Titre: Title:	Heuristiques de branchement basées sur le dénombrement pour la résolution de problèmes d'arbres de recouvrement contraints
Auteur: Author:	Simon Brockbank
Date:	2014
Type:	Mémoire ou thèse / Dissertation or Thesis
Référence: Citation:	Brockbank, S. (2014). Heuristiques de branchement basées sur le dénombrement pour la résolution de problèmes d'arbres de recouvrement contraints [Master's thesis, École Polytechnique de Montréal]. PolyPublie. https://publications.polymtl.ca/1462/

URL de PolyPublie: PolyPublie URL:	https://publications.polymtl.ca/1462/
Directeurs de recherche: Advisors:	Gilles Pesant, & Louis-Martin Rousseau
Programme: Program:	Génie informatique

UNIVERSITÉ DE MONTRÉAL

HEURISTIQUES DE BRANCHEMENT BASÉES SUR LE DÉNOMBREMENT POUR LA RÉSOLUTION DE PROBLÈMES D'ARBRES DE RECOUVREMENT CONTRAINTS

SIMON BROCKBANK
DÉPARTEMENT DE GÉNIE INFORMATIQUE ET GÉNIE LOGICIEL
ÉCOLE POLYTECHNIQUE DE MONTRÉAL

MÉMOIRE PRÉSENTÉ EN VUE DE L'OBTENTION
DU DIPLÔME DE MAÎTRISE ÈS SCIENCES APPLIQUÉES
(GÉNIE INFORMATIQUE)
JUIN 2014

UNIVERSITÉ DE MONTRÉAL

ÉCOLE POLYTECHNIQUE DE MONTRÉAL

Ce mémoire intitulé :

HEURISTIQUES DE BRANCHEMENT BASÉES SUR LE DÉNOMBREMENT POUR LA RÉSOLUTION DE PROBLÈMES D'ARBRES DE RECOUVREMENT CONTRAINTS

présenté par : BROCKBANK Simon
en vue de l'obtention du diplôme de : Maîtrise ès sciences appliquées
a été dûment accepté par le jury d'examen constitué de :

M. GAGNON Michel, Ph.D., président
M. PESANT Gilles, Ph.D., membre et directeur de recherche
M. ROUSSEAU Louis-Martin, Ph.D., membre et codirecteur de recherche
M. HERTZ Alain, Doct. ès Sc., membre

À ma mère, Louise, qui a fait preuve d'un immense support, qui était toujours là, qui a partagé avec moi les grands moments, joyeux comme difficiles, ma soeur, Sara, qui m'a toujours encouragé, ma copine, Catherine, qui m'a poussé à donner le meilleur de moi-même, à me dépasser, et mes amis, qui ont toujours cru en moi ...

REMERCIEMENTS

J'aimerais remercier le Fond de recherche du Québec — Nature et technologies (FQRNT) pour sa généreuse contribution à notre recherche. J'aimerais également remercier mon directeur de recherche, Gilles, qui a su me guider, me conseiller et m'encourager tout au long de ce projet de recherche. J'aimerais aussi remercier mon co-directeur, Louis-Martin Rousseau, pour son aide précieuse.

RÉSUMÉ

Ce mémoire se concentre sur la programmation par contraintes (CP), une approche puissante pour résoudre des problèmes combinatoires. Notre travail tourne autour de l'un des concepts clés de la CP : les heuristiques de branchement. Cette composante définit comment l'espace de recherche doit être exploré, quelles régions devraient être visitées en premier pour trouver une solution rapidement. Le progrès sur ce sujet est important, étant donné que la CP n'admet toujours pas d'approche générique efficace pour la recherche.

Les heuristiques de branchement basées sur le dénombrement comme maxSD se sont montrées efficaces pour une variété de problèmes de satisfaction de contraintes. Ces heuristiques ont besoin d'un algorithme dédié qui calcule la densité de solution locale pour chaque paire de variable-valeur, pour chaque contrainte, de façon semblable à ce qui a été fait pour les algorithmes de filtrage, pour appliquer l'inférence locale. Cependant, plusieurs contraintes n'ont toujours pas de tel algorithme.

Dans notre travail, nous dérivons un algorithme exact qui, en temps polynomial, calcule la densité de solution pour la contrainte d'arbre de recouvrement, à partir d'un résultat connu sur le nombre d'arbres de recouvrement dans un graphe non orienté. Nous étendons ensuite cet algorithme pour les graphes orientés, ce qui nous permet de calculer la densité de solution pour une contrainte d'anti-arborescence, également en temps polynomial.

Ensuite, nous comparons empiriquement les heuristiques de branchement basées sur ces résultats avec d'autres approches génériques. Tout d'abord, nous utilisons le problème d'arbres de recouvrement de degré contraint, sur des graphes non orientés pour démontrer l'efficacité de notre approche. Ensuite, pour les graphes orientés, nous utilisons le problème de la k-arborescence. Les heuristiques de branchement basées sur le dénombrement se montrent comme des approches très efficaces, autant pour le cas non orienté que pour le cas orienté, trouvant rapidement des solutions avec un minimum de retours en arrière.

ABSTRACT

This Master's thesis focuses on Constraint Programming (CP), a powerful approach to solve combinational problems. Our work revolves around one of the main components of CP : branching heuristics. This component defines how the search space must be explored, which areas should be visited first in order to quickly find a solution to the problem. Advances on this topic are critical, since CP lacks a generic effective search approach.

Counting-based branching heuristics such as maxSD were shown to be effective on a variety of constraint satisfaction problems. These heuristics require that we equip each family of constraints with a dedicated algorithm to compute the local solution density of variable assignments, much as what has been done with filtering algorithms to apply local inference. However, many constraints still lack such an algorithm.

In our work, we derive an exact polytime algorithm to compute solution densities for a spanning tree constraint, starting from a known result about the number of spanning trees in an undirected graph. We then extend the algorithm for directed graphs, which allows us to compute solution densities for a reverse arborescence constraint, also in polytime.

We then empirically compare branching heuristics based on those results with other generic heuristics. First, we use the degree contrained spanning tree, on undirected graphs, to demonstrate the effectiveness of our approach. Then, for the directed graphs, we use the k-arborescence problem. Counting-based branching heuristics prove to be very effective for both the undirected and directed case, finding solutions quickly and without many backtracks.

TABLE DES MATIÈRES

LISTE DES TABLEAUX

LISTE DES FIGURES

LISTE DES SIGLES ET ABRÉVIATIONS

IETF	Internet Engineering Task Force
OSI	Open Systems Interconnection
CP	Constraint Programming (Programmation par contraintes)
CSP	Constraint Satisfaction Problem (Problème de satisfaction de contraintes)
STA	Spanning Tree or Arborescence (Arbre ou arborescence de recouvrement)
maxSD	Maximum Solution Density (densité de solution maximum)
IBS	Impact-based Search

CHAPITRE 1

INTRODUCTION

Les problèmes combinatoires sont riches et variés. Ils permettent de décrire de nombreuses problématiques présentes en industrie ainsi que dans le milieu académique. Que ce soit pour concevoir des réseaux, des horaires ou encore pour ordonnancer des tâches dans une chaîne de montage, les problèmes combinatoires sont présents dans d'innombrables domaines.

Il existe plusieurs approches permettant de s'attaquer à la résolution de problèmes combinatoires NP-difficiles. La programmation en nombres entiers (IP, "Integer Programming") décrit les problèmes comme des problèmes mathématiques d'optimisation, dans lesquels les variables ne peuvent prendre que des valeurs entières. Dans plusieurs cas, les contraintes et les fonctions objectifs sont toutes linéaires. Lorsque c'est le cas, l'approche prend le nom de programmation de nombres entiers linéaire (ILP, "Integer Linear Programming"). Une fois que le problème est modélisé, plusieurs algorithmes existent pour le résoudre de façon exacte. La résolution de problème formulé en programmation linéaire en nombres entiers repose sur le calcul de sa relaxation linéaire, c'est-à-dire de la même formulation à laquelle on permet aux variables de prendre des valeurs fractionnaires. Afin d'obtenir une solution entière, deux méthodes possibles. D'abord le branchement (Branch-and-Bound), l'ajout successif de contraintes de séparation permet d'explorer l'ensemble de l'espace de recherche. Ensuite pour l'ajout de contraintes additionnelles (Cutting Planes) ayant la propriété de séparer la solution courante de la relaxation linéaire des solutions entières réalisables du problème. La combinaison de ces deux approches est aussi courante (Branch-and-Cut).

Les approches de recherche locale utilisent un voisinage, qui consiste à modifier légèrement une solution de base, afin d'obtenir de meilleures solutions au fur et à mesure que la recherche progresse. L'idée est d'explorer l'espace de recherche en utilisant différents mécanismes pour éviter de visiter plusieurs fois la même solution. Deux approches très connues sont le recuit simulé et la recherche taboue. Le recuit simulé fait souvent des changements améliorant directement la solution et rarement des changements moins prometteurs, en utilisant les probabilités. Plus la recherche avance, plus la probabilité de faire un choix peu prometteur diminue. La recherche taboue utilise une liste taboue afin d'interdire les changements déjà faits, pendant un certain temps. Pour toutes les approches, il existe des méthodes de diversifications ou de perturbation, qui consistent à modifier grandement une solution existante, pour aller explorer un espace de recherche plus éloigné du voisinage.

Les solveurs SAT, une autre approche couramment utilisée, décrivent les problèmes sous

la forme d'expressions booléennes à satisfaire. Le problème combinatoire devient alors un problème de décision, dans lequel le solveur tente de donner une valeur aux variables de façon à satisfaire l'expression booléenne ou prouver qu'elle est insatisfaisable.

La programmation par contraintes est une autre approche qui utilise le formalisme du CSP. Un CSP modélise un problème combinatoire à l'aide de variables, de leur domaine ainsi qu'un ensemble de contraintes, qui viennent expliciter les propriétés d'une solution au problème qui doit être résolu. Un CSP est défini comme un triplet $< X, D, C >$ où X représente l'ensemble des variables du problème, D représente l'ensemble des domaines des variables,c'est-à-dire les valeurs que les variables peuvent prendre, et C représente l'ensemble des contraintes sur les variables, qui restreignent les valeurs qu'elles peuvent prendre. Les contraintes indiquent les propriétés que la solution au problème doit respecter pour être admissible. Voici un exemple simple de CSP :

Exemple 1.1 (Exemple de CSP)
Soit un problème de planification d'examens finaux. Pour les 6 cours suivants (variables), les plages horaires suivantes(domaines, numérotées de 1 à 4) sont disponibles.

- $INF4705 \in \{1, 2, 3, 4\}$
- $INF8702 \in \{2, 3\}$
- $INF3710 \in \{1, 3, 4\}$
- $INF1600 \in \{3, 4\}$
- $INF1500 \in \{1, 2, 3\}$
- $INF6101 \in \{1, 3, 4\}$

Les examens ayant au moins un élève en commun doivent avoir une période différente, sinon il y aura conflit. Voici les contraintes représentant ces conflits potentiels :

- $INF4705 \neq INF3710$
- $INF4705 \neq INF1600$
- $INF3710 \neq INF1600$
- $INF8702 \neq INF6101$
- $INF1600 \neq INF1500$
- $INF1600 \neq INF6101$
- $INF1500 \neq INF6101$

Voici un exemple de solution à ce problème :
(où == implique l'affectation d'une valeur à une variable) :

- $INF4705 == 1$
- $INF8702 == 2$
- $INF3710 == 3$
- $INF1600 == 4$

- $INF1500 == 1$
- $INF6101 == 3$

La CP(Constraint Programming) est une approche permettant de résoudre les CSP qui sépare la définition du problème de sa résolution. Une fois modélisé, le problème est résolu au moyen d'un solveur. Le solveur CP peut être divisé en deux couches distinctes qui collaborent. La première est la recherche de solutions, dans laquelle le solveur tente d'attribuer des valeurs aux variables afin de respecter l'ensemble des contraintes définies sur celles-ci. De façon plus précise, le solveur fixe généralement une seule variable à la fois, à une seule valeur, puis fait appel à la deuxième couche : la couche de filtrage. Dans cette couche, le solveur filtre les domaines des variables, un contrainte à la fois. Cela signifie qu'il retire des domaines des variables les valeurs qui, une fois attribuées aux variables restantes, ne respecteraient plus la contrainte. Cette étape est répétée pour l'ensemble des contraintes du problème.

Exemple 1.2 (Exemple de Résolution d'un CSP avec la CP)
Reprenons le CSP de l'exemple 1.1. Les contraintes binaires, pour faciliter le filtrage, peuvent être regroupées. La contrainte globale alldifferent(S), où S est une ensemble, implique que chaque élément de cet ensemble doit avoir une valeur différente. En utilisant les contraintes binaires définies dans l'exemple 1.1, les ensembles suivants peuvent être construits :

- *ensemble* $A = \{INF4705, INF3710, INF1600\}$
- *ensemble* $B = \{INF8702, INF6101\}$
- *ensemble* $C = \{INF1600, INF1500, INF6101\}$

Finalement, définissons les trois contraintes alldifferent impliquant ces ensembles :

- *alldifferent(A), qui implique INF4705* $\neq$ *INF3710* $\neq$ *INF1600*
- *alldifferent(B), qui implique INF8702* $\neq$ *INF6101*
- *alldifferent(C), qui implique INF1600* $\neq$ *INF1500* $\neq$ *INF6101*

Pour résoudre ce CSP, commençons par fixer la variable INF4705 à la valeur 1. Comme INF4705 est impliquée dans la contrainte alldifferent(A), la valeur 1 doit être filtrée des domaines des variables INF3710 et INF1600. Voici donc les variables et leur domaines restants :

- $INF4705 == 1$
- $INF8702 \in \{2, 3\}$
- $INF3710 \in \{3, 4\}$
- $INF1600 \in \{3, 4\}$
- $INF1500 \in \{1, 2, 3\}$
- $INF6101 \in \{1, 3, 4\}$

Fixons maintenant la variable INF8702 à 2. Le filtrage devrait retirer cette valeur du

domaine de la variable INF6101, à cause de la contrainte alldifferent(B), mais elle ne s'y trouve pas, donc son domaine demeure le même. Voici les nouveaux domaines :

- $INF4705 == 1$
- $INF8702 == 2$
- $INF3710 \in \{3, 4\}$
- $INF1600 \in \{3, 4\}$
- $INF1500 \in \{1, 2, 3\}$
- $INF6101 \in \{1, 3, 4\}$

Fixons maintenant INF1600 à 4, ce qui impliquera le filtrage de cette valeur du domaine de INF3710, pour la contrainte alldifferent(A), mais aussi le filtrage pour les variables INF1500 et INF6101, pour la contrainte alldifferent(C) :

- $INF4705 == 1$
- $INF8702 == 2$
- $INF3710 \in \{3\}$
- $INF1600 == 4$
- $INF1500 \in \{1, 2, 3\}$
- $INF6101 \in \{1, 3\}$

Ces étapes sont répétées jusqu'à ce qu'une solution soit trouvée.

De façon plus graphique, le filtrage réduit l'espace de recherche de solutions en retirant les valeurs du domaine des variables dont l'affectation ne mènera pas vers une solution, avant que celle-ci soit faite.

Le filtrage peut être plus ou moins précis pour une contrainte donnée. Évidemment, plus il est précis, plus il est coûteux. Le filtrage le plus abordable est appelé cohérence de bornes. Lorsqu'une solution implique une certaine assignation variable-valeur, on dit que la valeur est supportée par cette solution. La cohérence de bornes garantit que la valeur maximale et minimale de chaque domaine a un support, mais ne garantit rien pour toutes les valeurs réelles entre ces deux dernières. La cohérence d'arcs, beaucoup plus précise, garantit que chaque valeur du domaine des variables est impliquée dans une solution.

Pour résoudre un problème, un solveur CP ne fait qu'alterner entre ses couches de recherche de solution et de filtrage, jusqu'à ce qu'une solution soit trouvée. Si jamais le domaine d'une variable devient vide suite au filtrage (aucune assignation possible), le solveur fait un "backtrack", qui consiste à revenir sur une décision prise préalablement et retourner à l'état qu'il avait lors de cette prise de décision. Une variable fixée à une certaine valeur est alors fixée à une autre valeur, ce qui relance le filtrage et à nouveau la recherche de solution. Ce processus est répété jusqu'à ce qu'une solution soit trouvée ou que l'espace de solutions ait été exploré en entier.

Exemple 1.3 (Exemple de "backtrack" avec la CP)
Reprenons le CSP de l'exemple 1.1 et restreignons le domaine de la variable $INF4705$ *aux valeurs* $\{3, 4\}$

Commençons par fixer la variable $INF3710$ *à la valeur 3. Comme* $INF3710$ *sont impliquées dans la contrainte alldifferent(A), la valeur 3 doit être filtrée des domaines des variables* $INF4705$ *et* $INF1600$. *Voici donc les variables et leur domaine restants :*

- $INF4705 \in \{4\}$
- $INF8702 \in \{2, 3\}$
- $INF3710 == 3$
- $INF1600 \in \{4\}$
- $INF1500 \in \{1, 2, 3\}$
- $INF6101 \in \{1, 3, 4\}$

La contraite Alldifferent(A) détecte alors une impasse puisque les variables INF4705 et Inf1600 ne peuvent prendre que la même valeur (4). Il y aura alors un backtrack sur l'affectation à INF3710. Une nouvelle affectation peut donc être faite, comme INF3710 == 1 par exemple.

Malheureusement, le nombre d'affectations possibles (donner une valeur de son domaine à une variable) est typiquement énorme pour un CSP. Il n'est donc pas possible de tester individuellement l'ensemble des assignations. De façon générale, les solveurs décident de donner ou non une certaine valeur à une variable. L'ordonnancement des branchements devient donc une partie importante de la recherche de solutions, car elle permet de l'orienter rapidement vers les parties de l'arbre de recherche les plus prometteuses. La CP utilise donc des heuristiques de branchement pour guider la recherche. Il n'existe présentement pas d'heuristique générique et efficace qui peut être utilisée de façon automatique par les solveurs CP. Il existe l'heuristique de choix de variable plus-petit domaine d'abord (minsize), qui est générique et relativement efficace ; des heuristiques plus spécialisées, plus efficaces pour certains problèmes sont nécessaires et ont été développées. Les heuristiques de branchement basées sur le dénombrement de solution ont récemment été proposées et s'avèrent très efficaces pour un bon nombre de problèmes. Elles nécessitent cependant un algorithme de dénombrement spécialisé, pour chaque famille de contraintes. De tels algorithmes existent pour un bon nombre de contraintes, mais il en reste certaines pour lesquels ces algorithmes n'ont pas encore été développés. La contrainte d'arbre de recouvrement, nécessaire pour traiter de nombreux problèmes de réseaux, n'a pas de tel algorithme.

Trouver un l'arbre de recouvrement dans un graphe est un problème bien connu. Pour le résoudre, un sous-ensemble d'arêtes connexe couvrant tous les sommets, sans former de cycle, est trouvé dans un graphe connexe et non orienté. Il est facile de le résoudre en temps

polynomial, même dans sa version d'optimisation : l'arbre de recouvrement minimal (MST : minimum spanning tree). À la figure 1.1, les arêtes (1,2), (1,4) et (2,3) en rouge couvrent tous les sommets sans former de cycle et constituent un arbre de recouvrement.

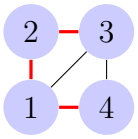


Figure 1.1 Exemple d'arbre de recouvrement

Bien que ce problème puisse être résolu en temps polynomial, ajouter certaines contraintes à l'arbre de recouvrement le rend NP-Difficile. Or, de tels problèmes sont fréquents : conception de réseaux informatiques, télécommunication ou élaboration de réseaux de transport. Le problème d'arbre de recouvrement avec degré contraint[28], le problème d'arbre de recouvrement hop-contraint[15] et l'arbre de recouvrement de diamètre contraint[3] sont des exemples de problèmes pour lesquels un arbre de recouvrement contraint doit être trouvé.

Ce mémoire s'attarde sur les heuristiques de branchement basées sur le dénombrement de solutions, en particulier pour les problèmes relatifs aux arbres de recouvrement, dans des graphes non orientés. Nos contributions principales sont les suivantes :

- L'élaboration d'un algorithme permettant de dénombrer les solutions pour la contrainte d'arbre de recouvrement non orientés, à partir de théorèmes mathématiques existants.
- L'implémentation et l'utilisation de cet algorithme dans une heuristique de branchement, qui guide la recherche en fonction de la densité de solutions.
- La résolution de problèmes d'arbres de recouvrement contraints avec la nouvelle heuristique proposée.
- La généralisation de l'algorithme de dénombrement pour les graphes orientés, ce qui implique une généralisation de l'heuristique de branchement qui en découle.
- L'utilisation de cet algorithme généralisé dans une heuristique de branchement, qui guide la recherche en fonction de la densité de solutions.
- La résolution de problèmes d'arborescences contraintes avec la nouvelle heuristique proposée.
- L'élaboration et la mise en place d'une procédure de mise à jour des matrices utilisées pour le calcul de la densité de solution, pour la contrainte d'arbre de recouvrement et d'anti-arborescence.
- L'élaboration et la mise en place d'un méthode permettant de calculer les densités de solutions de façon incrémentale, pour la contrainte d'arbre de recouvrement et la contrainte d'anti-arborescence.

Le mémoire est organisé comme suit : le chapitre 2 résume le travail précédant ce mémoire. Le chapitre 3 introduit nos algorithmes de dénombrement pour les arbres de recouvrements. Le chapitre 4 présente la mise en oeuvre des algorithmes du chapitre 3. Les résultats expérimentaux, issus de la comparaison des heuristiques de branchement basées sur le dénombrement avec d'autres approches, sont donnés dans le chapitre 5. Finalement, la conclusion est présentée dans le chapitre 6.

CHAPITRE 2

REVUE DE LITTÉRATURE

Dans ce chapitre, la recherche précédant le travail réalisé dans ce mémoire est abordée. Ce chapitre commence par la section 2.1, qui explique différentes heuristiques de branchement. Ensuite, la section 2.2 aborde les heuristiques de branchement basées sur le dénombrement et les algorithmes de dénombrement proposés pour plusieurs contraintes. Le travail fait sur les contraintes d'arbre de recouvrement est expliqué à la section 2.3. Enfin, les arbres ou arborescences contraints sont abordés à la section 2.4.

2.1 Heuristiques de branchement

Pour la CP, il n'existe pas à ce jour d'approche exacte, générique et intégrée au solveur pour choisir dans quel ordre les variables-valeurs, appelés branchements, doivent être choisies. Conséquemment, la CP fait appel aux heuristiques de branchement pour explorer rapidement les parties les plus prometteuses de l'arbre de recherche, afin de trouver efficacement des solutions aux CSP. En se basant sur plusieurs critères, tels la structure du problème, la taille des domaines ou l'impact d'une assignation sur les autres domaines, ces heuristiques indiquent quel branchement il est préférable de faire à un endroit précis dans l'arbre de recherche. Il existe un certain dilemme pour l'utilisation des heuristiques de recherche : les heuristiques très simples utilisent peu la structure du problème, donc guident moins bien la recherche. Les heuristiques plus complexes, guidant mieux la recherche, demandent un temps de calcul plus important. Dans cette section, différentes heuristiques de branchement seront décrites.

Les heuristiques de branchement se divisent en trois catégories principales, impliquant chacune un choix sur une composante particulière du branchement. La première est celle se concentrant sur le choix de la variable. Ce type d'heuristique détermine quelle variable il est préférable de fixer en premier. La deuxième catégorie est l'heuristique qui fait un choix de valeur. Une fois la variable choisie, cette approche détermine quelle valeur il est préférable de donner à la variable en premier. Enfin, la troisième et dernière catégorie fait à la fois le choix de variable et de valeur.

L'une des premières stratégies de branchement introduites fut celle du "fail-first principle", proposée par Haralick et Elliott [16]. L'idée derrière cette approche est fort simple : il s'agit de faire le plus restrictif, le plus difficile, dès le début. Ce faisant, faire le choix qui risque de mener vers un échec (branchement ne menant vers aucune solution) s'est montré très bénéfique.

Haralick et Elliott ont démontré que les approches respectant ce principe permettent d'obtenir de meilleurs résultats que les approches standards [16]. Un grand nombre d'heuristiques de branchement simples s'inspirent de ce principe.

Une heuristique appliquant directement ce principe est l'heuristique de choix de variable plus-petit-domaine-d'abord[16]. Cette heuristique très simple peut être appliquée à pratiquement n'importe quel problème. Le principe est le suivant : la variable dont le domaine comporte le plus petit nombre de valeurs est beaucoup plus prompte à causer un échec dans la recherche de solutions, étant donné que le choix de valeurs possibles pour cette variable est plus restreint. Puisque cette variable risque de causer des échecs, il est beaucoup plus judicieux de lui assigner une valeur dès le début de la recherche et obtenir un échec presque immédiatement, plutôt qu'après un grand nombre de branchements, ce qui reviendrait à perdre du temps.

Exemple 2.1 (Heuristique Plus-Petit-Domaine-d'Abord)
Soit le CSP simple avec les variables et domaines suivants :

- $a \in \{1, 2, 3\}$
- $b \in \{2, 3, 5\}$
- $c \in \{3, 4\}$
- $d \in \{3, 4\}$

et la contrainte globale alldifferent<a,b,c,d>, qui implique que les 4 variables ne doivent pas être fixées à la même valeur. (a ==2 et b==2 ne respecterait pas cette contrainte). Nous supposons ici un algorithme de filtrage simple qui ne considère que les contraintes binaires.

Si l'heuristique plus-petit-domaine-d'abord est appliquée, on branchera d'abord sur la variable c ou d, car les deux ont un domaine de taille 2, tandis que les variables a et b ont trois valeurs dans leur domaine. Par exemple, prenons c == 3. La valeur 3 doit donc être éliminée des autres domaines, car autrement, a,b ou d pourrait prendre la même valeur que c. Ce faisant, les nouveaux domaines deviennent :

- $a \in \{1, 2\}$
- $b \in \{2, 5\}$
- $c == 3$
- $d \in \{4\}$

La variable d sera fixée sans branchement, car elle n'a qu'une seule valeur dans son domaine.

Il existe aussi d'autres heuristiques de branchement qui tentent de maximiser le filtrage en choisissant des valeurs qui auront un impact important sur le domaine des autres variables. L'heuristique "Impact-based search", proposée par Refalo [37] choisit la variable ayant le

plus grand impact dans la recherche de solution. Cette approche mesure l'impact de chaque assignation en calculant le produit de la taille des domaines de l'ensemble des variables, de la façon suivante :

$P = |D_{x_1}| \times ... \times |D_{x_n}|$

où P représente le produit de la taille des domaines des variables x_1 à x_n

L'impact d'une affectation peut alors être calculé comme suit :

$I(x_i == a) = 1 - \frac{P_{\text{après}}}{P_{\text{avant}}}$

Où I est l'impact pour une assignation donnée. P_{avant} et $P_{\text{après}}$ représentent le produit de la taille des domaines avant et après l'assignation, respectivement.

Plus la taille de l'espace de recherche sera diminuée, plus l'impact calculé sera grand. Cependant, calculer l'impact pour toutes les assignations possibles à chaque branchement s'avère coûteux. Il est possible de faire une moyenne des impacts pour une variable donnée, en sommant l'impact de chaque assignation et en divisant par le nombre d'assignations. Cette méthode est beaucoup plus efficace. Cette stratégie globale permet de résoudre des exemplaires de problèmes qui étaient auparavant impossibles à résoudre avec des stratégies standards, comme plus-petit-domaine-d'abord.

Exemple 2.2 (Heuristique Impact-based search)
Soit le même CSP que celui utilisé dans l'exemple 2.1.

Calculons le produit des domaines P de base :

$P = |D_a| \times |D_b| \times |D_c| \times |D_d| = 3 \times 3 \times 2 \times 2 = 36$

Calculons maintenant l'impact de chaque assignation : Si a == 1 est choisi, la valeur 1 doit être retirée de tous les domaines (aucun dans ce cas-ci)

$I(a == 1) = 1 - \frac{P_{après}}{P_{avant}} = 1 - \frac{1 \times 3 \times 2 \times 2}{36} = 1 - \frac{12}{36} = \frac{24}{36}$

Si a== 2 est choisi, la valeur 2 sera retirée du domaine de b.

$I(a == 2) = 1 - \frac{P_{après}}{P_{avant}} = 1 - \frac{1 \times 2 \times 2 \times 2}{36} = 1 - \frac{8}{36} = \frac{28}{36}$

si a==3 est choisi, la valeur 3 devra être retiré des domaines de b,c et d

$I(a == 3) = 1 - \frac{P_{après}}{P_{avant}} = 1 - \frac{1 \times 2 \times 1 \times 1}{36} = 1 - \frac{2}{36} = \frac{34}{36}$

Voici les Impacts pour les autres assignations :

$I(b == 2) = \frac{28}{36}$
$I(b == 3) = \frac{34}{36}$
$I(b == 5) = \frac{24}{36}$
$I(c == 3) = \frac{32}{36}$
$I(c == 4) = \frac{27}{36}$
$I(d == 3) = \frac{32}{36}$
$I(d == 4) = \frac{27}{36}$

Comme le rapport le plus grand est celui de $I(a == 3)$*, il sera choisi.*

Une autre stratégie de branchement élaborée est l'heuristique dom/wdeg, proposée par Boussemart *et al.* [8]. Cette approche attribue un poids initial de 1 à chaque contrainte. À chaque fois qu'une contrainte cause un échec (le filtrage causé par cette contrainte vide le domaine d'une variable), son poids est augmenté de 1. Le degré pondéré d'une variable correspond à la somme des poids des contraintes qui l'impliquent. Cette information, si le "fail-first principle" est considéré, devient très intéressante, car elle permet d'ordonner les variables en fonction de la somme des poids des contraintes qui y sont associées. Un ratio divisant la taille du domaine par le degré pondéré de la variable peut alors être calculé. L'heuristique ne fait que choisir la variable ayant le plus petit rapport, ce qui correspond à choisir la variable qui a le plus grand potentiel de conflit d'abord.

Exemple 2.3 (Heuristique dom/wdeg)
Soit le même CSP que celui utilisé dans l'exemple 2.1. Ajoutons la contrainte arithmétique suivante :

$a + c > 6$

Les variables ont les degrés pondérés suivants initialement :

degré $a = 2$

degré $b = 1$

degré $c = 2$

degré $d = 1$

Calculons les ratios initiaux :

ratio $a = \frac{|D_a|}{deg_a} = \frac{3}{2}$

ratio $b = \frac{|D_b|}{deg_b} = \frac{3}{1}$

ratio $c = \frac{|D_c|}{deg_c} = \frac{2}{2}$

ratio $d = \frac{|D_d|}{deg_d} = \frac{2}{1}$

Comme la variable c admet le plus petit ratio, elle sera choisie d'abord. Fixons c==3. La contrainte alldifferent retire la valeur 3 de tous les domaines, tandis que la contrainte arithmétique retire toutes les valeurs du domaine de a, car aucune ne permet de respecter la contrainte. Les domaines deviennent :

- $a \in \{\}$
- $b \in \{2, 5\}$
- $c == 3$
- $d \in \{4\}$

Comme la contrainte arithmétique cause un échec, vidant le domaine de la variable a, son poids augmente et prend la valeur 2. Ce faisant, les degrés des variables a et c augmenteront et prendront la valeur 3. Après le backtrack, les domaines sont les suivants :

– $a \in \{1, 2, 3\}$
– $b \in \{2, 3, 5\}$
– $c \in \{4\}$
– $d \in \{3, 4\}$

les ratios deviennent :

ratio $a = \frac{|D_a|}{deg_a} = \frac{3}{3}$
ratio $b = \frac{|D_b|}{deg_b} = \frac{2}{1}$
ratio $c = \frac{|D_c|}{deg_c} = \frac{1}{3}$
ratio $d = \frac{|D_d|}{deg_d} = \frac{2}{1}$

La variable c sera choisie ensuite, car elle a le plus petit ratio.

L'activité des variables dans le filtrage du solveur peut également être utilisée pour guider la recherche. L'heuristique de branchement "Activity-based search", inspirée des solveurs SAT (heuristique VSID[27]), a été proposée par Michel et Van Hentenryck[26]. Elle mesure à quelle fréquence le domaine d'une variable est modifié. L'activité d'une variable x dans l'ensemble des variables X du CSP est calculée de la façon suivante :

$\forall x \in X - X'$ t.q. $|D(x)| > 1 : A(x) = A(x) \cdot \gamma$
$\forall x \in X' : A(x) = A(x) + 1$
$0 < \gamma < 1$

Où X' est l'ensemble des variables affectées. Il faut évidemment que le domaine de la variable soit plus grand que 1 pour que son activité soit considérée, autrement elle est déjà fixée. La valeur de γ, située entre 0 et 1, affecte à quelle vitesse l'activité d'une variable diminue d'une itération à l'autre. Plus sa valeur est petite, plus l'activité d'une variable décroit rapidement en fonction du temps. Ainsi, de façon générale, l'activité relative à une variable diminue avec le temps, mais augmente si son domaine est régulièrement affecté. L'activité est mise à jour à chaque noeud de l'arbre de recherche. Un ratio peut alors être calculé $\frac{A(x)}{|D(x)|}$. En utilisant ce ratio, il est possible d'ordonner les variables et de choisir celle qui est la plus active d'abord. Il est également possible d'estimer l'activité relative à l'assignation d'une valeur.

Exemple 2.4 (Heuristique Activity-based search)
Soit le même CSP que celui utilisé dans l'exemple 2.1

Choisissons un $\gamma = 0.5$ *et initialisons l'activité de chaque variable à 1.*

Fixons a == 2, comme première assignation. Les domaines des variables deviennent :

– $a == 2$
– $b \in \{3, 5\}$
– $c \in \{3, 4\}$

– $d \in \{3, 4\}$

Le domaine de la variables b est affecté, ce qui augmente son activité :

$A(b) = A(b) + 1 = 2$

La mise à jour des ratios doit également être faite pour les variables dont les domaines demeurent inchangés :

$A(c) = A(c) \times 0.5 = 0.5$

$A(d) = A(d) \times 0.5 = 0.5$

Calculons les ratios :

ratio $b = \frac{A(b)}{|D_b|} = \frac{2}{2} = 1$

ratio $c = \frac{A(c)}{|D_c|} = \frac{0.5}{2} = 0.25$

ratio $d = \frac{A(d)}{|D_d|} = \frac{0.5}{2} = 0.25$

Comme b a le plus grand ratio, c'est la variable la plus active. Elle sera choisie.

D'autres approches tentent plutôt de mesurer le bénéfice de fixer ou de ne pas fixer une certaine valeur à une variable à un moment donné, dans un problème d'optimisation. C'est le cas de la stratégie du regret[33]. L'idée derrière cette stratégie est la suivante : mesurer le bénéfice résultant de l'assignation d'une certaine valeur à une variable et ensuite mesurer la perte résultant de faire un choix différent. Cette perte est appelée regret. L'heuristique de branchement en découlant tente de minimiser le regret.

Exemple 2.5 (Heuristique de branchement regret)

Soit le CSP suivant, sous sa version optimisation, où la fonction objectif attribue les bénéfices (ici donnés arbitrairement) suivants (entre parenthèses) pour chaque valeur du domaine :

– $a \in \{1(5), 2(10), 3(12)\}$

– $b \in \{2(5), 3(2), 4(8)\}$

– $c \in \{3(3), 4(4)\}$

Considérations de l'heuristique

– *Considérons la valeur 4, pour la variable b. Si l'assignation b==4, dont le bénéfice est de 8 n'est pas faite, le mieux qu'il est possible de faire est b == 2, qui donne un bénéfice de 5. Conséquemment, si l'assignation b == 4 n'est pas faite, l'heuristique estimera un regret de 3.*

– *Considérons la valeur 3 pour la variable a. Le maximum possible est 12, avec a ==3. Ensuite, si a==3 n'est pas choisi, a==2 peut l'être, ce qui constitue un regret de 2.*

– *Pour la variable c, si c==4 est choisi, le bénéfice est de 4, tandis que l'alternative ne permet qu'un revenu de 3, ce qui résulte en un regret de 1.*

Comme l'affectation b == 4 admet le plus grand regret, elle sera faite en premier.

2.2 Heuristiques de branchement basées sur le dénombrement

Récemment, des heuristiques basées sur le dénombrement ont été proposées. Ces heuristiques sont très efficaces, mais nécessitent un algorithme de dénombrement adapté à chaque contrainte.

L'idée générale des heuristiques de dénombrement est de prendre une décision de branchement en se basant sur le nombre de solutions qui la supportent et qui incluent cette assignation particulière. De façon plus formelle, ces heuristiques utilisent les concepts de dénombrement de solutions et de densité de solution [31, 45, 34], qui sont définis de la façon suivante :

Définition 2.2.1 (Dénombrement de solution)
Étant donné une contrainte $c(x_1, ..., x_n)$ et les domaines finis respectifs $D_i, 1 < i < n$, $\#c(x_1, ..., x_n)$ représente le nombre de n-tuples dans la relation correspondante.

Définition 2.2.2 (Densité de solution)
Étant donné une contrainte $c(x_1, ..., x_n)$ et les domaines finis respectifs $D_i, 1 < i < n$, pour une variable x_i incluse dans la contrainte c et une valeur $d \in D_i$, nous appelons

$$\sigma(x_i, d, c) = \frac{\#c(x_1, \ldots, x_{i-1}, d, x_{i+1}, \ldots, x_n)}{\#c(x_1, \ldots, x_n)}$$

la densité de solution *d'une paire (x_i, d) dans c. La densité de solution mesure à quelle fréquence un choix variable-valeur fait partie d'une solution respectant la contrainte c*

L'idée d'utiliser le nombre de solutions pour guider la recherche a déjà été exploitée par le passé. Kask *et al.* [20] ont proposé une heuristique qui approxime le nombre de solutions qui étend une solution partielle. Celle-ci choisit l'assignation participant au plus grand nombre de solutions pour la variable courante. Une heuristique basée sur la distribution des solutions, proposée initialement par Hsu *et al.* [17] et ensuite améliorée par [22], utilise le framework "Expectation-Maximisation Belief Propagation" (EMPB). La probabilité qu'une variable prenne une certaine valeur dans une solution est calculée, puis utilisée pour guider la recherche. Les travaux utilisant la densité de solution[31, 45, 34] diffèrent des autres approches, car le calcul est fait individuellement pour chaque contrainte.

La densité de solution peut être calculée de façon exacte ou approximée, pour une contrainte donnée. Pour ce faire, un algorithme de dénombrement, adapté à une contrainte particulière, est nécessaire. En utilisant des informations intrinsèques aux contraintes, ainsi que des propriétés mathématiques, le nombre de solutions impliquant une assignation particulière peut être calculé ou approximé. Par exemple, le calcul du permanent, une fonction de la matrice similaire au déterminant, peut être utilisé pour évaluer la densité de solution pour la

contrainte alldifferent [45, 34], en construisant la matrice représentant le graphe de valeurs pour l'ensemble des variables de la contrainte.

Il existe cependant un compromis entre la précision de l'algorithme de dénombrement et son efficacité dans l'heuristique de recherche. Comme le dénombrement de solutions doit être fait avant chaque branchement, il doit être le plus efficace possible, sinon il serait moins coûteux de risquer un branchement moins bon et d'éventuellement atteindre une solution, que de faire un long calcul menant directement à celle-ci. Dans cette optique, Pesant *et al.* [34] bornent le permanent, ce qui est nettement plus rapide que de le calculer exactement. Bien que l'heuristique basée sur le dénombrement de solutions perde de la précision, le fait d'avoir un calcul plus performant permet d'opter pour une solution combinée : un calcul moins précis, mais moins coûteux, ce qui s'avère beaucoup plus efficace globalement.

Pesant *et al.* [34] décrivent des algorithmes de dénombrement pour de nombreuses autres contraintes : alldifferent symétrique, cardinalité globale, regular et knapsack, qui permettent de résoudre de nombreux autres problèmes. Pour la version symétrique de la contrainte alldifferent, les auteurs étendent l'algorithme développé pour la contrainte alldifferent de base. Ils calculent une borne supérieure du permanent de façon un peu moins précise qu'avec la contrainte de base. La contrainte de cardinalité globale est une généralisation de la contrainte alldifferent, donc les auteurs utilisent une fois de plus une borne supérieure sur le permanent pour dénombrer les solutions. Pour les contraintes regular et knapsack, Pesant *et al.* [34] proposent un algorithme exact. Les auteurs décrivent également un algorithme approché pour la contrainte knapsack.

Des algorithmes de dénombrement ont également été proposés pour d'autres contraintes globales : element[35] et spread/deviation[32]. Pour la contrainte element, un algorithme exact est proposé. Un algorithme exact est également décrit par l'auteur pour la contrainte spread/deviation.

Pesant *et al.* [34] ont généralisé le concept d'heuristique basée sur le dénombrement de solution sous la forme d'une heuristique générique, centrée sur la contrainte. Leur approche, maxSD (max Solution Density), combine le choix de variable et de valeur en itérant sur chaque variable et chaque valeur de son domaine. L'assignation ayant la densité de solution la plus élevée est choisie. Au fur et à mesure que la recherche de solution progresse, les densités de solutions sont recalculées ou mises à jour, ce qui permet d'utiliser cette heuristique jusqu'à ce que le problème soit résolu.

Bien que des algorithmes de dénombrement existent pour plusieurs contraintes, il en reste plusieurs pour lesquels un tel algorithme n'a pas encore été développé. La contrainte arbre de recouvrement, qui permet de modéliser plusieurs problèmes de réseaux, n'admet toujours pas d'algorithme de dénombrement. La prochaine section donne plus de détails sur ce type

de contrainte.

2.3 Contraintes d'arbres de recouvrement

La recherche dans la communauté CP sur les structures d'arbres imposées s'est concentrée sur les algorithmes de filtrage et non sur les heuristiques de branchement.

Beldiceanu *et al.* [6] ont introduit la contrainte d'arbre, qui permet le partitionnement d'un problème de digraphe avec une perspective de CP. Dans leur travail, une contrainte utilisant une variable de type ensemble pour représenter une anti-arborescence est proposée. Ce type de variable regroupe tous les sommets faisant partie de l'anti-arborescence dans un ensemble et ceux n'en faisant pas partie dans l'autre. Avec leur contrainte, les auteurs obtiennent une cohérence de domaine en $\mathcal{O}(nm)$, où n est le nombre de sommets du graphe, tandis que m est le nombre d'arêtes. Le filtrage de leur contrainte repose sur l'identification des points d'articulation du graphe, ainsi que sur ses racines et ses puits. Ces informations sont utilisées pour évaluer le nombre minimum et le maximum d'arbres nécessaires pour partitionner le graphe.

Dooms et Katriel [9] ont introduit la contrainte MST, qui requiert une variable arbre pour représenter l'arbre de recouvrement minimum du graphe sur lequel la contrainte est définie. Plusieurs variantes du problème de l'arbre de recouvrement minimum, comme le "minimum k-spanning tree" et le "Steiner tree" sont connus pour être NP-difficiles, bien que la version de base du problème puisse être résolue en temps polynomial. Ces problèmes peuvent être modélisés en combinant la contrainte d'arbre de recouvrement minimum et d'autres contraintes. Les auteurs ont proposé un algorithme de filtrage qui maintient la cohérence de borne en temps polynomial, pour plusieurs restrictions de cette contrainte. Leur algorithme divise les arêtes en trois ensembles : obligatoires, possibles et interdites. Par la suite, Dooms et Katriel [10] ont proposé une version de la contrainte avec un poids (weighted spanning tree contraint), dans laquelle l'arbre de recouvrement et le poids des arêtes sont des variables. Ils considèrent plusieurs algorithmes de filtrage. Dans leur travail, une variable d'ensemble est utilisée, indiquant quelles arêtes font partie de l'arbre de recouvrement.

Le filtrage proposé par Dooms et Katriel [10] a par la suite été simplifié et amélioré par Régin [38], qui a proposé un algorithme de filtrage incrémental, qui maintient plusieurs composantes connexes et représente les opérations de fusion des arbres disjoints dans l'algorithme de Kruskal. Conséquemment, la cohérence de domaine a été atteinte pour cette contrainte, en $\mathcal{O}(m + n \log n)$. Subséquemment, Régin *et al.* [39] ont amélioré la complexité de ce filtrage.

2.4 STAs Contraints

Bien que le problème de l'arbre de recouvrement minimal puisse se résoudre en $\mathcal{O}(n^2)$ avec les algorithmes de Prim et Kruskal, ajouter certaines contraintes sur celui-ci le rend NP-difficile. Plutôt que de trouver un arbre de recouvrement minimum, il peut être intéressant d'énumérer tous les arbres recouvrants possibles dans un graphe. Comme nous considérerons à la fois les graphes orientés et non orientés dans notre travail, nous utiliserons STA(Spanning tree or Arborescence) pour référer aux arbres (non orientés) ou arborescences (orientées) de recouvrement.

Beaucoup de recherche a été faite sur l'énumération des STAs dans les graphes. Gabow et Myers [13] ont d'abord introduit un algorithme qui trouve tous les STAs dans les graphes orientés ou non orientés. Leur approche utilise le "backtracking" et la recherche en profondeur, énumérant tous les STAs en $\mathcal{O}(V + E + EN)$, où V, E et N représentent le nombre de sommets, arêtes et STAs, respectivement. Kapoor et Ramesh [18] ont présenté un algorithme pour trouver les STAs dans les graphes non orientés avec des arêtes avec ou sans poids. En construisant d'abord un arbre et en utilisant une approche pour parcourir celui-ci, les auteurs sont capables de représenter les STAs en décrivant uniquement les changements relatifs d'un STA à l'autre, plutôt qu'en les décrivant entièrement . Ils améliorent ensuite la performance de leur approche pour les graphes orientés[19], en présentant un nouvel algorithme, qui énumère tous les STAs en $\mathcal{O}(NV + V^3)$. Leur approche implique l'échange d'arcs faisant partie ou non du premier STA décrit, ce qui permet de les énumérer tous. Uno [42] a également proposé une approche pour énumérer tous les STAs dans un graphe orienté, en $\mathcal{O}(E + ND(V, E))$, où $D(V, E)$ est la complexité de la structure de données nécessaire pour mettre à jour le STA, dans un graphe non orienté de V sommets et E arêtes.

Les approches énumérées dans le paragraphe précédent énumèrent les STAs en appliquant principalement des changements locaux. Notre travail se distingue de cette recherche par son objectif principal : utiliser l'information locale sur les arêtes (densité de solution) comme heuristique de branchement, pour construire un STA. Nous utilisons le filtrage de la contrainte d'arbre de recouvrement pour calculer le nombre de solutions impliquant une certaine arête. Au fur et à mesure que la recherche de solution progresse, l'information sur les arêtes est mise à jour, jusqu'à ce qu'un STA complet soit construit. Notre travail peut donc être utilisé pour énumérer les STAs, mais notre intérêt se situe plutôt au niveau des STAs contraints.

Il existe de nombreux problèmes impliquant une contrainte de degré sur le STA. Lokshtanov *et al.* [23] considèrent le "degree preserving spanning tree problem". Ils considèrent explicitement deux problèmes en particulier[24]. Le premier est le "full degree spanning tree problem" qui, étant donné un graphe connexe et non orienté, détermine si le graphe G contient

un STA T dans lequel au moins k sommets ont le même degré dans G que dans T. Les auteurs généralisent ce problème aux graphes orientés. Ils considèrent également le problème inverse, le "reduced degree spanning tree", dans lequel au moins k sommets doivent avoir un degré différent dans G et T. L'intérêt d'un tel problème se situe au niveau de la conception de réseaux de distribution d'eau, où un nombre minimal d'appareils mesurant le débit devait être installés[36].

Un exemple de problème plus général est le "minimum degree spanning tree problem"[12], qui consiste à trouver le STA ayant le plus petit degré maximum dans le graphe. Les auteurs proposent un algorithme approximatif pour résoudre ce problème, dont la généralisation permet également de résoudre le "minimum degree steiner tree problem". Ce dernier problème implique la construction d'un sous-graphe de recouvrement dont certaines composantes ne sont pas connexes.

Il existe également des contraintes qui ne sont pas reliées aux degrés des sommets du STA. Par exemple, Alon *et al.* [4] décrivent le "directed maximum leaf out branching problem", qui consiste à trouver le STA ayant le plus grand nombre de feuilles. Ce problème NP-difficile admet également une version non orientée, qui est également NP-difficile[14]. L'intérêt de ces problèmes se situe au niveau de la conception de protocoles de routage, où les nombres d'entrées et de sorties des routeurs est limité.

Beaucoup de travail a également été fait sur les STAs contraints dont les arêtes ont des poids, ce qui en fait des problèmes d'optimisation. Khandekar *et al.* [21] décrivent un algorithme approximatif pour résoudre le "minimum-cost degree constrained 2-node connected subgraph problem". Résoudre ce problème implique la recherche d'un MST dont tous les sommets ont un degré inférieur à une certaine valeur. Ce MST doit également être 2-sommets connecté, ce qui signifie que le retrait de n'importe quel sommet n'affecte pas la connexité du sous-graphe. Nutov [29] donne un algorithme approximatif pour résoudre le "directed weighted degree constrained network", problème qui implique la recherche d'un sous-graphe de coût minimum f-connecté (le retrait de f sommets ne déconnecte pas le sous-graphe) qui satisfait également des contraintes de degré. Un autre problème est le "minimum crossing spanning tree problem", considéré par [5]. Ce problème est résolu en trouvant un MST qui contient au moins b arêtes faisant partie d'un sous-ensemble d'arêtes $e \in E$. Les auteurs proposent une amélioration à la garantie d'approximation de l'algorithme existant, en plus d'une extension à la technique de résolution pour le cas orienté.

Notre travail se situe au niveau de la résolution des problèmes de satisfaction, pour les graphes orientés et non orientés. Nous ne considérons donc pas les problèmes d'optimisation impliquant un poids sur les arêtes, mais uniquement les problèmes où un STA respectant certaines contraintes doit être trouvé.

CHAPITRE 3

ALGORITHMES DE DÉNOMBREMENT

Dans ce chapitre, les algorithmes de dénombrement proposés pour la contrainte d'arbre de recouvrement et d'anti-arborescence sont détaillés. À la section 3.1, l'algorithme de dénombrement est démontré et expliqué pour les graphes non orientés. À la section 3.2, l'algorithme est étendu pour les graphes orientés.

3.1 Algorithme de dénombrement pour les graphes non orientés

Un graphe est défini comme $G = (V, E)$ où V est un ensemble de sommets et E est un ensemble d'arêtes, couplant les sommets de V. Un graphe non orienté a un ensemble d'arêtes qui couple les sommets de façon non ordonnée. Un graphe orienté, ou digraphe, a plutôt un ensemble d'arcs, qui couple les sommets de façon ordonnée. Les arcs ont une information de plus que les arêtes : un sens.

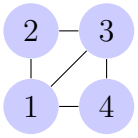


Figure 3.1 Graphe non orienté

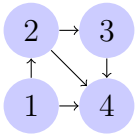


Figure 3.2 Graphe orienté

En comptant le nombre d'arêtes reliées à un sommet d'un graphe non orienté, le degré $deg(v)$ d'un sommet v est obtenu. Pour un graphe, il est possible de calculer le degré sortant et entrant du sommet. Le degré entrant, noté $deg^-(s)$, correspond au nombre d'arcs orientés vers le sommet, tandis que le degré sortant, noté $deg^+(s)$, correspond au nombre d'arcs dont l'origine est le sommet.

Un graphe $G' = (V', E')$ est défini comme un sous-graphe induit de $G = (V, E)$ si $V' \subseteq V$, $E' \subseteq E$ et si E' est l'ensemble des arêtes de E dont les deux sommets font partie de V'. G'

est un graphe connexe si pour chaque paire de sommets $u, v \in V$ il existe au moins un chemin entre u et v. Un arbre est un graphe non orienté acyclique connexe.

Continuons avec une définition plus formelle d'un arbre de recouvrement :

Définition 3.1.1 (Arbre de recouvrement (Spanning tree)[11])
Un arbre de recouvrement T d'un graphe G est un arbre $T(V, E')$ où $E' \subseteq E$.

Les graphes peuvent être représentés en mémoire de plusieurs façons. La plus commune est la matrice d'adjacence, une matrice carrée $n \times n$ qui indique si une arête entre les sommets i et j est présente, pour l'ensemble des arêtes possibles. La figure 3.3 est un exemple de matrice d'adjacence, pour le graphe non orienté de la figure 3.1, tandis que la figure 3.4 l'est pour le graphe orienté de la figure 3.2.

$$A = \begin{pmatrix} 0 & 1 & 1 & 1 \\ 1 & 0 & 1 & 0 \\ 1 & 1 & 0 & 1 \\ 1 & 0 & 1 & 0 \end{pmatrix}$$

Figure 3.3 Matrice d'adjacence pour le graphe non orienté 3.1

$$A = \begin{pmatrix} 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \end{pmatrix}$$

Figure 3.4 Matrice d'adjacence pour le graphe orienté 3.2

Pour un graphe non orienté, la matrice d'adjacence est toujours symétrique, ce qui implique que les matrices triangulaires supérieures et inférieures partage la même information. Ce n'est pas le cas pour la matrice d'adjacence du graphe orienté.

La *matrice Laplacienne* $L(G)$ d'un graphe non orienté G est formée en soustrayant la matrice d'adjacence de G de la matrice diagonale où l'entrée i correspond au degré du sommet i dans G. Plus formellement :

$$L(G) = D - A(G)$$

où D est la matrice diagonale dont l'entrée i correspond à $deg(v_i)$ et A est la matrice d'adjacence de G.

Par convention, nous ferons référence à cette matrice simplement par L. La figure 3.5 est un exemple de matrice Laplacienne pour le graphe non orienté 3.1.

$$L = \begin{pmatrix} 3 & -1 & -1 & -1 \\ -1 & 2 & -1 & 0 \\ -1 & -1 & 3 & -1 \\ -1 & 0 & -1 & 2 \end{pmatrix}$$

Figure 3.5 Matrice Laplacienne pour le graphe 3.1

Tout comme la matrice d'adjacence, la matrice Laplacienne est symétrique pour les graphes non orientés. Le *mineur*-(i, j) d'une matrice carrée M, noté M_{ij}, est le déterminant de la sous-matrice obtenue en retirant la i^e rangée et la j^e colonne de M. La matrice Laplacienne a une propriété fort intéressante : son mineur-(i, j), pour n'importe quelles rangée i et colonne j, est égal au nombre d'arbres de recouvrement du graphe correspondant.

Theoreme 3.1.1 (Kirchhoff's Matrix-Tree [41])
Dénotons par $\tau(G)$ le nombre d'arbres de recouvrement du graphe G de n sommets et par L_{ij} le mineur-(i, j) de la matrice Laplacienne de G, pour $1 \leq i, j \leq n$ quelconques. Alors

$$\tau(G) = |L_{ij}|.$$

Conséquemment, le nombre de solutions pour la contrainte `arbre de recouvrement` peut être calculé comme le déterminant d'une matrice carrée $(n-1) \times (n-1)$, en $\mathcal{O}(n^3)$.

Cependant, la matrice Laplacienne est une M-matrice[25], ce qui lui donne une propriété particulière : ses mineurs principaux sont toujours positifs[30]. Une M-matrice est définie de la façon suivante :

$A - sI - B$ où $B = (b_{ij})$ avec $b_{ij} \geq 0$, pour tous $1 \leq i, j \leq n$, et $s \geq p(B)$, le maximum du moduli des vecteurs propres de B.

Un mineur principal est le déterminant d'une sous-matrice formée par le retrait de la rangée et colonne correspondante (autrement dit, $i = j$). Par conséquent, si la rangée et colonne retirées de la matrice Laplacienne sont les mêmes, nous avons :

Corollaire 3.1.1
Dénotons par $\tau(G)$ le nombre d'arbres de recouvrement du graphe G de n sommets et par L_{ii} le mineur-(i, i) de la matrice Laplacienne de G, pour $1 \leq i \leq n$ quelconque. Alors

$$\tau(G) = L_{ii}.$$

Dans notre cas, les colonnes et rangées retirées pour le calcul du mineur-(i,j) seront toujours correspondantes ($i = j$). Si nous retirons la première rangée et colonne de la matrice Laplacienne de la figure 3.5, le mineur résultant est $L_{11} = 2 \times (3 \times 2 - (-1) \times (-1)) - (-1) \times (-1 \times 2 - (-1) \times 0) = 8$, comme illustré à la figure 3.6, où les 8 arbres de recouvrement possibles sont illustrés.

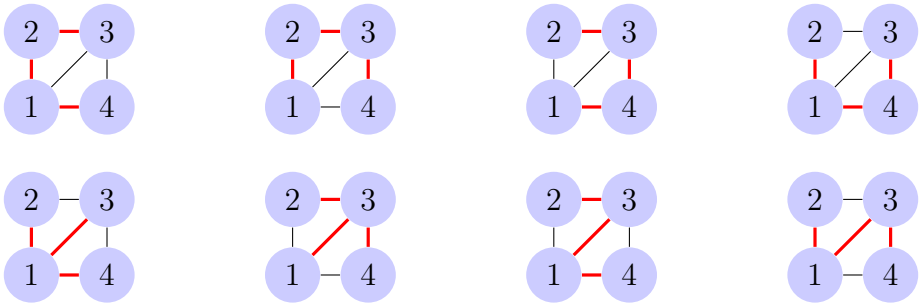


Figure 3.6 Arbres recouvrants (arêtes en rouge) présents dans le graphe 3.1

Nous sommes plutôt intéressés à calculer la densité de solution pour une arête $(i,j) \in E$. Pour ce faire, il est possible de compter le nombre d'arbres de recouvrement qui n'utilisent pas cette arête, $\tau(G \setminus \{(i,j)\})$, et ensuite, diviser cette quantité par le nombre total d'arbres de recouvrement existants dans le graphe. Le résultat est la densité de solution correspondant à l'affectation de la valeur 0 à la variable associée à l'arête retirée (i.e. $(i,j) \notin T$) :

$$\sigma((i,j), 0, \texttt{arbre de recouvrement}(G,T)) = \frac{\tau(G \setminus \{(i,j)\})}{\tau(G)}.$$

Soit $L' = L(G \setminus \{(i,j)\})$. Comment L' diffère de L ? Ces deux matrices sont identiques, sauf pour les cellules ℓ_{ii}, ℓ_{jj}, ℓ_{ij}, et ℓ_{ji}. En effet, comme nous pouvons choisir le retrait de n'importe quelles rangées et colonnes pour calculer le mineur, nous retirons la rangée et la colonne i. Cela nous évite de devoir modifier les cellules ℓ_{ii}, ℓ_{ij}, et ℓ_{ji}. Conséquemment, il ne reste qu'à modifier une seule cellule, de la façon suivante : $\ell'_{jj} = \ell_{jj} - 1$. Donc, la valeur présente dans cette cellule est la seule différence entre les matrices L' et L. La mise à jour est illustrée sur la figure 3.7, où $i = 1$ et $j = 2$. Sans les premières rangée et colonne, une seule mise à jour est réellement nécessaire.

La *formule Sherman-Morrison*[40] stipule que si une matrice M' est obtenue à partir d'une matrice M en remplaçant sa j^e colonne, $(M)_j$, par un vecteur colonne u alors

$$\det(M') = (1 + e_j^\top M^{-1}(u - (M)_j))\det(M).$$

Dans notre cas, $(u - (M)_j) = -e_j$, donc l'expression de droite de l'équation précédente

$$L(G) = \begin{pmatrix} 3 & -1 & -1 & -1 \\ -1 & 2 & -1 & 0 \\ -1 & -1 & 3 & -1 \\ -1 & 0 & -1 & 2 \end{pmatrix} \qquad L(G-(i,j)) = \begin{pmatrix} 2 & 0 & -1 & -1 \\ 0 & 1 & -1 & 0 \\ -1 & -1 & 3 & -1 \\ -1 & 0 & -1 & 2 \end{pmatrix}$$

Figure 3.7 Mise à jour de la matrice Laplacienne pour $i = 1$ et $j = 2$

se simplifie : $(1 - e_j^\top M^{-1} e_j)\det(M) = (1 - m_{jj}^{-1})\det(M)$.

Finalement, nous avons

$$\sigma((i,j), 0, \texttt{arbre de recouvrement}(G,T)) = \frac{L'_{ii}}{L_{ii}} = \frac{(1 - m_{jj}^{-1})L_{ii}}{L_{ii}} = 1 - m_{jj}^{-1},$$

et évidemment

$$\sigma((i,j), 1, \texttt{arbre de recouvrement}(G,T)) = m_{jj}^{-1}.$$

Calculer les densités de solutions s'avère donc très simple : pour chaque arête (i,j) adjacente à un sommet i, de façon à ce que $j < i$ (respectivement $j > i$), la valeur correspondante est la j^e (respectivement $(j-1)^e$, car la colonne et rangée i sont retirées) cellule sur la diagonale de la matrice inverse de M, la sous-matrice de la matrice Laplacienne L obtenue en enlevant la i^e rangée et colonne. Répéter ces opérations pour chaque sommet d'un ensemble recouvrant de sommets (vertex cover) permet d'obtenir la densité de solution relative à chaque arête. Cet ensemble a une taille d'au plus n (car il y a n sommets dans le graphe), donc l'ensemble de la procédure peut être fait en $\mathcal{O}(\gamma n^3)$, où gamma est dans l'ordre de $\mathcal{O}(n)$. Une couverture de sommets peut être définie de la façon suivante :

Définition 3.1.2 (Couverture de sommets [11])
Dans un graphe, un sommet couvre une arête si celle-ci lui est adjacente. Une couverture de sommet est un sous-ensemble de sommets qui couvre toutes les arêtes du graphe. La couverture de sommet minimum est celle dont le nombre de sommets en faisant partie est le plus petit. Trouver une telle couverture est NP-difficile.

Afin de pouvoir calculer la densité de solutions pour toutes les arêtes, il est nécessaire de répéter le calcul pour au moins chaque sommet faisant partie de la couverture de sommets minimum. Comme calculer cette couverture est difficile et coûteux, une couverture incluant un plus grand nombre de sommets est utilisée, ce qui est quand même plus rapide en pratique que de répéter le calcul pour l'ensemble des sommets du graphe.

La figure 3.8 illustre l'application de la formule de Sherman-Morrison. La valeur à la position $(2,2)$ de la matrice Laplacienne originale (et $(1,1)$ dans la sous-matrice M, dont la

rangée et la colonne 1 a été retirées) est mise à jour. Le vecteur colonne u, qui est identique à la colonne $(M)_j$, sauf pour la valeur à la position 1, vient remplacer la colonne $(M)_j$. Ce faisant, seule la valeur à la position $(1,1)$ change dans la nouvelle sous-matrice M', ce qui correspond à ce que la formule Sherman-Morrison énonce.

$$M = \begin{pmatrix} 2 & -1 & 0 \\ -1 & 3 & -1 \\ 0 & -1 & 2 \end{pmatrix} \qquad u = \begin{pmatrix} 1 \\ -1 \\ 0 \end{pmatrix}$$

$$M' = \begin{pmatrix} 1 & -1 & 0 \\ -1 & 3 & -1 \\ 0 & -1 & 2 \end{pmatrix}$$

Figure 3.8 Exemple d'application de la formule Sherman-Morrison

Soit M^{-1} la sous-matrice inversée de L, obtenue en retirant la première rangée et la première colonne, comme illustré à la figure 3.9. En inversant cette matrice, les densités de solutions pour les arêtes adjacentes au sommet 1 sont obtenues sur la diagonale. Si nous portons attention à l'arête $(1,2)$, on remarque qu'il y a bel et bien 5 arbres recouvrants qui incluent cette arête (en vert), comme illustré à la figure 3.10.

$$M^{-1} = \begin{pmatrix} 5/8 & 2/8 & 1/8 \\ 2/8 & 4/8 & 2/8 \\ 1/8 & 2/8 & 5/8 \end{pmatrix}$$

Figure 3.9 Matrice M inversée

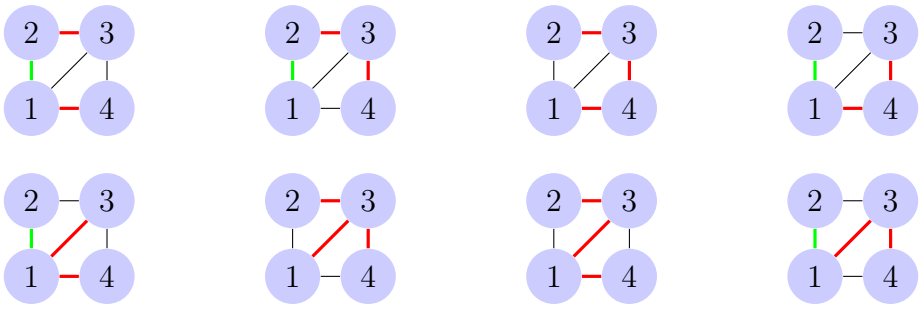


Figure 3.10 Arbres recouvrants (en rouge et vert) incluant l'arête $(1,2)$(en vert)

3.2 Algorithme de dénombrement pour les graphes orientés

L'extension naturelle d'un arbre de recouvrement pour un graphe orienté est une arborescence. Une arborescence implique habituellement une racine, qui est définie comme suit :

Définition 3.2.1 (Racine [11])
Une racine est un sommet à partir duquel il existe au moins un chemin vers tous les autres sommets du graphe.

Un graphe orienté peut également admettre un ou plusieurs puits.

Définition 3.2.2 (Puits [11])
Un puits est exactement l'inverse d'une racine. Pour chaque sommet, il existe au moins un chemin les reliant au puits.

Définition 3.2.3 (Arborescence [11])
Une arborescence est un graphe orienté qui est un arbre (sans l'orientation des arcs) et qui a une racine.

Dans nos travaux, nous considérerons les anti-arborescences, qui sont définies de la façon suivante :

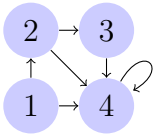
Définition 3.2.4 (Anti-arborescence [11])
Une anti-arborescence est un graphe orienté qui est un arbre (sans l'orientation des arcs) et qui a un puits.

Nos algorithmes de dénombrement pour les graphes orientés ont été conçus pour un type de graphe particulier : les "sink-rooted graphs" :

Définition 3.2.5 (Sink-rooted graph [11])
Soit $G = (V, E)$ un graphe orienté. Il peut être appelé un sink-rooted graph s'il y a un puits $s \in V$ auquel tous les autres sommets sont connectés.

Dans notre cas, le puits s aura également une boucle sur lui-même (self-loop), ce qui implique $deg^+(s) = 1$. Le graphe illustré à la figure 3.2 est bel et bien un "sink-rooted graph", étant donné que le sommet 4, le puits, n'admet aucune arête sortante (outre la boucle) et que chaque autre sommet a un chemin le reliant au puits.

La *matrice Laplacienne* est également définie pour les graphes orientés. Comme pour les graphes non orientés, elle est formée par la soustraction de la matrice d'adjacence de G à la matrice diagonale de degré. Pour le cas orienté, la matrice diagonale de degré ne considère que le degré sortant des sommets. Donc, dans cette matrice, la valeur de la i^e cellule correspond



$$L = \begin{pmatrix} 2 & -1 & 0 & -1 \\ 0 & 2 & -1 & -1 \\ 0 & 0 & 1 & -1 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Figure 3.11 Graphe orienté et sa matrice Laplacienne correspondante

au degré sortant (deg_+) du sommet i dans G. La figure 3.11 illustre un graphe orienté et sa matrice Laplacienne correspondante.

Le *mineur* $-(i,j)$ d'une matrice carrée M, dénoté M_{ij} est le déterminant de la sous-matrice obtenue en retirant la i^e rangée et j^e colonne de M. La matrice Laplacienne a la propriété suivante : son mineur-(s,s), pour la rangée et la colonne s correspondant au puits s du graphe orienté, est égal au nombre d'anti-arborescences du graphe correspondant.

Theoreme 3.2.1 (Kirchhoff's Matrix-Tree [41])
Soit $A(G)$ le nombre d'anti-arborescences du graphe G ayant n sommets, dont s est un puits. Donc,

$$A(G) = L_{ss}.$$

Donc, le nombre de solutions respectant la contrainte `anti-arborescence` peut aussi être calculé comme le déterminant d'une matrice carrée $(n-1) \times (n-1)$, en $\mathcal{O}(n^3)$.

Si nous retirons les dernières rangée et colonne (correspondant aux colonne et rangée du puits) de la matrice Laplacienne à la figure 3.11, le mineur résultant est $2 \times (2 \times 1 - (-1) \times (0)) - (-1) \times (0 \times 1 - (-1) \times 0) = 4$. Les 4 anti-arborescences sont illustrées à la figure 3.13.

$$L_{44} = \begin{pmatrix} 2 & -1 & 0 \\ 0 & 2 & -1 \\ 0 & 0 & 1 \end{pmatrix}$$

Figure 3.12 L_{44}, pour le digraphe 3.2

Nous sommes intéressés par le calcul de la densité de solution pour un arc $(i,j) \in E$. Comme pour le cas non orienté, une façon de calculer la densité de solution est de compter le nombre d'anti-arborescences n'utilisant pas cet arc, $A(G \setminus \{(i,j)\})$, et ensuite diviser ce nombre par le nombre total d'anti-arborescences. Le résultat est la densité de solution de la

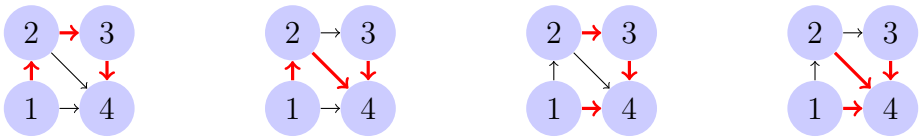


Figure 3.13 Anti-arborescences (arcs en rouge) présentes dans le digraphe 3.2

variable correspondante à laquelle une valeur de 0 (i.e. $(i,j) \notin A$) est affectée :

$$\sigma((i,j), 0, \texttt{Anti-arborescence}(G, A)) = \frac{A(G \setminus \{(i,j)\})}{A(G)}.$$

Nous pouvons donc dériver la densité de solution pour le cas orienté. Soit $L' = L(G \setminus \{(i,j)\})$. Pour le cas orienté, L et L' sont identiques sauf pour les celllules ℓ_{ii} et ℓ_{ij}. Les changements sont donc localisés sur une seule rangée, tel qu'illustré par la figure 3.14, où $i = 1$ et $j = 2$.

$$L(G) = \begin{pmatrix} 2 & -1 & 0 & -1 \\ 0 & 2 & -1 & -1 \\ 0 & 0 & 1 & -1 \\ 0 & 0 & 0 & 1 \end{pmatrix} \qquad L(G - \{i,j\}) = \begin{pmatrix} 1 & 0 & 0 & -1 \\ 0 & 2 & -1 & -1 \\ 0 & 0 & 1 & -1 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Figure 3.14 Mise à jour de la matrice Laplacienne pour $i = 1$ et $j = 2$

Comme tous les changements sont localisés dans une seule rangée de M, nous transposons la matrice (son déterminant reste invariant après une transposition) et appliquons la *formule Sherman-Morrison* comme auparavant :

$$\det(M') = (1 + e_i^\top (M^{-1})^\top (u - (M)_i)^\top) \det(M^\top).$$

Il y a deux cas possibles. Dans le premier, la destination de l'arc est un sommet quelconque. Dans le deuxième, la destination du sommet est le puits.

Dans le premier cas, $(u - (M)_i)^\top = e_j - e_i$ donc l'expression de droite de l'équation précédente se simplifie à

$$(1 + e_i^\top (M^\top)^{-1} (e_j - e_i)) \det(M)$$

et ensuite

$$(1 + e_i^\top (M^\top)^{-1}(e_j) + (1 + e_i^\top (M^\top)^{-1})(-e_i))\det(M)$$

.

En simplifiant, nous obtenons :

$$(1 + (m_{ij}^\top)^{-1} - (m_{ii}^\top)^{-1})\det(M) = (1 + m_{ji}^{-1} - m_{ii}^{-1})\det(M)$$

.

Donc finalement, nous avons

$$\sigma((i,j), 0, \texttt{Anti-Arborescence}(G, A)) = \frac{L'_{ss}}{L_{ss}} = \frac{(1 + m_{ji}^{-1} - m_{ii}^{-1})L_{ii}}{L_{ii}} = 1 + m_{ji}^{-1} - m_{ii}^{-1},$$

et évidemment

$$\sigma((i,j), 1, \texttt{Anti-asrborescence}(G, A)) = m_{ii}^{-1} - m_{ji}^{-1}.$$

Dans le deuxième cas, $(u - (M)_i)^\top = -e_i$ donc l'expression de droite de l'équation précédente se simplifie à

$$(1 + e_i^\top (M^\top)^{-1}(-e_i))\det(M)$$

et ensuite

$$(1 + e_i^\top (M^\top)^{-1}(-e_i))\det(M)$$

.

En simplifiant, nous obtenons :

$$(1 - (m_{ii}^\top)^{-1})\det(M) = (1 - m_{ii}^{-1})\det(M)$$

.

Donc finalement, nous avons

$$\sigma((i,j), 0, \texttt{Anti-arborescence}(G, A)) = \frac{L'_{ss}}{L_{ss}} = \frac{(1 - m_{ii}^{-1})L_{ii}}{L_{ii}} = 1 - m_{ii}^{-1},$$

et évidemment

$$\sigma((i,j), 1, \texttt{Anti-arborescence}(G, A)) = m_{ii}^{-1}.$$

Voici un exemple d'application de la formule Sherman-Morrison, pour le cas orienté, à la

figure 3.15.

Supposons j = 2

$$M = \begin{pmatrix} 2 & -1 & 0 \\ 0 & 2 & -1 \\ 0 & 0 & 1 \end{pmatrix} \qquad u = \begin{pmatrix} 1 & 0 & 0 \end{pmatrix}$$

$$M' = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 2 & -1 \\ 0 & 0 & 1 \end{pmatrix}$$

Figure 3.15 Exemple d'application de la formule Sherman-Morrison

Calculer la densité de solution pour la contrainte d'anti-arborescence s'avère aussi simple que pour la contrainte d'arbre de recouvrement.

Exemple 3.1

Soit M la sous-matrice de L obtenue en retirant la dernière colonne et rangée, comme à la figure 3.12. Alors

$$M^{-1} = \begin{pmatrix} 1/2 & 1/4 & 1/4 \\ 0 & 1/2 & 1/2 \\ 0 & 0 & 1 \end{pmatrix}$$

Et la densité pour l'arc $(1,2)$ est simplement : $m_{11}^{-1} - m_{21}^{-1} = \frac{1}{2} - 0 = \frac{1}{2}$. Pour un arc dirigé vers le puits, l'arc $(2,4)$, nous avons : $m_{22}^{-1} = \frac{1}{2}$.

La figure 3.16 illustre quelles anti-arborescences incluent l'arc $(1,2)$ (en vert, première rangée) et l'arc $(2,4)$ (en jaune, deuxième rangée).

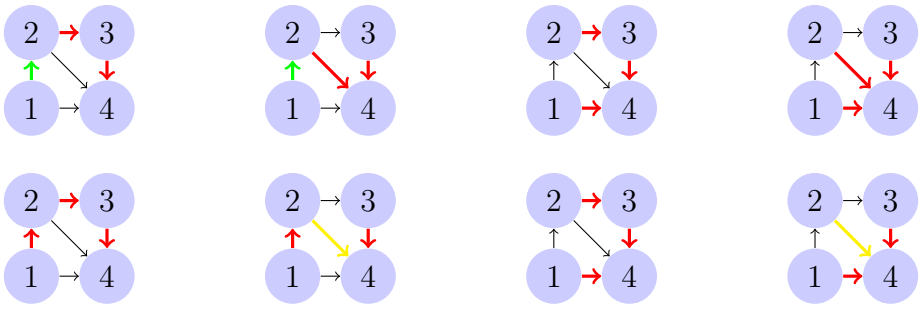


Figure 3.16 Anti-arborescences du digraphe de la figure 3.2, incluant l'arc $(1,2)$ (vert, première rangée) et l'arc $(2,4)$ (jaune, deuxième rangée)

CHAPITRE 4

MISE EN OEUVRE

Dans ce chapitre, les détails nécessaires à la mise en oeuvre des heuristiques de branchement basées sur le dénombrement sont abordés. Dans la section 4.1, les modèles et contraintes nécessaires à la recherche de STAs contraints sont détaillés. Dans la section 4.2, l'intégration des algorithmes de dénombrement à la recherche est expliquée. Enfin, dans la section 4.3, quelques problèmes reliés à l'implémentation et leur solution sont donnés.

4.1 Modèles et contraintes

Dans cette section, les détails relatifs à la modélisation des STAs sont donnés. Pour tous nos modèles, nos structures de données sont réversibles, afin qu'elles puissent être restaurées lors d'un backtrack. La librairie Ilog[1], utilisée pour modéliser les STAs, propose des variables de type *IlcRev*, qui conservent en mémoire toutes les valeurs prises par les éléments de la structure. Lorsqu'un backtrack est nécessaire, les structures retrouvent automatiquement l'état qu'elles avaient au moment où la décision de branchement est revue. Donc, aucun calcul n'est nécessaire pour restaurer les structures de données à l'état approprié.

Dans la sous-section 4.1.1, les modèles utilisés pour décrire les arbres de recouvrement contraints sont abordés. Dans la sous-section 4.1.2, les modèles pour résoudre les problèmes d'anti-arborescence sont décrits.

4.1.1 Cas non orienté

De façon générale, un arbre de recouvrement est représenté par l'ensemble des arêtes qui le constitue. Nous utilisons donc un vecteur de variables binaires, où chaque arête possible est représentée. Conséquemment, notre vecteur a une taille de n^2, où n est le nombre de sommets du graphe. Conséquemment, ce vecteur est symétrique, car pour chaque arête, il y a deux cellules dédiées, étant donné que chaque arête est représentée comme adjacente aux deux sommets qu'elle relient. Nous indiquons qu'une variable est requise par l'arbre de recouvrement par la valeur 1, tandis qu'une arête inexistante ou interdite est représentée par la valeur 0. Le tableau de variables est initialisé en fonction du graphe dans lequel un arbre de recouvrement est recherché.

Une solution au problème correspond au choix de $n - 1$ arêtes parmi celles disponibles, sans qu'un cycle soit créé dans le graphe. Cela correspond à notre première contrainte :

arbre de recouvrement (Spanning Tree). Cette contrainte borne d'abord le nombre d'arêtes pouvant être choisies et garantit qu'aucun cycle ne sera formé, étant donné que le graphe reste connexe. Pour borner le nombre d'arêtes, il suffit d'une contrainte arithmétique, qui somme le nombre de 1 dans le tableau de variables et la compare à la borne souhaitée : n-1.

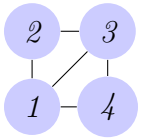
Définition 4.1.1 (Contrainte arbre de recouvrement)
Soit un graphe non orienté de n sommets. Le tableau de variables var *est une matrice de variables binaires de taille* n^2*. La contrainte d'arbre de recouvrement prend la forme suivante :*

$$sum(var) == n - 1$$

Cela signifie qu'il y aura un choix de n-1 arêtes exactement dans le graphe, ce qui forme l'arbre de recouvrement, à condition qu'aucun cycle ne soit formé.

Prévenir la formation de cycles dans le graphe s'avère un peu plus complexe, car une contrainte ne peut pas être utilisée directement. C'est par le filtrage que cette garantie prend forme. Pour ce faire, les composantes connexes formées par le choix des arêtes sont conservées en mémoire et mises à jour après chaque choix d'arête. Lorsqu'une arête est choisie, cela implique que les deux sommets qu'elle relie entrent dans la même composante connexe. Prévenir la formation d'un cycle revient à interdire le choix d'une arête reliant deux sommets qui font partie de la même composante connexe. Donc, initialement, chaque sommet forme sa propre composante connexe, dont il est l'unique représentant. Lorsqu'une arête est choisie, les deux sommets entrent dans la même composante connexe, soit celle du plus petit sommet (par convention). Pour joindre deux sommets dans la même composante connexe, il suffit de parcourir le tableau de composantes connexes et de remplacer les valeurs égale à celle du plus grand sommet par celle du plus petit sommet. Ensuite, le tableau des variables est parcouru pour retirer les arêtes reliant deux sommets figurant dans la même composante connexe. Cela correspond à retirer la valeur 1 du domaine de ces variables, comme illustré dans l'exemple 4.1. Le retrait des arêtes incohérentes est également fait dans la matrice Laplacienne. Le détail de ces mises à jour est donné dans la section 4.2.1.

Exemple 4.1 (Modèle pour l'arbre de recouvrement)
Soit le graphe suivant :



Voici la matrice de variables de branchement y étant associée :

$$var = \begin{pmatrix} \{0\} & \{0,1\} & \{0,1\} & \{0,1\} \\ \{0,1\} & \{0\} & \{0,1\} & \{0\} \\ \{0,1\} & \{0,1\} & \{0\} & \{0,1\} \\ \{0,1\} & \{0\} & \{0,1\} & \{0\} \end{pmatrix}$$

Étant donné qu'il y a 4 sommets, la matrice contient 16 variables binaires. Comme toutes les arêtes excluant les "self-loop" ((1,1), (2,2), (3,3), (4,4)), et (2,4), puis sa symétrique (4,2), font partie du graphe, la matrice admet les valeurs 0 et 1 pour toutes les autres cellules.

Voici le tableau qui donne la composante connexe de chaque sommet :

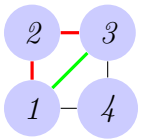
$$cc = \begin{pmatrix} 1 & 2 & 3 & 4 \end{pmatrix}$$

Si l'arête (1,2) est choisie, la matrice de variables de branchement sera mise à jour (cases (1,2) et (2,1))et le tableau de composante connexe le sera également.

$$var = \begin{pmatrix} \{0\} & \{1\} & \{0,1\} & \{0,1\} \\ \{1\} & \{0\} & \{0,1\} & \{0\} \\ \{0,1\} & \{0,1\} & \{0\} & \{0,1\} \\ \{0,1\} & \{0\} & \{0,1\} & \{0\} \end{pmatrix}$$

$$cc = \begin{pmatrix} 1 & 1 & 3 & 4 \end{pmatrix}$$

Si ensuite l'arête (2,3) est choisie, l'arête (1,3) doit être retirée du graphe, car autrement, les sommets 1 et 3 pourraient être directement reliés, ce qui formerait un cycle :



Conséquemment, les structures sont mises à jour de la façon suivante :

$$var = \begin{pmatrix} \{0\} & \{1\} & \{0\} & \{0,1\} \\ \{1\} & \{0\} & \{1\} & \{0\} \\ \{0\} & \{1\} & \{0\} & \{0,1\} \\ \{0,1\} & \{0\} & \{0,1\} & \{0\} \end{pmatrix}$$

$$cc = \begin{pmatrix} 1 & 1 & 1 & 4 \end{pmatrix}$$

4.1.2 Cas orienté

De façon générale, tout comme pour un arbre de recouvrement, une anti-arborescence est représentée par l'ensemble des arcs qui la constitue. Nous utilisons donc une matrice de variables binaires, où chaque arc possible est représenté, de taille n^2. Contrairement à la matrice pour le cas non orienté, celle pour le cas orienté n'est pas symétrique, donc chaque arc possible n'est représenté qu'une seule fois. Sa numérotation est en fonction de sa source, vers sa destination. Comme pour le cas non orienté, nous indiquons qu'une variable est requise par l'arborescence par la valeur 1, tandis qu'un arc inexistant ou interdit est représenté par la valeur 0. Le tableau de variables est également initialisé en fonction du graphe dans lequel l'anti-arborescence est recherchée.

La construction d'une anti-arborescence correspond au choix de $n-1$ arcs parmi ceux disponibles, sans qu'un cycle soit créé dans le graphe. Cela correspond à notre première contrainte : *anti-arborescence*. Tout comme la contrainte d'arbre de recouvrement, cette contrainte borne d'abord le nombre d'arcs pouvant être choisis et garantit qu'aucun cycle ne sera formé. Pour borner le nombre d'arcs, il suffit d'une contrainte arithmétique, qui somme le nombre de 1 dans la matrice de variables de branchement et qui compare cette somme à la borne souhaitée.

Limiter le nombre d'arcs choisis est insuffisant pour garantir la formation d'une anti-arborescence. Il faut également garantir qu'il n'existe qu'un seul chemin, à partir de chaque sommet, qui atteint le puits. Une autre façon de voir cette contrainte est de limiter le degré sortant de chaque sommet à 1. Si le degré sortant de chaque sommet est égal à 1 (sauf le puits, qui, évidemment, n'admet aucun arc sortant), il est impossible que plusieurs chemins mènent un sommet vers le puit, car il faut au minimum $n-1$ arcs pour que chaque sommet puisse être connecté au puit. Conséquemment, lorsqu'un arc est choisi, tous les arcs sortant de son sommet source sont retirés la matrice de variables. Pour ce faire, il suffit de le parcourir et de retirer 1 du domaine de tous les arcs dont le sommet source correspond à celui du sommet choisi. Le degré sortant d'un sommet est calculé en sommant la valeur prise par tous les arcs ayant ce sommet comme source. Comme seules les variables correspondantes aux arcs pris ont la valeur 1, la somme des valeurs de toutes les variables partageant le même sommet source revient à calculer le degré sortant. Pour empêcher les cycles, il faut faire appel au filtrage, comme dans le cas non orienté. En combinant le filtrage, le degré sortant limité à un et le nombre d'arcs choisis à $n-1$, il n'y a pas de cycles ni de multiples chemins dans la solution, ce qui en fait une anti-arborescence correcte.

Définition 4.1.2 (Contrainte anti-arborescence)
Soit un graphe orienté de n sommets. La matrice de variables binaires var *est de taille* n^2 *La*

contrainte d'anti-arborescence prend la forme suivante :

$$sum(var) == n - 1$$

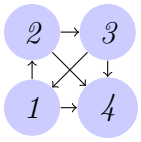
Cela signifie qu'il y aura un choix de n-1 arcs exactement dans le graphe. Chaque sommet v (excluant le puit) admet également la contrainte suivante :

$$deg(v) == 1 \quad \forall v \neq s$$

Cela signifie qu'un seul arc peut prendre le sommet v comme source.

Le filtrage d'arcs pouvant former des cycles est fait de façon presque identique à celui pour le cas non orienté. Tout comme pour le cas non orienté, les composantes connexes formées par le choix des arcs sont conservées en mémoire et mises à jour après chaque choix d'arc. Initialement, chaque sommet forme sa propre composante connexe, dont il est l'unique représentant. Lorsqu'un arc est choisi, les deux sommets entrent dans la même composante connexe, soit celle du plus grand sommet (par convention). Comme pour le cas non orienté, il suffit de parcourir le vecteur de composantes connexes et de remplacer les valeurs correspondant à l'un ou l'autre des sommets reliés par l'arc par celle du sommet le plus grand. Ensuite, le tableau des variables est parcouru pour retirer les arcs dont la source et la destination figurent dans la même composante connexe. Cela correspond à retirer la valeur 1 du domaine de ces variables, comme illustré dans l'exemple 4.2. La matrice Laplacienne doit également être cohérente. Les mises à jour nécessaires sont décrites dans la sous-section 4.2.3. Le retrait des arêtes incohérentes est également fait dans la matrice Laplacienne, dont le détail est donné dans la sous-section 4.2.3.

Exemple 4.2 (Modèle pour l'anti-arborescence)
Soit le graphe suivant :



Voici le tableau de variables y étant associé :

$$var = \begin{pmatrix} \{0\} & \{0,1\} & \{0\} & \{0,1\} \\ \{0\} & \{0\} & \{0,1\} & \{0,1\} \\ \{0,1\} & \{0\} & \{0\} & \{0,1\} \\ \{0\} & \{0\} & \{0\} & \{0\} \end{pmatrix}$$

Le tableau n'étant pas symétrique, seuls les arcs présents apparaissent, par exemple l'arc $(0,1)$*, à la cellule* $(0,1)$*. Évidemment, les arcs qui seraient des "self-loop" sont absents.*

Voici le tableau qui donne la composante connexe de chaque sommet :

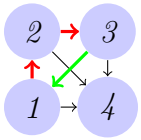
$$cc = \begin{pmatrix} 1 & 2 & 3 & 4 \end{pmatrix}$$

Si l'arête $(1,2)$ *est choisie, les variables seront mises à jour (cellule* $(1,2)$ *dans le tableau. Le tableau de composante connexe le sera également.*

$$var = \begin{pmatrix} \{0\} & \{1\} & \{0\} & \{0\} \\ \{0\} & \{0\} & \{0,1\} & \{0,1\} \\ \{0,1\} & \{0\} & \{0\} & \{0,1\} \\ \{0\} & \{0\} & \{0\} & \{0\} \end{pmatrix}$$

$$cc = \begin{pmatrix} 2 & 2 & 3 & 4 \end{pmatrix}$$

Si ensuite l'arête $(2,3)$ *est choisie, l'arête* $(3,1)$ *doit être retirée du graphe, car autrement, les sommets* 3 *et* 1 *pourraient être directement reliés, ce qui formerait un cycle :*



Conséquemment, les structures sont mises à jour de la façon suivante :

$$var = \begin{pmatrix} \{0\} & \{1\} & \{0\} & \{0\} \\ \{0\} & \{0\} & \{1\} & \{0\} \\ \{0\} & \{0\} & \{0\} & \{0,1\} \\ \{0\} & \{0\} & \{0\} & \{0\} \end{pmatrix}$$

$$cc = \begin{pmatrix} 3 & 3 & 3 & 4 \end{pmatrix}$$

4.1.3 Calcul de la densité de solution

Pour mettre en place le calcul de densité de solution, il est nécessaire d'implémenter les contraintes d'arbre de recouvrement et d'anti-arborescence. En utilisant la structure ("framework") présente dans le laboratoire, nous avons implémenté un type de contrainte particulier, que nous avons nous-même défini, qui inclut les méthodes nécessaires au calcul de la densité de solutions. Dans ce framework, une contrainte prend la forme d'une classe C++. Donc, la

contrainte particulière, *Countable Constraint*, est une classe abstraite de laquelle toutes les contrainte de dénombrement héritent, qui a l'interface définie à la figure 4.1.

```
class CountableConstraint
{
    CountableConstraint()
    recount();
    getDensity();
}
```

Figure 4.1 Interface de Countable Constraint

Une contrainte de base admet un constructeur, une méthode post, qui indique quand la propagation est appelée ainsi qu'une méthode propagate, qui effectue le filtrage approprié. Notre type de contrainte particulier ajoute à celles-ci les méthodes recount, qui fait le dénombrement de solutions puis le calcul des densités et getDensity, qui retourne la densité de solution pour une paire variable-valeur. La figure 4.2 donne l'interface d'une contrainte qui hérite de Countable Constraint.

```
class NewConstraint : CountableConstraint
{
    NewConstraint()
    post();
    propagate();
    recount();
    getDensity();
}
```

Figure 4.2 Interface de classe héritant de Countable Constraint

Pour la contrainte d'arbre de recouvrement, les fonctions remplissent les tâches indiquées à la définition 4.1.3 :

Définition 4.1.3 (Fonctions de la contrainte arbre de recouvrement)
Voici les fonctions et leurs tâches respectives :

- *constructeur*
  - *Initialise la matrice Laplacienne*
  - *Calcule la couverture de sommets, qui est indispensable pour calculer la densité de solution de chaque arête.*

  - *Ajoute au modèle la contrainte sur le tableau de variables, indiquant que seules $n-1$ arêtes peuvent être choisies.*
- *post*
  - *Indique quand la propagation doit avoir lieu, soit à chaque fois que le domaine d'une variable est modifié. En effet, puisqu'il s'agit de variables binaires, le retrait d'une valeur dans le domaine implique que cette variable est fixée, donc que l'arête correspondante est requise ou interdite dans l'arbre.*
- *propagate*
  - *Filtre des arêtes incohérentes pour éviter les cycles.*
  - *Met à jour la matrice Laplacienne*
- *recount*
  - *Calcule la densité de solution, au complet ou de façon incrémentale.*
- *getDensity*
  - *Retourne la densité de solution pour une paire variable-valeur. Évite d'avoir à recalculer la densité à chaque fois qu'elle est requise.*

La structure de la contrainte d'anti-arborescence est pratiquement identique à celle de l'arbre de recouvrement. Pour cette contrainte, les fonctions remplissent les tâches données par la définition 4.1.4 :

Définition 4.1.4 (Fonctions de la contrainte anto-arborescence)
Voici les fonctions et leurs tâches respectives :

- *constructeur*
  - *Initialise la matrice Laplacienne*
  - *Ajoute au modèle la contrainte sur le tableau de variables, indiquant que seules $n-1$ arêtes peuvent être choisies.*
- *post*
  - *Indique quand la propagation doit avoir lieu, soit à chaque fois que le domaine d'une variable est modifié.*
- *propagate*
  - *Filtre des arcs incohérents pour éviter les cycles.*
  - *Met à jour la matrice Laplacienne*
- *recount*
  - *Calcule la densité de solution, au complet ou de façon incrémentale.*
- *getDensity*
  - *Retourne la densité de solution pour une paire variable-valeur.*

Comme la densité de solution doit être connue pour chaque paire variable-valeur, il est plus

judicieux de la conserver en mémoire que de la calculer à chaque fois qu'elle est nécessaire. Pour cette raison, une structure réversible est utilisée pour conserver la densité de solution, sous la forme d'un tableau de nombres en points flottants.

Que cela soit pour le cas orienté ou le cas non orienté, le calcul de la densité de solution implique une inversion de la matrice Laplacienne, comme décrit au chapitre 3. Comme la densité de solution doit être calculée à nouveau après chaque branchement, il est impératif que ce calcul soit le plus performant possible. Dans cette optique, nous avons utilisé la librairie ALGLIB[7], qui gère entièrement l'inversion de matrice. Cette librairie utilise la *décomposition LU*, qui est beaucoup plus rapide que l'inversion directe. Il suffit d'utiliser les variables du format de la librairie, ce qui amène un léger surcoût, puis de faire appel à la méthode *rmatrixinverse*, qui retourne la matrice inversée. Ensuite, il suffit d'utiliser les cellules de cette matrice inversée pour obtenir les densités de solutions désirées (diagonale pour le cas non orienté et diagonale - terme pour le cas orienté).

Pour le cas non orienté, il faut au plus n inversions de la sous-matrice Laplacienne, soit une par sommet, afin que la rangée et la colonne y correspondant soient retirées de la Laplacienne avant l'inversion. Il est possible de réduire le nombre d'inversions en utilisant une couverture de sommets ("vertex cover"). Il peut s'avérer coûteux de calculer la couverture de sommets. Nous utilisons une approche heuristique qui calcule une couverture raisonnable, mais pas nécessairement minimum, ce qui nous permet de réduire significativement le nombre d'inversions nécessaires. Il faut également maintenir la couverture de sommets tout au long de la recherche de solutions, car plusieurs sommets seront contractés, ce qui sera expliqué ultérieurement. Une façon de faire est d'ajouter un des deux sommets contractés à la couverture de sommets, s'il n'y est pas déjà présent. Pour le cas orienté, il suffit d'une seule inversion de matrice par branchement, car seules les rangée et colonne correspondant au puits doivent être retirées de la matrice Laplacienne.

4.2 Intégration à la recherche arborescente

Dans cette section, nous décrivons quelques détails d'implémentation et les problèmes reliés à celle-ci. Au fur et à mesure que les décisions de branchement sont prises et que le filtrage de domaine est appliqué, certaines arêtes de G seront requises dans T, tandis que d'autres seront interdites. Ces changements doivent se refléter dans nos structures de données.

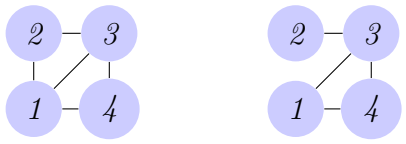
4.2.1 Mise à jour de la matrice Laplacienne, cas non orienté

Si l'arête (i, j) est interdite, elle est simplement retirée du graphe. Pour refléter ce changement dans la matrice Laplacienne, nous ajoutons un aux cellules ℓ_{ij} et ℓ_{ji}, qui représentent

la même arête, étant donné que la matrice est symétrique. Le degré des deux sommets aux extrémités de cette arête doit également être mis à jour, en soustrayant un aux cellules ℓ_{ii} et ℓ_{jj}. Cette procédure est illustrée dans l'exemple 4.3.

Exemple 4.3 (Retrait d'une arête de la matrice Laplacienne, cas non orienté)
Supposons que l'arête $(1, 2)$ *est maintenant interdite pour l'arbre de recouvrement :*

$$L = \begin{pmatrix} 3 & -1 & -1 & -1 \\ -1 & 2 & -1 & 0 \\ -1 & -1 & 3 & -1 \\ -1 & 0 & -1 & 2 \end{pmatrix} \qquad L = \begin{pmatrix} 2 & 0 & -1 & -1 \\ 0 & 1 & -1 & 0 \\ -1 & -1 & 3 & -1 \\ -1 & 0 & -1 & 2 \end{pmatrix}$$



Si l'arête (i, j) est requise, nous contractons les deux sommets qu'elle relie dans le graphe de façon à ce que (i, j) fasse implicitement partie de l'arbre de recouvrement. Contracter deux sommets dans le graphe revient à imposer implicitement l'arête les reliant dans l'arbre de recouvrement. Nous remplaçons les deux sommets par un seul nouveau sommet, qui sera relié à toutes les arêtes adjacentes à l'un ou l'autre des sommets contractés. Évidemment, l'arête reliant les deux sommets contractés est retirée du graphe.

Par convention, nous contractons toujours le sommet le plus grand vers le plus petit. Pour mettre à jour la matrice Laplacienne, nous commençons par ajouter au sommet i toutes les arêtes (j, k) : $\ell_{ik} \leftarrow \ell_{ik} + \ell_{jk}$. Cela peut créer des arêtes multiples, ce qui n'est pas un problème, étant donné que les densités de solution calculées sont également valables pour les multigraphes. Le degré du sommet i, ℓ_{ii}, est mis à jour en conséquence. Ensuite, comme le sommet j fusionne avec le sommet i, nous retirons toutes les arêtes y étant connectées, en remplaçant par zéro la valeur de toutes les cellules de la rangée et colonne j. Finalement, nous ajustons la valeur de la cellule ℓ_{jj} à 1, afin que les mineurs soient calculés correctement lorsque les rangées et colonnes j sont incluses. Autrement, le mineur aurait eu une valeur nulle, car la rangée et colonne j n'aurait inclut que des zéros. Cette procédure est illutrée dans l'exemple 4.4.

Exemple 4.4 (Choix d'une arête dans la matrice Laplacienne, cas non orienté)
Supposons que l'arête $(1, 2)$ *est maintenant requise dans l'arbre de recouvrement. Nous contractons donc le sommet* 2 *avec le sommet* 1.

$$L = \begin{pmatrix} 3 & -1 & -1 & -1 \\ -1 & 2 & -1 & 0 \\ -1 & -1 & 3 & -1 \\ -1 & 0 & -1 & 2 \end{pmatrix} \qquad L = \begin{pmatrix} 3 & 0 & -2 & -1 \\ 0 & 1 & 0 & 0 \\ -2 & 0 & 3 & -1 \\ -1 & 0 & -1 & 2 \end{pmatrix}$$

4.2.2 Mise à jour des densités de solution

Les densités de solutions vont changer au cours de la recherche et il serait utile d'éviter de les recalculer au complet à chaque branchement. Étant donné l'inverse de la matrice M, est-il possible de calculer incrémentalement l'inverse d'une matrice légèrement différente de M'. La *formule Sherman-Morrison*[40] révèle que si M' est obtenu à partir de M en remplaçant sa i^e colonne, $(M)_i$, par un vecteur colonne u comme auparavant, alors

$$M'^{-1} = M^{-1} - \frac{(M^{-1}(u - (M)_i))(e_i^\top M^{-1})}{1 + e_i^\top M^{-1}(u - (M)_i)}.$$

Ceci peut être calculé en $\mathcal{O}(n^2)$, car cela implique la multiplication d'un vecteur colonne de taille n et d'une matrice de taille n^2, ce qui implique de l'ordre de n^2 opérations.

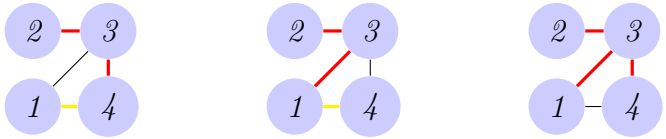
Dans certains cas, nous pouvons réduire cette complexité considérablement. Considérons l'arête interdite (i, j). Pour n'importe quelle arête (i, k), dont la densité de solution a été obtenue à partir de l'inverse de la sous-matrice dont la rangée et colonne i avaient été retirées de L, retirer l'arête (i, j) ne change la valeur que d'une seule cellule dans la sous-matrice, comme nous avons vu précédemment, ce qui simplifie la formule de la façon suivante :

$$M'^{-1} = M^{-1} - \frac{(M^{-1} \cdot (-e_j)) \cdot (e_j^\top \cdot M^{-1})}{1 - m_{jj}^{-1}} = M^{-1} + \frac{1}{1 - m_{jj}^{-1}} \cdot Q$$

où $Q = (q_{hk})$ est une matrice $(n-1) \times (n-1)$ avec $q_{hk} = m_{hj}^{-1} \cdot m_{jk}^{-1}$. Comme nous n'avons besoin que de la k^e cellule sur la diagonale, $m_{kk}^{-1} + (m_{kj}^{-1})^2/(1 - m_{jj}^{-1})$, la mise à jour est faite en temps constant. L'application de cette procédure est illustrée par l'exemple 4.5. Ce qui précède est également applicable pour n'importe quelle arête (j, k) avec la sous-matrice dont la rangée et colonne j ont été retirées de L.

Exemple 4.5 (Mise à jour de la matrice Laplacienne, version incrémentale)
Soit le cas suivant :

– *Densité de solution pour l'arête* $(1,4)$ *est* $\frac{5}{8}$.
– *L'arête* $(1,2)$ *est interdite*
– *La densité de solution mise à jour :* $\frac{5}{8} + (m_{42}^{-1})^2/(1 - m_{22}^{-1}) = \frac{5}{8} + (\frac{1}{8})^2/(1 - \frac{5}{8}) = \frac{2}{3}$.



Mettre en place cette mise à jour incrémentale est relativement complexe. Le but est d'inverser la matrice pour le premier calcul de densité de solutions et de ne jamais réinverser par la suite, d'apporter des changements directement à la matrice inversée. Pour ce faire, il faut conserver la matrice inverse en mémoire, dans une structure réversible. Il faut également conserver en mémoire tous les changements nécessaires de la matrice Laplacienne, non seulement lors de branchements, mais également lorsque des arêtes sont retirées par le filtrage. Pour ce faire, des structures de données supplémentaires, soit des vecteurs correspondants aux vecteurs u modifiés, sont nécessaires. Lorsqu'une arête est retirée ou choisie, les changements doivent être faits sur l'ensemble des matrices inverses, soit un changement par sommet dans la couverture de sommets.

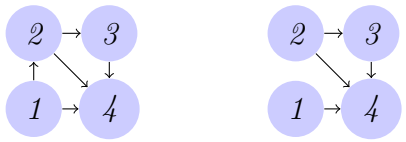
Il est également possible de faire ces mises à jour sous forme d'événements, où l'ensemble des modifications à faire est accumulé et conservé en mémoire. Lorsqu'une densité de solution est requise, toutes des modifications stockées sont apportées sur les matrices inverses, avant le calcul de la densité de solution. Pour la mise en place de cette procédure, il faut des structures qui gardent l'ensemble des modifications à faire, dans l'ordre où elles doivent être faites.

4.2.3 Mise à jour de la matrice Laplacienne, cas orienté

Comme pour le cas non orienté, si l'arête (i,j) est interdite, elle est simplement retirée du graphe. Pour refléter ce changement dans la matrice Laplacienne, nous ajoutons 1 à la cellule ℓ_{ij} uniquement, étant donné que la matrice n'est pas symétrique. Le degré du sommet source de cet arc doit également être mis à jour, en soustrayant un à la cellule ℓ_{ii}. Cette procédure est illustrée dans l'exemple 4.7.

Exemple 4.6 (Retrait d'un arc de la matrice Laplacienne, cas orienté)
Supposons que l'arête $(1,2)$ *est maintenant interdite pour l'anti-arborescence :*

$$L(G) = \begin{pmatrix} 2 & -1 & 0 & -1 \\ 0 & 2 & -1 & -1 \\ 0 & 0 & 1 & -1 \\ 0 & 0 & 0 & 1 \end{pmatrix} \qquad L(G) = \begin{pmatrix} 1 & 0 & 0 & -1 \\ 0 & 2 & -1 & -1 \\ 0 & 0 & 1 & -1 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$



Comme dans le cas non orienté, lorsqu'un arc (i, j) est requis dans l'anti-arborescence, nous le contractons dans le graphe, de façon à ce que (i, j) fasse implicitement partie de celle-ci. Il est impératif que la contraction se fasse de la source vers la destination, car dans la matrice Laplacienne, seuls les arcs sortant d'un sommet sont représentés. Cette réalité est représentée dans le vecteur qui garde en mémoire les sommets contractés et dans la matrice Laplacienne.

Pour mettre à jour la matrice Laplacienne, la rangée j demeure inchangée. Cependant, l'ensemble des arcs qui était dirigés vers le sommet i doivent maintenant être dirigés vers j, afin de refléter la contraction. Pour ce faire, tous les arcs dirigés vers le sommet i sont ajoutés à ceux dirigés vers le sommet j, ce qui revient à transférer les arcs de colonne dans la matrice Laplacienne, pour tous les sommets sauf i et j : $\ell_{kj} \leftarrow \ell_{kj} + \ell_{ki}$. Comme pour le cas non orienté, des arcs multiples peuvent être créés, ce qui n'affecte en rien la validité des densités de solution. Ensuite, comme le sommet i fusionne avec le sommet j, nous retirons tous les arcs sortants de i, en remplaçant par zéro la valeur de toutes les cellules des rangées et colonnes i. Finalement, nous ajustons la valeur de la cellule ℓ_{ii} à 1, afin que les mineurs soient calculés correctement lorsque la rangée et colonne i sont incluses. Autrement, comme dans le cas non orienté, le mineur aurait eu une valeur nulle, car la rangée et colonne i n'aurait inclue que des zéros. Cette procédure est illustrée dans l'exemple 4.7.

Exemple 4.7 (Choix d'un arc dans la matrice Laplacienne, cas orienté)
Supposons que l'arc $(1, 2)$ *est maintenant requis dans l'anti-arborescence. Nous contractons donc le sommet* 1 *avec le sommet* 2*.*

$$L(G) = \begin{pmatrix} 2 & -1 & 0 & -1 \\ 0 & 2 & -1 & -1 \\ 0 & 0 & 1 & -1 \\ 0 & 0 & 0 & 1 \end{pmatrix} \qquad L(G) = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 2 & -1 & -1 \\ 0 & 0 & 1 & -1 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Tout comme pour le cas non orienté, il est possible et moins coûteux de mettre à jour la matrice Laplacienne de façon incrémentale. En appliquant la *formule Sherman-Morrison*[40],

tel que pour le cas non orienté, la mise à jour peut être calculée en $\mathcal{O}(n^2)$, en multipliant le vecteur colonne de taille n par la matrice de taille n^2. Il n'est cependant pas possible de réduire cette complexité, car il faut toujours mettre à jour au minimum deux cellules. En effet, le retrait d'un arc (i, j) affecte les cellules (i, i) et (i, j), qui doivent réduire et augmenter leur valeur de 1, respectivement. Le choix d'un arc (i, j) implique la mise à jour de la rangée i au complet, ainsi que celle des rangée et colonne j. Par conséquent, le gain relatif à l'ajout du calcul incrémental pour le cas orienté est moindre par rapport à celui pour le cas non-orienté, malgré le fait que sa mise en oeuvre soit aussi complexe.

4.3 Complications

Le fait que les densités de solutions et les valeurs dans les sous-matrices à inverser soies conservées dans des nombres à points flottants nous a causé quelques complications, en particulier pour le cas orienté. En effet, étant donné que le calcul des densités de solutions pour le cas orienté implique une opération entre deux valeurs de la sous-matrice inversée, il n'était pas rare d'obtenir des densités plus petites que 0 ou plus grandes que 1, ce qui est impossible et incohérent, étant donné que la densité de solution est un rapport entre le nombre de solutions incluant un arc et le nombre total de solutions. Cette instabilité numérique est due à une opération entre 2 nombres infiniment grands ou deux nombres infiniment petits, en point flottant. Lorsque l'opération est faite entre ces deux nombres (typiquement une soustraction), le résultat est légèrement imprécis. Une densité qui devrait alors prendre une valeur de 0 prend une valeur légèrement différente (relativement à la taille des nombres sur lesquels l'opération est faite). Ce faisant, la densité de solutions pour la paire variable-valeur concernée est erronée, ce qui peut occasionner des branchements inadéquats et par le fait même s'avérer fort coûteux dans la recherche de solutions. Il est donc impératif de remédier à ce problème, sinon la qualité de l'heuristique de branchement basée sur le dénombrement est affectée.

Une autre limite de nos approches reliée aux points flottants est le fait qu'il ne soit pas possible d'obtenir des 0 plats (valeur en point flottant de 0 et non un nombre infiniment petit). Parfois, une arête ne participe à aucune solution, ce qui devrait lui attribuer une densité de solution de 0. Or, comme nous travaillons avec des structures utilisant des points flottants, un tel 0 n'existe pas. Une valeur infiniment petite sera utilisée pour remplacer ce zéro, ce qui ne cause pas de problème à première vue. Le problème est le suivant : au fur et à mesure que la recherche progresse, cette valeur infiniment petite le devient de moins en moins, car l'erreur au niveau du calcul de la matrice inverse se propage. Éventuellement, il est possible d'obtenir des valeurs incohérentes, ce qui encore une fois peut causer des branchements inadéquats.

Une solution à tous ces problèmes est l'introduction d'une valeur ϵ, qui joue le rôle de seuil. Pour les opérations entre deux nombres de grande taille, il suffit de comparer chacun des nombres à ϵ, avant de faire l'opération. Si ces nombres l'excède, la densité de solution qui devrait résulter de l'opération entre ces deux nombres est directement fixée à zéro (nous parlons ici du cas de la soustraction de deux grands nombres, pour le cas orienté en particulier). La même procédure est utilisé si des nombres infiniment petits sont rencontrés. En fixant directement la valeur de la densité de solution, l'instabilité numérique est évitée et n'affecte donc pas les branchements. Les imprécisions des calculs des densités qui devraient donner 0 sont également gérées de cette façon. Si la valeur de densité est inférieure à ϵ, la densité est fixée à zéro.

Lors de l'implémentation des mises à jour incrémentales, nous avons également constaté des problèmes reliés à l'instabilité numérique, qui sont eux aussi reliés à l'utilisation des points flottants. En effet, lorsque deux calculs différents menant au même résultat sont faits à l'aide de nombres en point flottant, il est possible qu'il y ait une légère variation au niveau des valeurs numériques. Or, lorsque les mises à jour sont faites sur les matrices inverses, le calcul fait n'est pas le même que si la matrice est inversée entièrement à nouveau. Conséquemment, les valeurs dans la matrice peuvent être très légèrement différentes (une vingtaine de chiffres derrière la virgule dans notre cas). Néanmoins, puisque l'heuristique de branchement choisit la paire variable-valeur ayant la plus grande densité, il est possible que le branchement soit différent, en fonction de l'approche utilisée, car l'une des paires a une densité légèrement plus grande (alors qu'elle devrait être identique) qu'une autre. Donc, la recherche de solution, et par extension le temps puis le nombre d'échecs, n'étaient pas exactement les mêmes pour certains exemplaires, pour l'approche standard et incrémentale. Malgré cette différence, la qualité des solutions et le nombre de branchement pour y arriver demeurent très semblables pour les deux approches, ce qui n'en fait pas une complication majeure.

CHAPITRE 5

EXPÉRIMENTATIONS ET DISCUSSION

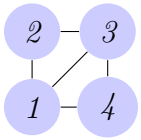
Ce chapitre illustre l'efficacité des heuristiques de branchement basées sur le dénombrement, en comparant une approche basée sur ce type d'heuristique à deux autres approches. La section 5.1 rapporte les résultats d'expérimentations impliquant la contrainte arbre de recouvrement, tandis que la section 5.2 fait de même pour la contrainte d'anti-arborescence.

5.1 Contrainte d'arbre de recouvrement

Pour démontrer l'efficacité de l'utilisation de la densité de solution avec la contrainte `arbre de recouvrement` pour guider l'heuristique de branchement sur quelques problèmes de STA contraints, nous avons considéré la recherche des arbres de recouvrement de degré contraint, dans un graphe donné. Pour ce problème, un cas spécial est celui où un degré maximum égal à 2 est imposé. Ce cas correspond à la recherche d'un chemin hamiltonien. Nous avons généré quelques graphes avec un générateur conçu pour produire des exemplaires avec lesquels il est difficile de trouver un chemin hamiltonien pour des algorithmes ayant recours au backtracking [43].

Le problème d'arbre de recouvrement de degré contraint est modélisé avec la contrainte d'arbre de recouvrement ainsi que par une série de contraintes arithmétiques, venant borner le degré de chaque sommet. Une contrainte arithmétique est donc présente pour chaque sommet, sommant l'ensemble des variables associées aux arêtes adjacentes à ce sommet et bornant cette somme à la valeur souhaitée. Le modèle est illustré par l'exemple 5.1.

Exemple 5.1 (Modèle pour l'arbre de recouvrement de degré contraint)
Soit le graphe suivant :



Voici le vecteur de variables de branchement y étant associé :

$$var = \begin{pmatrix} \{0\} & \{0,1\} & \{0,1\} & \{0,1\} \\ \{0,1\} & \{0\} & \{0,1\} & \{0\} \\ \{0,1\} & \{0,1\} & \{0\} & \{0,1\} \\ \{0,1\} & \{0\} & \{0,1\} & \{0\} \end{pmatrix}$$

Évidemment, la contrainte d'arbre de recouvrement est appliquée sur var. Si nous voulons borner le degré des sommets à 2, il faut ajouter les 4 contraintes arithmétiques suivantes, impliquant les arêtes adjacentes de chaque sommet :

- *Sommet 1 :* $var[1,2] + var[1,3] + var[1,4] \leq 2$
- *Sommet 2 :* $var[2,1] + var[2,3] \leq 2$
- *Sommet 3 :* $var[3,1] + var[3,2] + var[3,4] \leq 2$
- *Sommet 4 :* $var[4,1] + var[4,3] \leq 2$

Nous comparons les heuristiques de branchement suivantes : densité de solution maximale (maxSD), impact-based search (IBS) et sélection de variable et valeur aléatoire (random). L'heuristique maxSD considère l'information relative à la densité de solution de chaque contrainte et branche sur la paire variable-valeur qui correspond à la densité de solution la plus importante. Pour IBS, les impacts sont initialisés avec les informations du noeud racine de l'arbre de recherche. À un noeud particulier de l'arbre de recherche, les cinq meilleures variables en fonction des impacts approximés sont identifiées. Pour ce sous-ensemble, les impacts exacts sont calculés et le branchement est fait sur la meilleure variable (impact le plus élevé) et meilleure valeur(impact le plus bas). Cette procédure est cohérente avec ce qui est suggéré dans la documentation de IBM ILOG solver. Pour random, nous rapportons une moyenne de dix exécutions.

Tableau 5.1 Nombre de backtracks (gauche) et temps en secondes (droite) avant de trouver un arbre de recouvrement de degré maximum égal à 2. Chaque ligne représente une moyenne sur 10 exemplaires.

n	maxSD	IBS	random
15	0.2	229.8	49.0
20	1.5	533.0	976.6
25	2.1	1772.3	5919.6
30	71.7	12517.1	91454.4
35	112.2	18405.4	139861.3

n	maxSD	IBS	random
15	0.029	0.001	0.001
20	0.080	0.012	0.020
25	0.187	0.085	0.173
30	0.815	0.897	1.873
35	1.769	4.742	14.646

Nous avons d'abord généré des graphes aléatoires de 15, 20, 25, 30 et 35 sommets (10 exemplaires chacun). Le générateur garantit l'existence d'un chemin hamiltonien dans ces graphes, en construisant d'abord un chemin hamiltonien dans celui-ci. Ensuite, chacune des arêtes restantes est ajoutée ou non au graphe, en fonction d'une probabilité fixe dont la valeur dépend de la densité souhaitée. Plus le graphe doit être dense, plus la probabilité d'ajouter une arête sera importante. En regardant d'abord la borne de degré égal à 2, le tableau 5.1 de

Tableau 5.2 Nombre de backtracks (gauche) et temps en secondes (droite) avant de trouver un arbre de recouvrement de degré maximum égal à 3. Chaque ligne représente une moyenne sur 10 exemplaires.

n	maxSD	IBS	random
15	0.0	225.6	1.3
20	0.0	315.2	53.2
25	0.0	446.7	882.0
30	0.0	495.1	18589.8
35	0.0	566.8	20001.4

n	maxSD	IBS	random
15	0.039	0.002	0.001
20	0.100	0.013	0.001
25	0.222	0.021	0.311
30	0.441	0.039	0.093
35	0.852	0.063	2.333

gauche indique que l'utilisation de maxSD guide efficacement la recherche vers une solution, en faisant plusieurs ordres de grandeur de backtracks de moins que les deux autres approches. Bien que maxSD apparaisse comme plus lente sur les petits graphes, comme indiqué par le tableau 5.1 de droite, au fur et à mesure que les graphes grossissent, l'approche devient plus rapide que IBS et random.

Nous nous tournons ensuite vers les arbres de recouvrement de degré maximal égal à 3. (voir tableau 5.2). Il est clairement démontré que l'utilisation de la densité de solution pour trouver des arbres de recouvrement dans des graphes aléatoires est une approche très efficace. Un degré maximum de 3 est beaucoup moins restrictif qu'un degré maximum de 2, ce qui implique qu'un plus grand nombre d'arbres de recouvrement auront cette propriété. Conséquemment, les premiers arbres trouvés satisferont toutes les contraintes. Pour tous les graphes, l'heuristique de branchement basée sur la densité de solution trouve un arbre de recouvrement sans backtrack, contrairement aux autres approches. Malgré le fait qu'aucun backtrack n'est fait, maxSD demeure plus lente que IBS sur ces exemplaires étant donné que cette dernière ne requiert que quelques centaines de backtracks.

Nous avons également généré des *crossroad graphs* en utilisant le même générateur de graphe[43]. Ces graphes sont constitués de plusieurs petits sous-graphes aléatoires peu denses, uniquement connectés les uns aux autres par des arêtes ponts ("bridges"). À la figure 5.1, les ponts sont les arêtes $(3, 4)$ et $(6, 7)$. Nous avons généré 10 exemplaires de crossroad graphs contenant 3, 4 et 5 sous-graphes et nous avons tenté d'y trouver un chemin hamiltonien (arbre de recouvrement de degré 2). Les résultats sont présentés au tableau 5.3.

Utiliser maxSD sur ces graphes difficiles s'avère très efficace, car une solution est toujours trouvée avec un grand nombre de backtracks en moins. Pour les exemplaires constitués de 5 sous-graphes, random n'a pas pu trouver de solution pour un seul exemplaire en 2 heures de

Tableau 5.3 Nombre de backtracks (gauche) et temps en secondes (droite) avant de trouver un chemin hamiltonien dans les crossroad graphs. Chaque ligne représente une moyenne sur 10 exemplaires.

n	maxSD	IBS	random
3	0.2	7721.9	8530.5
4	0.1	262011.7	191195.8
5	0.4	162353.0	-

n	maxSD	IBS	random
3	0.085	0.255	0.062
4	0.280	26.379	3.674
5	0.676	586.679	-

temps de calcul. Dans ce cas , maxSD est également plus rapide que les deux autres approches, de plusieurs ordres de grandeur.

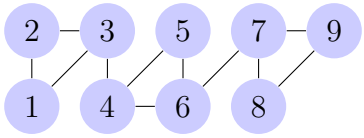


Figure 5.1 Exemple de crossroad graph

Nous avons aussi testé notre approche sur 3 topologies réelles, issues d'un opérateur de reéseau télécom européen. Le tableau 5.4 donne le nombre de sommets et d'arêtes de chaque exemplaire. L'exemplaire b5 est de loin le plus épars et devrait donc être le plus facile, tandis que b3 devrait être le plus difficle, étant donné son plus grand nombre de sommets et d'arêtes.

Tableau 5.4 Nombre de sommets (V) et d'arêtes(E) des trois topologies réelles

exemplaire	V	E
b1	45	63
b3	52	72
b5	52	58

Comme pour les graphes aléaloires, nous avons comparé les trois heuristiques de branchement suivante : maxSD, IBS et random. Nous testons l'efficacité de notre approche en cherchant des arbres de recouvrement dont le degré des sommets est restreint à 2 ou 3. Pour random, nous rapportons une moyenne de dix exécutions.

Tableau 5.5 Nombre de backtracks (gauche) et temps en secondes (droite) avant de trouver un arbre de recouvrement de degré maximum égal à 3.

exemplaire	maxSD	IBS	random
b1	0	664	3160299
b3	0	778	5351123
b5	0	626	89

n	maxSD	IBS	random
b1	1.31	0.08	186.07
b3	3.39	0.14	410
b5	1.86	0.11	0.04

Tableau 5.6 Nombre de backtracks (gauche) et temps en secondes (droite) avant de trouver un arbre de recouvrement de degré maximum égal à 2.

exemplaire	maxSD	IBS	random
b1	-	-	-
b3	127	-	-
b5*	28786	450440	108552

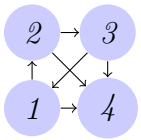
n	maxSD	IBS	random
b1	-	-	-
b3	6.38	-	-
b5*	877.12	85.29	9.21

Le tableau 5.5 démontre que maxSD guide rapidement vers une solution, sans aucun backtrack, tandis que IBS fait quelques backtracks et que random en fait un grand nombre, sauf pour l'exemplaire b5, en raison de sa plus grande simplicité. Cependant, maxSD se montre moins rapide que IBS pour ces exemplaires. Cela est expliqué par la taille des graphes impliqués, qui demande un effort de calcul considérable pour obtenir la densité de solutions à chaque branchement. Lorsque le degré est restreint à 2, le problème devient beaucoup plus difficile, comme illustré dans le tableau 5.6. En effet, pour le premier exemplaire, b1, aucune approche ne réussit à trouver une solution durant les 10 minutes de temps d'exécution allouées. Pour l'exemplaire b3, seul maxSD arrive à trouver une solution durant les 10 minutes de temps de calcul allouées, et ce, en quelques backtracks seulement. Ce résultat indique clairement que maxSD est une approche efficace pour les problèmes de satisfaction difficiles. Pour l'exemplaire b5, il n'existe aucun arbre de recouvrement dont le degré des sommets est restreint à 2. Donc, les trois approches épuisent l'ensemble de l'espace de recherche, ce qui constitue un preuve d'optimalité. MaxSD se montre plus efficace que l'approche aléatoire, mais également considérablement plus lente que IBS. MaxSD guide cependant mieux la recherche, faisait la preuve avec le plus petit nombre de backtracks, soit un ordre de grandeur de moins que les deux autres approches. Encore une fois, la taille du graphe explique le temps de calcul élevé.

5.2 Contrainte d'anti-arborescence

Pour démontrer l'efficacité de l'utilisation de la densité de solution avec la contrainte `anti-arborescence` pour guider l'heuristique de branchement, nous avons considéré la recherche d'anti-arborescences contraintes. De façon similaire au cas non orienté, l'anti-arborescence de degré contraint peut être modélisée par la contrainte anti-aborescence accompagnée par une contrainte arithmétique pour chaque sommet, venant borner le degré entrant de celui-ci. Le modèle de base est illustré par l'exemple 5.2.

Exemple 5.2 (Modèle pour l'anti-arborescence de degré contraint)
Soit le graphe suivant :



Voici le tableau de variables y étant associé :

$$var = \begin{pmatrix} \{0\} & \{0,1\} & \{0\} & \{0,1\} \\ \{0\} & \{0\} & \{0,1\} & \{0,1\} \\ \{0,1\} & \{0\} & \{0\} & \{0,1\} \\ \{0\} & \{0\} & \{0\} & \{0\} \end{pmatrix}$$

Évidemment, la contrainte d'anti-arborescence est appliquée sur var. Si nous voulons borner le degré entrant des sommets à 1, ils faut ajouter les 4 contraintes arithmétiques suivantes, impliquant les arcs entrants de chaque sommet :

- *Sommet 1 :* $var[3,1] \leq 1$
- *Sommet 2 :* $var[1,2] \leq 1$
- *Sommet 3 :* $var[2,3] \leq 1$
- *Sommet 4 :* $var[1,4] + var[2,4] + var[3,4] \leq 1$

Pour nos expériementations, nous avons considéré la recherche de k-arborescences, définie en 5.2.1.

Définition 5.2.1 (k-arborescence[21].)
Soit un graphe orienté $G = (V, E)$, une racine $r \in V$, un sous-ensemble de sommets candidats $S \subseteq V \setminus r$ de terminaux (feuilles possibles dans l'arbre) et un entier $k \leq |S|$. Une k-arborescence est une arborescence, dont la racine est r, qui a exactement k feuilles (sommets de degré sortant de 0) parmi S. Il ne peut pas y avoir de feuilles parmi les sommets non-candidats. Une contrainte sur le degré des sommets dans l'arborescence est également présente.

Nous pouvons résoudre ce problème à l'aide de notre contrainte d'anti-arborescence, en inversant le sens des arcs des exemplaires et en choisissant la racine r comme puits s. Nous modélisons la contrainte des k terminaux devant être choisis dans le sous-ensemble de sommets candidats à l'aide d'une contrainte de type GCC (Global Cardinality Constraint), qui stipule que exactement k sommets parmi ce sous-ensemble doivent avoir un degré entrant nul. La contrainte de degré est ajoutée comme à l'exemple 5.2. Pour s'assurer que seuls les sommets candidats puissent être des feuilles dans l'anti-arborescence, une deuxième contrainte de type GCC est utilisée sur les sommets non candidats, les forçant à avoir un degré entrant supérieur ou égal à 1.

Comme pour la contrainte d'arbre de recouvrement, nous utilisons les 3 topologies réelles, que nous avons converties en graphes orientés. Pour ce faire, nous avons simplement doublé chaque arête, pour en faire un arc dans chaque direction. Le puits choisi est toujours le dernier sommet, soit le sommet n. Le tableau 5.7 illustre le nombre de sommets et d'arcs pour les trois exemplaires.

Tableau 5.7 Nombre de sommets (V) et d'arcs(E) des trois topologies réelles

exemplaire	V	E
b1	45	126
b3	52	144
b5	52	106

Comme pour la contrainte anti-arborescence, nous considérons 3 approches, soit densité de solution maximale (maxSD), impact-based search (IBS), et sélection de variable et valeur aléatoire (random). Pour chacun des trois exemplaires, nous considérons 10 ensembles de sommets candidats différents de 25 sommets, construits de façon aléatoire. Nous fixons le puits au dernier sommet (sommet n). Pour random, nous rapportons une moyenne de dix exécutions.

Comme illustré par le tableau 5.8, maxSD est nettement plus efficace pour les exemplaires b1 et b3, tant au niveau du nombre de backtracks qu'au niveau du temps de calcul, qui domine les deux autres approches de 1 à 2 ordres de grandeur. Pour l'exemplaire b5, qui est le l'exemplaire plus facile, maxSD ne se démarque pas comme pour les deux autres exemplaires, car elle trouve une solution avec le plus petit nombre de backtracks, mais avec un temps de calcul semblable aux autres approches. Encore une fois, la taille du graphe impliqué et la difficulté relativement faible de l'exemplaire sont en cause.

La 13-arborescence, dont le degré est limité à 2, est un problème plus difficile que le pro-

Tableau 5.8 Nombre de backtracks (gauche) et temps en secondes (droite) avant de trouver une 10-arborescence (k = 10) pour chaque exemplaire. Chaque ligne représente une moyenne sur 10 exemplaires, dont les sous-ensembles de sommets candidats sont différents. Le degré entrant maximum des sommets est de 3.

exemplaire	maxSD	IBS	random
b1	0.2	570.0	2857.3
b3	111.7	3699.8	25080.9
b5	118.0	671.1	178.8

exemplaire	maxSD	IBS	random
b1	0.09	0.20	0.36
b3	0.24	1.48	24.26
b5	0.22	0.29	0.10

Tableau 5.9 Nombre de backtracks (gauche) et temps en secondes (droite) avant de trouver une 13-arborescence (k = 13) pour chaque exemplaire. Chaque ligne représente une moyenne sur 10 exemplaires, dont les sous-ensembles de sommets candidats son différents. Le degré maximal entrant des sommets est de 2.

exemplaire	maxSD	IBS	random
b1	317.2	1723.6	6925.1
b3	445.4	1946.7	91613.3
b5*	258.0	639.0	254.4

exemplaire	maxSD	IBS	random
b1	0.27	0.54	0.66
b3	0.53	0.59	10.78
b5*	0.36	0.31	0.08

blème de la 10-arboresence de degré 3. En effet, un grand nombre d'essais ont été réalisés pour trouver des exemplaires difficiles, ce qui explique le choix de ces paramètres particuliers. Le tableau 5.9 en témoigne, montrant un nombre de backtracks beaucoup plus important pour les trois heuristiques. Pour les exemplaires b1 et b3, maxSD guide beaucoup mieux la recherche de solutions, bien qu'elle ne soit pas beaucoup plus rapide que IBS. L'heuristique random se montre très inefficace pour ces deux exemplaires, ce qui indique leur difficulté. L'exemplaire b5 n'admet aucune solution à ce problème, donc les trois heuristiques démontrent la preuve d'optimalité. Comme cet exemplaire est plus facile que les deux autres, maxSD performe moins bien, générant autant de backtracks et prenant plus de temps que l'heuristique aléatoire. Il en va de même pour IBS.

CHAPITRE 6

CONCLUSION

Les travaux proposés dans ce mémoire traitent principalement des heuristiques de branchements et du dénombrement de solutions, deux aspects qui se sont révélés comme clé dans la programmation par contraintes. Guider la recherche vers les espaces de solutions prometteurs en utilisant le dénombrement s'avère très efficace pour plusieurs contraintes, dont celles étudiées dans le cadre de cette maitrise : l'arbre de recouvrement et l'anti-arborescence. Ce chapitre se divise comme suit : la section 6.1 rappelle et résume la contribution des travaux ; la section 6.2 expose les limitations de la contribution et la section 6.3 aborde les améliorations possibles et les perspectives d'avenir.

6.1 Synthèse des travaux

La densité de solution, étant donné une contrainte, est un rapport qui indique, pour une combinaison variable-valeur, la quantité de solutions auxquelles elle participe. Cette information peut être utilisée pour guider la recherche. Pour pouvoir utiliser cette approche, il faut un algorithme de dénombrement propre à chaque contrainte. Il existe plusieurs contraintes pour lesquelles un tel algorithme n'a pas été conçu, notamment pour les contraintes d'arbre de recouvrement et d'anti-arborescence.

Dans le chapitre 3, les algorithmes de dénombrement pour les contraintes d'arbre de recouvrement et d'anti-arborescence sont expliqués. La matrice Laplacienne, pouvant être formée à partir de la matrice d'adjacence d'un graphe, possède des propriétés intéressantes, qui permettent son utilisation pour le calcul de densité de solution. Pour la contrainte d'arbre de recouvrement, il est possible de calculer le déterminant de cette matrice pour obtenir directement le nombre d'arbres de recouvrement total dans le graphe. En inversant une sous-matrice issue de la matrice Laplacienne, le nombre d'arbres de recouvrement auxquels une arête précise participe peut être calculé en $\mathcal{O}(n^4)$. Pour la contrainte d'anti-arborescence, il est également possible de retirer de l'information de dénombrement à partir de la matrice Laplacienne. En retirant la rangée et colonne correspondant au puit du graphe "sink-rooted", le mineur de la matrice Laplacienne peut également être calculé, ce qui permet d'obtenir la densité de solution relative à un arc. Ce calcul peut être fait en $\mathcal{O}(n^3)$

Dans le chapitre 4, la mise en place des algorithmes de dénombrement, leur intégration à la recherche ainsi que les complications relatives à l'implémentation sont abordées. Tout

d'abord, les structures de données, les variables et les contraintes nécessaires à la modélisation des problèmes d'arbre de recouvrement et d'anti-arborescences sont détaillées. En utilisant des structures réversibles, les calculs relatifs au backtracking peuvent être évités. En utilisant un vecteur de variables binaires et des contraintes sur le degré entrant et sortant des sommets, plusieurs problèmes de STA contraints peuvent être mis en place. L'inversion de la matrice Laplacienne pour le calcul de la densité de solution peut être faite de façon efficace avec les fonctionnalités de la librairie ALGLIB, qui utilise la décomposition LU. La densité de solution peut être mise à jour de façon incrémentale, en utilisant la formule de Sherman-Morrison, pour le cas non orienté et le cas orienté. En modifiant directement la matrice inversée, un facteur n peut être retiré du calcul de la densité de solution, ce qui permet de le faire en temps $\mathcal{O}(\gamma n)$. Des complications reliées à l'utilisation de nombres en point flottant sont relevées, expliquées et résolues.

Le chapitre 5 compare l'approche maximisant la densité de solution à chaque branchement à d'autres approches pour démontrer son efficacité. Le problème de l'arbre de recouvrement de degré contraint est utilisé pour la contrainte d'arbre de recouvrement. Pour les graphes non orientés aléatoires et en particulier pour ceux composés de quelques composantes connexes reliées par des arêtes ponts, l'approche maximisant la densité de solution se démontre la plus efficace. Des graphes issus de topologies réelles sont également utilisés pour illustré l'efficacité de l'heuristique proposée. Le problème de k-arborecence est utilisé pour la contrainte d'anti-arborescence, dans le cas orienté. En réutilisant et en adaptant les topologies réelles précédemment utilisées, l'efficacité de l'heuristique de branchement basée sur le dénombrement est à nouveau démontrée, en particulier pour les exemplaires les plus difficiles.

6.2 Limitations de la solution proposée

Bien que l'heuristique de branchement basée sur les densités de solution s'avère efficace, elle admet certaines limitations. La plus flagrante est sans doute la mise à l'échelle. Au fur et à mesure que le graphe à traiter grossit, la matrice à inverser grossit également. Il est évident que l'inversion de matrice, pour des graphes d'une très grande taille (quelques centaines de sommets), sera très coûteuse peu importe la méthode utilisée. Si le coût relatif à l'inversion excède le bénéfice retiré du bon guidage de la recherche de solutions, il n'est plus utile d'utiliser la densité de solution pour guider la recherche.

6.3 Améliorations futures

Comme travail futur, nous pourrions résoudre d'autres types de STAs contraints. Plusieurs domaines, dont la conception de réseaux, la télécommunication et le transport ont des

problèmes qui impliquent la recherche de STAs. Le problème de l'arbre de recouvrement de degré contraint [28], le problème de l'arbre de recouvrement "hop-constrained" [15] et le STA minimum de diamètre contraint [3] en sont des exemples.

Nos travaux pourraient également avoir des applications pour les circuits Eulériens. En effet, l'algorithme BEST[2, 44] décrit une formule qui permet de compter le nombre de cycles Eulériens dans un graphe orienté, en temps polynomial. Or, ce calcul implique celui du nombre d'arborescences, dont nous donnons le détail dans nos travaux. Les circuits Eulériens ont de nombreuses applications, dont la planification de routes de déneigement ou de facteurs.

Nous planifions également d'examiner la compatibilité de la notre algorithme de densité de solution avec des algorithmes de filtrage plus puissants développés pour les contraintes de STA, tel que proposé dans la littérature.

RÉFÉRENCES

[1] (2010). IBM ILOG CPLEX Optimizer.
urlhttp ://www-01.ibm.com/software/integration/optimization/cplex-optimizer/.

[2] AARDENNE-EHRENFEST, T. et BRUIJN, N. (1987). *Circuits and Trees in Oriented Linear Graphs*. Modern Birkhäuser Classics. Birkhäuser Boston.

[3] ABDALLA, A. et DEO, N. (2002). Random-tree diameter and the diameter-constrained mst. *Int. J. Comput. Math.*, 79, 651–663.

[4] ALON, N., FOMIN, F. V., GUTIN, G., KRIVELEVICH, M. et SAURABH, S. (2009). Spanning directed trees with many leaves. *SIAM J. Discrete Math.*, 23, 466–476.

[5] BANSAL, N., KHANDEKAR, R. et NAGARAJAN, V. (2009). Additive guarantees for degree-bounded directed network design. *SIAM J. Comput.*, 39, 1413–1431.

[6] BELDICEANU, N., FLENER, P. et LORCA, X. (2005). The tree constraint. *CPAIOR*. 64–78.

[7] BOCHKANOV, S. (2010). Alglib. `http://mloss.org/software/view/231/`.

[8] BOUSSEMART, F., HEMERY, F., LECOUTRE, C. et SAIS, L. (2004). Boosting systematic search by weighting constraints. *ECAI*. 146–150.

[9] DOOMS, G. et KATRIEL, I. (2006). The *minimum spanning tree* constraint. *CP*. 152–166.

[10] DOOMS, G. et KATRIEL, I. (2007). The "not-too-heavy spanning tree" constraint. *CPAIOR*. 59–70.

[11] FOULDS, L. R. (1991). *Graph Theory Applications*. Springer.

[12] FÜRER, M. et RAGHAVACHARI, B. (1994). Approximating the minimum-degree steiner tree to within one of optimal. *J. Algorithms*, 17, 409–423.

[13] GABOW, H. N. et MYERS, E. W. (1978). Finding all spanning trees of directed and undirected graphs. *SIAM J. Comput.*, 7, 280–287.

[14] GAREY, M. R. et JOHNSON, D. S. (1979). *Computers and Intractability : A Guide to the Theory of NP-Completeness*. W. H. Freeman.

[15] GOUVEIA, L., SIMONETTI, L. et UCHOA, E. (2011). Modeling hop-constrained and diameter-constrained minimum spanning tree problems as steiner tree problems over layered graphs. *Math. Program.*, 128, 123–148.

[16] HARALICK, R. M. et ELLIOTT, G. L. (1980). Increasing tree search efficiency for constraint satisfaction problems. *Artif. Intell.*, 14, 263–313.

[17] HSU, E. I., KITCHING, M., BACCHUS, F. et MCILRAITH, S. A. (2007). Using expectation maximization to find likely assignments for solving csp's. *AAAI*. 224–230.

[18] KAPOOR, S. et RAMESH, H. (1995). Algorithms for enumerating all spanning trees of undirected and weighted graphs. *SIAM J. Comput.*, 24, 247–265.

[19] KAPOOR, S. et RAMESH, H. (2000). An algorithm for enumerating all spanning trees of a directed graph. *Algorithmica*, 27, 120–130.

[20] KASK, K., DECHTER, R. et GOGATE, V. (2004). Counting-based look-ahead schemes for constraint satisfaction. *CP*. 317–331.

[21] KHANDEKAR, R., KORTSARZ, G. et NUTOV, Z. (2013). On some network design problems with degree constraints. *J. Comput. Syst. Sci.*, 79, 725–736.

[22] LEBRAS, R., ZANARINI, A. et PESANT, G. (2009). Efficient generic search heuristics within the embp framework. *CP*. 539–553.

[23] LOKSHTANOV, D., RAMAN, V., SAURABH, S. et SIKDAR, S. (2009). On the directed degree-preserving spanning tree problem. *IWPEC*. 276–287.

[24] LOKSHTANOV, D., RAMAN, V., SAURABH, S. et SIKDAR, S. (2011). On the directed full degree spanning tree problem. *Discrete Optimization*, 8, 97–109.

[25] MERRIS, R. (1994). Laplacian matrices of graphs : a survey. *Linear Algebra and its Applications*, 197–198, 143 – 176.

[26] MICHEL, L. et HENTENRYCK, P. V. (2012). Activity-based search for black-box constraint programming solvers. *CPAIOR*. 228–243.

[27] MOSKEWICZ, M. W., MADIGAN, C. F., ZHAO, Y., ZHANG, L. et MALIK, S. (2001). Chaff : Engineering an efficient sat solver. *DAC*. 530–535.

[28] NARULA, S. C. et HO, C. A. (1980). Degree-constrained minimum spanning tree. *Computers & OR*, 7, 239–249.

[29] NUTOV, Z. (2011). Approximating directed weighted-degree constrained networks. *Theor. Comput. Sci.*, 412, 901–912.

[30] OSTROWSKI, A. (1937). Über die determinanten mit überwiegender hauptdiagonale. *Commentarii Mathematici Helvetici*, 10, 69–96.

[31] PESANT, G. (2005). Counting solutions of csps : A structural approach. *IJCAI*. 260–265.

[32] PESANT, G. (2011). Filtering and Counting for the Spread and Deviation Constraints. *Proc. Tenth International Workshop on Constraint Modelling and Reformulation (held during CP'11)*.

[33] PESANT, G., GENDREAU, M., POTVIN, J.-Y. et ROUSSEAU, J.-M. (1998). An exact constraint logic programming algorithm for the traveling salesman problem with time windows. *Transportation Science*, 32, 12–29.

[34] PESANT, G., QUIMPER, C.-G. et ZANARINI, A. (2012). Counting-based search : Branching heuristics for constraint satisfaction problems. *J. Artif. Intell. Res. (JAIR)*, 43, 173–210.

[35] PESANT, G. et ZANARINI, A. (2011). Recovering indirect solution densities for counting-based branching heuristics. *CPAIOR*. 170–175.

[36] POTHOF, I., SCHUT, J. et DER TOEGEPASTE WISKUNDE, U. T. F. (1995). *Graph-theoretic Approach to Identifiability in a Water Distribution Network*. Memorandum / Faculteit der Toegepaste Wiskunde, Universiteit Twente. Fac. of Applied Math., University of Twente.

[37] REFALO, P. (2004). Impact-based search strategies for constraint programming. *CP*. 557–571.

[38] RÉGIN, J.-C. (2008). Simpler and incremental consistency checking and arc consistency filtering algorithms for the weighted spanning tree constraint. *CPAIOR*. 233–247.

[39] RÉGIN, J.-C., ROUSSEAU, L.-M., RUEHER, M. et VAN HOEVE, W. J. (2010). The weighted spanning tree constraint revisited. *CPAIOR*. 287–291.

[40] SHERMAN, J. et MORRISON, W. J. (1950). Adjustment of an inverse matrix corresponding to a change in one element of a given matrix. *Annals of Mathematical Statistics*, 21, 124–127.

[41] TUTTE, W. (2001). *Graph Theory*, vol. 21 de *Encyclopedia of Mathematics and Its Applications*. Cambridge University Press. 333 pages.

[42] UNO, T. (1996). An algorithm for enumerating all directed spanning trees in a directed graph. *ISAAC*. 166–173.

[43] VANDEGRIEND, B. (1998). *Finding Hamiltonian Cycles : Algorithms, Graphs and Performance*. Mémoire de maîtrise, University of Alberta. Http ://webdocs.cs.ualberta.ca/∼joe/Theses/HCarchive/main.html.

[44] W. T. TUTTE, C. A. B. S. (1941). On unicursal paths in a network of degree 4. *The American Mathematical Monthly*.

[45] ZANARINI, A. et PESANT, G. (2009). Solution counting algorithms for constraint-centered search heuristics. *Constraints*, 14, 392–413.