Amélioration du filtrage de la contrainte WeightedCircuit pour le problème du commis voyageur

Mémoire

Raphaël Boudreault

Maîtrise en informatique - avec mémoire
Maître ès sciences (M. Sc.)

Québec, Canada

Amélioration du filtrage de la contrainte WEIGHTEDCIRCUIT pour le problème du commis voyageur

Mémoire

Raphaël Boudreault

Sous la direction de:

Claude-Guy Quimper

Résumé

Le *problème du commis voyageur*, aussi connu sous le nom de *problème du voyageur de commerce* ou *traveling salesman problem* (TSP) en anglais, est un problème classique de l'optimisation combinatoire et de la recherche opérationnelle. Il consiste, étant donné un certain nombre de villes et la distance entre chacune d'entre elles, à trouver un chemin de longueur minimale visitant chaque ville une seule fois et retournant à son point de départ. Le problème apparaît naturellement dans une multitude de problématiques de transport et industrielles, en plus de trouver des applications dans un important nombre de domaines en apparence non liés, allant de la logistique au séquençage de l'ADN. Toutefois, sa complexité informatique le rend difficile à résoudre.

Le solveur *Concorde* permet actuellement de résoudre de manière exacte des instances du TSP comportant des milliers de villes en seulement quelques secondes. Cependant, une limitation importante est qu'il ne permet pas de considérer des contraintes additionnelles telles que des fenêtres de temps pour chaque visite. La *programmation par contraintes* est une approche permettant facilement d'ajouter ces contraintes au problème.

Dans ce mémoire, nous revisitons l'approche *CP-based Lagrangian relaxation* (CP-LR) utilisée notamment pour les algorithmes de filtrage de l'état de l'art de la contrainte WeightedCircuit encodant le TSP en programmation par contraintes. Nous proposons deux nouveaux algorithmes basés sur notre approche CP-LR améliorée. Ceux-ci permettent d'obtenir un gain significatif sur le temps de résolution du TSP comparativement à l'implémentation de l'état de l'art.

Abstract

The *traveling salesman problem* (TSP) is a classic problem in combinatorial optimization and operations research. It consists, given a number of cities and the distance between each of them, to find a path of minimal distance visiting each city exactly once and returning to its starting point. The problem naturally appears in various transportation and industrial problems, in addition to having applications in several domains apparently unrelated, going from logistics to DNA sequencing. Its computational complexity makes it nonetheless difficult to solve.

The *Concorde* solver currently allows to exactly solve TSP instances having thousands of cities in only a few seconds. However, an important limitation is that it cannot consider additional constraints such as time windows for each visit. *Constraint programming* is an approach that easily allows these constraints to be added to the problem.

In this Master's thesis, we revisit the *CP-based Lagrangian relaxation* (CP-LR) approach used in particular for the state-of-the-art filtering algorithms of the WeightedCircuit constraint that encodes the TSP in constraint programming. We propose two new algorithms based on our improved CP-LR approach. These allow to obtain a significant gain on the TSP solving time when compared to the state-of-the-art implementation.

Table des matières

Liste des tableaux

Liste des figures

À mon grand-papa Antoine.
J'espère que cette lecture
t'évoquera de bons souvenirs.

Remerciements

J'aimerais prendre le temps de remercier convenablement mon directeur de recherche, Claude-Guy Quimper. Tout au long de ma maîtrise, mes réunions hebdomadaires avec lui m'ont été particulièrement précieuses. Ses conseils sont toujours judicieux, ses idées sont extrêmement pertinentes et son dévouement à la recherche et l'enseignement est agréablement apprécié. J'aimerais aussi le remercier pour sa confiance et toutes les opportunités qu'il m'a offertes. C'est tout de même grâce à lui que j'ai pu obtenir un stage Mitacs auprès de Thales, chez qui maintenant je travaille en recherche!

Je voudrais aussi particulièrement remercier Kim Rioux-Paradis qui m'a recrutée au *PPC Lab* en 2018 en me conseillant d'aller vers Claude-Guy. Tu as été une inspiration pour moi, alors je te remercie grandement de m'avoir montré que, oui, il y a des « jobs » après un baccalauréat en mathématiques.

Bien sûr, je dois aussi remercier toutes les personnes du *PPC Lab* que j'ai côtoyées et qui ont rendu mon parcours particulièrement agréable, notamment Yanick Ouellet, Nicolas Blais, Marc-André Ménard, Alexandre Mercier-Aubin, Catherine Bourdeau-Laferrière, Anthony Deschênes, Christopher Coulombe, et les autres! Vous étiez ma raison principale de me lever le matin (jusqu'à ce qu'on soit tous confinés).

Finalement, je dois évidemment remercier le CRSNG et le FRQNT pour le support financier de ma maîtrise.

Avant-propos

Ce mémoire présente les travaux de recherche réalisés au cours de ma maîtrise sous la direction de Claude-Guy Quimper. Ceux-ci ont mené à un article, intitulé *Improved CP-Based Lagrangian Relaxation Approach with an Application to the TSP*, dont les auteurs sont Raphaël Boudreault et Claude-Guy Quimper, présenté et publié le 19 août 2021 dans le cadre de la conférence scientifique en intelligence artificielle *30th International Joint Conference on Artificial Intelligence* (IJCAI 2021) [12]. Pour cet article, je possède le statut de premier auteur et Claude-Guy Quimper est mon directeur de recherche. Mon rôle dans la préparation de cet article a été d'effectuer le travail de recherche, élaborer les algorithmes, réaliser les expérimentations et rédiger l'article. Cet article est inséré sans modifications au texte au chapitre 3 de ce mémoire. Certaines équations mathématiques ont été légèrement retravaillées en respectant la notation introduite dans ce mémoire pour en améliorer la lisibilité et assurer sa cohérence.

Introduction

Le *problème du commis voyageur*, aussi connu sous le nom de *problème du voyageur de commerce* ou *traveling salesman problem* (TSP) en anglais, est un problème classique de l'optimisation combinatoire et de la recherche opérationnelle [4]. Il consiste, étant donné un certain nombre de villes et la distance entre chacune d'elles, à trouver un chemin de longueur minimale visitant chaque ville une seule fois et retournant à son point de départ. L'intérêt de ce problème est qu'il apparaît naturellement dans une multitude de problématiques de transport et industrielles. De plus, en remplaçant les concepts de « villes » et de « chemins » selon le domaine, le TSP et ses nombreuses variantes trouvent des applications dans un important nombre de domaines en apparence non liés, allant de la logistique au séquençage de l'ADN.

La principale difficulté du TSP réside dans l'explosion du nombre de possibilités en fonction du nombre de villes considérées. En effet, pour une collection de n villes, il existe $\frac{1}{2}(n-1)!$ chemins candidats. Par exemple, pour un nombre réaliste de villes tel que 50, on fait face à un nombre de possibilités de l'ordre de 10^{62}. Cela signifie qu'un algorithme essayant de façon exhaustive l'ensemble des possibilités et prenant 0,000 001 seconde pour évaluer la longueur d'un chemin nécessiterait tout de même au total plus de 10^{49} années de calcul. Ainsi, un algorithme raisonnable devra implicitement éliminer des lots de chemins afin de concentrer l'exploration sur les candidats prometteurs.

Plusieurs techniques sont actuellement utilisées dans la littérature pour résoudre le TSP malgré sa complexité informatique. Parmi celles-ci, le solveur *Concorde* permet de résoudre de manière exacte des instances du TSP comportant des milliers de villes en seulement quelques secondes [4]. Encore aujourd'hui, il est l'état de l'art pour ce problème. Toutefois, l'une des principales limitations de *Concorde* est qu'il ne permet de résoudre que le TSP *pur*, c'est-à-dire qu'il ne permet pas de considérer des contraintes additionnelles telles que des fenêtres de temps pour chaque visite, la capacité des véhicules et l'évitement du trafic, contraintes qui surviennent rapidement dans toute application pratique du TSP. Dans ces cas, il est nécessaire de considérer une tout autre approche.

La *programmation par contraintes* est une approche naturelle pour ce type de problèmes. Dans ce paradigme de programmation, les éléments à décider, appelés *variables*, sont munis chacun d'un ensemble de valeurs possibles appelé *domaine*. Les restrictions sur les valeurs que peuvent

prendre les variables sont exprimées sous la forme de *contraintes*, des énoncés logiques qui sont respectés sous certaines conditions. Afin de retirer des valeurs incohérentes du domaine des variables, chaque contrainte est associée à un *algorithme de filtrage* spécifique. Ces derniers sont à la base d'une résolution efficace d'un problème d'optimisation en programmation par contraintes. Pour notre problème d'intérêt, la contrainte qui encode implicitement le TSP est connue dans la littérature sous le nom de WeightedCircuit [7].

Les algorithmes de filtrage de l'état de l'art pour WeightedCircuit sont basés sur une *relaxation lagrangienne* du TSP [7, 8]. En bref, cette technique d'optimisation permet de violer certaines contraintes du problème en ajoutant de nouvelles variables, appelées *multiplicateurs de Lagrange*, qui pénalisent l'objectif lorsque c'est le cas. Le tout permet d'obtenir une longueur de chemin plus courte que la longueur minimale et celle-ci est utilisée pour le filtrage des variables du problème. Lorsque ce filtrage est combiné à l'optimisation globale des multiplicateurs, l'approche résultante est connue sous le nom de *CP-based Lagrangian relaxation* (CP-LR) [77, 78]. Cette dernière est au cœur des algorithmes de filtrage pour la contrainte WeightedCircuit, mais peut aussi être utilisée pour d'autres contraintes.

Dans ce mémoire, nous proposons d'améliorer le filtrage de WeightedCircuit en introduisant une approche générale CP-LR améliorée. Cette dernière modifie localement les multiplicateurs de Lagrange avant d'appeler les algorithmes de filtrage dans le but d'augmenter le nombre de valeurs filtrées. Nous introduisons donc deux nouveaux algorithmes basés sur celle-ci pour le filtrage de WeightedCircuit. Les résultats expérimentaux que nous obtenons sur des instances du TSP montrent que nos algorithmes permettent d'obtenir un gain significatif sur le temps de résolution lorsque comparés à l'implémentation de l'état de l'art.

Le présent mémoire est divisé comme suit. D'abord, le chapitre 1 introduit l'ensemble des concepts nécessaires à la compréhension de la problématique et les contributions de ce mémoire. Ensuite, le chapitre 2 présente l'état de l'art actuel en programmation par contraintes pour la résolution du TSP, donc le filtrage de WeightedCircuit. Finalement, le chapitre 3 expose l'ensemble des contributions de ce mémoire, en présentant l'approche CP-LR améliorée, son application au TSP et les résultats expérimentaux obtenus.

Chapitre 1

Notions préliminaires

Dans ce chapitre, nous introduisons l'ensemble des concepts nécessaires à la compréhension de la problématique et les contributions de ce mémoire. D'abord, les sections 1.1, 1.2 et 1.3 présentent, dans l'ordre, les notions entourant l'optimisation, l'optimisation linéaire et l'optimisation linéaire en nombres entiers, allant du plus général au plus spécifique. Puis, à la section 1.4, nous présentons les différentes notions de la théorie des graphes dont nous aurons besoin pour présenter le problème du commis voyageur à la section 1.5. Finalement, nous introduisons la programmation par contraintes à la section 1.6.

1.1 Optimisation

L'*optimisation* est une branche particulière des mathématiques qui apparaît naturellement dans une multitude de domaines [62]. Elle consiste essentiellement à déterminer la meilleure décision parmi un ensemble de possibilités en fonction de certains critères donnés. D'abord, la *modélisation* permet de formuler à l'aide d'expressions mathématiques les différentes composantes d'un problème d'optimisation. Puis, à l'aide de techniques mathématiques et numériques, la *résolution* cherche à définir le choix optimal pour le problème. L'optimisation est à l'intersection des mathématiques et de l'informatique, et joue un rôle essentiel notamment en recherche opérationnelle, mathématiques appliquées, économie, statistique, théorie des jeux et plusieurs applications industrielles.

Un problème d'optimisation est formé de *variables de décision*, ou simplement *variables*, qui correspondent aux éléments à décider. Un exemple de variable dans une entreprise pourrait être le nombre d'employés sur le plancher à une certaine heure, ou encore le prix de vente d'un bien. Le fait d'optimiser consiste alors d'affecter la meilleure valeur à l'ensemble des variables selon un objectif prédéfini. Cet objectif s'exprime par le biais d'une *fonction objectif* dont la valeur dépend des variables de décision du problème. Cette valeur est appelée *valeur objectif* et peut être *maximisée* ou *minimisée*. Il pourrait s'agir par exemple d'une fonction exprimant le revenu global de l'entreprise, où la meilleure décision serait celle qui maximise

cet objectif. Les *contraintes* d'un problème d'optimisation établissent les restrictions sur les valeurs que peuvent prendre les variables. La disponibilité des employés ou l'existence d'une limite budgétaire en sont des exemples.

Un problème d'optimisation quelconque peut être formulé par

$$\begin{aligned} \min_{\boldsymbol{x}} \quad & f(\boldsymbol{x}) \\ \text{s. à} \quad & \boldsymbol{x} \in X \end{aligned} \tag{PO}$$

où $\boldsymbol{x} = (x_1, x_2, \ldots, x_n)^T$ est le vecteur [1] des variables et f est la fonction objectif à valeurs dans $\mathbb{R}$. L'abréviation « s. à » indique les différentes contraintes du problème dont la fonction est « sujette à ». L'ensemble X est appelé l'*espace des solutions* et détermine les *solutions admissibles* du problème, c'est-à-dire les valeurs possibles de $\boldsymbol{x}$ respectant l'ensemble des contraintes du problème. Cet ensemble est généralement le résultat de la conjonction de plusieurs contraintes distinctes. La notation $\min_{\boldsymbol{x}}$ signifie que le problème (PO) est un problème de minimisation sur les variables $\boldsymbol{x}$. On dit d'une solution $\boldsymbol{x}^* \in X$ qu'elle est *optimale* si la fonction f atteint son minimum en $\boldsymbol{x}^*$, c'est-à-dire si pour tout $\boldsymbol{x} \in X$, $f(\boldsymbol{x}^*) \leq f(\boldsymbol{x})$. Notons que tout problème de maximisation peut être réécrit sous la forme d'un problème de minimisation en observant que maximiser $g(\boldsymbol{x})$ est équivalent à minimiser $f(\boldsymbol{x}) = -g(\boldsymbol{x})$.

1.2 Optimisation linéaire

1.2.1 Définition

Un *problème d'optimisation linéaire* est le problème d'optimisation (PO) où les variables $\boldsymbol{x}$ sont *continues*, c'est-à-dire $\boldsymbol{x} \in \mathbb{R}^n$, et où X est défini par des contraintes d'inégalité $g_i(\boldsymbol{x}) \leq 0$, pour $i \in \{1, \ldots, k\}$, avec g_i une fonction à valeurs dans $\mathbb{R}$ [19, 31, 62]. De plus, la fonction objectif f et les fonctions $g_1, \ldots, g_k$ doivent être *linéaires*, c'est-à-dire sous la forme

$$v_1 x_1 + v_2 x_2 + \ldots + v_n x_n + d = \boldsymbol{v}^T \boldsymbol{x} + d$$

où $\boldsymbol{v} = (v_1, v_2, \ldots, v_n)^T \in \mathbb{R}^n$ est un vecteur de coefficients et $d \in \mathbb{R}$ est une constante. Ainsi, un problème d'optimisation linéaire à n variables et m contraintes est formulé de manière générale par

$$\begin{aligned} \min_{\boldsymbol{x}} \quad & f(\boldsymbol{x}) = \boldsymbol{c}^T \boldsymbol{x} \\ \text{s. à} \quad & \boldsymbol{a_i}^T \boldsymbol{x} \leq b_i, \qquad i \in \{1, \ldots, m\} \\ & x_j \geq 0, \qquad j \in \{1, \ldots, n\} \end{aligned}$$

où $\boldsymbol{c} \in \mathbb{R}^n$, $\boldsymbol{a_i} \in \mathbb{R}^n$ pour $i \in \{1, \ldots, m\}$ et $\boldsymbol{b} = (b_1, b_2, \ldots, b_m)^T \in \mathbb{R}^m$. Notons que cette formulation permet de considérer d'autres types de contraintes, puisque

$$\boldsymbol{a_i}^T \boldsymbol{x} \geq b_i \iff -\boldsymbol{a_i}^T \boldsymbol{x} \leq -b_i$$

1. Dans ce mémoire, les vecteurs sont représentés **en gras** et considérés sous la forme de vecteurs colonne.

et

$$\boldsymbol{a_i}^T \boldsymbol{x} = b_i \iff \left(\boldsymbol{a_i}^T \boldsymbol{x} \leq b_i \wedge \boldsymbol{a_i}^T \boldsymbol{x} \geq b_i\right).$$

Sous forme matricielle, en posant A comme étant la matrice formée de m lignes et n colonnes où la $i^{\text{ème}}$ ligne correspond au vecteur $\boldsymbol{a_i}^T$, on obtient

$$\begin{aligned} \min_{\boldsymbol{x}} \quad & f(\boldsymbol{x}) = \boldsymbol{c}^T \boldsymbol{x} \\ \text{s. à} \quad & A\boldsymbol{x} \leq \boldsymbol{b} \\ & \boldsymbol{x} \geq 0. \end{aligned} \tag{PL}$$

À titre d'exemple, considérons le problème d'optimisation linéaire à deux variables suivant :

$$\begin{aligned} \min_{x,y} \quad & f(x,y) = -3x - 2y \\ \text{s. à} \quad & -\tfrac{1}{2}\,x + y \leq \tfrac{5}{2} && (1.1) \\ & \tfrac{3}{4}\,x + y \leq \tfrac{9}{2} && (1.2) \\ & x - y \leq \tfrac{5}{2} && (1.3) \\ & x, y \geq 0. \end{aligned}$$

L'espace des solutions déterminé par les contraintes du problème est illustré à la figure 1.1. Celui-ci est un *polygone convexe*, une forme géométrique définie par un nombre fini de droites ayant la propriété d'être convexe, c'est-à-dire « sans creux ». En dimensions supérieures, cette notion se généralise au *polytope convexe*, dont les limites sont plutôt définies par des *hyperplans*. Il est possible de montrer que l'espace des solutions d'un problème d'optimisation linéaire est toujours un polytope convexe. Pour assurer l'existence d'une solution optimale, notons qu'il faut aussi imposer que l'espace soit non vide et *borné*, c'est-à-dire qu'il n'est pas possible de faire tendre la valeur objectif vers $-\infty$. Dans notre exemple, toutes ces propriétés

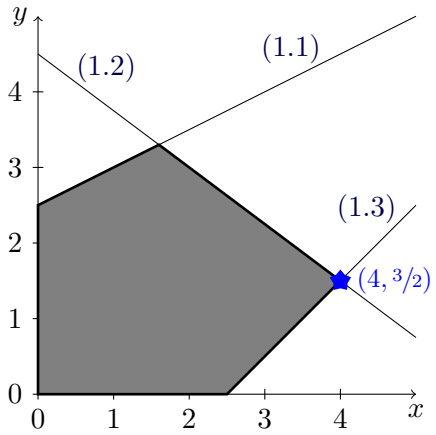


FIGURE 1.1 – Représentation graphique de l'espace des solutions, en gris, du problème en exemple.

sont vérifiées et la fonction objectif est minimisée au point $(x^*, y^*) := (4, 3/2)$ avec une valeur optimale $f(x^*, y^*) = -15$.

1.2.2 Résolution

Le fait que la solution optimale du problème corresponde à un sommet du polygone n'est pas un hasard. En effet, la convexité de l'espace des solutions nous assure que si une solution optimale existe, alors il y en a nécessairement une en un sommet du polygone. Dans le cas où le problème possèderait plus d'une solution optimale, l'ensemble des solutions optimales est défini par une arête du polygone et inclut les sommets de celle-ci. Ce phénomène se généralise aussi en dimensions supérieures, où les sommets d'un polytope à n dimensions correspondent aux intersections de n hyperplans. Ainsi, pour tout problème d'optimisation linéaire, la recherche de solutions peut se limiter à l'ensemble des sommets du polytope. Bien qu'il en existe une quantité finie, ce nombre peut être exponentiel en fonction du nombre de variables et de contraintes.

Une autre propriété intéressante découle de la convexité de l'espace des solutions et de la linéarité de la fonction objectif. À partir de toute solution admissible qui n'est pas optimale, il existe une direction dans l'espace des solutions vers laquelle la valeur objectif ne peut que diminuer ou rester inchangée. La direction recherchée sera généralement celle où l'objectif diminue le plus rapidement. La combinaison des précédentes observations est à la base de l'*algorithme du simplexe*, introduit par George Dantzig en 1947 et premier algorithme à résoudre ce type de problèmes dans sa généralité [22]. Son appellation est tirée de son lien avec le *simplexe* en mathématiques, un cas particulier du polytope convexe.

Pour utiliser l'algorithme du simplexe, il faut d'abord reformuler le problème (PL) sous la forme *standard*

$$\begin{aligned} \min_{\boldsymbol{x}} \quad & f(\boldsymbol{x}) = \boldsymbol{c}^T \boldsymbol{x} \\ \text{s. à} \quad & A\boldsymbol{x} = \boldsymbol{b} \\ & \boldsymbol{x} \geq 0, \end{aligned}$$

c'est-à-dire sous la forme d'un problème ne comportant que des contraintes d'égalité. Toute contrainte d'inégalité peut être transformée en égalité grâce à l'ajout d'une *variable d'écart* au problème. Pour notre exemple, en ajoutant les variables d'écart $s_1, s_2, s_3 \geq 0$, on obtient

$$\begin{aligned} \min_{x,y} \quad & f(x, y) = -3x - 2y \\ \text{s. à} \quad & -\tfrac{1}{2}\, x + y - s_1 = \tfrac{5}{2} \\ & \tfrac{3}{4}\, x + y - s_2 = \tfrac{9}{2} \\ & x - y - s_3 = \tfrac{5}{2} \\ & x, y, s_1, s_2, s_3 \geq 0. \end{aligned}$$

Géométriquement, chaque sommet du polygone de la figure 1.1 correspond à l'intersection de deux contraintes d'inégalité *actives*, c'est-à-dire où l'égalité est en fait atteinte. Par exemple, le sommet optimal (x^*, y^*) est obtenu au point où $\frac{3}{4}\,x + y = \frac{9}{2}$ (contrainte (1.2)) et $x - y = \frac{5}{2}$ (contrainte (1.3)). Remarquons que sous la forme standard, ce sommet correspond au choix de fixer les variables d'écart s_2 et s_3 à 0. En fait, chaque sommet correspond au choix de deux variables fixées à la valeur 0. Ces dernières sont dites *hors bases*, alors que les variables restantes sont appelées *de base*. De façon générale, pour un problème sous forme standard de n variables et m contraintes, les sommets du polytope correspondent à un choix de $n - m$ variables hors bases (m variables de base). De plus, deux sommets sont dits *adjacents* s'ils n'ont qu'une variable de base différente. Notons que cette définition concorde bien avec l'aspect géométrique de la notion d'adjacence de sommets.

Partant d'un sommet admissible du problème considéré, l'algorithme du simplexe effectue un parcours de sommet en sommet du polytope jusqu'à ce qu'il soit impossible de faire diminuer davantage la valeur objectif. Le sommet suivant chaque sommet intermédiaire visité doit être adjacent et est choisi en fonction de sa vitesse locale à décroître l'objectif en allant dans sa direction. Cette vitesse est déterminée par la notion de *coût réduit*. Pour toute variable hors base x_i, son coût réduit est défini comme étant le coût unitaire d'augmentation de la valeur objectif si la variable x_i prend une valeur différente de 0, c'est-à-dire devient une variable de base. Par exemple, un coût réduit de -3 pour une variable qui prendra la valeur 2 au nouveau sommet donnera une diminution de 6 de l'objectif. L'algorithme du simplexe fait donc le choix d'une variable dont le coût réduit est le plus négatif possible. Ce choix existera toujours si les précédentes propriétés sont vérifiées et que le sommet courant n'est pas optimal. Lorsqu'une solution optimale est atteinte, l'ensemble des coûts réduits sont positifs ou nuls et l'algorithme se termine.

L'algorithme du simplexe peut se réduire à une séquence d'opérations matricielles élémentaires dans un *tableau* de $m + 1$ lignes et $n + m + 1$ colonnes. En ajoutant des règles pour qu'un sommet ne puisse être sélectionné plus d'une fois, phénomène pathologique connu dans la littérature sous le nom de *cyclage*, l'algorithme est assuré de s'exécuter en un nombre fini d'étapes. Toutefois, puisque tous les sommets peuvent être parcourus dans le pire cas, son temps d'exécution peut être exponentiel en fonction de n et m. En pratique, l'algorithme du simplexe est particulièrement efficace sur une multitude de problèmes. De plus, sous certaines hypothèses probabilistes, les travaux de Borgwardt [11] montrent que le temps d'exécution moyen est polynomial en fonction de n et m, avec $O(mn^4)$ itérations.

D'autres algorithmes permettent aussi de résoudre les problèmes d'optimisation linéaire. En 1979, la méthode de l'ellipsoïde de Khachiyan [54] était le premier algorithme en temps polynomial pour les résoudre. Aujourd'hui, il en existe une multitude de variantes, appelées *méthodes de points intérieurs*. Contrairement à l'algorithme du simplexe, ces algorithmes calculent une séquence de points situés à l'intérieur de l'espace des solutions jusqu'à convergence [62]. Fina-

lement, certains cas particuliers tels que les problèmes de flots dans un réseau ont leur propre algorithme en temps polynomial [19].

1.3 Optimisation linéaire en nombres entiers

1.3.1 Définition

Un *problème d'optimisation linéaire en nombres entiers* est le problème d'optimisation linéaire (PL) où toutes les variables sont définies comme étant entières [19, 62]. Une *variable entière* x est contrainte à prendre une valeur entière, c'est-à-dire $x \in \mathbb{Z}$. Lorsque x est limitée aux valeurs 0 et 1, on parle plutôt d'une *variable binaire*. Ainsi, la forme générale est

$$\begin{aligned} \min_{\boldsymbol{x}} \quad & f(\boldsymbol{x}) = \boldsymbol{c}^T \boldsymbol{x} \\ \text{s. à} \quad & A\boldsymbol{x} \leq \boldsymbol{b} \\ & \boldsymbol{x} \geq 0, \boldsymbol{x} \in \mathbb{Z}^n. \end{aligned} \tag{PLE}$$

Lorsque seulement certaines variables sont entières, on dit que le problème est *mixte*. Dans la littérature, les techniques de résolution employées pour les problèmes en nombres entiers sont aussi applicables aux problèmes mixtes. Pour cette raison, nous considérons uniquement dans la suite la forme du problème (PLE).

Reprenons l'exemple de la section 1.2, mais cette fois avec des variables entières. La figure 1.2 illustre le nouvel espace des solutions déterminé par les contraintes du problème. Celui-ci n'est maintenant formé que de 13 solutions bien distinctes. L'espace des solutions est fini et *discret*, c'est-à-dire chaque solution est isolée dans le plan. Bien qu'il soit toujours borné, l'ensemble n'est évidemment plus convexe. De plus, la solution optimale est atteinte en $(x^*, y^*) \coloneqq (3, 2)$ avec une valeur objectif de $f(x^*, y^*) = -13$, et ne correspond plus à

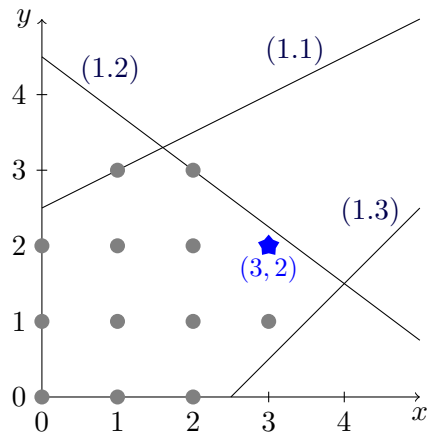


FIGURE 1.2 – Représentation graphique de l'espace des solutions, en gris, lorsque les variables x et y sont entières.

l'intersection de contraintes d'inégalité. Les techniques de résolution abordées à la section 1.2 ne peuvent donc plus être directement appliquées.

Lorsque l'espace des solutions est fini et discret, on parle de manière plus générale de problèmes d'*optimisation combinatoire* [18]. Bien que l'énumération soit naturellement envisageable pour de petits problèmes comme notre exemple, le temps de calcul demandé ne nous le permet généralement pas, puisqu'il augmente exponentiellement avec le nombre de variables entières. En particulier, le problème (PLE) est NP-difficile, c'est-à-dire qu'il n'existe aucun algorithme permettant de le résoudre en temps polynomial.

1.3.2 Relaxations et bornes

Pour résoudre le problème (PLE), l'approche générale est de considérer des méthodes itératives calculant des estimations de la valeur objectif optimale $z^* := f(\boldsymbol{x}^*)$. Dans un problème de minimisation, une *borne inférieure* de z^* est une valeur $z \leq z^*$ qui sous-estime la valeur objectif optimale. De manière similaire, une *borne supérieure* de z^* est une valeur $z \geq z^*$ qui surestime la valeur objectif optimale. Ainsi, il suffit de considérer simultanément une séquence croissante de bornes inférieures et une séquence décroissante de bornes supérieures jusqu'à ce que les valeurs estimées coïncident. À ce moment, une solution optimale au problème est trouvée et l'optimalité est démontrée.

En général, la seule façon de générer des bornes supérieures pour le problème (PLE) est de trouver des solutions admissibles. En effet, chaque solution admissible $\bar{\boldsymbol{x}}$ du problème fournit une borne supérieure $f(\bar{\boldsymbol{x}})$. Par ailleurs, le calcul de bornes inférieures est fait en utilisant différentes techniques de *relaxation*, c'est-à-dire en considérant plutôt certaines simplifications du problème original.

Relaxation linéaire

La *relaxation linéaire* du problème (PLE) est obtenue en retirant l'ensemble des contraintes d'intégralité sur $\boldsymbol{x}$, c'est-à-dire en permettant aux variables du problème de prendre une valeur réelle. Le problème (PL) est donc la relaxation linéaire du problème (PLE) et les techniques de résolution présentées à la section 1.2 s'appliquent. Cette technique permet efficacement d'obtenir une borne inférieure valide. Par exemple, la valeur $f(4, 3/2) = -15$ (figure 1.1) est bien une borne inférieure de la valeur optimale $f(3, 2) = -13$ (figure 1.2).

Relaxation de contraintes

Une *relaxation de contraintes* du problème (PLE) est obtenue en retirant une ou plusieurs contraintes du problème. L'espace des solutions se voit alors agrandi, ce qui génère un problème moins contraint et une borne inférieure de la valeur objectif optimale. En général,

l'ensemble choisi sera composé de contraintes compliquées afin de rendre le problème relaxé plus facile à résoudre que le problème original.

Relaxation lagrangienne

La *relaxation lagrangienne* est une technique qui consiste à retirer certaines contraintes du problème qui le rendent difficile à résoudre tout en ajoutant à la fonction objectif une pénalité de non-respect de ces contraintes [62, 76]. Plus précisément, considérons le problème d'optimisation linéaire suivant formé de deux familles de contraintes, $\mathcal{A}$: $A\boldsymbol{x} \leq \boldsymbol{b}$ et $\mathcal{B}$: $B\boldsymbol{x} \leq \boldsymbol{d}$, où $X \subseteq \mathbb{R}^n$ est un ensemble arbitraire :

$$\begin{aligned} \min_{\boldsymbol{x}} \quad & f(\boldsymbol{x}) = \boldsymbol{c}^T\boldsymbol{x} \\ \text{s. à} \quad & A\boldsymbol{x} \leq \boldsymbol{b} \\ & B\boldsymbol{x} \leq \boldsymbol{d} \\ & \boldsymbol{x} \in X. \end{aligned} \tag{P}$$

Supposons que $\mathcal{A}$ est formée de m contraintes difficiles. La relaxation lagrangienne déplace ces contraintes dans la fonction objectif tout en gardant la structure du problème original. En introduisant un vecteur $\boldsymbol{\lambda} = (\lambda_1, \ldots, \lambda_m)^T \in \mathbb{R}^m$ de *multiplicateurs de Lagrange* $\lambda_i \geq 0$, $i \in \{1, \ldots, m\}$, la fonction objectif $f(\boldsymbol{x})$ est modifiée pour être pénalisée lorsque les contraintes de $\mathcal{A}$ sont violées, menant au problème relaxé suivant :

$$\begin{aligned} \min_{\boldsymbol{x}} \quad & g_{\boldsymbol{\lambda}}(\boldsymbol{x}) = \boldsymbol{c}^T\boldsymbol{x} + \boldsymbol{\lambda}^T(A\boldsymbol{x} - \boldsymbol{b}) \\ \text{s. à} \quad & B\boldsymbol{x} \leq \boldsymbol{d} \\ & \boldsymbol{x} \in X. \end{aligned} \tag{RL($\boldsymbol{\lambda}$)}$$

Pour tout $\boldsymbol{\lambda} \geq 0$, si $\bar{\boldsymbol{x}}$ est la solution optimale du problème (RL($\boldsymbol{\lambda}$)) et $\boldsymbol{x}^*$ est la solution optimale du problème (P), $g_{\boldsymbol{\lambda}}(\bar{\boldsymbol{x}})$ est une borne inférieure de $f(\boldsymbol{x}^*)$. En effet, il suffit de remarquer que $g_{\boldsymbol{\lambda}}(\bar{\boldsymbol{x}}) \leq g_{\boldsymbol{\lambda}}(\boldsymbol{x}^*)$ par optimalité de $\bar{\boldsymbol{x}}$, et que

$$g_{\boldsymbol{\lambda}}(\boldsymbol{x}^*) = \boldsymbol{c}^T\boldsymbol{x}^* + \boldsymbol{\lambda}^T(A\boldsymbol{x}^* - \boldsymbol{b}) \leq \boldsymbol{c}^T\boldsymbol{x}^* = f(\boldsymbol{x}^*),$$

car $A\boldsymbol{x}^* \leq \boldsymbol{b}$ et $\boldsymbol{\lambda} \geq 0$.

En vue d'obtenir la plus grande borne inférieure possible, le *problème des multiplicateurs de Lagrange* cherche à maximiser la valeur de $g_{\boldsymbol{\lambda}}(\bar{\boldsymbol{x}})$ parmi $\boldsymbol{\lambda} \geq 0$. Plusieurs méthodes existent pour résoudre celui-ci, incluant mais non limitées à la *génération de colonnes* [43] et la *descente de sous-gradient* [1, 5, 44]. Cette dernière choisit de façon itérative les multiplicateurs de Lagrange en fonction de la solution optimale du problème (RL($\boldsymbol{\lambda}$)) jusqu'à convergence.

1.3.3 Résolution

Dans la littérature, il existe différentes stratégies pour résoudre de manière exacte le problème (PLE), c'est-à-dire pour trouver sa solution optimale et démontrer son optimalité [62,

76, 83]. Les méthodes de type *séparation et évaluation*, *branch and bound* en anglais, font partie des plus importantes. Introduites en 1960 par Land et Doig [58], elles consistent essentiellement à diviser récursivement l'espace des solutions en plusieurs sous-problèmes (*séparation*), tout en calculant une borne inférieure de la fonction objectif dans chacun de ces sous-problèmes (*évaluation*). L'étape de séparation est effectuée tant qu'une solution admissible n'a pas été trouvée dans le sous-espace considéré et que les bornes déterminées indiquent qu'il est toujours envisageable d'en trouver une. De plus, l'évaluation est généralement réalisée en considérant la relaxation linéaire du sous-problème courant. La procédure se termine lorsque l'espace des solutions est entièrement exploré et que l'optimalité est démontrée. Ces méthodes peuvent être représentées sous la forme d'un *arbre de recherche*.

Les *méthodes de plans sécants*, en anglais *cutting-plane methods*, sont d'autres stratégies de résolution couramment utilisées. Introduites par Gomory en 1958 [39], celles-ci reposent sur l'ajout itératif de contraintes à la relaxation du problème. Lorsque la solution optimale de la relaxation n'est pas admissible au niveau du problème original, ces méthodes cherchent à ajouter des contraintes d'inégalité à la relaxation pour raffiner l'espace de solutions et retirer la solution actuelle de celui-ci. Ces contraintes additionnelles sont appelées *plans sécants*. Il peut s'agir de plans sécants spécifiques au problème considéré ou de *coupes de Gomory*. Ces dernières ont l'avantage d'être générées automatiquement par l'algorithme du simplexe. La méthode *séparation et coupe*, en anglais *branch and cut*, intègre efficacement la génération de plans sécants à l'approche de séparation et évaluation [64].

1.4 Théorie des graphes

La *théorie des graphes* est une branche des mathématiques discrètes qui étudie les *graphes*, une structure abstraite permettant de représenter visuellement les relations entre divers objets. Celle-ci nous permet efficacement de modéliser une multitude de problèmes ayant diverses applications, incluant le *problème du commis voyageur* (voir section 1.5). Les définitions et les descriptions d'algorithmes qui suivent sont librement inspirées de Cormen *et al.* [19].

1.4.1 Graphe

Un *graphe non orienté*, ou simplement *graphe*, est une paire (V, E) de deux ensembles finis V et E, où E définit une relation binaire sur les éléments de V. L'ensemble V est appelé l'*ensemble des sommets* et contient les *sommets* du graphe (en anglais, *vertices*). L'ensemble E est appelé l'*ensemble des arêtes* et contient les *arêtes* du graphe (en anglais, *edges*). Une arête est une paire non ordonnée de sommets, c'est-à-dire un ensemble $\{u, v\}$ avec $u, v \in V$ et $u \neq v$. À titre d'exemple, la figure 1.3 illustre un graphe défini sur l'ensemble de sommets $\{1, 2, 3, 4, 5, 6\}$. Dans ce genre de représentation, les cercles et les lignes correspondent respectivement aux sommets et aux arêtes.

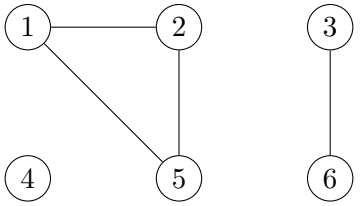


FIGURE 1.3 – Représentation du graphe $G = (V, E)$, où $V = \{1, 2, 3, 4, 5, 6\}$ et $E = \{\{1, 2\}, \{1, 5\}, \{2, 5\}, \{3, 6\}\}$.

Dans un *graphe orienté*, les arêtes de E sont des paires ordonnées de sommets, c'est-à-dire des couples (u, v) avec $u, v \in V$. Ainsi, dans ce type de graphe, les arêtes (u, v) et (v, u) sont distinctes et (u, u) est une arête permise. Il convient donc de représenter plutôt les arêtes à l'aide de flèches unidirectionnelles. Dans ce qui suit, nous ne considérons que le cas non orienté. Toutefois, il est bien de garder à l'esprit qu'il existe des définitions fortement similaires pour les graphes orientés.

Si $\{u, v\} \in E$, on dit que les sommets u et v sont *adjacents*, et que l'arête est *incidente* aux sommets u et v. De plus, on définit $\delta(u) \coloneqq \{e \in E : u \in e\}$ comme étant l'ensemble des arêtes de E qui sont incidentes au sommet $u \in V$. Par exemple, à la figure 1.3, le sommet 1 est adjacent aux sommets 2 et 5, et $\delta(1) = \{\{1, 2\}, \{1, 5\}\}$. Le graphe est *complet* si tous ses sommets sont adjacents.

Le *degré* d'un sommet $u \in V$ du graphe correspond au nombre d'arêtes qui lui sont incidentes, soit $|\delta(u)|$. De façon plus générale, on définit $\deg_A(u) \coloneqq |\{e \in A : u \in e\}|$ comme étant le nombre d'arêtes incidentes à $u \in V$ dans le sous-ensemble d'arêtes $A \subseteq E$. Par exemple, à la figure 1.3, le sommet 5 est de degré 2, et $\deg_{\{\{2,5\},\{3,6\}\}}(5) = 1$. Le sommet 4 est dit *isolé*, puisque son degré est nul.

1.4.2 Chaîne, cycle, coupe et connexité

Dans un graphe $G = (V, E)$, une *chaîne* de longueur k allant du sommet u au sommet u' est une séquence $\langle v_0, v_1, \ldots, v_k \rangle$ de sommets tels que $u = v_0$, $u' = v_k$ et $\{v_i, v_{i+1}\} \in E$ pour $i \in \{0, \ldots, k-1\}$. La longueur correspond au nombre d'arêtes dans la chaîne. Les sommets u et u' sont appelés les *extrémités* de la chaîne. Par exemple, à la figure 1.3, $\langle 1, 2, 5 \rangle$ est une chaîne de longueur 2 dont les extrémités sont 1 et 5.

Une chaîne $\langle v_0, v_1, \ldots, v_k \rangle$ est appelée un *cycle* si $k > 0$, $v_0 = v_k$ et toutes les arêtes de la chaîne sont distinctes. Le cycle est dit *élémentaire* si, en plus, tous les sommets qui la composent sont distincts, sauf ses deux extrémités. À la figure 1.3, la chaîne $\langle 1, 2, 5, 1 \rangle$ est un cycle élémentaire. Un graphe qui ne contient aucun cycle élémentaire est dit *acyclique*. Finalement, un *cycle hamiltonien* d'un graphe est un cycle élémentaire passant par l'ensemble des sommets de ce graphe, parcourant par le fait même chaque sommet une et une seule fois. La figure 1.4 illustre un graphe admettant un cycle hamiltonien.

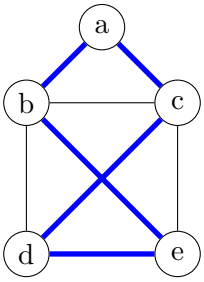


FIGURE 1.4 – Un graphe de sommets a, b, c, d, e admettant le cycle hamiltonien $\langle a, b, e, d, c, a\rangle$, en bleu.

Une *coupe* $C = (A, B)$ d'un graphe $G = (V, E)$ est une partition de l'ensemble des sommets V en deux sous-ensembles $A, B \subseteq V$ tels que $B = V \setminus A$. Son *ensemble de coupe* associé est défini par l'ensemble des arêtes qui la traverse, c'est-à-dire $\{\{i, j\} \in E : i \in A, j \in B\}$. Par exemple, à la figure 1.3, la coupe $(\{1, 2, 3\}, \{4, 5, 6\})$ détermine l'ensemble de coupe $\{\{1, 5\}, \{2, 5\}, \{3, 6\}\}$.

Un graphe $G = (V, E)$ est *connexe* si chaque sommet peut être atteint à partir de tous les sommets de G. Autrement dit, il est connexe si pour toute paire $u, v \in V$, il existe une chaîne ayant u et v comme extrémités. Aussi, une *composante connexe* d'un graphe est un sous-ensemble maximal de sommets induisant un graphe connexe. Le graphe de la figure 1.3 n'est pas connexe, mais comporte trois composantes connexes, soit $\{1, 2, 5\}$, $\{3, 6\}$ et $\{4\}$. Par ailleurs, le graphe de la figure 1.4 est connexe.

1.4.3 Arbre

Un *arbre* est un graphe qui est connexe et acyclique. Les sommets d'un arbre sont appelés *nœuds* et les sommets de degré 1 sont désignés spécifiquement par *feuilles*. La figure 1.5 illustre un exemple d'arbre. Pour un arbre $T = (V, E)$, on a $|E| = |V| - 1$. De plus, l'ajout d'une arête à E induit nécessairement un unique cycle dans le graphe. Par ailleurs, le retrait d'une arête de E rend le graphe non connexe avec exactement deux composantes connexes.

Un *arbre couvrant* d'un graphe connexe $G = (V, E)$ est un sous-ensemble acyclique d'arêtes $T \subseteq E$ connectant tous les sommets de G. L'ensemble T forme donc un arbre qui *couvre* le graphe G. La figure 1.6 illustre un exemple d'arbre couvrant. En fait, il s'agit même d'un *arbre couvrant minimal*, comme nous le verrons en détail à la section 1.4.4.

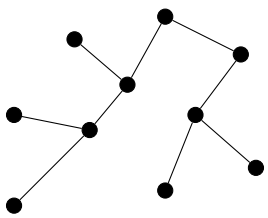


FIGURE 1.5 – Exemple d'arbre à 10 nœuds, dont 5 feuilles.

1.4.4 Arbre couvrant minimal

Dans plusieurs contextes, il est essentiel de plutôt considérer un *graphe pondéré*. Pour un graphe $G = (V, E)$, chaque arête de G est associée à un *poids*, aussi parfois appelé *coût*, une valeur numérique spécifique pour l'arête. Typiquement, le poids d'une arête est obtenu grâce à une *fonction de poids* $w\colon E \to \mathbb{R}$. Pour une arête $\{u, v\} \in E$, on dénote son poids par $w(u, v)$. Par extension, pour un ensemble d'arêtes $A \subseteq E$, le poids total $w(A)$ de l'ensemble est donné par la somme des poids des arêtes dans A, c'est-à-dire

$$w(A) \coloneqq \sum_{\{u,v\} \in A} w(u, v).$$

Étant donné un graphe connexe $G = (V, E)$ muni d'une fonction de poids w, le *problème de l'arbre couvrant de poids minimal* consiste à déterminer un arbre couvrant T du graphe G tel que son poids total $w(T)$ soit minimal. Cet arbre T est appelé un *arbre couvrant (de poids) minimal*. Notez que celui-ci n'est pas toujours unique. La figure 1.6 montre un exemple d'arbre couvrant minimal d'un graphe connexe.

Les deux propriétés suivantes donnent des conditions nécessaires et suffisantes pour qu'un arbre couvrant d'un graphe soit minimal [70, 82].

Propriété de cycles. *Un arbre couvrant T d'un graphe $G = (V, E)$ est minimal si, et seulement si, pour toute arête $\{i, j\} \in E \setminus T$, pour toute arête $(u, v) \in T$ de l'unique cycle dans $T \cup \{i, j\}$, on a $w(i, j) \geq w(u, v)$.*

Propriété de coupes. *Un arbre couvrant T d'un graphe $G = (V, E)$ est minimal si, et seulement si, pour toute arête $\{i, j\} \in T$, pour toute arête $(u, v) \in E \setminus T$ de l'ensemble de coupe formé par le retrait de $\{i, j\}$ de T, on a $w(i, j) \leq w(u, v)$.*

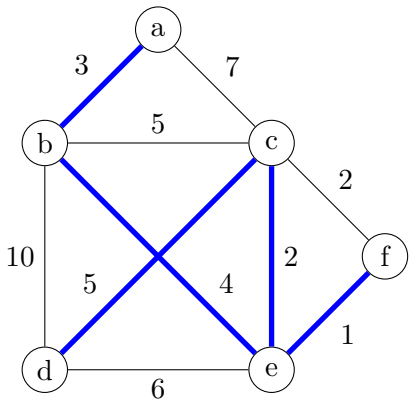


FIGURE 1.6 – Un arbre couvrant minimal, en bleu, pour un graphe connexe. Les poids sont indiqués à côté de chaque arête. Le poids total de l'arbre est de 15. Il n'est pas unique : $\{c, e\}$ peut être remplacée par $\{c, f\}$.

Plusieurs algorithmes faisant usage de ces propriétés existent dans la littérature pour résoudre le problème. Parmi les classiques, l'*algorithme de Kruskal*, introduit par Kruskal en 1956 [56], permet de construire aisément un arbre couvrant minimal de façon vorace. En triant d'abord les arêtes par ordre croissant de poids, chaque arête est parcourue puis ajoutée à l'arbre dans le cas où elle ne crée pas de cycle. En utilisant une structure de données particulière pour garder en mémoire les nœuds appartenant à la même composante connexe (*ensembles disjoints*, voir Tarjan [80]), l'algorithme s'exécute en pire cas $O(|E| \log |V|)$. L'*algorithme de Prim*, introduit par Prim en 1957 [66], est un autre algorithme vorace pour construire un arbre couvrant minimal. Partant d'un sommet arbitraire, les arêtes sont ajoutées une par une à l'arbre en choisissant toujours l'arête de poids minimal parmi celles incidentes aux sommets visités et ne créant pas de cycle. L'utilisation d'un tas de Fibonacci permet d'atteindre une complexité en pire cas de $O(|E| + |V| \log |V|)$. Finalement, la meilleure complexité en temps connue à ce jour pour calculer un arbre couvrant minimal est $O(|E|\, \alpha(|E|, |V|))$, où $\alpha(m, n)$ est la *fonction inverse d'Ackermann*, qui est en pratique au plus égale à 4 [15, 80].

1.5 Problème du commis voyageur

1.5.1 Définition

Le *problème du commis voyageur*, aussi connu sous le nom de *problème du voyageur de commerce* ou *traveling salesman problem* (TSP) en anglais, est le problème d'optimisation combinatoire auquel s'intéresse ce mémoire [4]. Il consiste à déterminer, étant donné un certain nombre de villes et la distance entre chacune d'elles, le plus court chemin parcourant chaque ville une seule fois et terminant à son point de départ. Plus formellement, étant donné un graphe non orienté $G = (V, E)$ connexe formé d'un ensemble V de villes et pondéré par la fonction $w \colon E \to \mathbb{R}$, où $w(u, v)$ est la distance entre les villes u et v, il s'agit du problème de déterminer un cycle hamiltonien C de G de poids total $w(C)$ minimal. La figure 1.7 illustre un exemple du TSP pour un graphe complet de 4 sommets.

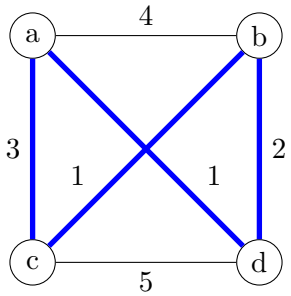


FIGURE 1.7 – Exemple du TSP pour un graphe complet de 4 sommets. Le cycle hamiltonien le plus court est $\langle a, d, b, c, a \rangle$, en bleu, de poids total 7.

Pour être plus précis, pour le problème précédemment décrit, il faudrait dire qu'il s'agit de la version *symétrique*. Lorsque le graphe considéré est orienté, et donc qu'il est possible d'avoir $w(u, v) \neq w(v, u)$ pour $(u, v) \in E$, on parle plutôt du TSP *asymétrique*. Dans ce mémoire, nous restreignons notre étude au cas symétrique du problème. Toutefois, il est à noter que toute instance asymétrique peut être transformée en instance symétrique en doublant le nombre de sommets considérés grâce à l'approche de Jonker et Volgenant [51].

Comme mentionné à l'introduction, la principale difficulté de ce problème réside dans l'explosion combinatoire du nombre de possibilités en fonction du nombre de sommets du graphe. Débutant le cycle en un sommet quelconque d'un graphe à n sommets, il existe initialement $n - 1$ successeurs potentiels. Puis, lorsque deux sommets sont choisis, il ne reste que $n - 2$ sommets possibles pour continuer le chemin, et ainsi de suite. Le processus de construction d'un cycle de n sommets génère donc au total $(n-1)(n-2)\cdots(2)(1) = (n-1)!$ possibilités. Ce nombre doit être divisé par deux, puisque le graphe est non orienté et qu'un cycle est équivalent qu'il soit parcouru dans un sens ou dans l'autre. Ainsi, pour une collection de n villes, il existe $\frac{1}{2}(n-1)!$ chemins candidats. Dans notre exemple à 4 sommets, cela implique uniquement 3 chemins à essayer. Pour 10 sommets, il y a 181 440 cycles possibles. Et pour 50 sommets, ce nombre atteint l'ordre de 10^{62}. Une recherche exhaustive n'est donc pas un algorithme de résolution viable. En fait, dans sa formulation la plus simple telle qu'énoncée précédemment, le TSP est un problème NP-difficile, c'est-à-dire qu'on ne connaît pas d'algorithme en temps polynomial pour le résoudre.

1.5.2 Historique

Bien que ses origines soient incertaines et pourraient remonter jusqu'au début des années 1800, le TSP fascine les chercheurs depuis sa première formulation mathématique en 1930 [4]. L'intérêt de celui-ci est qu'il apparaît naturellement dans une multitude de problématiques de transport et industrielles. De plus, le TSP et ses nombreuses variantes trouvent des applications dans un important nombre de domaines en apparence non liés, allant de la logistique au séquençage de l'ADN.

Dans les années 1940, le problème gagne rapidement en popularité grâce à sa complexité. La première avancée majeure dans la résolution du TSP survient en 1954 alors que Dantzig, Fulkerson et Johnson résolvent une instance de 49 villes aux États-Unis (à la main!) à l'aide d'une méthode de plans sécants [21]. Partant d'une relaxation prenant la forme d'un *problème d'affectation*, problème pour lequel il existe des algorithmes efficaces en temps polynomial [57], leur approche ajoute graduellement des plans sécants spécifiques à la solution relaxée, notamment des *contraintes d'élimination de sous-cycle* et des *inégalités de peigne* (voir Applegate *et al.* [4]), jusqu'à la solution optimale. Cette approche sera au cœur de l'ensemble des travaux portant sur la résolution exacte du TSP pour les années qui suivront.

À partir des années 1970, plusieurs travaux importants permettent de résoudre des problèmes allant jusqu'à 2392 villes, incluant notamment la *relaxation 1-tree* de Held et Karp [43, 44] (présentée en détail à la section 1.5.4), l'*heuristique de Lin-Kernighan* [61], l'utilisation de *peignes généralisés* [17, 41] et l'implémentation de la technique *séparation et coupe* [20, 40, 47, 65].

Dans les années 1990, Applegate, Bixby, Chvátal et Cook développent le solveur *Concorde* [3, 4]. En intégrant intelligemment l'ensemble des approches mentionnées précédemment, il permet de résoudre de manière exacte des instances du TSP comportant des milliers de villes en seulement quelques secondes. En 2006, il trouve la solution optimale d'un problème de 85 900 villes, résolvant ainsi la totalité des 110 instances difficiles de la librairie TSPLIB [72, 73]. L'une de ses particularités est qu'il utilise des algorithmes sophistiqués pour générer les meilleurs plans sécants possible et pour améliorer ceux déjà trouvés. Encore aujourd'hui, il est l'état de l'art pour ce problème. L'utilisation d'une combinaison des techniques de *Concorde* et de l'*heuristique LKH*, une implémentation efficace de l'heuristique de Lin-Kernighan [45, 46], a même récemment mené à la résolution d'un TSP tridimensionnel entre 109 399 étoiles [2].

Comme mentionné à l'introduction, l'une des principales limitations de *Concorde* est qu'il ne permet de résoudre que le TSP *pur*, c'est-à-dire qu'il ne permet pas de considérer des contraintes additionnelles sur le cycle à déterminer. Par exemple, il pourrait s'agir de fenêtres de temps pour chaque visite, la capacité des véhicules ou l'évitement du trafic, contraintes qui surviennent naturellement dans toute application pratique du TSP. Cette limitation motive donc l'utilisation d'une approche générique telle que la *programmation par contraintes*, présentée en détail à la section 1.6.

1.5.3 Formulation linéaire

Soit $G = (V, E)$ un graphe connexe, où $V \coloneqq \{1, 2, \ldots, n\}$ est l'ensemble des sommets (*villes*), E est l'ensemble des arêtes (*routes*) et $w(i, j) \in \mathbb{R}$ est le poids de l'arête $\{i, j\} \in E$ (*distance entre les villes i et j*). En introduisant une variable binaire x_e pour tout $e \in E$, où x_e prendra la valeur de 1 si l'arête e fait partie du cycle hamiltonien, le TSP peut être modélisé comme un problème d'optimisation linéaire en nombres entiers [4] :

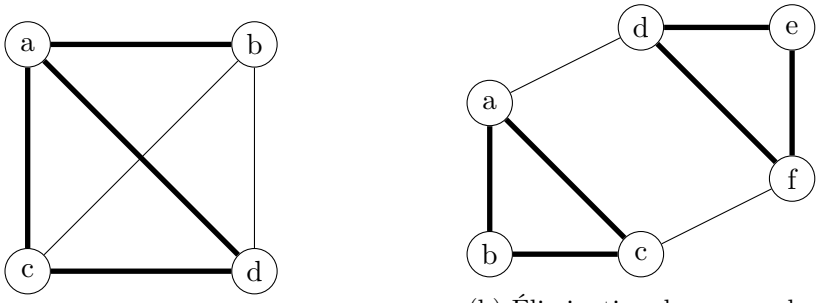
$$\min_{\boldsymbol{x}} \quad f(\boldsymbol{x}) = \sum_{e \in E} w(e) x_e \tag{1.4}$$

$$\text{s. à} \quad \sum_{e \in \delta(i)} x_e = 2 \qquad \forall i \in V \tag{1.5}$$

$$\sum_{\substack{i,j \in N, \\ i<j}} x_{\{i,j\}} \leq |N| - 1 \qquad \forall N \subsetneq V, |N| \geq 3 \tag{1.6}$$

$$x_e \in \{0, 1\} \qquad \forall e \in E. \tag{1.7}$$

La fonction objectif (1.4) à minimiser correspond au poids total du cycle. Les contraintes (1.5) sont les *contraintes de degré* et forcent chaque sommet à avoir deux arêtes incidentes. Les contraintes (1.6) sont les *contraintes d'élimination de sous-cycle* et exigent que les arêtes choisies ne forment qu'un unique cycle, donc que le cycle soit connexe. Notez qu'il existe $O(2^n)$ contraintes de ce type, un nombre exponentiel en fonction du nombre de sommets. Finalement, les contraintes (1.7) forcent les variables du problème à être binaires. La figure 1.8 illustre les différentes contraintes du modèle.



(a) Degré.
a et b n'ont pas un degré de 2.

(b) Élimination de sous-cycle.
$\langle a, b, c, a\rangle$ et $\langle d, e, f, d\rangle$ forment deux cycles distincts.

FIGURE 1.8 – Illustration des situations particulières empêchées par chaque type de contraintes. Les arêtes en gras seraient sélectionnées, c'est-à-dire auraient leur variable associée égale à 1.

1.5.4 Relaxation *1-tree*

La *relaxation 1-tree*, introduite en 1970 par Held et Karp [43, 44], est obtenue en relaxant les contraintes (1.5) de degré. La contrainte est uniquement conservée pour le sommet arbitraire 1 et une contrainte redondante est ajoutée au modèle, menant à la nouvelle formulation suivante, où les modifications sont identifiées en bleu :

$$\begin{aligned}
\min_{\boldsymbol{x}} \quad & f(\boldsymbol{x}) = \sum_{e\in E} w(e)x_e & & \\
\text{s. à} \quad & \sum_{e\in\delta(1)} x_e = 2 & \cancel{\forall i \in V} & \quad (1.8)\\
& \sum_{e\in E} x_e = |V| & & \quad (1.9)\\
& \sum_{i,j\in N, i<j} x_{\{i,j\}} \leq |N| - 1 & \forall N \subsetneq V\backslash\{1\}, |N| \geq 3 & \quad (1.10)\\
& x_e \in \{0,1\} & \forall e \in E. &
\end{aligned}$$

Soit $G' = (V', E')$ le graphe G sans le sommet 1, c'est-à-dire avec $V' \coloneqq V\backslash\{1\}$ et $E' \coloneqq E\backslash\delta(1)$. Les contraintes (1.8), (1.9) et (1.10) définissent la structure d'un *1-tree*[2] de G, un arbre couvrant de G' auquel on ajoute deux arêtes distinctes incidentes au sommet 1. Ainsi, la

2. Se traduit littéralement de l'anglais par *1-arbre*, puisqu'il s'agit presque d'un arbre, à un seul cycle près.

relaxation *1-tree* consiste simplement à déterminer un *1-tree* de G de poids total minimal. Cela peut être fait efficacement en calculant un arbre couvrant minimal de G', puis en y ajoutant les deux arêtes de poids minimal incidentes au sommet 1. On peut se convaincre qu'il s'agit bien d'une relaxation du TSP en notant que tout cycle hamiltonien dans un graphe est un *1-tree* et que si un *1-tree* minimal forme un cycle, alors il est une solution optimale du TSP. Un exemple simple de *1-tree* minimal est illustré à la figure 1.9.

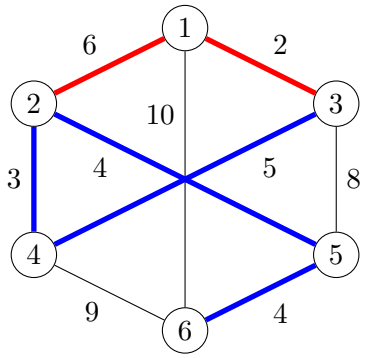


FIGURE 1.9 – Exemple d'un *1-tree* de poids minimal d'un graphe G qui n'est pas un cycle, représenté par les arêtes de couleur. Les arêtes bleues forment un arbre couvrant minimal de G'. Les arêtes rouges sont les deux arêtes de poids minimal de $\delta(1)$. La borne inférieure correspondante est 24 (optimale : 28).

La borne inférieure fournie par la relaxation *1-tree* peut être améliorée grâce à une relaxation lagrangienne des contraintes (1.5) de degré précédemment retirées. En introduisant un vecteur $\boldsymbol{\lambda} = (\lambda_1, \ldots, \lambda_n)^T \in \mathbb{R}^n$ de multiplicateurs de Lagrange [3], avec $\lambda_1 = 0$, le problème relaxé devient

$$\begin{aligned} \min_{\boldsymbol{x}} \quad & g_{\boldsymbol{\lambda}}(\boldsymbol{x}) = \sum_{e \in E} w(e) x_e + \sum_{i \in V} \lambda_i (\deg_T(i) - 2) \\ \text{s. à} \quad & T \text{ est un } \textit{1-tree}, \text{ où } e \in T \Leftrightarrow (x_e = 1) \\ & x_e \in \{0, 1\} \quad \forall e \in E. \end{aligned} \tag{1.11}$$

Étant donné $\boldsymbol{\lambda} \in \mathbb{R}^n$, on note $Z_{RL}(\boldsymbol{\lambda})$ la valeur optimale de ce problème. L'intérêt de cette formulation est que, pour tout $\boldsymbol{\lambda} \in \mathbb{R}^n$, $Z_{RL}(\boldsymbol{\lambda})$ est une borne inférieure valide du TSP. De plus, en remarquant que la fonction objectif (1.11) peut être réécrite sous la forme

$$\sum_{\{i,j\} \in E} (w(i,j) + \lambda_i + \lambda_j) x_{\{i,j\}} - 2 \sum_{i \in V} \lambda_i,$$

$Z_{RL}(\boldsymbol{\lambda})$ peut être obtenue en calculant un *1-tree* minimal avec la fonction de poids modifiée $\tilde{w}(i,j) \coloneqq w(i,j) + \lambda_i + \lambda_j, \forall \{i,j\} \in E$. La meilleure borne possible, aussi appelée *borne de*

3. Notez ici que les multiplicateurs de Lagrange peuvent prendre des valeurs négatives. Cela provient du fait qu'une contrainte d'égalité $\boldsymbol{ax} = b$ correspond simultanément à deux contraintes d'inégalité, $\boldsymbol{ax} \leq b$ et $-\boldsymbol{ax} \leq -b$. La pénalisation de la fonction objectif est donc de la forme $\lambda_1(\boldsymbol{ax} - b) + \lambda_2(-\boldsymbol{ax} - (-b)) = (\lambda_1 - \lambda_2)(\boldsymbol{ax} - b)$, pour $\lambda_1, \lambda_2 \geq 0$. Cette formulation est équivalente à $\lambda(\boldsymbol{ax} - b)$ pour $\lambda \in \mathbb{R}$.

Held-Karp, est alors la valeur optimale du problème des multiplicateurs de Lagrange, c'est-à-dire la valeur $Z_{RL}(\boldsymbol{\lambda})$ maximale parmi $\boldsymbol{\lambda} \in \mathbb{R}^n$.

L'approche originale de Held et Karp pour calculer la borne optimale repose sur une descente de sous-gradient [44]. Initialement, les multiplicateurs de Lagrange $\boldsymbol{\lambda}$ sont tous fixés à 0 ($\boldsymbol{\lambda}^{(0)} = \mathbf{0}$). À l'itération $k \geq 0$ de l'algorithme, les multiplicateurs $\boldsymbol{\lambda}^{(k)}$ donnent une borne $Z_{RL}(\boldsymbol{\lambda}^{(k)})$ et un *1-tree* minimal $T^{(k)}$. Si $T^{(k)}$ est un cycle hamiltonien de G, il s'agit nécessairement de la solution optimale du TSP et la procédure se termine. Sinon, les contraintes (1.5) de degré sont violées pour au moins un sommet de V'. Dans ce cas, pour chaque sommet $i \in V'$ tel que $\deg_{T^{(k)}}(i) \neq 2$, le multiplicateur associé λ_i est modifié de façon à pénaliser davantage la fonction objectif (1.11) et donc forcer le *1-tree* correspondant à respecter la contrainte. La mise à jour des multiplicateurs $\boldsymbol{\lambda}$ est de la forme

$$\lambda_i^{(k+1)} \coloneqq \lambda_i^{(k)} + p^{(k)}(\deg_{T^{(k)}}(i) - 2), \quad \forall i \in V',$$

où $p^{(k)}$ est la *longueur du pas* à l'itération k. Certaines conditions mathématiques sur cette longueur permettent d'assurer la convergence de l'algorithme [1]. Toutefois, en pratique, on choisit

$$p^{(k)} \coloneqq a^{(k)} \left(\frac{U - Z_{LR}(\boldsymbol{\lambda}^{(k)})}{\sum_{i \in V'} (\deg_{T^{(k)}}(i) - 2)^2} \right),$$

où U est une borne supérieure du TSP et $a^{(k)} \in (0, 2]$ est un paramètre correspondant à l'*agilité* du sous-gradient à l'itération k [1, 5, 36, 49]. Dans les travaux de Held et Karp [44], l'agilité est choisie arbitrairement et fixée pour l'ensemble des itérations. De façon plus répandue, elle est initialisée à 2, puis divisée par 2 après un certain nombre d'itérations sans amélioration de la borne [1, 36]. Dans le code de *Choco Graph* [27, 29] utilisé dans nos expériences du chapitre 3, les paramètres par défaut du sous-gradient définissent l'agilité par $a^{(0)} \coloneqq 2$ et

$$a^{(k)} \coloneqq 2^{1-b^{(k)}},\ b^{(k)} \coloneqq \frac{(\lfloor k/30 \rfloor)\,(\lfloor k/30 \rfloor + 1)}{2} \quad \forall k > 0.$$

1.6 Programmation par contraintes

1.6.1 Définition

La *programmation par contraintes*, en anglais *constraint programming* (CP), est un paradigme de programmation qui permet de résoudre des problèmes de recherche combinatoire, c'est-à-dire des problèmes où l'espace des solutions est fini et discret [75]. Il est à l'intersection de l'intelligence artificielle, l'informatique, les mathématiques et la recherche opérationnelle. La particularité de cette technique est qu'elle tente de séparer la *modélisation* du problème, étape durant laquelle un utilisateur doit exprimer les différentes caractéristiques de celui-ci, de la *résolution* du problème, étape entièrement prise en charge par le *solveur de contraintes* utilisé.

Un modèle en CP consiste d'abord en un ensemble de variables de décision, généralement de type entier ou booléen, munies chacune d'un ensemble de valeurs possibles appelé *domaine*. Le domaine d'une variable x, noté $\text{dom}(x)$, peut être donné sous la forme d'une énumération de valeurs ou d'un intervalle. Ensuite, l'ensemble des restrictions sur les variables et des relations entre elles sont exprimées sous la forme de contraintes choisies parmi le catalogue de contraintes du solveur utilisé. Celles-ci peuvent être arithmétiques ($x \leq 8$, $x \neq y$, $z > 0$), logiques ($x \vee y$, $y \Rightarrow z$, $\neg x \wedge w$) ou *globales*, c'est-à-dire des contraintes définies sur un nombre arbitraire de variables du problème. Par exemple, la contrainte globale $\textsc{AllDifferent}(x_1, x_2, \ldots, x_n)$ encode le fait que les n variables doivent prendre simultanément des valeurs distinctes. On définit la *portée* d'une contrainte comme étant l'ensemble des variables concernées par celle-ci. Dans le cas d'un problème d'optimisation, il faut aussi spécifier dans le modèle la fonction objectif qui sera à minimiser ou à maximiser. Les problèmes qui n'ont pas de fonction objectif sont appelés *problèmes de satisfaction* et consistent simplement à déterminer une solution satisfaisant l'ensemble des contraintes du modèle.

1.6.2 Arbre de recherche

En CP, la résolution s'effectue généralement grâce à une *recherche en profondeur* afin d'énumérer l'ensemble des solutions admissibles du problème [68, 75]. Elle consiste à générer un *arbre de recherche* où chaque nœud de l'arbre correspond à une *solution partielle*, c'est-à-dire une solution où uniquement quelques variables sont assignées à une valeur. La *racine* de l'arbre est la solution partielle initiale où aucune variable n'est assignée et est généralement représentée comme le nœud le plus haut. Chaque nœud hérite de la solution partielle de son nœud parent et y ajoute une *assignation*, c'est-à-dire un choix de variable non assignée et un choix de valeur parmi le domaine de cette variable. Cette action est appelée *branchement*.

Chaque contrainte du problème doit être munie dans le solveur d'un algorithme efficace de vérification, d'une procédure permettant de déterminer étant donné une solution si celle-ci satisfait la contrainte. Durant la recherche en arborescence, ces algorithmes sont appelés dès qu'une contrainte voit l'ensemble des variables de sa portée assignées. Si une incohérence est détectée ou qu'il n'existe plus de possibilités de branchement (le domaine d'au moins une variable est vide) à un nœud, un *retour arrière* est déclenché, retournant explorer son nœud parent dans l'arbre. Pour un problème de satisfaction, la recherche se termine lorsqu'une solution admissible est trouvée. Toutefois, pour un problème d'optimisation, elle ne peut se terminer que lorsque l'ensemble des nœuds de l'arbre sont explorés. À titre d'exemple, la figure 1.10 illustre un arbre de recherche possible pour le problème de satisfaction constitué de trois variables, $x, y, z \in \mathbb{Z}$, avec $\text{dom}(x) = \{4, 5\}$, $\text{dom}(y) = \{5, 6\}$ et $\text{dom}(z) = \{4, 25\}$, ainsi que deux contraintes, $x > z$ et $x \neq y$.

La taille de l'arbre de recherche complet peut être exponentielle en fonction du nombre de variables et du nombre de valeurs pour chaque variable. Toutefois, il existe diverses stratégies

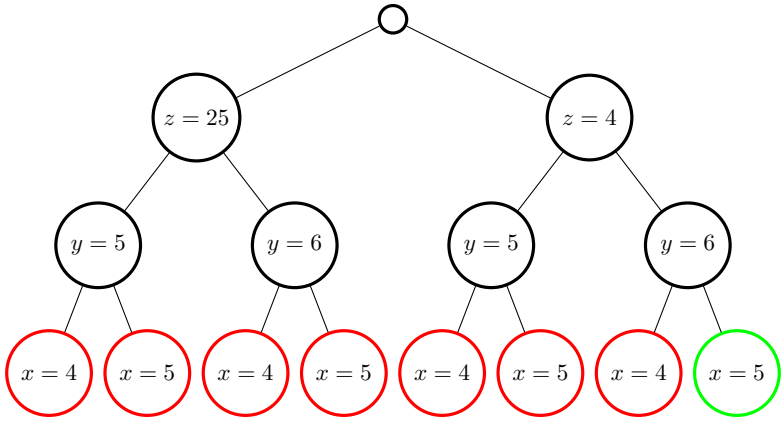


FIGURE 1.10 – Exemple d'arbre de recherche. Chaque nœud est identifié par son branchement. Les nœuds en rouge correspondent aux solutions ayant mené à une incohérence. L'unique solution est $x = 5$, $y = 6$ et $z = 4$ (en vert).

permettant de réduire drastiquement la taille de celui-ci, en omettant volontairement l'exploration de certaines branches spécifiques ou en effectuant de meilleurs choix de recherche. Notamment, une règle de sélection appropriée appelée *heuristique de branchement* pour les choix de variable et de valeur aurait pu nous permettre d'explorer en premier la branche la plus à droite de la figure 1.10.

1.6.3 Filtrage

Le *filtrage* est une technique qui consiste à couper dans l'arbre de recherche des branches ne menant certainement pas à une solution admissible ou menant à une moins bonne solution par rapport à l'objectif d'un problème d'optimisation. Durant la recherche, des *algorithmes de filtrage* spécifiques à chaque contrainte sont appelés à chaque nœud. Étant donné une contrainte, son algorithme de filtrage prend en entrée les domaines des variables de sa portée, puis infère des valeurs pouvant être retirées sans risque des domaines. En fait, dans un solveur de contraintes, chaque contrainte doit être munie de son propre algorithme de filtrage, qu'elle soit arithmétique, logique ou globale. Ces algorithmes sont généralement plus lents qu'un algorithme de vérification, et il convient de trouver un compromis entre temps d'exécution et nombre de valeurs retirées. Les algorithmes sont appelés itérativement : le fait de retirer une valeur d'un domaine peut permettre de filtrer une valeur grâce à une autre contrainte. C'est ce qu'on appelle la *propagation des contraintes* [68, 75].

Reprenons l'exemple de la section 1.6.2 avec son arbre de recherche correspondant illustré à la figure 1.10. Dès la racine de l'arbre, le filtrage de la contrainte « $x > z$ » détecte d'abord que la valeur la plus petite possible pour x doit être strictement plus grande que la valeur la plus petite du domaine de z, 4, ce qui retire la valeur 4 de $\mathrm{dom}(x)$. De façon similaire, le filtrage de cette contrainte détecte aussi que 25 doit être retirée de $\mathrm{dom}(z)$. Ensuite, sachant que $\mathrm{dom}(x) = \{5\}$ et $\mathrm{dom}(y) = \{5, 6\}$, le filtrage de la contrainte « $x \neq y$ » retire la valeur 5

de dom(y). Finalement, l'arbre de recherche d'une seule branche obtenu suivant ce filtrage est illustré à la figure 1.11.

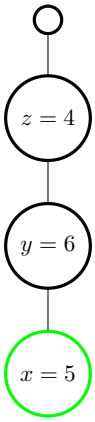


FIGURE 1.11 – Arbre de recherche de la figure 1.10 obtenu suite à l'application du filtrage à la racine.

Chapitre 2

Résolution du TSP en programmation par contraintes

Dans ce chapitre, nous présentons l'état de l'art actuel pour la résolution du TSP en programmation par contraintes. Les notions qui y sont introduites sont directement liées aux contributions de ce mémoire. D'abord, la section 2.1 définit formellement la contrainte WeightedCircuit, la contrainte globale qui encode implicitement le TSP. La section 2.2 présente en détail l'approche et les algorithmes de filtrage utilisés pour cette contrainte. Ensuite, la section 2.3 introduit les heuristiques de branchement couramment utilisées pour la résolution du TSP. Finalement, la section 2.4 présente les travaux de recherche intéressants qui ont été récemment effectués sur le sujet.

2.1 Contrainte WeightedCircuit

En programmation par contraintes, étant donné un graphe $G = (V, E)$ pondéré par une fonction $w\colon E \to \mathbb{Z}$, des variables binaires $\boldsymbol{x} = (x_{e_1}, \ldots, x_{e_{|E|}})^T$ ainsi qu'une variable entière z, la contrainte globale WeightedCircuit$(\boldsymbol{x}, z, G, w)$ est satisfaite si, et seulement si, l'ensemble d'arêtes $C \coloneqq \{e\colon x_e = 1\}$ détermine un cycle hamiltonien de G de poids total $w(C)$ d'au plus la valeur de z [7]. Cette contrainte est sémantiquement équivalente à

$$\textsc{Circuit}(\boldsymbol{x}, G) \wedge \left(\sum_{e \in E} w(e) x_e \leq z \right),$$

où Circuit$(\boldsymbol{x}, G)$ est une contrainte globale exigeant que l'ensemble d'arêtes C détermine un cycle hamiltonien de G. Introduite par Laurière en 1978 [59], cette dernière possède différents algorithmes efficaces de filtrage dans les solveurs de contraintes modernes [6, 14, 37, 53], la plupart généralement combinés à ceux de la contrainte globale AllDifferent [59, 69]. Bien qu'il serait possible de reprendre simplement l'ensemble de ces algorithmes pour WeightedCircuit, le filtrage serait alors restreint à la structure du graphe. En effet, ceux-ci ne permettent

pas d'inférer des valeurs à retirer du domaine des variables $\boldsymbol{x}$ à partir du poids des arêtes du graphe. Ainsi, cela justifie la pertinence d'une contrainte globale WeightedCircuit munie d'algorithmes de filtrage spécifiques utilisant ces poids.

Étant donné la contrainte WeightedCircuit fournie dans le catalogue du solveur utilisé, le TSP peut donc être formulé simplement par

$$\begin{aligned} \min_{\boldsymbol{x},z} \quad & z \\ \text{s. à} \quad & \textsc{WeightedCircuit}(\boldsymbol{x}, z, G, w), \end{aligned}$$

où le domaine initial de z est $[0, M]$, pour M une borne supérieure de $w(C)$ suffisamment grande. Une borne supérieure initiale valide peut notamment être obtenue grâce à l'heuristique LKH [45, 46].

2.2 Algorithmes de filtrage

2.2.1 Contexte

En 1997, Caseau et Laburthe [14] sont les premiers à étudier un algorithme de filtrage utilisant le poids des arêtes pour la contrainte WeightedCircuit. En plus d'introduire une contrainte globale d'élimination de sous-cycles (NoCycle), ils proposent le calcul de bornes inférieures de z via des relaxations de contraintes pour filtrer son domaine ainsi qu'empêcher des assignations d'arêtes au poids trop grand. Cette idée est rapidement reprise et formalisée par Focacci *et al.* sous le nom de *filtrage basé sur les coûts* (*cost-based filtering* en anglais) [32], puis appliquée à la résolution exacte du TSP ainsi que sa variante avec fenêtres de temps (TSPTW) [33-35].

Formellement, soit un problème d'optimisation générique avec $\boldsymbol{x} = (x_1, \ldots, x_n)^T$ de la forme $\min_{\boldsymbol{x}} f(\boldsymbol{x})$ sujet à certaines contraintes non spécifiées. Disons qu'à un certain nœud dans l'arbre de recherche, la meilleure solution trouvée fournit une borne supérieure U sur la fonction objectif et qu'il est possible d'obtenir une borne inférieure L en considérant une relaxation du problème. La technique de filtrage basée sur les coûts consiste à utiliser l'information du sous-problème relaxé courant pour retirer certaines valeurs du domaine des variables. Si $L > U$, une incohérence est détectée. Sinon, si $L[x_i = \mu]$ désigne la valeur optimale du sous-problème sous la contrainte additionnelle $x_i = \mu$ pour une valeur $\mu \in \text{dom}(x_i)$, $i \in \{1, \ldots, n\}$, et que $L[x_i = \mu] > U$, la valeur μ est retirée du domaine de x_i. Cette technique devient intéressante lorsqu'il existe un algorithme efficace pour obtenir $L[x_i = \mu]$. De plus, celle-ci s'inspire des techniques en recherche opérationnelle et programmation linéaire en nombres entiers où un raisonnement similaire appelé en anglais *reduced cost fixing* permet de filtrer certaines valeurs spécifiques (voir, par exemple, Wolsey [83]).

Lorsque la technique de filtrage basée sur les coûts est intégrée à la relaxation lagrangienne dans le contexte de la CP, on parle d'une approche *CP-based Lagrangian relaxation* (CP-LR). Elle est introduite par Sellmann et Fahle en 2001 [78], formalisée par Sellmann en 2004 [77], puis utilisée pour résoudre divers problèmes [10, 13, 30, 63, 79]. En particulier, les algorithmes de filtrage de l'état de l'art pour la contrainte WeightedCircuit sont basés sur celle-ci [7], présentés en détail à la section 2.2.2. Reprenant le contexte de la section 1.3.2–Relaxation lagrangienne, supposons que nous avons le problème (P) formé des familles de contraintes $\mathcal{A}$: $A\boldsymbol{x} \leq \boldsymbol{b}$ et $\mathcal{B}$: $B\boldsymbol{x} \leq \boldsymbol{d}$, que $\mathcal{A}$ forme les contraintes difficiles du problème et qu'il existe un algorithme de filtrage efficace basé sur les coûts pour $\mathcal{B}$, disons le *propagateur* Prop($\mathcal{B}$). En introduisant des multiplicateurs de Lagrange $\boldsymbol{\lambda}$, la relaxation lagrangienne appliquée aux contraintes $\mathcal{A}$ mène au problème (RL($\boldsymbol{\lambda}$)) dont la fonction objectif $g_{\boldsymbol{\lambda}}(\boldsymbol{x})$ dépend de $\boldsymbol{\lambda}$. La valeur optimale de ce problème étant une borne inférieure de l'objectif original, celle-ci est utilisée pour le filtrage Prop($\mathcal{B}$). La maximisation de cette borne parmi $\boldsymbol{\lambda} \geq 0$ en fonction de la structure de $\mathcal{A}$ produit une séquence de multiplicateurs $\boldsymbol{\lambda}^{(0)}, \boldsymbol{\lambda}^{(1)}, \ldots$ qui, pour chaque itération $i \geq 0$, sont associés à un problème unique de minimisation de $g_{\boldsymbol{\lambda}^{(i)}}(\boldsymbol{x})$. L'approche CP-LR consiste alors à utiliser le filtrage basé sur les coûts Prop($\mathcal{B}$) en utilisant la solution optimale de chacun des problèmes rencontrés durant le processus. Sellmann [77] montre que l'application du filtrage avec des multiplicateurs non optimaux peut mener à davantage de valeurs filtrées qu'avec les multiplicateurs maximisant la borne. Cela justifie donc l'utilisation répétée des algorithmes de filtrage durant le processus d'optimisation de la borne plutôt qu'une fois à la toute fin.

Par similarité avec WeightedCircuit, il convient de mentionner les travaux effectués parallèlement sur la contrainte WeightedSpanningTree. Introduite par Dooms et Katriel [25], celle-ci est satisfaite si, et seulement si, le graphe pondéré considéré admet un arbre couvrant de poids total d'au plus la valeur d'une variable entière. En notant qu'une borne inférieure est facilement obtenue en temps polynomial grâce au calcul d'un arbre couvrant minimal, les algorithmes de filtrage développés pour cette contrainte utilisent aussi une approche basée sur les coûts pour déterminer les arêtes de l'arbre [25, 70, 71]. Comme nous le verrons à la section 2.2.2, les algorithmes de filtrage pour WeightedCircuit en sont une extension naturelle.

2.2.2 État de l'art

En 2010, Benchimol *et al.* introduisent des algorithmes de filtrage spécifiques considérant le poids des arêtes pour la contrainte WeightedCircuit [7, 8]. Ceux-ci utilisent une approche CP-LR basée sur la relaxation *1-tree* du TSP (section 1.5.4). Étant donné des multiplicateurs de Lagrange $\boldsymbol{\lambda} = (\lambda_1, \ldots, \lambda_n) \in \mathbb{R}^n$, avec $V \coloneqq \{1, \ldots, n\}$ et $\lambda_1 = 0$, le filtrage consiste d'abord à déterminer $Z_{RL}(\boldsymbol{\lambda})$ grâce au calcul d'un *1-tree* minimal de G, disons T, en utilisant la fonction de poids modifiée $\tilde{w}(i,j) \coloneqq w(i,j) + \lambda_i + \lambda_j, \forall \{i,j\} \in E$. Soit $G' = (V', E')$, avec

$V' \coloneqq V \setminus \{1\}$ et $E' \coloneqq E \setminus \delta(1)$. Posons $T \coloneqq S \cup \{m_1, m_2\}$, où $S \subseteq E'$ est un arbre couvrant minimal de G' et $m_1, m_2 \in \delta(1)$ sont les deux arêtes de poids minimal incidentes au sommet 1. De plus, soit U, une borne supérieure actuelle de la variable z. Par définition de la contrainte, notons qu'il s'agit par le fait même d'une borne supérieure de la solution optimale du TSP.

À partir des valeurs de $Z_{RL}(\boldsymbol{\lambda})$ et U, un filtrage basé sur les coûts du *1-tree* est effectué pour identifier des *arêtes interdites*, arêtes ne pouvant faire partie d'aucune solution du problème, ainsi que des *arêtes obligatoires*, arêtes devant faire nécessairement partie de toute solution. Une arête $e \in E$ interdite aura sa variable associée x_e assignée à la valeur 0 ($\text{dom}(x_e) = \{0\}$). Une arête $e \in E$ obligatoire aura sa variable associée x_e assignée à la valeur 1 ($\text{dom}(x_e) = \{1\}$). Supposons que les arêtes interdites ont été retirées de E et notons l'ensemble des arêtes obligatoires (en anglais, *mandatory*) par M, avec $M \subseteq T$.

Dans les sections suivantes, nous utiliserons le graphe de la figure 2.1 afin d'illustrer les différentes règles de filtrage. Dans cet exemple, il n'y a pas d'arêtes obligatoires ($M = \emptyset$) et les multiplicateurs de Lagrange sont tous nuls. Les poids sont indiqués à côté de chaque arête et les arêtes en gras forment un *1-tree* minimal. Ainsi, $Z_{RL}(\boldsymbol{\lambda}) = 24 - 2(0) = 24$ et on suppose $U = 28$.

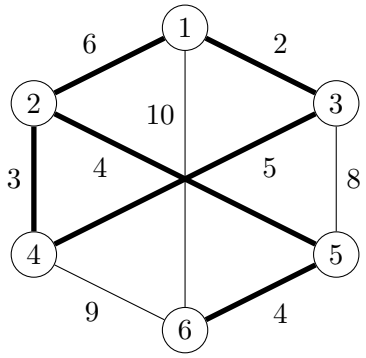


FIGURE 2.1 – Graphe de référence pour illustrer les règles de filtrage de la section 2.2.2.

Borne inférieure

Par définition, la valeur de $Z_{RL}(\boldsymbol{\lambda})$ est une borne inférieure de la solution optimale du TSP. Ainsi, si $Z_{RL}(\boldsymbol{\lambda}) > U$, une incohérence est détectée. Sinon, puisque $Z_{RL}(\boldsymbol{\lambda})$ est aussi une borne inférieure de la variable z, le domaine de z peut être filtré pour que sa borne inférieure soit d'au moins $Z_{RL}(\boldsymbol{\lambda})$. Dans notre exemple, $Z_{RL}(\boldsymbol{\lambda}) \leq U$, donc $\text{dom}(z)$ serait modifié de façon à ce que z ne puisse être que plus grande ou égale à 24.

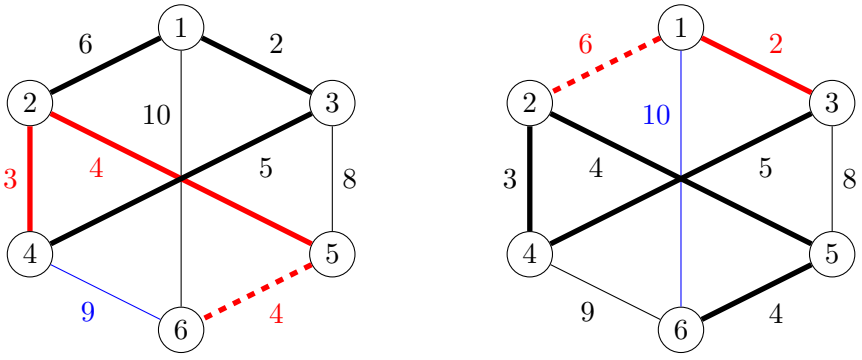
Détection d'arêtes interdites

Étant donné une arête $e \in E \setminus T$ qui n'est pas dans le *1-tree*, l'*arête de support* de e est l'arête $s \in T$ devant être retirée de T afin de conserver un *1-tree* minimal si e est forcée dans celui-ci. Si l'arête e n'est pas incidente au sommet 1, notons $C_e \subseteq S$ l'ensemble des arêtes de l'unique

cycle dans $S \cup \{e\}$. Ce cycle existe et est unique grâce à une propriété élémentaire des arbres (voir section 1.4.3). Par la propriété de cycles d'un arbre couvrant minimal, s est l'arête de plus grand poids dans $C_e \setminus M$. Sinon, l'arête e est incidente au sommet 1 et s est l'arête de plus grand poids parmi $\{m_1, m_2\} \setminus M$. Le *coût réduit* de l'arête e, noté $\bar{c}(e)$, est l'augmentation de $Z_{RL}(\boldsymbol{\lambda})$ lorsque e est forcée dans T. Ainsi, par définition, $\bar{c}(e) = \tilde{w}(e) - \tilde{w}(s)$, avec $\bar{c}(e) \coloneqq \infty$ si s n'existe pas. Le filtrage basé sur les coûts implique donc que $e \in E \setminus T$ doit être identifiée *interdite* si $Z_{RL}(\boldsymbol{\lambda})[x_e = 1] = Z_{RL}(\boldsymbol{\lambda}) + \bar{c}(e) > U$.

La figure 2.2 illustre cette règle de filtrage à partir de notre exemple. À la figure 2.2(a), l'arête d'intérêt est $e = \{4, 6\}$, en bleu, et n'est pas incidente au sommet 1. Si cette dernière est forcée dans le *1-tree*, les arêtes candidates à retirer sont celles du cycle $C_{\{4,6\}}$, identifiées en rouge, avec l'arête de plus grand poids $\{5, 6\}$ en pointillé (ou $\{2, 5\}$, au choix). $s = \{5, 6\}$ est donc l'arête de support de $\{4, 6\}$, et son coût réduit est $\bar{c}(\{4, 6\}) = \tilde{w}(4, 6) - \tilde{w}(5, 6) = 9 - 4 = 5$. Puisque $Z_{RL}(\boldsymbol{\lambda}) + \bar{c}(\{4, 6\}) > U$ $(29 > 28)$, l'arête $\{4, 6\}$ est identifiée *interdite*. À la figure 2.2(b), l'arête d'intérêt est $e = \{1, 6\}$, en bleu, et est incidente au sommet 1. Forcée dans le *1-tree*, les arêtes candidates sont $m_1 = \{1, 2\}$ et $m_2 = \{1, 3\}$, en rouge, avec l'arête de plus grand poids $\{1, 2\}$ en pointillé. $s = \{1, 2\}$ est l'arête de support de $\{1, 6\}$, et son coût réduit est $\bar{c}(\{1, 6\}) = \tilde{w}(1, 6) - \tilde{w}(1, 2) = 10 - 6 = 4$, ce qui n'est pas suffisant pour l'identifier *interdite* $(28 \not> 28)$.

L'application de cette règle de filtrage pour chaque arête $e \in E \setminus T$ demande de calculer le coût réduit de toutes les arêtes. Un algorithme simple pourrait être de déterminer un arbre couvrant minimal avec e obligatoire pour tout $e \in E \setminus T$. Toutefois, la complexité en temps serait alors de $O(|E|^2 \alpha(|E|, |V|))$ en pire cas, non efficace en pratique. Pour une arête $e \notin \delta(1)$, le calcul de son coût réduit consiste simplement à déterminer les arêtes de C_e, tâche qui peut être exécutée grâce à un algorithme de parcours en profondeur dans le graphe induit par S en $O(|V|)$ [19]. Pour toutes les arêtes, la complexité finale en pire cas serait $O(|E||V|)$.



(a) $e \notin \delta(1)$. (b) $e \in \delta(1)$.

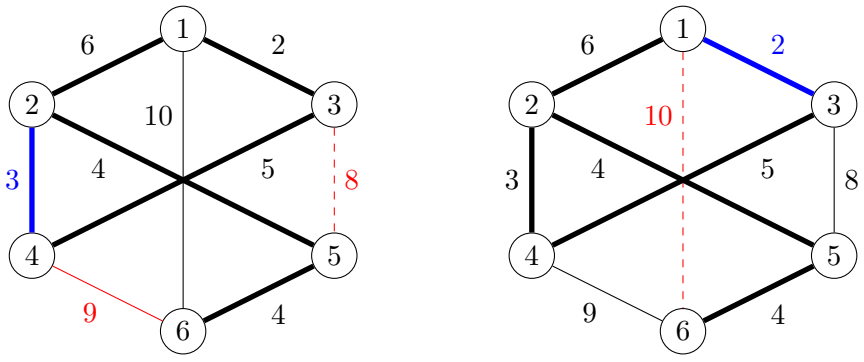
FIGURE 2.2 – Exemples de détection d'arêtes interdites.

Il est possible de faire encore mieux en adaptant les algorithmes utilisés dans le contexte de la contrainte WeightedSpanningTree [25, 70] et de vérification d'un arbre couvrant minimal [24, 55]. En particulier, l'approche de Régin [70] introduit une structure particulière d'arbre, appelée en anglais *connected component tree, ccTree* (littéralement *arbre de composante connexe*), qui permet de transformer le problème de trouver l'arête de poids maximal dans C_e en requête de trouver le plus petit ancêtre commun entre deux nœuds du *ccTree*, problème pour lequel il existe plusieurs algorithmes efficaces [9, 38, 81]. Le tout résulte en une complexité améliorée en pire cas de $O(|V| + |E| + |V| \log |V|)$. À noter que le *ccTree* est aussi connu sous le nom d'*arbre cartésien* dans la littérature [23, 74].

Détection d'arêtes obligatoires

Étant donné une arête $e \in T \setminus M$ dans le *1-tree*, l'*arête de remplacement* de e est l'arête $r \in E \setminus T$ devant être ajoutée à T afin de conserver un *1-tree* minimal si e est retirée de celui-ci. Si l'arête e n'est pas incidente au sommet 1, le retrait de l'arête e divise l'arbre couvrant S en exactement deux composantes connexes (voir section 1.4.3). Notons $R_e \subseteq E' \setminus S$ l'ensemble de coupe formé par le retrait de e de S, c'est-à-dire l'ensemble des arêtes incidentes à un nœud de chaque composante. Par la propriété de coupes d'un arbre couvrant minimal, r est l'arête de plus petit poids dans R_e. Sinon, l'arête e est incidente au sommet 1 et r est l'arête de plus petit poids dans $\delta(1) \setminus \{m_1, m_2\}$. Le *coût de remplacement* de l'arête e, noté $\hat{c}(e)$, est l'augmentation de $Z_{RL}(\boldsymbol{\lambda})$ lorsque e est retirée de T. Ainsi, par définition, $\hat{c}(e) = \tilde{w}(r) - \tilde{w}(e)$, avec $\hat{c}(e) \coloneqq \infty$ si r n'existe pas. Le filtrage basé sur les coûts implique donc que $e \in T \setminus M$ doit être identifiée *obligatoire* si $Z_{RL}(\boldsymbol{\lambda})[x_e = 0] = Z_{RL}(\boldsymbol{\lambda}) + \hat{c}(e) > U$.

La figure 2.3 illustre cette règle de filtrage à partir de notre exemple. À la figure 2.3(a), l'arête d'intérêt est $e = \{2, 4\}$, en bleu, et n'est pas incidente au sommet 1. Si cette dernière est retirée du *1-tree*, les arêtes candidates à ajouter sont celles de l'ensemble de coupe $R_{\{2,4\}}$, identifiées en rouge, avec l'arête de plus petit poids $\{3, 5\}$ en pointillé. $r = \{3, 5\}$ est donc l'arête de remplacement de $\{2, 4\}$, et son coût de remplacement est $\hat{c}(\{2, 4\}) = \tilde{w}(3, 5) - \tilde{w}(2, 4) =$



(a) $e \notin \delta(1)$. (b) $e \in \delta(1)$.

Figure 2.3 – Exemples de détection d'arêtes obligatoires.

$8 - 3 = 5$. Puisque $Z_{RL}(\boldsymbol{\lambda}) + \hat{c}(e) > U$ $(29 > 28)$, l'arête $\{2, 4\}$ est identifiée *obligatoire*. À la figure 2.3(b), l'arête d'intérêt est $e = \{1, 3\}$, en bleu, et est incidente au sommet 1. Retirée du *1-tree*, la seule arête de $\delta(1) \setminus \{m_1, m_2\}$ est $r = \{1, 6\}$, en rouge pointillé, et est l'arête de remplacement de $\{1, 3\}$. Son coût de remplacement est $\hat{c}(\{1, 3\}) = 10 - 2 = 8$, et donc $\{1, 3\}$ est aussi identifiée *obligatoire* $(32 > 28)$.

L'application de cette règle de filtrage pour chaque arête $e \in T \setminus M$ demande de calculer le coût de remplacement de toutes les arêtes. Comme dans le cas des arêtes interdites, un algorithme simple serait de déterminer un arbre couvrant minimal du graphe duquel e est retirée, pour tout $e \in T \setminus M$. La complexité totale serait alors de $O(|V||E|\alpha(|E|, |V|))$ en pire cas, tout aussi peu efficace.

Il est possible de faire mieux en remarquant que, pour $e \notin \delta(1)$, chaque arête candidate $r \in R_e$ forme un unique cycle dans $S \cup \{r\}$ qui contient l'arête e. Ainsi, étant donné l'ensemble des arêtes du graphe en ordre croissant de poids, l'algorithme suivant permet de déterminer le coût de remplacement des arêtes $e \notin \delta(1)$ du *1-tree*. Suivant l'ordre, on considère à tour de rôle chaque arête $r \in E' \setminus S$ qui n'est pas dans l'arbre. L'arête considérée forme un unique cycle dans $S \cup \{r\}$. Chaque arête non obligatoire et non déjà marquée de ce cycle est alors marquée comme ayant l'arête de remplacement r. Le processus est répété tant qu'une arête de $S \setminus M$ n'est pas marquée, c'est-à-dire n'a pas été assignée une arête de remplacement. Au final, la complexité en temps de cet algorithme est de $O(|E||V|)$ en pire cas.

Dans le contexte de la contrainte WeightedSpanningTree, Dooms et Katriel [25] proposent d'utiliser un algorithme de Tarjan pour le calcul des coûts de remplacement d'un arbre couvrant minimal en temps $O(|E|\alpha(|E|, |V|))$ en pire cas [81]. Régin *et al.* [71] proposent plutôt un algorithme plus pratique de même complexité. Repartant de l'algorithme précédemment décrit, ils y ajoutent la contraction des arêtes du graphe lorsqu'elles sont marquées. Les contractions sont effectuées en utilisant la structure pour ensembles disjoints de Tarjan [80], résultant en un algorithme de complexité totale améliorée de $O(|E|\alpha(|E|, |V|))$ en pire cas.

Filtrage additionnel

Par définition de la contrainte WeightedCircuit, seul un cycle hamiltonien de poids au plus z devrait satisfaire la contrainte. Toutefois, en ne considérant que les règles de filtrage précédemment mentionnées, certaines assignations non valides seraient fautivement admises et non filtrées. Ainsi, il convient d'ajouter certains algorithmes pour assurer sa cohérence.

D'abord, il faut forcer les arêtes de l'ensemble $C = \{e \in E \colon x_e = 1\}$ à déterminer un cycle hamiltonien de G. Il pourrait s'agir de toute contrainte additionnelle forçant le degré de chaque sommet à être égal à 2, par exemple

$$\sum_{j \in V} x_{\{i,j\}} = 2, \, \forall i \in V,$$

certaines propriétés spécifiques de filtrage données par l'algorithme de Prim (Benchimol *et al.* [7]), ou tout autre algorithme de filtrage tiré de la contrainte Circuit [14, 37, 59]. Comme le suggère fortement Fages [27], les algorithmes choisis devraient être les plus simples et rapides possible.

Ensuite, il faut aussi assurer le lien entre les arêtes du cycle et la variable z. La façon la plus simple est d'ajouter au modèle la contrainte

$$\sum_{e \in E} w(e) x_e \leq z.$$

D'autres opportunités de filtrage de z sont aussi simultanément possibles en considérant certaines propriétés du problème [27].

2.3 Heuristiques

En 2016, Fages *et al.* [28] présentent l'impact de différentes heuristiques de sélection des arêtes du graphe sur la résolution du TSP. Lorsqu'un branchement dans l'arbre de recherche doit être effectué, l'heuristique détermine l'arête $e \in E \setminus M$ pour laquelle on prend la décision $x_e = 1$. Alors que Benchimol *et al.* [7] choisissent dans leurs expériences de considérer l'arête ayant le plus grand coût de remplacement, d'autres règles de sélection sont explorées. À la lumière de leurs expériences sur le temps de résolution de plusieurs instances du TSP, deux heuristiques sont retenues pour leur efficacité générale : *maxCost* et *minDeltaDeg*. Lors d'un branchement, *maxCost* choisit simplement l'arête de plus grand poids. *minDeltaDeg*, quant à elle, choisit l'arête $\{i, j\}$ telle que $\deg_{E \setminus M}(i) + \deg_{E \setminus M}(j)$ est minimal. Celles-ci sont alignées avec le principe *échouer en premier* (en anglais, *Fail First*) [42] déclarant qu'il peut être mieux de faire de « mauvais » choix tôt dans l'arbre de recherche pour s'éloigner rapidement des régions sans solution. En effet, *maxCost*, en choisissant des arêtes coûteuses, génère rapidement des incohérences au niveau de la relaxation *1-tree*, alors que *minDeltaDeg*, en sélectionnant des arêtes où le graphe comporte peu de choix, génère des incohérences au niveau de la structure du graphe. Les deux heuristiques sont relativement simples et ne nécessitent aucune information provenant de la relaxation elle-même.

Les auteurs introduisent de plus une politique de sélection d'arêtes appelée *Last Conflict*. Basée sur une heuristique générique [60], celle-ci restreint le choix des arêtes à celles situées dans la région du graphe où la dernière incohérence a été détectée. Ainsi, dans le même ordre d'idée que le principe *échouer en premier*, cette règle explore des régions de l'arbre de recherche ayant mené à une incohérence dans l'optique d'en détecter d'autres. Plus précisément, suite au branchement d'une arête menant à une incohérence, un sommet arbitraire de cette arête est gardé en mémoire, puis la prochaine décision doit nécessairement inclure ce sommet. La variante *LCFirst* identifiée dans leurs travaux choisit simplement, pour une arête $\{i, j\}$, le sommet i. Les meilleurs résultats sont obtenus en combinant cette dernière aux deux pré-

cédentes heuristiques. Notons que l'ensemble de leurs algorithmes pour WeightedCircuit ont été implémentés dans le solveur de contraintes libre d'accès *Choco* [52, 67] et son extension *Choco Graph* [27, 29]. Cette implémentation de WeightedCircuit est généralement considérée celle de l'état de l'art.

2.4 Travaux récents

Certains travaux récents sur la résolution du TSP en programmation par contraintes que nous ne considérons pas dans la suite méritent toutefois d'être mentionnés. En 2016, Ducomman *et al.* [26] évaluent la performance du filtrage basé sur les coûts selon la borne inférieure obtenue par trois différentes relaxations : *1-tree* de Held et Karp (section 1.5.4), *problème d'affectation* [7] et *ln-path*, une relaxation basée sur les travaux de Christofides *et al.* en 1981 [16]. Cette dernière se montre particulièrement efficace lorsque des contraintes additionnelles sont présentes, par exemple pour le problème du TSP avec fenêtres de temps. Toutefois, pour le TSP original, la conclusion des auteurs est claire [26] : « *The best approach regarding the number of instances solved and quality of the bound is the Held and Karp's filtering* (La meilleure approche concernant le nombre d'instances résolues et la qualité de la borne est le filtrage de Held et Karp). »

La récente séquence de contributions d'Isoart et Régin a permis d'importantes améliorations à la résolution du TSP en programmation par contraintes, en plus de présenter de nouvelles directions de recherche particulièrement intéressantes. D'abord, ils introduisent en 2019 [48] un algorithme de filtrage appelé *k-cutset* basé sur la recherche d'ensembles de coupe de taille k dans le graphe, pour $k \in \{1, 2, 3\}$. Celui-ci permet d'identifier rapidement des arêtes interdites et obligatoires selon la structure du graphe. Ensuite, en 2020 [49], ils présentent une étude sur l'interaction entre le filtrage de la relaxation *1-tree* et le processus d'optimisation du sous-gradient pour le problème des multiplicateurs de Lagrange. Plus précisément, en observant que la borne calculée est sujette à énormément de variations en fonction du filtrage, ils posent la question suivante : à quel moment dans le processus itératif les algorithmes de filtrage devraient-ils être appelés? Pour y répondre, ils proposent un algorithme adaptatif appelé *Scope-sizing subgradient algorithm* (SSSA) qui permet d'éviter les oscillations de la borne et rend le processus complet plus robuste. Finalement, en 2020 [50], ils proposent une approche de résolution en parallèle. Cette dernière divise le problème d'optimisation en plusieurs sous-problèmes qui seront résolus indépendamment par plusieurs processeurs. Plusieurs difficultés sont discutées et une adaptation au parallélisme de l'heuristique *LCFirst* est présentée.

Chapitre 3

Improved CP-Based Lagrangian Relaxation Approach with an Application to the TSP

Détails de l'article

Titre :	*Improved CP-Based Lagrangian Relaxation Approach with an Application to the TSP.*
Auteurs :	Raphaël Boudreault et Claude-Guy Quimper.
Publication :	*Proccedings of the 30th International Joint Conference on Artificial Intelligence* (IJCAI 2021), pages 1374-1380, 2021 [12].

3.1 Résumé

La relaxation lagrangienne est une technique d'optimisation efficace et largement utilisée en optimisation. Dans le contexte de la programmation par contraintes, il est possible d'en faire usage en la combinant à la technique de filtrage basée sur les coûts. En anglais, cette approche est appelée *CP-based Lagrangian relaxation* (CP-LR). En particulier, les algorithmes de filtrage de l'état de l'art pour la contrainte WeightedCircuit, qui encode le problème du commis voyageur (TSP), sont basés sur celle-ci. Dans cet article, nous proposons une approche CP-LR améliorée modifiant localement les multiplicateurs de Lagrange afin d'augmenter le nombre de valeurs filtrées. Nous introduisons aussi deux nouveaux algorithmes basés sur cette dernière pour filtrer WeightedCircuit. Les résultats expérimentaux obtenus sur des instances du TSP montrent que nos algorithmes permettent d'obtenir un gain significatif sur le temps de résolution et la taille de l'arbre de recherche, lorsque comparés à l'implémentation de l'état de l'art.

3.2 Abstract

CP-based Lagrangian relaxation (CP-LR) is an efficient optimization technique that combines *cost-based filtering* with *Lagrangian relaxation* in a constraint programming context. The state-of-the-art filtering algorithms for the WeightedCircuit constraint that encodes the *traveling salesman problem* (TSP) are based on this approach. In this paper, we propose an improved CP-LR approach that locally modifies the Lagrangian multipliers in order to increase the number of filtered values. We also introduce two new algorithms based on the latter to filter WeightedCircuit. The experimental results on TSP instances show that our algorithms allow significant gains on the resolution time and the size of the search space when compared to the state-of-the-art implementation.

3.3 Introduction

In constraint programming (CP), an efficient way to perform domain filtering for an optimization problem is *cost-based filtering* [10]. Given a minimization problem and an upper bound on the objective value, the cost of a relaxed subproblem is used as lower bound. If assigning a variable to a value increases this lower bound beyond the upper bound, this assignment is inconsistent and the value is filtered out from the variable domain. In linear programming, *Lagrangian relaxation* is a common technique to obtain lower bounds. Difficult constraints are moved into the objective function while adding weights, called *Lagrangian multipliers*, that penalize the objective when these constraints are violated. Maximizing the objective function over the multipliers provides better lower bounds. During this optimization process, one could apply cost-based filtering to each of the resulting subproblems. From this idea, *CP-based Lagrangian relaxation* (CP-LR) was introduced [20] and used to solve many problems [5, 6, 9, 17, 21]. In particular, the state-of-the-art filtering algorithms for the WeightedCircuit constraint that encodes the *traveling salesman problem* (TSP) are based on a CP-LR approach [4].

We propose an improved CP-LR approach that adds a step before applying the cost-based filtering. This step globally increases the number of filtered values by locally modifying the Lagrangian multipliers. It results in a stronger filtering and a greater pruning of the search tree.

Section 3.4 presents the theory of cost-based filtering, Lagrangian relaxation, CP-LR, the TSP, and the WeightedCircuit constraint. Section 3.5 describes our improved CP-LR approach. Section 3.6 presents two new algorithms based on this approach to filter the WeightedCircuit constraint. Experiments on the TSP (Section 3.7) show a significant gain on both the resolution time and the size of the search space when compared to the state-of-the-art implementation of WeightedCircuit [7, 16].

3.4 Background

3.4.1 Cost-Based Filtering

Consider the generic optimization problem $\min_{\boldsymbol{x}} f(\boldsymbol{x})$, where $\boldsymbol{x} = (x_1, \ldots, x_n)^T$ subject to unspecified constraints. CP generally uses a branch-and-bound approach to explore, within a search tree, the possible assignments for $\boldsymbol{x}$. Say, at some point of the search, the best solution found has an objective value U and that we can compute a lower bound L by considering a *relaxed* version of the original problem. *Cost-based filtering* [10] consists in using the information of the current relaxed subproblem to filter values from the variable domains. If $L > U$, infeasibility is raised. Let $L[x_i = \mu]$ be the optimal objective value of the relaxed subproblem with the extra constraint $x_i = \mu$. If $L[x_i = \mu] > U$, then the value μ is filtered out from $\text{dom}(x_i)$. This method requires an efficient algorithm to compute $L[x_i = \mu]$. In the context of integer linear programming, reduced costs can be used to filter specific values with a similar reasoning (*reduced cost fixing*, see e.g. [22]).

3.4.2 Lagrangian Relaxation

Consider the following linear program formed of two constraint families, $\mathcal{A} : A\boldsymbol{x} \leq \boldsymbol{b}$ and $\mathcal{B} : B\boldsymbol{x} \leq \boldsymbol{d}$, where $X \subseteq \mathbb{R}^n$ is an arbitrary set:

$$\begin{aligned} Z = \min_{\boldsymbol{x}} \ & \boldsymbol{c}^T\boldsymbol{x} \\ \text{s.t.} \quad & A\boldsymbol{x} \leq \boldsymbol{b} \\ & B\boldsymbol{x} \leq \boldsymbol{d} \\ & \boldsymbol{x} \in X. \end{aligned} \tag{P}$$

Suppose $\mathcal{A}$ consists of difficult constraints. The *Lagrangian relaxation* technique lets these constraints go in the objective function while keeping a global view on the original problem. Introducing *Lagrangian multipliers* $\lambda_i \geq 0$, the objective function is penalized when the constraints of $\mathcal{A}$ are violated, resulting in the following relaxed problem:

$$\begin{aligned} Z_{LR}(\boldsymbol{\lambda}) = \min_{\boldsymbol{x}} \ & \boldsymbol{c}^T\boldsymbol{x} + \boldsymbol{\lambda}^T(A\boldsymbol{x} - \boldsymbol{b}) \\ \text{s.t.} \quad & B\boldsymbol{x} \leq \boldsymbol{d} \\ & \boldsymbol{x} \in X. \end{aligned} \tag{LR($\boldsymbol{\lambda}$)}$$

Any $\boldsymbol{\lambda} \geq 0$ makes $Z_{LR}(\boldsymbol{\lambda})$ a valid lower bound of Z. In order to get the tightest bound, the Lagrangian multiplier problem is to find $\boldsymbol{\lambda}$ that maximizes $Z_{LR}(\boldsymbol{\lambda})$ subject to $\boldsymbol{\lambda} \geq 0$. Various methods exist in the literature to solve this problem including subgradient descent algorithms [3, 19], where the choice of multipliers is guided by the solution of (LR($\boldsymbol{\lambda}$)) until convergence.

3.4.3 CP-Based Lagrangian Relaxation

Assume we are given the linear program (P) and that an efficient filtering algorithm $\textsc{Prop}(\mathcal{B})$ is known for $\mathcal{B}$. *CP-based Lagrangian relaxation* (CP-LR) [19, 20] consists of optimizing the Lagrangian multipliers for $\mathcal{A}$ while using $\textsc{Prop}(\mathcal{B})$ for each subproblem (LR($\boldsymbol{\lambda}$)) encountered during the gradient descent. Hence, while maximizing the lower bound $Z_{LR}(\boldsymbol{\lambda})$ over $\boldsymbol{\lambda}$, cost-based filtering is applied on the corresponding substructure $\mathcal{B}$.

As shown by Sellmann [19], suboptimal multipliers can be more efficient for filtering than the multipliers that optimize the bound. This justifies why the filtering should be performed during the multipliers optimization process rather than once at the end. While optimizing the bound is the main objective, one could also want to maximize the quantity of filtered values each time $\textsc{Prop}(\mathcal{B})$ is called. Thus, it is a hint that multipliers should play a greater role in the filtering step.

3.4.4 TSP and WeightedCircuit

Let $G = (V, E)$ be an undirected graph with nodes set $V \coloneqq \{1, \dots n\}$, edges set $E \subseteq \{\{x, y\} : x, y \in V, x \neq y\}$ and weight function $w : E \to \mathbb{Z}$. For an edge $\{i, j\} \in E$, we write $w(i, j)$ for its weight. The *(symmetric) traveling salesman problem* (TSP) consists in finding a Hamiltonian cycle in G of minimum weight, i.e. a minimal path visiting all nodes and returning to its starting point.

Let $\delta(i) \coloneqq \{e \in E : i \in e\}$ be the set of edges incident on node i. Introducing binary variables x_e for $e \in E$, the TSP can be modeled as an integer linear program [1]:

$$
\begin{aligned}
Z = \min_{\boldsymbol{x}} \ & \sum_{e \in E} w(e) x_e & & \\
\text{s.t.} \quad & \sum_{e \in \delta(i)} x_e = 2 & \forall i \in V & \qquad (3.1)\\
& \sum_{\substack{i,j \in N,\\ i<j}} x_{\{i,j\}} \leq |N| - 1 & \forall N \subsetneq V, |N| \geq 3 & \qquad (3.2)\\
& x_e \in \{0, 1\} & \forall e \in E. &
\end{aligned}
$$

Equations (3.1) are known as the *degree constraints* and require that each node have exactly two edges incident on it. Inequalities (3.2) are known as the *subtour elimination constraints* and ensure the connectivity of the tour.

The *1-tree relaxation* of Held and Karp [11, 12] results from relaxing the degree constraints (3.1). Let $G' = (V', E')$ be the graph G from which an arbitrary node labeled 1 is removed, i.e. with $V' \coloneqq V \setminus \{1\}$ and $E' \coloneqq E \setminus \delta(1)$. A *1-tree* of G is a spanning tree of G' to which we add two distinct edges incident on node 1. The 1-tree relaxation consists in finding a 1-tree of G of minimal weight. The sum of the edges' weights in the 1-tree is a lower bound of the

TSP. Introducing Lagrangian multipliers $\boldsymbol{\lambda} = (\lambda_1, \ldots, \lambda_n)^T \in \mathbb{R}^n$ with $\lambda_1 = 0$, this lower bound can be improved with a Lagrangian relaxation of the degree constraints (3.1). Let $\deg_A(i) \coloneqq |\{e \in A : i \in e\}|$ be the degree of node $i \in V$ in a set of edges A. We obtain the following relaxed problem:

$$\begin{aligned} Z_{LR}(\boldsymbol{\lambda}) = \min_{\boldsymbol{x}} & \sum_{e \in E} w(e) x_e + \sum_{i \in V} \lambda_i (\deg_T(i) - 2) \\ \text{s.t.} \quad & T \text{ is a 1-tree, where } e \in T \Leftrightarrow (x_e = 1) \\ & x_e \in \{0, 1\} \quad \forall e \in E. \end{aligned}$$

Observe that the objective function can be rewritten as

$$\sum_{\{i,j\} \in E} (w(i,j) + \lambda_i + \lambda_j) x_{\{i,j\}} - 2 \sum_{i \in V} \lambda_i.$$

Thus, $Z_{LR}(\boldsymbol{\lambda})$ can be found by computing a minimum spanning tree of G', denoted S, and adding the two minimal edges incident on node 1, denoted m_1 and m_2, using the weight function $\tilde{w}(i,j) \coloneqq w(i,j) + \lambda_i + \lambda_j$, $\forall \{i,j\} \in E$.

In CP, given the binary variables $\boldsymbol{x} = (x_{e_1}, \ldots, x_{e_{|E|}})^T$, the weight function w, and an integer variable z, the WeightedCircuit$(\boldsymbol{x}, z, G, w)$ constraint [4] is satisfied if the edges $e \in E$ with $x_e = 1$ form a Hamiltonian cycle on the nodes V with total weight at most z. The TSP is thus formulated as a minimization problem on z subject to this constraint. Its filtering algorithms rely on the identification of *mandatory* edges that *must* be part of any solution and *forbidden* edges that *cannot* be part of any solution, through the costs stemming from the 1-tree structure.

Let M be the set of mandatory edges and suppose that forbidden edges were removed from E. Let $\boldsymbol{\lambda}$ be the Lagrangian multipliers, $T \coloneqq S \cup \{m_1, m_2\}$ the minimum 1-tree and U a known upper bound on the value of Z.

For an edge $e \in E \setminus T$, the *support edge* $s \in T$ is the edge that needs to be removed from T if e is forced into the minimum 1-tree. If e is not incident on node 1, let $C_e \subseteq S$ be the edges lying on the unique cycle in $S \cup \{e\}$. The non-mandatory edge $s \in C_e \setminus M$ with maximum weight is the support edge of e. Else, e is incident on node 1 and the support edge s is the one in $\{m_1, m_2\} \setminus M$ with the greatest weight. The *reduced cost* of e, denoted $\bar{c}(e)$, is the increase of the objective value when forcing e in T. Thus, we have $\bar{c}(e) = \tilde{w}(e) - \tilde{w}(s)$ or ∞ if s does not exist. Cost-based filtering implies that $e \in E \setminus T$ is forbidden if $Z_{LR}(\boldsymbol{\lambda})[x_e = 1] = Z_{LR}(\boldsymbol{\lambda}) + \bar{c}(e) > U$.

For an edge $e \in T \setminus M$, the *replacement edge* r is the edge that replaces e in the minimum 1-tree if e is removed. If e is not incident on node 1, removing e from S divides the tree into two components. We define $R_e \subseteq E' \setminus S$ to be the cut-set of G' induced by the removal of edge e from S, i.e. the edges in E' incident on both components in $S \setminus \{e\}$. The replacement

edge r is the one with minimum weight in R_e. Otherwise, e is incident on node 1 and the replacement edge is the one in $\delta(1) \setminus \{m_1, m_2\}$ with the smallest weight. The *replacement cost* of e, denoted $\hat{c}(e)$, is the increase of the objective value when removing e from T. Thus, we have $\hat{c}(e) = \tilde{w}(r) - \tilde{w}(e)$ or ∞ if r does not exist. Cost-based filtering implies that $e \in T$ is mandatory if $Z_{LR}(\boldsymbol{\lambda})[x_e = 0] = Z_{LR}(\boldsymbol{\lambda}) + \hat{c}(e) > U$.

3.5 Improved CP-LR Approach

Consider the original problem (P) and its relaxation (LR($\boldsymbol{\lambda}$)). During the optimization of the Lagrangian multipliers $\boldsymbol{\lambda}$ for $\mathcal{A}$, we look closer at the filtering step of Prop($\mathcal{B}$) on the subproblem (LR($\boldsymbol{\lambda}$)). We know from Sellmann [19] that a worse lower bound could lead to more filtering and that this phenomenon is not restricted to CP-LR. Given a variable and a value for which we would like to perform cost-based filtering, could we *temporarily* change the Lagrangian multipliers so that this value is filtered? In this context, *temporarily* would mean that we find multipliers to filter the value, but we do not consider them in the general multiplier optimization process. We propose the following general framework:

1. Find conditions on the multipliers so that the relaxed solution $\boldsymbol{x}^* = (x_1^*, \ldots, x_n^*)^T$ of (LR($\boldsymbol{\lambda}$)) remains optimal.
2. For each variable x_i and value $\mu \in \text{dom}(x_i) \setminus \{x_i^*\}$, check whether there exist new multipliers $\boldsymbol{\lambda}'$ satisfying the conditions from step 1 such that $Z_{LR}(\boldsymbol{\lambda}')[x_i = \mu] > U$. If so, filter μ from $\text{dom}(x_i)$.

The algorithms executing these steps are left unspecified since the framework depends greatly on the nature of the problem. For any propagator, this approach provides a natural extension: the original cost-based filtering algorithm corresponds to the case where we have $\boldsymbol{\lambda}' = \boldsymbol{\lambda}$ in step 2. Also, note that it can be seen as a generalization of the *dual picking* technique [2].

Consider this integer program with the constraint $3x_2 - x_3 \leq 2$ relaxed into the objective using a Lagrangian $\lambda \geq 0$:

$$\begin{aligned} Z_{LR}(\lambda) = \min_{\boldsymbol{x}} \; & -2x_1 - x_2 - x_3 + \lambda(3x_2 - x_3 - 2) \\ \text{s.t.} \quad & 3x_1 + 2x_2 - x_3 \leq 2 \\ & x_1, x_2, x_3 \in \{0, 1\}. \end{aligned}$$

Suppose we know an efficient cost-based filtering algorithm for this relaxed problem and that an upper bound on Z is -1. Choosing $\lambda = 0$, we obtain $Z_{LR}(0) = -3$, with the solution $\boldsymbol{x}^* = (1, 0, 1)^T$. We compute $Z_{LR}(0)[x_1 = 0] = -2$, $Z_{LR}(0)[x_2 = 1] = -2$ and $Z_{LR}(0)[x_3 = 0] = -1$ and no value is filtered from these costs, even if the bound -3 is optimal. With our improved approach, say we want to look more closely to $x_3 = 0$. Step 1, we note that the solution $\boldsymbol{x}^*$ is optimal for every multiplier $\lambda \geq 0$. Step 2, looking at the subproblem

$Z_{LR}(\lambda')[x_3 = 0]$, we find that $\lambda' = \frac{1}{3}$ leads to $Z_{LR}(\frac{1}{3})[x_3 = 0] = -\frac{2}{3} > -1$. We infer that $x_3 \neq 0$.

3.6 Application to the TSP

Given Lagrangian multipliers $\boldsymbol{\lambda}$, the state-of-the-art filtering algorithms for the WEIGHTED-CIRCUIT constraint compute a minimum 1-tree and the support/replacement edges of every edge in the graph. A brute-force algorithm directly derived from the definitions computes the reduced/replacement costs as well as the sets C_e or R_e for each edge $e \in E$ in overall time $O(|V||E|)$. We propose to apply the improved framework of Section 3.5 supposing this pre-processing step was done.

Let the relaxed solution $\boldsymbol{x}^*$ correspond to the minimum 1-tree T which gives the lower bound $Z_{LR}(\boldsymbol{\lambda})$. By definition of the 1-tree relaxation, this solution is optimal if, and only if, T is a minimum 1-tree of G. Thus, for each edge $e \in E \setminus M$, the search for Lagrangian multipliers $\boldsymbol{\lambda}'$ is subject to the condition that T remains a minimum 1-tree of G.

Given an edge $\{i, j\} \in E \setminus M$, Lemma 1 specifies the conditions on how to find $\boldsymbol{\lambda}'$ that increases the value of $Z_{LR}(\boldsymbol{\lambda})[x_{\{i,j\}} = 0]$ or $Z_{LR}(\boldsymbol{\lambda})[x_{\{i,j\}} = 1]$.

Lemma 1. *Let $\{i, j\} \in E \setminus M$ be an edge of G. Suppose $\{k, l\}$ is the support (resp. replacement) edge of $\{i, j\}$ and $\boldsymbol{\lambda}' \coloneqq (\lambda_1, \ldots, \lambda_s + v, \ldots, \lambda_n)^T$ where $v \in \mathbb{R}$ and $s \in V \setminus \{1\}$. If v is chosen such as under this modification T remains a minimum 1-tree of G and $\{k, l\}$ the support (resp. replacement) edge of $\{i, j\}$, then*

$$Z_{LR}(\boldsymbol{\lambda}')[x_{\{i,j\}} = 1] = Z_{LR}(\boldsymbol{\lambda})[x_{\{i,j\}} = 1] + v \cdot (\deg_T(s) - 2) + v \cdot \Big(\mathbf{1}_{\{i,j\}}(s) - \mathbf{1}_{\{k,l\}}(s)\Big)$$

$$\Big(Z_{LR}(\boldsymbol{\lambda}')[x_{\{i,j\}} = 0] = Z_{LR}(\boldsymbol{\lambda})[x_{\{i,j\}} = 0] + v \cdot (\deg_T(s) - 2) + v \cdot \Big(\mathbf{1}_{\{k,l\}}(s) - \mathbf{1}_{\{i,j\}}(s)\Big)\Big)$$

where $\mathbf{1}_A(x) = 1$ iff $x \in A$ is the indicator function.

Proof. Suppose that $\{i, j\} \in E \setminus T$ and that $\{k, l\}$ is its support edge (the proof is similar for $\{i, j\} \in T$). Considering $\boldsymbol{\lambda}'$, let $\tilde{w}'(e)$ be the new weight of edge e. Since T is still a minimum 1-tree of G and $\{k, l\}$ the support edge of $\{i, j\}$, $Z_{LR}(\boldsymbol{\lambda}')[x_{\{i,j\}} = 1] = Z_{LR}(\boldsymbol{\lambda}') + \tilde{w}'(i, j) - \tilde{w}'(k, l)$ by definition of the reduced cost of $\{i, j\}$. We have

$$\begin{aligned} Z_{LR}(\boldsymbol{\lambda}') &= \sum_{e \in E} w(e)x_e + \sum_{i \in V} \lambda'_i(\deg_T(i) - 2) \\ &= \sum_{e \in E} w(e)x_e + \sum_{i \in V \setminus \{s\}} \lambda_i(\deg_T(i) - 2) + (\lambda_s + v)(\deg_T(s) - 2) \\ &= Z_{LR}(\boldsymbol{\lambda}) + v(\deg_T(s) - 2) \end{aligned}$$

and

$$\begin{aligned}\tilde{w}'(i,j) - \tilde{w}'(k,l) &= w(i,j) + \lambda'_i + \lambda'_j - w(k,l) - \lambda'_k - \lambda'_l \\ &= w(i,j) + \lambda_i + \lambda_j - w(k,l) - \lambda_k - \lambda_l + v \cdot c \\ &= \tilde{w}(i,j) - \tilde{w}(k,l) + v \cdot c\end{aligned}$$

where $c = \mathbf{1}_{\{i,j\}}(s) - \mathbf{1}_{\{k,l\}}(s) \in \{-1, 0, 1\}$. Recalling that

$$Z_{LR}(\boldsymbol{\lambda})[x_{\{i,j\}} = 1] = Z_{LR}(\boldsymbol{\lambda}) + \tilde{w}(i,j) - \tilde{w}(k,l)$$

and putting it all together, the result follows. □

Even though Lemma 1 imposes the condition that the support/replacement edge of $\{i,j\}$ remains unchanged, which is more than what is required by the step 1, it leads in the following to a SIMPLE algorithm (Section 3.6.1) and an α-SETS algorithm (Section 3.6.2).

3.6.1 The SIMPLE Algorithm

Given the edge $\{i,j\} \in E \setminus M$, the SIMPLE algorithm (Algorithm 1) checks whether λ_i and λ_j can be both modified so that $\text{dom}(x_{\{i,j\}})$ is filtered. If $\{i,j\} \in T$, line 1 computes the value Δ corresponding to how much the bound and the replacement cost of $\{i,j\}$ needs to be increased in order to declare the edge mandatory. For λ_i, line 2 calls MAXDECREASE to compute a value $v \leq 0$ such that $\lambda_i + v$ is a modification allowed by the hypotheses of Lemma 1 and that can only increase $Z_{LR}(\boldsymbol{\lambda})[x_{\{i,j\}} = 0]$. Line 3 performs the same process for λ_j. If SIMPLE is applied for $\{i,j\} \in E \setminus T$, we aim at increasing the values of the multipliers λ_i and λ_j so that $\{i,j\}$ is identified as forbidden. Line 4 computes the value Δ of how much the bound and the reduced cost of $\{i,j\}$ need to be increased. For λ_i, line 5 calls MAXINCREASE to compute a value $v \geq 0$ such that $\lambda_i + v$ is a modification allowed by the hypotheses of Lemma 1 and that increases $Z_{LR}(\boldsymbol{\lambda})[x_{\{i,j\}} = 1]$. Line 6 repeats the process for λ_j.

For $\{i,j\} \in T$ and the multiplier λ_i, line D1 of function MAXDECREASE computes a value $\alpha \geq 0$ ensuring T remains a minimum 1-tree under the modification $\lambda_i - \alpha$. Line D2 restricts this value not to exceed β to ensure that the replacement edge of $\{i,j\}$ remains unchanged. For $\{i,j\} \in E \setminus T$, the value α computed on line I1 of function MAXINCREASE ensures that T remains a minimum 1-tree while the value β computed on line I2 restricts the support edge of $\{i,j\}$ to remain unchanged. Both functions run in $O(|V|)$.

In the following, we show that for an edge $\{i,j\} \in T$, the SIMPLE algorithm correctly computes new values for λ_i and λ_j that can only increase $Z_{LR}(\boldsymbol{\lambda})[x_{\{i,j\}} = 0]$. The proof is similar in the case of an edge $\{i,j\} \in E \setminus T$. For the modification of λ_i, we first need the following lemma.

Lemma 2. *Considering* $\boldsymbol{\lambda}' = (\lambda_1, \ldots, \lambda_i - \alpha, \ldots, \lambda_n)^T$ *and the corresponding new reduced costs* $\bar{c}'(e)$ *for* $e \in E \setminus T$*, we have* $\bar{c}'(e) \geq 0 \ \forall e \in E \setminus T$.

Algorithm 1: SIMPLE(i, j)

if $\{i, j\} \in T$ **then**
  $\{k, l\} \leftarrow$ GETREPLACEMENTEDGE(i, j)
1 $\Delta \leftarrow U - (Z_{LR}(\boldsymbol{\lambda}) + \tilde{w}(k, l) - \tilde{w}(i, j))$
  if $i \notin \{1, k, l\} \wedge \deg_T(i) \leq 2$ **then** `// Decrease` λ_i
2 $v \leftarrow (-1) \cdot$ MAXDECREASE(i, j)
    $\Delta \leftarrow \Delta - (v \cdot (\deg_T(i) - 2) - v)$
  if $j \notin \{1, k, l\} \wedge \deg_T(j) \leq 2$ **then** `// Decrease` λ_j
3 $v \leftarrow (-1) \cdot$ MAXDECREASE(j, i)
    $\Delta \leftarrow \Delta - (v \cdot (\deg_T(j) - 2) - v)$
  if $\Delta < 0$ **then** $\mathrm{dom}(x_{\{i,j\}}) \leftarrow \mathrm{dom}(x_{\{i,j\}}) \setminus \{0\}$
else
  $\{k, l\} \leftarrow$ GETSUPPORTEDGE(i, j)
4 $\Delta \leftarrow U - (Z_{LR}(\boldsymbol{\lambda}) + \tilde{w}(i, j) - \tilde{w}(k, l))$
  if $i \notin \{1, k, l\} \wedge \deg_T(i) \geq 2$ **then** `// Increase` λ_i
5 $v \leftarrow$ MAXINCREASE(i, j)
    $\Delta \leftarrow \Delta - (v \cdot (\deg_T(i) - 2) + v)$
  if $j \notin \{1, k, l\} \wedge \deg_T(j) \geq 2$ **then** `// Increase` λ_j
6 $v \leftarrow$ MAXINCREASE(j, i)
    $\Delta \leftarrow \Delta - (v \cdot (\deg_T(j) - 2) + v)$
  if $\Delta < 0$ **then** $\mathrm{dom}(x_{\{i,j\}}) \leftarrow \mathrm{dom}(x_{\{i,j\}}) \setminus \{1\}$

Function MAXDECREASE(i, j)

D1 $\alpha \leftarrow \min\{$GETREDUCEDCOST$(i, k) \colon \{i, k\} \in E \setminus T\} \cup \{\infty\}$
if $j \neq 1$ **then**
  $r \leftarrow$ GETREPLACEMENTEDGE(i, j)
D2 $\beta \leftarrow \min\{\tilde{w}(i, k) - \tilde{w}(r) \colon \{i, k\} \in R_{\{i,j\}}\} \cup \{\infty\}$
  $\alpha \leftarrow \min\{\alpha, \beta\}$
return α

Function MAXINCREASE(i, j)

I1 $\alpha \leftarrow \min\{$GETREPLACEMENTCOST$(i, k) \colon \{i, k\} \in T \setminus M\} \cup \{\infty\}$
if $j \neq 1$ **then**
  $s \leftarrow$ GETSUPPORTEDGE(i, j)
I2 $\beta \leftarrow \min\{\tilde{w}(s) - \tilde{w}(i, k) \colon \{i, k\} \in C_{\{i,j\}} \setminus M\} \cup \{\infty\}$
  $\alpha \leftarrow \min\{\alpha, \beta\}$
return α

Proof. Consider $e \in E \setminus T$ and let $s, s' \in T$ be respectively the support edge of e before and after considering the multipliers $\boldsymbol{\lambda}'$. We have

$$\bar{c}'(e) = \tilde{w}'(e) - \tilde{w}'(s') \geq \tilde{w}'(e) - \tilde{w}(s') \geq \tilde{w}'(e) - \tilde{w}(s)$$

because $\tilde{w}'(x) \leq \tilde{w}(x) \, \forall x \in E$ and by definition of a support edge, $\tilde{w}(s') \leq \tilde{w}(s)$. Now, if $e \in \delta(i)$, we have

$$\tilde{w}'(e) - \tilde{w}(s) = (\tilde{w}(e) - \alpha) - \tilde{w}(s) = \bar{c}(e) - \alpha \geq 0$$

by definition of α in line D1. Else, $e \notin \delta(i)$ and

$$\tilde{w}'(e) - \tilde{w}(s) = \tilde{w}(e) - \tilde{w}(s) = \bar{c}(e) \geq 0.$$

In every case, $\bar{c}'(e) \geq 0$. □

By Lemma 2, the reduced costs remain positive, thus T remains a minimum 1-tree under the modification of λ_i. For every edge a incident on i in $R_{\{i,j\}}$, a is a candidate to be a replacement edge of $\{i, j\}$. If r is the current replacement edge of $\{i, j\}$, the value computed by β guarantees that $\tilde{w}'(a) \geq \tilde{w}'(r)$ with the multipliers $\boldsymbol{\lambda}'$, i.e. that r remains the replacement edge. Since $v \leq 0$ and the conditions $i \notin \{1, k, l\} \wedge \deg_T(i) \leq 2$ hold, the conclusion follows from Lemma 1. The same process is performed with λ_j without recomputing the reduced costs. Indeed, the modification of λ_i cannot decrease the reduced cost of edges incident on node j.

Simple only considers specific cases of Lemma 1 and some opportunities of increasing $Z_{LR}(\boldsymbol{\lambda})[x_{\{i,j\}} = \mu]$ are ignored. This allows us to assure the previous properties are true and keep the algorithm simple and efficient.

The values computed on lines D2 and I2 require the sets C_e and R_e. As an alternative that only uses the reduced costs computed by a faster pre-processing [4], we propose to replace the sets on lines D2 and I2 by

$$\{\tilde{w}(i,k) - \tilde{w}(r) \colon \{i,k\} \in E \setminus T, \tilde{w}(i,k) \geq \tilde{w}(r)\} \cup \{\infty\} \text{ and}$$

$$\{\tilde{w}(s) - \tilde{w}(i,k) \colon \{i,k\} \in T \setminus M, \tilde{w}(s) \geq \tilde{w}(i,k)\} \cup \{\infty\}.$$

In the following, we refer to the choice of the original sets as the *complete* policy and of these latter as the *relaxed* policy. In both cases, the overall time complexity is in $O(|V|)$.

3.6.2 The α-Sets Algorithm

Following step 1, we derive the conditions on the multipliers in order for the 1-tree $T = S \cup \{m_1, m_2\}$ to remain minimal.

$$\lambda'_a + \lambda'_b - \lambda'_c - \lambda'_d \leq w(c,d) - w(a,b) \qquad \forall \{a,b\} \in S \setminus M, \forall \{c,d\} \in R_{\{a,b\}} \quad \text{(A)}$$

$$\lambda'_b - \lambda'_d \leq w(1,d) - w(1,b) \qquad \forall \{1,b\} \in \{m_1, m_2\} \setminus M, \forall \{1,d\} \in \delta(1) \setminus \{m_1, m_2\}. \quad \text{(B)}$$

The constraints (A) come from the *cut property* of minimum spanning trees stating that the cost of an edge in the tree $\{a, b\}$ should not be greater than the cost of any edge $\{c, d\}$ in its cut-set $R_{\{a,b\}}$. The constraints (B) ensure that m_1 and m_2 are the two minimal edges incident on node 1. For an edge $\{i, j\} \in E \setminus M$, Lemma 1 also requires that its support/replacement edge $\{k, l\} \in E \setminus M$ remains unchanged. By definition, this can be formulated as

$$\lambda'_k + \lambda'_l - \lambda'_a - \lambda'_b \leq w(a, b) - w(k, l) \qquad \forall \{a, b\} \in R_{\{i,j\}}, \text{ if } \{i, j\} \in S; \tag{C}$$

$$\lambda'_l - \lambda'_b \leq w(1, b) - w(1, l) \qquad \forall \{1, b\} \in \delta(1) \setminus \{m_1, m_2\}, \text{ if } \{1, j\} \in \{m_1, m_2\}; \tag{D}$$

$$\lambda'_a + \lambda'_b - \lambda'_k - \lambda'_l \leq w(k, l) - w(a, b) \qquad \forall \{a, b\} \in C_{\{i,j\}}, \text{ if } \{i, j\} \in E' \setminus S; \tag{E}$$

$$\lambda'_b - \lambda'_l \leq w(1, l) - w(1, b) \qquad \forall \{1, b\} \in \{m_1, m_2\}, \text{ if } \{1, j\} \in \delta(1) \setminus \{m_1, m_2\}. \tag{F}$$

All of these constraints are linear and could be used to derive a linear program that maximizes $Z_{LR}(\boldsymbol{\lambda}')[x_{\{i,j\}} = \mu]$. However, given there are $O(|V|^4)$ constraints, even the best linear solvers take too much time for the filtering to pay off. Therefore, we solve an easier problem where constraints are gradually added and multipliers are locally modified.

Given an edge $\{i, j\} \in E \setminus M$ and Lagrangian multipliers $\boldsymbol{\lambda}$, the α-Sets algorithm tries to find an *α-set* $A \subseteq V$ in the graph: a set of nodes that will have their corresponding multiplier simultaneously modified and that will lead to an increased $Z_{LR}(\boldsymbol{\lambda})[x_{\{i,j\}} = \mu]$ value. Each node $u \in V$ is labeled with a value $\sigma_u \in \{-1, 0, +1\}$ initially set to 0, corresponding to whether the multiplier λ_u will decrease (-1), increase $(+1)$, remain unchanged (0). For a variable $\alpha \geq 0$ and each node u, we look for multipliers $\lambda'_u = \lambda_u + \sigma_u \cdot \alpha$. Starting from an initial node, the algorithm computes incrementally a set A of nodes and a set Ω of constraints taken from (A) to (F). During any step in the process, the substitution of λ'_u by $\lambda_u + \sigma_u \cdot \alpha \ \forall u \in V$ in each constraint $\omega \in \Omega$ leads to a system of linear inequalities that can be written in the form $c_\omega \cdot \alpha \leq m_\omega$, where the coefficient $c_\omega \in \{-2, -1, 0, 1, 2\}$ and $m_\omega \geq 0$ are two known constants. Maximizing α under these constraints, a value $\alpha^* \geq 0$ is found. If $\alpha^* = 0$, the algorithm looks for the restraining constraint and appends one of its related nodes to A in order to loosen the inequality. Doing so, new constraints must be taken into account and added to Ω. This process is repeated until $\alpha^* > 0$, leading to a valid α-set A and new multipliers $\boldsymbol{\lambda}'$. The increased value $Z_{LR}(\boldsymbol{\lambda}')[x_{\{i,j\}} = \mu]$ directly follows from the sum of all the changes according to Lemma 1, without having to recompute a minimum 1-tree. If it is still insufficient to filter $\text{dom}(x_{\{i,j\}})$, the procedure can be re-executed with the multipliers $\boldsymbol{\lambda} \coloneqq \boldsymbol{\lambda}'$ previously found and looks for another α-set.

This procedure searches for an α-set A where $\{k,l\} \in E \setminus M$ is the support/replacement edge of $\{i,j\}$.

1. Initialize a set of constraints Ω with the constraints from (C) to (F), depending on the nature of $\{i,j\}$.
2. **Choose** an initial node $u \in \{i,j,k,l\} \setminus \{1\}$ and $\sigma_u \in \{-1,+1\}$. Following Lemma 1, this choice must satisfy
   a) $\{i,j\} \in E \setminus T \Rightarrow \sigma_u \cdot \Big(\deg_T(u) - 2 + \mathbf{1}_{\{i,j\}}(u) - \mathbf{1}_{\{k,l\}}(u)\Big) > 0$;
   b) $\{i,j\} \in T \Rightarrow \sigma_u \cdot \Big(\deg_T(u) - 2 + \mathbf{1}_{\{k,l\}}(u) - \mathbf{1}_{\{i,j\}}(u)\Big) > 0$.

   If none of these 8 combinations works, no set A exists and the algorithm halts without filtering $\mathrm{dom}(x_{\{i,j\}})$.
3. Append node u to the set A.
4. Add to Ω all the constraints not already considered of type (A)-(B) where λ'_u has coefficient σ_u.
5. For each constraint $\omega \in \Omega$, compute the values c_ω and m_ω. Since ω is of the form
$$\lambda'_c + \lambda'_d - \lambda'_a - \lambda'_b \leq w(a,b) - w(c,d),$$
   we have $c_\omega = \sigma_a + \sigma_b - \sigma_c - \sigma_d$ and $m_\omega = \tilde{w}(a,b) - \tilde{w}(c,d)$. The maximal value $\alpha \geq 0$ can take, subject to the constraints in Ω, is given by
$$\alpha^* \leftarrow \min\left\{\frac{m_\omega}{c_\omega} : \omega \in \Omega \wedge c_\omega > 0\right\}.$$
6. If $\alpha^* > 0$, return the value α^* and the set A is found.
7. If $\alpha^* = 0$, find the constraint $\bar{\omega} \in \Omega$ that prevents obtaining a value $\alpha > 0$. This constraint is of the form $c_{\bar{\omega}} \cdot \alpha \leq \tilde{w}(a,b) - \tilde{w}(c,d)$, where $a,b,c,d \in V$.
8. **Choose** a node $u' \in \{a,b,c,d\} \setminus \{1\}$ and $\sigma_{u'} \in \{-1,+1\}$ such that
   a) $u' \notin P \cup (\{a,b\} \cap \{c,d\})$;
   b) $(u' \in \{a,b\} \wedge \deg_T(u') \geq 2) \Rightarrow \sigma_{u'} = +1$;
   c) $(u' \in \{c,d\} \wedge \deg_T(u') \leq 2) \Rightarrow \sigma_{u'} = -1$;
   d) $u' \in \{i,j,k,l\} \Rightarrow$ conditions of step 2.

   If none of these 8 combinations works, backtrack and reconsider the last **choice**. If no possibility remains, no set A exists and the algorithm halts without filtering $\mathrm{dom}(x_{\{i,j\}})$. Else, set $u \coloneqq u'$ and go to step 3.

Since the procedure can be repeated as long as an α-set A is found and $\mathrm{dom}(x_{\{i,j\}})$ is not filtered, a maximum number of iterations can be added. Also, to avoid considering too many constraints at the same time, a maximum cardinality C_m on A can be imposed. Our

implementation follows an iterative-deepening search that gradually increases C_m. For an α-set of C nodes, step 5 deals with $C \cdot O(|V||E|)$ constraints. As 4 nodes are possible for each choice, it leads to a worst-case time complexity in $O(C_m|V||E|4^{C_m})$.

This algorithm can be combined with the SIMPLE algorithm and the *complete* policy: we call the latter first to obtain new multipliers $\boldsymbol{\lambda}'$, and if it was not enough to filter the considered edge, we use them as the initial multipliers for α-SETS. We refer to this process as the HYBRID algorithm.

3.6.3 Example

On the graph of Figure 3.1(a) with $Z_{LR}(\boldsymbol{\lambda}) = 19$ and $U = 23$, $\{a, b\}$ is the support edge of $\{b, c\}$ and $Z_{LR}(\boldsymbol{\lambda})[x_{\{b,c\}} = 1] = Z_{LR}(\boldsymbol{\lambda}) + \tilde{w}(b, c) - \tilde{w}(a, b) = 20$. We apply HYBRID on edge $\{b, c\}$. First, SIMPLE finds that λ_c can be increased by 1, obtained from the minimum between $\alpha = 1$ (line I1) and $\beta = 2$ (line I2). This leads to the multipliers $\boldsymbol{\lambda}'$ on Figure 3.1(b), but $Z_{LR}(\boldsymbol{\lambda}')[x_{\{b,c\}} = 1] = 22 \leq U$. Choosing again to increase node c by $\alpha \geq 0$, α-SETS finds the constraint $\alpha \leq \tilde{w}'(d, e) - \tilde{w}'(c, d) = 0$. A valid choice is to simultaneously increase node e by α, leading to the new constraint $\alpha \leq \tilde{w}'(d, e) - \tilde{w}'(c, e) = 1$. Since $\alpha^* > 0$, the multipliers $\boldsymbol{\lambda}'$ are updated (Figure 3.1(c)) and $Z_{LR}(\boldsymbol{\lambda}')[x_{\{b,c\}} = 1] = 24 > U$. The edge $\{b, c\}$ is thus forbidden.

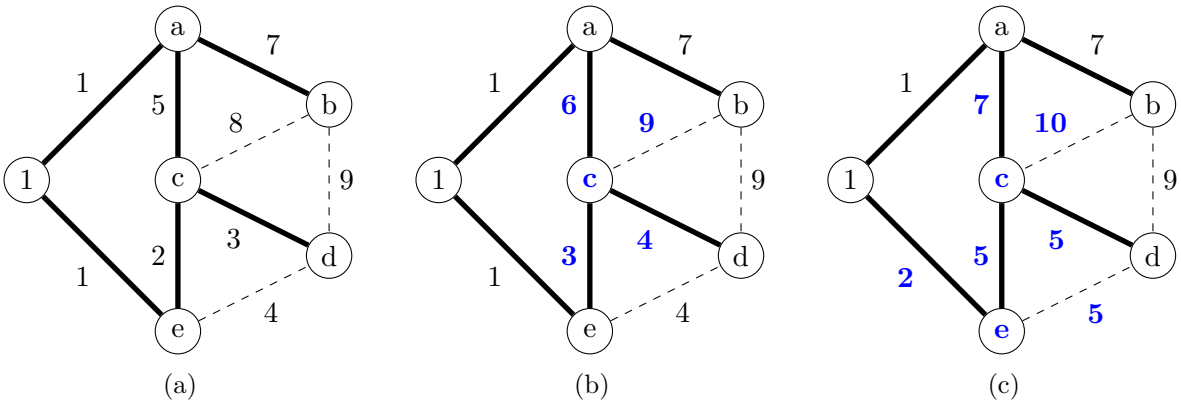


Figure 3.1 – Example of Section 3.6.3. Bold edges form a minimum 1-tree. Current penalized weights are indicated next to each edge.

3.7 Experiments

The algorithms were implemented[1] in Java 14 using the solver Choco 4.0.6 [16] and its extension Choco Graph 4.2.3 [7]. The experiments were performed on a CentOS Linux 7 machine using an Intel Xeon Silver 4110 CPU at 2.10 GHz and 32 GB of RAM. The TSP was modeled using the WEIGHTEDCIRCUIT constraint already implemented in Choco Graph using

1. The code is available at `http://www2.ift.ulaval.ca/~quimper/publications.php`.

the state-of-the-art algorithms [4]. It uses a subgradient descent algorithm to optimize the Lagrangian multipliers. We kept the default parameters, but we did not restart the algorithm when the lower bound of the 1-tree was increased. As an initial upper bound on the objective variable, we gave the bound provided by the LKH-2.0.9 heuristic [13]. The search strategy was fixed to *maxCost* with the *LCFirst* policy [8]. We chose the symmetric TSP instances from the TSPLIB library [18] between 96 and 500 nodes that could be solved by Choco under 8 hours and with at least 100 search nodes.

We compare the SIMPLE and HYBRID algorithms against the one from Choco. Our algorithms were only called at the very last iteration of the subgradient process. Furthermore, for HYBRID, since α-SETS is slowed down by the number of constraints, it was only called when $|E| \leq 2|V|$ with a limit of 2 on the cardinality of the α-sets and a maximum of 10 iterations when trying to reach the fixed point.

Table 3.1 shows considerable gains on the solving time and the size of the search space for almost every instance. SIMPLE with the *relaxed* policy gain an average reduction of 15% on the number of nodes and 18% on the solving time. The solving time is reduced for 86% of the instances. Conclusions are similar for the *complete* policy. However, the gains between the two policies are in general not significant enough to prefer the latter. The improvement is more prominent with the HYBRID algorithm. On average, the number of nodes and the solving time are respectively reduced by 36% and 30% compared to Choco. The solving time is reduced for 93% of the instances. It is in general the best method among all the ones we tested (79% of the instances). In rare cases (a280, d198), the number of nodes increases despite the potential amount of additional filtering. This pathological situation is a well-known fact in the context of CP-LR [15, 19] and should be more investigated.

Instance	Choco		Simple *Relaxed*		Simple *Complete*		Hybrid	
	N	**T**	**N**	**T**	**N**	**T**	**N**	**T**
gr96	520	3.2	436	2.7	365	2.4	**323**	**2.4**
kroA100	1488	8.3	1066	5.9	1211	6.8	**873**	**5.5**
kroB100	2312	12.3	1838	8.9	2042	11.0	**1462**	**7.9**
kroC100	360	2.3	247	1.5	263	1.7	**210**	**1.5**
kroD100	128	1.1	132	1.0	127	1.0	**114**	**0.9**
kroE100	1915	9.8	1578	8.4	1532	8.8	**1113**	**6.9**
gr120	324	3.0	199	1.9	254	2.5	**145**	**1.6**
pr124	224	2.5	195	2.1	169	2.1	**157**	**1.9**
ch130	908	8.7	768	7.5	673	7.2	**518**	**5.4**
pr136	72574	561.4	78781	558.2	72224	**503.0**	**66481**	519.3
gr137	923	9.9	**716**	**8.1**	756	9.4	746	9.8
pr144	149	2.8	65	**1.3**	66	1.4	**62**	1.4
ch150	934	10.9	681	7.8	710	9.6	**498**	**7.3**
kroA150	5652	65.0	2461	**26.8**	2910	35.4	**2329**	27.4
kroB150	112078	1218.6	109715	1115.5	85765	925.6	**67134**	**671.2**
pr152	335	6.2	641	9.4	446	8.2	**277**	**5.7**
si175	38068	486.0	35755	436.1	38775	520.6	**29915**	**387.3**
rat195	18785	361.1	14905	290.3	13113	266.0	**10281**	**215.9**
d198	**5702**	**101.4**	7322	112.2	6695	109.0	6765	118.1
kroA200	1176978	15766.3	863416	12654.3	884545	12399.6	**687347**	**9618.6**
kroB200	46484	739.2	33405	555.6	32779	514.3	**27392**	**430.0**
gr202	1568	20.2	991	11.9	853	11.8	**644**	**10.0**
tsp225	125761	2426.2	72357	1416.6	74639	1546.7	**51766**	**1174.0**
gr229	458131	7367.2	303679	4568.1	268029	4559.1	**215354**	**3272.8**
pr264	123	8.8	130	8.9	114	10.0	**88**	**8.6**
a280	**1605**	**30.7**	2429	40.2	1832	33.7	2005	37.3
lin318	3794	115.1	2296	72.8	2409	81.7	**1128**	**39.1**
gr431	415036	20485.0	309152	17077.4	278439	15917.5	**241454**	**15074.7**
Mean	89031	1779.8	65906	1393.3	63276	1339.5	**50592**	**1130.8**

Table 3.1 – Num. of search nodes (**N**) and solving time in seconds (**T**).

3.8 Conclusion

We introduced an improved approach for CP-LR problems that adds a new step in the filtering process to increase the number of filtered values. We applied it on the WeightedCircuit constraint filtering by introducing two new algorithms: Simple and α-Sets. Experimental results on TSP instances show they greatly improve the solving time compared to the state-of-the-art implementation of Choco. The recent work of Isoart and Régin on the TSP (*k-cutset* [14], *SSSA* [15]) is *a priori* completely compatible with our work and should be further investigated. Future research also includes testing the approach in other CP-LR contexts.

Références

[1] D. L. Applegate, R. E. Bixby, V. Chvátal et W. J. Cook, *The Traveling Salesman Problem: A Computational Study*. Princeton University Press, 2006, 606 p.

[2] O. S. Bajgiran, A. A. Cire et L.-M. Rousseau, « A first look at picking dual variables for maximizing reduced cost fixing », in *Integration of AI and OR Techniques in Constraint Programming*, D. Salvagnin et M. Lombardi, éd., sér. Lecture Notes in Computer Science, Cham : Springer International Publishing, 2017, p. 221-228.

[3] J. E. Beasley, « Lagrangian relaxation », in *Modern Heuristic Techniques for Combinatorial Problems*, USA : John Wiley & Sons, Inc., 1993, p. 243-303.

[4] P. Benchimol, W.-J. van Hoeve, J.-C. Régin, L.-M. Rousseau et M. Rueher, « Improved filtering for weighted circuit constraints », *Constraints*, t. 17, n° 3, p. 205-233, 2012.

[5] D. Bergman, A. A. Cire et W.-J. van Hoeve, « Improved constraint propagation via Lagrangian decomposition », in *Principles and Practice of Constraint Programming*, G. Pesant, éd., sér. Lecture Notes in Computer Science, Cham : Springer International Publishing, 2015, p. 30-38.

[6] H. Cambazard et J.-G. Fages, « New filtering for AtMostNValue and its weighted variant: a Lagrangian approach », *Constraints*, t. 20, n° 3, p. 362-380, 2015.

[7] J.-G. Fages, « Exploitation de structures de graphe en programmation par contraintes », Thèse, École des Mines de Nantes, 2014.

[8] J.-G. Fages, X. Lorca et L.-M. Rousseau, « The salesman and the tree: the importance of search in CP », *Constraints*, t. 21, n° 2, p. 145-162, 2016.

[9] T. Fahle et M. Sellmann, « Cost based filtering for the constrained knapsack problem », *Annals of Operations Research*, t. 115, n° 1, p. 73-93, 2002.

[10] F. Focacci, A. Lodi et M. Milano, « Cost-based domain filtering », in *Principles and Practice of Constraint Programming*, J. Jaffar, éd., sér. Lecture Notes in Computer Science, Berlin, Heidelberg : Springer, 1999, p. 189-203.

[11] M. Held et R. M. Karp, « The traveling-salesman problem and minimum spanning trees », *Operations Research*, t. 18, n° 6, p. 1138-1162, 1970.

[12] M. Held et R. M. Karp, « The traveling-salesman problem and minimum spanning trees: part II », *Mathematical Programming*, t. 1, n° 1, p. 6-25, 1971.

[13] K. Helsgaun, « An effective implementation of the Lin–Kernighan traveling salesman heuristic », *European Journal of Operational Research*, t. 126, n° 1, p. 106-130, 2000.

[14] N. Isoart et J.-C. Régin, « Integration of structural constraints into TSP models », in *Principles and Practice of Constraint Programming*, T. Schiex et S. de Givry, éd., sér. Lecture Notes in Computer Science, Springer International Publishing, 2019, p. 284-299.

[15] N. Isoart et J.-C. Régin, « Adaptive CP-based Lagrangian relaxation for TSP solving », in *Integration of Constraint Programming, Artificial Intelligence, and Operations Research*, E. Hebrard et N. Musliu, éd., sér. Lecture Notes in Computer Science, Cham : Springer International Publishing, 2020, p. 300-316.

[16] N. Jussien, G. Rochart et X. Lorca, « Choco: an open source Java constraint programming library », in *CPAIOR'08 Workshop on Open-Source Software for Integer and Contraint Programming (OSSICP'08)*, Paris, France, 2008, p. 1-10.

[17] J. Menana et S. Demassey, « Sequencing and counting with the multicost-regular constraint », in *Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems*, W.-J. van Hoeve et J. N. Hooker, éd., sér. Lecture Notes in Computer Science, Berlin, Heidelberg : Springer, 2009, p. 178-192.

[18] G. Reinelt, « TSPLIB—A traveling salesman problem library », *ORSA Journal on Computing*, t. 3, n° 4, p. 376-384, 1991.

[19] M. Sellmann, « Theoretical foundations of CP-based Lagrangian relaxation », in *Principles and Practice of Constraint Programming – CP 2004*, M. Wallace, éd., sér. Lecture Notes in Computer Science, Berlin, Heidelberg : Springer, 2004, p. 634-647.

[20] M. Sellmann et T. Fahle, « CP-based Lagrangian relaxation for a multimedia application », in *3rd International Workshop on Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems (CP-AI-OR)*, 2001, p. 1-14.

[21] M. Sellmann et T. Fahle, « Constraint programming based Lagrangian relaxation for the automatic recording problem », *Annals of Operations Research*, t. 118, n° 1, p. 17-33, 2003.

[22] L. Wolsey, « Branch and bound », in *Integer Programming*, John Wiley & Sons, Ltd, 2020, p. 113-138.

Conclusion

Dans ce mémoire, nous avons d'abord présenté au chapitre 1 plusieurs notions théoriques entourant l'optimisation combinatoire, la relaxation lagrangienne, la théorie des graphes et la programmation par contraintes. Nous avons aussi présenté en détail le problème du commis voyageur ainsi que sa relaxation *1-tree*. Ensuite, au chapitre 2, nous avons exposé la contrainte WeightedCircuit et les algorithmes de filtrage de l'état de l'art pour celle-ci. Nous avons vu que ces algorithmes utilisent une approche CP-LR basée sur la relaxation *1-tree*.

Au chapitre 3, nous avons introduit une approche CP-LR améliorée qui ajoute une nouvelle étape dans le processus itératif de filtrage modifiant localement les multiplicateurs de Lagrange pour augmenter le nombre de valeurs filtrées. Cette contribution se veut surtout d'apporter un tout nouveau regard dans la communauté de programmation par contraintes sur l'importance du choix des multiplicateurs, non pas uniquement pour l'optimisation de la borne, mais aussi pour assurer le filtrage de valeurs spécifiques. Grâce à deux algorithmes originaux, Simple et α-Sets, nous avons appliqué cette approche sur le filtrage de la contrainte WeightedCircuit encodant le TSP. Nos résultats expérimentaux sur différentes instances du TSP ont montré que l'utilisation de nos algorithmes permet de grandement améliorer le temps de résolution et le nombre de nœuds dans l'arbre de recherche comparativement à l'implémentation de l'état de l'art de la contrainte.

Bien que les performances des solveurs de contraintes soient encore bien en deçà de celles du solveur *Concorde*, l'état de l'art pour la résolution du TSP, nos contributions mettent en lumière plusieurs directions de recherche intéressantes. D'abord, nos nouveaux algorithmes devraient être testés sur des variantes du TSP munies d'une ou plusieurs contraintes additionnelles. Dans ces cas, l'utilisation de *Concorde* n'est plus envisageable, ce qui rend pertinente l'utilisation de la programmation par contraintes ainsi que de nos algorithmes. Ensuite, afin d'améliorer davantage la performance du filtrage de WeightedCircuit, nos travaux auraient avantage à être intégrés avec ceux d'Isoart et Régin (*k-cutset* [48], SSSA [49]) qui sont, *a priori*, entièrement compatibles avec notre approche. Finalement, en dehors du contexte du TSP, notre approche CP-LR générique devrait être appliquée à d'autres contraintes pour lesquelles leurs algorithmes de filtrage sont basés sur les coûts d'une relaxation lagrangienne, par exemple MultiCost-Regular [63] et AtMostWValue [13].

Bibliographie

[1] R. K. Ahuja, T. L. Magnanti et J. B. Orlin, « Lagrangian relaxation and network optimization, » in *Network Flows: Theory, Algorithms, and Applications*, USA : Prentice-Hall, Inc., 1993, p. 598-648.

[2] D. Applegate et al. « Star tours: HYG109399. » (2019), adresse : http://www.math.uwaterloo.ca/tsp/star/hyg.html (visité le 17/06/2021).

[3] D. L. Applegate, R. E. Bixby, V. Chvátal et W. J. Cook. « Concorde TSP solver. » (2003), adresse : https://www.math.uwaterloo.ca/tsp/concorde/index.html (visité le 17/06/2021).

[4] D. L. Applegate, R. E. Bixby, V. Chvátal et W. J. Cook, *The Traveling Salesman Problem: A Computational Study*. Princeton University Press, 2006, 606 p.

[5] J. E. Beasley, « Lagrangian relaxation, » in *Modern Heuristic Techniques for Combinatorial Problems*, USA : John Wiley & Sons, Inc., 1993, p. 243-303.

[6] N. Beldiceanu et E. Contejean, « Introducing global constraints in CHIP, » *Mathematical and Computer Modelling*, t. 20, n° 12, p. 97-123, 1994.

[7] P. Benchimol, W.-J. van Hoeve, J.-C. Régin, L.-M. Rousseau et M. Rueher, « Improved filtering for weighted circuit constraints, » *Constraints*, t. 17, n° 3, p. 205-233, 2012.

[8] P. Benchimol, J.-C. Régin, L.-M. Rousseau, M. Rueher et W.-J. van Hoeve, « Improving the Held and Karp approach with constraint programming, » in *Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems*, A. Lodi, M. Milano et P. Toth, éd., sér. Lecture Notes in Computer Science, Berlin, Heidelberg : Springer, 2010, p. 40-44.

[9] M. A. Bender, M. Farach-Colton, G. Pemmasani, S. Skiena et P. Sumazin, « Lowest common ancestors in trees and directed acyclic graphs, » *Journal of Algorithms*, t. 57, n° 2, p. 75-94, 2005.

[10] D. Bergman, A. A. Cire et W.-J. van Hoeve, « Improved constraint propagation via Lagrangian decomposition, » in *Principles and Practice of Constraint Programming*, G. Pesant, éd., sér. Lecture Notes in Computer Science, Cham : Springer International Publishing, 2015, p. 30-38.

[11] K.-H. Borgwardt, « The average number of pivot steps required by the simplex-method is polynomial, » *Zeitschrift für Operations Research*, t. 26, n° 1, p. 157-177, 1982.

[12] R. Boudreault et C.-G. Quimper, « Improved CP-based Lagrangian relaxation approach with an application to the TSP, » in *Proceedings of the Thirtieth International Joint Conference on Artificial Intelligence, IJCAI-21*, Z.-H. Zhou, éd., International Joint Conferences on Artificial Intelligence Organization, 2021, p. 1374-1380.

[13] H. Cambazard et J.-G. Fages, « New filtering for AtMostNValue and its weighted variant: a Lagrangian approach, » *Constraints*, t. 20, n° 3, p. 362-380, 2015.

[14] Y. Caseau et F. Laburthe, « Solving small TSPs with constraints, » in *Proceedings of the 14th International Conference on Logic Programming*, MIT Press, 1997, p. 316-330.

[15] B. Chazelle, « A minimum spanning tree algorithm with inverse-Ackermann type complexity, » *Journal of the ACM*, t. 47, n° 6, p. 1028-1047, 2000.

[16] N. Christofides, A. Mingozzi et P. Toth, « State-space relaxation procedures for the computation of bounds to routing problems, » *Networks*, t. 11, n° 2, p. 145-164, 1981.

[17] V. Chvátal, « Edmonds polytopes and weakly hamiltonian graphs, » *Mathematical Programming*, t. 5, n° 1, p. 29-40, 1973.

[18] W. J. Cook, W. H. Cunningham, W. R. Pulleyblank et A. Schrijver, *Combinatorial Optimization*, 1re éd. John Wiley & Sons, Ltd, 1997.

[19] T. H. Cormen, C. E. Leiserson, R. L. Rivest et C. Stein, *Introduction to Algorithms*, 3e éd. The MIT Press, 2009, 1312 p.

[20] H. Crowder et M. W. Padberg, « Solving large-scale symmetric travelling salesman problems to optimality, » *Management Science*, t. 26, n° 5, p. 495-509, 1980.

[21] G. Dantzig, R. Fulkerson et S. Johnson, « Solution of a large-scale traveling-salesman problem, » *Journal of the Operations Research Society of America*, t. 2, n° 4, p. 393-410, 1954.

[22] G. B. Dantzig, « Origins of the simplex method, » in *A History of Scientific Computing*, New York, NY, USA : Association for Computing Machinery, 1990, p. 141-151.

[23] E. D. Demaine, G. M. Landau et O. Weimann, « On cartesian trees and range minimum queries, » *Algorithmica*, t. 68, n° 3, p. 610-625, 2014.

[24] B. Dixon, M. Rauch et R. Tarjan, « Verification and Sensitivity Analysis of Minimum Spanning Trees in Linear Time, » *SIAM Journal On Computing*, t. 21, p. 1184-1192, 1992.

[25] G. Dooms et I. Katriel, « The "not-too-heavy spanning tree" constraint, » in *Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems*, P. Van Hentenryck et L. Wolsey, éd., sér. Lecture Notes in Computer Science, Berlin, Heidelberg : Springer, 2007, p. 59-70.

[26] S. Ducomman, H. Cambazard et B. Penz, « Alternative filtering for the Weighted Circuit constraint: comparing lower bounds for the TSP and solving TSPTW, » in *Thirtieth AAAI Conference on Artificial Intelligence*, 2016.

[27] J.-G. Fages, « Exploitation de structures de graphe en programmation par contraintes, » Thèse, École des Mines de Nantes, 2014.

[28] J.-G. Fages, X. Lorca et L.-M. Rousseau, « The salesman and the tree: the importance of search in CP, » *Constraints*, t. 21, n° 2, p. 145-162, 2016.

[29] J.-G. Fages, C. Prud'homme et X. Lorca. « Choco Graph. » (2019), adresse : https://github.com/chocoteam/choco-graph (visité le 31/07/2021).

[30] T. Fahle et M. Sellmann, « Cost based filtering for the constrained knapsack problem, » *Annals of Operations Research*, t. 115, n° 1, p. 73-93, 2002.

[31] M. C. Ferris, O. L. Mangasarian et S. J. Wright, *Linear Programming with MATLAB*, sér. MOS-SIAM Series on Optimization. Society for Industrial and Applied Mathematics, 2007, 270 p.

[32] F. Focacci, A. Lodi et M. Milano, « Cost-based domain filtering, » in *Principles and Practice of Constraint Programming*, J. Jaffar, éd., sér. Lecture Notes in Computer Science, Berlin, Heidelberg : Springer, 1999, p. 189-203.

[33] F. Focacci, A. Lodi et M. Milano, « A hybrid exact algorithm for the TSPTW, » *INFORMS Journal on Computing*, t. 14, n° 4, p. 403-417, 2002.

[34] F. Focacci, A. Lodi et M. Milano, « Embedding relaxations in global constraints for solving TSP and TSPTW, » *Annals of Mathematics and Artificial Intelligence*, t. 34, n° 4, p. 291-311, 2002.

[35] F. Focacci, A. Lodi, M. Milano et D. Vigo, « Solving TSP through the integration of OR and CP techniques, » *Electronic Notes in Discrete Mathematics*, CP98, Workshop on Large Scale Combinatorial Optimisation and Constraints, t. 1, p. 13-25, 1999.

[36] D. Fontaine, L. Michel et P. Van Hentenryck, « Constraint-based Lagrangian relaxation, » in *Principles and Practice of Constraint Programming*, sér. Lecture Notes in Computer Science, Cham : Springer International Publishing, 2014, p. 324-339.

[37] K. G. Francis et P. J. Stuckey, « Explaining circuit propagation, » *Constraints*, t. 19, n° 1, p. 1-29, 2014.

[38] H. N. Gabow et R. E. Tarjan, « A linear-time algorithm for a special case of disjoint set union, » *Journal of Computer and System Sciences*, t. 30, n° 2, p. 209-221, 1985.

[39] R. E. Gomory, « Outline of an algorithm for integer solutions to linear programs and an algorithm for the mixed integer problem, » in *50 Years of Integer Programming 1958-2008*, M. Jünger et al., éd., Berlin, Heidelberg : Springer Berlin Heidelberg, 2010, p. 77-103.

[40] M. Grötschel et O. Holland, « Solution of large-scale symmetric travelling salesman problems, » *Mathematical Programming*, t. 51, n° 1, p. 141-202, 1991.

[41] M. Grötschel et M. W. Padberg, « On the symmetric travelling salesman problem I: inequalities, » *Mathematical Programming*, t. 16, n° 1, p. 265-280, 1979.

[42] R. M. Haralick et G. L. Elliott, « Increasing tree search efficiency for constraint satisfaction problems, » *Artificial Intelligence*, t. 14, n° 3, p. 263-313, 1980.

[43] M. Held et R. M. Karp, « The traveling-salesman problem and minimum spanning trees, » *Operations Research*, t. 18, n° 6, p. 1138-1162, 1970.

[44] M. Held et R. M. Karp, « The traveling-salesman problem and minimum spanning trees: part II, » *Mathematical Programming*, t. 1, n° 1, p. 6-25, 1971.

[45] K. Helsgaun, « An effective implementation of the Lin–Kernighan traveling salesman heuristic, » *European Journal of Operational Research*, t. 126, n° 1, p. 106-130, 2000.

[46] K. Helsgaun. « LKH. » (2018), adresse : http://akira.ruc.dk/~keld/research/LKH/ (visité le 17/06/2021).

[47] S. Hong, « A linear programming approach for the traveling salesman problem, » thèse de doct., Johns Hopkins University, Baltimore, Maryland, USA, 1972.

[48] N. Isoart et J.-C. Régin, « Integration of structural constraints into TSP models, » in *Principles and Practice of Constraint Programming*, T. Schiex et S. de Givry, éd., sér. Lecture Notes in Computer Science, Springer International Publishing, 2019, p. 284-299.

[49] N. Isoart et J.-C. Régin, « Adaptive CP-based Lagrangian relaxation for TSP solving, » in *Integration of Constraint Programming, Artificial Intelligence, and Operations Research*, E. Hebrard et N. Musliu, éd., sér. Lecture Notes in Computer Science, Cham : Springer International Publishing, 2020, p. 300-316.

[50] N. Isoart et J.-C. Régin, « Parallelization of TSP solving in CP, » in *Principles and Practice of Constraint Programming*, H. Simonis, éd., sér. Lecture Notes in Computer Science, Cham : Springer International Publishing, 2020, p. 410-426.

[51] R. Jonker et T. Volgenant, « Transforming asymmetric into symmetric traveling salesman problems, » *Operations Research Letters*, t. 2, n° 4, p. 161-163, 1983.

[52] N. Jussien, G. Rochart et X. Lorca, « Choco: an open source Java constraint programming library, » in *CPAIOR'08 Workshop on Open-Source Software for Integer and Contraint Programming (OSSICP'08)*, Paris, France, 2008, p. 1-10.

[53] L. G. Kaya et J. N. Hooker, « A filter for the circuit constraint, » in *Principles and Practice of Constraint Programming*, F. Benhamou, éd., sér. Lecture Notes in Computer Science, Berlin, Heidelberg : Springer, 2006, p. 706-710.

[54] L. Khachiyan, « Polynomial algorithms in linear programming, » *USSR Computational Mathematics and Mathematical Physics*, t. 20, n° 1, p. 53-72, 1980.

[55] V. King, « A Simpler Minimum Spanning Tree Verification Algorithm, » *Algorithmica*, t. 18, p. 263-270, 1997.

[56] J. B. Kruskal, « On the shortest spanning subtree of a graph and the traveling salesman problem, » *Proceedings of the American Mathematical Society*, t. 7, n° 1, p. 48-50, 1956.

[57] H. W. Kuhn, « The hungarian method for the assignment problem, » *Naval Research Logistics Quarterly*, t. 2, n° 1-2, p. 83-97, 1955.

[58] A. H. Land et A. G. Doig, « An automatic method of solving discrete programming problems, » *Econometrica*, t. 28, n° 3, p. 497-520, 1960.

[59] J.-L. Lauriere, « A language and a program for stating and solving combinatorial problems, » *Artificial Intelligence*, t. 10, n° 1, p. 29-127, 1978.

[60] C. Lecoutre, L. Saïs, S. Tabary et V. Vidal, « Reasoning from last conflict(s) in constraint programming, » *Artificial Intelligence*, t. 173, n° 18, p. 1592-1614, 2009.

[61] S. Lin et B. W. Kernighan, « An effective heuristic algorithm for the traveling-salesman problem, » *Operations Research*, t. 21, n° 2, p. 498-516, 1973.

[62] J. Lundgren, P. Värbrand et M. Rönnqvist, *Optimization*. Lund : Studentlitteratur AB, 2010, 537 p.

[63] J. Menana et S. Demassey, « Sequencing and counting with the multicost-regular constraint, » in *Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems*, W.-J. van Hoeve et J. N. Hooker, éd., sér. Lecture Notes in Computer Science, Berlin, Heidelberg : Springer, 2009, p. 178-192.

[64] M. Padberg et G. Rinaldi, « Optimization of a 532-city symmetric traveling salesman problem by branch and cut, » *Operations Research Letters*, t. 6, n° 1, p. 1-7, 1987.

[65] M. Padberg et G. Rinaldi, « A branch-and-cut algorithm for the resolution of large-scale symmetric traveling salesman problems, » *SIAM Review*, t. 33, n° 1, p. 60-100, 1991.

[66] R. C. Prim, « Shortest connection networks and some generalizations, » *The Bell System Technical Journal*, t. 36, n° 6, p. 1389-1401, 1957.

[67] C. Prud'homme, J.-G. Fages et X. Lorca. « Choco-solver. » (2021), adresse : https://choco-solver.org/ (visité le 31/07/2021).

[68] C.-G. Quimper, « Chapitre 2 - La fouille et le filtrage, » Diapositives du cours IFT-4001 / IFT-7020, 2021.

[69] J.-C. Régin, « A filtering algorithm for constraints of difference in CSPs, » in *Proceedings of the Twelfth AAAI National Conference on Artificial Intelligence*, Seattle, Washington : AAAI Press, 1994, p. 362-367.

[70] J.-C. Régin, « Simpler and incremental consistency checking and arc consistency filtering algorithms for the weighted spanning tree constraint, » in *Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems*, L. Perron et M. A. Trick, éd., sér. Lecture Notes in Computer Science, Berlin, Heidelberg : Springer, 2008, p. 233-247.

[71] J.-C. Régin, L.-M. Rousseau, M. Rueher et W.-J. van Hoeve, « The weighted spanning tree constraint revisited, » in *Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems*, A. Lodi, M. Milano et P. Toth, éd., sér. Lecture Notes in Computer Science, Berlin, Heidelberg : Springer, 2010, p. 287-291.

[72] G. Reinelt, « TSPLIB—A traveling salesman problem library, » *ORSA Journal on Computing*, t. 3, n° 4, p. 376-384, 1991.

[73] G. Reinelt. « TSPLIB. » (2013), adresse : http://comopt.ifi.uni-heidelberg.de/software/TSPLIB95/ (visité le 17/06/2021).

[74] K. Rioux-Paradis et C.-G. Quimper, « The WeightedCircuitsLmax constraint, » in *Integration of Constraint Programming, Artificial Intelligence, and Operations Research*, W.-J. van Hoeve, éd., sér. Lecture Notes in Computer Science, Cham : Springer International Publishing, 2018, p. 495-511.

[75] F. Rossi, P. van Beek et T. Walsh, éd., *Handbook of Constraint Programming*, sér. Foundations of Artificial Intelligence. Elsevier, 2006, t. 2, 978 p.

[76] A. Schrijver, *Theory of Linear and Integer Programming*, sér. Wiley-Interscience Series in Discrete Mathematics and Optimization. Chichester : Wiley, 1998.

[77] M. Sellmann, « Theoretical foundations of CP-based Lagrangian relaxation, » in *Principles and Practice of Constraint Programming – CP 2004*, M. Wallace, éd., sér. Lecture Notes in Computer Science, Berlin, Heidelberg : Springer, 2004, p. 634-647.

[78] M. Sellmann et T. Fahle, « CP-based Lagrangian relaxation for a multimedia application, » in *3rd International Workshop on Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems (CP-AI-OR)*, 2001, p. 1-14.

[79] M. Sellmann et T. Fahle, « Constraint programming based Lagrangian relaxation for the automatic recording problem, » *Annals of Operations Research*, t. 118, n° 1, p. 17-33, 2003.

[80] R. E. Tarjan, « Efficiency of a good but not linear set union algorithm, » *Journal of the ACM (JACM)*, t. 22, n° 2, p. 215-225, 1975.

[81] R. E. Tarjan, « Applications of path compression on balanced trees, » *Journal of the ACM (JACM)*, t. 26, n° 4, p. 690-715, 1979.

[82] R. E. Tarjan, « Minimum spanning trees, » in *Data Structures and Network Algorithms*, sér. CBMS-NSF Regional Conference Series in Applied Mathematics, Society for Industrial and Applied Mathematics, 1983, p. 71-83.

[83] L. A. Wolsey, *Integer Programming*. John Wiley & Sons, 2020, 373 p.