Titre: Title:	Imitation du branchement fort pour les problèmes de rotations d'équipage
Auteur: Author:	Pierre Pereira
Date:	2022
Type:	Mémoire ou thèse / Dissertation or Thesis
Référence: Citation:	Pereira, P. (2022). Imitation du branchement fort pour les problèmes de rotations d'équipage [Master's thesis, Polytechnique Montréal]. PolyPublie. https://publications.polymtl.ca/10490/

POLYTECHNIQUE MONTRÉAL
affiliée à l'Université de Montréal

Imitation du branchement fort pour les problèmes de rotations d'équipage

PIERRE PEREIRA
Département de génie informatique et génie logiciel

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

Août 2022

POLYTECHNIQUE MONTRÉAL
affiliée à l'Université de Montréal

Ce mémoire intitulé :

Imitation du branchement fort pour les problèmes de rotations d'équipage

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

Quentin CAPPART, président
Daniel ALOISE, membre et directeur de recherche
François SOUMIS, membre et codirecteur de recherche
Claudio CONTARDO, membre externe

DÉDICACE

À ma famille, ma copine, et mes amis...

REMERCIEMENTS

Je remercie profondément *Daniel Aloise*, *François Soumis* et *Frédéric Quesnel* qui m'ont guidé lors de la réalisation de ce projet. Je souhaite également remercier *Yassine Yaakoubi* qui a bien voulu se greffer à ce projet afin d'apporter une aide supplémentaire très précieuse.

À mes amis rencontrés à Montréal, *Ilan*, *Guillaume*, *Alice*, *Enzo*, *Lucie*, *Tess*, *Matthieu*, *Etienne* et *Amaury*, je vous remercie de m'avoir apporté autant de chaleur lors de ce périple dans le froid canadien.

Je ne serai jamais arrivé jusqu'ici sans ma famille, pour qui j'éprouve un amour sans limite même si je ne l'exprime que trop rarement. Je souhaite aussi remercier la femme que j'aime, qui m'a accompagné durant ces deux années et qui a dû supporter une relation à distance que je lui ai imposé. Heureusement que tu étais bel et bien là lors de mes phases de doutes.

Ce mémoire est le fruit d'un travail réalisé en grande partie avec mon collègue et ami sincère *Emeric*. Je lui suis extrêmement reconnaissant pour tous les efforts et la bonne humeur qu'il a apporté, j'espère qu'il ne sous-estime pas son impact sur le résultat final de ce mémoire.

Aussi, je souhaite remercier mon école française Grenoble-INP ENSIMAG pour m'avoir permis de me lancer dans cette aventure canadienne.

Ce travail n'aurait jamais pu voir le jour sans l'aide et le support de mon entourage.

RÉSUMÉ

Les compagnies aériennes doivent résoudre des problèmes d'optimisations parmi les plus complexes en industrie. La planification de vol demande de respecter des multitudes de contraintes et met en jeu des ressources à la fois matérielles, humaines et économiques. C'est pourquoi il est crucial pour une compagnie de toujours chercher à améliorer les solveurs (logiciels de résolution de problèmes) chargés d'optimiser ces problèmes.

Un de ces problèmes est celui de la planification d'équipage, qui a pour but de déterminer un emploi du temps pour chaque personnel navigant de la compagnie. La résolution de ce problème permet de s'assurer qu'à chaque vol soit affecté un pilote et copilote tout en respectant les contraintes liées aux droits du travail des employés et aux préférences de chacun. Les équipages sont une source de dépense annuelle très importante pour une compagnie aérienne. Pour autant, il n'est actuellement pas possible de trouver une solution optimale à ce problème et c'est pourquoi il y a beaucoup d'investissements pour améliorer les solveurs afin de réduire les coûts des solutions obtenues.

Les meilleurs solveurs actuels se basent sur une recherche arborescente de solution : le *Branch-and-Bound*. Un arbre de recherche est construit, afin de parcourir l'espace des solutions du problème. Cependant, dans le cas de la planification d'équipage, le problème est trop gros pour se permettre de parcourir l'ensemble de l'espace des solutions. En pratique, on se contente d'explorer en profondeur une partie de cet espace afin d'obtenir une solution approximative du problème : on nomme cette méthode le *diving Branch-and-Bound*.

Le parcours en profondeur d'un tel arbre est guidé par une heuristique de branchement qui influence fortement la performance globale du solveur. L'heuristique de branchement doit prendre des décisions durant la résolution du problème afin de choisir comment parcourir l'arbre de recherche. La meilleure heuristique connue est le *strong branching*. Lorsque le solveur l'utilise il obtient en moyenne les solutions aux coûts les plus faibles. Cependant, l'utilisation de cette heuristique coûte cher en temps de calcul. En effet, elle augmente la quantité de sous-problèmes à résoudre durant le parcours de l'arbre ce qui allonge énormément les temps de calcul. C'est pourquoi en pratique ce n'est pas le strong branching qui est utilisé, mais le *branchement fractionnaire*. Ce dernier, lorsqu'il est utilisé dans un solveur ne permet pas d'obtenir les meilleures solutions approximatives mais il est beaucoup plus économe en temps de calcul (pour certains grands problèmes, il utilise de l'ordre de 4h de calculs contre 16h lorsque le strong branching est utilisé).

Le but de ce mémoire est de proposer une méthode qui construit de nouvelles heuristiques

de branchement afin d'améliorer la performance des solveurs des compagnies aériennes. Ces nouvelles heuristiques ont comme objectif de combiner la rapidité de calcul du branchement fractionnaire tout en prenant les bonnes décisions de branchement du strong branching. Pour cela, nous utilisons l'apprentissage automatique afin d'entraîner des modèles à imiter les décisions du strong branching. Ces modèles, une fois entraînés, sont beaucoup moins coûteux à utiliser que le strong branching et sont capables de prendre des décisions de branchements plus efficaces que le branchement fractionnaire. Nous étudions la faisabilité d'une telle méthode, les limites qu'elle implique et tentons d'aller au-delà de ce qui a été fait dans la littérature. Nous montrons que cette méthode permet effectivement de produire de nouvelles heuristiques de branchement qui, lorsqu'elles sont utilisées dans le solveur, donnent de meilleures solutions approximatives que celles obtenues par branchement fractionnaire tout en ayant un temps de résolution similaire.

ABSTRACT

Airlines have to solve crew pairing problems every months. These problems aim to find a good anonymous schedule so that every flights are assigned to pilots and copilots. Crew scheduling needs to verify geographical, temporal and administrative constraints while optimizing millions of variables, which is why only an approximate solution can be found in practice. These problems are approximately solved using a diving Branch-and-Price algorithm. The diving Branch-and-Price algorithm constructs a tree of solutions that is searched using a branching heuristic. The branching heuristic have a great impact on the final solution quality and on the total runtime of the algorithm. The best known heuristic is the strong branching, it raises the best approximate solutions but it is also computationally heavy. A better approach is to use the fractional branching, which is faster and still raises good solutions. In this work, we use machine learning to learn new branching heuristics based on the strong branching decisions. The goal is to have a branching heuristic as fast as the fractional branching while yielding better approximate solutions.

We study the possibility of using imitation learning to learn new branching heuristics with the objective of beating fractional branching based solvers. We compare the solutions and runtimes of our heuristics with the fractional and the strong branching heuristics. We also evaluate the impact of distributional shift on the heuristic performances as it could be a limit of the method. We show that imitation learning is a promising method to build new branching heuristics that can outperform the fractional branching solutions while having the same computational complexity.

TABLE DES MATIÈRES

LISTE DES TABLEAUX

LISTE DES FIGURES

LISTE DES SIGLES ET ABRÉVIATIONS

B&B	Branch-and-Bound
B&P	Branch-and-Price
CSP	Crew Scheduling Problem
CPP	Crew Pairing Problem
CAP	Crew Assignment Problem
MLP	Multilayer perceptron

CHAPITRE 1 INTRODUCTION

Les compagnies aériennes ont besoin de rapidement établir les emplois du temps de leurs pilotes et copilotes de lignes. Cependant c'est un problème très compliqué où seule une solution approximative est envisageable lorsqu'un grand nombre de vols est à prendre en compte (typiquement l'ensemble des vols sur 1 mois). Le solveur permettant de déduire ces solutions approximatives dépends notamment d'une heuristique qui influence beaucoup sur la qualité finale de la solution. Le but de ce projet de recherche est d'améliorer le solveur en apprenant une meilleure heuristique à l'aide d'apprentissage automatique.

1.1 Définitions et concepts de base

1.1.1 Programmation en nombres entiers

Le problème traité dans ce mémoire se formule à travers la programmation en nombres entiers. La programmation en nombres entiers donne un cadre rigoureux à un problème d'optimisation. Une fois modélisé, on peut appliquer divers algorithmes spécialisés pour résoudre le problème d'optimisation.

Programmation linéaire et programmation en nombres entiers

Le domaine de l'optimisation sous contraintes formule souvent les problèmes à travers un programme linéaire de la forme suivante :

$$\min_x \; c^T x \tag{1.1a}$$

$$\text{s.c. } Ax = b \tag{1.1b}$$

$$x \geq 0 \tag{1.1c}$$

où x est une variable multidimensionnelle à optimiser de sorte à minimiser les coûts décrits par le vecteur c (1.1a) tout en respectant les contraintes déterminés par A et b (1.1b).

On parle de programmation linéaire car la fonction objectif et les contraintes sont toutes les deux des fonctions linéaires de x. Ici x peut être n'importe quel réel non-négatif (1.1c). Cependant, dans le cadre de la programmation à nombres entiers la variable x à optimiser a comme contrainte supplémentaire qu'elle doit être *entière*. Il n'est pas possible de résoudre analytiquement un programme à nombre entiers. Cependant, on peut déterminer une borne

inférieure en résolvant une relaxation linéaire de ce dernier (on retire les contraintes associées à l'entièreté des variables), et une borne supérieure en trouvant une solution faisable entière.

Pour résoudre un programme linéaire, on utilise habituellement l'*algorithme du simplexe*. Cet algorithme commence par déduire une solution réalisable puis itère petit à petit en utilisant la convexité du problème afin d'améliorer la solution à chaque étape jusqu'à ce que cela ne soit plus possible : la solution finale est alors optimale. Géométriquement, l'espace des solutions réalisables d'un programme linéaire est un décrit par un polyèdre dont l'un des sommets est la solution optimale. L'algorithme du simplexe parcourt les sommets de ce polyèdre jusqu'à atteindre le sommet associé à la solution optimale.

Algorithme du Branch-and-Bound

L'algorithme du Branch-and-Bound (B&B) permet de résoudre des problèmes de programmation à nombres entiers. Il parcourt l'espace des solutions faisables de manière méthodique en estimant de plus en plus finement la borne (*bound*) inférieure et supérieure de la solution optimale. Ces deux bornes permettent de rapidement élaguer des parties de l'arbre et donc de réduire l'espace de recherche.

Pour ce faire, il construit itérativement un arbre de recherche où chaque nœud est une relaxation linéaire différente du problème initial. A chaque itération, il faut choisir un nœud feuille de l'arbre (étape de sélection de nœud) puis créer deux nœuds fils à partir de ce dernier (étape de sélection de variable). Lors de l'éclosion d'un nœud parent en nœuds fils, on sélectionne une "variable de branchement" $x = x_1$ (solution obtenue lors de la résolution de la relaxation linéaire du père) et on crée deux nouveaux nœuds fils. Ces nœuds fils représentent deux nouvelles versions du programme entier, contenant le même programme à optimiser que le père avec l'ajout d'une contrainte sur la variable de branchement x qui va diviser l'espace de recherche en deux. Ces nouveaux nœuds ont des contraintes supplémentaires : $x \leq \lfloor x_1 \rfloor$ pour le premier fils et $x \geq \lceil x_1 \rceil$ pour le second.

L'idée principale du B&B est la suivante : si la borne inférieure d'un nœud est supérieure à la borne supérieure globale connue, alors on peut élaguer la branche associée à ce nœud dans l'arbre (la meilleure solution des descendants de ce nœud ne sera jamais meilleure que celle déjà connue). La borne supérieure globale s'obtient avec la meilleure solution entière obtenue jusqu'à présent en parcourant l'arbre. La borne inférieure associée à un nœud est simplement la solution relaxée du programme de ce nœud. Plus on parcourt l'arbre en profondeur et plus cette solution relaxée va se dégrader. On arrête de parcourir l'arbre lorsque plus aucune branche de l'arbre ne peut être élaguée et qu'aucun nœud fils ne peut être construit. Si des solutions faisables ont été trouvées pendant la recherche, c'est alors la meilleure solution

faisable qui est optimale (sinon c'est qu'il n'existe pas de solution faisable).

La performance générale du B&B dépend fortement de la manière dont on construit l'arbre. Il y a deux heuristiques importantes : la sélection du nœud à parcourir et la sélection de la variable de branchement. De mauvaises heuristiques ne vont pas permettre d'élaguer efficacement les branches de l'arbre et vont rendre la résolution du programme entier très longue.

1.1.2 Diving Branch-and-Price

Le B&B a reçu beaucoup d'attention dans le domaine académique et il existe de nombreuses optimisations et déclinaisons différentes en fonction du type de problème à résoudre. Une de ces déclinaisons est le Branch-and-Price (B&P). Le B&P combine le B&B avec la génération de colonnes. Il permet de résoudre des problèmes lorsque le nombre de variables est trop grand pour se permettre d'optimiser le modèle en chargeant toutes les variables en mémoire.

De plus, lorsqu'il est impossible de résoudre un problème à l'optimalité en un temps acceptable, il est possible de parcourir l'arbre du B&B en profondeur et d'obtenir une solution faisable approximative du programme entier. Le parcours en profondeur va fixer les variables du problème à des nombres entiers jusqu'à avoir une solution faisable.

Combiner la génération de colonnes et le parcours en profondeur de l'arbre est un moyen efficace de traiter les plus gros problèmes de programmation à nombres entiers. L'algorithme résultant de cette combinaison se nomme le *diving Branch-and-Price*.

Génération de colonnes

La génération de colonnes a pour but de produire des variables (des *colonnes* du programme) x utiles à un programme linéaire afin de respecter les contraintes tout en minimisant la fonction objective. On dit que ces variables x sont à coûts réduits négatifs si elles permettent de réduire la fonction de coût tout en respectant les contraintes. Ces variables sont obtenues en résolvant des sous-problèmes liés au programme linéaire. Un exemple de sous-problème serait de trouver les chemins les plus courts validant certaines contraintes. Ces chemins peuvent ensuite être associés à une variable de décision pour le programme linéaire (le problème maître). Lorsqu'on utilise la génération de colonnes pour résoudre un programme linéaire, on alterne entre la résolution du problème maître et la résolution des sous-problèmes pour générer des solutions faisables de meilleures qualités. La solution est optimale, pour le problème maître, lorsque la génération de colonnes ne peut plus produire de colonnes à coût réduit négatif.

Dans un B&B, il est possible d'utiliser la génération de colonnes pour la résolution de chaque

relaxation linéaire des nœuds de l'arbre. Cette méthode se nomme le *Branch-and-Price*.

Parcours en profondeur du B&B

Lors d'un parcours en profondeur de l'arbre, on considère que l'heuristique de sélection de nœud choisit toujours le dernier nœud créé dans l'arbre (le plus profond). Ce faisant, on ne peut explorer tout l'espace de recherche et donc cette méthode ne permet pas d'obtenir une solution optimale mais plutôt de rapidement trouver une solution faisable entière du programme.

Lors de la création d'un nouveau nœud fils, l'heuristique de sélection de variable fixe la valeur de la variable sélectionnée x à $\lfloor x \rfloor$ ou à $\lceil x \rceil$. En bouclant entre la résolution du dernier nœud fils et en fixant petit à petit les variables, l'algorithme fini par trouver une solution faisable entière.

L'heuristique de sélection de variable (aussi appelée heuristique de branchement) est très importante, elle a une forte influence sur la qualité de la solution. On détaille maintenant deux des heuristiques les plus utilisées pour le diving B&P : le branchement fractionnaire et le strong branching.

Branchement fractionnaire

Le branchement fractionnaire choisit la variable fractionnaire x étant la plus proche d'un nombre entier. Mathématiquement, on peut écrire que cette heuristique choisit x tel que :

$$\min_x \min(x - \lfloor x \rfloor, \lceil x \rceil - x) \tag{1.2}$$

Cette heuristique est rapide et cherche à brancher sur les variables qui bousculent le moins la solution relaxée courante. Ce n'est pas l'heuristique qui donne les meilleures solutions en pratique car elle est un peu trop cupide, mais elle a l'avantage d'être rapide et simple.

Strong branching

Le strong branching est une autre heuristique de branchement. Cette méthode demande d'abord de résoudre tous les nœuds fils existants (toutes les variables fractionnaires possibles), puis choisit la variable qui a mené à la meilleure solution fille. Elle regarde donc un coup plus loin dans l'arbre et prends sa décision basée sur cette observation. Elle est très coûteuse en temps de calcul, mais c'est l'heuristique qui produit les meilleures solutions finales.

En pratique, on ne résout pas tous les candidats fils possibles, mais uniquement un sous-ensemble après avoir filtré les nœuds fils possibles. Malgré tout, cette heuristique est tellement lente que, lorsque le temps de résolution est un facteur important, on ne peut pas se permettre de l'utiliser.

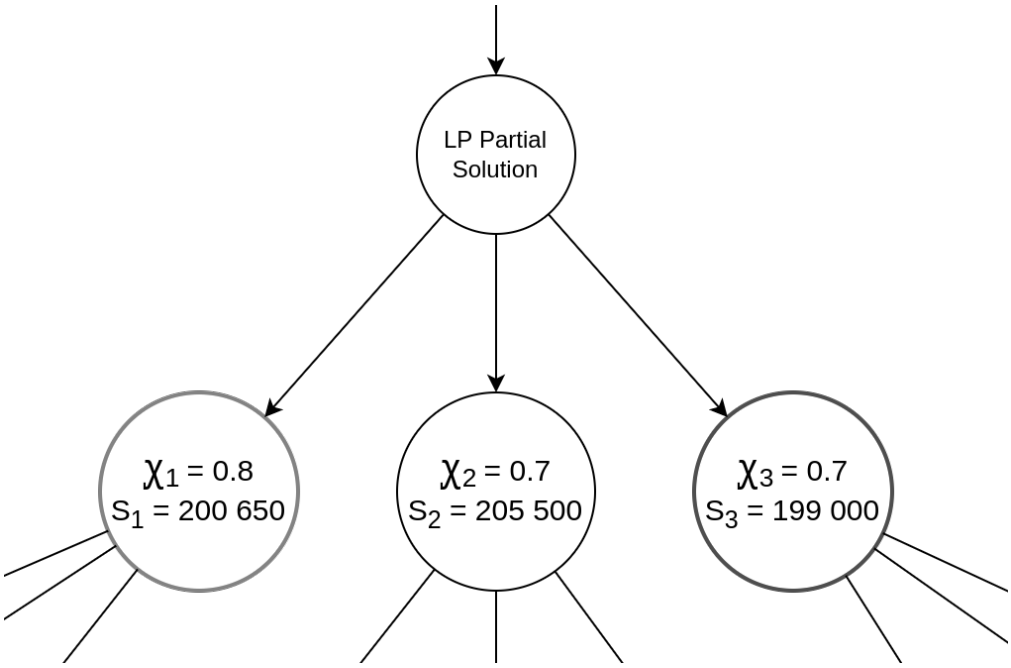

Figure 1.1 – Exemple d'une sélection de variable avec 3 candidats

La Figure 1.1 illustre la différence de branchement entre le branchement fractionnaire et le strong branching. Le branchement fractionnaire choisit le candidat 1 car il a la valeur fractionnaire χ_i la plus élevée. Le strong branching choisit le candidat 3 car c'est celui qui a la solution relaxée s_i la plus basse.

1.1.3 Apprentissage automatique

L'apprentissage automatique permet de valoriser des données en y extrayant de l'information pour produire un algorithme (un modèle) capable de prendre des décisions. Cette compilation de l'information de manière automatique permet de construire rapidement des règles de décisions complexes et performantes. Lors de la création d'un nouveau modèle, on dit qu'on l'entraîne sur les données afin qu'il apprenne ses règles de décisions. On nomme classiquement $x \in \mathbb{R}^{d_1}$ l'entrée d'un modèle f et $f(x) = \hat{y} \in \mathbb{R}^{d_2}$ la sortie de ce modèle. On appelle $\hat{y}$ la "prédiction" du modèle.

Apprentissage supervisé

L'apprentissage supervisé un moyen très utilisé pour entraîner des modèles d'apprentissage automatique. On représente le jeu de données comme un ensemble de couples (x_i, y_i) où x_i contient des caractéristiques sur lesquelles se base le modèle pour sa prédiction et y_i est la valeur qui doit être prédite. On note $(x_i, y_i)_1^N$ l'ensemble des N couples formant le jeu de données.

Le but de l'apprentissage supervisé est alors d'utiliser le jeu de données pour entraîner un modèle à prédire le mieux possible la relation entre les caractéristiques x_i et les étiquettes y_i. Mathématiquement, on souhaite avoir f tel quel $f(x_i) = \hat{y}_i \approx y_i \ \forall i$.

Plusieurs algorithmes d'apprentissages existent, en fonction du type de modèle choisi et des caractéristiques du jeu de données.

Apprentissage par renforcement

Un autre type d'apprentissage est l'apprentissage par renforcement. Cette façon d'entraîner un modèle ne nécessite pas de jeu de données au préalable mais demande en revanche d'avoir à disposition un environnement avec lequel le modèle peut interagir. Dans ce mode d'entraînement, le modèle apprend au fil de ses interactions avec l'environnement afin de maximiser une fonction de récompense.

Par rapport à l'apprentissage supervisé, le modèle génère lui-même son jeu de données à travers ses interactions avec l'environnement. Il peut se passer d'étiquettes y_i explicites, ce qui a un avantage lorsque celles-ci sont mal définies ou lorsqu'elles représentent mal l'objectif à optimiser.

Modèles paramétriques

Un moyen d'entraîner des modèles d'apprentissage automatique est de les paramétrer afin de plus facilement contrôler et analyser le comportement la fonction f. On note $\theta \in \mathbb{R}^d$ les paramètres du modèle et $f(x; \theta)$ sa prédiction.

On mesure la performance d'un modèle f à l'aide d'une fonction de perte L. Lors d'un apprentissage supervisé, on peut mathématiquement formuler l'entraînement d'un modèle comme étant l'optimisation des paramètres θ du modèle :

$$\theta = \arg\min_{\theta} L(\theta; (x_i, y_i)_1^N, f) \tag{1.3}$$

Un exemple de fonction de perte est la fonction d'entropie croisée :

$$L(\theta; (x_i, y_i)_1^N, f) = -\sum_{i=1}^{N} y_i \log f(x_i; \theta) \tag{1.4}$$

Où y_i et $f(x_i; \theta)$ sont des probabilités entre 0 et 1. Cette fonction de perte est minimale lorsque $f(x_i; \theta) = y_i$ c'est-à-dire lorsque le modèle effectue des prédictions parfaites.

Descente de gradient

Pour entraîner des modèles paramétriques, il est possible d'utiliser *la descente de gradient*. C'est un algorithme itératif qui va évaluer le gradient de la fonction de perte L par rapport aux paramètres θ afin de déterminer la direction que doivent prendre chaque élément de θ afin d'améliorer la performance du modèle.

Dans sa forme la plus simple, une étape de la descente de gradient se décrit ainsi :

$$\theta_{j+1} = \theta_j - \nu \nabla L(\theta_j; (x_i, y_i)_1^N, f) \tag{1.5}$$

Où $\nu \in \mathbb{R}^+$ est le pas de la descente de gradient. Cette itération continue jusqu'à convergence des paramètres θ_j.

Cette méthode fonctionne lorsque la fonction de perte L est dérivable. C'est une condition faible, ce qui lui permet d'être appliquée dans beaucoup de contextes d'optimisation. De plus, elle a l'avantage d'être de complexité $O(N)$ par itération (où N est la taille du jeu de données) ce qui la rend attractive pour traiter de grosses quantités de données. En revanche, elle ne permet pas de trouver le minimum analytique d'une fonction et peut rester bloquer dans des minima locaux d'une fonction non-convexe.

Réseaux de neurones

Les réseaux de neurones sont une famille de modèles paramétriques qui permettent une modélisation très flexible de la relation entre x et y. Ce sont des modèles non-linéaires entraînés à l'aide de la descente de gradient. Les calculs d'un réseau de neurones sont répartis à travers une série de couches qui vont l'une après l'autre améliorer la représentation vectorielle de l'entrée x. La dernière couche du modèle (la couche de sortie) va être capable de mieux effectuer sa prédiction à l'aide des transformations successives de x plutôt qu'avec la représentation vectorielle initiale x.

Une grande liberté de modélisation est possible au niveau des couches car tant que ces

dernières sont dérivables alors l'optimisation d'un tel modèle est possible. Ainsi, beaucoup d'architectures différentes existent que nous ne décrirons pas ici, mais nous pouvons au moins mentionner la couche cachée de base :

$$h_{i+1} = a_i(\theta_i h_i) \tag{1.6}$$

Où :

- $h_i \in \mathbb{R}^{d_i}$ est la représentation vectorielle de la couche i ;
- $\theta_i \in \mathbb{R}^{d_{i+1} \times d_i}$ est une matrice de paramètres ;
- a_i est une fonction dite "d'activation" non-linéaire.

1.1.4 Apprentissage par imitation

L'apprentissage par imitation est une catégorie d'apprentissage automatique supervisé. Ce type d'apprentissage a pour but d'entraîner un modèle à imiter un *oracle*. Souvent, l'oracle est coûteux à utiliser, voir impossible dans certaines situations, mais il sait parfaitement résoudre la tâche considérée. L'apprentissage par imitation cherche à l'approximer à l'aide d'un modèle beaucoup moins coûteux. Notons qu'en pratique l'oracle n'a pas besoin d'être parfait, l'apprentissage par imitation peut être utilisée pour imiter un expert coûteux.

Le fonctionnement de l'apprentissage par imitation est donc d'utiliser dans un premier temps l'oracle pour générer un jeu d'apprentissage étiqueté par ce dernier, puis d'entraîner le modèle sur ce jeu de données. La force d'une telle méthode est qu'elle permet un entraînement *hors-ligne*. C'est-à-dire qu'elle déplace la charge de calcul vers l'entraînement du modèle (phase de collecte du jeu de données et phase d'entraînement du modèle). Une fois entraîné, le modèle peut être exploité *en ligne* à faible coût. De plus, il est des fois difficile d'employer des méthodes d'apprentissage par renforcement car le modèle peut être dangereux s'il est déployé dans un environnement sans être déjà entraîné (ex : cas de la conduite autonome). Un risque important de l'apprentissage par imitation est le *distributional shift*. Le distributional shift peut arriver lorsque le modèle prend une série de décisions qui s'éloignent de la trajectoire théorique qu'aurait pris l'oracle. Le modèle s'éloigne alors petit à petit des situations rencontrées à l'entraînement et il est amené à prendre des décisions dans un environnement inconnu. Ce phénomène est un risque lorsque les décisions prises par le modèle ont une influence sur les décisions qu'il devra prendre par la suite, et cela peut entraîner une chute drastique de performance.

1.1.5 Problème de planification du personnel navigant

Les compagnies aériennes doivent établir l'emploi du temps de son personnel navigant (Crew Scheduling Problem CSP). C'est un problème difficile car il y a beaucoup de contraintes à prendre en compte. Le personnel navigant est constitué des pilotes et des copilotes (personnel navigant technique) et des agents de bord (personnel navigant commercial). La planification impose des contraintes temporaires, spatiales et administratives. Il faut s'assurer que chaque vol soit couvert, ainsi que les lois du travail spécifiques de la compagnie aérienne, et les réglementations des pays concernés, soient respectées.

Ce problème est résolu séparément pour le personnel navigant technique et le personnel commercial, car ils sont indépendants et n'ont pas les mêmes contraintes. Le CSP est défini pour couvrir un ensemble de vol sur une période donnée, typiquement sur 1 mois. La résolution se fait en deux étapes :

1. Rotations d'équipage (Crew Pairing Problem CPP) ;
2. Blocs mensuels d'équipage (Crew Assignment Problem CAP).

Les rotations d'équipage construisent des emplois du temps anonymes compatibles afin de minimiser les coûts (nuitées hors base, vols en tant que passager...). Ensuite, la construction des blocs mensuels permet d'associer ces emplois du temps au personnel de la compagnie aérienne en maximisant les préférences des employés.

Ce mémoire se concentre sur la résolution des CPPs.

Formulation d'un CPP

Une rotation est une série de vols séparés par des connexions, des vols en tant que passager et des périodes de repos. Pour que la rotation soit faisable, elle doit commencer et terminer à la même base aérienne et respecter toutes les contraintes temporelles et spatiales imposées par les vols. Un CPP peut être formulé par un programme à nombres entiers :

$$\min \sum_{p \in \Omega} c_p \chi_p \tag{1.7a}$$

$$\text{s.t.} \sum_{p \in \Omega} a_{fp} \chi_p = 1, \forall f \in \mathcal{F} \tag{1.7b}$$

$$\chi_p \in \{0, 1\}, \forall p \in \Omega \tag{1.7c}$$

Où $\mathcal{F}$ est l'ensemble des vols à couvrir et Ω est l'ensemble de toutes les rotations possibles. Le coût d'une rotation p est donné par c_p. L'objectif est de minimiser le coût total des rotations sélectionnées, modélisées par les variables binaires χ_p. Finalement, a_{fp} est une constante

binaire prenant la valeur 1 si la rotation p contient le vol f et 0 sinon.

Génération de colonnes pour le problème de rotation d'équipage

L'ensemble Ω contient en pratique des millions de rotations, il n'est donc pas envisageable de directement optimiser le programme tel quel. La génération de colonnes permet de ne générer qu'un infime sous-ensemble de ces rotations, celles qui ont un coût réduit négatif dans le problème maître. L'étape de génération de colonnes résout des sous-problèmes qui caractérisent Ω, des sous-problèmes de plus-court-chemins sous contraintes. Ces sous-problèmes sont résolus à l'aide de graphes dirigés acycliques.

Les CPPs sont trop gros pour être résolus optimalement. Une bonne solution faisable est obtenue à l'aide d'un diving Branch-and-Price.

1.2 Éléments de la problématique

Le CSP est un problème très important pour les compagnies aériennes. C'est en fait la seconde source de dépense annuelle après l'essence nécessaire aux avions. Une réduction des coûts liés aux CSPs d'1% amène à des économies de millions de dollars annuels. C'est donc un domaine de recherche très actif, ayant reçu beaucoup d'améliorations incrémentales depuis les années 1970, années où cette problématique a émergé.

Récemment, l'apprentissage automatique a été utilisé afin d'améliorer les algorithmes de recherche opérationnelle. Pendant des années, les chercheurs en recherche opérationnelle ont conceptualisé des heuristiques à force d'essai-erreurs. Cette étape laborieuse est en fait ce qu'est capable de faire l'apprentissage automatique. En effet, l'apprentissage automatique permet de construire des heuristiques en déterminant automatiquement des règles menant à de bonnes décisions. Ces règles peuvent être bien plus complexes que celles faites à la main par l'homme, en se basant sur plus de variables et en détectant des schémas statistiques invisibles à l'œil nu. Cependant, ce n'est pas sans défaut. Les heuristiques apprises de cette façon ne fonctionnent pas toujours aussi bien qu'espéré. On doit s'assurer que les relations statistiques apprises à l'entraînement sont bien celles que l'on retrouve lorsqu'on utilise le modèle. Il y a un risque de *sur-apprentissage* et de *distributional shifting*. La recherche actuelle a montré qu'il était possible d'améliorer l'état-de-l'art de la recherche opérationnelle à l'aide de l'apprentissage automatique.

1.3 Objectifs de recherche

Ce mémoire se concentre sur les CPPs. Ces problèmes sont résolus approximativement à l'aide d'un diving B&P. L'heuristique de sélection de variable a un impact crucial sur la performance de cet algorithme : sur la rapidité de l'exécution de l'algorithme ainsi que sur la qualité de la solution finale obtenue.

En pratique, on utilise actuellement le branchement fractionnaire comme heuristique de sélection de variable car il est rapide à utiliser et donne des solutions approximatives de bonne qualité. Cependant, il est connu que l'heuristique du strong branching donne des solutions finales moins coûteuses et qu'il serait souhaitable de l'utiliser à la place du branchement fractionnaire. Malheureusement, le strong branching est trop coûteux en temps de calcul, et il est impossible de l'utiliser en pratique si l'on souhaite résoudre les CPPs en un temps acceptable.

Pour cette raison, le but de ce mémoire est de construire une nouvelle heuristique à l'aide d'apprentissage par imitation afin de reproduire les décisions du strong branching à un coût

computationnel similaire à celui du branchement fractionnaire. La viabilité de la méthode est tout d'abord évaluée en utilisant les nouvelles heuristiques dans le solveur de référence GENCOL pour résoudre des problèmes de CPP. Ensuite, le distributional shift est étudié afin de vérifier si ce phénomène est à prendre en compte lors de l'utilisation de notre méthode.

1.4 Plan du mémoire

La suite de ce mémoire est composée des parties suivantes. Tout d'abord, le Chapitre 2 effectue une revue critique de la littérature autour du domaine des CPPs et de l'apprentissage automatique pour les heuristiques de branchement afin de cerner l'état de l'art du sujet. Une revue plus générale sur l'apprentissage automatique pour les problèmes de recherche opérationnelle se trouve dans la Section 4.2.1 de l'article. La démarche générale sur la façon dont le projet de maîtrise a été approché est décrite dans le Chapitre 3. Elle fait suite à la revue critique de littérature afin de justifier nos approches et différents choix de conceptions avant de commencer nos expériences de recherche. Le Chapitre 4 contient notre article où nous montrons la viabilité de l'apprentissage par imitation pour construire de nouvelles heuristiques de branchement. Le Chapitre 5 étudie l'impact du distributional shift sur les heuristiques apprise par imitation. Enfin, le Chapitre 7 conclue sur l'ensemble des travaux de ce mémoire, sur leurs limites et sur des améliorations potentielles futures.

CHAPITRE 2 REVUE CRITIQUE DE LITTÉRATURE

Cette revue de littérature se concentre sur le sujet de ce mémoire afin d'en justifier son approche. Une revue de littérature plus globale se trouve dans la Section 4.2.1 de notre article.

2.1 Crew Pairing Problem

La formulation des CPPs donnée par (1.7) se résout à l'aide d'un B&P [2, 3]. Cependant, nous travaillons sur de grosses instances de crew pairing, contenant des dizaines de milliers de vols sur une période d'un mois. Il n'est donc pas possible de déterminer une solution optimale à nos instances. C'est pour cela qu'en pratique on souhaite uniquement trouver une bonne solution approximative : une solution réalisable au coût le bas possible. Pour cela, on change légèrement l'algorithme du B&P en parcourant l'arbre en profondeur seulement [1]. Cela permet de rapidement trouver une solution réalisable, cependant la qualité de la solution dépends de l'heuristique de branchement utilisé pendant le parcours en profondeur [4]. Parce que le parcours s'effectue en profondeur, il n'est pas possible d'utiliser des heuristiques sophistiquées comme le *reliability branching* et le *pseudocost branching* [5] car elles demandent de brancher plusieurs fois sur les variables du problème. En pratique on utilise le branchement fractionnaire car il est rapide et permet d'obtenir de bonnes solutions [1]. C'est pourquoi c'est aussi le branchement utilisé par les compagnies aériennes. [6] a utilisé le *retrospective branching* qui permet d'annuler de mauvais branchements passés, cependant c'est ici aussi le branchement fractionnaire qui est utilisé pour fixer les variables.

Le strong branching est un autre branchement qui a été étudié [7], mais le temps de calcul d'une telle heuristique rends son utilisation impossible en pratique, bien qu'elle permette d'obtenir de meilleures solutions approximatives [4].

Le but de ce mémoire est de produire une nouvelle heuristique de branchement qui doit être aussi rapide que le branchement fractionnaire tout générant des solutions de meilleures qualités.

2.2 Learning to branch

Récemment, de nouvelles heuristiques de branchement ont été produites à l'aide de l'apprentissage automatique. [8] ont été les premiers à utiliser l'apprentissage par imitation, où l'heuristique apprend en imitant les décisions de branchement du strong branching. L'idée de

cette méthode est d'approximer l'heuristique du strong branching à l'aide d'un modèle mathématique plus léger. Il est alors possible de construire des heuristiques qui sont efficaces et rapides [9–11]. Les méthodes récentes utilisent des réseaux de neurones, qui permettent une modélisation plus complexe à l'aide de *Graph Neural Networks* [10] ou de *Gating models* [12].

Une autre façon d'apprendre l'heuristique de branchement est d'utiliser l'apprentissage par renforcement [13]. Ce faisant, on se débarrasse du besoin d'une heuristique de référence. L'heuristique apprise a alors la possibilité de découvrir par elle-même de bonnes et nouvelles décisions à prendre. Cependant, l'apprentissage par renforcement demande un entraînement très long, c'est une méthode prometteuse mais pas réalisable avec une puissance de calcul raisonnable.

L'apprentissage automatique offre la possibilité de générer de nouvelles heuristiques de branchement efficaces. Le strong branching étant l'heuristique produisant les meilleures solutions par rapport au branchement fractionnaire lors d'un diving B&P [4], cela justifie l'utilisation de l'apprentissage par imitation. Il est plus approprié aux CPPs car nos problèmes sont très grands et nous avons un bon branchement de référence à approximer.

Notons que la plupart des travaux cherchent à imiter le strong branching afin de résoudre les problèmes à l'optimalité [8, 10, 12]. Ainsi, ils imitent un strong branching qui réduit l'écart entre la borne inférieure et supérieure des solutions faisables le plus vite possible. Dans le cadre des CPPs, nous cherchons à rapidement obtenir une solution faisable entière au coût le plus faible possible, c'est pourquoi nous parcourons l'arbre du B&P en branchant sur le nœud qui augmente le moins possible la borne inférieure.

CHAPITRE 3 DÉMARCHE DE L'ENSEMBLE DU TRAVAIL DE RECHERCHE

Ce travail de recherche a commencé en équipe avec Emeric Courtade, étudiant en maîtrise recherche en Génie informatique à Polytechnique Montréal. Nous avons travaillé ensemble jusqu'à la division du projet en deux parties distinctes. Son sujet de recherche est d'étudier les caractéristiques des nœuds du B&P qui peuvent expliquer les décisions de branchement du strong branching. Ce faisant, il a remarqué que la variable fractionnaire χ_p est fortement corrélée à la décision finale du strong branching, ce qui explique les bonnes performances du branchement fractionnaire. Ses résultats font l'objet d'une discussion dans la Section 4.4 des résultats de l'article.

Après une revue de la littérature sur notre sujet de recherche, nous avons pris contact avec des chercheurs du domaine afin de mieux comprendre les enjeux de l'utilisation du machine learning dans la résolution des problèmes d'optimisation combinatoire. Nous avons eu l'occasion de discuter de notre sujet avec *Yassine Yaakoubi*, *Quentin Cappart* et *Maxime Gasse*. Yassine Yaakoubi a directement travaillé sur l'utilisation de l'apprentissage automatique pour améliorer la résolution des CPPs [14]. Quentin Cappart a étudié la capacité de généralisation des modèles entraînés à résoudre des problèmes de plus-court-chemin [15]. Maxime Gasse a étudié l'utilisation des *Graph Neural Networks* pour imiter le strong branching [10]. Suite à nos discussions fructueuses le long du projet, Yassine Yaakoubi a rejoint la liste des co-auteurs de l'article.

Les 6 premiers mois de la recherche ont été effectués à distance car les mesures sanitaires m'empêchaient de passer les frontières du Canada. Durant cette période, nous avons effectué la revue de littérature et commencer les entraînements de différents modèles. Une fois sur place, nous avons pris le temps de nous familiariser avec GENCOL, le solveur du GERAD permettant de résoudre les instances de CPP. Il a fallu le modifier afin de pouvoir rouler nos modèles à l'intérieur de ce dernier et qu'il puisse les utiliser comme nouvelles heuristiques de sélection de variable. Cette étape a pris plus de temps que prévu car il s'avère que le logiciel a beaucoup de dépendances. J'ai pu ensuite tester les différentes heuristiques entraînées par imitation afin de les évaluer sur les instances de références du CPP.

3.1 Justification de la méthodologie

Les heuristiques de branchement sont habituellement définies à la main par les chercheurs en recherche opérationnelle [16]. L'apprentissage automatique permet de construire de nouvelles heuristiques guidées par un processus d'optimisation. De plus, ces nouvelles heuristiques peuvent naturellement se spécialiser pour des familles de problèmes et donc mieux performer que des heuristiques généralistes.

Le branchement fractionnaire est celui utilisé en pratique par les compagnies aériennes car le strong branching demande trop de temps de calcul pour résoudre les instances de CPP. Nous avons donc pour objectif de produire de nouvelles heuristiques qui permettent de résoudre les CPPs aussi rapidement que le branchement fractionnaire tout en donnant de meilleures solutions. Pour cela, une fois que nous avons entraîné une nouvelle heuristique de branchement, nous l'utilisons dans le logiciel de résolution des instances de CPP et comparons les temps de calcul et la qualité de la solution avec ceux obtenus à l'aide du branchement fractionnaire.

Par rapport aux précédents travaux, notre approche se distingue car l'heuristique apprise est une heuristique de branchement pour un diving B&P alors que les autres travaux cherchent à apprendre une heuristique de branchement pour un B&B classique. Nos problèmes sont des problèmes de très grande taille contenant des millions de variables, ce qui a rarement été traité dans les précédents travaux (un exemple d'autres travaux similaires serait [11]). De plus, nos travaux cherchent spécifiquement à battre une heuristique dans un domaine particulier, celui du CPP, alors que la plupart des travaux cherchent à évaluer comment leur approche fonctionne sur plusieurs familles de problèmes différentes.

3.1.1 Apprentissage par imitation

Lors de la revue de littérature, nous avons vu qu'il était possible d'entraîner de nouvelles heuristiques de branchement par apprentissage par imitation ou par apprentissage par renforcement [17]. L'apprentissage par renforcement est intéressant car il ne nécessite pas d'heuristique de référence, et peut mener à des heuristiques ayant de nouvelles stratégies. Cependant, la résolution d'un CPP prend beaucoup de temps ce qui rend l'entraînement par renforcement difficilement envisageable. En effet, afin de pouvoir entraîner une telle heuristique, on doit l'utiliser pour résoudre de multiples fois des instances de CPP. L'état de l'art nous montre qu'il est difficile d'entraîner de telles heuristiques ainsi actuellement, alors que l'apprentissage par imitation donne de bons résultats en pratique [16]. Nous nous sommes alors concentrés sur l'utilisation de l'apprentissage par imitation en utilisant le strong branching comme heuristique de référence à imiter.

3.1.2 Normalisation des scores du strong branching

L'apprentissage par imitation étant un type d'apprentissage supervisé, il faut définir la façon dont nous allons étiqueter les données décisions du strong branching. On rappelle que le score d'un nœud de l'arbre du B&P est donné par la valeur de la solution de sa relaxation linéaire une fois que sa variable fractionnaire a été fixée à 1. Prédire directement le score du strong branching a été fait par [8] mais cela produit une heuristique qui ne peut s'adapter à des instances de tailles différentes que celles vues en entraînement. Il faut normaliser les scores afin de les ramener à la même échelle quelle que soit la taille du problème.

On peut souhaiter prédire le meilleur candidat et donc de faire une classification binaire (en associant 1 au meilleur candidat et 0 aux autres). Cependant en faisant ainsi on pénalise trop les bons candidats même s'ils ne sont pas les meilleurs. On met sur le même plan les mauvais candidats et les bons candidats, alors qu'il est possible que le bon candidat ait un score très proche du meilleur candidat. C'est pour cela qu'il vaut mieux penser les étiquettes comme étant un classement souple des candidats. Le classement souple permet de mieux pondérer le rang de chaque candidat en fonction de son score relatif avec les autres. Nous avons choisi de normaliser les scores de sorte à produire un classement souple des candidats, comme cela a été fait par [11]. On associe alors une probabilité à chaque candidat, de sorte que la somme des probabilités des candidats associés à un même nœud parent soit égale à 1 et que la probabilité reflète le score relatif d'un candidat par rapport aux autres.

Lors de la normalisation, nous nous sommes rendus compte que l'échelle variable de la valeur des solutions pose problème. Nous sommes obligés d'utiliser un paramètre de température afin de contrôler la force de normalisation utilisée par [11]. Cela nous a poussé à proposer une nouvelle normalisation, qui a pour but de mieux distinguer les bons des mauvais nœuds de branchement tout en ne demandant aucun paramètre supplémentaire à fixer.

3.1.3 Choix des modèles

Nous utilisons des réseaux de neurones car ils sont très permissifs et permettent un choix de modélisation varié, que ça soit par la façon dont on considère les caractéristiques des candidats en entrée ou même à la façon dont on peut décrire l'objectif à optimiser. Nous avons décidé de tester 3 architectures différentes de réseau de neurones.

La première architecture est un simple modèle sans couche cachée, ce qui revient à faire une régression logistique sur les probabilités des candidats. C'est un modèle linéaire avec peu de paramètres, il sert de point de référence aux autres modèles (en plus du branchement fractionnaire). Il généralise directement le branchement fractionnaire car il suffit de mettre

1 au paramètre associé à la valeur fractionnaire du candidat et 0 aux autres paramètres pour simuler le rang que produit le branchement fractionnaire. Le but de ce modèle est donc de valider notre méthodologie, car s'il est moins bon que le branchement fractionnaire c'est probablement qu'il y a une erreur dans notre méthodologie (plutôt que de croire que le modèle ne peut pas battre le branchement fractionnaire). Le deuxième type de modèle que nous avons utilisé est un réseau de neurones à couches cachées (Multilayer Perceptron MLP). Il permet une modélisation plus complexe que le modèle linéaire, et donc on peut espérer faire mieux. Comme pour le modèle linéaire, il considère les candidats séparément pour prédire leurs scores. Ce n'est pas le cas du troisième type de modèle que nous avons utilisé. Ce dernier utilise un calcul d'"attention" entre candidats afin de mieux pondérer les scores prédits en fonction des caractéristiques des autres candidats. On utilise pour cela un *transformer* [18]. On s'attend à ce qu'il soit le meilleur modèle car c'est celui qui a la modélisation la plus adaptée au problème. Cependant, ce type de modèle est plus sensible aux problèmes du *sur-apprentissage* et au *distributional shift*.

3.2 Cohérence de l'article par rapport aux objectifs de la recherche

L'article soumis utilise l'apprentissage automatique pour imiter les décisions du strong branching afin d'entraîner de nouvelles heuristiques de branchement pour les problèmes de CPP. Ces nouvelles heuristiques sont comparées aux performances (temps de résolution et qualité des solutions) du strong branching et du branchement fractionnaire. L'article s'inscrit dans l'objectif du projet de recherche qui est de proposer de nouvelles heuristiques de branchement à l'aide de l'apprentissage automatique.

CHAPITRE 4 ARTICLE 1 : LEARNING TO BRANCH FOR THE CREW PAIRING PROBLEM

Submitted “Operation Research Forum”, 28th of July 2022.

Authors

Pierre Pereira, pierre.pereira@polymtl.ca
Emeric Courtade, emeric.courtade@polymtl.ca
Daniel Aloise, daniel.aloise@polymtl.ca
Frederic Quesnel, frederic.quesnel@gerad.ca
François Soumis, francois.soumis@gerad.ca
Yassine Yaakoubi, yassine.yaakoubi@gerad.ca

Keywords crew pairing problem, heuristics, machine learning, branching strategy

Abstract

Crew pairing problems (CPP) are regularly solved by airlines to produce crew schedules. The goal of CPPs is to find a set of pairings (sequence of flights and rests forming one or more workdays) that cover all flights in a given period at a minimum cost. The CPP is a NP-hard combinatorial problem that is usually solved approximately using a branch-and-price heuristic. A common branching heuristics selects the closest-to-one pairing variable and fixes it without backtracking, which allows finding good solutions quickly. By contrast, variable selection strategies based on strong branching typically produce better solutions but are computationally prohibitive. This paper explores the possibility of learning efficient branching strategies for diving heuristics from strong branching decisions. To help the learning process, we propose a new normalisation of the branching scores which is able to better distinguish the good and bad branching candidates. Our results demonstrate that our learnt strategies make better branching decisions than the greedy strategy while executing in approximately the same computational time.

4.1 Introduction

Airline crew scheduling is an inherently difficult large-scale combinatorial optimization problem. Airlines have to plan ahead for the next month which flights their crew employees have to take, taking into account geographical and temporal constraints, as well as administrative constraints and crew preferences. Thus, the Crew Scheduling Problem (CSP) consists of finding a crew schedule that ensures all flights of a given period (typically a week or a month [1]) are assigned to one pilot and copilot. Crew expenses are the second highest fees of an airline company, after fuel [19], and therefore, any small improvement can lead to significant savings.

The CSP is typically divided into two sub-problems: the Crew Pairing Problem (CPP) and the Crew Assignment Problem (CAP). A CPP solution consists of a set of anonymous pairings that cover all flights over a time period at minimal cost. A pairing is a feasible sequence of flights separated by connections, deadheads and rest periods. For a pairing to be feasible, it needs to start and finish at the same crew base, meet the temporal and spatial constraints of the flights, and comply with collective agreements and airline regulations. Given a set of pairings, the CAP assigns them to all crew members, thus yielding a valid crew schedule. The objective of the CAP is to maximize crew satisfaction while respecting some additional constraints (e.g. a specific language must be spoken by at least one crew on some flight [20]).

In this work, we focus our attention to the solution of the monthly CPP, which can be formulated as a set-partitioning problem. Typical instances of the CPP contain up to tens of thousands of flights. Due to the size of the problem, the CPP is often solved in the literature by means of a Branch-and-Price (B&P) algorithm [1, 20]. B&P is a variant of the Branch-and-Bound algorithm where linear relaxations are solved using column generation. For the CPP, pairing candidates are generated by solving resource-constrained shortest-path sub-problems.

Because monthly CPP instances are too large to be optimally solved, solvers such as the state-of-the-art GENCOL [21] use a diving heuristic to find a near-optimal integer solution [2, 19]. Such heuristic relies on a good *variable selection strategy* to fix pairings. One of those strategies is based on the variables fractional values [4, 6], and quickly reaches a relatively good solution. By contrast, the *strong branching* strategy yields better final solutions [4], but is too computationally expensive to be used in practice.

Recently, Machine Learning (ML) has been more intensively used in the combinatorial optimization domain. ML exploits statistical redundancies of a family of problems to leverage good solution strategies to those problems. Using ML, we can consider producing a new branching strategy that might improve state-of-the-art combinatorial optimization solvers.

Specifically, one common method is to use imitation learning to approximate a computationally expensive oracle by a simpler model [8, 10]. The goal in this paper is to apply these ideas to our specific setting of diving B&P heuristics for CPPs. We do so by *proposing a new variable selection strategy for a diving heuristic* that relies on a ML model to select which pairing to fix at each node of the B&P tree. The ML model is trained offline in an imitation learning framework, using historical decisions from a strong branching diving heuristic as training data. Our contributions are:

— We demonstrate that imitation learning can be used to devise new data-driven diving B&P heuristics. To the best of our knowledge, this is the first time this is done in the literature.

— Our method relies on a new score normalization scheme that better distinguishes between good and bad branching candidates. We show that our ML-based variable selection strategy benefits from this new normalization.

— The diving branching heuristics based on those strategies produces better solutions than the traditional fractional branching algorithm, in a similar amount of time.

The rest of the paper is organized as follows. Section 4.2 presents a literature review of the CPP and of ML-based branching methods. Section 4.3 explains our methodology, Section 4.4 shows our results, and finally in Section 4.5, concluding remarks are given.

4.2 Literature review

4.2.1 Crew pairing problem

The CPP is typically formulated as a set-partitioning problem. Let $\mathcal{F}$ be the set of flights to be covered and Ω the set of all possibles pairings. The cost of a pairing p is given by c_p and a_{fp} is a binary coefficient that is equal to 1 if the pairing $p \in \Omega$ contains flight f, and 0 otherwise Finally, let χ_p be a variable equal to 1 if pairing p is selected, and 0 otherwise.

Thus, the CPP can be expressed as (4.1):

$$\min \sum_{p \in \Omega} c_p \chi_p \tag{4.1a}$$

$$\text{s.t.} \sum_{p \in \Omega} a_{fp} \chi_p = 1, \forall f \in \mathcal{F} \tag{4.1b}$$

$$\chi_p \in \{0, 1\}, \forall p \in \Omega \tag{4.1c}$$

For medium and large instances, Ω cannot be enumerated because it contains billions of pairings. Therefore, the CPP is commonly solved using a (B&P) algorithm [22] : linear

relaxations of the CPP are solved using column generation [2,3,7]. Column generation works by iteratively solving a *restricted master problem* (RMP) and one or several *subproblems*. The RMP solves the linear relaxation of (4.1), restricted to only some pairings of Ω. The goal of the subproblems is to find new negative-reduced-cost pairings to add to the RMP. The algorithm stops when no reduced cost candidates can be generated, at which point the current RMP solution is optimal. A branch-and-bound algorithm is applied to reach integrality. The column generation algorithm is used to solve the linear relaxation at each node.

In practice, exploring the whole branch-and-bound tree would be too computationally expensive, so a branching heuristic is used to quickly produce a good (but not necessarily optimal) solution [1]. In particular, *diving branching heuristics* explore the branching tree in a depth-first fashion and stop when an integer solution is reached, without performing backtracking. A common type of branching decision for the CPP is column fixing, in which one or several pairing variables have their values fixed to one. There exist several variable selection strategies. The *fractional branching* heuristic [4, 6] selects the variables with the highest fractional values. This heuristic is fast and yields good solutions in practice. To accelerate the diving process, it is possible to fix several candidates simultaneously as long as there are no conflicts between the pairings (no flight being covered more than once). This degrades the final solution but it is a powerful way of improving the computational time over the cost of the solution [23]

The *strong branching* heuristic employs another variable selection strategy which yields better final solutions than fractional branching [4], but is too computationally expensive to be used in practice. It evaluates several candidate variables for branching, and then solves one relaxed version of the underlying CPP problem for each candidate by fixing their associated values to one. The child node yielding the lowest-cost fractional solution is then selected, and the others discarded. An illustration of the described variable selection strategies is shown in Figure 4.1. In the figure, the strong branching strategy chooses the candidate 3 because it has the lowest relaxed solution value s_3 among the three candidate nodes, whereas the fractional branching strategy would have selected the candidate 1 because of its maximum fractional value χ_1 among the candidate variables.

Although fractional branching can be combined with other variable selection strategies [6,23], it remains the main branching strategy used to solve the CPP.

Other variable selection strategies exist, such as reliability branching and pseudocost branching [5], but they are not appropriate for a diving B&P heuristic where each variable is branched only once. A comparison between multiple diving branching strategies within B&P can be found in [4].

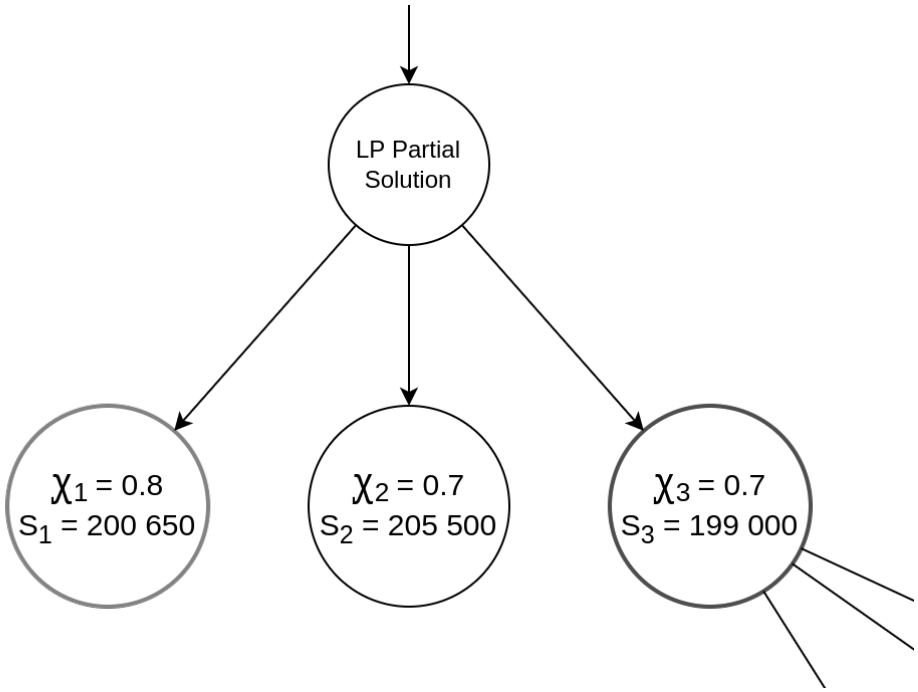

Figure 4.1 – Example of branching selection for three candidate nodes

We note that fixing a candidate to 0 is a decision of very low impact. Furthermore, such a decision is equivalent to forbidding a pairing, which is hard to do in column generation (although not impossible, see [24]). Therefore, we cannot use the product rule of [25] as it requires branching on both fixed candidate values, i.e., 0 and 1, to produce a score. Still, other types of branching rules can be found in the literature, such as *inter-task fixing* [1, 26, 27], which forces or forbid two flights to appear consecutively in a pairing.

Recent developments include the work of [6] where the authors proposed a *retrospective branching*, a branching heuristic which unfixes bad pairings during diving. Some works have been focused on the use of specific properties to the airline crew scheduling problem to improve the solver [20, 28].

Other works looked into applying machine learning to accelerate column-generation-based solvers for crew scheduling problems. [14] and [29] use machine learning to produce initial solutions and modify clusters of the CPP for the column-generation based solver. [30] use a *Graph Neural Network* to select which pairings to keep during the column generation in order to reduce the time spent when reoptimizing the restricted master problem. In the same line of work, [31] predict which pairings are useful by filtering the pairings during the column generation of a crew assignment problem. As far as we know, directly applying imitation learning to construct a new branching heuristic has never been done for the CPP.

4.2.2 Learning to branch

[8] were the first to try *imitating* a branching heuristic. They learnt to approximate strong branching with *Extremely Random Trees* on the standard benchmark MIPLIB [32]. [9] modeled the task as a ranking problem, where the relative score of each candidate is not important. Their approach is also original as the model is instance-specific, i.e., it is learnt while solving the given problem. [10] modeled MIPs using a bipartite graph and used a *Graph Neural Network* to learn from strong branching decisions. This way of encoding MIPs reduced the number of hand-crafted features by means of a raw representation of the MIP. [11] used this GNN idea to push it further and to learn heuristics for diving and branching. Since in practice, MIP solvers are not embedded in computers having GPUs for huge neural networks inference, [33] designed a cheaper framework that leverages the GNNs expressiveness while keeping it competitive when running inside a CPU-based solver. One recent innovation is the use of a parameterized search by [12] which combines two models. The first, called *NoTree*, is used to predict the value of a candidate based on its features. A second model, denoted *TreeGate*, is then used to encode the "tree state", thus adapting the inference of the first model. The second model allows better generalization over different sets of problems. [34] showed that it is often the case that when a candidate has been ranked number 2 by the strong branching heuristic, this candidate is the next best candidate for the next branching decision. The authors regularized their models by hardcoding this fact into their loss function, which resulted in faster solving times.

Other methods than imitation learning have been studied as well. [35] used machine learning to tune the neighbourhood size of the *local branching* heuristic. They showed that the neighbourhood optimal size has to be adapted dynamically for each MIP and for each iteration of the search. [36] have learnt an adaptive weighting of different heuristics to combine them dynamically. They also provide a theoretical study of the learning to branch task. In this line of work, [37] learn a scheduling of different well-known heuristics in order to rapidly find good feasible solutions.

Combining or improving known heuristics is a robust way of improving upon each heuristic separately but it does not allow for discovering new decision rules. [13] are the first to try to approach the learning to branch problem with reinforcement learning. This approach is attractive because it does not require any oracle (expert solver) and can potentially find new and better heuristics. Recently, [38] built upon the work of [13] to improve the reinforcement learning framework by introducing *Tree MDPs* that accelerate the convergence and improve the branching decision quality of the model. Nevertheless, training such heuristics needs tons of data, which is hardly practicable for large-scale problems. A survey about learning

to branch can be found in [17]. Finally, other works about the use of machine learning for combinatorial optimization problems can be found in the survey of [16].

4.3 Methodology

This section presents the methodology used in our experiments. We first present the variable selection strategies used within diving B&P heuristics for CPPs. We then present our general methodology. Finally, we explain how the dataset is collected, what normalization we use, the different models trained and how our training and evaluation pipeline is conducted.

4.3.1 Variable selection strategies

As explained in Section 4.2.1, CPPs are typically solved via B&P [2,3]. In this paper, we will compare our ML-based variable selection strategies with those used within the fractional and strong heuristics. The implemented fractional branching variant selects the single candidate χ_p having the highest (closest-to-one) fractional value and fixes it to 1. No computations are needed for this strategy as the fractional values are already computed from the parent node relaxation.

Our implementation of the strong branching variable selection strategy looks one step ahead by fixing and solving each candidate one by one, and then selects the candidate having the lowest relaxed optimal solution. Since we are interested in a feasible solution with the lowest possible cost, strong branching selects the candidate that increases the inferior bound the least. To reduce CPU times, it is common to evaluate a small subset of candidates. In our case, we only evaluate the top-5 candidates according to their fractional value. The drawback of strong branching is its computational overhead, though it is able to yield the best solutions.

4.3.2 Experimental framework

The goal of this paper is to imitate strong branching decisions with a faster approximation using the imitation learning framework [8]. This new learned variable selection strategy is expected to produce better solutions than fractional branching while having the same computational complexity.

The framework is very similar to that of [8]. We use imitation learning to produce new strategies based on an oracle obtained from strong branching decisions. To develop our heuristic, we proceed as follows:

1. (Dataset generation) To collect data, we solve several CPP instances using the strong

branching diving heuristic. At each strong branching evaluation step and for each candidate χ_i (among the top-5, according to their fractional value), we collect a set of features M_i and its strong branching score s_i. The dataset also contains the parent of each candidate so that sibling nodes can be grouped together and have their branching scores normalized together (see Section 4.3.2)

2. (Model training) A model is learned to predict a normalized version of the collected strong branching scores s_i based on the features M_i.
3. (Heuristic evaluation) During testing (evaluation), the trained model is used within the solver as a variable selection strategy. The resulting data-driven diving heuristic is therefore similar to fractional branching, except that the fixed variable is determined by the trained model rather than the fractional value of the variables.

An illustration of our methodological pipeline is shown in Figure 4.2.

Dataset collection

We used seven benchmark instances of the CPP literature from [1]. A summary of the instances is shown in Table 4.1. They consist of monthly CPPs, and can be divided into two groups according to their size: instances 1, 2 and 3 are considered small-scale, while the remaining ones 4, 5, 6 and 7 are considered large-scale. They are solved using the diving strong branching heuristic, so that a set of features is collected for each branching candidate.

Table 4.1 – The seven instances from [1] used to train and test our variable selection strategies

	Instance 1	Instance 2	Instance 3	Instance 4	Instance 5	Instance 6	Instance 7
Flights	1013	1500	1854	5613	5743	5886	7765
Airports	26	35	41	49	34	52	54

The features M_i used in our experiments are listed in Table 4.2. We followed the work of [8] by having three types of features:

— Static problem features (*nb_pairing_tasks, nb_cols_in_mp*).
— Dynamic problem features (*frac_val, var_cost, frac_conflicting_cols, min_cost_conflicting_cols, dual_cost*).
— Dynamic optimization features (*frac_pairing_tasks_fixed, depth*).

Nonetheless, some of our features are specific to CPPs (frac_pairing_tasks_fixed, nb_pairing_tasks). To limit the number of candidates, only the top-5 most promising candidates are retained

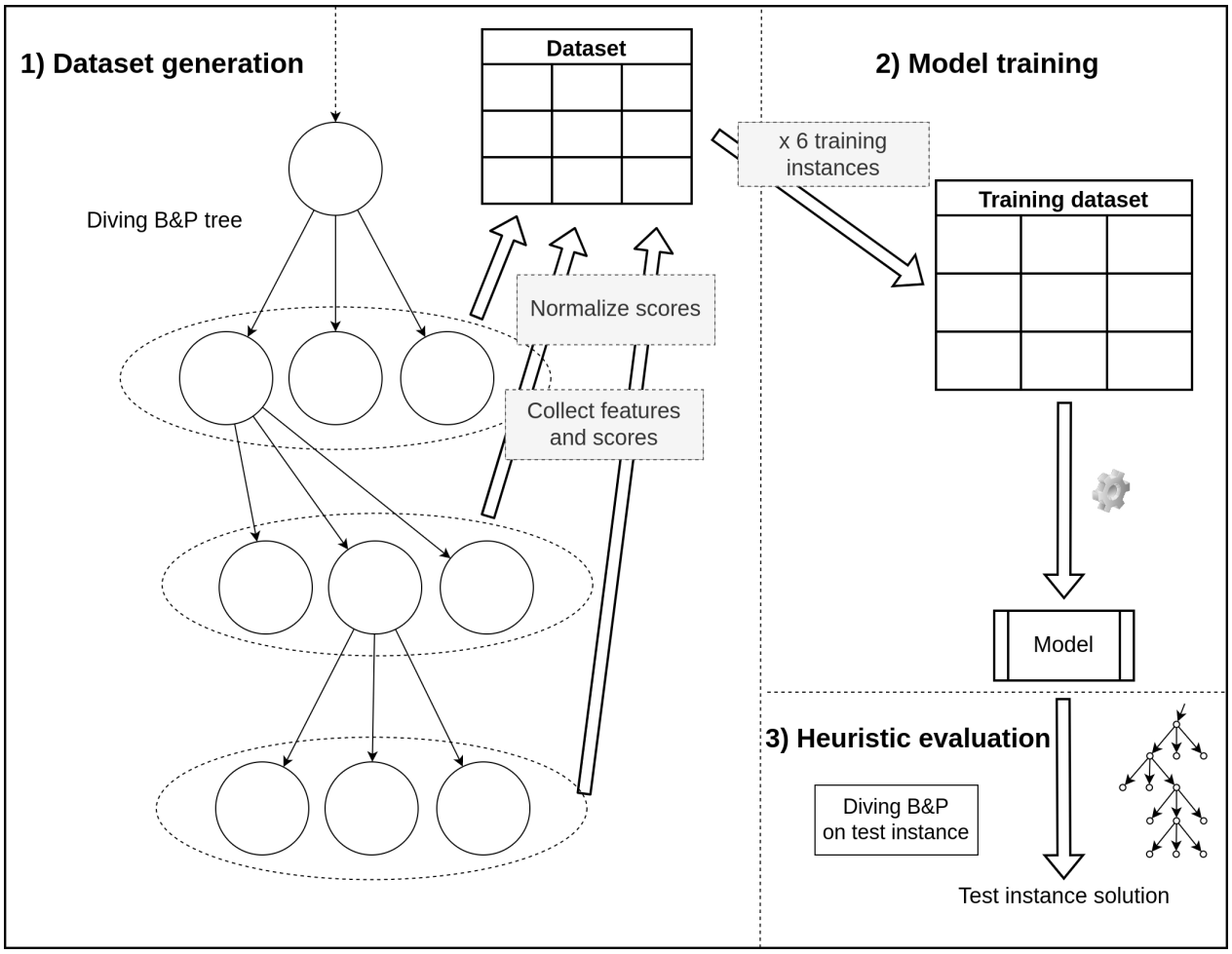

Figure 4.2 – Our methodology pipeline

according to their fractional value (i.e., the 5 candidates having the highest fractional χ_p value).

Models, training and evaluation

Three ML models are used in our study.[1] The first one is a linear model which is simple enough so that it should not present generalization issues. The second one is a multi-layer perceptron (MLP) with 3 hidden layers and 150 neurons in each hidden layer. These two models take into account only the features of a candidate to predict its score. Finally, we train a third model which is a very small transformer encoder [18]. It has 1 layer with an embedding size of 5, a feedforward hidden size of 10 and one attention head. We keep

1. The code used to train these models can be found at `https://github.com/Futurne/l2b_cpp`.

Table 4.2 – Our collected features

Name	Description
frac_val χ_p	Fractional value of the candidate in the relaxed parent solution.
var_cost c_p	Cost of the candidate in the objective function.
frac_conflicting_cols	Fraction of conflicting columns with this candidate.
min_cost_conflicting_cols	Minimum cost in the objective function of the candidate's conflicting columns.
dual_cost (min, avg, max)	Minimum, average and maximum dual cost of the candidate among the partitioning constraints.
frac_pairing_tasks_fixed	Fraction of pairing tasks fixed.
nb_pairing_tasks	Number of pairing tasks.
nb_cols_in_mp	Number of columns in the master program.
depth	Depth of the candidate in the B&P tree.

this transformer small in order to avoid overfitting. That model is able to consider all the candidates' features together to predict the score of a branching candidate.

A *leave-one-out* testing was used to assess the trained models, i.e., to test a model on one instance, we trained the model on all other instances. After the exclusion of the test instance, the remaining dataset is randomly split into training (80%) and validation (20%) data subsets to perform model selection. The trained model is then tested on the excluded instance within the solver.

We note that the goal of this paper is not to produce the best possible model for solving CPPs, but rather to investigate the possibility of learning a variable selection strategy for a diving branching heuristic by means of imitation learning. Therefore, a standard hyperparameter configuration was used without any particular fine-tuning: the training was performed for 100 epochs, using the ADAM optimizer [39] with a learning rate of 10^{-3} and the cross-entropy loss. We compute the validation loss on the validation dataset at each epoch. We kept as the final trained model the one having the lowest validation loss among the 100 epochs.

Normalizations of the strong branching scores

The score s_i of each candidate is the solution value of its relaxation once its corresponding fractional value has been fixed to 1. In order to train our models, we need to normalize this score so that the resulting normalized scores are comparable with each other. In doing so, we ensure that each normalized score falls within a fixed range of values, otherwise the range of s_i is affected by the underlying instance's size. Thus, we want the normalization to produce a meaningful ranking between the children nodes.

Recall that the model will be used to select among several candidates, the one that yields the lowest score s_i. However, we often observe that several candidates yield very similar

scores. Thus, the ranking should be *soft* so that the normalized score reflects how good the node is with respect to its siblings. In other words, our model should not overly penalize branching decisions that are good but not best. Therefore, we need a normalization that keeps equivalent candidates together while distinguishing the bad from the good candidates. Indeed, evaluating the quality of such normalization is challenging.

Our procedure starts by separating all candidates χ_i according to their parent node, and builds a vector s with their associated scores s_i. The model task consists in predicting the normalized rank of every child candidate, so that a decision can be taken regarding branching on the best predicted candidate. The normalized rank of a candidate is expressed as a probability score $p(\chi_i \mid M_i)$ such that the normalized ranks of all children of a parent node sum up to 1.

We test two normalization functions to transform the branching scores. The first one is the tempered softmax from [11]:

$$p(\chi_i \mid M_i) = \frac{\exp(-s_i/t)}{\sum_{k \in K} \exp(-s_k/t)}, \tag{4.2}$$

where t is a positive constant temperature hyperparameter that needs to be set, otherwise the softmax saturates to 1 for the best candidate and to 0 for the others; K is the set of all children of the same parent node (χ_i being one of these children).

The expression (4.2) is the standard norm for such a machine learning task [40]. The temperature t needs to be very well calibrated so that good candidates are grouped together. However, this normalization tends to bring good and bad candidates too close to one another, making it difficult for the ML predictor to provide a stable classification. In addition, this temperature is highly dependent on the instance at hand, making it difficult, if not impossible, to define a parameter value for each instance in advance.

To circumvent the above limitation, we propose an alternate norm (4.3) to better distinguish good and bad candidates.

$$y_i = \log s_i \tag{4.3a}$$

$$v_i = \frac{y_i - \bar{y}}{\sigma_y} \tag{4.3b}$$

$$p(\chi_i \mid M_i) = \frac{\exp(-v_i)}{\sum_{k \in K} \exp(-v_k)} \tag{4.3c}$$

where $\bar{y}$ is the mean value and σ_y is the standard deviation of the vector y.

The intuition behind this norm is to use the logarithm in (4.3a) to be less sensitive to the

scale of the scores, and to use a Gaussian normalization in (4.3c) before applying the softmax, to obtain a more discriminating difference between good and bad candidates. Both of those objectives are learnt with a negative cross-entropy loss [11, 40]. The idea of the two norms is that they are soft objectives for the classifier, meaning that the model is not penalized too much if it predicts a good candidate even if it is not the best among all. Note that, in contrast to the tempered softmax norm in (4.2), the proposed norm in (4.3) does not require any hyperparameter.

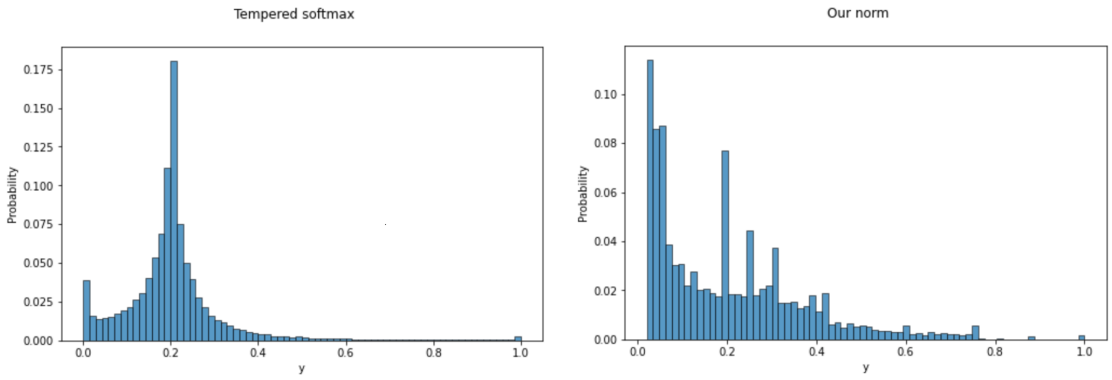

Figure 4.3 – A comparison between the classical tempered softmax (4.2) (left) and our norm (4.3) (right)

Figure 4.3 shows a comparison between the distributions of the two normalizations. The tempered softmax distribution is centered around the mean value which is 0.2 since we always have 5 candidates. But it needs a well calibrated temperature to make sure that good and bad candidates are well separated. Our norm produces a more discriminating rank without being too aggressive. An average candidate will have a lower probability with our norm than with a well calibrated tempered softmax norm. This leads to a flatter distribution in the histogram as shown in Figure 4.3 where candidates are spread across the probability distribution. When there are 5 average candidates, they are centered around 0.2. When there is one candidate that is clearly better than the others its probability score is pushed around 0.6 to 0.8, and the four remaining bad candidates then have their probabilities around 0 to 0.1.

4.4 Computational results

To assess the performance of the proposed methodological pipeline, each model (linear, MLP, transformer encoder) is trained to imitate the strong branching scores, using either the tempered softmax (4.2) or the proposed norm (4.3) as the normalization function. For each

model, we define a variable selection strategy that selects the candidate having the best predicted rank, according to the model. Those variable selection strategies are compared based on the quality of the proposed solutions and the execution time compared to those obtained using the strong branching heuristic.

The selection strategy based on the linear model is a direct improvement over the greedy strategy used by fractional branching, as the linear model has access to the fractional value of each candidate along with other features. The simplicity of this model prevents it from overfitting. In contrast, the transformer is more expressive than the linear model, and if the model generalizes well, the corresponding variable selection strategy is expected to make better decisions. Finally, since the inference of a model is way quicker than optimizing the relaxed version of each candidate, we expect the overall runtime of the machine-learning-based diving heuristics to be on the same order of magnitude as the fractional branching heuristic, and significantly faster than the strong branching one.

All B&P heuristics were implemented inside a state-of-the-art CPP solver based on the GENCOL optimization software [21]. All our experiments were performed on computers equipped with Intel Xeon E3-1226 v3 3.30GHz CPUs. Because of the inherent randomness of the solver, each diving heuristic was evaluated over five distinct runs. Consequently, the following tables present average values computed from these executions.

Table 4.3 shows the relative solution values of all diving heuristics compared with respect to strong branching (lower is better). Every solution is obtained by using a specific variable selection strategy h within the diving B&P heuristic. We compare the solution value S_h obtained with a strategy h to the solution value S_{SB} obtained with the strong branching heuristic. This is computed as a ratio between the two, i.e., $\frac{S_h - S_{SB}}{S_{SB}}$. We show in Table 4.3 the percentage of this score.

Table 4.3 – Solution values for each instance with respect to the strong branching strategy

Solution w.r.t. SB (%)	instance 1	instance 2	instance 3	instance 4	instance 5	instance 6	instance 7	mean
fractional branching	1.869	0.268	1.139	0.051	0.120	0.421	0.054	0.560
linear (our norm)	1.112	0.511	0.723	0.051	0.103	0.240	0.672	0.487
linear (softmax)	0.878	0.058	0.561	0.213	0.166	0.368	-0.043	0.314
MLP (our norm)	1.451	0.495	0.548	**-0.418**	0.213	0.473	0.146	0.415
MLP (softmax)	**0.681**	0.420	0.575	0.133	0.133	0.352	0.180	0.353
transformer (our norm)	0.916	0.420	**0.345**	-0.110	**0.066**	0.398	**-0.056**	**0.283**
transformer (softmax)	1.732	**-0.078**	0.892	0.565	0.265	**0.095**	0.121	0.513
mean (our models only)	1.128	0.304	0.607	0.072	0.158	0.321	0.170	

The data-driven variable selection strategies are on average better than that used by the fractional branching heuristic. As expected, a simple model such as the linear model already

yields lower-cost solutions compared with those obtained with fractional branching. The best model is the transformer trained with the new norm. In fact, the gap between the solution values obtained by the transformer diving heuristic and the strong branching is on average two times smaller than the gap between the values obtained by the fractional branching heuristic and the strong branching. We also observe from the table that the variable selection strategies based on the other machine learning models also perform relatively well.

Moreover, we observe in Table 4.3 that the simpler models (linear and MLP) do not benefit from the new normalization as they did not improve the results compared to their counterparts trained with the tempered softmax norm. This could be due to the fact that the Gaussian distribution is easier to learn. Furthermore, these models can only evaluate the candidates separately. In contrast, the transformer has access to all 5 candidates for prediction, which means that it is able to better discriminate between good and bad candidates. This could explain why the transformer is able to take advantage of the more informative rank of the candidates even if the distribution is not Gaussian.

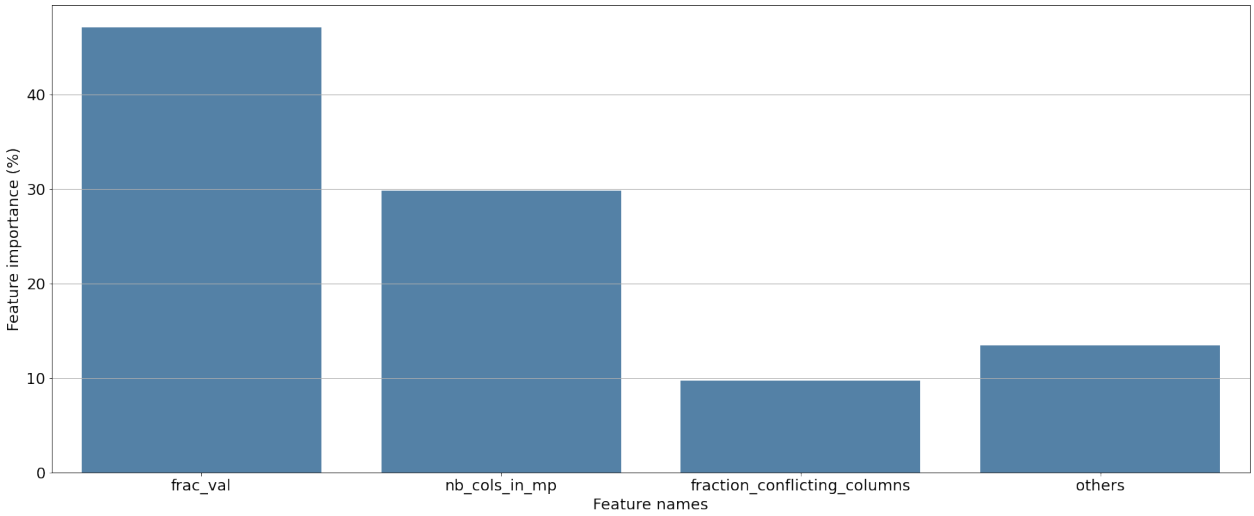

Figure 4.4 – Feature importance of the linear model trained with our norm

In Figure 4.4, we report the relative importance of the four most important features for the linear models trained with the new norm. Understanding which features contribute the most to the outcome of trained ML predictors can explain their behaviour and how they differ from other strategies. We measure the importance of each feature by taking the absolute value of its corresponding parameter, and then computing its relative percentage with respect to all the features. Features having an importance score smaller than 5% are grouped into the "others" category.

Notice that the *frac_val* value accounts for 47.1% of the prediction value. This explains why the performance of our heuristics are close to that of fractional branching. It also reveals that the fractional branching strategy is already a good proxy for the strong branching decision, which explains why it has been successfully used in practice.

Finally, Table 4.4 shows the runtimes of the different diving heuristics. It also presents the relative difference with respect to strong branching in parentheses. The relative runtime is computed by comparing the computational runtime T_h of a given diving heuristic h to the runtime T_{SB} obtained with the strong branching heuristic. This value is computed as $\frac{T_h - T_{SB}}{T_{SB}}$. The ML-based strategies are slightly slower than fractional branching. Nonetheless, they are significantly faster than strong branching. Unless the slight difference in execution time is unacceptable to an airline, the significant improvement in solution quality makes the proposed framework useful for airlines to reduce the cost of their CPP solutions.

4.5 Conclusions

In this work, we explored the use of machine learning for conceiving different data-driven variable selection strategies to be used within B&P diving heuristics. In particular, our strategies are learnt using imitation learning based on strong branching. This allows our heuristics to be as efficient as the fractional branching heuristic while making better branching decisions. Moreover, we also propose a new normalization for the strong branching scores that better distinguishes the good and the bad branching candidates. This norm shows better results than the standard normalization when used with a transformer, a model that is able to compare the branching candidates all together.

Finally, we note that our norm is harder to learn than the standard one because of its non-Gaussian distribution. A better norm would adapt the temperature hyperparameter of the standard norm based on the strong branching scores to make sure that good and bad candidates have well balanced probabilities after the softmax normalization. Besides, our branching decisions are limited by the candidates we consider at each stage of the tree. Since

Table 4.4 – Runtime for each instance with respect to the strong branching runtime

Time (s) (w.r.t. SB %)	instance 1	instance 2	instance 3	instance 4	instance 5	instance 6	instance 7	mean (%)
strong branching	178	227	889	40 481	44 639	51 478	109 365	–
fractional branching	**49 (-72.47)**	**55 (-75.75)**	218 (-75.50)	**9 037 (-77.68)**	**8 690 (-81.53)**	**11 450 (-77.76)**	**23 999 (-78.06)**	**-76.82**
linear (our norm)	52 (-70.79)	73 (-67.84)	246 (-72.35)	9 271 (-77.10)	9 880 (-77.87)	11 878 (-76.93)	26 560 (-75.71)	-74.08
linear (softmax)	57 (-67.98)	69 (-69.69)	236 (-73.48)	9 441 (-76.68)	9 833 (-77.97)	12 247 (-76.21)	24 777 (-77.34)	-74.19
MLP (our norm)	52 (-70.90)	71 (-68.55)	248 (-72.15)	9 743 (-75.93)	10 450 (-76.59)	12 406 (-75.90)	25 084 (-77.06)	-73.87
MLP (softmax)	57 (-68.75)	73 (-67.84)	246 (-72.31)	10 000 (-75.30)	10 013 (-77.57)	13 092 (-74.57)	26 191 (-76.05)	-73.06
transformer (our norm)	55 (-69.21)	62 (-72.69)	**217 (-75.57)**	10 713 (-73.93)	10 052 (-77.48)	12 027 (-76.64)	26 763 (-75.53)	-74.38
transformer (softmax)	54 (-69.78)	68 (-70.04)	244 (-72.60)	9 899 (-75.55)	10 739 (-76.94)	12 012 (-76.67)	25 142 (-77.01)	-73.94

the trained models can only select one of the five pre-selected top candidates (according to their fractional values), we strongly prevent them from branching on bad candidates besides reducing their expressiveness. As future research venues, one could explore the use of graph neural networks [10] with a raw representation of the candidates MIP models. Future research will also investigate the distributional shift that may occur when using such models, which can be sensitive to small feature variations, as well as ways to overcome that shift, for example by coupling graph neural networks and *DAgger* [11, 41].

Statements and Declarations

Acknowledgments We thank Quentin Cappart and Maxime Gasse for their valuable discussions and advice on the manner.

Funding Information This study was financially supported by Ad Opt.

Conflict of Interest On behalf of all authors, the corresponding author states that there is no conflict of interest.

Author Contributions Statement Every author helped during meetings with ideas and critical thinking about the research project. E. Courtade and P. Pereira did the experiments (code & computations). F. Quesnel and Benoît Rochefort helped with the GENCOL software for compilation and manipulation. Figures have been made by P. Pereira with the software *draw.io* and the library *matplotlib*. The manuscript has been written by P. Pereira, with some sections greatly enhanced by direct proposals of D. Aloise, F. Quesnel and Y. Yaakoubi. All authors reviewed the manuscript. Maxime Gasse and Quentin Cappart helped during fruitful conversations about the research project.

CHAPITRE 5 A PROPOS DU DISTRIBUTIONAL SHIFT

Durant les travaux de recherches, nous avons formulé l'hypothèse que le phénomène de distributional shift est un challenge supplémentaire à notre problème. En effet, par rapport aux approches classiques qui apprennent une heuristique pour un parcours d'arbre normal, nous parcourons l'arbre en profondeur. Cela implique que chaque décision de notre heuristique est définitive. De plus, la très grande taille de nos instances augmente les risques d'accumuler les mauvaises décisions. On peut alors supposer que si le modèle prend de mauvaises décisions de branchement, il risque de se retrouver dans un environnement qui ne ressemble pas à ce sur quoi il s'est entraîné : c'est le phénomène du distributional shift [42]. Dans un tel cas, le modèle a de grandes chances de ne pas savoir comment agir et à prendre des décisions hasardeuses. Plus le modèle est complexe et plus il a de chance de se tromper lorsqu'il est mis dans une situation légèrement différente à celles vues pendant l'entraînement. C'est pourquoi les modèles simples tels que les modèles linéaires sont moins sensibles au distributional shift.

Nous avons alors souhaité étudier ce phénomène pour peut-être le mettre en exergue, le but étant de montrer qu'il faut prendre en compte le distributional shift lors de la mise en place de la solution. Cependant nos expériences n'ont pas permis de montrer que le distributional shift est un problème important.

5.1 Méthodologie

Nous utilisons une méthodologie similaire à celle mise en place dans la Section 4.3 de l'article. On entraîne des heuristiques de branchement à l'aide de l'apprentissage par imitation en se basant sur les décisions du strong branching. Ces heuristiques sont ensuite utilisées pour obtenir des solutions approximatives des CPPs en les utilisant dans le logiciel GENCOL.

5.1.1 Génération du jeu de données

Nous utilisons les 7 instances classiquement étudiées pour les problèmes de CPP [1]. Des détails sur ces instances sont disponibles dans le Tableau 4.1.

Le jeu de données est collecté en utilisant le strong branching au sein du logiciel GENCOL et en sauvegardant chaque décision de branchement. Puisque le strong branching est très coûteux en calcul, on pré-sélectionne les candidats les plus prometteurs en prenant les 5 candidats ayant la valeur fractionnaire la plus haute. À chaque branchement, on sauvegarde les caractéristiques des candidats ainsi que la valeur de la solution relaxée obtenue grâce au

strong branching.

Nous collectons deux jeux de données différents. Un premier où le strong branching est utilisé à chaque branchement (exécution classique de l'heuristique), et un second en utilisant une technique de *distillation* [41] : une partie aléatoire des branchements se fait au hasard parmi les 5 candidats pré-sélectionnés. Le but de ce second jeu de données est de simuler des erreurs aléatoires du modèle qui imite le strong branching, afin de collecter des données sur des candidats que ne rencontrerait habituellement pas le strong branching. Ce second jeu de données permet donc de combattre le distributional shift [41] car un modèle entraîné sur ce dernier aura appris à traiter des cas plus variés que s'il avait été entraîné sur le premier jeu de données.

Dans les deux cas, on normalise les scores de chaque candidat (leur solution relaxée) en utilisant notre norme (4.3) (voir Section 4.3.2). On récolte les mêmes données que dans l'article (voir Section 4.3.2).

5.1.2 Modèles

Nous entraînons deux types de modèles : un modèle linéaire simple et un réseau de neurones à couches cachées (MLP). Le MLP possède 3 couches cachées de 150 neurones chacune. Le modèle linéaire est tellement simple qu'il ne devrait pas profiter de la distillation du second jeu d'entraînement car il ne peut pas apprendre de relation complexe entre les caractéristiques des candidats et la décision finale de branchement. En revanche, le MLP a une expressivité bien plus grande que le modèle linéaire. Il devrait être bien plus sensible au distributional shift.

5.1.3 Entraînement et évaluation

On ne souhaite pas entraîner l'heuristique sur l'instance sur laquelle elle va être testée ensuite. Pour cela, lorsqu'on évalue une heuristique sur une instance spécifique, on retire cette instance du jeu de données. Ce qui reste du jeu de données est ensuite divisé aléatoirement en jeu d'entraînement (80%) et de validation (20%).

Nous entraînons les heuristiques à l'aide de la descente de gradient stochastique, en faisant un nombre fixe de 100 itérations. À la fin d'un entraînement, l'heuristique finale est celle qui a eu une perte moyenne la plus faible sur le jeu de validation durant les 100 itérations. La fonction de perte est l'entropie croisée.

Nous entraînons séparément les heuristiques sur le premier jeu de données et sur le second, afin de comparer l'impact du type de jeu d'entraînement sur les performances des heuristiques.

5.2 Résultats expérimentaux

Nous souhaitons vérifier la présence du distributional shift lors de la résolution des instances de CPP. En premier nous vérifions s'il y a un lien entre la taille de l'instance et les performances des modèles entraînés sur le jeu de données sans distillation. En effet, les modèles plus complexes sont plus sensibles à un changement dans les distributions rencontrées. On devrait donc s'attendre à ce que le MLP produise des solutions moins bonnes sur les grandes instances par rapport aux autres heuristiques plus simples. En second, nous comparons les solutions obtenues par des modèles utilisant le jeu de données avec distillation et sans distillation. Le modèle linéaire ne devrait pas beaucoup bénéficier de la distillation due à sa simplicité. En revanche, le MLP entraîné sur le jeu avec distillation devrait s'améliorer sur les grandes instances par rapport au MLP entraîné sans distillation.

Les résolutions des instances ont été faites à l'aide du logiciel GENCOL [21] sur des ordinateurs équipés de CPUs Intel Xeon E3-1226 v3 3.30GHz. Le temps d'entraînement moyen d'un modèle est de 12 minutes (entraîné pour une instance de test).

Tableau 5.1 – Valeurs des solutions obtenues par les modèles entraînés sur le jeu de données sans distillation

Solutions	Instance 1	Instance 2	Instance 3	Instance 4	Instance 5	Instance 6	Instance 7
strong branching	197 458	270 507	347 772	760 969	1 121 056	1 052 447	1 520 184
branchement fractionnaire	201 148	271 231	351 731	761 356	1 122 398	1 056 873	1 521 010
modèle linéaire (no distill)	**199 312**	273 129	**350 321**	**761 119**	**1 121 952**	1 056 069	1 518 837
MLP (no distill)	200 534	**269 972**	350 769	761 571	1 124 815	**1 053 420**	**1 517 325**

Le Tableau 5.1 affiche les solutions obtenues à l'aide des modèles entraînés sur les données où la distillation n'a pas été utilisée. Pour chaque instance, la meilleure solution (la moins coûteuse) entre celles obtenues par le branchement fractionnaire, le modèle linéaire et le MLP est mise en gras. Nous pouvons voir que, par rapport au branchement fractionnaire, nos heuristiques sont compétitives en termes de qualité de solution. Cependant, si on regarde les performances des deux modèles uniquement, on s'attendrait à ce que le MLP soit meilleur sur les petites instances (1 - 3) et moins bon sur les grandes (4 - 7). Hors la performance relative des modèles ne semble pas être corrélée à la taille des instances comme le prédirait le distributional shift.

Le Tableau 5.2 compare les modèles entraînés sur le jeu de données avec distillation avec ceux entraînés sans distillation. Les modèles linéaires et les MLP sont comparés entre eux respectivement. Pour chaque instance, le meilleur modèle parmi celui qui est entraîné avec ou sans distillation est mis en gras. Alors qu'on s'attendrait à ce que les modèles entraînés avec

Tableau 5.2 – Comparaisons des solutions obtenues avec les deux types de jeux de données

Solutions	instance 1	instance 2	instance 3	instance 4	instance 5	instance 6	instance 7
linear (distill)	199 653	**271 888**	**350 288**	761 360	1 122 210	**1 054 977**	1 530 394
linear (no distill)	**199 312**	273 129	350 321	**761 119**	**1 121 952**	1 056 069	**1 518 837**
MLP (distill)	**200 323**	271 845	**349 679**	**757 785**	**1 123 448**	1 057 422	1 522 411
MLP (no distill)	200 534	**269 972**	350 769	761 571	1 124 815	**1 053 420**	**1 517 325**

distillation se démarquent sur les grandes instances (4 - 7), on ne remarque aucune tendance entre la taille des instances et l'entraînement avec et sans distillation. On peut donc supposer que ce phénomène n'est pas influent avec notre méthodologie.

5.3 Conclusion

Le distributional shift ne semble pas être un facteur important dans notre cas. La distillation a un impact sur ce qu'apprennent les heuristiques, mais cela relève plus d'un hyperparamètre à sélectionner plutôt que d'une méthode à appliquer systématiquement.

On s'attendait pourtant à ce que ce phénomène ait une forte influence sur la performance du MLP. On peut supposer que la pré-sélection des candidats de branchement empêche trop fortement aux heuristiques de prendre de très mauvaises décisions. Il est aussi possible que nos caractéristiques associées aux candidats soient suffisamment simples pour que les distributions ne changent pas drastiquement entre les candidats rencontrés via les branchements du strong branching et ceux rencontrés suite aux erreurs de nos modèles.

Il serait intéressant d'étudier ce phénomène lorsque l'on retire la pré-sélection des candidats. Il faudrait aussi augmenter la complexité des modèles entraînés, en utilisant par exemple un *transformer* [18] ou en ajoutant des couches cachées au réseau de neurones testé.

Finalement, s'il s'avère que le distributional shift n'est pas si problématique que ce que l'on pourrait croire, c'est une bonne nouvelle car cela justifie d'autant plus l'approche par imitation par rapport à un apprentissage par renforcement.

CHAPITRE 6 DISCUSSION GÉNÉRALE

Dans ce mémoire, nous avons étudié l'apprentissage par imitation pour entraîner de nouvelles heuristiques de branchement. Nous avons testé divers modèles et des normes différentes afin de voir ce qui fonctionne le mieux. De plus, nous avons étudié l'impact du distributional shift sur la méthodologie proposée.

Cependant, nous avons manqué de temps pour pousser plus loin nos analyses. Pour mesurer l'impact du distributional shift, nous aurions aimé tester plus de configurations possibles (le transformer par exemple), et laisser la possibilité aux heuristiques entraînées d'évaluer plus de candidats que les 5 candidats pré-sélectionnés. Pour aller plus loin, au vu des bonnes performances du branchement fractionnaire, nous avons eu l'idée d'apprendre un modèle qui aurait pour but de prédire l'erreur du branchement fractionnaire par rapport au strong branching. Ce modèle se baserait donc sur le classement initial produit par le branchement fractionnaire et modifierait ce classement en fonction des caractéristiques de chaque candidat de sorte à ce que le classement final soit similaire à celui du strong branching.

Enfin, on peut être surpris par les bonnes performances du branchement fractionnaire. En effet, cette heuristique produit déjà des solutions qui sont seulement 0.56% plus coûteuses des solutions trouvées à l'aide du strong branching. Cependant, il faut tout de même se replacer dans le contexte du problème : les CPPs sont les secondes sources de dépenses annuelles les plus élevées pour les compagnies aériennes. Sur nos plus grosses instances, 0.56% d'économie sur la solution revient à $8,400$\$ épargnés, c'est-à-dire plus de $100,000$\$ économisés à l'année !

CHAPITRE 7 CONCLUSION

Le but de ce projet de recherche est de produire de nouvelles heuristiques de branchement amenant à des solutions de meilleures qualités tout en étant aussi rapide que l'heuristique de branchement fractionnaire. Nous avons pu montrer qu'il était possible d'apprendre de nouvelles heuristiques à l'aide de l'apprentissage par imitation, et que ces heuristiques, utilisées dans un diving B&P, génèrent des solutions réalisables de meilleures qualités que celles générées par le branchement fractionnaire pour un temps de calcul similaire.

7.1 Synthèse des travaux

La revue de littérature a pointé du doigt la possibilité d'utiliser l'apprentissage par imitation [8] pour apprendre de nouvelles heuristiques de branchement pour améliorer les solutions des problèmes de CPP. Cependant il n'était pas évident de savoir à l'avance si cette méthode allait fonctionner pour une heuristique de branchement d'un diving B&P. En effet, dans le domaine du *learning to branch*, peu de travaux ont travaillé sur des problèmes aussi gros que les nôtres, et aucun n'a déjà travailler sur l'imitation d'une heuristique de branchement pour un diving B&P. Nous avons en plus le distributional shift [42] qui constitue une menace supplémentaire quant à la réussite de l'apprentissage.

Finalement, nous avons montré que l'apprentissage par imitation fonctionne dans notre cas. Bien que le temps total pour produire les solutions de CPP soit légèrement plus long qu'en utilisant le branchement fractionnaire, notre meilleure heuristique produit des solutions qui ont un écart avec la solution obtenue par strong branching deux fois plus faible que celles obtenues à l'aide du branchement fractionnaire. Nous avons expérimenté avec deux normes. La première est classiquement utilisée dans la littérature pour ce genre de problème [11,40], et la seconde est une norme que nous avons mis en place en espérant qu'elle facilite la distinction entre les bons et les mauvais candidats de branchement. Nos expériences ont montré que la norme classique semble plus simple à apprendre pour les modèles peu expressifs. En revanche, les modèles plus expressifs bénéficient de notre norme et sont finalement plus performants.

7.2 Limitations de la solution proposée

Une limitation principale de nos résultats est le fait que nous pré-sélectionnons les 5 candidats les plus prometteurs avant chaque branchement. Nos heuristiques sont donc contraintes dans leurs expressivités, et ne peuvent pas trop s'éloigner de ce que ferait le branchement

fractionnaire puisque c'est sur ce critère que sont pré-sélectionnés les candidats.

De plus, en pré-sélectionnant les candidats, nous réduisons grandement la possibilité de faire de très mauvais branchements. Cela a pu être bénéfique sur la qualité de la solution finale, mais ça a aussi pu empêcher une bonne analyse du distributional shift.

Finalement, il faut garder à l'esprit que l'apprentissage par imitation ne permet pas d'apprendre de nouvelles heuristiques qui feront mieux que ce que fait l'heuristique de référence. Pour nos travaux, le but est de battre le branchement fractionnaire donc ce n'est pas important, mais si on souhaite faire mieux que le strong branching, alors il faudra songer à utiliser une autre méthode (comme l'apprentissage par renforcement par exemple).

7.3 Améliorations futures

Nos travaux ne sont qu'une démonstration de l'utilisation de l'apprentissage automatique pour produire des heuristiques de branchement pour les CPPs. Il est possible d'utiliser de meilleurs modèles, comme les *Graph Neural Networks* [10] qui semblent plus adaptés lors d'un apprentissage d'heuristiques de branchement. Il serait aussi envisageable d'utiliser de nouvelles caractéristiques, qui pourrait décrire les candidats de manière plus riche. Nous avons travaillé sur les instances de bases des CPPs, mais il est possible d'augmenter la taille du jeu de données en faisant de petites variations dans la définition des problèmes pour produire de nouvelles instances. Tout cela permettrait, en se basant sur la même méthodologie que la nôtre, d'améliorer les heuristiques de branchement.

D'autres axes de recherches sont possibles. Pour accélérer la résolution des instances, il est commun de brancher sur plusieurs candidats *en même temps*. Lorsque l'on fait cela, la solution finale se dégrade rapidement. Il serait intéressant d'étudier comment brancher sur plusieurs candidats tout en maintenant une bonne solution finale. Aussi, l'apprentissage par renforcement est prometteur, et bien qu'il soit actuellement impraticable sur de tels gros problèmes, on peut espérer que des travaux futurs arriveront à l'utiliser pour des problèmes similaires aux CPPs afin d'apprendre de nouvelles heuristiques sans se baser sur une heuristique de référence.

RÉFÉRENCES

[1] A. Kasirzadeh, M. Saddoune et F. Soumis, "Airline crew scheduling : models, algorithms, and data sets," *EURO Journal on Transportation and Logistics*, vol. 6, n°. 2, p. 111–137, 2017.

[2] S. Lavoie, M. Minoux et E. Odier, "A new approach for crew pairing problems by column generation with an application to air transportation," *European Journal of Operational Research*, vol. 35, n°. 1, p. 45–58, 1988.

[3] C. Barnhart, E. L. Johnson, G. L. Nemhauser, M. W. P. Savelsbergh et P. H. Vance, "Branch-and-price : Column generation for solving huge integer programs," *Oper. Res.*, juin 1998.

[4] R. Sadykov, F. Vanderbeck, A. Pessoa, I. Tahiri et E. Uchoa, "Primal heuristics for branch and price : The assets of diving methods," *INFORMS J. Comput.*, avr. 2019.

[5] T. Achterberg, T. Koch et A. Martin, "Branching rules revisited," *Oper. Res. Lett.*, vol. 33, n°. 1, p. 42–54, janv. 2005.

[6] F. Quesnel, G. Desaulniers et F. Soumis, "A new heuristic branching scheme for the crew pairing problem with base constraints," *Computers & Operations Research*, vol. 80, p. 159–172, 2017.

[7] D. Klabjan, E. L. Johnson, G. L. Nemhauser, E. Gelman et S. Ramaswamy, "Solving large airline crew scheduling problems : Random pairing generation and strong branching," *Comput. Optim. Appl.*, vol. 20, n°. 1, p. 73–91, oct. 2001.

[8] A. M. Alvarez, Q. Louveaux et L. Wehenkel, "A machine learning-based approximation of strong branching," *INFORMS J. Comput.*, janv. 2017.

[9] E. Khalil, P. Le 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°. 1, Feb. 2016.

[10] M. Gasse, D. Chételat, N. Ferroni, L. Charlin et A. Lodi, "Exact combinatorial optimization with graph convolutional neural networks," dans *Advances in Neural Information Processing Systems 32*, 2019.

[11] V. Nair, S. Bartunov, F. Gimeno, I. von Glehn, P. Lichocki, I. Lobov, B. O'Donoghue, N. Sonnerat, C. Tjandraatmadja, P. Wang, R. Addanki, T. Hapuarachchi, T. Keck, J. Keeling, P. Kohli, I. Ktena, Y. Li, O. Vinyals et Y. Zwols, "Solving mixed integer programs using neural networks," 2021.

[12] G. Zarpellon, J. Jo, A. Lodi et Y. Bengio, "Parameterizing branch-and-bound search trees to learn branching policies," *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 35, nº. 5, p. 3931–3939, May 2021.

[13] M. Etheve, Z. Alès, C. Bissuel, O. Juan et S. Kedad-Sidhoum, "Reinforcement learning for variable selection in a branch and bound algorithm," dans *Integration of Constraint Programming, Artificial Intelligence, and Operations Research*, E. Hebrard et N. Musliu, édit. Cham : Springer International Publishing, 2020, p. 176–185.

[14] Y. Yaakoubi, F. Soumis et S. Lacoste-Julien, "Machine learning in airline crew pairing to construct initial clusters for dynamic constraint aggregation," *EURO Journal on Transportation and Logistics*, vol. 9, nº. 4, p. 100020, 2020.

[15] C. K. Joshi, Q. Cappart, L.-M. Rousseau et T. Laurent, "Learning the travelling salesperson problem requires rethinking generalization," *Constraints*, vol. 27, nº. 1, p. 70–98, avr. 2022.

[16] Y. Bengio, A. Lodi et A. Prouvost, "Machine learning for combinatorial optimization : A methodological tour d'horizon," *European Journal of Operational Research*, vol. 290, nº. 2, p. 405–421, 2021.

[17] A. Lodi et G. Zarpellon, "On learning and branching : a survey," *Top*, vol. 25, nº. 2, p. 207–236, juill. 2017.

[18] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. u. Kaiser et I. Polosukhin, "Attention is all you need," dans *Advances in Neural Information Processing Systems*, I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan et R. Garnett, édit., vol. 30. Curran Associates, Inc., 2017.

[19] M. Deveci et N. Çetin Demirel, "A survey of the literature on airline crew scheduling," *Engineering Applications of Artificial Intelligence*, vol. 74, p. 54–69, 2018.

[20] F. Quesnel, G. Desaulniers et F. Soumis, "A branch-and-price heuristic for the crew pairing problem with language constraints," *European Journal of Operational Research*, vol. 283, nº. 3, p. 1040–1054, 2020.

[21] J. Desrosiers, "Gencol : une équipe et un logiciel d'optimisation." *Stud. Inform. Univ.*, vol. 8, p. 61–96, 01 2010.

[22] G. Desaulniers, F. Lessard, M. Saddoune et F. Soumis, "Dynamic constraint aggregation for solving very large-scale airline crew pairing problems," *SN Oper. Res. Forum*, vol. 1, nº. 3, p. 19–23, août 2020.

[23] M. Gamache, F. Soumis, G. Marquis et J. Desrosiers, "A column generation approach for large-scale aircrew rostering problems," *Oper. Res.*, avr. 1999.

[24] D. Villeneuve et G. Desaulniers, "The shortest path problem with forbidden paths," *European Journal of Operational Research*, vol. 165, n°. 1, p. 97–107, 2005.

[25] T. Achterberg, "Scip : solving constraint integer programs," *Math. Prog. Comp.*, vol. 1, n°. 1, p. 1–41, juill. 2009.

[26] R. Anbil, J. Forrest et W. Pulleyblank, "Column generation and the airline crew pairing problem," *Documenta Mathematica*, vol. 35, 08 1998.

[27] G. Desaulniers, J. Desrosiers, Y. Dumas, S. Marc, B. Rioux, M. Solomon et F. Soumis, "Crew pairing at air france," *European Journal of Operational Research*, vol. 97, n°. 2, p. 245–259, 1997.

[28] M. Saddoune, G. Desaulniers et F. Soumis, "Aircrew pairings with possible repetitions of the same flight number," *Computers & Operations Research*, vol. 40, n°. 3, p. 805–814, mars 2013.

[29] Y. Yaakoubi, F. Soumis et S. Lacoste-Julien, "Structured convolutional kernel networks for airline crew scheduling," dans *Proceedings of the 38th International Conference on Machine Learning*, ser. Proceedings of Machine Learning Research, M. Meila et T. Zhang, édit., vol. 139. PMLR, 18–24 Jul 2021, p. 11 626–11 636.

[30] M. Morabit, G. Desaulniers et A. Lodi, "Machine-learning-based column selection for column generation," *Transportation Science*, juin 2021.

[31] F. Quesnel, A. Wu, G. Desaulniers et F. Soumis, "Deep-learning-based partial pricing in a branch-and-price algorithm for personalized crew rostering," *Computers & Operations Research*, vol. 138, p. 105554, 2022.

[32] T. Achterberg, T. Koch et A. Martin, "Miplib 2003," *Operations Research Letters*, vol. 34, n°. 4, p. 361–372, 2006.

[33] P. Gupta, M. Gasse, E. Khalil, P. Mudigonda, A. Lodi et Y. Bengio, "Hybrid models for learning to branch," dans *Advances in Neural Information Processing Systems*, H. Larochelle, M. Ranzato, R. Hadsell, M. Balcan et H. Lin, édit., vol. 33. Curran Associates, Inc., 2020, p. 18 087–18 097.

[34] P. Gupta, E. B. Khalil, D. Chetélat, M. Gasse, Y. Bengio, A. Lodi et M. P. Kumar, "Lookback for learning to branch," *arXiv*, juin 2022.

[35] D. Liu, M. Fischetti et A. Lodi, "Learning to search in local branching," *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 36, n°. 4, p. 3796–3803, Jun. 2022.

[36] M.-F. Balcan, T. Dick, T. Sandholm et E. Vitercik, "Learning to branch," dans *Proceedings of the 35th International Conference on Machine Learning*, ser. Proceedings of Machine Learning Research, J. Dy et A. Krause, édit., vol. 80. PMLR, 10–15 Jul 2018, p. 344–353.

[37] A. Chmiela, E. Khalil, A. Gleixner, A. Lodi et S. Pokutta, "Learning to schedule heuristics in branch and bound," dans *Advances in Neural Information Processing Systems*, M. Ranzato, A. Beygelzimer, Y. Dauphin, P. Liang et J. W. Vaughan, édit., vol. 34. Curran Associates, Inc., 2021, p. 24 235–24 246.

[38] L. Scavuzzo, F. Y. Chen, D. Chételat, M. Gasse, A. Lodi, N. Yorke-Smith et K. Aardal, "Learning to branch with tree mdps," 2022.

[39] D. P. Kingma et J. Ba, "Adam : A method for stochastic optimization," 2014.

[40] J. Guo, Y. Fan, L. Pang, L. Yang, Q. Ai, H. Zamani, C. Wu, W. B. Croft et X. Cheng, "A deep look into neural ranking models for information retrieval," *Information Processing & Management*, vol. 57, nº. 6, p. 102067, 2020.

[41] S. Ross, G. Gordon et D. Bagnell, "A reduction of imitation learning and structured prediction to no-regret online learning," dans *Proceedings of the Fourteenth International Conference on Artificial Intelligence and Statistics*, ser. Proceedings of Machine Learning Research, G. Gordon, D. Dunson et M. Dudík, édit., vol. 15. Fort Lauderdale, FL, USA : PMLR, 11–13 Apr 2011, p. 627–635.

[42] S. Ross et D. Bagnell, "Efficient reductions for imitation learning," dans *Proceedings of the Thirteenth International Conference on Artificial Intelligence and Statistics*, ser. Proceedings of Machine Learning Research, Y. W. Teh et M. Titterington, édit., vol. 9. Chia Laguna Resort, Sardinia, Italy : PMLR, 13–15 May 2010, p. 661–668.