Titre: Title:	Utilisation de réseaux de neurones pour le filtrage basé sur les coûts en programmation par contraintes
Auteur: Author:	Emile Jehaes
Date:	2026
Type:	Mémoire ou thèse / Dissertation or Thesis
Référence: Citation:	Jehaes, E. (2026). Utilisation de réseaux de neurones pour le filtrage basé sur les coûts en programmation par contraintes [Master's thesis, Polytechnique Montréal]. PolyPublie. https://publications.polymtl.ca/76207/

URL de PolyPublie: PolyPublie URL:	https://publications.polymtl.ca/76207/
Directeurs de recherche: Advisors:	Quentin Cappart, & Louis-Martin Rousseau
Programme: Program:	Génie logiciel

POLYTECHNIQUE MONTRÉAL
affiliée à l'Université de Montréal

Utilisation de réseaux de neurones pour le filtrage basé sur les coûts en programmation par contraintes

EMILE JEHAES
Département de génie informatique et génie logiciel

Mémoire présenté en vue de l'obtention du diplôme de *Maîtrise ès sciences appliquées*
Génie informatique et logiciel

Avril 2026

POLYTECHNIQUE MONTRÉAL
affiliée à l'Université de Montréal

Ce mémoire intitulé :

Utilisation de réseaux de neurones pour le filtrage basé sur les coûts en programmation par contraintes

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

Daniel ALOISE, président
Quentin CAPPART, membre et directeur de recherche
Louis-Martin ROUSSEAU, membre et codirecteur de recherche
Claude-Guy QUIMPER, membre externe

DÉDICACE

A ma famille et mes amis,
merci

REMERCIEMENTS

Je tiens à exprimer mes sincères remerciements à toutes les personnes qui ont rendu possible la réalisation de ce mémoire.

Je remercie tout d'abord mon directeur de recherche, le Professeur Quentin Cappart, de m'avoir confié ce projet de recherche et de m'avoir guidé tout au long de sa réalisation. Son soutien constant, sa disponibilité et ses conseils précieux ont été essentiels pour mener à bien ce travail. De plus, il m'a offert l'opportunité de participer au développement d'Abyss ainsi qu'à l'organisation du cours INF8175, qui ont constitué des expériences très enrichissantes.

Je remercie également mon co-directeur de recherche, le Professeur Louis-Martin Rousseau, ainsi que la Professeure Bistra Dilkina, pour leur contribution experte à ce projet. Leurs commentaires et suggestions ont grandement contribué à ce travail de recherche.

Je tiens également à remercier Tristan Riou, qui a travaillé avec moi au début de ce projet et qui m'a permis d'apprécier cette épreuve souvent solitaire qu'est la rédaction d'un mémoire. Dans le même esprit, je remercie tous les membres du laboratoire CORAIL pour les discussions enrichissantes et l'ambiance de travail exceptionnelle.

Enfin, je remercie ma famille et mes amis pour leur soutien inconditionnel tout au long de mes études.

RÉSUMÉ

La programmation par contraintes est un paradigme permettant d'obtenir des solutions exactes à des problèmes combinatoires. Ces problèmes sont utilisés dans de nombreux domaines de l'industrie, mais leur complexité, entrainant des temps de résolution longs, peut constituer un défi. Ces dernières années, plusieurs méthodes guidées par les données ont été introduites avec des résultats prometteurs. Ces approches visent surtout à améliorer les choix de branchement ou le calcul des bornes primales, en vue d'accélérer la résolution.

Ce mémoire se focalise sur le mécanisme de propagation de certaines contraintes globales. En effet, plusieurs de ces contraintes utilisent un filtrage basé sur le coût nécessitant le calcul de bornes via la relaxation lagrangienne. La qualité de ces bornes impacte directement l'efficacité du filtrage de la contrainte et, par conséquent, les performances de résolution. Cette qualité est déterminée par les multiplicateurs de Lagrange utilisés dans la relaxation. Le calcul de multiplicateurs appropriés est donc essentiel. Ceux-ci proviennent d'un processus itératif qui nécessite de résoudre une relaxation à chaque itération et s'avère donc coûteux en temps de calcul.

Pour pallier ce problème, ce mémoire propose une approche guidée par les données centrée sur ces multiplicateurs. Un réseau de neurones sur graphe est utilisé pour modéliser la relation entre les caractéristiques du problème et les multiplicateurs. Ce type de réseau de neurones est bien adapté à la tâche, car la plupart des problèmes d'optimisation combinatoire peuvent être modélisés comme un graphe.

L'approche proposée a été intégrée dans un solveur de programmation par contraintes, et évaluée sur plusieurs problèmes. Les résultats montrent une réduction du temps de résolution, indiquant que la prédiction des multiplicateurs peut améliorer le filtrage selon le coût de certaines contraintes globales.

Cette approche est indépendante du problème et peut donc être intégrée dans les propagateurs de certaines contraintes globales dans des solveurs de programmation par contraintes. Bien que la réduction de temps de calcul actuelle soit modérée, des recherches futures sur l'amélioration de la stratégie d'entraînement et de l'architecture du modèle devraient permettre d'obtenir des gains de performance plus importants.

ABSTRACT

Constraint Programming is a powerful paradigm for obtaining exact solutions to combinatorial optimization problems. Those problems are widely encountered in industries but remain a challenge to solve due to their complexity resulting in long solving times. Recent data-driven methods have shown promising results. Most approaches focus on enhancing the branching strategies or strengthening the primal bounds to accelerate solving time.

Instead, this work focuses on the propagation mechanism of global constraints. Some of these constraints require cost-based filtering, which relies on the computation of bounds via Lagrangian relaxation. The tightness of these bounds directly impacts the efficiency of the filtering and, consequently, the overall solving time. Since the quality of these bounds depends on the Lagrangian multipliers used in the relaxation, computing appropriate multipliers is essential. However, they are obtained from an iterative process which, at each step, solves the relaxation, making it computationally expensive.

To address this issue, we propose a data-driven method focused on predicting multipliers. We use Graph Neural Networks to model the relationship between instances and appropriate multipliers. This type of neural network is well-suited to the task, as most combinatorial problems can be modeled as graphs. Consequently, they can capture the structural complexity of the instances. In addition, their flexibility with respect to the input size provides a general framework for this task.

The proposed approach was integrated into a general constraint programming solver and evaluated on different problems. The experimental results showed consistent improvements in the solving time, indicating that learning-based multiplier prediction can improve the cost-based filtering of global constraints.

This approach is problem-independent and can be integrated into the propagators of certain global constraints in general constraint programming solvers. Although the current time reduction is modest, further research on training strategies and model architecture may yield greater performance gains.

TABLE DES MATIÈRES

LISTE DES TABLEAUX

LISTE DES FIGURES

LISTE DES SIGLES ET ABRÉVIATIONS

atMostNValue	Contrainte *Au Plus N Différentes Valeurs*
atMostWValue	Contrainte *Au plus un poids W*
COP	Problème d'optimisation sous contraintes
CP	Programmation par contraintes
CSP	Problème de satisfaction de contraintes
GNN	Réseau de neurones graphique
LB	Borne inférieure
LLM	Modèle de langage de grande taille
LSCP	Problème de couverture par un ensemble de lieux
ML	Apprentissage automatique
MST	Arbre couvrant de poids minimum
TSP	Problème du voyageur de commerce
TSPTW	Problème du voyageur de commerce avec fenêtre de temps
UB	Borne supérieure
UFLP	Problème de localisation d'entrepôts non capacitaires
weightedCircuit	Contrainte *Circuit Pondéré*

CHAPITRE 1 INTRODUCTION

Ce premier chapitre vise à présenter le contexte de ce mémoire. Nous allons d'abord discuter de l'importance des problèmes d'optimisation combinatoire dans le contexte actuel, puis présenter la problématique étudiée. Enfin, les objectifs de recherche seront exposés, avant de conclure par un plan du mémoire.

1.1 Mise en contexte

Les problèmes d'optimisation combinatoire (COP) font l'objet de nombreux travaux de recherche, car ils permettent de modéliser un spectre très large de situations. On peut notamment citer le problème du voyageur de commerce (TSP) et ses variantes, utilisés pour les problèmes de livraison et de planification de trajets, ou encore les problèmes d'attribution et d'ordonnancement rencontrés dans les contextes industriels.

Il est donc nécessaire de les résoudre de manière efficace. Pour la plupart des COP étudiés ici, ils sont $\mathcal{NP}$-difficiles, entrainant une résolution exacte à temps exponentiel. Différentes approches ont été envisagées afin de pallier cette difficulté et de trouver des solutions optimales en temps raisonnable, notamment la programmation par contraintes (CP). Bien qu'elle ne soit pas la seule méthode permettant de résoudre ces problèmes, ce paradigme est répandu et donne de bons résultats, comme le montre Concorde [2], l'état de l'art pour la résolution optimale du TSP.

Naturellement, la recherche visant à améliorer la CP demeure importante. Des améliorations sont proposées chaque année afin d'optimiser les solveurs. Avec l'essor de l'apprentissage automatique et, en particulier, de l'apprentissage profond, il est naturel d'envisager leur utilisation dans le contexte de la CP. Ce mémoire propose une méthode d'apprentissage automatique destinée à améliorer la résolution de COP en recourant à la CP.

Les solveurs de CP s'appuient sur un mécanisme de séparation et d'évaluation pour trouver la solution optimale. Celui-ci repose sur la division de l'espace de recherche en sous-problèmes, puis sur l'évaluation de ces derniers. La force de la programmation par contraintes réside dans l'utilisation des différentes contraintes du problème pour réduire l'espace de recherche dans les sous-problèmes, en filtrant les valeurs des domaines des variables qui ne peuvent plus conduire à une solution. Il existe de nombreux types de filtrage, et celui étudié dans ce mémoire est le filtrage selon le coût (*cost-based filtering*).

Ce filtrage repose sur un concept simple. L'objectif de la résolution d'un COP est de trouver

la solution optimale, c'est-à-dire celle qui respecte les contraintes tout en minimisant une fonction objectif (la situation est identique si l'on souhaite maximiser cette fonction : il suffit d'inverser son signe). Par conséquent, pendant la résolution, les solutions dont la valeur de la fonction objectif dépasse celle de la meilleure solution trouvée jusqu'ici ne sont pas retenues.

Le filtrage par le coût intervient pour éliminer les choix menant à de telles solutions. Pour ce faire, il est possible de calculer une borne inférieure sur la valeur de la solution optimale au sein d'un sous-problème. À partir de cette borne, on filtre les valeurs des domaines qui ne peuvent pas conduire à une solution respectant cette borne, ce qui réduit l'espace de recherche et accélère la résolution, sans compromettre l'obtention de la solution optimale.

Pour calculer ces bornes, on recourt à des relaxations du problème, c'est-à-dire à des versions simplifiées pouvant être résolues plus rapidement, de sorte que la solution fournit une borne inférieure sur la valeur de la solution optimale du problème original. La qualité de ces bornes est un point critique pour le filtrage par le coût, certaines bornes inférieures conduisant à un filtrage plus efficace. En pratique, ces bornes sont contrôlées par un ensemble de pénalités, lesquelles doivent être choisies avec soin.

1.2 Problématique visée dans ce mémoire

1.2.1 Calcul de la borne inférieure

Grâce à ce mécanisme, les bornes inférieures sont calculées avec un processus itératif, qui nécessite de résoudre à chaque itération une relaxation du problème et d'ajuster les pénalités en fonction de la solution trouvée. Ce processus est coûteux en temps, car malgré la simplicité de la relaxation, elle doit être résolue de nombreuses fois.

Le filtrage par le coût est un mécanisme très puissant, qui pour certaines contraintes globales peut parfois accélérer la résolution de plusieurs ordres de grandeur [3]. Cependant, son coût en temps reste un point critique qui limite son efficacité. Améliorer l'efficacité du filtrage par le coût est donc essentiel pour accélérer la résolution de COP avec la CP. De plus, ce mécanisme étant applicable à de nombreuses contraintes globales, son amélioration influence significativement plusieurs COP.

1.3 Objectifs de recherche et contributions

L'objectif principal de ce mémoire est d'accélérer le filtrage par le coût pour les contraintes globales, afin de réduire le temps de résolution des COP au sein des solveurs de CP. Pour ce faire, nous proposons une approche générale fondée sur l'apprentissage profond, visant à

accélérer le calcul des bornes utilisées par le filtrage.

Cette approche permet d'initialiser les pénalités à l'aide de l'apprentissage, réduisant le nombre d'itérations nécessaires à la détermination des pénalités adéquates, et par conséquent accélérant le filtrage par le coût.

Le caractère général de la méthode est validé en l'appliquant à trois contraintes globales différentes, puis à des problèmes concrets faisant appel à ces contraintes. Cette démarche permet d'évaluer l'impact de notre approche sur les performances. Enfin, l'évaluation est réalisée dans un solveur CP généraliste, confirmant que cette méthode est applicable dans un cadre pratique.

L'approche a été implémentée dans le solveur de CP Choco [4], dans les contraintes globales de "*Circuit Pondéré*" (*weightedCircuit*), "*Au Plus N Différentes Valeurs*" (*atMostNValue*) et "*Au Plus un Poids W*" (*atMostWValue*). Ces contraintes n'ont pas été modifiées, ce qui rend l'implémentation pleinement compatible avec Choco, les relaxations utilisées pour le filtrage par le coût étant déjà implémentées. L'évaluation est d'ailleurs réalisée sur différents problèmes utilisant ces contraintes.

1.4 Plan du mémoire

Ce mémoire est articulé comme suit :

Dans le chapitre 2, nous proposons une mise en contexte de notre contribution, centrée sur une revue de littérature introduisant tous les concepts utilisés dans notre contribution.

Le chapitre 3 présente le cœur de notre contribution. Tout d'abord, nous décrivons notre approche. Ensuite, l'architecture du réseau de neurones utilisé est détaillée, suivie de la méthode d'entraînement de ce réseau de neurones ainsi que la fonction de perte proposée et la théorie derrière celle-ci.

Le chapitre 4 est centré autour de l'évaluation de notre contribution. Les trois contraintes globales utilisées ainsi que les problèmes correspondants sont présentés. Ensuite, dans le chapitre 5 nous décrivons les détails pratiques de l'implémentation. Les résultats numériques seront présentés et discutés, avant de conclure sur les performances de notre approche. Enfin, la conclusion abordera les limitations et perspectives futures de notre contribution.

CHAPITRE 2 REVUE DE LITTÉRATURE ET NOTIONS THÉORIQUES

Ce chapitre introduit les notions théoriques utilisées dans le mémoire, ainsi que la littérature liée. Nous introduisons tout d'abord la programmation par contraintes avant de revenir sur la notion de filtrage basé sur le coût. Enfin, nous présentons la relaxation lagrangienne, et son usage dans le cadre du filtrage selon le coût. Nous terminons ce chapitre en présentant les réseaux de neurones graphiques (GNN).

2.1 Programmation par contraintes

2.1.1 Problème de satisfaction de contraintes

Un problème de satisfaction de contraintes (CSP) peut être défini par un triplet $\langle \mathcal{X}, \mathcal{D}, \mathcal{C} \rangle$ où :

- $\mathcal{X}$ est l'ensemble des variables du problème,
- $\mathcal{D}$ est l'ensemble des domaines de ces variables,
- $\mathcal{C}$ est l'ensemble des contraintes.

Une solution consiste donc en une assignation de valeurs respectant toutes les contraintes.

Par exemple, on peut citer le problème des N-reines : On souhaite placer N reines sur un échiquier de taille $N \times N$ sans qu'elles ne se menacent les unes et les autres.

- $\mathcal{X} = \{x_1, x_2, ..., x_N\}$: Lors du choix des variables de décision, on sait que deux reines ne peuvent pas être sur la même colonne. Chaque variable représente la ligne de la reine dans la colonne correspondante.
- $\mathcal{D} = \{1, 2, ..., N\} \forall x_i \in \mathcal{X}$: Le domaine contient les numéros de chaque ligne.
- $\mathcal{C}$: Les contraintes assurent que deux reines se menacent si elles sont :
  1. sur la même ligne : On ajoute donc la contrainte $x_i \neq x_j \forall 1 \leq i < j \leq N$
  2. sur la même colonne : Pas besoin de contraintes, nos variables vérifient déjà cela par la définition de leur domaine.
  3. sur la même diagonale : $x_i - i \neq x_j - j \quad \forall 1 \leq i < j \leq N$ et $x_i + i \neq x_j + j \quad \forall 1 \leq i < j \leq N$

Une solution est donc une assignation des valeurs x_i qui vérifie ces contraintes.

Cependant, on s'intéresse dans ce mémoire aux problèmes d'optimisation combinatoire (COP) qui sont des CSP auxquels on ajoute une fonction objectif f, à optimiser. Un COP est donc défini par un quadruplet $\langle \mathcal{X}, \mathcal{D}, \mathcal{C}, f \rangle$. L'objectif n'est plus seulement de trouver une solution

satisfaisant les contraintes, mais de déterminer celle qui minimise ou maximise la fonction objectif.

2.1.2 Séparation et évaluation

Comme mentionné brièvement dans l'introduction, on peut notamment résoudre les COP à l'aide de solveurs de programmation par contraintes [5]. Nous présentons maintenant en détail le mécanisme de *séparation et évaluation* utilisé par les solveurs de CP. Ce mécanisme partitionne les domaines des variables afin de générer des sous-problèmes. En répétant cette opération, on va générer un arbre de recherche. Après la séparation, on va aussi réduire les domaines, en utilisant un mécanisme de filtrage qui sera détaillé plus loin. Trois cas de figure sont possibles :

- Le domaine d'au moins une variable est vide : Cela signifie qu'aucune solution n'existe dans la branche actuelle.
- Le domaine de toutes les variables est de taille 1 : On obtient alors une solution satisfaisant le CSP.
- Dans les autres cas, la procédure continue.

On peut illustrer ceci sur le petit CSP suivant :

$$\begin{aligned} x + y &\leq z \\ x &\in \{2, 4\} \\ y &\in \{4, 8\} \\ z &\in \{6, 10\} \end{aligned}$$

On retrouve l'ensemble des variables $\mathcal{X} = \{x, y, z\}$, des contraintes $x + y \leq z$ ainsi que les domaines des variables. La figure 2.1 illustre la séparation du domaine de z, qui permet de filtrer certaines valeurs des domaines de x et y dans la branche de gauche conduisant à une solution. À droite par contre, il est nécessaire de continuer la procédure, car x et y ont encore plusieurs valeurs dans leur domaine.

La résolution d'un CSP consiste donc à générer cet arbre jusqu'à obtenir une assignation satisfaisant les contraintes. Dans le cas d'un COP, cependant, puisque nous souhaitons obtenir la solution optimale, il est nécessaire de poursuivre la recherche après la première solution afin d'identifier celle qui optimise la fonction objectif.

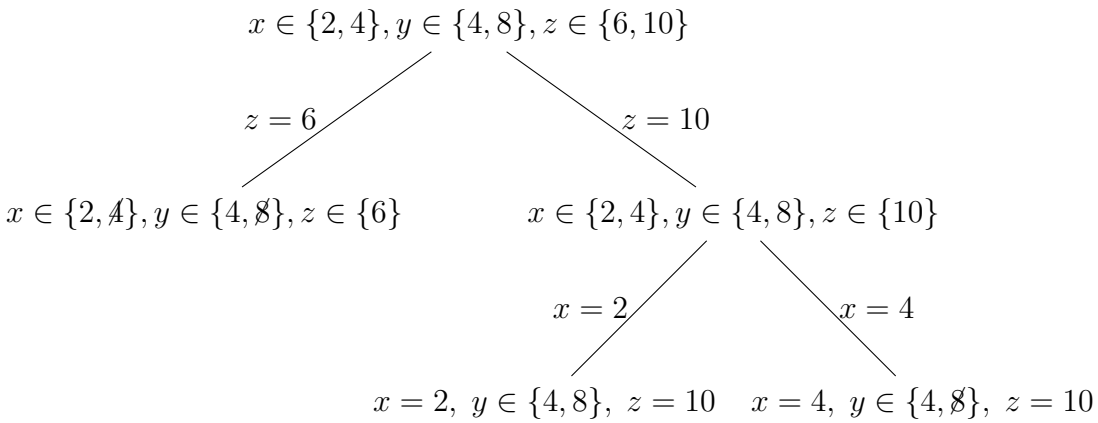

FIGURE 2.1 Exemple de l'algorithme de séparation et évaluation

2.1.3 Filtrage et propagation des contraintes

Dans le mécanisme de séparation et d'évaluation présenté précédemment, certaines valeurs des domaines des sous-problèmes sont filtrées. Dans la figure 2.1, on peut observer que dans la branche de gauche, la valeur 4 a été retirée du domaine de la variable x. Ce filtrage est permis par la contrainte du CSP, $x + y \leq z$. En effet, dans le sous-problème considéré, le domaine de z a été réduit à 6 ; le minimum de y étant 4, x ne peut donc plus prendre la valeur 4 tout en respectant la contrainte, ce qui implique $x \leq 2$ dans toute solution.

Le filtrage consiste ainsi à retirer du domaine les valeurs qui ne peuvent plus faire partie d'une solution. Ces valeurs sont identifiées en exploitant les contraintes du problème. La *propagation des contraintes* est le processus consistant à réduire les domaines des variables en éliminant les valeurs qui ne sont plus *supportées*. Une valeur est *supportée* pour une variable si elle peut figurer dans au moins une solution satisfaisant toutes les contraintes.

En pratique, détecter toutes les valeurs non supportées peut être coûteux. C'est pourquoi des algorithmes de propagation spécifiques existent pour certaines contraintes, offrant des filtrages plus ou moins efficaces. Pour la contrainte de *weightedCircuit* (*circuit pondéré*), par exemple, [6] explore plusieurs approches : le filtrage basé sur un problème d'affectation, la relaxation de Held-Karp [7], et le filtrage utilisant les n-chemins avec relaxation lagrangienne [8].

Le choix de l'algorithme de propagation est crucial : une propagation plus complète réduit la taille de l'arbre de recherche en détectant les conflits plus tôt, mais elle peut ralentir l'exécution. Trouver le compromis optimal entre qualité du filtrage et vitesse d'exécution est donc essentiel.

2.2 Filtrage selon le coût

Dans le cadre de la résolution d'un COP, comme on l'a mentionné dans l'introduction, nous ne sommes intéressés que par la solution optimale. Si une solution a déjà été trouvée, toute solution ayant une valeur objectif supérieure peut être ignorée.

En pratique, cela se fait en ajoutant une contrainte sous la forme :

$$f(\mathcal{X}) \leq z$$

avec z la valeur objectif de la meilleure solution trouvée jusqu'ici. Ceci va donc permettre d'accélérer la recherche, car on peut maintenant ignorer certaines branches. Si on parvient à prouver que toute solution dans un sous-problème aura une valeur objectif plus élevée que la meilleure trouvée jusqu'ici, alors on peut le filtrer.

Pour cela, [9] a proposé la méthode de filtrage selon le coût que nous utilisons ici. Le potentiel de cette méthode n'est plus à démontrer, et elle a été appuyée par de nombreuses publications [3], [9], [10]. On retiendra notamment [10] qui présente un usage du filtrage selon le coût qui permet d'obtenir la consistance d'arc plus efficacement. *La consistance d'arc* est une propriété de filtrage qui garantit que pour chaque variable et chaque valeur de son domaine, il y a au moins une valeur compatible dans chaque variable connectée par au moins une contrainte.

On maintient en tout temps une borne supérieure (UB) sur la valeur optimale de la fonction objectif du problème global. Celle-ci est généralement la meilleure solution trouvée jusqu'à ce point de la recherche. Ensuite, lors du filtrage d'un sous-problème, on calcule une borne inférieure (LB) sur la valeur de l'objectif dans ce sous-problème. On détaillera plus loin les méthodes utilisées afin de calculer ces bornes inférieures. Dès lors, on peut utiliser ces bornes pour filtrer certains sous-problèmes :

$$LB_{\text{sous-problème}} > UB \Rightarrow \text{on peut filtrer le sous-problème}$$

En effet, cette inégalité nous garantit qu'on ne pourra pas trouver une solution avec une meilleure valeur dans ce sous-problème, et on peut donc arrêter la recherche dans celui-ci. Cependant, le filtrage par le coût va plus loin, et permet également de filtrer certaines valeurs du domaine. En effet, supposons que pour une variable $x \in \mathcal{X}$ et une valeur de son domaine $v \in \mathcal{D}(x)$ on calcule les deux quantités suivantes :

- $LB_{x=v} := LB(\text{sous-problème avec } x = v)$
- $LB_{x\neq v} := LB(\text{sous-problème avec } x \neq v)$

On peut donc déduire les conditions suivantes :

— Si $LB_{x=v} > UB$, alors on peut filtrer v
— Si $LB_{x\neq v} > UB$, alors on peut assigner x à v.

Ces conditions découlent toujours simplement du fait que l'on ne recherche que des solutions améliorant le coût.

Lors du filtrage, on peut calculer ces bornes pour toutes les valeurs afin d'éliminer celles qui ne peuvent améliorer la solution. On va illustrer ceci sur un exemple dans la figure 2.2. On cherche à placer deux tâches pour minimiser le temps total entre le début de la première et la fin de la seconde. Elles ne peuvent pas être exécutées en même temps. Dans la figure 2.2, on a un exemple de solution déjà trouvée lors de la recherche, avec un temps total de 5. Si on applique le filtrage selon le coût, en considérant toutes les valeurs des domaines, on va obtenir la figure 2.3 où les valeurs des domaines qui ont été retirées sont en rouge. En effet, on peut vérifier aisément que placer une tâche sur une zone rouge entraine un temps total plus grand que 5.

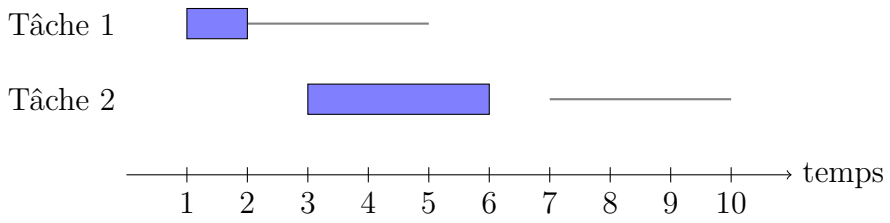

FIGURE 2.2 Exemple pour le filtrage par le coût : Une solution déjà trouvée

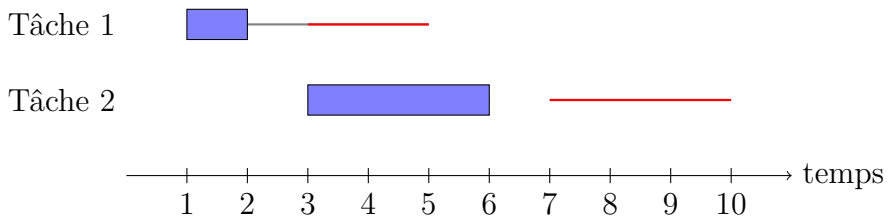

FIGURE 2.3 Exemple pour le filtrage par le coût : Résultat après filtrage

Ce type de filtrage peut donc grandement accélérer la recherche, mais calculer une borne inférieure à chaque filtrage pour toutes les assignations de valeurs possibles n'est pas faisable en temps raisonnable.

Pour éviter cela, on introduit la notion de coût réduit (cr) :

— $cr(x \neq v)$ correspond au coût minimal qui s'ajoutera à la borne inférieure si on retire v du domaine de x.

— $cr(x = v)$ correspond au coût minimal qui s'ajoutera à la borne inférieure si on fixe x à v.

On a dès lors les relations suivantes :

— $LB_{x=v} = LB_{\text{sous-problème}} + cr(x = v)$

— $LB_{x\neq v} = LB_{\text{sous-problème}} + cr(x \neq v)$

Et on peut donc réécrire les conditions de filtrage :

— Si $LB_{\text{sous-problème}} + cr(x = v) > UB$, alors on peut filtrer v

— Si $LB_{\text{sous-problème}} + cr(x \neq v) > UB$, alors on peut assigner x à v.

Pour l'exemple des tâches, la borne inférieure du sous-problème est 4, correspondant à la durée cumulée minimale des tâches. En effet, les tâches ne pouvant pas se superposer, le temps total doit être au moins aussi grand que la durée cumulée des tâches. Quant au coût réduit de planifier la tâche 2 de 7 à 10, il est de 2 car on ne pourra rien planifier de 5 à 7. On peut filtrer cet intervalle puisque $LB + cr = 4 + 2 > UB = 5$.

Le calcul des coûts réduits étant bien plus rapide que celui des bornes inférieures, c'est cette approche qui sera utilisée pour le filtrage. Ce filtrage permet de réduire fortement la taille de l'arbre de recherche pour certains problèmes NP-difficiles, mais il nécessite des calculs additionnels. En effet, la borne supérieure est obtenue simplement à l'aide des solutions trouvées, mais la borne inférieure doit être recalculée pour chaque sous-problème. En pratique, les coûts réduits sont eux obtenus de manière assez directe à partir de la borne inférieure, et ne ralentissent donc pas l'exécution.

Il est donc important de pouvoir obtenir des bornes inférieures de façon efficace. De plus, on peut filtrer des valeurs lorsque $LB + cr > UB$. Comme on essaie de filtrer le plus de valeurs au plus vite afin de réduire la taille de la recherche, on souhaite maximiser le nombre d'éléments du domaine qui vérifient cette inégalité. On veut donc rendre les bornes inférieures et supérieures les plus serrées possibles. Nous allons maintenant détailler le calcul des bornes inférieures.

2.3 Relaxation Lagrangienne

Une des méthodes courantes pour obtenir une borne inférieure sur la solution optimale d'un problème est de recourir à une relaxation. Cela signifie résoudre une version plus simple du problème, où certaines contraintes rendant la résolution difficile ont été retirées.

Ici, nous allons utiliser les relaxations lagrangiennes. Ce type de relaxation revient à retirer les contraintes problématiques, et à placer dans la fonction objectif un terme qui va pénaliser le non-respect de cette contrainte.

On va illustrer ceci sur un exemple. Supposons que nous souhaitons résoudre le COP suivant :

$$
\begin{aligned}
min_x \quad & p^T x && (2.1)\\
s.t. \; & Ax \geq b && (2.2)\\
& C(x) \geq d && (2.3)\\
& x, p \in \mathbb{R}^n, \quad A \in \mathbb{R}^{m_1 \times n}, \quad b \in \mathbb{R}^{m_1}, \quad d \in \mathbb{R}^{m_2} && (2.4)
\end{aligned}
$$

Avec $C(x)$ une fonction de x quelconque $\mathbb{R}^n \to \mathbb{R}^{m_2}$. On va supposer ici que c'est la contrainte 2.3 qui complique la résolution de ce problème. Pour obtenir des bornes inférieures efficaces, on considère une relaxation de cette contrainte.

Pour cela, nous allons la retirer et ajouter un terme de pénalité dans l'objectif :

$$
\begin{aligned}
min_x \quad & p^T x + \lambda^T (d - C(x)) && (2.5)\\
s.t. \; & Ax \geq b && (2.6)\\
& x, p \in \mathbb{R}^n, \quad A \in \mathbb{R}^{m_1 \times n}, \quad b \in \mathbb{R}^{m_1}, \quad d \in \mathbb{R}^{m_2} && (2.7)\\
& \lambda \in \mathbb{R}_+^{m_2} && (2.8)
\end{aligned}
$$

Nous avons maintenant un problème plus simple à résoudre. On peut remarquer que la pénalité dans l'objectif est contrôlée par des multiplicateurs non-négatifs λ qui sont introduits plus en détail par la suite. Cette solution optimale fournit alors une borne inférieure du problème initial.

Nous noterons $f_{\mathcal{L}}(x, \lambda)$ dans ce qui suit la fonction objectif de la relaxation lagrangienne et nous supposons que la contrainte relaxée peut s'écrire sous la forme $C(x) \geq d$. De plus, x^* est la solution optimale au problème initial, et $x^*_{\mathcal{L}}$ est la solution optimale de la relaxation lagrangienne. On obtient la relation suivante :

$$f_{\mathcal{L}}(x^*_{\mathcal{L}}, \lambda) \leq f(x^*) \qquad \forall \lambda \in \mathbb{R}_+^{m_2}$$

Cette relation est simple à vérifier. En effet, x^* est faisable pour la relaxation, et $x^*_{\mathcal{L}}$ étant la solution optimale de cette relaxation, on a :

$$f_{\mathcal{L}}(x^*_{\mathcal{L}}, \lambda) \leq f_{\mathcal{L}}(x^*, \lambda) = f(x^*) + \lambda^T (d - C(x^*))$$

De plus, x^* étant faisable pour le problème initial, cela signifie que $d - C(x^*) \leq 0$. Comme

les multiplicateurs sont positifs, on a donc :

$$f_{\mathcal{L}}(x^*_{\mathcal{L}}, \lambda) \leq f_{\mathcal{L}}(x^*, \lambda) = f(x^*) + \lambda^T(d - C(x^*)) \leq f(x^*)$$

En résolvant cette relaxation dans laquelle les contraintes difficiles sont retirées, on trouve donc la borne inférieure nécessaire au filtrage par le coût. Ainsi, on obtient une borne inférieure efficace à calculer, même lorsque la contrainte relaxée rend le problème initial difficile.

L'usage de la relaxation lagrangienne a été largement étudié et donne généralement de bons résultats. On peut notamment citer [11] qui utilise la relaxation lagrangienne pour du filtrage par le coût, en améliorant la qualité de la borne inférieure par des modifications locales des multiplicateurs de Lagrange pour le problème du voyageur de commerce. Une approche similaire a été appliquée à la contrainte de *AtMostNValue* dans [12].

2.3.1 Calcul des multiplicateurs de Lagrange

Dans le filtrage par le coût, on souhaite obtenir des bornes inférieures aussi serrées que possible afin de maximiser le filtrage et réduire la taille de l'arbre de recherche, diminuant ainsi le temps de résolution.

Cependant jusqu'ici, on n'a aucune garantie sur la qualité de la borne inférieure obtenue par la relaxation lagrangienne. En pratique, cette borne est contrôlée par les multiplicateurs non négatifs λ, appelés les multiplicateurs de Lagrange. Comme on souhaite trouver la borne la plus serrée possible, on cherche à résoudre le problème de maximisation de la borne inférieure :

$$max_{\lambda \in \mathbb{R}^{m_2}}\ min_{x \in \mathbb{R}^n}\ f(x) + \lambda^T(d - C(x))$$
$$s.t. \quad \text{contraintes non relaxées}$$

Malheureusement, ce problème ne peut plus être résolu efficacement. On va donc recourir à une méthode d'optimisation, à savoir les itérations de sous-gradient pour trouver les multiplicateurs de Lagrange qui maximisent la borne inférieure. Cela consiste à résoudre la relaxation lagrangienne pour des multiplicateurs initiaux (souvent 0), calculer le sous-gradient, mettre à jour les multiplicateurs dans la direction de ce sous-gradient, et recommencer. L'objectif de ce processus itératif est de trouver des multiplicateurs qui maximisent la borne inférieure.

Cependant, à chaque itération on doit résoudre le problème relaxé. Même si cette résolution est rapide, on doit répéter celle-ci à chaque itération de sous-gradient. De plus, ces itérations doivent être effectuées pour filtrer par le coût chaque nœud de l'arbre de recherche. Ce processus est donc coûteux en temps, et pourtant c'est celui qui est bien souvent utilisé dans

les solveurs faute d'en avoir un meilleur.

Les itérations de sous-gradient pour l'optimisation des multiplicateurs de Lagrange utilisent généralement un pas géométrique, et sont appliquées dans l'algorithme 1. Nous détaillerons dans le chapitre suivant comment nous obtenons le sous-gradient.

Algorithm 1 Itérations de sous-gradient

▷ **Param :** $\epsilon \in \mathbb{R}_+$, $\alpha_0 \in]0, 1[$

$\lambda \leftarrow \mathbf{0}$, $LB \leftarrow -\infty$, $\alpha \leftarrow \alpha_0$
while $|LB - LB_{\mathcal{L}}(\lambda)| \geq \epsilon$ **do**
  $LB \leftarrow LB_{\mathcal{L}}(\lambda)$
  $\boldsymbol{\lambda} \leftarrow max(\boldsymbol{\lambda} + \alpha \nabla_{\lambda} \mathbf{LB}, \mathbf{0})$
  $\alpha \leftarrow \alpha \cdot \alpha_0$
end while
return λ

Il est toutefois important de noter que maximiser la borne inférieure ne va pas nécessairement mener au meilleur filtrage. Ceci est notamment illustré dans [13]. L'objectif de ce mémoire est donc de réduire le temps pris par ces itérations en recourant à l'apprentissage automatique. Avant de détailler cette approche, nous introduisons les concepts fondamentaux de l'apprentissage automatique et des réseaux de neurones graphiques (GNN).

2.4 Réseaux de neurones graphiques

Un réseau de neurones est une fonction paramétrée dont les paramètres ont été calibrés de façon à produire les sorties attendues en fonction des entrées. Il est composé de différentes couches, généralement définies comme suit :

Soit $\boldsymbol{x}^{k-1} \in \mathbb{R}^{h^{k-1} \times n}$ l'entrée de la couche k, $\boldsymbol{W}^k \in \mathbb{R}^{h^k \times h^{k-1}}$ une matrice de paramètres et $\boldsymbol{b}^k \in \mathbb{R}^{h^k}$ un vecteur de paramètres. Soit σ^k une fonction non linéaire. Alors la sortie de la couche k est la suivante :

$$\boldsymbol{x}^k = \sigma^k \left(\boldsymbol{W}^k \cdot \boldsymbol{x}^{k-1} + \boldsymbol{b}^k \right)$$

En utilisant de nombreuses couches, on peut approximer des fonctions complexes à l'aide de ces réseaux de neurones. Ceux-ci prennent donc en entrée des vecteurs sur lesquels ils effectuent leur prédiction. Cependant, lors de la prédiction chaque entrée est traitée de manière indépendante dans un réseau de neurones classique.

Dans le cas qui nous intéresse dans ce mémoire, nous verrons qu'il est préférable de traiter toutes les entrées de façon liée. Pour cela, nous utiliserons les GNN. Les GNN ont été in-

troduits par [14], et ont depuis été utilisés dans de nombreux domaines, notamment pour l'apprentissage de relations [15] ou encore pour la prédiction de molécules [16]. Ce type de réseau prend directement en entrée la représentation d'un graphe. En particulier, le type de couche que nous utiliserons est l'architecture GraphSAGE présentée par [17], que nous allons détailler ci-après.

Le graphe est représenté sous la forme d'une matrice où chaque ligne encode la représentation d'un nœud avec des caractéristiques dépendant de la situation. De plus, le graphe est représenté par sa matrice d'adjacence. Pour les graphes dirigés ou pondérés, on peut également inclure une matrice de caractéristiques des arêtes en adaptant légèrement l'architecture, ce qui sera présenté ultérieurement.

Une couche de GraphSAGE est composée de deux étapes. À la première étape, chaque nœud reçoit un message de tous ses voisins, contenant leurs informations, puis va agréger ces messages, en utilisant par exemple comme proposé par [17] la moyenne. Dans ce qui suit, $\mathcal{N}(i)$ désigne le voisinage du nœud i, et $\boldsymbol{x}_i^{k-1} \in \mathbb{R}^{h^{k-1}}$ l'entrée de la couche k pour le nœud i. À nouveau, $\boldsymbol{W}_1^k \in \mathbb{R}^{h^{k\prime} \times h^{k-1}}$ et $\boldsymbol{b}^k \in \mathbb{R}^{h^{k\prime}}$ sont respectivement une matrice et un vecteur de paramètres. Alors on peut calculer le résultat de l'agrégation pour le nœud i :

$$\boldsymbol{x}_i^{k\prime} = \frac{1}{|\mathcal{N}(i)|} \sum_{j \in \mathcal{N}(i)} \left[\sigma^k \left(\boldsymbol{W}_1^k \cdot \boldsymbol{x}_j^{k-1} + \boldsymbol{b}^k \right) \right]$$

On peut voir que le message émis par chaque nœud est issu de sa représentation à la couche précédente. Ensuite, chaque nœud met à jour sa représentation en combinant sa propre représentation précédente et le message agrégé des voisins avec $\boldsymbol{W}_2^k \in \mathbb{R}^{h^k \times h^{k\prime}}$ et $\boldsymbol{W}_3^k \in \mathbb{R}^{h^k \times h^{k-1}}$ d'autres matrices de poids :

$$\boldsymbol{x}_i^k = \sigma^k \left(\boldsymbol{W}_3^k \boldsymbol{x}_i^{k-1} + \boldsymbol{W}_2^k \boldsymbol{x}_i^{k\prime} \right)$$

Ces deux étapes constituent donc une couche, et seront répétées à chaque couche du GNN. Dans la figure 2.4, on peut voir une représentation simplifiée d'une couche de GraphSAGE appliquée à un nœud x_1.

Selon la tâche considérée, on placera donc après ces couches GraphSAGE un réseau de neurones classique qui servira à la classification ou à la régression sur chaque nœud. Étant donné que toutes les couches sont différentiables, l'entraînement de ce réseau de neurones peut se faire par descente de gradient traditionnelle. C'est-à-dire que l'on prédit la sortie sur les données d'entraînement $F(W, X)$ avec W les paramètres du GNN, puis calcule l'erreur de prédiction à l'aide de la fonction de perte L par rapport à la sortie attendue Y. Enfin, on

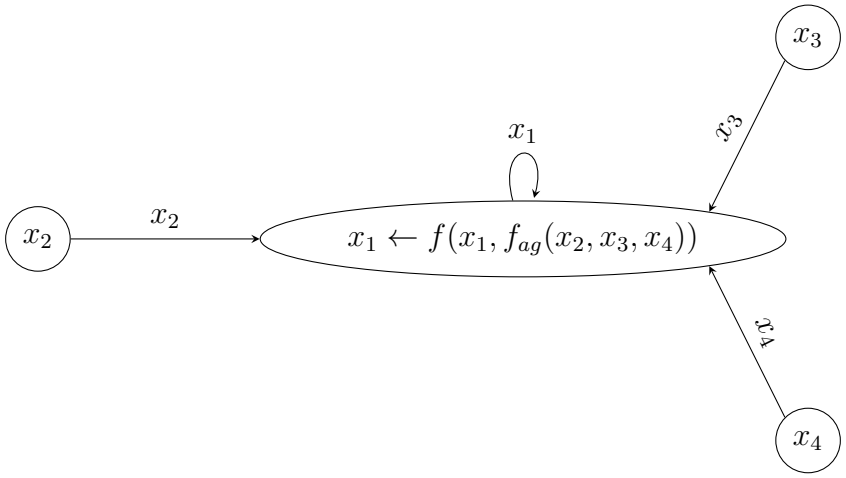

FIGURE 2.4 Représentation d'une couche du GNN appliquée à x_1. Ses voisins envoient leurs messages à x_1, qui les combine pour mettre à jour sa représentation. (La fonction f contient les différentes matrices de poids introduites précédemment et f_{ag} est la fonction d'agrégation)

rétropropage le gradient de l'erreur de prédiction dans le réseau de neurones afin de mettre à jour les poids (avec η le taux d'apprentissage) :

$$\hat{Y} = F(W, X)$$

$$W \leftarrow W - \eta \frac{\partial L(\hat{Y}, Y)}{\partial W}$$

Le gradient de la perte par rapport aux poids est calculé grâce à la règle de la dérivation à la chaîne, et du fait que l'on connaît

$$\frac{\partial L}{\partial \hat{Y}}$$

puisque la fonction de perte choisie est différentiable. Ceci sera détaillé dans le chapitre suivant. De plus

$$\frac{\partial \hat{Y}}{\partial W}$$

est connu car toutes les couches sont différentiables.

Néanmoins, ici on va utiliser une méthode d'apprentissage appelée l'apprentissage auto-supervisé par gradient. C'est-à-dire que l'on ne connaît pas les valeurs attendues Y. En revanche on sait calculer directement la direction dans laquelle ils se trouvent, donc le gradient de la perte. L'apprentissage sera donc guidé par ce gradient, et on pourra entraîner le GNN.

La représentation d'un COP sous la forme d'un graphe, ainsi que les liens avec les GNN, sont

au cœur de plusieurs publications récentes, dont notamment le tour d'horizon de [18].

2.5 État de l'art

Avant de décrire notre approche, il est important de situer ce mémoire dans le contexte de la littérature existante.

Premièrement, la programmation par contraintes n'est pas la seule approche pour résoudre les problèmes d'optimisation combinatoire. Parmi les alternatives, on retrouve :

— La programmation linéaire en nombres entiers mixtes (MILP) [19], qui est une méthode exacte très populaire pour les problèmes d'optimisation combinatoire, et la méthode état de l'art pour de nombreux problèmes. Les solveurs Gurobi [20] et CPLEX sont des exemples de solveurs MILP très performants. Cependant, l'approche MILP est applicable uniquement à des problèmes pouvant être formulés avec un objectif et des contraintes linéaires, ce qui n'est pas le cas de tous les problèmes d'optimisation combinatoire.

— La programmation dynamique repose sur la décomposition d'un problème en sous-problèmes plus petits. Cette approche est souvent limitée par la taille de l'espace de recherche, et est donc souvent combinée avec d'autres méthodes pour pallier ce problème. Par exemple [21] utilise la programmation dynamique pour de l'optimisation en plusieurs étapes, et le problème du sac à dos 0/1 qui peut être résolu en temps pseudo-polynomial grâce à la programmation dynamique.

Certaines approches ne cherchent pas nécessairement à trouver la solution optimale, mais plutôt une solution de bonne qualité en un temps raisonnable. On peut citer les algorithmes de recherche locale qui reposent sur des méta-heuristiques pour explorer l'espace de recherche. Ce champ de méthode est très diversifié, [22] introduit notamment une approche automatisant la conception de l'heuristique, qui permet notamment de résoudre le problème de routage de véhicules.

La programmation par contraintes permet quant à elle de résoudre de façon exacte un large éventail de problèmes d'optimisation combinatoire, incluant ceux qui ne peuvent pas être formulés avec des contraintes et un objectif linéaires. Bien entendu de très nombreux travaux ont été conduits afin d'améliorer les solveurs de programmation par contraintes. On peut notamment distinguer les axes d'amélioration suivants :

— La modélisation : Les contraintes globales permettent de mieux capturer la structure du problème, et une modélisation adéquate peut mieux exploiter les symétries, et donc réduire l'espace de recherche. [23] propose une nouvelle contrainte globale pour les problèmes d'horaires qui permet d'accélérer la résolution.

— Les algorithmes de propagation : Comme mentionné précédemment, le type de filtrage utilisé influence fortement la taille de l'arbre de recherche. Il affecte également la complexité numérique de l'exécution. De nouveaux algorithmes de filtrage sont régulièrement proposés, comme [24] qui propose un nouvel algorithme de filtrage selon le coût pour la contrainte de cardinalité globale. On a aussi [25] qui propose un algorithme de filtrage conçu pour la relaxation de Held-Karp. Ce dernier utilise également les bornes duales afin d'améliorer le filtrage, mais sans recourir à l'apprentissage automatique.
— Les stratégies de recherche et heuristiques de branchement : Le choix des variables à assigner, ainsi que des valeurs influencent également la taille de l'arbre de recherche, par exemple en permettant de détecter plus rapidement une incohérence. [26] présente une nouvelle heuristique de branchement pour les CSP, utilisant la recherche basée sur le comptage de solutions.
— Enfin, de plus en plus de travaux hybrides sont menés, combinant la programmation par contraintes et d'autres méthodes d'optimisation, comme l'approche MILP+CP pour le problème des gares de triage [27], ou encore [28] qui propose d'utiliser de la recherche locale pour améliorer les performances d'un solveur incomplet de CP pour les problèmes de placement de facilités.

Parmi ces méthodes hybrides, l'apprentissage automatique est de plus en plus présent dans l'état de l'art. Nous allons maintenant explorer ces différentes approches hybrides. Une revue de littérature plus complète mais plus ancienne est présentée dans [29]. Cependant, nous allons surtout nous intéresser ici aux publications les plus récentes, qui sont plus pertinentes pour notre travail. Nous garderons cependant leur séparation en trois catégories, qui semble toujours adaptée : l'apprentissage de bout en bout, l'apprentissage pour paramétrer les solveurs et l'apprentissage pour guider la recherche en interagissant avec les algorithmes d'optimisation.

2.5.1 Apprentissage de bout en bout

Un premier type d'usage du ML pour la programmation par contraintes est l'apprentissage de bout en bout. Cela consiste à entraîner un modèle de ML à résoudre directement le problème, c'est-à-dire à trouver la solution optimale. Tassel et al. [30] proposent une approche où ils utilisent l'apprentissage par renforcement pour entraîner un modèle à résoudre des petites instances avec un solveur CP pour donner un retour. Ce modèle prédit ensuite des règles permettant d'obtenir de bonnes solutions pour des instances plus grandes. Cependant, cette méthode n'a aucune garantie d'optimalité, et est restreinte aux problèmes d'ordonnancement de tâches. On notera également [31] qui pour une tâche similaire utilise de l'apprentissage profond. Bien que ces deux méthodes donnent des résultats très prometteurs, elles ne fournissent

aucune garantie d'optimalité et ne sont pas généralisables à d'autres types de problèmes.

2.5.2 Apprentissage pour paramétrer les solveurs

Dans cette seconde approche, l'objectif est d'utiliser le ML pour paramétrer les solveurs. Une bonne illustration de cette approche est présentée dans [32], où l'objectif est de prédire si une configuration d'hyper-paramètres va être efficace pour la résolution. Une nouvelle catégorie d'approche a émergé avec le bond en avant des modèles de langage large (LLM), qui vise à utiliser les LLM afin de modéliser les COP [33].

2.5.3 Apprentissage pour guider la recherche en interagissant avec les algorithmes d'optimisation

La troisième approche, qui constitue celle d'intérêt pour ce mémoire, est l'usage de ML pour guider la recherche pendant la résolution. De nombreuses pistes pour améliorer les solveurs ont déjà été explorées.

En ce qui concerne les heuristiques de recherche, on peut notamment citer [34] qui utilise un GNN pour apprendre une heuristique d'ordonnancement de variables pour les CSP et [35] qui utilise un GNN pour apprendre une heuristique de sélection de valeur. Ces deux approches sont intéressantes et plus généralisables que les publications mentionnées précédemment.

L'apprentissage automatique a également été utilisé pour améliorer le branchement, comme dans [36] qui utilise l'apprentissage automatique pour pondérer différentes méthodes de branchement pour un ensemble d'instances, afin de choisir celles qui donneront les meilleurs résultats, et [37] qui entraine un modèle de ML à reproduire une stratégie de branchement très coûteuse mais qui réduit l'arbre de recherche, afin de la reproduire à moindre coût.

Certaines approches hybrides comme [38] utilisent le ML pour prédire des solutions, dont les caractéristiques sont ensuite utilisées pour guider la recherche du solveur CP en favorisant les choix de variables et de valeurs qui sont similaires à ceux de la solution prédite.

Cependant, aucune de ces approches hybrides n'explore le contexte de la relaxation lagrangienne utilisée dans le filtrage par le coût, qui est celui de ce mémoire. [39] ont introduit un apprentissage auto-supervisé pour prédire les multiplicateurs de Lagrange utilisés lors de la décomposition lagrangienne. Celle-ci est très performante, et est assez proche de ce mémoire. Cependant, la différence majeure est l'usage de la décomposition lagrangienne, qui est différente de la relaxation lagrangienne utilisée ici.

Il convient également de mentionner le papier [1] qui a inspiré ce mémoire. La même méthode

y est proposée, mais celle-ci est limitée au problème du voyageur de commerce dans un solveur dédié, tandis que l'approche proposée ici est généralisable et utilisée dans un solveur de CP généraliste.

D'autres travaux ont été conduits sur les bornes, comme [40] mais ceux-ci s'appliquent aux diagrammes de décision et sortent du cadre de la programmation par contrainte, et ne sont pas directement applicables à notre cas.

On notera [41] qui utilise l'apprentissage automatique dans le cadre d'une relaxation lagrangienne, mais l'objectif est de prédire des solutions faisables de haute qualité. Dans [42], les auteurs utilisent l'apprentissage automatique combiné avec une décomposition lagrangienne (en MILP) mais pour prédire la qualité de la borne obtenue et la complexité de son calcul, ce qui diffère du travail proposé ici.

Dans l'objectif de prédire les bornes d'une relaxation lagrangienne, à l'exception de [1,11,12, 39] mentionnées précédemment, il n'existe pas à notre connaissance d'autres publications qui visent à améliorer les multiplicateurs de Lagrange dans le cadre de la relaxation lagrangienne afin d'augmenter le filtrage. Nous avons donc choisi de nous concentrer sur cette approche prometteuse et encore inexplorée.

CHAPITRE 3 APPRENTISSAGE DES MULTIPLICATEURS DE LAGRANGE

3.1 Question de recherche

Comme mentionné précédemment, lors du filtrage selon le coût, il est nécessaire d'effectuer un processus itératif, afin de trouver les bornes inférieures. Cependant, ce processus itératif qui optimise les multiplicateurs de Lagrange est coûteux. Dans [1], les auteurs proposent d'utiliser un GNN afin de prédire les multiplicateurs de Lagrange pour la contrainte *weightedCircuit* (circuit pondéré) dans le cadre d'un problème du voyageur de commerce. Cette approche avait donné des résultats intéressants mais était implémentée dans le solveur de [3] qui se limite au cas classique du problème du voyageur de commerce. La question centrale de ce mémoire est la suivante.

Peut-on généraliser cette approche à des solveurs CP généraux et à des problèmes comportant d'autres contraintes ?

3.2 Innovation

L'utilisation de l'apprentissage profond pour accélérer le calcul des bornes inférieures dans le filtrage selon le coût en CP n'a été envisagée que récemment, et il y a encore très peu de publications sur le sujet. On notera que [39] propose une approche similaire, qui est néanmoins destinée aux décompositions lagrangiennes. Quant au papier [1], nous cherchons à transformer cette procédure spécialisée en une méthode générale. De plus, la méthode initiale reposait sur certaines caractéristiques du problème. Nous souhaitons utiliser uniquement les données de la contrainte pour généraliser à d'autres contraintes.

Il est crucial de mentionner l'approche proposée dans [11] pour la contrainte de *weighted-Circuit* et [12] pour *AtMostNValue* qui vise également à améliorer le filtrage en travaillant sur les multiplicateurs de Lagrange. Cependant, ces approches ont pour objectif d'améliorer les multiplicateurs de Lagrange obtenus à la fin du processus itératif en leur appliquant des altérations locales, tandis que notre travail se concentre sur l'initialisation de ces itérations.

Bien que l'objectif soit de généraliser l'approche de [1] à d'autres contraintes, il reste nécessaire d'entrainer un modèle pour chaque contrainte globale que l'on souhaite améliorer. En revanche, une fois que le modèle est entrainé pour une contrainte globale, il peut être utilisé pour tout problème intégrant cette contrainte. C'est un point important, car cela signifie que

l'effort d'entrainement est amorti sur tous les problèmes utilisant la même contrainte globale.

3.3 Méthodologie

Comme mentionné précédemment, le but est de réduire le temps de calcul nécessaire pour trouver des multiplicateurs de Lagrange donnant une borne inférieure de bonne qualité. Pour cela, dans l'algorithme précédent, nous remplaçons l'initialisation nulle des multiplicateurs par une prédiction fournie avec un GNN. Notre objectif est donc de réduire le nombre d'itérations de sous-gradient nécessaires avant d'atteindre une borne satisfaisante. Cela réduira le nombre de résolutions de la relaxation et accélérera la résolution du problème.

La majorité des COP pouvant être représentés sous la forme d'un problème sur un graphe $G(V, E)$ (avec V l'ensemble des sommets et E les arêtes), nous utilisons donc des GNN dans ce mémoire.

Dans la figure 3.1, on peut voir la modification que l'on a apportée. La fonction $\Phi_W(.)$ est donc un GNN. Il est important de noter que la relaxation lagrangienne donne toujours des bornes valides. De ce fait, l'usage de notre réseau de neurones nous garantit également de toujours trouver une borne valide pour le filtrage, ce qui est indispensable.

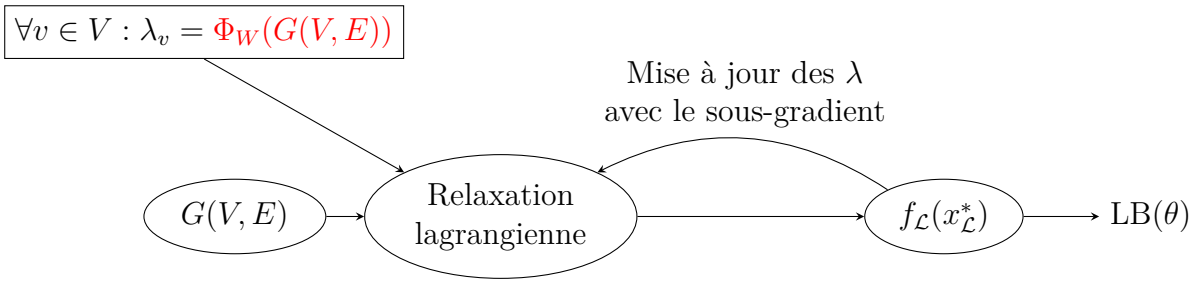

FIGURE 3.1 Itérations de sous-gradient avec initialisation avec le GNN. Schéma adapté de [1]

Concernant l'entrainement de ce GNN, un apprentissage supervisé pourrait donner des résultats intéressants, mais nous ne connaissons pas les multiplicateurs de Lagrange optimaux. Nous allons donc nous tourner vers l'apprentissage auto-supervisé par gradient. Pour appliquer ce type d'apprentissage, il est nécessaire de définir un gradient permettant de mettre à jour les poids du GNN.

3.3.1 Cible de l'apprentissage

L'objectif de l'apprentissage est d'obtenir les multiplicateurs de Lagrange qui maximisent le filtrage. Cependant, il est difficile de calculer ceux-ci. À la place, nous allons utiliser la borne

inférieure. En effet, maximiser la borne inférieure tend généralement à améliorer le filtrage, même s'il serait préférable, lorsque cela est possible, de maximiser directement le filtrage.

Notons $LB_{\mathcal{L}}(G(V,E),\lambda)$ la fonction qui, étant donné une instance de problème et les multiplicateurs de Lagrange, va calculer la borne inférieure obtenue en résolvant le problème relaxé. Nous voulons entrainer le GNN afin de maximiser cette borne, soit formellement :

$$max_W \quad LB_{\mathcal{L}}\left(G(V,E), \Phi_W(G(V,E))\right)$$

3.3.2 Entrainement du réseau de neurones

Nous cherchons à calculer le gradient suivant pour mettre à jour le réseau de neurones :

$$\frac{\partial LB_{\mathcal{L}}\left(G(V,E), \boldsymbol{\lambda}_W\right)}{\partial \boldsymbol{\lambda}_W}$$

(Nous notons $\boldsymbol{\lambda}_W$ la sortie du GNN). En utilisant ce gradient pour la rétropropagation, on met à jour les poids du GNN dans la direction qui maximise la borne inférieure, en appliquant la règle de la chaîne :

$$\frac{\partial LB_{\mathcal{L}}\left(G(V,E), \boldsymbol{\lambda}_W\right)}{\partial W} = \frac{\partial LB_{\mathcal{L}}\left(G(V,E), \boldsymbol{\lambda}_W\right)}{\partial \boldsymbol{\lambda}_W} \times \frac{\partial \boldsymbol{\lambda}_W}{\partial W}$$

Le terme de droite est connu, car les GNN n'utilisent que des opérations différentiables. Il nous reste donc à calculer le gradient de la borne inférieure par rapport aux multiplicateurs de Lagrange. En pratique, ce gradient est inaccessible et ne peut être utilisé pour l'entrainement. Par contre, on peut calculer facilement un gradient qui n'est valide que localement. En effet, si on considère le cas pour une assignation donnée $\boldsymbol{x}^*$, on obtient :

$$LB_{\mathcal{L}}\left(G(V,E), \boldsymbol{\lambda}_W\right) = f_{\mathcal{L}}(\boldsymbol{x}^*_{\mathcal{L}}, \boldsymbol{\lambda}_W)$$

Et on peut maintenant trouver un gradient local de la borne inférieure :

$$\begin{aligned}
\frac{\partial LB_{\mathcal{L}}\left(G(V,E), \boldsymbol{\lambda}_W\right)}{\partial \boldsymbol{\lambda}_W} &= \frac{\partial f_{\mathcal{L}}(\boldsymbol{x}^*_{\mathcal{L}}, \boldsymbol{\lambda}_W)}{\partial \boldsymbol{\lambda}_W} \\
&= \frac{\partial}{\partial \boldsymbol{\lambda}_W}\left[f(\boldsymbol{x}^*_{\mathcal{L}}) + \boldsymbol{\lambda}_W^T(d - C(\boldsymbol{x}^*_{\mathcal{L}}))\right] \\
&= d - C(\boldsymbol{x}^*_{\mathcal{L}})
\end{aligned}$$

Cette expression permet de mettre à jour les poids du GNN. Il est important de noter qu'elle

n'est valide que localement. En effet, si les multiplicateurs de Lagrange subissent une variation trop importante, alors $\boldsymbol{x}^*_{\mathcal{L}}$ n'est plus la solution optimale de la relaxation.

Cette expression peut être expliquée logiquement. En effet, si le terme $d_i - C_i(\boldsymbol{x}^*_{\mathcal{L}})$ est positif, et donc ne satisfait pas la contrainte qui a été relaxée, alors le retour donné au modèle est d'augmenter la pénalité (le multiplicateur) associée à cette contrainte. A l'inverse, si elle est satisfaite, alors le gradient va dans la direction d'une réduction du multiplicateur associé.

On a bien un apprentissage auto-supervisé avec gradient, car on ne connait pas les valeurs optimales, et on utilise un gradient pour diriger le GNN. Bien entendu, ce type d'apprentissage peut être combiné avec des méthodes classiques pour améliorer l'apprentissage profond, telles que l'apprentissage par lot et l'optimiseur Adam.

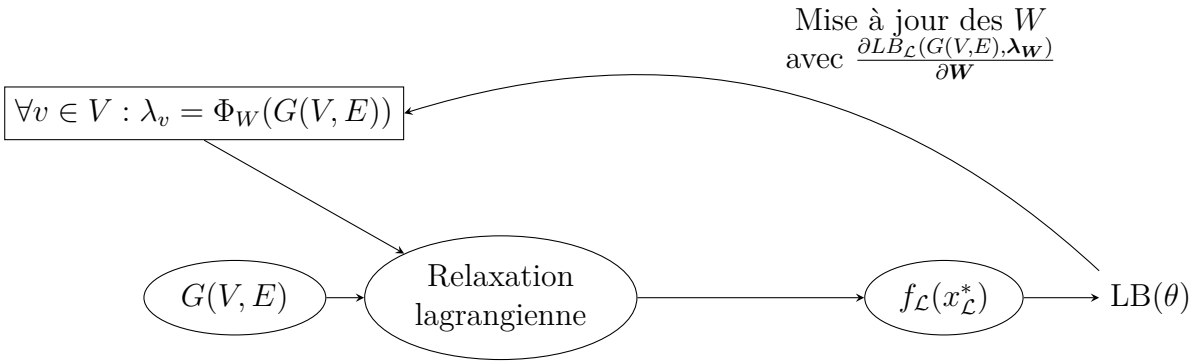

FIGURE 3.2 Entrainement du GNN. Schéma adapté de [1]

Dans la figure 3.2, on peut voir la boucle d'entrainement : le GNN prédit les multiplicateurs, la borne correspondante est calculée, puis le gradient obtenu est utilisé pour la rétropropagation.

Algorithm 2 Boucle d'entrainement du GNN

▷ **Param :** *nEpoch* est le nombre d'époques, *lr* est le taux d'apprentissage
▷ $\mathcal{L}(\lambda)$ retourne $\boldsymbol{x}^*_{\mathcal{L}}$ la solution au problème relaxé
▷ Φ_w un GNN de poids w initialisés aléatoirement

for e **from** 1 **to** *nEpoch* **do**
  for $G(V,E)$ **in** ensemble d'instances d'entrainement **do**
    $\boldsymbol{\lambda} \leftarrow \Phi_w(G(V,E))$
    $\boldsymbol{x}^*_{\mathcal{L}} \leftarrow \mathcal{L}(\boldsymbol{\lambda})$
    $\frac{\partial LB_{\mathcal{L}}(G(V,E),\boldsymbol{\lambda})}{\partial \boldsymbol{\lambda}} \leftarrow C(\boldsymbol{x}^*_{\mathcal{L}}) - d$
    backpropagate $\frac{\partial LB_{\mathcal{L}}(G(V,E),\boldsymbol{\lambda})}{\partial \boldsymbol{\lambda}}$ **to update** w
  end for
end for

L'algorithme 2 présente la boucle d'entrainement du GNN. C'est un entrainement classique.

Le seul point remarquable est que nous avons inversé le signe du gradient. En effet, les algorithmes d'optimisation de réseaux de neurones visent toujours à minimiser la fonction de perte. C'est pourquoi ils utilisent des variantes de descente de gradient.

Dans notre cas, nous souhaitons maximiser la borne inférieure. Ainsi, dans l'algorithme, nous utilisons $-\nabla$ pour la descente de gradient pour mettre à jour le réseau de neurones. De ce fait, la descente de gradient maximise la borne inférieure prédite par le GNN.

Nous pouvons maintenant clairement voir pourquoi maximiser la LB n'est pas strictement équivalent à maximiser le filtrage. En effet, pour filtrer, il faut que :

$$LB + cr > UB$$

Or dans notre cas, nous maximisons la LB, mais nous ne tenons pas compte des coûts réduits, qui sont pourtant dépendants du choix de multiplicateurs. Notre entrainement ne fournit donc pas exactement un modèle qui maximise le filtrage. Néanmoins en pratique, la maximisation de la borne inférieure donne tout de même de bons résultats.

3.4 Architecture du réseau de neurones

Dans le chapitre précédent, nous avons introduit les GNN ainsi que l'architecture GraphSage. Cependant, celle-ci ne prend pas en compte les éventuelles caractéristiques sur les arêtes du graphe.

Pour plusieurs contraintes globales, il est également nécessaire d'encoder l'information sur les arêtes du graphe. Nous utilisons donc une astuce similaire à ce qui avait été proposé dans [43]. Nous agrégeons également les informations sur les arêtes d'un nœud, puis nous les ajoutons à celles obtenues des voisins du nœud. Le message reçu est donc toujours

$$\boldsymbol{x}_i^{k\prime} = \frac{1}{|\mathcal{N}(i)|} \sum_{j \in \mathcal{N}(i)} \left[\sigma^k \left(\boldsymbol{W}_1^k \cdot \boldsymbol{x}_j^{k-1} + \boldsymbol{b}^k \right) \right]$$

En outre, on calcule une moyenne sur les arêtes adjacentes au nœud, avec E l'ensemble des arêtes du graphe et $\boldsymbol{E}_{uv} \in \mathbb{R}^{n_e}$ les n_e caractéristiques de l'arête de u à v :

$$e_i^k = \frac{1}{|\{v : (i,v) \in E\}|} \sum_{(i,v) \in E} [\boldsymbol{E}_{iv}]$$

Et on va introduire une nouvelle matrice de poids $\boldsymbol{W}_4^k \in \mathbb{R}^{h^k \times h^{n_e}}$. La mise à jour de la représentation du nœud i s'écrit alors :

$$\boldsymbol{x}_i^k = \sigma\left(\boldsymbol{W}_3^k \boldsymbol{x}_i^{k-1} + \boldsymbol{W}_2^k \boldsymbol{x}_i^{k\prime} + \boldsymbol{W}_4^k \boldsymbol{e}_i^k\right)$$

Cette formule combine les informations du nœud lui-même, de ses voisins et des arêtes adjacentes pour produire la nouvelle représentation du nœud.

3.5 Construction de l'ensemble d'entrainement

Maintenant que nous avons couvert les explications plus théoriques derrière le fonctionnement de notre approche, il reste à définir l'ensemble d'entrainement utilisé pour le GNN. Cet ensemble contiendra différentes instances d'un problème intégrant la contrainte globale que l'on souhaite améliorer. En pratique le seul nœud où un solveur de CP va filtrer sur une instance complète est à la racine. Cependant, nous souhaitons pouvoir appeler notre modèle dans l'arbre de recherche, et continuer à prédire des multiplicateurs de Lagrange qui donnent lieu à des LB adaptées.

Dans l'arbre de recherche, notre modèle ne recevra plus le problème complet, mais des sous-problèmes dont certaines valeurs ont déjà été retirées des domaines par les branchements et filtrages successifs. Afin de préparer le modèle à ces cas de figure, nous allons aussi joindre des instances de ces sous-problèmes à l'ensemble d'entrainement. Pour obtenir ces sous-problèmes, nous exécutons la résolution d'un problème complet, puis enregistrons certains sous-problèmes formés lors de la recherche. Ce choix rend le GNN plus robuste et améliore les performances, car les cas d'entrainement sont ainsi du même type que ceux rencontrés lors de la phase d'inférence.

CHAPITRE 4 APPLICATION SUR DES CONTRAINTES GLOBALES

Afin d'évaluer la méthodologie que nous proposons ici, nous l'appliquons à plusieurs problèmes dans un solveur général de programmation par contraintes. Nous avons utilisé le solveur par contraintes Choco [4] version 4.10.17 et implémenté l'approche en Java. L'implémentation du GNN sera présentée ultérieurement.

Trois contraintes globales ont été envisagées, la contrainte "circuit pondéré" (*weightedCircuit*), la contrainte "Au Plus N Valeurs" (*atMostNValue*) et la contrainte "Au Plus un Poids W" (*atMostWValue*). Nous présentons maintenant ces contraintes, les problèmes associés ainsi que les relaxations choisies.

4.1 Contrainte *weightedCircuit*

La contrainte *weightedCircuit* [44] est une contrainte bien connue, qui sert surtout à la modélisation des problèmes du voyageur de commerce. Elle peut être vue comme une combinaison de deux contraintes :

- La contrainte "*circuit*", qui impose qu'un certain trajet forme un circuit hamiltonien dans le graphe, passant exactement une fois par chaque nœud.
- Une contrainte de poids, qui encode le coût du circuit.

Celle-ci prend deux arguments, un graphe $G(V, E_w)$ (où E_w désigne l'ensemble des arêtes pondérées), et une variable de coût z, qui est le coût du circuit.

Commençons par formaliser la contrainte *weightedCircuit*. Il existe différentes formulations possibles selon la variable de décision choisie, mais dans notre cas nous optons pour une variable binaire x_e indiquant si l'arête e appartient à la solution. De plus, chaque arête e est associée à son coût, c_e. On peut maintenant réécrire cette contrainte comme une combinaison de contraintes plus simples :

Dans ce qui suit, $\delta(v)$ est l'ensemble des voisins d'un nœud. La contrainte 4.1 va stocker le coût du circuit dans la variable de coût z. La seconde contrainte, 4.2 vérifie que chaque nœud est connecté à exactement deux arêtes sélectionnées. La contrainte 4.3 sélectionne exactement $|V|$ arêtes. La dernière contrainte (4.4) élimine les sous-tournées, garantissant que le seul circuit formé inclut tous les nœuds.

$$\text{CircuitPondere}(G(V,E),z): \quad \sum_{e\in E} c_e x_e = z \tag{4.1}$$

$$\sum_{e\in\delta(v)} x_e = 2 \quad \forall v \in V \tag{4.2}$$

$$\sum_{e\in E} x_e = |V| \tag{4.3}$$

$$\sum_{e\in E(S)} x_e \leq |S| - 1 \quad \forall S \subset V,\ 2 \leq |S| \leq |V| - 1 \tag{4.4}$$

4.1.1 Problème du voyageur de commerce

En pratique, cette contrainte permet entre autres de modéliser le problème du voyageur de commerce (TSP). Dans ce problème, on dispose d'une liste de villes, ainsi que les distances entre elles. Le but est d'établir un parcours de longueur minimale qui passe par toutes les villes une seule fois et revient au point de départ. On remarque que cela correspond exactement à la contrainte *weightedCircuit*, en minimisant le coût total z :

$$\begin{aligned} TSP: \min_{x_e\in\{0,1\}} & \ z \\ s.t. \quad & \sum_{e\in E} c_e x_e = z \\ & \sum_{e\in\delta(v)} x_e = 2 \quad \forall v \in V \\ & \sum_{e\in E} x_e = |V| \\ & \sum_{e\in E(S)} x_e \leq |S| - 1 \quad \forall S \subset V,\ 2 \leq |S| \leq |V| - 1 \end{aligned}$$

Il convient maintenant de présenter la relaxation lagrangienne utilisée pour le filtrage par le coût. Dans le cadre du TSP, une des relaxations généralement utilisées est celle proposée par Held et Karp dans [45] et [7]. Ils proposent d'appliquer la relaxation lagrangienne à la contrainte 4.2, pour tous les nœuds excepté le premier. Cette relaxation est utilisée dans le solveur de l'état de l'art Concorde [2]. Ceci donne le problème relaxé suivant (problème 4.5) où HK est le problème relaxé selon la méthode introduite par Held et Karp.

Pour le premier nœud, rien n'a changé, mais pour les $|V| - 1$ autres nœuds, on a relaxé la contrainte sur le degré. Ainsi, $|V| - 1$ termes de pénalité et $|V| - 1$ multiplicateurs ont été

$$
\begin{aligned}
HK(\theta): \min_{x\in\{0,1\}} \quad & \sum_{e\in E} c_e x_e - \sum_{v\in V\backslash\{1\}} \theta_v (2 - \sum_{e\in\delta(v)} x_e) \\
\text{s.t.} \quad & \sum_{e\in\delta(1)} x_e = 2 \\
& \sum_{e\in E} x_e = |V| \\
& \sum_{e\in E(S)} x_e \leq |S| - 1 \quad \forall S \subset V\backslash\{1\},\ 2 \leq |S| \leq |V| - 1 \\
& \theta_v \in \mathbb{R} \quad \forall v \in V\backslash\{1\}
\end{aligned}
\tag{4.5}
$$

introduits.

Il est important de noter que les contraintes relaxées étaient des égalités. On ne se situe donc plus dans le cadre standard des multiplicateurs de Lagrange, car les multiplicateurs ont maintenant pour domaine $\mathbb{R}$. C'est pour différencier ces multiplicateurs de ceux de Lagrange qu'ils sont notés θ_v.

On peut aussi remarquer que le terme de pénalité a été soustrait de la fonction objectif. Ce choix est arbitraire, car les multiplicateurs n'étant plus seulement positifs ce signe ne change rien (en pratique ce choix permet la transformation en 1-arbre qui va suivre). Chaque nœud (sauf le premier) est associé à un multiplicateur, car la contrainte associée a été relaxée.

On peut maintenant reformuler l'objectif relaxé pour retrouver une formulation plus pratique pour la suite. Premièrement, on va ajouter une pénalité associée au sommet 1, qui vaut toujours 0. L'objectif peut être réécrit de la manière suivante puisque ce terme est multiplié par 0 :

$$
\min_{x\in\{0,1\}} \quad \sum_{e\in E} c_e x_e - \sum_{v\in V\backslash\{1\}} \theta_v (2 - \sum_{e\in\delta(v)} x_e) + \theta_1 (2 - \sum_{e\in\delta(1)} x_e)
$$

En distribuant les sommes, et en réarrangeant les termes, on trouve :

$$
\min_{x\in\{0,1\}} \quad \sum_{e\in E} c_e x_e + \sum_{v\in V} \left(\sum_{e\in\delta(v)} \theta_v x_e \right) - 2 \sum_{v\in V} \theta_v
$$

Dans la double somme, chaque arête $e = (u, v)$ apparait deux fois, une fois avec le multiplicateur de u et une fois avec v. En effet, on parcourt chaque nœud, et on considère toutes les arêtes incidentes. Ainsi chaque arête est comptée deux fois. L'objectif devient :

$$
\min_{x\in\{0,1\}} \quad \sum_{e\in E} c_e x_e + \sum_{e=(u,v)\in E} ((\theta_u + \theta_v) x_e) - 2 \sum_{v\in V} \theta_v
$$

En regroupant les deux premières sommes, on obtient la reformulation suivante du problème :

$$
\begin{aligned}
HK(\theta): \min_{x\in\{0,1\}} \quad & \sum_{e=(u,v)\in E} (c_e + \theta_u + \theta_v)x_e - 2\sum_{v\in V}\theta_v \\
\text{s.t.} \quad & \sum_{e\in\delta(1)} x_e = 2 \\
& \sum_{e\in E} x_e = |V| \\
& \sum_{e\in E(S)} x_e \leq |S| - 1 \quad \forall S \subset V\backslash\{1\},\ 2 \leq |S| \leq |V| - 1 \\
& \theta_1 = 0 \\
& \theta_v \in \mathbb{R} \quad \forall v \in V\backslash\{1\}
\end{aligned}
\tag{4.6}
$$

On peut dès lors remarquer que les solutions au problème relaxé sont des 1-arbres. Un 1-arbre est obtenu dans un graphe en construisant un arbre couvrant pour tous les nœuds $v \geq 2$. On connecte ensuite le nœud 1 au reste de l'arbre par deux arêtes.

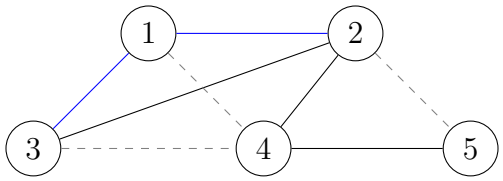

FIGURE 4.1 Exemple de 1-arbre

Dans le schéma 4.1, on voit un exemple de 1-arbre. Les arêtes noires forment l'arbre couvrant pour les nœuds 2 à 5 et en bleu les 2 arêtes qui ont été ajoutées pour connecter le premier nœud. On pourrait bien entendu choisir un autre sommet que le premier pour le 1-arbre, mais comme la numérotation des nœuds est arbitraire, on garde le premier par convention, sans perte de généralité.

Pour en revenir à notre problème, dans l'équation 4.6, cela revient à chercher un 1-arbre de coût minimal (MST), dans un graphe modifié. Dans ce graphe, les arêtes sont de coût $c_{uv} + \theta_u + \theta_v$. Ce problème peut être résolu avec la même complexité que la recherche d'un arbre couvrant de poids minimum, qui est polynomiale.

Ainsi, pour des multiplicateurs fixés, il nous suffit de calculer un 1-arbre afin d'obtenir une borne inférieure sur le coût du TSP. De plus, comme dit précédemment, puisque cette borne provient d'une relaxation Lagrangienne, elle est toujours valide.

4.1.2 Entrainement du GNN

Après avoir choisi notre relaxation Lagrangienne pour la contrainte *weightedCircuit*, on peut calculer le gradient qui est utilisé afin de mettre à jour les poids du réseau de neurones. En repartant de l'expression trouvée précédemment pour le gradient à maximiser :

$$d - C(x)$$

En tenant compte de la contrainte que l'on relaxe ici, on trouve :

$$\frac{\partial HK(\theta)}{\partial \theta_i} = 2 - \sum_{v \in V} [x_{iv}] = 2 - deg(x_i)$$

Pour rappel, ce gradient n'est pas celui que l'on passe à l'optimiseur mais l'inverse car l'optimiseur va faire une descente de gradient. Ce gradient doit être maximisé. Après avoir résolu le 1-arbre, il suffit de calculer le degré de chaque sommet dans la solution et on obtient la dérivée nécessaire à la mise à jour des poids.

Lorsque le degré d'un nœud est 2, le gradient est nul car on ne souhaite pas modifier ce multiplicateur puisque le sommet respecte déjà la contrainte. A l'inverse, si un sommet est d'un degré inférieur à 2, ce gradient est positif puisqu'on veut renforcer la pénalité, et négatif dans le cas contraire. Le gradient pour la descente de gradient est donc :

$$deg(x_i) - 2$$

4.1.3 Problème du voyageur de commerce avec fenêtre de temps

Un des objectifs d'intégrer l'approche dans un solveur général est de pouvoir résoudre des problèmes avec d'autres contraintes, tout en bénéficiant d'une amélioration du filtrage par le coût.

Pour la contrainte *weightedCircuit*, nous considérons la variante du TSP qui est le problème du voyageur de commerce avec fenêtre de temps (TSPTW). Le problème consiste toujours à trouver le circuit de coût minimal passant une et une seule fois par chaque ville, mais cette fois chaque ville n'est accessible que pendant un certain intervalle de temps. La distance entre les villes est interprétée comme un temps de trajet.

Pour le formaliser, nous allons ajouter quelques paramètres :

1. $Open_v \forall\ v \in V$: Le moment d'ouverture de chaque ville
2. $Close_v \forall\ v \in V$: Le moment de fermeture de chaque ville

À partir de la contrainte *weightedCircuit*, nous calculons une variable $succ_v \ \forall v \in V$ qui enregistre le successeur de chaque ville, à savoir la ville qui est visitée après celle-ci.

De plus, nous devons ajouter une variable qui encode le moment auquel nous arrivons à chaque ville, $arrival_v \ \forall v \in V$ et le temps d'attente éventuel avant que la ville n'ouvre $wait_v \ \forall v \in V$ (il faut attendre le temps nécessaire jusqu'à l'ouverture, i.e., $Open_v - arrival_v$).

Formalisons le TSPTW :

$$\min_{G,succ,arrival,wait,z} z \tag{4.7}$$

$$s.t. weightedCircuit(G, z) \rightarrow succ \tag{4.8}$$

$$arrival_v \leq Close_v \qquad \forall v \in V \tag{4.9}$$

$$arrival_v + wait_v \geq Open_v \qquad \forall v \in V \tag{4.10}$$

$$arrival_{succ_v} = arrival_v + wait_v + c_{v,succ_v} \qquad \forall v \in V \setminus \{1\} \tag{4.11}$$

$$arrival_{succ_1} = c_{1,succ_1} \tag{4.12}$$

$$wait_v \geq 0 \qquad \forall v \in V \tag{4.13}$$

Quant à l'entrainement et la dérivée, c'est là que réside la force de notre approche. En effet, comme cela a déjà été fait pour la contrainte *weightedCircuit*, il n'y a plus de travail supplémentaire pour d'autres problèmes utilisant la contrainte.

4.2 Contrainte *atMostNValue*

Nous allons maintenant nous intéresser à une seconde contrainte couramment utilisée, dite "*Au Plus N Différentes Valeurs*" (*atMostNValue*) aussi introduite dans [44]. Elle est majoritairement utilisée dans des problèmes liés à la planification ou au placement de facilités. Cette contrainte prend en entrée un ensemble de variables $\{X_1, ..., X_n\}$ avec $X_i \in D(X_i) \qquad \forall i \in \{1, ..., n\}$ (avec $D(X_i)$ le domaine de la variable X_i) et un entier N. Elle se note :

$$atMostNValue(\{X_1, ..., X_n\}, N)$$

Cette contrainte vérifie que les variables de l'ensemble prennent au plus N différentes valeurs. Afin de la formaliser, nous utilisons le programme linéaire proposé par Bessière et al. [46] que Berthiaume et Quimper [12] ont également utilisé pour leur travail sur la contrainte *atMostNValue*. On définit $m \in \mathbb{N}$ tel que $D(X_i) \subseteq \{1, ..., m\}$ pour tout i, sans perte de généralité.

On peut dès lors écrire le CSP associé à cette contrainte comme :

$$
\begin{aligned}
CSP - atMostNValue(X, N) : & \sum_{j=1}^{m} y_j \leq N \\
s.t. \quad & \sum_{j \in D(X_i)} y_j \geq 1 \qquad \forall X_i \in X \\
& \mathbf{y} \in \{0,1\}^m
\end{aligned}
$$

y_j est une variable binaire indiquant le fait qu'une certaine valeur est utilisée au moins une fois. Le programme linéaire proposé consiste à relaxer ces variables booléennes :

$$
\begin{aligned}
LP : \min_{\mathbf{y}} & \sum_{j=1}^{m} y_j && (4.14) \\
s.t. \quad & \sum_{j \in D(X_i)} y_j \geq 1 \qquad \forall X_i \in X && (4.15) \\
& \mathbf{0} \leq \mathbf{y} && (4.16) \\
& \mathbf{y} \leq \mathbf{1} && (4.17)
\end{aligned}
$$

On considère désormais des vecteurs de variables réelles comprises entre 0 et 1 comme variables de décision. Afin d'appliquer notre méthodologie, il est nécessaire de partir d'une relaxation lagrangienne. Nous utilisons celle proposée par Cambazard et Fages [47], qui consiste à relaxer la contrainte 4.15 :

$$
\begin{aligned}
LR : \min_{\mathbf{y}} & \sum_{j=1}^{m} y_j + \sum_{i=1}^{n} \lambda_i \left(1 - \sum_{j \in D(X_i)} y_j\right) \\
s.t. \quad & \mathbf{0} \leq \mathbf{y} \\
& \mathbf{y} \leq \mathbf{1} \\
& \boldsymbol{\lambda} \geq \mathbf{0}
\end{aligned}
$$

Les multiplicateurs λ sont positifs afin de les distinguer des multiplicateurs θ de *weighted-Circuit*, qui peuvent être négatifs. On reste ici dans le cadre standard des multiplicateurs de Lagrange, car les contraintes relaxées sont des inégalités.

4.2.1 Problème de couverture par un ensemble de lieux

Comme problème lié à cette contrainte, nous utilisons le problème de couverture par un ensemble de lieux (*Location Set Covering Problem* (LSCP)). Ce problème a été introduit en 1971 par Toregas et al. [48], dans le contexte des services d'urgence.

Il consiste à choisir parmi un ensemble de lieux candidats ceux où l'on va construire des hôpitaux, de façon à ce que chaque citoyen ait au moins un hôpital proche de chez lui. L'objectif dans ce problème est d'ouvrir le moins d'hôpitaux possible. On peut exprimer ce problème en utilisant la contrainte *atMostNValue* :

$$\begin{aligned} \min_{\boldsymbol{X},N} \quad & N \\ s.t. \quad & atMostNValue(\{X_1, ..., X_n\}, N) \\ & X_i \in D(X_i) \qquad \forall i \in \{1, ..., n\} \\ & N \in \mathbb{N} \end{aligned}$$

N est le nombre d'hôpitaux à ouvrir, les X_i représentent les différents citoyens, et les valeurs possibles pour chaque X_i correspondent aux emplacements d'hôpitaux accessibles depuis ce citoyen. En résolvant ce problème d'optimisation, on obtient le nombre minimal N à construire qui satisfait tous les citoyens.

4.2.2 Entrainement du GNN

Comme précédemment, après avoir choisi la relaxation pour notre contrainte, nous pouvons calculer le gradient utilisé dans le réseau de neurones. Nous notons ici les multiplicateurs de Lagrange λ, car il s'agit de contraintes d'inégalité. On peut repartir de la valeur trouvée précédemment :

$$\frac{\partial LR(\boldsymbol{\lambda})}{\partial \lambda} = d - C(x)$$

Et on va donc obtenir :

$$\frac{\partial LR(\boldsymbol{\lambda})}{\partial \lambda_i} = 1 - \sum_{j \in D(X_i)} y_j$$

De même que pour *weightedCircuit*, cette formule est assez intuitive. En effet, pour rappel y_j est la variable binaire qui indique si la valeur j est prise par au moins une des variables X_i. La dérivée partielle est positive si aucune valeur du domaine de X_i n'a été sélectionnée (ce qui incite le modèle à en choisir une), nulle si exactement une valeur est prise, et négative si

plus d'une valeur est utilisée (car la contrainte est sur-satisfaite, la pénalité peut alors être réduite). À nouveau, c'est le gradient inverse qui est utilisé pour la rétropropagation.

4.2.3 Problème p-médian

Comme pour le TSP, nous considérons également un problème plus complet, qui inclut d'autres contraintes que *atMostNValue*. Il s'agit du problème p-médian qui a été utilisé et formalisé dans [12]. On a un ensemble de clients $\{X_1, ..., X_n\}$ qui peuvent chacun être servis par certains fournisseurs : X_i est la variable représentant le fournisseur assigné au client i, avec $X_i \in D(X_i)$, et ce service est associé à un coût $c_{i,j}$ qui dépend du client et du fournisseur (la distance de livraison par exemple). L'objectif est de servir tous les clients à coût minimal, tout en limitant le nombre de fournisseurs utilisés à N. On modélise ce problème :

$$\min_{X,C} \sum_{i=1}^{n} C_i \tag{4.18}$$

$$s.t. \quad atMostNValue(\mathbf{X}, N) \tag{4.19}$$

$$Element(X_i, \{c_{i,1}, c_{i,2}, ..., c_{i,m}\}, C_i) \quad \forall i \in \{1, ..., n\} \tag{4.20}$$

La contrainte *Element* 4.20 impose que $C_i = c_{i,X_i}$. Le modèle reste identique à celui du problème simple, comme on utilise la même contrainte globale.

4.3 Contrainte *atMostWValue*

Cette contrainte est une variante pondérée de $atMostNValue$. La différence est un paramètre de poids supplémentaire associé à chaque valeur des domaines. Ainsi, la limite sur le nombre de valeurs choisies N devient une contrainte sur le poids total W. Elle est formulée comme suit :

$$atMostWValue(\{X_1, ..., X_n\}, W, \{w_1, ..., w_m\})$$

En utilisant la même procédure que pour *atMostNValue*, on peut donc trouver un programme

linéaire très similaire, où seul l'objectif est modifié par rapport à 4.14–4.17 :

$$LP : \min_{\mathbf{y}} \sum_{j=1}^{m} (w_j y_j) \tag{4.21}$$

$$s.t. \quad \sum_{j \in D(X_i)} y_j \geq 1 \qquad \forall X_i \in X \tag{4.22}$$

$$\mathbf{0} \leq \mathbf{y} \tag{4.23}$$

$$\mathbf{y} \leq \mathbf{1} \tag{4.24}$$

La relaxation utilisée est la même, et on trouve :

$$LR(\lambda) : \min_{\mathbf{y}} \sum_{j=1}^{m} (w_j y_j) + \sum_{i=1}^{n} \lambda_i \left(1 - \sum_{j \in D(X_i)} y_j\right)$$

$$s.t. \quad \mathbf{0} \leq \mathbf{y}$$

$$\mathbf{y} \leq \mathbf{1}$$

$$\boldsymbol{\lambda} \geq \mathbf{0}$$

Le sous-gradient utilisé pour l'optimisation est donc exactement le même, car la partie de l'objectif qui a été modifiée est constante selon λ et on a toujours :

$$\frac{\partial LR(\boldsymbol{\lambda})}{\partial \lambda_i} = 1 - \sum_{j \in D(X_i)} y_j$$

4.3.1 LSCP pondéré

Pour le problème simple, on peut reprendre la version pondérée de celui utilisé pour *atMostN-Value*. Il s'illustre toujours à l'aide du contexte des services d'urgence, avec une variation : On dispose désormais d'un budget total pour la construction des centres d'urgence, et selon les localisations choisies, le coût de construction varie. Il faut donc trouver un ensemble de lieux qui minimise le coût, tout en assurant la couverture de tous les citoyens.

On trouve la formalisation suivante :

$$\begin{aligned} &\min_{\boldsymbol{X}, W} \quad W \\ &s.t. \quad atMostWValue(\{X_1, ..., X_n\}, W, \{w_1, ..., w_m\}) \\ &\qquad X_i \in D(X_i) \qquad \forall i \in \{1, ..., n\} \\ &\qquad W \in \mathbb{R}^+ \end{aligned}$$

4.3.2 Problème de localisation d'entrepôts non capacitaires

Afin d'évaluer notre approche dans un contexte plus complet, nous considérons également un problème utilisant d'autres contraintes. Nous utilisons ici le problème de localisation d'entrepôts non capacitaires (*Uncapacitated facility location problem* (UFLP)). Dans ce problème, on dispose d'un budget W pour ouvrir des entrepôts parmi un ensemble de localisations possibles. L'objectif est de servir tous les clients en minimisant le coût de service et celui d'ouverture des entrepôts. Les variables sont :

- $\{X_1, ..., X_n\}$: $X_i = j$ signifie que le client i est servi par l'entrepôt j.
- $A \in \{0, 1\}^{n,m}$: $A_{i,j} = 1$ si le client i est servi par j, 0 autrement.
- $\{C_1, ..., C_n\}$: Le coût de service de chaque client.

Les paramètres sont :

- W : Le budget maximum.
- $C_{i,j}$: Le coût pour le client i d'être servi par j.
- $\{w_1, ..., w_m\}$: Le coût de construction de l'entrepôt j.

En formalisant le problème, on obtient :

$$UFLP : \min_{X,A,C} \sum_{i=1}^{n} C_i + \sum_{j=1}^{m} \left(\max_i (A_{i,j}) w_j \right) \tag{4.25}$$

$$s.t. \quad atMostWValue(\{X_1, ..., X_n\}, W, \{w_1, ..., w_m\}) \tag{4.26}$$

$$A_{i,j} \leftarrow reify(X_i = j) \qquad \forall i \in \{1, ..., n\} \quad \forall j \in \{1, ..., m\} \tag{4.27}$$

$$Element(X_i, \{c_{i,1}, ..., c_{i,m}\}, C_i) \qquad \forall i \in \{1, ..., n\} \tag{4.28}$$

La contrainte *reify* 4.27 assure que $A_{i,j} = 1$ si et seulement si $X_i = j$ et 0 autrement.

On a maintenant introduit les 3 contraintes globales utilisées pour illustrer notre approche, ainsi que les problèmes dans lesquels elles seront évaluées.

4.4 Implémentation

Comme mentionné précédemment, nous avons décidé de travailler avec le solveur Choco [4], car ce solveur contenait déjà des implémentations pour le filtrage par le coût, ce qui permettait de se consacrer uniquement à l'ajout de l'apprentissage. Nous avons donc choisi et entrainé un modèle pour chaque contrainte. Les GNN ont été implémentés avec la libraire JGNN (Java Graph Neural Networks) [49], qui était la seule librairie permettant l'implémentation de GNN en java. Elle n'est pas encore finalisée et ne permet pas l'utilisation d'un GPU. En

conséquence, les modèles sont assez lents et peuvent prendre jusqu'à 20 millisecondes pour une prédiction. Cependant, nous avons reproduit les mêmes modèles avec PyTorch et en les exécutant sur GPU, le temps de prédiction n'excédait pas 1 ms.

Concernant l'usage du modèle, étant donné le coût d'une prédiction il n'était pas faisable de l'utiliser en chaque nœud de l'arbre de recherche. Nous avons donc gardé la méthode introduite dans [1], à savoir d'appeler le modèle sur les 10 premiers nœuds de l'arbre de recherche.

4.4.1 *weightedCircuit*

La relaxation de Held et Karp est déjà implémentée dans Choco, en suivant l'algorithme 3. Les pénalités initiales sont souvent héritées du nœud précédent de l'arbre de recherche, ou initialisées à 0. Dans notre cas, nous appelons le modèle dès l'initialisation.

Ensuite, comme mentionné précédemment, on résout successivement la relaxation (le 1-arbre) en résolvant un MST puis en ajoutant deux arêtes. À chaque étape, nous tentons de filtrer des arêtes, grâce à la solution de la relaxation et les coûts réduits. Les itérations de sous-gradient utilisent un pas de Newton qui est contrôlé par deux paramètres, I et S. Dans l'implémentation initiale de Choco, ils valent 5 et 30.

Le modèle GNN pour *weightedCircuit* comporte trois couches GraphSAGE, suivies de deux couches entièrement connectées. La taille des couches cachées est toujours de 32 neurones. Ce choix suit la configuration du papier initial.

4.4.2 *atMostNValue* et *atMostWValue*

Un propagateur utilisant la relaxation lagrangienne pour chacune de ces contraintes est déjà implémenté dans [12] pour Choco. Nous avons utilisé cette implémentation, dont les itérations de sous-gradient sont dans l'algorithme 4. Il est très similaire à celui de *weightedCircuit*, seul le calcul du pas diffère légèrement et bien entendu la relaxation n'est pas la même.

Pour ces deux contraintes, puisque les instances sont moins complexes que pour le TSP, nous avons utilisé une architecture comportant seulement deux couches GraphSAGE avant la partie dense, identique à celle précédente. La taille des couches cachées a aussi été réduite à 16 afin d'accélérer les prédictions.

Algorithm 3 Itérations de sous-gradient implémentées dans Choco

▷ **Param :** $(I, S) = (5, 30)$.
▷ **Pre :** G est une instance de TSP sous forme d'un graphe.
▷ **Pre :** UB est une borne supérieure connue sur la solution.
▷ **Pre :** θ_0 est la liste des pénalités.
▷ **Pre :** LB_0 est une borne inférieure sur la solution, calculée par la relaxation lagrangienne avec θ_0.

$LB_{stock} \leftarrow 0$
$LB \leftarrow LB_0$
$\theta \leftarrow \theta_0$
$G' \leftarrow G + CoutPenalise(G, \theta_0)$
while $LB_{stock} < LB$ **do**
  $LB_{stock} \leftarrow LB$
  $\alpha, \beta = 2, 0.5$
  for i **from** 1 **to** I **do**
    for s **from** 1 **to** S **do**
      $mst = CalculeMST(G, \theta)$
      $LB_{HK} = Cout(mst) - 2 \cdot \sum_{\theta_i \in \theta} \theta_i$
      $G \leftarrow G + FiltreArete(G, LB_{HK}, \theta, mst)$
      if $LB_{HK} > LB$ **then**
        $LB \leftarrow LB_{HK}$
      end if
      $\partial\theta \leftarrow 2 - deg_{mst}(\theta)$
      $\theta \leftarrow \theta + PasDeNewton(\alpha, \beta, UB) \cdot \partial\theta$
      $G' \leftarrow G' + CoutPenalise(G, \theta)$
    end for
    $\alpha = \alpha \cdot \beta$
    $\beta = \beta/2$
  end for
end while
return θ_n

4.5 Discussion des limitations du modèle

Bien que l'emploi des GNN permette de capturer la complexité des graphes représentant les instances, plusieurs difficultés ont été rencontrées lors de l'entraînement des modèles. En effet, malgré l'utilisation de techniques d'optimisation telles que le traitement par lots ou l'optimiseur Adam, l'entrainement présentait une instabilité notable, avec d'importantes oscillations de la fonction de perte, même avec des taux d'apprentissage très faibles. Si cela n'affecte pas la performance du modèle ni le fonctionnement global de l'approche, cela complique son implémentation et rend l'obtention de résultats stables plus difficile.

Algorithm 4 Itérations de sous-gradient implémentées dans [12]

▷ **Param :** $(\mu_0, m) = (5, 10)$.
▷ **Pre : y** de taille m contenant y_j de la relaxation
▷ **Pre :** UB une borne supérieure sur la solution
▷ **Pre :** λ_0 une liste de pénalités initiales.
▷ **Pre :** G le graphe représentant l'instance

$LB \leftarrow 0$
$\lambda \leftarrow \lambda_0$
$iter \leftarrow 0$
while $iter < 1000$ **do**
  if $LB \leq UB$ **then**
    $G \leftarrow G + FiltreArete(G, LB, UB, y, \lambda)$
  end if
  $y, LB = Solve(G, \lambda)$
  $CoutReduit = CoutReduit(y, LB)$
  $\mu = \mu_0 / 2^{\lfloor iter/m \rfloor}$
  $\lambda \leftarrow \lambda + \mu * SousGradient(y)$
  Si le pas est trop petit ou $UB < LB$ *alors :* ***break***
  $iter \leftarrow iter + 1$
end while
return λ

4.6 Résumé

En conclusion, la méthodologie générale peut être appliquée relativement facilement à différentes contraintes. De plus, au sein d'une même contrainte, aucune adaptation n'est requise pour traiter différents problèmes, ce qui constitue l'un des principaux atouts de cette approche.

Enfin, les choix d'architecture ont principalement été hérités du papier initial et n'ont pas été optimisés. En raison des longs temps d'entraînement, il n'a pas été possible de réaliser une étude approfondie des différentes architectures possibles.

CHAPITRE 5 RÉSULTATS EXPÉRIMENTAUX ET ANALYSE

Notre implémentation est disponible sur GitHub [1] afin de faciliter la reproductibilité de nos résultats.

5.1 *weightedCircuit*

Comme nous souhaitons proposer une méthode indépendante du problème, le modèle n'utilise que des caractéristiques données en entrée de la contrainte. Pour *weightedCircuit*, nous avons donc choisi les caractéristiques suivantes sur les nœuds du graphe :

- La distance au nœud le plus proche.
- La distance moyenne aux autres nœuds du graphe.
- Le degré du nœud dans la solution du 1-arbre.
- Un indicateur binaire : si ce nœud est la racine.

Pour les arêtes :

- La longueur de l'arête.
- Un indicateur binaire : si l'arête a été filtrée par le solveur.
- Un indicateur binaire : si l'arête a été marquée comme appartenant à la solution par le solveur.

5.1.1 Entrainement du modèle

Afin d'entrainer le réseau de neurones, nous avons choisi d'utiliser des instances aléatoires afin d'obtenir un modèle plus général ne reposant pas sur une structure récurrente des données. Nous avons généré 100 instances de TSP, où les villes sont distribuées aléatoirement sur le plan $[0, 100]^2$. Afin d'obtenir l'ensemble d'instances partiellement résolues, on a appliqué la méthode introduite en section 3.5, en générant 10 instances partiellement résolues par instance. Elles correspondent aux 10 premiers nœuds de l'arbre de recherche, afin que l'ensemble d'entrainement reflète au mieux l'utilisation prévue lors de la validation.

5.1.2 Problème du voyageur de commerce

Comme mentionné précédemment, pour la contrainte *weightedCircuit*, nous avons choisi le problème du voyageur de commerce, car il fait exclusivement appel à cette contrainte.

1. `https://github.com/corail-research/learning-cost-based-filtering`

Processus expérimental

Les instances utilisées pour entrainer le modèle contiennent 100 villes.

Pour rappel l'implémentation initiale dans Choco utilisait comme paramètres pour les itérations de sous-gradients ($I = 5, S = 30$) (voir algorithme 3). Cependant en pratique ces paramètres conduisent à un très grand nombre d'itérations à chaque nœud (au moins 150). Par conséquent, les multiplicateurs utilisés fournissent des bornes très proches de la valeur optimale, avec un écart inférieur à 1% dans la majorité des cas. En pratique, il est possible d'obtenir un filtrage presque aussi bon en réduisant ce nombre d'itérations, ce qui se traduit par une accélération du temps de résolution. En ce qui concerne la borne supérieure, nous avons suivi la même approche que l'article de référence, avec une borne à 2% de la valeur optimale.

Nous considérons ici trois cas possibles lors du traitement d'un nœud :

- $Iter+$: Nous utilisons les valeurs par défaut de Choco ($I = 5, S = 30$).
- $Iter-$: Nous utilisons des valeurs réduites afin de contrôler le nombre d'itérations ($I = 2, S = 10$). De plus, la boucle *while* de l'algorithme 3 n'est exécutée qu'une fois.
- GNN : Nous appelons le modèle sur ce nœud pour initialiser Θ dans l'algorithme par la prédiction.

Les configurations sont notées

$$(\text{paramètres pour les 10 premiers nœuds}, \text{paramètres pour les nœuds restants})$$

Cela correspond aux paramètres utilisés lors de la résolution des 100 instances de test, et permet de voir l'impact du modèle sur les premiers nœuds, mais aussi de voir si simplement faire plus d'itérations de sous-gradient au début a un impact similaire à l'utilisation du modèle.

Les différentes configurations expérimentales sont donc :

- **(Iter+, Iter+)** : Implémentation initiale de Choco.
- **(Iter-, Iter-)** : Accélération la procédure du sous-gradient, au prix d'une réduction du filtrage.
- **(Iter+, Iter-)** : Plus d'itérations sur les 10 premiers nœuds. Cette configuration permet d'observer si augmenter le nombre d'itérations de sous-gradient au début produit un effet similaire à l'utilisation du modèle.
- **(GNN & Iter-, Iter-)** : Un des cas d'usage du modèle, en réduisant les itérations durant toute la résolution.
- **(GNN & Iter+, Iter-)** : Le second cas d'usage du modèle, mais avec la nuance

que nous exécutons plus d'itérations après avoir initialisé les multiplicateurs avec le modèle.

Comparaison avec l'article initial

Avant de passer aux résultats, il est important de noter certaines différences avec l'article qui a inspiré ce mémoire [1].

- La référence : Comme le solveur utilisé n'est pas le même, il est naturel de s'attendre à une amélioration différente. De plus, Choco est bien meilleur que le précédent solveur pour calculer les multiplicateurs de Lagrange. Sur des instances aléatoires, Choco atteint une borne inférieure à plus de 98% de la valeur optimale dès la racine, alors que les bornes du solveur de l'article étaient entre 90% et 95% de l'optimum. Ce point fait une grande différence car un bon filtrage sur les premiers nœuds de l'arbre a une grande influence sur le reste de la résolution. Dans l'article, une partie de l'accélération provenait donc d'une augmentation de la borne inférieure. Dans notre cas, cette augmentation n'a jamais été observée car la borne inférieure est vraisemblablement déjà très proche de la valeur optimale (et parfois même égale à la solution optimale). La seule amélioration possible réside donc dans l'accélération du calcul de cette borne, ce qui mène à des gains de temps plus modestes.
- Les caractéristiques : Précédemment, le modèle utilisait également des caractéristiques du problème (les coordonnées des différents nœuds dans le plan). Cependant, dans l'objectif de rendre l'approche généralisable, il n'est plus possible de dépendre des caractéristiques spécifiques aux problèmes.

Résultats expérimentaux

Les résultats obtenus pour les cinq configurations, sur des problèmes de TSP, sont présentés dans le tableau 5.1. Le temps alloué était de 30 minutes. Les métriques considérées sont :

- Le nombre de nœuds explorés
- Le nombre d'itérations de sous-gradient effectuées (et donc le nombre de résolutions de la relaxation).
- Le temps de résolution.
- Le temps de résolution sans tenir compte du temps d'inférence. Cette métrique est prise en compte, car comme indiqué précédemment la librairie JGNN utilisée n'est pas encore optimisée et implique des ralentissements. En pratique, en utilisant PyTorch ce temps d'inférence sera réduit.

On peut faire plusieurs observations suite à ce résultat :

	Nœuds	Itérations (k)	Temps de résolution (s)	Temps de résolution sans le temps d'inférence (s)
$(Iter+, Iter+)$	**8155**	2229,85	32,28	32,28
$(Iter-, Iter-)$	14541	483,88	14,83	14,83
$(Iter+, Iter-)$	15028	498,05	15,27	15,27
$(GNN\&Iter-, Iter-)$	14244	478,83	15,32	14,83
$(GNN\&Iter+, Iter-)$	12951	**437,26**	**14,03**	**13,52**

TABLEAU 5.1 Résultats moyens pour un TSP simple avec des instances aléatoires de 100 villes

Premièrement, l'implémentation initiale de Choco est en effet sous-optimale, bien que son filtrage soit bien meilleur, avec au moins 33% de nœuds explorés en moins, le temps passé par nœud est beaucoup plus important comme le montre le nombre élevé d'itérations.

Ensuite, la méthodologie $(GNN\&Iter+, Iter-)$ est plus rapide que les autres (plus de 8% de temps de résolution en moins que la meilleure sans modèle). Cette amélioration est toutefois inférieure à celle rapportée dans l'article (17%), ce qui provient très vraisemblablement du changement de solveur car notre cas de référence est plus performant.

5.1.3 Problème du voyageur de commerce avec fenêtre de temps

Nous résolvons maintenant le problème avec des contraintes additionnelles. Il convient de noter que la taille des instances a dû être réduite à 20-30 villes. En effet, au-delà de ce nombre, aucune instance ne terminait, et pour certaines aucune solution n'était obtenue dans les 30 minutes imparties.

Afin de générer ces instances, la configuration des villes est toujours celle du TSP. Les contraintes de temps ont été générées à l'aide du code de [50], qui permet de ne générer que des instances avec une solution possible. Les paramètres utilisés sont les suivants :

— *max_tw_gap* : La durée maximale des fenêtres d'ouverture des villes constituant le tour faisable.

— *max_tw_size* : toutes les fenêtres d'ouverture des villes sont comprises dans l'intervalle $[0, max\_tw\_size]$.

Nous utilisons ici 3 ensembles de données contenant chacun 100 instances.

— Medium20 : 20 villes, max_tw_gap = 10, max_tw_size = 100

— Medium30 : 30 villes, max_tw_gap = 10, max_tw_size = 100

— Large20 : 20 villes, max_tw_gap = 50, max_tw_size = 300 ce qui entraine une résolution plus lente et augmente les calculs non liés à la contrainte *weightedCircuit*.

Les ensembles d'entrainement et de test sont générés selon la même procédure que précé-

demment. Pour ce cas, nous n'utilisons plus de borne supérieure, car elle est plus difficile à obtenir.

Résultats expérimentaux

Dans un premier temps, nous utilisons un modèle entrainé sur des instances de 30 villes. Les résultats sont présentés dans le tableau 5.2. Les métriques utilisées sont toujours les mêmes avec 2 ajouts :

- La valeur de la solution trouvée à l'issue du temps imparti (la valeur optimale, ou la plus petite solution trouvée si l'instance n'a pas fini son exécution)
- Le nombre d'instances sur les 100 n'ayant pas terminé dans le temps imparti.

Seules les deux dernières mesures (Meilleure valeur trouvée, et nombre de délais expirés (*time-out*)) prennent en compte les instances qui n'ont pas terminé leur résolution, elles n'entrent pas dans le calcul des moyennes pour les autres.

On ne considère plus la paramétrisation initiale de Choco, qui était également bien plus lente sur ce problème, et n'apporte donc rien pour la comparaison.

Medium20	Nœuds (k)	Itérations (M)	Temps de résolution (s)	T. de résolution sans temps d'inférence (s)	Meilleure valeur trouvée	# de délais expiré
$(Iter-, Iter-)$	161,18	3421,30	46,79	46,79	1594,01	4
$(Iter+, Iter-)$	153,91	3260,51	44,36	44,36	1785,98	5
$(GNN\&Iter-, Iter-)$	198,62	4194,96	56,66	56,62	**1403,09**	**3**
$(GNN\&Iter+, Iter-)$	**147,39**	**3145,16**	**44,15**	**44,11**	1594,1	5
Medium30	**Nœuds (k)**	**Itérations (M)**	**Temps de résolution (s)**	**T. de résolution sans temps d'inférence (s)**	**Meilleure valeur trouvée**	**# de délais expiré**
$(Iter-, Iter-)$	586,81	12059,86	282,92	282,92	14936,18	57
$(Iter+, Iter-)$	600,05	12343,87	283,99	283,99	**14368,7**	**55**
$(GNN\&Iter-, Iter-)$	**577,09**	**11846,5**	**272,86**	**272,81**	14651,48	56
$(GNN\&Iter+, Iter-)$	590,28	12151,54	298,19	298,14	14651,48	**55**
Large20	**Nœuds (k)**	**Itérations (M)**	**Temps de résolution (s)**	**T. de résolution sans temps d'inférence (s)**	**Meilleure valeur trouvée**	**# de délais expiré**
$(Iter-, Iter-)$	1054,60	23358,34	331,38	331,38	**7972,86**	58
$(Iter+, Iter-)$	**1035,16**	**22886,2**	**328,48**	**328,48**	8161,76	59
$(GNN\&Iter-, Iter-)$	1071,26	23739,76	344,93	344,89	8159,93	**57**
$(GNN\&Iter+, Iter-)$	1056,97	23425,05	340,41	340,37	8350,25	58

TABLEAU 5.2 Résultats pour le TSPTW avec un modèle entrainé sur des instances de 30 villes

Premièrement, il apparait que les résultats sont cette fois un peu plus mitigés, bien que l'utilisation du modèle soit tout de même bénéfique.

L'utilisation d'un modèle sur des instances de taille similaire à l'ensemble d'entrainement s'avère efficace. En effet, pour l'ensemble Medium20, la version utilisant le modèle avec moins d'itérations a moins d'instances avec un délai expiré, et trouve une meilleure solution en moyenne à l'issue du temps imparti. L'approche combinant plus d'itérations initiales et le modèle permet une réduction modeste du temps d'exécution.

Pour l'ensemble Medium30, une observation similaire se confirme : l'une des approches avec le modèle présente le meilleur temps d'exécution, et le gain est cette fois plus important. Pour les délais expirés, il n'y a pas d'amélioration cette fois.

Enfin, l'ensemble Large20, plus difficile que les précédents, se comporte comme prévu et présente des performances moindres. En effet, bien que l'on ait réussi à réduire le nombre d'instances ne finissant pas, on perd en temps moyen sur les instances résolues.

De ces résultats, nous pouvons conclure que lorsque d'autres contraintes entrent en jeu, notre approche reste pertinente, mais devient moins efficace, en particulier si la complexité du problème ne provient plus de la contrainte *weightedCircuit*.

Généralisation

Nous résolvons maintenant ces ensembles de données en utilisant le modèle entrainé pour les problèmes de simple TSP. L'intérêt ici est d'observer ce qui se passe lorsque la taille des instances diffère fortement entre entrainement et application. En effet, ce modèle avait été entrainé sur des instances de 100 villes, et on l'utilise maintenant sur des instances de 20 ou 30 villes. Les résultats sont dans le tableau 5.3.

De nouveau, l'usage du modèle semble améliorer les performances. Pour Medium20, les meilleurs temps de résolutions et solutions finales correspondent à l'usage du modèle (ou sont les mêmes). Pour Medium30 l'usage du modèle ralentit un peu le temps d'exécution, mais a permis de résoudre 3 instances de plus dans le temps imparti. Enfin pour Large20, les résultats sont cette fois en faveur de notre approche car l'usage du modèle est au moins aussi bon que l'implémentation initiale, ou l'améliore légèrement.

Il ressort que même si des changements importants de taille des instances réduisent les performances, notre approche permet tout de même d'améliorer les résolutions. Ces résultats pourraient être améliorés en optimisant le choix des paramètres.

Medium20	Nœuds (k)	Itérations (M)	Temps de résolution (s)	T. de résolution sans temps d'inférence (s)	Meilleure valeur trouvée	# de délais expiré
$(Iter-, Iter-)$	161,18	3421,30	44,61	44,61	**1594,01**	**4**
$(Iter+, Iter-)$	153,91	3260,51	41,75	41,75	1785,98	5
$(GNN\&Iter-, Iter-)$	**149,75**	**3167,31**	**40,38**	**40,34**	**1594,01**	**4**
$(GNN\&Iter+, Iter-)$	170,85	3618,70	46,59	46,55	**1594,01**	**4**
Medium30	Nœuds (k)	Itérations (M)	Temps de résolution (s)	T. de résolution sans temps d'inférence (s)	Meilleure valeur trouvée	# de délais expiré
$(Iter-, Iter-)$	**586,81**	**12059,8**	**272,44**	**272,44**	14936,18	57
$(Iter+, Iter-)$	600,05	12343,87	309,54	309,54	**14369,2**	57
$(GNN\&Iter-, Iter-)$	597,51	12270,23	282,12	282,07	14651,48	**54**
$(GNN\&Iter+, Iter-)$	586,86	12072,82	275,41	275,34	15222,18	56
Large20	Nœuds (k)	Itérations (M)	Temps de résolution (s)	T. de résolution sans temps d'inférence (s)	Meilleure valeur trouvée	# de délais expiré
$(Iter-, Iter-)$	1111,71	24589,03	343,34	343,34	7972,75	**57**
$(Iter+, Iter-)$	1133,52	25036,77	344,94	344,94	8162,11	59
$(GNN\&Iter-, Iter-)$	1161,80	25695,79	351,86	351,82	7780,84	**57**
$(GNN\&Iter+, Iter-)$	**1033,53**	**22916,6**	**327,37**	**327,33**	**7590,74**	60

TABLEAU 5.3 Résultats pour le TSPTW avec un modèle entrainé sur des instances de 100 villes

5.2 *atMostNValue*

Pour cette contrainte, le choix de caractéristiques est assez similaire. Pour les nœuds qui représentent des variables (*les citoyens* ou *les clients*), on utilise 2 caractéristiques :

- Un indicateur binaire : Si parmi le domaine de cette variable, au moins une valeur est déjà marquée comme obligatoire (c'est-à-dire qu'elle fait partie de toute solution).
- La taille du domaine restant pour la variable (sans les valeurs déjà filtrées).

Pour les valeurs (*les hôpitaux* ou *les entrepôts*), trois caractéristiques sont considérées :

- Un indicateur binaire si la valeur est marquée comme obligatoire.
- Un indicateur binaire si la variable a été filtrée.
- Le nombre de variables ayant cette valeur dans leur domaine.

5.2.1 Problème de couverture par un ensemble de lieux

Les instances considérées pour cette partie contiennent 100 clients et 100 entrepôts, et chaque client est connecté aléatoirement à N entrepôts.

$$N \sim \mathcal{N}(9, 3)$$

Les ensembles d'entrainement sont générés comme expliqué précédemment, et contiennent toujours 100 instances initiales et 1000 instances partiellement résolues.

Il est à noter que la librairie JGNN ne permet pas les graphes contenant différents types de nœuds, nous avons dû manuellement implémenter les couches de transmission de message, ce qui entraine un temps d'inférence plus long. De plus, comme notre approche n'est pas encore pleinement intégrée au solveur, il faut également récupérer les caractéristiques des nœuds.

Résultats expérimentaux

On travaille cette fois avec l'algorithme 4. Comme précédemment, nous avons considéré différents paramètres pour les itérations de sous-gradients, contrôlé ici par μ_0 et m. À nouveau, les 10 premiers nœuds peuvent être traités différemment du reste.

L'implémentation initiale ne sera mentionnée que lorsqu'elle donne des résultats intéressants. En effet, comme pour *weightedCircuit* elle est nettement moins performante dans la plupart des cas. Elle utilisait $\mu_0 = 5, m = 10$ partout.

Dans notre cas, nous utilisons $m = 5$ pour tous les nœuds. Cette valeur a été choisie car elle maximise les performances du solveur sans modèle, ce qui permet de l'évaluer en le comparant à une référence performante.

Dans le tableau 5.4 nous comparons 2 approches :

- Référence : L'implémentation de référence avec $m = 5$
- GNN : L'implémentation avec $m = 5$ et l'usage du GNN sur les 10 premiers nœuds pour initialiser les multiplicateurs de Lagrange.

	Nœuds	Itérations (k)	Temps de résolution (ms)	Temps de résolution sans le temps d'inférence (ms)
Référence	590.90	132.60	**777.42**	777.42
GNN	**552.60**	**126.93**	2009.41	**723.91**

Tableau 5.4 Résolution d'un LSCP avec ou sans modèle, pour des instances de 100 variables avec 100 valeurs possibles.

On peut voir que l'usage du modèle améliore les performances sauf pour le temps de résolution brut. En effet, comme mentionné, l'inférence est ici particulièrement lente, et le problème étant plutôt simple et donc résolu rapidement, cela a un lourd impact. Comme expliqué plus tôt, ce point n'est pas particulièrement un problème car une fois une librairie efficace utilisée et après avoir complètement intégré le modèle dans le solveur, ce temps d'inférence sera drastiquement réduit.

5.2.2 Problème p-médian

On peut maintenant passer au problème contenant des contraintes additionnelles. Les instances utilisées sont toujours les mêmes, on a simplement ajouté des coûts sur chaque assignation variable valeur en les générant aléatoirement dans $[0, 4]$. L'objectif en n'utilisant pas de problèmes euclidiens (où le problème peut être représenté graphiquement) est d'éviter la présence de structure sur laquelle le modèle peut se reposer. De cette façon, on l'évalue dans un contexte plus complexe.

Les paramétrisations du solveur considérées sont toujours les mêmes, avec un délai d'expiration à 1800 secondes. On remet ici les résultats du solveur initial (appelé Choco, avec $\mu_0 = 5 \& m = 10$) car ils sont maintenant suffisamment proches du reste pour être pertinents. On peut voir les résultats obtenus dans le tableau 5.5.

Notre approche est systématiquement meilleure (sauf sur le nombre de nœuds, qui naturellement est plus faible avec l'implémentation de Choco), mais de nouveau les gains sur des problèmes ayant plus de contraintes sont assez modérés. De plus, ici l'objectif n'était plus lié à la contrainte de couverture, et il est possible que le gain soit plus faible car la complexité du problème ne provient plus uniquement de cette contrainte.

Instances complexes

L'idée est maintenant d'observer ce qu'il se passe sur des instances plus complexes. Celles utilisées ici proviennent de [12], et sont particulièrement difficiles à résoudre par le filtrage basé sur le coût. On a donc évalué nos 3 solveurs dans le tableau 5.6.

On peut voir que pour ce type d'instance, non seulement notre approche ralentit l'exécution mais l'implémentation initiale est meilleure que la référence. On peut donc conclure que bien que pour des cas quelconques, utiliser le modèle permet d'améliorer les résultats, dans les cas limites où le filtrage par le coût est inefficace, notre approche ne permet pas de pallier cette difficulté.

	Nœuds (k)	Itérations (M)	Temps de résolution (s)	T. de résolution sans temps d'inférence (s)	Meilleure valeur trouvée	# de délais expiré
Choco	**324,09**	101,73	560,45	560,45	110,31	50
Référence	473,35	83,36	512,25	512,25	110,24	50
GNN	443,54	**79,53**	**510,38**	**508,04**	**110,20**	50

Tableau 5.5 Résultats pour le problème p-médian sur des instances aléatoires

	Nœuds (k)	Itérations (M)	Temps de résolution (s)	T. de résolution sans temps d'inférence (s)	Meilleure valeur trouvée	# de délais expiré
Choco	**169, 93**	55,69	**363,13**	**363,13**	**127,63**	33
Référence	294,47	51,58	445,00	445,00	128,04	33
GNN	290,53	**50,94**	463,76	461,42	128,00	33

TABLEAU 5.6 Résultats sur des instances de p-médian difficiles pour le filtrage par le coût.

5.3 *atMostWValue*

Intéressons-nous à la contrainte *atMostWValue*. Du point de vue des caractéristiques utilisées au modèle, la seule différence est l'ajout d'une caractéristique aux valeurs, à savoir le coût w_i associé. L'architecture du modèle est la même que pour *atMostNValue*.

5.3.1 LSCP pondéré

Les instances utilisées pour ce problème sont générées avec 150 variables et valeurs, puis chaque variable est reliée aléatoirement à N valeurs avec

$$N \sim \mathcal{N}(10, 3)$$

Pour le coût des valeurs, il est proportionnel au nombre de fois que cette valeur apparait dans le domaine d'une variable, avec un bruit gaussien. Les itérations de sous-gradients sont identiques à celles utilisées pour la contrainte précédente, et nous adoptons les mêmes configurations. Nous avons choisi d'utiliser une borne supérieure pour ce problème, car les instances n'étaient pas résolues dans le temps imparti sans borne primale. Nous avons choisi d'utiliser la valeur optimale comme borne supérieure, car dans [1], cela réduisait l'efficacité de l'approche. Nous évaluons ainsi notre méthodologie dans des conditions moins favorables. Dans le tableau 5.7 on peut voir la comparaison entre la référence et le modèle

Une fois encore, sur un problème contenant uniquement la contrainte améliorée par le modèle,

	Nœuds (k)	Itérations (M)	Temps de résolution (s)	T. de résolution sans temps d'inférence (s)
Référence	52,38	18,96	210,16	210,16
GNN	**49,82**	**18,03**	**202,26**	**196,95**

TABLEAU 5.7 Résultats pour le LSCP pondéré

l'impact de l'approche est positif en tout point.

5.3.2 Problème de localisation d'entrepôts non capacitaires

Nous considérons à présent un problème plus complexe. Les instances sont toujours générées avec la même méthode, on va maintenant associer un coût aléatoire à chaque paire de variable-valeur. Dans un premier temps, ce coût est choisi parmi $\{0, 1, 2\}$ Afin de limiter les temps d'exécution, on a réduit la taille des instances à 50 variables et 50 valeurs.

Le modèle permet à nouveau d'accélérer la résolution. Ce résultat change lorsque les coûts des paires variable-valeur sont générés aléatoirement entre 0 et 5, comme on peut le voir dans le tableau 5.9

Il apparait que cette fois le résultat semble très mitigé, avec le modèle qui ajoute une instance qui ne se termine pas, et ne gagne pas de temps sur celle qui se finissent. Comme lors du problème p-médian, lorsque la partie complexe n'est plus la contrainte que l'on tente d'améliorer, l'utilisation du modèle n'a plus d'impact ou encore un impact négatif.

5.4 Discussion des résultats

Dans l'ensemble, il apparait que lorsque la contrainte améliorée est au cœur du problème, notre approche permet d'améliorer les performances du solveur de manière fiable et significative. De plus, on a pu vérifier que les performances restent stables malgré des variations importantes de la taille des instances.

Lorsque d'autres contraintes entrent en jeu, les performances deviennent plus mitigées. Cependant, le modèle permet souvent d'accélérer la résolution ou de réduire le nombre d'instances non-résolues. Par contre si la contrainte améliorée n'est pas la source principale de complexité du problème, l'utilisation du modèle est inutile.

	Nœuds (k)	Itérations (M)	Temps de résolution (s)	T. de résolution sans temps d'inférence (s)	Meilleure valeur trouvée	# de délais expiré
Référence	846,82	**105,10**	286,77	286,77	529,45	0
GNN	**837,70**	106,51	**272,20**	**271,48**	529,45	0

Tableau 5.8 Résultats pour le UFLP avec des coûts entre 0 et 2

	Nœuds (k)	Itérations (M)	Temps de résolution (s)	T. de résolution sans temps d'inférence (s)	Meilleure valeur trouvée	# de délais expiré
Référence	2455,89	**234,15**	**654,68**	654,68	817,53	**15**
GNN	**2449,06**	235,67	654,83	**654,12**	**817,41**	16

Tableau 5.9 Résultats pour le UFLP avec des coûts entre 0 et 5

CHAPITRE 6 CONCLUSION

6.1 Résumé des travaux

Ce travail présente une approche hybride utilisant l'apprentissage automatique afin d'améliorer les solveurs de programmation par contraintes. Celle-ci permet de prédire les multiplicateurs de Lagrange associés à une contrainte, afin d'accélérer les itérations de sous-gradients utilisées lors de la résolution de problèmes d'optimisation combinatoire.

Comme les résultats l'ont montré, cette approche est indépendante du problème et cible la contrainte à améliorer, sans nécessiter de modifications du solveur ni du reste du modèle. Il reste néanmoins nécessaire d'entrainer le modèle pour chaque nouvelle contrainte ciblée. Cette opération est relativement rapide et ne demande que des instances représentatives. De plus, l'emploi de GNN permet de résoudre des instances de différentes tailles sans nécessiter de réentrainer le modèle ce qui favorise la généralisation.

Enfin, cette méthode a été appliquée sur trois contraintes globales, en considérant également des problèmes comprenant des contraintes additionnelles. En résumé, l'approche proposée est prometteuse et permet d'accélérer la résolution des problèmes d'optimisation combinatoire, malgré certaines limites à prendre en compte. Des améliorations futures sont nécessaires pour maximiser les bénéfices.

6.2 Limitations de la solution proposée

L'approche proposée présente encore plusieurs limitations. Tout d'abord, les améliorations sont pour l'instant modestes et ne sont pas systématiques. Bien que les gains soient intéressants sur des problèmes simples, ils diminuent pour des problèmes plus complexes. De plus, si d'autres contraintes influencent la complexité du problème et que celle-ci ne dépend plus uniquement de la contrainte ciblée, les bénéfices de l'approche sont limités, voire peuvent devenir négatifs.

Enfin, la cible d'entrainement actuellement utilisée, à savoir la hauteur de la borne inférieure, n'est pas exactement ce que l'on souhaite maximiser. L'objectif réel est de maximiser le filtrage obtenu par la contrainte. Il serait donc préférable de travailler directement avec ce signal. Bien que ces deux mesures soient liées, elles ne sont pas parfaitement corrélées, ce qui peut limiter l'efficacité du modèle.

6.3 Améliorations futures

Ce mémoire ouvre plusieurs pistes d'amélioration. Il serait tout d'abord pertinent d'explorer d'autres cibles pour l'entrainement du modèle, afin que l'objectif d'apprentissage corresponde mieux à l'objectif réel. Par ailleurs, les performances pourraient être renforcées en testant d'autres architectures de GNN ou en employant des techniques d'apprentissage plus avancées. Enfin, une intégration directe du modèle dans un solveur, capable d'exploiter pleinement ses structures internes, permettrait de tester l'approche sur des cas pratiques et de mieux évaluer son efficacité dans des scénarios réels.

RÉFÉRENCES

[1] A. Parjadis *et al.*, “Learning Lagrangian Multipliers for the Travelling Salesman Problem,” dans *30th International Conference on Principles and Practice of Constraint Programming (CP 2024)*, ser. Leibniz International Proceedings in Informatics (LIPIcs), P. Shaw, édit., vol. 307. Dagstuhl, Germany : Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2024, p. 22 :1–22 :18. [En ligne]. Disponible : https://drops.dagstuhl.de/entities/document/10.4230/LIPIcs.CP.2024.22

[2] D. L. Applegate *et al.*, *The Traveling Salesman Problem : A Computational Study*. Princeton University Press, 2006. [En ligne]. Disponible : http://www.jstor.org/stable/j.ctt7s8xg

[3] P. Benchimol *et al.*, “Improved filtering for weighted circuit constraints,” *Constraints*, vol. 17, n°. 3, p. 205–233, 2012. [En ligne]. Disponible : https://doi.org/10.1007/s10601-012-9119-x

[4] C. Prud’homme et J.-G. Fages, “Choco-solver : A java library for constraint programming,” *Journal of Open Source Software*, vol. 7, n°. 78, p. 4708, 2022. [En ligne]. Disponible : https://doi.org/10.21105/joss.04708

[5] F. Rossi, P. van Beek et T. Walsh, édit., *Handbook of Constraint Programming*. Elsevier, 2006.

[6] S. Ducomman, H. Cambazard et B. Penz, “Alternative filtering for the weighted circuit constraint : comparing lower bounds for the tsp and solving tsptw,” dans *Proceedings of the Thirtieth AAAI Conference on Artificial Intelligence*, ser. AAAI’16. AAAI Press, 2016, p. 3390–3396.

[7] M. Held et R. M. Karp, “The traveling-salesman problem and minimum spanning trees : Part ii,” *Mathematical Programming*, vol. 1, n°. 1, p. 6–25, 1971. [En ligne]. Disponible : https://doi.org/10.1007/BF01584070

[8] N. Christofides, A. Mingozzi et P. Toth, “Exact algorithms for the vehicle routing problem, based on spanning tree and shortest path relaxations,” *Mathematical Programming*, vol. 20, n°. 1, p. 255–282, 1981. [En ligne]. Disponible : https://doi.org/10.1007/BF01589353

[9] F. Focacci, A. Lodi et M. Milano, “Cost-based domain filtering,” dans *Principles and Practice of Constraint Programming – CP’99*, J. Jaffar, édit. Berlin, Heidelberg : Springer Berlin Heidelberg, 1999, p. 189–203.

[10] G. Claus, H. Cambazard et V. Jost, “Arc-consistency and linear programming duality : an analysis of reduced cost based filtering,” 07 2022.

[11] R. Boudreault et C.-G. Quimper, “Improved cp-based lagrangian relaxation approach with an application to the tsp,” dans *Proceedings of the Thirtieth International Joint Conference on Artificial Intelligence, IJCAI-21*, Z.-H. Zhou, édit. International Joint Conferences on Artificial Intelligence Organization, 8 2021, p. 1374–1380, main Track. [En ligne]. Disponible : https://doi.org/10.24963/ijcai.2021/190

[12] F. Berthiaume et C.-G. Quimper, “Local alterations of the lagrange multipliers for enhancing the filtering of the atmostnvalue constraint,” dans *International Conference on the Integration of Constraint Programming, Artificial Intelligence, and Operations Research*. Springer, 2024, p. 68–83.

[13] M. Sellmann, “Theoretical foundations of cp-based lagrangian relaxation,” dans *Principles and Practice of Constraint Programming – CP 2004*, M. Wallace, édit. Berlin, Heidelberg : Springer Berlin Heidelberg, 2004, p. 634–647.

[14] M. Gori, G. Monfardini et F. Scarselli, “A new model for learning in graph domains,” dans *Proceedings. 2005 IEEE International Joint Conference on Neural Networks, 2005.*, vol. 2, 2005, p. 729–734 vol. 2.

[15] M. Schlichtkrull *et al.*, “Modeling relational data with graph convolutional networks,” dans *The Semantic Web : 15th International Conference, ESWC 2018, Heraklion, Crete, Greece, June 3–7, 2018, Proceedings*. Berlin, Heidelberg : Springer-Verlag, 2018, p. 593–607. [En ligne]. Disponible : https://doi.org/10.1007/978-3-319-93417-4_38

[16] J. Gilmer *et al.*, “Neural message passing for quantum chemistry,” dans *Proceedings of the 34th International Conference on Machine Learning - Volume 70*, ser. ICML’17. JMLR.org, 2017, p. 1263–1272.

[17] W. L. Hamilton, R. Ying et J. Leskovec, “Inductive representation learning on large graphs,” dans *Proceedings of the 31st International Conference on Neural Information Processing Systems*, ser. NIPS’17. Red Hook, NY, USA : Curran Associates Inc., 2017, p. 1025–1035.

[18] Q. Cappart *et al.*, “Combinatorial optimization and reasoning with graph neural networks,” dans *Proceedings of the Thirtieth International Joint Conference on Artificial Intelligence, IJCAI-21*, Z.-H. Zhou, édit. International Joint Conferences on Artificial Intelligence Organization, 8 2021, p. 4348–4355, survey Track. [En ligne]. Disponible : https://doi.org/10.24963/ijcai.2021/595

[19] F. Clautiaux et I. Ljubić, “Last fifty years of integer linear programming : A focus on recent practical advances,” *European Journal of Operational Research*, vol. 324, n°. 3, p.

707–731, 2025. [En ligne]. Disponible : https://www.sciencedirect.com/science/article/pii/S0377221724008877

[20] Gurobi Optimization, LLC, “Gurobi Optimizer Reference Manual,” 2026. [En ligne]. Disponible : https://www.gurobi.com

[21] M. Mankowski et M. Moshkov, “Extensions of dynamic programming for multi-stage combinatorial optimization,” *Theoretical Computer Science*, vol. 844, p. 106–132, 2020. [En ligne]. Disponible : https://www.sciencedirect.com/science/article/pii/S0304397520304618

[22] R. Martín-Santamaría *et al.*, “On the automatic generation of metaheuristic algorithms for combinatorial optimization problems,” *European Journal of Operational Research*, vol. 318, n°. 3, p. 740–751, 2024. [En ligne]. Disponible : https://www.sciencedirect.com/science/article/pii/S0377221724004296

[23] S. Cloutier et C.-G. Quimper, “Cumulative Scheduling with Calendars and Overtime,” dans *30th International Conference on Principles and Practice of Constraint Programming (CP 2024)*, ser. Leibniz International Proceedings in Informatics (LIPIcs), P. Shaw, édit., vol. 307. Dagstuhl, Germany : Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2024, p. 7 :1–7 :16. [En ligne]. Disponible : https://drops.dagstuhl.de/entities/document/10.4230/LIPIcs.CP.2024.7

[24] M. Schmied et J.-C. Régin, “Efficient implementation of the global cardinality constraint with costs,” vol. 307. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2024, p. 27 :1–27 :18. [En ligne]. Disponible : https://drops.dagstuhl.de/entities/document/10.4230/LIPIcs.CP.2024.27

[25] P. Benchimol *et al.*, “Improving the held and karp approach with constraint programming,” dans *Proceedings of the 7th International Conference on Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems*, ser. CPAIOR’10. Berlin, Heidelberg : Springer-Verlag, 2010, p. 40–44. [En ligne]. Disponible : https://doi.org/10.1007/978-3-642-13520-0_6

[26] G. Pesant, C. Quimper et A. Zanarini, “Counting-based search : Branching heuristics for constraint satisfaction problems,” *Journal of Artificial Intelligence Research*, vol. 43, p. 173–210, févr. 2012. [En ligne]. Disponible : http://dx.doi.org/10.1613/jair.3463

[27] P. Han *et al.*, “Integrated optimization of train makeup problem and resource scheduling in railway marshalling yards : A hybrid milp-cp approach with logic-based benders decomposition,” *Transportation Research Part B : Methodological*, vol. 200, p. 103306, 2025. [En ligne]. Disponible : https://www.sciencedirect.com/science/article/pii/S0191261525001559

[28] P. Iosif *et al.*, “A CP/LS Heuristic Method for Maxmin and Minmax Location Problems with Distance Constraints,” dans *30th International Conference on Principles and Practice of Constraint Programming (CP 2024)*, ser. Leibniz International Proceedings in Informatics (LIPIcs), P. Shaw, édit., vol. 307. Dagstuhl, Germany : Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2024, p. 14 :1–14 :21. [En ligne]. Disponible : https://drops.dagstuhl.de/entities/document/10.4230/LIPIcs.CP.2024.14

[29] Y. Bengio, A. Lodi et A. Prouvost, “Machine learning for combinatorial optimization : A methodological tour d’horizon,” *European Journal of Operational Research*, vol. 290, n°. 2, p. 405–421, 2021. [En ligne]. Disponible : https://www.sciencedirect.com/science/article/pii/S0377221720306895

[30] P. Tassel, M. Gebser et K. Schekotihin, “An end-to-end reinforcement learning approach for job-shop scheduling problems based on constraint programming,” *Proceedings of the International Conference on Automated Planning and Scheduling*, vol. 33, p. 614–622, 07 2023.

[31] I. Echeverria, M. Murua et R. Santana, “Leveraging constraint programming in a deep learning approach for dynamically solving the flexible job-shop scheduling problem,” *Expert Systems with Applications*, vol. 265, p. 125895, 2025. [En ligne]. Disponible : https://www.sciencedirect.com/science/article/pii/S0957417424027623

[32] I. Bleukx *et al.*, “Model-based algorithm configuration with adaptive capping and prior distributions,” dans *Integration of Constraint Programming, Artificial Intelligence, and Operations Research*, P. Schaus, édit. Cham : Springer International Publishing, 2022, p. 64–73.

[33] W. Shi *et al.*, “Constraintllm : A neuro-symbolic framework for industrial-level constraint programming,” dans *Proceedings of the 2025 Conference on Empirical Methods in Natural Language Processing*. Association for Computational Linguistics, 2025, p. 16010–16030. [En ligne]. Disponible : http://dx.doi.org/10.18653/v1/2025.emnlp-main.809

[34] W. Song *et al.*, “Learning variable ordering heuristics for solving constraint satisfaction problems,” *Engineering Applications of Artificial Intelligence*, vol. 109, mars 2022, publisher Copyright : © 2021 Elsevier Ltd.

[35] T. Marty *et al.*, “Learning and fine-tuning a generic value-selection heuristic inside a constraint programming solver,” *Constraints*, vol. 29, n°. 3, p. 234–260, 2024. [En ligne]. Disponible : https://doi.org/10.1007/s10601-024-09377-4

[36] M.-F. Balcan *et al.*, “Learning to branch,” dans *Proceedings of the 35th International Conference on Machine Learning*, ser. Proceedings of Machine Learning Research, J. Dy et A. Krause, édit., vol. 80. PMLR, 10–15 Jul 2018, p. 344–353. [En ligne]. Disponible : https://proceedings.mlr.press/v80/balcan18a.html

[37] E. Khalil *et al.*, “Learning to branch in mixed integer programming,” *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 30, n°. 1, Feb. 2016. [En ligne]. Disponible : https://ojs.aaai.org/index.php/AAAI/article/view/10080

[38] Y. Shen *et al.*, “Adaptive solution prediction for combinatorial optimization,” *European Journal of Operational Research*, vol. 309, n°. 3, p. 1392–1408, 2023. [En ligne]. Disponible : https://www.sciencedirect.com/science/article/pii/S0377221723000838

[39] S. Bessa *et al.*, “Learning valid dual bounds in constraint programming : Boosted lagrangian decomposition with self-supervised learning,” *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 39, n°. 11, p. 11 113–11 121, Apr. 2025. [En ligne]. Disponible : https://ojs.aaai.org/index.php/AAAI/article/view/33208

[40] Q. Cappart *et al.*, “Improving optimization bounds using machine learning : Decision diagrams meet deep reinforcement learning,” *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 33, p. 1443–1451, 07 2019.

[41] Y. Zhu *et al.*, “A reinforcement learning embedded surrogate lagrangian relaxation method for fast solving unit commitment problems,” *IEEE Transactions on Power Systems*, vol. 40, n°. 5, p. 3806–3818, 2025.

[42] J. Weiner *et al.*, “Ranking constraint relaxations for mixed integer programs using a machine learning approach,” *EURO Journal on Computational Optimization*, vol. 11, p. 100061, 2023. [En ligne]. Disponible : https://www.sciencedirect.com/science/article/pii/S2192440623000059

[43] W. W. Lo *et al.*, “E-graphsage : A graph neural network based intrusion detection system for iot,” dans *NOMS 2022-2022 IEEE/IFIP Network Operations and Management Symposium*. IEEE, avr. 2022, p. 1–9. [En ligne]. Disponible : http://dx.doi.org/10.1109/NOMS54207.2022.9789878

[44] N. Beldiceanu et É. Contejean, “Introducing global constraints in chip,” *Mathematical and Computer Modelling*, vol. 20, p. 97–123, 1994. [En ligne]. Disponible : https://api.semanticscholar.org/CorpusID:119440837

[45] M. Held et R. M. Karp, “The traveling-salesman problem and minimum spanning trees,” *Operations Research*, vol. 18, n°. 6, p. 1138–1162, 1970. [En ligne]. Disponible : http://www.jstor.org/stable/169411

[46] C. Bessiere *et al.*, “Filtering algorithms for the NValue constraint,” *Constraints*, vol. 11, n°. 4, p. 271–293, 2006.

[47] H. Cambazard et J.-G. Fages, “New filtering for atmostnvalue and its weighted variant : A lagrangian approach,” *Constraints*, vol. 20, n°. 3, p. 362–380, 2015. [En ligne]. Disponible : https://doi.org/10.1007/s10601-015-9191-0

[48] C. Toregas *et al.*, "The location of emergency service facilities," *Operations research*, vol. 19, n°. 6, p. 1363–1373, 1971.

[49] E. Krasanakis, S. Papadopoulos et I. Kompatsiaris, "Jgnn : Graph neural networks on native java," *SoftwareX*, vol. 23, p. 101459, 2023. [En ligne]. Disponible : https://www.sciencedirect.com/science/article/pii/S2352711023001553

[50] Q. Cappart *et al.*, "Combining reinforcement learning and constraint programming for combinatorial optimization," *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 35, p. 3677–3687, 05 2021.