Contraintes de Partitionnement de Graphe

Xavier Lorca

UNIVERSITÉ DE NANTES
FACULTÉ DES SCIENCES ET DES TECHNIQUES

ÉCOLE DOCTORALE

**SCIENCES ET TECHNOLOGIES
DE L'INFORMATION ET DES MATÉRIAUX**

Année 2007

N° attribué par la bibliothèque

Contraintes de Partitionnement de Graphe

THÈSE DE DOCTORAT

Discipline : Informatique

Spécialité : Contraintes

*Présentée
et soutenue publiquement par*

Xavier Lorca

*Le 29 Octobre 2007 à l'École Nationale Supérieure
des Techniques Industrielles et des Mines de Nantes*

Devant le jury ci-dessous :

Président	:	Jean-Xavier Rampon, Professeur	Université de Nantes
Rapporteurs	:	Alexander Bockmayr, Professeur	Freie Universität Berlin
		Gilles Pesant, Professeur	Université de Montréal
Examinateurs	:	Giuseppe F. Italiano, Professeur	Università de Roma
		Jean-Charles Régin, HDR	Ilog
Directeur de thèse	:	Nicolas Beldiceanu, Professeur	École des Mines de Nantes

Équipe d'accueil	:	Contraintes – LINA
Laboratoire d'accueil	:	Département Informatique de l'École des Mines de Nantes La Chantrerie – 4, rue Alfred Kastler – 44 307 Nantes

Remerciements

En fait, cette thèse c'est l'histoire de forêts où l'on trouve des arbres avec des racines et des branches mais étrangement à deux sommets.
L'accès à ces forêts est filtré car on est contraint d'y éviter des puits mais aussi d'y exhiber des tas de choses bizarres et inquiétantes qui peuvent violer de façon incomparable et sans aucune contrainte les règles fondamentales de bon comportement...
Ma parole, l'informatique c'est une jungle ! Plus sérieusement comme tu peux le constater, nous n'avons rien pigé.

Tes parents,
Benoît et Rose-marie

Preuve s'il en est que cette thèse n'a pas rendu l'informatique plus accessible aux non-initiés... Pourtant, le soutien inconditionnel de mes parents durant toutes ces années m'a donné la force de mener ce travail à terme. Un petit mot pour Bertille, la plus douce et tendre des compagnes, qui ne m'a jamais reproché de lui préférer un ordinateur durant ses rares week-end de repos.

Revenons à la science, enfin pas exactement...

À Naren (pardon, Monsieur Jussien) pour son « *pouvoir de nuisance* » qui, j'ai le regret de lui annoncer, ne nuit à personne mais donne du baume au cœur à tous ! Juste quelques mots empruntés pour quelqu'un qui apporte à tous sans rien demander, ni rien montrer : « *La puissance ne consiste pas à frapper fort ou souvent, mais à frapper juste* ».

À Sophie et Philippe pour les pauses café du matin, à heure précise...

À Nicolas, Richard, Christophe, Hadrien, Guillaume, Florian, Fabien pour toutes ces discussions inutiles et stériles mais oh combien réconfortantes dans les périodes difficiles.

À gsigms, rurunuela, nephente, emi.cst, wind pour tous ces neurones dépensés et pour ma main gauche qui maintenant connaît quelques raccourcis clavier.

À Aibo pour le lien social qu'il a créé pendant des semaines difficiles... Le salon de l'agriculture n'a pas du faire autant d'entrées que mon bureau.

Finalement, la science !

Ceux par qui tout commence, Frédéric Koriche et Christian Bessière sans qui je ne me serais pas lancé dans cette formidable expérience humaine et professionnelle.

À toutes les personnes avec qui j'ai pu échanger et collaborer durant ces trois années (j'espère n'oublier personne !) : Hadrien Cambazard, Grégoire Dooms, Pierre Flener, Fabien Hermenier, Narendra Jussien, Irit Katriel, Jean-Marc Menaud, Luis Quesada, et Guillaume Richaud.

Enfin, à Nicolas Beldiceanu, mon encadrant, il n'existe probablement pas de mots quantifiant ce qu'il m'a apporté. Les conversations que nous avons partagées, les commentaires et expertises dont il m'a fait part, m'ont permis de pousser toujours plus loin ma curiosité scientifique, sans pour autant perdre de vue les objectifs initiaux.

Table des matières

Introduction

La résolution de problèmes combinatoires est une problématique récurrente dans beaucoup de domaines, on citera par exemple la biologie ou encore la logistique. En particulier, on retiendra des applications pratiques dans des domaines aussi variés que la reconstruction de « super-arbres » en phylogénie, la planification de missions ou la construction de tournées de véhicules en logistique.

Pour toutes ces applications, un réflexe naturel pour un scientifique consiste à tenter d'abstraire la problématique donnée en dessinant des points (appelés *sommets*) pouvant représenter des individus dans le cas de la phylogénie, ou des lieux géographiques dans le cas de la planification de missions et le cas des tournées de véhicules. Puis, un second réflexe consiste à lier ces points par des traits (appelés *arcs* ou *arêtes* suivant qu'il existe ou non une orientation) afin de modéliser une relation logique entre eux. Cette relation peut consister en un lien de descendance entre individus dans le cas de la phylogénie, ou en une condition d'accessibilité entre lieux géographiques dans le cas de la planification de missions et le cas des tournées de véhicules. Cette manière abstraite de représenter un problème présente l'avantage de mettre en évidence la structure du problème que l'on veut résoudre, au travers d'une vision globale permettant un raisonnement efficace. Dans ce contexte, la théorie des graphes a permis de proposer des solutions théoriques à de nombreux problèmes pratiques.

Le reconstruction de super-arbres, la planification de missions et les tournées de véhicules, bien que provenant de domaines différents, sont des applications qui se ramènent à un problème particulier dans les graphes : le *partitionnement*. Il s'agit d'un regroupement de l'ensemble initial des sommets du graphe considéré en une collection de sous-ensembles respectant une certaine topologie (c.-à-d., une relation logique) appelée *patron*. En particulier, dans le cas de la reconstruction de super-arbres en phylogénie, le patron recherché vérifie que les arcs unifiant les individus ne peuvent pas former de cycles et que chaque individu est lié par un arc à un unique autre, représentant son père dans la relation filiale. Un tel patron dans la théorie des graphes est appelé *arbre*. De manière similaire, il en existe bien d'autres types, les plus connus étant le *chemin* et le *cycle*.

Cependant, les problèmes pratiques se résument rarement à un problème académique « pur » comme peuvent l'être des problèmes classiques de la théorie des graphes tel que le problème de chemin Hamiltonien ou celui des K-chemins disjoints. En effet, ils combinent très souvent le problème de partitionnement par un patron donné avec un ensemble de restrictions sur la topologie des sommets et des arcs. La diversité des contraintes opérationnelles mises en jeu constitue alors une limite à leur résolution par des approches considérant de manière séparée le problème de partitionnement et les restrictions supplémentaires imposées.

À ce titre, la *programmation par contraintes* offre un paradigme flexible et déclaratif permettant de prendre en compte ces restrictions sans pour autant remettre en question les fondements théoriques des problèmes de partitionnement de graphe. Du point de vue de la programmation par contraintes, la diversité des restrictions possibles se traduit bien souvent par un nombre non négligeable de contraintes distinctes modélisant le problème. Cependant, la seule modélisation d'un problème par un ensemble de contraintes distinctes n'est pas toujours suffisante dans la pratique. En effet, dans les applications liées au partitionnement de graphe la plupart des contraintes additionnelles modélisent des restrictions sur la topologie des partitions autorisées. Ces restrictions sont donc totalement liées entre elles via le type de patron (arbre, chemin, cycle) autorisé pour effectuer la partition.

Objectifs de la thèse

Les contraintes globales liées à la couverture de graphes sont généralement liées à la notion de coût associé à la couverture. Il s'agit donc de contraintes d'optimisation visant, par exemple, à fournir une couverture de coût minimum. On peut citer par exemple la contrainte de couverture par un chemin de coût minimum [Sel03, KH05], mais aussi la contrainte de couverture par un arbre de coût minimum [DK06]. Cependant, cette thèse se concentre sur les problèmes de *satisfaction de contraintes* liés au partitionnement de graphe par des arbres mettant en jeu un certain nombre de restrictions sur la topologie des partitions autorisées. Ainsi, nous ne traitons pas le volet optimisation (gestion des coûts de couverture) qui peut être associé aux contraintes de partitionnement de graphe. Notre travail se focalise plus particulièrement sur l'étude des *interactions* existantes entre le problème de partitionnement et les restrictions classiques (telles que les relations de précédences ou d'incomparabilités entre les sommets du graphe à partitionner) imposées par les applications concrètes telles que la reconstruction de super-arbres en phylogénie ou la planification de missions. Nous nous attacherons plus particulièrement à montrer comment prendre en compte de manière globale un certain nombre de ces restrictions au sein d'une contrainte de partitionnement de graphe par des arbres.

Un autre aspect essentiel portera sur la mise en œuvre d'une telle contrainte. Nous montrerons que la prise en compte des contraintes modélisant les restrictions sur les partitions autorisées, nécessite de faire communiquer (voire de partager) de manière efficace les structures de données mises en œuvre pour chaque restriction. Nous débattrons aussi de la nécessité de déployer, autant que faire se peut, des algorithmes totalement dynamiques pour gérer les mises à jour de ces structures de données. Pour cela, nous montrerons en quoi cette gestion dynamique permet de s'abstraire significativement d'un problème récurrent à la plupart des contraintes globales liées aux graphes : la sensibilité des algorithmes à la *densité*[1] du graphe considéré. En effet, intuitivement on peut penser que, plus le graphe à partitionner est dense, plus le nombre de partitions possibles de ce graphe est grand et plus le problème est finalement « facile » à résoudre. Cependant, la complexité de la plupart des algorithmes de graphes est directement liée au nombre d'arêtes[2] (resp., d'arcs). Cela constitue un facteur limitatif important quand à la taille des graphes possibles à traiter en pratique. On arrive donc au paradoxe suivant : d'une part, plus la densité du graphe considéré est élevée plus le problème est intuitivement facile, et d'autre part, plus la densité est élevée, plus le temps de résolution est important. C'est dans ce contexte qu'une gestion totalement dynamique des structures de données modélisant la contrainte permet de s'abstraire partiellement du facteur limitatif que constitue la densité du graphe.

Méthodologie de l'approche

Ce document couvre aussi bien les aspects théoriques liés au partitionnement de graphe que la mise en œuvre pratique d'une contrainte répondant à cette problématique. Il est organisé au travers des trois parties suivantes :

Cadre général : la première partie est consacrée à la mise en place du cadre de notre étude. Nous discuterons tout d'abord de la modélisation d'une famille de graphes dans le cadre des contraintes globales et des niveaux de consistances associés aux contraintes de partitionnement de graphe. Puis, nous rappellerons certaines définitions essentielles liées à la théorie des graphes qui seront utilisées tout au long de ce document.

Théorie des contraintes d'arbre : la seconde partie propose tout d'abord quatre contraintes de partitionnement de graphe par des arbres, deux variantes dans le cas des graphes non-orientés [BKL06], deux autres dans le cas des graphes orientés [BFL05]. Cependant, dans la plupart des applications pratiques, ces contraintes seules ne peuvent modéliser complètement tout le problème. Il faut bien souvent prendre en compte des restrictions sur les configurations possibles des partitions ; par exemple :

[1] La *densité* d'un graphe représente le ratio entre nombre d'arêtes (resp., d'arcs) du graphe et le nombre total d'arêtes (resp., d'arcs) possibles dans ce graphe. Par exemple, pour un graphe d'ordre n contenant m arêtes, la densité est donné par le ratio $\frac{m}{n^2}$.

[2] Remarquons qu'un grand nombre d'algorithmes de graphes sont soit basés sur un *parcours en profondeur d'abord*, soit un *parcours en largeur d'abord*.

- préciser qu'un sommet (ou un ensemble de sommets) doit être atteint avant un autre (un sous-ensemble d'autres) dans toute partition ;
- préciser que deux sommets (ou deux sous-ensembles de sommets) sont incomparables dans toute partition ;
- préciser qu'un sommet ne peut avoir qu'un nombre restreint de prédécesseurs.

Ainsi, nous proposons d'étendre une des deux contraintes d'arbre introduite dans le cas des graphes orientés en prenant en compte des contraintes additionnelles telles que les contraintes de précédence, d'incomparabilité ou encore de degré [BFL06]. Finalement, beaucoup d'applications pratiques sont directement liées à un patron de partitionnement spécifique : les chemins. Dans ce contexte, nous proposons une spécialisation de cette contrainte d'arbre au cas particulier des chemins [BL07]. Ce travail s'est focalisé sur l'évaluation du nombre minimum de chemins pouvant partitionner un graphe orienté puisqu'il s'agit de la problématique la plus courante dans les applications pratiques liées au partitionnement par des chemins.

Stratégies de mises en œuvre et applications : la troisième partie détaille l'implémentation et quelques applications de la contrainte d'arbre étendue aux contraintes additionnelles. Après avoir présenté une première implémentation de la contrainte basée sur les structures de données restaurables [AB90] fournies par le résolveur de contraintes *Choco* (`http://choco.sf.net/`), nous nous intéresserons à montrer comment déployer une telle contrainte indépendamment du résolveur utilisé [RLJ07] (c.-à-d., sans utiliser les structures de données spécifiques au résolveur *Choco*). Cette approche sera illustrée au travers de la mise en œuvre pratique d'une telle contrainte dans les résolveurs *Choco* et *Gecode* (`http://gecode.org/`). Ensuite, la contrainte d'arbre sera utilisée sur trois types d'applications : le problème de reconstruction de « super-arbres » en phylogénie, le problème de la couverture partielle d'un graphe par un chemin sous contraintes de précédences, et le problème de chemin Hamiltonien.

Première partie

Contexte de l'Étude

Chapitre 1

Cadre général

Sommaire

La programmation par contraintes [Mac77, Lau78, vH89] est une discipline située à la croisée de nombreux domaines comme la recherche opérationnelle, l'analyse numérique, le calcul symbolique et l'intelligence artificielle. Elle apporte une nouvelle approche à la résolution de problèmes combinatoires en tentant de combiner et d'unifier le meilleur de chacune de ces disciplines. Les premiers travaux dans ce domaine remontent aux recherches effectuées à la frontière entre l'intelligence artificielle et l'infographie [Mon74] dans les années 1970. Les deux dernières décennies ont permis de prendre conscience que les enjeux d'un paradigme déclaratif, permettant de modéliser, d'implémenter et de résoudre des problèmes combinatoires étaient essentiels. Ainsi, l'unification des différentes approches proposées autour de la résolution de problèmes combinatoires complexes a donné lieu à l'émergence d'un cadre commun tant conceptuel que pratique : la programmation par contraintes. Notons que les industriels, tels que Bull, Cosytec, IBM, ICL, Ilog, Prologia, ou Siemens, ont été les premiers à réaliser l'intérêt pratique de la programmation par contraintes dans des solutions globales de résolution et/ou d'optimisation de problèmes combinatoires réels.

1.1 Un paradigme déclaratif basé sur les mécanismes de propagation et de recherche

L'idée intuitive de la programmation par contraintes est de proposer une méthode de résolution de problèmes combinatoires basée sur la déclaration de contraintes (pouvant être vues comme des conditions portant sur des variables) devant être satisfaites par toute solution valide du problème considéré. Ainsi, dans le cadre discret, un Problème de Satisfaction de Contraintes (CSP) peut être défini de la manière suivante :

- un ensemble $\mathcal{V}$ de variables (au sens mathématique du terme) prenant leur valeur dans l'ensemble des entiers ;
- un ensemble $\mathcal{D}$ de domaines finis, représentant l'ensemble des valeurs possibles que chaque variable peut prendre ;
- un ensemble $\mathcal{C}$ de contraintes (relations logiques) portant sur des sous-ensembles des variables.

Une *solution* d'un CSP est une assignation des variables (c.-à-d., l'affectation d'une valeur à chaque variable) satisfaisant simultanément toutes les contraintes. C'est le *résolveur de contraintes* qui encapsule le mécanisme de résolution permettant d'aboutir à toutes ou une partie des solutions satisfaisant l'ensemble des contraintes. L'aspect déclaratif de la programmation par contraintes réside dans le fait que

l'utilisateur final doit seulement déclarer l'ensemble des variables modélisant son problème et décrire les conditions les liant (c.-à-d., les contraintes).

Le cœur de la programmation par contraintes repose sur le couple *propagation-recherche* : La partie propagation essaie de déduire de nouvelles informations à partir de l'état courant des variables. Quand à la partie recherche, elle consiste en un parcours, généralement une exploration dites en *profondeur d'abord*, de l'espace de recherche du CSP associé. Un algorithme effectuant uniquement un parcours de cet espace de recherche enumérera toutes les assignations de variables possibles jusqu'à, soit trouver une solution valide pour le CSP, soit conclure qu'il n'existe aucune solution satisfaisant simultanément toutes les contraintes. Un tel algorithme, que l'on peut qualifier d'algorithme de recherche exhaustif, a une complexité en temps exponentielle. C'est à ce niveau que le mécanisme de propagation vient *intelligemment* épauler la recherche. En effet, il permet au résolveur de contraintes de détecter et de supprimer des parties inutiles (dites *inconsistantes*) de l'espace de recherche sans avoir à les parcourir explicitement. La réduction de l'espace de recherche est effectuée au travers de la propagation de chaque contrainte mise en jeu dans le CSP, via un mécanisme appelé *filtrage*. Le filtrage consiste en l'examen de chaque variable non encore assignée (ou instanciée), en vue de supprimer de son domaine toute valeur inconsistante, c.-à-d., toute valeur qui si elle était assignée à cette variable violerait au moins une des contraintes mises en jeu dans le CSP. Malheureusement, détecter et supprimer pour chaque variable toutes les valeurs inconsistantes relève très souvent d'un problème NP-difficile. C'est pour cela que le couple « propagation-recherche » est généralement indissociable et qu'un des enjeux majeurs de la programmation par contraintes reste la recherche d'un juste équilibre entre *l'efficacité*, en terme du temps de calcul, et le rendement effectif[1], en terme de filtrage. En d'autres termes, lors de la conception de contraintes, il est primordial de trouver un équilibre entre le temps de calcul de l'algorithme effectuant le filtrage de la contrainte et la quantité effective de valeurs inconsistantes détectées (ce qui permet bien souvent en pratique, de réduire de manière effective la taille de l'espace de recherche à parcourir).

1.2 Les contraintes globales

Une *contrainte globale* [BvH03] permet d'exprimer une relation logique entre un nombre non fixé de variables mises en jeu dans un CSP. La contrainte globale la plus connue est, sans conteste, la contrainte de différence [Rég94] $\mathit{alldifferent}(v_1, v_2, \ldots, v_n)$ qui exprime que toutes les valeurs affectées aux variables $v_1, v_2, \ldots, v_n$ doivent être deux à deux distinctes. L'intérêt d'une telle contrainte dépasse clairement le gain en expressivité qu'elle offre par rapport à une représentation basée sur $\frac{n\times(n-1)}{2}$ contraintes de différence binaires du type $v_i \neq v_j$. En effet, le rendement effectif de l'algorithme de filtrage qui lui est associé dépasse largement la capacité déductive de « simples » contraintes de différences binaires. D'une manière plus générale, c'est le comportement recherché pour ce type de contrainte dont l'efficacité, en terme de temps de calcul, constitue la principale limite. Ainsi, comme c'est le cas pour la contrainte *alldifferent*, si un algorithme de filtrage polynomial supprime du domaine de chaque variable toute valeur n'appartenant à aucune solution compatible alors on dira que cet algorithme effectue un *filtrage complet* des domaines des variables. Cependant, beaucoup de contraintes globales servent à modéliser des problèmes qui sont intrinsèquement NP-complets, ce qui, bien entendu, entraîne que seul un *filtrage partiel* des domaines des variables peut être effectué au travers d'un algorithme polynomial. C'est par exemple le cas de la contrainte *nvalue*[2] dont l'un des aspects est lié au problème de « minimum hitting set » [Bel01, BHHW04]. Finalement, le caractère « global » de ce type de contrainte permet aussi de simplifier le travail du résolveur par le fait qu'il fournit toute ou partie de la structure du problème à résoudre.

1.3 Les contraintes de partitionnement de graphe

Les contraintes décrivant les propriétés de graphes sont considérées comme une nouvelle étape de la recherche en programmation par contraintes. Quelques exemples connus sont les contraintes de *chemin Hamiltonien* et *d'arbre couvrant* proposées dans ALICE [Lau78], qui ont été suivies des contraintes de *cycle*

[1] Le *rendement effectif* d'un algorithme de filtrage peut être vu comme sa *capacité déductive* en terme de valeurs inconsistantes détectées dans le domaine de chaque variable qu'il met en jeu.

[2] La contrainte *nvalue* permet de restreindre le nombre de valeurs distinctes assignées à un ensemble de variables.

et de *chemin* introduites dans les premières versions de CHIP [DvHS+88] et d'Ilog Solver [Pug94]. Ces dernières années ont donné lieu à une explosion des travaux sur les contraintes associées aux propriétés de graphes. On peut en particulier retenir les contraintes de cycle [BC94, PS02, KH06], de chemins [JKK+99, Sel03, CB04, QvRDC06], d'arbres [PU06b], de connexité [PU06a], ou encore des travaux plus généraux portant sur des ensembles de propriétés de graphes [BPR05, BCDP07].

La complexité de développement des contraintes de partitionnement de graphe et, plus généralement, des contraintes mettant en jeu des propriétés de graphes, repose à la fois sur la représentation des graphes en terme de structures de données, et sur l'efficacité de l'implémentation des algorithmes maintenant en jeu les propriétés nécessaire pour filtrer les domaines des variables. Bien entendu, le choix de la représentation et l'efficacité d'implémentation sont très fortement liés. Cet aspect des contraintes de partitionnement de graphe sera discuté plus en détail dans la suite de ce document.

Chapitre 2

Programmation par contraintes et théorie des graphes

Sommaire

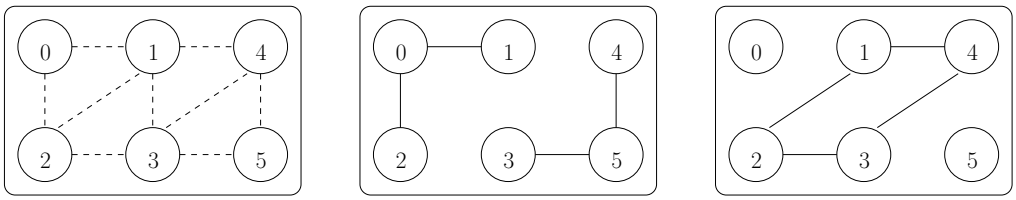
La modélisation de graphes dans les contraintes globales constitue un point clé dans la résolution de problèmes liés au partitionnement de graphe. Lors de la conception de telles contraintes, le type de variables choisies constitue un facteur décisif quand à l'implémentation des algorithmes qu'elles mettent en jeu. Dans ce contexte, nous discutons tout d'abord de différentes manières de représenter les graphes, en terme de variables, au sein du paradigme qui constitue la programmation par contraintes (section 2.1). Ensuite, nous détaillons plus précisément les niveaux de consistance les plus élevés atteignables par une contrainte prenant en paramètre des graphes (section 2.2 page 21). Finalement, nous rappelons certains concepts fondamentaux de la théorie des graphes qui seront utilisés tout au long de ce document (section 2.3 page 21).

2.1 Représenter une famille de graphes

Les problèmes liés au partitionnement de graphe peuvent souvent se voir comme la recherche d'un graphe partiel[1], vérifiant certaines propriétés, induit par un graphe initial constituant la donnée de base du problème. Par la suite, nous appellerons famille de graphes l'ensemble des graphes partiels induits par un graphe initial. L'objet de cette section est d'introduire certaines représentations de familles de graphes en programmation par contraintes, permettant de modéliser l'ensemble des graphes partiels induit par un graphe initial donné.

Nous introduisons dans cette section deux modélisations possibles d'une famille de graphes en terme de variables, l'une permettant de modéliser une famille de graphes non-orientés, l'autre dédiée au cas d'une famille de graphes orientés. Dans les deux cas, les représentations permettent d'exprimer tout graphe partiel d'un graphe orienté (resp. non-orienté) $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ engendré par un sous-ensemble d'arcs (resp. arêtes) $\mathcal{S} \subseteq \mathcal{E}$.

[1] Un graphe partiel $\mathcal{G}' = (\mathcal{V}, \mathcal{S})$ engendré par un graphe $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ est tel que $\mathcal{S} \subseteq \mathcal{E}$.



(a) $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ (b) $\mathcal{G}' = (\mathcal{V}, \mathcal{E}')$, $\mathcal{E}' \subseteq \mathcal{E}$ (c) $\mathcal{G}'' = (\mathcal{V}, \mathcal{E}'')$, $\mathcal{E}'' \subseteq \mathcal{E}$

FIG. 2.1 – Le graphe non-orienté $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ représente le graphe initial. Les graphes $\mathcal{G}'$ et $\mathcal{G}''$ sont deux représentants de la famille des graphes non-orientés induits par $\mathcal{G}$; il s'agit de deux graphes partiels non-orientés induits par $\mathcal{G}$.

Les deux types classiques de variables connues en programmation par contraintes sont les variables entières et les variables ensemblistes. Elles peuvent être définies de la manière suivante :

Définition 1 (Variable entière)**.** *Une* variable entière V_i *prend sa valeur dans un ensemble fini d'entiers noté* $\mathcal{D}(V_i)$*. La plus petite et la plus grande valeur de* $\mathcal{D}(V_i)$ *sont respectivement notées* $\min(V_i)$ *et* $\max(V_i)$*.*

Définition 2 (Variable ensembliste)**.** *Le domaine d'une* variable ensembliste V_s *est un ensemble d'ensembles finis d'entiers. Il est décrit par sa* borne inférieure *(ou* noyau*)* $\underline{V_s}$ *et sa* borne supérieure *(ou* enveloppe*)* $\overline{V_s}$*. Toute valeur de* $\underline{V_s}$ *est contenu dans* V_s *et toute valeur de* V_s *est contenu dans* $\overline{V_s}$*. Lorsque la variable ensembliste est* fixée *alors* $\underline{V_s} = \overline{V_s}$*. Les valeurs dans* $\underline{V_s}$ *sont dites* valeurs obligatoires *et les valeurs de* $\overline{V_s} \setminus \underline{V_s}$ *sont dites* valeurs potentielles*.*

Nous sommes maintenant en mesure de modéliser une famille de graphes non-orientés (figure 2.1) et une famille de graphes orientés (figure 2.2 page ci-contre).

Définition 3 (Famille de graphes non-orientés)**.** *Étant donné un graphe initial* $\mathcal{G} = (\mathcal{V}, \mathcal{E})$*, la famille de graphes représentant l'ensemble des graphes partiels de* $\mathcal{G}$ *est définie par l'association d'une variable ensembliste* V_v *à chaque sommet* v *de* $\mathcal{V}$ *tel que, pour chaque arête* $(u, v) \in \mathcal{E}$*, si* $u \in \underline{V_v}$ *alors* $v \in \overline{V_u}$*.*

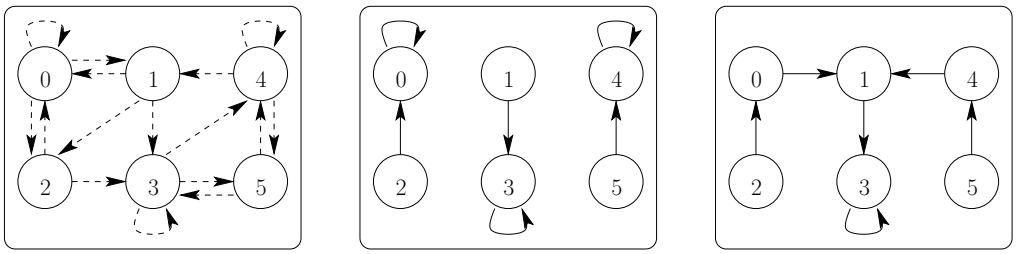
Propriété 1. *Toute assignation complète des variables ensemblistes modélisant un graphe* $\mathcal{G} = (\mathcal{V}, \mathcal{E})$*, telle que* $u \in V_v$*, définit un graphe partiel de* $\mathcal{G}$*.*

Définition 4 (Famille de graphes orientés)**.** *Étant donné un graphe orienté* $\mathcal{G} = (\mathcal{V}, \mathcal{E})$*, la famille de graphes représentant l'ensemble des graphes partiels de* $\mathcal{G}$ *est définie par l'association d'une variable entière* V_v *à chaque sommet* v *de* $\mathcal{V}$ *tel que, pour chaque arc* $(v, w) \in \mathcal{E}$*, on a bien* $w \in \mathcal{D}(V_v)$*.*

Propriété 2. *Toute assignation complète des variables entières modélisant un graphe orienté* $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ *définit un graphe partiel* $\mathcal{G}' = (\mathcal{V}, \mathcal{S})$ *de* $\mathcal{G}$ *tel que pour chaque sommet* $v \in \mathcal{V}$*, il existe un unique successeur* $w \in \mathcal{V}$ *où* $(v, w) \in \mathcal{S}$*.*

Notons que ces deux modélisations ne constituent pas la seule approche possible. En effet, Dooms *et al.* proposent un cadre spécifique de modélisation directe des graphes en programmation par contraintes : *CP(Graph)* [DDD05]. L'intuition de ce modèle est basée sur le fait que proposer une représentation d'un graphe par, à la fois une relation sur ses sommets (comme celle que nous proposons), et une relation sur ses arcs (généralement, il s'agit d'une modélisation par une matrice de variables booléennes statuant sur l'appartenance d'un arc à au moins une solution) est strictement plus expressive qu'une représentation basée uniquement sur l'une de ces deux alternatives. Cette intuition prend sa source dans les travaux de B. Courcelle [Cou97] basés sur une logique univalente du second ordre permettant d'exprimer certaines propriétés de graphes (par exemple, l'existence d'un chemin entre deux sommets donnés du graphe). Malheureusement, cette logique n'est pas exploitée pour évaluer (ou filtrer) par rapport à une propriété de graphe.

Dans la suite de ce document, nous proposerons une modélisation des graphes basée sur les définitions 3 et 4. Cependant, le choix d'écarter la modélisation introduite dans CP(Graph) a été motivée par le fait que nous ne souhaitions pas, pour l'heure, introduire ce nouveau type de variables dans le résolveur de



(a) $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ (b) $\mathcal{G}' = (\mathcal{V}, \mathcal{E}')$, $\mathcal{E}' \subseteq \mathcal{E}$ (c) $\mathcal{G}'' = (\mathcal{V}, \mathcal{E}'')$, $\mathcal{E}'' \subseteq \mathcal{E}$

FIG. 2.2 – Le graphe orienté $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ représente le graphe initial. Les graphes orientés $\mathcal{G}'$ et $\mathcal{G}''$ sont deux représentants de la famille des graphes induit par $\mathcal{G}$; Il s'agit de deux graphes partiels orientés induits par $\mathcal{G}$ vérifiant que chaque sommet de $\mathcal{V}$ possède un unique successeur.

contrainte *Choco* (`http://choco.sf.net/`) qui sera utilisé pour les différentes expérimentations. En effet, même si les variables graphes permettent un bon niveau d'expressivité pour des contraintes de graphes, les difficultés de gestion des interactions avec les autres contraintes du résolveur, qui utilisent des variables entières et ensemblistes, nous ont convaincus de rester sur une approche plus « classique ».

2.2 Niveaux de consistance

Afin de pouvoir caractériser le comportement de l'algorithme de filtrage d'une contrainte mentionnant des variables entières et/ou ensemblistes d'un point de vue déductif, nous introduisons la notion de niveaux de consistance. Plus précisément, à partir des deux définitions de variables entières et ensemblistes, nous introduisons les définitions relatives aux deux niveaux de consistances les plus élevés respectivement atteignables, par une contrainte, dans le cas de variables entières seules et le cas de l'utilisation conjointe de variables entières et de variables ensemblistes.

Définition 5 (Arc-Consistance Généralisée [Mac77, MF85])**.** *Une contrainte $\mathcal{C}$, définie sur les variables entières V_1^d, ..., V_l^d, atteint l'*arc-consistance généralisée *ssi pour chaque paire (V^d, v) telle que V^d est une variable entière de $\mathcal{C}$ et $v \in \mathcal{D}(V^d)$, il existe au moins une solution respectant $\mathcal{C}$ dans laquelle la valeur v est assignée à la variable V^d.*

Définition 6 (Consistance-Hydride [BHH+05, BKL06])**.** *Une contrainte $\mathcal{C}$, définie sur les variables entières V_1^d, ..., V_l^d et sur les variables ensemblistes V_{l+1}^s, ..., V_n^s, est* hybride-consistante *ssi :*

1. *pour chaque paire (V^d, v) telle que V^d est une variable entière de $\mathcal{C}$ et $v \in \mathcal{D}(V^d)$, il existe au moins une solution respectant $\mathcal{C}$ dans laquelle la valeur v est assignée à V^d ;*
2. *pour toute paire (V^s, v) telle que V^s est une variable ensembliste de $\mathcal{C}$, si $v \in \underline{V^s}$ alors v appartient à l'ensemble assigné à V^s dans* toutes *les solutions respectant $\mathcal{C}$ et si $v \in \overline{V^s} \setminus \underline{V^s}$ alors v appartient à l'ensemble assigné à V^s dans au moins une solution et est exclue de cet ensemble dans au moins une solution.*

2.3 Définitions et propriétés de graphes utiles

De nombreux algorithmes de filtrage mettant en jeu des contraintes modélisant des problèmes de graphes (ou des contraintes globales modélisables en terme de la recherche d'un graphe vérifiant certaines propriétés) sont intrinsèquement liés à des notions classiques faisant partie de la théorie des graphes. Dans ce contexte, l'objet de cette section est de rappeler un ensemble de définitions, notations et théorèmes liés à la théorie des graphes qui seront utilisés tout au long de ce document.

2.3.1 Définitions générales

Nous introduisons dans cette section les définitions communes aux graphes, qu'ils soient orientés ou non.

Définition 7 (Graphe). *Un graphe est un couple $(\mathcal{V}, \mathcal{E})$, où $\mathcal{V}$ est un ensemble d'objets appelés les sommets du graphe, et $\mathcal{E}$ est une relation binaire sur $\mathcal{V} \times \mathcal{V}$. Les éléments de $\mathcal{E}$ sont appelés les arêtes (arcs) du graphe.*

Définition 8 (Ordre d'un graphe). *L'*ordre *d'un graphe $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ est défini par le nombre de sommets dans ce graphe, c.-à-d., le nombre d'éléments dans l'ensemble $\mathcal{V}$.*

Définition 9 (Sous-graphe [Ber70]). *Étant donné un graphe $\mathcal{G} = (\mathcal{V}, \mathcal{E})$, un* sous-graphe $\mathcal{G}'$ *de $\mathcal{G}$ engendré par $\mathcal{V}' \subset \mathcal{V}$ est défini par $(\mathcal{V}' \subseteq \mathcal{V}, \{(i,j) \in \mathcal{E} \mid i,j \in \mathcal{V}'\})$.*

Définition 10 (Graphe partiel [Ber70]). *Étant donné un graphe $\mathcal{G} = (\mathcal{V}, \mathcal{E})$, un* graphe partiel $\mathcal{G}'$ *de $\mathcal{G}$ engendré par $\mathcal{E}' \subset \mathcal{E}$ est défini par $(\mathcal{V}, \mathcal{E} \cap \mathcal{E}')$.*

Définition 11. *Étant donnés deux graphes $\mathcal{G}_1 = (\mathcal{V}_1, \mathcal{E}_1)$ et $\mathcal{G}_2 = (\mathcal{V}_2, \mathcal{E}_2)$, on peut définir les opérations suivantes :*

- *$\mathcal{G}_1 \cup \mathcal{G}_2$ dénote l'*union *des graphes $\mathcal{G}_1$ et $\mathcal{G}_2$, qui est définie par le graphe $(\mathcal{V}_1 \cup \mathcal{V}_2, \mathcal{E}_1 \cup \mathcal{E}_2)$.*
- *$\mathcal{G}_1 \cap \mathcal{G}_2$ dénote l'*intersection *des graphes $\mathcal{G}_1$ et $\mathcal{G}_2$, qui est définie par le graphe $(\mathcal{V}_1 \cup \mathcal{V}_2, \mathcal{E}_1 \cap \mathcal{E}_2)$.*
- *$\mathcal{G}_1 \subseteq \mathcal{G}_2$ dénote l'*inclusion *des arêtes (des arcs) de $\mathcal{G}_1$ dans $\mathcal{G}_2$, définie par $\mathcal{V}_1 = \mathcal{V}_2$, $\mathcal{E}_1 \cup \mathcal{E}_2 = \mathcal{E}_2$ et $\mathcal{E}_1 \cap \mathcal{E}_2 = \mathcal{E}_1$;*
- *$\mathcal{G}_1 \setminus \mathcal{V}_2$ dénote la* restriction *de $\mathcal{G}_1$ aux sommets de $\mathcal{V}_1 \setminus \mathcal{V}_2$. On définit la restriction sur les sommets par $(\mathcal{V}_1 \setminus \mathcal{V}_2, \{(i,j) \in \mathcal{E}_1 \mid i \notin \mathcal{V}_2 \wedge j \notin \mathcal{V}_2\})$.*
- *$\mathcal{G}_1 \setminus \mathcal{E}_2$ dénote la restriction de $\mathcal{G}_1$ aux arêtes (aux arcs) de $\mathcal{E}_1 \setminus \mathcal{E}_2$. On définit la restriction sur les arêtes (les arcs) par $(\mathcal{V}_1, \{(i,j) \in \mathcal{E}_1 \mid (i,j) \notin \mathcal{E}_2\})$.*
- *$TC(\mathcal{G}_1)$ dénote la* fermeture transitive *de $\mathcal{G}_1$, qui est le graphe $(\mathcal{V}_1, \mathcal{E}'_1)$ tel que pour tout v, w dans $\mathcal{V}_1$ il existe une arête (un arc) (v,w) dans $\mathcal{E}'_1$ ssi il existe un chemin (une chaîne), non vide, de v à w dans $\mathcal{G}_1$.*
- *$TR(\mathcal{G}_1)$ dénote la* réduction transitive *de $\mathcal{G}_1$, qui est le plus petit graphe (au sens de l'inclusion d'arêtes ou d'arcs) tel que $TC(\mathcal{G}_1) = TC(TR(\mathcal{G}_1))$.*

Définition 12 (Différence symétrique). *Soit S un ensemble fini et E, F deux parties de S, on appelle différence symétrique de E et F, notée $E \oplus F$, l'ensemble constitué par la réunion des éléments de E qui ne sont pas dans F, et des éléments de F qui ne sont pas dans E. Formellement, on écrira :*

$$E \oplus F = \{x \in S \mid x \in (E \setminus F) \vee x \in (F \setminus E)\}$$

2.3.2 Dans le cadre des graphes non-orientés

Nous introduisons maintenant certains concepts relatifs aux graphes non-orientés qui seront utilisés par les contraintes d'arbres introduites dans ce document (ces différentes définitions sont extraites de [Ber70, GM85]) :

Définition 13 (Chaîne). *Une* chaîne *de longueur $q > 0$ est une séquence $\alpha = (u_1, u_2, \ldots, u_q)$ d'arêtes d'un graphe $\mathcal{G}$ telle que chaque arête de la séquence ait une extrémité en commun avec l'arête précédente, et l'autre extrémité avec l'arête suivante.*

Définition 14 (Cycle). *Un* cycle *est une chaîne $\alpha = (u_1, u_2, \ldots, u_q)$ telle que : (1) la même arête ne figure pas deux fois dans la séquence α et, (2) les deux sommets aux extrémités de la chaîne coïncident.*

Définition 15 (Composante connexe). *Un graphe $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ est dit* connexe *ssi pour toute paire de sommets u et v, il existe une chaîne dans $\mathcal{G}$ reliant ces deux sommets. On peut alors extraire la relation d'équivalence :*

$$i \mathcal{R} j \Leftrightarrow \begin{cases} \text{il existe une chaîne de } i \text{ à } j. \\ \text{ou } i = j. \end{cases}$$

Les classes de cette équivalence constituent une partition de $\mathcal{V}$ en sous-graphes connexes de $\mathcal{G}$, appelés composantes connexes *de $\mathcal{G}$. $CC(i)$ désigne la composante connexe de $\mathcal{G}$, maximale au sens de l'inclusion, contenant le sommet i de $\mathcal{V}$.*

Définition 16 (Point d'articulation, isthme). *Étant donné un graphe non-orienté $\mathcal{G} = (\mathcal{V}, \mathcal{E})$. Un* point d'articulation *de $\mathcal{G}$ est un sommet dont la suppression augmente le nombre de composantes connexes de $\mathcal{G}$. Un* isthme *de $\mathcal{G}$ est une arête e de $\mathcal{E}$ dont le retrait augmente le nombre de composantes connexes de $\mathcal{G}$.*

Définition 17 (2-connexité). *Un graphe est dit* 2-connexe *si et seulement s'il est connexe, d'ordre $n \geq 3$ et, n'admet pas de points d'articulation.*

Définition 18 (Couplage, sommet saturé, couplage maximum). *Soit $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ un graphe non-orienté. Un* couplage *dans $\mathcal{G}$ est un sous-ensemble d'arêtes $M \subset \mathcal{E}$, tel que deux arêtes de M ne sont pas adjacentes. Un sommet u de $\mathcal{G}$ est dit* saturé *par le couplage M s'il existe une arête de M incidente à u. Un* couplage maximum *est un couplage de cardinalité maximale.*

2.3.3 Dans le cadre des graphes orientés

Nous introduisons maintenant certains concepts relatifs aux graphes orientés utilisés dans ce document :

Définition 19 (Sommet puits). *Un sommet* puits *dans un graphe orienté $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ est un sommet p tel qu'il n'existe pas d'arc (p, i) dans $\mathcal{E}$, quel que soit $i \neq p$ dans $\mathcal{V}$.*

Définition 20 (Chemin). *Un* chemin *de longueur $q > 0$ est une séquence $\alpha = (u_1, u_2, \ldots, u_q)$ d'arcs d'un graphe $\mathcal{G}$ telle que chaque arc de la séquence ait son origine en commun avec l'extrémité terminale de l'arc précédent, et son extrémité terminale avec l'origine de l'arc suivant.*

Définition 21 (Circuit). *Un* circuit *est un chemin $\alpha = (u_1, u_2, \ldots, u_q)$ telle que : (1) le même arc ne figure pas deux fois dans la séquence α et, (2) les deux sommets aux extrémités du chemin coïncident.*

Définition 22 (Composante fortement connexe et graphe réduit [GM85]). *Un graphe orienté $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ est dit* fortement connexe *ssi, étant donné deux sommets quelconques i et j de $\mathcal{V}$, il existe un chemin de i vers j. On peut alors extraire la relation d'équivalence suivante :*

$$i \ \mathcal{R} \ j \Leftrightarrow \begin{cases} \text{il existe à la fois un chemin de } i \text{ à } j \text{ et de } j \text{ à } i. \\ \text{ou } i = j. \end{cases}$$

Les classes d'équivalence induites sur $\mathcal{V}$ par cette relation forment une partition $\mathcal{V}_1, \ldots, \mathcal{V}_q$. Les sous-graphes de $\mathcal{G}$ engendrés par les sous-ensembles $\mathcal{V}_1, \ldots, \mathcal{V}_q$ sont appelés composantes fortement connexes *de $\mathcal{G}$. Le* graphe réduit *$\mathcal{G}_r$ est défini par le quotient du graphe $\mathcal{G}$ avec la relation de forte connexité $\mathcal{G}_r = \mathcal{G} \setminus \mathcal{R}$; À chaque composante fortement connexe (CFC) de $\mathcal{G}$ est associé un sommet de $\mathcal{G}_r$ et à l'ensemble d'arcs reliant une première CFC de $\mathcal{G}$ à une seconde CFC de $\mathcal{G}$ correspond un arc unique de $\mathcal{G}_r$. La notation $CFC(i)$ désigne la composante fortement connexe de $\mathcal{G}$, maximale au sens de l'inclusion, contenant le sommet $i \in \mathcal{V}$.*

Définition 23 (Degré [Ber70]). *Soit $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ un graphe orienté $\mathcal{G} = (\mathcal{V}, \mathcal{E})$. Si un sommet u de $\mathcal{V}$ est l'extrémité initiale d'un arc (u, v) de $\mathcal{E}$, on dit que cet arc est* incident à u vers l'extérieur. *Le nombre d'arcs dans lesquels u est l'extrémité initiale dans $\mathcal{G}$ se note $d^+_{\mathcal{G}}(u)$. Ce nombre est désigné par le terme* demi-degré extérieur *de u. Un arc* incident à u vers l'intérieur *et le* demi-degré intérieur *$d^-_{\mathcal{G}}(u)$ se définissent de la même manière. Le* degré *d'un sommet u est donné par $d_{\mathcal{G}(u)} = d^+_{\mathcal{G}}(u) + d^-_{\mathcal{G}}(u)$.*

Définition 24 (Anti-arborescence). *Un graphe partiel $\mathcal{T}$ d'un graphe orienté $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ est une* anti-arborescence *de racine $r \in \mathcal{V}$, si c'est un arbre enraciné en r (c.-à-d., le sommet r est sommet puits de $\mathcal{T}$), orienté des feuilles vers la racine.*

Définition 25 (Sommet dominant [LT79]). *Étant donné un graphe orienté $\mathcal{G}$ et deux sommets i et j de $\mathcal{G}$ tels qu'il existe au moins un chemin de i à j ; un sommet d est un* sommet dominant *le sommet j par rapport au sommet i ssi il n'existe pas de chemin de i à j dans $\mathcal{G} \setminus \{d\}$. L'ensemble des sommets dominants j par rapport à i est noté $DOM_{<\mathcal{G},i>}(j)$.*

Les notations suivantes sont spécifiquement introduites dans le cadre du partitionnement d'un graphe orienté par des arbres (Chapitres 4 page 47 et 5 page 65).

Notation 1 (Composante puits d'un graphe). *Étant donné une graphe $\mathcal{G}$ et une cfc $\mathcal{C}$ de $\mathcal{G}$, s'il n'existe pas un arc (u, v) de $\mathcal{G}$ tel que $u \in \mathcal{C}$ et $v \notin \mathcal{C}$ alors, $\mathcal{C}$ est appelée* composante puits.

Notation 2 (Racines potentielles d'un graphe). *Un sommet v de $\mathcal{V}$, tel que (v, v) appartient à $\mathcal{E}$, est appelé* racine potentielle. *Par extension, un arc (v, v) de $\mathcal{E}$ est appelé* boucle. *Une cfc de $\mathcal{G}$ contenant au moins une racine potentielle est appelée* composante enracinée.

Notation 3 (Sommet particulier dans une composante fortement connexe). *Un sommet u de $\mathcal{V}$ est appelé* porte *de la cfc le contenant ssi il existe un arc (u, v) appartenant à $\mathcal{E}$ tel que v n'appartient pas à la cfc contenant u. Par extension, un arc (u, v) de $\mathcal{E}$ tel que u et v n'appartiennent pas à la même cfc est appelé* arc connectant.

2.3.4 Théorie des flots

Cette section rappelle trois définitions issues de [Ber70, p.72] définissant les notions essentielles de la théorie des flots.

Définition 26. *Soit $\mathcal{A}$ un ensemble de sommets d'un graphe $\mathcal{G} = (\mathcal{V}, \mathcal{E})$, soit $\omega^+(\mathcal{A})$ l'ensemble d'arcs sortant de $\mathcal{A}$, et soit $\omega^-(\mathcal{A})$ l'ensemble d'arcs entrant sur $\mathcal{A}$. Un* cocycle *est un ensemble d'arcs $\omega(\mathcal{A}) = \omega^+(\mathcal{A}) \cup \omega^-(\mathcal{A})$.*

Définition 27 (Réseau). *Soit $\mathcal{N} = (\mathcal{V} \cup \{s, t\}, \mathcal{E})$ un graphe orienté tel qu'à chaque arc (i, j) de $\mathcal{E}$ est associée une* borne inférieure ℓ_{ij} *représentée par un entier non-négatif, et une* capacité c_{ij} *représentée, elle-aussi, par un entier non-négatif ($\ell_{ij} \leq c_{ij}$). Pour chaque sommet i, distinct de s et t, il existe un chemin dans $\mathcal{N}$ de s à i et un chemin de i à t. Il y a aussi un arc de t à s, appelé* arc de retour *de $\mathcal{N}$.*

Définition 28 (Flot). *Un* flot *dans un réseau $\mathcal{N} = (\mathcal{V} \cup \{s, t\}, \mathcal{E})$ est défini par une fonction $f : \mathcal{E} \mapsto \mathbb{N}$ telle que :*

1. Contraintes de capacité *: quel que soit (i, j) de $\mathcal{E}$, $f(i, j)$ appartient à $[l_{ij}, c_{ij}]$.*
2. Conservation du flot *:*

$$\forall x \in \mathcal{V}, \sum_{(i,x) \in \omega^-(\{x\})} f(i, x) = \sum_{(x,j) \in \omega^+(\{x\})} f(x, j) \tag{2.1}$$

Un flot global $\mathcal{F}(\mathcal{N})$ *d'un réseau $\mathcal{N}$ est fourni par :*

$$\mathcal{F}(\mathcal{N}) = \sum_{(s,j) \in \omega^+(\{s\})} f(s, j) \tag{2.2}$$

Théorème 1 (Hoffman). *Étant donné un réseau $\mathcal{N} = (\mathcal{V}, E)$ défini par une fonction $f : \mathcal{E} \mapsto \mathbb{N}$ telle que pour tout (i, j) de $\mathcal{E}$, $f_{ij} \in [\ell_{ij}, c_{ij}]$, il existe un flot compatible dans $\mathcal{N}$ ssi pour tout cocycle $\omega(\mathcal{A})$ de $\mathcal{N}$, on a :*

$$\sum_{(i,j) \in \omega^+(\mathcal{A})} c_{ij} - \sum_{(i,j) \in \omega^-(\mathcal{A})} \ell_{ij} \geq 0 \tag{2.3}$$

2.4 Implémentation des graphes et complexité

Nous nous intéressons ici à préciser les hypothèses que nous avons faites dans ce document sur les temps de calculs permettant d'effectuer les opérations basiques manipulant des graphes. Nous nous plaçons dans le cadre d'un graphe orienté ou non $\mathcal{G} = (\mathcal{V}, \mathcal{E})$, tel que $|\mathcal{V}| = n$ et $|\mathcal{E}| = m$.

Nous allons supposer que les graphes sont représentés par des structures de données Java du type `BitSet` ou `HashSet`. Plus précisément, nous supposerons qu'un graphe peut être vu de deux manières distinctes :

– Le graphe est représenté par un tableau $T[\ ]$ de `BitSet`. Quelque soit $i \in [0, n]$, $T[i]$ désigne le `BitSet` représentant l'ensemble des voisins (successeurs) du sommet d'index i.

- Le graphe est représenté par une collection d'objets de type `Sommet` définis par un index de type `Integer` et une liste de type `HashSet<Sommet>` désignant l'ensemble des sommets voisins (successeurs) du sommet i.

Avec ce type d'implémentation, les opérations basiques ont les complexités suivantes :

- Le *test d'appartenance* d'une arête (arc) au graphe $\mathcal{G}$ est exécuté en temps constant.
- L'*énumération* des voisins (successeurs) d'un sommet du graphe $\mathcal{G}$ est exécutée en temps proportionnel au nombre de voisins (successeurs) du sommet considéré.
- L'*ajout* d'une arête (arc) dans le graphe $\mathcal{G}$ est effectué en temps constant.
- La *suppression* d'une arête (arc) dans le graphe $\mathcal{G}$ est effectuée en temps constant.
- L'*intersection* de l'ensemble des arêtes (arcs) de deux graphes $\mathcal{G}$ et $\mathcal{G}'$ est effectuée en temps proportionnel à la somme du nombre d'arêtes dans $\mathcal{G}$ et $\mathcal{G}'$.

Dans la suite du document, les complexités annoncées supposerons que les opérations basiques décrites ci-dessus sont respectées.

Deuxième partie

Théorie des contraintes de partitionnement par des arbres

Chapitre 3

Contraintes d'arbre dans les graphes non-orientés

Sommaire

Dans ce chapitre, nous présentons deux contraintes qui partitionnent les sommets d'un graphe non-orienté $\mathcal{G} = (\mathcal{V}, \mathcal{E})$, où $|\mathcal{V}| = n$ et $|\mathcal{E}| = m$, en un ensemble d'arbres disjoints. La première contrainte, *resource-forest*, spécifie que chaque arbre dans la forêt doit contenir au moins un sommet *ressource*. L'ensemble des ressources est un sous-ensemble de sommets `R` inclus dans $\mathcal{V}$. Nous décrivons un algorithme de consistance-hybride (voir définition 6 page 21) pour la contrainte *resource-forest* ayant une complexité de $O(m + n)$. La seconde contrainte, *proper-forest*, est une variante de la première ne nécessitant pas que chaque arbre contienne une ressource. Cependant, tout arbre construit doit être un arbre *propre*, c.-à-d., un arbre contenant au moins deux sommets. Nous avons développé un algorithme de consistance-hybride ayant une complexité en $O(mn)$ au pire des cas, et en $O(m\sqrt{n})$ dans la plupart des cas.

La contrainte *resource-forest* peut s'appliquer au problème classique de conception de réseaux qui consiste à considérer un graphe non-orienté $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ où $\mathtt{R} \subseteq \mathcal{V}$ est un ensemble de sommets correspondant à un type de ressources, par exemple des imprimantes. Les sommets restant représentent des tâches (ou clients, ou encore utilisateurs). Le problème est de couvrir tous les sommets du graphe par des arbres (réseaux) tels que chaque arbre contienne au moins un sommet ressource (ce qui signifie par exemple que chaque réseau doit avoir au moins une imprimante).

La deuxième variante de la contrainte de partitionnement, appelée *proper-forest*, spécifie que le graphe doit être une forêt contenant `NTREE` *arbres propres*, comme le définit A. Cayley en 1889 [Cay89]. Un arbre propre est un graphe connexe, sans cycle, contenant au moins deux sommets. Remarquons que dans le cas de la contrainte *proper-forest* la notion de ressource n'existe pas (ou, de manière équivalente on peut considérer que tout sommet est une ressource). Elle peut être utilisée pour la conception de réseaux insensibles aux coupures, par exemple, chaque réseau doit contenir au moins deux ordinateurs de sorte

qu'ils puissent se soutenir l'un l'autre. La contrainte *proper-forest* est plus complexe que son pendant avec ressources. En effet, nous détaillerons un algorithme de filtrage ayant une complexité en $O(mn)$, dominée par la complexité de déterminer les arêtes du graphe qui appartiennent à au moins un couplage de cardinalité maximum. Comme nous le verrons, ce cas se produit lorsque le domaine de `NTREE` est fixé à une certaine valeur. Dans tous les autres cas, l'algorithme est basé sur la recherche d'un couplage de cardinalité maximum dans le graphe, ce qui est effectué avec une complexité en $O(m\sqrt{n})$.

Comme ces deux contraintes mettent en jeu aussi bien des variables entières que des variables ensemblistes, leurs algorithmes de filtrage garantissent la consistance-hybride. La section 3.1 introduit les contraintes *resource-forest* et *proper-forest*. Les sections 3.2 page 32 et 3.3 page 38 présentent respectivement les algorithmes de filtrage des contraintes *resource-forest* et *proper-forest*.

3.1 Les contraintes *resource-forest* et *proper-forest*

Dans cette section, nous introduisons et motivons les contraintes *resource-forest* et *proper-forest*, nous définissons les graphes correspondants, et fournissons des exemples qui illustrent la sémantique de chaque contrainte ainsi que la problématique du filtrage dans le cadre de la consistance-hybride.

Dans beaucoup de problèmes de partitionnement de graphe, l'ensemble des sommets du graphe à partitionner est constitué, d'une part, par un ensemble de sommets *ressources* et d'autre part, par un ensemble de sommets *tâches*. Indépendamment du patron utilisé pour partitionner le graphe, cette distinction entre deux types de sommet traduit le besoin de garantir l'existence d'au moins un sommet ressource dans chaque partition. Cette distinction entre les sommets ressources et les sommets tâches fut initialement introduite dans le cadre de la contrainte *cycle* [BC94]. Un exemple d'application pour la contrainte *cycle* est le problème de planification de tournées de véhicules qui consiste à allouer un ensemble de camions (ressources) livrant des marchandises à un ensemble de magasins (tâches) en attente de produits. Dans notre cas, la contrainte *resource-forest* peut être utilisée pour modéliser le problème d'allocation de ressources matérielles partagées dans un réseau. Ici, une ressource correspond à un type de matériel informatique qui doit être partagé sur un réseau (par ex., une imprimante) et une tâche représente le matériel susceptible d'utiliser cette ressource (par ex., un ordinateur). Une solution (c.-à-d., une partition du graphe initial par une forêt) est un réseau dans lequel chaque ordinateur sera connecté à au moins une imprimante.

En 1889, A. Cayley [Cay89] introduisit la définition formelle d'un arbre comme celle d'un graphe connexe, sans cycle, contenant au moins deux sommets. Dans la suite, nous appellerons les arbres définis par Cayley des *arbres propres*. Alors, une *forêt propre* sera définie comme un ensemble fini d'arbres propres. Ainsi, la contrainte *proper-forest* partitionne les sommets d'un graphe non-orienté en un ensemble de sommets disjoints formant des arbres propres. Plus formellement, la figure 3.1 présente la modélisation des contraintes *resource-forest* et *proper-forest*.

Les contraintes *resource-forest* et *proper-forest* sont définies par une variable entière `NTREE` ainsi que par la collection `VER` des n sommets $v_1, v_2, \ldots, v_n$ du graphe à partitionner. À chaque sommet v_i est associé un ensemble d'attributs définis de la manière suivante :
- `L` est un entier compris entre 1 et n qui peut être interprété comme le *nom* du sommet v_i ;
- `N` est une variable ensembliste dont les éléments sont des entiers (c.-à-d., les noms de sommets) compris entre 1 et n. Les bornes inférieures et supérieures de `N` peuvent être respectivement interprétées comme l'ensemble des *voisins obligatoires* et l'ensemble des *voisins possibles ou obligatoires* du sommet v_i ;
- `R` (seulement dans le cas de la contrainte *resource-forest*) est un booléen qui est vrai si le sommet v_i est un sommet ressource et faux sinon ;

Fig. 3.1 – Définition des paramètres des contraintes *resource-forest* et *proper-forest*.

Notation 4 (Attributs associés à une contrainte d'arbre)**.** *Quel que soit un entier i compris entre* 1 *et* n, `VER`$[i]$ *désigne le i-ème item de la collection* `VER`*, alors que* `VER`$[i]$`.L`, `VER`$[i]$`.N`, *et* `VER`$[i]$`.R` *représentent respectivement les attributs* `L`, `N` *et* `R` *de* `VER`$[i]$.

Lorsque l'on considère des contraintes globales, il est très souvent plus facile de raisonner directement sur le graphe modélisant la contrainte globale en question que de raisonner directement sur ses variables (voir également les contraintes *cycle* [BC94], *path* [Sel02, Sel03], et *alldifferent* [Rég94]). Dans le cadre des contraintes *resource-forest* et *proper-forest*, le modèle de graphe est direct (voir définition 3 page 20) : il s'agit d'un graphe non-orienté $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ dans lequel les sommets représentent des éléments de VER et les arêtes représentent la relation de voisinage entre ces derniers. Dans toutes les figures qui suivent, chaque arête du graphe est *typée* (solide ou pointillé) pour indiquer si elle représente une relation de voisinage obligatoire (solide) ou potentielle (pointillé).

Sachant qu'il s'agit d'un pré-traitement linéaire dans le nombre de sommets du graphe $\mathcal{G}$, nous supposerons dans la suite de ce chapitre que le graphe $\mathcal{G}$ associé ne contient aucune boucle et qu'il est symétrique, c.-à-d., $i \in \mathtt{VER}[j].\underline{\mathtt{N}} \Leftrightarrow j \in \mathtt{VER}[i].\underline{\mathtt{N}}$ (dans ce cas nous dirons que i et j sont des *voisins obligatoires*) et $i \in \mathtt{VER}[j].\overline{\mathtt{N}} \Leftrightarrow j \in \mathtt{VER}[i].\overline{\mathtt{N}}$ (dans ce cas, si i et j ne sont pas des voisins obligatoires alors se sont des *voisins potentiels*). Remarquons que cette étape de pré-traitement permet de découvrir que la contrainte n'a pas de solution. Ceci peut se produire si $i \in \mathtt{VER}[i].\underline{\mathtt{N}}$ (il y a une boucle obligatoire) ou si $\exists i, j : i \in \mathtt{VER}[j].\underline{\mathtt{N}} \wedge j \notin \mathtt{VER}[i].\overline{\mathtt{N}}$ (i est un voisin obligatoire de j mais j n'est pas un voisin possible de i). Formellement, le graphe associé aux contraintes *resource-forest* et *proper-forest* est défini de la manière suivante :

Définition 29. *Que l'on considère une contrainte* resource-forest *ou une contrainte* proper-forest*, le* graphe associé *est le graphe non-orienté* $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ *pour lequel* $\mathcal{V} = \{v_i : i \in [1, n]\}$ *et* $(v_i, v_j) \in \mathcal{E}$ *ssi* $i \in \mathtt{VER}[j].\overline{\mathtt{N}} \ \wedge \ j \in \mathtt{VER}[i].\overline{\mathtt{N}}$*. Nous distinguons les arêtes solides de celles en pointillé de la manière suivante : l'arête* $(v_i, v_j) \in \mathcal{E}$ *est* solide *si* i *et* j *sont des voisins obligatoires et elle sera en* pointillé *si* i *et* j *sont des voisins possibles. Pour finir, nous noterons le nombre d'arêtes,* $|\mathcal{E}|$*, dans le graphe par* m*.*

Dans le cas de la contrainte resource-forest*, nous ferons une distinction entre les sommets ressources et les sommets tâches ; l'ensemble des sommets ressources* R *est* $\{v_i : \mathtt{VER}[i].\mathtt{R} = \text{true}\}$*. Tout sommet dans* $\mathcal{V} \setminus \mathtt{R}$ *est un sommet tâche.*

La contrainte *resource-forest* spécifie que le graphe associé doit être une forêt dans laquelle chaque arbre contient au moins une ressource. La contrainte *proper-forest* spécifie, quant à elle, que le graphe associé doit être une forêt propre. Formellement, on dira :

Définition 30. *Une contrainte* resource-forest(NTREE, VER) *est satisfaite ssi les conditions suivantes sont vérifiées :*

1. $\forall i \in [1, n] : \mathtt{VER}[i].\mathtt{L} = i$ *;*
2. $\forall i, j \in [1, n] : i \in \mathtt{VER}[j].\mathtt{N} \Leftrightarrow j \in \mathtt{VER}[i].\mathtt{N}$ *(c.-à-d., la relation de voisinage est symétrique) ;*
3. *Le graphe* $\mathcal{G}$ *associé contient* NTREE *composantes connexes telles que chacune contient au moins un sommet de* R *et ne contient pas de cycles.*

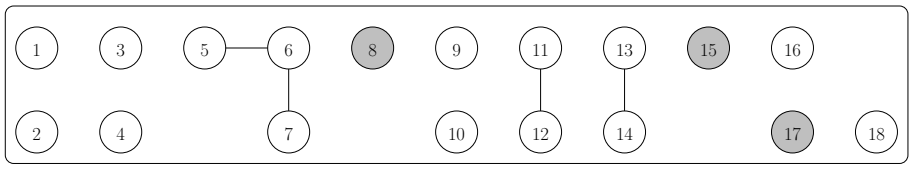
Définition 31. *Une contrainte* proper-forest(NTREE, VER) *est satisfaite ssi les conditions suivantes sont vérifiées :*

1. $\forall i \in [1, n] : \mathtt{VER}[i].\mathtt{L} = i$ *;*
2. $\forall i, j \in [1, n] : i \in \mathtt{VER}[j].\mathtt{N} \Leftrightarrow j \in \mathtt{VER}[i].\mathtt{N}$ *(c.-à-d., la relation de voisinage est symétrique) ;*
3. *Le graphe* $\mathcal{G}$ *associé est une forêt de* NTREE *arbres propres disjoints.*

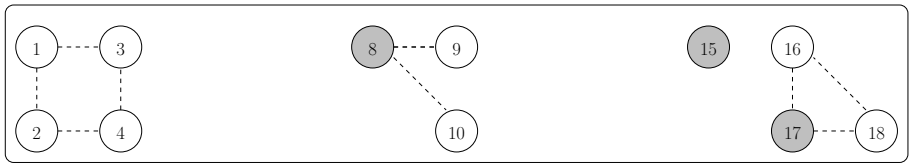
Avant de pouvoir décrire les algorithmes de filtrage des contraintes *resource-forest* et *proper-forest*, nous introduisons le graphe obligatoire $\mathcal{G}_{sure}$ et le graphe possible $\mathcal{G}_{maybe}$ associé au graphe $\mathcal{G}$. Un exemple est fourni dans la Figure 3.2 page suivante.

Définition 32. *(*Graphe obligatoire*) Étant donné une contrainte* resource-forest *ou* proper-forest *et son graphe associé* $\mathcal{G}$*, le graphe* $\mathcal{G}_{sure}$ *contient toutes les arêtes qui doivent être dans la forêt. Formellement,* $\mathcal{G}_{sure} = (\mathcal{V}, \mathcal{E}_{sure})$*, où* $\mathcal{E}_{sure}$ *est l'ensemble des arêtes solides de* $\mathcal{G}$*.*

Définition 33. *(*Graphe possible*) Étant donné une contrainte* resource-forest *ou* proper-forest *et son graphe associé* $\mathcal{G}$*, le graphe* $\mathcal{G}_{maybe}$ *contient le sous-graphe induit par les sommets qui ne sont pas incidents aux arêtes obligatoires. Formellement,* $\mathcal{G}_{maybe} = (\mathcal{V}_{maybe}, \mathcal{E}_{maybe})$*, où* $\mathcal{V}_{maybe}$ *contient tous les sommets qui sont isolés dans* $\mathcal{G}_{sure}$ *et* $\mathcal{E}_{maybe} = \mathcal{E} \cap (\mathcal{V}_{maybe} \times \mathcal{V}_{maybe})$*.*



(a) Le graphe $\mathcal{G}_{sure}$ associé avec le graphe de la Figure 3.3(a) page suivante.



(b) Le graphe $\mathcal{G}_{maybe}$ associé avec le graphe de la Figure 3.3(a) page ci-contre.

FIG. 3.2 – Les graphes $\mathcal{G}_{sure}$ et $\mathcal{G}_{maybe}$ associés avec le graphe de la figure 3.3(a) page suivante.

L'exemple suivant sera utilisé dans la première partie de ce chapitre pour illustrer les contraintes *resource-forest* et *proper-forest*.

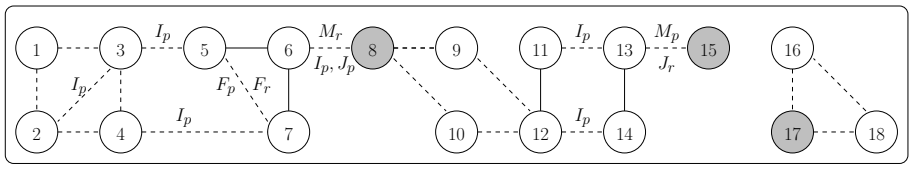
Exemple 1. *La figure 3.3(a) page suivante montre le graphe $\mathcal{G}$ donné en entrée, où les arêtes obligatoires sont marquées d'un trait solide et les autres sont marquées en pointillé. Les figures 3.3(b) page ci-contre et 3.3(c) page suivante montrent deux solutions possibles pour la contrainte* resource-forest*, la première avec deux arbres et la deuxième avec trois. Dans les deux cas, la contrainte* resource-forest *est effectivement satisfaite car chacun des arbres contient au moins un sommet ressource (c.-à-d., les sommets grisés* $8, 15$ *et* 17*). Les Figures 3.3(b) page ci-contre et 3.3(d) page suivante montrent deux solutions possibles pour la contrainte* proper-forest*, avec respectivement deux et sept arbres propres (c.-à-d., aucun des arbres ne correspond à un sommet isolé).*

Un algorithme atteignant la consistance-hybride pour la contrainte resource-forest *sur $\mathcal{G}$ devrait découvrir qu'au vu des valeurs présentes dans le domaine de* NTREE*, l'arête marquée par M_r, c.-à-d., l'arête* $(6, 8)$*, est obligatoire. En effet, l'absence de l'arête* $(6, 8)$ *crée une composante connexe sans sommet ressource. D'autre part, l'arête* $(5, 7)$ *(étiquetée par F_r) est interdite car elle fermerait un cycle dans $\mathcal{G}$. De plus, le domaine de* NTREE *devrait être restreint à l'intersection entre ses valeurs précédentes et l'ensemble* $\{2, 3\}$*. En effet, il existe exactement* 2 *composantes connexes dans $\mathcal{G}$ ainsi que* 3 *sommets ressources. Si, en entrée, $\mathcal{D}$(*NTREE*) =* $\{2\}$*, l'algorithme devrait également découvrir que l'arête marquée par J_r, c.-à-d., l'arête* $(13, 15)$*, est obligatoire. En effet, cette arête est un isthme dans le graphe $\mathcal{G}$. La Section 3.2 justifiera ce filtrage par la suite.*

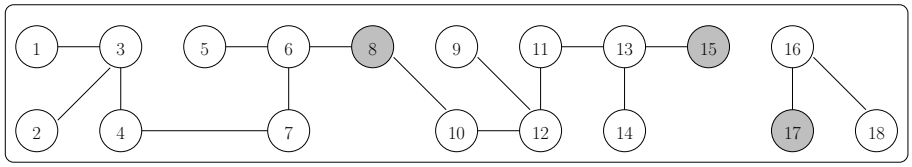
Un algorithme atteignant la consistance-hybride pour la contrainte proper-forest *sur $\mathcal{G}$ devrait découvrir que l'arête* $(13, 15)$ *(étiquetée par M_p) est obligatoire car il s'agit d'un isthme du graphe $\mathcal{G}$ et que par définition, on ne peut pas réduire un arbre propre à un sommet unique. D'autre part, l'arête* $(5, 7)$ *(étiquetée par F_p) est interdite car elle fermerait un cycle dans $\mathcal{G}$. Ensuite, cet algorithme devrait restreindre le domaine de* NTREE *à l'intersection entre ses valeurs précédentes et l'ensemble* $\{2, 3, 4, 5, 6, 7\}$*. En effet, le nombre minimum d'arbres autorisés est directement lié au nombre de composantes connexes dans $\mathcal{G}$ (2 dans notre cas) et le nombre maximum d'arbres autorisés peut être obtenu en calculant la somme entre le nombre de composantes connexes de $\mathcal{G}_{sure}$ de taille au moins* 2 *et le nombre maximum d'arbres de taille* 2 *constructibles dans $\mathcal{G}_{maybe}$. Si, en entrée, $\mathcal{D}$(*NTREE*) =* $\{7\}$*, l'algorithme devrait également découvrir que les arêtes marquées par I_p, c.-à-d.,* $(2, 3)$*,* $(3, 5)$*,* $(4, 7)$*,* $(6, 8)$*,* $(11, 13)$ *et* $(12, 14)$ *sont interdites. Par la suite, la Section 3.3 page 38 justifiera ce dernier filtrage.*

3.2 Filtrer la contrainte *resource-forest*

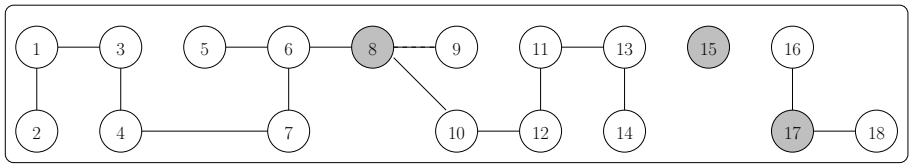
Dans cette section nous étudions la contrainte *resource-forest*. Tout d'abord, la section 3.2.1 page ci-contre détaille une condition nécessaire et suffisante pour l'existence d'une solution satisfaisant cette contrainte. Ensuite les sections 3.2.2 page 34 et 3.2.3 page 35 introduisent et démontrent que la contrainte *resource-*



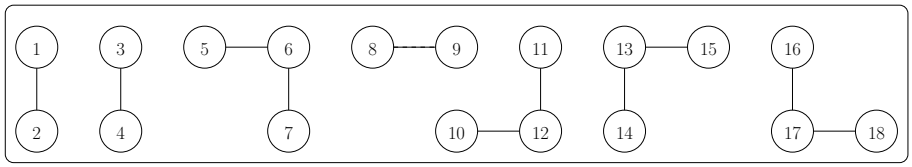
(a) Pour un graphe non-orienté : $J_{r,p}$ arête obligatoire quand max(NTREE) = 2, I_p arête interdite quand min(NTREE) = 7, $F_{r,p}$ arête interdite dans la cas général et, $M_{r,p}$ arête obligatoire dans la cas général.



(b) Une solution avec 2 arbres pour les contraintes *resource-forest* et *proper-forest*.



(c) Une solution avec 3 arbres pour la contrainte *resource-forest*.



(d) Une solution avec 7 arbres pour la contrainte *proper-forest*.

FIG. 3.3 – Dans la figure 3.3(a) certaines arêtes, intervenant dans le filtrage, sont étiquetées par les symboles M_α, F_α, J_α et I_α. L'indice α est fixé à la première lettre de la contrainte à laquelle est lié le filtrage en question (c.-à-d., r pour la contrainte *resource-forest* et p pour la contrainte *proper-forest*). M dénote les arêtes obligatoires, F les arêtes interdites, J les arêtes obligatoires lorsque le nombre maximum d'arbres possibles est atteint, et I les arêtes interdites lorsque le nombre minimum d'arbres possibles est atteint.

-forest peut être filtrée jusqu'à la consistance-hybride. Finalement, la section 3.2.4 page 37 étudie la complexité de l'algorithme de filtrage proposé.

3.2.1 Existence d'une solution pour la contrainte *resource-forest*

Le théorème 2 fournit une condition nécessaire et suffisante quant à l'existence d'une solution pour la contrainte *resource-forest*. Les deux premières conditions de ce théorème assurent qu'il existe une partition du graphe en une forêt telle que chaque arbre contienne au moins un sommet ressource. La troisième condition assure, quant à elle, que le nombre d'arbres de la forêt appartienne au domaine de la variable NTREE.

Théorème 2. *Il existe une solution pour la contrainte* resource-forest(NTREE, VER) *ssi les conditions suivantes sont vérifiées :*

(1) $\mathcal{G}_{sure}$ *ne contient aucun cycle ;*

(2) *toute composante connexe de* $\mathcal{G}$ *contient au moins un sommet ressource ;*

(3) $\mathcal{D}(\texttt{NTREE}) \cap [\texttt{MINTREE}, \texttt{MAXTREE}] \neq \emptyset$*, où* MINTREE *est le nombre de composantes connexes de* $\mathcal{G}$ *et* MAXTREE *est le nombre de composantes connexes de* $\mathcal{G}_{sure}$ *comportant au moins un sommet ressource.*

Démonstration. **Les conditions (1), (2) et (3) sont suffisantes :** pour prouver que les trois conditions sont suffisantes, nous supposons qu'elles sont vérifiées et nous montrons que pour chaque valeur $k \in [\texttt{MINTREE}, \texttt{MAXTREE}]$, nous pouvons construire une forêt couvrante de $\mathcal{G}$ avec k arbres, chacun contenant au moins un sommet ressource.

Cas 1 : $k = \texttt{MAXTREE}$. Soit $\mathcal{T} = \{C_1, \cdots, C_p\}$ les composantes connexes de $\mathcal{G}_{sure}$. Par définition de MAXTREE, exactement MAXTREE d'entre elles contiennent au moins un sommet ressource. Par la condition (2), chaque composante qui ne contient pas un sommet ressource est connectée par une chaîne dans $\mathcal{G}$ à une composante en contenant au moins un. Pour obtenir une solution avec k arbres, nous fusionnons donc chaque composante qui ne contient pas de sommets ressources avec une qui en contient un, et nous retournons un arbre couvrant chaque composante.

Cas 2 : $k < \texttt{MAXTREE}$. Nous construisons d'abord une forêt de MAXTREE arbres comme dans le cas 1 et, ensuite, nous fusionnons deux arbres jusqu'à ce qu'il ne reste plus que k arbres : tant qu'il y a trop d'arbres, nous en sélectionnons deux qui sont connectés par une arête e et nous les fusionnons en ajoutant e à la forêt. Comme MINTREE est le nombre de composantes connexes dans $\mathcal{G}$, tant que k est strictement supérieur à MINTREE nous avons la garantie de trouver deux arbres qui peuvent être fusionnés.

Les conditions (1), (2) et (3) sont nécessaires : si $\mathcal{G}_{sure}$ contient un cycle, la solution ne peut pas être une forêt. S'il existe une composante connexe de $\mathcal{G}$ ne contenant pas de sommet ressource alors cette dernière ne pourra pas être rattachée par une chaîne à un sommet ressource et l'on n'aura donc pas de solution. Si $\mathcal{D}(\texttt{NTREE}) \cap [\texttt{MINTREE}, \texttt{MAXTREE}] = \emptyset$ alors nous avons $\max(\texttt{NTREE}) < \texttt{MINTREE}$ ou $\min(\texttt{NTREE}) > \texttt{MAXTREE}$. Dans chaque cas, la contrainte est insatisfiable : en effet, nous ne pouvons pas créer moins de MINTREE arbres car un arbre doit être connexe. Maintenant nous montrons que l'on ne peut pas créer plus de MAXTREE arbres. Remarquons d'abord qu'une composante connexe de $\mathcal{G}_{sure}$ ne peut pas être cassée, donc chaque composante de $\mathcal{G}_{sure}$ ne peut pas appartenir à plus d'un arbre. De plus, les sommets d'une composante ne contenant pas un sommet ressource doivent appartenir au même arbre que les sommets d'une composante contenant un sommet ressource. □

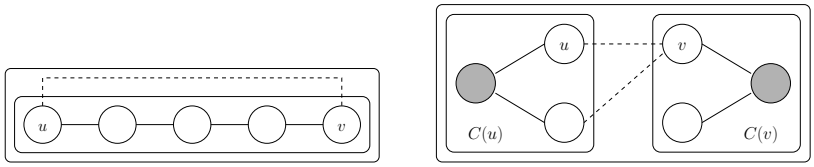
3.2.2 *Consistance-Hybride* pour la contrainte *resource-forest*

L'algorithme 1 de filtrage pour la contrainte *resource-forest* atteint la consistance-hybride. Dans un premier temps, cet algorithme vérifie que la contrainte possède au moins une solution, en utilisant la caractérisation fournit par le théorème 2 page précédente (voir ligne 1). Les lignes 2 à 6 effectuent un filtrage en accord avec le fait qu'une solution ne doit pas comporter de cycles (tout en ignorant le nombre d'arbres de la forêt et le fait que chaque arbre doit contenir au moins un sommet ressource). À la ligne 6 l'algorithme supprime toute arête en pointillé (u, v) où u et v sont connectés par une suite d'arêtes solides ; comme les arêtes solides doivent figurer dans toute solution, cette arête en pointillé créerait un cycle (par ex., l'ajout de l'arête en pointillé $(5, 7)$ de la figure 3.3(a) page précédente créerait le cycle d'arêtes solides $\{(5, 7), (7, 6), (6, 5)\}$). Les lignes 7 à 8 identifient les arêtes en pointillé qui doivent être dans une solution car leurs retraits créeraient un groupe isolé de sommets (c.-à-d., une composante connexe) sans sommet ressource, et les basculent en arêtes solides (par ex., l'arête $(6, 8)$ de la figure 3.3(a) page précédente). Pour finir, la ligne 9 met à jour le domaine de la variable NTREE en utilisant la condition (3) du théorème 2 page précédente.

Les lignes 10 à 13 sont exécutées uniquement lorsque la variable NTREE est fixée. Dans ce cas, le nombre d'arbres dans une solution est fixé, et s'il est égal à MINTREE (comme définit dans le Théorème 2 page précédente), tous les isthmes de $\mathcal{G}$ sont obligatoires et sont transformés en arêtes solides. En effet, dans le cas contraire, le nombre de composantes connexes du graphe, et par conséquent le nombre d'arbres de toute solution, serait strictement supérieur à MINTREE (par ex., l'arête $(13, 15)$ de la figure 3.3(a) page précédente devient obligatoire si l'on doit avoir exactement $\texttt{MINTREE} = 2$ arbres). D'un autre côté, si la valeur de NTREE est fixée à MAXTREE, alors les composantes connexes de $\mathcal{G}_{sure}$ comportant au moins un sommet ressource ne peuvent être fusionnées entre elles sous peine de perdre un arbre et par conséquent de ne plus pouvoir atteindre MAXTREE. Ainsi une arête en pointillé entre deux composantes de ce type ne doit pas être dans la forêt et est donc supprimée.

Algorithme 1 Algorithme de consistance-hybride pour la contrainte *resource-forest*.

1. **Si** la contrainte n'a pas de solution (Théorème 2 page 33) **alors** Sortir sur un échec ;
2. Calculer les composantes connexes (CC) de $\mathcal{G}_{sure}$;
3. **Pour chaque** $v \in \mathcal{V}$ **faire**
4. $C(v) \leftarrow$ la CC de $\mathcal{G}_{sure}$ contenant v ;
5. **Pour chaque** arête $(u, v) \in \mathcal{E}$ en pointillé **faire**
6. **Si** $C(u) = C(v)$ **alors** supprimer (u, v) du graphe ;
7. **Pour chaque** arête e en pointillé **faire**
8. **Si** le retrait de e crée une CC de $\mathcal{G}$ sans sommet ressource **alors** Basculer e en arête solide ;
9. $\mathcal{D}(\texttt{NTREE}) \leftarrow \mathcal{D}(\texttt{NTREE}) \cap [\texttt{MINTREE}, \texttt{MAXTREE}]$;
10. **Si** $\mathcal{D}(\texttt{NTREE}) = \{\texttt{MINTREE}\}$ **alors**
11. **Pour chaque** arête e en pointillé qui est un isthme de $\mathcal{G}$ **faire** Basculer e en arête solide ;
12. **Si** $\mathcal{D}(\texttt{NTREE}) = \{\texttt{MAXTREE}\}$ **alors**
13. **Pour chaque** arête en pointillé $e = (u, v)$ pour laquelle $C(u) \neq C(v)$ et, $C(u)$ et $C(v)$ contiennent un sommet ressource **faire** Supprimer e du graphe ;



(a) (b)

FIG. 3.4 – Illustration de la preuve du lemme 1 : figure 3.4(a), les sommets u et v sont connectés par des arêtes solides. L'arête en pointillé (u, v) crée un cycle, par conséquent, elle est impossible. Elle est supprimée à la ligne 6 de l'algorithme 1. Figure 3.4(b), l'arête (u, v) connecte deux composantes connexes contenant chacune au moins un sommet ressource. Si $\mathcal{D}(\texttt{NTREE}) = \{\texttt{MAXTREE}\} = \{2\}$, cette arête est supprimée à la ligne 13. En effet, si elle était forcée, la solution ne comporterait qu'un seul arbre.

3.2.3 Correction et complétude

Nous prouvons que l'algorithme 1 atteint le niveau de consistance-hybride en démontrant les trois conjectures suivantes :

1. aucune arête en pointillé du graphe $\mathcal{G}$, ni aucune valeur de $\mathcal{D}(\texttt{NTREE})$ n'est supprimée si elle appartient à au moins une solution (lemme 1) ;
2. chaque arête en pointillé et chaque valeur de $\mathcal{D}(\texttt{NTREE})$ restante participent à au moins une solution ; chaque arête en pointillé est exclue d'au moins une solution (lemme 2 page suivante) ;
3. chaque arête en pointillé transformée en arête solide appartient à toute solution. (lemme 3 page 37)

Lemme 1. *L'algorithme 1 ne supprime aucune arête en pointillé du graphe, ni aucune valeur du domaine de la variable* NTREE *appartenant à une solution.*

Démonstration. Supposons qu'il existe une solution S contenant une arête (u, v) supprimée à la ligne 6. Alors, il existe une chaîne composée d'arêtes solides depuis u jusqu'à v (figure 3.4(a)), et ses arêtes sont aussi dans la solution. Mais alors, la forêt ainsi construite contient un cycle : c'est une contradiction. Si une arête en pointillé (u, v) était supprimée à la ligne 13 alors, $\mathcal{D}(\texttt{NTREE}) = \{\texttt{MAXTREE}\}$, c'est à dire que le nombre d'arbres composant la forêt serait égal au nombre de composantes connexes de $\mathcal{G}_{sure}$ contenant un sommet ressource (figure 3.4(b)). De telles composantes connexes ne peuvent être « cassées » dans la solution, et de plus, nous savons qu'elles ne peuvent pas être fusionnées car alors le nombre d'arbres dans la forêt serait inférieur à NTREE. Donc, (u, v) n'appartient à aucune solution. Si l'algorithme supprime une valeur possible de $\mathcal{D}(\texttt{NTREE})$ alors, la troisième condition du théorème 2 page 33 est clairement contredite. □

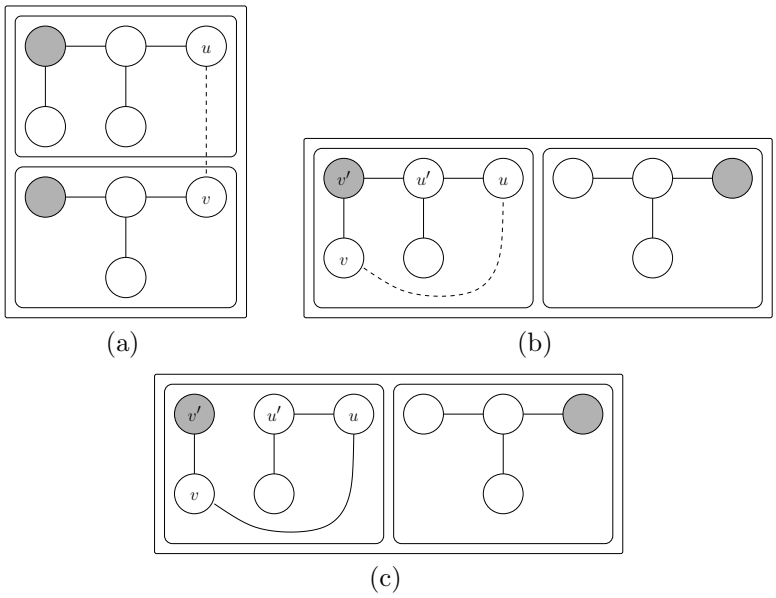


FIG. 3.5 – Illustration de la preuve du lemme 2. figure 3.5(a) : S est la forêt représentée par les arêtes solides et $S' = S \cup (u, v)$ est toujours une forêt. Figures 3.5(b) et 3.5(c) : l'arête (u', v') qui appartient à la solution S est remplacée par l'arête (u, v) pour produire la solution S'.

Lemme 2. *Après avoir appliqué l'algorithme 1 page précédente, chaque arête restante dans $\mathcal{G}$ et chaque valeur du domaine de* NTREE *participe à au moins une solution, et toute arête en pointillé est exclue d'au moins une solution.*

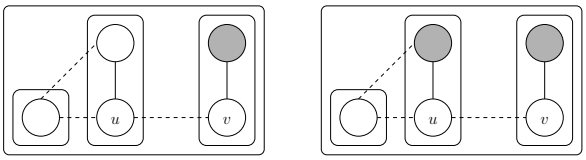
Démonstration. Soit S une solution contenant MAXTREE arbres, obtenue par la construction fournie dans la première partie de la preuve du théorème 2 page 33. Soit (u, v) une arête qui n'a pas été supprimée de $\mathcal{G}$. Si elle appartient à S, nous savons qu'elle participe à au moins une solution. Sinon, c'est une arête en pointillé (car toute arête solide appartient à S). Nous décomposons la preuve en trois assertions relatives à chaque partie du lemme.

Assertion 1 : nous montrons que toute arête restante (u, v) en pointillé participe à au moins une solution.

Cas 1 : si les sommets u et v sont dans des arbres distincts de S alors, $S' = S \cup (u, v)$ est toujours une forêt (c.-à-d., S' ne contient pas de cycles, comme dans la Figure 3.5(a)). Évidemment, chaque arbre de S' contient au moins un sommet ressource car chaque arbre de S contenait un sommet ressource. Si S' ne peut pas être une solution, c'est parce qu'elle contient moins que MINTREE arbres : on sait que S' est composée de MAXTREE $-$ 1 arbres donc, MINTREE $=$ MAXTREE, c.-à-d., le nombre de CC de $\mathcal{G}_{sure}$ contenant un sommet ressource est égal au nombre de CC de $\mathcal{G}$. Ceci implique que chaque CC de $\mathcal{G}$ contient au moins une CC de $\mathcal{G}_{sure}$ qui possède un sommet ressource. Donc, dans la solution S, soit l'arbre contenant le sommet u ne contenait pas de sommets ressource, soit c'était l'arbre contenant v, quoiqu'il en soit cela contredit notre hypothèse que S était une solution : c'est une contradiction.

Cas 2 : maintenant, supposons que les sommets u et v sont dans un même arbre de S (voir figure 3.5(b)). Comme l'arête (u, v) n'a pas été supprimée à la ligne 6, nous savons que u et v ne sont pas dans la même CC de $\mathcal{G}_{sure}$. Donc, dans la chaîne connectant u à v dans S, il y a au moins une arête en pointillé (u', v'). Soit S' une forêt obtenue depuis S en supprimant l'arête (u', v') et en insérant à la place l'arête (u, v), c.-à-d., $S' = (S \setminus \{(u', v')\}) \cup \{(u, v)\}$ (figure 3.5(c)). Toute arête solide appartient toujours à S', S' a le même nombre d'arbres que S et, S' induit la même partition sur les sommets que S. Donc, S' est une solution contenant (u, v).

Assertion 2 : nous montrons que toute valeur restante $k \in \mathcal{D}$(NTREE) appartient à une solution. Si l'algorithme ne change pas le nombre de CC de $\mathcal{G}_{sure}$ contenant un sommet ressource ou le nombre de CC de $\mathcal{G}$, alors le changement ne peut provenir que de l'application du théorème 2 page 33. Supposons qu'il change ces valeurs. Le nombre de CC de $\mathcal{G}_{sure}$ qui ont un sommet ressource ne peut pas augmenter



(a) (b)

FIG. 3.6 – Illustration de la preuve du lemme 3. figure 3.6(a) : (u, v) est obligatoire car sans elle, l'arbre contenant u ne contient pas un sommet ressource. figure 3.6(b) : si $\mathcal{D}(\texttt{NTREE}) = \{1\}$ alors, l'isthme (u, v) est obligatoire.

car les arêtes solides ne sont ni supprimées, ni transformées en arêtes en pointillé. Si ce nombre diminue, c'est parce que deux CC de $\mathcal{G}_{sure}$ ont été fusionnées, et ceci ne peut se produire que si une arête en pointillé qui les connecte est transformée en arête solide. Cependant, une arête en pointillé ne peut être transformée en arête solide que dans deux cas :

- une de ses extrémités appartient à une CC de $\mathcal{G}$ ne contenant pas un sommet ressource ;
- cette arête est un isthme de $\mathcal{G}$ et $\mathcal{D}(\texttt{NTREE}) = \{\texttt{MINTREE}\}$.

Évidemment, cette arête appartient à toute solution avec $k = \texttt{MINTREE}$ arbres. Ainsi, le nombre de CCs de $\mathcal{G}_{sure}$ ne peut pas augmenter car nous n'ajoutons pas d'arêtes dans $\mathcal{G}$. Il ne peut pas non plus diminuer car toute arête supprimée n'est jamais un isthme.

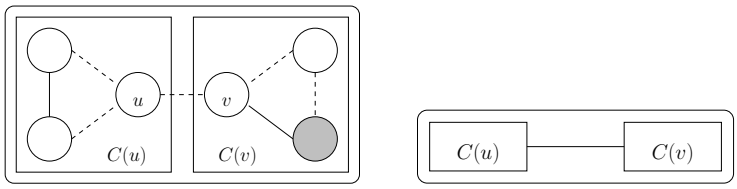
Assertion 3 : nous montrons que chaque arête en pointillé (u, v) est exclue d'au moins une solution. Nous avons montré précédemment qu'il existe une solution utilisant (u, v). Soit $S' = S \setminus \{(u, v)\}$. Si S' est une solution, alors nous savons que (u, v) est exclue d'au moins une solution. Supposons maintenant que S' n'est pas une solution. Ceci peut arriver soit parce qu'elle contient un arbre sans sommet ressource, soit parce qu'elle contient trop d'arbres. Si le retrait de l'arête (u, v) crée un arbre T sans sommet ressource alors, comme (u, v) n'était pas une arête solide créée par la ligne 8, nous savons que cet arbre appartient à une CC qui contient un sommet ressource, donc nous pouvons reconnecter T avec un sommet ressource en ajoutant une arête appropriée à la forêt. Dans l'autre cas, si tout arbre de S' contient un sommet ressource mais que S' est composé de trop d'arbres alors, c'est que le nombre d'arbres dans S est égal à la valeur maximum de $\mathcal{D}(\texttt{NTREE}) \leq \texttt{MAXTREE}$. Supposons qu'il ne soit pas possible de fusionner deux arbres en ajoutant une arête en pointillé autre que (u, v). Alors, le nombre d'arbres dans S est égal au nombre de CC dans $\mathcal{G}$, c.-à-d., MINTREE. Donc, $\mathcal{D}(\texttt{NTREE}) = \{\texttt{MINTREE}\}$. Si (u, v) était un isthme, il aurait été transformé en arête solide à la ligne 11. Donc, il existe un cycle qui contient (u, v). Comme nous ne sommes pas capable de fusionner deux arbres après la suppression de (u, v), c'est que pour chaque arête en pointillé du cycle, toutes les extrémités des arêtes en pointillé appartiennent au même arbre. Ceci implique que u et v sont dans le même arbre après le retrait de (u, v), ce qui signifie qu'il existe un cycle dans la forêt S : c'est une contradiction. □

Lemme 3. *Toute arête en pointillé que l'algorithme 1 page 35 transforme en arête solide participe à toutes les solutions.*

Démonstration. Supposons que cela ne soit pas le cas, c.-à-d., il existe une arête en pointillé (u, v) qui a été transformée en arête solide mais qui est exclue d'au moins une solution S. Si (u, v) a été transformée à la ligne 8 alors, $\mathcal{G}$ possède une CC qui ne contient pas de sommet ressource, donc S n'est pas une forêt valide (figure 3.6(a)). Donc, la transformation en arête solide a été faite à la ligne 11. Dans ce cas, nous savons que la variable NTREE est fixée à MINTREE, c.-à-d., le nombre d'arbres dans S est égal au nombre de CC dans $\mathcal{G}$ (figure 3.6(b)). Mais alors tout isthme de $\mathcal{G}$, et par conséquent l'arête (u, v), doit appartenir à S : c'est une contradiction. □

3.2.4 Complexité

Toutes les étapes de l'algorithme, à l'exception des lignes 7 et 8, requièrent la détection des cycles, le calcul des composantes connexes et l'identification des isthmes. Toutes ces étapes peuvent être effectuées en temps linéaire [GM85, p.18] dans le nombre d'arêtes de $\mathcal{G}$. Nous allons maintenant montrer que les



(a) Le graphe $\mathcal{G}$ et ses deux composantes 2-connexes.

(b) Le graphe réduit $\mathcal{G}_r$ associé à $\mathcal{G}$.

FIG. 3.7 – Le graphe réduit $\mathcal{G}_r$, associé à un grahe $\mathcal{G}$, crée par contraction de chaque composante 2-connexe de $\mathcal{G}$ en un unique sommet.

lignes 7 et 8 peuvent, elles aussi être vérifiées en temps linéaire. Clairement, une arête, dont le retrait crée une composante connexe sans sommet ressource, est un isthme dans $\mathcal{G}$. Mais tous les isthmes n'ont pas cette propriété. Nous créons donc un graphe réduit $\mathcal{G}_r$ en contractant chaque composante 2-connexe de $\mathcal{G}$ en un seul sommet (figure 3.7). Le graphe $\mathcal{G}_r$ est un arbre dont les arêtes sont toutes des isthmes de $\mathcal{G}$. Nous dirons qu'un sommet de $\mathcal{G}_r$ est une ressource si un des sommets de la composante 2-connexe correspondante est une ressource. Nous devons donc identifier quelles arêtes de $\mathcal{G}_r$ sont des arêtes dont le retrait pourrait créer une composante connexe de $\mathcal{G}_r$ sans sommets ressources. En d'autres termes, nous avons réduit notre problème au même problème mais sur des arbres. Ceci peut se produire pour une arête dont un des sommets adjacents est une racine d'un sous-arbre sans ressource. Ainsi, nous sélectionnons arbitrairement un sommet ressource de $\mathcal{G}_r$ et effectuons un DFS de $\mathcal{G}_r$ depuis ce sommet ressource. Chaque fois que l'on fait un retour arrière depuis un sommet v, on communique à son parent p si une ressource a été rencontrée dans le sous-arbre enraciné sur v. Si ce n'est pas le cas, alors l'arête (p, v) est transformée en une arête solide si ce n'était pas déjà le cas. Ainsi, nous venons de montrer que :

Théorème 3. *L'algorithme 1 page 35 filtre la contrainte* resource-forest *jusqu'à la consistance-hybride avec une complexité temporelle de* $O(m + n)$.

3.3 Filtrer la contrainte *proper-forest*

Dans cette section, nous étudions la contrainte *proper-forest*. Tout d'abord, la section 3.3.1 détaille une condition nécessaire et suffisante pour l'existence d'une solution satisfaisant cette contrainte. Ensuite les sections 3.3.2 page ci-contre et 3.3.3 page 40 introduisent et démontrent que la contrainte *proper-forest* peut être filtrée jusqu'à la consistance-hybride. Finalement, la section 3.3.4 page 42 étudie la complexité de l'algorithme de filtrage proposé.

3.3.1 Existence d'une solution pour la contrainte *proper-forest*

Le théorème 4 fournit une condition nécessaire et suffisante d'existence d'une solution pour une contrainte *proper-forest*. Cette condition assure qu'il est possible de partitionner le graphe en une forêt propre et que le nombre d'arbres propres dans cette forêt appartient au domaine de la variable `NTREE`.

Théorème 4. *Il existe une solution pour la contrainte* proper-forest(`NTREE`, `VER`) *ssi les conditions suivantes sont vérifiées :*

(1) *$\mathcal{G}$ n'a pas de sommets isolés (c.-à-d., de composante connexe constituée d'un seul sommet) ;*

(2) *$\mathcal{G}_{sure}$ ne contient aucun cycle ;*

(3) *$\mathcal{D}$(`NTREE`) $\cap$ [`MINTREE`, `MAXTREE`] $\neq \emptyset$, où* `MINTREE` *est le nombre de composantes connexes de $\mathcal{G}$ et* `MAXTREE` *est le nombre de composantes connexes de $\mathcal{G}_{sure}$ de taille au moins deux, plus la taille d'un couplage de cardinalité maximum de $\mathcal{G}_{maybe}$.*

Démonstration. **Les conditions (1), (2) et (3) sont suffisantes :** pour prouver que les trois conditions précédentes sont suffisantes, nous supposons qu'elles sont vérifiées et nous montrons que, pour chaque

valeur k de l'intervalle [MINTREE, MAXTREE], il est possible de construire une forêt couvrante de $\mathcal{G}$ avec k arbres propres. Commençons avec k = MAXTREE et choisissons ensuite une valeur arbitraire k dans [MINTREE, MAXTREE].

- Soit $\mathcal{T} = \{T_1, \cdots, T_p\}$ une forêt couvrante maximale de $\mathcal{G}_{sure}$, c.-à-d., chaque T_i est un sommet isolé dans $\mathcal{G}_{sure}$ ou une forêt couvrante d'une composante connexe de $\mathcal{G}_{sure}$. Remarquons qu'un arbre T_i de taille un est également un sommet de $\mathcal{G}_{maybe}$;
- Pour construire une forêt couvrante de cardinalité MAXTREE, il faut calculer un couplage de cardinalité maximum $\mathcal{M}$ dans $\mathcal{G}_{maybe}$ et modifier la forêt $\mathcal{T}$ de la manière suivante :
  - Par définition de $\mathcal{G}_{maybe}$, chaque arête du couplage maximum $\mathcal{M}$ connecte deux singletons T_i et T_j de $\mathcal{T}$. Alors, il faut les fusionner en un arbre comportant deux sommets ;
  - Pour chaque sommet u de $\mathcal{V}_{maybe}$, correspondant à un arbre T_i de taille un, qui n'est pas saturé dans le couplage maximum $\mathcal{M}$, sélectionner un voisin quelconque v de u et inclure l'arête (u, v) dans la forêt couvrante. Notons que cela n'augmente pas le nombre d'arbres propres de notre forêt car on rattache simplement le sommet u à un arbre propre déjà existant. En effet, le sommet v ne peut être un sommet isolé de $\mathcal{G}_{maybe}$ car cela contredirait le fait que $\mathcal{M}$ soit un couplage maximum. En d'autres termes, fusionner l'arbre T_i avec l'arbre T_v auquel v appartient. La condition (1) garantie que ceci est possible. La forêt obtenue est composée d'exactement MAXTREE arbres ;
- Si k est strictement inférieur à MAXTREE alors, on repart de la construction précédente et l'on fusionne des arbres propres jusqu'à avoir k arbres propres : tant qu'il y a trop d'arbres propres, on en sélectionne deux qui sont liés par une arête e de $\mathcal{G}_{maybe}$ et on les fusionne en incluant e dans la forêt propre. Comme MINTREE est le nombre de composantes connexes de $\mathcal{G}$, tant que k est strictement supérieur à MINTREE nous sommes assurés de trouver deux arbres propres qui peuvent être effectivement fusionnés.

Les conditions (1), (2) et (3) sont nécessaires : clairement, si le graphe $\mathcal{G}$ contient un sommet isolé v, alors v n'appartient pas à un sous-graphe de $\mathcal{G}$ qui est un arbre propre. Si $\mathcal{G}_{sure}$ contient un cycle, la solution doit contenir un cycle donc cela ne pourra pas être une forêt propre. Finalement, si $\mathcal{D}(\text{NTREE}) \cap [\text{MINTREE}, \text{MAXTREE}] = \emptyset$ alors on a $\max(\text{NTREE}) < \text{MINTREE}$ ou $\min(\text{NTREE}) > \text{MAXTREE}$. Dans chacun des deux cas précédents, la contrainte n'a pas de solution : nous ne pouvons, en effet, avoir moins de MINTREE arbres propres car un arbre propre doit être connexe. Pour voir que l'on ne peut pas créer plus de MAXTREE arbres propres, il faut remarquer que le nombre d'arbres propres est au plus le nombre de composantes connexes de $\mathcal{G}_{sure}$ (nous ne pouvons pas casser une composante connexe de $\mathcal{G}_{sure}$) et qu'un sommet de $\mathcal{G}_{maybe}$ peut soit former un nouvel arbre propre avec un autre sommet de $\mathcal{G}_{maybe}$, soit être fusionné avec un arbre propre existant (et donc ne pas augmenter le nombre d'arbres propres). Un plus petit arbre propre étant forcément composé de deux sommets reliés par une arête, un agencement des sommets de $\mathcal{G}_{maybe}$ par paires maximisera le nombre d'arbres propres. Clairement, un couplage de cardinalité maximum dans $\mathcal{G}_{maybe}$ permet de construire un tel agencement et donc de maximiser le nombre d'arbres propres construits. □

3.3.2 *Consistance-Hybride* pour la contrainte *proper-forest*

L'algorithme 2 de filtrage associé à la contrainte *proper-forest* atteint la consistance-hybride. Premièrement, il vérifie que la contrainte possède au moins une solution, en utilisant la caractérisation fournie par le théorème 4 page ci-contre. Les lignes 2 à 5 et 16 à 17 filtrent en accord avec le fait qu'une solution ne doit contenir que des arbres propres (en ignorant le nombre d'arbres propres de la forêt). Les lignes 16-17 identifient les arêtes en pointillé qui doivent être dans la solution car leur retrait isolerait un sommet, et donc il faut les rendre solides (par ex., l'arête $(13, 15)$ de la figure 3.3(a) page 33). À la ligne 5, l'algorithme supprime toute arête (u, v) en pointillé (par ex., l'arête $(5, 7)$ de la figure 3.3(a) page 33) telle que u et v sont connectés par une chaîne d'arêtes solides ; comme les arêtes solides doivent être présentes dans la solution, cette arête en pointillé créerait un cycle. Pour finir, la ligne 6 met à jour le domaine de la variable NTREE en utilisant la condition (3) du théorème 4 page précédente.

Les lignes 7 à 15 sont exécutées seulement lorsque la variable NTREE est fixée. Dans ce cas, le nombre d'arbres propres dans une solution est donc fixé et s'il est égal à MINTREE ou MAXTREE (ces valeurs sont définies dans le théorème 4 page précédente), on peut alors effectuer un filtrage supplémentaire : si NTREE = MINTREE alors tous les isthmes de $\mathcal{G}$ sont obligatoires et sont transformés en arêtes solides, car

Algorithme 2 Algorithme de consistance-hybride pour la contrainte *proper-forest*.

```
1. Si la contrainte n'a pas de solution (voir Théorème 4 page 38) alors Sortir sur un échec ;
2. Calculer les composantes connexes (CC) de G_sure .
3. Pour chaque v ∈ V faire C(v) ← la CC de G_sure contenant v ;
4. Pour chaque arête (u, v) ∈ E en pointillé faire
5.  Si C(u) = C(v) alors supprimer (u, v) du graphe ;
6. D(NTREE) ← D(NTREE) ∩ [MINTREE, MAXTREE] ;
7. Si D(NTREE) = {MINTREE} alors
8.  Pour chaque arête (u, v) en pointillé qui est un isthme de G faire
9.    Basculer (u, v) en arête solide ;
10. Si D(NTREE) = {MAXTREE} alors
11.  Pour chaque arête (u, v) en pointillé faire
12.    Supprimer (u, v) de G si une des conditions suivante est vérifiée :
13       a. |C(u)| > 1, |C(v)| > 1, et C(u) ≠ C(v) ;
14.      b. (u, v) ∈ E_maybe mais n'appartient à aucun couplage maximum dans G_maybe ;
15.      c. |C(u)| > 1 et v est saturé dans tout couplage maximum de G_maybe .
16. Pour chaque arête (u, v) ∈ E en pointillé faire
17.  Si u n'a qu'un voisin v dans G alors Basculer (u, v) en arête solide ;
```

sinon le nombre de composantes connexes du graphe, et donc le nombre d'arbres de toute solution, serait strictement supérieur à MINTREE. Dans l'exemple associé à la figure 3.3(a) page 33, l'arête $(6, 8)$ (marquée par J_p) est obligatoire lorsque $\texttt{NTREE} = 2$. Maintenant, si $\texttt{NTREE} = \texttt{MAXTREE}$ alors trois ensembles d'arêtes sont interdits et sont supprimés du graphe ; nous verrons qu'inclure l'une d'entre elles dans une solution va réduire le nombre d'arbres que l'on peut construire à une valeur inférieure à MAXTREE. Par exemple, si $\texttt{NTREE} = 7$ alors, les arêtes marquées par I_p dans la figure 3.3(a) page 33 sont supprimées : $(11, 13)$ et $(12, 14)$ à la ligne 13, $(2, 3)$ à la ligne 14, et $(3, 5)$, $(4, 7)$ et $(8, 6)$ à la ligne 15.

3.3.3 Correction et complétude

Pour prouver la correction de l'algorithme, nous allons montrer que :

1. Aucune arête en pointillé du graphe n'est supprimée si elle appartient à une solution et aucune valeur de $\mathcal{D}$(NTREE) n'est supprimée s'il existe au moins une couverture de taille correspondante (lemme 4).
2. Chaque arête en pointillé restante dans $\mathcal{G}$ et chaque valeur de $\mathcal{D}$(NTREE) participent à au moins une solution ; chaque arête en pointillé restante est exclue d'au moins une solution (lemme 5 page suivante).
3. Chaque arête en pointillé transformée en arête solide appartient à toute solution (lemme 6 page 42).

Lemme 4. *L'algorithme 2 ne supprime aucune arête en pointillé du graphe, ni aucune valeur du domaine de la variable* NTREE *appartenant à une solution.*

Démonstration. Soit (u, v) une arête qui a été supprimée par l'algorithme 2. Supposons qu'il existe une solution S qui contienne (u, v). Si l'arête (u, v) a été supprimée à la ligne 5, alors il existe une chaîne formée d'arêtes solides depuis u jusqu'à v, et ces arêtes font partie de la solution. Mais alors la forêt S contient un cycle, c'est une contradiction. Il faut donc supposer que l'arête (u, v) a été supprimée aux lignes 10 à 15. Dans ce cas, nous savons que le nombre d'arbres dans S est égal à MAXTREE (le nombre de composantes connexes de $\mathcal{G}_{sure}$ de cardinalité au moins deux plus la taille d'un couplage de cardinalité maximale dans $\mathcal{G}_{maybe}$). Nous verrons que s'il existe toujours une solution S' lorsque la contrainte est appliquée sur un graphe $\mathcal{G}'$, obtenu depuis $\mathcal{G}$ en rendant l'arête (u, v) solide, alors ceci viole la condition (3) du théorème 4 page 38 car S' aurait plus que $\texttt{MAXTREE}'$ arbres (où $\texttt{MAXTREE}'$ est la valeur MAXTREE calculée sur le graphe $\mathcal{G}'$). Si (u, v) était supprimé à la ligne 13, alors $\texttt{MAXTREE}' = \texttt{MAXTREE} - 1$ car u et v ne sont pas dans $\mathcal{G}_{maybe}$ et deux composantes connexes de $\mathcal{G}_{sure}$ ont été fusionnées. La solution $S' = S$ a MAXTREE (i.e., $> \texttt{MAXTREE}'$) arbres. Si (u, v) était supprimé à la ligne 14, alors u et v formerait une composante connexe de taille deux dans $\mathcal{G}'_{sure}$. Ceci augmente MAXTREE de un. D'un autre côté, la taille d'un couplage

de cardinalité maximum dans $\mathcal{G}'_{maybe} = \mathcal{G}_{maybe} \setminus \{u, v\}$ est inférieure de deux à celle de $\mathcal{G}_{maybe}$, car sinon l'arête (u, v) appartiendrait à un couplage de cardinalité maximum. Ainsi, $\texttt{MAXTREE}' = \texttt{MAXTREE} - 1$ et $S' = S$ est une solution avec MAXTREE arbres. Au final, si (u, v) est supprimé à la ligne 15, alors transformer (u, v) en une arête solide ajoute v dans la composante connexe contenant u dans $\mathcal{G}'_{sure}$. Comme v n'est pas dans $\mathcal{G}'_{maybe}$, la taille d'un couplage de cardinalité maximum dans $\mathcal{G}'_{maybe}$ est inférieure de un à celle de $\mathcal{G}_{maybe}$ (sinon $\mathcal{G}_{maybe}$ aurait un couplage de cardinalité maximum dans lequel v ne serait pas saturé, ce qui constituerait une contradiction). Supposons encore que $\texttt{MAXTREE}' = \texttt{MAXTREE} - 1$ et $S' = S$ soit une solution avec MAXTREE arbres. Si l'algorithme supprime une valeur utile de $\mathcal{D}(\texttt{NTREE})$, ceci contredit trivialement le théorème 4 page 38. □

Lemme 5. *Après avoir appliqué l'algorithme 2 page précédente, toute arête restant dans $\mathcal{G}$ et toute valeur restant dans $\mathcal{D}(\texttt{NTREE})$ participent à au moins une solution ; toute arête en pointillé est exclue d'au moins une solution.*

Démonstration. Nous décomposons la preuve en deux assertions respectivement relatives à (1) montrer que toute arête restante dans $\mathcal{G}$ et toute valeur restante dans $\mathcal{D}(\texttt{NTREE})$ participent à au moins une solution, puis (2) montrer que toute arête en pointillé est exclue d'au moins une solution :
Assertion 1 : nous avons déjà montré dans la preuve du théorème 4 page 38 que chaque valeur dans $[\texttt{MINTREE}, \texttt{MAXTREE}]$ participe à au moins une solution. Montrons donc que chaque arête (u, v) restant appartient à une forêt dans au moins une solution. Premièrement, nous construisons une solution S avec MAXTREE arbres comme précédemment. Si (u, v) appartient à la forêt, tout va bien. Sinon, soit $S' = S \cup (u, v)$. Si S' n'est pas une solution pour la contrainte, c'est soit à cause du fait que S' n'est pas une forêt ou soit parce que le nombre d'arbres dans S' n'est pas dans $\mathcal{D}(\texttt{NTREE})$. Si ce n'est pas une forêt, c'est parce que l'ajout de (u, v) a crée un cycle, mais dans ce cas (u, v) aurait été supprimée à la ligne 5. Ainsi, le nombre d'arbres de l forêt S', qui est égal à $\texttt{MAXTREE} - 1$, n'est pas dans $\mathcal{D}(\texttt{NTREE})$. S'il existe une valeur dans $\mathcal{D}(\texttt{NTREE})$ qui est inférieure au nombre d'arbres de S', nous pouvons fusionner des arbres comme nous l'avons montré dans la preuve du théorème 4 page 38 jusqu'à ce que l'on obtienne une solution. Sinon, on doit avoir $\mathcal{D}(\texttt{NTREE}) = \{\texttt{MAXTREE}\}$. Mais dans ce cas, l'arête (u, v) aurait du être supprimée à la ligne 13.
Assertion 2 : nous montrons que chaque arête en pointillé est exclue d'au moins une solution. Soit (u, v) une arête en pointillé. Nous avons prouvé ci-dessus qu'il existe une solution S qui utilise (u, v). Soit $S' = S \setminus \{(u, v)\}$. Si S' est une solution pour la contrainte alors tout va bien. Supposons que S' n'est pas une solution, ceci peut se produire si S' contient un sommet isolé ou si elle est composée d'un trop grand nombre d'arbres :

- Supposons que le retrait de (u, v) crée un arbre contenant l'unique sommet u (c.-à-d., u est un sommet isolé). Comme (u, v) n'a pas été transformée en arête solide à la ligne 17, nous savons que u a un autre voisin $w \neq v$, avec lequel il est lié par une arête en pointillé. Donc, si v n'était pas un arbre de taille 1 ou que le nombre d'arbres était strictement supérieur à MINTREE, nous aurions fusionné u avec un arbre voisin (celui contenant w) et obtenu ainsi une solution ne contenant pas l'arête (u, v). Cependant, si le nombre d'arbres est exactement MINTREE et que à la fois u et v sont des sommets isolés, fusionner chacun d'entre eux dans un arbre existant nous laisserait $\texttt{MINTREE} - 1$ arbres. Heureusement, ce cas n'est pas possible. En effet, supposons que cela se soit produit, nous savons que w (qui est un voisin de u) appartient à un arbre de la solution qui contient au moins deux sommets et qui ne contient pas u ou v. Ainsi, la composante connexe de u (et v) dans $\mathcal{G}$ est présente dans au moins deux arbres de la solution, et la solution doit avoir plus de MINTREE arbres.
- Supposons qu'après le retrait de (u, v) il n'existe pas d'arbres de taille 1 et qu'il n'y ait pas un arbre de trop. Si l'on peut, nous fusionnons deux arbres en utilisant une arête e en pointillé autre que (u, v). Si ce n'est pas possible, alors c'est que le nombre d'arbres est égal au nombre de composantes connexes de $\mathcal{G} \setminus \{e\}$, qui est égal à MINTREE. Si l'arête (u, v) était un isthme alors, il aurait était transformé en une arête solide à la ligne 9. Ainsi, il y a un cycle contenant (u, v). Comme nous ne sommes pas en mesure de fusionner deux arbres après la suppression de (u, v), c'est la même chose pour chaque arête en pointillé sur le cycle, donc les deux extrémités appartiennent au même arbre. Ceci implique que u et v sont dans le même arbre après la suppression de l'arête (u, v), ce qui signifie qu'il existe un cycle dans la forêt S : c'est une contradiction.

□

Lemme 6. *Chaque arête en pointillé que l'algorithme 2 page 40 fait passer à l'état solide appartient à toute solution.*

Démonstration. Supposons qu'il existe une arête (u, v) transformée de pointillé à solide mais qui n'appartient pas à une solution S. Si (u, v) est transformée en arête solide à la ligne 17 de l'algorithme 2 page 40 alors $\mathcal{G}$ a un sommet isolé, ainsi S ne peut pas être une forêt propre. Donc, la transformation a du être effectuée à la ligne 9. Dans ce cas, nous savons que la variable `NTREE` est fixée à `MINTREE`, c.-à-d., le nombre d'arbres dans S est égal au nombre de composantes connexes de $\mathcal{G}$. Mais alors tout isthme de $\mathcal{G}$, et par conséquent l'arête (u, v), doit appartenir à S. Ceci constitue une contradiction. □

3.3.4 Complexité

La complexité pour vérifier l'existence d'une solution satisfaisant la contrainte est dominée par la complexité de calculer la borne `MAXTREE` (tout le reste peut être effectué en temps linéaire). Pour trouver `MAXTREE`, nous devons calculer la cardinalité d'un couplage maximum dans $\mathcal{G}_{maybe}$ et la meilleure borne supérieure connue est $O(m\sqrt{n})$ [MV80]. Les lignes 3 à 7 peuvent être évaluées en temps linéaire sur le nombre d'arêtes : nous devons calculer les CC de $\mathcal{G}_{sure}$ et faire un parcours des arêtes en pointillé. Trouver tous les isthmes de $\mathcal{G}$ à la ligne 8 et détecter les sommets n'ayant pas plus d'un voisin (feuilles) à la ligne 17 peut être aussi effectué en temps linéaire sur le nombre d'arêtes : $O(m)$.

Jusqu'ici la limite de complexité de la contrainte est donnée par le test de faisabilité qui est effectué en $O(m\sqrt{n})$. Si la variable `NTREE` est instanciée à la valeur `MAXTREE`, nous devons exécuter aussi les lignes 10 à 15. La ligne 13 est évidente. Pour la ligne 14, nous devons déterminer quelles arêtes du graphe $\mathcal{G}_{maybe}$ appartiennent à au moins un couplage de cardinalité maximum de $\mathcal{G}_{maybe}$. Pour le cas des graphes bipartis, ceci peut être effectué en temps linéaire dès qu'un couplage maximum est connu [Rég94]. Cependant, dans notre cas nous devons effectuer cette tâche dans le cas d'un graphe quelconque. Nous allons décrire ci-dessous un algorithme qui peut effectuer ceci avec une complexité bornée par $O(mn)$. Finalement, pour la ligne 15 nous avons besoin d'un algorithme qui, partant d'un graphe et d'un couplage maximum sur ce graphe, détecte les sommets saturés dans tout couplage maximum. Nous montrerons que ceci peut être effectué en temps linéaire sur le nombre d'arêtes.

Théorème 5. *L'algorithme 2 page 40 filtre la contrainte* proper-forest *jusqu'à la consistance-hybride avec une complexité borné par $O(mn)$ si $\mathcal{D}(\texttt{NTREE}) = \{\texttt{MAXTREE}\}$ et $O(m\sqrt{n})$ sinon.*

Identifier les sommets saturés dans tout couplage maximum

Soit $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ un graphe non-orienté, soit $\mathcal{M} \subseteq \mathcal{E}$ un couplage de cardinalité maximum dans $\mathcal{G}$, soit $\mathcal{V}(\mathcal{M})$ l'ensemble des sommets qui sont saturés par $\mathcal{M}$ et $\overline{\mathcal{V}(\mathcal{M})} = \mathcal{V} \setminus \mathcal{V}(\mathcal{M})$. Nous allons identifier l'ensemble $\mathcal{S}_v$ des sommets de $\mathcal{G}$ qui sont saturés dans tout couplage maximum de $\mathcal{G}$. Clairement, $\mathcal{S}_v \subseteq \mathcal{V}(\mathcal{M})$.

Lemme 7. *Un sommet $v \in \mathcal{V}(\mathcal{M})$ n'est pas dans $\mathcal{S}_v$ ssi il existe une chaîne alternée P depuis v jusqu'à $u \in \overline{\mathcal{V}(\mathcal{M})}$, quittant le sommet v par une arête saturée.*

Démonstration. (Voir figure 3.8 page suivante) S'il existe une telle chaîne P, la différence symétrique $\mathcal{M} \oplus P$ est un couplage de cardinalité égale à la cardinalité du couplage $\mathcal{M}$ (c.-à-d., un couplage maximum) qui ne sature pas v, donc $v \notin \mathcal{S}_v$.

Dans l'autre sens, supposons que $v \notin \mathcal{S}_v$. Alors, il existe un couplage maximum $\mathcal{M}'$ dans $\mathcal{G}$ qui ne sature pas v. Considérons la différence symétrique $\mathcal{M} \oplus \mathcal{M}'$ entre les deux couplages. Trivialement, le degré d'un sommet de $\mathcal{M} \oplus \mathcal{M}'$ est au plus 2. Il existe une unique arête de $\mathcal{M} \oplus \mathcal{M}'$ adjacente à v, concrètement il s'agit de l'arête saturant v dans $\mathcal{M}$. Nous pouvons donc construire de manière gloutonne une chaîne alternée commençant avec cette arête. Si nous atteignons un sommet de $\overline{\mathcal{V}(\mathcal{M})}$, tout va bien. Tant que nous atteignons un sommet $u \in \mathcal{V}(\mathcal{M})$ par une arête non-saturée, nous pouvons quitter ce sommet par son arête saturée. Maintenant, supposons que nous atteignons un sommet $u \in \mathcal{V}(\mathcal{M})$ par une arête saturée et que l'on ne puisse pas continuer. C'est à dire, que le sommet u n'est pas saturé dans $\mathcal{M}'$. Alors, la chaîne que nous avons construit depuis v jusqu'à u est une chaîne augmentante pour $\mathcal{M}'$, ce qui contredit notre hypothèse que $\mathcal{M}'$ était de cardinalité maximum. □

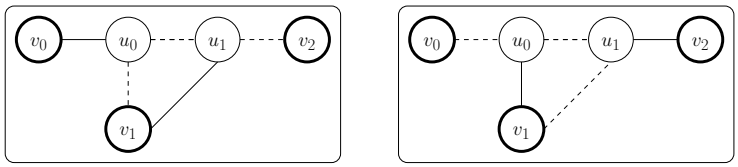


FIG. 3.8 – figure 3.8(a) : $\mathcal{M} = \{(v_0, u_0), (v_1, u_1)\}$ et $\mathcal{V}(\mathcal{M}) = \{u_0, v_0, u_1, v_1\}$. Le sommet v_0 n'est pas saturé dans tout couplage maximum parce qu'il existe une chaîne augmentante $P = \{v_0, u_0, v_1, u_1, v_2\}$ tel que $(v_0, u_0) \in \mathcal{M}$ et $v_2 \notin \mathcal{V}(\mathcal{M})$. Figure 3.8(b) : un couplage maximum qui ne sature pas v_0. De manière similaire, nous déduisons que v_1 et v_2 ne sont pas saturés dans tout couplage maximum.

Algorithme 3 Détection des sommets saturés dans tout couplage maximum de $\mathcal{G}$.

procedure `AlwaysSaturated`$(\mathcal{G}, \mathcal{M})$
1. SAT $= \mathcal{V}(\mathcal{M})$;
2. **Pour chaque** $v \in \mathcal{V}$ **faire**
3. visited_by_matching_edge$(v) \leftarrow no$;
4. visited_by_non_matching_edge$(v) \leftarrow no$;
5. **Pour chaque** `racine` $r \in \overline{\mathcal{V}(\mathcal{M})}$ **faire** `DDFS`$(\mathcal{G}, \mathcal{M}, \text{SAT}, r$`,yes)` ;

procedure `DDFS`$(\mathcal{G}, \mathcal{M}, \text{SAT}, v$`,matching)`
1. **Si** `matching = yes` **alors** `(*` v `est une racine ou` v `est atteint par une arête saturée *)`
2. **Si** visited_by_matching_edge(v) `= yes` **alors** `sortir` ;
3. SAT $\leftarrow$ SAT $\setminus \{v\}$;
4. visited_by_matching_edge$(v) \leftarrow$ `yes` ;
5. **Pour chaque** $(v, u) \in \mathcal{E} \setminus \mathcal{M}$ **faire** `DDFS`$(\mathcal{G}, \mathcal{M}, \text{SAT}, u$`,no)` ;
7. **Sinon** `(*` v `était atteint par une arête non-saturée *)`
8. **Si** visited_by_non_matching_edge(v) `= yes` **alors** `sortir` ;
9. visited_by_non_matching_edge$(v) \leftarrow$ `yes` ;
10. **Si** $\exists (v, u) \in \mathcal{M}$ **alors** `DDFS`$(\mathcal{G}, \mathcal{M}, \text{SAT}, u$`,yes)` ;

L'algorithme initialise SAT $= \mathcal{V}(\mathcal{M})$ et ensuite supprime les sommets de SAT s'il existe une chaîne alternée comme décrit dans le lemme 7 page ci-contre. Les chaines alternées seront identifiées par un « parcours en profondeur d'abord » du graphe, en commençant sur un sommet non-saturé de $\mathcal{M}$ et évoluant sur des chaines alternées. Tant que le parcours atteint un sommet $v \in$ SAT par une arête saturée, l'algorithme supprime v de SAT. Notons que dans le cas d'un parcours en profondeur d'abord classique, chaque sommet est marqué soit « visité », soit « non-visité ». Dans notre cas, nous autorisons uniquement le déplacement le long des chaines alternées. Ainsi, le statut de chaque sommet sera déterminé par deux états possible. Le premier détermine s'il a été visité par une arête saturée et le second s'il a été visité par une arête non-saturée. Remarquons que chaque sommet sera au plus visité deux fois et chaque arête est traversée uniquement lorsque son sommet source est lui-même visité, donc le temps d'exécution est toujours linéaire. Nous appelons cette procédure un « double parcours en profondeur d'abord »(DDFS dans la suite). L'algorithme 3 détaille cette procédure. La racine du DDFS est un sommet non-saturé. Chacun à leur tour les sommets non-saturés deviennent racine, mais observons que les états ne sont pas remis à leur valeur par défaut (c'est le cas dans les parcours en profondeur d'abord avec sources multiples), donc nous ne savons plus quel sommet non-saturé était la racine lorsque plusieurs sommets ont été visités. La procédure récursive DDFS reçoit en paramètres le graphe $\mathcal{G}$, un couplage maximum $\mathcal{M}$, l'ensemble courant SAT, le sommet racine courant v et un paramètre booléen appelée « matching » spécifiant si l'arête par laquelle le sommet a été atteint est saturée ou non. Le lemme suivant assure que l'algorithme 3 identifie correctement l'ensemble $\mathcal{S}_v$ des sommets saturés.

Lemme 8. *Lorsque l'algorithme 3 se termine,* SAT $= \mathcal{S}_v$*, c.-à-d., la variable* SAT *contient l'ensemble des sommets du graphe qui sont saturés dans tout couplage maximum de* $\mathcal{G}$*.*

Démonstration. Pour un sommet $v \notin \mathcal{V}(\mathcal{M})$, il est trivial que v n'est pas dans $\mathcal{S}_v$ et, qu'il n'a jamais été

inséré dans SAT par l'algorithme.

Si $v \in \mathcal{V}(\mathcal{M}) \setminus \mathcal{S}_v$ alors, le lemme 7 page 42 assure qu'il existe une chaîne alternée P depuis un sommet non-saturé u jusqu'à v, tel que P termine par une arête saturée. Il est facile de montrer par induction sur le nombre d'arêtes non-saturées appartenant à P que le sommet v sera visité par la procédure DDFS avec la variable « matching » mise à « vrai », et par conséquent que v sera supprimé de SAT.

Finalement, si $v \in SAT$ alors, $v \in \mathcal{V}(\mathcal{M})$ et donc il est inséré dans SAT. S'il est supprimé plus tard de SAT, c'est parce qu'il est atteint par une arête saturée d'une chaîne alternée depuis un sommet non-saturé. Mais alors, le Lemme 7 page 42 assure que v n'est pas dans $\mathcal{S}_v$: c'est une contradiction. □

Identifier les arêtes qui appartiennent à au moins un couplage maximum

Soit un graphe $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ et soit $\mathcal{M} \subseteq \mathcal{E}$ un couplage maximum de $\mathcal{G}$. Nous souhaitons caractériser l'ensemble des arêtes appartenant à au moins un couplage maximum de $\mathcal{G}$. Nous devons donc trouver les arêtes non-saturées qui appartiennent soit à une chaîne alternée commençant par une arête non saturée et terminant par une arête saturée, soit à un cycle alterné. Pour chaque sommet u, nous devons identifier l'ensemble des sommets qui sont atteignables depuis u par une chaîne alternée élémentaire débutant et finissant sur des arêtes non-saturées. Cette information est calculée à l'aide d'une dérivation de l'algorithme « blossom-shrinking » proposé dans [Edm65], décrite en détail dans [Rég99].

Maintenant, nous décrivons brièvement comment cette information peut être utilisée pour déterminer si une arête non-saturée (x, y) appartient à un cycle alterné. Une telle arête peut appartenir à un cycle alterné seulement si il existe une arête (u, v) telle que $(x, u) \in M$ et $(y, v) \in M$. Supposons qu'il existe une chaîne alternée élémentaire entre u et v qui commence et finit avec des arêtes non-saturées. Comme x et y sont des sommets incidents à des arêtes saturées les connectant avec le sommet final de la chaîne, la chaîne ne peut pas en visiter plus d'un d'entre eux : si la chaîne quitte x (resp. y) par l'arête saturée, il atteint u (resp. v) par une arête saturée. Comme la chaîne doit finir avec une arête non-saturée, il doit visiter u (resp. v) deux fois, c.-à-d., qu'il ne s'agit pas d'une chaîne élémentaire. Par symétrie, la chaîne ne peut pas entrer dans x (resp. y) par une arête saturée. Mais il s'agit bien d'une chaîne alternée, donc il ne peut pas à la fois entrer et sortir par des arêtes non-saturées. Par conséquent, une telle chaîne ne visite pas x et y. En d'autre termes, la chaîne élémentaire et la chaîne (u, x, y, v) se combinent en un cycle élémentaire alterné.

Finalement, s'il existe un cycle alterné contenant (x, y) alors, il doit contenir la chaîne (u, x, y, v). En résumé, un parcours est effectué pour chaque sommet du graphe afin de déterminer quelle paire de sommets sont connectés par une chaîne élémentaire alternée débutant et terminant avec des arêtes non-saturées. Ceci est effectué dans une complexité temporelle de $O(mn)$. À partir de cette information, les arêtes (x, y) non-saturées sont parcourues et, pour chacune on détermine en temps constant s'il existe des arêtes saturées (x, u) et (y, v), et si oui, si la chaîne (u, x, y, v) peut être complétée pour former un cycle élémentaire alterné.

3.4 Synthèse sur les contraintes d'arbres dans le cas non-orienté

Cette section résume les principaux résultats de complexité théorique pour les contraintes de partitionnement de graphe par des arbres dans les graphes non-orientés. Elle rappelle les propriétés de graphes essentielles pour caractériser de manière complète tout arc d'un graphe donné vis-à-vis des contraintes d'arbres. Nous rappelons que ces contraintes sont définies sur un graphe $\mathcal{G} = (\mathcal{V}, \mathcal{E})$, non-orienté, tel que $|\mathcal{V}| = n$ and $|\mathcal{E}| = m$. Dans la suite, les tableaux 3.1 page ci-contre et 3.2 page suivante utilisent les notations suivantes :

Notation 5. *Pour un graphe non-orienté H le nombre de CC maximales au sens de l'inclusion est donné par $|CC(H)|$, un couplage de cardinalité maximum de H est noté $\mu(H)$.*

Le tableau 3.1 page ci-contre résume les meilleures complexités connues pour vérifier la faisabilité et atteindre la consistance-hybride pour chaque contrainte d'arbre. Le tableau 3.2 page suivante résume les principales propriétés de graphes utilisées pour déterminer des bornes réalisables sur le nombre d'arbres autorisés pour couvrir le graphe, ainsi que les conditions d'existence d'arbres « bien formés » vis-à-vis de la définition de chaque contrainte. Ce dernier tableau montre que trois propriétés basiques de la théorie

Type de contrainte	*proper-forest*	*resource-forest*
vérifier la faisabilité	$O(m\sqrt{n})$	$O(n+m)$
consistance-hybride	$O(mn)$ [pire cas], $O(m\sqrt{n})$ [$\mathcal{D}(\texttt{NTREE}) = \{\texttt{MAXTREE}\}$]	$O(n+m)$

TAB. 3.1 – Meilleures complexités connues pour les contraintes de partitionnement par des arbres dans le cas de graphes non-orientés.

des graphes définissent complètement les deux contraintes : les composantes connexes, le couplage de cardinalité maximum et, la détection de cycles. Pour chacune des contraintes, des conditions nécessaires et des règles de filtrage ont été déduites avec des algorithmes connus (par ex., « parcours en profondeur d'abord », couplage de cardinalité maximum, détection de composantes connexes, etc), ainsi que de nouveaux algorithmes (par ex., identification des sommets saturés dans tout couplage de cardinalité maximum). Notons toutefois qu'en terme de propriétés de graphes les bornes `MINTREE` et `MAXTREE` de la contrainte *proper-forest* correspondent parfaitement aux bornes sur le nombre de composantes connexes fournies dans [BPR05]. L'algorithme de filtrage correspondant est également presque automatiquement retrouvé de manière mécanique dans [BCDP07].

Type de graphe	*proper-forest*	*resource-forest*
`MINTREE`	$\lvert CC(\mathcal{G})\rvert$	$\lvert CC(\mathcal{G})\rvert$
`MAXTREE`	$\lvert CC(\mathcal{G}_{sure})\rvert +$ $\lvert \mu(\mathcal{G}_{maybe})\rvert$	$\lvert CC(\mathcal{G}_{sure})$ avec au moins un sommet *ressource*$\rvert$
Arbres « bien formés »	pas de cycles dans $\mathcal{G}_{sure}$ pas de sommets isolés dans $\mathcal{G}$	pas de cycle dans $\mathcal{G}_{sure}$ un sommet *ressource* dans chaque $CC(\mathcal{G}_{sure})$
Nombre d'arbres valides	$\mathcal{D}(\texttt{NTREE}) \cap [\texttt{MINTREE}, \texttt{MAXTREE}] \neq \emptyset$	

TAB. 3.2 – Propriétés de graphe caractérisant les solutions des contraintes de partitionnement par des arbres dans les graphes non-orientés.

Chapitre 4

Contraintes d'arbre dans les graphes orientés

Sommaire

Dans ce chapitre, nous nous intéressons à la contrepartie orientée des contraintes *resource-forest* et *proper-forest* introduites au chapitre 3 page 29. Ces deux nouvelles contraintes, nommées *tree* et *proper-tree*, partitionnent les sommets d'un graphe orienté en un ensemble d'anti-arborescences disjointes (par abus de langage nous appellerons par la suite « arbre » une anti-arborescence). Les partitions compatibles pour les contraintes *tree* et *proper-tree* seront composées d'un ensemble d'arbres, c.-à-d., d'un ensemble de composantes connexes, sans circuits telles que chaque sommet possède un unique successeur, qu'il existe un chemin élémentaire de chaque sommet vers la racine et que la racine soit un sommet portant une boucle sur lui-même. Dans le cas de la contrainte *proper-tree*, c'est le nombre d'arbres propres (c.-à-d., d'arbres composés d'au moins deux sommets) présents dans la partition qui est contraint.

Bien que ces deux contraintes ne permettent pas de modéliser tous les aspects d'un problème, elles interviennent dans de nombreuses applications pratiques en combinaison avec des contraintes additionnelles permettant de restreindre la forme des partitions autorisées. C'est le cas en particulier pour la reconstruction de « super-arbres » en phylogénie, pour la planification de missions, ou pour la construction de tournées de véhicules en logistique. Nous étudierons leurs interactions avec certaines contraintes additionnelles dans le chapitre 5 page 65.

Dans ce chapitre, nous montrons comment l'algorithme de filtrage de la contrainte *tree* garantit d'atteindre l'arc consistance généralisée alors que, dans le cas de la contrainte *proper-tree*, l'évaluation du nombre minimum d'arbres propres est un problème NP-dur. La section 4.1 page suivante introduit les contraintes *tree* et *proper-tree*. Les sections 4.2 page 49 et 4.3 page 53 présentent respectivement les algorithmes de filtrage des contraintes *tree* et *proper-tree*.

4.1 Les contraintes *tree* et *proper-tree*

Dans cette section, nous introduisons plus formellement les contraintes *tree* et *proper-tree* en décrivant dans un premier temps les paramètres de chaque contrainte puis en définissant le graphe permettant de les modéliser. Les contraintes *tree* et *proper-tree* sont définies comme suit (figure 4.1) :

Les contraintes *tree* et *proper-tree* sont respectivement données sous la forme *tree*(NTREE, VER) et *proper-tree*(NPROP, VER), où respectivement NTREE et NPROP sont deux variables entières et VER est la collection des n sommets VER$[1], \ldots,$ VER$[n]$ du graphe à partitionner. À chaque sommet $v_i =$ VER$[i]$ est associé un ensemble d'attributs définis de la manière suivante :
- L est un entier compris entre 1 et n qui peut être interprété comme le nom du sommet v_i ;
- S est une variable entière définie sur l'intervalle $[1, n]$. Elle représente, dans une solution de la contrainte *tree* ou *proper-tree*, l'unique successeur du sommet v_i dans une couverture. Si i appartient au domaine de la variable VER$[i]$.S, alors on dira que v_i est une *racine potentielle*.

FIG. 4.1 – Définition des paramètres des contraintes *tree* et *proper-tree*.

Comme nous l'avons déjà observé dans le cadre des contraintes *resource-forest* et *proper-forest*, lorsque l'on définit le sens d'une contrainte globale, il est bien souvent plus facile de raisonner directement sur un graphe modélisant la contrainte, que de raisonner sur les variables et leurs domaines. Ainsi, les deux contraintes *tree* et *proper-tree* vont être modélisées par le graphe orienté $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ dont les sommets correspondent aux éléments de la collection VER et les arcs représentent la relation de successeur entre eux (voir Définition 4 page 20). Dans le cadre des contraintes *tree* et *proper-tree*, on définit $\mathcal{G}$ de la manière suivante :

Définition 34. *Le* graphe orienté associé $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ *aux contraintes* tree(NTREE, VER) *et* proper-tree(NPROP, VER) *est défini par* $\mathcal{V} = \{v_i \mid i \in [1, n]\}$ *et* $\mathcal{E} = \{(v_i, v_j) \mid j \in \mathcal{D}(\text{VER}[i].\text{S})\}$. $m = |\mathcal{E}|$ *désigne le nombre d'arcs du graphe* $\mathcal{G}$.

Nous sommes maintenant en mesure de définir les notions d'arc sûr et d'arc possible : un arc (i, j) du graphe $\mathcal{G}$ associé à une contrainte *tree* est dit *S-arc* (un *arc sûr*) si $\mathcal{D}(\text{VER}[i].\text{S}) = \{j\}$, sinon (i, j) est dit *M-arc* (un *arc possible*). Un sommet i est dit *S-succ* si tout arc sortant de i est S-arc, sinon i est dit *M-succ*. De la même manière, un sommet i est dit *S-pred* si tout arc entrant est S-arc, sinon i est dit *M-pred*. Ces notations nous permettent d'introduire les notions de graphes des successeurs surs et possibles associés au graphe $\mathcal{G}$:

Définition 35 (Graphes des successeurs sûrs et possibles)**.** *Étant donné le graphe orienté* $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ *associé aux contraintes* tree(NTREE, VER) *et* proper-tree(NPROP, VER) *:*
- *Le graphe des* successeurs surs $\mathcal{G}_{sure}$ *contient tous les arcs qui doivent appartenir à la partition. Formellement,* $\mathcal{G}_{sure} = (\mathcal{V}, \mathcal{E}_{sure})$*, où* $\mathcal{E}_{sure}$ *est l'ensemble des S-arcs de* $\mathcal{G}$ *;*
- *Le graphe des* successeurs possibles $\mathcal{G}_{maybe}$ *contient tous les arcs qui peuvent éventuellement appartenir à une partition. Formellement,* $\mathcal{G}_{maybe} = (\mathcal{V}_{maybe}, \mathcal{E}_{maybe})$*, où* $\mathcal{V}_{maybe}$ *contient tous les sommets qui sont incidents à au moins un M-arc, et* $\mathcal{E}_{maybe}$ *est l'ensemble des M-arcs de* $\mathcal{G}$.

Nous pouvons maintenant définir les propriétés que doit respecter toute solution satisfaisant la contrainte *tree*. Une telle contrainte spécifie que le graphe orienté $\mathcal{G}$ qui lui est associé doit être une forêt composée de NTREE arbres. De manière formelle, on peut définir une solution respectant la contrainte *tree* par :

Définition 36. *Un graphe orienté* $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ satisfait *la contrainte d'arbre* tree(NTREE, VER) *si et seulement si :*

1. *pour chaque* $i \in [1, n]$ *on a* VER$[i]$.L $= i$ *;*
2. *le graphe* $\mathcal{G}$ *est composé de* NTREE *composantes connexes ;*
3. *chaque composante connexe de* $\mathcal{G}$ *ne contient pas de circuits impliquant plus d'un seul sommet.*

De manière similaire, la contrainte *proper-tree* spécifie que le graphe orienté $\mathcal{G}$ qui lui est associé doit être une forêt composée de NPROP arbres propres :

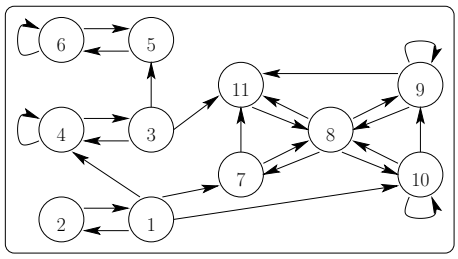
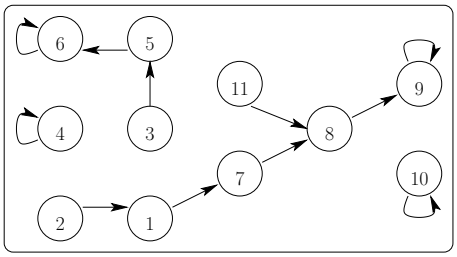


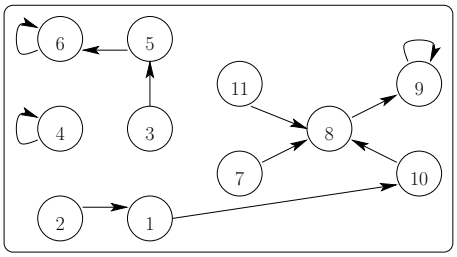
FIG. 4.2 – Graphe $\mathcal{G}$ associé aux contraintes d'arbres *tree* et *proper-tree*.

Label (i)	$\mathcal{D}(\texttt{VER}[i].\texttt{S})$	Label (i)	$\mathcal{D}(\texttt{VER}[i].\texttt{S})$
1	$\{2, 4, 7, 10\}$	7	$\{8, 11\}$
2	$\{1\}$	8	$\{7, 9, 10, 11\}$
3	$\{4, 5, 11\}$	9	$\{8, 9, 11\}$
4	$\{3, 4\}$	10	$\{8, 9, 10\}$
5	$\{6\}$	11	$\{8\}$
6	$\{5, 6\}$		

FIG. 4.3 – La première colonne donne le nom de chaque sommet du graphe à couvrir. La seconde colonne décrit les successeurs potentiels de chaque sommet (c.-à-d., dans la figure 4.2 le successeur de chaque sommet).



(a) Une forêt compatible dans le cas $\texttt{NTREE} = 4$.



(b) Une forêt compatible dans le cas $\texttt{NPROP} = 2$. Le sommet isolé 4 n'est pas compté.

FIG. 4.4 – Deux solutions compatibles dans le cas des contraintes *tree* (a) et *proper-tree* (b).

Définition 37. *Un graphe orienté* $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ satisfait *la contrainte d'arbre* proper-tree($\texttt{NPROP}, \texttt{VER}$) *si et seulement si :*

1. *pour chaque* $i \in [1, n]$ *on a* $\texttt{VER}[i].\texttt{L} = i$ *;*
2. *le graphe* $\mathcal{G}$ *est composé de* NPROP *composantes connexes de taille supérieure ou égale à 2, mis à part les sommets isolés ;*
3. *chaque composante connexe de* $\mathcal{G}$ *ne contient pas de circuits impliquant plus d'un seul sommet.*

Exemple 2. *La figure 4.2 donne un exemple de graphe que l'on cherche à partitionner par des arbres (ou des arbres propres). La table 4.3 décrit ce même graphe en terme de domaines de variables successeur. Les figures 4.4(a) et 4.4(b) fournissent une couverture par une forêt compatible formée respectivement de 4 arbres et 2 arbres propres.*

4.2 Filtrer la contrainte *tree*

Dans cette section nous étudions la contrainte *tree*. Tout d'abord, la section 4.2.1 détaille une condition nécessaire et suffisante pour l'existence d'une solution satisfaisant cette contrainte. Ensuite les sections 4.2.2 page suivante et 4.2.3 page 51 montrent que la contrainte *tree* peut être filtrée jusqu'à atteindre l'arc-consistance généralisée (voir définition 5 page 21). Finalement, la section 4.2.4 page 53 étudie la complexité de l'algorithme de filtrage proposé.

4.2.1 Existence d'une solution pour une contrainte *tree*

Le théorème suivant caractérise les conditions d'existence d'une solution pour une contrainte d'arbre. La première condition assure que l'on peut effectivement créer une forêt couvrante alors que la seconde

condition vérifie qu'il existe au moins une valeur compatible k dans le domaine de NTREE permettant de construire k arbres.

Théorème 6. *Il existe une solution pour la contrainte* tree(NTREE, VER) *ssi les conditions suivantes sont vérifiées :*

(1) *toute composante puits de $\mathcal{G}$ est une composante enracinée de $\mathcal{G}$;*

(2) $\mathcal{D}(\texttt{NTREE}) \cap [\texttt{MINTREE}, \texttt{MAXTREE}] \neq \emptyset$*, où* MINTREE *est le nombre de sommets puits dans le graphe réduit $\mathcal{G}_r$ associé à $\mathcal{G}$ et* MAXTREE *est le nombre de racines potentielles ou fixées dans $\mathcal{G}$.*

Démonstration. On prouve d'abord que la conjonction des conditions (1) et (2) est nécessaire. Pour la condition (1) : si une composante puits de $\mathcal{G}$ n'est pas une composante enracinée alors, il existe au moins un circuit dans $\mathcal{G}$ parmi les sommets associés à cette composante puits et la contrainte ne peut pas être satisfaite. Pour la condition (2) : si $\mathcal{D}(\texttt{NTREE}) \cap [\texttt{MINTREE}, \texttt{MAXTREE}] = \emptyset$ alors, soit max(NTREE) est strictement inférieur à MINTREE, soit min(NTREE) est strictement supérieur à MAXTREE. Dans le premier cas, la contradiction est évidente par le fait qu'il n'existe pas de chemins entre deux sommets appartenant à des composantes puits distinctes de $\mathcal{G}$. Dans le second cas, la contradiction provient du fait que chaque arbre de la forêt possède exactement une racine.

On prouve maintenant que la conjonction des conditions (1) et (2) est suffisante. Pour cela, nous donnons un algorithme en deux étapes, qui pour chaque valeur t de l'intervalle [MINTREE, MAXTREE] construit un partitionnement de $\mathcal{G}$ en t arbres. La première étape sélectionne t racines potentielles et choisit pour chaque composante fortement connexe (*CFC*) de $\mathcal{G}$ le sommet qui sera soit la racine d'un nouvel arbre, soit qui sera une porte liant la *CFC* à une autre. La seconde étape construit pour chaque *CFC* une forêt couvrante dont chaque arbre est enraciné soit sur la racine potentielle choisie, soit sur le sommet porte sélectionné.

ÉTAPE 1

- Choisir une racine potentielle r pour chaque *CFC* puits de $\mathcal{G}$ et forcer la boucle (r, r) sur cette racine. Soit $\mathcal{R}_1$ l'ensemble des racines sélectionnées ;
- Si t est strictement supérieur à MINTREE alors, choisir un ensemble $\mathcal{R}_2$ de $t -$ MINTREE racines potentielles dans $\mathcal{G}$, distinct de $\mathcal{R}_1$, et forcer la boucle sur chaque sommet de $\mathcal{R}_2$;
- Pour chaque *CFC* pour laquelle aucune boucle n'a été forcée, choisir un sommet v qui est un sommet porte et forcer l'un des arcs connectant sortant de v. Soit $\mathcal{R}_3$ l'ensemble des sommets porte ainsi sélectionnés.

ÉTAPE 2

Pour chaque *CFC* $\mathcal{S} = (\mathcal{V}_\mathcal{S}, \mathcal{E}_\mathcal{S})$ de $\mathcal{G}$ soient :

- $\mathcal{H} = \mathcal{V}_\mathcal{S} \cap (\mathcal{R}_1 \cup \mathcal{R}_2 \cup \mathcal{R}_3)$ (c.-à-d., les sommets de $\mathcal{S}$ sélectionnés à l'étape 1) ;
- $\mathcal{L} = \mathcal{V}_\mathcal{S} - \mathcal{H}_\mathcal{S}$ (c.-à-d., le reste des sommets de $\mathcal{S}$).

Pour chaque composante fortement connexe $\mathcal{S}$ de $\mathcal{G}$, l'algorithme 4 page ci-contre $TreeCovering(\mathcal{S}, \mathcal{H}, \mathcal{L})$ construit une partition des sommets de $\mathcal{S}$ en $|\mathcal{H}|$ arbres ayant comme racines les sommets de $\mathcal{H}$ (la partition est renvoyée dans le paramètre de sortie $\mathcal{F}$). Notons que l'algorithme 4 page suivante est correct car comme pour tout sommet $v \in \mathcal{L}$, il existe un chemin vers au moins un sommet de $\mathcal{H}$ (car $\mathcal{S}$ est une *CFC*) alors, on s'assure que l'ensemble $\mathcal{L}$ tend vers l'ensemble vide au fil des itérations de la boucle ligne 2, c.-à-d., que tout sommet de $\mathcal{V}$ est bien couvert par exactement un arbre.

Ainsi, nous venons de montrer comment construire un partitionnement de $\mathcal{G}$ en t arbres, quel que soit t appartenant à l'intervalle [MINTREE, MAXTREE]. De plus, comme $\mathcal{D}(\texttt{NTREE}) \cap [\texttt{MINTREE}, \texttt{MAXTREE}] \subseteq [\texttt{MINTREE}, \texttt{MAXTREE}]$, on a bien au moins une solution pour la contrainte *tree*. □

4.2.2 Arc-consistance généralisée pour la contrainte *tree*

Dans cette section, nous introduisons l'algorithme de filtrage de la contrainte *tree*. Cet algorithme repose essentiellement sur la détection des arcs du graphe $\mathcal{G}$ qui empêchent certains sommets d'atteindre les racines potentielles de $\mathcal{G}$. Intuitivement, on cherche à s'assurer que chaque sommet de $\mathcal{G}$ pourra être couvert par un arbre bien formé, c.-à-d., un arbre enraciné sur une racine potentielle de $\mathcal{G}$.

Algorithme 4 Construire une partition des sommets de $\mathcal{S}$ en $|\mathcal{H}|$ arbres ayant leurs racines dans $\mathcal{H}$.

procedure *TreeCovering*$(\mathcal{S}, \mathcal{H}, \mathcal{L}) : \mathcal{F}$
1. $\mathcal{F} \leftarrow \emptyset$;
2. **Tant que** $\mathcal{L} \neq \emptyset$ **faire**
3. **Si** il existe un arc $(v, h) \in \mathcal{E}_{\mathcal{S}}$ **tel que** $v \in \mathcal{L}$ et $h \in \mathcal{H}$ **alors**
4. $\mathcal{F} \leftarrow \mathcal{F} \cup \{(v, h)\}$;
5. $\mathcal{H} \leftarrow \mathcal{H} \cup \{v\}$;
6. $\mathcal{L} \leftarrow \mathcal{L} \setminus \{v\}$;
7. **Retourner** $\mathcal{F}$;

Algorithme 5 Filtrage associé à la contrainte *tree*(NTREE, VER).

procedure FilteringTree($\mathcal{G}$)
1. **Si** *tree*(NTREE, VER) n'a pas de solution (théorème 6 page ci-contre) **alors** Retourner un échec ;
2. $\mathcal{D}$(NTREE) $\leftarrow \mathcal{D}$(NTREE) $\cap$ [MINTREE, MAXTREE] ;
3. **Si** $\mathcal{D}$(NTREE) = {MAXTREE} **alors** Forcer toutes les boucles de $\mathcal{G}$;
4. **Si** $\mathcal{D}$(NTREE) = {MINTREE} **alors**
5. Supprimer toutes les boucles des composantes non puits de $\mathcal{G}$;
6. $\mathcal{D} \leftarrow$ l'ensemble des sommets dominants les racines potentielles de $\mathcal{G}$;
7. $\mathcal{D} \leftarrow \mathcal{D} \cup$ l'ensemble des racines potentielles de $\mathcal{G}$;
8. Appliquer la procédure FilteringDominators($\mathcal{G}, \mathcal{G}, \mathcal{D}$) décrite par l'algorithme 6 page suivante ;
9. Retourner le graphe $\mathcal{G}$ filtré ;

L'algorithme 5 effectue un filtrage atteignant l'arc-consistance pour la contrainte *tree*. La ligne 1 vérifie si la contrainte possède au moins une solution. Pour cela, une procédure calculant les composantes fortement connexes de $\mathcal{G}$ est nécessaire. La ligne 2 ajuste le domaine de NTREE en utilisant les bornes MINTREE et MAXTREE introduites par le théorème 6 page précédente. La ligne 3 force toutes les boucles sur les racines potentielles de $\mathcal{G}$ lorsque la borne MAXTREE est effectivement atteinte par NTREE. Ensuite, les lignes 4 et 5 interdisent les boucles sur les racines potentielles appartenant à des composantes non puits de $\mathcal{G}$ lorsque la borne MINTREE est atteinte par NTREE. Finalement, les lignes 6 à 9 détectent les sommets dominants les racines potentielles de $\mathcal{G}$, puis pour chaque sommet dominant, un parcours en profondeur d'abord est effectué (dans la procédure FilteringDominators de l'algorithme 6) en vue de détecter les arcs qui pourraient empêcher certains sommets d'être reliés à une racine potentielle. Les figures 4.5 et 4.6 proposent sur un exemple de visualiser l'effet du filtrage proposé par l'algorithme 5.

4.2.3 Correction et complétude

Pour prouver que l'algorithme 5 atteint bien l'arc-consistance, nous montrons que :

1. aucun arc de $\mathcal{G}$ n'est supprimé par l'algorithme 5 s'il appartient à au moins une solution satisfaisant la contrainte *tree*(NTREE, VER) (Lemme 9) ;
2. tout arc restant dans $\mathcal{G}$ après application de l'algorithme 5 participe à au moins une solution satisfaisant la contrainte *tree*(NTREE, VER) (Lemme 10 page suivante).

Lemme 9. *L'algorithme 5 ne retire aucun arc du graphe $\mathcal{G}$ pouvant appartenir à une solution satisfaisant la contrainte* tree(NTREE, VER).

Démonstration. Supposons qu'il existe une solution $\mathcal{S}$ satisfaisant la contrainte *tree*(NTREE, VER) contenant un arc (u, v) supprimé par l'algorithme 5.

Tout d'abord, supposons que (u, v) est supprimé par la ligne 3, c.-à-d., dans le cas où NTREE = MAXTREE. On est donc dans le cas où le sommet u est une racine potentielle de $\mathcal{G}$ et $u \neq v$. Comme (u, v) appartient à S alors, S est une forêt composée de MAXTREE $- 1$ arbres : c'est une contradiction.

Maintenant, supposons que (u, v) est supprimé par les lignes 4 et 5, c.-à-d., dans le cas où NTREE = MINTREE. On est donc dans le cas où $u = v$ et le sommet u appartient à une composante non puits de $\mathcal{G}$. Comme (u, v) appartient à S alors, S est une forêt composée de MINTREE $+ 1$ arbres. En effet, MINTREE

Algorithme 6 Filtrage des successeurs dans $\mathcal{G}$ de l'ensemble des sommets dominants $\mathcal{D}$.

```
/* Filtre les successeurs de chaque sommet dominant d qui ne peut pas */
/* atteindre une racine contenue dans U sans passer par le sommet d */
procedure FilteringDominators(G,U,D) : G
1. Pour chaque sommet d ∈ D faire
2.  Pour chaque sommet v de G_maybe faire
3.    reach_potential_root(v) ← no ;
4.    visited(v) ← no ;
5.  Visit(G,U,d,d) ;
6.  Pour chaque successeur s de d dans G_maybe faire
7.    Si reach_potential_root(s) = no alors supprimer l'arc (d,s) de G_maybe ;
8. Retourner le graphe G filtré ;
/* Marque les successeurs de u pouvant atteindre une racine contenue dans U */
procedure Visit(G,U,d,u) : booléen
9. visited(u) ← yes ;
10. Si u est une racine potentielle de U alors reach_potential_root(u) ← yes ;
11. Pour chaque successeur v de u tel que visited(v) = no faire
12.  reach_potential_root(u) ← Visit(G,U,d,v) ∨ reach_potential_root(u) ;
13. Pour chaque successeur v de u tel que visited(v) = yes et v ≠ d faire
14.  reach_potential_root(u) ← reach_potential_root(u) ∨ reach_potential_root(v) ;
15. Renvoyer reach_potential_root(u) ;
```

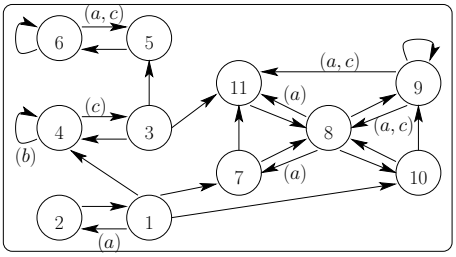


FIG. 4.5 – Effet de l'algorithme de filtrage 5 page précédente sur la figure 4.2 page 49.

Type	Condition sur NTREE	Lignes dans l'algorithme 5
(a)	-	6-8
(b)	NTREE = 2	4-5
(c)	NTREE = 3	3

FIG. 4.6 – Correspondance entre le filtrage décrit par les arcs marqués par les lettres (a), (b) et (c) de la figure 4.5 et l'algorithme de filtrage 5 page précédente.

est donné par le nombre de composantes puits du graphe $\mathcal{G}$, donc forcer l'arc (u, v) revient à forcer une boucle dans une composante non puits : c'est une contradiction.

Finalement, supposons que (u, v) est supprimé par les lignes 6 à 8, c.-à-d. que le sommet u est un sommet dominant toutes les racines potentielles de $\mathcal{G}$ par rapport au sommet v. Cela signifie que l'arbre de S contenant l'arc (u, v) est enraciné sur un sommet sans boucle (car depuis l'arc (u, v), on ne peut plus atteindre de racine sans passer par u) : c'est une contradiction. □

Lemme 10. *Tout arc restant dans le graphe $\mathcal{G}$, après application de l'algorithme 5 page précédente, participe à au moins une solution satisfaisant la contrainte* tree(NTREE, VER).

Démonstration. Soit (u, v) un arc de $\mathcal{G}$ restant après application de l'algorithme 5 page précédente. Soit $\mathcal{G}_{(u,v)}$ le graphe $\mathcal{G}$ dans lequel l'arc (u, v) a été forcé.

Supposons tout d'abord que $\mathcal{G}_{(u,v)}$ viole la condition (1) du théorème 6 page 50. Cela signifie qu'il existe une composante puits dans le graphe $\mathcal{G}_{(u,v)}$ qui ne contient pas de racine potentielle. Donc, si forcer l'arc (u, v) dans $\mathcal{G}_{(u,v)}$ a augmenté le nombre de *CFC* de $\mathcal{G}$ alors, le sommet u était un sommet dominant les racines potentielles de $\mathcal{G}$. Mais alors, dans $\mathcal{G}_{(u,v)}$, le sommet v ne peut plus atteindre de racine potentielle. Donc, l'arc (u, v) aurait été supprimé par la procédure FilteringDominators appelée par la ligne 8 de l'algorithme 5 page précédente : c'est une contradiction.

Supposons maintenant que $\mathcal{G}_{(u,v)}$ viole la condition (2) du théorème 6 page 50. Cela signifie que $\mathcal{D}$(NTREE)∩[MINTREE, MAXTREE] définit un ensemble vide. On peut distinguer deux cas : soit (a) $\mathcal{D}$(NTREE) <

MINTREE, soit (b) $\mathcal{D}(\texttt{NTREE}) > \texttt{MAXTREE}$. Soient $CFC(u)$ et $CFC(v)$ les CFC de $\mathcal{G}$ contenant respectivement les sommets u et v.

Dans le cas (a), remarquons que la valeur de MINTREE dans $\mathcal{G}_{(u,v)}$ a augmenté exactement de 1 par rapport à sa valeur dans $\mathcal{G}$. En effet, pour augmenter la valeur de MINTREE, le sommet u doit être l'unique sommet porte de $CFC(u)$ et $CFC(u) = CFC(v)$. Mais alors, forcer l'arc (u, v) dans $\mathcal{G}$ transforme $CFC(u)$ en une composante puits de $\mathcal{G}_{(u,v)}$ ce qui augmente MINTREE de 1. On peut donc en déduire que si $\mathcal{D}(\texttt{NTREE}) < \texttt{MINTREE}$ dans $\mathcal{G}_{(u,v)}$ alors, $\mathcal{D}(\texttt{NTREE}) = \texttt{MINTREE}$ dans $\mathcal{G}$. Ainsi, toutes les racines potentielles de $CFC(u)$ auraient été supprimées par la ligne 5 de l'algorithme 5 page 51. D'autre part, dans le cas $u \neq v$, le sommet u étant l'unique porte de $CFC(u)$, l'arc (u, v) aurait été supprimé par la procédure appelée par la ligne 8 de l'algorithme 5 page 51 puisque u était un sommet dominant les racines potentielles de $\mathcal{G}$ (celles éventuellement présentent dans $CFC(u)$ ont été supprimées par la ligne 5) : c'est une contradiction.

Dans le cas (b), remarquons que la valeur de MAXTREE dans $\mathcal{G}_{(u,v)}$ a diminué exactement de 1 par rapport à sa valeur dans $\mathcal{G}$. En effet, pour diminuer la valeur de MAXTREE, le sommet u doit être une racine potentielle de $\mathcal{G}$ et $u \neq v$. Forcer l'arc (u, v) supprime la boucle sur le sommet u et diminue de exactement 1 la valeur de MAXTREE dans $\mathcal{G}_{(u,v)}$. On peut donc en déduire que si $\mathcal{D}(\texttt{NTREE}) > \texttt{MAXTREE}$ dans $\mathcal{G}_{(u,v)}$ alors, $\mathcal{D}(\texttt{NTREE}) = \texttt{MAXTREE}$ dans $\mathcal{G}$. Ainsi, toutes les racines potentielles de $\mathcal{G}$ auraient été forcées par la ligne 3 de l'algorithme 5 page 51 ce qui aurait conduit à la suppression de l'arc (u, v) dans $\mathcal{G}$: c'est une contradiction. □

4.2.4 Complexité

On s'intéresse tout d'abord à la complexité de la condition nécessaire et suffisante introduite par le théorème 6 page 50 qui est évaluée à la ligne 1 de l'algorithme 5 page 51. La complexité repose sur le calcul des composantes fortement connexes de $\mathcal{G}$ qui peut être effectué en $O(m+n)$ par une recherche en profondeur d'abord proposée dans [Tar72]. La ligne 2 ajuste le domaine de la variable NTREE relativement aux bornes minimum (MINTREE) et maximum (MAXTREE) sur le nombre d'arbres possibles pour couvrir $\mathcal{G}$. Ensuite, les lignes 3 à 5 peuvent être trivialement effectuées en $O(n)$. La ligne 6 calcule les sommets dominant les racines potentielles de $\mathcal{G}$ en $O(n^2)$ par l'algorithme proposé dans [CHK01]. Finalement, la ligne 8 appelle la procédure décrite par l'algorithme 6 page précédente. Cette procédure effectue un parcours en profondeur d'abord (lignes 9 à 15) à partir de chaque sommet dominant les racines potentielles de $\mathcal{G}$. Globalement, les lignes 1 à 8 de la procédure sont calculées en $O(mn)$. D'où, le théorème :

Théorème 7. *L'algorithme 5 page 51 filtre la contrainte* tree(NTREE, VER) *en temps* $O(mn)$.

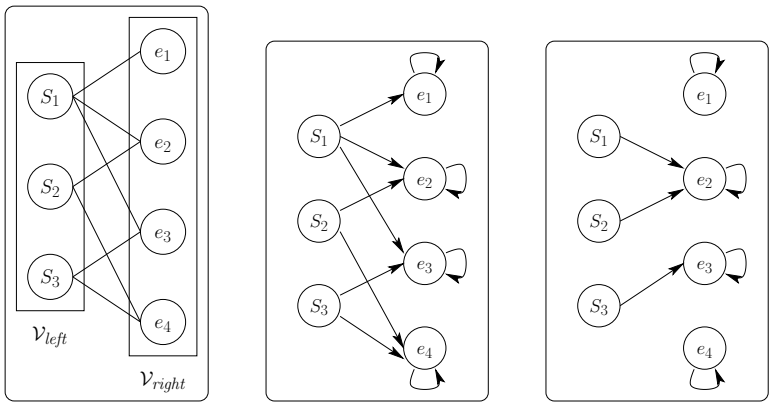
4.3 Filtrer la contrainte *proper-tree*

De nombreuses applications pratiques ne nécessitent pas directement la couverture de tous les sommets du graphe $\mathcal{G}$ associé à la contrainte *proper-tree*. Par exemple, on peut souhaiter trouver un certain nombre de chemins empruntant un sous-ensemble des sommets de $\mathcal{G}$ (c'est par exemple le cas dans l'exemple de planification de missions donné en introduction du chapitre 5 page 65). L'objet de cette section est de montrer comment modéliser et prendre compte une telle situation à l'aide de la variable NPROP représentant le nombre d'arbres propres autorisés dans une partition valide.

La section 4.3.1 montre que le graphe associé à la contrainte *proper-tree* ne peut pas être filtré jusqu'à l'arc-consistance généralisée en temps polynomial. Ensuite, la section 4.3.2 page suivante détaille l'évaluation des bornes minimum et maximum sur le nombre d'arbres propres pouvant couvrir le graphe associé à la contrainte *proper-tree*. Puis, la section 4.3.3 page 57 introduit une condition nécessaire sur l'existence d'une solution satisfaisant la contrainte *proper-tree*. Le filtrage relatif à cette condition nécessaire est introduit en section 4.3.4 page 57 et démontré en section 4.3.5 page 59. Finalement, la section 4.3.6 page 61 étudie la complexité de l'algorithme de filtrage proposé.

4.3.1 Complexité théorique

Théorème 8. *Propager l'arc consistance généralisée pour la contrainte* proper-tree(NPROP, VER) *est NP-complet.*



(a) Graphe bipartie $\mathcal{B}$. (b) Le graphe $\mathcal{G}$. (c) Une solution.

FIG. 4.7 – Modélisation d'un problème de « hitting set » par une contrainte *proper-tree*, avec NPROP = 2 et $|\mathcal{V}_{left}| = |\mathcal{C}|$.

Démonstration. Premièrement, un certificat polynomial pour la contrainte *proper-tree* est fourni. Ce certificat repose sur un algorithme déterministe polynomial vérifiant qu'une assignation totale des variables mises en jeu dans la contrainte *proper-tree* est une solution valide, c.-à-d., satisfait la contrainte. Nous assurons qu'une assignation complète des variables de *proper-tree* constitue une forêt bien formée vérifiant que (1) le graphe orienté $\mathcal{G}_{sure}$ associé à *proper-tree* contient NPROP composantes connexes comportant au moins deux sommets, et (2) $\mathcal{G}_{sure}$ ne contient aucun circuit mettant en jeu plus d'un sommet (c.-à-d., une boucle sur un sommet).

Dans un deuxième temps, nous montrons que toute instance du problème NP-complet « hitting set »[1] [GJ78] est reformulable en temps polynomial sous la forme d'une contrainte *proper-tree*(NPROP, VER). Une modélisation classique du problème « hitting set » repose sur un graphe biparti $\mathcal{B} = (\mathcal{V}_{left}, \mathcal{V}_{right}, \mathcal{E})$ défini de la manière suivante :

- un sommet de $\mathcal{V}_{left}$ est associé à chaque sous-ensemble de $\mathcal{C}$;
- un sommet de $\mathcal{V}_{right}$ est associé à chaque élément de $\mathcal{S}$;
- il existe une arête dans $\mathcal{E}$ entre un sommet de $\mathcal{V}_{left}$ et un sommet de $\mathcal{V}_{right}$ ssi l'élément associé au sommet de $\mathcal{V}_{right}$ appartient au sous-ensemble associé au sommet de $\mathcal{V}_{left}$.

Une solution au problème du « hitting set » consiste alors à trouver une assignation de chaque sommet de $\mathcal{V}_{left}$ à un sommet de $\mathcal{V}_{right}$ telle qu'au plus k sommets de $\mathcal{V}_{right}$ soient utilisés pour assigner tous les sommets de $\mathcal{V}_{left}$. Nous proposons une transformation polynomiale du problème « hitting set » en une contrainte *proper-tree*(NPROP, VER). Pour cela, nous définissons le graphe associé $\mathcal{G}$ par $(\mathcal{V}_{left} \cup \mathcal{V}_{right}, \{(i,j) \in \mathcal{V}_{left} \times \mathcal{V}_{right} \mid i = j \vee (i,j) \in \mathcal{E}\})$. Le problème du « hitting set » consiste alors à trouver une partition de $\mathcal{G}$ avec au plus k arbres propres (voir figure 4.7). □

Lemme 11. *Déterminer le nombre minimum d'arbres propres* MINPROP *participant à une partition d'un graphe $\mathcal{G}$ en arbres est NP-dur.*

Démonstration. Soit P le problème de satisfaction qui, étant donné un graphe orienté $\mathcal{G}$, détermine l'existence d'une partition par des arbres de $\mathcal{G}$ telle que k arbres soient propres. D'après le théorème 8 page précédente, P est un problème NP-complet. Supposons que $\mathcal{G}$ contienne exactement n racines potentielles et que l'on connaisse un oracle répondant au problème de satisfaction P alors, MINPROP est calculable en $\log_2(n)$ appels à l'oracle sur P. □

4.3.2 Bornes sur le nombre d'arbres propres

L'évaluation du nombre minimum d'arbres propres pouvant couvrir le graphe $\mathcal{G}$ étant un problème NP--dur, nous allons nous attacher à déterminer une borne inférieure du nombre minimum d'arbres propres

[1] Étant donné une collection $\mathcal{C}$ de sous-ensembles de l'ensemble $\mathcal{S}$ et un entier k, le problème du « hitting set » vérifie l'existence d'un sous-ensemble $\mathcal{S}' \subseteq \mathcal{S}$ tel que $|\mathcal{S}'| \leq k$ et $\mathcal{S}'$ contient au moins un élément de chaque sous-ensemble de $\mathcal{C}$.

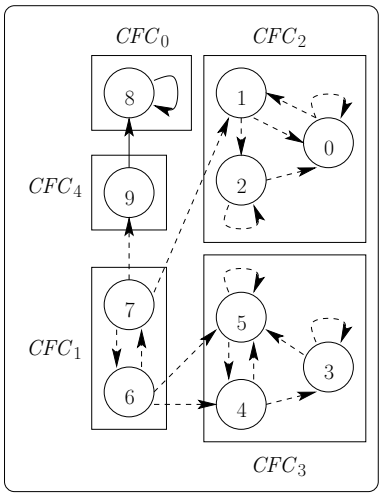


FIG. 4.8 – Évaluation de la borne `MINPROP` sur le graphe $\mathcal{G}$ associé à une contrainte *proper-tree*. Les composantes puits CFC_2 et CFC_3 vérifient la condition (1), donc chacune va former un arbre propre. La composante CFC_0 est atteinte par un arc de $\mathcal{G}_{sure}$, $(9, 8)$, donc elle forme aussi un arbre propre. La borne `MINPROP` est donc égale à 3.

autorisés pour couvrir $\mathcal{G}$.

Proposition 1. *Étant donné une contrainte* proper-tree *et le graphe $\mathcal{G}$ associé, une borne inférieure sur le nombre d'arbres propres,* `MINPROP`*, utilisés dans toute partition de $\mathcal{G}$ est définie par le nombre de composantes puits de $\mathcal{G}$ telles que :*

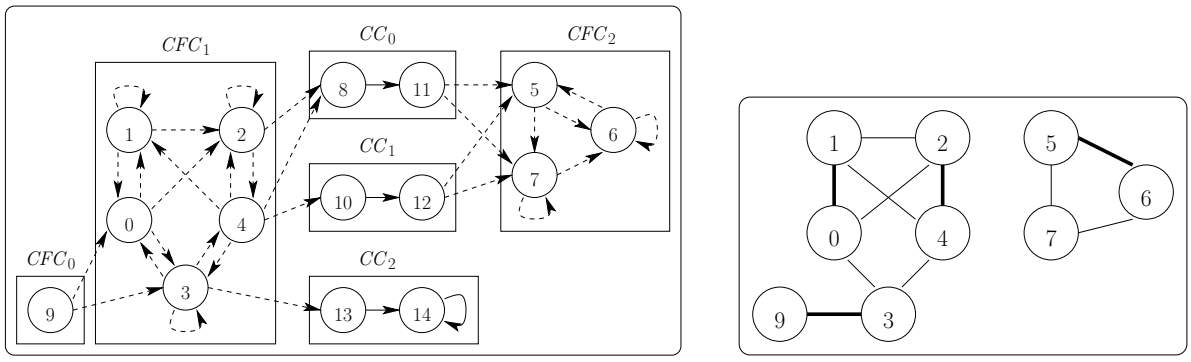
(1) *il existe un sommet sans boucle dans cette composante ;*

(2) *ou, il existe un arc de $\mathcal{G}_{sure}$ vers un sommet de cette composante.*

Démonstration. Dans une composante puits définie par la condition (1), tout sommet sans boucle doit atteindre une racine potentielle de cette composante. Cela constitue un arbre propre avec au moins deux sommets. La condition (2) décrit une composante connexe de $\mathcal{G}_{sure}$ de taille supérieure ou égale 2. Comme une telle composante connexe ne peut pas être refractionnée en composantes de taille strictement inférieure à 2, elle forme un arbre propre. □

Exemple 3. *Le graphe $\mathcal{G}$ représenté par la figure 4.8 contient trois composantes puits CFC_0, CFC_2 et CFC_3. La composante CFC_0 est atteinte par un arc de $\mathcal{G}_{sure}$ (l'arc $(9, 8)$) donc elle forme un arbre propre. D'autre part, les composantes puits CFC_2 et CFC_3 contiennent respectivement les sommets* 1 *et* 4 *qui ne sont pas racines potentielles. Comme ils devront quoiqu'il arrive en atteindre une, on sait que les composantes CFC_2 et CFC_3 engendreront chacune un arbre propre. Finalement, pour le graphe $\mathcal{G}$, on sait que* `MINPROP` *est égal à* 3.

Intéressons-nous maintenant au nombre maximum d'arbres propres pouvant couvrir un graphe $\mathcal{G}$ associé à une contrainte *proper-tree*. Ce nombre est intuitivement lié au nombre de racines potentielles présentes dans le graphe $\mathcal{G}$. En effet, on ne peut pas construire plus d'arbres bien formés que de racines potentielles disponibles initialement. Seulement, nous ne devons pas prendre en compte les sommets isolés. Finalement, parmi les sommets qui ne sont incidents à aucun arc de $\mathcal{G}_{sure}$, il faut évaluer le nombre maximum d'arbres propres possibles. Un arbre propre étant composé d'au moins deux sommets, on cherche à maximiser le nombre d'arbres propres dans le sous-graphe de $\mathcal{G}_{maybe}$ induit par les sommets qui ne sont incidents à aucun arc de $\mathcal{G}_{sure}$. Avant de présenter la formule calculant la borne supérieure sur le nombre d'arbres propres couvrant le graphe $\mathcal{G} = (\mathcal{V}, \mathcal{E})$, nous introduisons le graphe non-orienté $\mathcal{G}^{root}_{maybe} = (\mathcal{V}^{root}_{maybe}, \mathcal{E}^{root}_{maybe})$ défini par :



(a) Le graphe $\mathcal{G}$.

(b) Le graphe $\mathcal{G}^{root}_{maybe}$ associé à $\mathcal{G}$.

FIG. 4.9 – Un graphe $\mathcal{G}$ associé à une contrainte *proper-tree* est représenté par la figure 4.9(a). Le graphe $\mathcal{G}^{root}_{maybe}$ associé à $\mathcal{G}$ est donné par la figure 4.9(b). Un couplage de cardinalité maximum de $\mathcal{G}^{root}_{maybe}$ est représenté par les arêtes en gras de la figure 4.9(b), on a $|\mu(\mathcal{G}^{root}_{maybe})| = 4$. Les composantes connexes de $\mathcal{G}_{sure}$ de taille ≥ 2 sont données par CC_0, CC_1 et CC_2 (contenant les arcs en trait plein). Le nombre de racines potentielles ou fixées est donné par $|\mathcal{R}| = 6$. La borne `MAXPROP` est donc égale à $\min(6, 4+3) = 6$.

- l'ensemble des sommets de $\mathcal{V}^{root}_{maybe}$ est défini par l'union de l'ensemble $\mathcal{V}_{root}$ des sommets racines potentielle de $\mathcal{G}_{maybe}$ avec l'ensemble $\mathcal{V}_{prec}$ des sommets précédant au moins une racine potentielle dans $\mathcal{G}_{maybe}$;
- l'ensemble des arêtes $\mathcal{E}^{root}_{maybe}$ est défini par $\{(u,v) \in \mathcal{E} \mid v \in \mathcal{V}_{root}\}$.

De plus, nous noterons dans la suite par $\mu(\mathcal{G}^{root}_{maybe})$ un couplage de cardinalité maximum dans le graphe $\mathcal{G}^{root}_{maybe}$ et, par $CC^{\geq 2}_{sure}$ l'ensemble des composantes connexes de $\mathcal{G}_{sure}$, de taille supérieure ou égale à 2. Finalement, on notera par $\mathcal{R}$ l'ensemble des racines (potentielles ou fixées) de $\mathcal{G}$.

Proposition 2. *Étant donné une contrainte* proper-tree *et le graphe $\mathcal{G}$ associé, une borne supérieure sur le nombre d'arbres propres,* `MAXPROP`*, utilisés dans une partition de $\mathcal{G}$ est définie par :*

$$\min(|\mathcal{R}|, |CC^{\geq 2}_{sure}| + |\mu(\mathcal{G}^{root}_{maybe})|)$$

Démonstration. Montrons que la borne `MAXPROP` est correcte (en d'autre termes, `MAXPROP` est une borne supérieure du nombre maximum d'arbres propres). Pour cela, considérons les graphes $\mathcal{G}_{maybe}$ et $\mathcal{G}_{sure}$ associés au graphe $\mathcal{G}$ définissant la contrainte *proper-tree*. Chaque composante connexe de $\mathcal{G}_{sure}$ de taille supérieure ou égale à 2 va participer a, au plus, un arbre propre. Il reste donc à montrer combien d'arbres propres peuvent être créés au plus dans $\mathcal{G}_{maybe}$. Pour cela, considérons le graphe $\mathcal{G}^{root}_{maybe}$ dérivé de $\mathcal{G}_{maybe}$ et montrons que l'on ne peut pas construire plus de $|\mu(\mathcal{G}^{root}_{maybe})|$ arbres propres dans $\mathcal{G}_{maybe}$. À cet effet, montrons que si tous les sommets de $\mathcal{V}_{root}$ sont saturés dans le couplage $\mu(\mathcal{G}^{root}_{maybe})$ alors, on ne peut pas créer plus de $|\mu(\mathcal{G}^{root}_{maybe})|$ arbres propres dans $\mathcal{G}_{maybe}$. S'il existe un sommet de $\mathcal{V}_{root}$ non saturé dans le couplage alors : soit la racine potentielle correspondante n'a pas de prédécesseur dans $\mathcal{G}_{maybe}$ et donc cette racine ne peut pas former un arbre propre ; soit elle possède un prédécesseur u dans $\mathcal{G}_{maybe}$ mais, il est forcément saturé dans le couplage $\mu(\mathcal{G}^{root}_{maybe})$ sinon il existerait une chaîne augmentante saturant l'arête correspondante. □

Exemple 4. *Du graphe $\mathcal{G}$ représenté par la figure 4.9(a) on peut extraire d'une part le graphe $\mathcal{G}^{root}_{maybe}$, représenté par la figure 4.9(b), et d'autre part le graphe $\mathcal{G}_{sure}$, représenté par les composantes CC_0, CC_1 et CC_2 de la figure 4.9(a). Un couplage maximum de $\mathcal{G}^{root}_{maybe}$ fournit un arrangement par paires des sommets non instanciés de $\mathcal{G}$ précédant au moins une racine potentielle. Cet arrangement maximise, par définition, le nombre d'arbres propres constructibles dans $\mathcal{G}_{maybe}$ (représenté par les composantes CFC_0, CFC_1 et CFC_2 de la figure 4.9(a)). On a donc $|\mu(\mathcal{G}^{root}_{maybe})| = 4$. D'autre part, les composantes connexes CC_0, CC_1 et CC_2 du graphe $\mathcal{G}_{sure}$ sont de taille ≥ 2 donc, elles peuvent potentiellement former chacune un arbre propre. Ainsi, on obtient :* `MAXPROP` $= \min(|\mathcal{R}|, |CC^{\geq 2}_{sure}| + |\mu(\mathcal{G}^{root}_{maybe})|) = \min(6, 3+4) = 6$ *arbres propres au maximum dans le graphe $\mathcal{G}$.*

4.3.3 Existence d'une solution pour la contrainte *proper-tree*

Le théorème suivant donne deux conditions nécessaires d'existence d'une solution pour la contrainte d'arbre *proper-tree*. La première condition assure que l'on peut effectivement créer une forêt couvrante alors que la seconde condition vérifie qu'il existe au moins une valeur k dans le domaine de NPROP compatible avec les bornes sur le nombre d'arbres propres. Remarquons que cette condition est en tout point similaire à la condition nécessaire et suffisante introduite pour la contrainte *tree* dans la section 4.2 page 49. Cependant, dans ce cas, on s'attache à évaluer la consistance du domaine de NPROP par rapport aux bornes MINPROP et MAXPROP. Les bornes MINPROP et MAXPROP n'étant pas forcément réalisables, il s'agit seulement de conditions nécessaires.

Théorème 9. *S'il existe une solution pour la contrainte* proper-tree(NPROP, VER) *alors, les conditions suivantes sont vérifiées :*

(1) *toute composante puits de $\mathcal{G}$ est une composante enracinée de $\mathcal{G}$;*

(2) $\mathcal{D}(\texttt{NPROP}) \cap [\texttt{MINPROP}, \texttt{MAXPROP}] \neq \emptyset$*, où* MINPROP *est une borne inférieure du nombre minimum d'arbres propres pouvant couvrir $\mathcal{G}$ (proposition 1 page 55) et,* MAXPROP *est une borne supérieure du nombre maximum d'arbres propres pouvant couvrir $\mathcal{G}$ (proposition 2 page précédente).*

Démonstration. Pour la condition (1) : Si une composante puits de $\mathcal{G}$ n'est pas une composante enracinée alors, il existe au moins un circuit dans $\mathcal{G}$ parmi les sommets associés à cette composante puits et la contrainte ne peut pas être satisfaite. Pour la condition (2) : Si $\mathcal{D}(\texttt{NPROP}) \cap [\texttt{MINPROP}, \texttt{MAXPROP}] = \emptyset$ alors, soit max(NPROP) est strictement inférieur à MINPROP, soit min(NPROP) est strictement supérieur à MAXPROP. Dans le premier cas, la contradiction est évidente par le fait que, pour chaque composante puits de $\mathcal{G}$ prise en compte dans la borne MINPROP (proposition 1 page 55), on a montré qu'au moins un arbre propre serait créé. Dans le second cas, la borne MAXPROP étant correcte (proposition 2 page précédente), l'existence d'une solution pour la contrainte *proper-tree*, contenant un nombre d'arbres propres supérieur à MAXPROP est impossible. □

4.3.4 Algorithme de filtrage pour la contrainte *proper-tree*

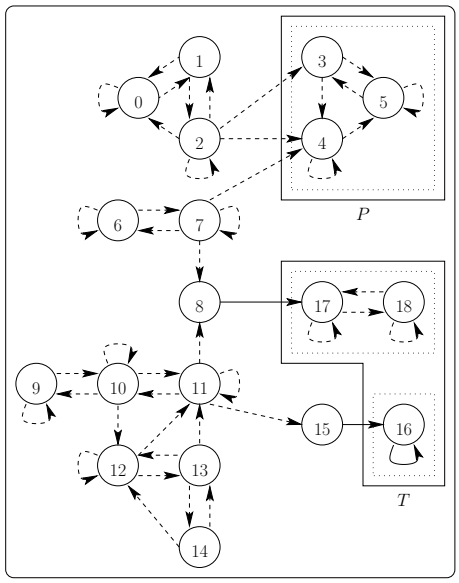
Nous décrivons ici le filtrage relatif à la contrainte *proper-tree*(NPROP, VER) par rapport aux bornes MINPROP et MAXPROP associées aux propositions 1 page 55 et 2 page précédente. Tout d'abord, nous décrivons comment la notion de sommet dominant dans le graphe $\mathcal{G}$ va à nouveau (c'était déjà le cas pour la contrainte *tree* en section 4.2 page 49) permettre de filtrer certains arcs inconsistants de $\mathcal{G}_{maybe}$ par rapport à la borne MINPROP. Puis, nous montrerons comment les couplages maximaux du graphe $\mathcal{G}^{root}_{maybe}$ associé à $\mathcal{G}_{maybe}$ vont également permettre (c'était déjà le cas pour la contrainte *proper-forest* au chapitre 3 page 29) de filtrer les arcs inconsistants de $\mathcal{G}_{maybe}$ par rapport à la borne MAXPROP.

Cependant, avant de s'intéresser au filtrage de la contrainte *proper-tree* relativement aux bornes MINPROP et MAXPROP, remarquons que l'algorithme de filtrage proposé en section 4.2 page 49 pour la contrainte *tree* reste également valide dans le cas de la contrainte *proper-tree*, excepté pour la partie concernant les bornes MINTREE et MAXTREE, qui sont spécifiques à la contrainte *tree*. En effet, il suffit pour s'en convaincre d'observer que les deux contraintes ne diffèrent que par les variables NTREE et NPROP spécifique à chacune. On remarque alors, que le nombre d'arbres (NTREE) pouvant partitionner un graphe $\mathcal{G}$ est toujours supérieur ou égal au nombre d'arbres propres (NPROP) pouvant couvrir ce même graphe.

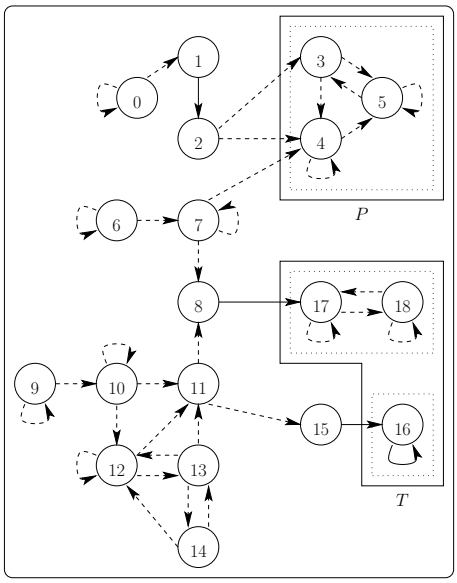
Exemple 5. *La figure 4.10(a) page suivante présente un graphe $\mathcal{G}$ associé à une contrainte* proper-tree(NPROP, VER)*. Ce graphe sera utilisé pour illustrer l'effet des algorithmes de filtrage que nous allons introduire dans la suite de cette section. Les rectangles représentent les sommets des composantes puits de $\mathcal{G}$ atteintes par un arc de $\mathcal{G}_{sure}$ ($\mathcal{T}$) et des composantes puits de $\mathcal{G}$ contenant un sommet sans boucle ($\mathcal{P}$). Les arcs en pointillé sont les arcs de $\mathcal{G}_{maybe}$ et les arcs en traits pleins sont ceux de $\mathcal{G}_{sure}$.*

Filtrage lorsque NPROP $\leq$ MINPROP.

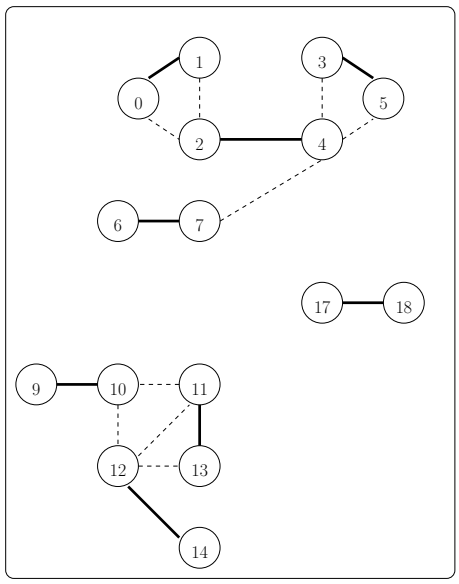
L'algorithme 7 page 59 commence tout d'abord par calculer la borne MINPROP (ligne 1) relative aux nombres minimum d'arbres propres autorisés dans le graphe $\mathcal{G}$ associé à la contrainte *proper-tree*. Puis, nous répercutons cette borne sur la variable NPROP (ligne 2).



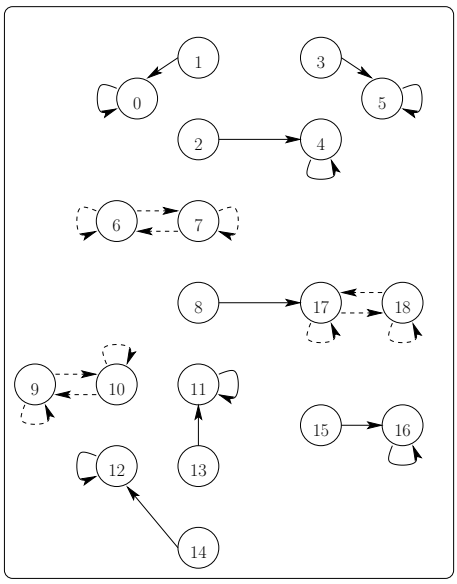
(a) Le graphe $\mathcal{G}$ associé à la contrainte *proper-tree*.



(b) $\mathcal{G}$ après filtrage lorsque NPROP $\leq$ MINPROP $= 3$.



(c) Un couplage maximum dans $\mathcal{G}^{root}_{maybe}$.



(d) $\mathcal{G}$ après filtrage lorsque NPROP $\geq$ MAXPROP $= 10$.

FIG. 4.10 – La figure 4.10(a) représente un graphe $\mathcal{G}$ associé à la contrainte *proper-tree*(NPROP, VER). La figure 4.10(b) représente l'état du graphe $\mathcal{G}$ après le filtrage relatif à la borne MINPROP. La figure 4.10(c) représente la graphe $\mathcal{G}^{root}_{maybe}$ associé à $\mathcal{G}$, les arêtes en gras sont les arêtes appartenant au couplage maximum $\mu(\mathcal{G}^{root}_{maybe})$ du graphe $\mathcal{G}^{root}_{maybe}$ associé au graphe $\mathcal{G}$. Les arêtes en pointillé sont les arêtes n'appartenant à aucun couplage de $\mathcal{G}^{root}_{maybe}$. La figure 4.10(d) représente l'état du graphe $\mathcal{G}$ après le filtrage relatif à la borne MAXPROP.

Le filtrage du graphe $\mathcal{G}_{maybe}$ relatif à la borne MINPROP nécessite de vérifier pour chaque sommet du graphe $\mathcal{G}_{maybe}$, s'il peut atteindre une composante puits de $\mathcal{G}_{maybe}$ dont on est certain qu'elle formera un arbre propre (ces composantes sont enregistrées dans les ensembles $\mathcal{T}$ et $\mathcal{P}$, lignes 4 et 5). L'intuition de l'algorithme est basée sur la détection des arcs de $\mathcal{G}_{maybe}$ qui vont empêcher certains sommets d'être reliés aux arbres propres déjà créés ou aux composantes puits dont on est sûr qu'elles formeront au moins

Algorithme 7 Filtrer une contrainte *proper-tree*(NPROP, VER) avec NPROP $\leq$ MINPROP.

```
/* Restriction de NPROP */
1. Calculer MINPROP ;
2. Ajuster min(NPROP) avec MINPROP ;
/* Filtrage de G_maybe par rapport à MINPROP */
3. Si D(NPROP) = MINPROP alors
4.   T ← les sommets des composantes puits de G atteintes par un arc de G_sure ;
5.   P ← les sommets des composantes puits de G contenant un sommet sans boucle ;
6.   D ← les sommets dominants les racines potentielles de T ∪ P (racines de G inclues) ;
8. Appliquer la procédure FilteringDominators(G, T ∪ P, D) décrite par l'algorithme 6 page 52 ;
9. Retourner le graphe G filtré ;
```

un arbre propre (les composantes enregistrées dans $\mathcal{T}$ et $\mathcal{P}$). Ce filtrage correspond parfaitement à celui introduit dans le cas de la contrainte *tree*. En effet, dans le cas de *tree*, on considérait que toute racine du graphe $\mathcal{G}$ pouvait former un arbre alors que, dans le cas de *proper-tree*, on ne considère qu'un sous-ensemble de racines de $\mathcal{G}$ donné par l'ensemble des racines contenues dans $\mathcal{T} \cup \mathcal{P}$. La procédure permettant de filtrer chaque sommet dominant est décrite par l'algorithme 6 page 52 `FilteringDominators`() appliqué sur le graphe $\mathcal{G}$ par rapport au sous-ensemble $\mathcal{T} \cup \mathcal{P}$ des sommets de $\mathcal{G}$.

Remarquons que dans le cas du filtrage sur la borne MINPROP, il est inutile d'appliquer le filtrage, pourtant valide, de la contrainte *tree* comme nous l'avions dit en introduction. En effet, si dans la contrainte *tree* les sommets dominants sont calculés par rapport à toutes les racines du graphe $\mathcal{G}$, dans le cas de *proper-tree*, les sommets dominants sont calculés par rapport à un sous-ensemble des racines de $\mathcal{G}$ (celles contenues dans $\mathcal{T} \cup \mathcal{P}$). Donc, tout sommet dominant de $\mathcal{G}$ pris en compte par le filtrage de la contrainte *tree* est pris en compte par le filtrage de la contrainte *proper-tree* lorsque la borne MINPROP est atteinte.

Exemple 6. *La figure 4.10(b) page précédente représente l'état du graphe $\mathcal{G}$ de la figure 4.10(a) page ci-contre obtenu après filtrage par l'algorithme 7. Dans la figure 4.10(a) page ci-contre, les sommets* $\{1, 2, 7, 10, 11\}$ *sont des sommets dominant les racines potentielles de $\mathcal{T} \cup \mathcal{P}$. L'algorithme 7 supprime la boucle sur les racines potentielles* 2 *et* 11 *car il s'agit de sommets de $\mathcal{G}$ qui dominent les racines potentielles de $\mathcal{T} \cup \mathcal{P}$ pour au moins un sommet qui lui n'est pas racine potentielle, dans notre cas les sommets* 1, 13 *et* 14. *Puis, il supprime les arcs sortant de chacun des sommets dominants pouvant empêcher l'accès à une racine potentielle de $\mathcal{T} \cup \mathcal{P}$. Ainsi, les arcs* $(1, 0)$, $(2, 0)$, $(2, 1)$, $(7, 6)$, $(10, 9)$ *et* $(11, 10)$ *sont supprimés.*

Filtrage lorsque NPROP $\geq$ MAXPROP.

L'algorithme 8 page suivante commence tout d'abord par calculer la borne MAXPROP (ligne 1) relative aux nombres maximum d'arbres propres autorisés dans le graphe $\mathcal{G}$ associé à une contrainte *proper-tree*. Puis, nous répercutons cette borne sur la variable NPROP (ligne 2).

Le filtrage du graphe $\mathcal{G}_{maybe}$ relatif à la borne MAXPROP nécessite de détecter les arêtes (u, v) saturées dans tout couplage maximum de $\mathcal{G}^{root}_{maybe}$. En effet, ces arêtes nous assurent que les sommets u et v feront partie du même arbre propre (contenant soit les arcs (u, v) et (v, v), soit les arcs (v, u) et (u, u)) dans toute couverture valide de $\mathcal{G}_{maybe}$ par $|\mu(\mathcal{G}^{root}_{maybe})|$ arbres propres (lignes 3 à 14). L'intuition de ce filtrage est basée sur le fait que l'on cherche à créer le plus grand nombre d'arbres propres dans $\mathcal{G}_{maybe}$. Dans ce but, les lignes 3 à 14 suppriment tous les arcs de $\mathcal{G}_{maybe}$ qui ne peuvent pas y contribuer.

Exemple 7. *La figure 4.10(d) page précédente représente l'état du graphe $\mathcal{G}$ de la figure 4.10(a) page ci-contre obtenu après filtrage par l'algorithme 8 page suivante. Dans la figure 4.10(c) page ci-contre, les arêtes (u, v) appartenant à tout couplage de cardinalité maximum de $\mathcal{G}^{root}_{maybe}$ conduisent au filtrage des arcs sortant des sommets u et v. Ainsi, les arcs entre les sommets* 0 *et* 2, 1 *et* 2, 2 *et* 3, 3 *et* 4, 4 *et* 5, 4 *et* 7, 10 *et* 11, 10 *et* 12, 12 *et* 13, 13 *et* 14 *vont être supprimés. Finalement, la boucle sur le sommet* 2 *est supprimée par les lignes 12 à 14 puisque l'arc* $(2, 4)$ *appartient à $\mathcal{G}_{maybe}$ mais pas l'arc* $(4, 2)$.

4.3.5 Correction

Pour prouver que les algorithmes 7 et 8 page suivante sont corrects, nous montrons que :

Algorithme 8 Filtrer une contrainte *proper-tree*(NPROP, VER) avec NPROP $\geq$ MAXPROP.

```
/* Restriction de NPROP */
1. Calculer MAXPROP ;
2. Ajuster max(NPROP) avec MAXPROP ;
/* Filtrage de G_maybe par rapport à MAXPROP */
3. Si D(NPROP) = MAXPROP alors
4.  Pour chaque arête (u, v) de G^root_maybe faire
5.    Si (u, v) appartient à tout couplage de cardinalité maximum de G^root_maybe alors
6.      Pour chaque arc (u, w) de G_maybe tel que w ≠ v et w ≠ u faire
7.        Supprimer (u, w) de G_maybe ;
8.      Pour chaque arc (v, w) de G_maybe tels que w ≠ v et w ≠ u faire
9.        Supprimer (v, w) de G_maybe ;
10.     Si u est une racine potentielle ou v est une racine potentielle mais pas les deux alors
11.       Forcer la boucle sur u ou sur v suivant le cas ;
12.     Si u et v sont deux racines potentielles alors
13.       Si (u, v) est dans G_maybe mais pas (v, u) alors Supprimer la boucle sur u ;
14.       Sinon Supprimer la boucle sur v ;
15. Retourner le graphe G filtré ;
```

1. aucun arc de $\mathcal{G}$ n'est supprimé par l'algorithme 7 page précédente s'il appartient à au moins une solution satisfaisant la contrainte *proper-tree* (Lemme 12) ;
2. aucun arc de $\mathcal{G}$ n'est supprimé par l'algorithme 8 s'il appartient à au moins une solution satisfaisant la contrainte *proper-tree* (Lemme 13).

Lemme 12. *L'algorithme 7 page précédente ne supprime aucun arc de $\mathcal{G}_{maybe}$ appartenant à au moins une solution satisfaisant la contrainte* proper-tree(NPROP, VER) *lorsque* NPROP $\leq$ MINPROP.

Démonstration. On rappelle tout d'abord que toute composante puits contenue dans $\mathcal{T} \cup \mathcal{P}$ forme au moins un arbre propre. Maintenant, supposons qu'il existe une solution S satisfaisant la contrainte *proper-tree*. Nous pouvons distinguer deux hypothèses possibles pour le filtrage effectué à la ligne 8 de l'algorithme 7 page précédente :

- Soit u un sommet racine de S tel que la boucle sur u aurait été supprimée par la ligne 8. Alors, par définition, u est un sommet dominant les racines potentielles de $\mathcal{T} \cup \mathcal{P}$ dans $\mathcal{G}$. De plus, il existe un sommet v, qui n'est pas une racine potentielle, tel que tout chemin de v à une racine potentielle de $\mathcal{T} \cup \mathcal{P}$ passe par le sommet u. Donc, fixer la boucle sur u va créer un nouvel arbre enraciné sur u. Il s'agit d'un arbre propre puisque v est forcément rattaché à u par un chemin. Mais, u pouvait lui-même être rattaché à une racine potentielle de $\mathcal{T} \cup \mathcal{P}$, c.-à-d., à des composantes qui vont chacune former au moins un arbre propre. Ainsi, fixer la boucle sur u crée un arbre propre supplémentaire : c'est une contradiction ;
- Soit (u, v) un arc de S qui aurait été supprimé par la ligne 8. Alors, par définition, u est un sommet dominant les racines potentielles de $\mathcal{T} \cup \mathcal{P}$ dans $\mathcal{G}$. De plus, forcer l'arc (u, v) dans $\mathcal{G}$ implique que ni u, ni v ne pourront atteindre une racine potentielle de $\mathcal{T} \cup \mathcal{P}$. Donc les sommets u et v vont appartenir à un nouvel arbre propre qui ne sera pas enraciné sur un sommet de $\mathcal{T} \cup \mathcal{P}$. Ce nouvel arbre est forcément supplémentaire car u aurait pu être rattaché par un chemin à une racine potentielle de $\mathcal{T} \cup \mathcal{P}$, et v pouvait toujours être rattaché à u. Ainsi, fixer l'arc (u, v) crée un arbre propre supplémentaire : c'est une contradiction.

□

Lemme 13. *L'algorithme 8 ne supprime aucun arc de $\mathcal{G}_{maybe}$ appartenant à au moins une solution satisfaisant la contrainte* proper-tree(NPROP, VER) *lorsque* NPROP $\geq$ MAXPROP.

Démonstration. On montre tout d'abord que tout arc (u, v) de $\mathcal{G}_{maybe}$ supprimé par l'algorithme ne crée pas un sommet w ne pouvant plus atteindre de racine de $\mathcal{G}$. Supposons que cela ne soit pas le cas. De la suppression d'un arc (u, v), on peut étudier deux cas :

(1) $u \neq v$: c'est le filtrage des lignes 5 à 11. On peut en déduire que u et v dominent les racines de $\mathcal{G}$ par rapport à w. Par conséquent, le sommet v est une racine potentielle mais pas le sommet u et, u ne précède pas directement une autre racine que v sinon le retrait de (u, v) n'aurait pas isolé w des racines de $\mathcal{G}$. Mais alors, dans $\mathcal{G}^{root}_{maybe}$, u est atteint par l'unique arête (u, v) donc, (u, v) est saturée dans au moins un couplage de cardinalité maximum de $\mathcal{G}^{root}_{maybe}$ et n'a pas pu être supprimée par les lignes 5 à 11 : c'est une contradiction.

(2) $u = v$: c'est le filtrage des lignes 12 et 14. Dans ce cas, on sait que $u = v$. Mais alors, la boucle sur u était la seule racine atteignable par w. Ceci est absurde car, si (u, v) est supprimée de $\mathcal{G}_{maybe}$ alors, c'est qu'il existe un arc (u, r) tel que r est une racine de $\mathcal{G}$. Donc, après le retrait de (u, v), le sommet w peut toujours atteindre une racine de $\mathcal{G}$.

Supposons maintenant qu'il existe une solution S telle qu'il existe un arc (u, v) dans S supprimé par l'algorithme 8 page ci-contre. Étudions les deux cas de filtrages possibles :

(1) $u \neq v$: c'est le filtrage des lignes 5 à 11. Cela signifie qu'il existe un arc (u, w) dans $\mathcal{G}_{maybe}$ tel que l'arête (u, w) correspondante dans $\mathcal{G}^{root}_{maybe}$ soit saturée dans tout couplage maximum. Si l'arête (u, w) est une arête toujours saturée alors, les sommets u et w forment un arbre propre dans toute partition de $\mathcal{G}^{root}_{maybe}$ maximisant le nombre d'arbres propres (voir la contrainte *proper-forest*). Par conséquent, ils forment aussi un arbre propre dans toute partition de $\mathcal{G}_{maybe}$ maximisant le nombre d'arbres propres. Ainsi, la suppression de l'arc (u, w) de $\mathcal{G}_{maybe}$ diminue le nombre d'arbres propres couvrant $\mathcal{G}_{maybe}$: c'est une contradiction.

(2) $u = v$: c'est le filtrage des lignes 12 et 14. C'est une conséquence directe de la preuve pour le cas $u \neq v$. En effet, si pour une arête (u, w) toujours saturée dans $\mathcal{G}^{root}_{maybe}$ ne correspond qu'un unique arc (u, w) de $\mathcal{G}_{maybe}$ alors, la boucle sur u ne peut pas être fixée. En effet, elle empêcherait les sommets u et w de former un arbre propre : c'est une contradiction.

□

4.3.6 Complexité

Le calcul de la borne `MINPROP` repose sur la connaissance des composantes connexes du graphe $\mathcal{G}_{sure}$ et des composantes fortement connexes de $\mathcal{G}$, la complexité est donc en $O(m+n)$. Le calcul de la borne `MAXPROP` nécessite de calculer un couplage de cardinalité maximum dans le graphe $\mathcal{G}^{root}_{maybe}$ associé à $\mathcal{G}_{maybe}$, ceci peut être effectué en $O(m\sqrt{n})$ [MV80].

Le filtrage relatif à la borne `MINPROP` nécessite de connaitre les composantes puits de $\mathcal{G}_{maybe}$ contenant un sommet sans boucle et celles atteintes par un arc de $\mathcal{G}_{sure}$. Reconnaître de telles composantes fortement connexes de $\mathcal{G}_{maybe}$ coûte au plus $O(n)$ (en admettant que les composantes puits de $\mathcal{G}_{maybe}$ soient connues). Il faut aussi détecter l'ensemble des sommets dominant les racines potentielles de $\mathcal{T} \cup \mathcal{P}$, ce qui est calculé en $O(n^2)$ comme nous l'avons montré en section 4.2 page 49. Finalement, pour chaque sommet dominant, un parcours en profondeur est exécuté en $O(mn)$ par la procédure `FilteringDominators` introduite par l'algorithme 6 page 52.

Finalement, le filtrage relatif à la borne `MAXPROP` nécessite de savoir détecter les arêtes de $\mathcal{G}^{root}_{maybe}$ saturées dans tout couplage de cardinalité maximum de $\mathcal{G}^{root}_{maybe}$. Comme nous l'avons montré en section 3.3.4 page 42 ceci peut être respectivement effectué en $O(mn)$. On en déduit les théorèmes suivant :

Théorème 10. *Les algorithmes 7 page 59 et 8 page précédente filtrent une contrainte* proper-tree *avec une complexité temporelle de* $O(nm)$.

4.4 Synthèse sur les contraintes d'arbre dans les cas orienté et non-orienté

Cette section met en lumière les similitudes et les différences entre, d'une part les contraintes *resource-forest* et *proper-forest* introduites par le chapitre 3 page 29, et d'autre part les contraintes *tree* et *proper-tree* que nous venons de voir. Nous rappelons que toutes les quatre sont définies sur un graphe $\mathcal{G} = (\mathcal{V}, \mathcal{E})$, orienté ou non suivant le cas, tel que $|\mathcal{V}| = n$ et $|\mathcal{E}| = m$.

Le Tableau 4.1 résume les meilleures complexités connues pour vérifier la faisabilité et filtrer chaque contrainte d'arbre. Il faut remarquer que, comme bien souvent, le calcul d'une forêt couvrante pour un graphe $\mathcal{G}$ donné est globalement plus coûteux dans le cas des graphes orientés, que dans le cas des graphes non-orientés.
Dans la suite, le tableau 4.2 page ci-contre utilise les notations suivantes :

Notation 6. *Pour un graphe H donné,*
- $|CC(H)|$ *est le nombre de composantes connexes maximales au sens de l'inclusion ;*
- $\mu(H)$ *est un couplage de cardinalité maximum de H ;*
- $|CFC_{sink}(H)|$ *dénote le nombre de composantes fortement connexes puits dans H ;*
- $|R_H|$ *est le nombre de racines potentielles dans H ;*
- $|CFC_{proper}(H)|$ *est le nombre de composantes puits de H telles que pour chaque composante décomptée, il existe un sommet sans boucle ou il existe un arc sûr l'atteignant.*

Le tableau 4.2 page suivante résume les principales propriétés de graphes utilisées pour déterminer des bornes réalisables sur le nombre d'arbres autorisés pour couvrir le graphe, ainsi que les conditions d'existence d'arbres « bien formés » vis-à-vis de la définition de chaque contrainte. Ce dernier tableau montre que quatre propriétés de la théorie des graphes définissent les quatre contraintes : les composantes connexes (dans le cas de graphes non-orientés), les composantes fortement connexes (dans le cas des graphes orientés), le couplage de cardinalité maximum et, la détection de cycles et circuits. Pour chacune des contraintes, des conditions nécessaires et des règles de filtrages ont été déduites avec des algorithmes connus (par ex., « parcours en profondeur d'abord », couplage de cardinalité maximum, détection de composantes connexes, calcul des sommets dominants), ainsi que de nouveaux algorithmes (par ex., identification des sommets saturés dans tout couplage de cardinalité maximum). Ce tableau met bien en évidence l'équivalence sémantique existante entre la contrainte *proper-forest* et la contrainte *proper-tree* d'une part et, entre la contrainte *resource-forest* et la contrainte *tree* d'autre part.

Dans le premier cas, les bornes `MINTREE` de *proper-forest* et `MINPROP` de *proper-tree* évaluent toutes deux les composantes (connexes ou fortement connexes) du graphe $\mathcal{G}$ dont on est certain qu'elles vont former un arbre propre dans $\mathcal{G}$. De manière similaire, la borne `MAXTREE` de *proper-forest* et `MAXPROP` de *proper-tree* évaluent toutes deux le nombre maximum d'arbres propres de $\mathcal{G}$ en comptant : d'une part le nombre de composantes connexes de $\mathcal{G}_{sure}$ qui vont former (ou potentiellement former) un nouvel arbre propre ($|CC(\mathcal{G}_{sure})|$ pour *proper-forest* et $|CC^{\geq 2}_{sure}|$ pour *proper-tree*) ; d'autre part, le nombre maximum d'arbres propres pouvant être formé par le graphe des arcs « possibles » ($|\mu(\mathcal{G}_{maybe})|$ pour *proper-forest* et $|\mu(\mathcal{G}^{root}_{maybe})|$ pour *proper-tree*). Notons toutefois que `MINPROP` et `MAXPROP`, dans le cas de la contrainte *proper-tree*, ne sont pas des bornes forcément réalisables.

Dans le second cas, les bornes `MINTREE` de *resource-forest* et `MINTREE` de *tree* évaluent toutes deux les composantes (connexes ou fortement connexes) du graphe $\mathcal{G}$ dont on est certain qu'elles vont former un arbre. On remarque bien, que dans un cas, toutes les composantes connexes sont considérées, alors que dans l'autre seules les composantes puits le sont. En effet, le passage aux graphes orientés impose un sens de circulation des feuilles vers la racine. Pour les bornes `MAXTREE` de *resource-forest* et `MAXTREE` de *tree*, elles évaluent toutes deux le nombre de sommets pouvant servir de racine à un arbre (le nombre de sommets ressources pour *resource-forest* et le nombre de racines potentielles pour *tree*).

Type de graphe	Non-orienté		Orienté	
	proper-forest	*resource-forest*	*tree*	*proper-tree*
vérifier la faisabilité	$O(m\sqrt{n})$	$O(n+m)$	$O(n+m)$	$O(m\sqrt{n})$
consistance	$O(mn)$ [pire cas], $O(m\sqrt{n})$ [classique]	$O(n+m)$	$O(mn)$	$O(mn)$

Tab. 4.1 – Meilleures complexités connues pour les contraintes structurelles de partitionnement par des arbres.

Type de graphe	Non-orienté	
	proper-forest	*resource-forest*
`MINTREE`	$\lvert CC(\mathcal{G})\rvert$	$\lvert CC(\mathcal{G})\rvert$
`MAXTREE`	$\lvert CC(\mathcal{G}_{sure})\rvert + \lvert\mu(\mathcal{G}_{maybe})\rvert$	$\lvert CC(\mathcal{G}_{sure})$ avec au moins un sommet *ressource*$\rvert$
Arbres « bien formés »	pas de cycles dans $\mathcal{G}_{sure}$, pas de sommets isolés dans $\mathcal{G}$	pas de cycle dans $\mathcal{G}_{sure}$, un sommet *ressource* dans chaque $CC(\mathcal{G}_{sure})$
Nombre d'arbres valides	$\mathcal{D}(\texttt{NTREE}) \cap [\texttt{MINTREE}, \texttt{MAXTREE}] \neq \emptyset$	
Type de graphe	Orienté	
	proper-tree	*tree*
`MINPROP` | `MINTREE`	$\lvert CFC_{proper}(\mathcal{G})\rvert$	$\lvert CFC_{sink}(\mathcal{G})\rvert$
`MAXPROP` | `MAXTREE`	$\min(\lvert\mathcal{R}\rvert, \lvert CC^{\geq 2}_{sure}\rvert + \lvert\mu(\mathcal{G}^{root}_{maybe})\rvert)$	$\lvert\mathcal{R}\rvert$
Arbres « bien formés »	au moins une racine potentielle dans chaque $CFC_{sink}(\mathcal{G})$	
Nombre d'arbres valides	$\mathcal{D}(\texttt{NPROP}) \cap [\texttt{MINPROP}, \texttt{MAXPROP}] \neq \emptyset$	$\mathcal{D}(\texttt{NTREE}) \cap [\texttt{MINTREE}, \texttt{MAXTREE}] \neq \emptyset$

TAB. 4.2 – Propriétés de graphe caractérisant les solutions des contraintes de partitionnement par des arbres.

Chapitre 5

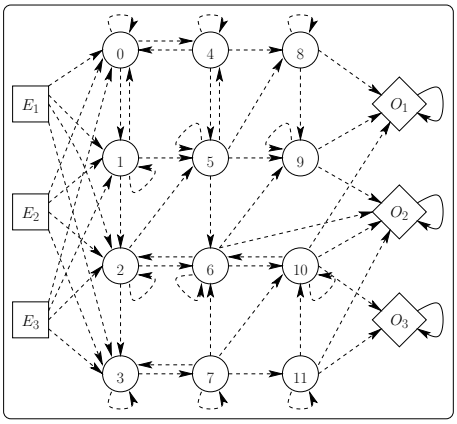
Contraintes additionnelles liées au partitionnement

Sommaire

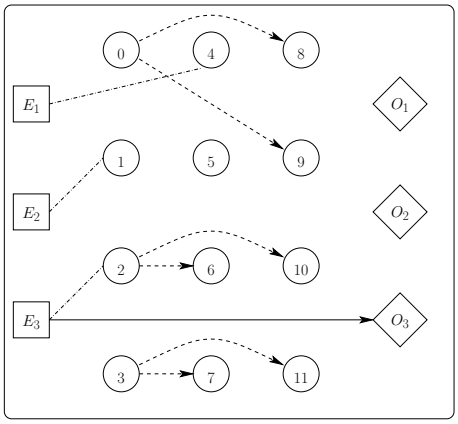
De nombreux problèmes combinatoires reposent sur le partitionnement d'un graphe en un ensemble d'arbres. Bien souvent, le partitionnement seul n'est pas suffisant. En effet, la modélisation du problème nécessite la prise en compte de contraintes additionnelles telles qu'une restriction sur le degré de chaque sommet, des relations de précédences ou d'incomparabilités entre certains sommets du graphe à partitionner.

Le problème de planification de missions [Gue07] est un problème mentionnant de telles contraintes additionnelles hétérogènes. Ainsi, il constitue un bon exemple réel illustrant la nécessité de prendre en compte directement des contraintes additionnelles au cœur de la contrainte *tree* introduite au chapitre 4 page 47. Un problème de planification de missions peut se modéliser par un ensemble de ressources (les sommets E_1, E_2, E_3 de la figure 5.1 page suivante) devant atteindre un certain nombre d'objectifs (les sommets O_1, O_2, O_3 de la figure 5.1 page suivante). Pour cela, les ressources se déplacent dans un environnement, par exemple une ville modélisée sous la forme d'un graphe orienté (les sommets $i \in \{0, 1, \ldots, 11\}$

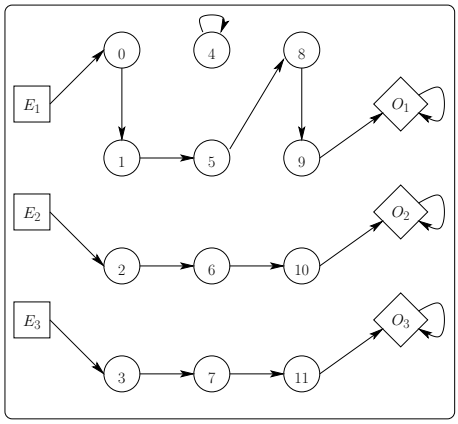
et les arcs de la figure 5.1(a)). Chaque ressource peut être contrainte de sorte que certains sommets de l'environnement lui soient interdits (les ressources E_1, E_2 et E_3 ne peuvent respectivement pas passer par les sommets 4, 1 et 2). Certains objectifs ne peuvent être affectés qu'à certaines ressources (la ressource E_3 est pré-affectée à l'objectif O_3). Ce type de problème nécessite aussi la prise en compte d'objectifs secondaires. Par exemple, on doit pouvoir modéliser le fait qu'atteindre un sommet donné de l'environnement nécessite le passage par d'autres sommets dans la suite du parcours (les arcs $(0,8), (0,9), (2,5), (2,10), (3,7), (3,11)$ de la figure 5.1(b)). Finalement, une solution satisfaisant l'ensemble de ces contraintes est la couverture du graphe modélisant les ressources, l'environnement et les objectifs, par des chemins disjoints (figure 5.1(c)). Remarquons que les sommets non utilisés correspondent à des sommets isolés portant une boucle dans la solution (le sommet 4 de la figure 5.1(c)).



(a) Graphe des scénarios possibles.



(b) Contraintes sur la planification.



(c) Une solution consistante.

FIG. 5.1 – Un exemple de planification de missions avec des contraintes hétérogènes. La figure 5.1(a) représente le graphe à couvrir, c.-à-d., le graphe $\mathcal{G}$ qui sera associé à notre contrainte *tree*. L'objectif est de proposer une couverture de ce graphe par trois chemins débutant sur les sommets E_1, E_2, E_3 et terminant sur les sommets O_1, O_2, O_3. La figure 5.1(b) donne les contraintes de la couverture recherchée : les arcs en trait plein représentent des contraintes de précédences, les arcs en pointillé correspondent à des contraintes de précédences conditionnelles (c.-à-d., s'il existe une contrainte de précédence conditionnelle entre les sommets u et v alors, si un chemin atteint le sommet u, ce même chemin doit ensuite atteindre le sommet v) et les arêtes en pointillé discontinus à des contraintes d'incomparabilités. Finalement, la figure 5.1(c) représente une solution satisfaisant l'ensemble les contraintes représentées par la figure 5.1(b).

Intuitivement, les contraintes sur le degré de chaque sommet permettent de restreindre la forme de la partition à des chemins ou des arbres binaires par exemple. De même les contraintes de pré-

cédence [Que06]) sont nécessaires pour exprimer un ordre partiel entre les sommets du graphe. Les contraintes d'incomparabilité (parfois également appelées contraintes de dominance [ADK+03, Thi04, BDMN04]) expriment que deux sommets ne sont pas situés sur un même chemin. La contrainte sur le nombre d'arbres « propres » permet de faire une distinction sur la forme des arbres dans la couverture. Ainsi, un arbre peut être défini comme une composante connexe sans circuit. Néanmoins, on peut distinguer les arbres contenant au moins deux sommets (arbre propre A. Cayley [Cay89]) des arbres réduits à un sommet isolé. Finalement, les contraintes de précédence conditionnelle permettent d'exprimer une forme conditionnelle des contraintes de précédence « pures ». Une telle contrainte, entre deux sommets u et v du graphe à couvrir, induit que si u appartient à un arbre propre alors u précède v dans cet arbre. Parmi les exemples pratiques d'utilisation de ses contraintes, on pourra citer le problème de reconstruction de super-arbres en phylogénie [ASSU81, BEGS02, GPSW03, NW96, Ste92], le partitionnement d'un graphe par des chemins [CB04, Que06] ou des circuits [Bou99].

Dans le chapitre 4 page 47, nous avons étudié les contraintes *tree* et *proper-tree*. L'objectif de ce chapitre est d'unifier et d'étendre ces deux contraintes en une contrainte *extended-tree* prenant en compte les contraintes additionnelles introduites précédemment :

- *contraintes de précédence* entre sommets : un sommet u *précède* un sommet v dans un graphe $\mathcal{G}$ si u et v appartiennent au même arbre et qu'il existe un chemin de u à v dans cet arbre ;
- *contraintes de précédence conditionnelle* : si deux sommets u et v sont tels que « u précède conditionnellement v » et u, v sont dans le même arbre propre alors u précède v dans la partition ;
- *contraintes d'incomparabilité* : deux sommets u et v sont *incomparables* s'il n'existe pas de chemin de u à v ou de v à u dans la partition ;
- *contraintes de degré* : le *demi-degré négatif* de chaque sommet du graphe (c.-à-d., le nombre de prédécesseurs de chaque sommet) est restreint ;
- *contraintes sur le nombre d'arbres propres* : un *arbre propre* étant défini comme un arbre contenant au moins deux sommets, on force une partition à être composée d'un certain nombre d'arbres propres.

La figure 5.2 présente les paramètres de la contrainte *extended-tree* qui permettent d'exprimer un ensemble de restrictions sur les partitions effectivement autorisées.

La contrainte *extended-tree* étendue est définie par *extended-tree*(`NTREE`, `NPROP`, `VER`), où `NTREE` (le nombre d'arbres de taille au moins 1) et `NPROP` (le nombre d'arbres de taille au moins 2) sont deux variables entières et `VER` représente la collection de sommets du graphe $\mathcal{G}$ associé. Chaque sommet v_i possède les attributs suivants :

- `L` est un entier compris entre 1 et n. Il représente le nom du sommet v_i dans la collection ;
- `S` est une variable entière définie sur l'intervalle $[1, n]$. Elle représente l'unique successeur (ou père) du sommet v_i dans une couverture. Si $i \in \mathcal{D}(\texttt{VER}[i].\texttt{S})$ alors, on dira que v_i est une *racine potentielle* ;
- `P` est un ensemble (éventuellement vide) d'entiers composés d'éléments de $[1, n]$ représentant les noms des sommets du graphe. On peut l'interpréter comme l'ensemble des *descendants obligatoires* du sommet v_i ;
- `C` est un ensemble d'entiers (éventuellement vide) composés d'éléments de l'intervalle $[1, n]$ représentant les noms des sommets du graphe. On dira que si le sommet v_i appartient à un arbre propre alors il doit exister un chemin de v_i vers chacun des sommets de $\texttt{VER}[i].\texttt{C}$;
- `I` est un ensemble (éventuellement vide) d'entiers composés d'éléments de l'intervalle $[1, n]$ qui représentent les noms des sommets du graphe. On peut l'interpréter comme l'ensemble des sommets *incomparables* avec v_i. Notons que la relation d'incomparabilité étant symétrique, pour un sommet v_i l'ensemble des sommets éventuellement incomparables avec v_i est dans l'intervalle $[i + 1, n]$;
- `D` est une variable entière définie sur $[0, n-1]$. Elle peut être interprétée comme le *demi-degré intérieur* de v_i (en ne comptant pas les boucles engendrées par les racines potentielles).

Fig. 5.2 – Paramètres de la contrainte *extended-tree* prenant en compte un ensemble de contraintes additionnelles modélisant des restrictions sur les partitions autorisées.

Un point clé dans la résolution de problèmes combinatoires est d'éviter la découverte répétitive de la même inconsistance (ce phénomène est généralement appelé *thrashing*). Ainsi, la contribution de ce chapitre est un ensemble de conditions nécessaires relatives aux restrictions additionnelles liées à la contrainte *extended-tree*. Ces conditions prennent en compte une partie des interactions existantes entre les contraintes supplémentaires dans le but de prévenir le phénomène de thrashing. Par conséquent, la contrainte *extended-tree* proposée sera en mesure de résoudre différents problèmes pratiques comme le problème de reconstruction de super-arbres en phylogénie et les problèmes de chemins disjoints partiellement ordonnés, qui sont généralement traités par des approches ad-hoc distinctes.

La contrainte *extended-tree* est définie à partir d'un graphe orienté $\mathcal{G}$ tel que proposé dans la définition 34 page 48 : le *graphe des successeurs* $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ est décrit par $\mathcal{V} = \{v_i \mid i \in [1, n]\}$ et $\mathcal{E} = \{(v_i, v_j) \mid j \in \mathcal{D}(\texttt{VER}[i].\texttt{S})\}$. nous rappelons maintenant les notions d'arc sûr et d'arc possible : un arc (i, j) du graphe $\mathcal{G}$ associé à une contrainte *tree* est dit *S-arc* (un *arc sûr*) si $\mathcal{D}(\texttt{VER}[i].\texttt{S}) = \{j\}$, sinon (i, j) est dit *M-arc* (un *arc possible*). Un sommet i est dit *S-succ* si tout arc sortant de i est S-arc, sinon i est dit *M-succ*. De la même manière, un sommet i est dit *S-pred* si tout arc entrant est S-arc, sinon i est dit *M-pred*. Ces notations nous permettent d'introduire les notions de graphes des successeurs surs et possibles associés au graphe $\mathcal{G}$:

v_i	VER[i].L	VER[i].S	Figure 5.3(a) page ci-contre				Figure 5.3(b) page suivante			
			VER[i].P	VER[i].C	VER[i].I	VER[i].D	VER[i].P	VER[i].C	VER[i].I	VER[i].D
v_1	1	$\{1,3\}$	-	-	-	$[0,6]$	-	-	-	$[0,2]$
v_2	2	$\{4,5,6\}$	-	-	-	$[0,6]$	-	-	$\{3\}$	$[0,2]$
v_3	3	$\{1,4\}$	-	-	-	$[0,6]$	$\{1\}$	-	-	$[0,2]$
v_4	4	$\{1,3,6\}$	-	-	-	$[0,6]$	-	-	$\{6\}$	$[0,2]$
v_5	5	$\{2,5\}$	-	-	-	$[0,6]$	-	-	-	$[0,2]$
v_6	6	$\{1,4,5\}$	-	-	-	$[0,6]$	-	-	-	$[0,2]$

v_i	VER[i].L	VER[i].S	Figure 5.3(c) page ci-contre				Figure 5.3(d) page suivante			
			VER[i].P	VER[i].C	VER[i].I	VER[i].D	VER[i].P	VER[i].C	VER[i].I	VER[i].D
v_1	1	$\{1,3\}$	-	-	-	$[0,3]$	-	-	-	$[0,1]$
v_2	2	$\{4,5,6\}$	$\{4\}$	-	-	$[0,3]$	$\{3,4,6\}$	-	-	$[0,1]$
v_3	3	$\{1,4\}$	-	-	$\{6\}$	$[0,3]$	$\{1\}$	-	$\{5\}$	$[0,1]$
v_4	4	$\{1,3,6\}$	-	-	$\{5,6\}$	$[0,3]$	-	-	$\{5\}$	$[0,1]$
v_5	5	$\{2,5\}$	-	-	-	$[0,3]$	-	-	-	$[0,1]$
v_6	6	$\{1,4,5\}$	-	-	-	$[0,3]$	$\{1\}$	-	-	$[0,1]$

v_i	VER[i].L	VER[i].S	Figure 5.3(e) page ci-contre				Figure 5.3(f) page suivante			
			VER[i].P	VER[i].C	VER[i].I	VER[i].D	VER[i].P	VER[i].C	VER[i].I	VER[i].D
v_1	1	$\{1,3\}$	$\{5\}$	-	$\{2\}$	$[0,2]$	-	-	-	$[0,1]$
v_2	2	$\{4,5,6\}$	-	-	$\{3\}$	$[0,2]$	-	$\{4\}$	-	$[0,1]$
v_3	3	$\{1,4\}$	-	-	-	$[0,2]$	-	-	-	$[0,1]$
v_4	4	$\{1,3,6\}$	-	-	-	$[0,2]$	-	-	-	$[0,1]$
v_5	5	$\{2,5\}$	-	-	-	$[0,2]$	-	-	-	$[0,1]$
v_6	6	$\{1,4,5\}$	-	-	-	$[0,2]$	-	$\{3\}$	-	$[0,1]$

TAB. 5.1 – Les trois premières colonnes décrivent le graphe $\mathcal{G}$ associé à la contrainte *extended-tree*. Chacune des autres colonnes représente un ensemble de contraintes supplémentaires pour laquelle chaque sous-figure de la figure 5.3 page suivante donne une solution possible.

Définition 38 (Graphes des successeurs surs et possibles)**.** *Étant donné une contrainte* extended-tree *et son graphe associé* $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ *:*

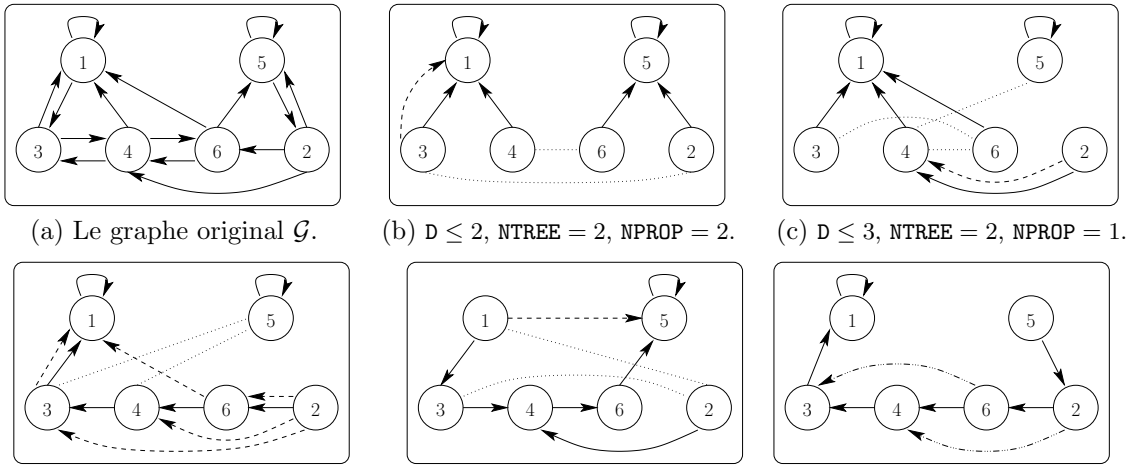
- *Le graphe des* successeurs surs $\mathcal{G}_{sure}$ *contient tous les arcs qui doivent appartenir à la partition. Formellement,* $\mathcal{G}_{sure} = (\mathcal{V}, \mathcal{E}_{sure})$*, où* $\mathcal{E}_{sure}$ *est l'ensemble des S-arcs de* $\mathcal{G}$ *;*
- *Le graphe des* successeurs possibles $\mathcal{G}_{maybe}$ *contient tous les arcs qui peuvent éventuellement appartenir à une partition. Formellement,* $\mathcal{G}_{maybe} = (\mathcal{V}_{maybe}, \mathcal{E}_{maybe})$*, où* $\mathcal{V}_{maybe}$ *contient tous les sommets qui sont incidents à au moins un M-arc, et* $\mathcal{E}_{maybe}$ *est l'ensemble des M-arcs de* $\mathcal{G}$*.*

La contrainte *extended-tree*(NTREE, NPROP, VER) impose que le graphe $\mathcal{G}$ qui lui est associé soit une forêt composée de NTREE arbres, dont NPROP sont des arbres propres, telle que toutes les contraintes de

précédence, de précédence conditionnelle, d'incomparabilité, et de degré soient vérifiées. Formellement, on a :

Définition 39. *Un graphe orienté* $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ satisfait *la contrainte d'arbre étendue* extended-tree(NTREE, NPROP, VER) *si et seulement si :*

- *pour chaque* $i \in [1, n]$ *on a* VER$[i]$.L $= i$ *;*
- $\mathcal{G}$ *est composé de* NTREE *composantes connexes ;*
- $\mathcal{G}$ *est composé de* NPROP *composantes connexes de taille au moins deux ;*
- *à l'exception de la boucle sur la racine, aucune composante connexe de* $\mathcal{G}$ *ne contient de circuits ;*
- *pour chaque sommet* i*, il existe un chemin élémentaire dans* $\mathcal{G}$ *de* i *à* j *pour tout* j *appartenant à* VER$[i]$.P *;*
- *pour chaque sommet* i*, quel que soit* j *dans l'ensemble* VER$[i]$.C*, il existe un chemin élémentaire dans* $\mathcal{G}$ *de* i *à* j *ssi* j *appartient à une composante connexe comportant au moins deux sommets ;*
- *pour chaque sommet* i*, quelque soit* j *appartenant à* VER$[i]$.I*, il n'existe aucun chemin de* i *à* j *et aucun chemin de* j *à* i *dans* $\mathcal{G}$ *;*
- *pour chaque sommet* i*, il existe* VER$[i]$.D *sommets* j*, avec* j *distinct de* i*, tels que* j *soit un prédécesseur de* i *(c.-à-d.,* VER$[j]$.S $= i$*) dans* $\mathcal{G}$ *;*



(a) Le graphe original $\mathcal{G}$. (b) D ≤ 2, NTREE $= 2$, NPROP $= 2$. (c) D ≤ 3, NTREE $= 2$, NPROP $= 1$.

(d) D ≤ 1, NTREE $= 2$, NPROP $= 1$. (e) D ≤ 2, NTREE $= 1$, NPROP $= 1$. (f) D ≤ 1, NTREE $= 1$, NPROP $= 1$.

FIG. 5.3 – (a) Le graphe $\mathcal{G}$ associé à la contrainte *extended-tree* détaillée dans l'exemple 8. Dans les sous-figures (b,c,d,e,f) 5 solutions vérifiant les contraintes de précédence, de précédence conditionnelle, d'incomparabilité, et de degré, décrites dans la table 5.1 page précédente, sont représentées avec des arcs en trait plein. Les contraintes de précédence sont représentées par des arcs avec des tirets , les contraintes de précédence conditionnelle par des arcs avec des tirets non-continus, et les contraintes d'incomparabilité par des arcs en pointillé.

Exemple 8. *La figure 5.3(a) et les trois premières colonnes de la table 4.3 page 49 représentent le graphe orienté* $\mathcal{G}$ *pour lequel cinq solutions sont présentées dans les figures 5.3(b) à 5.3(f) en accord avec les contraintes décrites dans la table 4.3 page 49 :*

- *partie (b) :* $\mathcal{D}$(NTREE) $= \{2\} = \mathcal{D}$(NPROP)*,* $\mathcal{D}$(VER$[i]$.D) $= [0, 2]$*, pour chaque sommet* v_i *de* $\mathcal{G}$*, le sommet* v_3 *doit précéder le sommet* v_1*, et les paires de sommets* (v_2, v_3) *et* (v_4, v_6) *sont incomparables ;*
- *partie (c) :* $\mathcal{D}$(NTREE) $= \{2\}$*,* $\mathcal{D}$(NPROP) $= \{1\}$*,* $\mathcal{D}$(VER$[i]$.D) $= [0, 3]$*, pour chaque sommet* v_i *de* $\mathcal{G}$*, le sommet* v_2 *doit précéder le sommet* v_4*, et les paires de sommets* (v_3, v_6)*,* (v_4, v_5) *et* (v_4, v_6) *sont incomparables ;*
- *partie (d) :* $\mathcal{D}$(NTREE) $= \{2\}$*,* $\mathcal{D}$(NPROP) $= \{1\}$*,* $\mathcal{D}$(VER$[i]$.D) $= [0, 1]$*, pour chaque sommet* v_i *de* $\mathcal{G}$*, les arcs* (v_2, v_3)*,* (v_2, v_4)*,* (v_2, v_6)*,* (v_3, v_1)*, et* (v_6, v_1) *représentent des contraintes de précédence, et les paires de sommets* (v_3, v_5) *et* (v_4, v_5) *sont incomparables ;*
- *partie (e) :* $\mathcal{D}$(NTREE) $= \{1\} = \mathcal{D}$(NPROP)*,* $\mathcal{D}$(VER$[i]$.D) $= [0, 2]$*, pour chaque sommet* v_i *de* $\mathcal{G}$*, le sommet* v_1 *précède le sommet* v_5*, et les paires de sommets* (v_1, v_2) *et* (v_2, v_3) *sont incomparables ;*

– *partie (f) :* $\mathcal{D}(\texttt{NTREE}) = \{1\} = \mathcal{D}(\texttt{NPROP})$, $\mathcal{D}(\texttt{VER}[i].\texttt{D} = [0,1]$, *pour chaque sommet* v_i *de* $\mathcal{G}$, *si le sommet* v_2 *(resp.* v_6*) appartient à un arbre comportant au moins deux sommets alors le sommet* v_2 *(resp.* v_6*) précède le sommet* v_4 *(resp.* v_3*) ;*

Dans la suite de ce chapitre, nous allons détailler des conditions nécessaires ainsi que des règles de filtrage relatives à chaque restriction prise en compte dans la contrainte *extended-tree*. Tout d'abord, la section 5.1 étudiera la complexité théorique de la contrainte *extended-tree*. La section 5.2 page 73 s'intéressera ensuite à l'étude des interactions évidentes entre les variables NTREE et NPROP. Puis, les sections 5.3 page 74, 5.4 page 77 et 5.5 page 79 étudieront respectivement les contraintes de précédence, de précédence conditionnelle et d'incomparabilité. La section 5.6 page 80 s'intéressera aux interactions liant les contraintes de précédence et d'incomparabilité. La section 5.7 page 83 étudiera les restrictions sur le degré de chaque sommet du graphe à partitionner. Finalement, la section 5.8 page 84 proposera une synthèse sur les complexités de la prise en compte de chacune des restrictions.

5.1 Étude de la complexité théorique

L'objectif de cette section est de clarifier la complexité théorique d'un certain nombre des restrictions prises en compte dans la contrainte *extended-tree*. Nous nous intéressons en particulier à l'étude des paramètres qui font de la propagation de l'arc consistance généralisée (GAC) un problème NP-complet. Cependant, il faut noter que dans le contexte d'une contrainte *extended-tree* ne prenant en compte qu'un ensemble de contraintes d'incomparabilité, nous n'avons pas su classer la complexité de propager la GAC pour la contrainte *extended-tree*.

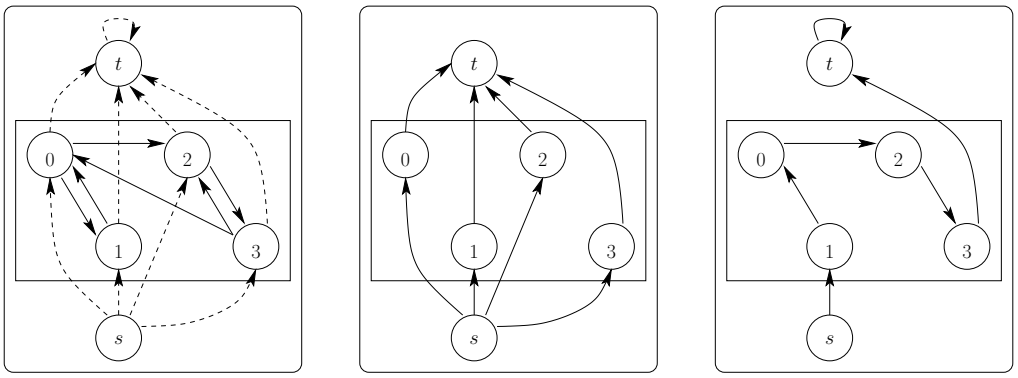
5.1.1 Le cas où NPROP seul est contraint

Dans cette section, nous nous intéressons à la complexité théorique de la contrainte *extended-tree* dans le cas où le nombre d'arbres propres est contraint. Dans ce contexte, nous faisons les hypothèses suivantes :
- $\mathcal{D}(\texttt{NPROP}) \subseteq \mathcal{D}(\texttt{NTREE})$, c.-à-d. que la variable NPROP est contrainte ;
- il n'existe pas de contraintes de précédences entre les sommets du graphe, c.-à-d. que pour tout sommet i de $\mathcal{G}$, on a $\texttt{VER}[i].\texttt{P} = \emptyset$;
- il n'existe pas de contraintes de précédences conditionnelles entre les sommets du graphe, c.-à-d. que pour tout sommet i de $\mathcal{G}$, on a $\texttt{VER}[i].\texttt{C} = \emptyset$;
- il n'existe pas de contraintes d'incomparabilités entre les sommets du graphe, c.-à-d. que pour tout sommet i de $\mathcal{G}$, on a $\texttt{VER}[i].\texttt{I} = \emptyset$;
- le demi-degré intérieur de chaque sommet de $\mathcal{G}$ n'est pas contraint, c.-à-d. que pour tout sommet i de $\mathcal{G}$, on a $\texttt{VER}[i].\texttt{D} = [0, n]$.

Dans ce contexte, la contrainte *extended-tree* est en tout point similaire à la contrainte *proper-tree* introduite au chapitre 4 page 47 (nous en discuterons plus en détails en section 5.2 page 73). Par conséquent, la contrainte *extended-tree* ne peut pas être propagée, en temps polynomial, jusqu'à atteindre l'arc consistance généralisée. En effet, nous avons déja montré pour la contrainte *proper-tree* que propager l'arc-consistance généralisée pour une telle contrainte est un problème NP-complet (voir théorème 8 page 53).

5.1.2 Le cas où l'ensemble des contraintes de précédence est non vide

Dans cette section, nous nous intéressons à la complexité théorique de la contrainte *extended-tree* par rapport à un ensemble de contraintes de précédence. Nous faisons les hypothèses suivantes :
- $\mathcal{D}(\texttt{NTREE}) \subset \mathcal{D}(\texttt{NPROP})$, c.-à-d. que la variable NPROP n'est pas contrainte ;
- l'ensemble des contraintes de précédence entre les sommets du graphe est non vide, c.-à-d. que pour tout sommet i de $\mathcal{G}$, on a $\texttt{VER}[i].\texttt{P} \neq \emptyset$;
- il n'existe pas de contraintes de précédence conditionnelle entre les sommets du graphe, c.-à-d. que pour tout sommet i de $\mathcal{G}$, on a $\texttt{VER}[i].\texttt{C} = \emptyset$;
- il n'existe pas de contraintes d'incomparabilité entre les sommets du graphe, c.-à-d. que pour tout sommet i de $\mathcal{G}$, on a $\texttt{VER}[i].\texttt{I} = \emptyset$;



(a) Le graphe $\mathcal{G}$ associé à *extended-tree*.

(b) Le graphe $\mathcal{G}_{prec}$ associé à *extended-tree*.

(c) Une solution satisfaisant *extended-tree*$(1, [0, n], \texttt{VER})$.

FIG. 5.4 – Modélisation d'un problème de chemin Hamiltonien par une contrainte *extended-tree*, avec $\texttt{NTREE} = \texttt{NPROP} = 1$. La figure 5.4(a) représente le graphe $\mathcal{G}$ associé à la contrainte *extended-tree*. Les sommets contenus dans le rectangle représentent le graphe $\mathcal{H}$ pour lequel on cherche à statuer sur l'existence d'un chemin Hamiltonien. La figure 5.4(b) dépeint le graphe de précédence $\mathcal{G}_{prec}$ associé à la contrainte *extended-tree*. Finalement, la figure 5.4(c) donne une solution satisfaisant la contrainte *extended-tree*.

– le demi-degré intérieur de chaque sommet de $\mathcal{G}$ n'est pas contraint, c.-à-d. que pour tout sommet i de $\mathcal{G}$, on a $\texttt{VER}[i].\texttt{D} = [0, n]$.

Dans ce contexte, nous montrons maintenant que la contrainte *extended-tree* ne peut pas être propagée, en temps polynomial, jusqu'à atteindre l'arc-consistance généralisée.

Théorème 11. *Propager la GAC pour une contrainte* extended-tree *par rapport aux contraintes de précédences est NP-complet.*

Démonstration. En premier lieu, nous décrivons un certificat polynomial pour la contrainte *extended-tree* par rapport à ses contraintes de précédences. Il s'agit d'un algorithme polynomial déterministe vérifiant qu'une assignation totale des variables mises en jeu dans la contrainte *extended-tree* est une solution valide, c.-à-d., satisfaisant la contrainte. Nous commençons par vérifier qu'une assignation complète des variables de *extended-tree* constitue une forêt contrainte : (1) le graphe orienté $\mathcal{G}_{sure}$ associé à *extended-tree* contient NTREE composantes connexes dont NPROP contiennent au moins deux sommets, et (2) $\mathcal{G}_{sure}$ ne contient aucun circuit mettant en jeu plus d'un sommet (c.-à-d. une boucle sur un sommet). Ensuite, pour chaque arc (u, v) de $\mathcal{G}_{prec}$, nous vérifions qu'il existe bien un chemin de u à v dans $\mathcal{G}_{sure}$ (en utilisant un algorithme de recherche en profondeur d'abord depuis le sommet u dans $\mathcal{G}_{sure}$).

Maintenant, nous montrons que toute instance du problème NP-complet de chemin Hamiltonien[1] [GJ78] peut être reformulée en temps polynomial comme une contrainte d'arbre (figure 5.4). Soit $\mathcal{H} = (\mathcal{N}, \mathcal{U})$ le graphe pour lequel nous cherchons à statuer sur l'existence d'un chemin Hamiltonien le couvrant. Nous définissons le graphe $\mathcal{G}$ associé à la contrainte *extended-tree*, ainsi que le graphe de précédence $\mathcal{G}_{prec}$. Soit $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ où $\mathcal{V} = \mathcal{N} \cup \{s, t\}$ l'ensemble des sommets du graphe à couvrir et $\mathcal{E} = \mathcal{U} \cup \{(s, j) \mid j \neq s,\ j \in \mathcal{V}\} \cup \{(i, t) \mid i \neq t,\ i \in \mathcal{V}\}$ l'ensemble des arcs possibles. Moins formellement, s est l'unique prédécesseur dans $\mathcal{G}$ de tous les sommets de $\mathcal{N}$ et t est le successeur de tous les sommets de $\mathcal{U}$. Soit $\mathcal{G}_{prec}$ le graphe de précédence défini par $(\mathcal{V}, \{(s, j) \mid j \neq s,\ j \in \mathcal{V}\} \cup \{(i, t) \mid i \neq t,\ i \in \mathcal{V}\})$. Le problème de chemin Hamiltonien consiste alors à trouver une partition de $\mathcal{G}$ en $\texttt{NTREE} = 1$ arbre satisfaisant les contraintes de précédence définies par $\mathcal{G}_{prec}$. □

[1] Étant donné un graphe orienté $\mathcal{H} = (\mathcal{N}, \mathcal{U})$, le *problème de chemin Hamiltonien* statue sur l'existence d'un chemin élémentaire contenant tous les sommets de $\mathcal{N}$.

5.1.3 Le cas où l'ensemble des contraintes de précédence conditionnelle est non vide

Cette section s'intéresse à la complexité théorique de la contrainte *extended-tree* par rapport à un ensemble de contraintes de précédence conditionnelle. Dans ce contexte, nous faisons les hypothèses suivantes :
- $\mathcal{D}(\texttt{NTREE}) \subset \mathcal{D}(\texttt{NPROP})$, c.-à-d. que la variable NPROP n'est pas contrainte ; ;
- il n'existe pas de contraintes de précédences entre les sommets du graphe, c.-à-d. que pour tout sommet i de $\mathcal{G}$, on a $\texttt{VER}[i].\texttt{P} = \emptyset$;
- l'ensemble des contraintes de précédences conditionnelles entre les sommets du graphe est non vide, c.-à-d. que pour tout sommet i de $\mathcal{G}$, on a $\texttt{VER}[i].\texttt{C} \neq \emptyset$;
- il n'existe pas de contraintes d'incomparabilités entre les sommets du graphe, c.-à-d. que pour tout sommet i de $\mathcal{G}$, on a $\texttt{VER}[i].\texttt{I} = \emptyset$;
- le demi-degré intérieur de chaque sommet de $\mathcal{G}$ n'est pas contraint, c.-à-d. que pour tout sommet i de $\mathcal{G}$, on a $\texttt{VER}[i].\texttt{D} = [0, n]$.

Finalement, nous montrons maintenant que la contrainte *extended-tree* ne peut pas être propagée, en temps polynomial, jusqu'à atteindre l'arc-consistance généralisée. Intuitivement, il s'agit de la même preuve que pour le théorème 5.1.2 page 70 lié aux contraintes de précédence. En effet, en fixant $\texttt{NTREE} = 1$, nous nous assurons que toute contrainte de précédence conditionnelle est sémantiquement équivalente à une contrainte de précédence classique (puisqu'elles vont appartenir à coup sûr à un arbre propre). Ceci devient évident en considérant que $|\mathcal{V}| \geq 2$ sachant qu'alors on doit avoir tous les sommets du graphe $\mathcal{G}$ à partitionner dans le même arbre.

Théorème 12. *Propager la GAC pour une contrainte* extended-tree *par rapport aux contraintes de précédence conditionnelle est NP-complet.*

Démonstration. Voir théorème 5.1.2 page 70. □

5.1.4 Le cas où le demi-degré intérieur des sommets du graphe est contraint

Dans cette section, nous nous intéressons à la complexité théorique de la contrainte *extended-tree* par rapport à un ensemble de contraintes sur le demi-degré intérieur de chaque sommet. Nous faisons les hypothèses suivantes :
- $\mathcal{D}(\texttt{NTREE}) \subset \mathcal{D}(\texttt{NPROP})$, c.-à-d. que la variable NPROP n'est pas contrainte ;
- il n'existe pas de contraintes de précédence entre les sommets du graphe, c.-à-d. que pour tout sommet i de $\mathcal{G}$, on a $\texttt{VER}[i].\texttt{P} = \emptyset$;
- il n'existe pas de contraintes de précédence conditionnelle entre les sommets du graphe, c.-à-d. que pour tout sommet i de $\mathcal{G}$, on a $\texttt{VER}[i].\texttt{C} = \emptyset$;
- il n'existe pas de contraintes d'incomparabilité entre les sommets du graphe, c.-à-d. que pour tout sommet i de $\mathcal{G}$, on a $\texttt{VER}[i].\texttt{I} = \emptyset$;
- le demi-degré intérieur de chaque sommet de $\mathcal{G}$ est contraint, c.-à-d. que pour tout sommet i de $\mathcal{G}$, on a $\texttt{VER}[i].\texttt{D} \subset [0, n]$.

Finalement, nous montrons maintenant que la contrainte *extended-tree* ne peut pas être propagée, en temps polynomial, jusqu'à atteindre l'arc-consistance généralisée.

Théorème 13. *Propager la GAC pour une contrainte* extended-tree *par rapport aux contraintes de degré est NP-complet.*

Démonstration. En premier lieu, nous allons décrire un certificat polynomial pour la contrainte *extended-tree* par rapport à ses contraintes de degré. Il s'agit d'un algorithme polynomial déterministe vérifiant qu'une assignation totale des variables mises en jeu dans la contrainte *tree* est une solution valide, c.-à-d., satisfaisant la contrainte. Nous commençons par vérifier qu'une assignation complète des variables de *tree* constitue une forêt contrainte par (1) le graphe orienté $\mathcal{G}_{sure}$ associé à *extended-tree* contient NTREE composantes connexes dont NPROP contiennent au moins deux sommets, et (2) $\mathcal{G}_{sure}$ ne contient aucun circuit mettant en jeu plus d'un sommet (c.-à-d. une boucle sur un sommet). Ensuite, pour chaque sommet i du graphe $\mathcal{G}_{sure}$, nous vérifions que le demi-degré intérieur de i est égal à $\texttt{VER}[i].\texttt{D}$.

Maintenant, nous montrons que toute instance du problème NP-complet de chemin Hamiltonien [GJ78] peut être reformulée en temps polynomial comme une contrainte d'arbre. Pour cela, nous considérons le

Algorithme 9 Filtrer une contrainte *extended-tree*(NTREE, NPROP, VER) par rapport à NTREE et NPROP.

```
1. Appliquer l'algorithme 5 page 51 relatif à la contrainte tree ;
2. Appliquer les algorithmes 7 page 59 et 8 page 60 relatifs à la contrainte proper-tree ;
/* Restriction de NPROP et NTREE relative à l'interaction */
3. Si min(NPROP) > max(NTREE) (théorème 14) alors Retourner un échec ;
4. Si min(NTREE) ≤ min(NPROP) alors min(NTREE) ← min(NPROP) ;
5. Si max(NTREE) ≤ max(NPROP) alors max(NPROP) ← max(NTREE) ;
```

graphe orienté $\mathcal{H} = (\mathcal{U}, \mathcal{A})$ pour lequel on cherche à statuer sur l'existence d'un chemin Hamiltonien. Associons à la contrainte *extended-tree*, le graphe $\mathcal{G}$ défini par $(\mathcal{H} \cup \{r\}, \mathcal{A} \cup \{(u, r) \mid u \in \mathcal{H} \cup \{r\}\})$, et pour chaque sommet i de $\mathcal{G}$ fixons $\texttt{VER}[i].\texttt{D} = [0, 1]$, c.-à-d., que chaque sommet de $\mathcal{G}$ peut être atteint au plus une fois. Le problème de chemin Hamiltonien consiste alors à trouver une partition de $\mathcal{G}$ en $\texttt{NTREE} = 1$ arbre satisfaisant les contraintes de degré. □

5.1.5 Le cas où l'ensemble des contraintes d'incomparabilités est non vide

Dans le contexte d'une contrainte *extended-tree* définie uniquement par un ensemble de contraintes d'incomparabilité, c.-à-d. que les hypothèses suivantes sont vérifiées :

- $\mathcal{D}(\texttt{NTREE}) \subset \mathcal{D}(\texttt{NPROP})$, c.-à-d. que la variable NPROP n'est pas contrainte ;
- il n'existe pas de contraintes de précédences entre les sommets du graphe, c.-à-d. que pour tout sommet i de $\mathcal{G}$, on a $\texttt{VER}[i].\texttt{P} = \emptyset$;
- il n'existe pas de contraintes de précédences conditionnelles entre les sommets du graphe, c.-à-d. que pour tout sommet i de $\mathcal{G}$, on a $\texttt{VER}[i].\texttt{C} = \emptyset$;
- l'ensemble des contraintes d'incomparabilités entre les sommets du graphe est non vide, c.-à-d. que pour tout sommet i de $\mathcal{G}$, on a $\texttt{VER}[i].\texttt{I} \neq \emptyset$;
- le demi-degré intérieur de chaque sommet de $\mathcal{G}$ n'est pas contraint, c.-à-d. que pour tout sommet i de $\mathcal{G}$, on a $\texttt{VER}[i].\texttt{D} = [0, n]$.

nous n'avons pas pu statuer sur la complexité théorique de la propagation de l'arc-consistance généralisée.

5.2 Interaction entre les variables NTREE et NPROP

Comme nous l'avons dit dans la section 4.3 page 53, de nombreuses applications pratiques ne nécessitent pas directement la couverture de tous les sommets du graphe $\mathcal{G}$ associé à la contrainte *extended-tree*. Tout d'abord, il faut remarquer que par définition de la contrainte *extended-tree*, les algorithmes de filtrage relatifs aux contraintes *tree* et *proper-tree* (introduites au chapitre 4 page 47) restent valides dans le cas de la contrainte *extended-tree*. L'objet de cette section est d'étudier l'interaction entre les variables NTREE et NPROP de la contrainte *extended-tree*. Nous introduirons tout d'abord une condition nécessaire à la faisabilité de la contrainte *extended-tree* par rapport aux variables NTREE et NPROP.

Nous commençons par introduire une condition nécessaire assurant l'inégalité $\texttt{NTREE} \geq \texttt{NPROP}$ pour la contrainte *extended-tree* :

Théorème 14. *S'il existe une solution pour la contrainte* extended-tree(NTREE, NPROP, VER) *alors, l'inégalité* $\min(\texttt{NPROP}) \leq \max(\texttt{NTREE})$ *est vérifiée entre ces deux variables.*

Démonstration. Dans toute partition du graphe $\mathcal{G}$ associé à une contrainte *extended-tree*(NTREE, NPROP, VER), il ne peut pas exister plus d'arbres propres que d'arbres. En effet, un arbre propre contient au moins deux sommets dont une racine potentielle, alors qu'un arbre contient au moins un sommet dont une racine potentielle. □

Nous sommes maintenant en mesure d'introduire le filtrage relatif aux bornes de NTREE et NPROP. L'algorithme 9 utilise tout d'abord le filtrage relatif aux contraintes *tree* et *proper-tree* introduit au chapitre précédant. Puis, la condition de faisabilité sur les domaines de NTREE et NPROP est vérifiée. Enfin, les domaines de NTREE et NPROP sont ajustés si nécessaire.

5.3 Relation de précédence entre les sommets du graphe

Nous nous intéressons dans cette section au traitement des contraintes de précédence. Rappelons que ces contraintes de précédence sont fournies par l'attribut `P` de la collection de sommets `VER` du graphe à partitionner. Tout d'abord, nous proposons une forme normale de l'ensemble de ces contraintes de précédence basée sur un graphe orienté acyclique (DAG) ne contenant aucun arc transitif. Puis, nous étudierons ces contraintes au travers (1) d'une borne supérieure sur le nombre d'arbres autorisés `NTREE`, (2) de conditions nécessaires quant à l'existence d'une solution satisfaisant un ensemble de contraintes de précédence, et (3) de règles de filtrage extraites des conditions nécessaires.

Définition 40 (Graphe de précédences et graphe de précédences contracté)**.** *Étant donné une contrainte* extended-tree *et le graphe associé* $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ *:*

- *le* graphe de précédences $\mathcal{G}_{prec}$ *de* $\mathcal{G}$ *est défini par* $TR(\mathcal{V}, \{(i,j) \in \mathcal{V}^2 \mid i \in \texttt{VER}[j].\texttt{P}\})$ *;*
- *le* graphe de précédences contracté $\mathcal{G}^H_{prec}$ *de* $\mathcal{G}$ *est défini par :*
  - *pour chaque composante connexe de* $\mathcal{G}_{sure}$*, il existe un sommet de* $\mathcal{G}^H_{prec}$ *;*
  - *il existe un arc entre deux sommets de* $\mathcal{G}^H_{prec}$ *ssi il existe un arc dans* $\mathcal{G}_{prec}$ *entre deux sommets appartenant à des composantes connexes distinctes de* $\mathcal{G}_{sure}$ *;*

Dans la suite de ce chapitre, nous ferons l'hypothèse que $\mathcal{G}_{prec}$ ne contient aucun sommet portant une boucle sur lui-même, c.-à-d., nous interdisons tout arc pouvant être interprété comme le fait qu'un sommet de $\mathcal{G}_{prec}$ se précède lui-même. De plus, notons que tout arc de $\mathcal{G}_{sure}$ qui ne correspond pas à une boucle s'interprète comme une contrainte de précédence. Ceci équivaut à maintenir l'invariant suivant :

$$\mathcal{G}_{sure} \setminus \{(u,v) \in \mathcal{E}_{sure} \mid u = v\} \subseteq \mathcal{G}_{prec} \tag{5.1}$$

Cet invariant permet de s'assurer que le graphe des arcs surs $\mathcal{G}_{sure}$, privé de ces boucles, doit toujours être un graphe partiel du graphe de précédences $\mathcal{G}_{prec}$.

5.3.1 Limitations du nombre maximum d'arbres

Nous étudions en premier lieu une borne supérieure sur le nombre d'arbres autorisés pour partitionner le graphe $\mathcal{G}$ associé à la contrainte d'arbre *extended-tree*, en prenant partiellement en compte les contraintes de précédences définies par $\mathcal{G}_{prec}$. Une première borne supérieure donnée en section 4.2.1 page 49 pour le cas de la contrainte *tree* et notée `MAXTREE`, consistait à dénombrer les racines du graphe $\mathcal{G}$. L'intuition de la nouvelle borne repose sur le dénombrement des composantes connexes de $\mathcal{G}_{prec}$ qui doivent nécessairement venir se connecter à une autre composante connexe de $\mathcal{G}_{prec}$. Ainsi, une valeur booléenne out_i est associée à chaque composante connexe $CC(i)$ de $\mathcal{G}_{prec}$, et sa valeur dépend de l'existence dans $CC(i)$ d'un sommet dont aucun successeur n'est contenu dans $CC(i)$:

$$out_i = \begin{cases} 1 & \text{si } \exists u \in CC(i) : \forall v \in \mathcal{D}(\texttt{VER}[u].\texttt{S}) : v \notin CC(i) \\ 0 & \text{sinon} \end{cases}$$

Ainsi, la composante connexe $CC(i)$ devra être obligatoirement fusionnée avec une autre composante connexe dès lors que out_i est égal à 1 :

Proposition 3. *Étant donné une contrainte d'arbre* extended-tree *et son graphe de précédences* $\mathcal{G}_{prec}$ *telle que* $\mathcal{G}_{prec}$ *est partitionnée en* k *composantes connexes. Une borne supérieure de* `NTREE` *est donnée par :*

$$\texttt{MAXTREE}_{prec} = k - \sum_{i=1}^{k} out_i \tag{5.2}$$

Démonstration. Directement fournie par la définition de out_i. □

Proposition 4. $\texttt{MAXTREE}_{prec}$ *est une borne supérieure plus fine que* `MAXTREE`*.*

Démonstration. Soit $p = \texttt{MAXTREE}$ le nombre de racines potentielles de $\mathcal{G}$. Nous savons que :

$$k - p \leq \sum_{i=1}^{k} out_i \leq k$$

Ainsi :

$$0 \leq k - \sum_{i=1}^{k} out_i \leq p$$

et donc, $0 \leq \texttt{MAXTREE}_{prec} \leq \texttt{MAXTREE}$. □

5.3.2 Filtrer la contrainte *extended-tree* par rapport à un ensemble de précédences

Puisque la recherche d'une condition nécessaire et suffisante pour la contrainte *extended-tree* par rapport aux contraintes de précédence est un problème NP-difficile (théorème 11 page 71), nous exhibons maintenant une condition nécessaire quant à l'existence de solutions pour une telle contrainte. Cette condition nécessaire est constituée de quatre conditions, chacune maintenant une propriété de la partition cherchée :

1. maintenir un nombre compatible d'arbres autorisés pour partitionner le graphe ;
2. garantir que la partition est sans circuits ;
3. satisfaire toutes les contraintes de précédences dans la partition ;
4. assurer que chaque arbre est enraciné sur une racine potentielle.

Théorème 15. *S'il existe une solution satisfaisant la contrainte* extended-tree *alors, les conditions suivantes sont vérifiées :*

1. ***nombre d'arbres :*** $\mathcal{D}(\texttt{NTREE}) \cap [\texttt{MINTREE}, \texttt{MAXTREE}_{prec}] \neq \emptyset$*, où* MINTREE *est le nombre de composantes puits dans* $\mathcal{G}$ *et* $\texttt{MAXTREE}_{prec}$ *est défini par la proposition 3 page ci-contre ;*
2. ***pas de circuit :*** $\mathcal{G}_{prec}$ *ne contient pas de circuit ;*
3. ***compatibilité :*** *la fermeture transitive de* $\mathcal{G}_{prec}$ *est inclue dans la fermeture transitive de* $\mathcal{G}$ *(c.-à-d.,* $TC(\mathcal{G}_{prec}) \subseteq TC(\mathcal{G})$*) ;*
4. ***racines compatibles :*** *pour chaque composante puits* $\mathcal{S}$ *de* $\mathcal{G}$*, au moins un sommet est à la fois une racine potentielle dans* $\mathcal{G}$ *et un puits dans* $\mathcal{G}_{prec}$*.*

Démonstration. Une preuve est fournie pour chaque condition :

1. Si $\mathcal{D}(\texttt{NTREE}) \cap [\texttt{MINTREE}, \texttt{MAXTREE}_{prec}] = \emptyset$ alors, $\max(\texttt{NTREE}) < \texttt{MINTREE}$ ou $\min(\texttt{NTREE}) > \texttt{MAXTREE}_{prec}$. Si la première inégalité est vérifiée alors, il existe plus de composantes puits distinctes dans $\mathcal{G}$ que le nombre maximum d'arbres autorisés. Comme nous savons qu'il n'existe pas de chemin entre deux sommets appartenant à des composantes puits distinctes de $\mathcal{G}$ alors, il n'existe pas de solution satisfaisant la contrainte *extended-tree*. La proposition 3 page précédente assure que si la seconde inégalité est vérifiée alors, la contrainte *extended-tree* ne peut pas être satisfaite.
2. Supposons que $\mathcal{G}_{prec}$ contienne un circuit et qu'il existe une solution satisfaisant la contrainte *extended-tree*. Alors, il existe deux chemins $P = \langle u_1, \ldots, u_k \rangle$ et $P' = \langle u_k, \ldots, u_1 \rangle$ dans $\mathcal{G}_{prec}$. Le chemin P force le sommet u_1 à précéder le sommet u_k dans toute solution. De manière équivalente, le chemin P' force le sommet u_k à précéder le sommet u_1 dans toute solution. Mais alors, il n'existe aucune solution pouvant satisfaire simultanément P et P' : c'est une contradiction.
3. Supposons que $TC(\mathcal{G}_{prec}) \not\subseteq TC(\mathcal{G})$. Alors, il existe au moins un arc (u, v) dans $TC(\mathcal{G}_{prec})$ tel que (u, v) n'appartienne pas à $TC(\mathcal{G})$. Ceci signifie qu'il existe au moins une contrainte de précédence qui ne peut pas être satisfaite car il n'existe pas de chemin de u à v dans $\mathcal{G}$.
4. Supposons au contraire qu'il existe une composante puits S telle que chaque racine potentielle r possède au moins un successeur dans $\mathcal{G}_{prec}$. Dans ce cas, chaque racine potentielle devant précéder un sommet de $\mathcal{G}$, la composante puits S ne peut avoir de racine.

Chaque condition étant nécessaire, leur conjonction est aussi une condition nécessaire.

□

Algorithme 10 Filtrer la contrainte *extended-tree* par rapport à un ensemble de précédences.

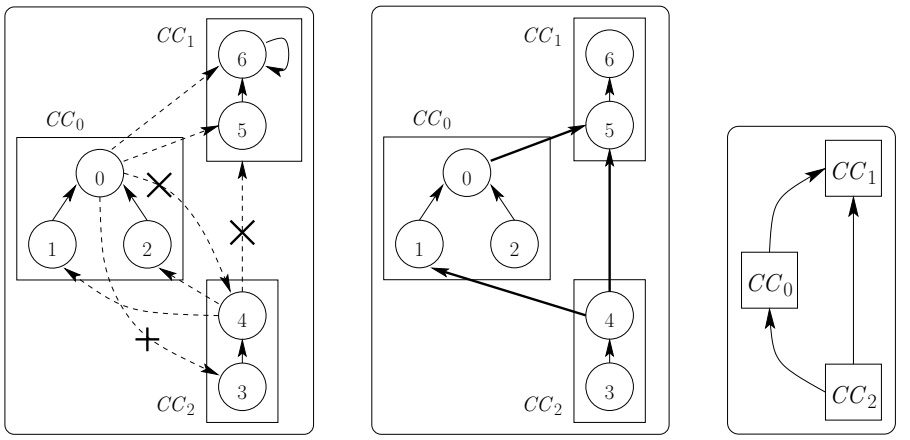
```
/* STEP 1 : Initialisation */
1. Pour chaque sommet u ∈ V faire
2.  CC(u) ← la composante connexe maximale de G_sure contenant le sommet u ;
3.  r(u) ← l'unique sommet puits de CC(u) ;
/* STEP 2 : Normalisation du graphe de précédences : remonter les précédences
            sur les racines des arbres « sûrs » */
4. Pour chaque arc (u,v) ∈ G_prec tel que u ≠ r(u) et v ∉ CC(u) faire
5.  remplacer (u,v) dans G_prec par (r(u),v) ;
/* STEP 3 : Vérifier la faisabilité */
6. Si la contrainte extended-tree n'a pas de solution (voir théorème 15) alors
7.  sortir sur un échec ;
/* STEP 4 : Mettre à jour le domaine de NTREE */
8. D(NTREE) ← D(NTREE) ∩ [MINTREE, MAXTREE_prec] ;
/* STEP 5 : Maintenir les conditions « sans circuits », « compatibles »,
            « racines compatibles » */
9. Pour chaque arc (u,v) de G_maybe faire
10.  supprimer (u,v) de G_maybe si une des conditions suivante est vérifiée :
11.    a. (CC(u), CC(v)) crée un circuit dans G^H_prec ;
12.    b. (CC(u), CC(v)) crée un arc transitif dans G^H_prec ;
13.    c. u = v et CC(u) n'est pas un puits de G^H_prec ;
/* STEP 6 : Normaliser le graphe de précédences */
14. G_prec ← TR(G_prec ∪ G_sure) ;
```

5.3.3 Algorithme de filtrage et complexité

À partir de la condition nécessaire introduite par le théorème 15 page précédente, l'algorithme 10 filtre la contrainte *extended-tree* par rapport à ses contraintes de précédence. Cet algorithme débute par une phase d'initialisation (STEP 1) qui calcule les composantes connexes de $\mathcal{G}_{sure}$ et marque l'unique sommet puits de chaque composante connexe (en effet, par définition de $\mathcal{G}_{sure}$, chacune de ses composantes connexes est un arbre). Cette phase d'initialisation est effectuée en $O(n+m)$. Ensuite, l'étape suivante (STEP 2) consiste à normaliser le graphe $\mathcal{G}_{prec}$ associé aux contraintes de précédence : pour chaque sommet u qui n'est pas une racine (c.-à-d., qui ne porte pas une boucle sur lui même) d'une composante connexe $CC(u)$ de $\mathcal{G}_{sure}$, toute contrainte de précédence (u,v) est remplacée par la précédence $(r(u),v)$ dans $\mathcal{G}_{prec}$, où $r(u)$ représente la racine de $CC(u)$ (en effet, observons que tout chemin de u à v visite $r(u)$ avant d'atteindre v). Cette étape de normalisation est effectuée en $O(m)$. Ensuite, la troisième étape (STEP 3) vérifie la faisabilité de la contrainte *extended-tree* par rapport à ses contraintes de précédence (théorème 15 page précédente) ; ce traitement coûte $O(nm)$, c.-à-d., le temps de calcul de la fermeture transitive de $\mathcal{G}_{prec}$ et de $\mathcal{G}$. La quatrième étape (STEP 4) met à jour le domaine de NTREE par rapport à la borne inférieure, MINTREE, définie dans la section 4.2.1 page 49, et par rapport à la borne supérieure, $\texttt{MAXTREE}_{prec}$, donnée dans la proposition 3 page 74 ; cette opération est effectuée en $O(n+m)$. La cinquième étape (STEP 5) vérifie la compatibilité de chaque arc (u,v) de $\mathcal{G}_{maybe}$ avec le graphe $\mathcal{G}_{prec}$. Pour cela, il faut vérifier que (u,v) ne crée ni de circuits, ni d'arcs transitifs dans $\mathcal{G}_{prec}$, et que si $u=v$ alors, il s'agit d'une racine potentielle compatible avec $\mathcal{G}_{prec}$. Cependant, il faut pouvoir ignorer les arcs de $\mathcal{G}_{prec}$ correspondant à des arcs de $\mathcal{G}_{sure}$, car il s'agit de contraintes de précédence déjà « satisfaites ». C'est pour cela, que la détection de circuits, d'arcs transitifs et de racines incompatibles est effectuée sur le graphe contracté $\mathcal{G}^H_{prec}$ (voir figure 5.5 page suivante). Il suffit de détecter les arcs transitifs et les arcs formant des circuits dans $\mathcal{G}^H_{prec}$ par un parcours en profondeur d'abord de $\mathcal{G}^H_{prec}$. Cette étape est calculée en $O(m)$. Finalement, la sixième étape (STEP 6) met à jour la forme normale de $\mathcal{G}_{prec}$ (c.-à-d., la réduction transitive) par rapport à $\mathcal{G}_{prec}$. Cette étape se ramène finalement au calcul de la fermeture transitive d'un graphe : $O(nm)$.

Lemme 14. *L'algorithme 10 ne supprime aucun arc de $\mathcal{G}_{maybe}$ ni aucune valeur du domaine de* NTREE *appartenant à une solution satisfaisant la contrainte* extended-tree.

Démonstration. Supposons qu'une valeur k de $\mathcal{D}$(NTREE) soit supprimée et qu'il existe une partition de $\mathcal{G}$ en k arbres satisfaisant la contrainte *extended-tree*. Alors, soit MINTREE n'est pas une borne inférieure



(a) Le graphe $\mathcal{G}$. (b) Le graphe $\mathcal{G}_{prec}$. (c) Le graphe $\mathcal{G}^H_{prec}$.

FIG. 5.5 – La figure 5.5(a) représente le graphe $\mathcal{G}$ associé à une contrainte *extended-tree*. Les rectangles représentent les composantes connexes de $\mathcal{G}_{sure}$. Les arcs barrés par des croix sont supprimés par l'étape (STEP 5) de l'algorithme 10 page précédente. La figure 5.5(b) représente le graphe de précédences associé à la contrainte *extended-tree*. Les contraintes de précédence induites par les arcs sûrs de $\mathcal{G}_{sure}$ sont contenues dans les rectangles représentant les composantes connexes de $\mathcal{G}_{sure}$. La figure 5.5(c) représente le graphe de précédences contracté de $\mathcal{G}_{prec}$.

pour `NTREE`, soit $\texttt{MAXTREE}_{prec}$ n'est pas une borne supérieure pour `NTREE`. La condition 1 du théorème 15 page 75 rend cette conclusion impossible : c'est une contradiction.

Supposons qu'il existe un arc (u, v) de $\mathcal{G}_{maybe}$ supprimé mais qu'il existe une partition de $\mathcal{G}$ satisfaisant la contrainte *extended-tree* et contenant un tel arc. Alors, seule l'étape 5 de l'algorithme 10 page ci-contre a pu conduire vers une telle situation. Les conditions a, b et c de cette étape respectent les conditions 2, 3 et 4 du théorème 15 page 75 ; toutefois, les conditions a, b et c se restreignent au graphe $\mathcal{G}^H_{prec}$, ce qui permet d'ignorer les composantes connexes de $\mathcal{G}_{sure}$, c.-à-d. les morceaux d'arbres déjà construits. Ainsi les conditions 2, 3 et 4 du théorème 15 page 75 rendent cette conclusion impossible : c'est une contradiction. □

Théorème 16. *L'algorithme 10 page précédente filtre une contrainte* extended-tree *avec un complexité au pire cas de* $O(nm)$.

Démonstration. Voir discussion précédant le lemme 14 page ci-contre. □

5.4 Relation de précédence conditionnelle

Beaucoup d'applications pratiques n'imposent pas de contraintes de précédence absolues entre les sommets du graphe à couvrir. Cependant, il est bien souvent nécessaire de formuler seulement des préférences. Par exemple, dans le contexte d'un problème de tournées de véhicules, on veut parfois pouvoir spécifier que, si un camion A atteint un dépôt D, alors il doit ensuite impérativement livrer les clients $C_1, \cdots, C_k$. Plus formellement, ces préférences spécifient qu'il doit exister un chemin d'un sommet u à un sommet v dans toute partition valide si et seulement si le sommet u appartient à un arbre contenant au moins deux sommets.

Dans cette section, nous étudierons tout d'abord un ensemble de règles permettant de détecter au plus tôt l'appartenance de deux sommets mis en jeu dans une contrainte de précédence conditionnelle à un même arbre de la partition (section 5.4.1 page suivante). Puis, nous étudierons la complexité des algorithmes effectuant ce traitement (section 5.4.2 page suivante).

Définition 41 (Graphe de précédence conditionnelle)**.** *Le* graphe de précédence conditionnelle $\mathcal{G}_{cond}$ *d'une contrainte* extended-tree *associée à un graphe* $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ *est défini par* $(\mathcal{V}, \{(u, v) \in \mathcal{V}^2 \mid v \in \texttt{VER}[u].\texttt{C}\})$.

Algorithme 11 Filtrer la contrainte *extended-tree* par rapport aux contraintes de précédences conditionnelles.

```
1. Pour chaque sommet u de G_cond faire
2.  Si u n'est ni une racine potentielle, ni un sommet puit de G_prec alors
3.    Pour chaque successeur v de u dans G_cond faire
4.      Ajouter (u, v) dans G_prec ;
5.      Supprimer (u, v) de G_cond ;
```

5.4.1 Filtrer une contrainte *extended-tree* par rapport aux précédences conditionnelles

Intuitivement, ce type de contrainte conditionnelle peut être traité directement à l'aide des contraintes de précédence que nous avons introduites dans la section 5.3 page 74. En effet, si l'on considère une contrainte de précédence conditionnelle entre les sommets u et v du graphe $\mathcal{G}$ associé à la contrainte *extended-tree*, cette contrainte de précédence conditionnelle deviendra sémantiquement équivalente à une contrainte de précédence classique lorsque l'on sera assuré que u va appartenir à un arbre propre de la partition. Finalement, le traitement de ces contraintes consiste à détecter au plus tôt le fait que deux sommets u, v mis en jeu dans une précédence conditionnelle appartiennent à un même arbre dans toute solution valide. Dès lors, il suffit d'ajouter l'arc (u, v) dans le graphe de précédence $\mathcal{G}_{prec}$ pour effectuer le filtrage relatif à cette contrainte de précédence conditionnelle.

Proposition 5. *Étant donné une contrainte* extended-tree *et les graphes $\mathcal{G}$, $\mathcal{G}_{prec}$ et $\mathcal{G}_{cond}$, une contrainte de précédence conditionnelle modélisée par l'arc (u, v) de $\mathcal{G}_{cond}$ est sémantiquement équivalente à une contrainte de précédence entre les sommets u et v dans $\mathcal{G}_{prec}$ si les conditions suivantes sont vérifiées :*

1. *le sommet u n'est pas un puits de $\mathcal{G}_{prec}$;*
2. *le sommet u n'est pas une racine potentielle.*

Démonstration. Par définition des contraintes de précédence conditionnelle, pour tout arc (u, v) de $\mathcal{G}_{cond}$, si le sommet u appartient à un arbre de taille au moins 2 alors on sait que u doit précéder le sommet v dans toute partition satisfaisant la contrainte *extended-tree*. Cette implication peut être transformée sans perte de généralité en : pour tout arc (u, v) de $\mathcal{G}_{cond}$, si le sommet u appartient à un arbre de taille au moins 2 alors (u, v) est un arc de $\mathcal{G}_{prec}$. En effet par définition des contraintes de précédence, dire que (u, v) est un arc de $\mathcal{G}_{prec}$ est équivalent à dire que u doit précéder le sommet v dans toute partition satisfaisant la contrainte *extended-tree*.

Ainsi, la première condition assure que le sommet u possède au moins un successeur dans toute partition satisfaisant la contrainte. Alors, on sait que u appartient à un arbre de taille au moins 2 dans toute partition valide. Finalement, (u, v) est une contrainte de précédence.

De manière similaire, la seconde condition assure que le sommet u ne peut pas être la racine d'un arbre donc, u a au moins un successeur dans toute partition satisfaisant la contrainte *extended-tree* et par suite, u appartient à un arbre de taille au moins 2 dans toute partition valide. Donc, (u, v) est une contrainte de précédence. □

5.4.2 Algorithmique et complexité

L'algorithme 11 vérifie pour chaque sommet u de $\mathcal{V}$ qu'il n'est ni une racine potentielle dans le graphe $\mathcal{G}$, ni un puits de $\mathcal{G}_{prec}$. Si les deux conditions précédentes sont vérifiées alors l'algorithme ajoute toutes les contraintes de précédences conditionnelles, du type (u, v), $v \in \texttt{VER}[u].\texttt{C}$, dans le graphe de précédences $\mathcal{G}_{prec}$ puis les supprime de $\mathcal{G}_{cond}$.

Cette opération nécessite un parcours de chaque sommet de $\mathcal{V}$, la décision sur le statut de puits et/ou racine potentielle pour chaque sommet peut être effectuée en temps constant, la complexité au pire cas est $O(n)$. Pour chaque sommet vérifiant les conditions, la migration des contraintes de précédence conditionnelle l'impliquant vers le graphe de précédences est directement fonction du nombre de précédences conditionnelles, au pire cas, on pourra retenir une complexité en $O(m)$.

Algorithme 12 Filtrer la contrainte *extended-tree* par rapport aux contraintes d'incomparabilités.

1. **Si** `la contrainte` *`extended-tree`* `n'a pas de solution (voir théorème 17)` **alors**
2. `sortir sur un échec ;`
3. **Pour chaque** `arc` $e \in \mathcal{G}_{maybe} \cap \mathcal{G}_{inc}$ **faire** `supprimer` e `de` $\mathcal{G}_{maybe}$ `;`

5.5 Relation d'incomparabilité entre les sommets du graphe

Dans cette section, nous nous intéressons aux contraintes d'incomparabilité. Rappelons que ces contraintes sont décrites par l'attribut `I` de la collection de sommets `VER`. Les contraintes d'incomparabilité sont modélisées via un graphe d'incomparabilité non-orienté. Finalement, la contrainte est étudiée au travers de conditions nécessaires caractérisant l'existence de solutions, ainsi qu'au travers de règles de filtrage dérivées de ces conditions.

Définition 42 (Graphe d'incomparabilités)**.** *Le* graphe d'incomparabilités $\mathcal{G}_{inc}$ *d'une contrainte* extended-tree *associée à un graphe* $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ *est défini par* $(\mathcal{V}, \{(u, v) \in \mathcal{V}^2 \mid u \in \texttt{VER}[v].\texttt{I} \vee v \in \texttt{VER}[u].\texttt{I}\})$.

5.5.1 Filtrer la contrainte *extended-tree* par rapport aux incomparabilités

Nous exhibons maintenant une condition nécessaire quant à l'existence de solutions pour la contrainte *extended-tree* par rapport aux contraintes d'incomparabilité. Cette condition nécessaire est constituée de deux parties, chacune maintenant une propriété de la partition recherchée : premièrement, nous garantissons que la partition ne viole aucune contrainte d'incomparabilité mise en jeu dans le graphe $\mathcal{G}_{inc}$; deuxièmement, nous maintenons l'accessibilité d'une racine potentielle pour chaque sommet non racine potentielle du graphe $\mathcal{G}$.

Dans la suite, pour un sommet u de $\mathcal{G}$, nous notons $inc(u)$ l'ensemble des sommets de $\mathcal{G}$ qui sont incomparables avec u. En d'autre termes, $inc(u)$ représente le voisinage de u dans $\mathcal{G}_{inc}$.

Théorème 17. *S'il existe une solution satisfaisant la contrainte* extended-tree *alors, les conditions suivantes sont vérifiées :*

1. ***compatibilité :*** $TC(\mathcal{G}_{sure}) \cap \mathcal{G}_{inc}$ *est un graphe sans arcs ;*
2. ***accessibilité :*** *pour chaque sommet u de $\mathcal{G}$, il existe au moins un chemin, atteignant une racine potentielle de $\mathcal{G}$, ne contenant aucun sommet du voisinage de u dans $\mathcal{G}_{inc}$.*

Démonstration. Nous fournissons une preuve pour chaque condition :

1. supposons qu'il existe un chemin d'un sommet u à un sommet v dans $\mathcal{G}_{sure}$. Si u et v sont incomparables alors, la contrainte d'arbre ne peut être satisfaite ;
2. supposons qu'il existe un sommet u de $\mathcal{G}$ tel que pour tout sommet r racine potentielle et tout chemin P de u à r, il existe un sommet v dans P tel que u et v soient incomparables. Alors, la contrainte *extended-tree* ne peut pas être satisfaite puisque u ne peut atteindre aucune racine potentielle sans violer au moins une contrainte d'incomparabilité.

Chaque condition du théorème étant nécessaire alors, leur conjonction est également une condition nécessaire. □

5.5.2 Algorithme de filtrage et complexité

Tout d'abord, l'algorithme 12 vérifie (lignes 1 et 2) la condition nécessaire relative aux contraintes d'incomparabilité (théorème 17) ; le calcul de la fermeture transitive de $\mathcal{G}_{sure}$ impose une complexité au pire cas de l'ordre de $O(n^2)$. En effet, le graphe $\mathcal{G}_{sure}$ est une forêt partitionnant le graphe $\mathcal{G}$, donc le nombre d'arcs dans $\mathcal{G}_{sure}$ est exactement de l'ordre du nombre de sommets dans $\mathcal{G}_{sure}$ (dans un arbre chaque sommet a exactement un père). Ensuite, la ligne 3 détecte et supprime les arcs de $\mathcal{G}_{maybe}$ violant une contrainte d'incomparabilité ; cette détection nécessite le calcul de l'intersection des graphes $\mathcal{G}_{maybe}$ et $\mathcal{G}_{inc}$, c.-à-d. au pire cas $O(|\mathcal{E}_{maybe}| + |\mathcal{E}_{inc}|) \approx O(m)$.

Lemme 15. *L'algorithme 12 page précédente ne supprime aucun arc de $\mathcal{G}_{maybe}$ appartenant à une solution satisfaisant la contrainte* extended-tree.

Démonstration. Supposons qu'il existe une solution valide S pour *extended-tree* telle que deux sommets u et v du graphe $\mathcal{G}$ soient incomparables et que l'arc (u, v) appartienne à S alors, l'incomparabilité entre les sommets u et v est violée dans S : c'est une contradiction avec S est une solution valide pour *extended-tree*. □

Théorème 18. *L'algorithme 12 page précédente filtre une contrainte* extended-tree *avec une complexité temporelle de $O(n^2)$.*

Démonstration. Basée sur le calcul de la fermeture transitive de $\mathcal{G}_{sure}$. Voir discussion précédent le lemme 15. □

5.6 Interactions entre précédences et incomparabilités

La combinaison de différentes restrictions portant sur la contrainte *extended-tree* constitue un problème difficile. Nous avons donc étudié deux aspects : l'un centré sur l'étude directe de règles de filtrages permettant de prendre en compte l'interaction entre différentes restrictions, et l'autre centré sur la déduction de nouvelles restrictions à partir de celles déjà existantes et de l'état du graphe $\mathcal{G}$ associé à la contrainte *extended-tree*. Ce dernier aspect est relativement original car il peut être vu comme une règle de filtrage indirecte.

5.6.1 Amélioration du filtrage via les interactions

Nous nous intéressons maintenant à l'interaction entre les contraintes de précédence et d'incomparabilité. De cette interaction, deux conditions nécessaires sont extraites ainsi que l'algorithme de filtrage dérivé.

Nous introduisons une condition nécessaire quant à l'existence de solutions satisfaisant la contrainte *extended-tree* associée à une combinaison de contraintes de précédence et d'incomparabilité. Cette condition est composée de deux conjonctions : d'une part, nous devons assurer que la partition satisfait toutes les contraintes de précédences (modélisées par $\mathcal{G}_{prec}$) sans violer de contraintes d'incomparabilités (modélisées par $\mathcal{G}_{inc}$) ; d'autre part, nous devons toujours garantir pour chaque sommet du graphe l'accessibilité à une racine potentielle.

Théorème 19. *S'il existe une solution satisfaisant une contrainte* extended-tree *mettant en jeu une combinaison de contraintes de précédence et d'incomparabilité alors, les conditions suivantes sont vérifiées :*

1. ***compatibilité :*** *$TC(\mathcal{G}_{prec}) \cap \mathcal{G}_{inc} = \emptyset$;*
2. ***accessibilité :*** *pour chaque arête (u, v) de $\mathcal{G}_{inc}$, il n'existe aucun sommet w tel que les arcs (w, u) et (w, v) soient simultanément présents dans $TC(\mathcal{G}_{prec})$.*

Démonstration. Une preuve est détaillée pour chaque condition :

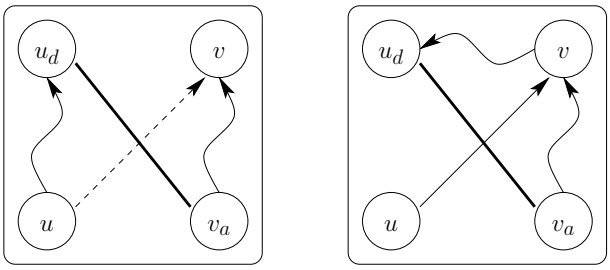
1. compatibilité : c'est une condition directement dérivée de la condition 1 du théorème 17 page précédente. En effet, toute contrainte d'incomparabilité entre deux sommets de $\mathcal{G}_{prec}$ qui appartient à un même chemin de $\mathcal{G}_{prec}$ conduit à une contradiction avec la contrainte *extended-tree*.
2. accessibilité : supposons qu'il existe un sommet w tel que les arcs (w, u) et (w, v) appartiennent à $TC(\mathcal{G}_{prec})$, supposons de plus que $(u, v) \in \mathcal{G}_{inc}$, alors les sommets u et v appartiennent à un même chemin dans toute solution : c'est une contradiction avec l'hypothèse $(u, v) \in \mathcal{G}_{inc}$.

Chaque condition étant nécessaire, leur conjonction est également une condition nécessaire. □

L'algorithme 13 page suivante vérifie tout d'abord la faisabilité de la contrainte *extended-tree* associée à une combinaison de contraintes de précédences et d'incomparabilités (théorème 19) ; en acceptant que le calcul de l'union et l'intersection de deux graphes coûte au pire cas $O(m)$, le coût global de cette étape peut se ramener au calcul de la fermeture transitive de $\mathcal{G}_{prec}$, c.-à-d., $O(nm)$. Ensuite, les lignes 3 à 6 ont une complexité globale au pire cas de $O(m^2)$. En effet, pour chaque arc (u, v) de $\mathcal{G}_{maybe}$, la condition

Algorithme 13 Filtrer une contrainte *extended-tree* combinant des contraintes de précédences et d'incomparabilités.

1. **Si** la contrainte *extended-tree* n'a pas de solution (voir théorème 19 page ci-contre) **alors**
2. sortir sur un échec ;
3. **Pour chaque** arc $(u, v) \in \mathcal{G}_{maybe}$ **faire**
4. supprimer (u, v) de $\mathcal{G}_{maybe}$ si une des conditions suivante est vérifiée :
5. a. $TC(\mathcal{G}_{prec} \cup \{(u, v)\}) \cap \mathcal{G}_{inc} \neq \emptyset$;
6. b. $\exists (u, u_d), (v_a, v) \in TC(\mathcal{G}_{prec}) : (u_d, v_a) \in \mathcal{G}_{inc}$;



(a) Interaction entre incomparabilités et précédences.

(b) Cause de l'incompatibilité.

FIG. 5.6 – Dans la figure 5.6(a) l'arc en trait épais représente une contrainte d'incomparabilité entre les sommets u_d et v_a, les arcs en traits plein incurvés représentent des chemins dans le graphe de précédences et l'arc (u, v) en pointillé représente un arc du graphe $\mathcal{G}_{maybe}$. La figure 5.6(b) représente la contradiction engendrée par l'ajout de l'arc (u, v) dans $\mathcal{G}_{sure}$. Il existe alors un chemin de v_a vers u_d dans $\mathcal{G}_{prec}$.

(a) permet de supprimer (u, v) s'il n'y pas de compatibilité de (u, v) avec les graphes de précédences et d'incomparabilités, et, la condition (b) détecte les arcs de $\mathcal{G}_{maybe}$ violant la condition d'accessibilité introduite par le théorème 19 page précédente.

Lemme 16. *L'algorithme 13 ne supprime aucun arc de $\mathcal{G}_{maybe}$ appartenant à une solution satisfaisant la contrainte* extended-tree*.*

Démonstration. Le cas (a) de l'algorithme 13 (ligne 5) est directement dérivé de la condition 1 du théorème 19 page ci-contre. En effet, supposons qu'un arc (u, v) de $\mathcal{G}_{maybe}$ tel que $TC(\mathcal{G}_{prec} \cup \{(u, v)\}) \cap \mathcal{G}_{inc} = \emptyset$ soit supprimé mais qu'il existe cependant une solution contenant (u, v) satisfaisant la contrainte. Alors, il existe une contradiction avec l'hypothèse $TC(\mathcal{G}_{prec} \cup \{(u, v)\}) \cap \mathcal{G}_{inc} = \emptyset$, puisque cette dernière assure que (u, v) viole au moins une contrainte d'incomparabilité.

Le cas (b) de l'algorithme 13 (ligne 6) est intuitivement lié à la condition 2 du théorème 19 page ci-contre. En effet, si un tel arc (u, v) est ajouté dans $\mathcal{G}_{sure}$ alors, le sommet u doit précéder à la fois les sommets u_d et v, et donc il existe dans toute solution satisfaisant la contrainte un chemin depuis u, atteignant d'abord le sommet v (par hypothèse $(u, v) \in \mathcal{E}_{sure}$) et ensuite le sommet u_d (voir figure 5.6). De plus, nous savons que le sommet v_a précède v alors, par transitivité, il existe un chemin depuis v_a jusqu'à u_d dans toute solution satisfaisante pour la contrainte : c'est une contradiction avec l'hypothèse que les sommets u_d et v_a) sont incomparables. □

Théorème 20. *L'algorithme 13 filtre une contrainte* extended-tree *avec une complexité au pire cas de $O(m^2)$.*

Démonstration. Voir discussion précédant le lemme 16. □

5.6.2 Déduction de nouvelles contraintes de précédence

Cette section montre comment certaines propriétés de graphes mises en jeu dans les graphes $\mathcal{G}$, $\mathcal{G}_{prec}$, et $\mathcal{G}_{inc}$ associés à la contrainte *extended-tree* permettent de déduire de nouvelles contraintes (ici des

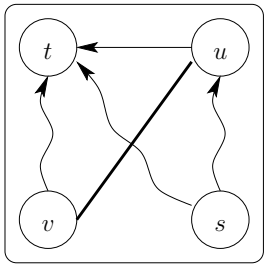


FIG. 5.7 – L'arc de précédence (u, t) (représenté par l'arc en trait plein) peut être ajouté dans $\mathcal{G}_{prec}$. L'arête en trait épais représente une contrainte d'incomparabilité entre les sommets u et v. Les arcs pleins incurvés désignent des chemins de $\mathcal{G}_{prec}$.

Algorithme 14 Déduction de contraintes de précédences.

```
/* Déduction de précédences depuis G */
1. calculer les sommets dominants dans le graphe G ;
2. Pour chaque dominant d de G faire
3.  Pour chaque (u,v) ∈ E_prec tel que d ∈ DOM_⟨G,u⟩(v) faire
4.    ajouter les arcs (u,d) et (d,v) dans G_prec ;
5.    supprimer l'arc (u,v) de G_prec ;
/* Déduction de précédences depuis G_inc et G_prec */
6. Pour chaque sommet s avec au moins deux successeurs dans TC(G_prec) faire
7.  Pour chaque successeur u de s tel que ∃v : (u,v) ∈ G_inc faire
8.    Si il existe un sommet t ∈ G_prec tel que les arcs
      (v,t) et (s,t) soient dans TC(G_prec) alors ajouter l'arc (u,t) à G_prec ;
```

contraintes de précédence) issues de l'interaction forte existante entre ces trois graphes.

Déduction de précédences à partir du graphe $\mathcal{G}$

Nous montrons maintenant comment la relation de dominance entre les sommets du graphe $\mathcal{G}$ associé à la contrainte *extended-tree* permet de mettre en lumière de nouvelles contraintes de précédence.

Étant donné le graphe $\mathcal{G}$ associé à la contrainte *extended-tree*, soit S une composante fortement connexe de $\mathcal{G}$, et soit u et v deux sommets de S tels que (u, v) soit un arc de $\mathcal{G}_{prec}$. Pour chaque dominant d de v par rapport u ($DOM_{\langle S,u\rangle}(v)$), les arcs (u, d) et (d, v) sont des contraintes de précédence valides puisque tout chemin de u vers v dans $\mathcal{G}$ atteint d'abord le sommet d avant d'atteindre le sommet v. Alors, ajouter (u, d) et (d, v) dans $\mathcal{G}_{prec}$ rend l'arc (u, v) transitif, qui peut donc être supprimé de $\mathcal{G}_{prec}$. Les lignes 2 à 5 de l'algorithme 14 détectent un tel patron dans $\mathcal{G}$ en $O(nm)$, alors que les sommets dominants sont calculés en $O(n^2)$ [CHK01]. De manière pratique, les sommets dominants de $\mathcal{G}$ sont bien-entendu calculés dynamiquement par rapport aux différentes racines potentielles.

Déduction de précédences à partir des graphes $\mathcal{G}_{prec}$ et $\mathcal{G}_{inc}$

L'interaction entre les contraintes de précédence (via le graphe $\mathcal{G}_{prec}$) et les contraintes d'incomparabilité (via $\mathcal{G}_{inc}$) permet également de révéler de nouvelles contraintes de précédence.

Étant donné quatre sommets distincts u, v, s et t, supposons qu'il existe une incomparabilité (u, v) dans $\mathcal{G}_{inc}$ et que les arcs $\{(v, t), (s, u), (s, t)\}$ appartiennent à la fermeture transitive $TC(\mathcal{G}_{prec})$. Alors, l'arc (u, t) peut être ajouté dans $\mathcal{G}_{prec}$. Dans la figure 5.7, les sommets t et u ne peuvent pas être incomparables car ils sont tous deux des descendants du sommet s. De plus, le sommet u ne peut pas être atteint depuis s après avoir atteint t, sinon les sommets u et v appartiendraient à un même chemin et l'incomparabilité entre eux ne serait plus respectée. Ainsi, le seul cheminement admissible, ordonnant t et u, consiste à ajouter une contrainte de précédence de u vers t. Les lignes 6 à 8 de l'algorithme 14 détectent un tel patron dans $TC(\mathcal{G}_{prec})$ avec une complexité temporelle de $O(mn)$.

5.7 Contraindre le demi-degré intérieur de chaque sommet

La contrainte de degré associée à chaque sommet du graphe $\mathcal{G}$ permet de spécifier le demi-degré intérieur de chacun d'eux, c.-à-d., le nombre de prédécesseurs de chaque sommet dans une partition. Ainsi, prendre en compte le demi-degré intérieur de chaque sommet du graphe $\mathcal{G}$ associé à la contrainte *extended-tree* permet de modéliser des problèmes de partitionnement par des chemins, ou des arbres binaires, sans avoir besoin de créer une contrainte globale spécifique pour chaque patron. Cette section montre comment représenter une telle contrainte à partir d'une modélisation basée sur la contrainte globale de cardinalité *gcc* [Rég96].

La contrainte globale de cardinalité [Rég96] $gcc(v_1, \ldots, v_n, c_{x_1}, \ldots, c_{x_m})$ est décrite par n variables d'assignation $v_1, \ldots, v_n$ et m variables d'occurrence $c_{x_1}, \ldots, c_{x_m}$. À chaque variable d'assignation v_j est associé un domaine $D_j \subseteq \{c_{x_1}, \ldots, c_{x_m}\}$, et à chaque variable d'occurrence c_{x_i} est associé un intervalle $E_i = [L_i, U_i]$. Une solution satisfaisant la contrainte de cardinalité est une assignation des valeurs aux variables telle que pour tout $j \in [1, n]$, la valeur assignée à v_j soit dans D_j et pour tout $i \in [1, m]$, le nombre de variables assignées à la valeur x_i est au moins égal à L_i et au plus égal à U_i. Pour un état de l'art complet sur la contrainte *gcc* voir [KT05].

Dans la suite, nous étudions la contrainte sur le demi-degré intérieur de chaque sommet du graphe $\mathcal{G}$ associé à la contrainte *extended-tree* au travers de l'approche utilisant un modèle à base de flot. D'autre part, nous nous placerons dans le contexte de la contrainte *gcc* originelle atteignant l'arc-consistance généralisée [Rég96]. Nous montrons comment il est possible de prendre en compte dans le modèle flot la gestion des racines (potentielles ou fixées) présentent dans le graphe $\mathcal{G}$ associé à la contrainte *extended-tree*.

Dans le contexte de la contrainte *extended-tree*, à chaque sommet v_i du graphe orienté $\mathcal{G}$ est associée une variable entière VER[i].D, dont le domaine $[d^i_{\min}, d^i_{\max}]$ représente le nombre de prédécesseurs possibles pour v_i dans une solution. Soulignons que les boucles sur les racines potentielles sont ignorées. Cette contrainte de degré est très similaire à la contrainte *gcc*, excepté pour le traitement des racines potentielles : une valeur i affectée à v_i est simplement ignorée, c.-à-d., qu'elle n'est pas comptée dans le nombre d'occurrences de la valeur i. La figure 5.8 montre le graphe de flot associé à cette contrainte *gcc* modifiée. Observons la présence d'un sommet ℓ supplémentaire correspondant aux racines potentielles du graphe. Formellement, ce graphe de flot est défini de la manière suivante :

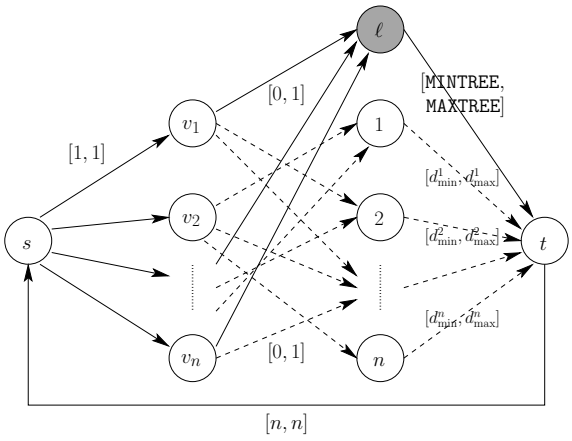


FIG. 5.8 – Graphe de flot associé à $\mathcal{G}$. Chaque sommet $v_i \in \mathcal{V}_{left}$ représente un sommet de $\mathcal{G}$, et chaque sommet $i \in \mathcal{V}_{right}$, avec $i \neq \ell$, représente un sommet de $\mathcal{G}$. Le sommet ℓ modélise l'ensemble des racines potentielles.

Définition 43. *Le* graphe de flot*, noté* $\mathcal{G}_f = (\mathcal{V}_f, \mathcal{E}_f)$*, d'une contrainte* extended-tree(NTREE, VER) *est tel que :*

- $\mathcal{V}_f$ *est l'union des ensembles de sommets suivants :*
  - $\mathcal{V}_{left} = \{\text{VER}[i].\text{S} \mid i \in [1, n]\}$*, notés* v_i *;*
  - $\mathcal{V}_{right} = \{\text{VER}[i].\text{L} \mid i \in [1, n]\}$*, notés* i *;*
  - $\{s, t, \ell\}$ *où* s*,* t*, et* ℓ *sont respectivement appelés les sommets* source*,* puits*, et* boucle *;*

- $\mathcal{E}_f$ *est l'union des arcs valués suivants :*
  - *il existe un arc de* s *à chaque* $v_i \in \mathcal{V}_{left}$ *de capacité* $[1,1]$ *;*
  - *un arc* (t,s) *de capacité* $[n,n]$ *;*
  - *il existe un arc de* $v_i \in \mathcal{V}_{left}$ *à* $j \in \mathcal{V}_{right}$ *de capacité* $[0,1]$ *ssi* $j \in \mathcal{D}(\texttt{VER}[i].\texttt{S})$ *et* $j \neq i$ *;*
  - *il existe un arc de* $v_i \in \mathcal{V}_{left}$ *à* ℓ *de capacité* $[0,1]$ *ssi* $i \in \mathcal{D}(\texttt{VER}[i].\texttt{S})$ *;*
  - *il existe un arc de* $i \in \mathcal{V}_{right}$ *à* t *de capacité* $\texttt{VER}[i].\texttt{D} = [d^i_{\min}, d^i_{\max}]$ *;*
  - *un arc* (ℓ,t) *de capacité* $[\texttt{MINTREE}, \texttt{MAXTREE}]$.

Remarquons que pour chaque $i \in \mathcal{V}_{right}$, si $i \neq \ell$ alors $d^i_{\min}$ et $d^i_{\max}$ représentent respectivement les valeurs minimale et maximale du degré négatif du sommet v_i de $\mathcal{G}$. Si $i = \ell$ alors $[d^i_{\min}, d^i_{\max}]$ est dominé par les nombres minimum (`MINTREE`) et maximum (`MAXTREE`) d'arbres possibles pour couvrir le graphe $\mathcal{G}$. En pratique, nous utilisons l'algorithme de filtrage classique basé sur le graphe de flot décrit dans [Rég96] pour supprimer tous les arcs de $\mathcal{G}$ incompatibles avec la contrainte sur le degré de chaque sommet.

La complexité de l'algorithme de filtrage dédié à la contrainte de degré est donnée par celle de la *gcc*, à savoir dans notre cas $O(mn^2)$.

5.8 Synthèse

Le tableau 5.2 résume les complexités théoriques permettant de filtrer la contrainte *extended-tree*. Le tableau est décomposé en quatres parties horizontales. La première partie rappelle les meilleures bornes connues pour les variables `NTREE` et `NPROP`. La seconde partie résume les conditions nécessaires relatives aux contraintes additionnelles décrites dans ce chapitre. La troisième partie référence les algorithmes permettant d'effectuer le filtrage relatif à chaque contrainte additionnelle prise en compte par la contrainte *extended-tree*. Finalement, la dernière partie rappelle les algorithmes permettant de mettre à jour le graphe de précédences $\mathcal{G}_{prec}$ par rapport aux différents graphes modélisant des contraintes annexes. Pour chaque algorithme et proposition, une borne supérieure du temps de calcul est fournie, en fonction des grandeurs n et m représentant respectivement le nombre de sommets et le nombre d'arcs dans le graphe $\mathcal{G}$. Notons que ces deux grandeurs sont suffisantes pour fournir une borne supérieure de la complexité du temps de calcul. En effet, soit m_{prec} et m_{inc} respectivement le nombre d'arcs et d'arêtes dans $\mathcal{G}_{prec}$ et $\mathcal{G}_{inc}$. D'une part, on sait que $m_{prec} \leq m$ puisque $\mathcal{G}_{prec}$ est un graphe acyclique sans arcs transitifs ; d'autre part, $m_{inc} \approx m$ car il ne peut pas exister plus de contraintes d'incomparabilité que de liens possibles entre les sommets du graphe $\mathcal{G}$.

	Interaction	*Effets*	*Théorèmes, propositions, et algorithmes*	*Complexité* théorique
Bornes	$\mathcal{G}$	min(`NTREE`)	Section 4.2.1 page 49, `MINTREE`	$O(m+n)$
	$\mathcal{G}_{prec}$	max(`NTREE`)	Proposition 3 page 74	
	$\mathcal{G}$	min(`NPROP`)	Proposition 1 page 55	
	$\mathcal{G}$	max(`NPROP`)	Proposition 2 page 56	$O(m\sqrt{n})$
Faisabilité	$\mathcal{G}_{prec}$	`echec`	Théorème 15 page 75 (précédences)	$O(mn)$
	$\mathcal{G}_{inc}$		Théorème 17 page 79 (incomparabilités)	
	$\mathcal{G}_{prec}$ & $\mathcal{G}_{inc}$		Théorème 19 page 80 (préc. et inc.)	
Filtrage	$\mathcal{G}$	$\mathcal{G}$	Algorithmes 7 page 59 et 8 page 60 (`NPROP`)	$O(mn)$
	$\mathcal{G}_{prec}$		Algorithme 10 page 76 (précédences)	$O(mn)$
	$\mathcal{G}_{inc}$		Algorithme 12 page 79 (incomparabilités)	$O(n^2)$
	$\mathcal{G}_{prec}$ & $\mathcal{G}_{inc}$		Algorithme 13 page 81 (préc. et inc.)	$O(m^2)$
	$\mathcal{G}_f$		Global Cardinality [Rég96] (degrés)	$O(mn^2)$
Déductions Internes	$\mathcal{G}$ & $\mathcal{G}_{cond}$	$\mathcal{G}_{prec}$	Algorithme 11 page 78	$O(m)$
	$\mathcal{G}$ & $\mathcal{G}_{prec}$		Algorithme 14 page 82	$O(mn)$
	$\mathcal{G}_{inc}$ & $\mathcal{G}_{prec}$			

Tab. 5.2 – Résumé des complexités de chaque proposition, théorème et algorithme permettant de maintenir la contrainte *extended-tree*.

Chapitre 6

Un cas particulier d'arbres : les chemins

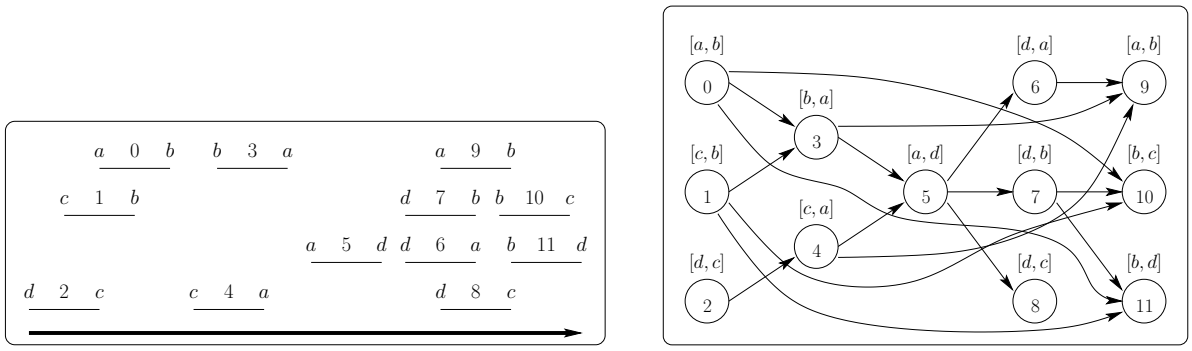
Sommaire

DAns une certaine mesure les contraintes de chemin sont reliées à plus d'un titre aux contraintes d'arbre. D'une part, on peut remarquer que tout chemin ne constitue ni plus ni moins qu'un arbre, tel que chaque sommet possède exactement deux voisins à l'exception des sommets sources et des sommets destinations. D'autre part, dans un arbre[1] il existe un chemin de tout sommet interne vers la racine. En conséquence, il serait surprenant que l'étude d'une contrainte de chemin ne recoupe pas les propriétés de graphes qui avaient déjà été utilisées dans le contexte des contraintes d'arbre. Cependant, les contraintes structurelles misent en jeu dans le contexte des chemins sont beaucoup plus fortes que dans le contexte des arbres, ce qui en justifie pleinement l'étude.

Les contraintes de partitionnement par des chemins sont omniprésentes dans de nombreuses applications pratiques comme les problèmes de *tournées de véhicules* [BQ64] ou de *robustesse dans les réseaux* [Suu74]. Dans le contexte des contraintes de partitionnement par des chemins, les conditions nécessaires prévenant la création de circuits et maintenant le fait que chaque sommet du graphe ne possède pas plus d'un prédécesseur ont déjà été introduites dans la littérature [JKK+99, Sel03, QvRDC06, BFL06]. Cependant, aucune de ces conditions ne considère réellement le nombre de chemins à construire. Le but de ce chapitre est de proposer une condition nécessaire relative au nombre de chemins requis pour partitionner un graphe orienté de manière à ce que chaque sommet appartienne à un chemin et un seul.

Considérons un graphe orienté $\mathcal{G} = (\mathcal{V}, \mathcal{A})$, une partition de $\mathcal{G}$ en chemins est une collection de chemins sommets-disjoints $P_1 = (V_1, A_1), \ldots, P_K = (V_K, A_K)$ dans $\mathcal{G}$, dont l'union est représentée par $\mathcal{V}$, c.-à-d., $V_i \cap V_j = \emptyset$, $i \neq j$, et $\bigcup_{i=1}^{K} V_i = \mathcal{V}$. Le problème des $\mathtt{K}$ *chemins sommets-disjoints* [Ste03, Vyg95] (que l'on notera K-NDP dans la suite) cherche à déterminer une partition de $\mathcal{G}$ telle que $\mathtt{K} \in [\underline{\mathtt{K}}, \overline{\mathtt{K}}]$, où $\underline{\mathtt{K}}$ désigne le nombre minimum de chemins d'une partition de $\mathcal{G}$ et $\overline{\mathtt{K}}$ est la taille du sous-ensemble $\mathcal{T} \subseteq \mathcal{V}$ de sommets potentiellement finaux pour chaque chemin.

[1] Rappelons que l'on se place dans le contexte d'anti-arborescence, ce qui implique que l'orientation se fait des feuilles vers la racine.



(a) (b)

FIG. 6.1 – En partie (a), un ensemble de tâches fixées dans le temps (décrit par l'axe orienté en gras). Chaque tâche t est décrite par un triplet s, i, d dont les éléments correspondent respectivement au *lieu de départ*, *l'identifiant de la tâche* et le *lieu de destination*. La partie (b) représente le graphe correspondant, respectant à la fois la continuité temporelle et géographique.

Trouver une partition de $\mathcal{G}$ en chemins est un problème généralement NP-complet [FHW80, Ste03], même lorsque le nombre de chemins est fixé à $\mathtt{K} = 2$. Il existe cependant de nombreuses classes de graphes particuliers pour lesquelles ce problème de partitionnement est résolvable en temps polynomial [AR90, YC94, Sch03]. Dans le contexte d'une contrainte maintenant une partition par des chemins sommets-disjoints, nous ne pouvons hélas faire aucune hypothèse sur la classe du graphe $\mathcal{G}$. Nous allons donc proposer une condition nécessaire basée sur la structure de $\mathcal{G}$. Cette condition sera directement dérivée du graphe réduit (voir définition 22 page 23) associé à $\mathcal{G}$ ainsi que de la relation de dominance 25 page 23 entre les sommets de $\mathcal{G}$. Le problème des K chemins sommets-disjoints est utile dans beaucoup d'applications pratiques. L'une d'entre elle consiste à couvrir un ensemble de tâches (par ex., des livraisons, des vols) avec un minimum de ressources (par ex., des camions, des avions). Chaque tâche est représentée comme un intervalle défini par une *date de départ au plus tôt*, une *date de fin au plus tard* et une *durée fixe*. De plus, une tâche possède aussi un *lieu de départ* et une *destination*. Dans toute solution valide, une tâche t_1 peut précéder immédiatement une tâche t_2 si (1) la date de fin au plus tôt de t_1 précède la date de début au plus tard de t_2, et si (2) la destination de t_1 est identique au lieu de départ de t_2.

Exemple 9. *Pour illustrer l'application précédente, considérons l'ensemble des tâches fixées dans le temps, représentées par la figure 6.1(a). La figure 6.1(b) fournit le graphe $\mathcal{G}$ correspondant qui peut être couvert par* 6 *chemins sommets-disjoints. Remarquons que le graphe $\mathcal{G}$ est acyclique puisque toutes les tâches sont complètement fixées dans le temps.*

Dans le contexte des contraintes de partitionnement [CB04, BFL06, QvRDC06], la contribution de ce chapitre est de montrer comment la combinaison d'un raisonnement sur les *flots* et d'un raisonnement sur la relation de *dominance* entre les sommets du graphe fournit une condition nécessaire pour la contrainte de partitionnement par des chemins. Cette contrainte a exactement les mêmes arguments que la contrainte *tree* introduit en section 4.2 page 49. Dans la suite de cette partie, la section 6.1.1 page ci-contre présente une approche basée sur les flots dans le contexte des graphes orientés sans circuits (DAG). Puis, la section 6.1.2 page 89 généralisera ce modèle au cas avec circuits, en prenant en considération les « goulots d'étranglement » du graphe réduit : les sections 6.1.2 page 90 et 6.1.2 page 92 montreront comment déterminer des bornes plus précises sur le nombre de chemins minimum et maximum à partir de (1) la relation de dominance entre les sommets du graphe $\mathcal{G}$ et (2) la manière dont les sommets de deux composantes fortement connexes sont reliées entre elles. Finalement, la section 6.2 page 93 montrera comment exploiter une partie du modèle de flot associé au problème des K chemins sommets-disjoints dans le but d'améliorer le filtrage lié à la contrainte de partitionnement par des chemins.

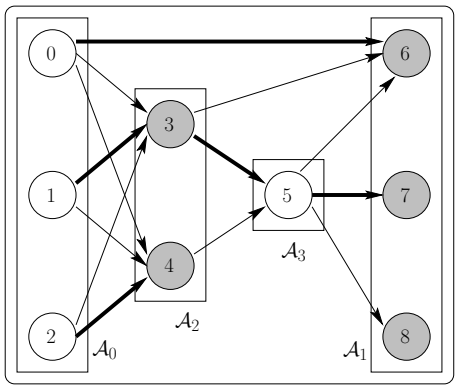


FIG. 6.2 – Un graphe orienté acyclique $\mathcal{G}$, où les sommets potentiellement finaux sont colorés en gris. Les arcs épais représentent une partition en chemins de cardinalité minimale.

6.1 Évaluation du nombre minimum de chemins nécessaires pour partitionner un graphe

6.1.1 Le problème K-NDP dans le cas de graphes orientés acycliques

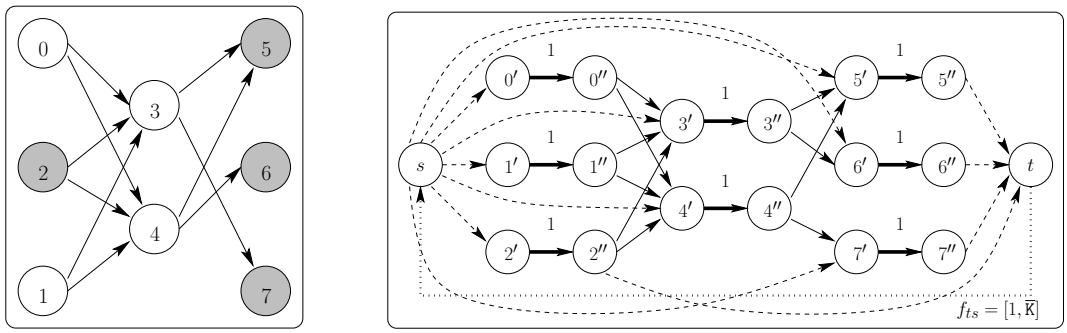
Cette section s'intéresse au cas des graphes orientés sans circuits pour lesquels le problème K-NDP peut être résolu en temps polynomial [AR90, YC94, Sch03]. Lorsque le nombre de chemins K n'est pas fixé, le point clé dans la résolution d'un tel problème concerne l'évaluation du nombre de chemins minimum nécessaires pour partitionner le graphe $\mathcal{G}$.

Une première piste pour évaluer cette borne inférieure consiste à regarder la taille d'une antichaîne maximale [2] (aussi appelée *largeur*) du graphe $\mathcal{G}$ (voir [Dil50] pour une caractérisation complète). En effet, une fois que l'antichaîne est traversée, il n'y a aucun moyen de revenir en arrière puisque $\mathcal{G}$ est sans circuit. Cependant, même si la largeur de $\mathcal{G}$ constitue une borne inférieure réalisable pour le problème de *couverture* par des chemins (voir [Sch03, p.219]), ce n'est pas le cas dans le contexte du problème de partitionnement par des chemins sommets-disjoints. L'exemple suivant illustre ce point :

Exemple 10. *La figure 6.2 présente les deux antichaînes maximales* $\mathcal{A}_0 = \{0, 1, 2\}$ *et* $\mathcal{A}_1 = \{6, 7, 8\}$ *de* $\mathcal{G}$*. La largeur de* $\mathcal{G}$ *étant égale à* 3*, il existe une couverture de* $\mathcal{G}$ *avec* 3 *chemins (par ex.,* $\langle 0, 6\rangle$*,* $\langle 1, 3, 5, 7\rangle$ *et* $\langle 2, 4, 5, 8\rangle$*). Cependant, remarquons que dans le contexte du partitionnement, les antichaînes* $\mathcal{A}_2 = \{3, 4\}$ *et* $\mathcal{A}_3 = \{5\}$ *forment un goulot d'étranglement entre les sommets de* $\mathcal{A}_0$ *et les sommets de* $\mathcal{A}_1$*. Ainsi, partitionner* $\mathcal{G}$ *avec* 3 *chemins sommets-disjoints est clairement impossible. En fait,* 4 *chemins sont nécessaires pour partitionner* $\mathcal{G}$ *en chemins sommets-disjoints (par ex.,* $\langle 0, 6\rangle$*,* $\langle 1, 3, 5, 7\rangle$*,* $\langle 2, 4\rangle$*, et* $\langle 8\rangle$*).*

Dans le cadre de graphes orientés sans circuits, cette section introduit un modèle de flot classique (voir exemple 11 page suivante) fournissant une condition nécessaire et suffisante pour le problème K-NDP, ainsi qu'une borne inférieure réalisable sur le nombre minimum de chemins sommets-disjoints nécessaires pour partitionner $\mathcal{G}$. Dans cette approche, on doit construire pour chaque K-partition en chemins du graphe, un K flot dans un graphe dérivé. Une idée simple est d'ajouter, pour chaque sommet i, deux arcs (s, i) et (i, t) (où s et t sont deux sommets supplémentaires). Alors, pour chaque chemin $\langle i_1, \ldots, i_n\rangle$ de la K partition, on peut ajouter les deux arcs (s, i_1) et (i_n, t), qui permettent de construire un K-flot. Le biais de cette construction est que le graphe résultant admet toujours un flot nul, ce qui conduit à une borne triviale inutile. Dans le but d'imposer un flot non-nul au travers de chaque sommet du graphe, nous scindons chaque sommet i en deux sommets i' et i'', et nous ajoutons l'arc (i', i'') en imposant qu'une unité de flot le traverse. Ainsi, la conservation du flot nous assure l'existence d'un flot non-nul de i' à i''.

[2] Une *antichaîne* dans un ensemble partiellement ordonné P est un sous-ensemble A de P tel que chaque paire d'éléments de A est incomparable, c.-à-d., pour tout x, y dans A, on ne peut avoir ni $x \leq y$, ni $y \leq x$ (dans notre contexte cela s'interprète comme le fait qu'il n'y a ni de chemin de x à y, ni de chemin de y à x).



(a) $\mathcal{G}$ (b) $\mathcal{N}$

FIG. 6.3 – Pour le graphe orienté $\mathcal{G}$ représenté en partie (a), la partie (b) fournie le réseau correspondant $\mathcal{N}$. Les arcs représentés par des tirets sont des arcs *extrémités de chemins*, les arcs en gras représentent les arcs associés aux *sommets dédoublés*, l'arc en pointillé représente *l'arc de retour*, et les arcs pleins sont les arcs associés au DAG.

Définition 44. *Étant donné un graphe orienté sans circuit $\mathcal{G}$, le réseau associé $\mathcal{N} = (\mathcal{X}, \mathcal{E}, c)$ est défini de la manière suivante :*

- *À chaque sommet i de $\mathcal{G}$ correspond deux sommets i' et i'' (resp. appelés sommet* entrée *et sommet* sortie*), ainsi qu'un arc (i', i'') de $\mathcal{E}$ associé aux* sommets dédoublés*. Soient $\mathcal{I}'$ et $\mathcal{I}''$ les deux ensembles de sommets correspondant. Finalement, deux sommets supplémentaires s (*source*) et t (*puits*) sont ajoutés à l'ensemble des sommets $\mathcal{X}$;*
- *À chaque arc (i, j) de $\mathcal{G}$ corresponds un arc (i'', j') de $\mathcal{E}$, où i'' est le sommet de sortie de i et j' le sommet d'entrée de j ;*
- *Il existe un arc (s, i') dans $\mathcal{E}$, pour tout sommet $i' \in \mathcal{I}'$, ainsi qu'un arc (i'', t) pour tout sommet de sortie i'' appartenant à $\mathcal{I}''$;*
- *Un arc de retour de t à s est finalement ajouté.*

Nous sommes maintenant en mesure de définir la fonction f associée au réseau $\mathcal{N}$. Dans ce but, nous distinguons différentes classes d'arcs correspondant respectivement aux *arcs de retour*, aux arcs associés aux *sommets dédoublés*, aux arcs du DAG $\mathcal{G}$, et aux arcs *extrémités de chemins*.

Définition 45. *Étant donné un graphe orienté sans circuit $\mathcal{G}$, le flot porté par chaque arc du réseau $\mathcal{N}$, associé à $\mathcal{G}$, est défini de la manière suivante :*

1. *l'arc de* retour *: $f(t, s) = f_{ts} \in [1, \overline{\mathtt{K}}]$, où $\overline{\mathtt{K}} = |\mathcal{T}|$;*
2. *les arcs associés aux* sommets dédoublés *: $f(i', i'') = f_i = 1$;*
3. *les arcs du* dag *: $f(i'', j') = f_{ij} \in [\ell_{ij}, c_{ij}]$ où $\ell_{ij} = 0$ et $c_{ij} = \overline{\mathtt{K}}$;*
4. *les arcs* extrémités de chemins *: $f(s, i) = f(i, t) = f_{si} = f_{it} \in [0, \overline{\mathtt{K}}]$, pour tout (s, i) et (i, t) de $\mathcal{E}$.*

Exemple 11. *La figure 6.3 illustre les définitions 44 et 45. Chaque sommet i du graphe $\mathcal{G}$ de la figure 6.3(a) est dédoublé en deux sommets distincts i' et i'' dans la figure 6.3(b). L'arc en pointillé de la figure 6.3(b) correspond à* l'arc de retour *de la définition 45, avec un flot $[1, \overline{\mathtt{K}}]$, les arcs en gras correspondent aux arcs associés aux* sommets dédoublés*, avec un flot unitaire, les arcs représentés par des tirets correspondent aux* extrémités de chemins*, avec un flot $[0, \overline{\mathtt{K}}]$, et les arcs plein correspondent quant à eux aux arcs associés au graphe $\mathcal{G}$, avec un flot $[0, \overline{\mathtt{K}}]$.*

Observons que chaque circuit dans $\mathcal{N}$ contient l'arc (t, s) car le graphe $\mathcal{G}$ est sans circuit. Alors, la conservation du flot assure que le flot global de $\mathcal{N}$ est fourni par :

$$\mathcal{F}(\mathcal{N}) = \sum_{(s,j) \in \omega^+(\{s\})} f_{sj} = f_{ts} \tag{6.1}$$

Dans la suite, le flot global de $\mathcal{N}$ sera assimilé au flot f_{ts} traversant l'arc de retour (t, s).

Théorème 21. *Un graphe orienté sans circuit $\mathcal{G}$ peut être partitionné en* K *chemins sommets-disjoints ssi il existe un flot compatible $f_{ts} =$ K dans le réseau $\mathcal{N}$ associé à $\mathcal{G}$.*

Démonstration. Montrons d'abord que la condition est nécessaire. Supposons que (a) il existe une partition P en chemins sommets-disjoints de $\mathcal{G}$ de taille K, et (b) qu'il n'existe pas un flot compatible dans $\mathcal{N} = (\mathcal{X}, \mathcal{E})$ avec $\mathcal{F} =$ K. Montrons qu'il existe une contradiction. Pour cela, nous allons montrer qu'il est possible de dériver, de la partition P, un flot compatible $\mathcal{F}$ pour le réseau $\mathcal{N}$. Par définition du réseau $\mathcal{N}$, pour chaque arc (u', u'') de $\mathcal{N}$, associé à un sommet dédoublé, le flot $f(u', u'')$ est fixé à 1. Le flot sur l'arc de retour f_{ts} est fixé à K. On dérive un flot compatible $\mathcal{F} =$ K dans $\mathcal{N}$, à partir de la partition P, de la manière suivante :

- Pour chaque arc (u, v) de P, le flot $f(u'', v')$ sur l'arc correspondant dans $\mathcal{N}$ est fixé à 1.
- Pour chaque sommet source u d'un chemin de P, le flot $f(s, u')$ dans $\mathcal{N}$ est fixé à 1.
- Pour chaque sommet puits v d'un chemin de P, le flot $f(v'', t)$ dans $\mathcal{N}$ est fixé à 1.
- Tous les autres arcs du réseau $\mathcal{N}$ ont un flot nul.

La construction est naturellement valide car le flot traversant l'arc de retour correspondant exactement au nombre de chemins dans la partition P (K), le flot sortant de la source s, le flot entrant sur le puits t du réseau $\mathcal{N}$ sont aussi égaux à K, et chaque arc de $\mathcal{N}$ correspondant à un arc de la partition P véhicule exactement une unité de flot. L'existence d'un partition en K dans le graphe $\mathcal{G}$ implique donc bien l'existence d'un flot compatible de taille K dans le réseau $\mathcal{N}$ dérivé de $\mathcal{G}$.

Nous prouvons maintenant que la condition est suffisante. Nous montrons qu'à partir de n'importe quel flot compatible de $\mathcal{N}$, $\mathcal{F}(\mathcal{N}) = f_{ts} =$ K, on peut construire une partition de $\mathcal{G}$ en K chemins sommets-disjoints. Dans ce but, considérons un flot compatible f_{ts} de $\mathcal{N}$ et le DAG $\mathcal{N}' = (\mathcal{X}', \mathcal{E}')$ défini à partir du flot f_{ts} de la manière suivante :

- $\mathcal{X}' = \mathcal{X} \setminus \{s, t\}$.
- $\mathcal{E}' = \{(i, j) \in \mathcal{E} \setminus (t, s) \mid f_{ij} = 1\}$.

Le flot f_{ts} assure que $\mathcal{N}'$ est un DAG composé de K composantes connexes telles que chacune corresponde à un chemin élémentaire. Cette propriété est directement dérivée du fait que pour chaque arc du DAG (i, j) de $\mathcal{N}$, soit $f_{ij} = 1$ ou $f_{ij} = 0$ (car dans le cas des graphes sans circuit chaque arc associé aux *sommets dédoublés* (i', i'') a un flot $f_i = 1$). Alors, en contractant chaque arc de $\mathcal{N}'$ associé aux *sommets dédoublés* en un unique sommet, $\mathcal{N}'$ devient un graphe partiel de $\mathcal{G}$ induit par $\mathcal{E}'$, et composé de K chemins sommets-disjoints. □

Lemme 17. *Soit un graphe orienté sans circuit $\mathcal{G}$ et son réseau $\mathcal{N}$ associé. La borne inférieure du nombre de chemins sommets-disjoints partitionnant $\mathcal{G}$ est donnée par le flot minimum $f^\star_{ts}$ de $\mathcal{N}$: $f^\star_{ts} = \underline{\mathrm{K}}$.*

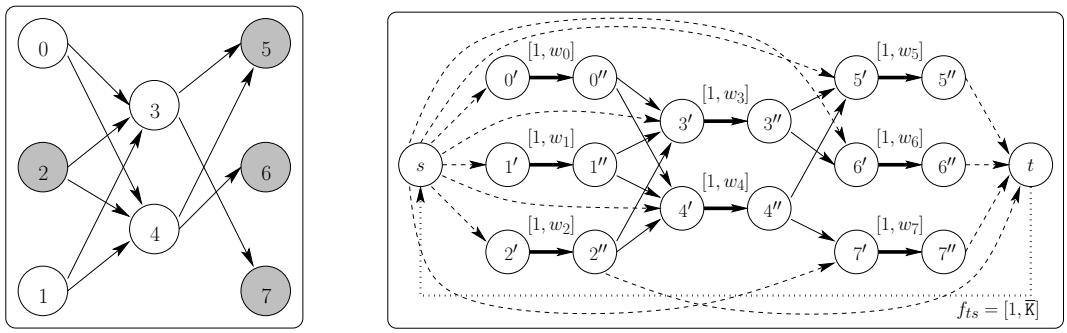
Démonstration. Directement dérivée du théorème 21. □

6.1.2 Le problème K-NDP dans le cas de graphes quelconques

Comme nous l'avons vu dans la section précédente, dans le cas où K n'est pas fixé, le point clé d'une approche résolvant le problème K-NDP repose sur l'évaluation d'une borne inférieure du nombre minimum de chemins partitionnant le graphe $\mathcal{G}$. En effet, considérons un graphe orienté $\mathcal{G}$ et un ensemble $\mathcal{T}$ de sommets potentiellement finaux. À partir d'une partition $\mathcal{P}$ quelconque de taille $p \in [1, |\mathcal{T}|]$, on peut construire une partition $\mathcal{P}'$ de taille $p' \in [p+1, |\mathcal{T}|]$ en décomposant, éventuellement, un des chemins contenant au moins deux sommets potentiellement finaux. Cependant, déterminer à partir de $\mathcal{P}$, une partition $\mathcal{P}''$ de taille $p'' \in [1, p-1]$ est un problème NP-complet (supposons que $p = 2$ alors, $p'' = 1$ et par conséquent, nous devons construire un chemin Hamiltonien).

Dans la section 6.1.1 page 87, un modèle de flot a été proposé dans le cadre de graphes orientés sans circuit. Cette section montre comment étendre aux graphes quelconques ce modèle, en considérant le graphe réduit $\mathcal{G}_r$ associé à $\mathcal{G}$. Un sommet du graphe réduit $\mathcal{G}_r$ correspond à une composante fortement connexe du graphe initial $\mathcal{G}$. Par conséquent nous devrons distinguer les arcs du DAG (c.-à-d., les arcs de $\mathcal{G}_r$) et les arcs associés aux *sommets dédoublés* (c.-à-d., les arcs représentant les *CFC* de $\mathcal{G}$), et déterminer la borne inférieure en tenant compte des deux informations. Ainsi, les définitions 44 page ci-contre et 45 page précédente sont étendues au cas des graphes contenant éventuellement un circuit de la manière suivante (figure 6.4 page suivante) :

- le réseau $\mathcal{N} = (\mathcal{X}, \mathcal{E}, c)$ est dérivé à partir du graphe réduit $\mathcal{G}_r$ associé à $\mathcal{G}$;



(a) $\mathcal{G}_r$ (b) $\mathcal{N}$

FIG. 6.4 – Le graphe orienté $\mathcal{G}_r$ représenté par les parties (a) et (b) fournit le réseau $\mathcal{N}$ correspondant. Les arcs avec des tirets représentent des arcs extrémités de chemins, les arcs en gras représentent les arcs associés aux sommets dédoublés, l'arc en pointillé (t, s) représente l'arc de retour du réseau.

– le flot lié à chaque arc associé aux sommets dédoublés (i', i'') de $\mathcal{N}$ est maintenant évalué de la manière suivante : $f(i', i'') = f_i \in [\ell_i, c_i]$, où ℓ_i (resp. c_i) correspond à une borne inférieure (resp. supérieure) sur le nombre minimum (resp. maximum) de chemins sommets-disjoints pouvant partitionner C_i. Nous avons $f_i \in [1, w_i]$ où w_i représente le nombre de sommets dans C_i qui sont soit des sommets potentiellement finaux dans $\mathcal{G}$, ou des sommets directement connectés, par un seul arc de $\mathcal{G}$, à une autre *CFC* C_j de $\mathcal{G}$.

Alors, la condition nécessaire et suffisante introduite par le théorème 21 page précédente peut être directement généralisée à une condition nécessaire dans le cas de graphes quelconques :

Théorème 22. *S'il existe une partition en chemins sommets-disjoints de $\mathcal{G}$ de taille $\mathtt{K} \in [\underline{\mathtt{K}}, \overline{\mathtt{K}}]$ alors, il existe un flot compatible $f_{ts} = \mathtt{K}$ dans le réseau $\mathcal{N}$ correspondant au graphe $\mathcal{G}$.*

De la même manière, le nombre minimum de chemins introduit par le lemme 17 page précédente est généralisé à une borne inférieure sur le nombre de chemins sommets-disjoints partitionnant le graphe $\mathcal{G}$.

Lemme 18. *Étant donné un graphe orienté $\mathcal{G}$ et son réseau correspondant $\mathcal{N}$, une borne inférieure sur le nombre de chemins sommets-disjoints partitionnant $\mathcal{G}$ est donnée par le plus petit flot compatible $f^\star_{ts}$ de $\mathcal{N}$: $f^\star_{ts} \in [\underline{\mathtt{K}}; \overline{\mathtt{K}}]$.*

Dans la suite, nous proposons deux manières d'améliorer l'évaluation du plus petit flot compatible dans le réseau $\mathcal{N}$ associé au graphe orienté $\mathcal{G}$: premièrement, dans la section 6.1.2, nous montrons comment estimer le nombre de chemins partitionnant chaque *CFC* de $\mathcal{G}$ dans le but d'améliorer l'évaluation du flot traversant chaque arc de $\mathcal{N}$ associé aux sommets dédoublés. Deuxièmement, dans la section 6.1.2 page 92, nous montrons comment estimer le nombre de chemins entre deux *CFC* dans le but de raffiner l'évaluation du flot traversant les arcs du DAG.

Estimer le nombre de chemins partitionnant une *CFC*

Une première voie pour améliorer la précision de la relaxation du problème K-NDP, lorsque des *CFC* de $\mathcal{G} = (\mathcal{V}, \mathcal{A})$ contiennent plus d'un sommet (c.-à-d., $\mathcal{G}$ n'est pas un DAG), est de raffiner les bornes sur le flot $f_i \in [\ell_i, c_i]$ pour chaque arc associé aux sommets dédoublés représentant la *CFC* C_i de $\mathcal{G}$. Cette section montre comment améliorer l'évaluation de ℓ_i (c.-à-d., du nombre minimum de chemins partitionnant C_i) qui était à l'origine fixée à la valeur 1 dans la définition 45 page 88. L'idée est d'identifier un sommet d de C_i, dont le retrait augmente le nombre de *CFC* de $C_i \setminus \{d\}$ (i.e., le graphe correspondant aux *CFC* de C_i auquel le sommet d a été supprimé), dans le but de ré-appliquer le Lemme 18 sur le nouveau DAG construit à partir du graphe réduit associé à $C_i \setminus \{d\}$. Ceci est réalisé en utilisant la relation de dominance entre les sommets de C_i, définie de la manière suivante :

Définition 46 ([LT79])**.** *Étant donné un graphe orienté G et deux sommets i, j de G tel qu'il existe au moins un chemin de i à j, un sommet d est un* dominant *du sommet j par rapport à un sommet i ssi il*

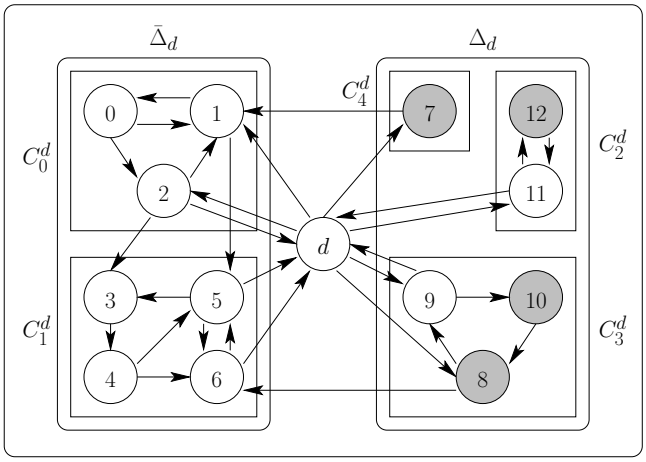


FIG. 6.5 – Cette figure représente pour une *CFC* C_i de $\mathcal{G}$, un de ces sommets dominant d de $D_i = \{1, 4, 8, 9, 11, d\}$ par rapport aux sommets $\mathcal{T}_i$ représentés en gris. Les rectangles contenus dans Δ_d et $\bar{\Delta}_d$ représentent les *CFC* créées par le retrait du sommet d de la composante fortement connexe C_i.

n'existe aucun chemin de i à j dans $G \setminus \{d\}$. L'ensemble des dominants de j par rapport à i est noté par $DOM_{\langle \mathcal{G}, i \rangle}(j)$.

Notation 7. *Étant donné un graphe orienté $\mathcal{G} = (\mathcal{V}, \mathcal{A})$, soit C_i une CFC de $\mathcal{G}$:*
- *$\mathcal{C}_i = (C_i, \mathcal{A}_i)$ représente le sous-graphe de $\mathcal{G}$ correspondant à C_i ;*
- *$\mathcal{T}_i$ représente le sous-ensemble des sommets de $\mathcal{V}_i$ qui sont, soit des sommets potentiellement finaux de $\mathcal{G}$, soit des sommets atteignant, en un seul arc, une autre CFC C_j de $\mathcal{G}$;*
- *soit D_i l'ensemble des sommets dominants de C_i définis par rapport à l'ensemble $\mathcal{T}_i$:*

$$D_i = \{d \mid \exists v \in \mathcal{V}_i, \forall w \in \mathcal{T}_i, d \in DOM_{\langle \mathcal{C}_i, v \rangle}(w)\} \tag{6.2}$$

La figure 6.5 illustre le fait que le retrait du sommet dominant $d \in D_i$ crée deux sortes de *CFC* (la figure 6.5 illustre ce point) :
- Δ_d est l'ensemble, possiblement vide, des nouvelles *CFC* à partir des quelles un sommet de $\mathcal{T}_i$ est atteint par au moins un chemin ne contenant pas d. Soit $\Delta_d^\star$ l'ensemble des sommets contenus dans les *CFC* de Δ_d.
- $\bar{\Delta}_d$ est l'ensemble, possiblement vide, des nouvelles *CFC* à partir desquelles aucun sommet de $\mathcal{T}_i$ ne peut être atteint par un chemin ne contenant pas d. Soit $\bar{\Delta}_d^\star$ l'ensemble des sommets contenus dans les *CFC* de $\bar{\Delta}_d$.

Exemple 12. *Dans la CFC C_i de $\mathcal{G}$ représentée par la figure 6.5, nous avons $\mathcal{T}_i = \{7, 8, 10, 12\}$. L'ensemble des sommets dominants de C_i par rapport à $\mathcal{T}_i$ est $D_i = \{1, 4, 8, 9, 11, d\}$. Par exemple, tout chemin du sommet 3 à un sommet de $\mathcal{T}_i$ rencontre les sommets 4 et d. Ainsi, les sommets 4 et d sont des dominants du sommet 3 par rapport aux sommets de $\mathcal{T}_i$. La figure 6.5 illustre aussi le partitionnement de la CFC C_i dans les composantes fortement connexes, produites par le retrait du sommet d :*
- *$\Delta_d = \{\mathcal{C}_2^d, \mathcal{C}_3^d, \mathcal{C}_4^d\}$ et $\Delta_d^\star = \{7, 8, 9, 10, 11, 12\}$,*
- *$\bar{\Delta}_d = \{\mathcal{C}_0^d, \mathcal{C}_1^d\}$ et $\bar{\Delta}_d^\star = \{0, 1, 2, 3, 4, 5, 6\}$.*

Nous introduisons maintenant une proposition qui va nous permettre de réduire le problème d'estimation du nombre minimum de chemins sommets-disjoints partitionnant C_i au problème de trouver le nombre minimum de chemins sommets-disjoints partitionnant Δ_d.

Proposition 6. *S'il existe une partition en chemins sommets-disjoints de $\mathcal{G}$ alors, pour chaque sommet dominant $d \in D_i$ d'une CFC C_i par rapport à l'ensemble des sommets $\mathcal{T}_i$, il existe un chemin Hamiltonien, terminant sur un prédécesseur de d, dans le sous-graphe de $\mathcal{G}$ induit par l'ensemble des sommets de $\bar{\Delta}_d^\star$.*

Démonstration. Rappelons-nous que, par construction de $\bar{\Delta}^{\star}_{d}$, un chemin ne peut pas finir sur un sommet de $\bar{\Delta}^{\star}_{d}$. Puisque d domine tout sommet de $\mathcal{T}_i \subset \Delta^{\star}_{d}$ par rapport à tout sommet de $\bar{\Delta}^{\star}_{d}$, d est la seule sortie possible pour les sommets de $\bar{\Delta}^{\star}_{d}$. Par conséquent, s'il n'existe pas un chemin Hamiltonien dans $\bar{\Delta}^{\star}_{d}$ alors, le sommet d est atteint par au moins deux chemins sommets-disjoints couvrant $\bar{\Delta}^{\star}_{d}$: c'est une contradiction. □

Proposition 7. *Considérons le graphe réduit $\mathcal{G}_{\Delta_d}$ (resp. $\mathcal{G}_{\bar{\Delta}_d}$) associé au sous-graphe d'une CFC C_i de $\mathcal{G}$ induit par les sommets de $\Delta^{\star}_{d}$ (resp. $\bar{\Delta}^{\star}_{d}$). Soit x_{Δ_d} une borne inférieure du nombre minimum de chemins sommets-disjoints partitionnant $\mathcal{G}_{\Delta_d}$. Une borne inférieure du nombre minimum de chemins sommets-disjoints partitionnant $\mathcal{C}_i$ (noté l^d_i), par rapport à un sommet dominant d de C_i, est définie par :*
- *$x_{\Delta_d} - 1$, s'il existe un arc (u, v) dans $\mathcal{C}_i$ tel que $u \in \Delta^{\star}_{d}$, $v \in \bar{\Delta}^{\star}_{d}$, et la CFC contenant le sommet v est un sommet source dans $\mathcal{G}_{\bar{\Delta}_d}$.*
- *x_{Δ_d}, sinon.*

Dans la pratique, la borne inférieure x_{Δ_d} peut être obtenue en réalisant le Lemme 18 page 90 basé sur la relaxation du flot traversant le réseau associé au graphe $\mathcal{G}_{\Delta_d}$ par rapport à l'ensemble des sommets potentiellement finaux $\mathcal{T}_i$. Finalement, la valeur maximale des l^d_i ($d \in D_i$) fournit une borne inférieure du nombre minimum de chemins dans la partition couvrant $\mathcal{C}_i$.

Proposition 8. *Soit ℓ_i le nombre minimum de chemins sommets-disjoints partitionnant la CFC C_i de $\mathcal{G}$ et soit D_i l'ensemble des sommets dominants de $\mathcal{C}_i$, une borne inférieure de ℓ_i est donnée par :*

$$max(\{l^d_i \mid d \in D_i\}) \tag{6.3}$$

Estimer le nombre maximum de chemins entre deux composantes fortement connexes

Une seconde manière de préciser la relaxation du problème K-NDP est d'ajuster les bornes sur le flot $f_{ij} \in [l_{ij}, c_{ij}]$, pour chaque arc du associé au graphe réduit correspondant au graphe $\mathcal{G}$ à partitionner. Cette section montre comment améliorer la précision de la borne supérieure c_{ij}, qui était originellement fixée à $\overline{\mathtt{K}} = |\mathcal{T}|$ dans la Définition 45 page 88 (c.-à-d., le nombre de sommets potentiellement finaux dans $\mathcal{G}$). Ceci est réalisé en calculant la taille d'un couplage de cardinalité maximum entre les sommets i incidents à un arc sortant d'une *CFC* C_i, et entrant sur un sommet j d'une *CFC* C_j.

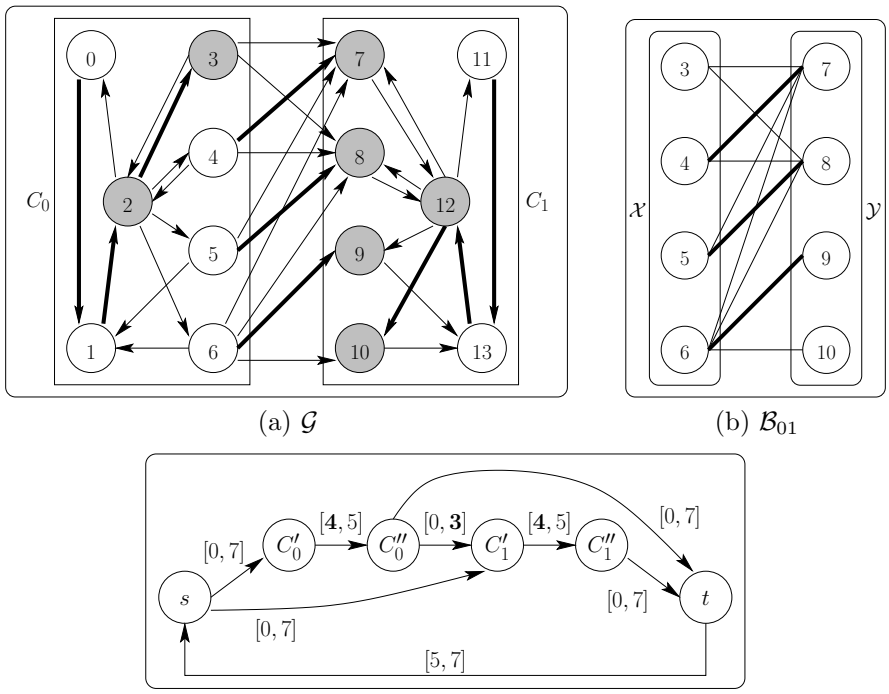
Notation 8. *Étant donné deux CFC distinctes C_i et C_j de $\mathcal{G}$ et un sommet i de $\mathcal{C}_i$,*
- *φ^{+}_{ij} représente le nombre d'arcs sortant de i et entrant sur un sommet j de C_j.*
- *φ^{-}_{ij} représente le nombre d'arcs entrant sur un sommet j de C_j et sortant du sommet i.*

Étant donné deux *CFC* distinctes C_i et C_j de $\mathcal{G}$ telles qu'il existe au moins un arc d'un sommet de C_i à un sommet de C_j, le nombre maximum de chemins sortant de C_i et entrant directement sur C_j peut être calculé à partir d'un graphe bipartie, extrait depuis C_i et C_j, définit de la manière suivante :

Définition 47. *Le graphe biparti $\mathcal{B}_{ij} = (\mathcal{X}, \mathcal{Y}, \mathcal{E})$ associé avec une paire de CFC distinctes $\mathcal{C}_i$ et $\mathcal{C}_j$ de $\mathcal{G} = (\mathcal{V}, \mathcal{A})$, tel qu'il existe un arc depuis un sommet de C_i vers un sommet de C_j, est défini par :*
- *$\mathcal{X} = \{i \in C_i \mid \varphi^{+}_{ij} > 0\}$;*
- *$\mathcal{Y} = \{j \in C_j \mid \varphi^{-}_{ij} > 0\}$;*
- *$\mathcal{E} = \{(i, j) \in \mathcal{A} \mid i \in \mathcal{X},\ j \in \mathcal{Y}\}$.*

Proposition 9. *Soit $\mathcal{M}_{ij}$ la taille d'un couplage de cardinalité maximum dans le graphe biparti $\mathcal{B}_{ij}$ associé à une paire de CFC distinctes, C_i et C_j, de $\mathcal{G}$. La capacité c_{ij}, traversant un arc du DAG (i'', j') du réseau $\mathcal{N}$ associé à $\mathcal{G}$, est bornée par $\mathcal{M}_{ij}$: $c_{ij} \leq \mathcal{M}_{ij}$.*

Exemple 13. *À partir du graphe $\mathcal{G}$ représenté par la figure 6.6(a) page suivante, nous construisons le graphe biparti $\mathcal{B}_{01}$ représenté par la figure 6.6(b) page ci-contre. Un couplage de cardinalité maximum de $\mathcal{B}_{01}$, de taille $\mathcal{M}_{01} = 3$, est dépeint par les arcs en gras de la figure 6.6(b) page suivante. Le nombre minimum de chemins sommets-disjoints partitionnant $\mathcal{G}$ est égal à 5 (les arcs en gras de la figure 6.6(a) page ci-contre représentent les chemins suivants : $\{0, 1, 2, 3\}$, $\{4, 7\}$, $\{5, 8\}$, $\{6, 9\}$ et $\{11, 13, 12, 10\}$). La figure 6.6(c) page suivante montre le réseau résultant $\mathcal{N}$ associé à $\mathcal{G}$. Notons que, pour les arcs associés*



(a) $\mathcal{G}$ (b) $\mathcal{B}_{01}$

(c) Réseau $\mathcal{N}$ associé à $\mathcal{G}$ (les bornes en gras correspondent aux bornes raffinées en utilisant les sommets dominants ou le couplage).

FIG. 6.6 – La Partie (a) représente le graphe orienté $\mathcal{G}$ composé par deux *CFC* C_0 et C_1. Les arcs en gras représentent une solution compatible maximisant le nombre de chemins sommets-disjoints partitionnant $\mathcal{G}$ (5 chemins). Les sommets en gris représentent les sommets potentiellement finaux de $\mathcal{G}$. La Partie (b) illustre le graphe bipartie $\mathcal{B}_{01}$ extrait des *CFC* C_0 et C_1. La Partie (c) représente le réseau $\mathcal{N}$ associé au graphe $\mathcal{G}$.

aux sommets dédoublés (C'_0, C''_0)*, la borne inférieure du nombre minimum de chemins couvrant la CFC* C_0 *est raffinée à la valeur* 4 *grâce à la détection du sommet dominant* 2 *(voir section 6.1.2 page 90). De manière similaire, la borne inférieure sur le nombre de chemins couvrant la CFC* C_1 *est raffinée à la valeur* 4 *grâce au sommet dominant* 12*. Dans le même ordre d'idée, la borne supérieure sur le nombre maximum de chemins disjoints pouvant sortir de la CFC* C_0*, et atteignant la CFC* C_1 *(représenté par la capacité de l'arc associés aux sommets dédoublés* (C''_0, C'_1)*), est raffinée à la valeur* $\mathcal{M}_{01} = 3$*.*

Ainsi, la valeur d'un flot minimum de $\mathcal{N}$ *est égale à* 5*. En effet, supposons que la valeur soit* 4 *(c.-à-d.,* $f_{ts} = 4$*) alors,* 4 *unités de flot peuvent être injectées sur l'arc* (s, C'_0) *car on sait que* $\ell_0 = 4$*, et que au plus* 3 *unités doivent atteindre* C'_1 *: il y a donc une contradiction avec* $\ell_1 = 4$*. Ainsi, il n'existe pas de flot compatible avec* $f_{ts} = 4$*.*

6.2 Une contrainte de partitionnement par des chemins

Une contrainte de partitionnement par des chemins peut être définie par un graphe orienté $\mathcal{G} = (\mathcal{V}, \mathcal{A})$, un nombre de chemins `NPATH` appartenant à un ensemble fini donné, et un ensemble $\mathcal{T} \subseteq \mathcal{V}$ de sommets potentiellement finaux de $\mathcal{G}$. Sans perte de généralités, la description de la contrainte de chemin proposée suit la description faite de la contrainte d'arbre *tree* introduite au chapitre 4 page 47. Ceci nous permet d'utiliser directement les algorithmes de filtrage développés pour la contrainte *tree* afin de prévenir la création de circuits et de chemins ne terminant pas sur des sommets potentiellement finaux. En plus, il est alors possible d'utiliser la restriction sur le degré de chaque sommet du graphe proposée pour la contrainte *extended-tree* au chapitre 5 page 65 pour contraindre chaque sommet du graphe a posséder au

plus un prédécesseur (en veillant toujours à ignorer les boucles).

La contrainte de chemin peut être définie sous la forme *path*(NPATH, NODES), où NPATH est une variable domaine spécifiant le nombre de chemins autorisés dans la partition, et NODES est la collection de n sommets NODES$[1], \ldots,$ NODES$[n]$ représentant le graphe $\mathcal{G}$. Chaque sommet $v_i =$ NODES$[i]$ possède les attributs suivants, complétant la description de $\mathcal{G}$:
- L est un entier défini sur $[1, n]$. Il peut être interprété comme l'*identifiant* du sommet v_i ;
- S est une variable domaine (ou variable *successeur*) dont le domaine est constitué des éléments (l'identifiant des sommets du graphe) de $[1, n]$. Il peut être interprété comme l'unique successeur du sommet v_i. De plus, si $i \in dom($NODES$[i].$S$)$ alors, nous dirons que v_i est un sommet *potentiellement final* pour un chemin de la partition.

FIG. 6.7 – Paramètres de la contrainte *path*(NPATH, NODES).

Pour chaque $i \in [1, n]$, les termes NODES$[i]$.L et NODES$[i]$.S représentent respectivement les attributs L et S de NODES$[i]$. De plus, le graphe $\mathcal{G}$ associé à la contrainte de chemin peut être formellement défini de la manière suivante :

Définition 48. *Le graphe orienté $\mathcal{G} = (\mathcal{V}, \mathcal{A})$, $|\mathcal{V}| = n$ et $|\mathcal{A}| = m$, ainsi que l'ensemble $\mathcal{T}$ associés à une contrainte* path(NPATH, NODES) *sont définis comme suit :*
- $\mathcal{V} = \{i \mid i \in [1, n]\}$;
- $\mathcal{A} = \{(i, j) \mid j \in dom($NODES$[i].S),\ i \neq j)\}$;
- $\mathcal{T} = \{i \mid i \in dom($NODES$[i].S)\}$.

Finalement, pour compléter la description de la contrainte, nous introduisons la description formelle d'une instanciation complète et valide des variables. Pour des raisons de lisibilité, nous allons directement raisonner sur le graphe $\mathcal{G}$ qui modélise la contrainte de chemin.

Définition 49. *Une* instanciation complète *des variables associées à une contrainte* path(NPATH, NODES) *est valide ssi :*
- $\forall i \in [1, n]$: NODES$[i]$.L $= i$;
- $\mathcal{G}$ *est composé de* NPATH *composantes connexes telles que chacune soit un chemin élémentaire terminant sur un sommet potentiellement final de $\mathcal{T}$.*

Le reste de cette section est organisée de la manière suivante : la section 6.2.1 fournie deux conditions nécessaires au partitionnement du graphe $\mathcal{G}$ associé à la contrainte de chemin, dérivées de la condition nécessaire introduite par le théorème 22 page 90. La section 6.2.2 montre comment exploiter cette condition nécessaire dans le but de filtrer la variable NPATH ainsi que les variable successeurs modélisant le graphe $\mathcal{G}$.

6.2.1 Faisabilité

Basée sur le théorème 22 page 90 de la section 6.1.2, nous introduisons un algorithme vérifiant une condition nécessaire pour la faisabilité de la contrainte de chemin vis-à-vis d'un nombre maximum de chemins autorisés max(NPATH) :
- vérifier l'existence d'un flot compatible dans le réseau $\mathcal{N}$ associé à $\mathcal{G}$;
- pour chaque *CFC* C_i de $\mathcal{G}$, pour chaque sommet dominant d de C_i, vérifier qu'il existe au plus un chemin couvrant le graphe $\mathcal{G}_{\bar{\Delta}_d}$ (dérivé de la proposition 6 page 91).[3] Cela peut être effectué en vérifiant que le graphe réduit $\bar{\mathcal{G}}_{\bar{\Delta}_d}$ correspond bien à un chemin élémentaire.

6.2.2 Filtrage

Cette section montre comment filtrer les domaines des variables successeurs NODES$[1]$.S$, \ldots,$ NODES$[n]$.S, ainsi que celui de la variable NPATH à partir du graphe $\mathcal{G}$ associé à la contrainte de chemin. Tout d'abord, il faut ajuster le minimum de NPATH au flot minimum compatible dans le réseau $\mathcal{N}$ associé à $\mathcal{G}$. Puis, pour chaque *CFC* C_i de $\mathcal{G}$, pour chaque sommet dominant $d \in C_i$, supprimer chaque arc (i, j) tel que :

[3] Rappelons-nous que $\mathcal{G}_{\bar{\Delta}_d}$ est le graphe réduit associé au sous-graphe de $\mathcal{G}$ induit par les sommets de $\bar{\Delta}^\star_d$.

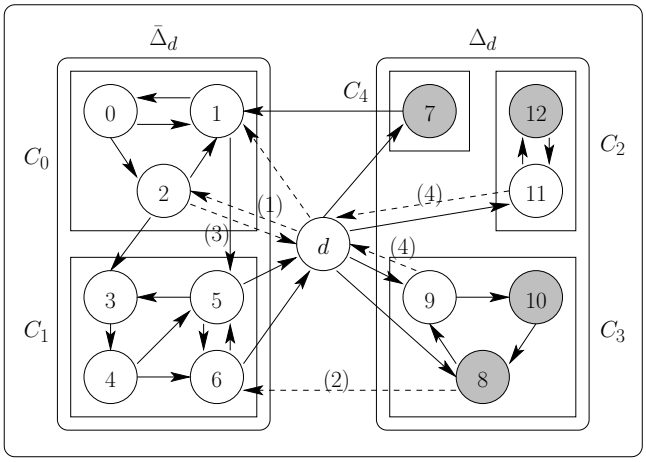


FIG. 6.8 – À partir de la Figure 6.5 page 91 de la Section 6.1.2 page 90, les arcs en pointillés représentent les arcs impossibles de $\mathcal{G}$ détectés par l'algorithme introduit en Section 6.2.2 page précédente.

- cas (1) : $i = d$ et $j \in \bar{\Delta}_d^\star$;
- cas (2) : $i \in \Delta_d^\star$ et $j \in \bar{\Delta}_d^\star$ tels que la *CFC*, créée par le retrait de d et contenant j, n'est pas un sommet source dans le graphe $\mathcal{G}_{\bar{\Delta}_d}$;
- cas (3) : $j = d$ et $i \in \bar{\Delta}_d^\star$ tels que la *CFC*, créée par le retrait de d et contenant i, n'est pas un sommet puit dans le graphe $\mathcal{G}_{\bar{\Delta}_d}$;
- cas (4) : $i \in \Delta_d^\star$, $j = d$ et, $\bar{\Delta}_d^\star \neq \emptyset$.

Exemple 14. *La figure 6.8 illustre les arcs impossibles détectés par l'algorithme précédent. Les arcs* $(d, 1)$ *et* $(d, 2)$ *sont détectés par le cas (1), l'arc* $(8, 6)$ *l'est par le cas (2), l'arc* $(2, d)$ *par le cas (3) et, les arcs* $(9, d)$ *et* $(11, d)$ *sont finalement détectés par le cas (4).*

Il faut observer que l'on aurait aussi pu supprimer tout arc (i, j) de $\mathcal{G}$, tel que i et j appartiennent à des *CFC* distinctes C_i et C_j, si l'arc de $\mathcal{N}$ associé aux sommets dédoublés, correspondant à l'arc (i, j), ne peut porter aucun flot (dans tout flot compatible de $\mathcal{N}$). Ce filtrage n'a pas été inclus dans l'algorithme précédent puisqu'à l'heure actuelle, nous ne connaissons pas de méthode « efficace » captant le fait qu'un arc d'un réseau ne peut porter aucun flot dans tout flot compatible de ce réseau.

6.3 Synthèse

Les algorithmes de filtrage proposés dans les contraintes de chemins précédentes étaient uniquement basés sur la prévention de circuits, ainsi que sur des contraintes maintenant le degré de chaque sommet (chaque sommet ayant au plus un prédécesseur/successeur). Cette partie considère le problème de partitionnement par des chemins pour lequel le nombre de chemins n'est pas fixé mais limité à une valeur donnée. Dans ce contexte, nous proposons un modèle de flot combinant la structure du graphe réduit associé à $\mathcal{G}$, la structure de chaque composante fortement connexe de $\mathcal{G}$ vis-à-vis de la relation de dominance entre les sommets, et la manière dont les sommets entre paires de composantes fortement connexes sont connectés.

Cependant, plusieurs questions restent ouvertes. Premièrement, dans notre approche, chaque sommet dominant est considéré de manière indépendante des autres, et on peut supposer qu'il existe une interaction entre les sommets dominants, de laquelle il serait peut être possible de déduire une amélioration des bornes sur chaque arc associé aux sommets dédoublés du réseau, ainsi qu'un filtrage plus précis. Deuxièmement, dans la section 6.2.2 page précédente, nous avons suggéré une filtrage lié à la détection des arcs qui n'appartiennent à aucun flot compatible du réseau $\mathcal{N}$. À notre connaissance, ce problème, bien que polynomial, ne connait pas de réponse efficace (autre qu'un algorithme basé sur un test pour chaque arc du réseau).

Troisième partie

Implémentation et applications

Chapitre 7

Implémenter une contrainte d'arbre

Sommaire

Les contraintes globales les plus puissantes sont généralement rattachées à un résolveur particulier : *cycle*, *diffn*, *cumulative* avec le résolveur *CHIP* [BC94], *tree* [BFL05] avec le résolveur *Choco*, *standard deviation* [SDDR07] avec *Ilog Solver*. Cette tendance tient probablement au fait que, d'un point de vue implémentation, les contraintes globales sont fortement liées aux services du noyau associé au résolveur pour lequel elles sont développées. En effet, les contraintes globales mettent très souvent en jeu, à la fois des structures de données internes, mais aussi des structures de données fournies par le résolveur. Ces dernières sont le plus souvent des structures dites *restaurables* (« backtrackables ») offertes par les résolveurs en vue de faciliter l'implémentation des contraintes. Cependant, tous les résolveurs ne fournissent pas les mêmes structures restaurables, ce qui conduit finalement l'implémentation des contraintes globales à dépendre fortement des structures de données proposées par le résolveur utilisé. Ce phénomène entraîne aussi des différences de performance notables lorsqu'une même contrainte est déployée sur plusieurs résolveurs. C'est en particulier le cas pour la contrainte *alldifferent* (une des rares contrainte existante dans la plupart des résolveurs).

L'implémentation réelle des contraintes globales étant bien souvent passée sous silence, dans ce chapitre, nous proposons de discuter dans ce chapitre de l'implémentation des contraintes *tree* et *extended-tree* (détaillées aux chapitres 4 page 47 et 5 page 65). Au travers, tout d'abord, de l'implémentation originale de la contrainte *tree* puis de la prise en compte des contraintes additionnelles introduites au travers de la contrainte *extended-tree*. Nous montrons en quoi une implémentation « directe » de telles contraintes rend leur utilisation sur des problèmes de grande taille, en particulier, à cause d'une extrême sensibilité des algorithmes utilisés à la densité des graphes. Ensuite, nous introduisons une nouvelle implémentation de la contrainte *tree*. Cette implémentation est basée sur une approche dynamique de la gestion des graphes permettant à la fois de diminuer la sensibilité des algorithmes à la densité mais aussi, de découpler la contrainte *tree* du résolveur de contrainte l'utilisant. Nous expérimentons cette nouvelle approche au travers des résolveurs *Choco* (`http://choco.sf.net/`) et *Gecode* (`http://gecode.org/`) qui utilise tous deux des systèmes de gestion de la mémoire complètement différents (c.-à-d., le *trail* [AB90] dans le cas de Choco et la *copie* [Sch99] dans le cas de Gecode).

7.1 Implémentation Originale

Dans cette section nous nous attachons à montrer comment la contrainte *tree* est implémentée au sein du résolveur Choco, puis comment nous prenons en compte les contraintes additionnelles introduites dans le cadre de la contrainte *extended-tree*. Finalement, nous concluons cette section sur une série d'expérimentations montrant les limites de la première approche pour les instances de taille moyenne.

7.1.1 La contrainte *tree*

On relève souvent deux situations lorsque l'on doit implémenter une contrainte globale : dans un premier cas, le résolveur de contraintes est connu et par la même les mécanismes de gestion de la mémoire sont imposés ; dans le second cas, on peut choisir un résolveur adapté aux exigences de la contrainte à implémenter c.-à-d., choisir un résolveur proposant les structures de données les plus prometteuses pour la mise en œuvre de la contrainte. Dans le cas de la contrainte *tree*, nous nous sommes concentrés sur le résolveur Choco. En effet, il permet une gestion relativement fine des évènements[1] se produisant sur les variables durant le mécanisme de propagation-recherche. De plus, chaque variable associée à un problème (lors de la déclaration de ce dernier) connaît les contraintes la mettant en jeu, et symétriquement, chaque contrainte connait le sous-ensemble de variables sur lequel elle s'applique. Ainsi, pour une contrainte donnée, différents traitements peuvent être effectués pour chaque type d'évènements se produisant sur les variables misent en jeu dans la contrainte. Ceci conduit à pouvoir atteindre facilement un point fixe pour les algorithmes de filtrages car, chaque contrainte peut spécifier si elle souhaite être réveillée par un évènement qu'elle a produit, ou exclusivement par des évènements extérieurs. Le principal bénéfice d'une telle fonctionnalité est que pour une contrainte n-aire donnée, l'algorithme de filtrage peut être découpé selon chaque type d'évènement se produisant sur les domaines des variables mises en jeu dans la contrainte.

La figure 7.1 décrit la contrainte *tree* telle qu'elle est implémentée dans le résolveur Choco. Le détail de chaque algorithme de propagation a été introduit au chapitre 4 page 47.

Pose de la contrainte *tree* :
- Calculer les bornes `MINTREE` et `MAXTREE`.
- S'il existe au moins une solution satisfaisant la contrainte alors, effectuer la propagation relative à la contrainte :
  1. Mettre à jour `NTREE` par rapport à `MINTREE` et `MAXTREE`.
  2. Propager par rapport au sommets dominants les racines de $\mathcal{G}$.
  3. Propager par rapport à min(`NTREE`) et max(`NTREE`).

Pour chaque évènement se produisant sur le domaine d'une variable mise en jeu dans la contrainte faire :
- Si cet évènement se produit sur la variable `NTREE` alors :
  1. Propager par rapport à min(`NTREE`) et max(`NTREE`).
- Si cet évènement se produit sur une variable modélisant un sommet de $\mathcal{G}$ alors :
  1. Mettre à jour `NTREE` par rapport à `MINTREE` et `MAXTREE`.
  2. Propager par rapport aux éventuels nouveaux sommets dominants découvert dans $\mathcal{G}$.

FIG. 7.1 – Squelette de l'algorithme de filtrage implémentant la contrainte *tree* dans le résolveur Choco.

Le résolveur de contraintes Choco est basé sur un système de mémorisation automatique des modifications (dit « trailing » [AB90])[2] comme l'instanciation d'une variable, le retrait de valeurs dans les domaines, mais aussi, les effets de tels évènements sur les structures de données restaurables propres à la contrainte. Dans notre cas, ces effets consistent exclusivement en la modification de la structure de donnée

[1] Le résolveur *Choco* distingue trois types principaux d'évènements : l'instanciation d'une variable, le retrait d'une valeur dans le domaine d'une variable, et la mise à jour de la borne inférieure (resp. borne supérieure) du domaine d'une variable.

[2] Une approche basée sur le « trailing » mémorise pour chaque évènement modifiant une structure de données, l'information nécessaire pour défaire cette modification.

représentant le graphe $\mathcal{G}$ et ses propriétés (composantes fortement connexes, sommets dominants). Par exemple, si un arc (i, j) est supprimé de $\mathcal{G}$ (c.-à-d., $j \notin \mathcal{D}(\texttt{VER}[i].\texttt{S})$) alors, ce retrait peut entraîner :

1. La diminution du nombre de racines potentielles, si $i = j$. Ceci conduit à une mise à jour de `MAXTREE` ;
2. L'augmentation du nombre de composantes puits. Ceci entraine une mise à jour de `MINTREE` ;
3. La création de nouveaux sommets dominants des racines potentielles dans $\mathcal{G}$.

Ainsi, pour pouvoir mémoriser les informations nécessaires à la restauration des structures de données mise en jeu dans la contrainte, le résolveur Choco propose différentes structures de données *restaurables* utilisant des types de données mémorisables sur les `entiers`, les `booléens` et les `BitSets`. La contrainte *tree* utilise ces structures de données pour dynamiquement mémoriser et restaurer les propriétés de graphes utiles comme les composantes fortement connexes et les sommets dominants du graphe $\mathcal{G}$. Plus précisément, la contrainte *tree* s'appuie sur une représentation du graphe $\mathcal{G}$ basée sur un tableau de `BitSet` de taille n (où n désigne le nombre de sommets du graphe). D'autre part, la liste des sommets dominants ainsi que la liste des composantes fortement connexes est elle aussi mémorisée à l'aide de `BitSet`. Ainsi, lors de chaque modification des domaines des variables misent en jeu dans la contrainte *tree*, on est capable de retrouver quelle modification est survenue sur le graphe $\mathcal{G}$, ses sommets dominants, ou ses composantes fortement connexes.

7.1.2 La contrainte *extended-tree*

Nous nous intéressons dans cette section à l'implémentation de la contrainte *extended-tree* introduite au chapitre 5 page 65. Chacune des restrictions que la contrainte *extended-tree* met en jeu peut être modélisée à l'aide d'un graphe additionnel : $\mathcal{G}_{prec}$ pour les contraintes de précédence, $\mathcal{G}_{cond}$ pour les contraintes de précédence conditionnelle, $\mathcal{G}_{inc}$ pour les contraintes d'incomparabilités, et $\mathcal{G}_f$ pour le graphe de flot associé aux contraintes sur le degré de chaque sommet du graphe. Ainsi, nous utilisons toujours la représentation sous la forme de tableaux de `BitSet` introduite pour la contrainte *tree*.

Nous détaillons maintenant le schéma général de l'algorithme de filtrage de la contrainte *extended-tree* lors de son réveil initial. Étant donné que la propagation de cette contrainte relève d'un problème difficile, les algorithmes que nous avons proposés au chapitre 5 page 65 ne sont pas complets. Ainsi, nous proposons lors du réveil de la contrainte de saturer les algorithmes de filtrage proposés pour chaque restriction, dans le but de détecter au plus tôt un maximum de valeurs inconsistantes dans les domaines des variables.

Lors du réveil de la contrainte *extended-tree* :
- Calculer les bornes `MINTREE` et $\texttt{MAXTREE}_{prec}$.
- Calculer les bornes `MINPROP` et `MAXPROP`.
- Mettre à jour le graphe de précédence $\mathcal{G}_{prec}$ par rapport aux graphes $\mathcal{G}$ et $\mathcal{G}_{inc}$.
- Synchroniser $\mathcal{G}_{prec}$ et $\mathcal{G}_{cond}$ par rapport au graphe $\mathcal{G}$. Cette synchronisation consiste à mettre à jour $\mathcal{G}_{prec}$ vis à vis des arcs de $\mathcal{G}_{maybe}$ devenus des arcs de $\mathcal{G}_{sure}$, et à ajouter dans $\mathcal{G}_{prec}$ les arcs de $\mathcal{G}_{cond}$ qui sont devenus des contraintes de précédence.
- Mettre à jour le graphe de flot $\mathcal{G}_f$ associé aux contraintes sur le degré de chaque sommet de $\mathcal{G}$.
- S'il existe au moins une solution satisfaisant la contrainte *extended-tree* alors :
  1. Appliquer l'algorithme de propagation relatif à la contrainte *tree*.
  2. Appliquer l'algorithme de propagation relatif à la contrainte *proper-tree*.
  3. Propager par rapport aux contraintes de précédences.
  4. Propager par rapport aux contraintes d'incomparabilités.
  5. Propager par rapport aux contraintes sur le degré de chaque sommet de $\mathcal{G}$.
- Tant qu'une modification est survenue sur l'un des graphes $\mathcal{G}$, $\mathcal{G}_{prec}$, $\mathcal{G}_{inc}$, $\mathcal{G}_f$ alors commencer un nouveau réveil.

FIG. 7.2 – Schéma général de l'algorithme de filtrage lors du réveil initial de la contrainte *extended-tree*.

La motivation principale de la mise en place d'une boucle de saturation effectuant elle-même le réveil de la contrainte, repose sur le fait que toute modification intervenant sur l'un des graphes $\mathcal{G}$, $\mathcal{G}_{prec}$, $\mathcal{G}_{cond}$,

$\mathcal{G}_{inc}$, $\mathcal{G}_f$ peut potentiellement entraîner la détection de nouveaux arcs impossibles dans le graphe $\mathcal{G}$, ou de valeurs impossibles dans les domaines de `NTREE` et `NPROP`. De manière réciproque, tout retrait d'arc de $\mathcal{G}$ ou toute mise à jour des bornes de `NTREE` et `NPROP` peut entraîner la modification des graphes $\mathcal{G}_{prec}$, $\mathcal{G}_{cond}$, $\mathcal{G}_{inc}$ et $\mathcal{G}_f$. Il s'agit donc bien d'une boucle de saturation permettant d'atteindre un point fixe où les structures $\mathcal{G}$, $\mathcal{G}_{prec}$, $\mathcal{G}_{cond}$, $\mathcal{G}_{inc}$, $\mathcal{G}_f$ sont dans un état stable vis-à-vis des algorithmes de filtrages.

7.1.3 Les limites de l'approche : illustration avec la contrainte *tree*

En nous replaçant maintenant dans le contexte de la contrainte *tree*, nous montrons quelles sont les limites d'un telle implémentation. En particulier, nous allons mettre en lumière l'extrême sensibilité des algorithmes « classiques » face à la densité du graphe $\mathcal{G}$ associé à la contrainte. Le tableau 7.1 illustre cette sensibilité. Les expérimentations ont été effectuées avec la version 1.2.04 du résolveur Choco sur une machine Intel Xeon avec 2.4GHz de CPU et 1Go RAM dont, 128Mo alloué à la machine virtuelle Java. Pour chaque taille de graphe dans $\{25, 50, 75, 100, 150, 200\}$, et les densités tirées dans l'intervalle $[0.05; 1]$ avec un pas de 0.05, nous avons généré et résolu 30 instances (soit 3600 graphes au total).

Ordre du graphe	Densité	Temps moyen (*ms*)
25	≤ 0.5	55
	> 0.5	90
50	≤ 0.5	610
	> 0.5	1532
75	≤ 0.5	3856
	> 0.5	8896
100	≤ 0.5	13040
	> 0.5	32568
150	≤ 0.5	69220
	> 0.5	219174
200	≤ 0.5	204497
	> 0.5	> 300000

TAB. 7.1 – Évaluation de l'implémentation de la contrainte *tree* originale sur des graphes aléatoires.

Dans la suite de ce chapitre, nous présentons une nouvelle technique d'implémentation permettant de s'abstraire partiellement de cette dépendance vis-à-vis de la densité du graphe considéré. Pour cela, nous allons essayer de prendre en compte au mieux le caractère « local » des modifications engendrées par les évènements survenant sur les domaines des variables mises en jeu dans la contrainte *tree*. En effet, il est facile de constater que chaque modification sur les domaines des variables représentant des sommets du graphe $\mathcal{G}$ associé à la contrainte, ne modifie que le voisinage du sommet concerné dans le graphe $\mathcal{G}$. Ceci pose donc la question suivante : est-il toujours nécessaire de recalculer des propriétés telles que les composantes fortement connexes de $\mathcal{G}$ de manière globale après chaque évènement survenant sur le graphe ? La réponse à cette question ne résume-t-elle pas à l'étude (ou tout simplement l'utilisation) d'algorithmes de graphes *dynamiques* maintenant ces propriétés ?

7.2 Vers une implémentation « portable »

L'intuition de cette nouvelle implémentation est essentiellement basée sur l'utilisation d'algorithmes spécifiques permettant d'éviter de recalculer sur le graphe $\mathcal{G}$ des propriétés qui pourraient être maintenues de manière incrémentale. Pour cela, l'état de l'art sur les algorithmes de graphes proposes de nombreux algorithmes dit *totalement dynamiques* [EGI97] maintenant différentes propriétés de graphes utiles pour la contraintes *tree*. Le caractère *totalement dynamiques* de ces algorithmes autorise le maintient des propriétés tant après l'ajout que le retrait d'arcs dans le graphe $\mathcal{G}$ associé à la contrainte. Cette bonne caractéristique conduit à poser deux questions :

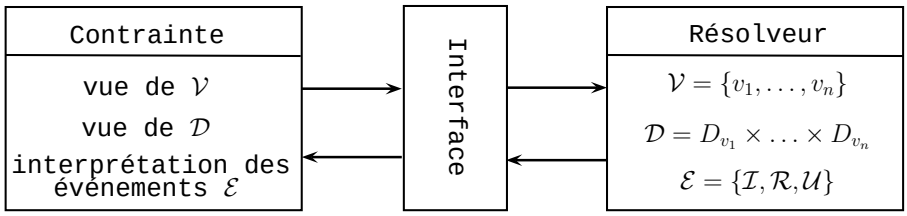


FIG. 7.3 – Une *contrainte adaptable* définie par rapport à trois types de données : une vue des variables $\mathcal{V}$ qui définissent le problème, une vue des domaines de chaque variable $\mathcal{D}$ et, une interprétation des évènements qui se produisent sur les domaines des variables. Ces évènements peuvent être des *instanciations* de variables ($\mathcal{I}$), des retraits de valeurs dans les domaines ($\mathcal{R}$) et, des mises à jour ($\mathcal{U}$) de bornes inférieures ou supérieures des domaines.

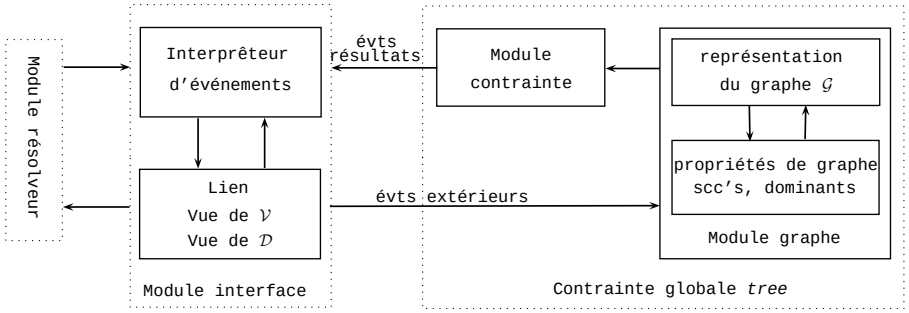


FIG. 7.4 – Schéma général d'implémentation de la contrainte *tree* adaptable.

- Est-il réellement nécessaire d'utiliser des structure de données restaurables s'il existe des algorithmes autorisant un maintient totalement dynamique ? En effet, il existe dans la théorie des graphes de nombreux algorithmes permettant de maintenir (sans recalcul global) des propriétés (comme les composantes fortement connexes, la fermeture et la réduction transitive, etc) lors de l'ajout ou le retrait d'arcs dans le graphe considéré.
- Si finalement les structures de données restaurables ne sont pas utilisées (c.-à-d., qu'il n'est pas nécessaire de mémoriser les variations des structures de la contrainte), quel lien exact reste-t-il entre la contrainte et le résolveur ? Vraisemblablement ce lien est extrêmement ténu, ce qui conduit à se dire que dans le cas de la contrainte *tree*, on doit être capable de fournir une implémentation indépendante du résolveur utilisé.

Ceci ouvre alors de nouvelles perspectives : d'une part, la contrainte doit être portable sur n'importe quel résolveur de PPC, et d'autre part, elle doit aussi pouvoir être utilisé sur des résolveurs de problèmes autres que des résolveurs PPC comme peuvent l'être COMET, un environnement de développement pour la recherche locale [HM07], ou PaLM, une extension du résolveur Choco basée sur les explications [JB00]. Ainsi, nous allons maintenant montrer que l'implémentation de la contrainte *tree* au travers d'une gestion dynamique des structures de données transforme cette dernière en « plugin » pour différents résolveur (voir figure 7.3).

7.2.1 Mise en œuvre

D'un côté, il est possible de résumer les limites de la complexité théorique de la contrainte *tree* au calcul répétitif de certaines propriétés de graphes (composantes fortement connexes (*CFC*), fermeture transitive, sommet dominants) sur le graphe $\mathcal{G}$ entre deux étapes de recherche effectuées par le résolveur. D'un autre côté, les techniques de recherches sont uniquement basées sur la sélection et le retrait de valeurs dans les domaines des variables, ou sur la restauration de valeurs dans les domaines. Dans le contexte de la contraintes *tree*, ces techniques sont donc amenées à modifier uniquement la variable `NTREE` ainsi que l'ensemble des arcs contenus dans le graphe $\mathcal{G}$.

Une nouvelle approche implémentant la contrainte d'arbre peut naturellement être décomposée de la

manière suivante (voir figure 7.4 page précédente) :

1. Le **module graphe** est basé sur une structure de données totalement dynamique modélisant la graphe $\mathcal{G}$ et ses propriétés associées (c.-à-d., les composantes fortement connexes, les sommets dominants et la fermeture transitive). Ce module contient les primitives mettant à jour les structures de données par rapport aux retraits (lors du filtrage) et aux ajouts d'arcs (lors de la restauration). Ces primitives calculent naturellement chaque propriétés sur le sous-graphe nécessaire induit par le graphe original $\mathcal{G}$.
2. Le **module contrainte** propose les algorithmes de filtrage (basés exclusivement sur les propriétés maintenues par le module graphe) détectant et supprimant les arcs de $\mathcal{G}$ inconsistants avec la contrainte *tree*.
3. Le **module interface** maintient une relation bijective entre les évènements survenant sur les domaines des variables (c.-à-d., retraits/ajouts de valeurs dans les domaines) et les évènements survenant sur le graphe $\mathcal{G}$ (c.-à-d., retraits/ajouts d'arcs).

En pratique, dans l'implémentation courante de la contrainte *tree* adaptable, lors de chaque événement survenant sur un domaine d'une variable : premièrement, cet événement est interprété par le module interface ; ensuite, le module graphe met à jour ses structures de données ; ensuite, les algorithmes de filtrages de la contrainte sont appliqués ; finalement, les évènements résultants fournis par la contrainte sont interprétés en terme de mises à jour des domaines des variables.

Nous détaillons maintenant les fonctionnalités du module interface. On distingue quatre évènements concernant le domaine des variables d'une contrainte : les retraits de valeurs dans les domaines, l'instanciation de variables, la mise à jour des bornes inférieures et des bornes supérieures. Cependant, les évènements liés à l'instanciation et à la mise à jour de bornes peuvent être naturellement ramenés à un ensemble d'évènements de type retrait. Ainsi, pour chaque type d'évènements reçus par le module interface depuis le résolveur, une transformation de l'événement reçu en un ensemble de retraits d'arcs est effectuée par le module graphe. Cependant, tous les retraits ne sont pas traités de la même manière par l'interface. En effet, l'événement à l'origine d'un ensemble de retraits n'est pas totalement ignoré par le module graphe, dans le but de pouvoir améliorer l'efficacité du traitement. Par exemple, un ensemble de retraits liés à un événement d'instanciation sur une variable conduit à une modification du voisinage du sommet correspondant dans le graphe $\mathcal{G}$. Cette information peut être prise en compte dans le but d'effectuer ces modifications de manière plus efficace.

En d'autre termes, cette nouvelle approche pour implémenter une contrainte globale conduit à une ré-organisation du code de la contrainte. Chaque partie est clairement isolée et identifiée : la gestion de la structure de données avec la modification des domaines (évènements), les algorithmes liés à la propagation (le cœur de la contrainte), et le maintient des propriétés de graphe nécessaires pour la propagation (le module graphe). Cette isolation des différentes parties composant une contrainte globale est un point relativement important dans la conception de contraintes globales avancées.

7.2.2 Expérimentations

Nous nous intéressons maintenant à une série d'expérimentations évaluant la contrainte *tree* adaptable. Dans un premier temps, nous comparons les résultats obtenus avec l'implémentation originale *ad-hoc* de la contrainte. Ensuite, nous évaluons les performances de la contrainte *ad-hoc* sur deux résolveurs de contraintes : Gecode et Choco. Comme précédemment, Les expérimentations ont été effectuées avec la version 1.2.04 du résolveur Choco et la version 1.3.1 du résolveur Gecode sur une machine Intel Xeon avec 2.4GHz de CPU et 1Go RAM dont, 128Mo alloué à la machine virtuelle Java.

Implémentation *ad-hoc* versus implémentation *adaptable*

L'objectif de ces expérimentations est de montrer que la contrainte *tree* adaptable surclasse largement la version ad-hoc initialement développée. De plus, l'approche purement dynamique proposée par cette nouvelle implémentation est, en moyenne, beaucoup moins sensible à la variation de la densité du graphe. Ce comportement constitue une très nette amélioration pratique par rapport à l'implémentation ad-hoc orignale [BFL06].

Ordre du graphe	Densité	Temps moyen (ms)	
		Ad-hoc	Adaptable
25	≤ 0.5	55	45
	> 0.5	90	38
50	≤ 0.5	610	310
	> 0.5	1532	307
75	≤ 0.5	3856	1174
	> 0.5	8896	1064
100	≤ 0.5	13040	3156
	> 0.5	32568	2682
150	≤ 0.5	69220	11543
	> 0.5	219174	9645
200	≤ 0.5	204497	33763
	> 0.5	> 300000	26315

TAB. 7.2 – Évaluation de la contrainte *tree* adaptable par rapport son implémentation ad-hoc.

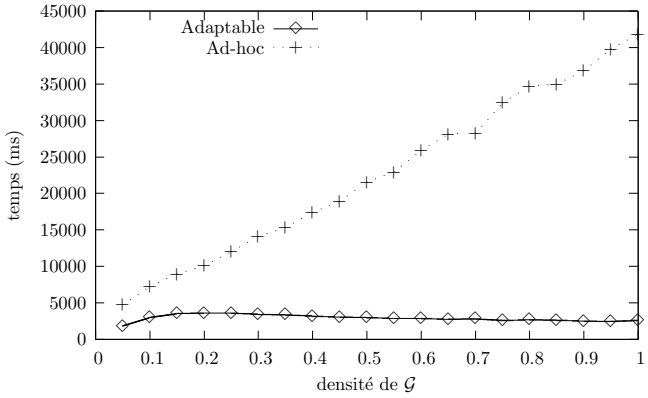


FIG. 7.5 – Résultats détaillés pour un graphe orienté d'ordre 100, la courbe en pointillé représente les résultats de l'implémentation ad-hoc de la contrainte *tree* tandis que la courbe en traits pleins donne les résultats de l'implémentation adaptable.

Cet ensemble d'expérimentations démontre deux apports essentiels de la version adaptable de la contrainte *tree*. Comme précédemment, pour chaque taille de graphe dans $\{25, 50, 75, 100, 150, 200\}$, et les densités tirées dans l'intervalle $[0.05; 1]$ avec un pas de 0.05, nous avons généré et résolu 30 instances (soit 3600 graphes étudiés). Notons que la durée d'exécution maximum est limitée à 300000ms, et qu'une heuristique de choix de variable et de valeur aléatoire est utilisée.

Premièrement, la table 7.2 met en avant une amélioration globale de l'implémentation ad-hoc de la contrainte *tree*. En effet, la version adaptable est 3.8 fois plus rapide dans le cas de graphes avec une densité inférieure ou égale à 0.5, et 10 fois plus rapide dans le cas des graphes avec une densité supérieure à 0.5. Deuxièmement, les figures 7.5 et 7.6 page suivante fournissent les comportements respectifs des contraintes ad-hoc et adaptables pour un graphe donné d'ordre 100. La figure 7.6 page suivante met en avant le ratio entre des temps d'exécution de la contrainte adaptable et de la version ad-hoc. Dans les deux cas, la version adaptable s'avère plus performante que la version ad-hoc dans le cas de graphes denses.

Nous pouvons maintenant détailler les raisons expliquant l'écart de performances entre les deux implémentations. Premièrement, nous allons nous intéresser à l'implémentation de la condition de faisabilité introduite au théorème 6 page 50. Deuxièmement, nous montrerons comment le filtrage a pu être amélioré en pratique.

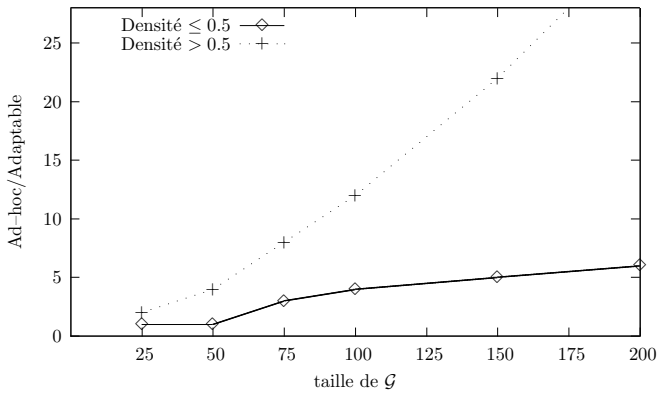


FIG. 7.6 – La courbe en pointillé représente le ratio entre les temps d'exécution de la contrainte ad-hoc et de la contrainte adaptable, dans le cas d'une densité du graphe $\mathcal{G}$ supérieure à 0.5. La courbe en traits pleins représente le même ratio dans le cas d'une densité inférieure ou égale à 0.5.

Lors de chaque réveil de la contrainte, la version ad-hoc de la contrainte *tree* recalcule les composantes fortement connexes (*CFC*) associées au graphe $\mathcal{G}$ par une procédure de recherche en profondeur d'abord introduite par Tarjan [Tar72], ayant une complexité en $O(n+m)$. Cependant, recalculer les *CFC* lors de chaque réveil est inutile. En effet, durant les étapes de propagation/recherche, les retraits ou les ajouts d'arcs dans $\mathcal{G}$ induisent uniquement des modifications locales de $\mathcal{G}$. Alors, le coût de recalcul des scc peut être significativement réduit en s'intéressant uniquement au sous-graphe nécessaire induit par les *CFC* précédemment calculées. En pratique, durant la recherche, la taille des *CFC* ne cesse de diminuer (alors que leur nombre augmente) jusqu'à être réduites à un unique sommet. Ainsi, le maintien dynamique des *CFC* est la première raison des améliorations fournies par la contrainte *tree*.

Dans l'algorithme de filtrage original, la contrainte détecte les sommets dominant [LT79] l'ensemble des racines potentielles du graphe $\mathcal{G}$. Pour chaque sommet dominant, l'algorithme de filtrage détecte et supprime les arcs sortants qui ne peuvent pas atteindre au moins une racine potentielle. Ainsi, pour chaque sommet dominant, il faut calculer un arbre de recherche en profondeur d'abord pour détecter si une racine potentielle peut être atteinte ou pas. Globalement, ceci est effectué en $O(nm)$. Dans le cas de la contrainte adaptable une nouvelle approche est proposée. Nous associons au graphe $\mathcal{G}$, sa fermeture transitive. Le calcul de cette fermeture transitive de $\mathcal{G}$ n'est effectué explicitement que durant l'étape de propagation initiale en $O(nm)$. Ensuite, un maintien dynamique de la fermeture transitive est effectué par une mise à jour incrémentale calculée sur un sous-graphe de $\mathcal{G}$ induit par les évènements ayant modifié $\mathcal{G}$. En pratique, ce maintient dynamique est efficace. Finalement, la connaissance de la fermeture transitive de $\mathcal{G}$ fournit les conditions d'accessibilités nécessaires pour effectuer le filtrage des arcs sortants des sommets dominant directement lorsque ces sommets sont identifiés par l'algorithme.

Évaluation de la portabilité

L'objectif de cette expérimentation est d'illustrer la portabilité de la nouvelle implémentation de la contrainte *tree*. Le choix des résolveurs de contraintes Gecode et Choco montre une légère dégradation dans le cas du résolveur Gecode. Nous montrerons que cette dégradation est liée au module interface gérant le lien entre la contrainte *tree* et le résolveur.

Gecode est un résolveur centré sur la propagation c.-à-d., que les contraintes ne connaissent pas les causes de leur réveil. En d'autres termes, ce résolveur ne peut pas dynamiquement enregistrer les évènements survenant sur les variables durant la recherche. Ainsi, c'est le module interface qui calcule ces évènements dans le but de les traduire en termes de retraits/ajouts d'arcs dans le module graphe.

Le module interface est un observateur surveillant les modifications survenant sur les domaines des variables misent en jeu. Lors de la pose de la contrainte, l'observateur mémorise une copie des domaines de chaque variable. Ensuite, lors de chaque réveil de la contrainte, l'observateur compare l'état courant des domaines de chaque variables à leur copie. Si un domaine a été modifié depuis le dernier réveil,

Taille du graphe	Temps d'exécution *Choco* (ms)			Temps d'éxécution *Gecode* (ms)		
	Module interface	Module contrainte	Module graphe	Module interface	Module contrainte	Module graphe
25	5	10	27	6	10	27
50	5	57	229	24	56	228
75	11	190	863	58	186	879
100	18	440	2287	120	439	2310
150	50	1466	9524	368	1329	9596
200	82	3214	27551	812	3009	27097

TAB. 7.3 – Comparaison des temps d'exécution de la contrainte *tree* adaptable sur deux résolveurs distincts (Choco et Gecode). Pour chaque résolveur, le temps d'exécution de chaque partie est détaillé.

l'observateur met à jour la copie des domaines et envoie les modifications d'arcs au module graphe. Ainsi, dans le pire cas il y a $O(|\mathcal{V}| \times |\mathcal{D}(v)|)$ comparaisons de valeurs. En pratique, ce traitement peut être effectué en utilisant des structures de données classiques comme les Java `BitSet`.

Ce calcul conduit à un surcoût du module interface lorsque la contrainte est utilisée avec le résolveur Gecode. Notons que ce n'est pas le cas avec le résolveur Choco car il s'agit d'un résolveur de contraintes centré sur les variables. Ainsi, le module interface connaît quel type de modifications est survenu sur les domaines des variables sans avoir à les parcourir. Par conséquent, lorsque le module interface est réveillé, il transforme uniquement les évènements survenant sur les variables en évènements sur le graphe et les envoie au module graphe.

Pour chaque taille de graphe dans $\{25, 50, 75, 100, 150, 200\}$, et les densités tirées dans $\{0.05, 0.2, 0.4, 0.5, 0.6, 0.8, 0.95\}$, nous avons généré et résolu 50 instances (soit 2100 graphes en tout). Le tableau 7.3 résume les temps de calcul de chaque partie composant la contrainte *tree* adaptable (Figure 7.4 page 103), à la fois pour les résolveurs de contraintes Choco et Gecode. Notons que pour chaque résolveur une heuristique aléatoire de choix de variables/valeurs est utilisée.

La colonne « Module interface » du tableau 7.3 illustre parfaitement le surcoût dut à l'interface dans le cas du résolveur Gecode. De plus, nous pouvons noter que les temps d'exécution relatifs à la contrainte elle-même sont équivalents dans les deux résolveurs : les colonnes « Module contrainte » et « Module graphe » illustrent parfaitement ce résultat.

7.3 Conclusion

Dans ce chapitre, nous venons de débattre de l'intérêt de séparer l'implémentation des contraintes globales liées au partitionnement d'un quelconque résolveur. Nous avons illustré, par le biais de la contrainte *tree*, que cette séparation permettait non seulement de rendre la contrainte portable sur plusieurs résolveurs mais aussi, de gagner en performances pures par la mise en place d'algorithmes de graphes totalement dynamiques.

Il faut noter que l'indépendance de la contrainte *tree* est directement liée au fait que les propriétés de graphe qu'elle met en jeu peuvent être maintenues de manière totalement dynamique. Un effet de bord relativement intéressant pour la communauté réside dans le fait que la conception de contraintes globales « portables » permet une large diffusion de ces dernières sur différents résolveurs. En effet, la plupart du temps les contraintes globales « avancées » sont rattachées à un unique résolveur et ne sont que très rarement déployées sur d'autres.

Toutefois, comme nous l'avons montré dans ce chapitre, la contrainte *tree* nécessite le maintien de différentes propriétés de graphe. Pour chacune de ces propriétés, il existe dans l'état de l'art un ou plusieurs algorithmes incrémentaux détaillant une implémentation efficace. Cependant, chacun de ces algorithmes nécessitent des structures de données spécifiques. Un challenge intéressant (qui peut aussi constituer une limite) consiste donc à déployer une structure de données unique pour la contrainte, maintenant l'ensemble des propriétés en évitant de dégrader les meilleures complexités théoriques connues pour chacune d'entre elles. Pour cette première approche, ce compromis peut encore être amélioré.

Chapitre 8

Applications

Sommaire

Nous nous intéressons dans ce chapitre à une série d'expérimentations de la contrainte *extended-tree* sur trois problèmes pratiques. Tout d'abord, la section 8.1 étudie le problème de phylogénie de super-arbre au travers d'un certain nombre d'instances issues de problème réels, et compare les résultats obtenus par la contrainte *extended-tree* vis-à-vis d'une approche dédiée à cette application. Ensuite, la section 8.2 page 113 présente les résultats de la contrainte *extended-tree* sur un problème de cheminement qui consiste en la construction d'un chemin induit par un ordre partiel entre un sous-ensemble des sommets du graphe initial. Nous comparons, là encore, les résultats de la contrainte *extended-tree* avec une approche spécifique à ce problème. Finalement, la section 8.3 page 114 fournit les résultats de la contrainte *extended-tree* pour un ensemble d'instances aléatoires du problème de chemin Hamiltonien.

Toutes les expérimentations présentées dans cette section ont été effectuées avec la version 1.2.04 du résolveur Choco sur une machine Intel Pentium 4 avec 3GHz de CPU et 1Go de RAM (512Mo ont été alloués à la machine virtuelle Java).

8.1 Reconstruction de super-arbres en phylogénie

Un des objectifs de la phylogénie est de reconstruire la généalogie des espèces vivantes. Cette généalogie est idéalement représentée par un arbre appelé « arbre de la vie ». Il est constitué de feuilles représentant les espèces contemporaines et de sommets internes représentant des espèces éteintes dont on ne connait pas forcément le nom. Un problème central en phylogénie est la construction d'un super-arbre généalogique [BEGS02] compatible avec plusieurs arbres généalogiques déjà connus. Il existe dans la littérature différentes définitions de la compatibilité d'un super-arbre avec un ensemble d'arbres donnés :

Définition 50 (Compatibilités forte, faible et stable)**.**

- *Un arbre $\mathcal{T}$ est dit* fortement compatible *avec un arbre $\mathcal{T}'$, si $\mathcal{T}'$ est topologiquement équivalent à un sous-arbre de $\mathcal{T}$ respectant l'agencement des noms des sommets.[NW96]*
- *Un arbre $\mathcal{T}$ est dit* faiblement compatible *avec un arbre $\mathcal{T}'$, si $\mathcal{T}'$ peut être obtenu à partir de $\mathcal{T}$ par une série de* contradictions *d'arcs.*[1] *[Ste92]*
- *Un arbre $\mathcal{T}$ est dit* stablement compatible *avec un ensemble d'arbres $\mathcal{S}$, si $\mathcal{T}$ est faiblement compatible avec chaque arbre de $\mathcal{S}$ et chaque sommet interne de $\mathcal{T}$ peut être nommé par au moins un sommet interne d'un arbre de $\mathcal{S}$.*

[1] La *contradiction* d'un arc $a = (v, w)$ est la substitution des sommets v et w par un unique sommet dont les arcs incidents sont ceux qui étaient incidents à v et w à l'exception de a.

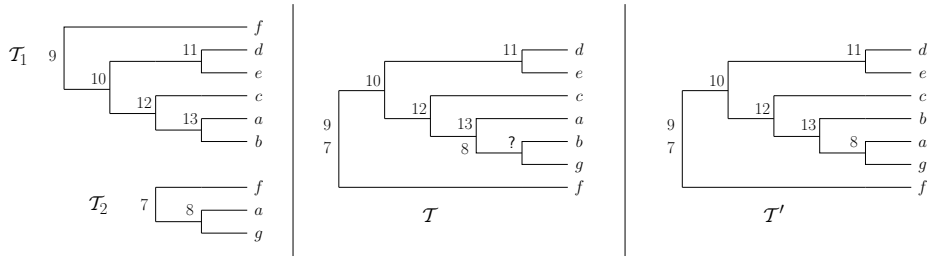


FIG. 8.1 – Une instance d'un problème de super-arbre et deux solutions compatibles.

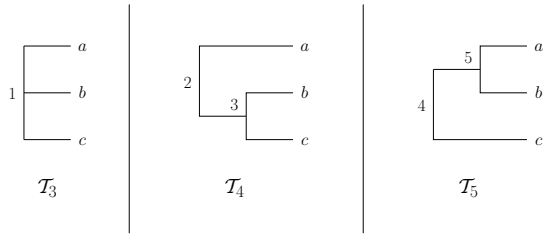


FIG. 8.2 – Trois « petits » arbres phylogéniques.

Pour le problème de reconstruction de super-arbres, les compatibilités forte, faible et stable coïncident ssi tous les arbres donnés en entrée sont binaires [NW96]. L'existence de solutions reste valable lorsque l'on restreint la compatibilité faible à la compatibilité stable.

Par exemple, les arbres $\mathcal{T}_1$ et $\mathcal{T}_2$ de la figure 8.1 acceptent $\mathcal{T}$ et $\mathcal{T}'$ comme super-arbres à la fois pour la compatibilité faible et forte. Comme on peut le voir, tous les sommets internes de $\mathcal{T}'$ peuvent être nommés par un sommet interne des deux arbres donnés en entrée, mais ce n'est pas le cas pour le père des sommets b et g de l'arbre $\mathcal{T}$. Par conséquent, $\mathcal{T}$ et quatre autres super-arbres sont tendancieux car ils font des *hypothèses sur l'existence d'espèces éteintes* qui n'appartiendraient à aucun des arbres donnés en entrée. Considérons aussi les trois arbres de la figure 8.2 : les arbres $\mathcal{T}_3$ et $\mathcal{T}_4$ acceptent l'arbre $\mathcal{T}_4$ comme super-arbre par rapport à la définition de compatibilité faible, en effet, il suffit de contracter l'arc $(3, 2)$ pour obtenir $\mathcal{T}_3$ à partir de $\mathcal{T}_4$. Cependant, les arbres $\mathcal{T}_3$ et $\mathcal{T}_4$ n'acceptent pas de super-arbre vis-à-vis de la définition de compatibilité forte. En effet, le plus récent ancêtre commun aux sommets b et c, noté $mrca(b, c)$, est le même sommet que $mrca(a, b)$ dans l'arbre $\mathcal{T}_3$, il est noté 1, mais ce n'est pas le même dans l'arbre $\mathcal{T}_4$, puisque le sommet correspondant au $mrca(b, c) = 3$ est un descendant du sommet correspondant au $mrca(a, b) = 2$. De la même manières, les arbres $\mathcal{T}_4$ et $\mathcal{T}_5$ n'acceptent de superarbres ni par rapport à la compatibilité faible, ni par rapport à la compatibilité forte.

Le premier algorithme répondant au problème de reconstruction d'un super-arbre a été donné dans [ASSU81]. Cet algorithme est basé sur la définition de compatibilité forte, sa complexité est de l'ordre de $O(\ell^2)$, où ℓ est le nombre de feuilles des arbres donnés en entrée. Beaucoup d'algorithmes dérivés ont depuis émergés. On retiendra surtout l'algorithme *OneTree* [NW96]. La première approche par programmation par contraintes a été proposée dans [GPSW03] sans utiliser de contraintes globales. Concernant la compatibilité faible, un algorithme classique est, par exemple, donné dans [Ste92]. Finalement, concernant la compatibilité stable, l'algorithme issue de la linguistique [BK07] peut être directement spécialisé pour le problème de reconstruction de super-arbre.

Nous nous plaçons maintenant dans le contexte de la compatibilité stable, le problème de reconstruction de super-arbre est défini à partir d'un ensemble $\mathcal{T}_1, \ldots, \mathcal{T}_k$ d'arbres et peut être modélisé par une contrainte *extended-tree*$(1, 1, \texttt{VER})$ telle que :

- le graphe $\mathcal{G}$ associé à la contrainte est un graphe complet orienté $(\mathcal{V}, \mathcal{E})$ tel que l'ensemble des sommets est défini par $\mathcal{V} = \mathcal{N}(\mathcal{T}_1) \cup \cdots \cup \mathcal{N}(\mathcal{T}_k)$ et l'ensemble des arcs par $\mathcal{E} = \{(u, v) \mid u, v \in \mathcal{V}\}$, où $\mathcal{N}(\mathcal{T})$ est l'ensemble des sommets de l'arbre $\mathcal{T}$;
- le graphe de précédence $\mathcal{G}_{prec} = (\mathcal{V}, \mathcal{E}_{prec})$ est directement donné à partir des arbres $\mathcal{T}_1, \ldots, \mathcal{T}_k$;
- le graphe d'incomparabilité est généré à partir des sommets incomparables dans chaque arbre

$\mathcal{T}_1, \ldots, \mathcal{T}_k$;
- toutes les feuilles des arbres $\mathcal{T}_1, \ldots, \mathcal{T}_k$ doivent rester des feuilles dans toute solution donc, leur demi-degré intérieur est fixé à zéro : $\texttt{VER}[i].\texttt{D} = 0$. Tous les autres sommets de $\mathcal{V}$ ont leur degré contraint à $\texttt{VER}[i].\texttt{D} \in [1, 2]$ si un super-arbre binaire est recherché, et $\texttt{VER}[i].\texttt{D} \in [1, n-1]$ dans tous les autres cas, où $n = |\mathcal{V}|$.

Une heuristique de choix de variables par taille de domaine croissante est utilisée lors de chaque réveil de la contrainte *extended-tree* (c.-à-d., lors de chaque instanciation décidée par le résolveur), et l'heuristique de choix de valeur privilégie, pour un sommet v_i donné, le père v_j de v_i tel que il existe un plus court chemin de v_i à v_j dans le graphe de précédence $\mathcal{G}_{prec}$. Cette dernière heuristique est basée sur l'intuition que plus la distance entre v_i et v_j est grande dans $\mathcal{G}_{prec}$, plus les chances de satisfaire toutes les contraintes de précédence mises en jeu sur un chemin de v_i à v_j sont faibles.

Le tableau 8.1 compare les performances de notre modèle en Choco (vérifiant la compatibilité stable), avec le modèle Choco introduit dans [GPSW03] (vérifiant la compatibilité forte). Ce dernier modèle est disponible à l'adresse suivante `http://www.dcs.gla.ac.uk/~pat/supertrees/`. Pour une instance donnée, la colonne « satisfiabilité » indique l'existence d'un super-arbre, et, entre parenthèses, nous précisons s'il existe ou non un super-arbre binaire. Précisons que la mention « n/a » est ajouté lorsqu'une colonne n'a pas de sens dans le contexte considéré. Au niveau des instances choisies, il y a tout d'abord deux arbres S_1 et S_2, issus de l'étude des araignées disponible dans la base de données TreeBase (voir `http://www.treebase.org/`, étude S1x6x97c14c42c30). Pour ces instances, il faut noter que le fait qu'un des deux arbres ne soit pas binaires implique qu'il n'existe pas de super-arbre binaire les unifiant. Nous nous sommes intéressé ensuite à deux arbres C_1 et C_2 issus d'une base de données étudiant les chats extraite de journaux spécifiques. Comme pour les araignées, l'existence d'un arbre non binaire implique qu'il n'existe pas de super-unifiant ces deux arbres. Enfin, nous avons étudié 129 espèces d'oiseaux marins au travers des arbres A à G. Ces instances ont été extraites à partir d'un journal d'ornithologie [KP02]. On peut remarquer que seuls les arbres A, E et G ne sont pas binaires ce qui implique que toute combinaison les contenant ne peut pas être unifiée en un super-arbre binaire.

TAB. 8.1: Comparaison des deux approches de programmation par contraintes connues pour le problème phylogénétique de reconstruction de super-arbres.

instance	#espèces	approche	$\lvert\mathcal{G}\rvert$	satisfiabilité	#échecs	temps(ms)
$S_1 + S_2$	17	*extended-tree*	18	yes (no)	0	48
		Prosser	n/a	yes	1	155
$C_1 + C_2$	23	*tree*	26	yes (no)	0	75
		Prosser	n/a	yes	2	254
$A + B$	30	*extended-tree*	52	yes (no)	0	302
		Prosser	n/a	yes	32	648
$A + C$	34	*extended-tree*	63	yes (no)	0	406
		Prosser	n/a	no	0	810
$A + D$	47	*extended-tree*	85	yes (no)	0	398
		Prosser	n/a	yes	0	1972
$A + E$	95	*extended-tree*	191	yes (no)	0	10393
		Prosser	n/a	n/a	n/a	out of memory
$A + F$	33	*extended-tree*	58	yes (no)	0	127
		Prosser	n/a	yes	4	710
$A + G$	49	*extended-tree*	82	yes (no)	0	409
		Prosser	n/a	yes	9	2135
$B + C$	32	*extended-tree*	65	no (no)	0	32
		Prosser	n/a	no	1770	12866

instance	#espèces	approche	$\lvert\mathcal{G}\rvert$	satisfiabilité	#échecs	temps(ms)
$B+D$	43	*extended-tree*	85	yes (yes)	0	301
		Prosser	n/a	yes	3	1683
$B+E$	95	*extended-tree*	195	no (no)	0	892
		Prosser	n/a	n/a	n/a	out of memory
$B+F$	33	*extended-tree*	62	yes (no)	0	144
		Prosser	n/a	yes	0	606
$B+G$	44	*extended-tree*	81	yes (no)	0	1440
		Prosser	n/a	yes	35	1765
$C+D$	52	*extended-tree*	101	yes (no)	0	630
		Prosser	n/a	yes	0	2979
$C+E$	96	*extended-tree*	203	yes (no)	0	27180
		Prosser	n/a	n/a	n/a	out of memory
$C+F$	38	*extended-tree*	74	yes (no)	0	393
		Prosser	n/a	yes	3	979
$C+G$	49	*extended-tree*	93	yes (no)	0	1530
		Prosser	n/a	no	0	2371
$D+E$	104	*extended-tree*	220	no (no)	0	1126
		Prosser	n/a	n/a	n/a	out of memory
$D+F$	46	*extended-tree*	91	yes (yes)	0	630
		Prosser	n/a	yes	4	1776
$D+G$	59	*extended-tree*	112	yes (no)	0	910
		Prosser	n/a	no	35	4403
$E+F$	96	*extended-tree*	199	no (no)	0	1035
		Prosser	n/a	n/a	n/a	out of memory
$E+G$	100	*extended-tree*	211	no (no)	0	1211
		Prosser	n/a	n/a	n/a	out of memory
$F+G$	43	*extended-tree*	83	no (no)	0	62
		Prosser	n/a	n/a	n/a	$> 5 \cdot 10^5$
$A+C+E$	99	*extended-tree*	215	yes (no)	0	49224
		Prosser	n/a	n/a	n/a	out of memory
$A+B+D+F$	72	*extended-tree*	139	yes (no)	0	8139
		Prosser	n/a	yes	59	4811
$A+B+D+G$	82	*extended-tree*	157	no (no)	0	347
		Prosser	n/a	n/a	n/a	out of memory
$A+C+D+F$	76	*extended-tree*	150	yes (no)	0	8690
		Prosser	n/a	no	0	2553
$A+C+D+G$	86	*extended-tree*	168	yes (no)	0	12650
		Prosser	n/a	n/a	n/a	out of memory

Notre modélisation actuelle ne prend pas en compte les symétries contrairement à l'approche proposée par Prosser dans [GPSW03]. En effet, dans ce problème les sommets internes d'un arbre donné ont tous un unique sommet père. Cependant, lorsque l'on tente d'unifier plusieurs arbres certains sommets internes ont des rôles totalement symétriques. Par exemple, pour les deux arbres donnés dans la figure 8.1 page 110, nous trouvons 36 super-arbres alors qu'il n'en existe que 8 qui ne sont pas symétriques par rapport à la compatibilité stable. Nous avons néanmoins ajouté une procédure permettant de contacter les branches des arbres telles que chaque sommets ne possède qu'un unique prédécesseurs. Ceci, permet par exemple d'obtenir 10 super-arbres au lieu des 36 initiaux sur l'exemple de la figure 8.1 page 110.

OSPMN instances	*extended-tree*			*DomReachability+Path* [Que06]	
	réveils	échecs	temps (ms)	échecs	temps (ms)
$SPMN\_22$	7	0	52	5	110
$SPMN\_22\_full$	3	0	36	0	70
$SPMN\_52b$	20	0	1115	6	920
$SPMN\_52\_full$	6	0	562	0	580
$SPMN\_52order\_a$	6	0	592	0	500
$SPMN\_52order\_b$	1	0	17	4	280

Tab. 8.2 – Résultats des expérimentations du problème OSPMN introduites dans [Que06].

Cependant, notre modélisation ne met en jeu que $\Theta(\ell)$ variables, où ℓ est le nombre de feuilles dans les arbres donnés en entrée, alors que le modèle de Prosser nécessite $\Theta(\ell^2)$ variables. Les conséquences en terme de temps de calcul et de consommation de mémoire sur les instances les plus grandes sont clairement visibles dans le tableau 8.1 page 111.

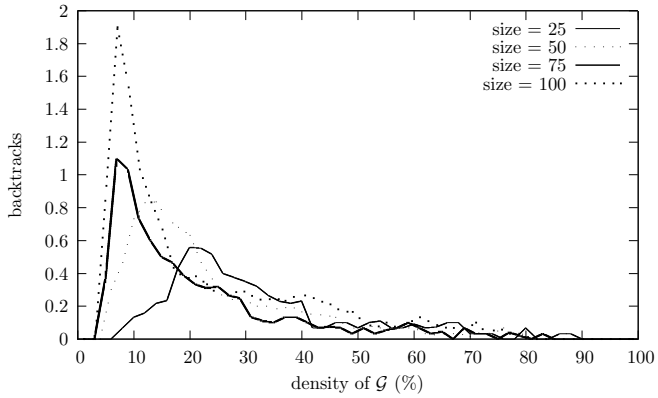
En ce qui concerne les algorithmes spécialisés, nous pouvons retenir les approches proposées dans [BK07, NW96] qui ont ont un temps de calcul de l'ordre de $O(\ell^2)$. Ces algorithmes dominent, sans surprise, les approches basées sur la programmation par contraintes (nous avons relevé sur les instances du tableau 8.1 page 111 un temps d'exécution maximum de l'ordre de 100ms). Cependant, ces algorithmes ne fournissent aucune flexibilité en comparaison des approches utilisant la programmation par contraintes. De plus, il est de plus en plus nécessaire de prendre en compte un certain nombre de restrictions pour ce problème. On retiendra en particulier la notion de date de divergence, d'espèces privilégiées, etc. Chacune de ces restrictions peut être prise en compte dans notre modèle sans remettre en question le modèle précédant alors que ce n'est évidemment pas le cas pour un algorithme dédié.

8.2 Couverture par un chemin sous contraintes de précédence

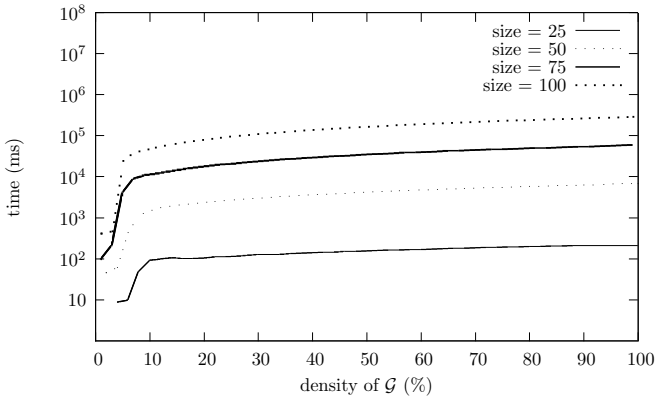
Nous évaluons maintenant la contrainte *extended-tree* au travers d'un problème de couverture de graphe par un chemin sous contraintes de précédence. Ce problème est une restriction du problème des k chemins sommets-disjoints [GJ78, page 217], prenant en compte un ordre partiel entre certain sommets du graphe à couvrir. dans la suite ce problème sera appelé OSPMN pour « Ordered Simple Path problem with Mandatory Nodes ». Un intérêt de s'intéresser à cette restriction est de pouvoir comparer notre approche utilisant la contrainte *extended-tree* avec une modélisation spécifique du problème proposée dans [Que06]. Cependant, dans ce contexte, nous avons restreint notre étude à la recherche d'un unique chemin compatible.

Le problème OSPMN est modélisé par une contrainte *extended-tree* dont le graphe associé $\mathcal{G}$ est le graphe à couvrir tel qu'une boucle a été ajoutée sur chaque sommet. Ensuite, l'ensemble des contraintes de précédence est contenu dans une composante connexe du graphe de précédence $\mathcal{G}_{prec}$. Ces contraintes sont telles que chaque sommet obligatoirement couvert succède le sommet étant l'origine du chemin et précède le dernier sommet du chemin. d'autre part, s'il existe un ordre imposé entre deux sommets obligatoires alors il existe un arc les liant dans $\mathcal{G}_{prec}$. Tous les sommets obligatoires sont isolés dans $\mathcal{G}_{prec}$. Pour compléter notre modèle, nous utilisons une heuristique de choix de variable construisant de manière incrémentale le chemin couvrant le graphe. Plus précisément, l'heuristique choisie comme nouvelle variable à instancier, la variable correspondant à la valeur choisie à l'étape précédente. En terme d'arcs dans le graphe, on peut l'interpréter comme le fait que si un arc (v_i, v_j) de $\mathcal{G}$ est forcé à une étape donné alors, un arc sortant de v_j sera sélectionné à l'étape suivante.

Le tableau 8.2 montre que notre modèle se comporte de manière très acceptable, en particulier au niveau du nombre d'échecs pour trouver une solution, par rapport au précédent modèle [Que06]. Précisons seulement que ce modèle est basé sur une approche utilisant les contraintes *domReachability*, *noCycle*, et *allDifferent* implémentées dans le résolveur *Gecode*(*CP*(*Graph*)). Au niveau des temps de calcul, les deux approches sont quasiment similaires.



(a) Évolution du nombre d'échecs moyen en fonction de la densité du graphe $\mathcal{G}$.



(b) Évolution du temps de calcul en fonction de la densité du graphe $\mathcal{G}$.

FIG. 8.3 – Étude du comportement de la contrainte *extended-tree* sur le problème de chemin Hamiltonien.

8.3 Couverture par un chemin Hamiltonien

Notre modèle pour le problème de chemin Hamiltonien utilise toujours la contrainte *extended-tree* pour laquelle on force que le degré de chaque sommet du graphe $\mathcal{G}$ associé soit fixé à 1, excepté pour le sommet origine du chemin. D'autre part, des contraintes de précédences sont ajoutés de sorte que l'origine du chemin précède tous le sommets du graphe $\mathcal{G}$, et tous les sommets de $\mathcal{G}$ précèdent le sommet destination du chemin. Pour chaque densité dans l'ensemble $\{0\%, 10\%, \ldots, 100\%\}$, un total de 50 graphes pour chaque taille parmi $\{25, 50, 75, 100\}$ sont générés aléatoirement. De plus, pour chaque graphe généré, les sommets origine et destination sont tirés aléatoirement parmi tous les sommets du graphe.

La figure 8.3(a) montre que les instances les plus difficiles (selon le critère du nombre d'échecs) concerne des graphes ayant une densité comprise entre 8% et 30%. Intuitivement, ce résultat n'est pas surprenant. En effet, plus le graphe initial est dense, plus la probabilité d'existence d'un chemin Hamiltonien est grande [KS83, Pós76]. Si le graphe contient moins d'arcs que de sommets, la probabilité d'existence d'un tel chemin est nulle, alors que, dans le cas d'un graphe complet, l'existence d'un chemin Hamiltonien est garantie. À partir de cette observation, la partie gauche de l'intervalle $[8\%, 30\%]$ de la figure 8.3(a) constitue une zone de problèmes « faciles » à résoudre (relativement au nombre d'échecs). La partie droite de cet intervalle est elle aussi composée de problèmes généralement facile à résoudre car, comme nous l'avons dit, la probabilité d'existence d'un chemin Hamiltonien augmente avec la densité. Cependant, il peut paraitre étrange de n'obtenir qu'une moyenne de 1.9 échecs dans le pire cas alors que le problème est connu comme NP-complet. Pourtant, ce chiffre ne fait que mettre en lumière la difficulté de

générer des graphes pour lesquels le problème de chemin Hamiltonien est dur. Néanmoins, nous sommes parfaitement conscient des limites de notre approche. Par exemple, les graphes ayant la topologie du graphe de Tutte [Tut46] [2] nécessitent autour d'une centaine d'échecs avant d'obtenir une réponse sur la non-existence d'un chemin Hamiltonien dans de tels graphes.

La figure 8.3(b) confirme les complexités théoriques annoncées dans le tableau 5.2 de la section 5.8. Cependant, on peut remarquer que l'intervalle de densités $[8\%, 30\%]$, que venons de décrire comme une zone où les graphes générés sont généralement plus complexes à résoudre en terme du nombre d'échecs, n'est pas visible sur le graphique du temps de calcul. L'explication de ce comportement repose sur le fait que le temps d'éxécution dépend directement de la densité du graphe et que le nombre moyen d'échecs constatés dans la zone de densité $[8\%, 30\%]$ ne compense pas l'augmentation du temps de calcul liée à la densité dans cet intervalle. En conclusion, cette expérimentation est une fois encore un plaidoyer en faveur d'une implémentation totalement incrémentale de la contrainte *extended-tree* (voir chapitre 7 page 99). En effet, une approche totalement incrémentale permettrait de s'abstraire partiellement du facteur densité et donc de ne pas avoir des temps de calcul prohibitifs alors que l'existance d'un chemin Hamiltonien est quasiment garantie.

[2] Le graphe de emphTutte est graphe cubique, 3-connexe, n'admettant pas de chemin Hamiltonien.

Conclusion

L'objectif de cette thèse était de proposer une étude des contraintes de partitionnement de graphe. Nous avons volontairement limité le cadre de cette étude aux problèmes de satisfaction de contraintes, car, si la majorité des travaux concernent l'aspect optimisation des contraintes de partitionnement, le côté purement structurel a trop souvent été négligé. En effet, la compréhension des propriétés de graphes (composantes fortement connexes, sommets dominant, etc) mises en jeu constitue un point fondamental pour la prise en compte de restrictions supplémentaires.

Tout d'abord, nous nous sommes intéressé à l'étude de contraintes globales pour le partitionnement de graphe par des arbres, tant dans le contexte des graphes non-orientés que dans le contexte des graphes orientés. En particulier, nous avons montré qu'il était possible d'atteindre un niveau de consistance maximal dans le cas de graphes non-orientés. En ce qui concerne le cas orienté, nous avons montré que, suivant la définition retenue pour la notion d'arbre (doit-il contenir ou non au moins deux sommets ?), il était aussi possible d'atteindre un niveau de consistance maximal dès lors que le nombre minimum de sommets dans un arbre n'est pas contraint.

Pourtant, ces contraintes de partitionnement seules n'ont qu'un intérêt limité en pratique. En effet, la plupart des applications pouvant se modéliser sous la forme d'un problème de partitionnement de graphe nécessitent l'introduction de restrictions portant sur la topologie des partitions valides. Dans ce contexte, nous avons étudié un certain nombre de restrictions classiques en nous appuyant, dans la mesure du possible, sur les propriétés structurelles mises en avant pour le problème « pur » de partitionnement de graphe par des arbres. En particulier, nous avons retenu la prise en compte d'un ordre partiel entre les sommets du graphe à partitionner (au travers de contraintes de précédence et de précédence conditionnelles), la notion de contraintes d'incomparabilité entre certains sommets du graphe, et la restriction sur le nombre de prédécesseurs possibles pour chaque sommet dans toute partition valide. En outre, nous nous sommes aussi attaché à étudier les interactions fortes existantes entre chacune de ces restrictions lorsqu'elles sont combinées.

Finalement, le dernier volet de cette thèse a concerné un aspect bien souvent passé sous silence : l'implémentation pratique (d'un point de vue ingénierie) de telles contraintes globales. En effet, partant du constat que la plupart des contraintes globales dépassent rarement le stade d'une implémentation spécifique à un résolveur de contraintes particulier (dont le code n'est souvent pas disponible), nous nous sommes interrogé sur les moyens de diffuser « plus rapidement » ces dernières. Le déploiement et le maintient d'une nouvelle contrainte globale sur les résolveurs existant est un travail difficilement envisageable en pratique. Cependant, nous avons montré que dans le contexte des contraintes globales modélisables par des graphes, il était possible de découpler totalement la contrainte du résolveur, à la condition que les algorithmes de graphes mis en jeu dans la contrainte puissent être totalement dynamiques. Ainsi, nous avons illustré cette nouvelle manière d'envisager l'implémentation des contraintes globales de graphes au travers d'une contrainte de partitionnement par des arbres : la contrainte *tree*.

Perspectives

Au niveau théorique, il reste encore beaucoup de pistes à explorer dans la lignée des travaux que cette thèse vient de détailler. Tout d'abord, le principale étude repose sur l'unification des quatre principales contraintes de partitionnement de graphe (arbre, chemin, cycle et map), ainsi que des contraintes additionnelles permettant de les utiliser dans des applications pratiques. Cependant, il reste aussi de nombreuses pistes à explorer dans le contexte des contraintes d'arbre et en particulier autour du nombre d'arbres

propres (c.-à-d., des arbres contenant plus d'un sommet) qui prend tout son sens dans les problèmes de couverture partielle de graphes. En effet, nous pensons qu'un filtrage complet du graphe associé à la contrainte *proper-tree* est possible lorsque `NPROP` est fixé à 1. L'étude de l'interaction entre les variables `NTREE` et `NPROP` au sein de la contrainte *extended-tree* semble aussi être un problème très intéressant, puisque l'interaction entre ces deux paramètres pourrait permettre de s'attaquer, par exemple, à une relaxation du problème de super-arbre en phylogénie ou de tournées de véhicules en logistique. Par exemple, en phylogénie, dans de nombreux cas, l'ensemble des arbres donnés en entrée n'est pas unifiable en un unique arbre. Cependant, l'interaction des variables `NTREE` et `NPROP` pourrait permettre de fournir une solution maximisant le nombre d'espèces unifiées. Pour cela, une simple relaxation des domaines des variables `NTREE` et `NPROP` (fixés à 1 dans la modélisation initiale du problème) en $\texttt{NTREE} = [0; n]$ et $\texttt{NPROP} = 1$, permet de maximiser le nombre d'espèces unifiées en recherchant la valeur minimale pour `NTREE` satisfaisant la contrainte *extended-tree*.

Au niveau des applications pratiques et de l'implémentation, nous continuerons l'intégration les contraintes additionnelles au sein de la contrainte *extended-tree* « portable ». En parallèle, nous ajouterons dans les restrictions additionnelles, la notion de fenêtres temporelles qui nous permettra de modéliser pleinement des problèmes de planification de missions et/ou de tournées de véhicules.

Questions spécifiques

Finalement, nous conclurons cette thèse par une liste de questions et de problèmes spécifiques rencontrés au cours de notre étude :

- L'évaluation d'une borne supérieure sur le nombre d'arbres propres (`NPROP`) pouvant couvrir un graphe orienté. Plus précisément, est-ce que l'évaluation de cette borne relève d'un problème polynomial ou non ?
- La complexité d'un algorithme de filtrage atteignant l'arc-consistance généralisée pour une contrainte d'arbre restreinte par un ensemble d'incomparabilités entre les sommets du graphe à partitionner ;
- L'existence d'algorithme « efficace » pour le problème de la détection des arcs d'un réseau n'appartenant à aucun flot maximum. Un tel algorithme serait particulièrement intéressant dans le contexte de la contrainte de chemin que nous avons présenté ;
- Bien que nous ayons montré que propager l'arc-consistance généralisée pour la contrainte *extended-tree* relève d'un problème NP-complet lorsqu'on ne fait aucune hypothèse sur la structure des contraintes de précédence, d'incomparabilité et de degré, une question ouverte réside dans le fait de savoir si certaines configurations des contraintes de précédence et d'incomparabilité n'autorisent pas une propagation complète.
- Au niveau de l'implémentation, nous avons clairement démontré l'intérêt d'implémenter les algorithmes maintenant des propriétés de graphes sous une forme totalement dynamique. Même si l'état de l'art actuel nous fournit beaucoup d'algorithmes basé sur le retrait ou l'ajout d'un unique arc dans le graphe, dans le contexte de la programmation par contraintes, le réveil d'une contrainte globale modélisant un problème de graphe est très souvent lié au retrait d'un ensemble d'arcs dans le graphe associé à la contrainte. Un traitement individuel du retrait de chaque arc, même par un algorithme efficace, est toujours possible mais semble peu pertinent. En effet, les retraits d'arcs ne relèvent jamais d'un phénomène épart mais plutôt d'une modification locale ayant pour but de restreindre (ou d'enrichir) le voisinage d'un sommet (par ex., l'instanciation d'une variable conduit à retirer tous les arcs sortant d'un sommet excepté celui correspondant à la valeur affectée). Cette remarque faite, on peut alors s'interroger sur la recherche d'algorithmes incrémentaux permettant de gérer, non plus un ajout ou un retrait d'arc, mais un ensemble d'ajouts ou de retraits relatif à un unique sommet.

Bibliographie

[AB90] A. Aggoun and N. Beldiceanu. Time Stamps Techniques for the Trailed Data in Constraint Logic Programming Systems. In *SPLT*, pages 487–510, 1990.

[ADK+03] E. Althaus, D. Duchier, A. Koller, K. Mehlhorn, J. Niehren, and S. Thiel. An Efficient Graph Algorithm for Dominance Constraints. 48(1) :194–219, 2003. Special Issue of SODA 2001.

[AR90] S. Rao Arikati and C. Pandu Rangan. Linear algorithm for optimal path cover problem on interval graphs. *Inf. Process. Lett.*, 35(3) :149–153, 1990.

[ASSU81] A.V. Aho, Y. Sagiv, T.G. Szymanski, and J.D. Ullman. Inferring a tree from lowest common ancestors with an application to the optimization of relational expressions. *SIAM Journal of Computing*, 10(3) :405–421, 1981.

[BC94] N. Beldiceanu and E. Contejean. Introducing global constraint in CHIP. *Mathl. Comput. Modelling*, 20(12) :97–123, 1994.

[BCDP07] N. Beldiceanu, M. Carlsson, S. Demassey, and T. Petit. Global Constraint Catalog : Past, Present and Future. *Constraints*, 12(1) :21–62, 2007.

[BDMN04] M. Bodirsky, D. Duchier, S. Miehle, and . Niehren. A new algorithm for normal dominance constraints. In *Proceedings of SODA'04*, pages 59–67, 2004.

[BEGS02] Olaf R.P. Bininda-Emonds, J.L. Gittleman, and M.A. Steel. The (super)tree of life : Procedures, problems, and prospects. *Annual Reviews of Ecological Systems*, 33 :265–289, 2002.

[Bel01] N. Beldiceanu. Pruning for the Minimum Constraint Family and for the Number of Distinct Values Constraint Family. In *Principles and Practice of Constraint Programming CP'01*, pages 211–224. Springer-Verlag, 2001.

[Ber70] C. Berge. *Graphes et Hypergraphes*. Dunod, Paris, 1970.

[BFL05] N. Beldiceanu, P. Flener, and X. Lorca. The *tree* constraint. In *International Conference on Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems (CP-AI-OR'05)*, volume 3524 of *LNCS*, pages 64–78. Springer-Verlag, 2005.

[BFL06] N. Beldiceanu, P. Flener, and X. Lorca. Combining tree partitioning, precedence, incomparability, and degree constraints, with an application to phylogenetic and ordered-path problems. Technical Report 2006-020, Department of Information Technology, Uppsala University, Sweden, April 2006.

[BHH+05] C. Bessière, E. Hebrard, B. Hnich, Z. Kızıltan, and T. Walsh. The *range* and *roots* Constraints : Specifying Counting and Occurrence Problems. In *IJCAI-05*, pages 60–65, 2005.

[BHHW04] C. Bessière, E. Hebrard, B. Hnich, and T. Walsh. The Complexity of Global Constraints. In *AAAI*, pages 112–117, 2004.

[BK07] M. Bodirsky and M. Kutz. Determining the consistency of partial tree descriptions. *Artificial Intelligence*, 171 :185–196, 2007.

[BKL06] N. Beldiceanu, I. Katriel, and X. Lorca. Undirected forest constraints. In Barbara Smith and Chris Beck, editors, *International Conference on Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems (CP-AI-OR'06)*, Lecture Notes in Computer Science, Cork, Ireland, June 2006. Springer Verlag.

[BL07] N. Beldiceanu and X. Lorca. Necessary condition for path partitioning constraints. In *International Conference on Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems (CP-AI-OR'07)*, volume 4510 of *LNCS*. Springer-Verlag, 2007.

[Bou99] E. Bourreau. *Traitement de contraintes sur les graphes en programmation par contraintes*. PhD thesis, University of Paris 13, France, 1999.

[BPR05] N. Beldiceanu, T. Petit, and G. Rochart. Bounds of Graph Characteristics. In P. van Beek, editor, *CP'05*, volume 3709 of *LNCS*, pages 742–746. Springer-Verlag, 2005.

[BQ64] M.L. Balinski and R.E. Quandt. On an Integer Program for Delivery Problem. *Operations Research*, 12(2) :300–304, 1964.

[BvH03] C. Bessière and P. van Hentenryck. To Be or Not to Be... a Global Constraint. In *Principles and Practice of Constraint Programming CP'03*, pages 789–794, 2003.

[Cay89] A. Cayley. A theorem on trees. *Quart. J. Math.*, 23 :376–378, 1889.

[CB04] H. Cambazard and E. Bourreau. Conception d'une contrainte globale de chemin. *JNPC*, pages 107–120, 2004. In French.

[CHK01] K. Cooper, T. Harvey, and K. Kennedy. A Simple, Fast Dominance Algorithm. *Software Practice and Experience*, 4 :1–10, 2001.

[Cou97] B. Courcelle. *The Expression Of Graph Properties And Graph Transformations In Monadic Second-Order Logic*. World Scientific, New-Jersey, London, 1997.

[DDD05] G. Dooms, Y. Deville, and P.E. Dupont. CP(Graph) : Introducing a Graph Computation Domain in Constraint Programming. In *Principles and Practice of Constraint Programming (CP'05)*, volume 3709 of *LNCS*, pages 211–225, 2005.

[Dil50] R.P. Dilworth. A decomposition theorem for partially ordered sets. *Annals of Mathematics*, 51 :161–166, 1950.

[DK06] G. Dooms and I. Katriel. The *minimum spanning tree* Constraint. In *Principles and Practice of Constraint Programming CP'06*, pages 152–166. Springer-Verlag, 2006.

[DvHS+88] M. Dincbas, P. van Hentenryck, H. Simonis, A. Aggoun, T. Graf, and F. Berthier. The Constraint Logic Programming Language CHIP. In *FGCS*, pages 693–702, 1988.

[Edm65] J. Edmonds. Path, trees, and flowers. *Can. J. Math.*, 17 :449–467, 1965.

[EGI97] D. Eppstein, Z. Galil, and G. Italiano. *Dynamic graph algorithms*. CRC Press, 1997.

[FHW80] S. Fortune, J.E. Hopcroft, and J. Wyllie. The Directed Subgraph Homeomorphism Problem. *Theor. Comput. Sci.*, 10 :111–121, 1980.

[GJ78] M.R. Garey and D.S. Johnson. *Computers and Intractability, A Guide to the Theory of NP-Completeness*. Freeman, New York, 1978.

[GM85] M. Gondran and M. Minoux. *Graphes et algorithmes*. Eyrolles, Paris, 2nd edition, 1985.

[GPSW03] I.P. Gent, P. Prosser, B. Smith, and W. Wei. Supertree construction using constraint programming. In Francesca Rossi, editor, *CP 2003*, volume 2833 of *LNCS*, pages 837–841. Springer-Verlag, 2003.

[Gue07] C. Guettier. Solving Planning and Scheduling Problems in Network based Operations. In *Principles and Practice of Constraint Programming CP'07*. To appear, 2007.

[HM07] Pascal Van Hentenryck and Laurent Michel. Growing COMET. In Frédéric Benhamou, Narendra Jussien, and Barry O'Sullivan, editors, *Trends in Constraint Programming*, chapter 17, pages 291–297. ISTE, London, UK, May 2007.

[JB00] N. Jussien and V. Barichard. The PaLM system : explanation-based constraint programming. In *Proceedings of TRICS : Techniques foR Implementing Constraint programming Systems, a post-conference workshop of CP 2000*, pages 118–133, Singapore, 2000.

[JKK+99] U. Junker, S. E. Karisch, N. Kohl, B. Vaaben, T. Fahle, and M. Sellmann. A Framework for Constraint Programming Based Column Generation. In *Principles and Practice of Constraint Programming CP'99*, pages 261–274. Springer-Verlag, 1999.

[KH05] I. Katriel and P. Van Hentenryck. Maintaining Longest Paths in Cyclic Graphs. In *Principles and Practice of Constraint Programming CP'05*, pages 358–372. Springer-Verlag, 2005.

[KH06] L.G. Kaya and J.N. Hooker. A filter for the circuit constraint. In *Principles and Practice of Constraint Programming CP'06*, volume 4204, pages 706–710. Springer-Verlag, 2006.

[KP02] M. Kennedy and R.D.M. Page. Seabird supertrees : Combining partial estimates of procellariiform phylogeny. *The Auk, A Quarterly Journal of Ornithology*, pages 88–108, 2002.

[KS83] J. Komlós and E. Szemerédi. Limit distribution for the existence of a Hamilton cycle in a random graph. *Discrete Mathematics*, 43 :55–63, 1983.

[KT05] I. Katriel and S. Thiel. Complete Bound Consistency for the Global Cardinality Constraint. *Constraints*, 10(3) :191–217, 2005.

[Lau78] J-L. Laurière. A Language and a Program for Stating and Solving Combinatorial Problems. *Artificial Intelligence*, 10 :29–127, 1978.

[LT79] T. Lengauer and R.E. Tarjan. A fast algorithm for finding dominators in a flowgraph. *ACM Trans Program. Lang. Syst.*, 1(1) :121–141, 1979.

[Mac77] A.K. Mackworth. Consistency in Networks of Relations. *Artif. Intell.*, 8(1) :99–118, 1977.

[MF85] A.K. Mackworth and E.C. Freuder. The Complexity of Some Polynomial Network Consistency Algorithms for Constraint Satisfaction Problems. *Artif. Intell.*, 25(1) :65–74, 1985.

[Mon74] U. Montanari. Networks of constraints : Fundamental properties and applications to picture processing. *Inf. Sci.*, 7 :95–132, 1974.

[MV80] S. Micali and V. V. Vazirani. An $\mathcal{O}(\sqrt{|V|} \cdot |E|)$ algorithm for finding maximum matching in general graphs. In *FOCS 1980*, pages 17–27, New York, 1980. IEEE.

[NW96] M. Pyng Ng and N.C. Wormald. Reconstruction of rooted trees from subtrees. *Discrete Applied Mathematics*, 69 :19–31, 1996.

[Pós76] L. Pósa. Hamiltonian circuits in random graphs. *Discrete Mathematics*, 14 :359–364, 1976.

[PS02] G. Pesant and P. Soriano. An Optimal Strategy for the Constrained Cycle Cover Problem. *Ann. Math. Artif. Intell.*, 34(4) :313–325, 2002.

[PU06a] P. Prosser and C. Unsworth. A connectivity constraint using bridges. In *ECAI*, pages 707–708, 2006.

[PU06b] P. Prosser and C. Unsworth. Rooted tree and spanning tree constraints. Technical Report cppod-13-2006, CP Pod research group, May 2006.

[Pug94] J.-F. Puget. A C++ Implementation of CLP. In *Second Singapore International Conference on Intelligent Systems (SPICIS)*, pages 256–261, Singapore, November 1994.

[Que06] L. Quesada. *Solving Constrained Graph Problems Using Reachability Constraints Based on Transitive Closure and Dominators*. PhD thesis, Université catholique de Louvain, Louvain-la-Neuve, Belgium, 2006.

[QvRDC06] L. Quesada, P. van Roy, Y. Deville, and R. Collet. Using dominators for solving constrained path problems. In *PADL'06*, volume 3819 of *LNCS*, pages 73–87, 2006.

[Rég94] J.-C. Régin. A filtering algorithm for constraints of difference in CSP. In *AAAI'94*, pages 362–367, 1994.

[Rég96] J.-C. Régin. Generalized arc consistency for global cardinality constraint. In *AAAI'96*, pages 209–215, 1996.

[Rég99] J.-C. Régin. The symmetric AllDiff constraint. In *IJCAI-99*, pages 420–425, 1999.

[RLJ07] G. Richaud, X. Lorca, and N. Jussien. A portable and efficient implementation of global constraints : the tree constraint case. In S. Abreu and V. S. Costa, editors, *Proceedings of the 7th International Colloqium on Implementation of Constraint and Logic Programming Systems*, Porto, Portugal, september 2007.

[Sch99] Christian Schulte. Comparing Trailing and Copying for Constraint Programming. In Danny De Schreye, editor, *Proceedings of the Sixteenth International Conference on Logic Programming*, pages 275–289, Las Cruces, NM, USA, 1999. The MIT Press.

[Sch03] A. Schrijver. *Combinatorial Optimization*. Springer, Berlin, 2003.

[SDDR07] P. Schaus, Y. Deville, P. Dupont, and J.-C. Régin. The Deviation Constraint. In *International Conference on Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems (CP-AI-OR'07)*, volume 4510 of *LNCS*, 2007.

[Sel02] M. Sellmann. *Reduction techniques in Constraint Programming and Combinatorial Optimization*. PhD thesis, University of Paderborn, 2002.

[Sel03] M. Sellmann. Cost-based filtering for shortest path constraints. In *Principles and Practice of Constraint Programming CP'03*, volume 2833 of *LNCS*, pages 694–708. Springer-Verlag, 2003.

[Ste92] M. Steel. The complexity of reconstructing trees from qualitative characters and subtrees, 1992.

[Ste03] G. Steiner. On the k-path partition of graphs. *Theor. Comput. Sci.*, 290(3) :2147–2155, 2003.

[Suu74] J.W. Suurballe. Disjoint Paths in a Network. *Networks*, 4 :125–145, 1974.

[Tar72] R.E. Tarjan. Depth-first search and linear graph algorithms. In *SIAM J. Comput.*, volume 1, pages 146–160, 1972.

[Thi04] S. Thiel. *Efficient Algorithms for Constraint Propagation and for Processing Tree Descriptions*. PhD thesis, Universität des Saarlandes, Saarbrücken, 2004.

[Tut46] W. T. Tutte. On Hamiltonian circuits. *Journal of the London Mathematical Society*, 21 :98–101, 1946.

[vH89] P. van Hentenryck. *Constraint satisfaction in logic programming*. MIT Press, Cambridge, MA, USA, 1989.

[Vyg95] J. Vygen. NP-completeness of some edge-disjoint paths problems. *Discrete Appl. Math.*, 61(1) :83–90, 1995.

[YC94] J. Yan and G.J. Chang. The path-partition problem in block graphs. *Inf. Process. Lett.*, 52(6) :317–322, 1994.

Contraintes de Partitionnement de Graphe

Xavier LORCA

Mots-clés : programmation par contraintes, contraintes globales, partitionnement de graphe

Les problèmes combinatoires basés sur le partitionnement de graphe permettent de modéliser un grand nombre d'applications pratiques. On retiendra des exemples aussi variés que la reconstruction de « super-arbres » en phylogénie, la planification de missions, ou la construction de tournées de véhicules en logistique. Ces applications, bien que provenant de domaines différents, peuvent toutes se voir comme un problème de partitionnement de graphe par des patrons tels que des cycles, des chemins, ou des arbres. Cependant, les problèmes pratiques se résument rarement à des problèmes « pur » comme peuvent l'être le problème de chemin Hamiltonien ou le problème des K-chemins disjoints. En effet, ils combinent bien souvent le problème de partitionnement avec un ensemble de restrictions sur la topologie des sommets et des arcs. La diversité des contraintes opérationnelles misent en jeu constitue souvent une limite à leur résolution par des approches considérant de manière séparée le problème de partitionnement et les restrictions supplémentaires imposées.

Cette thèse se concentre sur les problèmes de *satisfaction de contraintes* liés au partitionnement de graphe par des arbres mettant en jeu un certain nombre de restrictions sur la topologie des partitions autorisées. Notre travail se focalise d'une part sur la compréhension des propriétés structurelles inhérentes aux contraintes de partitionnement par des arbres, et d'autre part sur l'étude des *interactions* existantes entre le problème de partitionnement et les restrictions classiques (telles que les relations de précédences ou d'incomparabilités entre les sommets du graphe à partitionner). Nous nous attachons plus particulièrement à montrer comment prendre en compte de manière globale un certain nombre de ces restrictions au sein d'une contrainte de partitionnement de graphes par des arbres. Un autre aspect essentiel porte sur la mise en œuvre d'une telle contrainte : nous montrons en quoi une gestion dynamique des structures de données permet de s'abstraire significativement d'un problème récurrent à la plupart des contraintes globales liées aux graphes : la sensibilité des algorithmes de graphes à la *densité* des graphes manipulés par la contrainte.

Graph Partitioning Constraints

Xavier LORCA

Keywords : Constraint Programming, Global Constraints, Graph Partitioning

Combinatorial problems based on graph partitioning enable to represent many practical applications. Examples based on phylogenetic supertree problem, mission planning, or the routing problems in logistic, perfectly illustrate such applications. Nevertheless, these problems are not based on the same partitioning pattern : Generally, patterns like cycles, paths, or trees are distinguished. Moreover, the practical applications are not often limited to theoretical problems like Hamiltonian path problem, or K-node disjoint paths problems. Indeed, they usually combine the graph partitioning problem with several restrictions related to the topology of nodes and arcs. The diversity of implied constraints in real-life applications is a practical limit to the resolution of such problems by approaches considering the partitioning problem independently from each additional restriction.

This thesis focuses on *constraint satisfaction problems* related to tree partitioning problems enriched by several additional constraints that restrict the possible partitions topology. On the one hand, our study focuses the structural properties of tree partitioning constraints. On the ohter hand, it is dedicated to the *interactions* between the tree partitioning problem and classical restrictions (such as precedence relations or incomparability relations between nodes) involved in practical applications. Precisely, we show how to globally take into account several restrictions within one single tree partitioning constraint. Another interesting aspect of this thesis is related to the implementation of such a constraint. In the context of graph-based global constraints, we show how a fully dynamic management of data structures makes the runtime of filtering algorithms independent of the graph density.