

POLYTECHNIQUE MONTRÉAL
affiliée à l'Université de Montréal

Vers une représentation générique des problèmes combinatoires à des fins d'apprentissage

LÉO BOISVERT
Département de génie informatique et génie logiciel

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

Août 2024

POLYTECHNIQUE MONTRÉAL

affiliée à l'Université de Montréal

Ce mémoire intitulé :

Vers une représentation générique des problèmes combinatoires à des fins d'apprentissage

présenté par **Léo BOISVERT**

en vue de l'obtention du diplôme de *Maîtrise ès sciences appliquées*

a été dûment accepté par le jury d'examen constitué de :

Lama SÉOUD, présidente

Quentin CAPPART, membre et directeur de recherche

Amine MHEDHBI, membre

DÉDICACE

À ma famille et amis

REMERCIEMENTS

Évidemment, ce travail aurait été impossible sans les contributions -de plusieurs natures- d'une multitude de personnes. J'aimerais premièrement remercier mon directeur Quentin Cappart d'avoir cru en moi. Mes études de premier cycle ont été très difficiles et malgré mes résultats passables, il m'a donné une chance et m'a grandement supporté et guidé dans ce projet. Merci infiniment.

Selon le cliché, les études supérieures sont une épreuve psychologique vécue dans l'isolement total, où l'on est incapable de parler de son projet à quiconque, le tout sous pression. Je ne doute pas que ce soit vrai dans certains cas, mais ça ne pourrait pas être plus loin de ma réalité grâce aux formidables collègues et amis qu'ont été, sans ordre particulier, Tom Marty, Louis Popovic, Liliane-Caroline Demers, Daphné Lafleur, Hélène Verhaeghe, Thomas Jacquet, Loïc Grumiaux, Hugo Barral, Max Bourgeat, Gaël Reynal et Amaury Guichard. C'est bien à contrecœur que vous me faites dire que les francophones européens ne sont "pas si pires" ;)

Enfin, je dois remercier mon cercle rapproché, mes indéfectibles hypemen, mes *day ones*, mes fans #1 sans qui rien ne se ferait, et ce, dans la misère. Tout d'abord mes amis du collectif "État profond" : AJRD, Julio Naranja, JJ Slim et Young Piet Oudolf. Et finalement ma famille : maman, papa, Arthur, Henri, Marie et Alix.

RÉSUMÉ

Les dernières années ont vu un intérêt croissant pour l'utilisation d'approches basées sur l'apprentissage pour résoudre des problèmes combinatoires, soit de manière autonome, soit en conjonction avec des algorithmes d'optimisation traditionnels. Dans les deux scénarios, un défi réside dans l'encodage des problèmes combinatoires en une structure compatible avec l'algorithme d'apprentissage donné. De nombreux travaux existants ont proposé des représentations spécifiques aux problèmes, souvent sous la forme d'un graphe, pour tirer parti des avantages des *réseaux neuronaux sur graphes*. Cependant, ces approches manquent de généralité, car la représentation ne peut pas être facilement transférée d'un problème combinatoire à un autre. Bien que des tentatives aient été faites pour combler cette lacune, elles offrent encore seulement une généralité partielle. En réponse à ce défi, ce mémoire plaide en faveur de progrès vers une représentation entièrement générique des problèmes combinatoires pour les approches basées sur l'apprentissage. L'approche que nous proposons consiste à construire un graphe en décomposant toute contrainte d'un problème combinatoire en un arbre syntaxique abstrait et en exprimant les relations (par exemple, une variable impliquée dans une contrainte) à travers les arêtes. De plus, nous introduisons une architecture de réseau neuronal sur graphe capable d'apprendre efficacement à partir de cette représentation. L'outil fourni fonctionne sur des problèmes combinatoires exprimés dans le format XCSP3, un format universel de modélisation de problèmes de programmation par contraintes et largement utilisé dans la communauté scientifique. Notre approche permet de gérer toutes les contraintes disponibles dans la compétition mini-track de 2023. Les résultats expérimentaux sur quatre problèmes combinatoires démontrent que notre architecture atteint des performances comparables à celles des architectures dédiées tout en conservant la généralité.

ABSTRACT

In recent years, there has been a growing interest in using learning-based approaches for solving combinatorial problems, either in an end-to-end manner or in conjunction with traditional optimization algorithms. In both scenarios, the challenge lies in encoding the targeted combinatorial problems into a structure compatible with the learning algorithm. Many existing works have proposed problem-specific representations, often in the form of a graph, to leverage the advantages of *graph neural networks*. However, these approaches lack generality, as the representation cannot be easily transferred from one combinatorial problem to another one. While some attempts have been made to bridge this gap, they still offer a partial generality only. In response to this challenge, this thesis advocates for progress toward a fully generic representation of combinatorial problems for learning-based approaches. The approach we propose involves constructing a graph by breaking down any constraint of a combinatorial problem into an abstract syntax tree and expressing relationships (e.g., a variable involved in a constraint) through the edges. Furthermore, we introduce a graph neural network architecture capable of efficiently learning from this representation. The tool provided operates on combinatorial problems expressed in the XCSP3 format, handling all the constraints available in the 2023 mini-track competition. Experimental results on four combinatorial problems demonstrate that our architecture achieves performance comparable to dedicated architectures while maintaining generality.

TABLE DES MATIÈRES

DÉDICACE	iii
REMERCIEMENTS	iv
RÉSUMÉ	v
ABSTRACT	vi
TABLE DES MATIÈRES	vii
LISTE DES TABLEAUX	x
LISTE DES FIGURES	xi
LISTE DES SIGLES ET ABRÉVIATIONS	xii
CHAPITRE 1 INTRODUCTION	1
1.1 Mise en contexte	1
1.2 Objectifs de recherche et contributions	1
1.3 Plan du mémoire	2
CHAPITRE 2 NOTIONS DE BASE ET REVUE DE LITTÉRATURE	3
2.1 Plan du chapitre	3
2.2 Théorie de la complexité	3
2.2.1 Classe P : résolution polynomiale	4
2.2.2 Classe NP : vérification en temps polynomial	4
2.2.3 Une question centrale : P est-il égal à NP ?	4
2.2.4 Problèmes NP-complets	4
2.2.5 Problèmes NP-difficiles	5
2.3 Problèmes combinatoire et optimisation	5
2.3.1 Modélisation de problèmes combinatoires	5
2.4 Résolution de problèmes combinatoires	7
2.4.1 Méthodes exactes	8
2.4.2 Métaheuristiques	10
2.5 Apprentissage automatique	12
2.5.1 Apprentissage supervisé	12

2.5.2	Réseaux de neurones	12
2.6	Réseaux de neurones sur graphes	14
2.6.1	Intuition et notation	14
2.6.2	Le GNN, étape par étape	15
2.7	Optimisation combinatoire et apprentissage	17
2.7.1	Utilité des méthodes à base d'apprentissage	18
2.7.2	GNNs et optimisation	19
2.7.3	Problèmes de satisfiabilité	19
2.8	Question de recherche	20
CHAPITRE 3 Article 1 : TOWARDS A GENERIC REPRESENTATION OF COMBINA-		
	TORIAL PROBLEMS FOR LEARNING-BASED APPROACHES	21
3.1	Introduction	21
3.2	Encoding Combinatorial Problems as a Graph	22
3.3	Learning from the Encoding with a Graph Neural Network	24
3.4	Experiments	26
3.4.1	Results : Accuracy of the Approaches	27
3.4.2	Analysis : Generalization to Larger Instances	28
3.4.3	Discussion : Limitations and Challenges	29
3.5	Conclusion and Perspectives	29
CHAPITRE 4 DÉTAIL DE LA CONSTRUCTION DES GRAPHS		
		30
4.1	Conventions utilisées dans ce chapitre	30
4.2	Types de noeuds	30
4.2.1	Modèle	30
4.2.2	Valeurs	30
4.2.3	Opérateurs	32
4.2.4	Contraintes	33
4.3	Contraintes supportées	34
4.3.1	Contrainte d'intension binaire	34
4.3.2	Contrainte en extension	35
4.3.3	Contrainte de Somme	39
4.3.4	Contrainte Élément	40
CHAPITRE 5 CONCLUSION		
		42
5.1	Synthèse des travaux	42
5.2	Limitations de la solution proposée et pistes de travaux futurs	42

RÉFÉRENCES	44
----------------------	----

LISTE DES TABLEAUX

Tableau 3.1	Comparative performance of different architectures on various problems. .	28
Tableau 4.1	Contrainte en extension	36
Tableau 4.2	Short table	38

LISTE DES FIGURES

Figure 2.1	Relations entre les classes P, NP, NP-complet et NP-difficile, sous l'hypothèse $P \neq NP$	5
Figure 2.2	Exemple : CSP et coloration de graphe	6
Figure 2.3	Exemple : COP et coloration de graphe	7
Figure 2.4	Recherche avec retour arrière	8
Figure 2.5	Branch and bound	9
Figure 2.6	Principales tâches pour les GNNs	14
Figure 2.7	Propagation de l'information dans un GNN	17
Figure 3.1	Encoding of a combinatorial problem presented as a running example.	23
Figure 3.2	Analysis of the generalization ability on larger instances.	28
Figure 4.1	Types de noeuds	31
Figure 4.2	Sous-graphe variable-valeur	32
Figure 4.3	Sous-graphe variable-valeur	32
Figure 4.4	Sous-graphe Contrainte-Variable-Opérateur pour $(\neg x_1 \vee x_2) \wedge (x_1 \vee \neg x_2)$	33
Figure 4.5	Sous-graphe Variable-Opérateur pour $3x_1$	33
Figure 4.6	Sous-graphe pour $3x_1 \leq 4x_2$	34
Figure 4.7	Sous-graphe générique pour $x + 2 \leq 4$	36
Figure 4.8	Sous-graphes génériques pour la contrainte intension binaire	37
Figure 4.9	Contrainte table	38
Figure 4.10	Contrainte short table	39
Figure 4.11	Sous-graphe générique pour $4x_1 + 3x_2 \leq x_3$	40
Figure 4.12	Sous-graphe générique pour $5x_1 + 2x_2 \leq 3$	40
Figure 4.13	Contrainte élément basique	41
Figure 4.14	Contrainte élément avec indice	41

LISTE DES SIGLES ET ABRÉVIATIONS

CP	Programmation par contraintes
CSP	Problème de satisfaction de contraintes
COP	Problème d'optimisation de contraintes
GNN	Réseau de neurones sur graphe
MILP	Programmation linéaire en nombres entiers
TSP	Problème du voyageur de commerce

CHAPITRE 1 INTRODUCTION

1.1 Mise en contexte

Depuis les débuts de la science informatique, l’optimisation combinatoire a suscité l’intérêt de la communauté scientifique. La multitude de problèmes combinatoires qui se présentent naturellement dans la nature et leur intérêt théorique ont mené à de vastes travaux de recherche. Grâce à des années de projets, il est maintenant possible de résoudre des instances de grande taille de problèmes tels que le voyageur de commerce (TSP) ou celui de la satisfiabilité booléenne (SAT) à l’aide de méthodes exactes [1,2] ou d’heuristiques [3]. Pour résoudre des instances de grande taille, les heuristiques ou méta-heuristiques sont souvent privilégiées, étant donnée leur rapidité pour aboutir à des solutions de qualité. Or, celles-ci sont le fruit d’années d’expérience et sont faites à la main par des chercheuses pour chaque type de problème. Bien que les heuristiques soient performantes, leur caractère artisanal en limite l’applicabilité et la généralité. En effet, pour chaque type de problème, il existe une panoplie d’heuristiques et de métaheuristiques spécialisées [4], ce qui illustre leur caractère propre à chaque type de problème. De plus, les heuristiques et métaheuristiques ne s’améliorent pas avec l’expérience, ce qui serait avantageux dans des contextes industriels où des problèmes similaires sont résolus quotidiennement.

En parallèle de l’optimisation, l’apprentissage profond [5] a connu une ascension fulgurante depuis le début des années 2010. Cette approche de l’intelligence artificielle est basée sur l’apprentissage et s’améliore avec l’expérience. L’apprentissage profond a réalisé d’importants progrès dans des domaines tels que la compréhension du langage [6], la vision par ordinateurs [7] et la capacité de jouer à des jeux vidéo [8], pour ne nommer que ceux-ci. Ces progrès dans de très différents domaines ont amené à penser que l’apprentissage profond pourrait être bénéfique pour l’optimisation combinatoire, grâce à son amélioration par l’expérience [9].

Parmi les architectures d’apprentissage profond, les réseaux de neurones sur graphes (GNNs) se sont avérés prometteurs, dû à leur flexibilité et leur capacité de représenter naturellement les problèmes combinatoires. Malgré ces avantages, leur utilisation dans un cadre d’optimisation combinatoire demeure complexe [10]. De plus, les approches du passé ont privilégié des représentations des problèmes sous forme de graphe qui sont propres au type d’instance en question.

1.2 Objectifs de recherche et contributions

L’objectif de ce travail de recherche est de concevoir une représentation sous forme de graphe capable d’exprimer un problème arbitraire en vue d’utiliser des méthodes à base d’apprentissage.

Pour y arriver, nous proposons une nouvelle méthodologie de représentation de problèmes combinatoires sous forme de graphes. Pour valider la pertinence de cette représentation, nous avons validé les performance de celle-ci sur la version décisionnelle de 4 types de problèmes combinatoires : le problèmes du voyageur de commerce (TSP), de coloration de graphe, du *Knapsack* et de la satisfiabilité booléenne (SAT). Sur ces problèmes, nous arrivons à des performances similaires à celles des approches spécifiques aux problèmes dans la littérature. Pour ce faire, nous avons développé une bibliothèque Python qui supporte toutes les contraintes de la section mini de la compétition XCSP3 [11], un format largement adopté au sein de la communauté de programmation par contraintes. Tout le code est disponible publiquement sur GitHub ¹.

1.3 Plan du mémoire

Ce mémoire est divisé en 5 chapitres, dont l'introduction en cours forme le premier. Le chapitre 2 décrit les notions de base en modélisation de problèmes combinatoires, en apprentissage profond et en réseaux de neurones sur graphes. Ce chapitre couvre aussi les éléments clés de la littérature sur ces différents sujets. Le chapitre 3 est un article qui a été soumis à la conférence CPAIOR. Le chapitre 4 propose une description détaillée de la représentation des problèmes sous forme de graphes et une analyse des résultats qui vont au-delà de celles décrites dans l'article. Finalement, le chapitre 5 conclut ce mémoire en soulignant les contributions et les pistes d'amélioration futures.

1. Le code est disponible au <https://github.com/jardinetsouffleton/Graph-Representation>.

CHAPITRE 2 NOTIONS DE BASE ET REVUE DE LITTÉRATURE

2.1 Plan du chapitre

Ce chapitre vise à poser les bases théoriques essentielles et à parcourir la littérature pertinente pour une compréhension des défis et des solutions dans le domaine de l'optimisation combinatoire et de l'apprentissage automatique. Notre premier objectif est de définir la complexité computationnelle des problèmes combinatoires afin de motiver l'utilité de l'apprentissage. Nous entamons donc par un survol de la théorie de la complexité computationnelle. Celle-ci nous permet de classifier les problèmes informatiques selon leur degré de difficulté en termes de résolution et de vérification. Cette section abordera des concepts clés tels que les classes P, NP, la question de savoir si P égale NP, ainsi que les notions de problèmes NP-complets et NP-difficiles.

Ensuite, nous nous pencherons sur les problèmes combinatoires. Cette section couvrira la modélisation, les problèmes de satisfaction de contraintes (CSP), les problèmes d'optimisation de contraintes (COP) et les stratégies de résolution, incluant les méthodes exactes et les métaheuristiques. L'objectif est de fournir un cadre clair pour la compréhension des défis et des techniques présentement utilisées pour résoudre ces types de problèmes.

La section suivante sera consacrée aux bases de l'apprentissage automatique, posant le cadre nécessaire pour comprendre cette discipline clé. Elle sera suivie d'une description détaillée des réseaux de neurones sur graphes (GNNs), en soulignant leur importance et leur application dans la résolution des problèmes d'optimisation combinatoire.

Enfin, nous conclurons le chapitre en examinant l'intersection entre l'optimisation combinatoire et l'apprentissage automatique, mettant en lumière comment ces deux domaines peuvent se compléter et conduire à des avancées dans la résolution de problèmes.

2.2 Théorie de la complexité

La théorie de la complexité computationnelle est une des pierres angulaires de l'informatique théorique. Celle-ci offre un cadre pour classifier les problèmes informatiques selon leur difficulté de résolution et de vérification. Au cœur de cette théorie se trouvent les classes P et NP, qui représentent des catégories fondamentales de problèmes basées sur le temps requis pour les résoudre et les vérifier.

2.2.1 Classe P : résolution polynomiale

La classe P englobe les problèmes de décision qui peuvent être résolus en temps polynomial par une machine de Turing déterministe. En d'autres termes, pour un problème appartenant à la classe P, il existe un algorithme capable de fournir une réponse (oui ou non) en un temps qui croît polynomialement avec la taille de l'entrée. Cette classe symbolise l'efficacité algorithmique : les problèmes en P sont considérés comme étant "faciles" ou du moins traitables de manière pratique. Des exemples typiques incluent le tri d'une liste, la recherche d'un chemin dans un graphe ou le calcul du plus grand diviseur commun de deux nombres.

2.2.2 Classe NP : vérification en temps polynomial

La classe NP, quant à elle, contient les problèmes pour lesquels une solution proposée peut être vérifiée en temps polynomial par une machine de Turing déterministe. Cette définition met l'accent sur la facilité de vérification des solutions plutôt que sur la facilité de les trouver. Un problème est dans NP si, étant donné une "preuve" ou une solution, il est possible de vérifier l'exactitude de cette solution en temps polynomial. Il est important de noter que tous les problèmes en P sont également en NP, car si un problème peut être résolu rapidement, sa solution peut évidemment être vérifiée rapidement.

2.2.3 Une question centrale : P est-il égal à NP ?

L'une des questions les plus éminentes et non résolues en informatique théorique est de savoir si P est égal à NP. Si $P = NP$, cela signifierait que tous les problèmes qui peuvent être vérifiés rapidement peuvent également être résolus rapidement. Cela aurait d'énormes implications, non seulement en informatique, mais dans de nombreux domaines où des problèmes complexes doivent être résolus. Malgré que cette question soit toujours ouverte, on considère en pratique que $P \neq NP$.

2.2.4 Problèmes NP-complets

Au sein des problèmes NP, les NP-complets sont les problèmes les plus difficiles des problèmes NP. Leur importance réside dans l'hypothèse largement admise que $P \neq NP$, suggérant qu'aucune solution en temps polynomial n'existe pour ces problèmes. Des exemples emblématiques incluent la version décisionnelle du problème du voyageur de commerce (TSP) et celui de la satisfiabilité booléenne (SAT), ce dernier par le théorème de Cook [12].

2.2.5 Problèmes NP-difficiles

Les problèmes NP-difficiles, englobant tous les problèmes NP-complets, se caractérisent par leur résistance à une résolution efficace. Ils sont au moins aussi difficiles que les problèmes NP. Leur complexité intrinsèque ne permet pas, à ce jour, de garantir une solution en temps polynomial pour les pires cas. Cette catégorie englobe une diversité de problèmes, allant au-delà des limites des problèmes strictement NP-complets, incluant même des problèmes qui ne sont pas nécessairement décisionnels.

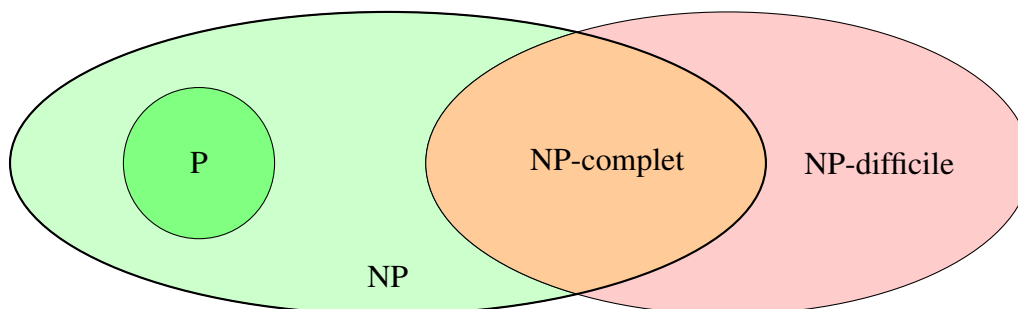


FIGURE 2.1 Relations entre les classes P, NP, NP-complet et NP-difficile, sous l'hypothèse $P \neq NP$

2.3 Problèmes combinatoire et optimisation

L'optimisation combinatoire s'attèle à trouver la meilleure solution parmi un ensemble fini -mais souvent très grand- d'options. Beaucoup des problèmes d'optimisation combinatoire sont NP-complet ou NP-difficile. Cela implique fréquemment de naviguer à travers des espaces de recherche immenses.

2.3.1 Modélisation de problèmes combinatoires

La modélisation de problèmes combinatoires est un aspect fondamental en informatique théorique et en recherche opérationnelle. Elle implique la représentation formelle de problèmes complexes sous une forme qui permet leur analyse et leur résolution algorithmique.

Un modèle de problème combinatoire est composé d'un ensemble fini de variables de décision, chacune pouvant adopter un nombre fini de valeurs. Les contraintes spécifient les valeurs admissibles pour ces variables. Ainsi, ce modèle définit un problème de satisfaction de contraintes (CSP).

Formellement, un CSP est représenté par un triplet $\langle X, D, C \rangle$ où :

- $X = \{x_1, \dots, x_n\}$: un ensemble fini de variables,
- $D = \{v_1, \dots, v_n\}$: un ensemble fini de valeurs,
- $C = \{c_1, \dots, c_n\}$: un ensemble fini de contraintes.

Chaque variable $x \in X$ sélectionne sa valeur dans un sous-ensemble de D , noté $D(x)$, appelé son domaine. Une contrainte c est une relation sur un sous-ensemble de variables qu'on appelle sa portée. Le graphe de cette relation définit les solutions admissibles pour la contrainte. Le problème de satisfaction de contraintes consiste à trouver une assignation pour chaque variable de manière à satisfaire toutes les contraintes, ou à démontrer qu'une telle assignation est impossible. Une assignation qui respecte toutes les contraintes est appelée *solution réalisable*.

Exemple de CSP

Illustrons ceci à l'aide d'un problème concret : le problème de coloration de graphe. Ce problème consiste à attribuer une couleur à chaque sommet d'un graphe $G = (V, E)$ de telle sorte que deux sommets adjacents n'aient pas la même couleur.

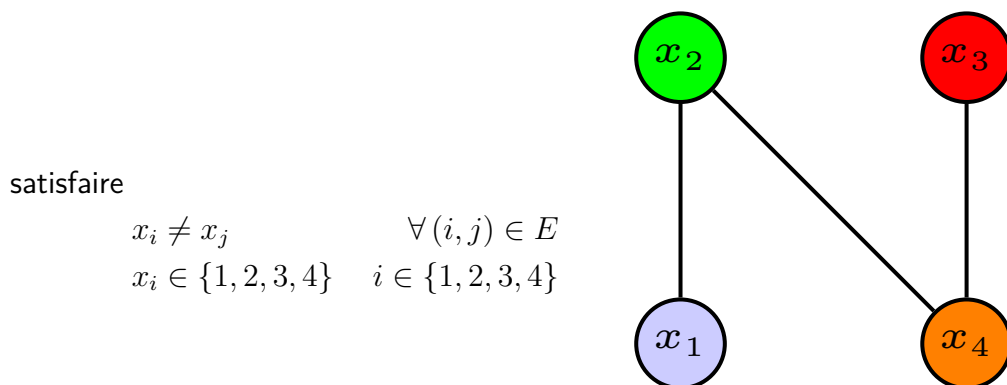


FIGURE 2.2 Exemple : CSP et coloration de graphe

Dans cet exemple, on retrouve un problème à 4 variables, où chacune peut prendre une couleur numérotée de 1 à 4 et où l'on retrouve 3 contraintes : $x_1 \neq x_2$, $x_2 \neq x_4$ et $x_3 \neq x_4$. Le graphe est colorié en respectant les contraintes avec l'assignation $x_1 = 1, x_2 = 2, x_3 = 3, x_4 = 4$.

Problèmes d'optimisation combinatoire

Dans certains cas, en plus de satisfaire aux contraintes d'un problème, il faut optimiser -i.e. maximiser ou minimiser- une quantité donnée. Cette quantité est exprimé par une fonction objectif, qui, quand ajoutée à un CSP, crée un *problème d'optimisation de contraintes* (COP).

Formellement, un COP est représenté par un 4-uplet $\langle X, D, C, f \rangle$ où :

- $\langle X, D, C \rangle$ est un CSP
- $f : D(x_1) \times \dots \times D(x_n) \rightarrow \mathbb{R}$ est une fonction à minimiser sur l'ensemble des solutions réalisables du CSP $\langle X, D, C \rangle$

Exemple de COP

Pour illustrer le concept de COP, revenons à l'exemple précédent. On peut convertir celui-ci en COP en exigeant que le nombre de couleurs utilisées soit minimisé. Ceci peut être modélisé de la façon suivante

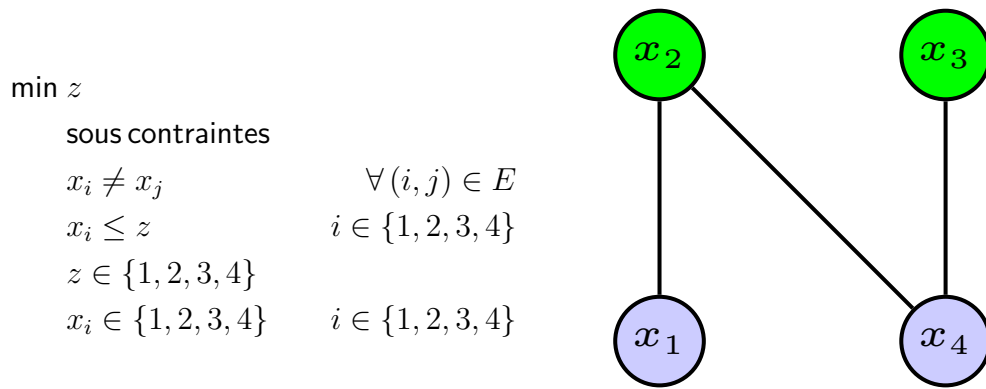


FIGURE 2.3 Exemple : COP et coloration de graphe

Dans cet exemple, on retrouve le même CSP que décrit dans la figure 2.2, auquel on a rajouté une fonction objectif visant à minimiser la valeur de la variable z , celle-ci devant être plus grande ou égale que toutes les valeurs des autres variables utilisées. Le graphe est colorié en respectant les contraintes avec l'assignation $x_1 = 1, x_2 = 2, x_3 = 2, x_4 = 1$. Dans le contexte de ce COP, cette solution est supérieure à celle de l'exemple du CSP puisqu'elle utilise moins de couleurs pour satisfaire aux contraintes.

2.4 Résolution de problèmes combinatoires

Tel que mentionné plus tôt, les problèmes combinatoire peuvent admettre des espaces de recherche énormes. Par exemple, pour le problème de coloration de graphe, si l'on admet k couleurs possibles et que l'on a n variables, il existe k^n solutions possibles. Bien que toutes ne soient pas nécessairement réalisables, il demeure que cette taille de problème requiert des méthodes de solution plus sophistiquées qu'un simple essai-erreur. On peut regrouper les méthodes de solution en deux grandes classes : les méthodes exactes et les heuristiques/méta-heuristiques.

2.4.1 Méthodes exactes

Les méthodes exactes de résolution des problèmes combinatoires visent à trouver une solution optimale ou à prouver qu'aucune solution n'existe. Ces méthodes explorent systématiquement l'espace des solutions possibles en utilisant diverses techniques pour éliminer les chemins non prometteurs et améliorer l'efficacité de la recherche. Nous allons aborder ces méthodes en nous basant sur l'exemple de la coloration de graphe.

Recherche exhaustive

La recherche exhaustive consiste à tester toutes les combinaisons possibles de valeurs pour les variables. Pour un problème de coloration de graphe avec n sommets et k couleurs, cette méthode évalue systématiquement chacune des k^n affectations possibles. Bien que garantissant de trouver une solution si elle existe, cette approche devient rapidement impraticable à mesure que n et k augmentent, en raison de la croissance exponentielle de l'espace de recherche.

Algorithmes de retour arrière (Backtracking)

Le retour arrière est une amélioration de la recherche exhaustive. Il explore l'arbre de recherche de manière récursive, en assignant des valeurs aux variables une à une. Lorsqu'une assignation viole une contrainte, l'algorithme "revient en arrière" pour essayer une autre valeur. Cette méthode évite l'évaluation de nombreuses combinaisons inutiles, réduisant ainsi l'espace de recherche.

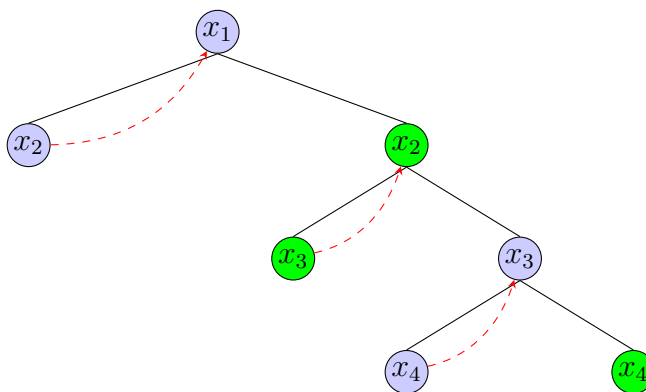


FIGURE 2.4 Recherche avec retour arrière

Exemple avec coloration de graphe Considérons un graphe avec n sommets et k couleurs. Le backtracking commence par colorier le premier sommet avec la première couleur. Si cette couleur ne viole aucune contrainte, l'algorithme passe au sommet suivant. Si toutes les couleurs sont testées

et qu'aucune n'est valide, l'algorithme revient au sommet précédent pour essayer une autre couleur. Visuellement, ceci est représenté dans la figure 2.4.

Le principe de "branch-and-bound"

Le principe du "branch and bound", ou branchement et bornage, est une méthode systématique pour résoudre les problèmes d'optimisation discrète. Ce procédé consiste à diviser l'espace de recherche en sous-ensembles (branchement) plus petits et plus gérables, tout en évaluant les limites inférieure et supérieure des solutions possibles dans chaque sous-ensemble. Si la limite d'un sous-ensemble ne peut pas surpasser la meilleure solution actuelle, ce sous-ensemble est écarté (borné), réduisant ainsi le volume de recherche. Cette méthode est particulièrement efficace pour trouver la meilleure solution possible ou pour prouver qu'aucune solution ne peut satisfaire les critères donnés.

Exemple avec la coloration de graphe

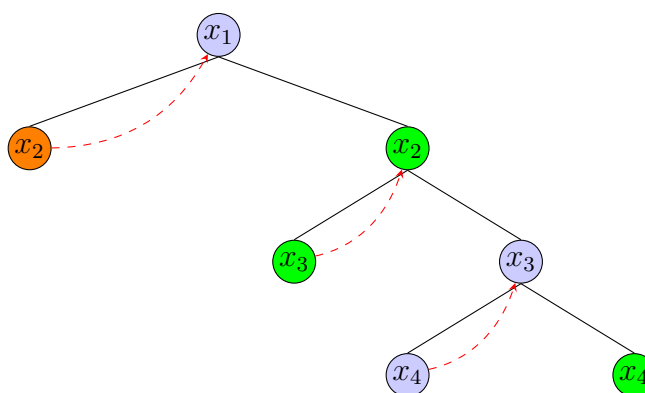


FIGURE 2.5 Branch and bound

Dans cet exemple, la recherche commence par explorer la partie droite de l'arbre et trouve une solution qui n'utilise que deux couleurs. Ensuite, la recherche se poursuit dans la partie gauche de l'arbre. Comme on arrive à une solution à deux couleurs dès le premier noeud de cette portion de l'arbre, on conclut qu'aucune solution ne surpassera celle qui a été obtenue dans la partie droite. Ainsi, cette portion de l'arbre de recherche est élaguée.

Heuristiques de Branchement

Les heuristiques de branchement représentent une composante essentielle dans les méthodes de résolution exactes comme le branchement et bornage (branch and bound). Elles jouent un rôle crucial

dans l'amélioration de l'efficacité de ces méthodes en réduisant l'espace de recherche. Ces heuristiques déterminent l'ordre dans lequel les nœuds de l'arbre de décision sont explorés et comment l'espace de recherche est partitionné.

L'idée principale derrière les heuristiques de branchement est de prioriser certaines branches de l'arbre de recherche, en se basant sur une estimation de leur potentiel à conduire vers une solution optimale ou à faciliter l'élagage d'autres branches non prometteuses. Cette priorisation peut être basée sur divers critères, tels que l'heuristique de sélection de variable, la réduction maximale du domaine, ou des estimations basées sur la fonction objectif [13].

Avantages et Inconvénients

L'avantage principal des heuristiques de branchement est leur capacité à améliorer considérablement l'efficacité des algorithmes de résolution exacte. En concentrant l'exploration sur les parties les plus prometteuses de l'espace de recherche, elles peuvent réduire de manière significative le temps nécessaire pour trouver une solution optimale ou pour prouver son inexistence.

Cependant, l'inconvénient majeur réside dans le fait que l'efficacité de ces heuristiques dépend fortement de la nature spécifique du problème. Il n'existe pas d'heuristique de branchement universellement optimale, et le choix de la meilleure heuristique peut nécessiter une connaissance approfondie du problème en question.

2.4.2 Métaheuristiques

Les métaheuristiques jouent un rôle crucial dans la résolution de problèmes combinatoires, en particulier lorsque la taille ou de la complexité du problème ralentit la résolution. Contrairement aux méthodes exactes, ces approches ne garantissent pas de trouver la solution optimale, mais elles visent à trouver des solutions de bonne qualité dans un délai raisonnable.

Principales métaheuristiques

Parmi les principales métaheuristiques utilisées pour résoudre des problèmes combinatoires, on trouve :

Recuit simulé : Inspiré par le processus de recuit en métallurgie, cette méthode simule le refroidissement lent d'un matériau. Le recuit simulé commence avec une température élevée qui diminue progressivement. À haute température, l'algorithme peut accepter des solutions de moins bonne qualité, ce qui permet d'éviter de rester coincé dans un minimum local. À mesure que la température diminue, l'algorithme devient plus sélectif, se concentrant sur l'amélioration de la qualité de

la solution. Le recuit simulé est exposé en plus de détails dans ce chapitre par Henderson, Jacobson et Johnson (2003) [14].

Recherche tabou : Cette technique emploie une mémoire à court terme, la liste tabou, pour mémoriser les dernières modifications apportées à la solution et éviter de revenir immédiatement sur ces modifications. Cela aide à explorer de manière plus systématique l'espace de recherche et à éviter les cycles. La recherche tabou utilise également des critères d'aspiration qui peuvent permettre de surmonter les restrictions de la liste tabou si cela conduit à des améliorations significatives de la solution. Une description plus détaillée de la méthode est présentée dans ce chapitre par Gendreau et Potvin (2005) [15].

Algorithmes génétiques : Ces algorithmes imitent la théorie de l'évolution biologique. Ils opèrent sur une population de solutions potentielles, utilisant des opérations inspirées de la génétique, telles que la sélection (choix des meilleures solutions), le croisement (combinaison de deux solutions pour en créer de nouvelles) et la mutation (modification aléatoire d'une solution). Au fil des générations, la population évolue vers de meilleures solutions. Nous invitons le lecteur intéressé à consulter la revue de Katoch et al. (2021) [16].

Optimisation par essaims de particules : Ce modèle s'inspire du comportement social des essaims. Chaque particule de l'essaim représente une solution potentielle et se déplace dans l'espace de recherche en fonction de sa propre expérience et de celle des autres particules. Les particules s'ajustent en fonction de la position de la meilleure particule de l'essaim, conduisant l'ensemble vers des solutions optimales. Une revue de ces techniques est présentée dans Shami et al. (2022) [17].

Recherche locale itérative : Cette approche consiste à appliquer de manière répétée un algorithme de recherche locale à partir de différentes solutions initiales. Après chaque exécution de la recherche locale, une perturbation est appliquée à la solution actuelle pour échapper aux minima locaux, et une nouvelle recherche locale est lancée. L'itération se poursuit jusqu'à ce qu'un critère d'arrêt soit atteint. Un survol de cette méthode peut être trouvé dans ce chapitre par Stützle et Ruiz (2018) [18].

Avantages et inconvénients

Les métaheuristiques sont flexibles et souvent efficaces pour trouver des solutions acceptables dans des délais raisonnables. Elles sont particulièrement utiles pour les problèmes de grande taille ou ceux où la structure de la solution optimale est inconnue.

Le principal inconvénient est qu'elles ne garantissent pas de trouver la solution optimale. De plus, le réglage des paramètres peut être complexe et avoir un impact significatif sur la performance de

l'algorithme.

2.5 Apprentissage automatique

Cette section décrit les grandes lignes de l'apprentissage automatique (ML), en se concentrant sur l'apprentissage profond. Elle est adaptée de [19].

2.5.1 Apprentissage supervisé

L'apprentissage supervisé est une tâche centrale en apprentissage automatique. Le but est de construire une fonction f capable de prédire la valeur de sortie correspondant à un intrant donné, en se basant sur un ensemble de données étiquetées. Dans ce contexte, chaque élément de l'ensemble de données consiste en une paire intrant-extrant, où l'intrant est généralement représenté par un vecteur de caractéristiques et l'extrant est l'étiquette ou la valeur cible associée.

La fonction f , souvent appelée modèle, est ajustée de manière à minimiser la différence entre les prédictions qu'elle génère et les valeurs des extrants dans les données d'entraînement. Cette différence est quantifiée à l'aide d'une fonction de perte, qui mesure l'erreur du modèle. L'objectif principal de l'apprentissage supervisé est donc de minimiser cette fonction de perte.

Fonction de perte

La fonction de perte joue un rôle crucial dans l'apprentissage supervisé, car elle guide l'ajustement du modèle. Elle quantifie l'erreur entre les prédictions du modèle et les valeurs attendues (étiquettes). Le choix de la fonction de perte dépend du type de tâche d'apprentissage : pour la régression, on utilise souvent l'erreur quadratique (la différence au carré entre la prédiction et la valeur réelle), tandis que pour la classification, des fonctions telles que l'entropie croisée sont fréquemment employées.

Le processus d'optimisation ajuste les paramètres du modèle de manière à minimiser la fonction de perte sur l'ensemble d'entraînement. Cette optimisation est généralement réalisée à l'aide d'algorithmes tels que la descente de gradient, qui ajuste progressivement les paramètres pour réduire l'erreur.

2.5.2 Réseaux de neurones

Les réseaux de neurones sont des approximateurs de fonctions continues. Ils sont à la base de l'apprentissage profond et ont été à l'origine de percées dans une variété de tâches telles que la

régression, la classification et la création de modèles génératifs. Ces méthodes trouvent des applications dans de nombreux domaines, y compris le traitement naturel du langage et la vision par ordinateur. Les réseaux de neurones de base, appelés réseaux denses, sont composés de séries de couches de neurones. Chaque couche est paramétrée par une matrice de poids W_l et un vecteur de biais b_l . On dénotera par θ l'ensemble des paramètres du réseau de neurones : les matrices W_l et les vecteurs b_l .

Le réseau de neurones fonctionne comme une fonction f_θ qui applique une transformation linéaire - multiplication par une matrice de poids W_l et addition d'un vecteur de biais b_l - suivie d'une fonction d'activation non linéaire ϕ_l . Avec le vecteur d'entrée x , le réseau complet prend alors la forme suivante :

$$f_\theta(x) = \phi_L(W_L \dots \phi_1(W_1 \cdot \phi_0(W_0 x + b_0) + b_1) \dots + b_L)$$

Étant donné un nombre suffisant de neurones, par le théorème d'approximation universelle [20, 21] de telles fonctions peuvent approximer avec une précision peuvent approximer toute fonction continue à un niveau de précision arbitraire.

Apprentissage

Ces modèles sont ensuite entraînés à l'aide d'un algorithme d'optimisation stochastique. Ceci se fait grâce à une mise à jour des paramètres avec un algorithme de descente de gradient sur une fonction que l'on veut optimiser. Illustrons ceci dans le contexte d'un problème de régression. Pour ce problème, le réseau sera entraîné à prédire une quantité $y_i \in \mathbb{R}$ d'après un vecteur $x_i \in \mathbb{R}^n$. Pour ce faire, le modèle apprendra à l'aide de n paires (x_i, y_i) . Autrement dit, à l'aide de ces données, le réseau viendra à ajuster ses paramètres θ pour minimiser l'erreur quadratique

$$L(\theta) = \sum_i (y_i - f_\theta(x_i))^2.$$

Pour ce faire, on calcule le gradient de L par rapport aux poids θ . Afin de calculer le gradient par rapport à toutes les couches du réseau, l'algorithme de rétropropagation est utilisé [22]. Au moment de l'implémentation, ceci se fait à même les bibliothèques d'apprentissage profond, qui peuvent calculer $\nabla_\theta L$ à l'aide de leurs modules de différentiation automatique. Finalement, les paramètres du réseau sont mis à jour dans la direction qui minimise L . La forme de base de descente de gradient est décrite de la façon suivante :

$$\theta \leftarrow \theta - \alpha \nabla_\theta L(\theta)$$

où α est un hyper-paramètre appelé "pas d'apprentissage". Ce processus est itéré jusqu'à ce que le

modèle converge vers une solution optimale ou satisfaisante.

2.6 Réseaux de neurones sur graphes

Dire que les réseaux de neurones ont révolutionné l'apprentissage automatique est un euphémisme. Toutefois, ceux-ci ne fonctionnent qu'avec des données sous formes de vecteurs ou de matrices. Or, plusieurs problèmes et réalités sont naturellement représentés sous formes de graphes, structures pour lesquelles les réseaux de neurones de base ne sont pas adaptés. Bien qu'il est techniquement possible d'utiliser des réseaux de neurones denses en prenant en entrée la représentation matricielle des graphes, cette approche omet certaines caractéristiques importantes des graphes en question, notamment l'invariance sous la permutation des noeuds [23]. Ainsi, une approche dédiée aux graphes s'est imposée : les réseaux de neurones sur graphes (GNNs).

Les GNNs font l'objet d'une sous-discipline de l'apprentissage profond en pleine expansion. Introduits en 1997 [24], puis formalisés en 2009 [25], ils ont trouvé de nombreuses applications dans des domaines aussi variés que les sciences sociales [26], la biologie moléculaire [27], les systèmes de recommandation [28] ou encore l'optimisation combinatoire [29].

Les principales tâches pour lesquelles les GNNs sont utilisés sont la classification de sommets, la prédiction d'existence d'arête ou encore la classification de graphes. Celles-ci sont illustrées dans la figure 2.6.

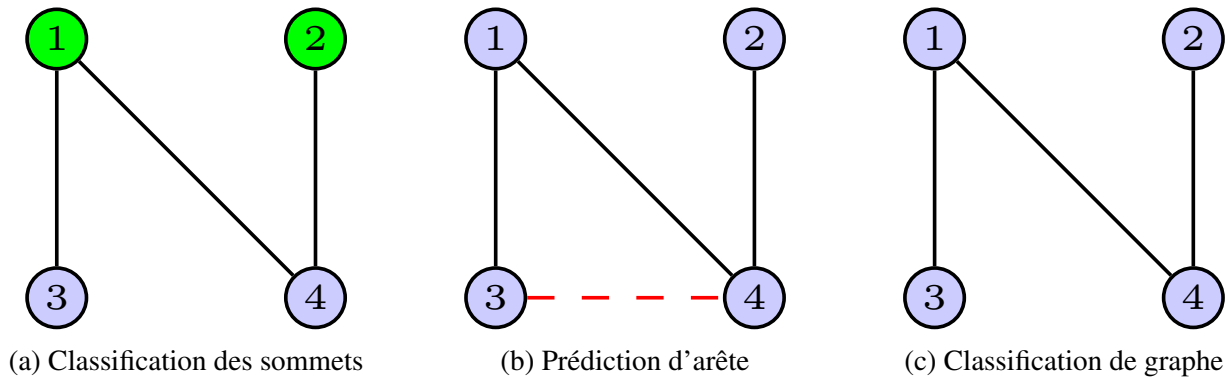


FIGURE 2.6 Principales tâches pour les GNNs

2.6.1 Intuition et notation

Les GNN prennent en entrée un graphe $G = (V, E)$, dont les arêtes et sommets peuvent être assortis d'un vecteur de caractéristiques, que l'on dénotera respectivement $\mathbf{f}_v, v \in V$ et $\mathbf{h}_{u,v}, (u, v) \in E$.

À un très haut niveau, les GNNs visent à apprendre un encodage pour représenter les arêtes et/ou sommets d'un graphe donné. Pour ce faire, les GNN vont généralement recourir à l'échange de messages entre sommets adjacents [30]. La procédure implique la conception, l'envoi, l'aggrégation et le traitement des messages entre un sommet v et les noeuds qui lui sont adjacents. On représentera par $N(v)$ les sommets adjacents au sommet v . Finalement, on note que les GNNs, comme les réseaux de neurones traditionnels, apprennent par descente de gradients, donc les opérations auxquelles ils font appel doivent être différentiables.

2.6.2 Le GNN, étape par étape

Cette sous-section donne les détails sur ces différentes étapes du processus d'échange de messages. On note qu'il existe une multitude d'architectures de GNNs. Afin de demeurer générique dans nos explications, nous nous baserons sur le formalisme tel que décrit par Hamilton [23].

Création des messages

Chaque nœud $w \in N(v)$ crée un message à destination du nœud v selon une fonction différentiable M_θ . La fonction M_θ peut faire appel à des fonctions différentiables pour les sommets adjacents, le sommet lui-même et les arêtes impliquées. Le message de w à v au temps t est donné par :

$$m_t(w \rightarrow v) = M_\theta(\mu_t(w), \mu_t(v), \mathbf{h}_{w,v}) \quad (2.1)$$

où μ est une fonction différentiable sont des matrices de paramètres et $\mathbf{h}_{w,v}$ représente le vecteur de caractéristiques de l'arête $(w, v) \in E$.

Envoi et agrégation des messages

Chaque nœud v envoie et reçoit des messages de ses voisins. Comme chaque sommet peut recevoir une quantité variable de messages provenant des sommets voisins, il faut les *agréger* afin que ceux-ci aient une dimension appropriée. Par exemple, l'opérateur d'aggrégation peut prendre le minimum, le maximum ou la somme, composante par composante, des vecteurs caractéristiques des sommets voisins. On dénote l'opérateur d'aggrégation des messages par un $\oplus$. L'aggrégation des messages reçus par le sommet v au temps t est donnée par :

$$A_t(v) = \bigoplus_{w \in N(v)} m_t(w \rightarrow v) \quad (2.2)$$

Mise à jour de l'état latent

L'état latent -le vecteur de caractéristiques- de v est mis à jour via la fonction différentiable ϕ_θ :

$$\mu_t(v) = \phi_\theta(\mu_t(v), A_t(v), \mathbf{f}_v) \quad (2.3)$$

où θ dénote une matrice de paramètres. On note qu'il est préférable que la fonction ϕ soit invariante sous la permutation pour garantir que l'ordre des nœuds voisins dans le graphe n'affecte pas le résultat de la mise à jour de l'état latent. Cette invariance est importante car les graphes, contrairement à d'autres structures de données comme les séquences ou les grilles, n'ont pas d'ordre intrinsèque de leurs éléments. Par conséquent, le calcul de l'état latent d'un nœud ne devrait pas dépendre de l'ordre dans lequel les informations de ses voisins sont traitées. Une fonction ϕ invariante sous la permutation assure que le GNN peut maintenir une cohérence dans la représentation, quelle que soit la manière dont les nœuds sont ordonnés dans le graphe d'entrée. Cela est essentiel pour permettre au GNN d'apprendre des représentations fiables et généralisables des données structurées sous forme de graphes.

Le processus en entier

Ces étapes sont répétées en succession pour T itérations de manière à ce que le vecteur de caractéristiques de chaque sommet v aie reçu de l'information provenant de plus loin que son voisinage immédiat. La valeur de T indique le degré de localité utilisé pour faire apprendre des encodages. Par exemple, si $T = 4$, alors l'encodage final d'un sommet contiendra de l'information provenant de sommets à une distance de 4 du sommet en question. Ceci est illustré dans la figure 2.7, en prenant le sommet x_1 comme référence.

Algorithme d'inférence

Après un certain nombre de d'itérations telles que décrites plus haut, le pipeline de sortie -de prédiction- est adapté en fonction de la tâche. Pour une tâche de classification des sommets, les encodages finaux des sommets sont passés dans un réseau adapté à la classification, par exemple un réseau dense. Ainsi, les encodages de tous les sommets passent par ce réseau et il en résulte une prédiction pour chaque sommet.

Pour une tâche de classification de graphe, il faut agréger l'information des encodages finaux de manière à produire un vecteur représentant les probabilités associées aux différentes classes. Comme dans l'étape d'aggrégation décrite plus haut, ceci peut se faire avec différents opérateurs, notamment la moyenne.

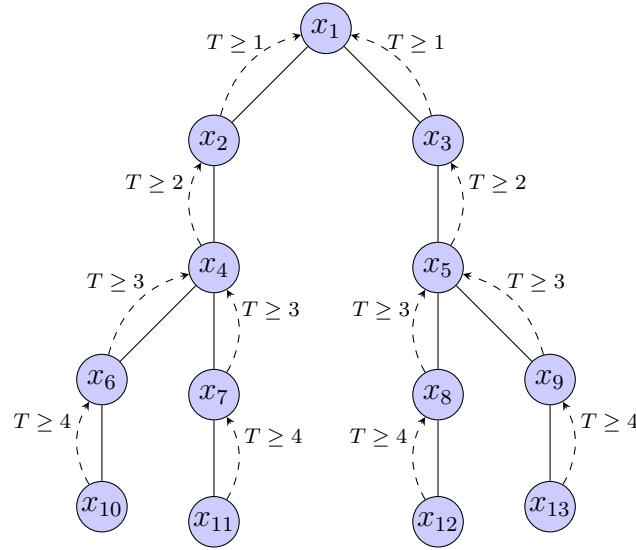


FIGURE 2.7 Propagation de l'information dans un GNN

2.7 Optimisation combinatoire et apprentissage

Les problèmes NP-difficiles et NP-complets ont fait l'objet de décennies de recherche. Au fil du temps, une multitude de méthodes de solution ont été proposées, mais parmi celles-ci, les méthodes à base d'apprentissage demeurent marginales. Toutefois, celles-ci offrent la promesse de fournir rapidement des solutions décentes et d'apprendre à l'aide de problèmes similaires provenant du passé. Ces deux avantages peuvent être très avantageux dans des contextes industriels où des instances similaires doivent être résolues jour après jour.

Considérons, par exemple, le cas d'un livreur de colis qui opère régulièrement dans une région donnée. Au fil du temps, il accumule une quantité considérable de connaissances sur ses parcours antérieurs, les conditions de trafic, les préférences des clients, etc. En exploitant ses connaissances, le livreur peut arriver à trouver de meilleurs itinéraires et à exploiter les similarités entre les tournées à faire d'un jour à l'autre. En se servant des données historiques, les méthodes d'apprentissage peuvent aider à optimiser les futures tournées de livraison en adaptant les itinéraires en fonction des conditions et des expériences passées. Cette capacité d'adaptation et d'apprentissage continu peut s'avérer extrêmement utile dans des contextes industriels où des problématiques similaires se présentent régulièrement, comme dans le cas de notre livreur.

En outre, cette section propose de détailler d'autres avantages des méthodes basées sur l'apprentissage en optimisation combinatoire, soulignant leur potentiel à transformer la manière dont ces problèmes complexes sont abordés et résolus.

2.7.1 Utilité des méthodes à base d'apprentissage

Tel que mentionné en introduction, le processus de développement de nouvelles heuristiques est complexe et demande une expertise avec les problèmes en question. Dans le but d'accélérer ce processus -et plus largement pour faire avancer la recherche en optimisation-, plusieurs équipes de recherche ont développé des méthodes à base d'apprentissage pour résoudre des problèmes NP complets/NP difficiles. Voyant la pertinence de ces travaux, Prouvost et al. (2020) [9] ont regroupé les différentes utilités des méthodes à base d'apprentissage pour l'optimisation. Les auteurs motivent l'utilité des méthodes à base d'apprentissage à l'aide d'un exemple d'entreprise de livraison qui doit résoudre, jour après jour, des problèmes de tournées de véhicules. Bien que ceux-ci ne soient pas identiques de jour en jour, ceux-ci présentent tout de même certaines similarités qui pourraient être exploitées par un algorithme qui apprend à résoudre ces problèmes, ce qui n'est pas le cas des méthodes classiques de solution.

La première mention de l'emploi de réseaux de neurones pour la résolution de problèmes combinatoires remonte à 1985 [31]. Dans cet article, les auteurs décrivent des façons d'utiliser des réseaux de neurones pour résoudre des instances du problème du voyageur de commerce (TSP). Or, comme la puissance de calcul était limitée, ces recherches n'ont pas été poursuivies avant plusieurs années. Près de 30 ans plus tard, on assiste à l'explosion de l'apprentissage profond avec le concours *ImageNet* [7] où une équipe de chercheurs de l'université de Toronto remporte le prestigieux concours de classification d'images -ImageNet- à l'aide de l'apprentissage profond. Cette percée remet cette méthode à l'avant plan de la recherche en intelligence artificielle (IA) et démontre la possibilité d'utiliser l'apprentissage profond à grande échelle. Quelques années plus tard, Vinyals et al. (2015) [32] développent les *Pointer networks*, un type de réseau de neurones récurrents. Ils les entraînent à résoudre des instances de TSP de taille 5 à 50 en s'entraînant sur des instances de taille 5 à 20. Pour les tailles 5 à 20, leur méthode arrive aux solutions optimales ou en est très proche, mais les performances se dégradent à mesure que les tailles des instances augmentent. Peu après, Bello et al. (2016) [33] développe une méthode basée sur l'apprentissage par renforcement pour résoudre le TSP. Leur modèle est entraîné sur des instances de taille 20 à 50 et parvient à trouver égal ou approcher les solutions optimales, obtenues à l'aide du solveur Concorde [34]. Dans Khalil et al (2016) [35], les auteurs développent une approximation d'un modèle linéaire utilisé pour faire des décisions de branchement dans une exploration de *Branch & bound*. Plusieurs autres approches basées sur l'apprentissage ont été proposées pour les décisions de branchement. Les lecteurs intéressés sont invités à consulter Lodi et Zarpellon (2017) [36] pour une revue plus approfondie.

2.7.2 GNNs et optimisation

Depuis, la plupart des avancées ont été réalisés à l'aide de GNNs. Khalil et al. (2017) [37] entraînent des GNNs sur des instances de taille 50 à 100 pour les problèmes de couverture par sommets, le TSP et le problème de coupe maximum. Pour le TSP, ils obtiennent des performances à moins de 10% de l'optimum pour des problèmes de tailles 50 à 1200. Pour les deux autres problèmes, ils arrivent à des performances à moins de 2% de l'optimum pour les tailles 50 à 1200. Kool et al. (2019) [38] utilisent un *Graph Attention Network* [39] dans le cadre d'une politique d'apprentissage de type REINFORCE [40] et parviennent à surpasser d'autres méthodes basées sur l'apprentissage telles que le *Pointer Network* [32]. Leur méthode démontre une performance qui s'approche de celle des solveurs tels que Google OR-Tools [41]. Utilisant aussi une architecture de GNN, Gasse et al. (2019) utilisent aussi un GNN pour faire des décisions de branchement en se basant sur un graphe représentant l'entière des problèmes et non une représentation condensée de ceux-ci. Fu et al. (2021) [42] emploient des GNNs dans un contexte de *Monte Carlo Tree Search* et arrivent à trouver des solutions compétitives pour des TSP de taille allant jusqu'à 10 000 sommets. Leur méthode arrivent à surpasser les méthodes à base d'apprentissage passées et améliore la capacité de généralisation de celles-ci. Le lecteur intéressé est invité à consulter Cappart et al. (2023) [10] pour en apprendre plus.

2.7.3 Problèmes de satisfiabilité

Pour les problèmes de satisfiabilité, Palm et al. (2018) [43] ont démontré qu'il était possible d'entraîner des GNNs pour résoudre des Sudoku n'ayant qu'une seule solution. S'inspirant de ces résultats, Selsam et al. (2018) [44] ont développé l'architecture NeuroSAT, qui est à la base des travaux du présent mémoire. À l'aide de cette architecture, ils sont parvenus à entraîner un modèle capable de prédire avec 85% de précision si un problème de satisfiabilité booléenne avait une solution ou non. Selsam et al. (2019) [45] ont ensuite développé une méthode de solution de tels problèmes guidée par un GNN. Ce premier article de Selsam et al. a inspiré de nombreux autres chercheurs à adapter son architecture afin de prédire la satisfiabilité d'autres problèmes. On note les travaux de Prates et al (2018) [46] où les auteurs arrivent à 80% de précision pour la tâche de prédiction de la version décisionnelle du TSP. Ensuite, Lemos et al. (2019) [47] ont appliqué une procédure similaire pour prédire la version décisionnelle du problème de coloration de graphe. Finalement, Liu et al. (2020) [48] emploie une méthode similaire pour prédire la satisfiabilité sur des instances de Knapsack. On note que pour tous ces papiers, les auteurs se sont basés sur des graphes qui encodent les problèmes de façon propre type de problème. Autrement dit, la représentation sous forme de graphe des problèmes SAT ne peut être utilisée que pour des problèmes SAT, la représentation des TSP n'est valide que pour des TSP, etc.

2.8 Question de recherche

Les approches à base d'apprentissage sont limitées puisqu'elles ne fonctionnent que pour un certain type de problème, tel que mentionné dans la section précédente. Ainsi, il demeure que ces approches pourraient bénéficier d'une représentation générique des problèmes. Suivant cette piste, notre question de recherche est la suivante : Est-il possible de concevoir une représentation des problèmes combinatoires qui soit entièrement générique ? Une telle représentation est capable de représenter un problème combinatoire arbitraire et permettre de réaliser des performances comparables à celles des représentations propres aux problèmes. Pour étudier ceci, nous avons développé une représentation générique capable de représenter des problèmes combinatoires. Cette représentation est capable de générer des graphes à partir de toute instance du format XCSP3 [11], avec les contraintes permises dans la mini-track. À l'aide de cette représentation, nous sommes parvenus à approcher les performances des architectures spécifiques aux problèmes étudiés : la coloration de graphe, le TSP décisionnel, le problème SAT et le Knapsack.

CHAPITRE 3 Article 1 : TOWARDS A GENERIC REPRESENTATION OF COMBINATORIAL PROBLEMS FOR LEARNING-BASED APPROACHES

Co-authors : Hélène Verhaeghe, Quentin Cappart

Published in : The 21st International Conference on the Integration of Constraint Programming, Artificial Intelligence, and Operations Research (CPAIOR2024)

Publication date : 2024/05/25

Les contributions de Léo Boisvert sont : Conception de la représentation graphique des problèmes combinatoires, design et implémentation des réseaux de neurones utilisés pour l'apprentissage, expérimentation, analyse des résultats, rédaction de l'article et recherche bibliographique.

3.1 Introduction

Combinatorial optimization has drawn the attention of computer scientists since the discipline emerged. Combinatorial problems, such as the *traveling salesperson problem* or the *Boolean satisfiability* have been the focus of many decades of research in the computer science community. We can now solve large problems of these kinds efficiently with exact methods [1,2] and heuristics [3]. Heuristics are handcrafted procedures that crystallize expert knowledge and intuition about the structure of the problems they are applied to. While heuristics have found success and applications for the resolution of combinatorial problems either as a direct-solving process or integrated into a search procedure, the rise of deep learning in many different fields [6–8, 49] has attracted the attention of researchers [9]. Among deep learning architectures, *graph neural networks* (GNNs) [25] have proven to be a powerful and flexible tool for solving combinatorial problems. However, as identified by Cappart et al. (2023) [10], it is still cumbersome to integrate GNN and machine learning into existing solving processes for practitioners. One reason is that a dedicated architecture must be designed and trained for each combinatorial problem. In addition to potentially expensive computing resources for training, this also requires the existence of a large and labeled training set.

Building such a problem-specific graph representation has been the preferred choice of most related approaches, such as NeuroSAT [44] leveraging an encoding for SAT formula, or such as approaches tightly linked to the traveling salesperson problem [46, 50]. These approaches suffer from a lack of generality as the architecture could not be trivially exported from one combinatorial problem to another one. Few attempts have been realized to bridge this gap, but they still provide only a partial genericity. For instance, Gasse et al. (2019) [29] proposed a bipartite graph linking variables and constraints when a variable is involved in a given constraint. However, this approach only encodes binary mixed-integer programs. Chalumeau et al. (2021) [51] later introduced a tri-

partite graph where variables, values, and constraints are specific types of vertices. This approach also lacks genericity as the method requires retraining when the number of variables changes. To our knowledge, the last attempt in this direction has been realized by Marty et al. (2023) [52] who leveraged a tripartite graph allowing decorating each vertex type with dedicated features. Although any combinatorial problem can theoretically be encoded with this framework, some information is lost with the encoding. For instance, it was not clear how the constraint $3x_1 \leq 4x_2$, could be differentiated from the constraint $2x_1 \leq 5x_2$. On the one hand, both constraints can be represented as an *inequality*, but we lose information about the variables’ coefficients. On the other hand, both constraints can be encoded as two distinct relationships, but in this case, we lose the fact that we have an inequality in both cases. More formally, their encoding function was not *injective*. Different combinatorial problems could have the same encoding with no option to differentiate them. Besides, the experiments proposed only targeted relatively pure problems (*maximum independent set*, *maximum cut*, and *graph coloring*). Similar limitations are observed in the approach of Tönschoff et al. (2022) [53].

Based on this context, this paper progresses towards a fully generic representation of combinatorial problems for learning-based approaches. Intuitively, our idea is to break down any constraint of a combinatorial problem as an *abstract syntax tree* and connect similar items (e.g., the same variables or constraints) through an edge. Then, we introduce a GNN architecture able to leverage this graph. To demonstrate the genericity of this approach, our architecture directly operates on combinatorial problems expressed with the XCSP3 format [11] and can handle all the constraints available in the 2023 mini-track competition. Experiments are carried out on four problems (featuring standard intension, and global constraints such as ALLDIFFERENT [54], TABLE [55], and NEGATIVE-TABLE [56], ELEMENT and SUM) and aims to predict the satisfiability of the decision version of combinatorial problems. The results show that our generic architecture gives performances close to problem-specific architectures and outperforms the tripartite graph of Marty et al. (2023) [52]. To ensure the reproducibility of the results, the implementation and the models trained are released on GitHub.

3.2 Encoding Combinatorial Problems as a Graph

Formally, a combinatorial problem $\mathcal{P}$ is defined as a tuple $\langle X, D(X), C, O \rangle$, where X is the set of variables, $D(X)$ is the set of domains, C is the set of constraints, and $O : (X \rightarrow \mathbb{R})$ is an objective function. A valid solution is an assignment of each variable to a value of its domain such that every constraint is satisfied. The optimal solution is a valid solution such that no other solution has a better value for the objective. Our goal is to build a function $\Phi : \langle X, D(X), C, O \rangle \mapsto \mathcal{G}(V, E)$, where $\mathcal{G}$ is a graph and V, E are its vertex and arc sets, respectively. We want this function to be *injective*,

i.e., an encoding refers to at most one combinatorial problem. We propose to do so by introducing an encoding consisting in a heterogeneous and undirected graph featuring 5 types of vertices : *variables*, *constraints*, *values*, *operators*, and *model*. The core idea is to split each constraint as a sequence of elementary operations, to merge vertices representing the same variable or value, and connect together all the relationships. An illustration of an encoding is proposed in Figure 3.1. Intuitively, this process is similar to building the abstract syntax tree of a program. Formally, the encoding gives a graph $\mathcal{G}(V_{\text{VAR,CST,VAL,OPE,MOD}}, f_{\text{VAR,CST,VAL,OPE,MOD}}, E)$ with the five types of vertices, and specific features attached to each vertex. Each type is defined as follows.

Values (V_{VAL}). A *value-vertex* is introduced for each constant appearing in $\mathcal{P}$. Such values can appear inside either a domain or a constraint. All the values are distinct, i.e., they are represented by a unique vertex. The type of the value (integer, real, etc.) is added as a feature (f_{VAL}) to each *value-vertex*, using a one-hot encoding.

Variables (V_{VAR}). A *variable-vertex* is introduced for each variable appearing in $\mathcal{P}$. A vertex related to a variable $x \in X$ is connected through an edge to each value v inside the domain $D(x)$. Such as the *value-vertices*, all variables are represented by a unique vertex. The type of the variable (boolean, integer, set, etc.) is added as a feature (f_{VAR}) using a one-hot encoding.

Constraints (V_{CST}). A *constraint-vertex* is introduced for each constraint appearing in $\mathcal{P}$. A vertex related to a constraint $c \in C$ is connected through an edge to each value v that is present in the constraint. The type of the constraint (*inequality*, *allDifferent*, *table*, etc.) is added as a feature (f_{CST}) using a one-hot encoding.

Operators (V_{OPE}). Tightly related to constraints, *operator-vertices* are used to unveil the combinatorial structure of constraints. Specifically, operators represent modifications or restrictions on variables involved in constraints (e.g., arithmetic operators, logical operators, views, etc.). An *operator-vertex* is added each time a variable $x \in X$ is modified by such operations. The vertex is connected to the vertices related to the impacted variables. The type of the operator ($+$, $\times$, $\wedge$, $\vee$, etc.) is added as a feature (f_{OPE}) using a one-hot encoding. If the operator uses a

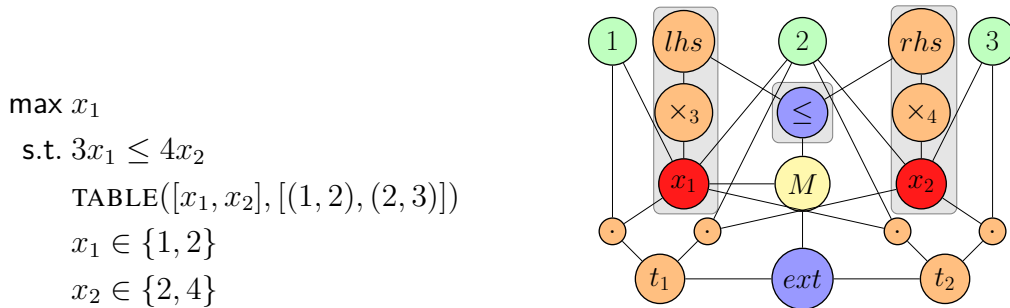


FIGURE 3.1 Encoding of a combinatorial problem presented as a running example.

numerical value to modify a variable, this value is used as a feature (f_{OPE}) as well.

Model (V_{MOD}). There is only one *model-vertex* per graph. This vertex is connected to all *constraint-vertices* and all *variable-vertices* involved in the objective function. Its semantics is to gather additional information about the combinatorial problem (e.g., the direction of the optimization) by means of its feature vector (f_{MOD}).

Example.

There are 3 *value-vertices* depicted in the encoding of Figure 3.1 (in green) and 2 *variable-vertices* (in red). As x_1 contains values 1 and 2 in its domain, they are connected with an edge, and similarly for the domain of x_2 . There are 2 *constraint-vertices* (in blue), one for the inequality ($\leq$) and one for the TABLE constraint (**ext**). The gray area in the figure shows how the constraint $3x_1 \leq 4x_2$ is scattered with operators (in orange). Specifically, it involves a multiplication ($\times$, with feature 3) of x_1 on the right-side (**rhs**) and a multiplication ($\times$, with feature 4) of x_2 on the left-side (**lhs**). The outer operators are then connected to the related constraint, e.g., **rhs** and **lhs** for the inequality ($\leq$). The constraint $\text{TABLE}([x_1, x_2], [(1, 2), (2, 3)])$ is expressed in a similar way. It involves two tuples t_1 and t_2 . Finally, the *model-vertex* (in yellow) is connected to the two constraints, and to variable x_1 , as it is part of the objective function.

As a final remark, this encoding can be used to represent any combinatorial problem in a unique way. To do so, a parser for each constraint, describing how a constraint has to be split with the involved operators and variables, must be implemented. We currently support all the constraints formalized in XCSP3-CORE modeling format [11] and used for the mini-solver tracks of the XCSP-2023 competition : *binary intension*, *table*, *negative table*, *short table*, *element* and *sum*. Our repository contains the required documentation to build a graph from an instance in the XCSP3-CORE formalism.

3.3 Learning from the Encoding with a Graph Neural Network

To achieve the next step and *learn* from this representation, we designed a tailored *graph neural network* (GNN) architecture to leverage this encoding. A GNN is a specialized neural architecture designed to compute a latent representation (known as an *embedding*) for each node of a given graph [25]. This process involves iteratively aggregating information from neighboring nodes. Each aggregation step is denoted as a *layer* of the GNN and incorporates learnable weights. There are various ways to perform this aggregation, leading to different variants of GNNs documented in the literature [39, 57, 58]. The model is differentiable and can then be trained with gradient descent methods.

Let $\mathcal{G}(V_{\text{VAR,CST,VAL,OPE,MOD}}, f_{\text{VAR,CST,VAL,OPE,MOD}}, E)$ be the graph encoding previously obtained, and let $h_{t,v}^{[i]} \in \mathbb{R}^p$ be a p -dimensional vector representation of a vertex $v \in V_t$ (t refers to a vertex type from $\mathcal{T} : \{\text{VAR, VAL, CST, OPE, MOD}\}$) at iteration $i \in \{0, \dots, I\}$. The inference process of a GNN consists in computing the next representations ($h_{t,v}^{[i+1]}$) from the previous ones for each vertex. This operation is commonly referred to as *message passing*. First, we set $h_{t,v}^{[0]} = f_{t,v}$ for each type, where $f_{t,v}$ is the vector of features related to vertex $v \in V_t$. Then, the representations at each iteration are obtained with LayerNorm [59] and LSTM layers [60]. The whole inference process is formalized in Algorithm 1. First, the initial embedding of each vertex is set to its feature (line 5) and the hidden states of the LSTM are initialized to 0 (line 6), as commonly done. Then, I steps of message-passing are carried out (main loop). At each iteration i , a *message* ($\mu_{t_1,v}^{[i]}$) is obtained for each vertex. The computation is done in three steps (line 8) : (1) the embedding of each neighbor of a given type is summed up, (2) the resulting value is fed to a standard multi-layer perceptron ($\text{MLP}_{t_1,t_2}^{\text{in}}$), note that there is a specific module for each edge type, (3) the messages related to each type are concatenated together ($\oplus$) to obtain the global message for each vertex. The result is then given as input to an LSTM cell (line 9, one cell for each type) and is used to obtain the embedding at the next layer. We note that each LSTM has its internal state ($\gamma_{t,v}^{[i]}$) updated. At the end of the loop, each vertex has a specific embedding ($h_{t,v}^{[I]}$). After the last iteration, we compute the vertex-type dependent output by passing $h_{t,v}^{[I]}$ through a standard multi-layer perceptron $\text{MLP}_t^{\text{out}}$ (line 10). This produces the output embeddings for all nodes, which are then averaged (line 11). Finally, the sigmoid function (σ) is used to obtain a value between 0 and 1 as output.

Algorithm 1: Inference process of our GNN architecture.

```

1 ▷ Pre :  $\mathcal{G}(V_{\text{VAR,CST,VAL,OPE,MOD}}, f_{\text{VAR,CST,VAL,OPE,MOD}}, E)$  is the graph encoding.
2 ▷ Pre :  $\mathcal{T} : \{\text{VAR, VAL, CST, OPE, MOD}\}$  is the set of vertex-types.
3 ▷ Pre :  $I$  is the number of iterations of the GNN.
4
5  $h_{t,v}^{[0]} := f_{t,v} \quad \forall v \in V_t, \forall t \in \mathcal{T}$ 
6  $\gamma_{t,v}^{[0]} := 0 \quad \forall v \in V_t, \forall t \in \mathcal{T}$ 
7 for  $i$  from 1 to  $I$  do
8    $\mu_{t_1,v}^{[i]} := \bigoplus_{t_2 \in \mathcal{T}} \text{MLP}_{t_1,t_2}^{\text{in}} \left( \sum_{u \in \mathcal{N}_{t_2}(v)} h_{t_2,u}^{[i-1]} \right) \quad \forall v \in V_{t_1}$ 
9    $h_{t,v}^{[i]}, \gamma_{t,v}^{[i]} := \text{LSTM}_t \left( \mu_{t,v}^{[i]}, \gamma_{t,v}^{[i-1]} \right) \quad \forall v \in V_t$ 
10  $\nu_{t,v} := \text{MLP}_t^{\text{out}} \left( h_{t,v}^{[I]} \right) \quad \forall v \in V_t, \forall t \in \mathcal{T}$ 
11  $\hat{y} := \sigma \left( \frac{1}{|\mathcal{T}| \times |V|} \sum_{t \in \mathcal{T}} \sum_{v \in V_t} \nu_{t,v} \right)$ 
12 return  $\hat{y}$ 

```

3.4 Experiments

The goal of this section is to evaluate the quality of our approach to solve combinatorial tasks. Four combinatorial problems are considered : *Boolean satisfiability* (SAT), *graph coloring* (COL), *knapsack* (KNAP) and the *traveling salesperson problem* (TSP). For the TSP, we considered two models : one based on the TABLE constraint (TSP-EXT), and a second one based on ELEMENT constraints (TSP-ELEM). For each problem, we trained a model to solve in an end-to-end manner their *decision version* : "Given a combinatorial problem $\mathcal{P}$ and a target k , is there a solution satisfying all the constraints of $\mathcal{P}$ with a cost less than k ?". Note that this corresponds to the decision version of a minimization problem. When there is no objective function, we have a plain constrained satisfaction problem (e.g., SAT). We highlight that we do not output the solution, but only an answer (yes or no) if the problem has a solution. This task has been considered in recent related works [44, 46–48]. Two types of baselines are considered for comparison : problem-dependent architectures designed in related works, and the tripartite graph proposed by Marty et al. (2023) [52]. The evaluation metric considered is the accuracy in predicting the answer correctly to the decision problem. The next paragraphs describe the experimental protocol for each problem and implementation details.

Boolean Satisfiability. Instances are generated with the random generator of Selsam et al. (2018) [44].

Briefly, the generator builds random pairs of SAT instances of n variables by adding new random clauses until the problem becomes unsatisfiable. Once the problem becomes unsatisfiable, it changes the sign of the first literal of the problem, rendering it satisfiable. Both the satisfiable and unsatisfiable instances are included in the dataset. Still following Selsam et al. (2018), we built a dataset containing millions of instances having 10 to 40 literals. The SAT-dependent architecture considered in our analysis has been also introduced in the same paper. We used a training set of size 3,980,000 and a validation set of size 20,000.

Traveling Salesperson Problem. Instances are generated with the random generator of Prates et al. (2018) [46]. The generation consists in (1) creating n points in a $(\sqrt{2}/2 \times \sqrt{2}/2)$ square, (2) building the distance matrix with the Euclidean distance, and (3) solving them using the Concorde solver [34] to obtain the optimal tour cost C^* . Two instances are then created : a feasible one and an unfeasible one with target costs of $1.02C^*$ and $0.98C^*$, respectively. We build two TSP models : a first one where the distance constraints are expressed with an extension constraint (TSP-EXT), and a second one with an element constraint (TSP-ELEM). The motivation is to analyze the impact of the model on the resulting graph encoding and the performances. Still following Prates et al. (2018), we built the dataset with a number of cities n sampled uniformly from 20 to 40. The TSP-dependent architecture considered in our analysis has been also introduced in the same paper. We used a training set of size 850,000

and a validation set of size 50,000.

Graph Coloring. Instances are generated following Lemos et al. (2019) [47]. This generator builds graphs with 40 to 60 vertices. For each graph, a single edge is added such that the k -colorability is altered. The instances are produced in pairs : one where the optimal value is k and another one where it is higher. Our encoding leverages a standard model of the graph coloring featuring binary intension ($\neq$) constraints. The COL-dependent architecture has been also introduced in the same paper. We used a training set of size 140,000 and a validation set of size 10,000.

Knapsack. We built instances containing 20 to 40 items and solved them to optimality to find the optimal value V^* . Then, we created two instances, a feasible one and an unfeasible one with a target cost of $1.02V^*$ and $0.98V^*$, respectively. Our encoding leverages a standard knapsack model featuring a SUM constraint. The KNAPSACK-specific model was based on the model of Liu et al. (2020) [48]. We used a training set of size 950,000 and a validation set of size 50,000.

Implementation Details. All models were trained with PyTorch [61] and PyTorch-Geometric [62] on a single Nvidia V100 32 GB GPU for up to 4 days or until convergence. We selected the model having the best performance on the validation set. To make the comparisons between specific and generic graphs fair, we trained all our models using a single GPU, and we tuned each model by varying the number of hidden units in the MLP and LSTM layers. Concerning the problem-specific architectures, we reused the same hyperparameters as described in their original publications. All models are trained with Adam optimizer [63] coupled with a learning rate scheduler and a weight decay of 10^{-8} . The main hyperparameters used for our different models are detailed in the accompanying code. All our models are expressed using XCSP3 formalism. We implemented a parser to build the graph from this representation. Our code and trained models are publicly available.

3.4.1 Results : Accuracy of the Approaches

Table 3.1 summarizes the accuracy in predicting the correct answer on the validation set for each problem and baseline. Interestingly, we observe that our approach achieves similar or close performances as the problem-specific architectures for all the problems. We see it as a promising result, as our approach, thanks to its genericity, can be directly used for all the problems without designing a new dedicated architecture. On the other hand, the approach of Marty et al. (2023) [52], fails to achieve similar results except for the graph coloring. This is because this representation does not preserve the combinatorial structure of complex constraints (COL has only constraints like $x_1 \neq x_2$).

TABLEAU 3.1 Comparative performance of different architectures on various problems.

Architecture	SAT	TSP-EXT	TSP-ELEM	COL	KNAP
Problem-specific	94.3%	96.3%		77.0 %	98.8%
Tripartite [52]	50.0%	50.0%		84.6%	50.0%
Ours	94.4%	84.5%	91.4%	84.4%	97.9%

While TSP-ELEM achieves results close to the TSP-specific approach, TSP-EXT falls short significantly. This highlights the importance of using an appropriate combinatorial model for the encoding. Specifically, the encoding TSP-ELEM has a size of 1841 vertices and 13042 edges, while TSP-EXT yields a graph of 5661 vertices and 28322 edges, for a same instance of 40 cities. This is consequently larger encoding, which is not desirable as it makes the model harder to train.

3.4.2 Analysis : Generalization to Larger Instances

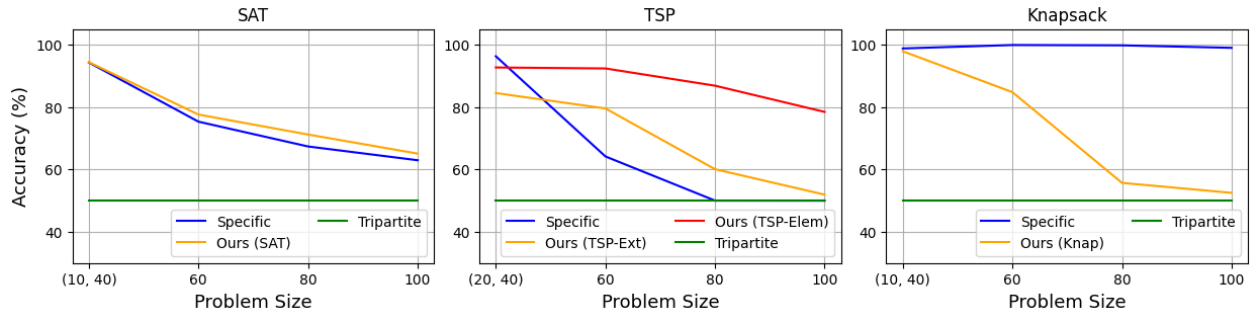


FIGURE 3.2 Analysis of the generalization ability on larger instances.

Figure 3.2 shows the generalization ability of the previous model, without retraining, on new instances with 60, 80 and 100 variables (5000 instances for each size). We observe that our generic representation provides a better generalization than the problem-specific architecture for SAT and TSP. Notably, TSP-ELEM offers a far better generalization than TSP-EXT, confirming the impact of the input model and the size of the graph. Interestingly, we observe a strong generalization ability of the specific architecture for the knapsack. Our preliminary analysis indicates that this is achieved thanks to a GNN aggregation function based on a weighted sum. Understanding in detail the root causes of this generalization is part of our future work.

3.4.3 Discussion : Limitations and Challenges

Although the empirical results showed the promise of this generic architecture, there are still open challenges that must be addressed. First, the training time required to obtain such results is prohibitive, although we have considered only relatively small instances so far. This mainly occurs because our encoding generates large graphs. This opens the door to integrating compression methods, dedicated to shrinking the size of the encoding without losing information. A parallel can be done with the SMARTTABLE [64] constraints, which encodes tables more compactly. This also highlights the importance of having a good input model, yielding to a small embedding (e.g., TSP-ELEM versus TSP-EXT). Besides, the current task is restricted to solving the decision version of combinatorial problems with an end-to-end approach. The next step will be to challenge this architecture when integrated with a full-fledged solver, such as what has been proposed by Gasse et al. (2019) for binary mixed-integer programs [29], and by Cappart et al. (2021) for constraint programming [65]. Finally, the generalization ability of a model trained for a specific problem (e.g., on the TSP) to another but similar problems (e.g., TSP with time windows) is an interesting aspect to investigate.

3.5 Conclusion and Perspectives

This paper introduced a first version of a generic procedure to encode arbitrary combinatorial problems into a graph for learning-based approaches. The encoding proposed is injective, meaning that each encoding can be obtained by only one combinatorial problem. Besides, a tailored graph neural network has been proposed to learn from this encoding. Experimental results showed that our approach could achieve similar results as problem-specific architectures, without requiring the need to hand-craft a dedicated representation. All the constraints involved in the 2023 mini-track competition of XCSP3 are currently supported. Adding new constraints only requires the implementation of a parser. Our next steps are to challenge the approach on bigger and more complex problems, and on other combinatorial tasks (e.g., learning branching heuristics).

CHAPITRE 4 DÉTAIL DE LA CONSTRUCTION DES GRAPHS

Ce chapitre explique dans le détail comment chaque type de noeud intervient dans la conception de graphes associés à des problèmes combinatoires. Tout au long de ce chapitre, nous utiliserons le terme "caractéristique" pour désigner le vecteur d'attributs associé aux différents noeuds dans le graphe. Autrement dit, nous traduirons le terme *feature* par caractéristique. Nous emploierons aussi le terme encodage pour traduire *embedding*.

4.1 Conventions utilisées dans ce chapitre

Afin de faciliter la compréhension, des représentations visuelles des graphes sont incluses dans ce chapitre. Comme les types de noeuds de chaque graphes se répètent et de manière à alléger le texte, nous suivrons les conventions suivantes pour les types de noeuds. Les noeuds de type valeur seront représenté en vert, tel qu'illustré dans la figure 4.1a. Les noeuds de type variable seront représentés en rouge tel qu'illustré dans la figure 4.1b. Les noeuds de type opérateur seront représentés en orange tel qu'illustré dans la figure 4.1c. Les noeuds de type contrainte seront représentés en bleu tel qu'illustré dans la figure 4.1d. Les noeuds de type modèle seront représentés en jaune tel qu'illustré dans la figure 4.1e.

4.2 Types de noeuds

4.2.1 Modèle

Il n'y a qu'un seul noeud de ce type par graphe. Ce sommet est connecté à tous les noeuds de type contrainte ainsi qu'à tous les noeuds de type impliqués dans la fonction objectif du modèle. Sa fonction est de rassembler des informations supplémentaires sur le problème combinatoire au moyen de son vecteur de caractéristiques.

4.2.2 Valeurs

Nous créons un nœud de type valeur pour chaque valeur qui peut être prise par les variables. Les nœuds de type valeur sont connectés aux variables qui peuvent prendre ces valeurs et peuvent être soit des entiers, soit des valeurs réelles. Les valeurs entières sont supposées être catégorielles. Si toutes les valeurs sont du même type -si toutes les valeurs sont des entiers, par exemple-, les caractéristiques initiales de ces nœuds ne sont que la caractéristique de la valeur en question. Sinon, les caractéristiques seront stockées dans la dimension réservée pour le type de valeur. Les valeurs



(a) Noeud de type valeur



(b) Noeud de type variable



(c) Noeud de type opérateur



(d) Noeud de type contrainte



(e) Noeud de type modèle

FIGURE 4.1 Types de noeuds

entières partageront toutes la même caractéristique initiale de 1, tandis que celles à valeur réelle auront leur valeur numérique comme caractéristique. Illustrons ceci avec un exemple : dans un problème de trois variables entières x_1, x_2, x_3 , avec les domaines suivants :

$$— x_1 \in \{1, 2\}$$

$$— x_2 \in \{2, 3\}$$

$$— x_3 \in \{3, 4\}$$

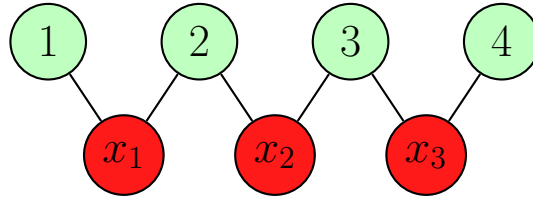


FIGURE 4.2 Sous-graphe variable-valeur

Comme toutes les valeurs sont entières, elles auront toutes la même caractéristique initiale de [1].

Regardons maintenant un exemple contenant des variables entières et réelles :

$$— x_1 \in \{1, 2\} \text{ Variable entière}$$

$$— x_2 \in \{3.0, 4.0\} \text{ Variable réelle}$$

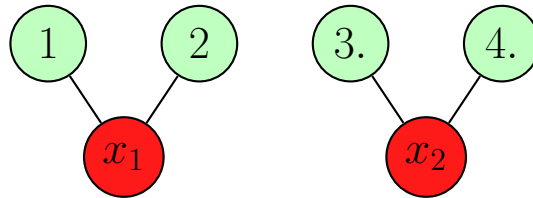


FIGURE 4.3 Sous-graphe variable-valeur

Comme nous avons à la fois des variables entières et réelles, les caractéristiques initiales des nœuds seront :

- Nœud 1 : [1, 0] le "1" dans la première dimension indique qu'il s'agit d'une variable entière
- Nœud 2 : [1, 0] le "1" dans la première dimension indique qu'il s'agit d'une variable entière
- Nœud 3 : [0, 3] la valeur numérique est inscrite dans la dimension 2
- Nœud 4 : [0, 4] la valeur numérique est inscrite dans la dimension 2

4.2.3 Opérateurs

Ces nœuds représentent des modifications ou des restrictions sur les variables impliquées dans les contraintes d'un problème. Les exemples d'opérateurs incluent les opérateurs arithmétiques/logiques

- $\neg$, $\times$, $+$, etc. - ainsi que les opérateurs d'assignation : l'opérateur d'assignation utilisé pour lier une valeur à une variable dans un tuple d'extension ou l'opérateur de tuple utilisé pour combiner plusieurs assignations variable-valeur pour créer un tuple d'extension. On note que les opérateurs de comparaison ($\leq$, $\geq$, $=$, etc.) sont catégorisés comme des contraintes dans le cas des contraintes d'intension, mais comme des opérateurs dans les contraintes de somme. Ceci sera abordé dans la section suivante. À moins que l'opérateur n'ait une signification numérique naturelle, les caractéristiques initiales pour ces nœuds sont un encodage de type *one-hot* du type d'opérateur. Par exemple, le problème SAT à 2 variables suivant :

$$(\neg x_1 \vee x_2) \wedge (x_1 \vee \neg x_2)$$

donnerait le sous-graphe variable-contrainte-opérateur, illustré dans la Figure 4.4.

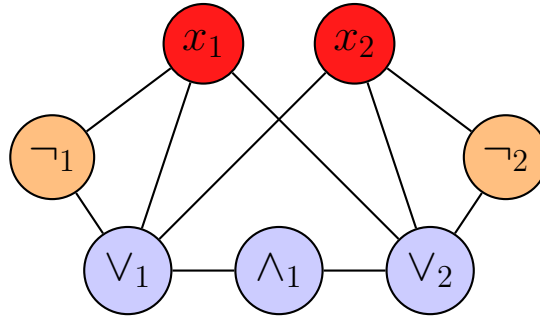


FIGURE 4.4 Sous-graphe Contrainte-Variable-Opérateur pour $(\neg x_1 \vee x_2) \wedge (x_1 \vee \neg x_2)$

Dans ce cas, comme il n'y a qu'un seul type d'opérateurs dans le graphe, les opérateurs " $\neg_1$ " et " $\neg_2$ " auraient tous les deux les caractéristiques initiales [1].

Lorsque les opérateurs ont une signification numérique naturelle, elle est encodée dans leurs caractéristiques initiales. Par exemple, $3x_1$ est représenté dans la Fig. 4.5.

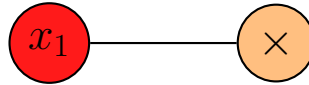


FIGURE 4.5 Sous-graphe Variable-Opérateur pour $3x_1$

Dans ce cas, l'encodage initial de l'opérateur " $\times$ " serait [3], car il s'agit d'une multiplication par 3.

4.2.4 Contraintes

Les contraintes imposent des restrictions sur les combinaisons de variables. Ces nœuds sont connectés aux variables, ainsi qu'à tout opérateur impliqué. Les variables sont toujours directement connec-

tées aux contraintes dans lesquelles elles apparaissent. Comme pour les opérateurs, à moins que l'opérateur n'ait une signification numérique naturelle, les caractéristiques initiales de ces nœuds sont un encodage de type *one-hot* du type d'opérateur. Pour les contraintes d'intension où il y a des côtés à une équation, nous utilisons un opérateur "côté droit", indiquant que certaines variables sont sur différents côtés d'une équation. Par exemple, la représentation graphique de la contrainte $3x_1 \leq 4x_2$ est illustrée dans la figure 4.6.

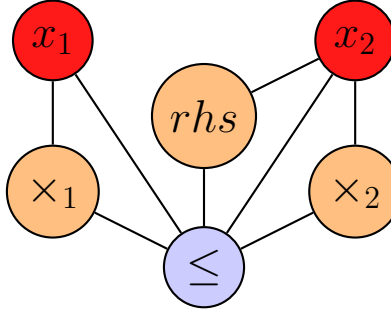


FIGURE 4.6 Sous-graphe pour $3x_1 \leq 4x_2$

Dans cet exemple, les caractéristiques initiales pour les nœuds seraient :

- | | | |
|-----------------------|-----------------------|---------------|
| — $rhs : [1, 0]$ | — $\times_2 : [0, 4]$ | — $x_1 : [1]$ |
| — $\times_1 : [0, 3]$ | — $\leq : [1]$ | — $x_2 : [1]$ |

4.3 Contraintes supportées

Comme point de départ, nous souhaitons prendre en charge certaines contraintes afin de construire des graphes génériques. Puisque nous voulons que les modèles soient génériques, nous avons sélectionné le formalisme XCSP3 [11] comme premières contraintes que nous allons supporter. À la date de rédaction de ce document, nous prenons désormais en charge toutes les contraintes de la mini-track XCSP3 2023. Les contraintes et les représentations graphiques associées sont énumérées dans cette section.

4.3.1 Contrainte d'intension binaire

La contrainte d'intension binaire dans XCSP3-mini permet les contraintes suivantes. Dans les descriptions, x , y et z désignent des variables entières, k désigne une valeur entière, $\odot$ désigne un opérateur relationnel dans $\{<, \leq, \geq, >, =, \neq\}$, et $\oplus$ désigne un opérateur arithmétique binaire dans $\{+, -, \cdot, /, \%, ||\}$, avec $||$ étant la distance.

Nous supportons plusieurs cas de comparaisons impliquant des variables et des constantes. Ces cas se déclinent de la manière suivante :

Premièrement, nous examinons des situations où une variable est comparée à une constante. Cela se manifeste de deux manières : soit la variable x est comparée à la constante k (noté $x \odot k$), soit c'est l'inverse, la constante k comparée à la variable x (noté $k \odot x$).

Ensuite, nous traitons le cas où une variable est comparée à une autre variable, ce qui se traduit par $x \odot y$.

Par ailleurs, nous analysons les cas où une expression combinant une variable, un opérateur, et une constante est comparée à une variable. Ces situations se présentent sous quatre formes :

1. $(x \oplus k) \odot y$,
2. $(k \oplus x) \odot y$,
3. $x \odot (y \oplus k)$,
4. $x \odot (k \oplus y)$.

Enfin, nous considérons le cas où une expression mélangeant deux variables avec un opérateur est comparée à une variable. Ceci se décline en deux configurations :

1. $(x \oplus y) \odot z$,
2. $x \odot (y \oplus z)$.

Ces différents cas représentent les diverses manières dont les variables et les constantes peuvent interagir dans nos comparaisons.

La représentation des contraintes en intension s'articule autour de plusieurs éléments clés. Tout d'abord, l'opérateur de comparaison est incarné par un nœud de type "contrainte". Parallèlement, un nœud de type "opérateur" est utilisé pour représenter l'opérateur arithmétique d'addition. Concernant les variables, elles sont connectées directement à l'opérateur de comparaison et, au besoin, à un opérateur arithmétique également. Les constantes, quant à elles, sont des caractéristiques des opérateurs. À titre d'exemple, une contrainte telle que $x + 2 \leq 4$ serait illustrée tel qu'indiqué dans la figure 4.7. Enfin, si la situation l'exige, un nœud opérateur est mis en place pour relier la variable du côté droit à l'opérateur de comparaison.

Une représentation visuelle des différents cas de figure est donnée dans la figure 4.8.

4.3.2 Contrainte en extension

La contrainte extension -aussi connue sous le nom de contrainte table- est composée de tuples, tous liés à la contrainte. D'abord, nous aborderons la contrainte extension de base, puis nous traiterons de la contrainte *short table* [56].

Caractéristiques :

1. $x : [1]$
2. $+: [2]$
3. $\leq : [4]$

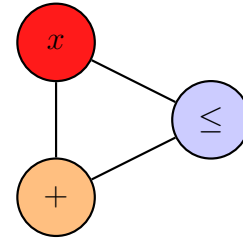


FIGURE 4.7 Sous-graphe générique pour $x + 2 \leq 4$

Tuples

Comme la contrainte extension est une restriction des domaines, les nœuds de type valeur sont connectés aux nœuds de tuple (sous-type d'opérateur). Par exemple, supposons que nous ayons le problème suivant :

$$\begin{aligned}
 &3 \text{ variables entières : } x_1, x_2, x_3 \\
 &x_1 \in \{1, 2, 3\} \\
 &x_2 \in \{2, 3, 4\} \\
 &x_3 \in \{3, 4, 5\}
 \end{aligned} \tag{4.1}$$

Ceci peut être représenté à l'aide du tableau suivant :

TABLEAU 4.1 Contrainte en extension

x_1	x_2	x_3
1	2	3
2	3	4
3	4	5

Graphiquement, ce problème est représenté dans la figure 4.9

Negative Table

Les contraintes *negative table* sont gérées exactement de la même manière. Si une contrainte table et une contrainte *negative table* sont toutes deux présentes dans le même modèle, elles seront codées avec des caractéristiques de départ différentes - avec un encodage de type *one-hot* différent de leur type.

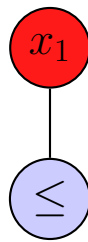
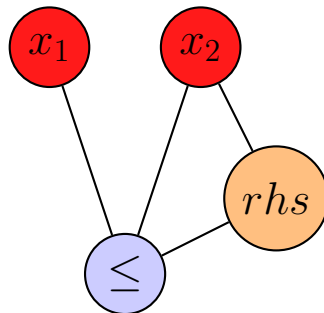
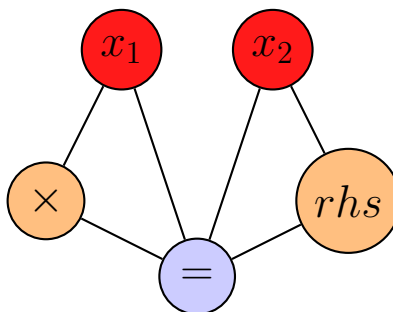
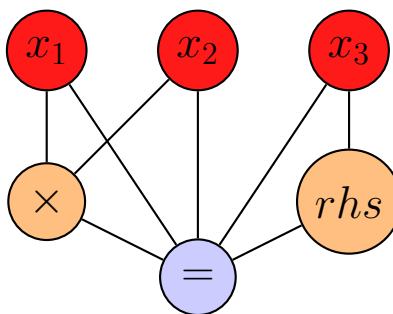
(a) Cas 1 : $x_1 \leq 4$ (b) Cas 2 : $x_1 \leq x_2$ (c) Cas 3 : $3x_1 = x_2$ (d) Cas 4 : $x_1 \cdot x_2 = x_3$

FIGURE 4.8 Sous-graphes génériques pour la contrainte intension binaire

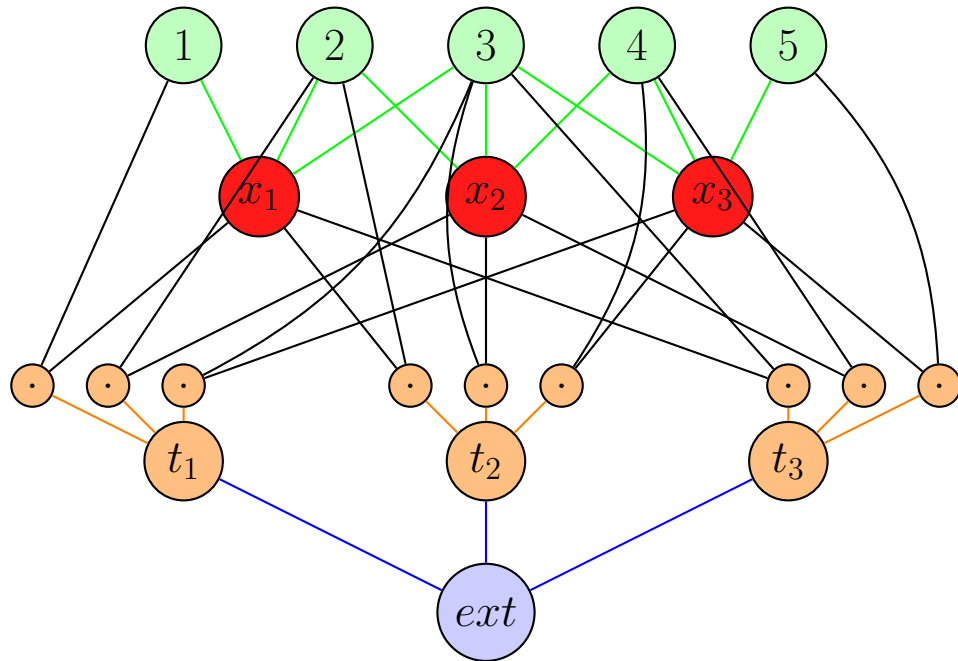


FIGURE 4.9 Contrainte table

Short table

La *short table* est représenté comme la contrainte de tableau, mais sans connexion à la variable qui peut prendre des valeurs arbitraires, car son domaine est illimité. Illustrons la contrainte représentée par le tableau suivant.

TABLEAU 4.2 Short table

x_1	x_2	x_3
*	2	3
2	*	4
3	4	*

Visuellement, cette contrainte est représentée dans la figure 4.10

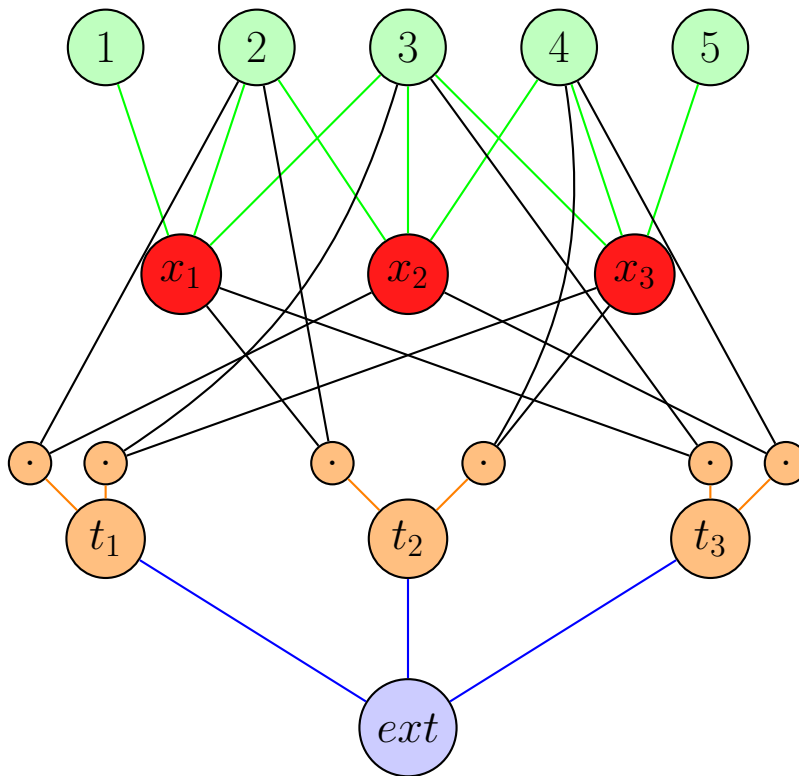


FIGURE 4.10 Contrainte short table

4.3.3 Contrainte de Somme

Les contraintes de somme sont des sommes pondérées de variables combinées avec un opérateur de comparaison. Les variables, du côté gauche de l'équation, sont connectées aux opérateurs de multiplication qui ont leurs valeurs numériques comme caractéristiques. Les opérateurs de multiplication sont connectés à la contrainte somme. L'opérateur de comparaison est connecté à la contrainte de somme. Si l'opérande de comparaison est une valeur numérique, alors l'opérateur de comparaison l'aura comme caractéristique. Sinon (par exemple, l'opérande est une variable), cette variable sera connectée à l'opérateur de comparaison. Par exemple, la contrainte $4x_1 + 3x_2 \leq x_3$ ainsi que les caractéristiques des noeuds du graphe sont décrits dans la figure 4.11 et la contrainte $5x_1 + 2x_2 \leq 3$ est décrite dans la figure 4.12.

Caractéristiques :

1. $x_1 : [1]$
2. $x_2 : [1]$
3. $x_3 : [1]$
4. $sum : [1]$
5. $\times_1 : [4, 0]$
6. $\times_2 : [3, 0]$
7. $\leq : [0, 1]$

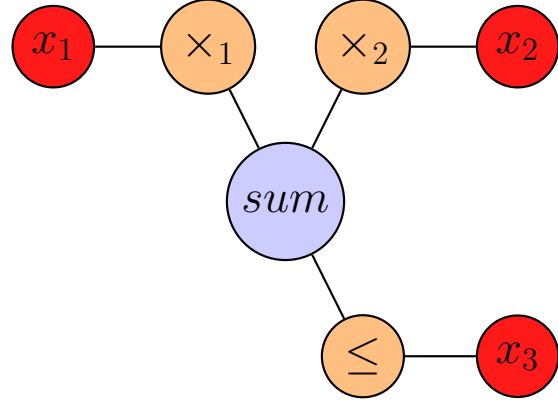


FIGURE 4.11 Sous-graphe générique pour $4x_1 + 3x_2 \leq x_3$

Caractéristiques :

1. $x_1 : [1]$
2. $x_2 : [1]$
3. $sum : [1]$
4. $\times_1 : [5, 0]$
5. $\times_2 : [2, 0]$
6. $\leq : [0, 3]$

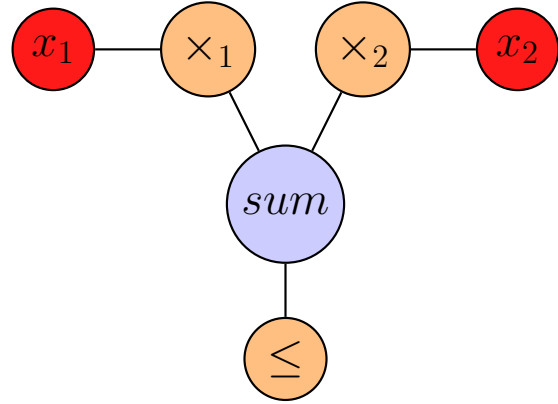


FIGURE 4.12 Sous-graphe générique pour $5x_1 + 2x_2 \leq 3$

4.3.4 Contrainte Élément

XCSP3-mini prend en charge 2 versions de la contrainte élément. La première affirme qu'un élément v est dans un ensemble $X = \{x_0, x_1, \dots\}$ si et seulement si il existe i tel que $0 \leq i < |X|$ et $x_i = v$. Formellement :

$$v \in X \iff \exists i : 0 \leq i < |X| \wedge x_i = v$$

Dans notre graphe, nous représenterons une telle contrainte en ayant un nœud de contrainte (élément). Nous connecterons ensuite le nœud de valeur qui doit être pris par l'une des variables au nœud de contrainte, par l'intermédiaire d'un nœud opérateur (sous-type valeur). Finalement, les variables seront connectées à la contrainte par l'intermédiaire de nœuds opérateurs de (sous-type

"attribution de valeur") Graphiquement, la contrainte élément où l'un de x_1, x_2 doit prendre la valeur 4, aurait la représentation suivante illustrée dans la figure 4.13.

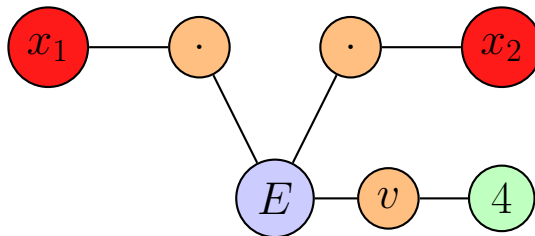


FIGURE 4.13 Contrainte élément basique

La deuxième version de la contrainte élément affirme qu'un élément à l'indice i dans l'ensemble $X = \{x_0, x_1, \dots\}$ a la valeur k si et seulement si $x_i = k$.

$$v \in X \iff \exists i : 0 \leq i < |X| \wedge x_i = v$$

La représentation graphique de ce second cas de figure est dans la figure 4.14. Dans notre représentation graphique, nous aurons un nœud opérateur indiquant l'indice $-i$ dans la représentation graphique- qui sera relié à l'indice i ainsi qu'à la contrainte élément $-E$. Enfin, un nœud opérateur pour l'opérateur de comparaison utilisé dans la contrainte serait créé et connecté au nœud de valeur et au nœud de contrainte. Graphiquement, la contrainte élément où l'élément à l'indice i de x_1, x_2 doit prendre la valeur 4, aurait la représentation ci-bas.

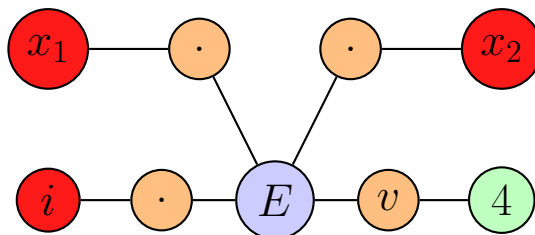


FIGURE 4.14 Contrainte élément avec indice

CHAPITRE 5 CONCLUSION

5.1 Synthèse des travaux

Ce mémoire se concentre sur la création d’une représentation générique pour les problèmes combinatoires, adaptée aux approches basées sur l’apprentissage. Cette recherche propose une méthode pour encoder ces problèmes en structures compatibles avec les algorithmes d’apprentissage, et une architecture dédiée de réseau de neurones sur graphes (GNNs). La thèse aborde la complexité de ces problèmes et les défis liés à leur représentation. Elle introduit une architecture de GNN capable d’apprendre efficacement à partir de ces représentations, gérant diverses contraintes dans les problèmes combinatoires complexes. L’efficacité de cette approche est validée par des résultats expérimentaux sur plusieurs types de problèmes, démontrant des performances comparables aux architectures dédiées tout en offrant une polyvalence accrue.

5.2 Limitations de la solution proposée et pistes de travaux futurs

Bien que les résultats empiriques démontrent le potentiel de cette architecture générique, il existe des défis à relever. Premièrement, le temps d’entraînement nécessaire pour obtenir de tels résultats est prohibitif, d’autant plus que les instances utilisées pour ce mémoire demeurent relativement petites. Ceci s’explique entre autres par la taille des graphes générés pour représenter les problèmes et suggère que l’intégration de méthodes de compression pour réduire la taille de l’encodage sans perdre d’information serait grandement bénéfique et permettrait une meilleure adoption de ces méthodes. Une piste future semblable à la contrainte SMARTTABLE [64] pourrait être envisagée. On note ensuite qu’il est également important d’avoir un bon modèle d’entrée pour un encodage plus petit, tel que démontré par les différences de performance entre les modèles en extension et ELEMENT pour le TSP. Une étude plus approfondie des performances pour différents modèles pourrait être envisagée pour clarifier ce point.

Ensuite, ces travaux sont basés sur un modèle d’apprentissage seulement. Afin d’étendre les méthodes d’apprentissage, il serait pertinent de comparer différents types de GNN pour identifier les plus performants. Sur ce point, il serait intéressant d’étudier la performance des GNNs basés sur le raisonnement algorithmique neuronal [66]. Cette dernière approche a récemment été tentée par Georgiev et al (2023) [67] et pourrait être étudiée plus en profondeur.

En outre, la tâche actuelle est limitée à la résolution de la version décisionnelle des problèmes combinatoires avec une approche de bout en bout (*end-to-end*). L’étape suivante consistera à intégrer cette architecture avec un solveur complet, comme proposé par Gasse et al. (2019) [29]

pour les programmes mixtes binaires, et par Cappart et al. (2021) [10] pour la programmation par contraintes.

On note ensuite que les GNNs présentés dans cette recherche sont entraînés et adaptés spécifiquement pour la classification binaire sur les versions décisionnelles de problèmes combinatoires de petite taille. De ceci découlent plusieurs limites. Premièrement, les modèles sont entraînés sur des tâches qui sont d’une utilité limitée à un contexte de recherche. Deuxièmement, la tâche de classification binaire a peu d’applications pratiques, que ce soit en recherche ou dans l’industrie. Afin d’en étendre l’applicabilité, il faudrait un travail d’ingénierie pour revoir le design des GNNs pour qu’ils soient adaptés à des tâches plus génériques. Finalement, on souligne que le temps d’entraînement nécessaire pour atteindre les performances rapportées dans ce travail rend cette technique prohibitive : certains modèles prennent jusqu’à 4 jours pour être entraînés. Afin d’étendre l’utilité de cette recherche, il faudrait réviser la création des graphes en les compressant pour permettre un apprentissage plus rapide. Une piste de recherche dans cette direction serait le modèle spécifique au problème du sac à dos décrit dans [48]. Cette architecture de GNN introduit des agrégations avec des sommes pondérées et semble généraliser très bien à des instances de grande taille en plus de réduire la taille des graphes. Une autre avenue potentielle serait de paralléliser l’entraînement des modèles sur plusieurs GPUs.

Finalement, la capacité de généralisation d’un modèle formé pour un problème spécifique à d’autres problèmes similaires est un aspect intéressant à étudier. L’objectif ultime de cette recherche est d’avoir une manière unifiée de représenter ces problèmes afin d’entraîner des modèles génériques pour les problèmes combinatoires et l’étude de la généralisation inter-problème permettrait d’explorer ceci.

Cette recherche introduit une procédure générique pour encoder les problèmes combinatoires en graphes pour des approches basées sur l’apprentissage. L’encodage proposé est injectif et un réseau neuronal graphique sur mesure a été proposé pour apprendre de cet encodage. Les résultats expérimentaux montrent que cette approche peut atteindre des résultats similaires à ceux des architectures spécifiques au problème, sans nécessiter une représentation dédiée. Toutes les contraintes du concours mini-track 2023 de XCSP3 sont actuellement prises en charge. L’ajout de nouvelles contraintes nécessite uniquement l’implémentation d’un analyseur. Les prochaines étapes consisteront à tester l’approche sur des problèmes plus grands et complexes, et sur d’autres tâches combinatoires.

RÉFÉRENCES

- [1] D. L. Applegate, R. E. Bixby, V. Chvátal, W. Cook, D. G. Espinoza, M. Goycoolea et K. Helsgaun, “Certification of an optimal tsp tour through 85,900 cities,” *Operations Research Letters*, vol. 37, n^o. 1, p. 11–15, Jan 2009.
- [2] J. Marques-Silva, I. Lynce et S. Malik, *Chapter 4 : Conflict-driven clause learning SAT solvers*, ser. Frontiers in Artificial Intelligence and Applications. IOS Press BV, 2021, p. 133–182, publisher Copyright : © 2021 The authors and IOS Press. All rights reserved.
- [3] K. Helsgaun, *An Extension of the Lin-Kernighan-Helsgaun TSP Solver for Constrained Traveling Salesman and Vehicle Routing Problems : Technical report*. Roskilde Universitet, déc. 2017.
- [4] M. Gendreau, J.-Y. Potvin *et al.*, *Handbook of metaheuristics*. Springer, vol. 2.
- [5] I. Goodfellow, Y. Bengio et A. Courville, *Deep learning*, 2016.
- [6] T. Brown, B. Mann, N. Ryder, M. Subbiah, J. D. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell *et al.*, “Language models are few-shot learners,” *Advances in neural information processing systems*, vol. 33, p. 1877–1901, 2020.
- [7] A. Krizhevsky, I. Sutskever et G. E. Hinton, “Imagenet classification with deep convolutional neural networks,” dans *Advances in Neural Information Processing Systems*, vol. 25. Curran Associates, Inc., 2012. [En ligne]. Disponible : https://proceedings.neurips.cc/paper_files/paper/2012/hash/c399862d3b9d6b76c8436e924a68c45b-Abstract.html
- [8] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski *et al.*, “Human-level control through deep reinforcement learning,” *nature*, vol. 518, n^o. 7540, p. 529–533, 2015.
- [9] 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^o. 2, p. 405–421, 2021.
- [10] Q. Cappart, D. Chételat, E. B. Khalil, A. Lodi, C. Morris et P. Velickovic, “Combinatorial optimization and reasoning with graph neural networks,” *Journal of Machine Learning Research*, vol. 24, n^o. 130, p. 1–61, 2023.
- [11] F. Boussemart, C. Lecoutre, G. Audemard et C. Piette, “Xcsp3-core : A format for representing constraint satisfaction/optimization problems,” *arXiv preprint arXiv :2009.00514*, 2020.
- [12] S. A. Cook, “The complexity of theorem-proving procedures,” dans *Proceedings of the Third Annual ACM Symposium on Theory of Computing*, ser. STOC ’71. New York, NY,

- USA : Association for Computing Machinery, 1971, p. 151–158. [En ligne]. Disponible : <https://doi.org/10.1145/800157.805047>
- [13] D. R. Morrison, S. H. Jacobson, J. J. Sauppe et E. C. Sewell, “Branch-and-bound algorithms : A survey of recent advances in searching, branching, and pruning,” *Discrete Optimization*, vol. 19, p. 79–102, 2016.
- [14] D. Henderson, S. H. Jacobson et A. W. Johnson, *The Theory and Practice of Simulated Annealing*. Boston, MA : Springer US, 2003, p. 287–319. [En ligne]. Disponible : https://doi.org/10.1007/0-306-48056-5_10
- [15] M. Gendreau et J.-Y. Potvin, *Tabu Search*. Boston, MA : Springer US, 2005, p. 165–186. [En ligne]. Disponible : https://doi.org/10.1007/0-387-28356-0_6
- [16] S. Katoch, S. S. Chauhan et V. Kumar, “A review on genetic algorithm : past, present, and future,” *Multimedia Tools and Applications*, vol. 80, n^o. 5, p. 8091–8126, févr. 2021.
- [17] T. M. Shami, A. A. El-Saleh, M. Alswaiti, Q. Al-Tashi, M. A. Summakieh et S. Mirjalili, “Particle swarm optimization : A comprehensive survey,” *IEEE Access*, vol. 10, p. 10 031–10 061, 2022.
- [18] T. Stützle et R. Ruiz, *Iterated Local Search*. Cham : Springer International Publishing, 2018, p. 579–605. [En ligne]. Disponible : https://doi.org/10.1007/978-3-319-07124-4_8
- [19] B. Omrani, “Apprentissage par renforcement d’heuristiques de branchement en programmation par contraintes,” Thèse de doctorat, Polytechnique Montréal, 2021.
- [20] K. Hornik, M. Stinchcombe et H. White, “Multilayer feedforward networks are universal approximators,” *Neural Networks*, vol. 2, n^o. 5, p. 359–366, 1989.
- [21] B. C. Csáji *et al.*, “Approximation with artificial neural networks.”
- [22] D. Rwerth, G. Hinton et R. Williams, “Learning representations by back-propagating error,” *Nature*, vol. 323, n^o. 9, p. 533–536, 1986.
- [23] W. L. Hamilton, *Graph representation learning*. Morgan & Claypool Publishers, 2020.
- [24] A. Sperduti et A. Starita, “Supervised neural networks for the classification of structures,” *IEEE Transactions on Neural Networks*, vol. 8, n^o. 3, p. 714–735, 1997.
- [25] F. Scarselli, M. Gori, A. C. Tsoi, M. Hagenbuchner et G. Monfardini, “The graph neural network model,” *IEEE Transactions on Neural Networks*, vol. 20, n^o. 1, p. 61–80, Jan 2009.
- [26] Y. Wu, D. Lian, Y. Xu, L. Wu et E. Chen, “Graph convolutional networks with markov random field reasoning for social spammer detection,” *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 34, n^o. 01, p. 1054–1061, avr. 2020.
- [27] J. Jumper, R. Evans, A. Pritzel, T. Green, M. Figurnov, O. Ronneberger, K. Tunyasuvunakool, R. Bates, A. Žídek, A. Potapenko *et al.*, “Highly accurate protein structure prediction with alphafold,” *Nature*, vol. 596, n^o. 7873, p. 583–589, 2021.

- [28] S. Wu, F. Sun, W. Zhang, X. Xie et B. Cui, “Graph neural networks in recommender systems : a survey,” *ACM Computing Surveys*, vol. 55, n°. 5, p. 1–37, 2022.
- [29] M. Gasse, D. Chetelat, N. Ferroni, L. Charlin et A. Lodi, “Exact combinatorial optimization with graph convolutional neural networks,” dans *Advances in Neural Information Processing Systems*, vol. 32. Curran Associates, Inc., 2019. [En ligne]. Disponible : <https://proceedings.neurips.cc/paper/2019/hash/d14c2267d848abeb81fd590f371d39bd-Abstract.html>
- [30] J. Gilmer, S. S. Schoenholz, P. F. Riley, O. Vinyals et G. E. Dahl, “Neural message passing for quantum chemistry,” dans *International conference on machine learning*. PMLR, 2017, p. 1263–1272.
- [31] J. J. Hopfield et D. W. Tank, ““neural” computation of decisions in optimization problems,” *Biological Cybernetics*, vol. 52, n°. 3, p. 141–152, Jul 1985.
- [32] O. Vinyals, M. Fortunato et N. Jaitly, “Pointer networks,” dans *Advances in Neural Information Processing Systems*, vol. 28. Curran Associates, Inc., 2015. [En ligne]. Disponible : <https://papers.nips.cc/paper/2015/hash/29921001f2f04bd3baee84a12e98098f-Abstract.html>
- [33] I. Bello, H. Pham, Q. V. Le, M. Norouzi et S. Bengio, “Neural combinatorial optimization with reinforcement learning,” *ArXiv*, vol. abs/1611.09940, 2016. [En ligne]. Disponible : <https://api.semanticscholar.org/CorpusID:3649804>
- [34] D. Applegate, R. Bixby, V. Chvatal et W. Cook, “Concorde tsp solver,” 2006.
- [35] E. Khalil, P. L. Bodic, L. Song, G. Nemhauser et B. Dilkina, “Learning to branch in mixed integer programming,” *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 30, n°. 11, févr. 2016. [En ligne]. Disponible : <https://ojs.aaai.org/index.php/AAAI/article/view/10080>
- [36] A. Lodi et G. Zarpellon, “On learning and branching : a survey,” *Top*, vol. 25, p. 207–236, 2017.
- [37] E. Khalil, H. Dai, Y. Zhang, B. Dilkina et L. Song, “Learning combinatorial optimization algorithms over graphs,” dans *Advances in Neural Information Processing Systems*, vol. 30. Curran Associates, Inc., 2017. [En ligne]. Disponible : <https://proceedings.neurips.cc/paper/2017/hash/d9896106ca98d3d05b8cbdf4fd8b13a1-Abstract.html>
- [38] W. Kool, H. van Hoof et M. Welling, “Attention, learn to solve routing problems!” dans *7th International Conference on Learning Representations, ICLR 2019, New Orleans, LA, USA, May 6-9, 2019*. OpenReview.net, 2019. [En ligne]. Disponible : <https://openreview.net/forum?id=ByxBFsRqYm>
- [39] P. Veličković, G. Cucurull, A. Casanova, A. Romero, P. Liò et Y. Bengio, “Graph attention networks,” dans *International Conference on Learning Representations*, 2018.

- [40] R. S. Sutton, D. McAllester, S. Singh et Y. Mansour, “Policy gradient methods for reinforcement learning with function approximation,” dans *Advances in Neural Information Processing Systems*, vol. 12. MIT Press, 1999. [En ligne]. Disponible : https://papers.nips.cc/paper_files/paper/1999/hash/464d828b85b0bed98e80ade0a5c43b0f-Abstract.html
- [41] L. Perron et F. Didier, “Cp-sat,” Google. [En ligne]. Disponible : https://developers.google.com/optimization/cp/cp_solver/
- [42] Z.-H. Fu, K.-B. Qiu et H. Zha, “Generalize a small pre-trained model to arbitrarily large tsp instances,” dans *Proceedings of the AAAI conference on artificial intelligence*, vol. 35, n^o. 8, 2021, p. 7474–7482.
- [43] R. Palm, U. Paquet et O. Winther, “Recurrent relational networks,” dans *Advances in Neural Information Processing Systems*, vol. 31. Curran Associates, Inc., 2018. [En ligne]. Disponible : https://proceedings.neurips.cc/paper_files/paper/2018/hash/b9f94c77652c9a76fc8a442748cd54bd-Abstract.html
- [44] D. Selsam, M. Lamm, B. Bünz, P. Liang, L. de Moura et D. L. Dill, “Learning a sat solver from single-bit supervision,” 2018.
- [45] D. Selsam et N. Bjørner, “Guiding high-performance sat solvers with unsat-core predictions,” dans *Theory and Applications of Satisfiability Testing—SAT 2019 : 22nd International Conference, SAT 2019, Lisbon, Portugal, July 9–12, 2019, Proceedings 22*. Springer, 2019, p. 336–353.
- [46] M. Prates, P. H. Avelar, H. Lemos, L. C. Lamb et M. Y. Vardi, “Learning to solve np-complete problems : A graph neural network for decision tsp,” dans *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 33, 2019, p. 4731–4738.
- [47] H. Lemos, M. Prates, P. Avelar et L. Lamb, “Graph colouring meets deep learning : Effective graph neural network models for combinatorial problems,” dans *2019 IEEE 31st International Conference on Tools with Artificial Intelligence (ICTAI)*. IEEE, 2019, p. 879–885.
- [48] M. Liu, F. Zhang, P. Huang, S. Niu, F. Ma et J. Zhang, “Learning the satisfiability of pseudo-boolean problem with graph neural networks,” dans *Principles and Practice of Constraint Programming : 26th International Conference, CP 2020, Louvain-la-Neuve, Belgium, September 7–11, 2020, Proceedings 26*. Springer, 2020, p. 885–898.
- [49] D. Bahdanau, K. Cho et Y. Bengio, “Neural machine translation by jointly learning to align and translate,” dans *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*, Y. Bengio et Y. LeCun, édit., 2015. [En ligne]. Disponible : <http://arxiv.org/abs/1409.0473>
- [50] C. K. Joshi, Q. Cappart, L.-M. Rousseau et T. Laurent, “Learning the travelling salesperson problem requires rethinking generalization,” *Constraints*, vol. 27, n^o. 1-2, p. 70–98, 2022.

- [51] F. Chalumeau, I. Coulon, Q. Cappart et L.-M. Rousseau, “Seapearl : A constraint programming solver guided by reinforcement learning,” dans *Integration of Constraint Programming, Artificial Intelligence, and Operations Research : 18th International Conference, CPAIOR 2021, Vienna, Austria, July 5–8, 2021, Proceedings 18*. Springer, 2021, p. 392–409.
- [52] T. Marty, T. François, P. Tessier, L. Gautier, L.-M. Rousseau et Q. Cappart, “Learning a generic value-selection heuristic inside a constraint programming solver,” dans *29th International Conference on Principles and Practice of Constraint Programming (CP 2023)*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2023.
- [53] J. Tönshoff, B. Kisin, J. Lindner et M. Grohe, “One model, any csp : Graph neural networks as fast global search heuristics for constraint satisfaction,” 2022.
- [54] J.-C. Régin, “A filtering algorithm for constraints of difference in CSPs,” dans *AAAI*, vol. 94, 1994, p. 362–367.
- [55] J. Demeulenaere, R. Hartert, C. Lecoutre, G. Perez, L. Perron, J.-C. Régin et P. Schaus, “Compact-table : efficiently filtering table constraints with reversible sparse bit-sets,” dans *Principles and Practice of Constraint Programming : 22nd International Conference, CP 2016, Toulouse, France, September 5-9, 2016, Proceedings 22*. Springer, 2016, p. 207–223.
- [56] H. Verhaeghe, C. Lecoutre et P. Schaus, “Extending compact-table to negative and short tables,” dans *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 31, 2017.
- [57] Y. Li, R. Zemel, M. Brockschmidt et D. Tarlow, “Gated graph sequence neural networks,” dans *International Conference on Learning Representations*, 2016.
- [58] F. Monti, D. Boscaini, J. Masci, E. Rodola, J. Svoboda et M. M. Bronstein, “Geometric deep learning on graphs and manifolds using mixture model cnns,” dans *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2017, p. 5115–5124.
- [59] J. L. Ba, J. R. Kiros et G. E. Hinton, “Layer normalization,” *arXiv preprint arXiv :1607.06450*, 2016.
- [60] S. Hochreiter et J. Schmidhuber, “Long short-term memory,” *Neural computation*, vol. 9, p. 1735–80, Dec 1997.
- [61] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimeshine, L. Antiga *et al.*, “Pytorch : An imperative style, high-performance deep learning library,” *Advances in neural information processing systems*, vol. 32, 2019.
- [62] M. Fey et J. E. Lenssen, “Fast graph representation learning with pytorch geometric,” *arXiv preprint arXiv :1903.02428*, 2019.
- [63] D. P. Kingma et J. Ba, “Adam : A method for stochastic optimization,” dans *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015*,

- Conference Track Proceedings*, Y. Bengio et Y. LeCun, édit., 2015. [En ligne]. Disponible : <http://arxiv.org/abs/1412.6980>
- [64] J.-B. Mairy, Y. Deville et C. Lecoutre, “The smart table constraint,” dans *Integration of AI and OR Techniques in Constraint Programming : 12th International Conference, CPAIOR 2015, Barcelona, Spain, May 18-22, 2015, Proceedings 12*. Springer, 2015, p. 271–287.
 - [65] Q. Cappart, T. Moisan, L.-M. Rousseau, I. Prémont-Schwarz et A. A. Cire, “Combining reinforcement learning and constraint programming for combinatorial optimization,” dans *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 35, n^o. 5, 2021, p. 3677–3687.
 - [66] P. Veličković et C. Blundell, “Neural algorithmic reasoning,” *Patterns*, vol. 2, n^o. 7, p. 100273, juill. 2021, arXiv :2105.02761 [cs, math, stat].
 - [67] D. Georgiev, D. Numeroso, D. Bacciu et P. Liò, “Neural algorithmic reasoning for combinatorial optimisation,” *arXiv preprint arXiv :2306.06064*, 2023.