Guider la recherche arborescente pour la résolution de problèmes industriels : apprentissage par renforcement et méthodes de Monte Carlo

Valentin Antuori

▶ To cite this version:

Valentin Antuori. Guider la recherche arborescente pour la résolution de problèmes industriels : apprentissage par renforcement et méthodes de Monte Carlo. Informatique [cs]. INSA Toulouse, 2022. Français. NNT : 2022ISAT0017 . tel-03869385v1

HAL Id: tel-03869385

https://laas.hal.science/tel-03869385v1

Submitted on 28 Sep 2022 (v1), last revised 24 Nov 2022 (v2)

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

THÈSE

En vue de l'obtention du

DOCTORAT DE L'UNIVERSITÉ DE TOULOUSE

Délivré par :
l'Institut National des Sciences Appliquées de Toulouse (INSA de Toulouse)

Présentée et soutenue le *08/07/2022* **par :**
Valentin Antuori

Guider la recherche arborescente pour la résolution de problèmes industriels : apprentissage par renforcement et méthodes de Monte Carlo

---

JURY

Emmanuel Hébrard	Chargé de Recherche CNRS	Directeur de thèse
Marie-José Huguet	Professeure des Universités	Directrice de thèse
Anastasia Paparrizou	Chargée de Recherche CNRS	Examinatrice
Louis-Martin Rousseau	Professeur	Rapporteur
Marc Sevaux	Professeur des Universités	Rapporteur
Sylvie Thiébaux	Professeure	Examinatrice
Vincent T'kindt	Professeur des Universités	Président du jury

École doctorale :
Ecole Doctorale Mathématiques, Informatique et Télécommunications de Toulouse
Spécialité :
Informatique et Télécommunications
Unité de recherche :
Laboratoire d'Analyse et d'Architecture des Systèmes (LAAS-CNRS)
Directeurs de thèse :
Emmanuel Hébrard et *Marie-José Huguet*
Rapporteurs :
Louis-Martin Rousseau et *Marc Sevaux*

Ce n'est qu'en essayant continuellement que l'on finit par réussir,
en d'autre terme, plus ça rate, plus on a de chance que ça marche.

Aphorisme Shadok

Remerciements

J'aimerais remercier Mr Louis-Martin Rousseau et Mr Marc Sevaux d'avoir rapporté ce manuscrit de thèse ainsi que les membres du jury de soutenance, présidé par Mr Vincent T'kindt et composé de Mme Anastasia Paparrizou et Mme Sylvie Thiébaux pour m'avoir accordé de leur temps.

Également pour le temps qu'ils m'ont consacré pendant trois ans, ainsi que pour leurs qualités humaines et scientifiques remarquables, j'aimerais remercier mes directeurs de thèse Marie-José Huguet et Emmanuel Hébrard. Merci également à mes encadrants industriels Siham Essodaigui et Alain Nguyen qui sont à l'origine de cette thèse, la confiance et l'autonomie qu'ils m'ont accordés durant ces trois ans ont été très appréciables.

Merci à toute l'équipe ROC pour la bonne humeur, le soutien, les pauses café et autres joyeusetés, nombreux seraient les jaloux de l'environnement et des conditions de travail que l'on a en tant que doctorant dans cette équipe. Une pensée particulière à Tom entre autre pour les rallonges de pause, les parties d'Unlock et de Civilization. Merci au chapitre IAA de Renault pour leur accueil lors de mes rares passages aux Plessis Robinson, et en particulier à Amine, et Hervé pour les échanges que l'ont a pu avoir autour des problèmes traités.

Pour tous les curieux, les étudiants, et les chercheurs, je remercie Alexandra Elbakyan.

Enfin j'aimerais remercier ma famille, notamment mes parents qui m'ont soutenu et poussé dans une voie à laquelle je n'aurais pas pensé emprunter 9 ans auparavant. Merci également à mes amis et à leur progéniture pour leurs encouragements et leur bonne humeur, mon parcours aurait été différent si certains n'avaient pas éveillé ma curiosité scientifique. Et merci à ma compagne Delphine pour le soutien sans faille à mes entreprises.

Résumé

La résolution de nombreux problèmes de recherche opérationnelle et plus spécifiquement de problèmes d'optimisation combinatoire, s'appuie sur des algorithmes de recherche arborescente. Dans un contexte industriel, il est fréquent que les problèmes combinatoires traités soient de très grande taille et/ou qu'on ne dispose seulement que d'un faible budget temporel à consacrer à leurs résolutions. Dès lors l'exploration complète de l'arbre de recherche est impossible, et la qualité d'une méthode arborescente repose alors sur sa capacité à s'orienter rapidement vers les zones de l'espace de recherche les plus prometteuses.

Il est fréquent qu'un même problème doive être résolu de manière périodique, tout en intégrant de légères variations. Il apparaît alors que la conception d'une heuristique pour guider la recherche arborescente peut passer par l'apprentissage automatique. Il semble ainsi possible d'utiliser un modèle d'apprentissage, entraîné sur un ensemble de données, sur de nouvelles données qui n'auraient que très peu variées.

De plus, une telle approche permettrait de spécialiser les heuristiques en entraînant le même modèle sur plusieurs ensembles de données issus de contextes différents pour un même problème. Cet apprentissage qui se fait en amont de la résolution peut également être combiné à un mécanisme d'apprentissage lors de la résolution du problème. Un tel mécanisme permet à l'algorithme une adaptation au problème plus précise encore.

Dans ce manuscrit nous nous intéressons à l'apport de méthodes d'apprentissage par renforcement et de méthodes de Monte Carlo pour la résolution de problèmes d'optimisation combinatoire issus de besoins industriels. Plus particulièrement, nous proposons deux approches dont le but est de guider l'exploration d'un arbre de recherche. La première approche consiste à concevoir une heuristique basée sur une combinaison linéaire de critères pertinents pour le problème, critères pouvant provenir de connaissances métier. Les poids de cette combinaison linéaire sont réglés via un algorithme d'apprentissage par renforcement, et l'heuristique obtenue est intégrée dans un algorithme de recherche arborescente. La seconde approche est une recherche arborescente de Monte Carlo combinée avec une recherche en profondeur d'abord. Le but est alors de découvrir, par l'expérience, quelle partie de l'arbre explorer. Ces deux approches peuvent être combinées et sont suffisamment génériques pour être adaptées aux deux problèmes industriels que nous étudions dans ce manuscrit. Le premier problème concerne la planification du déplacement de chariots pour transporter des pièces dans un atelier d'assemblage tout en respectant les cadences de production. Le second est un problème de chargement de camions en logistique amont comportant des contraintes liées à l'ordre de passage chez les fournisseurs et aux réglementations sur l'équilibre de la charge aux essieux. Pour ces deux problèmes les approches proposées surpassent les méthodes utilisées dans l'entreprise.

Mots-clés : Optimisation combinatoire, Apprentissage par renforcement, Recherche arborescente de Monte Carlo, Programmation par contraintes

Abstract

The resolution of many operation research problems, and more precisely combinatorial optimization problems relies on tree search algorithms. In an industrial context, combinatorial problems are usually very large and/or must be solved in a small amount of time. Consequently, the complete exploration of the search tree is often impossible, and tree-based methods should rely on their capacity to quickly find the most promising areas of the search space.

It is frequent that the same problem must be solved periodically, while incorporating slight variations. Thus, machine learning seems well suited to design heuristics to guide tree search in a context where we can expect historical data sets to be highly representative of the current problem.

Moreover, such an approach would allow to specialize heuristics by training the same model on several datasets from different contexts for the same problem. This learning phase, which is offline, can also be combined with an online learning mechanism. Such a mechanism allows the algorithm to adapt itself to the problem even more precisely.

In this thesis, we are interested in the use of reinforcement learning methods, and Monte Carlo methods for solving industrial combinatorial problems. More precisely, we propose two approaches to guide the exploration of a tree search. The first approach consists in designing a heuristic based on a linear combination of relevant criteria for the problem. The weights of this linear combination are learnt with a reinforcement learning algorithm, and the resulting heuristic is integrated into a tree search algorithm. The second approach is a Monte Carlo tree search combined with a depth-first search, with the objective of discovering which part of the tree to explore. These two approaches are studied in two industrial application cases. The first one is the problem of routing items between machines in an assembly line while respecting the production rates. The second one is a packing problem in inbound logistics, in which we have to fill trucks while respecting constraints related to the order of the suppliers visit and on axle load balance. For these two problems, the proposed approaches outperform the methods used in the company.

Keywords : Combinatorial Optimization, Reinforcement Learning, Monte Carlo Tree Search, Constraint Programming

Table des matières

Introduction **1**

1 Contexte et état de l'art **5**

1.1 Programmation par Contraintes 5
1.1.1 Formalisme de la programmation par contraintes 6
1.1.2 Méthodes de recherche arborescente 9
1.1.3 Autres approches 14
1.2 Apprentissage par renforcement 16
1.2.1 Principes de l'apprentissage par renforcement 17
1.2.2 Algorithmes d'apprentissage par renforcement 20
1.2.3 Fonction d'approximation 22
1.2.4 Algorithme REINFORCE 24
1.2.5 Apprentissage par renforcement pour l'optimisation combinatoire 27
1.3 Recherche arborescente de Monte Carlo 27
1.3.1 Principes de la méthode MCTS 28
1.3.2 Description de l'algorithme 28
1.3.3 Algorithme de MCTS pour l'optimisation combinatoire . . . 32
1.4 Apprentissage pour l'optimisation combinatoire 34
1.4.1 Apprendre à résoudre 35
1.4.2 Apprendre à chercher 36
1.4.3 Apprendre à configurer 39

2 Proposition d'un cadre générique pour la résolution de problèmes combinatoires **41**

2.1 Apprentissage par renforcement d'heuristiques 41
2.1.1 Processus de décision markovien pour la résolution de problème combinatoire 42
2.1.2 Politique de décision pour caractériser une heuristique de choix de valeurs 45
2.1.3 Algorithme d'apprentissage d'une heuristique de valeurs . . . 48
2.1.4 Intégration de l'heuristique dans une recherche arborescente . 50
2.2 Recherche arborescente de Monte Carlo 51
2.2.1 Utilisation des bornes 51
2.2.2 Recherche en profondeur 52
2.2.3 Compromis dynamique 53
2.3 Synthèse 54

3 Problème de tournées de chariots dans un atelier d'assemblage 57
3.1 Description du problème . . . 58
3.2 Travaux connexes . . . 62
3.2.1 Liens avec des problèmes de la littérature . . . 62
3.2.2 Modèle existant basé sur `LocalSolver` . . . 64
3.3 Modèles de programmation par contraintes . . . 65
3.3.1 Modèle d'ordonnancement . . . 66
3.3.2 Modèle de voyageur de commerce . . . 67
3.3.3 Stratégie de branchement . . . 69
3.4 Évaluation expérimentale des modèles de programmation par contraintes 69
3.4.1 Jeux de données . . . 69
3.4.2 Configuration . . . 71
3.4.3 Résultats expérimentaux . . . 71
3.5 Apprentissage par renforcement d'heuristiques de choix de valeurs . 74
3.5.1 Processus de décision markovien . . . 74
3.5.2 Caractérisation de l'heuristique de branchement . . . 75
3.5.3 Intégration de l'heuristique apprise dans des méthodes arborescentes . . . 76
3.5.4 Intégration de l'heuristique apprise dans une méthode de recherche locale . . . 77
3.5.5 Résultats expérimentaux . . . 81
3.6 Recherche arborescente de Monte Carlo . . . 85
3.6.1 Algorithme MCTS pour le problème de tournées de chariots . 85
3.6.2 Résultats expérimentaux . . . 87
3.7 Synthèse . . . 90

4 Problème de chargement de camions 93
4.1 Description du problème . . . 94
4.2 Travaux connexes . . . 98
4.2.1 Liens avec des problèmes de la littérature . . . 98
4.2.2 Algorithme existant dans l'entreprise . . . 101
4.3 Proposition d'une méthode de recherche arborescente . . . 103
4.3.1 Approche itérative . . . 103
4.3.2 Stratégies de placement et d'exploration . . . 104
4.3.3 Caractérisation de l'incomplétude . . . 110
4.4 Apprentissage par renforcement de la stratégie de branchement . . . 116
4.4.1 Processus de décision markovien . . . 116
4.4.2 Caractérisation de l'heuristique de branchement . . . 118
4.4.3 Résultats expérimentaux . . . 123
4.5 Recherche arborescente de Monte Carlo . . . 130
4.5.1 Algorithme MCTS pour le problème de chargement de camions 130
4.5.2 Résultats expérimentaux . . . 132
4.6 Synthèse . . . 136

Conclusion et perspectives **137**

Bibliographie **141**

Introduction

La résolution de nombreux problèmes d'optimisation combinatoire s'appuie sur des méthodes de recherche arborescente. Plusieurs mécanismes entrent en jeu pour l'efficacité du processus de résolution de ces méthodes. Parmi ceux-ci, on peut citer les différentes stratégies d'exploration de l'espace de recherche pour séparer cet espace et établir l'ordre de visite des différentes parties, les méthodes de filtrage pour réduire cet espace en supprimant des solutions non admissibles, ou les méthodes de relaxation pour évaluer des parties de l'espace de recherche et supprimer celles qui sont non pertinentes par rapport à un objectif à optimiser.

Néanmoins, dans un contexte industriel, ces problèmes sont souvent de très grande taille, et doivent être résolus en temps contraint. L'exploration complète d'un arbre de recherche s'avère impossible, et la qualité d'une méthode arborescente repose sur sa capacité à s'orienter rapidement vers des zones prometteuses de l'espace de recherche, en plus de réduire sa taille. Cependant, il est fréquent, dans ce contexte industriel, qu'un même problème d'optimisation combinatoire soit résolu périodiquement, tout en intégrant de légères variations sur les données du problème. Il semble alors que l'utilisation de l'apprentissage automatique pour guider la résolution de tels problèmes pourrait être pertinente, afin d'améliorer les performances des méthodes arborescentes.

L'apprentissage automatique a connu ces dernières années un regain de popularité et de nombreuses avancées dans le domaine de la vision par ordinateur, celui des jeux, et naturellement dans le domaine de l'optimisation combinatoire. Une des problématiques pour ce dernier domaine consiste à guider les méthodes de résolution, par exemple en utilisant des heuristiques d'exploration basées sur un modèle d'apprentissage.

Enfin, il arrive parfois qu'un même algorithme soit utilisé pour deux instances d'un même problème venant de contextes différents, et présentant alors des caractéristiques différentes. Une telle approche permettrait d'entraîner un modèle d'apprentissage pour chacun des contextes, afin de spécialiser l'algorithme. L'apprentissage d'un modèle peut se dérouler, en amont de la résolution sur des données réelles et les heuristiques d'exploration produites pourraient alors être intégrées dans une recherche arborescente. De plus, il peut être combiné à un mécanisme d'apprentissage durant le processus de résolution du problème pour s'adapter au problème considéré dans le but de déterminer les zones intéressantes à explorer plus fortement.

Cette thèse vise à étudier l'apport de méthodes d'apprentissage automatique combinées à des algorithmes d'optimisation combinatoire pour proposer de nouvelles approches hybrides afin de guider l'exploration d'un arbre de recherche. Cette thèse est issue du dispositif CIFRE avec l'entreprise Renault et les approches proposées sont évaluées pour la résolution de problèmes combinatoires réels rencontrés dans un contexte industriel.

Plus précisément, nous proposons un cadre général pour la création d'heuristiques de choix de valeurs, réglée par un algorithme d'apprentissage par renforcement. Nous intégrons ensuite ces heuristiques dans des algorithmes de recherche arborescente. Ces heuristiques prennent appui sur une combinaison linéaire de critères issus de connaissances sur le problème étudié. Nous proposons également des adaptations de la recherche arborescente de Monte Carlo pour la résolution de problèmes d'optimisation combinatoire, dont le but est de trouver rapidement les zones intéressantes de l'espace de recherche à travers l'évaluation de procédures gloutonnes lancées en plusieurs points de l'arbre de recherche. Ces deux méthodes sont intégrés, la recherche arborescente de Monte Carlo s'appuyant sur les heuristiques issues de l'apprentissage par renforcement, notamment dans une hybridation avec une recherche en profondeur d'abord.

Elles sont appliquées à deux problèmes industriels, le premier consiste à planifier le déplacement de pièces au moyen de chariots dans un atelier d'assemblage avec contraintes de capacité et de fenêtres de temps. Le second problème étudié est un problème de chargement de camions, se ramenant à un problème de placement en deux dimensions, avec des contraintes sur l'ordre d'insertion des objets et sur la charge aux essieux [1].

Organisation du manuscrit Le manuscrit se décompose en quatre chapitres.

Le chapitre 1 présente le contexte scientifique et technique de la thèse. Il commence par une présentation du formalisme de la programmation par contraintes et des principes d'une méthode de recherche arborescente, suivi par une présentation de l'apprentissage par renforcement puis de la recherche arborescente de Monte Carlo. Enfin, nous proposons un état de l'art sur l'apport de méthodes d'apprentissage automatique pour l'optimisation combinatoire.

Le chapitre 2 présente les contributions méthodologiques de la thèse. Nous y décrivons le cadre pour l'apprentissage d'heuristiques de choix de valeurs, basé sur une combinaison linéaire de critères qualifiant chacune des décisions possibles. Les poids sont alors obtenus par un algorithme d'apprentissage par renforcement. Dans ce chapitre, nous proposons également nos adaptations de la recherche arborescente de Monte Carlo. Ces adaptations sont l'utilisation de l'évolution des bornes de la fonction objectif comme récompense, l'hybridation avec une recherche en profondeur d'abord, et un mécanisme de compromis dynamique permettant de résoudre des problèmes de grande taille.

Les deux chapitres suivants détaillent la mise en oeuvre de cette méthodologie sur deux problèmes industriels. Le chapitre 3 s'intéresse à un problème de déplacement de pièces dans un atelier d'assemblage de l'entreprise. Après avoir évalué différentes formulations de ce problème, nous proposons d'appliquer le cadre présenté au chapitre 2. Une heuristique basée sur une combinaison linéaire de quatre critères est réglée par apprentissage par renforcement. Cette heuristique est ensuite

1. Les codes développés pendant la thèse sont disponible en ligne sur `https://gitlab.laas.fr/roc/valentin-antuori`.

intégrée dans deux approches arborescentes et une recherche locale à démarrages multiples avant d'être intégrée dans une recherche arborescente de Monte Carlo. Les évaluations expérimentales, sur des jeux de données industrielles ou simulées, mettent en évidence l'apport de nos différentes contributions.

Le chapitre 4 considère un problème de chargement de camions. Nous présentons d'abord une recherche arborescente prenant appui sur la méthode actuellement utilisée dans l'entreprise. Cette méthode ne permet pas d'explorer entièrement l'espace des solutions, et nous proposons de caractériser le type de solutions que nous ne pouvons pas atteindre. Nous définissons ensuite une heuristique basée sur quatre critères permettant de qualifier chaque décision, heuristique que nous intégrons dans notre recherche arborescente, puis dans une recherche arborescente de Monte Carlo. Les expérimentations menées permettent d'illustrer l'apport des différentes méthodes proposées.

Le manuscrit se termine par une conclusion, et des perspectives de ces travaux.

Publications Les contributions réalisées pour la résolution du problème de tournées de chariots ont donné lieu à plusieurs publications internationales. Ces publications portent sur les chapitres 2 et 3 du manuscrit :

- V. Antuori, E. Hebrard, MJ. Huguet, S. Essodaigui, A. Nguyen - *A Constraint Programming Approach for Planning Itemps Wransportation in a Workshop Context* - International Workshop on Project Management and Scheduling (PMS) April 15-17, 2020
- V. Antuori, E. Hebrard, MJ. Huguet, S. Essodaigui, A. Nguyen - *Leveraging Reinforcement Learning, Constraint Programming and Local Search : A Case Study in Car Manufacturing* - International Conference on Principles and Practice of Constraint Programming (CP), September 7-11, 2020
- V. Antuori, E. Hebrard, MJ. Huguet, S. Essodaigui, A. Nguyen - *Combining Monte Carlo Tree Search and Depth First Search Methods for a Car Manufacturing Workshop Scheduling Problem* - International Conference on Principles and Practice of Constraint Programming (CP), October 25-29, 2021

Des communications francophones ont également été présentées : ROADEF 2020 (version française de la présentation à PMS 2020) ; JFPC 2021 (version française de la présentation à CP 2020) ; RJCIA-AFIA 2021 (version française de la présentation à CP 2021).

En dehors de ce travail de thèse, une publication sur le filtrage par arc-cohérence de singleton pour le décodage de la machine Enigma a également été réalisée :

- V. Antuori, T. Portoleau, L. Rivière, E. Hébrard - *On How Turing and Singleton Arc Consistency Broke the Enigma Code* - International Conference on Principles and Practice of Constraint Programming (CP),October 25-29, 2021.

Chapitre 1

Contexte et état de l'art

Sommaire

1.1 Programmation par Contraintes 5
- 1.1.1 Formalisme de la programmation par contraintes 6
- 1.1.2 Méthodes de recherche arborescente 9
- 1.1.3 Autres approches 14

1.2 Apprentissage par renforcement 16
- 1.2.1 Principes de l'apprentissage par renforcement 17
- 1.2.2 Algorithmes d'apprentissage par renforcement 20
- 1.2.3 Fonction d'approximation 22
- 1.2.4 Algorithme REINFORCE 24
- 1.2.5 Apprentissage par renforcement pour l'optimisation combinatoire 27

1.3 Recherche arborescente de Monte Carlo 27
- 1.3.1 Principes de la méthode MCTS 28
- 1.3.2 Description de l'algorithme 28
- 1.3.3 Algorithme de MCTS pour l'optimisation combinatoire . . . 32

1.4 Apprentissage pour l'optimisation combinatoire 34
- 1.4.1 Apprendre à résoudre 35
- 1.4.2 Apprendre à chercher 36
- 1.4.3 Apprendre à configurer 39

Ce chapitre présente le contexte scientifique de cette thèse qui se base sur des méthodes de recherche arborescente et des méthodes d'apprentissage pour la résolution de problèmes d'optimisation combinatoire. Dans la première section, nous présentons le formalisme de la programmation par contraintes. La deuxième section est consacrée aux principes de l'apprentissage par renforcement. La troisième section présente la recherche arborescente de Monte Carlo. Enfin, dans la quatrième section, nous proposons un état de l'art de méthodes hybridant des techniques d'optimisation combinatoire avec des techniques d'apprentissage.

1.1 Programmation par Contraintes

D'une manière générale, un problème d'optimisation combinatoire est la recherche d'un élément dans un ensemble discret tel qu'une certaine mesure sur cet élément soit maximisée, ou minimisée.

Il existe plusieurs formalismes pour modéliser un problème d'optimisation combinatoire. Dans cette section, nous présentons le formalisme de la programmation par contraintes (PPC) qui est celui que nous avons considéré. Nous exposons ensuite le principe d'une recherche arborescente puis nous présentons d'autres méthodes utilisées pour la résolution de problèmes combinatoire.

1.1.1 Formalisme de la programmation par contraintes

Soit $\mathcal{X} = (x_1, ..., x_n)$ une séquence finie de variables. On note $\mathcal{D} = (\mathcal{D}(x_1), ..., \mathcal{D}(x_n)$ le domaine des variables de $\mathcal{X}$, avec $\mathcal{D}(x) \subset \mathbb{Z}$ l'ensemble fini de valeurs que peut prendre la variable x. On note $\min(\mathcal{D}(x))$ (resp. $\max(\mathcal{D}(x))$) la valeur minimale (resp. maximale) de $\mathcal{D}(x)$. On dit qu'une variable x est affectée à une valeur v si son domaine est réduit à cette valeur, c'est à dire si $\mathcal{D}(x) = \{v\}$. On définit l'inclusion de domaines pour une séquence de variables $\mathcal{X}$ par $\mathcal{D}_1 \subseteq \mathcal{D}_2 \Leftrightarrow \forall x \in \mathcal{X}$: $\mathcal{D}_1(x) \subseteq \mathcal{D}_2(x)$. Le domaine vide, noté $\perp$, correspond au domaine où $\mathcal{D}(x) = \emptyset$ pour toutes les variables x de $\mathcal{X}$.

On appelle tuple $\tau = (v_1, ..., v_n)$ une séquence de valeurs. Une contrainte c définie sur une séquence de variables $Y(c) = (x_{a_1}, ..., x_{a_k})$ est un sous ensemble fini de $\mathbb{Z}^k$ qui décrit l'ensemble des tuples autorisés par c sur les variables de $Y(c)$. $|Y(c)|$ est appelé l'arité de c et $Sc(c) = Y$ sa portée (*scope*).

Une contrainte peut être donnée en extension, en explicitant tous les tuples autorisés, ou en intention par une formule.

Exemple *Soient trois variables* x_1, x_2 *et* x_3 *telles que* $\mathcal{D}(x_1) = \mathcal{D}(x_2) = \mathcal{D}(x_3) = \{1, 2, 3\}$. *Pour définir une contrainte représentant le fait que toutes les variables ont des valeurs différentes on peut la définir en extension en donnant la liste des tuples acceptés :* $(1, 2, 3), (1, 3, 2), \ (2, 1, 3), (2, 3, 2), (3, 1, 2), (3, 2, 1)$ *ou bien la définir en intention :* $(x_1 \neq x_2) \wedge (x_1 \neq x_3) \wedge (x_2 \neq x_3)$.

Un type particulier de contraintes en intention est appelé *contrainte globale*. Ces contraintes peuvent être appliquées sur un nombre arbitraire de variables et capturent généralement un schéma récurrent à de nombreux problèmes. On définit ici deux célèbres contraintes globales, la contrainte AllDifferent [107] et la contrainte Element [75].

La contrainte AllDifferent impose que toutes les variables de sa portée prennent une valeur différente. Soient n variables $x_1, ..., x_n$, la contrainte AllDifferent sur ces n variables s'écrit :

$$\textsc{AllDifferent}(x_1, ..., x_n) : x_i \neq x_j, \ \forall i \in \{1, ..., n-1\}, \forall j \in \{i+1, ..., n\}$$

La contrainte Element permet d'associer une variable d'une séquence, à une seconde variable, au moyen d'une troisième variable qui représente l'indice dans la séquence. Soit Y une séquence de variables, et soient deux variables x et z. La contrainte Element s'écrit :

$$\textsc{Element}(x, Y, z) : Y[x] = z$$

Une instance $\mathcal{P}$ d'un **problème de satisfaction de contraintes** (*Constraint Satisfaction Problem*, CSP) est alors définie par un triplet $(\mathcal{X}, \mathcal{D}, \mathcal{C})$ avec :

— $\mathcal{X} = (x_1, ..., x_n)$ une séquence de variables

— $\mathcal{D} = (\mathcal{D}(x_1), ..., \mathcal{D}(x_n))$ le domaine des variables de $\mathcal{X}$

— $\mathcal{C}$ un ensemble de contraintes définies sur des sous-ensembles de $\mathcal{X}$.

Une instanciation $I = (v_1, ..., v_k)$ sur $Y = (x_1, ..., x_k) \subseteq \mathcal{X}$ est une affectation des valeurs $v_1, ..., v_k$ aux variables $x_1, ..., x_k$. On la note également $I = ((x_1, v_1), ..., (x_k, v_k))$. On représente par $I[x]$ la valeur de la variable x donnée par l'instanciation I. Pour une instanciation $I = (v_1, ..., v_n)$ sur $\mathcal{X}$ et une séquence de variables $Y = (x_{Y_1}, ..., x_{Y_k})$ de $\mathcal{X}$, on note $I\pi_Y = (v_{Y_1}, ..., v_{Y_k})$ la projection de l'instanciation I sur Y.

Pour un CSP $\mathcal{P} = (\mathcal{X}, \mathcal{D}, \mathcal{C})$. On dit qu'une instanciation I sur $Y \subseteq \mathcal{X}$ est

— *Valide* si pour toutes variables $x \in Y$, $I[x] \in \mathcal{D}(x)$.

— *Localement cohérente* si pour toutes contraintes $c \in \mathcal{C}$ avec $Sc(c) \subseteq Y$, on a $I\pi_{Sc(c)} \in c$.

— *Solution* de $\mathcal{P}$ si $Y = \mathcal{X}$ et si I est valide et localement cohérente.

— *Globalement cohérente* s'il existe une instanciation I' sur $\mathcal{X}$ qui est une solution de $\mathcal{P}$, et que $I'\pi_Y = I$. Autrement dit, I peut être étendue à une solution.

Résoudre une instance d'un CSP consiste à trouver une solution si elle existe, ou à prouver qu'il n'en existe aucune.

Une instance $\mathcal{P}$ d'un **problème d'optimisation sous contraintes** (*Constraint Optimization Problem*, COP) est une instance d'un CSP auquel on adjoint une fonction objectif f à maximiser ou à minimiser. Une solution de ce problème est une solution du CSP sous-jacent telle que valeur de la fonction objectif f à maximiser (resp. minimiser) soit supérieure ou égale (resp. inférieure ou égale) à celles de toutes les autres solutions du CSP.

Exemple *Pour illustrer une modélisation avec la programmation par contraintes, nous prenons l'exemple du problème de course d'orientation [66]. Ce problème se décrit comme suit. Soit $G = (V, A)$ un graphe non orienté complet avec $V = \{1, ..., n\}$ l'ensemble des noeuds et $A = \{(i, j) \mid i, j \leq n\}$ l'ensemble des arêtes. Un profit p_i est associé à chaque noeud $i \in V$, et une distance $d_{i,j}$ est associée à chaque arête $(i, j) \in A$. Le problème de la course d'orientation consiste à déterminer un chemin partant du noeud 1, arrivant au noeud n et ne dépassant pas une longueur totale D_{max}, afin de maximiser la somme des profits des noeuds empruntés. On admet que les profit des noeuds de départ et d'arrivée sont nuls $p_1 = p_n = 0$, et que ceux de tous les autres noeuds $i \in \{2, ...n - 1\}$ sont positifs $p_i > 0$.*

Avec le formalisme de la programmation par contraintes, on peut modéliser ce problème avec un unique ensemble de variables $\mathcal{X} = (x_1, ..., x_n)$ représentant le chemin parcouru. Une variable x_i représente le noeud visité en i-ème position dans

le chemin, elle vaut 0 *lorsque l'arrivée est déjà atteinte. On ajoute donc au graphe un sommet fictif, noté* 0*, de profit nul* $p_0 = 0$*, relié à tous les autres sommets et à lui même avec une distance nulle :* $d_{0,i} = 0 \ \forall i \in \{0, ..., n\}$*. Le domaine des variables est ainsi défini par* $\mathcal{D}(x_i) = \{0, .., n\}$ *pour tout* $i \in \{1..n\}$ *.*

On peut alors définir les contraintes de ce problème. On doit s'assurer que l'on visite en premier le sommet de départ, et une fois le sommet arrivée visité, toutes les variables suivantes doivent prendre la valeur 0. Ces deux contraintes peuvent s'exprimer de la façon suivante :

$$x_1 = 1 \tag{1.1}$$

$$x_i = n \vee x_i = 0 \implies x_{i+1} = 0 \qquad \forall i \in \{2, .., n-1\} \tag{1.2}$$

Puis, il faut spécifier que le noeud d'arrivée doit être visité au moins une fois. On peut utiliser la contrainte globale AtLeast *qui signifie qu'au moins une valeur parmi* $\{x_1, ...x_n\}$ *doit prendre la valeur* n*.*

$$\textsc{AtLeast}(1, \{x_1, ...x_n\}, n) \tag{1.3}$$

Ensuite on doit contraindre chaque sommet à n'être pas visité plus d'une fois. Cela peut s'exprimer par plusieurs contraintes globales AtMost *signifiant qu'au plus une valeur parmi* $\{x_1, ...x_n\}$ *doit prendre la valeur* k *pour tout* $k \in \{1, ..., n\}$ *:*

$$\textsc{AtMost}(1, \{x_1, ...x_n\}, k) \qquad \forall k \in \{1, .., n\} \tag{1.4}$$

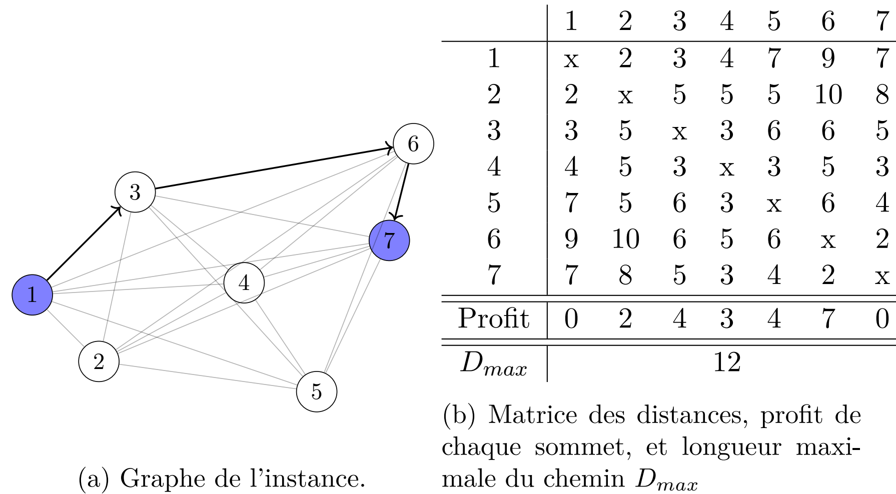
La longueur totale du chemin ne doit pas dépasser D_{max}*. On peut utiliser pour cela la contrainte globale* Element *en deux dimension :*

$$\sum_{i=1}^{n-1} d_{x_i, x_{i+1}} \leq D_{max} \tag{1.5}$$

Enfin, la fonction objectif effectue la somme des profits de tous les sommets visités et s'appuie également sur la contrainte globale Element *:*

$$f(\mathcal{X}) = \sum_{i=1}^{n} p_{x_i} \tag{1.6}$$

Une instance du problème de course d'orientation est fournie par la figure 1.1. Le graphe du problème est présenté dans la figure 1.1(a). Dans ce graphe, les noeuds 1 et 7 sont respectivement les noeuds de départ et d'arrivée. La matrice des distances de ce graphe, les profits associés à chaque noeud ainsi que la longueur totale du chemin sont donnés dans la figure 1.1(b). Une solution est proposée sur le graphe dans la figure 1.1(a). Elle visite les sommets 1, 3, 6 puis 7. Cette solution est de longueur 11 (3+6+2), et a un profit de 11 (0+4+7+0). Dans le modèle, cette solution est encodée par l'instantiation $I = (1, 3, 6, 7, 0, 0, 0)$*.*



(a) Graphe de l'instance.

	1	2	3	4	5	6	7
1	x	2	3	4	7	9	7
2	2	x	5	5	5	10	8
3	3	5	x	3	6	6	5
4	4	5	3	x	3	5	3
5	7	5	6	3	x	6	4
6	9	10	6	5	6	x	2
7	7	8	5	3	4	2	x
Profit	0	2	4	3	4	7	0
D_{max}	12						

(b) Matrice des distances, profit de chaque sommet, et longueur maximale du chemin D_{max}

FIGURE 1.1 – Exemple d'une instance du problème de course d'orientation

Pour trouver une solution à une instance d'un CSP ou d'un COP, plusieurs approches existent. Dans la section suivante nous présentons le principe d'une méthode de recherche arborescente.

1.1.2 Méthodes de recherche arborescente

Le principe d'une méthode de recherche arborescente est de séparer récursivement l'espace de recherche de sorte qu'à chaque étape, on crée plusieurs sous-problèmes à résoudre. Cette récursion engendre alors une structure arborescente à explorer, appelée arbre de recherche. L'ordre d'exploration de cet arbre de recherche est important et plusieurs ordres sont possibles comme la recherche en largeur, ou la recherche en profondeur. Dans cette section, on s'intéresse à l'exploration complète de cette arborescence par une **recherche en profondeur d'abord** avec retours-arrière (*Depth First Search*, DFS) sous l'angle de la programmation par contraintes.

Plus précisément, la racine de l'arborescence correspond au CSP d'origine. D'une manière générale, à chaque noeud de l'arbre, on crée plusieurs branches, chacune correspondant à une contrainte différente que l'on ajoute au CSP qui sera résolu par cette branche. Le plus souvent ces contraintes sont unaires, c'est à dire qu'a chaque noeud, on choisit une variable, et l'on va contraindre son domaine. Le type de contrainte ajoutée définit ce qu'on appelle la stratégie de branchement. Trois stratégies de branchement sont le plus souvent utilisées :

— *L'énumération* : Dans cette stratégie, on crée une branche par valeur du domaine de la variable sélectionnée, et la contrainte ajoutée à chacune des branches est une égalité. Autrement dit, on affecte une valeur différente à la variable dans chacune des branches.

— *La séparation de domaine* : Dans cette stratégie, la variable n'est pas forcément affectée, elle est séparée en deux avec une contrainte d'infériorité une contrainte de supériorité ou égalité (en trois avec une affectation).

— *La décision binaire* : Cette stratégie repose sur une affectation et une réfutation, c'est à dire une contrainte d'inégalité, pour une valeur de la variable sélectionnée. Généralement la branche gauche est une affectation, quand la branche droite est la réfutation.

Dans ce manuscrit, nous considérerons la décision binaire lorsqu'un solveur de programmation par contraintes sera employé, et l'énumération lorsque la recherche arborescente sera conçue ad-hoc.

Un algorithme simple de recherche en profondeur avec retours-arrière consiste à créer plusieurs branches à chaque noeud de l'arbre en fonction de la stratégie de branchement choisie, et de visiter la branche la plus à gauche. A chaque noeud, on peut aussi vérifier chaque contrainte dont les variables de sa portée sont affectées. Si la contrainte n'est pas satisfaite, alors cette branche est coupée car elle ne mène à aucune solution du CSP. On effectue alors un retour-arrière avant de continuer le long de la branche voisine la plus à gauche non visitée. Si les contraintes du problème sont vérifiées, lorsque toutes les variables sont affectées, alors une solution a été trouvée.

L'algorithme 1 présente une telle recherche en profondeur avec décisions binaires. Cet algorithme prend en entrée un CSP. A chaque étape, il choisit un couple (variable, valeur), et crée deux appels récursifs : le premier sur le CSP induit par l'affectation de la variable à la valeur, et le second créé par la réfutation de cette valeur pour cette variable. Chaque contrainte dont les variables de sa portée sont affectées est vérifiée et la branche courante est coupé si ce n'est pas le cas, opérant alors un retour-arrière dans la pile d'appels, on sait qu'aucune solution ne se trouve dans ce sous-arbre. Lorsqu'une telle impasse est détectée, on appelle cela un conflit. Lorsque toutes les variables sont affectées l'algorithme a trouvé une solution.

Algorithme 1 : Recherche en profondeur binaire pour un CSP

Données : $(\mathcal{X}, \mathcal{D}, \mathcal{C})$: un CSP

1 **Fonction** `recherche-binaire`$(\mathcal{X}, \mathcal{D}, \mathcal{C})$ **:**
2   **pour** $c \in \mathcal{C}$ **faire**
3   **si** $|\mathcal{D}(x)| = 1 \; \forall x \in Sc(c) \wedge \prod_{x \in Sc(c)} \mathcal{D}(x) \notin c$ **alors**
4   **retourner** *Faux*
5   **si** $\forall x \in \mathcal{X} : |\mathcal{D}(x)| = 1$ **alors**
6   **retourner** *Vrai*
7   Choisir (x_i, v_i) tel que $|\mathcal{D}(x_i)| > 1 \wedge v_i \in \mathcal{D}(x_i)$
8   $\mathcal{D}' \leftarrow (\mathcal{D}(x_1), ..., \mathcal{D}(x_{i-1}), \{v_i\}, \mathcal{D}(x_{i+1}), ..., \mathcal{D}(x_{|\mathcal{X}|}))$
9   $\mathcal{D}'' \leftarrow (\mathcal{D}(x_1), ..., \mathcal{D}(x_{i-1}), \mathcal{D}(x_i) \setminus \{v_i\}, \mathcal{D}(x_{i+1}), ..., \mathcal{D}(x_{|\mathcal{X}|}))$
10   **retourner** `recherche-binaire`$(\mathcal{X}, \mathcal{D}', \mathcal{C}) \vee$ `recherche-binaire`$(\mathcal{X}, \mathcal{D}'', \mathcal{C})$

Pour une instance d'un COP, on cherche la solution qui optimise la fonction objectif. Souvent on utilise une variable objectif, que l'on contraint alors par la

fonction objectif. Pour un problème de minimisation (resp. maximisation), un algorithme de type DFS peut être utilisé. Chaque nouvelle solution produite est une borne supérieure (resp. inférieure) du problème, et on ajoute alors une contrainte sur la variable objectif pour qu'elle soit inférieure (resp. supérieure) à cette borne. Le principe est de résoudre itérativement plusieurs problèmes de CSP, la solution optimale étant la dernière solution trouvée, la dernière contrainte rendant le problème infaisable. Il peut alors être utile avant la recherche arborescente de bien définir le domaine de la variable objectif, en précalculant des bornes, les plus serrées possibles pour accélérer la résolution.

La recherche arborescente telle que présentée dans cette section est une méthode dite *complète*, c'est à dire qu'aucune solution ne peut être manquée par la méthode, et l'on peut prouver l'infaisabilité. A l'opposé, une méthode incomplète peut manquer des solutions, et ne peut alors pas prouver l'infaisabilité. On parle également de méthodes exactes et approchées dans le contexte de l'optimisation.

1.1.2.1 Inférence et propagation

Pour élaguer l'espace de recherche, on peut utiliser des règles d'inférence qui vont réduire la taille des domaines en s'appuyant sur les contraintes. On utilise alors dans la recherche arborescente des algorithmes de filtrage que l'on appelle propagateurs. Formellement, un propagateur, p pour une contrainte c est une fonction qui associe à tout domaine $\mathcal{D}$ un domaine réduit $p(\mathcal{D}) \subseteq \mathcal{D}$ tel que :

— Soit un domaine $\mathcal{D}'$ tel que $\mathcal{D} \subseteq \mathcal{D}'$, alors $p(\mathcal{D}) \subseteq p(\mathcal{D}')$: le propagateur est monotone.

— Pour tout tuple $\tau \in \mathcal{D}$ tel que $\tau \in c$ on a $\tau \in p(\mathcal{D})$: le propagateur est dit *correct*.

— Si $|\mathcal{D}(x)| = 1$ pour tout $x \in Sc(c)$, alors $p(\mathcal{D}) = \mathcal{D}$ si $\prod_{x \in Sc(c)} \mathcal{D}(x) \in c$, sinon $p(\mathcal{D}) = \perp$: le propagateur doit pouvoir vérifier une contrainte.

Lorsqu'un propagateur retourne un domaine vide $\perp$, on considère qu'il y a un conflit car l'affectation actuelle ne peut pas s'étendre à une solution. A chaque noeud de l'arbre, les propagateurs sont appelés jusqu'à un point fixe, c'est à dire que soit le domaine est vide, soit le domaine n'est plus modifié par les propagateurs. Une telle procédure remplace alors les lignes 2 à 4 de l'algorithme 1. Chaque contrainte peut être associée à un ensemble de propagateurs. Il y a bien sûr un compromis à faire entre le niveau de filtrage voulu, et le temps passé à effectuer ce filtrage. On qualifie ce niveau de filtrage par la notion de cohérence. Plusieurs niveaux de cohérence sont considérés dans la littérature [24], le plus célèbre et le plus utilisé étant l'arc-cohérence [118] que l'on définit ici.

Soit une contrainte c définie sur $(x_{a_1}, ..., x_{a_k})$, on dit qu'une valeur $v \in \mathcal{D}(x_i)$ pour une variable $x_i \in Sc(c)$ a un support dans c si il existe un tuple $\tau \in c$, tel que $\tau[i] = v$ et que $\tau \in \prod_{x \in Sc(c)} \mathcal{D}(x)$. On dit alors qu'une contrainte c est arc-cohérente si et seulement si $\forall x \in Sc(c)$, $\forall v \in \mathcal{D}(x)$, il existe un support sur c

pour v. Un algorithme d'arc-cohérence consiste alors à retirer toutes les valeurs des variables qui n'ont pas de support sur une contrainte.

1.1.2.2 Heuristiques de branchement

Dans une recherche en profondeur, pour chacune des stratégies de branchement évoquées, il faut définir ce qu'on appelle communément l'*ordre des variables* à sélectionner et l'*ordre des valeurs* à explorer en premier.

L'ordre des variables définit la structure de l'arbre. Il est important, car un bon ordre de variables permet de découvrir les conflits plus tôt, et donc l'arbre de recherche comportera moins de noeuds. Un bon ordre des valeur est aussi important car il permet de trouver des solutions plus rapidement et in fine d'élaguer l'arbre. C'est donc l'ordre des variables qui dessine l'espace de recherche que l'on va explorer, mais l'ordre des valeurs qui va décider de la stratégie d'exploration de cet espace.

Pour obtenir des méthodes arborescentes efficaces, on doit donc trouver l'ordre des variables, et l'ordre des branches à visiter qui permettent de minimiser la taille de l'arbre de recherche. Ces ordres sont déterminés de manière heuristique, et on parle d'heuristiques de choix de variables et d'heuristique de choix de valeurs. Plus généralement, on trouve le terme d'heuristiques de branchement pour désigner ces heuristiques.

Enfin, ces heuristiques peuvent être statiques, c'est à dire que l'ordre peut être pré-établi avant la résolution du problème, et ne changera pas. Il peut aussi être dynamique et fonction du noeud courant, de l'état des domaines, etc. On peut également établir deux catégories selon un autre prisme, les heuristiques de branchement peuvent être génériques, adaptées à n'importe quelle problème, ou alors spécifiques, et donc dédiées à un problème en particulier.

Dans ce manuscrit on s'intéresse aux heuristiques de choix de valeurs, c'est à dire à quelle branche explorer en premier pour trouver une solution rapidement. Le type d'heuristique que l'on propose est spécifique au problème étudié, et est dynamique.

Ordre des variables L'approche la plus souvent utilisée pour la définition d'une heuristique de choix de variables est le principe du *First-Fail*, c'est à dire d'essayer en priorité là où on a le plus de chances d'échouer [71]. En effet, détecter un conflit le plus tôt possible dans l'arbre va permettre de réduire sa taille.

Parmi des exemples d'ordres statiques de choix de variables, on peut citer l'ordre lexicographique, c'est à dire l'ordre des variables apparaissant dans la description de l'instance du problème et l'ordre décroissant des degrés, défini par le nombre de contraintes dans laquelle la variable est impliquée. Pour les ordres dynamiques il existe des heuristiques simples telles que *Mindom* dont le principe est de sélectionner la variable dont le domaine courant est le plus petit, et d'autres plus sophistiquées telles que *Impact-Based Search* [145], ou bien *Weighted Degree* [26]. Dans *Impact-Based Search*, le principe est d'estimer l'impact du choix de la variable sur la réduction des domaines, sur la base des précédentes sélections de cette va-

riable dans d'autres zones de l'arbre de recherche. L'heuristique *Weighted Degree* maintient pour chaque contrainte un compte w du nombre de fois où elle a été responsable d'un échec. On sélectionne alors la variable qui maximise le quotient de la taille de son domaine par la somme des poids w des contraintes dans lesquelles elle est impliquée et dont au moins une autre variable n'est pas affectée.

Ordre des valeurs Il existe moins d'heuristiques pour l'ordre des valeurs. Une approche générique possible consiste à sélectionner une valeur pour laquelle on a déjà trouvé une solution pour ce couple variable/valeur [12].

Exemple *Pour le problème de course d'orientation, on peut choisir l'ordre lexicographique des variables x et une stratégie de branchement énumérative. L'ordre des valeurs consiste alors à choisir quel est le prochain noeud à visiter en premier en considérant les sommets que l'on a déjà vus. On peut par exemple prendre l'ordre décroissant d'utilité des sommets : $\frac{p_i}{d_{j,i}}$ pour le sommet i, avec j le dernier sommet visité.*

1.1.2.3 Redémarrage et randomisation

De légers changements dans les heuristiques de choix de variables ou de valeurs peuvent drastiquement modifier le temps d'exécution d'une recherche en profondeur car de mauvais choix à un niveau proche de la racine peut mener à une grande variabilité en terme de performance. Plus particulièrement, la difficulté d'un problème vient surtout de la combinaison d'une instance et de choix déterministes. Il a été montré sur des instances que l'on génère aléatoirement, qu'un algorithme déterministe a une probabilité non négligeable de rencontrer une instance d'un problème qui mettra un temps exponentiellement plus long que toutes les autres instances rencontrées jusqu'à présent [67]. A partir de ces observation, on constate une amélioration des temps de résolution en introduisant de l'aléatoire dans les heuristiques de choix de variables et de valeurs, ainsi qu'une politique de redémarrage. Le principe est de reprendre la recherche à partir de la racine lorsque la recherche atteint un certain seuil sur un critère donné (en gardant les bornes sur la fonction objectif ainsi que les informations collectées par les heuristiques de branchement). Généralement ce critère est le nombre de conflits, et le seuil est ajusté à chaque redémarrage. On comprend alors qu'en ajoutant de l'aléatoire dans les heuristiques, les parties de l'espace de recherche explorées ne seront pas les même d'un redémarrage à l'autre.

Deux politiques de redémarrage sont fréquemment utilisées. La première est une politique de redémarrage à croissance géométrique [173], qui fixe une limite sur le nombre de conflits de $b * f^{k-1}$ pour le k-ème redémarrage, où b pour base et f pour facteur, sont deux paramètres. La seconde politique de redémarrage est appelée *Luby* du nom d'un de ses auteurs [116], elle est basée sur la suite $1, 1, 2, 1, 1, 2, 4, 1, 1, 2, 1, 1, 2, 4, 8, ...$ Pour le k-ème redémarrage, on multiplie le k-ème nombre de la suite par une base b pour définir la limite en nombre de conflits. La suite est définie récursivement comme suit :

$$\texttt{Luby}(k) = \begin{cases} 2^{t-1} & \text{si } \exists t \in \mathbb{N}, k = t^2 - 1 \\ \texttt{Luby}(k - 2^{t-1} + 1) & \text{si } \exists t \in \mathbb{N}, 2^{t-1} \leq k < 2^t - 1 \end{cases} \tag{1.7}$$

1.1.3 Autres approches

Pour résoudre des problèmes d'optimisations combinatoire il existe d'autres méthodes que celles basées sur la programmation par contraintes. Certaines sont exactes comme par exemple les méthodes issues de la programmation linéaire en nombres entiers. D'autres méthodes produisent des solutions approchées comme les algorithmes gloutons, algorithmes de recherche locale, ou les métaheuristiques.

Programmation Linéaire en Nombres Entiers (PLNE) Modéliser un problème de minimisation (resp. maximisation) en PLNE consiste à le décrire sous forme d'équations linéaires impliquant des variables entières. La résolution s'appuie sur une recherche arborescente, que l'on appelle procédure de séparation et d'évaluation (*Branch and Bound*). Elle utilise généralement une stratégie de branchement de type séparation de domaines, et à chaque noeud, une borne inférieure (resp. supérieure) de la solution est calculée par une relaxation du problème c'est à dire une version simplifiée du problème que l'on peut résoudre facilement. La plus connue est la relaxation continue, dans laquelle les variables peuvent prendre des valeurs non entières. Dans ce cas, la stratégie de branchement consiste à sélectionner une variable dont la valeur dans la solution relâchée n'est pas entière, et séparer l'espace de recherche en deux parties avec une branche où on la force à être inférieure à sa partie entière, et l'autre où on la force à être supérieure ou égale à sa partie entière. A tout moment, si la borne inférieure est supérieure à la borne supérieure (resp. la borne supérieure est inférieure à la borne inférieure), c'est à dire à la meilleure solution trouvée jusqu'à présent, alors la branche est coupée. On retrouve souvent un ordre d'exploration de type *meilleur d'abord* (*Best First*), dans lequel le prochain noeud sélectionné n'est pas le plus profond de l'arbre mais celui qui optimise un critère. On parle alors plutôt d'heuristique de sélection de noeud que de sélection de valeur dans ce contexte. Une heuristique fréquente est de sélectionner le noeud qui a la plus petite (resp. grande) borne inférieure (resp. supérieur) pour un problème de minimisation (resp. maximisation).

Exemple *Le problème de course d'orientation peut être modélisé par un programme linéaire en nombres entiers. Dans ce modèle il y a deux types de variables :*

— *Pour tout paire de sommet i, j : $x_{i,j}$ vaut* 1 *si la visite du sommet j suit celle du sommet i,* 0 *sinon.*

— *Pour tout sommet i : une variable intermédiaire $u_i \in \mathcal{N}$ est créée pour pouvoir exprimer une contrainte permettant d'empêcher de faire des boucles.*

Le modèle s'exprime ainsi :

$$\max \sum_{i=2}^{n-1} \sum_{j=2}^{n} p_i x_{i,j} \tag{1.8}$$

$$\sum_{j=2}^{n} x_{1,j} = \sum_{i=2}^{n-1} x_{i,n} = 1 \tag{1.9}$$

$$\sum_{i=1}^{n-1} x_{i,k} = \sum_{j=2}^{n} x_{k,j} \leq 1 \quad \forall k \in \{2, ..., n-1\} \tag{1.10}$$

$$\sum_{i=1}^{n-1} \sum_{j=2}^{n} d_{i,j} x_{i,j} \leq D_{max} \tag{1.11}$$

$$u_i - u_j + 1 \leq (n-1)(1 - x_{i,j}) \quad \forall \{i,j\} \in \{2, ..., n\} \tag{1.12}$$

$$2 \leq u_i \leq n \quad \forall i \in \{2, ..., n\} \tag{1.13}$$

$$x_{i,j} \in \{0, 1\} \quad \forall \{i,j\} \in \{2, ..., n\} \tag{1.14}$$

$$u_i \in \mathbb{N} \quad \forall i \in \{2, ..., n\} \tag{1.15}$$

La fonction objectif, qui est la maximisation du profit est exprimée par l'équation 1.8. La première contrainte (eq 1.9) permet que le départ et l'arrivée soit visités. La contrainte 1.10 assure la connectivé du chemin, et que chaque sommet soit au plus visité une fois. L'équation 1.11 traduit le respect de longueur maximale du chemin. Les deux contraintes suivantes (eq 1.12 et 1.13) permettent d'éviter au chemin de former des sous-tours en ordonnant les passages des sommets visités. Enfin les deux dernières contraintes sont appelées des contrainte d'intégrité et définissent le domaine des variables (binaire, entière ou continue).

Algorithmes gloutons Appliquer à chaque étape les choix de variables et de valeurs déterminés par une heuristique de branchement conduit à un algorithme glouton. Ce type d'algorithme ne remet pas en cause les choix successifs effectués et ne développe qu'une branche d'un arbre de recherche. Pour certains problèmes spécifiques, il existe des algorithmes gloutons garantissant l'obtention de la solution optimale (par exemple pour déterminer un arbre couvrant de coût minimal dans un graphe) ou garantissant la qualité d'une solution par rapport à la solution optimale, on parle alors d'algorithmes d'approximation (par exemple pour déterminer dans un graphe une couverture par les sommets de cardinalité minimale). Dans le cas général, un algorithme glouton ne présente aucune garantie sur l'optimalité des solutions retournées. De plus, lorsque le problème est fortement contraint, même l'obtention d'une solution réalisable n'est pas garantie en utilisant un algorithme glouton (comme par exemple pour le problème de voyageur de commerce avec contraintes de fenêtres de temps).

Recherche locale Une méthode de recherche locale ne permet pas d'explorer tout l'espace de recherche et détermine des solutions approchées sans garantie d'op-

timalité.

Une recherche locale consiste à partir d'une solution initiale, à se déplacer dans l'espace de recherche de proche en proche. On définit la notion de voisinage par l'ensemble des solutions que l'on peut atteindre à partir d'une solution donnée. À chaque itération de ce type d'algorithme, on choisit une solution dans le voisinage de la solution courante qui devient la nouvelle solution courante, et ainsi de suite jusqu'à un critère d'arrêt. Généralement la meilleure solution voisine est choisie (celle qui a une meilleure valeur pour la fonction objectif), ce qui implique que l'algorithme finit dans un minimum local, c'est à dire qu'aucune solution voisine n'est améliorante. A partir de cet algorithme (appelé descente), plusieurs méthodes ont été proposées soit pour sortir des minima locaux, comme la recherche tabou [64] ou la recherche à voisinages variables [126], soit pour intégrer de l'aléatoire dans la sélection des voisins avec le recuit simulé [96]. On appelle de telles méthodes des métaheuristiques.

Exemple *Pour le problème de course d'orientation, un exemple de voisinage est l'ensemble des solutions accessibles en remplaçant un sommet de la solution actuelle par un sommet qui n'a pas encore été visité. Un second voisinage trivial et l'ensemble des solutions que l'on peut obtenir en ajoutant un sommet à la solution courante, par exemple le sommet le plus proche du dernier sommet courant de la solution partielle.*

1.2 Apprentissage par renforcement

Dans ce manuscrit, nous utilisons l'apprentissage par renforcement pour apprendre des heuristiques de choix de valeurs, heuristiques qui seront ensuite intégrées dans des méthodes de recherche arborescente. L'apprentissage par renforcement semble être une méthode d'apprentissage adaptée pour être utilisée dans ce cadre. Tout d'abord, les méthodes d'apprentissage par renforcement ne nécessitent pas de données étiquetées contrairement aux méthodes d'apprentissage supervisé. Des données étiquetées dans le cadre de l'optimisation combinatoire peuvent être difficile à obtenir. En effet, si nous cherchons à évaluer chaque décision à partir d'un noeud donné, les étiquettes prendraient alors la forme d'une valeur d'objectif pour chacune des décisions. Pour obtenir nos données d'apprentissage il nous faudrait connaître la valeur optimale de la fonction objectif pour une grande quantité de noeuds différents, il serait alors difficile de les obtenir sans explorer l'arbre de recherche entièrement.

L'autre avantage de l'apprentissage par renforcement est que ce cadre s'adapte bien au problème d'optimisation combinatoire. En effet, la résolution d'un tel problème peut se présenter sous la forme d'un processus de décision markovien, qui est le formalisme utilisé en apprentissage par renforcement. Dans cette section, nous décrivons les bases de l'apprentissage par renforcement et des processus de décision markovien, avant de décrire l'algorithme REINFORCE que nous utilisons dans la suite du manuscrit. Pour une introduction plus complète de la discipline,

nous renvoyons vers l'ouvrage de référence "Reinforcement Learning: An Introduction" [163]. Enfin, nous donnerons ensuite un aperçu bibliographique de l'utilisation de l'apprentissage par renforcement dans le cadre de l'optimisation combinatoire.

1.2.1 Principes de l'apprentissage par renforcement

L'apprentissage par renforcement est la tâche d'apprendre à un agent à interagir avec un environnement de façon à maximiser des récompenses reçues au cours du temps. L'agent doit découvrir les actions à effectuer pour les différents états de l'environnement afin de réaliser son objectif sur le long terme. Son but est d'apprendre une association entre les différents états de l'environnement et les actions à réaliser. Il s'agit d'un problème de prise de décisions séquentielles dans lequel une action peut influencer les récompenses ainsi que les états de l'environnement dans le futur. À chaque étape, l'agent prend une décision en fonction de l'état courant de l'environnement, ce qui va à la fois modifier l'environnement, et lui attribuer une récompense.

La formalisation d'un processus d'apprentissage par renforcement repose classiquement sur un processus de décision markovien (pour *Markov Decision Process*, MDP) [14]. Un MDP est caractérisé par un quadruplet $\{S, A, \tau, R\}$ avec :

- Un ensemble d'états S qui caractérisent l'environnement.
- Un ensemble d'actions A qui décrivent les prises de décisions de l'agent. On note $A(s)$ l'ensemble des actions qui peuvent être prises dans l'état $s \in S$.
- Une fonction de transition $\tau : S{*}A{*}S \rightarrow [0, 1]$ pour représenter le passage d'un état à un autre en fonction d'une action. $\tau(s, a, s')$ représente la probabilité de passer dans l'état s' lorsque l'on est dans l'état s et que l'on prend l'action a.
- Une fonction de récompense $R : S * A * S \rightarrow \mathbb{R}$ qui associe à chaque transition un réel [1].

Les interactions entre agent et environnement sont schématisées dans la figure 1.2. A chaque étape t du problème de décisions séquentielles, connaissant l'état courant $s_t \in S$ de l'environnement, l'agent prend une décision, l'action $a_t \in A(s_t)$. L'environnement transmet à l'agent le nouvel état s_{t+1} et la récompense associée $R(s_t, a_t, s_{t+1}) = r_{t+1} \in \mathbb{R}$.

Lorsqu'il existe au moins un état terminal, les prises de décisions séquentielles de l'agent correspondent à un *épisode*. Par exemple, on retrouve ce type de processus dans les jeux, où lorsqu'une partie est finie, une nouvelle partie indépendante de la première peut recommencer à partir d'un nouvel état initial. Chaque partie du jeu correspond ainsi à un épisode.

Une politique de décision, notée $\pi(a \mid s)$ consiste à déterminer quelle action $a \in A(s)$ appliquer pour chaque état $s \in S$. La politique de décision peut être

1. Nous choisissons d'utiliser une fonction de récompense déterministe pour alléger certaines notations, une fonction de récompense plus générale se définit par $R : S * A * S * \mathbb{R} \rightarrow [0, 1]$.

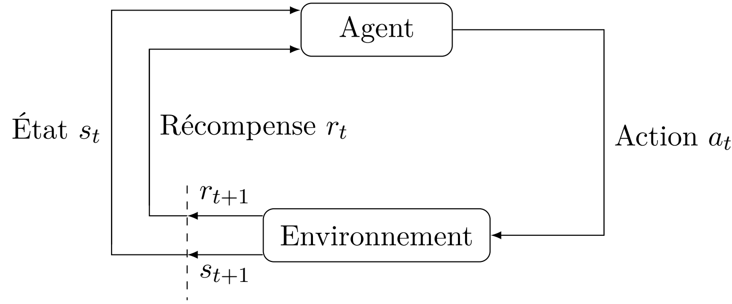


FIGURE 1.2 – Interactions entre l'agent et l'environnement ([163])

stochastique $\pi : S * A \rightarrow [0, 1]$ et associer à chaque état s une distribution de probabilités sur les actions possibles : $\sum_{a \in A(s)} \pi(s, a) = 1$. La politique de décision peut aussi être déterministe, elle est alors définie par $\pi : S \rightarrow A$. Le but de l'apprentissage par renforcement est de déterminer la politique de décision (stochastique ou déterministe) optimale, c'est à dire celle qui va maximiser en moyenne la somme des récompenses au cours du temps. Pour un épisode donné, en notant T le nombre d'étapes pour parvenir à atteindre un état terminal, on appelle *gain*, noté $G(t)$, la somme des récompenses obtenues à partir de l'étape t pour cet épisode :

$$G(t) = r_{t+1} + r_{t+2} + \ldots + r_T \tag{1.16}$$

Il est courant d'ajouter un facteur $\gamma < 1$ exponentiellement décroissant pour calculer cette valeur. Ce facteur est initialement prévu pour le cas non épisodique, dans lequel $T = \infty$). La valeur du *gain* à une étape t s'écrit alors :

$$G(t) = \sum_{k=t+1}^{T} \gamma^{k-t-1} r_k \tag{1.17}$$

$$= r_{t+1} + \gamma G(t+1) \tag{1.18}$$

Pour une politique π donnée, on peut alors définir des fonctions d'évaluation pour les états et pour les actions :

— $v_\pi(s)$: fonction d'évaluation de l'état s pour la politique π (*State-Value Function*). Elle correspond à l'espérance du *gain* en suivant la politique π à partir de l'état s :

$$v_\pi(s) = \mathbb{E}_\pi[G(t) \mid s_t = s] \tag{1.19}$$

où $\mathbb{E}_\pi[.]$ veut dire espérance lorsque l'agent suit la politique π et t représente n'importe quelle étape.

— $q_\pi(s, a)$: fonction d'évaluation de l'action a à partir d'un état s pour la politique π (*Action-Value Function*). Elle représente l'espérance du *gain* en pre-

nant l'action a à partir de l'état s puis en suivant la politique π :

$$q_\pi(s,a) = \mathbb{E}_\pi[G(t) \mid s_t = s, a_t = a] \tag{1.20}$$

Ces deux définitions peuvent être réécrites pour faire apparaître la relation qu'il y a entre un état et ses états successeurs. En effet, on peut représenter les états successeurs en faisant une somme sur toutes les actions possibles, puis pour chacune, une seconde somme sur toute les transitions possibles. On pondère l'expression par la probabilité d'une action (selon la politique), et par la probabilité de la transition. Les équations 1.24 et 1.28 sont appelées équations de Bellman.

$$\begin{aligned}
v_\pi(s) &= \mathbb{E}_\pi[G(t) \mid s_t = s] &(1.21)\\
&= \mathbb{E}_\pi[r_{t+1} + \gamma G(t+1)] &(1.22)\\
&= \sum_a \pi(a \mid s) \sum_{s',r} \tau(s,a,s')(r + \gamma \mathbb{E}_\pi[G(t+1) \mid s_{t+1} = s']) &(1.23)\\
&= \sum_a \pi(a \mid s) \sum_{s',r} \tau(s,a,s')(r + \gamma v_\pi(s')) &(1.24)
\end{aligned}$$

$$\begin{aligned}
q_\pi(s,a) &= \mathbb{E}_\pi[G(t) \mid s_t = s, a_t = a] &(1.25)\\
&= \mathbb{E}_\pi[r_{t+1} + \gamma G(t+1)] &(1.26)\\
&= \sum_{s',r} \tau(s,a,s')(r + \gamma \mathbb{E}_\pi[G(t+1) \mid s_{t+1} = s']) &(1.27)\\
&= \sum_{s',r} \tau(s,a,s')(r + \gamma \sum_{a'} \pi(a' \mid s') q_\pi(s',a')) &(1.28)
\end{aligned}$$

L'objectif de l'apprentissage par renforcement est de trouver une politique optimale π^*, c'est à dire telle que pour tout état s et pour toute politique π, $v_{\pi^*}(s) \geq v_\pi(s)$.

On peut définir la valeur optimale de chaque état, qui correspond à la valeur de l'état sous la meilleure politique :

$$v^*(s) = \max_\pi v_\pi(s), \quad \forall s \in S \tag{1.29}$$

On peut également définir la valeur optimale pour chaque action de la même manière :

$$q^*(s,a) = \max_\pi q_\pi(s,a), \quad \forall s \in S \ \forall a \in A(s) \tag{1.30}$$

Soit une politique optimale π^*, on a donc $v_{\pi^*}(s) = v^*(s)$ et $q_{\pi^*}(s,a) = q^*(s,a)$ pour tout état $s \in S$ et toute action $a \in A(s)$.

Enfin, l'équation (1.29) peut être ré-écrite en fonction de (1.30). En effet, la valeur optimale d'un état est égale à la valeur optimale de la meilleure action. On obtient alors une forme indépendante de toute politique. L'équation obtenue 1.36

est appelée équation d'optimalité de Bellman :

$$
\begin{aligned}
v^*(s) &= \max_{a \in A(s)} q^*(s,a) && (1.31)\\
&= \max_{a \in A(s)} q_{\pi^*}(s,a) && (1.32)\\
&= \max_{a \in A(s)} \mathbb{E}_{\pi^*}[G(t) \mid s_t = s, a_t = a] && (1.33)\\
&= \max_{a \in A(s)} \mathbb{E}_{\pi^*}[R_{t+1} + \gamma G(t+1) \mid s_t = s, a_t = a] && (1.34)\\
&= \max_{a \in A(s)} \mathbb{E}[R_{t+1} + \gamma v^*(s_{t+1})] \mid s_t = s, a_t = a] && (1.35)\\
&= \max_{a \in A(s)} \sum_{s',r} \tau(s,a,s')r + \gamma v^*(s') && (1.36)
\end{aligned}
$$

On peut également écrire l'équation d'optimalité de Bellman pour la valeur optimale des actions $q_{\pi^*}(s,a)$:

$$
q^*(s,a) = \sum_{s',r} \tau(s,a,s')(r + \gamma \max_{a' \in A(s')} q^*(s',a')) \tag{1.37}
$$

Les équations d'optimalité de Bellman représentent un système d'équations non linéaires, avec une équation par état (pour la fonction de valeur d'état), où le nombre d'inconnues est le nombre d'états. Une fois le système d'équations résolu, on connaît alors la valeur optimale de chaque état. Il est ensuite facile de déterminer une politique optimale à partir de là. En effet, un algorithme glouton choisissant l'action menant à l'état dont la valeur optimale est la plus élevée sera alors un algorithme optimal. De la même manière, si l'on connaît la valeur optimale de chaque action (si l'on résout le système d'équations donné par 1.37), alors l'algorithme glouton qui choisit l'action qui maximise cette valeur à chaque étape sera optimal.

Dans la section suivante, on donne deux exemples d'algorithme pour trouver une politique optimale dans un MDP. Ces deux algorithmes peuvent résoudre des problèmes d'une taille raisonnable seulement, car ils nécessitent une taille en mémoire linéaire par rapport au nombre d'états ou de paires état-action qui eux peuvent être exponentiellement grands.

1.2.2 Algorithmes d'apprentissage par renforcement

Dans cette section, on décrit le fonctionnement de deux algorithmes d'apprentissage par renforcement, le premier algorithme est basé sur la programmation dynamique et nécessite une connaissance complète de l'évolution de l'environnement. Le second algorithme, appelé *Sarsa*, ne nécessite pas cette connaissance, et il se base sur des épisodes pour estimer les valeurs des actions. Ces deux algorithmes nécessitent une place en mémoire importante car ils stockent la valeur de chaque état pour le premier, et la valeur de chaque paire état-action pour le second. Ils ne

sont alors pas adaptés pour les problèmes à très grande échelle.

1.2.2.1 Programmation dynamique

Les algorithmes d'apprentissage par renforcement doivent souvent répondre à deux problématiques : évaluer une politique donnée π, c'est à dire calculer les valeurs $v_\pi(s)$ pour tout état $s \in S$ (ou évaluer $q_\pi(s, a)$ pour toutes les actions $a \in A(s)$ et pour tout $s \in S$), mais également trouver la politique optimale.

Tout d'abord, on peut évaluer par programmation dynamique une politique donnée π en utilisant l'équation de Bellman (1.24) comme une règle de mise à jour. Soit $v_k(s)$, l'estimation de $v_\pi(s)$ à l'itération k, avec $v_0(s)$ choisi arbitrairement pour tout $s \in S$. On calcule alors $v_{k+1}(s)$ comme suit :

$$v_{k+1}(s) = \sum_a \pi(a \mid s) \sum_{s',r} \tau(s, a, s')(r + \gamma v_k(s')) \tag{1.38}$$

Pour un nombre suffisamment grand d'itérations, un algorithme basé sur cette règle de mise à jour converge vers la valeur réelle de chaque état. En pratique, il suffit de s'arrêter lorsque $\max_{s \in S}(v_{k+1}(s) - v_k(s))$ représentant l'écart maximal de valeur d'un état entre deux itérations, est suffisamment petit.

Pour déterminer la politique optimale déterministe par programmation dynamique, on s'appuie sur cette règle de mise à jour afin d'évaluer la politique courante π_t, puis pour calculer une nouvelle politique π_{t+1} telle que $\pi_{t+1}(s) = \arg\max_a \sum_{s',r} \tau(s, a, s')(r + \gamma v_{\pi_t}(s'))$. Ces deux étapes sont répétées jusqu'à convergence de la politique entre deux itérations : $\pi_t = \pi_{t+1}$.

Ce premier algorithme de programmation dynamique nécessite, à chaque nouvelle politique, plusieurs itérations pour le calcul de la valeur des états de la politique courante avant la mise à jour de la politique. L'algorithme 2 s'inspire de la méthode décrite ci-dessus en combinant les deux étapes en une seule. En effet, l'étape de mise à jour ne se fait plus en fonction de tous les états successeurs, mais seulement en fonction du meilleur état successeur.

Cet algorithme que l'on appelle itération de valeur (*Value Iteration*) [143, 21] converge alors généralement plus rapidement que la première méthode décrite précédemment. Il utilise également un unique tableau V pour stocker l'évaluation courante de chaque état, ce qui améliore également la convergence. En effet, lors de la mise à jour il se peut que l'on utilise un état déjà mis à jour dans cette itération, on prend donc en compte les informations plus rapidement [22].

1.2.2.2 Sarsa

L'algorithme Sarsa [151, 162] (algorithme 3), cherche à estimer la valeur optimale des actions q^* à partir d'interactions avec l'environnement. L'algorithme génère des épisodes, et l'apprentissage se fait via ces épisodes. Ce type d'algorithme peut être alors utilisé lorsqu'on ne connaît pas entièrement la dynamique de l'environnement contrairement à l'algorithme présenté précédemment. Comme exposé dans la sec-

Algorithme 2 : Itération de valeur

Données : $\theta > 0$: paramètre de seuil pour la précision de l'algorithme
1 $\forall s \in S : V(s) \leftarrow 0$
2 **répéter**
3 $\quad \Delta \leftarrow 0$
4 $\quad$ **pour** $s \in S$ **faire**
5 $\quad\quad v \leftarrow V(s)$
6 $\quad\quad V(s) \leftarrow \max_a \sum_{s',r} \tau(s,a,s')(r + \gamma V(s'))$
7 $\quad\quad \Delta = \max(\Delta, |V(s) - v|)$
8 **jusqu'à** $\Delta < \theta$;
9 $\forall s \in S : \pi(s) \leftarrow \arg\max_a \sum_{s',r} \tau(s,a,s')(r + \gamma V(s'))$
10 **retourner** π

tion précédente (section 1.2.1), une fois les valeurs de q^* obtenues, il est facile de déterminer une politique optimale, en agissant de façon gloutonne vis à vis de cette valeur.

La méthode stocke une table Q correspondant à une estimation de q^* pour chaque paire état-valeur. Chaque itération de l'algorithme consiste à générer un épisode en étant glouton vis à vis de Q, puis en mettant à jour la table par les connaissances générées par cet épisode, affinant alors cette estimation. Néanmoins, rester glouton vis à vis de Q ne permet pas l'exploration. En effet, pour être en mesure de trouver la valeur optimale des actions, toutes les actions doivent être suffisamment sélectionnées, on doit alors être en mesure de sélectionner n'importe quelle action à tout moment, pas seulement la meilleure du moment. Cette problématique est connue comme le problème de l'exploration. On parle aussi généralement du dilemme exploitation/exploration. L'exploitation consiste à continuer à être glouton par rapport aux connaissances que l'on a acquises jusque là, au détriment de l'exploration qui nous permet de découvrir de nouvelles voies. Pour remédier à ce problème, on peut utiliser une politique ε-*greedy*. Cette politique, paramétrée par un $\varepsilon > 0$ choisit alors la meilleure action au regard de Q avec une probabilité de $1-\varepsilon$, et une action aléatoire le reste du temps. Cela permet alors, à horizon infini, de sélectionner toutes les actions un nombre infini de fois. Le principe de l'algorithme Sarsa est de mettre à jour Q à chaque étape de l'épisode en utilisant l'observation de la récompense et l'estimation de la prochaine action prise au cours de l'épisode. La méthode utilise alors un paramètre de pas d'apprentissage α pour faire cette mise à jour.

1.2.3 Fonction d'approximation

Le nombre d'états d'un problème peut être arbitrairement grand. Par conséquent, il est impossible d'utiliser les méthodes précédentes aussi bien d'un point de vue de la place mémoire que du point de vue du temps nécessaire pour visiter tous les états et actions suffisamment de fois. Pour pallier à cette difficulté, on va

Algorithme 3 : Sarsa

Données : $\alpha \in]0,1]$: pas d'apprentissage, $\varepsilon > 0$

1 $\forall s \in S, \forall a \in A(s) : Q(s,a) \leftarrow 0$
2 **répéter**
3 $s \leftarrow$ initial
4 $a \leftarrow \begin{cases} \arg\max_a Q(s,a) & \text{avec une probabilité } 1-\varepsilon \\ \text{Valeur aléatoire parmis } A(s) & \text{avec une probabilité } \varepsilon \end{cases}$
5 **répéter**
6 Appliquer a, recevoir r et s'
7 $a' \leftarrow \begin{cases} \arg\max_a Q(s',a) & \text{avec une probabilité } 1-\varepsilon \\ \text{Valeur aléatoire parmis } A(s') & \text{avec une probabilité } \varepsilon \end{cases}$
8 $Q(s,a) = Q(s,a) + \alpha(r + \gamma Q(s',a') - Q(s,a))$
9 $s \leftarrow s'$
10 $a \leftarrow a''$
11 **jusqu'à** *s terminal*;
12 **jusqu'à** *convergence*;

chercher à généraliser l'expérience acquise sur une partie des états pour l'utiliser sur d'autres états. Pour cela, on approxime la valeur des états (ou la valeur des actions) par une fonction d'approximation, paramétrée par un vecteur de paramètres $\boldsymbol{\theta} \in \mathbb{R}^d$ dont la dimension d est bien plus petite que le nombre d'états. Cette fonction prend alors en entrée un état, c'est à dire un ensemble d'attributs le caractérisant. On note $\hat{v}_\pi(s, \boldsymbol{\theta}) \approx v_\pi(s)$, l'approximation de la valeur de l'état s pour la politique courante π et un vecteur de poids $\boldsymbol{\theta}$. Le nombre de paramètres étant alors plus petit que le nombre d'états, modifier la valeur de $\boldsymbol{\theta}$ change donc l'estimation de plusieurs états en même temps. Cette fonction d'approximation peut prendre plusieurs formes, allant de la combinaison linéaire d'attributs de l'état s, jusqu'à un réseau de neurones profond prenant en entrée ces mêmes attributs.

On peut ainsi combiner l'apprentissage par renforcement avec les techniques de l'apprentissage supervisé. En apprentissage supervisé, on dispose de données étiquetées. Pour chaque exemple x d'un jeu de données, on connaît son étiquette y, la tâche d'apprentissage supervisé consiste à appromixer ces valeurs des étiquettes au travers d'exemples. Pour ce faire, on minimise généralement l'erreur quadratique moyenne d'une fonction d'approximation, notée $f(x, \boldsymbol{\theta})$:

$$\min_{\boldsymbol{\theta}} \sum_{(x,y)} (y - f(x, \boldsymbol{\theta}))^2$$

Idéalement, en apprentissage par renforcement, les étiquettes sont les valeurs des états (ou des actions) $v_\pi(s)$ d'une politique donnée pour chaque état (ou action).

On cherche donc à minimiser :

$$\min_{\boldsymbol{\theta}} \sum_{s \in S} \mu_\pi(s)(v_\pi(s) - \hat{v}_\pi(s, \boldsymbol{\theta}))^2$$

avec $\mu_\pi(s)$ la probabilité d'être dans l'état s en suivant la politique π. Ce terme est nécessaire car on veut être plus précis pour les états rencontrés plus souvent. Si on disposait de $v_\pi(s)$, on pourrait alors pratiquer une descente de gradient et mettre à jour $\boldsymbol{\theta}$ à chaque étape t de l'épisode. Cette mise à jour est définie comme suit :

$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \frac{1}{2}\alpha \nabla (v_\pi(s_t) - \hat{v}_\pi(s_t, \boldsymbol{\theta}))^2 \quad (1.39)$$

$$= \boldsymbol{\theta} + \alpha(v_\pi(s_t) - \hat{v}_\pi(s_t, \boldsymbol{\theta}))\nabla \hat{v}_\pi(s_t, \boldsymbol{\theta}) \quad (1.40)$$

avec α le paramètre de pas d'apprentissage, et $\nabla f(\boldsymbol{\theta})$ pour toute expression $f(\boldsymbol{\theta})$ est le vecteur des dérivées partielles de l'expression par rapport aux composantes du vecteur $\boldsymbol{\theta}$. Le terme $\frac{1}{2}$ est utilisé pour la concision, car il fait disparaître le 2 qui vient du terme en carré en développant l'expression, sans changer la validité de l'algorithme.

Bien entendu on ne dispose pas de la valeur de $v_\pi(s_t)$, étant donné que c'est ce que l'on cherche à calculer. On va alors utiliser comme étiquette la cible de la mise à jour. Par exemple pour l'algorithme 3, la mise à jour est la suivante :

$$Q(s, a) = Q(s, a) + \alpha(r + \gamma Q(s', a') - Q(s, a))$$

On cherche ici à ce que la valeur de $Q(s, a)$ se rapproche de celle de $r + \gamma Q(s', a')$. On peut alors voir la valeur de $r + \gamma Q(s', a')$ comme l'étiquette correspondant à l'entrée (s, a). Comme on utilise une fonction paramétrée à la place de la table Q, notre étiquette sera alors $r + \gamma \hat{Q}(s', a', \boldsymbol{\theta})$.

La règle de mise à jour à l'étape t d'un épisode pour l'algorithme Sarsa devient ainsi :

$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \alpha(r_{t+1} + \gamma \hat{Q}(s_{t+1}, a_{t+1}, \boldsymbol{\theta}) - \hat{Q}(s_t, a_t, \boldsymbol{\theta}))\nabla \hat{Q}(s_t, a_t, \boldsymbol{\theta}) \quad (1.41)$$

Les algorithmes de gradient basés sur une règle de mise à jour dont l'étiquette dépend elle même de $\boldsymbol{\theta}$ sont appelés *semi-gradient*, car ces régles ne comportent qu'une partie seulement du vrai gradient, elles prennent en compte l'effet du changement de $\boldsymbol{\theta}$ sur l'estimation, mais pas sur l'étiquette.

L'algorithme 4 illustre une adaptation de Sarsa (algorithme 3) en utilisant une fonction d'approximation plutôt qu'une table.

1.2.4 Algorithme REINFORCE

Dans la suite du manuscrit, on s'intéresse à une méthode d'apprentissage par renforcement qui ne s'appuie pas sur les fonctions de valeur d'un état ou d'une action, mais qui s'intéresse directement à une politique paramétrée par un vecteur

Algorithme 4 : Sarsa avec fonction d'évaluation

Données : Fonction de valeur d'action paramétré différentiable
$\hat{q} : S * A * \mathbb{R}^d \to \mathbb{R}$, $\alpha \in]0, 1]$: pas d'apprentissage, $\varepsilon > 0$

1 $\forall w \in \boldsymbol{\theta} : w \leftarrow 0$
2 **répéter**
3 $s \leftarrow$ initial
4 $a \leftarrow \begin{cases} \arg\max_a \hat{Q}(s, a, \boldsymbol{\theta}) & \text{avec une probabilité } 1 - \varepsilon \\ \text{Valeur aléatoire parmis } A(s) & \text{avec une probabilité } \varepsilon \end{cases}$
5 **répéter**
6 Appliquer a, recevoir r et s'
7 **si** s' *est terminal* **alors**
8 $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta}(R - \hat{Q}(s, a, \boldsymbol{\theta}))\nabla\hat{Q}(s, a, \boldsymbol{\theta})$
9 **sinon**
10 $a' \leftarrow \begin{cases} \arg\max'_a \hat{Q}(s', a', \boldsymbol{\theta}) & \text{avec une probabilité } 1 - \varepsilon \\ \text{Valeur aléatoire parmis } A(s') & \text{avec une probabilité } \varepsilon \end{cases}$
11 $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta}(R + \gamma\hat{Q}(s', a', \boldsymbol{\theta}) - \hat{Q}(s, a, \boldsymbol{\theta}))\nabla\hat{Q}(s, a, \boldsymbol{\theta})$
12 $a \leftarrow a'$
13 $s \leftarrow s'$
14 **jusqu'à** s *terminal*;
15 **jusqu'à** *convergence*;

$\boldsymbol{\theta} \in \mathbb{R}^d$ de taille d. Un avantage d'une telle approche et que l'on peut incorporer des connaissances du problème considéré directement dans une politique. On note alors $\pi_{\boldsymbol{\theta}}(a \mid s)$ la probabilité de choisir l'action a dans l'état s pour la politique π paramétrée par le vecteur $\boldsymbol{\theta}$. Dans le but de maintenir l'exploration, on veut que la politique reste stochastique, et que pour tout état s et toute action $a \in A(s)$, $\pi_{\boldsymbol{\theta}}(a \mid s) > 0$ pour toute valeur de $\boldsymbol{\theta}$. L'objectif est alors de trouver la valeur de $\boldsymbol{\theta}$ qui va maximiser une mesure de performance habituellement notée $J(\boldsymbol{\theta})$. Dans le cas de tâche épisodique qui nous intéresse, la performance est la valeur de l'état initial en suivant la politique paramétrée : $J(\boldsymbol{\theta}) = v_{\pi_{\boldsymbol{\theta}}}(s_0)$, avec s_0 l'état initial, c'est à dire l'espérance de la somme des récompenses obtenues en suivant l'heuristique paramétrée pour un épisode. La méthode que l'on présente dans cette partie opère une ascension de gradient sur cette mesure de performance, pour mettre à jour itérativement la valeur du vecteur $\boldsymbol{\theta}$ de la façon suivante :

$$\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t + \alpha * \nabla J(\boldsymbol{\theta}_t) \tag{1.42}$$

avec t l'itération en cours, et α est appelé dans ce contexte le pas d'apprentissage (*learning rate*). Ce type de méthode est appelée méthode de gradient de la politique (*policy gradient method*).

La question est alors de savoir comment calculer le gradient de cette mesure de performance. En effet, il est difficile de savoir comment faire varier $\boldsymbol{\theta}$ pour améliorer

la valeur de $J(\boldsymbol{\theta})$ qui va dépendre à la fois des actions choisies à chaque état, mais également de la distribution des états sous cette politique. Le théorème du gradient de la politique [164, 121] permet d'obtenir une expression du gradient qui n'implique pas la dérivée de la distribution des états. On se place ici dans le cas où $\gamma = 1$ (le paramètre de décroissances exponentielle dans le calcul du *gain* eq. 1.18), nous reviendrons au cas général par la suite.

$$\nabla J(\boldsymbol{\theta}) \propto \sum_{s \in S} \mu_{\pi_{\boldsymbol{\theta}}}(s) \sum_{a \in A} q_{\pi_{\boldsymbol{\theta}}}(s,a) \nabla \pi_{\boldsymbol{\theta}}(a \mid s) \tag{1.43}$$

où $\propto$ veut dire proportionnel, et $\mu_{\pi_{\boldsymbol{\theta}}}$ est la distribution des états sous la politique $\pi_{\boldsymbol{\theta}}$.

Le principe de l'algorithme $REINFORCE$ [178] est d'effectuer une descente de gradient stochastique, c'est à dire en utilisant une approximation du gradient réel, s'appuyant sur des tirages de Monte Carlo.

On remarque que le premier terme du membre de droite de l'équation 1.43, représentant la probabilité d'être dans un état donné, pondère en fait le reste de l'expression. De plus, par un jeu de réécriture (équation 1.45), on peut faire apparaître une pondération du reste du membre de droite par $\pi_{\boldsymbol{\theta}}(a \mid s)$, la probabilité de choisir l'action a dans l'état s. On peut alors réécrire l'expression entièrement sous la forme d'une espérance (équation 1.46) en remplaçant la somme sur les états, et la somme sur les action par des échantillons s provenant de $\mu_{\pi_{\boldsymbol{\theta}}}$ et a provenant de $\pi_{\boldsymbol{\theta}}$.

$$\nabla J(\boldsymbol{\theta}) \propto \sum_{s \in S} \mu_{\pi_{\boldsymbol{\theta}}}(s) \sum_{a \in A} q_{\pi_{\boldsymbol{\theta}}}(s,a) \nabla \pi_{\boldsymbol{\theta}}(a \mid s) \tag{1.44}$$

$$= \sum_{s \in S} \mu_{\pi_{\boldsymbol{\theta}}}(s) \sum_{a \in A} \pi_{\boldsymbol{\theta}}(a \mid s) q_{\pi_{\boldsymbol{\theta}}}(s,a) \frac{\nabla \pi_{\boldsymbol{\theta}}(a \mid s)}{\pi_{\boldsymbol{\theta}}(a \mid s)} \tag{1.45}$$

$$= \mathbb{E}_{s \sim \mu_{\pi_{\boldsymbol{\theta}}}, a \sim \pi_{\boldsymbol{\theta}}} \left[q_{\pi_{\boldsymbol{\theta}}}(s,a) \frac{\nabla \pi_{\boldsymbol{\theta}}(a \mid s)}{\pi_{\boldsymbol{\theta}}(a \mid s)} \right] \tag{1.46}$$

$$= \mathbb{E}_{s \sim \mu_{\pi_{\boldsymbol{\theta}}}, a \sim \pi_{\boldsymbol{\theta}}} \left[q_{\pi_{\boldsymbol{\theta}}}(s,a) \nabla \ln \pi_{\boldsymbol{\theta}}(a \mid s) \right] \tag{1.47}$$

En générant des épisodes en suivant la politique $\pi_{\boldsymbol{\theta}}$ on obtient des échantillons qui peuvent être alors utilisés pour estimer $\nabla J(\boldsymbol{\theta})$, et pour chaque étape t de l'épisode, on peut utiliser $G(t)$ comme estimation de $q_{\pi_{\boldsymbol{\theta}}}(s_t, a_t)$ (eq. 1.20). La méthode REINFORCE consiste à utiliser des tirages aléatoires en suivant la politique courante pour approximer le gradient, et à mettre à jour le paramètre $\boldsymbol{\theta}$ (et donc la politique). Dans le cas où $\gamma < 1$, alors un facteur γ^t apparaît devant $G(t)$ (voir algorithme 5), ce facteur est régulièrement omis dans les variantes modernes de l'algorithme, ce qui conduit à une estimation biaisée du véritable gradient de $J(\boldsymbol{\theta})$ [132].

L'algorithme 5 présente la procédure complète d'apprentissage effectuée par la méthode REINFORCE.

Cet algorithme, bien que convergeant, souffre de la grande variance de l'estima-

Algorithme 5 : REINFORCE

Données : Une politique $\pi_{\boldsymbol{\theta}}$ différentiable, paramétré par $\boldsymbol{\theta}$, $\alpha \in]0,1]$: pas d'apprentissage

1 $\forall w \in \boldsymbol{\theta} : w \leftarrow 0$
2 **répéter**
3 Générer un épisode $s_0, a_0, r_1, ...s_{T-1}, a_{T-1}, r_T$ en suivant $\pi_{\boldsymbol{\theta}}$
4 **pour** $t = 0, 1..., T-1$ **faire**
5 $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \alpha * \gamma^t G(t) \nabla \ln \pi_{\boldsymbol{\theta}}(a_t \mid s_t)$
6 **jusqu'à** *convergence*;

tion du gradient et la convergence peut être longue. Une variation notable consiste alors à soustraire une valeur de référence au *gain* $G(t)$ dans la règle de mise à jour de l'algorithme. Cette valeur de référence peut être par exemple une estimation de la valeur de l'état $\hat{v}_\pi(S_t, \boldsymbol{w})$ paramétrée par un second vecteur de paramètres $\boldsymbol{w}$, comme décrit dans la section 1.2.3 que l'on apprend en même temps que $\boldsymbol{\theta}$. Plusieurs variantes modernes d'algorithmes de politique du gradient ont été proposées [127, 156, 155].

1.2.5 Apprentissage par renforcement pour l'optimisation combinatoire

L'apprentissage par renforcement pour la résolution de problèmes d'optimisation combinatoire a connu un essor ces dernières années, que ce soit pour apprendre directement à résoudre un problème [15, 51] ou bien pour apprendre à déterminer une heuristique de choix de valeurs [35, 30]. Nous renvoyons vers la section 1.4 qui présente un panel de méthodes utilisant l'apprentissage, dont l'apprentissage par renforcement, pour résoudre des problèmes combinatoires.

1.3 Recherche arborescente de Monte Carlo

La recherche arborescente de Monte Carlo (MCTS pour *Monte Carlo Tree Search*) est une recherche arborescente dans laquelle la recherche est guidée par des simulations de Monte Carlo. La sélection d'un noeud de l'arbre de recherche est vue comme un problème de bandit manchot [167]. Ceci permet d'orienter l'expansion de l'arbre en assurant un compromis entre exploitation des informations issues des précédentes simulations, et exploration de nouvelles parties de l'arbre. La méthode MCTS a d'abord été proposée dans le contexte des jeux [97], et plus particulièrement pour le jeu de Go [46, 36], avant d'être utilisée dans le contexte plus général de l'optimisation combinatoire [131, 58, 120, 152].

1.3.1 Principes de la méthode MCTS

La méthode MCTS développe un arbre de recherche de manière guidée et peut être vue comme une sorte de recherche *Best First*. Dans cet arbre, un noeud représente un état du problème. À chaque noeud interne de l'arbre est associé un ensemble d'actions, chacune menant à un noeud enfant. À l'origine, dans le contexte des jeux, le but de la méthode est de trouver le prochain coup à jouer, c'est à dire celui qui a le plus de chance de mener à une victoire. Ce coup est ensuite joué, puis la méthode reprend avec le sous arbre déjà développé qui correspond à ce coup (suivi de celui de l'adversaire dans le cas de jeux à deux joueurs). Pour l'optimisation combinatoire, on peut élargir ce but en considérant que la méthode vise à trouver un chemin de la racine vers un noeud représentant une solution du problème, afin de maximiser une fonction objectif. L'originalité de la méthode MCTS se base sur deux principes :

— à chaque itération, la méthode exécute une simulation à partir d'un noeud ouvert. A partir de l'état du problème associé à un noeud ouvert, une simulation consiste en un tirage de décisions aléatoires jusqu'à atteindre un état final. Ce mécanisme permet d'avoir une estimation du potentiel d'un noeud ouvert.

— pour ajouter de nouveaux noeuds dans l'arbre, et sélectionner les noeuds ouverts d'où partent les simulations, la méthode se base sur un principe de bandit manchot [167]. Ce principe assure un compromis entre l'exploitation des branches qui ont conduit à de bons résultats lors des simulations des itérations précédentes, et l'exploration de nouvelles parties de l'arbre non encore découvertes.

Une itération de la méthode MCTS est composée de quatre phases :

— La phase de sélection qui, à partir de la racine parcourt l'arbre jusqu'à arriver à un noeud ouvert.

— La phase d'expansion qui permet l'ajout de nouveaux noeuds enfants de celui sélectionné dans la première phase.

— La phase de simulation qui effectue une simulation à partir du noeud sélectionné ou d'un de ses enfants nouvellement ajouté.

— La phase de rétropropagation qui propage le résultat de cette simulation à tous les noeuds traversés dans les phases précédentes de cette itération, du noeud à partir duquel est partie la simulation jusqu'à la racine.

En s'appuyant sur les expériences passées pour guider l'expansion de l'arbre, l'algorithme pratique donc une forme d'apprentissage par renforcement, en adaptant sa stratégie de recherche au problème considéré.

1.3.2 Description de l'algorithme

La recherche arborescente de Monte Carlo est, comme son nom l'indique, un algorithme de recherche arborescente. Elle développe un arbre, dans lequel chaque noeud v correspond à un état du problème, et chaque arc e correspond à une

décision. La racine de l'arbre est alors l'état du problème vierge de toute décision. On note $A(v)$ l'ensemble des arcs sortant de v, et donc des décisions disponibles au noeud v. On note $v' = v|e$ le noeud obtenu par l'emprunt de l'arc e à partir du noeud v, l'arc reliant deux noeuds est noté $a(v, v')$. Puisqu'on est dans un arbre, chaque noeud v a un unique parent, que l'on note $p(v)$.

La méthode garde en mémoire l'arbre déjà exploré $\mathcal{T}$, de plus pour chacun des arcs issus d'un noeud v, la méthode mémorise un triplet $\langle N(v,e), Pr(v,e), Q(v,e)\rangle$, avec :

— $N(v, e)$ est le nombre de fois où l'arc e a été emprunté. Par soucis de simplification de notation, on mémorise également le nombre de visites d'un noeud, soient le noeud v et un arc $e \in A(v)$, on a $N(v|e) = N(v, e)$.

— $Pr(v, e)$ est une probabilité à priori, ou une préférence pour emprunter l'arc e lorsque l'on est au noeud v. Cette valeur n'est pas présente dans les versions initiales de la méthode MCTS et a été introduite par [157].

— $Q(v, e)$ est l'estimation de la valeur de l'arc. Cette valeur est calculée en moyennant les résultats de toutes les simulations lancées à partir d'un noeud du sous-arbre enraciné au noeud v.

Dans la méthode MCTS, il y a deux stratégies pour parcourir l'espace de recherche : une stratégie de parcours de l'arbre $\mathcal{T}$, et une stratégie de simulation pour atteindre une solution du problème.

— La stratégie de parcours de l'arbre est utilisée lors de la phase de sélection pour naviguer dans l'arbre en assurant un compromis entre exploitation et exploration. C'est cette stratégie qui évolue en fonction des itérations passées pour diriger la méthode afin de découvrir les zones de l'espace de recherche les plus prometteuses.

— La stratégie de simulation est utilisée lors de la phase de simulation pour aboutir à une solution de manière gloutonne. Originellement, les simulations utilisaient un tirage aléatoire, mais on peut utiliser n'importe quel algorithme glouton incluant une dose d'aléatoire. La stratégie de simulation est statique, dans le sens où elle n'évoluera pas durant l'exécution de la méthode.

La stratégie de parcours de l'arbre est celle qui est utilisée pour naviguer parmi les noeuds stockés dans l'arbre $\mathcal{T}$, elle dispose de la mémoire des itérations précédentes. La stratégie de simulation permet quand à elle de naviguer dans le reste de l'espace de recherche qui est en dehors de $\mathcal{T}$.

L'algorithme MCTS itère à travers quatre phases jusqu'à ce qu'un critère d'arrêt soit rencontré.

Sélection La phase de sélection démarre au noeud racine de $\mathcal{T}$ et finit lorsqu'on rencontre un noeud qui n'a pas encore été exploré. À chaque noeud $v \in \mathcal{T}$, l'arc sélectionné e^* est celui qui maximise une fonction exprimant le compromis entre exploitation et exploration. La fonction généralement utilisée est la limite supérieure de confiance (*Upper Comfidence Bound*) [10] ou une de ses dérivées comme

celle présentée ci-dessous. Le calcul de ce compromis, présenté dans l'équation 1.48, s'appuie sur les informations stockées à chaque noeud pour sélectionner le prochain arc e^*. Le prochain noeud est ainsi $v|e^*$. Le processus se poursuit jusqu'à aboutir à un noeud v n'ayant jamais été visité, c'est à dire tel que $N(v) = 0$.

$$e^* = \arg \max_{e \in A(v)} Q(v,e) + c * Pr(v,e) \frac{\sqrt{N(v)}}{N(v,e)+1} \tag{1.48}$$

Dans l'équation 1.48, le premier terme $Q(v,e)$ est appelé le terme d'exploitation, et le second terme $Pr(v,e)\frac{\sqrt{N(v)}}{N(v,e)+1}$ est appelé le terme d'exploration. Le compromis entre ces deux termes est assuré par le paramètre c.

Le principe de cette équation est de sélectionner l'arc qui maximise la valeur $Q(v,e)$ à laquelle est ajouté un bonus qui décroît à chaque visite pour promouvoir l'exploration. La probabilité à priori $Pr(v,e)$ permet de biaiser l'exploration avec des connaissances sur l'état du problème. Ces connaissances peuvent être heuristiques et basées sur des connaissances expertes du problème. On retrouve généralement ce second terme sous la forme $\sqrt{\frac{\ln N(v)}{N(v,e)+1}}$ comme à l'origine dans [97]. La forme retenue ici provient de [157], et à l'avantage de pouvoir injecter des connaissances sur le problème lors de la sélection sans changer le principe de compromis entre exploitation et exploration de l'équation originelle.

Lors de cette phase de sélection, on retrouve parfois une stratégie d'initialisation remplaçant l'équation 1.48 lors des premières premières visites d'un noeud, consistant par exemple à sélectionner aléatoirement parmi les arcs non sélectionnés tant que tous n'ont pas été sélectionnés au moins une fois [62], ou encore en alternant entre noeud non visité et noeud non visité aléatoirement [87]

Pour le jeu de Go, la valeur du premier terme $Q(v,e)$ est comprise entre -1 et 1. Si la valeur est proche de 1 (resp. -1), alors l'action correspondante mène très probablement à une victoire (resp. defaite), et cette valeur est estimée par les simulations des itérations précédentes. Le réglage du paramètre c peut donc être effectué une unique fois pour tous les appels à l'algorithme. Cependant, dans le contexte de l'optimisation combinatoire, le résultat d'une simulation est généralement liée à la valeur de la fonction objectif, et donc la valeur de $Q(v,e)$ peut fortement varier selon les instances. Cette variabilité des instances peut alors rendre difficile le réglage du paramètre de compromis. Plusieurs solutions sont proposées dans la littérature autour de la normalisation de la valeur de la fonction objectif. Cette normalisation peut se faire soit dans la phase de sélection, soit au niveau de la sélection [87], soit en normalisant le résultat de la phase de simulation directement [131, 120, 153].

Expansion Soit v le noeud obtenu lors de la phase de sélection. La phase d'expansion sert à ajouter à l'arbre $\mathcal{T}$ un ou plusieurs noeuds enfants du noeud v. Pour

chacun des arcs e ajoutés, on initialise le triplet de données : $N(v, e)$ et $Q(v, e)$ sont initialisés à 0 et la probabilité à priori $Pr(v, e)$ est initialisée selon une méthode ad-hoc. Si aucune distribution n'est disponible, on peut utiliser la distribution uniforme $1/|A(s)|$, ou bien ignorer ce mécanisme en mettant cette valeur à 1 pour chaque noeud.

Plusieurs variantes existent pour cette phase d'expansion. Notamment, elle peut être combinée soit avec la phase de sélection, soit avec la phase de simulation. Par exemple, le noeud ajouté peut être le premier noeud traversé par la phase de simulation [46].

Simulation Dans la phase de simulation, le noeud obtenu par la phase de sélection, ou l'un de ses fils nouvellement ajouté (selon les versions) est étendu jusqu'à un état final du problème en appliquant la stratégie de simulation. Dans les premières variantes de la méthode MCTS, ces simulations sont réalisées à partir d'un échantillonnage aléatoire des décisions possibles à partir du noeud.

Dans le cadre de l'optimisation combinatoire, n'importe quelle procédure gloutonne stochastique adaptée au problème considéré peut être utilisée. Dans cette phase de simulation, on retrouve des hybridations de l'algorithme avec d'autres méthodes comme l'utilisation de la relaxation linéaire continue en guise de simulation [153], de la recherche locale [65], de la propagation unitaire dans le contexte de la satisfaisabilité booléenne [141], ou encore d'un solveur de programmation par contraintes [114].

Rétropropagation La phase de rétropopagation consiste à mettre à jour les arcs parcourus lors de la phase de sélection avec les résultats issus de la phase de simulation. A l'issue de la phase de simulation, la solution obtenue est évaluée. Dans le contexte des jeux, l'évaluation consiste à retourner 1 pour une victoire et 0 pour une défaite. Pour l'optimisation combinatoire on utilise généralement la valeur de la fonction objectif comme évaluation d'une simulation [152, 87, 153]. D'autre méthodes ont été considérées, comme retourner directement un nombre entre 0 et 1 en fonction de la qualité de la solution par rapport à la meilleure solution courante [120, 131], ou bien se baser sur la profondeur du conflit rencontré dans le cadre de l'utilisation d'un solveur exact pour la phase de simulation [114].

Soit l'évaluation ϕ d'une simulation, la phase de rétropropagation consiste à mettre à jour les caractéristiques de tous les arcs empruntés jusqu'à la phase de simulation lors de cette itération. Ainsi, pour chacun de ces arcs e empruntés à partir du noeud v on a :

$$
\begin{aligned}
Q(v, e) &\leftarrow Q(v, e) + \frac{\phi - Q(v, e)}{N(v, e) + 1} \\
N(v, e) &\leftarrow N(v, e) + 1
\end{aligned}
$$

Ainsi, pour chacun des arcs sortants e d'un noeud v de l'arbre, $Q(v, e)$ est la moyenne des évaluations des $N(v, e)$ épisodes qui sont passés par cet arc lors de la

phase de sélection.

On peut se poser la question de la pertinence de la moyenne pour stocker le résultat des simulations, notamment pour le cas de l'optimisation combinatoire. Il peut être pertinent de ne stocker que la valeur maximale obtenue dans le sous-arbre, plutôt qu'une moyenne [144, 180]. Il semble néanmoins intéressant de noter que l'utilisation d'heuristiques dédiées pendant la phase de simulation réalise déjà une forme de sélection de solutions élites dont on fait ensuite la moyenne.

Apprentissage par renforcement La recherche arborescente de Monte Carlo est un algorithme d'apprentissage par renforcement, qui estime les valeurs des actions qui sont stockées dans l'arbre $\mathcal{T}$. Chaque noeud v correspond à un état s, et la valeur stockée dans chaque arc $Q(v,e)$ est en fait une estimation de $q_\pi(s,a)$ où s est l'état correspondant au noeud v et a est l'action correspondante à l'arc e, sous la politique formée par la conjonction de la stratégie de parcours de l'arbre et la stratégie de simulation. Dès lors, on peut utiliser la notion de récompense retournée par l'environnement à chaque action plutôt qu'une récompense unique de l'état final. Chaque épisode démarre à la racine de l'arbre et se termine par l'état final atteint par la simulation. Pour la phase de rétropopagation, pour chaque arc e traversé par la phase de sélection à partir du noeud v à l'étape t, on utilise alors le *gain* $G(t)$ de l'épisode entier :

$$\begin{aligned} Q(v,e) &\leftarrow Q(v,e) + \frac{G(t) - Q(v,e)}{N(v,e)+1} \\ N(v,e) &\leftarrow N(v,e) + 1 \end{aligned}$$

1.3.3 Algorithme de MCTS pour l'optimisation combinatoire

La recherche arborescente de Monte Carlo, principalement connue pour son exploitation dans les jeux, a plusieurs fois été appliquée dans le contexte de l'optimisation combinatoire. On retrouve une grande variété de problèmes, que ce soit d'ordonnancement [152, 114, 87, 124, 103, 111], de transport [131, 120], de *Bin Packing* [106, 108], des problèmes dynamiques de logistique [140], de stockage [148] ou d'affectation de ressources [23], le problème de satisfaisabilité booléenne [65, 141, 91], ou bien des problèmes multi-objectifs [175, 130]. L'algorithme de MCTS a également été employé pour trouver une *backdoor* dans des modèles de PLNE (un sous-ensemble de variables d'un problème dont on peut prouver l'optimalité en branchant uniquement dans celui-ci) [94]. Après simulation, les auteurs utilisent un solveur de PLNE pour évaluer la solution, c'est à dire si l'ensemble de variables obtenu est proche ou non d'être une *backdoor*. Suite au succès de l'algorithme AlphaZero [158], plusieurs travaux se sont concentrés sur son implémentation directe afin d'apprendre un réseau de neurones par des recherches arborescentes de Monte Carlo successives dans le cadre de l'optimisation combinatoire [1, 81, 182, 106]. Proche de cela, Rimélé et al. [148] propose l'utilisation de la méthode MCTS pour générer un ensemble de données dans le but d'apprendre une politique pour un

problème d'allocation en temps réel.

La plupart de ces travaux intégrant la méthode MCTS diffèrent seulement de quelques variations : par exemple certaines utilisent l'équation 1.48 pour la phase de sélection [103, 180], d'autres utilisent la forme plus classique sans probabilité à priori [108, 114] ; d'autres encore une variante conçue pour le jeu à un joueur [124] ; un terme incluant la variance [94] ; ou même une sélection ε-*greedy* à la place [152, 114]. D'autres variations incluent la normalisation de $Q(v, e)$. Comme évoqué précédemment, dans les jeux cette valeur est comprise entre 0 et 1 et indique la probabilité de victoire estimée par les simulations des itérations précédentes. Une approche consiste à attribuer une victoire (1) à une simulation qui améliore la solution courante, et sinon une valeur entre 0 et 1 dépendante d'un paramètre de seuil [120, 131]. Abe et al. [1] proposent de normaliser $Q(v, e)$ en utilisant la moyenne et l'écart type de la somme des récompenses obtenues par des simulations aléatoires exécutées à chaque ouvertures de noeuds. Une autre proposition consiste à utiliser l'inverse de $Q(v, e)$ et à diviser également le paramètre c par la valeur du noeud père [148]. Plus classiquement, une proposition simple est de normaliser en divisant par la différence entre la plus grande valeur et la plus petite valeur de $Q(v, e)$ parmi les enfants de v après avoir soustrait cette même plus petite valeur [180].

Certains travaux proposent des mécanisme spécifiques, ou hybrident la méthode MCTS avec d'autres approches. Sabharwal et al. [153] proposent d'utiliser une recherche arborescente pour guider un solveur PLNE pour un problème de maximisation. Dans leur méthode, une simulation consiste à résoudre une relaxation linéaire. Ils utilisent alors la valeur de la relaxation linéaire comme récompense de la simulation, et normalisent $Q(v, e)$ par la valeur de la relaxation linéaire obtenue à la racine de l'arbre. Enfin, lors de la phase de rétropropagation, ils mettent à jour les arcs tels que chaque arc stocke la plus grande valeur de relaxation linéaire. D'une manière similaire, Loth et al. [114] ont proposé d'hybrider la recherche arborescente de Monte Carlo avec un solveur de PPC. Chaque simulation est une recherche en profondeur d'abord opérée par le solveur de PPC qui s'arrête au premier conflit, mais la dernière branche explorée par la simulation est gardée en mémoire. Ainsi une prochaine simulation n'explorera pas ce qui a déjà été exploré par une simulation passée. Chaque arc est associé à un littéral, qui correspond à une affectation, ou à une réfutation, et les statistiques sont stockées non pas pour un arc mais pour chaque littéral, partageant alors les informations pour une affectation ou une réfutation indépendemment de sa position dans l'arbre. Enfin, lors de la phase de rétropropagation, la récompense est calculée pour chaque littéral traversé lors de la phase de sélection en se basant sur la profondeur relative atteinte lors du conflit de la phase de simulation. Plus précisément, à chaque arc de l'arbre $\mathcal{T}$, est stocké le nombre d'étapes moyen entre l'arc et les conflits obtenus lors des simulations précédentes. Pour évaluer une simulation, on regarde pour chaque arc traversé pendant la phase de sélection, si le nombre d'étapes entre le conflit et cet arc est plus important que le nombre moyen d'étapes. Si c'est le cas, alors une récompense de 1 est attribué au littéral correspondant, 0 sinon. L'idée avancée par les auteurs est que pour un ordre de variables fixé par le solveur, les branches longues sont les plus à

même d'indiquer une bonne direction pour trouver une solution. La méthode utilise également une politique de redémarrage, pour pouvoir profiter d'un ordre de sélection de variables dynamique tel que *Weigthed Degree*, supprimant ainsi les valeurs stockées pour chaque littéral. Cette évaluation de chacun des littéraux plutôt que des arcs est inspirée de l'heuristique RAVE (Rapid Action Value Estimation) [62], et permet alors les redémarrages en ayant la capacité d'estimer rapidement les valeurs des décisions. Pour la satisfaisabilité booléenne, Goffinet and Ramanujan [65] propose d'intégrer à la méthode MCTS une recherche locale pendant les simulations, pour résoudre un problème de MaxSAT. La phase de simulation commence par une instanciation de toutes les variables, puis exécute une recherche locale en interdisant de modifier les variables affectées pendant la phase de sélection. Pour le problème SAT, Keszocze et al. [91] propose une recherche arborescente de Monte Carlo intégrant l'apprentissage de clauses lors des conflits (*Conflict-Driven Clause Learning*), allant plus loin que la simple intégration de la propagation unitaire qui avait déjà été proposée [141]. Les nouvelles clauses apprises lors de conflits peuvent alors amener à supprimer des noeuds dans l'arbre lors de la phase de sélection. Les auteurs proposent de mettre à jour les statistiques de tous les arcs menant à ce noeud en supprimant tout ce qui a trait au sous-arbre enraciné en ce noeud. Les simulations se terminent dès qu'un conflit est rencontré, et l'évaluation d'une simulation consiste à la proportion de clauses satisfaites lors du conflit.

1.4 Apprentissage pour l'optimisation combinatoire

Dans cette section, nous donnons un aperçu de la littérature sur l'apport de méthodes d'apprentissage automatique pour la résolution de problèmes d'optimisation combinatoire. Si l'utilisation de l'apprentissage automatique au service de l'optimisation combinatoire n'est pas nouvelle [78], cette idée a connu un essor important ces dernières années au vu des très nombreux *surveys* récents que l'on peut trouver sur cette thématique [17, 112, 119, 100, 171, 125, 174, 29, 165, 184]. Depuis 2021, il existe également une compétition intitulée ML4CO (*the Machine Learning for Combinatorial Optimization Competition*) [60].

Il existe plusieurs caractéristiques permettant de classifier les différentes approches basées sur l'apprentissage automatique.

1. Apprend-on de manière hors ligne, c'est à dire avant la résolution, ou en-ligne, c'est à dire pendant la résolution ?
2. Apprend-on à partir de données collectées, par un processus d'apprentissage supervisé, ou apprend-on de l'expérience via l'apprentissage par renforcement ?
3. Apprend-on dans le but d'intégration d'un modèle appris dans un algorithme existant, apprend-on un modèle qui donne des solutions directement, ou apprend-on à sélectionner quelle approche utiliser pour résoudre le problème ?

Dans cette revue de la littérature, on se concentre plutôt sur cette dernière question, en se focalisant sur le but des méthodes proposées plutôt que sur les moyens

(première question), ou sur le moment de l'apprentissage (seconde question). Nous reprenons alors la classification des méthodes proposés par Bengio et al. [17].

1.4.1 Apprendre à résoudre

Les méthodes de cette catégorie ont le point commun d'être entièrement centrées sur leur modèle d'apprentissage automatique qui est entraîné dans le but de générer une solution. Certaines méthodes améliorent ensuite les solutions obtenues via une recherche locale [51, 109], ou utilisent un algorithme de recherche en faisceau (*Beam Search*) pour sélectionner la solution[129, 88], mais le coeur de ces méthodes est bien le modèle d'apprentissage. Dans la littérature de telles méthodes sont appelées des méthodes d'apprentissage *End-to-End* [17, 100]. Par ailleurs, dans cette catégorie *Apprendre à résoudre*, on trouve de nombreux travaux autour du problème de voyageur de commerce (*Travelling Salesman Problem*, TSP).

Vinyals et al. ont introduit une structure de réseau de neurones récurrents, appelée *pointer network* [172], permettant de retourner en sortie une permutation à partir d'une séquence passée en entrée. Ce modèle est entraîné de manière supervisée, en calculant des solutions optimales pour, entre autre le problème de TSP. Dans la continuité directe de ces travaux, plusieurs approches d'apprentissage par renforcement via une méthode de politique du gradient ont été proposées [15, 79]. Dans ces articles, les méthodes permettent d'échantillonner un grand nombre de solutions pour retourner la meilleure solution. Une telle approche d'échantillonage permet alors de continuer l'entraînement sur une instance spécifique grâce aux solutions générées au prix d'un temps de calcul supplémentaire.

Des méthodes similaires utilisant d'autres architectures de réseaux de neurones encodant la connectivité entre sommets du graphe de distance, ont été entraînées par des algorithmes de politique du gradient pour le TSP [51, 98]. Ces travaux améliorent les résultats obtenus par les premières méthodes pour le TSP et ont également été étendues au problème de tournées de véhicules [129, 98, 138]. D'autres approches ont été proposées en apprentissage supervisé, sur des structures de réseaux de neurones en graphe, dans lesquels l'objectif est de directement prédire les arcs de la solution optimale d'un TSP [88], ou bien les sommets de la solution optimale d'un problème de stable maximum (*Maximum Independent Set*) [109]. Dans ces deux approches, le modèle entraîné par apprentissage supervisé produit une carte de probabilités des arcs/sommets, et une procédure de recherche est utilisée pour reconstruire une solution. Enfin, des structures similaires de réseaux de neurones en graphe ont été entraînées également par apprentissage par renforcement pour le problème du TSP [89], mais également pour le problème de couverture par les sommets et de coupe maximale [92]. Là encore, un algorithme secondaire est utilisé pour former une solution à partir des données en sortie du modèle d'apprentissage, soit une recherche en faisceau [89], soit une procédure gloutonne [92].

Ces approches ne permettent généralement d'obtenir des solutions proches de l'optimum que sur des instances de l'ordre d'une centaine de sommets pour le TSP, sans arriver à généraliser sur des instances de taille plus importante. Pour compa-

raison, le solveur spécialisé *Concorde* [8] trouve et prouve la solution optimale d'un TSP à 100 sommets en 3 minutes.

Néanmoins, les approches proposées par [109] et [92] intègrent le modèle appris à chaque étape de décision de l'algorithme utilisé pour construire une solution : une recherche en largeur pour le premier et une procédure gloutonne pour le second. De ce fait, le modèle d'apprentissage utilise alors l'état courant du problème pour aider à guider la prochaine décision, et ces deux approches pourraient alors être classées dans la seconde catégorie.

Une dernière approche qu'on peut classer dans la catégorie *apprendre à résoudre* consiste à apprendre une approximation du problème à résoudre en produisant une variante plus facile de ce problème, variante pour laquelle on dispose d'un algorithme performant [136, 137]. Le principe de cette approche est de proposer un modèle d'apprentissage qui permet de transformer le problème originel en un autre problème proche. Pour assurer l'efficacité de la méthode, la solution optimale du problème approché doit être une bonne solution du problème initial. Cette approche a été appliquée à un problème stochastique d'ordonnancement de route et d'ordonnancement à une machine.

1.4.2 Apprendre à chercher

Nous classons dans cette catégorie tout mécanisme d'apprentissage qui est intégré dans un algorithme classiquement utilisé pour résoudre des problèmes d'optimisation combinatoire.

Dans cette catégorie, on retrouve de nombreux travaux sur l'apprentissage d'heuristiques de branchement dans le contexte de la PLNE [112]. Une heuristique efficace de choix de variables dans une méthode de séparation et d'évaluation (*Branch and Bound*) est le branchement fort(*Strong Branching*) [7] qui permet de diminuer efficacement l'espace de recherche. Cette heuristique s'appuie sur un score pour chaque variable, basé sur la relaxation linéaire de chacune des branches, impliquant un temps de calcul important [2]. Plusieurs travaux en apprentissage automatique essaient dès lors d'approximer cette heuristique de branchement que ce soit en apprenant le classement des variables [95, 70, 183], ou bien en estimant directement le score de chaque variable [4, 3], ou bien encore en prédisant la prochaine variable à sélectionner [59, 68, 161, 128]. La plupart des modèles sont entraînés via le paradigme de l'apprentissage par imitation [84], qui reprend le cadre de l'apprentissage par renforcement mais dans lequel on dispose d'une politique experte qui peut alors générer des exemples sur lesquels apprendre (ici un solveur utilisant le branchement fort). Il est à préciser que pour certaines approches l'apprentissage à lieu en ligne [95, 3], mais que les bons résultats de ces approches sont obtenus hors ligne avec des modèles de réseaux de neurones en graphe [59, 128]. Malheureusement, lorsque ces modèles sont intégrés dans un solveur sans pouvoir bénéficier de l'accélération des temps de calcul permis par l'utilisation d'une carte graphique dédiée, le temps de calcul augmente de manière importante [68]. Une approche possible est alors hybride, elle considère un tel modèle basé sur un réseau de neurones en

graphe à la racine de l'arbre, et un modèle plus léger s'appuyant sur les calculs du premier pour le reste de l'arbre [68]. Enfin, ces approches, bien que génériques et généralisant bien sur la taille des problèmes doivent être entraînés sur le même type de problème que ceux sur lesquels ils vont être utilisé pour obtenir des résultats satisfaisant. Autrement dit, ils nécessitent un modèle pour chaque type de problème. Zarpellon et al. ont proposé une approche visant à généraliser plus largement, pour avoir un unique modèle applicable à plusieurs problèmes en intégrant des caractéristiques des arbres de recherche dans le modèle d'apprentissage [185].

Non loin de cette problématique, Cappart et al. propose l'utilisation de l'apprentissage par renforcement pour apprendre une heuristique d'ordre des variables dans des diagrammes de décisions [19]. Le but est d'obtenir une borne inférieure et une borne supérieure pour un problème de stable maximal et de coupe maximale [28]. L'approche a été intégrée ensuite dans une méthode de séparation et d'évaluation et obtient de bons résultats [31, 135]. Une économie substantielle de temps calcul est réalisé en utilisant le modèle d'apprentissage pour déterminer l'ordre des variables du diagramme de décision seulement pour les 50 premiers noeuds et seulement pour le calcul de la borne supérieure. D'autre travaux autour du calcul de bornes ont été proposés, que ce soit l'estimation de bornes inférieure pour un problème de minimisation dans des méthodes de recherche arborescentes non exact [74, 80], ou la prédiction d'amélioration de la solution en exécutant un algorithme glouton à partir d'un noeud donné de l'arbre de recherche [93]. Enfin, il est également possible d'apprendre à sélectionner les coupes dans la méthode des plans sécants ou dans un *Branch and Cut* [166, 82].

L'autre versant des heuristiques de branchement dans une recherche arborescente concerne le choix de la branche à explorer. Cela consiste soit à choisir le prochain noeud à étendre dans le cas d'une recherche de type *Best First*, soit à choisir l'ordre des branches à visiter dans le cas d'une recherche en profondeur d'abord. Plusieurs approches ont été proposées pour la sélection des noeuds dans le cas d'une recherche arborescente [74, 80], ou dans le cas d'une recherche en faisceau, dans lequel le modèle d'apprentissage calcule le coût estimé d'un noeud jusqu'à la solution [83]. En ce qui concerne la sélection de la branche dans une recherche en profondeur Cappart et al. ont proposé une formalisme unifiant l'apprentissage par renforcement et la programmation par contraintes par un modèle de programmation dynamique [30]. Le modèle d'apprentissage sert dans ce contexte comme heuristique de choix de valeurs à chaque noeud de l'arbre. Cette approche a notamment été étendue en proposant un solveur de programmation par contraintes intégrant l'apprentissage par renforcement [35].

La plupart de ces travaux présentent l'avantage d'être intégrés au sein d'un solveur générique de PLNE ou de PPC préexistant. Néanmoins, l'apprentissage, qu'il soit supervisé ou par renforcement nécessite généralement d'être spécifique au problème étudié.

Les méthodes de recherche locale et les métaheuristiques ont aussi été étudiées sous l'angle de l'apprentissage automatique [119]. Depuis longtemps, des méca-

nismes d'adaptation ont été proposés dans les métaheuristiques, que ce soit pour la sélection du voisinage dans une recherche à voisinage large [149], ou pour le réglage du degré d'aléatoire dans la construction de solutions initiales [139].

Les méthodes d'apprentissage peuvent être intégrées dans une métaheuristique de plusieurs manières [119]. La prédiction de la qualité des mouvements, ou l'évaluation des solutions peut être confiée à un modèle d'apprentissage. Par exemple Lucas [117] propose deux métriques pour guider une recherche locale. La première est la valeur de la fonction objectif. Lorsqu'un minimum local est atteint, une seconde métrique est utilisée, la distance à une zone prometteuse dans l'espace des caractéristiques décrivant une solution. Ces zones ont été définies par apprentissage supervisé sur des données comprenant des solutions de bonnes et de mauvaises qualités. Ensuite, un modèle d'apprentissage peut permettre d'obtenir les solutions initiales nécessaires à certaines méthodes, que ce soit via un mécanisme d'apprentissage par renforcement en ligne permettant de s'adapter à l'instance considérée [187], ou par toute méthode qui permet la génération de solutions [128]. Un modèle d'apprentissage peut aussi être utilisé pour sélectionner les voisinages à utiliser lorsque plusieurs sont disponibles [38, 115], pouvant relever d'une certaine forme d'adaptation que nous abordons dans le prochain paragraphe. La collecte et la fouille des données est également une discipline de l'apprentissage automatique, et, pour un problème de tournées de véhicules, des travaux [9] ont proposé d'extraire les séquences de noeuds retrouvées fréquemment dans des minima locaux. Ces séquences les plus fréquentes sont alors insérées dans les minima locaux suivants pour tenter de trouver de nouvelles solutions. Par ailleurs, le contrôle dynamique des paramètres est une tâche que l'on peut confier à un modèle d'apprentissage [37, 133].

Enfin, le dernier type d'approche entrant dans la catégorie *Apprendre à chercher* est l'adaptation en ligne du comportement de l'algorithme. Ces méthodes se basent le plus souvent sur une approche de type bandit manchot, dans le but de garder un compromis entre exploitation et exploration. Elles sont proches des méthodes présentées dans la section suivante (procédures de sélection d'algorithme et de réglage de paramètres), mais nous les classons dans cette catégorie, car ces mécanismes sont intégrés dans un algorithme global.

Plusieurs travaux ont été proposés sur le choix du niveau de propagation dans un solveur de programmation par contraintes [160]. En effet, en PPC il faut souvent faire un compromis entre le temps passé à chaque noeud de la recherche arborescente, et le filtrage des domaines permettant de réduire l'espace de recherche. La sélection d'heuristiques de branchement a également été étudiée sous cet angle, par exemple par la sélection de l'heuristique dans un portefeuille [176, 179]. En s'appuyant sur une approche de type bandit manchot pour la sélection des noeuds à étendre, la recherche arborescente de Monte Carlo peut être classée dans ce type de méthodes, avec un apprentissage pour la sélection des branches évoqué plus haut. Paparrizou and Wattez [134] proposent d'intégrer, après chaque redémarrage, un choix entre une heuristique de référence adaptative (telle que *weighted degree*) et une sélection aléatoire de l'ordre des variables, suggérant alors un apprentissage

différent de l'heuristique de référence. Dans un contexte différent, Liberto et al. [110] proposent d'adapter le choix de l'heuristique de branchement d'un solveur PLNE au sous-problème courant. En amont de la résolution, ils exécutent un algorithme de partitionnement des données (*Clustering*) pour regrouper différents types d'instances et de problèmes, et affectent une heuristique à chaque groupe (via une procédure de sélection d'algorithmes). Pendant la résolution, le sous-problème courant est classé dans un groupe et l'heuristique associée est alors utilisée. Enfin, nous avons évoqué dans le paragraphe précédent le principe de la recherche à voisinage large adaptative [149], reposant sur le même type de principe. Néanmoins Turkes et al. [168] ont montré par une méta-analyse, que ce mécanisme d'adaptation dans cette méthode n'apporte pas vraiment d'amélioration par rapport à une sélection aléatoire d'opérateurs de voisinage.

1.4.3 Apprendre à configurer

Dans cette catégorie sont classées toutes les méthodes qui permettent de sélectionner l'algorithme ou les paramètres utilisés pour la résolution. Les méthodes prédisent alors la meilleure configuration à utiliser pour résoudre l'instance du problème considéré, en amont de la résolution. La plupart de ces méthodes relèvent alors du domaine de la configuration d'algorithmes qui est un domaine très étudié [77]. Plus particulièrement, la sélection d'algorithmes est un terrain propice à l'utilisation de modèles d'apprentissage, notamment pour prédire les performances d'un algorithme donné pour une instance donnée [101]. Parmi les méthodes proposées, beaucoup ont connu un certain succès comme SATZilla qui prédit le temps d'exécution de chacun des solveurs d'un portefeuille pour sélectionner le meilleur en vue de résoudre un problème de satisfaisabilité booléenne [181]. De façon similaire, Kruber et al. [102] proposent d'utiliser un modèle de classification prédisant si une décomposition de Dantzig-Wolfe donnée pour un problème de PLNE va être plus rapide que la formulation originelle, et donc in fine de choisir quel solveur appeler.

Chapitre 2

Proposition d'un cadre générique pour la résolution de problèmes combinatoires

Sommaire

2.1 Apprentissage par renforcement d'heuristiques 41
2.1.1 Processus de décision markovien pour la résolution de problème combinatoire 42
2.1.2 Politique de décision pour caractériser une heuristique de choix de valeurs 45
2.1.3 Algorithme d'apprentissage d'une heuristique de valeurs . . . 48
2.1.4 Intégration de l'heuristique dans une recherche arborescente . 50
2.2 Recherche arborescente de Monte Carlo 51
2.2.1 Utilisation des bornes 51
2.2.2 Recherche en profondeur 52
2.2.3 Compromis dynamique 53
2.3 Synthèse 54

Dans ce chapitre, nous présentons les deux contributions méthodologiques de la thèse pour résoudre des problèmes d'optimisation combinatoire : un cadre pour la création d'heuristiques de choix de valeurs dans un arbre de recherche que l'on peut régler avec l'apprentissage par renforcement, et des adaptations de la méthode de recherche arborescente de Monte Carlo pour des problèmes d'optimisation combinatoire. Ces deux méthodes peuvent être combinées et leurs applications à deux cas d'étude seront présentées dans les chapitres suivants.

2.1 Apprentissage par renforcement d'heuristiques

Nous proposons d'utiliser l'algorithme REINFORCE pour apprendre une heuristique de choix de valeurs pour des problèmes combinatoires. On se place dans le cadre où l'ordre des variables à instancier est déjà défini (qu'il soit dynamique ou statique), et on veut évaluer les différents choix de valeurs pour ces variables. L'objectif est d'utiliser cette heuristique comme guide au sein de méthodes de recherche arborescente, afin d'identifier les branches à explorer en priorité pour obtenir rapidement de bonnes solutions. Cette heuristique doit également être facile à calculer,

étant donné qu'elle sera appelée à chaque ouverture de noeuds dans l'arbre de recherche.

L'heuristique de choix de valeurs repose sur une fonction d'évaluation qui est appelée pour chacune des décisions possibles. Dans le cadre de cette thèse, on propose d'utiliser comme fonction d'évaluation une combinaison linéaire de critères, qui dépendent du problème considéré. Le paramètre à apprendre est alors le vecteur de poids de cette combinaison linéaire. Cependant, les techniques développées dans ce chapitre se généralisent aisément à tout modèle différentiable, l'algorithme d'apprentissage utilisé étant l'algorithme REINFORCE présenté en section 1.2.4.

L'avantage d'une telle méthode est la possibilité d'injecter des connaissances du problème dans l'heuristique sous la forme de critères pour évaluer une décision. De plus, la simplicité d'un tel modèle permet un apprentissage rapide comparé à des modèles plus complexes. Cette rapidité permet de considérer l'apprentissage comme une phase de pré-traitement de la résolution. En effet, il n'est pas rare que pour un problème, on puisse séparer les instances en plusieurs ensembles aux caractéristiques données. Avec un modèle rapide à entraîner, on peut alors se permettre de spécialiser l'apprentissage pour un ensemble d'instances.

2.1.1 Processus de décision markovien pour la résolution de problème combinatoire

Cette section explicite la correspondance que l'on peut réaliser entre un processus de décision markovien (MDP) et la construction d'une solution d'un problème d'optimisation combinatoire. En effet, un tel processus implique des prises de décision successives, chacune modifiant l'état du problème, assimilé ici à l'environnement. L'état d'un problème permet de connaître quelles décisions futures sont alors possibles, c'est à dire respectent les contraintes. A chacune des décisions, un signal de récompense peut être fourni par l'intermédiaire de l'évaluation de la fonction objectif (ou de bornes sur celle-ci) ou par la violation de contraintes. Ce MDP est déterministe[1], c'est à dire que la fonction de transition τ et de la fonction de récompense R se définissent par : $\tau : S * A \rightarrow S$ et $R : S * A \rightarrow \mathbb{R}$. L'espace d'états de ce MDP correspond alors à l'espace de recherche du problème combinatoire.

Dans cette section, on cherche à établir quelle est la branche d'un arbre de recherche qui est la plus susceptible de nous mener à une bonne solution. Notre approche est alors différente de celles qui considèrent la minimisation de l'arbre de recherche directement comme le but du processus de décision markovien [55, 74, 35]. Un épisode dans ces travaux correspond à l'exploration complète de l'arbre de recherche, quand dans dans notre cas, un épisode correspond à un algorithme glouton et finit à la première solution trouvée.

On s'intéresse alors à des problèmes d'optimisation combinatoire pour lesquels la construction d'une solution réalisable est facile, et peut être obtenue en temps polynomial par un algorithme glouton. En effet, pour appliquer l'algorithme REINFORCE on va devoir générer un grand nombre de solutions réalisables du problème

1. Dans ce manuscrit on s'intéresse à des problèmes déterministes seulement.

considéré. Il est cependant toujours envisageable de travailler sur une version relâchée du problème, en intégrant la violation des contraintes dans la fonction objectif pendant l'étape d'apprentissage. Par exemple, la construction d'une solution d'un problème de voyageur de commerce peut entrer dans ce cadre. L'ordre déterminé des variables peut être l'ordre de visite des sommets, et il est facile de construire une solution admissible en choisissant à chaque étape quel est le prochain sommet à visiter. Si on rajoute des fenêtres de temps à chaque sommet, trouver une solution réalisable devient NP-Complet [154]. On peut alors intégrer la violation des fenêtres de temps avec une forte pénalité dans la fonction objectif.

Soit un problème d'optimisation combinatoire exprimé sous la forme d'un triplet $(\mathcal{X}, \mathcal{D}, \mathcal{C})$, où $\mathcal{X}$ est l'ensemble des variables, $\mathcal{D}$ est le domaine de $\mathcal{X}$ et $\mathcal{C}$ est l'ensemble des contraintes, ainsi que d'une fonction objectif f. On suppose que l'on dispose d'une séquence d'instanciation des variables. On suppose également que l'on peut évaluer la valeur de la fonction objectif pour une instanciation donnée. On note alors $f(I)$ l'évaluation de la valeur de l'objectif pour l'instanciation I. Cette évaluation prend la forme de borne inférieure ou supérieure sur la fonction objectif. On définit le MDP associé comme suit :

- A chaque état $s \in S$ correspond une instanciation $I = ((x_1, v_1), ..., (x_k, v_k))$ sur $Y = (x_1, ..., x_k) \subseteq \mathcal{X}$. Cette instanciation est valide, c'est à dire que, $I[x_i] \in \mathcal{D}(x_i)$, pour tout $x_i \in Y$. Par l'hypothèse que la construction de solutions admissibles est facile, on peut également dire que l'instanciation est globalement cohérente, c'est à dire qu'elle peut être étendue à une solution. Enfin, cet état est un état final si $Y = \mathcal{X}$.
- Soit un état $s \in S$ décrit par $I = ((x_1, v_1), ..., (x_k, v_k))$. On note la prochaine variable à instancier x_{k+1}. L'ensemble des actions $A(s)$ de l'état s est défini par l'ensemble des valeurs $v' \in \mathcal{D}(x_{k+1})$ tel que l'instanciation $((x_1, v_1), ..., (x_{k+1}, v'))$ est globalement cohérente. On parle d'action ou de décision de manière interchangeable.
- Dans l'état $s \in S$ décrit par $I = ((x_1, v_1), ..., (x_k, v_k))$, si on applique l'action a associé à la valeur $v' \in \mathcal{D}(x_{k+1})$, l'état courant noté s' correspond alors à l'instanciation $((x_1, v_1), ..., (x_{k+1}, v'))$.
- Pour la fonction de récompense, nous proposons d'utiliser l'accroissement marginal ou le décroissement marginal de l'évaluation de la fonction objectif entre chaque décision. Ce point est discuté plus en détail dans le prochain paragraphe.

L'utilisation de l'évaluation de la fonction objectif est une proposition ouverte et plusieurs propositions sont possibles, selon que l'on étudie un problème en minimisation ou en maximisation. Dans la suite du manuscrit, nous étudions deux problèmes de minimisation, et utilisons une borne inférieure de la fonction objectif pour attribuer cette récompense. L'accroissement marginal de cette borne inférieure est donc une pénalité que l'on veut minimiser. On remarque que si l'on ajoute la

somme des récompenses d'un épisode à la borne inférieure de l'état initial, alors on obtient la valeur de la fonction objectif pour cet épisode.

En utilisant le facteur de décroissance exponentielle pour le calcul du *gain* (eq. 1.18), c'est à dire si $\gamma < 1$, le *gain* d'un épisode pour un état donné, accorde alors plus d'importance aux pénalités proches de cet état, minimisant alors l'impact des pénalités futures. Cela peut être utile lorsque les épisodes sont longs.

Le choix d'une borne inférieure plus ou moins fine modifiera alors les pénalités accordées. Par exemple, dans le cas d'un problème de voyageur de commerce, la décision de visiter un sommet éloigné du reste du graphe sera pénalisée fortement en cas d'une borne inférieure triviale consistant à sommer les arcs déjà empruntés. L'accroissement du coût de l'arbre couvrant de poids minimum comme pénalité permet de prendre en compte le fait que ce sommet doit être visité dans tous les cas. Le choix de cette borne est d'autant plus important lorsque $\gamma < 1$, pour lequel les décisions coûteuses du point de vue de l'accroissement de cette borne risquent alors d'être repoussées vers la fin.

Dans le cas d'un problème de maximisation, on peut de la même manière utiliser le décroissement d'une borne supérieure comme une pénalité.

Dans la suite de ce chapitre, nous restons dans le cadre de l'apprentissage par renforcement dont le but est de maximiser la somme des récompenses qu'elles soient positives ou négatives (on considère des récompenses négatives pour les pénalités).

Exemple *Pour représenter la résolution du problème de course d'orientation (présenté en 1.1) comme un MDP, il faut d'abord définir un ordre sur les variables à instancier. On rappelle que pour ce problème, on dispose d'un unique ensemble de variables $\mathcal{X} = (x_1, ..., x_n)$ représentant le chemin parcouru. On choisit l'ordre lexicographique des x_i, que l'on va instancier de $i = 1$ à $i = n$. Après k décisions, un état s correspond à l'instanciation des k premières variables $I = ((x_1, v_1), ..., (x_k, v_k))$. L'ensemble $A(s)$ des actions possibles à partir de cet état s correspond aux valeurs admissibles pour la variable x_{k+1} : $\{v \mid v \notin \{v_1, ..., v_k\} \wedge \sum_{i=1}^{k-1} d_{x_i,x_{i+1}} + d_{x_k,v} + d_{v,n} \leq D_{max}\}$. On peut omettre n (noeud arrivée) de $A(s)$ étant donné que si $A(s)$ n'est pas vide, alors visiter directement n est dominé par toute autre décision. Si $A(s)$ est vide, alors le chemin est fini : la variable x_{k+1} reçoit la valeur n , et toutes les variables restantes x_i, avec $k+1 < i \leq n$ sont nécessairement mises à 0. Soient d^- l'arc de plus petite valeur dans le graphe induit par $V' = V \setminus \{v_1, ..., v_{k-1}\}$ et σ la liste des sommets de $V' \setminus \{v_{k-1}\}$ trié par ordre décroissant de profit. Si on note $K = \lceil \frac{D_{max} - \sum_{i=1}^{k-1} d_{v_i,v_{i+1}}}{d^-} \rceil$, alors $U = \sum_{i=1}^{k} p_{v_i} + \sum_{j=1}^{K-1} p_{\sigma_j}$ est une borne supérieure de la fonction objectif pour l'état actuel. K est une borne supérieure du nombre d'arcs que l'on peut prendre dans l'état actuel, on peut alors visiter au plus $K - 1$ sommet, le dernier arc nous permettant de rejoindre l'arrivée. La pénalité attribuée après chaque action est donc le décroissement de cette borne supérieure.*

D'un autre coté, il est possible d'utiliser une borne supérieure (resp. inférieure) de la fonction objectif pour l'instanciation courante dans le cadre d'un problème en minimisation (resp. maximisation), pour obtenir une récompense. Dans ce cas, il est

possible que la borne utilisée n'évolue pas de manière monotone, et les récompenses peuvent donc être positives ou négatives.

Exemple *Pour le problème de course d'orientation on peut calculer une borne inférieure de la fonction objectif très simple sur la valeur de la fonction objectif en calculant* $\sum_{i=1}^{k} p_{v_i}$. *La récompense associée à l'action de sélection du sommet* $v_{k+1} \in A(s)$ *est alors son profit* $p_{v_{k+1}}$.

2.1.2 Politique de décision pour caractériser une heuristique de choix de valeurs

Pour appliquer l'algorithme REINFORCE et entraîner une politique de décision permettant de choisir les actions, on a besoin de définir cette politique.

On suppose qu'il existe une ou plusieurs règles de priorité pour classer les différentes décisions possibles à partir d'un état donné du problème. Une telle règle de priorité repose généralement sur un critère numérique à maximiser ou minimiser.

Ainsi, pour le problème du sac à dos, le choix des objets en suivant l'ordre décroissant du ratio poids/valeur est un exemple d'une telle règle de priorité. Pour le problème de tournées de véhicules, un autre exemple de règle de priorité est le choix des sommets à visiter en suivant l'ordre établi par la règle du plus proche voisin exprimant un critère de distance au dernier sommet.

Pour un problème combinatoire donné, il existe généralement plusieurs règles de priorité heuristiques, en particulier lorsque le problème met en jeu plusieurs contraintes. Plusieurs critères de sélection de la prochaine décision à prendre entrent alors en compétition. Par exemple, pour les problèmes de tournées de véhicules avec capacité limitée et fenêtres de temps, on peut ajouter un critère relatif aux fenêtres de temps, comme le temps restant pour visiter un client avec la date d'échéance, ainsi qu'un critère relatif à la capacité des véhicules, comme le poids des objets à charger afin de donner une priorité aux livraisons encombrantes.

On propose de définir une politique de décision en s'appuyant sur ces différents critères de sélection.

Pour une action a dans l'état s, soit $\boldsymbol{\lambda}(s, a)$ le vecteur des critères de sélection. On définit une fonction d'évaluation $g_{\boldsymbol{\theta}}(s, a) = \boldsymbol{\theta}^{\intercal}\boldsymbol{\lambda}(s, a)$ avec $\boldsymbol{\theta}$ un vecteur de paramètres de taille $|\boldsymbol{\lambda}(s, a)|$. Par convention, la politique de décision cherche l'action a qui maximise $g_{\boldsymbol{\theta}}(s, a)$. On souhaite donc que la valeur de $g_{\boldsymbol{\theta}}(s, a)$ soit plus grande lorsque l'action a est plus désirable, c'est à dire lorsqu'elle mène à des récompenses importantes, et en particulier on considère des critères λ avec la même propriété. Dans ce manuscrit $g_{\boldsymbol{\theta}}(s, a)$ est une fonction linaire, mais toute fonction différentiable peut être utilisé à la place.

Une fois calculée l'évaluation de chacune des actions de $A(s)$, on peut :

— Établir une politique de décision déterministe en choisissant l'action a qui maximise $g_{\boldsymbol{\theta}}(s, a)$

— Définir une politique de décision stochastique en calculant une distribution de probabilités sur les actions avec la fonction `softmax` :

$$\pi_{\boldsymbol{\theta}}(a \mid s) = \frac{e^{g_{\boldsymbol{\theta}}(s,a)/\beta}}{\sum_{a' \in A(s)} e^{g_{\boldsymbol{\theta}}(s,a')/\beta}} \forall a \in A(s) \tag{2.1}$$

Avec β un paramètre, appelé *température*, permettant de contrôler la distribution de sortie de la fonction `softmax`. Pour $\beta = 1$, la distribution n'est pas modifiée. Plus β s'approche de 0, et plus la distribution résultante sera discriminante. À l'inverse, lorsque β est grand, aucune action particulière n'émergera franchement. On peut aussi voir ce paramètre comme le degré de confiance que l'on accorde à la politique.

Ces politiques sont paramétrées par le vecteur $\boldsymbol{\theta}$, on va alors régler ce vecteur de poids avec l'algorithme REINFORCE en utilisant la politique stochastique.

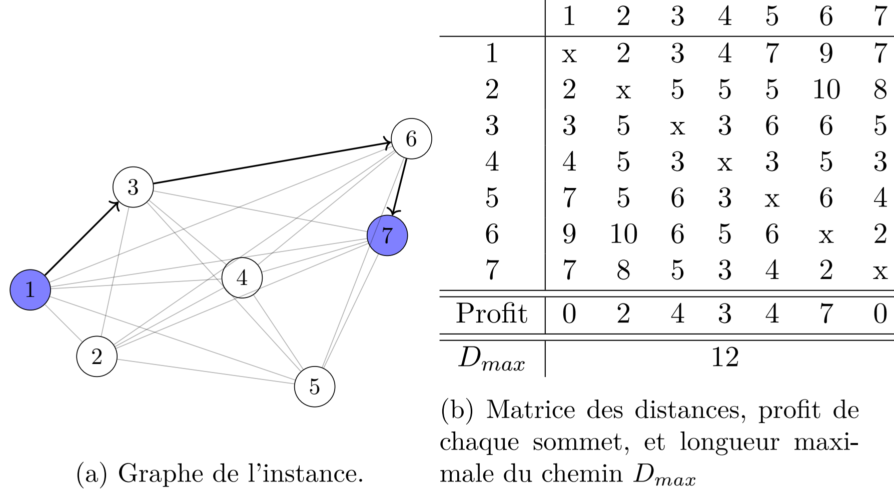
Exemple *Dans l'exemple du problème de la course d'orientation on dispose de plusieurs critères pour qualifier les différents sommets. Ces critères sont en partie inspirés par Golden et al., qui proposent également d'utiliser une combinaison linéaire pour les assembler [66]. Soit un état s correspondant à l'affectation des k premières variables $I = (x_1, v_1), ..., (x_k, v_k)$, pour chacun des noeuds candidats $i \in A(s)$, on considère les trois critères de sélection suivants :*

$$\lambda_1(s,i) = \frac{p_i}{d_{v_k,i}}$$
$$\lambda_2(s,i) = -(d_{i,1} + d_{i,n})$$
$$\lambda_3(s,i) = -d_{i,C(G')}$$

Avec $C(G')$ le centre du sous-graphe de G induit par $V' = V \setminus \{v_1, ..., v_k\}$.

Le premier critère $\lambda_1(s,i)$ représente une forme d'utilité : c'est le profit du noeud i divisé par sa distance au noeud actuel. Le second critère $\lambda_2(s,i)$ mesure la distance du noeud i aux deux noeuds essentiels de chaque chemin que sont le départ et l'arrivée. Ce critère est statique, dans le sens ou pour un sommet donné, sa valeur ne change pas selon l'état. Enfin, le dernier critère $\lambda_3(s,i)$ est la distance du noeud i au centre du sous-graphe induit par les noeuds restants. Ces trois critères ont été mis dans le sens de la maximisation, une action est plus désirable si ces critères sont grands.

Pour l'instance proposée dans la figure 2.1, à l'état initial s_0, les noeuds candidats sont les noeuds de 2 à 6 pour lesquels les fonctions d'évaluation dépendant de



	1	2	3	4	5	6	7
1	x	2	3	4	7	9	7
2	2	x	5	5	5	10	8
3	3	5	x	3	6	6	5
4	4	5	3	x	3	5	3
5	7	5	6	3	x	6	4
6	9	10	6	5	6	x	2
7	7	8	5	3	4	2	x
Profit	0	2	4	3	4	7	0
D_{max}	12						

(a) Graphe de l'instance.

(b) Matrice des distances, profit de chaque sommet, et longueur maximale du chemin D_{max}

FIGURE 2.1 – Exemple d'une instance du problème de course d'orientation

ces trois critères et du vecteur de paramètres $\boldsymbol{\theta}$ sont les suivantes :

$$
\begin{aligned}
g_{\boldsymbol{\theta}}(s_0, 2) &= \boldsymbol{\theta}_1 * 1 + \boldsymbol{\theta}_2 * -10 + \boldsymbol{\theta}_3 * -5 \\
g_{\boldsymbol{\theta}}(s_0, 3) &= \boldsymbol{\theta}_1 * 4/3 + \boldsymbol{\theta}_2 * -8 + \boldsymbol{\theta}_3 * -3 \\
g_{\boldsymbol{\theta}}(s_0, 4) &= \boldsymbol{\theta}_1 * 0.75 + \boldsymbol{\theta}_2 * -7 + \boldsymbol{\theta}_3 * 0 \\
g_{\boldsymbol{\theta}}(s_0, 5) &= \boldsymbol{\theta}_1 * 4/7 + \boldsymbol{\theta}_2 * -11 + \boldsymbol{\theta}_3 * -3 \\
g_{\boldsymbol{\theta}}(s_0, 6) &= \boldsymbol{\theta}_1 * 7/9 + \boldsymbol{\theta}_2 * -11 + \boldsymbol{\theta}_3 * -5
\end{aligned}
$$

Dans la pratique, l'ordre de grandeur des critères de sélection peut varier d'une instance à l'autre, et il convient généralement de normaliser ces critères. Cette normalisation permet de pouvoir apprendre et utiliser une valeur unique de $\boldsymbol{\theta}$ sans tenir compte de la variation de la valeur de ces critères pour un ensemble d'instance.

Exemple *Pour le problème de course d'orientation, on peut normaliser les trois critères de sélection dans [0, 1] en divisant chaque critère par les valeurs extrêmes issues de l'instance considérée :*

$$
\begin{aligned}
\lambda_1(s,i) &= \frac{p_i}{d_{v_k,i}} / \frac{\max_{j \in V} p_j}{\min_{(l,j) \in A} d_{l,j}} \\
\lambda_2(s,i) &= -(d_{i,1} + d_{i,n}) / (\max_{j \in V} d_{1,j} + \max_{j \in V} d_{j,n}) \\
\lambda_3(s,i) &= -d_{i,C(G')} / \max_{j \in G'} d_{j,C(G')}
\end{aligned}
$$

Pour l'exemple de la figure 1.1, dans l'état initial s_0 du problème et avec $\boldsymbol{\theta}^\intercal =$

$(0.5, 0.25, 0.25)$ *on obtient :*

$$g_{\boldsymbol{\theta}}(s_0, 2) \approx \boldsymbol{\theta}_1 * 0.29 + \boldsymbol{\theta}_2 * -0.59 + \boldsymbol{\theta}_3 * -1 \approx -0.25$$
$$g_{\boldsymbol{\theta}}(s_0, 3) \approx \boldsymbol{\theta}_1 * 0.38 + \boldsymbol{\theta}_2 * -0.47 + \boldsymbol{\theta}_3 * -0.6 \approx -0.08$$
$$g_{\boldsymbol{\theta}}(s_0, 4) \approx \boldsymbol{\theta}_1 * 0.21 + \boldsymbol{\theta}_2 * -0.41 + \boldsymbol{\theta}_3 * 0 \approx 0$$
$$g_{\boldsymbol{\theta}}(s_0, 5) \approx \boldsymbol{\theta}_1 * 0.16 + \boldsymbol{\theta}_2 * -0.65 + \boldsymbol{\theta}_3 * -0.6 \approx -0.23$$
$$g_{\boldsymbol{\theta}}(s_0, 6) \approx \boldsymbol{\theta}_1 * 0.22 + \boldsymbol{\theta}_2 * -0.65 + \boldsymbol{\theta}_3 * -1 \approx -0.3$$

En appliquant une politique de décision déterministe avec cette valeur de $\boldsymbol{\theta}$, le prochain noeud sélectionné est 4 afin de permettre de maximiser la fonction d'évaluation.

On peut transformer ces valeurs pour les fonctions d'évaluation de chaque action en distribution de probabilités par la fonction `softmax`. La table 2.1, donne les probabilités de chacune des actions selon la valeur du paramètre β. La première ligne donne la distribution de probabilités obtenue avec $\beta = 1$, la seconde avec $\beta = 0.5$ et la troisième ligne pour $\beta = 0.1$.

β	$\pi_{\boldsymbol{\theta}}(2 \mid s_0)$	$\pi_{\boldsymbol{\theta}}(3 \mid s_0)$	$\pi_{\boldsymbol{\theta}}(4 \mid s_0)$	$\pi_{\boldsymbol{\theta}}(5 \mid s_0)$	$\pi_{\boldsymbol{\theta}}(6 \mid s_0)$
1	0.18	0.22	0.24	0.19	0.17
0.5	0.16	0.24	0.28	0.17	0.15
0.1	0.05	0.27	0.60	0.06	0.03

TABLE 2.1 – Probabilité de chacune des action données par la fonction `softmax` selon plusieurs valeurs de température pour l'exemple du problème de la course d'orientation

Choisir la prochaine action consiste alors à tirer aléatoirement une action en suivant la probabilité calculée. En modifiant la valeur du paramètre β on peut intégrer plus ou moins d'aléatoire dans la politique.

2.1.3 Algorithme d'apprentissage d'une heuristique de valeurs

Comme présenté en 1.2.4, l'algorithme REINFORCE consiste à générer des épisodes en suivant une politique de décision, et à mettre à jour cette politique en prenant en compte le résultat de cet épisode. Pour implémenter cet algorithme dans le contexte de la résolution d'un problème d'optimisation combinatoire, on doit être capable de générer une solution en suivant la politique de décision, puis de calculer l'estimation du gradient de l'objectif (équation 1.47) à chacune des étapes.

La génération de solutions en suivant la politique de décision se fait simplement en calculant la distribution de probabilités à chaque étape, puis en sélectionnant une action aléatoirement en suivant cette distribution (pour rappel la politique doit être stochastique). Enfin, à chaque étape t de l'épisode, la règle de mise à jour est : $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \alpha * \gamma^t G(t) \nabla \ln \pi_{\boldsymbol{\theta}}(a_t \mid s_t)$, avec :

$$G(t) = \sum_{k=t+1}^{T} \gamma^{k-t-1} r_k \tag{2.2}$$

$$\nabla \ln \pi_{\boldsymbol{\theta}}(a_t \mid s_t) = (\boldsymbol{\lambda}(s_t, a_t) - \sum_{a' \in A(s_t)} \pi_{\boldsymbol{\theta}}(a' \mid s_t) * \boldsymbol{\lambda}(s_t, a'))/\beta \tag{2.3}$$

La mise à jour à une étape donnée t utilise alors le produit du *gain* et de la différence entre la valeur de chaque critère pour l'action a_t sélectionnée à l'étape t et la moyenne pondérée par la politique actuelle de ce critère.

Exemple *Dans l'exemple du problème de course d'orientation de la figure 2.1, prenons la solution proposée dans le graphe 2.1(a) comme résultat d'un épisode. On utilise comme récompense, l'accroissement de la borne inférieure triviale consistant à utiliser la somme des profits collectés jusqu'à présent. Cet épisode peut être décrit comme une séquence d'états, d'actions et de récompenses comme suit :*

$$s_0, a_0 = 3, r_1 = 4, s_1, a_1 = 6, r_2 = 7, s_2$$

On peut alors calculer $G(0)$ *(avec* $\gamma = 0.9$*) et* $\nabla \ln \pi_{\boldsymbol{\theta}}(a_0 \mid s_0)$ *pour* $\beta = 1$ *et* $\boldsymbol{\theta}^{\mathsf{T}} = (0.5, 0.25, 0.25)$ *:*

$$\begin{aligned} G(0) &= r_1 + \gamma * r_2 = 10.3 \\ \nabla \ln \pi_{\boldsymbol{\theta}}(A_0 \mid S_0) &\approx \begin{bmatrix} 0.38 \\ -0.47 \\ -0.6 \end{bmatrix} - \begin{bmatrix} 0.256 \\ -0.542 \\ -0.601 \end{bmatrix} \\ &= \begin{bmatrix} 0.124 \\ 0.072 \\ 0.001 \end{bmatrix} \end{aligned}$$

On met ensuite à jour $\boldsymbol{\theta}$ *:*

$$\boldsymbol{\theta} \leftarrow \begin{bmatrix} 0.5 \\ 0.25 \\ 0.25 \end{bmatrix} + \alpha * 10.3 \begin{bmatrix} 0.124 \\ 0.072 \\ 0.001 \end{bmatrix}$$

On réitère ensuite le même processus pour $G(1)$ *avant de générer un nouvel épisode avec la nouvelle valeur de* $\boldsymbol{\theta}$*.*

Enfin, il convient de discuter de l'ensemble d'apprentissage. Il est d'usage en apprentissage supervisé d'utiliser un ensemble de données d'entraînement et un ensemble de données de test séparé. En effet dans le contexte de l'apprentissage supervisé, on cherche généralement des modèles de prédiction qui doivent pouvoir généraliser et répondre à de nouvelles données non utilisées pour l'entraînement.

Dans notre contexte, on ne cherche pas particulièrement la généralisation dans le sens où notre modèle d'apprentissage est relativement simple, et est donc rapide

à entraîner. De ce fait, il peut être appris directement sur l'ensemble des données, et la phase d'apprentissage peut être vue comme une phase de pré-traitement d'un processus de résolution comme par exemple une recherche arborescente. Cette remarque ne tient plus si l'on complexifie le modèle d'apprentissage, car le temps d'apprentissage serait prohibitif, et ne pourrait plus être considéré comme une phase de pré-traitement.

Dans certains cas d'applications industrielles, on dispose parfois de plusieurs ensembles d'instances aux caractéristiques différentes. Cela peut être par exemple un problème de logistique que l'on retrouve dans plusieurs ateliers aux caractéristiques différentes (déplacements dans l'atelier, gammes de produits, temps de travail légal, etc...). Il existe alors plusieurs instances dans chaque atelier, avec des caractéristiques communes. Notre approche permet dans ce cas, d'utiliser un ensemble d'apprentissage correspondant à un ensemble d'instances aux caractéristiques communes. Le but est alors d'apprendre une valeur de $\boldsymbol{\theta}$ par ensemble d'instances (par atelier par exemple) afin de spécialiser l'algorithme. Si le problème varie dans le temps alors on pourra soit réutiliser la même valeur de $\boldsymbol{\theta}$ apprise initialement, soit réapprendre sur les nouvelles données. En initialisant la valeur de $\boldsymbol{\theta}$ à celle que l'on disposait précédemment, l'algorithme devrait alors converger plus rapidement que lors de l'apprentissage initial.

Sur un ensemble d'instance, une itération de l'algorithme consiste alors à générer plusieurs épisodes provenant de plusieurs instances avant de mettre à jour $\boldsymbol{\theta}$. Dans le chapitre 3 nous mettons à jour ce paramètre après avoir généré plusieurs épisodes pour chacune des instances de l'ensemble d'instance considéré. Dans le chapitre 4 nous mettons à jour le paramètre après avoir généré un épisode pour chaque instance.

De plus, pour la même raison que l'on veut normaliser les critères, on peut vouloir normaliser les récompenses. En effet la valeur de $G(t)$ est directement utilisée dans la mise à jour de $\boldsymbol{\theta}$, et l'amplitude de cette valeur peut varier entre instances, on veut alors que pour l'ensemble d'apprentissage cette valeur soit du même ordre de grandeur.

2.1.4 Intégration de l'heuristique dans une recherche arborescente

Une fois la valeur de $\boldsymbol{\theta}$ réglée, il est très simple d'intégrer l'heuristique de choix de valeurs obtenue dans une recherche arborescente. En effet, à chaque ouverture de noeud, correspondant à l'état s, il suffit de calculer $\boldsymbol{\lambda}(s,a)$ pour tout $a \in A(s)$, puis de calculer la combinaison linéaire $g_{\boldsymbol{\theta}}(s,a)$. Si l'on veut une heuristique déterministe de choix de valeurs, alors l'évaluation des décisions nous permet de définir un ordre d'exploration en les triant par évaluation décroissante. Si l'on veut une heuristique stochastique de choix de valeurs, il faut alors calculer la distribution de probabilités avec la fonction `softmax`, le paramètre de température β permettant alors d'ajuster l'aléatoire dans l'heuristique. Cette distribution nous permet de sélectionner une décision avec de l'aléatoire. Dans le cas où notre fonction d'évaluation dépend des autres actions disponibles (ce ne sera pas le cas dans ce manuscrit), lors de retours-

arrière, on peut éventuellement recalculer les évaluations et la nouvelle distribution de probabilités. Sinon, on peut suivre l'ordre des valeurs calculé à l'ouverture du noeud.

2.2 Recherche arborescente de Monte Carlo

La seconde approche étudiée dans cette thèse consiste à s'appuyer sur la recherche arborescente de Monte Carlo (MCTS) pour résoudre des problèmes d'optimisation combinatoire. En théorie, cet algorithme parcourt entièrement l'espace de recherche, et doit donc retourner la solution optimale pour un budget temporel infini. En pratique, cet algorithme ne fait que très peu d'élagage et l'exploration complète sera prohibitive. Il sera alors utilisé pour trouver des solutions approchées.

Dans notre approche de l'utilisation de MCTS, on se place dans le cadre de processus de décision markovien décrit à la section 2.1. On rappelle donc qu'on dispose d'un ordre des variables à instancier et que l'on peut évaluer la fonction objectif pour n'importe quelle instanciation. Enfin dans la version que nous présentons dans cette section, l'hypothèse de la facilité de construction d'une solution réalisable est nécessaire pour la phase de simulation. Dans le cas où l'on utilise une version relâchée du problème, on peut restreindre cette hypothèse à la phase de simulation seulement et tenir compte de toutes les contraintes pour les noeuds de l'arbre développé.

Dans cette section, nous décrivons les trois contributions à la méthode MCTS pour la résolution de problèmes d'optimisation combinatoire.

2.2.1 Utilisation des bornes

La plupart des application de la recherche arborescente de Monte Carlo utilisent la notion de récompense seulement lors de l'arrivée à un état final [152, 141], comme c'est le cas pour les jeux, où la seule récompense est donnée avec la victoire ou défaite, et cette récompense ne s'accorde que lorsque l'on connaît le résultat, c'est à dire à un état final. Dans notre approche, nous considérons qu'une récompense peut être associée à chaque arc (action) visité à partir d'un noeud (état) donné. Cela permet notamment d'ajouter un facteur de décroissance exponentielle lors du calcul du *gain* (eq. 1.18). Certaines approches d'apprentissage par renforcement utilise l'accroissement marginal d'une borne inférieure triviale de la fonction objectif (en minimisation) comme pénalité [92, 28, 30, 81], mais à notre connaissance aucune application de la recherche arborescente de Monte Carlo n'utilise de facteur de décroissance dans le calcul du *gain*. La motivation de cette proposition est la suivante. Lorsque les branches sont très longues, on peut conjecturer qu'à mesure qu'on s'éloigne d'une décision initiale, son impact sur la qualité de la solution finale s'amenuise. Il y une forme de rendement décroissant de l'information lors des procédures gloutonnes utilisées pendant la phase de simulation. Cette utilisation de l'accroissement de l'évaluation de la fonction objectif comme une récompense à chaque décision, couplé à un facteur de décroissance exponentielle permet donc

de diminuer l'impact à long terme sur l'évaluation des décisions explorées dans la recherche arborescente de Monte Carlo.

Pour décrire le calcul de la phase de rétropropagation, nous nous plaçons dans le cas d'un problème de minimisation et utilisons l'opposé de l'accroissement marginal de la borne inférieure de la fonction objectif comme récompense. Soit une feuille v de l'arbre de recherche à partir de laquelle est lancée une simulation. L'état final de cette simulation est de profondeur T par rapport à la racine de l'arbre. Pour chacune des étapes t de l'épisode complet (démarrant à la racine) et s'arrêtant à l'état final obtenu par la simulation, on suppose qu'on dispose de l'instanciation I_t correspondante, et de sa borne inférieure $f_{LB}(I_t)$.

La phase de rétropropagation consiste alors pour chacun des arcs e emprunté à partir de v' à l'étape t, de la racine jusqu'à v, à calculer le *gain* $G(t)$ puis à mettre à jour $Q(v', e)$ et $N(v', e)$ comme suit :

$$G(t) = \sum_{k=t}^{T-1} \gamma^{k-t}(f_{LB}(I_t) - f_{LB}(I_{t+1})) \tag{2.4}$$

$$Q(v', e) \leftarrow Q(v', e) + \frac{G(t) - Q(v', e)}{N(v', e) + 1} \tag{2.5}$$

$$N(v', e) \leftarrow N(v', e) + 1 \tag{2.6}$$

On note également que l'utilisation des bornes permet l'élagage de l'arbre, aussi bien en évitant l'ajout de noeud dont on sait qu'ils ne mènent à aucune solution améliorante lors de la phase d'expansion, que pendant la phase de sélection lorsqu'un noeud déjà visité ne peut plus mener vers une solution améliorante. La suppression de noeuds dans l'arbre n'est pas initialement prévue dans la méthode et pose la question de comment traiter les informations contenues dans l'arc menant à ce noeud. Ces informations correspondent aux données collectées pour toutes les itérations qui sont passées à travers cet arc lors de la phase de sélection. Nous avons choisi dans ces travaux lorsqu'un noeud est supprimé, de supprimer également toutes les informations concernant les itérations qui sont passées par ce noeud de tous les arcs de la racine vers le père de ce noeud de la même manière que dans Keszocze et al. [91]. En effet, toutes ces informations ne sont plus pertinentes pour le reste de la recherche car elles concernent des parties de l'espace de recherche qui ne sont plus accessibles.

2.2.2 Recherche en profondeur

Nous proposons une hybridation de la méthode MCTS originelle avec une recherche en profondeur d'abord (DFS). En effet, il arrive parfois que les simulations faites via un algorithme glouton nous amènent dans des zones intéressantes de l'espace de recherche qu'on aimerait pouvoir intensifier. C'est pourquoi nous proposons, lorsque les solutions retournées par l'algorithme glouton sont estimées de bonne qualité, d'intensifier l'exploration autour de cet état final via un budget de conflits avec une méthode de recherche en profondeur avec retours-arrière. Ce budget est

dépendant de la qualité de la solution obtenue par l'algorithme glouton.

Autour de cette idée, plusieurs variations sont possibles. Nous donnons ici une approche générale possible.

La première partie de la phase de simulation hybride est identique à une simulation classique : à partir du noeud sélectionné par les premières phases, une procédure gloutonne stochastique est lancée jusqu'à obtention d'un état final qui est une solution du problème. On désigne cette solution par l'instanciation I. On accorde un budget de DFS suivant cette simulation selon trois paramètres, un paramètre de seuil Γ, ainsi que β^+ et β^-, deux paramètres définissant le budget maximal et minimal pour le DFS. Le budget accordé, noté β, vaut alors :

$$\beta = \begin{cases} \beta^+ & \text{si } I \text{ est améliorant} \\ \beta^- + (\beta^+ - \beta^-) * \Phi(I, \Gamma) & \text{si } \Phi(I, \Gamma) > 0 \\ 0 & \text{sinon} \end{cases}$$

Avec $\Phi(I, \Gamma)$ une fonction dans $[0, 1]$ qui évalue l'instanciation obtenue par la procédure gloutonne par rapport au seuil Γ. Nous donnerons deux variations autour de cette approche pour définir le budget dans les chapitres 3 et 4.

A partir de β, le budget de conflits, on peut effectuer un DFS à partir de l'état final atteint par la simulation. Pendant ce DFS, on peut utiliser des bornes sur la valeur de la fonction objectif et donc élaguer des parties de l'arbre. Il est alors très probable que le DFS commence par faire des retours-arrière jusqu'à obtenir un noeud valide du point de vue des bornes. Une limite pour ce DFS est de ne pas dépasser le cadre du sous-arbre enraciné en le noeud d'où est partie la phase de simulation initiale. On empêche alors tout retour-arrière remontant au-delà de ce noeud. Si cela devait arriver, on considère alors que la recherche est complète pour ce sous arbre, le noeud correspondant est alors clos, et il retiré de l'arbre. Lorsque le budget β est entièrement consommé, alors la meilleure solution obtenue, soit lors du glouton initial, soit lors du DFS (ce qui voudrait dire alors qu'une nouvelle solution globale a été obtenue) est alors retenue pour le calcul du *gain* et la phase de rétropropagation.

2.2.3 Compromis dynamique

En pratique, à mesure que la méthode MCTS progresse, on se rend compte que la profondeur moyenne à laquelle sont lancées les simulations stagne. La méthode MCTS exécute alors une forme de recherche en largeur et n'est plus bien adaptée pour l'exploration de l'espace de recherche. Cela peut être problématique dans le cas où les problèmes considérés sont de très grande taille, c'est à dire pour lesquels la profondeur des états finaux est importante. C'est le rôle du terme d'exploration de dévier de la meilleure action actuellement estimée. On peut contrôler cette balance entre exploitation et exploration avec le paramètre c (eq. 1.48), mais une valeur trop petite produit l'effet inverse, l'algorithme se transforme en une forme de recherche en profondeur d'abord. On propose alors de régler dynamiquement ce coefficient

en fonction de la profondeur de l'arbre, et en fonction de la profondeur du noeud considéré lors du calcul de l'équation 1.48. Soit $\delta(\mathcal{T})$, la profondeur de l'arbre de recherche $\mathcal{T}$ en mémoire, et t l'étape de sélection (0 à la racine), le coefficient exploitation/exploration sera :

$$\Lambda^{\delta(\mathcal{T})-t} * c \tag{2.7}$$

avec $\Lambda \leq 1$ un paramètre (la valeur 1 désactivant le mécanisme).

Dans l'utilisation classique de la méthode MCTS pour les jeux, il y a généralement une phase de validation (ou de *commit*). Après un certain nombre d'itérations, une action est définitivement choisie et devient la nouvelle racine de l'arbre. Puis, la méthode MCTS reprend pour explorer seulement le sous-arbre choisi. En effet, dans le contexte d'un jeu, un coup doit être joué, et on peut garder en mémoire les parties de l'arbre sous ce coup que l'on a déjà exploré, en oubliant les autres.

Dans le contexte de l'optimisation combinatoire ce mécanisme de commit a été utilisé dans [152]. L'algorithme n'est alors plus complet étant donné que cette action est irréversible. Le compromis dynamique proposé s'approche de ce mécanisme de *commit*. En effet, pour des noeuds proches de la racine $\Lambda^{\delta(\mathcal{T})-t}$ va tendre vers 0 à mesure que l'arbre s'allonge. Les premières décisions vont alors tendre à être toujours celles qui ont la meilleure valeur de $Q(v, e)$. Inversement, à un noeud proche d'une feuille, ce terme va être proche du coefficient c original, et donc favoriser plus l'exploration (selon la valeur de c choisie). L'avantage de l'approche basée sur un coefficient dynamique est que l'algorithme reste complet car aucune partie de l'arbre n'est supprimée.

2.3 Synthèse

Dans ce chapitre, nous avons proposé deux approches pour la résolution de problèmes d'optimisation combinatoire. La première est un cadre pour l'apprentissage d'heuristiques de choix de valeurs pour un problème donné. Cette heuristique prend appui sur des connaissances que l'on a du problème pour définir des critères de sélection de la prochaine décision à prendre. Ces critères sont assemblés en une combinaison linéaire dont les poids sont réglés via un algorithme d'apprentissage par renforcement : l'algorithme REINFORCE. Ce cadre n'est pas limité aux combinaisons linéaires, et peut être utilisé avec tout autre modèle différentiable. Nous avons ensuite montré qu'une telle heuristique est facilement intégrable dans une recherche arborescente.

La seconde approche est la recherche arborescente de Monte Carlo que nous proposons d'hybrider avec une recherche en profondeur d'abord pour la phase de simulation. Nous proposons également un mécanisme de compromis dynamique, ainsi que l'utilisation du facteur de décroissance exponentielle dans le calcul du *gain* pour les problèmes à grande échelle pour lesquels la profondeur de l'arbre exploré peut être importante.

Ces deux approches peuvent être combinées, la recherche arborescente de Monte

Carlo utilisant alors l'heuristique pour la phase de simulation.

Dans les deux chapitres suivants, nous développons ces contributions sur deux problèmes industriels. Nous proposons d'apprendre une heuristique de choix de valeurs basée sur une combinaison linéaire de critères puis de l'intégrer dans une recherche en profondeur avec retours-arrière, avant de l'utiliser conjointement avec une recherche arborescente de Monte Carlo.

Chapitre 3

Problème de tournées de chariots dans un atelier d'assemblage

Sommaire

3.1 Description du problème 58

3.2 Travaux connexes 62

3.2.1 Liens avec des problèmes de la littérature 62

3.2.2 Modèle existant basé sur `LocalSolver` 64

3.3 Modèles de programmation par contraintes 65

3.3.1 Modèle d'ordonnancement 66

3.3.2 Modèle de voyageur de commerce 67

3.3.3 Stratégie de branchement 69

3.4 Évaluation expérimentale des modèles de programmation par contraintes 69

3.4.1 Jeux de données 69

3.4.2 Configuration 71

3.4.3 Résultats expérimentaux 71

3.5 Apprentissage par renforcement d'heuristiques de choix de valeurs 74

3.5.1 Processus de décision markovien 74

3.5.2 Caractérisation de l'heuristique de branchement 75

3.5.3 Intégration de l'heuristique apprise dans des méthodes arborescentes 76

3.5.4 Intégration de l'heuristique apprise dans une méthode de recherche locale 77

3.5.5 Résultats expérimentaux 81

3.6 Recherche arborescente de Monte Carlo 85

3.6.1 Algorithme MCTS pour le problème de tournées de chariots . 85

3.6.2 Résultats expérimentaux 87

3.7 Synthèse 90

Le problème industriel étudié dans ce chapitre vise à planifier les tournées d'une équipe d'opérateurs logistiques dans un atelier d'assemblage du secteur automobile.

Les opérateurs logistiques conduisent chacun un véhicule leur permettant de déplacer des chariots de pièces entre les machines qui les produisent et les machines qui les consomment. Chaque opérateur est affecté à un ensemble de machines, le problème à résoudre consiste à planifier les déplacements de ces opérateurs en respectant des fenêtres temporelles imposées par les cycles de production/consommation des pièces transportées. Nous nous sommes intéressés au problème de conception de la tournée d'un seul opérateur, c'est à dire à la planification des déplacements des différents chariots pour un ensemble donné de machines.

La première section de ce chapitre comporte la description formelle du problème étudié. Dans la seconde section, nous positionnons ce problème par rapport à des problèmes classiques de la littérature et nous présentons la méthode utilisée dans l'entreprise pour le résoudre. La troisième section détaille deux modèles de programmation par contraintes que nous avons proposés. Une évaluation expérimentale de ces modèles est détaillée dans la quatrième section. La cinquième section montre comment l'utilisation de l'apprentissage par renforcement peut améliorer les résultats obtenus. Enfin, dans la sixième section, l'apport d'une recherche arborescente de Monte Carlo sera présenté et évalué.

3.1 Description du problème

La ligne d'assemblage étudiée se compose de m types de pièces, que l'on appelle **composants**, à déplacer entre deux machines spécifiques correspondant à leurs lieux de production et leurs lieux de consommation respectifs. Chaque composant dispose de son propre type de chariot pouvant contenir plusieurs exemplaires de la pièce. On dispose d'un total de quatre chariots par composant. La situation initiale est que deux chariots, un chariot vide et un chariot plein, sont positionnés devant chacun des postes de production et de consommation. Quand un chariot d'un composant est plein au poste de production, un opérateur doit l'emmener au poste de consommation associé. De manière symétrique, lorsqu'un chariot est vide au poste de consommation, le chariot doit être ramené par l'opérateur au poste de production.

On appelle **temps de cycle de production**, c_i, le temps mis pour produire et consommer, un chariot de composant i. (ce temps est identique pour le poste de production et le poste de consommation d'un même type de pièces). Sur un horizon temporel H, l'atelier fonctionne en continu, et la fin d'un cycle de production marque le début du prochain cycle jusqu'à H. On note n_i le nombre de cycles d'un composant i, on a donc $n_i = \lfloor \frac{H}{c_i} \rfloor$.

Pour un composant i, à chaque cycle $k \leq n_i$, l'opérateur doit réaliser deux opérations de collecte et deux opérations de livraison : la collecte pe_i^k et la livraison de_i^k du chariot vide du poste de consommation vers le poste de production, ainsi que la collecte pf_i^k et la livraison df_i^k du chariot plein du poste de production vers le poste de consommation.

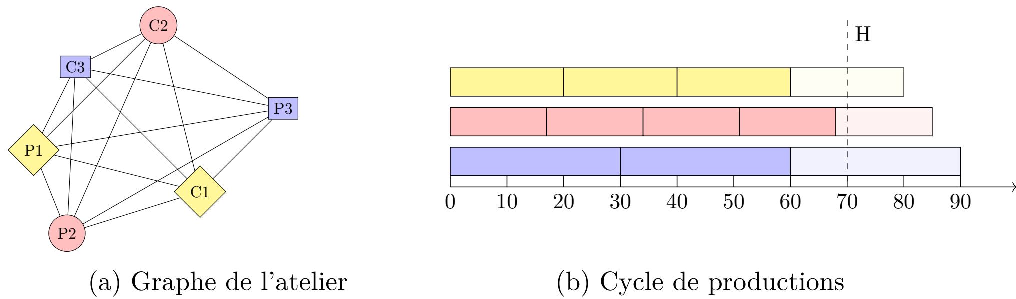
Soit P l'ensemble des opérations de collecte et soit D l'ensemble des opérations

de livraison, on note $\mathcal{A} = P \cup D$ **l'ensemble des opérations** de déplacement de chariots avec : $P = \bigcup_{i=1}^{m}\left(\bigcup_{k=1}^{n_i}\{pe_i^k, pf_i^k\}\right)$ et $D = \bigcup_{i=1}^{m}\left(\bigcup_{k=1}^{n_i}\{de_i^k, df_i^k\}\right)$.

Sur un horizon donné, l'atelier ne doit jamais s'arrêter. Par conséquence, les quatre opérations de déplacement de chariots du k-ème cycle du composant i doivent être réalisées durant l'intervalle temporel correspondant à ce cycle : $[(k-1)c_i, kc_i]$. Pour une opération $a \in \mathcal{A}$, on dénote par r_a sa **date de disponibilité**, et par d_a sa **date d'échéance**. Les dates de disponibilité et d'échéance correspondent aux différents cycles de production des composants. La **durée d'une opération** a est désignée par p_a et correspond au temps nécessaire pour accrocher ou décrocher le chariot.

Les durées des trajets entre chaque paire de l'ensemble des postes de production et des postes de consommation sont connues. Les opérations de collecte et de livraison étant associées à un lieu donné, on peut alors définir la **durée de trajet** $D_{a,b}$ entre deux opérations a et b. Ces temps de trajet sont symétriques : $D_{a,b} = D_{b,a}, \forall\{a,b\} \in \mathcal{A}$, et respectent l'hypothèse de l'inégalité triangulaire : $\forall\{a,b,c\} \in \mathcal{A} : D_{a,b} \leq D_{a,c} + D_{c,b}$

Exemple *Soit un atelier d'assemblage composé de* 3 *composants : jaune, bleu et rouge. Pour chacun des composants i, il y a un poste de production* P_i *et un poste de consommation* C_i*. On dispose de la durée de trajet entre chaque paire de postes de travail. Le graphe de la figure 3.1(a) représente l'atelier dans lequel ces trois composants sont produits et consommés ainsi que l'ensemble des routes possibles dans l'atelier entre chaque poste de consommation et de production. Le graphe considéré est alors complet. Chaque composant a un temps de cycle spécifique, illustré par la figure 3.1(b) : cycle de* 20 *unités de temps pour le composant jaune, de* 17 *unités pour le composant rouge, et de* 30 *unités pour le composant bleu). Ces cycles s'enchaînent jusqu'à l'horizon* H *(fixé à* $H = 70$ *dans cet exemple). Il y a donc trois cycles pour le composant jaune, quatre pour le composant rouge et deux cycles pour le composant bleu.*



(a) Graphe de l'atelier (b) Cycle de productions

FIGURE 3.1 – Exemple du problème de tournée de chariots

Quatre chariots dédiés sont associés à chaque composant i. Pour chaque cycle de chaque composant, il y a quatre opérations de collecte ou de livraison de chariots. Par exemple, pour le composant jaune, il faut réaliser quatre opérations de déplacement de chariot entre la date 0 *de début du premier cycle et la date* 20 *de*

fin du premier cycle. Quatre opérations de déplacement de chariots doivent ensuite être effectuées pour le second cycle (entre 20 *et* 40*), ainsi que quatre autres opérations pour le troisième cycle (entre* 40 *et* 60*). Pour l'ensemble des composants et des cycles de la figure 3.1, il y a un total de* 36 *opérations à planifier (*12 *pour le composant jaune,* 16 *pour le composant rouge et* 8 *pour le composant bleu).*

Une solution de ce problème peut se représenter comme une séquence d'opérations. On cherche alors une séquence σ des n opérations. On note $s(\sigma, j)$ la date de début de la j-ème opération de la séquence σ et $e(\sigma, j)$ sa date de fin. Pour qu'une solution soit valide, il faut respecter les trois contraintes suivantes :

Contraintes de fenêtres de temps. Chacune des opérations doit être effectuée dans une fenêtre de temps. La date de début d'une opération est :

$$s(\sigma, j) = \begin{cases} r_{\sigma(j)} & \text{si } j = 1 \\ \max(r_{\sigma(j)}, s(\sigma, j-1) + p_{\sigma(j-1)} + D_{\sigma(j-1),\sigma(j)}) & \text{sinon} \end{cases}$$

La date de fin $e(\sigma, j) = s(\sigma, j) + p_{\sigma(j)}$ de l'opération $\sigma(j)$ doit être inférieure à sa date d'échéance :

$$\forall j \in [1, n],\ e(\sigma, j) \leq d_{\sigma(j)} \tag{3.1}$$

Contrainte de précédence. Chaque opération de collecte doit précéder l'opération de livraison associée. Soit $\rho = \sigma^{-1}$ l'association des opérations à leur position dans la séquence, cette contrainte se note ainsi :

$$\rho(pf_i^k) < \rho(df_i^k) \ \wedge\ \rho(pe_i^k) < \rho(de_i^k) \qquad \forall i \in [1, m]\ \forall k \in [1, n_i] \tag{3.2}$$

De plus, sous l'hypothèse d'une durée nulle des opérations (accrochage et décrochage des chariots), on remarque qu'il n'y a que deux ordres possibles pour les quatre opérations d'un même cycle de production. En effet, la collecte du chariot plein (resp. vide) et la livraison du chariot vide (resp. plein) se font au même endroit, à savoir le lieu de production (resp. consommation) du composant, et doivent être faites dans le même intervalle temporel. Comme la première livraison (soit d'un chariot plein soit d'un chariot vide) et la seconde collecte se font au même endroit, effectuer la seconde collecte avant de faire la première livraison est dominé du point de vue de la longueur du train de chariots. Dans les données réelles considérées, la durée de traitement des opérations est négligeable par rapport aux durées de déplacement et aux fenêtres de temps, on décide alors d'imposer une contrainte de précédence entre la première livraison et la seconde collecte pour chaque cycle de production :

$$\rho(df_i^k) < \rho(pe_i^k) \ \vee\ \rho(de_i^k) < \rho(pf_i^k)\ \forall i \in [1, m]\ \forall k \in [1, n_i] \tag{3.3}$$

Cette contrainte permet de réduire fortement l'espace de recherche en imposant une unique décision binaire pour l'ordre relatif des quatre opérations de chacun des cycles.

Contrainte de longueur du train de chariots. L'opérateur peut accrocher les chariots les uns aux autres, de manière à former un train de chariots. Nous faisons l'hypothèse que l'ordre des chariots dans le train n'affecte pas la durée de décrochage. De ce fait, une opération de collecte ne doit pas nécessairement être directement suivie par la livraison associée. Cependant, la longueur totale du train de chariots ne doit pas dépasser une longueur maximale fixée $T_{\max}$. On note t_i^* la longueur des chariots utilisés pour transporter un composant i. L'occupation du train pour une opération donnée a_i, associée à un composant i, est notée t_{a_i}. Sa valeur (positive ou négative) dépend de la nature de l'opération (collecte ou livraison) : $t_{a_i} = t_i^*$ si $a_i \in P$ et $t_{a_i} = -t_i^*$ si $a_i \in D$.

La contrainte limitant l'occupation maximale du train de chariots s'écrit alors :

$$\forall j \in [1, n], \ \sum_{l=1}^{j} t_{\sigma(l)} \leq T_{\max} \tag{3.4}$$

Optimisation Pour la mise en oeuvre des contributions présentées au chapitre 2, nous avons besoin de considérer une fonction objectif pour attribuer les récompenses. Pour transformer le problème de satisfaction étudié en problème d'optimisation, nous proposons de relâcher les contraintes 3.1 sur les dates d'échéance des opérations et de considérer comme objectif la minimisation du retard maximal.

Soit $L(\sigma, j) = e(\sigma, j) - d_{\sigma(j)}$ le retard de l'opération $\sigma(j)$, l'objectif de minimisation du retard maximal s'écrit $\max\{L(\sigma, j) \mid 1 \leq j \leq |\sigma|\}$. On définit également le retard maximal sur une sous-séquence d'opérations : $L(\sigma, j, l) = \max\{L(\sigma, q) \mid j \leq q \leq l\}$, et on note $L(\sigma)$ le retard de l'ensemble de la séquence d'opérations σ : $L(\sigma, 1, |\sigma|)$. Lorsque la valeur $L(\sigma)$ devient nulle (ou négative), le problème de satisfaction est résolu.

Pour le problème de minimisation du retard maximal, les opérations peuvent finir après leurs dates d'échéance, il est alors nécessaire de rendre explicite les contraintes de précédence entre les opérations de déplacement de chariots d'un même composant des différents cycles de production :

$$\max(\rho(df_i^{k-1}), \rho(de_i^{k-1})) < \min(\rho(pe_i^k), \rho(pf_i^k)) \forall i \in [1, m] \ \forall k \in [2, n_i] \tag{3.5}$$

Avec cette relaxation des contraintes sur les dates d'échéance, il est facile de construire une solution avec une procédure gloutonne dans l'ordre de la séquence, en respectant les contraintes de précédences entre les opérations et la longueur du train.

Une autre relaxation du problème initial de satisfaction est possible, elle consiste à relâcher la contrainte 3.4 sur la longueur du train de chariots, et de considérer

comme objectif la minimisation de cette longueur maximale. Cependant, en relâchant cette contrainte, construire une solution reste difficile en raison des contraintes sur les fenêtres de temps. Pour les approches proposés, nous avons besoin de pouvoir construire une solution facilement.

Complexité Trouver une solution réalisable au problème de chariot est NP-Complet. En effet, on peut réduire une instance du problème de voyageur de commerce à une instance de notre problème. Il faut, pour cela, choisir un sommet (qui correspondra à notre point de départ pour le voyageur de commerce) que l'on duplique, ces deux sommets sont éloignés d'une distance $2M$ avec M largement supérieur à k la longueur maximale accordée au voyageur de commerce. On ajoute également M à la longueur de tous les autres arcs sortants de ces deux sommets dupliqués. Les autres arcs sont inchangés. Ensuite, chaque sommet du graphe est dupliqué (à nouveau pour celui que l'on avait sélectionné précédemment), pour représenter le point de consommation et le point de production. On a alors une instance de notre problème composée de $|V|+1$ composants (V l'ensemble des sommets du graphe du problème d'origine), on fixe le même temps de cycle c_i de $2M + k$ pour tous les composants, et l'instance considérée est d'horizon $2M + k$ également. La longueur maximale du train est de 1, chaque chariot à une longueur de 1, et les durées des opérations sont nulles. Une solution valide de cette instance commence et finit par le sommet qui correspond au point de départ du voyageur de commerce, et doit enchaîner immédiatement chaque collecte avec sa livraison. De plus, puisque les distances entre postes de production et de consommation sont nulles, la solution qui enchaîne les quatre opérations de chaque composant est dominante. Par conséquent, il existe une solution à ce problème de chariots, si et seulement si il existe une solution d'une distance au plus k pour le problème de voyageur de commerce.

Le tableau 3.1 présente une synthèse des notations utilisées pour le problème de tournées de chariots.

3.2 Travaux connexes

Le problème de tournées de chariots étudié est un cas particulier du problème de collectes et livraisons à un véhicule avec fenêtres de temps et capacité (la longueur du train). Nous présentons dans cette section les liens avec les problèmes de la littérature, et l'approche utilisée par Renault.

3.2.1 Liens avec des problèmes de la littérature

Ce problème de tournées de chariots est à classer dans la catégorie des problèmes *one-to-one* [18], c'est à dire que chaque point de collecte (resp. de livraison) est associé à un unique point de livraison (resp. de collecte).

Dans la littérature, plusieurs approches exactes ont été proposées pour des problèmes de collecte et livraison à un véhicule. On trouve des algorithmes de séparation

m	Nombre de composants
n	Nombre total d'opérations
$\mathcal{A}$	Ensemble des opérations
P	Ensemble des opérations de collecte
D	Ensemble des opérations de livraison
H	Horizon temporel
c_i	Temps de cycle de production du composant i
n_i	Nombre de cycles de production du composant i
pe_i^k	Opération de collecte du chariot plein du cycle k du composant i
de_i^k	Opération de livraison, du chariot plein du cycle k du composant i
pf_i^k	Opération de collecte du chariot vide du cycle k du composant i
df_i^k	Opération de livraison du chariot vide du cycle k du composant i
r_a	Date de disponibilité de l'opération a
d_a	Date d'échéance de l'opération a
p_a	Durée de l'opération a
t_i^*	Longueur du chariot utilisé pour transporter le composant i
t_a	Occupation du train de chariots pour l'opération a
$D_{a,b}$	Durée du trajet entre les opérations a et b
σ	Solution représentée comme une séquence des n opérations
$\sigma(j)$	Opération située en j-ème position dans la séquence σ
$s(\sigma, j)$	Date de début de la j-ème tâche de la séquence d'opérations σ
$e(\sigma, j)$	Date de fin de la j-ème tâche de la séquence d'opérations σ
$L(\sigma, j)$	Retard de la j-ème opération de la séquence σ
$L(\sigma, j, l)$	Retard maximal de la sous séquence entre les positions j et l de σ
$L(\sigma)$	Retard maximal de la séquence σ

Table 3.1 – Notations pour le problème de tournées de chariots

et évaluation [90], certain avec coupe [53, 150, 76], ainsi que de la programmation dynamique [53, 50]. Seulement la plupart de ces approches ne prennent en compte ni la capacité, ni les fenêtres de temps. Des approches incomplètes ont également été proposées. Il existe plusieurs heuristiques constructives dans le cas sans fenêtre de temps ni capacité [63, 146]. Carrabs et al. [33] proposent une recherche à voisinages variables (*Variable Neighborhood Search*) pour une variante du problème avec contrainte sur le chargement. Plusieurs heuristiques de perturbation ont été proposées [147], heuristiques qui peuvent être assimilées à de la recherche locale. Les fenêtres de temps sont prise en compte par van der Bruggen et al. [169] avec une méthode de recherche locale qui relâche initialement les contraintes sur les fenêtres de temps avant de minimiser la longueur du tour ainsi que par Landrieu et al. [105], avec une méthode de recherche tabou.La contrainte de fenêtres de temps ainsi que la contrainte de capacité a été prise en compte dans une approche de type Monte Carlo (*Nested Monte Carlo* [34]) [54]

Les contraintes de fenêtre de temps et de capacité sont régulièrement prises en compte dans les problèmes de transport à la demande (*Dial-a-Ride Problem*). Cependant, dans cette catégorie de problèmes, la fonction objectif intègre souvent

une notion de qualité de service [43].

Pour le problème que l'on considère, il n'y a pas d'objectif à optimiser, mais déterminer une solution réalisable est difficile. Par ailleurs, les cycles de production-consommation entraînent une structure temporelle très particulière. En effet, pour chaque composant les quatre opérations d'un même cycle doivent être effectuées pendant la même fenêtre de temps, et aux deux mêmes endroits, de plus, ces opérations seront répétées pour chacun des cycles jusqu'à l'horizon temporel fixé. L'hétérogénéité des temps de cycles de production (pour chaque composant, le temps de cycle peut être différent) entraînent des décalages entre les fenêtres temporelles des opérations de déplacement des chariots des différents composants. Ainsi, un plan de route réalisable pour un horizon temporel court ne peut pas être seulement répété pour un horizon temporel plus long car cela peut conduire à une solution non réalisable.

Les méthodes de type recherche à voisinage large (Large Neighborhood Search, LNS) sont fréquemment utilisées pour résoudre des problèmes de tournées de véhicules [149]. Pour notre problème, ces méthodes sont très impactées par la structure temporelle. En effet, lorsqu'une requête est retirée de la séquence, le nombre d'alternatives pour la réinsertion est limité par les précédences dues aux cycles de production. Des expériences préliminaires ont été faites avec une méthode de type *LNS* mais cette voie n'a pas été poursuivie car ces essais n'ont pas été concluants.

3.2.2 Modèle existant basé sur `LocalSolver`

On présente dans cette section le modèle `LocalSolver` qui est utilisé par l'entreprise pour la résolution de problème de tournées de chariots. `LocalSolver` est un solveur commercial s'appuyant sur une méthode de recherche locale pour la résolution de problèmes combinatoires. Le modèle développé est un modèle de type voyageur de commerce. Il est basé sur deux variables de type liste (type de variable spécifique à `LocalSolver`) utilisées pour représenter la séquence d'opérations, et deux variables par opération correspondant aux dates de début et à la longueur du train :

— variable de type liste ***seq*** de longueur n , où $seq_j = a$ signifie que l'opération a est effectuée en position j ;

— variable de type liste ***pos*** de longueur n, où $pos_a = j$ permet d'associer à chaque opération a sa position j dans la séquence ;

— des variables entières $train_j$ indiquant la longueur du train après l'opération effectué en position j dans la séquence ;

— des variables entières end_j indiquant la date de fin de l'opération effectuée à la position j

Le modèle est défini par les contraintes suivantes :

$$end_1 = r_{seq_1} + p_{seq_1} \tag{3.6}$$
$$train_1 = t_{seq_1} \tag{3.7}$$
$$end_j = \max(r_{seq_j}, end_{j-1} + D_{seq_{j-1},seq_j}) + p_{seq_j} \quad \forall j \in [2, n] \tag{3.8}$$
$$train_j = train_{j-1} + t_{seq_j} \quad \forall j \in [2, n] \tag{3.9}$$
$$end_j \leq d_{seq_j} \quad \forall j \in [1, n] \tag{3.10}$$
$$train_j \leq T_{\max} \quad \forall j \in [1, n] \tag{3.11}$$
$$pos_{pf_i^k} < pos_{df_i^k} \wedge pos_{pe_i^k} < pos_{de_i^k} \quad \forall i \in [1, m]\ \forall k \in [1, n_i] \tag{3.12}$$
$$pos_{df_i^k} < pos_{pe_i^k} \vee pos_{de_i^k} < pos_{pf_i^k} \quad \forall i \in [1, m]\ \forall k \in [1, n_i] \tag{3.13}$$
$$pos_a = \text{INDEXOF}(\boldsymbol{seq}, a) \quad \forall a \in [1, n] \tag{3.14}$$
$$\text{COUNT}(\boldsymbol{seq}) = n \tag{3.15}$$

Les contraintes (3.6) et (3.7) correspondent à l'initialisation des compteurs de temps et de la longueur du train, respectivement. La contrainte (3.8) assure l'accumulation des durées des opérations et des durées de trajet tandis que la contrainte (3.9) représente la longueur du train à chaque étape. La modélisation dans `LocalSolver` nous permet d'utiliser une variable comme indice de tableau. Les contraintes (3.10) et (3.11) permettent de respecter les date d'échéance et la longueur maximale du train. S'ajoutent ensuite la contrainte (3.12) qui assure les précédences entre collectes et livraisons, et la contrainte (3.13) qui définit l'ordre des opérations au sein d'un même cycle de production. La contrainte (3.14) permet de relier les variables $\boldsymbol{seq}$ et pos_a : $pos_a = j \Leftrightarrow seq_j = a$. Enfin, la dernière contrainte (3.15) est similaire à la contrainte globale ALLDIFFERENT [107] et assure que les variables $\boldsymbol{seq}$ sont une permutation.

Un modèle `LocalSolver` nécessite une fonction objectif à minimiser. Le modèle initial comprend alors la minimisation du retard maximal avec la relaxation de la contrainte 3.10. Cependant, des expériences préliminaires ont montré que, de manière surprenante, le modèle en satisfaction, avec un objectif fictif, obtenait de meilleurs résultats, c'est donc cette version que nous retenons ici. Comme nous le montrerons dans la section 3.4 ce modèle est peu efficace pour résoudre les instances de l'entreprise.

3.3 Modèles de programmation par contraintes

Deux modèles de programmation par contraintes ont été conçus pour résoudre le problème de tournées de chariots et sont présentés dans cette section. Le premier modèle est une variante d'un modèle d'ordonnancement à une ressource, et le second modèle est une variante d'un modèle de voyageur de commerce. Une évaluation comparative de ces différents modèles sera exposée dans la section suivante.

3.3.1 Modèle d'ordonnancement

Pour le problème considéré, l'importance des contraintes temporelles et l'absence d'objectif à optimiser suggère qu'un modèle de programmation par contraintes orienté ordonnancement pourrait être efficace [13]. Le problème de tournées de chariots peut être représenté par un problème d'ordonnancement à ressource disjonctive unique (l'opérateur) devant réaliser quatre types de tâches (les opérations de collecte et livraison de chariots vides et pleins).

Soit $\mathcal{A}$ l'ensemble des n opérations, deux ensembles de variables sont définies :

- les variables $start_a \in [r_a, d_a - p_a]$ représentent la date de début de chaque opération $a \in \mathcal{A}$;
- les variables booléennes $x_{ab} \in \{0,1\}$ indique l'ordre relatif de chaque paire d'opérations $\{a,b\} \subset \mathcal{A}$.

Le modèle de programmation par contraintes proposé est alors le suivant :

$$x_{ab} = \begin{cases} 1 \Leftrightarrow start_b \geq start_a + p_a + D_{a,b} \\ 0 \Leftrightarrow start_a \geq start_b + p_b + D_{b,a} \end{cases} \qquad \forall \{a,b\} \subset \mathcal{A} \qquad (3.16)$$

$$x_{df_i^k pe_i^k} \vee x_{de_i^k pf_i^k} \qquad \forall i \in [1,m]\ \forall k \in [1,n_i] \qquad (3.17)$$

$$x_{pf_i^k df_i^k} = 1 \wedge x_{pe_i^k de_i^k} = 1 \qquad \forall i \in [1,m]\ \forall k \in [1,n_i] \qquad (3.18)$$

$$\textsc{Reservoir}(\boldsymbol{x}, \boldsymbol{start}, T_{\max}) \qquad (3.19)$$

Les deux ensembles de variables sont liés par la contrainte (3.16). La contrainte (3.17) représente les deux ordres possibles pour les quatre opérations d'un même cycle et la contrainte (3.18) marque la précedence entre une opération de collecte et l'opération de livraison associée. Enfin, la limite maximale sur la longueur du train est exprimée par la contrainte globale Reservoir (3.19) sur l'ensemble des variables $\boldsymbol{x}$ et $\boldsymbol{start}$. En effet, la limite sur la longueur peut être vue comme une ressource de type réservoir à capacité limitée, qui est remplie par les opérations de collecte, et vidée par les opérations de livraison, la quantité de ressource associée à chaque opération étant la longueur du chariot concerné.

Pour cette contrainte globale, une version légère du propagateur proposé dans [104] a été développée. Seul l'algorithme qui permet de trouver des bornes sur les dates de début des tâches grâce au graphe de précédence impliqué par les variables booléennes et par les contraintes de précédences issues du problème a été implémenté. La détection de conflits, grâce au calcul de la borne inférieure sur l'occupation de la ressource (la longueur du train) avant chaque opération, est également utilisé.

En pratique, le nombre de variables booléennes est bien plus petit que n^2. En effet, les fenêtres temporelles des cycles de production imposent un grand nombre de précédences qui s'ajoutent à celles directement imposées par la contrainte (3.18). Ces précédences peuvent être alors ignorées ou directement postées en ignorant la

variable booléenne associée : si a précède b et $r_b < d_a + p_a + D_{a,b}$, alors on poste la contrainte $start_b \geq start_a + p_a + D_{a,b}$. De plus, grâce à la contrainte 3.3, on remarque qu'une seule variable booléenne est nécessaire pour les quatre opérations d'un même cycle, il suffit d'ordonner les deux opérations de collecte pour obtenir l'ordre relatif des quatre opérations du cycle. On peut donc modifier la contrainte 3.16 en conséquence pour les paires d'opérations de collecte de même cycle a et b, de manière à ce que chaque valeur de la variable x_{ab} soit associée à plusieurs équivalence au lieu d'une seule. Ces deux remarques permettent alors de se passer d'un grand nombre de variables booléennes.

3.3.2 Modèle de voyageur de commerce

Le second modèle proposé se base sur le modèle de voyageur de commerce avec fenêtres de temps présenté dans [52]. Pour cela, nous avons besoin de deux opérations fictives supplémentaires, 0 et $n+1$, représentant le départ et l'arrivée de la tournée. Ces deux opérations fictives ont une durée nulle, n'utilisent pas de ressource et ont une durée de trajet nulle par rapport à toutes les autres opérations.

On note $\mathcal{A}^+ = \mathcal{A} \cup \{0, n+1\}$ l'ensemble des opérations, étendu par les deux opérations fictives. Les variables considérées dans ce modèle sont d'une part celles utilisées dans le modèle précédent : $start_a$ (dates de début des opérations) et x_{ab} (ordre relatif entre paire d'opérations). D'autre part, nous considérons les variables suivantes :

- des variables $next_a \in \mathcal{A}^+$ qui indiquent quelle opération va directement suivre l'opération a, pour toute opération $a \in \mathcal{A}^+$;
- des variables $train_a \in [0, T_{\max}]$ indiquant la longueur du train avant l'exécution de l'opération a, pour toute opération $a \in \mathcal{A}^+$;
- des variables $pos_a \in \mathcal{A}^+$ donnant pour chaque opération sa position dans la tournée, pour toute opération $a \in \mathcal{A}^+$.

Le modèle de programmation par contraintes proposé est le suivant :

$$next_{n+1} = 0 \tag{3.20}$$
$$start_0 = 0 \tag{3.21}$$
$$train_0 = 0 \tag{3.22}$$
$$pos_0 = 0 \land pos_{n+1} = n + 1 \tag{3.23}$$
$$start_{next_a} \geq start_a + p_a + D_{a,next_a} \qquad \forall a \in \mathcal{A} \cup \{0\} \tag{3.24}$$
$$train_{next_a} = train_a + t_a \qquad \forall a \in \mathcal{A} \cup \{0\} \tag{3.25}$$
$$\textsc{Circuit}(next) \tag{3.26}$$
$$pos_b > pos_a + 1 \implies next_a \neq b \qquad \forall \{a, b\} \subset \mathcal{A}^+ \tag{3.27}$$
$$\textsc{AllDifferent}(pos) \tag{3.28}$$
$$x_{ab} = \begin{cases} 1 \implies next_b \neq a \\ 0 \implies next_a \neq b \end{cases} \qquad \forall \{a, b\} \subset \mathcal{A}^+ \tag{3.29}$$
$$x_{ab} = \begin{cases} 1 \Leftrightarrow pos_b > pos_a \\ 0 \Leftrightarrow pos_a > pos_b \end{cases} \qquad \forall \{a, b\} \subset \mathcal{A}^+ \tag{3.30}$$
$$pos_a = \sum_{0 \leq b < a} x_{ba} + \sum_{a < b \leq n} 1 - x_{ab} \qquad \forall a \in \mathcal{A}^+ \tag{3.31}$$

Les contraintes (3.20), (3.21), (3.22) et (3.23) sont des initialisations pour la construction d'une tournée et la longueur du train. En particulier, la contrainte (3.20) permet d'encoder la réalisation d'un circuit complet.

La contrainte (3.24) garantit l'accumulation des durées des opérations et des durées des trajets durant le tour. La contrainte (3.25) permet de représenter la longueur du train tout au long de la séquence. Ces deux contraintes font appel à la contrainte globale Element [75].

De plus, la contrainte globale Circuit (3.26) [107] force les variables *next* à former un circuit Hamiltonien. Contrairement au modèle de voyageur de commerce présenté dans [52], le modèle proposé pour le problème de tournées de chariots n'utilise pas la contrainte Weighted Circuit. En effet, la propagation de cette contrainte s'appuie sur la qualité de la borne supérieure sur la longueur totale du tour. Dans notre problème, cette borne n'est pas aussi forte que dans le problème du voyageur de commerce où elle correspond à l'objectif.

Enfin, les contraintes restantes (3.27, 3.28, 3.29, 3.30 et 3.31) font le lien entre les différentes variables et sont des contraintes redondantes qui servent à améliorer le filtrage.

Notons que pour les même raisons évoquées pour le premier modèle, nous n'avons besoin que d'un nombre de variables booléennes bien inférieur à n^2. Il faut pour cela poster directement les implications des contraintes 3.29 et 3.30, et considérer des constantes (de valeur 1 ou 0 selon la précédence) pour la contrainte 3.31. En ce qui concerne l'ensemble des variables booléennes qui ordonnent les opérations d'un même cycle, là encore il faut modifier ces mêmes contraintes (plusieurs équi-

valences ou implications pour les contraintes 3.29 et 3.30 et double prise en compte de la variable booléenne concernée dans 3.31).

3.3.3 Stratégie de branchement

Pour chacun des modèles de programmation par contraintes la stratégie de branchement est un arbre binaire, la branche gauche correspond à l'affectation d'une variable à une valeur, et la branche droite correspond à sa réfutation. Pour les deux modèles, des expériences préliminaires ont montré qu'une stratégie de recherche dédiée est la clé pour résoudre ce problème. En raison de la structure des précédences issues des cycles de production du problème, il apparaît que construire une solution dans l'ordre des opérations à effectuer est efficace.

Pour le modèle de type voyageur de commerce, l'ordre des variables consiste à sélectionner les variables $next$, en commençant par $next_0$, puis $next_{next_{j-1}}$ à profondeur $j > 1$ de l'arbre de recherche.

Pour le modèle de type ordonnancement, seules les variables booléennes sont des variables de décision. Ainsi, il convient de fixer les disjonctions entre paires d'opérations de manière cohérente avec les cycles de production des opérations, c'est à dire les disjonctions d'une opération d'un composant i du cycle k seront décidées seulement lorsque les disjonctions des opérations des cycles précédents $k' < k$ du même composant auront été décidées. Également, parmi cet ensemble de disjonctions candidates, on va vouloir décider celles dont les dates de début au plus tôt des opérations concernées sont les plus petites possibles. Concrètement, à chaque étape on calcule l'ensemble des paires d'opérations $\mathcal{C}$ telles que pour toute paire $a, b \in \mathcal{C}$, $|\mathcal{D}(x_{ab})| > 1$ et $(\min(\mathcal{D}(start_a)), \min(\mathcal{D}(start_b)))$ est Pareto minimal. La prochaine variable x_{ab} est choisie aléatoirement dans cet ensemble.

En ce qui concerne le choix de la valeur, pour le modèle de type voyageur de commerce, on choisit dans le domaine de la variable l'opération a ayant la plus petite date de début au plus tôt $\min(\mathcal{D}(start_a))$. Pour le modèle de type ordonnancement, on choisit la valeur de la variable booléenne pour que l'opération ayant la plus petite date de début au plus tôt précède la seconde opération.

Ces deux stratégies de branchement agissent alors comme une heuristique de "plus proche voisin", en ajoutant la prise en compte des date de disponibilité des opérations.

3.4 Évaluation expérimentale des modèles de programmation par contraintes

3.4.1 Jeux de données

Nous disposons pour ce problème d'un jeu de données industrielles et d'un jeu de données générées aléatoirement. Le jeu de données industrielles correspond à trois ensembles distincts de composants que l'on nomme S, L et R (correspondant initialement à des opérateurs à qui sont affectés ces ensembles de pièces). En pratique, ces

jeux de données industrielles sont relativement sous contraints, le processus d'affectation des ensembles de composants aux opérateurs n'étant pas optimisé. Les différences entre ces trois ensembles de composants sont leur cardinalité, et la synchronicité des différents cycles de production. Par exemple, dans l'ensemble S, il n'y a que 5 types de composants, et leurs cycles de production sont presque tous semblables et plutôt courts. Les deux autres ensembles sont composés de plus de 30 composants avec des cycles de production très variés : certains sont courts d'autres très longs. Pour chaque ensemble S, L et R, on forme trois instances en faisant varier l'horizon temporel considéré : le premier horizon correspond à un quart de travail d'un opérateur (7h15), le second horizon correspond à une journée de 3 quarts et le troisième horizon correspond à une semaine de 6 jours. Ces trois horizons valent respectivement 43 500, 130 500 et 783 000 *centièmes de minutes*. On dispose ainsi de 9 instances contenant de 400 à plus de 10 000 opérations.

Le second jeu de données a été généré aléatoirement [1]. Pour se rapprocher des données industrielles, on reprend le même procédé d'organisation que dans celles-ci. Les composants y sont rassemblés en sous-ensembles que l'on appelle une gamme, et à cette gamme est associée une certaine quantité demandée quotidiennement. Tous les composants d'une gamme vont alors être produits dans les mêmes quantités, et donc avoir avoir des cycles de production proches, dépendant de la capacité du chariot qui leur est associé. Dans les données générées, on reproduit ce schéma. Une instance est composée d'un certain nombre de gammes (de 3 à 7), chacune associée à un nombre aléatoire de composants. Pour chacun des composants d'une même gamme, la quantité journalière produite est la même, mais les chariots sont différents (capacité et longueur). Les cycles de production de chaque composant sont obtenus en divisant la durée d'une journée par le nombre de trajets requis pour déplacer la quantité produite (dépendant donc de la capacité du chariot). On obtient alors des sous-ensembles de composants avec les mêmes temps de cycles de production. Pour désynchroniser ces temps de cycle au sein d'une même gamme, pour chaque composant pris séparément, on va perturber aléatoirement la capacité des chariots en ajoutant (en plus ou en moins) un petit décalage. Cette désynchronisation est appliquée à la moitié des ensembles de composants.

Enfin, chaque poste de production et de consommation a été positionné aléatoirement dans un plan et les distances ont été calculées à partir de ce plan, puis arrondies pour garder des nombres entiers (en vérifiant l'inégalité triangulaire). Pour se rapprocher des instances industrielles, le plan est divisé en zones (entre 6 et 10 ou 1 zone unique selon les ensembles de composants), et le placement des postes de production et consommation est alors restreint à ces zones. Cela forme alors des *clusters* de machines.

La satisfiabilité des instances générées aléatoirement a été montrée pour le plus petit horizon temporel possible, correspondant à la durée du plus long cycle de production de l'ensemble de composants, qui est d'à peu près 10 000 unité de temps en fonction des instances. Ces résultats ont été obtenus avec le modèle de program-

1. Ces instances sont disponibles sur `https://gitlab.laas.fr/vantuori/trolley-pb`.

mation par contraintes d'ordonnancement. Cependant, en raison des fenêtres de temps, il est difficile de garantir la satisfiabilité de ces instances sur de plus grands horizons temporels.

Ce travail préliminaire pour la génération d'instances nous a permis de sélectionner 40 ensembles de composants aux caractéristiques diverses. Pour chaque ensemble, on considère les trois mêmes horizons temporels (*quart*, *jour* et *semaine*) que pour les données industrielles. Il y a donc un total de 120 instances pour ce jeu de données.

Ces instances ont été classées en 4 catégories selon le nombre de composants qui compose l'instance. Ces 4 classes sont notées A, B C et D avec 15 composants pour la catégorie A, 20 composants pour la catégorie B, 25 composants pour la catégorie C et 30 composants pour la catégorie D. Le nombre moyen de gammes par instance est de 2.4 pour la catégorie A, 4.4 pour la catégorie B, 5.9 pour la catégorie C et 6.6 pour la catégorie D. Enfin, le nombre de zones est également réparti dans l'ensemble des catégories. La catégorie A est donc la plus facile, moins de composants veut dire moins d'opérations à faire en même temps, avec des opérations plus synchronisées. La catégorie D à l'inverse est la plus difficile.

3.4.2 Configuration

Les modèles de programmation par contraintes ont été implémentés avec le solveur Choco 4.10 [142], et la version de `LocalSolver` utilisée est la 9.0.

Pour les instances industrielles, les expérimentations ont été menées sur un Intel(R) Core(TM) i7-8700 CPU @ 3.20GHz3.

Pour les instances aléatoires, au vu de leur nombre, le cluster de calcul du laboratoire a été utilisé, il est composé de Intel(R) Xeon E5-2695 v3 @ 2.30GHz et Intel(R) Xeon E5-2695 v4 @ 2.10GHz. Nous avons lancé chaque instance 10 fois pour chacun des modèles de programmation par contraintes, en ajoutant une stratégie de redémarrage basée sur une séquence de *Luby* [116] avec une base de 500. Nous avons également ajouté de l'aléatoire pour le modèle de voyageur de commerce (celui d'ordonnancement contenant déjà une part d'aléatoire dans le choix des variables), en choisissant aléatoirement la prochaine opération parmi les deux opérations ayant les dates de début au plus tôt les plus petites. Le temps limite de résolution est fixé à 1h par instance et par méthode.

A cause d'un problème de licence nous n'avons pas pu lancer les expérimentations avec `LocalSolver` pour les instances aléatoires. Cependant, pour les tests partiels que nous avons pu effectuer pour ces instances aléatoires, les résultats de `LocalSolver` étaient similaires à ceux obtenus sur les instances industrielles.

3.4.3 Résultats expérimentaux

La table 3.2 donne les résultats des deux modèles de programmation par contraintes (PPC) proposés ainsi que du modèle basé sur `LocalSolver` sur les instances industrielles. La première colonne (Cl) indique l'ensemble de composants concernés (S, L

ou R) et la seconde colonne l'horizon temporel (H). Pour évaluer les deux modèles de programmation par contraintes, deux indicateurs sont considérés : le temps CPU (en secondes) pour résoudre l'instance et le nombre de conflits dans la recherche arborescente (*fail*) du solveur. Pour `LocalSolver`, seul le temps CPU est donné (lorsqu'une solution satisfiable est obtenue).

Table 3.2 – Comparaison des modèles pour les instances industrielles

Cl	H	`Ordo`		`TSP`		`LocalSolver`
		CPU	#conflit	CPU	#conflit	CPU
S	`Quart`	0.275	0	3.999	0	26
S	`Jour`	0.840	0	48.855	0	566
S	Semaine	17.747	0	memory out		timeout
L	`Quart`	7.129	0	36.033	8	121
L	`Jour`	23.468	0	344.338	8	3539
L	`Semaine`	318.008	6	memory out		timeout
R	`Quart`	8.397	0	41.615	14	1065
R	`Jour`	44.215	0	417.116	15	timeout
R	`Semaine`	timeout	773738	memory out		timeout

Dans la table 3.2, on observe que le modèle PPC d'ordonnancement est le plus rapide des trois modèles évalués. Le modèle `LocalSolver` est celui qui nécessite le plus de temps CPU, et, dans le temps limite imparti, il ne permet pas de résoudre les instances de plus grande taille (horizon d'une *semaine* pour S, L et R, horizon d'un *jour* pour R). Le modèle PPC de voyageur de commerce comporte un nombre important de variables et de contraintes, et prend trop de place en mémoire pour les horizons les plus longs (limite fixée à 16Go pour les instances concernées). Enfin, pour le modèle PPC de voyageur de commerce, les instances résolues avec un faible nombre de conflits, montrent, que la première branche de l'arbre de recherche peut être très lente à explorer en raison des propagations effectuées. Le modèle PPC d'ordonnancement résout toutes les instances industrielles sans conflit à l'exception des instances L et R sur l'horizon d'une *semaine*.

L'obtention de solutions sans conflit pour le modèle PPC d'ordonnancement, montrent que ces instances sont assez peu contraintes. Néanmoins, ces instances n'en demeurent pas triviales. En effet, des expérimentations préliminaires utilisant des heuristiques génériques telles que *Weighted Degree* [26] sur le même modèle PPC d'ordonnancement n'ont permis de résoudre que 3 instances dans le temps limite. Ainsi, pour le problème étudié, même sur des instances sous contraintes, de mauvais choix intervenant tôt dans l'arbre de recherche peuvent mener à des sous-arbres infaisables dont il est difficile de sortir. En particulier, le nombre de conflits pour l'instance R sur l'horizon d'une *semaine* montre cette situation (nombre important de conflits, pas de solution obtenue au bout du temps limite fixé). Le facteur clé

TABLE 3.3 – Comparaison des modèles pour les instances aléatoires

Cl	H	Ordo			TSP		
		#S	CPU	#conflit	#S	CPU	#conflit
A	Quart	7.1	418	300K	4.0	56	366
	Jour	4.0	29	213	3.6	802	1267
	Semaine	3.1	866	1976	0.0	mem. out	
B	Quart	2.1	389	150K	0.9	844	11K
	Jour	1.0	201	15K	0.0	–	
	Semaine	0.0	–		0.0	mem. out	
C	Quart	0.0	–		0.0	–	
	Jour	0.0	–		0.0	–	
	Semaine	0.0	–		0.0	mem. out	
D	Quart	0.0	–		0.0	–	
	Jour	0.0	–		0.0	–	
	Semaine	0.0	–		0.0	mem. out	
Total		14.4%			7.1%		

pour la résolution de ces instances est donc de s'appuyer sur une stratégie de branchement efficace pour que l'ordre d'instanciation des variables suive l'ordre de la séquence d'opérations en cours de construction. En effet, à cause de la structure temporelle du problème, la propagation permet de filtrer seulement les dates de début au plus tôt des opérations temporellement proches de la décision que l'on vient de prendre.

La table 3.3 présente les résultats obtenus par les modèles de programmation par contraintes pour les instances aléatoires. On rappelle qu'il y a 10 instances pour chaque catégorie (A, B, C, D) et pour chaque horizon temporel (*quart*, *jour*, *semaine*). Les indicateurs reportés sont le temps CPU, le nombre de conflits pour les instances qui ont été résolues, moyenné sur les 10 instances lancées 10 fois chacune, et le nombre moyen d'instances résolues (#S). La dernière ligne de la table indique le pourcentage d'instances résolues sur les 1200 exécutions de la méthode correspondante (10 exécutions pour chacune des 120 instances.

Comme pour les instances industrielles, le modèle PPC d'ordonnancement domine le modèle PPC de voyageur de commerce qui nécessite une trop grande place en mémoire pour les instances de plus grande taille. Cependant, les instances aléatoires sont plus contraintes comme en témoigne le faible nombre d'instances résolues par ces modèles ("-" dans la table indique un dépassement du temps de calcul d'une heure). On peut relever également, de manière moins flagrante le rapport temps CPU/nombre de conflits qui indique que descendre une branche peut être coûteux (par exemple, 7 conflits/seconde pour les instance A-*jour* pour le modèle d'ordonnancement, et 7 conflits/seconde également pour les instances A-*quart* pour le

modèle de voyageur de commerce).

Nous retirons deux conclusions de ces expérimentations : premièrement les modèles PPC ne peuvent pas résoudre de manière fiable des instances difficiles. Renforcer la propagation ne s'avère pas pertinent, comme on peut le voir sur le modèle de voyageur de commerce. Ce modèle (qui fait plus de filtrage) a tendance à être moins efficace, et ne passe pas à l'échelle. Deuxièmement, construire la séquence d'opérations de manière “chronologique" permet d'obtenir de bons résultats ; ceci explique pourquoi les modèles de PPC surpasse la recherche locale de `LocalSolver`. Ainsi, les plongées gloutonnes du solveur PPC peuvent parfois obtenir une solution sans conflit. L'obtention d'heuristiques performantes et de bonnes stratégies pour les exploiter, semblent alors être primordial pour résoudre ce problème de tournées de chariots. C'est pourquoi on se propose d'utiliser la méthode décrite dans le chapitre 2 pour obtenir une telle heuristique.

3.5 Apprentissage par renforcement d'heuristiques de choix de valeurs

Nous utilisons dans cette partie le cadre décrit dans le chapitre 2 pour obtenir l'heuristique désirée en s'appuyant sur l'algorithme REINFORCE. Cette heuristique est ensuite intégrée dans le modèle de programmation par contraintes d'ordonnancement présenté précédemment (section 3.3.1) ainsi que dans deux approches supplémentaires : une recherche à divergences limitées et une recherche locale à démarrages multiples.

3.5.1 Processus de décision markovien

Nous considérons la version en minimisation du problème de tournées de chariots où l'objectif est de minimiser le retard maximal $L(\sigma)$ décrit en fin de section 3.1. Pour rappel, afin mettre en place un processus d'apprentissage par renforcement, nous avons besoin d'exprimer la recherche d'une séquence d'opérations qui minimise le retard maximal comme un processus de décision Markovien.

On définit les variables seq, tel que $seq_j = a$ signifie que la tache a est effectuée à la position j. L'ordre choisi pour instancier ces variables est l'ordre lexicographique.

Le MDP est alors le suivant :

- Un état est une instanciation $I = ((seq_1, a_1), ..., (seq_k, a_1k))$ sur les variables seq. Il représente une séquence d'opérations. On note alors un état par σ et par raccourci la j-ème opération de la séquence se note $\sigma(j)$. Dans la suite, on parle d'état ou de séquence de manière interchangeable. Tout état final est une solution réalisable (ie. toutes les opérations ont été placées dans la séquence).
- Les actions $A(\sigma)$ sont les opérations qui peuvent étendre σ tout en respectant les contraintes de précédence (3.2), de capacité (3.4), d'ordre dans les cycles

de production (3.3) ainsi que les précédences entre les cycles de production pour un même composant (3.5).

— En appliquant l'action a dans l'état σ, l'état courant devient alors σ' qui correspond à la même séquence d'opération σ étendue de l'opération a.

— On peut évaluer la valeur de la fonction objectif pour une séquence donnée par le retard maximal de cette séquence $L(\sigma)$. On définit la récompense de la transition de l'état σ à l'état σ' en prenant l'action a, par l'accroissement marginal de cette borne inférieure : $R(s,a) = L(\sigma') - L(\sigma)$.

Dans ce chapitre, on utilise alors une pénalité à minimiser plutôt qu'une récompense a maximiser comme nous l'avons vu dans le chapitre 2. Pour l'algorithme REINFORCE on effectue alors une descente de gradient plutôt qu'une ascension remplaçant alors la mise à jour de $\boldsymbol{\theta}$ par une différence au lieu d'une somme dans l'algorithme 5.

3.5.2 Caractérisation de l'heuristique de branchement

La politique recherchée consiste alors à évaluer chacune des actions pouvant étendre la séquence courante, de manière à minimiser la somme des pénalités sur le long terme, ce qui correspond à minimiser le retard maximal. Comme décrit dans le chapitre 2, on s'appuie pour cela sur un ensemble de critères pour évaluer une action. Nous proposons ici quatre critères.

Soit l'état courant σ et soit l'action $a \in A(\sigma)$ avec $a \in A(\sigma)$. On note $lst(a,\sigma)$ la date de début au plus tard de l'opération a tel que les quatre opérations du cycle de a respectent leur date d'échéance. Cette date est facile à calculer et est fonction du nombre de tâches restantes à faire dans le cycle, ainsi que des temps de trajets entre point de collecte, et point de livraison.

Pour chaque état σ, et pour chaque opération $a \in A(\sigma)$, on considère les critères suivants :

$$\lambda_1(\sigma,a) = (lst(a,\sigma) - \max(r_a, e(\sigma,|\sigma|) - L(\sigma) + D_{\sigma(|\sigma|),a}))/\max_{b\in\mathcal{A}}\{d_b - r_b\} \quad (3.32)$$

$$\lambda_2(\sigma,a) = \max(r_a - (e(\sigma,|\sigma|) - L(\sigma)), D_{\sigma(|\sigma|),a})/\max_{\{b,c\}\in\mathcal{A}}\{D_{b,c}\} \quad (3.33)$$

$$\lambda_3(\sigma,a) = 1 - |t_a|/T_{\max} \quad (3.34)$$

$$\lambda_4(\sigma,a) = \begin{cases} 1 & \text{si } a \in P \\ 0 & \text{sinon} \end{cases} \quad (3.35)$$

Pour chacun des quatre critères, une valeur haute signifie que l'action n'est pas désirable du point de vue de ce critère, et ils ont été normalisés dans $[0,1]$.

Le critère $\lambda_1(\sigma,a)$ de l'équation 3.32 représente l'urgence de l'opération, c'est à dire la distance temporelle à sa date d'échéance si elle était sélectionnée. Le critère $\lambda_2(\sigma,a)$, fourni par l'équation 3.33, est la distance temporelle à la dernière tâche de la séquence. Pour ces deux critères, on utilise $e(\sigma,|\sigma|) - L(\sigma)$ à la place de $e(\sigma,|\sigma|)$ pour représenter la date de fin de la dernière opération dans la séquence afin d'ignorer l'impact des choix précédents sur le retard actuel de la séquence

courante. Le critère $\lambda_3(\sigma, a)$, de l'équation 3.34, est la longueur du chariot. En effet, étant donné que toutes les opérations doivent être faites, faire les opérations qui occupent le plus la ressource le plus tôt possible laisse plus de liberté pour la suite. Enfin, le critère $\lambda_4(\sigma, a)$, donné par l'équation 3.35, pénalise les opérations de collecte, étant donné qu'on veut éviter de laisser un chariot dans le train trop longtemps (et donc favoriser les opérations de livraison).

La fonction d'évaluation est une combinaison linéaire de ces quatre critères $g_{\boldsymbol{\theta}}(\sigma, a) = \boldsymbol{\theta}^\intercal \boldsymbol{\lambda}(\sigma, a)$. Comme décrit dans la section 2.1, les poids de cette combinaison linéaire sont déterminés avec un algorithme d'apprentissage par renforcement (REINFORCE). On utilise ensuite la fonction `softmax` pour transformer la fonction d'évaluation en distribution de probabilités pour obtenir la politique stochastique suivante :

$$\forall a \in A(\sigma) \pi_{\boldsymbol{\theta}}(a \mid \sigma) = \frac{e^{(1-g_{\boldsymbol{\theta}}(\sigma,a))/\beta}}{\sum_{b \in A(\sigma)} e^{(1-g_{\boldsymbol{\theta}}(\sigma,b))/\beta}} \tag{3.36}$$

On rappelle que le paramètre de température $\beta > 0$ permet de contrôler la distribution, plus β est proche de 0 et plus la politique tend vers une politique déterministe. Lorsque $\boldsymbol{\theta}$ est normalisé, alors $g_{\boldsymbol{\theta}}(\sigma, a)$ retourne une valeur entre 0 et 1, et contrairement à ce qui était décrit dans le chapitre 2, une valeur basse signifie une action désirable, et une valeur haute signifie une action indésirable. On inverse alors ce comportement dans la fonction `softmax` en prenant le complément à 1 de la fonction, modifiant donc les signes des deux termes de la différence dans le calcul du gradient (équation 2.3).

3.5.3 Intégration de l'heuristique apprise dans des méthodes arborescentes

La politique déterminée par apprentissage par renforcement, est intégrée comme heuristique de choix de valeurs au sein de deux méthodes arborescentes basées sur le même modèle de programmation par contraintes de type ordonnancement présenté en section 3.3 : une recherche arborescente avec retour arrière et redémarrages et une recherche arborescente à divergences limitées (*Limited Discrepancy Search*). Pour ces méthodes, la contrainte sur les dates d'échéances (eq. 3.1) n'est pas relâchée, et elles produisent alors des solutions sans retard.

Adaptation du modèle PPC Pour utiliser la politique stochastique apprise, nous avons besoin d'introduire un nouvel ensemble de variables : seq_j indique quelle est l'opération réalisée à la position j, pour tout $j \in \{1, ..., n\}$. On relie cette variable aux autres de la façon suivante :

$$x_{seq_j seq_l} = 1 \ \forall j < l \in [1, n] \tag{3.37}$$

Au cours de la recherche arborescente, le branchement s'effectue sur les variables

seq dans l'ordre lexicographique, la propagation de cette contrainte et le filtrage de cette variable peut ainsi s'appliquer avec une forme légère de propagation (*forward checking*) :

— Lorsque l'on branche sur $seq_j = a$, comme toute les variables seq_l $\forall l < j$ sont déjà affectées, on peut établir $x_{ab} = 1$ $\forall b \in \mathcal{A} \backslash \{seq_l \mid l < j\}$.

— Après l'affectation $seq_j = a$, on peut enlever la valeur a du domaine des variables seq_{j+1} si il existe une opération $b \in \mathcal{A} \backslash \{seq_l \mid l \leq j\}$ tel que le domaine de la variable x_{ab} est réduit à $\{0\}$ (si elle existe).

— Lorsque la variable seq_j est instanciée, on peut restreindre le domaine de seq_{j+1} aux seules opérations qui peuvent étendre la séquence $\{seq_1, ..., seq_j\}$ au vu des contraintes de précédences (3.2, 3.5, 3.3) et de la contrainte sur la longueur du train (3.4).

En conséquence, nous n'avons plus besoin du propagateur de la contrainte de ressource `Réservoir` (eq. 3.19).

Stratégies d'exploration Pour explorer l'arbre de recherche, deux stratégies basées sur l'heuristique apprise par apprentissage par renforcement sont proposées :

1. *Redémarrages rapides.* On choisit la prochaine opération aléatoirement en suivant la distribution de probabilités donnée par la politique `softmax` (3.36). Dans le but d'explorer rapidement différentes parties de l'arbre de recherche, des redémarrages rapides sont utilisés, en suivant une séquence de *Luby* [116] avec une base de 15.
2. *Recherche à divergences limitées.* Étant donné que l'on cherche à s'appuyer le plus possible sur l'heuristique et d'en dévier le moins possible, la recherche à divergences limitées [72] semble être une bonne stratégie d'exploration de l'arbre de recherche. Pour cela, nous considérons une version déterministe de la politique : $\pi(\sigma) = \arg\min_a f(\sigma, a)$. On utilise une version itérative de cette méthode à divergences : le nombre de divergences démarre à 0, puis il est incrémenté jusqu'à arriver à une valeur maximale donnée en paramètre (réglé à 20 dans les expérimentations, ce maximum n'étant jamais atteint dans la limite de temps fixée).

3.5.4 Intégration de l'heuristique apprise dans une méthode de recherche locale

La politique issue de l'apprentissage par renforcement produit généralement une solution comportant des retards. Une méthode de recherche locale peut alors conduire à améliorer cette solution afin de réduire ce retard ou d'obtenir un retard nul.

Nous proposons une méthode de recherche locale à démarrages multiples dans laquelle la génération de solution initiale se fait avec l'heuristique basée sur la politique stochastique avant d'être améliorée par une plus profonde descente. Les solutions initiales générées respectent toutes les contraintes sauf les dates d'échéances

(3.1), et les voisinages considérés préservent également ces contraintes. Dans le voisinage d'une solution, on applique le mouvement local qui fait décroître le plus le retard maximal $L(\sigma)$, jusqu'à ce qu'un tel mouvement ne puisse plus être trouvé. Deux voisinages sont proposés, et la complexité d'une itération de la méthode de recherche locale (calculer le voisinage et appliquer le meilleur mouvement) est en $O(nm)$.

Rappelons que $L(\sigma, j, l) = \max\{L(\sigma, q) \mid j \leq q \leq l\}$ est le retard maximal pour toutes les opérations comprises entre les positions j et l de la séquence σ.

Voisinage swap Le premier type de mouvement consiste à échanger deux opérations dans la séquence, c'est à dire à échanger les valeurs de $\sigma(j)$ et $\sigma(l)$ pour deux positions j et l.

La première condition à vérifier et de s'assurer que l'ordre des opérations au sein des opérations d'un même composant reste valide après le **swap**, c'est à dire qu'il respecte les contraintes (3.2), (3.5) et (3.3) : une opération de collecte (resp. de livraison) doit toujours être entre deux opérations de livraison (resp. de collecte) du même composant. Pour chaque opération a, une plage valide entre la position de son prédécesseur $pr(a)$ et celle de son successeur $su(a)$ peut être calculée en temps constant si l'on garde au préalable une association des opérations à leur position. Ainsi, pour chaque $j \in [1, n]$ on considère les échanges seulement entre les positions j and l pour $l \in [j + 1, su(\sigma(j))]$ et tel que $pr(\sigma(l)) \leq j$.

La seconde condition pour que ce mouvement soit valide, est que l'échange ne doit pas entraîner une violation de la contrainte (3.4), c'est à dire la longueur maximale du train de chariot. Soit $\tau_j = \sum_{q=1}^{j-1} t_{\sigma(q)}$ la longueur du train avant la j-ième opération. Après l'échange on a $\tau_{j+1} = \tau_j + t_{\sigma(l)}$ qui doit être inférieur à $T_{\max}$. À toute autre position jusqu'à l, la différence sera de $t_{\sigma(l)} - t_{\sigma(j)}$, et on doit vérifier la contrainte seulement à la position ou la longueur est maximale, c'est à dire : $\max\{\tau_q \mid j \leq q \leq l\} + t_{\sigma(l)} - t_{\sigma(j)} \leq T_{\max}$. Cela peut être fait en temps constant (amorti) pour chaque échange avec l'opération $\sigma(j)$ en calculant la longueur maximale du train de manière incrémentale pour chaque $l \in [j + 1, su(\sigma(j))]$.

Puis, on doit calculer la valeur du retard maximal de la nouvelle séquence σ' dans laquelle les opérations aux positions j et l ont été échangées, c'est à dire calculer le coût marginal de l'échange. Plusieurs parties sont à considérer : (1) le retard maximal avant la position j ne change pas ; (2) on doit calculer les nouveaux retards $L(\sigma', j)$ et $L(\sigma', l)$ aux position j et l respectivement ; (3) on doit calculer $L(\sigma', j + 1, l - 1)$, le nouveau retard maximal pour les opérations strictement entre j et l ainsi que (4) $L(\sigma', l + 1, n)$ le retard maximal pour les opérations strictement après l.

La nouvelle date de fin $e(\sigma', j)$ de l'opération $\sigma'(j) = \sigma(l)$ et donc le retard à cette position $L(\sigma', j) = e(\sigma', j) - d_{\sigma'(j)}$ peut se calculer en $O(1)$ de la manière suivante :

$$e(\sigma', j) = p_{\sigma'(j)} + \max(r_{\sigma'(j)}, (e(\sigma', j - 1) + D_{\sigma'(j-1),\sigma'(j)}))$$

Ensuite, les opérations $\sigma(j+1), \ldots, \sigma(l-1)$ restent dans le même ordre et $\sigma(j+1)$ est décalé d'une valeur $\Delta = e(\sigma', j) + D_{\sigma'(j),\sigma'(j+1)} - e(\sigma, j) - D_{\sigma(j),\sigma(j+1)}$. Cependant, les opérations ultérieures peuvent ne pas être toutes décalées dans le temps de la même manière.

En effet, lorsque $\Delta < 0$ il peut exister une opération dont la date de disponibilité empêche le décalage de Δ. Soit $g(\sigma, j) = r_{\sigma(j)} - s(\sigma, j)$ le *décalage à gauche maximal*(exprimé en valeur négative) pour la j-ième opération, et soit $g(\sigma, j, l) = \max\{g(\sigma, q) \mid j \leq q \leq l\}$.

Proposition 1 *Si la séquence ne change pas entre les positions j et l, un décalage temporel $\Delta < 0$ à la position j mène à un décalage temporel de* $\max(\Delta, g(\sigma, j, l))$ *à la position l.*

Soit $L_\Delta(\sigma, j, l)$ le retard maximal sur l'intervalle $[j, l]$ de la séquence σ décalée de Δ à partir de la position j. On peut définir $L_{-\infty}(\sigma, j, l)$ le retard maximal sur l'intervalle $[j, l]$ pour un décalage temporel négatif infini :

$$L_{-\infty}(\sigma, j, l) = \max\{L(\sigma, q, l) + g(\sigma, j, q) \mid j \leq q \leq l\} \tag{3.38}$$

L'équation 3.38 revient à appliquer le décalage maximale possible à chaque opération entre j et l et à retenir l'opération qui serait la plus en retard ensuite.

Proposition 2 *Si* $\Delta < 0$ *alors* $L_\Delta(\sigma, j, l) = \max(\Delta + L(\sigma, j, l), L_{-\infty}(\sigma, j, l))$.

Les valeurs de $L(\sigma, j, l)$, $g(\sigma, j, l)$ et $L_{-\infty}(\sigma, j, l)$ peuvent être calculées de manière incrémentale :

$$\begin{aligned} L(\sigma, j, l+1) &= \max(L(\sigma, j, l), L(\sigma, l+1)) \\ g(\sigma, j, l+1) &= \max(g(\sigma, j, l), g(\sigma, l+1)) \\ L_{-\infty}(\sigma, j, l+1) &= \max(L_{-\infty}(\sigma, j, l), g(\sigma, j, l+1) + L(\sigma, l+1)) \end{aligned}$$

Inversement, lorsque $\Delta > 0$ des décalages temporels peuvent être "absorbés" par le temps d'attente avant une opération. Cependant il y a généralement peu d'intérêt à déplacer une opération avant une position j avec un temps d'attente non négatif (c'est à dire quand $s(\sigma, j) = r_{\sigma(j)}$) étant donné que cette opération et toutes les suivantes ne pourront pas bénéficier d'une réduction d'un temps de trajet. De plus, le calcul du nouveau retard maximal après un tel échange ne semble pas pouvoir être amorti, ce qui augmenterait la complexité final d'une itération d'un facteur n. On considère alors seulement les échanges dont la première position j est telle que $\forall l > j,\ g(\sigma, l) < 0$. En conséquence, si le décalage temporel est positif $\Delta > 0$, alors $L_\Delta(\sigma, q, l) = \Delta + L(\sigma, q, l)$.

On peut donc, pour un échange entre les position j et l, lorsque $j < l-1$ calculer en temps constant (amorti), le nouveau retard $L(\sigma', j+1, l-1) = L_\Delta(\sigma, j+1, l-1)$ pour les opérations situées dans l'intervalle $[j+1, l-1]$.

Le nouveau retard $L(\sigma', l)$ à la position l est calculé de manière similaire à $L(\sigma', j)$ étant donné que l'on connaît la nouvelle date de début de $\sigma'(l-1)$ par les étapes précédentes.

Enfin, dans le but de calculer le nouveau retard maximal $L(\sigma', l+1, n)$ des opérations suivantes, on précalcule $L(\sigma, j, n)$, $g(\sigma, j, n)$ et $L_{-\infty}(\sigma, j, n)$ pour chaque position $1 \leq j \leq n$ une fois après chaque mouvement en $O(n)$. Puis, $L(\sigma', l+1, n)$ peut être obtenu en $O(1)$ pour chaque échange potentiel par la proposition 2.

Pour le voisinage **swap**, on peut vérifier la validité et calculer le coût marginal d'un échange en temps constant amorti, et appliquer l'échange en temps linéaire. La complexité d'une itération est donc en $O(nm)$ étant donné que pour un composant i, la somme des tailles des plages valides pour pour toutes les opérations de collecte et de livraison pour ce composant est de l'ordre de $O(n)$. En effet, soient $a_i^1, \ldots, a_i^{4n_i}$ les opérations pour le composant i ordonnées comme dans σ, alors $su(a_i^k) = \rho(a_i^{k+1})$ et $\sum_{k=1}^{4n_i} su(a_i^k) - \rho(a_i^k) = \rho(a_i^{4n_i}) - \rho(a_i^1) \in \Theta(n)$).

Voisinage toggle Comme indiqué dans la section 3.1, on impose une contrainte de précédence entre la première opération de livraison et la seconde opération de collecte pour un cycle d'un composant donné (eq. 3.3). On n'a alors plus que deux ordres possibles pour les quatre opérations du k-ième cycle de production du composant i : soit $pf_i^k < df_i^k < pe_i^k < de_i^k$ ou bien $pe_i^k < de_i^k < pf_i^k < df_i^k$. Le voisinage **toggle** consiste à passer de l'un à l'autre de ces deux ordres, en échangeant les valeurs de $\sigma(pf_i^k)$ et $\sigma(pe_i^k)$ ainsi que les valeurs de $\sigma(df_i^k)$ et $\sigma(de_i^k)$.

Soient $j_1 = pf_i^k, j_2 = df_i^k, j_3 = pe_i^k, j_4 = de_i^k$ les positions des quatre opérations du composant i et cycle k dans la solution courante, et sans perte de généralité, on suppose $j_1 < j_2 < j_3 < j_4$. Soit σ' la séquence obtenue par l'application du mouvement *toggle* sur le cycle k du composant i dans σ.

Pour calculer le coût marginal de ce mouvement, on doit calculer le nouveau retard aux positions des quatre opérations impliquées $L(\sigma', j_1)$, $L(\sigma', j_2)$, $L(\sigma', j_3)$ et $L(\sigma', j_4)$. On doit également calculer le nouveau retard maximal des quatre intervalles suivants :

$$\begin{aligned} L(\sigma', j_1+1, j_2-1) &= L_{\Delta_1}(\sigma, j_1+1, j_2-1) \\ L(\sigma', j_2+1, j_3-1) &= L_{\Delta_2}(\sigma, j_2+1, j_3-1) \\ L(\sigma', j_3+1, j_4-1) &= L_{\Delta_3}(\sigma, j_3+1, j_4-1) \\ L(\sigma', j_4+1, n) &= L_{\Delta_4}(\sigma, j_4+1, n) \end{aligned}$$

Calculer le coût marginal peut être fait via la même méthode que celle définie pour le voisinage *swap* : on peut d'abord calculer la nouvelle date de fin à la position j_1, puis calculer la valeur de Δ_1 à partir d'elle, qui peut ensuite être utilisée pour calculer $L_{\Delta_1}(\sigma, j_1+1, j_2-1)$ en $O(j_2 - j_1 - 1)$, etc. Cependant, la différence est qu'il y a moins de mouvements possibles ($n/4$), et que le calcul du coût marginal ne peut pas être amorti. La complexité est donc la même : $O(nm)$

Recherche locale Ces deux voisinages sont appliqués dans une méthode de recherche locale initialisée à partir d'une séquence obtenue par application de l'heuristique stochastique. Dans un premier temps, seul le voisinage *swap* est appliqué : on sélectionne le meilleur mouvement de *swap*, c'est à dire celui faisant diminuer le plus le retard maximal. Cette procédure est répétée jusqu'à ce qu'il n'y ait plus de mouvement *swap* améliorant. Dans un second temps, le voisinage *toggle* est considéré : le premier mouvement *toggle* améliorant est appliqué avant de revenir à la recherche de *swap* améliorant. Le processus entier est répété jusqu'à obtenir un minimum local. Lorsqu'un minimum local est obtenu, une nouvelle séquence d'opération est construire par l'heuristique stochastique et le processus est répété jusqu'à un temps limite.

3.5.5 Résultats expérimentaux

Cette section présente les résultats expérimentaux obtenus par l'algorithme REINFORCE, puis propose une comparaison des différentes méthodes décrites dans les sections précédentes.

3.5.5.1 Evaluation de l'algorithme d'apprentissage REINFORCE

Pour cette première expérimentation, on applique l'algorithme REINFORCE sur les quatre catégories d'instances générées aléatoirement (A, B, C, D) pour l'horizon temporel *jour*, ce qui correspond à 40 instances (voir section 3.4). On effectue l'apprentissage par *batch* de taille 240, c'est à dire que pour chaque instance, 6 épisodes sont générés avant de mettre à jour $\boldsymbol{\theta}$ et notre critère d'arrêt est le nombre d'itération, fixé à 2000. L'objectif est de comparer plusieurs exécutions de l'algorithme avec plusieurs paramétrages ou plusieurs ensembles de données d'apprentissage pour un même budget, le nombre de 2000 itérations a été fixé empiriquement. Le paramètre de température a été fixé à $\beta = 0.1$ et le pas d'apprentissage $\alpha = 2^{-12}/\overline{n}$ avec $\overline{n}$ le nombre moyen d'opérations dans les instances considérées. Les pénalités n'ont pas été normalisées pour cette expérimentation, et dans nos instances on a remarqué que le retard maximal tend à être plus grand quand le nombre d'opérations de l'instance augmente. On utilise alors le pas d'apprentissage pour compenser cet effet. Ces 2000 itérations prennent environ 2h sur un Intel(R) Core(TM) i7-8700 CPU @ 3.20GHz.

Dans la figure 3.2, on présente le résultat de cet apprentissage pour différentes valeurs de γ. Pour rappel, ce paramètre est un facteur décroissant exponentiellement sur le calcul du *gain* (eq. 1.18). Il permet de de paramétrer à quel point les récompenses futures sont importantes dans un état donné. Dans cette figure, chaque point donne la valeur du retard maximal moyen des instances du *batch* à une itération donnée. On donne les résultats de l'algorithme pour quatre valeurs de γ, $\gamma = 1$ en rouge, $\gamma = 0.99$ en bleu, $\gamma = 0.95$ en vert et $\gamma = 0.9$ en noir. On remarque alors immédiatement que le résultat de l'apprentissage avec la valeur $\gamma = 1$ converge très rapidement vers un plateau pour lentement continuer à décroître vers un minimum

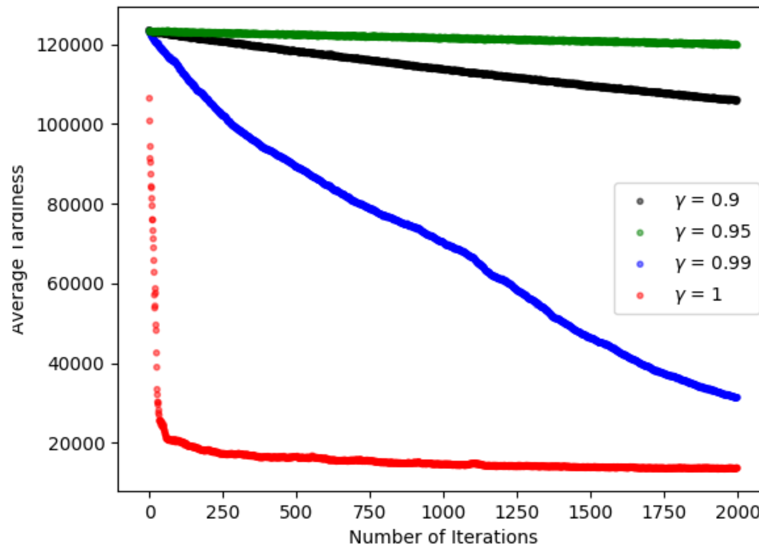


FIGURE 3.2 – Convergence pour différentes valeurs de γ

local. Ce plateau est beaucoup plus bas que pour les autres valeurs de γ qui ne convergent pas au bout de 2000 itérations. On peut alors penser que plus γ est petit, et moins bon le résultat sera, mais de manière surprenante, l'apprentissage avec $\gamma = 0.9$ semble obtenir de meilleurs résultats que l'apprentissage avec $\gamma = 0.95$.

Pour cette seconde expérimentation, on cherche à évaluer la qualité de l'algorithme REINFORCE en faisant varier l'ensemble des données considérées pour apprendre, dans les mêmes conditions que pour la première expérimentation, et avec $\gamma = 1$. La table 3.4 présente les résultats obtenus (valeurs moyennes du retard maximal) après apprentissage de différentes valeurs du vecteur de poids $\boldsymbol{\theta}$ sur plusieurs ensembles de données. Pour la première colonne, nous avons appris les valeurs de $\boldsymbol{\theta}$ pour chaque instance prise séparément. Pour la seconde colonne, on a appris une valeur de $\boldsymbol{\theta}$ pour chaque catégorie d'instances, et pour la dernière colonne, on a appris une unique valeur de $\boldsymbol{\theta}$ pour toutes les instances. Les apprentissages sont réalisés seulement pour l'horizon d'un *jour* à chaque fois. Dans la table, on reporte pour chaque classe (A, B, C, D) le retard moyen, sur les 10 instances de la classe. Le retard est obtenu par application de procédures gloutonnes suivant la politique $\pi_{\boldsymbol{\theta}}$ unique de chaque instance (avec une température $\beta = 0.01$).

On voit que l'avantage d'apprendre une valeur spécifique de $\boldsymbol{\theta}$ pour une classe d'instances plutôt que sur l'ensemble total est très faible. En ce qui concerne l'apprentissage pour une instance, il y a un gain par rapport à un apprentissage global, mais cet avantage s'estompe à mesure que la difficulté des instances augmente (environ 10% de gain pour la classe A, 6% pour classe B, 3% pour la classe C et 2% pour la classe C). Notre modèle est très simple et utilise peu de critères, il n'est donc pas surprenant que le gain sur un apprentissage spécifique soit peu impor-

tant. Cependant, cela veut dire qu'une unique valeur de $\boldsymbol{\theta}$ apprise sur l'ensemble de données est alors pertinent dans la plupart des cas.

Cl	Par Inst.	Par Cl.	Tous
A	1011	1130	1128
B	6417	6766	6797
C	14622	15184	15069
D	20285	20578	20640

TABLE 3.4 – Retard moyen obtenus avec plusieurs ensembles d'apprentissage

Dans la suite, on utilise comme poids pour la combinaison linéaire de la fonction d'évaluation, les valeurs $\boldsymbol{\theta}_{\mathsf{T}} = (0.25, 0.56, 0.15, 0.04)$ qui sont celles obtenues en fin d'apprentissage avec $\gamma = 1$ sur l'ensemble des 40 instances de l'horizon d'un *jour*. Ces poids indiquent qu'une bonne heuristique est principalement un compromis entre l'urgence et le temps de trajet.

3.5.5.2 Comparaison des méthodes exploitant l'heuristique apprise

Les expérimentations réalisées dans cette section s'appuient sur le jeu de données de 120 instances générées aléatoirement et dont les caractéristiques ont été présentées en section 3.4. Les méthodes évaluées sont : la recherche arborescente basée sur la politique stochastique et des redémarrages rapides (`CP-softmax`) ; la méthode à divergences limitées (`LDS`) et la recherche locale à démarrages multiples (`Multi-LS`). Pour chacune des 120 instances, chaque méthode a été lancée 10 fois avec un temps maximal d'une heure pour chaque exécution. Les méthodes `CP-softmax` et `LDS` ont été développées avec le solveur Choco 4.10 [142], la méthode `Multi-LS` ne s'appuie sur aucun outil existant et a été entièrement développée en Java. Les expérimentations ont été effectuées sur un cluster composé de Intel(R) Xeon E5-2695 v3 @ 2.30GHz et Intel(R) Xeon E5-2695 v4 @ 2.10GHz.

Chaque méthode s'appuie sur la même heuristique de choix de valeurs issue de la politique $\pi_{\boldsymbol{\theta}}(a \mid \sigma)$ avec les mêmes valeurs pour le vecteur de poids $\boldsymbol{\theta}$ appris avec l'algorithme REINFORCE, qui a été normalisé ($\sum_{i=1}^{4} \boldsymbol{\theta}_i = 1$), et avec $\beta = 1/150$.

Pour les deux méthodes (`CP-softmax` et `Multi-LS`) utilisant la version stochastique de la politique, on ajoute une première itération avec une version déterministe, c'est à dire que la politique devient stochastique seulement après le premier redémarrage pour la méthode basée sur la programmation par contraintes, et à partir de la seconde itération pour la méthode de recherche locale.

Les résultats sont présentés dans le tableau 3.5. Les deux premières colonnes du tableaux rappellent les quatre catégories d'instances A, B, C et D chacune déclinée sur trois horizons différents : le quart de travail d'un opérateur, une journée de

trois quarts et une semaine de six jours. Pour chacune de ces catégories et chacun de ces horizons, on dispose de dix instances. Ainsi, chaque ligne dans le tableau correspond à 10 instances, et chaque méthode a été exécutée 10 fois pour chacune des instances. Les résultats présentés sont des moyennes sur ces 10 exécutions par instance.

Dans ce tableau, les résultats des deux modèles de programmation par contraintes présentés en section 3.3 sont rappelés (colonnes `Ordo` et `TSP`) puis on y compare les trois méthodes basées sur l'heuristique issue de l'apprentissage par renforcement (colonnes `CP-softmax`, `LDS` et `Multi-LS`).

Les indicateurs relevés sont : le nombre d'instances résolues [2] (#S), le temps CPU moyen pour résoudre une instance (CPU), et le nombre moyen de conflits pour résoudre une instance (#conflit) pour les méthodes basées sur un solveur de programmation par contraintes. La méthode de recherche locale `Multi-LS` permet de retourner des solutions qui violent la contrainte de dates d'échéance. L'indicateur (LMAX) donne la valeur de retard maximal moyen en centième de minute et permet d'évaluer la difficulté des instances restantes à résoudre. Enfin, la dernière ligne donne le pourcentage d'instances résolues sur les 1200 exécutions de la méthode correspondante (10 exécutions pour chacune des 120 instances).

TABLE 3.5 – Comparaison des méthodes sur les instances générées aléatoirement

Cl	H	`Ordo`			`TSP`			`CP-softmax`			`LDS`			`Multi-LS`		
		#S	CPU	#conflit	#S	CPU	#conflit	#S	CPU	#conflit	#S	CPU	#conflit	#S	CPU	LMAX
A	`Quart`	7.1	418	300K	4.0	56	366	**9.0**	2	15	**9.0**	2	9	**9.0**	0	10
	`Jour`	4.0	29	213	3.6	802	1267	**9.0**	15	815	8.0	21	71	**9.0**	176	193
	`Semaine`	3.1	866	1976	0.0	mem. out		**8.0**	118	27	5.0	68	0.0	7.0	155	2433
B	`Quart`	2.1	389	150K	0.9	844	11K	**6.0**	4	77	**6.0**	15	85	**6.0**	2	467
	`Jour`	1.0	201	15K	0.0	–		**5.2**	341	20K	4.0	12	19	4.6	346	3218
	`Semaine`	0.0	–		0.0	mem. out		**3.5**	423	715	1.0	99	0.0	1.0	0	26915
C	`Quart`	0.0	–		0.0	–		**4.0**	103	5366	**4.0**	715	4090	**4.0**	255	1941
	`Jour`	0.0	–		0.0	–		**1.0**	12	7	**1.0**	18	27	**1.0**	1	9498
	`Semaine`	0.0	–		0.0	mem. out		**1.0**	807	366	0.0	–		0.0	–	71104
D	`Quart`	0.0	–		0.0	–		**1.9**	697	24K	1.0	442	1058	1.6	1165	2677
	`Jour`	0.0	–		0.0	–		0.0	–		0.0	–		0.0	–	13994
	`Semaine`	0.0	–		0.0	mem. out		0.0	–		0.0	–		0.0	–	107186
Total		14.4%			7.1%			40.5%			32.5%			36%		

L'analyse de ces résultats montre que les deux modèles initiaux de programmation par contraintes (`Ordo` et `TSP`) sont largement dominés par les trois méthodes basées sur l'heuristique apprise par renforcement. La méthode `CP-softmax` est la meilleure et semble mieux passer à l'échelle que les deux autres, avec un total de 40.5% d'instances résolues contre 32.5% et 36% pour la recherche à divergences limitées `LDS` et la recherche locale à démarrages multiples `Multi-LS` respectivement. Il est difficile de tirer des conclusions des autres indicateurs, cependant, on peut re-

2. Pour la méthode de recherche locale qui résout le problème d'optimisation, une instance est considérée comme résolue si son retard maximal est négatif ou nul

marquer deux choses en regardant la colonne `CP-softmax` : tout d'abord, le temps CPU pour résoudre une instance est relativement court, et le nombre de conflits est peu élevé comparativement aux deux modèles initiaux, notamment sur les instances de catégorie A et B pour l'horizon d'un *quart*. Cela semble indiquer que c'est bien l'heuristique apprise qui permet de trouver les solutions rapidement. La seconde remarque porte sur le ratio conflits/seconde qui peut être très faible pour les instances d'horizon plus long. Par exemple, l'unique instance de la catégorie C pour l'horizon d'un *jour* est résolue en provoquant moins de 1 conflit par seconde.

Néanmoins, de nombreuses instances restent non résolues, et on remarque avec les résultats de la recherche locale que le retard maximal des meilleurs solutions sur ces instances reste conséquent.

En ce qui concerne les instances industrielles (tableau non fourni ici) le paramètre $\boldsymbol{\theta}$ global appris sur les instances générées aléatoirement donne de bons résultats. En effet, toutes les instances sont facilement résolues par les trois méthodes, exceptée l'instance R pour l'horizon d'une *semaine* qui n'est résolue que par la méthode `CP-softmax`. Cependant, si on apprend une valeur de $\boldsymbol{\theta}$ spécifique pour ces instances industrielles, il s'avère que toutes les instances sont résolues directement en appliquant seulement un algorithme glouton déterministe utilisant la nouvelle valeur du vecteur de poids $\boldsymbol{\theta}$, sauf pour l'instance L avec l'horizon d'une *semaine*. Néanmoins, cette instance est facilement résolue par les trois méthodes (`CP-softmax`, `LDS` ou `Multi-LS`) proposées avec cette valeur de $\boldsymbol{\theta}$.

3.6 Recherche arborescente de Monte Carlo

Dans cette section, nous intégrons l'heuristique issue de l'apprentissage par renforcement dans une recherche arborescente de Monte Carlo (MCTS). Pour cela, on se place dans le cadre du processus de décision markovien décrit dans la section précédente, on utilise donc la version relâchée du problème, pour permettre d'effectuer les simulations avec une procédure gloutonne. Un noeud de l'arbre v correspond alors à un état s, correspondant lui même à une séquence d'opérations σ. On utilise ces termes de manières interchangeables.

3.6.1 Algorithme MCTS pour le problème de tournées de chariots

On propose de décrire les phases de l'algorithme et la variation que nous avons choisie pour chacune.

Sélection. On rappelle que pour ce problème nous minimisons une pénalité et donc que $Q(v,e) < Q(v,e')$ signifie que l'action correspondante à l'arc e est préférable. Nous proposons de normaliser le terme d'exploitation dans [-1, 1] comme

suit :

$$\overline{Q(v,e)} = \begin{cases} 2 * \frac{Q^+ - Q(v,e)}{Q^+ - Q^-} - 1 & \text{si } N(v,e) > 0 \\ 0 & \text{sinon} \end{cases} \tag{3.39}$$

avec $Q^+ = \max\{Q(v,e) \mid e \in A(v), N(v,e) > 0\}$ and $Q^- = \min\{Q(v,e) \mid e \in A(v), N(v,e) > 0\}$ représentant respectivement les valeurs maximale et minimale des arcs sortants déjà explorés. On utilise alors $\overline{Q(v,e)}$ comme terme d'exploitation dans l'équation 1.48 pour sélectionner le prochain noeud. On utilise également le mécanisme de compromis dynamique décrit dans le chapitre 2.

Expansion. Soit v, le noeud, correspondant à la séquence σ, qui a été sélectionné dans la phase de sélection. On ajoute un noeud pour chaque action $a \in A(v)$ si pour la séquence σ' obtenue après l'application de a on a $L(\sigma') = 0$. Autrement dit, on n'ajoute aucun noeud de l'arbre correspondant à une opération qui sera en retard. Si aucune action n'est ajoutée alors v est supprimé. Les probabilités à priori de chacun des noeuds ajoutés sont données par $\pi_{\boldsymbol{\theta}}$ avec une température $\beta = 0.1$ dans la fonction `softmax` (eq. 2.1).

Simulation. La simulation est appliquée à partir du noeud sélectionné pendant la phase de sélection et se fait en deux temps. D'abord, une procédure gloutonne est exécutée, suivant la politique $\pi_{\boldsymbol{\theta}}$ avec une température $\beta = 0.005$ jusqu'à ce que la borne inférieure sur le retard maximale dépasse la meilleure solution obtenue jusqu'à présent. Ensuite, on définit un budget de retours-arrière entre 0 et β^+, un paramètre dépendant de la position $\phi(\sigma)$ du premier retard dans la séquence obtenue σ. Si σ est une solution améliorante, alors le budget est maximal (β^+). Sinon, soit ϕ^* la taille de la plus grande séquence sans retard trouvée depuis le début de la résolution, le budget est alors :

$$\beta = \begin{cases} \beta^+ & \text{si } \phi(\sigma) \geq \phi^* \\ \beta^+ (1 - \frac{\phi^* - \phi(\sigma)}{\phi^* - \Gamma * \phi^*})^2 & \text{si } \phi^* > \phi(\sigma) > \Gamma * \phi^* \\ 0 & \text{sinon} \end{cases} \tag{3.40}$$

Avec $\Gamma < 1$ le paramètre de seuil.

Une fois le budget du DFS entièrement consommé (ou si aucun budget n'a été accordé), la plus longue séquence sans retard trouvé est ensuite étendue par une procédure gloutonne à une solution complète, et la borne supérieure n'est plus utilisé pendant cette phase pour obtenir cette solution. C'est cette solution complète qui est retournée par la phase de simulation.

Durant le DFS on utilise la politique $\pi_{\boldsymbol{\theta}}$ sur chacun des noeuds enfants, et pour contrôler la largeur de l'arbre on propose de ne garder que les noeuds dont la probabilité est supérieure à un seuil fixé à 10^-6, ce qui laisse approximativement entre 1 et 3 enfants. Ces enfants sont ensuite triés selon leur probabilité, et un

unique noeud est tiré aléatoirement puis échangé avec le premier. Cet ordre sera alors l'ordre d'exploration du DFS. Étant donné que les branches peuvent être très longues, cela suffit pour garder suffisamment de variétés dans les solutions, tout en évitant de mauvaises décisions selon la politique apprise. C'est aussi la raison pour laquelle on utilise un budget de retours-arrière plutôt que de conflits étant donné que de nombreux noeuds peuvent n'avoir qu'un seul enfant. Enfin, nous avons remarqué expérimentalement que l'ajout d'une stratégie de redémarrages géométriques durant cette phase permettait d'améliorer les résultats obtenus. La recherche redémarre alors au noeud à partir duquel la simulation initiale a démarré et non pas à la racine de l'arbre.

Rétropropagation. La phase de rétropropagation se déroule comme celle décrite dans la section 1.3.

3.6.2 Résultats expérimentaux

On utilise pour ces expérimentations numériques l'ensemble des instances générées aléatoirement et décrites dans la section 3.4.1. Nous avons comparé différentes variantes de la recherche arborescente de Monte Carlo. Nous avons lancé pour chacune de ces variantes 10 exécutions sur chacune des 120 instances avec un temps limite d'1 heure par instance. Ces expérimentations ont été réalisées sur un cluster composé de processeur Xeon E5-2695 v3 @ 2.30GHz. Les méthodes ont été implémentées en C++ et compilées avec GCC-8.0.

3.6.2.1 Comparaison de méthodes MCTS

On compare six variantes de la recherche arborescente de Monte Carlo :

— `MCTS` est l'algorithme standard de recherche arborescente de Monte Carlo, sans aucune des adaptations proposées dans le chapitre 2.2. De plus, on considère une unique récompense à l'état final des simulations, sans aucun facteur de décroissance exponentielle.

— `MCTS+DFS` est le même algorithme que `MCTS` avec l'ajout de la phase de DFS dans la simulation.

— `G-MCTS` est une variante de `MCTS` pour laquelle on utilise l'accroissement de la borne inférieure comme récompense à chaque décision, avec un facteur de décroissance exponentielle.

— `G-MCTS+DFS` ajoute le DFS à la phase de simulation

— `G-MCTS+DFS+CD` étend `G-MCTS+DFS` en ajoutant le compromis dynamique tel que décrit dans 2.2 lors de la phase de sélection.

— `G-MCTS+SAT-DFS+CD` est une variante de `G-MCTS+DFS+CD` dans laquelle une borne supérieure sur la fonction objectif est fixée à 1 dans le DFS, et dans le DFS seulement. L'intensification consiste alors à chercher à trouver une solution sans aucun retard. Cependant la borne supérieure est à nouveau

relâchée dans la dernière partie de la phase de simulation qui produit alors une solution complète avec une procédure gloutonne, et donc la méthode au globale fournit une borne supérieure.

Les valeurs des différents paramètres de ces méthodes sont reportés dans la table 3.6. On rappelle que c est le compromis entre exploitation et exploration et Λ est le facteur de décroissance appliqué à c pour le compromis dynamique. γ est le facteur de décroissance appliqué au calcul du *gain* (eq.1.18). Enfin Γ et β^+ sont respectivement le paramètre de seuil et le nombre maximal de retours-arrière pour le DFS. Les valeurs de ces paramètres ont été choisies via des expérimentations préliminaires, et la combinaison présentée est celle ayant conduit aux meilleurs résultats.

c	1
Λ	0.995
γ	0.9977
α	0.9
β^+	50000
Redémarrage (base)	100
Redémarrage (facteur)	1.2

TABLE 3.6 – Valeurs des paramètres des méthodes MCTS

Les résultats sont présentés dans les tables 3.7 et 3.8 dans lesquelles on reporte le nombre d'instances résolues (#S) ainsi que le retard maximal moyen (LMAX). Pour chacune des méthodes, on considère qu'une instance est résolue si et seulement si la fonction objectif est nul, c'est à dire si il n'y a aucun retard. La table 3.7 détaille les performances des différentes variations de la méthode MCTS par classes d'instances. Dans cette table, chaque ligne correspond à 100 exécutions par méthode (10 instances et 10 exécution), le nombre d'instances résolues (#S) est moyenné sur les 10 exécutions. Dans la table 3.8, le même résultat est présenté mais agrégé par horizon temporel. Dans cette table, le nombre d'instances résolues est fourni en pourcentage (sur les 400 exécutions par ligne et par méthode).

Dans ces tables, on peut voir les avantages d'utiliser le DFS durant la phase de simulation, qui permet de résoudre plus d'instances de l'horizon le plus long (*semaine*). En effet ces instances sont trop grandes pour être résolues avec des procédures gloutonnes uniquement, et le DFS permet d'intensifier la recherche dans les parties les plus profondes de l'arbre qui ne sont pas explorées par le MCTS seul. Ces tables montrent également le bénéfice de l'utilisation du facteur décroissant en terme de valeur de la fonction objectif sur les horizons les plus long d'un *jour* et d'une *semaine*. Cependant, les résultats sur l'horizon le plus court (*quart*) sont dégradés. Au global, la combinaison de ces deux mécanismes produit de meilleurs résultats que les deux versions avec un seul de ces mécanismes. De plus, on peut voir les effets du compromis dynamique sur l'horizon d'un *jour*. Cet horizon reste trop

Cl	H	MCTS		MCTS+DFS		G-MCTS		G-MCTS+DFS		G-MCTS+DFS+CD		G-MCTS+SAT-DFS+CD	
		#S	Lmax	#S	Lmax	#S	Lmax	#S	Lmax	#S	Lmax	#S	Lmax
A	Quart	**10.0**	**0**	**10.0**	**0**	**10.0**	**0**	**10.0**	**0**	**10.0**	**0**	**10.0**	**0**
	Jour	9.0	135	9.0	133	9.0	115	9.0	77	**10.0**	**0**	9.8	**0**
	Semaine	6.8	1996	7.8	1850	7.0	**1800**	**8.0**	1839	7.7	1840	**8.0**	1858
B	Quart	8.0	420	7.9	**258**	**9.0**	372	8.2	353	8.1	349	8.7	439
	Jour	5.0	2954	5.4	2959	6.0	2522	6.0	2439	6.3	**2121**	**7.0**	2134
	Semaine	1.0	21070	2.9	20771	1.0	20572	3.2	20541	3.1	**20355**	**3.6**	20635
C	Quart	**4.9**	**1676**	4.8	1708	4.0	1901	4.5	1727	4.0	1824	4.0	2012
	Jour	1.0	9503	1.1	9248	1.0	8683	2.6	**8656**	**3.6**	8747	3.5	9022
	Semaine	0.0	**64442**	0.8	64713	0.0	64480	0.9	64584	0.9	64474	**1.0**	64445
D	Quart	**4.0**	2154	3.3	2146	3.0	2338	3.3	**2018**	3.0	2304	3.0	2621
	Jour	0.0	13659	0.0	13664	0.0	12657	0.0	12723	**1.3**	**12225**	1.1	12340
	Semaine	0.0	101474	0.0	101444	0.0	**100533**	0.0	100760	0.0	100954	0.0	100840
Total		41%		44%		42%		46%		48%		**50%**	

Table 3.7 – Comparaison de différentes versions de MCTS

H	MCTS		MCTS+DFS		G-MCTS		G-MCTS+DFS		G-MCTS+DFS+CD		G-MCTS+SAT-DFS+CD	
	#S	Lmax	#S	Lmax	#S	Lmax	#S	Lmax	#S	Lmax	#S	Lmax
Quart	**0.67**	1063	0.65	1028	0.65	1153	0.65	**1024**	0.63	1119	0.64	1268
Jour	0.38	6562	0.39	6501	0.40	5994	0.44	5974	0.53	**5773**	**0.54**	5874
Semaine	0.20	47245	0.29	47195	0.20	**46846**	0.30	46931	0.29	46906	**0.32**	46944

Table 3.8 – Résultats agrégés par horizon temporel

long pour que le MCTS puisse progresser suffisamment profondément dans l'arbre de recherche pour trouver des solutions. Ce mécanisme, en forçant l'exploitation permet d'améliorer les résultats sur le nombre d'instances résolues. Enfin, l'ajout du DFS en satisfaction permet sans surprise de résoudre plus d'instances, en obtenant de moins bons résultats en terme de valeur de la fonction objectif.

En résumé, si chaque ajout à la méthode d'origine pris séparément, n'améliore que légèrement les résultats, la combinaison de chacune de ces modifications permet d'obtenir jusqu'à 9% d'instances résolues supplémentaires.

3.6.2.2 Comparaison avec les méthodes précedentes

Dans cette seconde partie, on compare les deux meilleures versions de la méthode (`G-MCTS+DFS+CD` et `G-MCTS+SAT-DFS+CD`) avec les approches présentées dans la section 3.5.4. On reporte ici seulement les résultats obtenus par la méthode de programmation par contraintes avec redémarrages rapides qui a obtenu les meilleurs résultats ainsi que l'approche de recherche locale qui donnait en plus une borne supérieure sur le retard maximal. Il est intéressant de noter que toutes ces approches exploitent la même heuristique issue de l'apprentissage par renforcement. Cette comparaison est fournie dans la table 3.9. Dans cette table, on peut voir que les

deux versions de la recherche arborescente de Monte Carlo surpassent clairement les méthodes développées dans la section 3.5.4 aussi bien du point de vue du nombre d'instances résolues que de la fonction objectif. Plus particulièrement, la dominance est clairement marquée pour les instances d'horizon d'un *quart* et d'un *jour* en terme de nombre d'instances résolues. Cependant, aucune des méthodes MCTS ne surpasse `CP-softmax` sur l'horizon d'une *semaine*. Au final, entre l'approche `CP-softmax`, de type programmation par contraintes, et la version `G-MCTS+SAT-DFS+CD`, il y a une différence de 9.5% en terme de nombre d'instances résolues en faveur de la seconde. Cependant, on remarque que la moitié de l'ensemble d'instances reste sans solution, mais nous pensons qu'une grande majorité de celles-ci ne sont en fait pas satisfiables (en particulier pour l'horizon d'une *semaine* de la classe D).

Cl	H	CP-softmax	Multi-LS		G-MCTS+DFS+CD		G-MCTS+SAT-DFS+CD	
		#S	#S	LMAX	#S	LMAX	#S	LMAX
A	Quart	9.0	9.0	10	**10.0**	**0**	**10.0**	**0**
	Jour	9.0	.90	193	**10.0**	**0**	9.8	**0**
	Semaine	**8.0**	7.0	2433	7.7	**1840**	**8.0**	1858
B	Quart	6.0	6.0	467	8.1	**349**	**8.7**	439
	Jour	5.2	4.6	3218	6.3	**2121**	**7.0**	2134
	Semaine	3.5	1.0	26915	3.1	**20355**	**3.6**	20635
C	Quart	**4.0**	**4.0**	1941	**4.0**	**1824**	**4.0**	2012
	Jour	1.0	1.0	9498	**3.6**	**8747**	3.5	9022
	Semaine	**1.0**	0.0	71104	0.9	**64474**	**1.0**	64445
D	Quart	1.9	1.6	2677	**3.0**	**2304**	**3.0**	2621
	Jour	0.0	0.0	13994	**1.3**	**12225**	1.1	12340
	Semaine	0.0	0.0	107186	0.0	100954	0.0	**100840**
	Total	40.5%	36%		48%		**50%**	

TABLE 3.9 – Comparaison des méthodes

3.7 Synthèse

Dans ce chapitre, nous avons traité d'un problème de déplacement de chariots dans un atelier d'assemblage. Nous avons exprimé le problème sous forme de deux modèles de programmation par contraintes, le premier inspiré de modèle d'ordonnancement disjonctif, le second tiré d'un modèle de voyageur de commerce avec fenêtre de temps. Nous avons établi que le premier modèle était le plus performant mais n'était pas en mesure de résoudre des problèmes plus contraints que les données industrielles bien que surpassant l'approche mise en place dans l'entreprise. A partir de ce constat, nous avons alors développé une heuristique basée sur quatre critères dont la combinaison a été obtenue via un algorithme d'apprentissage par renforcement. Nous avons proposé ensuite trois approches prenant appui sur cette heuristique : deux méthodes issues du modèle d'ordonnancement de programmation par contraintes et une recherche locale. Ces trois méthodes permettent d'améliorer largement les résultats. Enfin, nous avons proposé plusieurs variantes de

la recherche arborescente de Monte Carlo pour ce problème. Ces variantes mettent en avant l'utilité des mécanismes que nous proposons pour cette méthode : l'utilisation d'un facteur décroissant exponentiellement dans l'évaluation des simulations, l'hybridation de la méthode avec une recherche en profondeur, et un mécanisme de compromis dynamique entre exploitation et exploration. Nous avons montré que l'utilisation conjointe de ces mécanismes permettent d'améliorer les résultats obtenus par les trois méthodes précédentes.

Chapitre 4

Problème de chargement de camions

Sommaire

4.1 Description du problème 94
4.2 Travaux connexes 98
4.2.1 Liens avec des problèmes de la littérature 98
4.2.2 Algorithme existant dans l'entreprise 101
4.3 Proposition d'une méthode de recherche arborescente . . . 103
4.3.1 Approche itérative 103
4.3.2 Stratégies de placement et d'exploration 104
4.3.3 Caractérisation de l'incomplétude 110
4.4 Apprentissage par renforcement de la stratégie de branchement 116
4.4.1 Processus de décision markovien 116
4.4.2 Caractérisation de l'heuristique de branchement 118
4.4.3 Résultats expérimentaux 123
4.5 Recherche arborescente de Monte Carlo 130
4.5.1 Algorithme MCTS pour le problème de chargement de camions 130
4.5.2 Résultats expérimentaux 132
4.6 Synthèse 136

Le second problème industriel considéré dans cette thèse est lié aux activités de logistique pour approvisionner des usines en pièces. Il s'agit d'un problème d'acheminement de pièces par transport routier, depuis des fournisseurs vers une usine, tout en respectant des contraintes liées aux demandes de l'usine et à la disponibilité des pièces, des contraintes sur les capacités et disponibilités des camions, etc.

La résolution de ce problème complexe, considérant plusieurs échelons temporels et différentes fonctions logistiques de l'entreprise, a été décomposée en plusieurs sous-problèmes traités successivement. L'enjeu qui nous intéresse dans ce chapitre est celui de l'optimisation du chargement des camions. Plus précisément, on dispose de piles rectangulaires d'emballages de pièces affectées à des camions dont la tournée est déjà établie. Cette pré-affectation étant contrainte par les demandes et disponibilités des pièces, il est très fréquent d'avoir besoin de camions supplémentaires, dit de *débord*, pour acheminer les pièces prévues sur cette tournée là. L'objectif est alors de minimiser le nombre de véhicules nécessaires pour transporter un ensemble

donné de pièces, et de minimiser la place occupée dans le dernier camion, qu'on appelle le *métrage linéaire*. Le but est de pouvoir compléter ce dernier camion de débord avec d'autres piles (ceci est effectué dans une phase de post-traitement).

La hauteur des piles étant fixée, le problème du chargement des camions se ramène à un problème de placement en deux dimensions. En complément des contraintes de placement et celle lié à la masse des piles, une contrainte spécifique liée à l'équilibrage de la charge aux essieux des camions doit être considérée. Cette contrainte de charge aux essieux n'était pas intégrée dans le module de calcul de chargement de véhicules au moment de commencer ces travaux. Elle était prise en compte a posteriori, ce qui conduisait à rejeter des solutions de chargement qui s'avéraient en fait non réalisables. Cette contrainte est aujourd'hui intégrée dans ce module, mais d'une manière non-satisfaisante, qui conduit cette fois à perdre de nombreuses solutions réalisables potentiellement de bonne qualité.

Par ailleurs, un rang de visite est défini par sous-ensembles de piles d'emballage. En pratique, ces sous-ensembles de piles sont regroupés chez les différents fournisseurs et un ordre de visite des fournisseurs a été pré-calculé en amont du calcul de chargement des camions. Les camions se chargeant par le fond, une pile d'un rang donné ne peut pas être insérée dans le camion s'il contient des piles d'un rang supérieur.

Ce chapitre est articulé de la façon suivante. Tout d'abord, nous décrivons formellement le problème traité, puis nous le positionnons par rapport à des problèmes de la littérature et nous présentons la méthode actuellement utilisée dans l'entreprise. Nous détaillons ensuite la recherche arborescente que nous avons développée, et qui sera utilisée comme brique de base dans les deux sections suivantes que sont l'apprentissage par renforcement d'une heuristique de choix de valeurs, et l'utilisation de la recherche arborescente de Monte Carlo.

4.1 Description du problème

On dispose d'un ensemble I d'objets rectangulaires (c'est à dire les piles d'emballage de pièces) que l'on doit faire rentrer dans des camions identiques. Ces camions sont contraints par deux dimensions, leur largeur X_{MAX} et longueur Y_{MAX}, et peuvent contenir une masse maximale m_{MAX}. Chaque objet est caractérisé par ses deux dimensions w_i et l_i, avec $w_i \leq l_i$, sa masse m_i, et une contrainte d'orientation $o_i \in \{0, 1, 2\}$ avec

- $o_i = 0$: aucune contrainte d'orientation
- $o_i = 1$: la longueur de la pile doit être dans la **longueur** du camion
- $o_i = 2$: la longueur de la pile doit être dans la **largeur** du camion

Les objets ont également un rang $r_i \in \mathbb{N}$, qui détermine leur ordre d'insertion. L'objectif est de ranger tous les objets dans un nombre minimal de camions et de minimiser la place occupée dans le camion le moins rempli, mesurée par ce qu'on

appelle le métrage linéaire. Le métrage linéaire est la longueur entre le fond du camion et l'extrémité de l'objet le plus éloigné du fond.

Une solution de ce problème peut se représenter par une partition $\mathcal{P}$ de l'ensemble des objets I, représentant l'affectation des objets aux différents camions. Pour chacun des objets $i \in I$, on considère deux variables représentant les coordonnées du coin gauche inférieur $(X(i), Y(i))$ dans le camion auquel il a été affecté et une variable $lw(i) \in \{0, 1\}$ pour son orientation : $lw(i) = 1$ si la longueur de l'objet est dans la longueur du camion, 0 sinon.

Exemple *Un exemple de solution de chargement d'un camion est représentée dans la figure 4.1. On considère que la largeur du camion est placée sur l'axe des abscisses et que la longueur du camion est placée sur l'axe des ordonnées. On rappelle que la hauteur des objets (piles) n'a pas besoin d'être considérée explicitement pour le calcul d'un chargement. Sur cette figure, chaque objet est identifié par les coordonnées de son coin gauche inférieur. Par exemple, l'objet en bas à gauche de cette figure est identifié par les coordonnées* $(0, 0)$*, l'objet en haut à droite a comme coordonnées* $(35, 30)$*.*

Dans chacun des camions, le chargement doit respecter les contraintes suivantes :

Contrainte d'orientation. L'orientation des objets doit respecter l'orientation imposée :

$$(o_i = 1 \implies lw(i) = 1) \wedge (o_i = 2 \implies lw(i) = 0) \qquad \forall i \in I \tag{4.1}$$

Contrainte de non chevauchement. Les objets ne doivent pas se chevaucher dans les camions. On a alors pour toute paire d'objets placée dans le même camion, $\{i, j\} \subseteq I' \subseteq \mathcal{P}$:

$$\begin{aligned} &X(i) + (lw(i) * w_i + (1 - lw(i)) * l_i) \leq X(j) \vee \\ &Y(i) + (lw(i) * l_i + (1 - lw(i)) * w_i) \leq Y(j) \vee \\ &X(j) + (lw(j) * w_j + (1 - lw(j)) * l_j) \leq X(i) \vee \\ &Y(j) + (lw(j) * l_j + (1 - lw(j)) * w_j) \leq Y(i) \end{aligned} \tag{4.2}$$

Contrainte de confinement. Aucun objet $i \in I$ ne doit dépasser les limites du camion :

$$\begin{aligned} X(i) &\geq 0 \\ Y(i) &\geq 0 \\ X(i) + (lw(i) * w_i + (1 - lw(i)) * l_i) &\leq X_{MAX} \\ Y(i) + (lw(i) * l_i + (1 - lw(i)) * w_i) &\leq Y_{MAX} \end{aligned} \tag{4.3}$$

M_t	Poids du tracteur
D_1	Distance entre les deux essieux du tracteur
D_2	Distance entre l'essieu avant et le centre de masse du tracteur
D_3	Distance entre l'essieu avant et la sellette [1]
M_c	Poids de la SR (semi remorque) vide
D_4	Distance entre la sellette et le tridem arrière [2] arrière
D_5	Distance entre le centre de masse de la SR et le tridem arrière
D_6	Distance entre le fond de la SR et la sellette (porte à faux)
N_1^{MAX}	Poids max sur l'essieu moteur
N_2^{MAX}	Poids max sur le tridem arrière

Table 4.1 – Données du camion pour calculer la charge aux essieux

Contrainte de rang. Le rang des objets indique l'ordre dans lequel les objets doivent être insérés. Pour tout ensemble $I' \subseteq \mathcal{P}$ d'objets placés dans le même camion, la position des objets dans I' doit respecter leur rang de chargement :

$$r_i < r_j \implies Y(i) \leq Y(j) \qquad \forall \{i, j\} \subseteq I', \forall I' \subseteq \mathcal{P} \tag{4.4}$$

Contrainte de masse. Chaque camion est contraint par la masse de son chargement :

$$\sum_{i \in I'} m_i \leq m_{MAX} \qquad \forall I' \subseteq \mathcal{P} \tag{4.5}$$

Contrainte de charge aux essieux. Enfin, la contrainte de charge aux essieux peut s'exprimer comme une contrainte sur le centre de masse du chargement. Ce centre de masse doit se trouver dans un intervalle dépendant de la masse du chargement. On note $\phi^-(M)$ et $\phi^+(M)$ la valeur minimale (respectivement maximale) que peut prendre le centre de masse d'un chargement de masse M, avec

$$\phi^-(M) = D_6 + D_4 - \frac{\frac{D_4(N_1^{MAX} * D_1 - M_t * D_2)}{D_3} - M_c * D_5}{M}$$
$$\phi^+(M) = D_6 + D_4 - \frac{D_4(M + M_c - N_2^{MAX}) - M_c * D_5}{M}$$

Ce calcul utilise de nombreuses données du camion que l'on trouve synthétisées dans la table 4.1. En résumé, si le centre de masse du chargement est inférieur à $\phi^-(M)$, alors l'essieu moteur sera surchargé. Si le centre de masse du chargement est supérieur à $\phi^+(M)$ ce sera l'essieu de la semi-remorque qui sera surchargé.

La contrainte sur la charge aux essieux consiste à vérifier que le centre de masse

est dans cet intervalle :

$$\sum_{i \in I'} \frac{Y(i) + (lw(i) * l_i + (1 - lw(i)) * w_i)/2}{m_i} \in [\phi^-(\sum_{i \in I'} m_i), \phi^+(\sum_{i \in I'} m_i)] \quad \forall I' \subseteq \mathcal{P} \tag{4.6}$$

Fonction objectif. L'objectif du problème de chargement de camions est de minimiser le nombre de sous-ensembles (la cardinalité) de la partition $\mathcal{P}$, ce qui correspond au nombre de camions, puis de minimiser le métrage linéaire du camion le moins rempli, c'est à dire de minimiser :

$$\min_{I' \subseteq \mathcal{P}} \max_{i \in I'} Y(i) + (lw(i) * l_i + (1 - lw(i)) * w_i)$$

Ces deux objectifs peuvent être agrégés, l'objectif est alors de minimiser :

$$(|\mathcal{P}| - 1) * Y_{MAX} + \min_{I' \subseteq \mathcal{P}} \max_{i \in I'} Y(i) + (lw(i) * l_i + (1 - lw(i)) * w_i) \tag{4.7}$$

Exemple *La figure 4.1 illustre un placement d'objets dans un camion et le métrage linéaire correspondant. Dans cette figure, il y a 7 objets de 3 rangs différents : 20*35 et 15*40 en bleu, 15*35 et 15*23 en rouge, 20*25, 15*15 et 25*45 en jaune. Le fournisseur des piles bleues est visité avant le fournisseur des piles rouges, lui même visité avant le fournisseur des piles jaunes. Le métrage linéaire correspondant est la plus grande ordonnée de l'arête supérieure d'un objet, ici le métrage linéaire est* 75 *unités.*

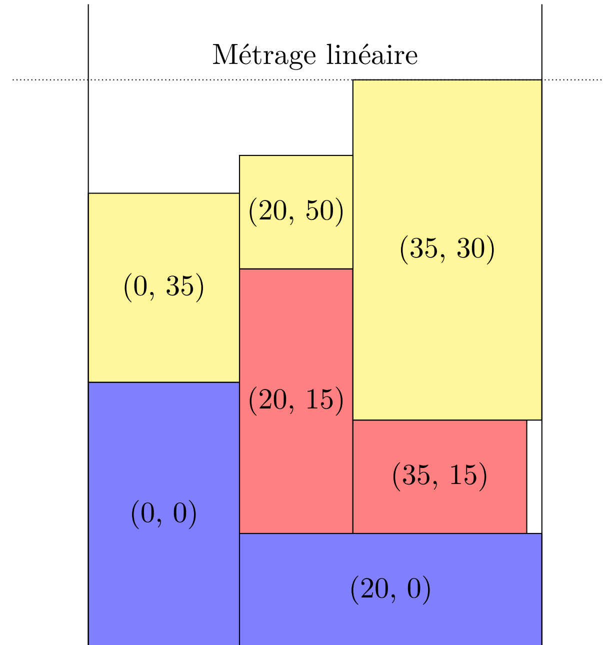


FIGURE 4.1 – Exemple de placement d'objets dans un camion

En suivant la représentation de la figure 4.1 et pour simplifier les notations, on peut considérer que les deux dimensions des objets i sont la hauteur h_i et la longueur l_i [3]. Excepté pour le calcul de bornes, tout objet qui peut être mis dans les deux orientations est alors dupliqué, chacun correspondant à une orientation, et la variable $lw(i)$ indique quelle version de l'objet i est choisi. Le bas du camion est l'axe des abscisses, sa hauteur est Y_{MAX}, et sa longueur est X_{MAX}. Dans la suite, on parle d'*objet* lorsqu'ils peuvent être pivotés (chacun décrit par une largeur et une longueur), sinon on parle de *rectangle* (avec une longueur et une hauteur fixé). On appelle aussi les camions, des boîtes lorsque le problème considéré est plus général que le problème de chargement de camions avec les contraintes métiers décrites.

4.2 Travaux connexes

Le problème étudié entre dans la catégorie des problèmes dit de *packing*. On présente dans cette partie plusieurs méthodes de la littérature pour la résolution de ce type de problème. Nous présentons ensuite l'algorithme actuellement utilisé dans l'entreprise.

4.2.1 Liens avec des problèmes de la littérature

Il s'agit d'un problème en deux dimensions, pour lequel les objets peuvent avoir plusieurs orientations, et sans coupe guillotine (sans contrainte sur le placement des objets les uns par rapport aux autres) que l'on note 2BP|R|F (pour *Two-Dimensionnal Bin Packing Problem*, *Rotation*, *Free*) [113]. Dans le problème industriel étudié, sont ajoutées, la contrainte de masse 4.5, la contrainte de charge aux essieux 4.6 ainsi que la contrainte sur le rang 4.4 proche d'une contrainte que l'on trouve dans la littérature sous le nom de contrainte de déchargement (*Unloading Constraint*) [45]. L'objectif de minimisation du métrage linéaire du camion le moins rempli est un objectif de *Strip Packing* (SP), problème qui consiste à trouver un placement des objets dans une boite de hauteur infinie minimisant la hauteur totale du rangement. La composante principale de ces problèmes, qui est le placement des objets dans des boites se trouve également dans deux autres problèmes, le problème du sac à dos en deux dimensions (*Two Dimensional Knapsack Problem*, 2D-KP), dans lequel chaque objet à un profit et l'on doit maximiser la somme des profits des objets contenus dans une boite, ainsi que le problème de *Rectangle Packing* (RP) dans lequel on veut minimiser la surface de la boîte pour empaqueter tous les objets, appelé aussi *Orthogonal Packing Problem* (OPP) pour la version en décision.

Ces problèmes sont tous NP-Difficile [123, 113, 99]. On peut montrer facilement que notre problème est également difficile à partir du problème 2BP|R|F en désactivant les contraintes métiers, en donnant un unique rang et un poids nul pour tous

3. La longueur se mesure sur l'axe des ordonnées et la hauteur se mesure sur l'axe des abscisses. La hauteur dans ce contexte n'a pas à voir avec la hauteur des piles qui n'est pas considérée dans ce problème.

les objets.

Méthodes exactes et heuristiques pour les problèmes de *packing*. Parmi les approches exactes, plusieurs travaux se focalisent sur le calcul de bornes inférieures [123, 6, 122, 25, 32]. La plupart de ces bornes concerne les problèmes sans rotation (les orientations des objets sont fixées). Une procédure possible pour les appliquer est alors de découper les objets en un ensemble de carrés [49]. Une procédure de séparation et évaluation a été proposée pour le SP par Martello et al. [123], laquelle considère une extension de la solution partielle courante pour chaque objet pouvant être placé dans les positions à gauche et le plus bas possible. Pour éliminer les redondances qui peuvent être créées par une telle procédure, Fekete and Schepers [56, 57] propose une représentation d'un placement comme k graphes d'intervalles, un pour chaque dimension, dans lesquels un sommet est un objet, et un arc entre deux sommets signifie un chevauchement dans la projection sur l'axe de la dimension considérée. Une procédure énumérative a été proposée pour trouver les graphes qui forment une solution valide, en évitant alors de nombreuses redondances. Une autre proposition est de modifier la procédure énumérative de [123] pour ne considérer qu'une seule position (la plus à gauche, puis la plus basse) pour chaque objet, et de considérer également le choix de ne pas mettre d'objet à cette emplacement [40].

Une procédure énumérative à deux niveaux de branchement pour le BP est proposée par Martello and Vigo [122]. Le premier niveau consiste en une recherche arborescente pour l'affectation des objets aux boites. Le second niveau permet de vérifier qu'une affectation est valide, elle consiste alors à résoudre un OP. Après un échec de la réfutation, évalué par des calculs de bornes inférieures, et un échec d'une validation heuristique, une procédure énumérative est alors exécutée. Celle ci est inspirée de Hadjiconstantinou and Christofides [69] pour le 2D-KP, proche de celles de Martello et al. [123] pour le SP.

Une autre approche exacte consiste à décomposer le problème et à le résoudre dans une procédure de séparation et d'évaluation avec génération de colonnes (*branch and price*). Une couverture d'ensembles est utilisé comme problème maître, chaque ensemble est un rangement dans une boite. Le problème secondaire est un 2D-KP résolu soit heuristiquement, soit en deux temps par une procédure de sélection d'objets selon les profits d'abord, puis un modèle de programmation par contraintes pour vérifier le placement (OP). La programmation par contraintes présente un modèle compact pour le RP et l'OP utilisant une contrainte globale de non chevauchement en deux dimensions (`non overlap2D`), ou des disjonctions de deux contraintes de non chevauchement pour chaque paire de rectangle, ainsi que deux contraintes (redondantes) `cumulatives`, une pour chaque dimension [41, 159, 99]. Ce modèle utilise alors la même représentation que l'approche par graphe d'intervalle décrite précédemment [56, 57]

La contrainte d'ordre sur les objets a été prise en compte dans une autre approche basée sur une décomposition [45] pour l'OP. Cette méthode résout itérati-

vement un problème en une dimension d'un BP contigu en découpant les objets et la boite horizontalement. Chaque solution relâchée donne les ordonnées, et un problème secondaire consiste à chercher alors les abscisses des rectangles en vérifiant la contrainte d'ordre du déchargement, ainsi, soit une solution est trouvée, soit une coupe est ajoutée à la relaxation. Des méthodes similaires en découpant verticalement la boite et les objets ont été également proposées [44, 40]. Cette approche à partir de cette relaxation, ainsi que des règles de dominance ont été proposées pour le RP par Korf et al. [99] qui présente également une méthode de programmation par contraintes. La méthode de PPC se base sur le placement relatif des objets les uns par rapport aux autres, ainsi que sur des mécanismes de filtrage de domaine, de cassage de symétries ainsi que des heuristiques de branchement dédiées.

Pour le problème de placement des objets, de nombreuses heuristiques se basent sur la règle du plus bas à gauche (*bottom left*). Le principe est de trier les objets, puis de les insérer dans l'ordre, le plus bas possible, et ensuite de les loger le plus à gauche possible. Cependant, certaines solutions, potentiellement optimales, ne sont accessibles par aucune permutation [11]. Une autre approche, qu'on appelle *best-fit*, consiste à d'abord sélectionner l'emplacement avant le rectangle. En particulier, la méthode employée par Renault, et la nôtre, s'appuient sur une telle méthode [27, 170] et sera décrite dans la section suivante. Un autre type de méthode consiste à découper la boite en niveaux [42]. Le premier niveau est le fond de la boîte et chaque niveau supérieur est défini par une ligne horizontale sur le rectangle le plus haut du niveau précédent. Les objets sont triés puis placés le plus à gauche possible dans un des niveaux choisis selon une règle de placement. Un niveau est créé si aucun ne convient. Plusieurs règles de placement existe : *next-fit* considère le dernier niveau créé et *first-fit* considère le premier niveau qui peut contenir l'objet considéré.

Pour le problème de BP, plusieurs méthodes consistent à résoudre d'abord un problème de SP, puis à utiliser cette solution pour obtenir une solution du BP [113, 20, 39]. Par exemple, Chung et al. [39] proposent de construire une solution de SP par niveau. Puis, chaque niveau est interprété comme un objet de la longueur de la boite, et le problème devient alors un BP en une dimension, résolue heuristiquement. A l'inverse, l'autre approche heuristique classique consiste pour chacun des objets à d'abord sélectionner la boîte, puis placer l'objet [20]. La sélection de la boîte se fait selon le même type de règle, placement dans la boîte courante, garder une liste de boîte et sélectionner la première où peut loger l'objet, ou alors selon un critère dépendant d'où va pouvoir loger l'objet. Le placement dans la boîte peut se faire ensuite par niveau ou *bottom left*, ou bien tout autre heuristique de placement en deux dimensions.

On remarque que la plupart des heuristiques peuvent être adaptées dans le cas avec rotation. Dans la littérature, de nombreuses variantes d'heuristiques [48, 73, 186, 86, 47, 16, 177, 113] ont été proposées et ne peuvent être toutes détaillées ici. Nous nous focalisons dans la partie suivante sur celle développée dans l'entreprise.

4.2.2 Algorithme existant dans l'entreprise

L'algorithme existant dans l'entreprise est un algorithme ad-hoc utilisant des procédures gloutonnes. Il prend en entrée un ensemble d'objets (avec une largeur, une longueur et les orientations possibles), et commence par calculer une borne inférieure sur le nombre de camions nécessaires comme suit :

$$\max\{\frac{\sum_{i \in I} w_i * l_i}{Y_{MAX} * X_{MAX}}, \frac{\sum_{i \in I} m_i}{m_{MAX}}\}$$

Ensuite, pour chacun des camions, il maintient un ensemble d'objets déjà affectés, et en sélectionne un nouveau à chaque itération qu'il essaie d'affecter à chacun des camions. Pour ce faire, une sous-procédure gloutonne de placement est appelée, avec en entrée, l'ensemble des objets déjà affectés au camion auquel s'ajoute l'objet courant. Finalement, l'objet courant est définitivement ajouté au camion dont le placement minimise une métrique donnée. Si l'algorithme échoue à trouver un plan de chargement pour ce nombre de camions, il recommence en l'incrémentant d'une unité.

La sous-procédure gloutonne est un algorithme de type *best-fit* inspiré de la littérature [27, 170]. L'algorithme commence par trier les rectangles selon une métrique (les objets pouvant être pivotés sont dupliqués comme décrit dans la section 4.1, on parle alors de rectangles), puis initialise ensuite ce qu'on appelle la *ligne d'horizon* du camion. Dans la méthode, cette *ligne d'horizon* est en fait une liste d'entiers, de longueur X_{MAX}, où *ligne d'horizon*$[i] = j$ indique la hauteur totale j de la solution courante au point d'abscisse i. À chaque itération, l'algorithme identifie le segment le plus bas dans la *ligne d'horizon*, c'est à dire les indices consécutifs qui ont la plus faible valeur, et y place le premier rectangle de la liste triée qui loge dans ce segment. Si le rectangle est plus petit que le segment, alors l'algorithme le place sur le coté droit ou le coté gauche du segment selon une politique de placement donnée. Enfin, la *ligne d'horizon* est mise à jour pour refléter ce placement (le rectangle doublon s'il existe est supprimés de la liste). Si aucun rectangle ne loge dans le segment, alors la *ligne d'horizon* est ajustée pour que ce segment soit élevé au niveau de son plus bas voisin. Le processus est répété jusqu'à ce que tous les objets soient insérés.

Finalement, une procédure d'aplanissement est exécutée, elle consiste à retirer l'objet qui donne le métrage linéaire, et à essayer de le réinsérer dans son autre orientation sur le segment le plus bas (après plusieurs ajustements de la *ligne d'horizon* si nécessaire). Cette procédure d'aplanissement est répétée tant que le métrage linéaire diminue.

L'ordre dans lequel les objets sont sélectionnées dans la procédure maître est l'ordre décroissant des surfaces, puis de la masse en cas d'égalité, et enfin de longueur. La sous-procédure de placement est exécutée pour quatre politiques de placement, et six tris d'objets. Les quatre politiques de placement sont :

- Placement à gauche
- Placement à droite
- Placement contre le segment voisin le plus haut

— Placement contre le segment voisin le plus bas

Les tris considérés sont :

— Longueur décroissante des rectangles

— Hauteur décroissante des rectangles

— Surface décroissante

Ces trois tris sont utilisés soit de façon déterministe soit avec une perturbation aléatoire, ce qui conduit à six tris différents au total.

De ces vingt-quatre procédures gloutonnes, la sous-procédure de placement retourne celle qui minimise le métrage linéaire. La procédure maître vérifie alors que cette meilleure solution respecte la hauteur Y_{MAX} et la contrainte de charge aux essieux, et si ce n'est pas le cas elle ne retiendra pas ce camion pour l'affectation finale.

Pour la contrainte de charge aux essieux, cette approche pose problème, car elle impose que la contrainte soit vérifiée à chacune des étapes d'insertion d'un nouvel objet, et non pas pour le chargement final uniquement. La solution utilisée actuellement est alors d'ignorer complètement cette contrainte pendant toute la procédure, et de vérifier a posteriori si la solution retournée par l'algorithme la respecte. Si ce n'est pas le cas, alors l'algorithme est relancé à nouveau en la respectant comme décrit précédemment. En pratique, de nombreuses instances sont sous-contraintes vis à vis de la charge aux essieux, et cette méthode est fonctionnelle, mais certain ensemble d'instances sont beaucoup plus contraintes, et cette méthode montre alors des limites.

Pour chacune des solutions s obtenues, on calcule un indicateur de remplissage qui est le ratio entre la surface définie par la longueur du camion et la hauteur du métrage linéaire de la solution, et la surface des objets dans le camion. Parmi celles qui ont une valeur de ratio inférieure à 1.2, on sélectionne le camion correspondant à la solution qui minimise le métrage linéaire. Si aucune solution n'obtient de ratio inférieur à 1.2, alors on sélectionne le camion qui correspond à la solution minimisant ce ratio.

L'algorithme 6, appelé *BinLoading*, présente la méthode existante. Il prend en entrée un nombre de camions, un ensemble d'objets, une liste de politiques de placement, une liste de fonctions de tri pour la sous-procédure de placement. La liste d'objets est triée, et pour chacun des élément, on teste l'insertion dans les camions. La sous-procédure de placement *BestFit* est ainsi appelée pour chacune des combinaisons de politique de placement et de tri, et le meilleur métrage linéaire est retourné. Si aucune solution n'est admissible, alors l'algorithme finit et il est rappelé avec $N + 1$ camions. Sinon, on affecte l'objet courant au camion selon le ratio de remplissage, défini dans le paragraphe précédent, par la fonction `Selection`. Par simplification, l'algorithme 6 retourne l'affectation finale des objets aux camions mais l'implémention réelle garde en mémoire les placements calculés par la sous-procédure, et retourne alors le chargement de chacun des camions.

L'algorithme de la sous-procédure *BestFit* a servi de base pour notre proposition d'une méthode de recherche arborescente intégrant la contrainte de charge aux

Algorithme 6 : Algorithme existant de remplissage de camion

```
1  Fonction BinLoading(N : Nombre de camion, I : ensemble d'objet, P :
     Politiques de placement, T : Fonctions de trie) :
2      L ← Tri(I)              // Trie par surface, masse puis largeur
3      C ← Liste d'ensemble vide de taille N
4      pour elt ∈ L faire
5          Res ← Liste de taille N
6          pour i ∈ {1, ..., N} faire
7              S ← C[i] ∪ elt
8              Res[i] ← min{BestFit(S, p, t) ∀p, t ∈ P ∗ T}
9          idx ← Selection(Res)
10         si idx = nil alors
11             retourner ⊥
12         C[idx] ← C[idx] ∪ elt
13         RetirerDoublon(L, elt)
14     retourner C
```

essieux. Une description plus détaillée du principe d'utilisation de la *ligne d'horizon* (stockée sous forme d'une liste de segment) sera donnée dans la section 4.3.2.

4.3 Proposition d'une méthode de recherche arborescente

Afin d'améliorer la méthode de résolution existant actuellement et d'intégrer la contrainte de charge aux essieux, nous avons développé une méthode de recherche arborescente. Dans un premier temps, on présente l'approche itérative qui permet de traiter le remplissage des camions séquentiellement. Dans un second temps, on détaille la stratégie de placement des objets, et la stratégie d'exploration. Dans la troisième section, nous discuterons de la complétude de notre méthode de placement.

4.3.1 Approche itérative

On décide d'aborder le problème de chargement en le résolvant de manière itérative, un camion après l'autre. Ainsi, chaque sous-problème se ramène à un problème de placement en deux dimensions avec comme objectif de minimiser la surface des objets exclus du camion considéré. Une fois le sous-problème résolu, on forme une nouvelle instance avec ces objets exclus. On répète ce processus jusqu'à ce qu'aucun objet ne soit exclu, et pour ce dernier sous-problème, l'objectif sera alors de minimiser le métrage linéaire.

L'approche proposée est gloutonne du point de vue des camions, en effet une fois un nouveau sous-problème formé avec les objets exclus, il est impossible de revenir

en arrière pour modifier le placement des objets dans le camion précédent. Cette approche n'est alors pas complète pour le problème de *bin packing*, et résoudre à l'optimal tous les sous-problèmes itérativement n'assure pas que le nombre total de camions soit minimisé. La solution obtenue par notre approche n'est ainsi qu'une borne supérieure du nombre optimal de camions.

La principale motivation de notre approche réside dans la façon dont l'entreprise traite le problème de chargement des camions (et dans les données industrielles à disposition). En effet, la pré-affectation des objets aux camions fait que de nombreuses instances ne nécessite alors qu'un unique camion. La résolution de ce problème permet d'avoir une validation ou infirmation sur le nombre de camions estimé à priori et de pouvoir ré-ajuster différentes décisions logistiques.

Pour le sous-problème de chargement d'un unique camion (que l'on appelle problème de *rectangle packing*), toutes les notations présentées dans la section 4.1 restent valides sauf celles utilisées pour la représentation d'une solution. Une solution de ce sous-problème est caractérisée par un sous-ensemble $\mathcal{O} \in I$ des objets qui sont placés dans le camion, avec pour chaque objet ses coordonnées et son orientation. Les contraintes décrites en section 4.1 ne concernent alors que les objets de ce sous-ensemble.

Le premier objectif considéré correspond à minimiser la surface des objets exclus :

$$z_1(\mathcal{O}) = \sum_{i \in \mathcal{O}_{\text{EXCL}}} w_i * l_i \tag{4.8}$$

avec $\mathcal{O}_{\text{EXCL}} = I \setminus \mathcal{O}$. On remarque que minimiser $z_1(\mathcal{O})$ revient à maximiser la surface des objets de $\mathcal{O}$.

Le second objectif est de minimiser le métrage linéaire lorsque $z_1(\mathcal{O}) = 0$:

$$z_2(\mathcal{O}) = \min \max_{i \in \mathcal{O}} Y(i) + (lw(i) * l_i + (1 - lw(i)) * w_i) \tag{4.9}$$

4.3.2 Stratégies de placement et d'exploration

On propose une méthode de recherche arborescente pour résoudre le problème de *rectangle packing*. Le principe de cette méthode est de transformer en recherche arborescente l'algorithme glouton de *best-fit* [27, 170] utilisé par l'entreprise et décrit en section 4.2.2.

Le but est d'explorer entièrement l'espace des solutions accessibles par cette procédure. Cet algorithme créé une solution à partir d'une permutation des rectangles, et à chaque étape, cherche dans cette permutation, le premier rectangle qui peut loger dans le segment le plus bas. Plutôt que d'explorer toutes les permutations des objets, nous proposons à chaque niveau de l'arbre de créer une branche pour chaque rectangle logeant dans le segment le plus bas. On considère dans cette section que chaque objet non carré pouvant être pivoté a été dupliqué en deux rectangles i, j, et on note $l_i \neq l_j$ leurs longueurs(sur l'axe des abscisses) et $h_i \neq h_j$ leurs hauteurs

(sur l'axe des ordonnées). Lorsqu'un des deux rectangles est placé, le second est supprimé. Bien entendu pour le calcul des bornes et des objectifs, on ne considère que les objets du problème originel et non pas les rectangles dupliqués. En ce qui concerne la politique de placement des objets dans le segment (gauche ou droite), nous proposons de toujours placer l'objet sur la gauche du segment.

La recherche arborescente que nous proposons revient à appliquer une version proche de ce qui a été proposée par Clautiaux et al. [40], en privilégiant le fond de la boîte plutôt que le coté gauche. Privilégier le fond plutôt que le coté permet de respecter plus facilement la contrainte de rang que les auteurs n'intègrent pas. La différence est que notre procédure n'est pas complète et ce point sera discuté dans la section 4.3.3. Rendre notre méthode complète revient à introduire un mécanisme qui rendrait alors les méthodes semblables.

4.3.2.1 Création des branches

L'état courant du niveau de remplissage du camion va être représenté par une succession de *segments* parallèles à l'axe des abscisses, que l'on appelle la *ligne d'horizon* [27, 85]. Plus formellement, on définit :

— un segment comme un triplet (X^l, X^r, Y) tel que $X^l < X^r$, X^l (resp. X^r) $\in [0, X_{MAX}]$, $Y \in [0, Y_{MAX}]$.

— la *ligne d'horizon* comme une séquence de segments $(X_1^l, X_1^r, Y_1), \ldots, (X_n^l, X_n^r, Y_n)$ telle que pour chaque $i \in [2, n]$, $X_{i-1}^r = X_i^l$.

À la racine de l'arbre de recherche, la *ligne d'horizon* est composée d'un unique segment $(0, X_{MAX}, 0)$. À chaque noeud, le segment le plus bas est sélectionné (le plus à gauche en cas d'égalité), c'est à dire le segment s tel que pour tout autre segment t on a $Y_s < Y_t$ ou $Y_s = Y_t$ et $X_s^r < X_t^r$. La recherche d'un tel segment se fait en temps linéaire par rapport au nombre de segments, $O(n)$ dans le pire cas cas. Il est possible d'utiliser un tas binaire pour stocker la *ligne d'horizon* [85], ce qui ferait baisser la complexité d'une telle recherche ($O(log(n))$), cependant, en pratique, le nombre de segments pour notre problème est très faible, et l'implémention d'une telle structure adaptée aux retours-arrière de la recherche en profondeur est une tâche d'ingénierie complexe pour un gain négligeable.

On ajoute alors un noeud fils pour chacun des rectangles encore disponibles i qui loge sur le segment s, c'est à dire si $l_i < X_s^r - X_s^l$. Pour chacune de ces branches, une *ligne d'horizon* différente est alors créée. Pour chaque rectangle, deux cas se présentent :

— Soit le rectangle placé occupe tout le segment s, c'est à dire $l_i = X_s^r - X_s^l$, alors le segment s est remplacé par un nouveau segment $(X_s^l, X_s^r, Y_s + h_i)$ dans la *ligne d'horizon*.

— Soit le rectangle placé n'occupe pas entièrement le segment s, c'est à dire si $l_i < X_s^r - X_s^l$, alors le segment s est remplacé par deux segments dans la *ligne d'horizon* : $(X_s^l, X_s^l + l_i, Y_s + h_i)$ et $(X_s^l + l_i, X_s^r, Y_s)$. Ensuite, les paires de

segments consécutifs (X_i^l, X_i^r, Y_i) et $(X_{i+1}^l, X_{i+1}^r, Y_{i+1})$ telles que $Y_i = Y_{i+1}$ sont fusionnées en un unique segment : (X_i^l, X_{i+1}^r, Y_i).

Exemple *La figure 4.2 illustre ce schéma de branchement. La figure du haut montre l'état d'une solution du problème à un noeud de l'arbre de recherche. Dans cette solution, trois rectangles ont déjà été placés. La ligne d'horizon est composée des trois segments en double barre rouge, le segment le plus bas est pointé par la flèche. Trois rectangles non encore placés peuvent loger dans ce segment, et trois noeuds enfants sont donc créés (dans la figure du bas). Le noeud enfant de gauche place un rectangle qui remplit exactement le segment le plus bas. La ligne d'horizon résultante est alors composée à nouveau de trois segments notés en rouge. Dans le noeud enfant central le rectangle placé ne remplit pas exactement le segment, mais sa hauteur permet de rejoindre le segment situé à gauche. La ligne d'horizon résultante est donc également composée de trois segments. Sur le noeud enfant de droite, le rectangle placé ni ne remplit exactement le segment, ni ne rejoint le segment de gauche. La ligne d'horizon résultante est alors composée de quatre segments. Pour chacun des noeuds enfants, une flèche indique le nouveau segment le plus bas.*

Si aucun rectangle ne peut être placé sur le segment le plus bas (X_i^l, X_i^r, Y_i), alors l'espace correspondant ne peut pas être utilisé. On considère qu'on peut le remplir avec un rectangle fictif ayant les dimensions adéquates et de poids nul : $(X_i^r - X_i^l) * (\min(Y_{i-1}, Y_{i+1}) - Y_i)$. L'espace occupé par ce rectangle fictif sera considéré comme un espace vide dans la solution. On appelle cette procédure un *pont*. Ensuite la *ligne d'horizon* est mise à jour comme décrit précédemment.

Exemple *La figure 4.3 illustre le principe de la procédure de pont. Aucun rectangle ne peut être placé sur le segment le plus bas de la figure de gauche. La procédure de pont est alors exécutée. Elle produit le résultat de la figure de droite, un rectangle fictif (hachuré) est placé, et la ligne d'horizon (en double trait rouge) est alors mise à jour. Le nouveau segment le plus bas est le segment pointé par la flèche.*

Il peut être utile parfois de faire un *pont* même si des rectangles peuvent être placés. On ajoute alors une dernière branche correspondante à ce qu'on appelle un mouvement de *pont*, qui correspond à la procédure décrite au paragraphe précèdent, à condition que $X_s^r - X_s^l < X_{MAX}$, c'est à dire que la taille du segment soit inférieure à la largeur du camion.

Exemple *La solution de la figure 4.4 ne peut être obtenue sans effectuer de pont. En effet, une fois les rectangles* 1 *et* 2 *placés, seul le rectangle 4 peut être placé sur le segment le plus bas qui correspond à l'arête supérieure du rectangle* 2*. Ainsi, dans l'arbre de recherche sans pont, la seule branche créée est celle correspondant au placement du rectangle* 4*. Avec la création de pont, une autre branche sera créée dans laquelle un rectangle fictif sera placé au dessus du rectangle* 2 *permettant ensuite le placement du rectangle* 3.

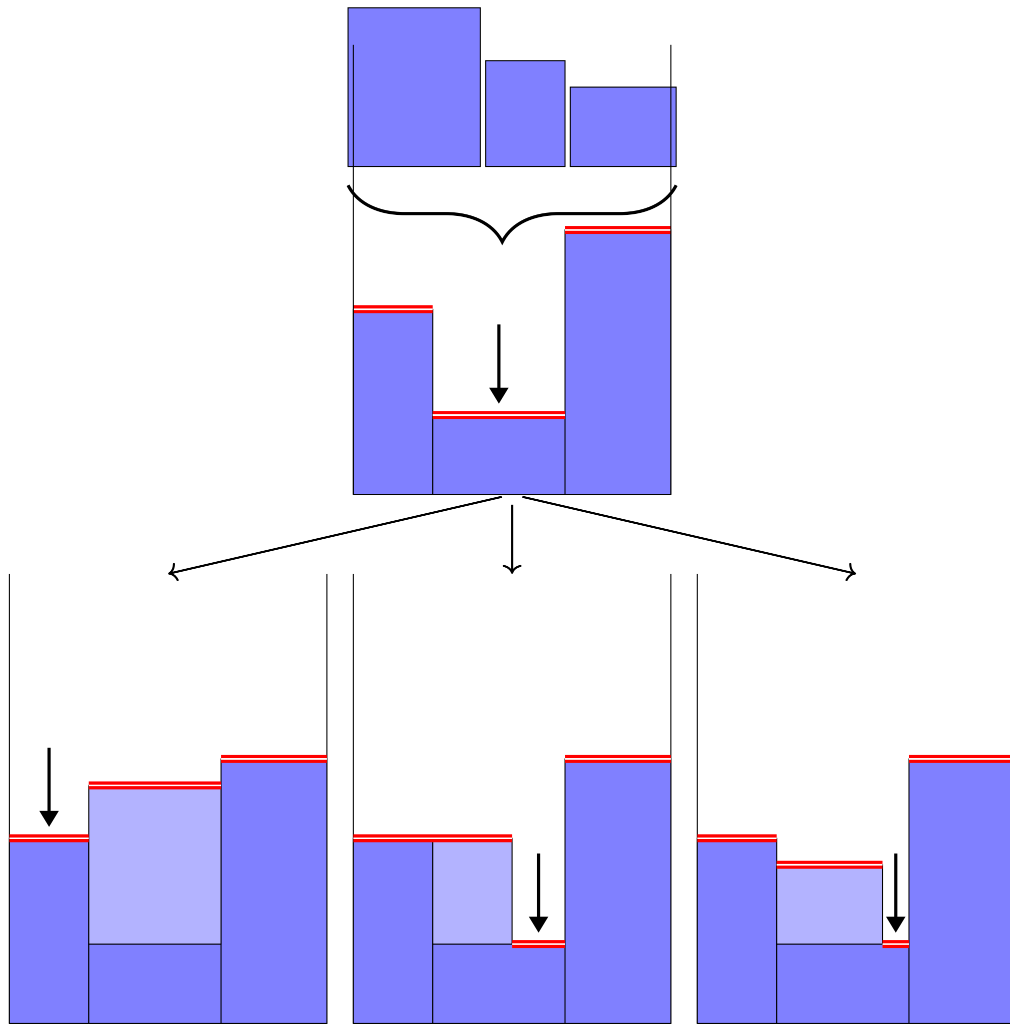


FIGURE 4.2 – Exemple de branchement

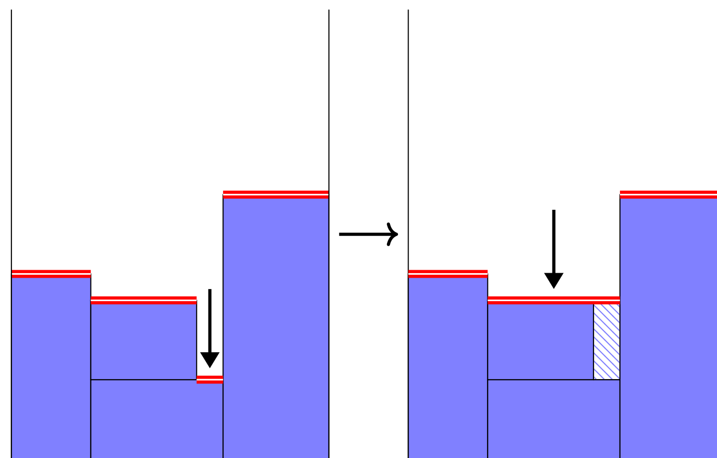


FIGURE 4.3 – Procédure de *pont*

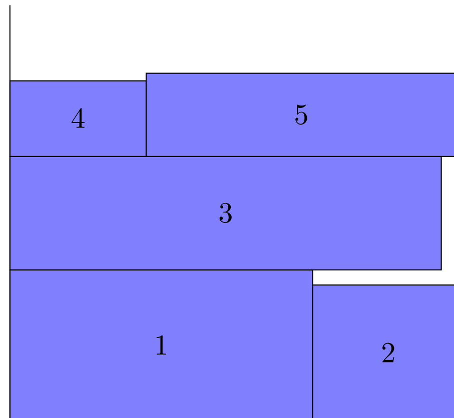


FIGURE 4.4 – Exemple de solution inaccessible sans mouvement de *pont*

Avec ce schéma de branchement, une solution correspond alors à une séquence de rectangles placés selon la règle *bottom-left* et d'actions de *pont*.

Plus généralement, si l'on considère le problème avec rotation dans lequel on doit choisir l'orientation des rectangles, on peut encoder une solution par une séquence σ d'entiers dans laquelle 0 représente le mouvement de *pont*, et pour tout élément $i \in \sigma$ différent de 0, on a $|i| \in I$, avec $i > 0$ si la longueur du rectangle est dans la longueur du camion, et $i < 0$ sinon.

4.3.2.2 Diminution du facteur de branchement

Dans les données industrielles, de nombreux objets ont les mêmes dimensions, rangs et orientations. On peut alors, lors de l'étape de création des noeuds fils, ne considérer qu'un seul rectangle pour chaque orientation au lieu de considérer tous les objets ayant les mêmes caractéristiques. La contrainte sur la masse du chargement (4.5) étant très peu restrictive, on peut également ne considérer qu'un rectangle si plusieurs rectangles ont les mêmes dimensions, rangs et caractéristiques, mais une masse différente, car il est très peu probable de perdre des solutions par une telle procédure. En contrepartie le facteur de branchement s'en trouve très nettement diminué.

4.3.2.3 Vérification des contraintes

Toutes les contraintes, mise à part la contrainte sur le centre de masse (4.5) sont faciles à vérifier au moment même du branchement en ajoutant des conditions sur les rectangles candidats au placement sur un segment donné. La contrainte de chevauchement des paires d'objets (4.2) est directement respectée par la règle de placement décrite dans la section précédente. Pour la contrainte de confinement sur l'axe verticale, il suffit d'écarter les rectangles qui produiraient un nouveau segment ayant une hauteur supérieure à la hauteur maximale autorisée. Pour respecter la contrainte sur le chargement total du camion (4.5), on écarte tout objet qui fait

dépasser la masse maximale autorisé à chaque noeud de l'arbre.

Pour la contrainte de rang des objets, il suffirait, lorsque l'on sélectionne un objet d'un rang k, d'écarter tout les objets de rang inférieur à k. Les données industrielles fournies, décrites en section 4.4.3, comportent un nombre important d'instances où un unique camion est nécessaire. Dès lors, pour ce type d'instances, il est inutile de placer un rectangle d'un rang donné tant qu'un rectangle d'un rang inférieur reste à placer. On remarque, via une expérimentation préliminaire qu'implémenter la contrainte de cette manière (c'est à dire forcer le placement de tous les rectangles d'un rang avant de placer ceux des rangs suivants) est plus performant sur un temps de résolution contraint comme c'est le cas dans l'entreprise. On retiendra alors cette solution pour la suite qui présente également un avantage métier. En effet, un changement de rang est un changement de fournisseur. Si un camion ne contient que des objets d'un unique rang, il peut alors se rendre directement à l'usine sans passer par aucun des autres fournisseurs, ce qui représente un gain financier (à nombre de camions équivalent).

Enfin, la contrainte sur le centre de masse (4.5) ne se comporte pas de façon monotone par rapport au chargement : il est possible qu'un chargement qui respecte cette contrainte ne la respecte plus si on supprimait des objets, parce que le centre de masse se déporterait. De ce fait, vérifier cette contrainte pour une solution partielle n'est pas trivial. Si on ne considère que les objets placés à un noeud v de l'arbre de recherche, la contrainte peut être violée et pourtant il existe un placement dans le sous-arbre enraciné en v qui la satisfait. Néanmoins, à chaque noeud on regarde si la contrainte n'est pas violé pour le chargement actuel. Cette vérification se fait en temps constant amorti en calculant l'équation (4.5). Si elle est respectée pour le chargement, alors on considère ce noeud comme une solution admissible. Il se peut qu'aucune des feuilles de l'arbre explorées ne respecte la contrainte, il faut alors considérer tout chargement partielle respectant cette contrainte comme solution admissible.

4.3.2.4 Bornes inférieures

A chaque niveau de l'arbre, le noeud courant peut être élagué grâce à des bornes inférieures, soit sur la surface des objets exclus, soit sur le métrage linéaire si la meilleure solution courante n'exclut aucun objet c'est à dire si $\mathcal{O}_{\text{EXCL}} = \emptyset$.

Pour la séquence d'objet σ, certains objets ne respectent pas la contrainte 4.3 dans les deux orientations (contrainte de confinement) ou bien ne respectent pas la contrainte 4.5 (contrainte de masse). Soit $E(\sigma)$ l'ensemble des objets exclus de cette façon. On étend cet ensemble en y ajoutant tout les objets, dont un objet de rang inférieur a été exclu :

$$E(\sigma) \leftarrow E(\sigma) \cup \{i \mid \min_{j \in E(\sigma)} r_j < r_i, i \in I\}$$

En effet, comme évoqué précédemment pour traiter la contrainte de rang nous avons choisis de placer d'abord tout les objets d'un rang donné, avant de considérer les

objets des rang suivants.

Soit $LB_{\text{EXCL}} = \sum_{i \in I} l_i * w_i - X_{MAX} * Y_{MAX}$ une borne inférieure sur la surface des objets exclus. On remarque que LB_{EXCL} peut être inférieure à 0. On note $R(\sigma)$ la surface occupée par les rectangles fictifs de la séquence σ, placés par des procédures de *pont*. On obtient alors une borne inférieure $lb_1(\sigma)$ sur la surface des objets qui seront exclus pour une séquence σ donnée :

$$lb_1(\sigma) = \max(LB_{\text{EXCL}} + R(\sigma), E(\sigma)) \tag{4.10}$$

La seconde borne inférieure $lb_2(\sigma)$ est utilisée dans le cas où l'on a déjà trouvé une solution n'excluant aucun objet, et qu'on passe alors sur le second objectif, elle exprime une borne inférieure sur le métrage linéaire :

$$lb_2(\sigma) = \max(cY(\sigma), \frac{\sum_{i \in I} w_i * l_i + R(\sigma)}{X_{MAX}}) \tag{4.11}$$

avec $cY(\sigma)$ le métrage linéaire de la séquence d'objets σ :

$$cY(\sigma) = \max_{i \in \sigma | i \neq 0} (Y(|i|) + (lw(|i|) * l_{|i|} + (1 - lw(|i|)) * w_{|i|})) \tag{4.12}$$

Ces deux bornes sont utilisées pour élaguer l'arbre. Soit $\hat{\sigma}$ la meilleure solution trouvée depuis le début de l'algorithme. A chaque noeud correspondant à la séquence σ, le noeud courant est supprimé si $lb_1(\sigma) > z_1(\hat{\sigma})$. Sinon, si $z_1(\hat{\sigma}) = 0$, alors le noeud sera supprimé si $lb_2(\sigma) > z_2(\hat{\sigma})$. Les bornes inférieures de la littérature [123, 5] n'ont pas été utilisé car elles concernent le problème sans rotation des objets, une procédure de découpage des rectangles en carrés pour utiliser de telles bornes est alors une perspective envisageable pour l'amélioration de notre méthode [49]. Dans la section 4.4.3.2 nous utilisons cependant une procédure, permettant de réduire la taille de X_{MAX} et donc d'améliorer les bornes utilisées.

4.3.3 Caractérisation de l'incomplétude

La méthode proposée dans la section précédente 4.3.2 n'est pas complète pour l'approche considérée (le problème de *Rectangle Packing*). En effet, la règle de placement proposée (plus bas segment et placement à gauche) ne permet pas d'atteindre certaines solutions. De même, la suppression des rectangles identiques en dimensions et en rangs, mais de poids différents peut nous faire perdre des solutions vis à vis des contraintes de masse et de charge aux essieux (contraintes 4.5 et 4.6). Enfin, notre gestion de la contrainte de rang nous empêche également des sélectionner un rectangle lorsque des rectangles de rangs inférieurs sont encore disponibles.

Ces deux dernières raisons, sont des choix délibérés, pour diminuer la taille de l'arbre de recherche, et accélérer la résolution, et peuvent facilement être modifiées dans un but de complétude. Ces deux contraintes sont aussi très liées à l'application industrielle et moins génériques que le problème de placement en deux dimensions que nous considérons dans cette section.

Par soucis de simplification et dans une optique de généricité, dans cette section on ignore alors les contraintes métiers (contraintes de rang 4.4, de masse 4.5 et de charge aux essieux 4.6). De plus, on considère le cas sans orientation du problème, qui correspond pour nous au cas où chaque objet a une orientation forcée (pour tout $i \in I$, $o_i \neq 0$), et pour chaque rectangle i on considère sa hauteur h_i et sa longueur l_i. Les résultats de cette section peuvent facilement s'étendre au cas avec rotation

On s'intéresse dans cette partie au problème de *Strip Packing*, c'est à dire la minimisation du métrage linéaire, dans lequel on place tous les rectangles dans une boite de hauteur infinie, et l'on cherche alors à minimiser le métrage linéaire du placement. On s'intéresse surtout au placement des objets et on veut savoir sous quelles conditions notre procédure énumérative est complète ou non.

Dans ce problème, une solution est un couple $\langle X, Y \rangle$ où est $X(i)$ est la fonction donnant l'abscisse du coin en bas à gauche du rectangle i, et $Y(i)$ son ordonnée.

Il a été montré que les algorithmes qui suivent une règle de placement *bottom-left* (dans lesquels la solution est donnée sous forme d'une séquence où chaque rectangle est placé le plus bas possible, puis calé à gauche) ne sont pas complets, et il existe des instances où aucune permutation ne permet d'atteindre la solution optimale [11]. Un tel contre exemple est montré en figure 4.5. Cette solution est montrée optimale et unique par Baker et al. (aux symétries près) mais elle est inaccessible par une procédure de type *bottom-left*, car l'interstice entre les rectangles 2 et 3 ne peut pas être obtenu. En effet, après avoir placé les deux premiers rectangles (1 et 2), tous les rectangle restant peuvent loger dans le segment à droite du rectangle 2. Si on y met le rectangle 3, il sera alors calé à gauche, contre le rectangle 2, et donc les rectangles 4 et 5 ne pourront pas être placés comme ils le sont actuellement. Le placement de tous les autres rectangles dans ce segment mènent alors à des solutions différentes et non optimales.

Notre procédure produit également une séquence de rectangles à insérer mais toutes les permutations de rectangles ne sont pas visitées. Notre méthode se concentre d'abord sur les espaces disponibles (le plus bas segment) pour le remplir, quand Baker et al. considèrent les approches qui se concentrent d'abord sur la séquence de rectangles, puis sur l'insertion. Par exemple, la solution de la figure 4.4 peut être donnée sous la forme de séquence de rectangles $\langle 1, 2, 3, 4, 5 \rangle$, qui ne peut être produite par notre procédure sans mouvement de *pont*. Cependant, en ajoutant le mouvement de *pont*, on peut obtenir la séquence $\langle 1, 2, B, 3, 4, 5 \rangle$ pour obtenir cette solution, avec B le mouvement de *pont*. Néanmoins, la solution de la figure 4.5 reste inaccessible pour les même raisons que données précédemment : l'interstice entre les objets 2 et 3 ne peut être obtenu.

Dans la prochaine section, nous caractérisons les différents types d'espace vide d'une solution, et montrons sous quelles conditions une solution est accessible avec notre procédure. Ensuite, nous identifions les solutions qui ne sont pas accessibles par notre méthode.

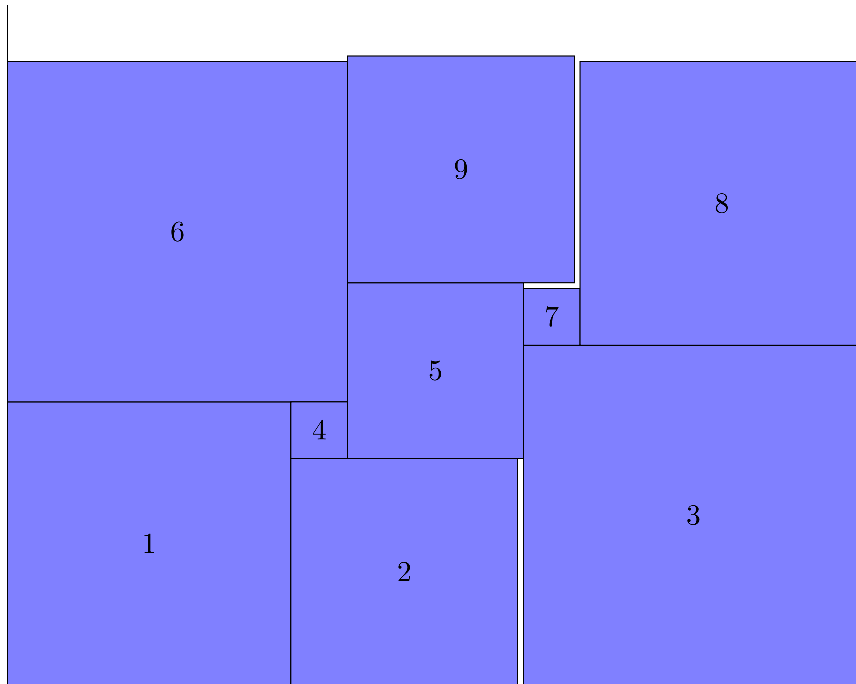


FIGURE 4.5 – Exemple de solution inaccessible [11]

4.3.3.1 Définitions, propriétés

On a besoin pour la suite de définir des notions qui seront utilisées dans cette section seulement. On appelle la procédure de placement décrite dans la section 4.3.2 la procédure LS-B (pour *Lowest Segment - Bridge*).

Définition 1 *Dans une solution, on dit qu'un rectangle i est :*

- *en contact gauche avec un autre rectangle j si $X(j) + w_j = X(i)$ et soit $Y(i) \leq Y(j) < Y(i) + l_i$ soit $Y(j) \leq Y(i) < Y(j) + l_j$.*
- *en contact bas avec j si $Y(j) + l_j = Y(i)$ et soit $X(i) \leq X(j) < X(i) + w_i$ soit $X(j) \leq X(i) < X(j) + w_j$.*
- *en contact droit avec un autre rectangle j si j est en contact gauche avec i.*
- *en contact haut avec un autre rectangle j si j est en contact bas avec i.*

Définition 2 *Un rectangle i est ajusté dans une solution s'il existe au moins un autre rectangle j tel que i est en contact gauche avec j ou $X(i) = 0$, et s'il existe un autre rectangle k tel que i soit en contact bas avec k ou $Y(i) = 0$.*

En d'autre terme, un rectangle est ajusté s'il ne peut être ni déplacé vers la gauche ni déplacé vers le bas sans collision avec un autre rectangle ou sortir des limites.

Définition 3 *Une solution $\langle X, Y \rangle$ est dite ajustée si tous les rectangles sont ajustés.*

Définition 4 *Dans une solution ajustée, on dit qu'un rectangle i a pour support gauche (resp. en bas) un rectangle j si j est le rectangle qui minimise $Y(j)$ tel que que i est en contact gauche (resp. bas) avec j*

En d'autre terme, si j est un support de i, alors le placement de i dépend de j pour la procédure LS-B.

Théorème 1 *Pour toute solution, il existe une solution équivalente ajustée au moins aussi bonne du point de vue du métrage linéaire*

Preuve Soit une solution $\langle X, Y \rangle$ non ajustée. Il existe alors au moins un rectangle i non ajusté, qui peut alors être déplacé soit vers le bas soit vers la gauche. Si l'on ajuste ce rectangle, alors il y a deux cas :

— Soit la nouvelle solution n'est pas ajustée, et ce processus peut alors être répété.

— Soit la nouvelle solution est ajustée. Les rectangles qui ont été déplacés ayant été translatés seulement vers le bas ou vers la gauche, le métrage linéaire de cette nouvelle solution est au moins aussi bon que celui de $\langle X, Y \rangle$. ■

Dans la suite, toutes les solutions sont considérées comme ajustées sauf lorsque le contraire est précisé.

Définition 5 *Soit une solution $\langle X, Y \rangle$. On appelle espace vide tout espace recouvert d'aucun rectangle dont le contour est dessiné soit par les bords de la boite, soit par les arêtes d'autres rectangles.*

On propose de découper les espaces vides par un empilement de rectangles que l'on appelle rectangles vides.

Définition 6 *Tout espace vide est décomposé en un nombre minimal de rectangles vides tel que chacun des rectangles vides soit en contact gauche avec un unique rectangle non vide ou contre le bord gauche, et en contact droit avec un unique rectangle non vide ou contre le bord droit.*

Une autre manière de voir ce découpage est le prolongement les arêtes parallèles à l'axe des abscisses de tous les rectangles qui entourent un espace vide vers l'intérieur de cet espace vide jusqu'à intersection. Un exemple d'un tel découpage est visible dans la figure 4.6. Dans cet exemple, l'espace vide au milieu est découpé en trois rectangles vides en prolongeant les arêtes horizontales de tous les rectangles en contact avec l'espace vide.

On remarque que dans une solution ajustée, un rectangle vide i ne peut être en contact haut qu'avec un seul rectangle vide à la fois, sinon il existerait un rectangle non vide qui serait non ajusté entre ces deux rectangles vides, car il serait alors en contact bas avec seulement i.

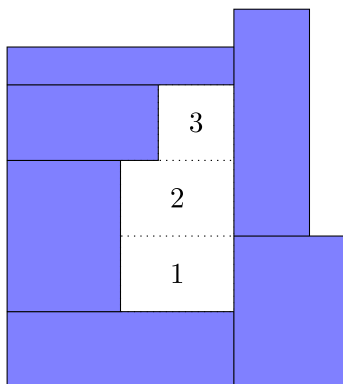


FIGURE 4.6 – Exemple de découpage d'un espace vide pour un sous-ensemble de rectangles d'une solution

On se propose maintenant de caractériser deux types de rectangles vides (appelés *simples* ou *complexes*). Puis, on se pose la question de savoir s'il est possible d'obtenir une solution contenant de tels rectangles avec la procédure de placement LS-B. Pour une solution donnée, on considère ces rectangles à part entière, et donc leurs coordonnées sont également données par les fonctions X et Y, mais ils ne peuvent pas être des supports d'autres rectangles.

Pour une solution avec n rectangles vides obtenus par le découpage explicité ci-dessus, on numérote ces rectangles vides de $I+1$ à $I+n$. Soit p la permutation de $\{1, \ldots, I+n\}$ telle que, pour toutes paires de rectangles consécutifs p_i, p_{i+1} :

$$Y(p_i) < Y(p_{i+1}) \vee Y(p_i) = Y(p_{i+1}) \wedge X(p_i) < X(p_{i+1})$$

.

Théorème 2 *Toute solution ajustée sans espace vide peut être créée par la procédure LS-B.*

Preuve De façon triviale, sans espace vide, pour chaque rectangle p_i on a : soit $X(p_i) = 0$, soit il existe un autre rectangle p_j support gauche de p_i avec $j < i$. De plus, pour chaque rectangle p_i on a : soit $Y(p_i) = 0$, soit il existe un autre rectangle p_j support bas de p_i avec $j < i$. Ainsi, la permutation p peut être placée dans cet ordre par la procédure LS-B pour obtenir la solution désirée. ■

Les lemmes suivants servent à caractériser les rectangles en contact gauche (prop. 1) et en contact droit (prop. 2) des rectangles vides.

Lemme 1 *Soit p_i un rectangle vide et soit p_j un rectangle non vide tels que p_i est en contact gauche avec p_j. On a $i > j$.*

Preuve Par l'absurde, si $i < j$ alors soit $Y(p_i) < Y(p_j)$ et p_i aurait alors un second rectangle à son contact gauche, ce qui est impossible, soit $Y(p_i) = Y(p_j)$ et $X(p_i) < X(p_j)$ ce qui contredirait le fait que p_i soit en contact gauche avec p_j. ■

Lemme 2 *Soit p_i un rectangle vide et soit p_j un rectangle non vide tels que p_i soit en contact droit avec p_j. Si $j > i$ alors $Y(p_i) = Y(p_j)$ et $j = i + 1$*

Preuve Par l'absurde, si $Y(p_i) < Y(p_j)$ alors p_i aurait deux rectangles a son contact droit ce qui par construction n'est pas possible et si $Y(p_i) > Y(p_j)$ alors on a $i > j$. Enfin si $Y(p_i) = Y(p_j)$ et $j > i$, mais que $j > i + 1$, alors p_i ne peut pas être en contact droit avec p_i car on aurait nécessairement p_{i+1} entre les deux. ■

Définition 7 *Un rectangle vide p_i est dit complexe s'il est en contact droit avec un rectangle non vide p_j dont le support gauche p_k est tel que $i < k$. Tout les autres rectangles vides sont dits simples*

En d'autre terme, un rectangle vide complexe apparaît lorsque le support gauche d'un rectangle apparaît après lui dans la permutation p, formant un “porte-à-faux”. Dans l'exemple 4.6, les rectangles vides 2 et 3 sont complexes car le rectangle avec lequel ils sont en contact droit a son support gauche plus haut. Le rectangle vide 1 est un rectangle simple.

Lemme 3 *Si un rectangle vide p_i est simple et en contact droit avec un rectangle p_j alors $i > j$.*

Preuve Supposons que le rectangle vide p_i soit en contact droit avec un rectangle p_j tel que $i < j$. Puisque $i < j$, on a nécessairement $Y(i) \leq Y(j)$. Donc p_j est en contact gauche avec un rectangle non vide p_k tel que $Y(i) < Y(k)$, et par conséquence $i < k$. Par définition, p_i est complexe, ce qui est une contradiction. ■

Théorème 3 *Toute solution ajustée dont les rectangles vides sont tous simples peut être obtenue en suivant la procédure LS-B.*

Preuve Soit $\langle X, Y \rangle$ une solution ajustée contenant uniquement des rectangles vides *simples* et soit p, la permutation des rectangles (vides compris), comme décrit précédemment.

Soit p_i le premier rectangle vide. La solution restreinte aux rectangles $p_1, ..., p_{i-1}$ est accessible en suivant la procédure LS-B car elle ne crée aucun espace vide (théorème 2). Après ce placement, le segment le plus bas débute aux coordonnées $(X(p_i), Y(p_i))$ car :

- soit $X(p_i) = 0$ (resp. $Y(p_i) = 0$) ;
- soit l'unique rectangle en contact droit avec p_i dans la solution (resp. tous les rectangles qui sont en contact haut avec p_i) est dans $p_1, ..., p_{i-1}$ (lemme 1).

On sait alors que ce segment le plus bas est de largeur l_{p_i} car :

- soit $X(p_i) + l_{p_i} = X_{MAX}$;
- soit p_i est en contact droit avec un rectangle p_j tel que $j < i$, par le lemme 3, car p_i est un rectangle vide *simple*.

Finalement, la hauteur du *pont* est le minimum des hauteurs des deux segments voisins, et donc égale à celle du rectangle vide (h_i), par définition.

Par conséquent, le rectangle vide crée par le *pont* correspond exactement au rectangle p_i, et la procédure LS-B obtient le même rangement jusqu'à l'indice i si on remplace le rectangle p_i par le mouvement de *pont*. On peut donc poursuivre ce raisonnement pour le prochain rectangle vide simple. Autrement dit, la séquence égale à p mais où chaque rectangle vide est remplacé par le mouvement de *pont* permet à la procédure LS-B de calculer la solution $\langle X, Y \rangle$. ∎

Théorème 4 *Soit une solution ajustée $\langle X, Y \rangle$. Si cette solution contient au moins un rectangle vide complexe, alors elle ne peut être obtenue par la procédure LS-B.*

Preuve Soit p_i un rectangle vide complexe, et soient p_j le rectangle en contact droit avec p_i, et p_j son support gauche (avec $i < j$ car p_i est *complexe*). L'existence de p_i suppose qu'on ait $Y(p_j) < Y(p_k)$ et donc $j < k$. Or la procédure LS-B est incapable de placer les rectangles dans un ordre différent de celui de p, et donc le rectangle p_j devrait être placé avant son support gauche, ce qui est impossible. ∎

Pour rendre la procédure complète, il faudrait modifier la gestion de la *ligne d'horizon* lorsqu'un mouvement de *pont* est effectué, pour supprimer le segment plutôt que de le rehausser. Puis, lorsqu'un rectangle est placé en porte-à-faux au dessus de ce segment supprimé, le réintégrer dans la *ligne d'horizon* en tronquant la partie sous le rectangle en porte-à-faux. Nous n'avons pas implémenté un tel mécanisme, qui revient alors à appliquer directement l'algorithme proposé par Clautiaux et al. [40]. Dans leur méthode à chaque noeud, une branche est créée pour chaque rectangle qui loge à la position disponible la plus à gauche. L'ensemble des positions correspond au premier point de chaque segment de la *skyline*. Ils ajoutent également une branche dans laquelle cette position ne sera occupée par aucun rectangle. La position est alors désactivée. Un rectangle fictif est ensuite placé lorsque qu'un espace vide est situé au contact gauche avec un rectangle nouvellement placé, créant une nouvelle position.

4.4 Apprentissage par renforcement de la stratégie de branchement

Dans cette section, on applique l'approche proposée dans le chapitre 2 afin d'obtenir une heuristique pour le choix des rectangles dans la procédure LS-B en s'appuyant sur plusieurs critères. La pondération entre ces critères est déterminée avec une méthode d'apprentissage par renforcement. L'heuristique apprise est ensuite intégrée dans une recherche en profondeur.

4.4.1 Processus de décision markovien

Pour le problème à un camion que l'on traite il n'y a pas de modification majeure à apporter pour le considérer comme un MDP. En effet, on construit une séquence

σ de rectangles que l'on place dans l'ordre, et l'environnement nous donne la liste des prochains rectangles admissibles $A(\sigma)$ jusqu'à un état final, c'est à dire lorsque $A(\sigma)$ est vide.

Un état du MDP est défini par une séquence σ de rectangles déjà placés comme décrit précédemment, et les actions $A(\sigma)$ disponibles dans un état σ correspondent à l'ensemble des couples rectangle-orientation qui rentrent dans le segment le plus bas courant. On ignore le mouvement *pont* pour l'apprentissage par renforcement.

En ce qui concerne la récompense on doit légèrement adapter le cadre présenté dans le chapitre 2. D'abord nous devons définir ce qu'est un épisode dans notre problème. Il y a plusieurs choix. Soit nous considérons le problème réel de minimisation du nombre de camions et un épisode se termine lorsque tous les rectangles sont placés, soit nous considérons le problème de minimisation de la surface exclue dans un unique camion, et alors un épisode s'arrête lorsque celui ci est plein. On décide d'utiliser cette seconde option pour être cohérent avec notre recherche arborescente.

Pour homogénéiser les récompenses, on considère la notion d'espace inutile. L'espace inutile d'une solution est tout espace qui n'est pas pas utilisé dans un camion, en dessous du métrage linéaire si $z_1(\sigma) = 0$. Formellement, l'espace inutile $W(\sigma)$ d'une solution décrite par la séquence de rectangles σ est :

$$W(\sigma) = \begin{cases} Y_{MAX} * X_{MAX} - \sum_{i\in\sigma|i\neq 0} l_{|i|} * h_{|i|} & \text{si } z_1(\sigma) > 0 \\ z_2(\sigma) - \sum_{i\in\sigma|i\neq 0} l_{|i|} * h_{|i|} & \text{sinon} \end{cases} \tag{4.13}$$

Minimiser l'espace inutile revient alors à minimiser la surface exclue, ainsi que le métrage linéaire. On utilise alors l'accroissement marginale d'une borne inférieure sur cette objectif comme une pénalité à minimiser. On utilise alors comme borne inférieure $R(\sigma)$ (la surface des rectangles vides de σ), lorsque σ n'est pas une solution complète et la valeur réelle de $W(\sigma)$ lorsque l'épisode est fini. De plus, on normalise les récompenses par la surface d'un camion $X_{MAX} * Y_{MAX}$. La somme des récompenses correspond alors au pourcentage de surface inutile dans un camion. Dans la suite de la section, on considère donc comme objectif la minimisation de l'espace inutile normalisé.

Étant donné qu'un état final d'un épisode peut ne pas être une solution admissible à cause de la contrainte de charge aux essieux, on considère a posteriori que la fin de l'épisode coïncide avec le dernier état valide, c'est à dire la dernière solution admissible rencontrée. Une autre possibilité aurait été d'accorder une importante pénalité en cas de contrainte non respectée, mais il faut alors décider du montant de cette pénalité et décider de quand l'accorder. Avec le choix retenu, la solution retournée sera un camion peu rempli, avec beaucoup d'espace inutile et les valeurs de *gains* associées à cet épisode seront alors mauvaises.

4.4.2 Caractérisation de l'heuristique de branchement

Dans cette section on considère quatre critères pour qualifier un rectangle i dans un état donné défini par σ. On note alors h_i sa hauteur et l_i sa longueur, indépendemment de son orientation.

On définit quatre critères qualifiant un rectangle pour un segment donné. Le premier critère correspond tout simplement au ratio entre la longueur du rectangle, et la longueur du segment. Le second critère est une estimation de la longueur des rectangles que l'on peut espérer placer sur ce segment.

Pour ce faire, un prétraitement est nécessaire. Il consiste à résoudre le problème de la somme de sous-ensembles. Le troisième critère se base sur le centre de masse actuel et un centre de masse cible qui respecterait au mieux cette contrainte dans le pire cas. Enfin, le dernier critère correspond à l'espace vide créé par le placement de l'objet. Ce dernier critère est utilisé pour contrebalancer le premier critère, un rectangle peut être très long par rapport au segment considéré mais très haut, et pourrait créer un *pont* important ensuite.

Les deux premiers critères sont définis par :

$$\lambda_1(\sigma, i) = \frac{l_i}{l(\sigma)} \tag{4.14}$$

$$\lambda_2(\sigma, i) = \frac{\omega(l(\sigma) - l_i) + l_i}{l(\sigma)} \tag{4.15}$$

avec $l(\sigma)$ la taille du plus bas segment du premier point de décision après avoir placé la séquence de rectangles σ (c'est à dire après avoir effectué les *ponts* obligatoires nécessaires). La fonction ω prend en entrée la taille du segment après placement du rectangle i, et retourne une estimation de la longueur qui sera effectivement utilisé dans ce segment. Elle utilise le résultat de la résolution d'un problème de somme de sous-ensemble effectué en prétraitement. L'intuition derrière cette fonction est d'imaginer que l'on tire aléatoirement des rectangles jusqu'à ce que le segment deviennent trop petit pour loger un rectangle, quelle serait alors l'espérance de la longueur des rectangles que l'on aura placé après i. Les détails de cette fonction seront donnés après la présentation des deux critères restants.

Pour le troisième critère, on a besoin d'une estimation sur la position du centre de masse lorsque le camion sera plein. On appelle $\zeta(\sigma)$ cette estimation. Pour la calculer, on considère tout espace vide au dessus de la *ligne d'horizon* comme s'il était parfaitement rempli et équilibré. En d'autre terme, on utilise une relaxation continue du problème comme estimation du centre de masse final. Lorsque $\sigma = \emptyset$, on a $\zeta(\sigma) = Y_{MAX}/2$. A chaque insertion d'un nouveau rectangle, on peut calculer le centre de masse réel des rectangles déjà placés. Pour obtenir la nouvelle valeur de $\zeta(\sigma)$, on doit alors calculer le centre de masse d'un objet factice, occupant toute la place restante du camion. Pour calculer ce centre de masse, on décompose l'objet en une suite de rectangles en considérant pour chacun des segments de la nouvelle

ligne d'horizon, un rectangle d'une longueur égale au segment, et d'une hauteur s'étendant jusqu'à Y_{MAX}. On attribue à cet objet occupant tout le reste du camion une masse proportionnelle au ratio de sa surface par la surface des objets restants à placer dans le camion (ou la masse totale restante si sa surface est supérieure à la surface restante à placer). Soit la surface S d'un tel objet, sa masse M se calcule comme suit :

$$W = \sum_{i \in I \setminus \sigma} m_i * \frac{S}{\max\{\sum_{j \in I \setminus \sigma}(w_j * l_j), S\}}$$

On peut alors calculer $\zeta(\sigma)$ comme le centre de masse de tous les objets du camions et de cet objet factice.

Soit $t = (\phi^-(\sum_{i \in I} m_i) + \phi^+(\sum_{i \in I} m_i))/2$, le centre de l'intervalle admissible du centre de masse si le camion est chargé au maximum, le troisième critère est défini par :

$$\lambda_3(\sigma, i) = (t - \zeta(\sigma)) * (Y(\kappa(\sigma)) + h_i/2 - t) * (d(i) - \underline{d(I)}) \tag{4.16}$$

avec $\underline{d(I)}$ la densité moyenne des objets de I, et $d(i) = h_i * l_i/m_i$ la densité du rectangle i. La valeur t est alors utilisée comme une cible à atteindre pour le centre de masse du chargement.

Ce critère est composé de trois facteurs. Le premier est la différence entre la cible pour respecter au mieux la contrainte, et l'estimation. Le second facteur est la différence entre le centre de masse de l'objet que l'on place sur le segment le plus bas $(Y(\kappa(\sigma)) + h_i/2)$, et le centre de masse ciblé. Le troisième facteur est la différence entre la densité du rectangle, et la densité moyenne des objets. Si l'estimation courante est sous la cible et si le centre de masse de l'objet placé dépasse la cible, alors, on sélectionne en priorité les objets denses. Par contre, si le centre de masse de l'objet est sous la cible également, alors il ne faut pas mettre d'objet dense. A l'inverse, lorsque l'estimation courante dépasse la cible, et le centre de masse de l'objet la dépasse, alors on sélectionne des objets peu denses pour ne pas aggraver l'estimation. Par contre, lorsque le centre de masse de l'objet dépasse la cible, alors on prend des objets denses pour s'en approcher. Pour alléger la notation, on omet la normalisation dans l'équation 4.16. Pour chacun des trois termes, on divise alors par les valeurs extrêmes de chacun de ces trois termes (la plus grande moitié du camion coupée en 2 par t pour les deux premiers termes, et la plus grande valeur absolue de la différence du troisième terme pour celui-ci).

Le dernier critère dépend de la hauteur des segments. On note $Y(\kappa(\sigma))$ la hauteur du segment $\kappa(\sigma)$ du $\kappa(\sigma)$. On note également $\overrightarrow{\kappa}(\sigma))$ le segment à droite du plus bas segment $\overrightarrow{\kappa}(\sigma)$, ce segment est celui qui vient juste après dans la *ligne d'horizon*. Le dernier critère est donc :

$$\lambda_4(\sigma, i) = \begin{cases} -\frac{(l(\kappa(\sigma)) - l_i) * \min\{Y(\kappa(\sigma)) + h_i, Y(\overrightarrow{\kappa}(\sigma))\}}{\min_{j \in I} l_j * h_j} & \text{si } \min_{j \in I} l_j < l(\kappa(\sigma)) - l_i \\ 0 & \text{sinon} \end{cases} \tag{4.17}$$

Le placement du rectangle dans le segment considéré provoque un espace vide si le rectangle le moins large disponible à la prochaine décision ne peut pas être placé dans le reste du segment. On utilise le rectangle le moins long de l'ensemble entier comme une borne inférieure, pour éviter de devoir chercher ce plus petit rectangle disponible à la prochaine action (dépendant des contraintes de rang et de masse). Le numérateur est la surface de cet espace vide, et, le dénominateur la rapporte à la surface du plus petit rectangle de l'instance. On prend l'opposé de ce critère pour garder tous les critères dans le sens de la maximisation, ce critère accorde en fait une pénalité.

Moyenne du remplissage. Nous décrivons dans ce paragraphe comment déterminer la fonction ω nécessaire au calcul du second critère (eq 4.15). Soient $l(\sigma)$, la longueur du plus bas segment après avoir placé la séquence de rectangles σ et i un rectangle de $A(\sigma)$. Après le placement de i, la longueur de ce segment sera $l(\sigma) - w_i$. Si cette longueur est nulle, il n'y a pas d'espace gaspillé, mais il n'est pas forcément souhaitable de minimiser cette valeur. En effet, un espace résiduel plus grand pourrait être plus facile à combler. Décider s'il existe un sous-ensemble de rectangles dont la somme des côtés est égal à une valeur donnée L correspond au problème de la somme de sous-ensembles. Ce problème est NP-complet au sens faible, il peut être résolu de manière efficace par la programmation dynamique en $O(n * L)$ pour n entiers. On pose $L = X_{MAX}$ et on considère le multi-ensemble des longueurs l_i et des largeurs h_i des rectangles $i \in I$.[4] Cette technique permet de raffiner les bornes (eq. 4.10 et 4.11), et a déjà utilisé dans ce but [25].

Pour estimer la densité de solutions pour chaque somme réalisable, on modifie l'algorithme classique de programmation dynamique pour calculer le nombre de solutions pour chacune de ces valeurs.

L'algorithme 7 calcule la même matrice V telle que $V(i, l)$ est la somme maximale inférieure ou égale à L que l'on peut obtenir avec les i plus petits éléments. Il calcule une deuxième matrice D telle que $D(i, l)$ est le nombre de combinaisons d'éléments parmi les i premiers permettant d'atteindre la somme l. Pour un élément unique, il n'y a qu'une solution possible (ligne 3).

— Si le i-ème élément permet de trouver une somme strictement plus grande pour la borne l (ligne 16), alors le nombre de solutions $D(i, l)$ est égal au nombre de solutions sans l'élément i pour la borne résiduelle $l - e(i)$ (ligne 18).

— Si au contraire utiliser cet élément dégrade strictement la somme (ligne 19), alors le nombre de solutions est est le même que lorsqu'on ignore cet élément : $D(i, l) = D(i - 1, l)$ (ligne 21).

— Finalement, si la somme maximale que l'on peut obtenir avec l'élément i est égale à celle que l'on peut obtenir avec les éléments $\{1, \ldots, i - 1\}$ (ligne 22), alors le nombre de solutions est la somme des deux cas (ligne 24).

4. On peut observer qu'il s'agit d'une relaxation du problème réel puisqu'un rectangle peut-être représenté deux fois dans une solution, une fois dans sa longueur, et une fois dans sa hauteur.

Lorsque l'algorithme termine, la dernière ligne de la matrice $D(|I|, \cdot)$ donne le nombre de réalisations de chaque somme dans $[\min_{i \in I} l_i, X_{MAX}]$.

Algorithme 7 : Somme de sous-ensembles

```
    Données : I : Ensemble d'entiers positifs, L : Somme cible
 1  e ← TrieCroissant(I)
 2  V ← Matrice de 0 de taille |I| * L
 3  D ← Matrice de 0 de taille |I| * L
 4  ∀l ∈ {e(1), ..., L} : V(1, l) ← e(1), D(1, l) ← 1
 5  pour i ∈ {2, ..., |I|} faire
 6      pour l ∈ {e(1), ..., L} faire
 7          si e(i) > l alors
 8              V(i, l) ← V(i − 1, l)
 9              D(i, l) ← D(i − 1, l)
10          sinon
11              lw ← e(i)
12              nw ← 1
13              si e(i) + e(1) ≤ l alors
14                  lw ← e(i) + V(i − 1, l − e(i))
15                  nw ← D(i − 1, l − e(i))
16              si lw > V(i − 1, l) alors
17                  D(i, l) ← nw
18                  V(i, l) ← lw
19              sinon si lw < V(i − 1, l) alors
20                  D(i, l) ← D(i − 1, l)
21                  V(i, l) ← V(i − 1, l)
22              sinon
23                  D(i, l) ← D(i − 1, l) + nw
24                  V(i, l) ← lw
25  retourner V, D
```

Cet algorithme est exécuté avant la résolution, et on dispose alors des tables V et D pour la fonction ω du second critère :

$$\lambda_2(\sigma, i) = \frac{\omega(l(\sigma) - l_i) + l_i}{l(\sigma)} \tag{4.18}$$

$$\omega(L) = \begin{cases} \frac{\sum_{l \in T(L)} V(|I|,l) * D(|I|,l)}{\sum_{l \in T(L)} D(|I|,l)} & \text{si } T(L) \neq \emptyset \\ 0 & \text{sinon} \end{cases} \tag{4.19}$$

$$T(L) = \{l \in \mathbb{N} \mid V(|I|, l) = l,\ L - l_{\bar{i}} < V(|I|, l) \leq L\} \tag{4.20}$$

avec $\bar{i}$ le rectangle le moins long encore disponible au noeud de l'arbre considéré.

L'ensemble T permet de ne sélectionner que les sommes tels que l'espace résiduel laissé après le placement des rectangles constituant cette somme soit moins large que le plus petit rectangle disponible. On calcule ensuite une moyenne en pondérant par la densité. On obtient alors une estimation de la longueur des rectangle que l'on peut espérer placer sur ce segment.

Une fois ces critères calculés pour un rectangle i et pour une séquence donnée de rectangles σ, on définit la fonction d'évaluation $g_{\boldsymbol{\theta}}(\sigma, i)$, tel que décrit dans le chapitre 2 :

$$g_{\boldsymbol{\theta}}(\sigma, i) = \boldsymbol{\theta}^{\intercal} \boldsymbol{\lambda}(\sigma, i) \tag{4.21}$$

Une fois cette valeur calculée pour chacun des rectangles possibles de $A(\sigma)$, on calcule la distribution de probabilités avec la fonction `softmax` :

$$\pi_{\boldsymbol{\theta}}(i \mid \sigma) = \frac{e^{g_{\boldsymbol{\theta}}(\sigma,i)/\beta}}{\sum_{j \in A(\sigma)} e^{g_{\boldsymbol{\theta}}(\sigma,j)/\beta}} \forall i \in A(\sigma) \tag{4.22}$$

Pour définir un ordre d'exploration des branches de la recherche arborescente on va tirer aléatoirement un des éléments selon cette distribution, puis trier les différentes actions selon $\pi_{\boldsymbol{\theta}}$, et échanger le premier élément de cette liste triée, avec l'élément tiré aléatoirement. Pour intégrer les actions de *pont*, on ajoute ensuite l'action lorsqu'elle est disponible comme la dernière branche explorée.

Validation des critères. Les critères que nous proposons constituent chacun une règle que l'on pourrait suivre seule pour guider la recherche. On peut valider que ces critères sont pertinents en regardant quelle serait la performance d'une procédure gloutonne ne suivant qu'un unique critère de manière déterministe. On peut alors comparer cette valeur à la performance d'une procédure gloutonne qui sélectionne les rectangles aléatoirement.

La table 4.2 montre les résultats obtenus sur le jeu de données `Son` (présenté en section 4.4.3 et comportant 1166 instances). Pour cette table, on arrête l'exécution lorsque le premier camion est rempli pour chaque instance. On garde alors la meilleure solution admissible, respectant donc la contrainte sur le centre de masse, comme décrit dans la section 4.4.1. Chaque colonne correspond à l'espace vide normalisé moyen en suivant un unique critère comme heuristique, la première colonne (rnd) étant l'objectif obtenu avec une sélection aléatoire, moyenné sur 10 exécutions et sur l'ensemble des instances. La colonne λ_i correspond à l'heuristique déterministe qui maximise à chaque étape le i-ème critère décrit dans cette section. La table 4.3 montre la procédure opposée, c'est à dire l'objectif moyen en suivant l'opposé de chacun des critères, c'est à dire que l'on sélectionne le rectangle qui minimise le critère considéré.

On remarque dans la table 4.2 que chacun des critères donne en moyenne une solution qui est meilleure que l'aléatoire, et donc qu'ils semblent tous pertinents. Également, si l'on suit l'opposé du critère, les résultats sont moins bons qu'un tirage

	rnd	λ_1	λ_2	λ_3	λ_4
Son	0.174	0.113	0.118	0.121	0.154

Table 4.2 – Comparaison de chaque critère contre de l'aléatoire

	rnd	$-\lambda_1$	$-\lambda_2$	$-\lambda_3$	$-\lambda_4$
Son	0.174	0.185	0.216	0.465	0.191

Table 4.3 – Comparaison de chaque critère contre de l'aléatoire

aléatoire. Cela veut alors dire que ces critères ont un réel impact sur l'objectif considéré.

Enfin, pour le troisième critère, on remarque dans la table 4.3 que suivre l'opposé de ce critère augmente fortement l'objectif pour l'ensemble d'instances `Son`. Ce critère vise à faire respecter la contrainte de charge aux essieux, on voit ici que suivre l'opposé du critère mène alors à violer cette contrainte, et donc obtenir des camions peu remplis.

4.4.3 Résultats expérimentaux

Dans cette section, nous commençons par décrire les instances utilisées. Puis, nous évaluons la qualité de l'heuristique de choix des rectangles obtenue avec l'algorithme REINFORCE. Enfin, nous comparons les résultats d'une méthode arborescente basée sur cette heuristique avec ceux produits par la méthode implémentée dans l'entreprise.

4.4.3.1 Instances

Nous disposons de trois jeux d'instances industrielles. Ces trois jeux de données sont composés respectivement de 1171, 1190 et 7130 instances du problème de *bin packing* originel et concerne trois usines de l'entreprise que nous appelons respectivement `M`, `D` et `S`.

La difficulté de ces instances vient surtout du très court budget de temps à disposition pour les résoudre. En effet, la résolution de ce problème intervient dans un algorithme de résolution plus large et le placement en deux dimensions n'est qu'une brique dans le processus de planification de l'entreprise. De plus, la faible diversité dans les dimensions des objets fait que notre traitement des objets de même dimensions pour réduire la taille de l'arbre de recherche est très efficace. La table 4.7 référence les caractéristiques de ces trois jeux d'instances. On y trouve pour chacune des usines le nombre d'instances, le nombre moyen d'objets de chaque instance ($\overline{I}$), le nombre moyen d'objets de dimensions différentes par instance ($\overline{D}$), et le ratio de surface entre l'objet qui a la plus grande surface, et l'objet qui a la plus petite ($\frac{\max_{i \in I} w_i * w_i}{\min_{i \in I} w_i * w_i}$) que l'on note $\overline{R}$. On note que ces instances apparaissent alors être de petites tailles, d'autant plus pour notre stratégie de recherche. En effet, notre facteur de branchement correspond à la colonne $\overline{D}$ que l'on multiplie

Usine	Nb instances	LB	$\overline{I}$	$\overline{D}$	$\overline{R}$	CPU
M	1171	1244	14.61	1.43	1.07	38s
S	7130	9556	13.61	2.27	2.26	136s
D	1190	1573	15.30	3.39	2.06	34s

FIGURE 4.7 – Caractéristiques des trois jeux d'instances industrielles

Usine	Nb instances	LB	$\overline{I}$	$\overline{D}$	$\overline{R}$
Son	1166	4269	87.02	11.39	10.95

FIGURE 4.8 – Caractéristiques du jeu d'instances générées

par deux (pour les rotations) auquel on ajoute le mouvement de *pont*. La profondeur de l'arbre, si on ignore le mouvement de *pont* correspond alors à la colonne $\overline{I}$. On indique également une borne inférieure (LB) sur le nombre de camions nécessaires pour l'ensemble des instances. Cette borne est obtenue en divisant la surface totale des objets par la surface d'un camion, réduite en utilisant le résultat produit par l'algorithme de somme de sous-ensembles sur les deux dimensions.

Sur ce grand nombre d'instances, on distingue des instances triviales, des instances faciles, et quelques instances difficiles. Le temps en secondes accordé actuellement par l'entreprise pour obtenir une solution pour chacun de ces trois jeux d'instances est indiqué dans la dernière colonne (CPU). Ce temps correspond au temps total pour résoudre toutes les instances d'une usine, et non pas un temps par instance. Ce n'est pas une limite stricte mais cela nous donne l'ordre de grandeur pour définir le budget à accorder à nos algorithmes.

Nous avons également généré un ensemble supplémentaire d'instances plus variées à partir de celles de `S` que l'on nomme `Son`. Pour créer ce jeu d'instances, on combine à chaque fois plusieurs instances de `S` dont les objets sont destinés à parcourir le même trajet en une seule journée. Le résultat est qu'il y a moins d'instances mais avec plus d'objets. Ensuite, ces objets sont, soit laissés tels quels, soit coupés en plusieurs. Les objets peuvent être coupés en deux soit dans le sens de la longueur, soit dans le sens de la largeur, au milieu, soit les deux, pour produire chacun 2 ou 4 objets, puis la masse de l'objet initiale est répartie aléatoirement entre ces 2 ou 4 objets. Le tout est un processus aléatoire.

L'orientation des objets créés est la même que l'orientation de l'objet initial dans les jeux d'instances. Enfin, certaines instances restant trop peu variées, ou comportant trop peu d'objets, ont été ensuite retirées. Les instances de `Son` sont alors plus variées que les instances industrielles initiales, tout en gardant des tailles assez standardisées pour certains objets, et de la répétition dans les dimensions des objets. L'objectif est de venir augmenter le facteur de branchement de notre arbre de recherche. Les caractéristiques de cet ensemble d'instances sont résumées dans le tableau 4.8.

4.4.3.2 Evaluation de l'algorithme d'apprentissage REINFORCE

Dans cette section, on applique l'algorithme REINFORCE pour régler les poids $\boldsymbol{\theta}$ de la combinaison linéaire de la fonction d'évaluation.

On propose d'apprendre un jeu de paramètres $\boldsymbol{\theta}$ sur chacun des ensembles d'instances `Son`, `M`, `S` et `D`. Chacun de ces ensembles est séparé en deux pour former un ensemble d'apprentissage et un ensemble de test de même taille. Puis, pour évaluer la pertinence du réglage de $\boldsymbol{\theta}$ sur un sous-ensemble de données non vues lors de l'apprentissage, on regarde la valeur moyenne de l'objectif considéré lors de l'apprentissage (l'équation 4.13) en construisant des solutions avec les poids obtenus après apprentissage. On compare alors cette valeur aux valeurs obtenues avec une sélection aléatoire, ou une sélection déterministe basée sur la maximisation d'un unique critère.

On exécute l'algorithme REINFORCE sur chaque sous-ensembles d'apprentissage séparément sur 15000 itérations chacun, avec le paramètre $\gamma = 1$. Pour rappel, ce paramètre est un coefficient exponentiellement décroissant dans la prise des récompenses futures pour le calcul du *gain* à une étape donnée. Dans ce problème, un épisode étant relativement court, ce coefficient n'est pas pertinent. Une itération de l'algorithme correspond à la génération d'un épisode pour chacune des instances de l'ensemble d'apprentissage puis à la mise à jour du paramètre $\boldsymbol{\theta}$ en fonction de ces épisodes tel que décrit dans le chapitre 2. A chaque début d'itération, $\boldsymbol{\theta}$ est normalisé tel que $\sum_i \boldsymbol{\theta}_i = 1$, et au début du processus les poids sont initialisés à 1 (puis normalisés à la première itération à 0.25 donc). Les 15000 itérations prennent entre 1h et 2h à être exécutées sur un Intel(R) Core(TM) i7-8700 CPU @ 3.20GHz.

On s'intéresse en détail aux résultats pour l'ensemble `Son`. Quatre valeurs pour le pas d'apprentissage α ont été testées et la température β est fixée à 0.1. Dans la figure 4.9, on présente les résultats de cet apprentissage pour ces quatre valeurs de α : $\{0.1, 0.01, 0.001, 0.0001\}$. L'algorithme a été exécuté via un outil à vocation générique développé dans le cadre de ces travaux, qui prend le problème de l'apprentissage en maximisation. On utilise alors l'opposé des pénalités définies précédemment à maximiser.

Pour cette figure, on a récolté la valeur de l'objectif moyen à chaque itération, que l'on montre ici sous la forme de moyenne glissante de taille 50, affichant donc le premier point de chaque courbe pour l'itération 24, moyennant l'espace inutile de l'itération 0 à l'itération 49.

Au regard de cette figure, si l'apprentissage semble bien fonctionner il est à relativiser par l'échelle de l'ordonnée utilisée ici qui va de -0.11 à -0.098. En effet, à l'initialisation du processus, l'espace inutile moyen est aux alentours de $11,2\%$, ce qui est déjà bien meilleur que les $17,4\%$ obtenus par sélection aléatoire (visible à la table 4.5 pour l'ensemble d'entraînement). Au mieux, cette valeur atteint environ $9,8\%$. La progression est alors seulement de $1,4$ points. Néanmoins, on note l'impact du paramètre α dans ce processus. Pour une valeur trop petite $\alpha = 10^{-4}$, le processus peine à apprendre. La différence entre $\alpha = 10^{-1}$ et $\alpha = 10^{-2}$ est la vitesse à laquelle on atteint ce qui semble être un plateau en terme d'objectif moyen.

α	Moyenne	écart type
0.1	-0.098692	$2386.4141 * 10^{-6}$
0.01	-0.098675	$2350.3321 * 10^{-6}$
0.001	-0.098747	$2359.7206 * 10^{-6}$

TABLE 4.4 – Moyenne et écart type sur les 5000 dernières itérations pour différentes valeurs de α (sur 50000 itérations pour $\alpha = 0.001$), sur l'ensemble d'apprentissage `Son`

Enfin, pour $\alpha = 10^{-3}$ le plateau n'est pas atteint en moins de 15000 itérations. Il est atteint au bout d'environ 30000 itérations sans le dépasser néanmoins (résultat non présenté).

Dans la figure 4.4, on regarde plus en détail l'objectif obtenu en fin d'apprentissage pour les trois valeurs de α qui convergent vers le plateau (pour $\alpha = 0.001$ on s'appuie sur un apprentissage jusqu'à 50000 itérations non montré ici). On montre la moyenne de l'objectif des 5000 dernières itérations, ainsi que l'écart type. On peut voir que l'objectif moyen est sensiblement le même. On pourrait également penser qu'une convergence plus lente augmente légèrement la stabilité une fois ces valeurs obtenues. Cependant la différence d'écart type pour $\alpha = 0.1$ et $\alpha = 0.01$ n'est que de $3.6 * 10^{-5}$ en faveur du second. L'écart type repart très légèrement à la hausse lorsque le pas d'apprentissage est plus petit, une différence de $9.4 * 10^{-6}$. Ces faibles valeurs d'écart type sont à rapporter à la faible différence entre la valeur d'objectif en début d'apprentissage et la valeur moyenne finale.

Finalement, les différentes valeurs de α semblent alors indiquer que, pour ce problème, seul le temps du processus pour arriver à un minimum local est modifié lorsque ce paramètre change, sans changer fortement la valeur de ce minimum local, ni même la stabilité lorsque l'on y arrive.

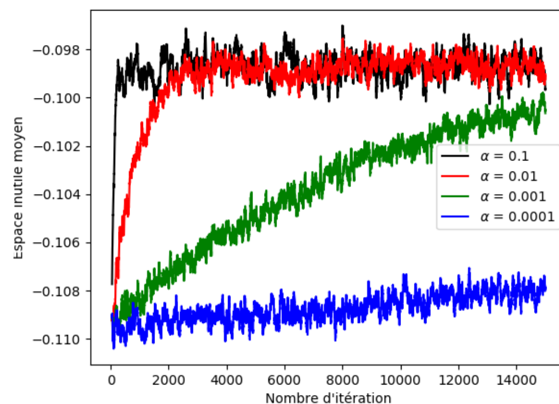


FIGURE 4.9 – Moyenne glissante sur 50 itérations de la valeur moyenne de l'espace inutile sur l'ensemble d'apprentissage de `Son`

Pour la suite, on choisit la valeur de $\boldsymbol{\theta} \approx (0.55, 0.19, 0.08, 0.19)$ obtenue lors

	rnd	$\boldsymbol{\theta}^*$	$\boldsymbol{\theta}^1$	$\boldsymbol{\theta}^2$	$\boldsymbol{\theta}^3$	$\boldsymbol{\theta}^4$
$\texttt{Son}^e$	0.178	**0.100**	0.125	0.154	0.140	0.162
$\texttt{Son}^t$	0.168	**0.096**	0.116	0.149	0.138	0.157

TABLE 4.5 – Comparaison entre une sélection aléatoire, et une sélection basée sur différentes valeurs de $\boldsymbol{\theta}$

de l'apprentissage avec $\alpha = 0.01$ et pour laquelle la valeur d'objectif moyen a été minimale sur une itération de l'algorithme (pour une valeur de 9.0%).

Dans la table 4.5, on compare la valeur moyenne d'objectif obtenue sur une sélection aléatoire avec la valeur moyenne d'objectif obtenue pour différentes valeurs de $\boldsymbol{\theta}$, et ce pour l'ensemble d'entraînement et l'ensemble de test de `Son` que l'on nomme respectivement $\texttt{Son}^e$ et $\texttt{Son}^t$.

La température β utilisée est de 0.1. La première colonne (rnd) donne l'objectif moyen pour une sélection aléatoire. La colonne $\boldsymbol{\theta}^i$ donne l'objectif moyen pour une valeur de $\boldsymbol{\theta}$ avec un 1 pour le i-ème critère et des 0 sinon. Enfin, la colonne $\boldsymbol{\theta}^*$ donne l'objectif moyen obtenu pour la valeur de $\boldsymbol{\theta}$ retenue après apprentissage. Pour chaque ensemble d'instances, et pour chaque colonne, le résultat affiché est une moyenne sur 10 exécutions. Cette table nous montre que la combinaison linéaire des quatre critères, réglée via l'algorithme REINFORCE forme une meilleure heuristique que chacun des quatre critères pris séparément. On voit également que ce réglage fait sur un ensemble d'apprentissage fonctionne bien sur un ensemble de test que l'algorithme n'a jamais vu. Ces deux ensembles d'instances ont néanmoins des caractéristiques semblables, provenant du même ensemble à la base mais notre modèle étant très simple avec très peu de paramètres, il semble peu probable de tomber dans le travers du surapprentissage.

On répète le processus pour le jeu d'instances industrielles avec $\alpha = 0.01$. Il est à noter que pour ces instances, les critères semblent moins pertinents, en particulier le premier critère. En effet, de nombreuses piles sont formatées de telle sorte qu'elles peuvent rentrer par paire dans la largeur du camion. En sélectionnant la pile qui maximise l'occupation du segment, on risque d'orienter ces piles d'une façon qui ne permet plus de former ces paires.

Le processus d'apprentissage pour ces trois ensembles d'instances industrielles semble moins efficace que pour `Son`. La valeur initiale de l'objectif est très proche de la valeur finale qui est atteinte en une poignée d'itérations, et pour `M`, il n'y a aucune différence entre ces valeurs. On observe sur la table 4.6, les résultats issus de cet apprentissage. Comme pour la table précédente, on compare différentes valeurs de $\boldsymbol{\theta}$ sur chacun des ensembles d'apprentissage et de test, $\boldsymbol{\theta}^i$ représentant une valeur de $\boldsymbol{\theta}$ avec un 1 pour le i-ème critère et des 0 sinon, et $\boldsymbol{\theta}^*$ la valeur de $\boldsymbol{\theta}$ apprise pour l'ensemble d'instances considérées. On donne également les valeurs obtenues si on utilise le $\boldsymbol{\theta}$ appris sur `Son`, noté $\boldsymbol{\theta}^{\texttt{Son}}$. On observe que, pour les instances de `M`, l'heuristique obtenue n'est pas meilleure qu'une heuristique suivant seulement le troisième critère. Cet ensemble d'instances est effectivement plus difficile du point de vue de la contrainte de charge aux essieux. Cependant, pour les instances de `S`

	rnd	$\boldsymbol{\theta}^*$	$\boldsymbol{\theta}^{\texttt{Son}}$	$\boldsymbol{\theta}^1$	$\boldsymbol{\theta}^2$	$\boldsymbol{\theta}^3$	$\boldsymbol{\theta}^4$
$\texttt{M}^e$	0.255	**0.236**	0.237	0.257	0.257	**0.236**	0.255
$\texttt{M}^t$	0.231	**0.215**	0.216	0.233	0.232	0.216	0.232
$\texttt{S}^e$	0.119	**0.085**	0.090	0.124	0.101	0.113	0.100
$\texttt{S}^t$	0.123	**0.088**	0.090	0.127	0.106	0.115	0.106
$\texttt{D}^e$	0.132	**0.084**	0.095	0.139	0.093	0.122	0.097
$\texttt{D}^t$	0.130	**0.081**	0.093	0.150	0.093	0.122	0.094

TABLE 4.6 – Comparaison entre une sélection aléatoire, et une sélection basée sur différentes valeurs de $\boldsymbol{\theta}$ - Instances industrielles

et de D, les deux $\boldsymbol{\theta}^*$ obtenus après apprentissage sont pertinents et se généralisent bien sur l'ensemble de test. On note également une petite différence entre $\boldsymbol{\theta}^{\texttt{Son}}$, et les $\boldsymbol{\theta}^*$ des deux ensembles d'instances, et plus particulièrement sur l'ensemble D.

4.4.3.3 Comparaison de méthodes

Dans cette section, on compare la recherche arborescente décrite dans la section 4.3.2 (que l'on note `DFS`) et la méthode utilisée dans l'entreprise décrite en section 4.2.2 (que l'on note `BinLoading`), sur les trois ensembles d'instances industrielles ainsi que sur l'ensemble d'instances générées `Son`. On traite ici du problème de *bin packing*, avec pour objectif premier de minimiser le nombre total de camions sur un ensemble d'instances.

Instances industrielles. Nous proposons de comparer trois versions de notre recherche arborescente, chacune correspondante à trois valeurs de $\boldsymbol{\theta}$:

- `DFS-0` : Correspond à une ordre aléatoire des objets à chaque point de décision ($\boldsymbol{\theta}^\intercal = (0, 0, 0, 0)$)
- `DFS-*` : Correspond à la valeur de $\boldsymbol{\theta}$ apprise spécifiquement sur un sous-ensemble de l'ensemble d'instances considérées (décrit dans la section 4.4.3.2)
- `DFS` : Correspond à la valeur de $\boldsymbol{\theta}$ apprise sur un sous-ensemble d'instances de `Son` (décrit dans la section 4.4.3.2)

Chacune de ces méthodes a été exécutée 10 fois sur chacune des instances sur un cluster composé de Intel(R) Xeon E5-2695 v3 @ 2.30GHz et de Intel(R) Xeon E5-2695 v4 @ 2.10GHz. Elles ont été implémentées en C++ et compilées avec GCC-9.4. La méthode `BinLoading` n'a été lancée qu'une fois sur un Intel(R) Core(TM) i7-8665U CPU @ 1.90GHz.

La façon dont nous décomposons le problème fait qu'il est difficile de donner un temps limite précis pour résoudre une instance. On propose alors de donner un budget de conflits par camion. L'objectif est de paramétrer ce budget pour obtenir des temps de calcul proches de ceux considérés par l'entreprise. Ce temps de calcul est atteint avec une limite fixée à 25000 conflits. On rappelle que l'algorithme utilisé par l'entreprise ne s'appuie que sur des procédures gloutonnes et est donc très rapide.

Les résultats sont présentés dans la table 4.10. Pour cette table, le paramètre de température a été réglé à $\beta = 0.1$, et une stratégie de redémarrage géométrique est ajoutée sur la base d'un nombre de conflits démarrant à 100 avec un facteur de 1.2. Ces paramètres ont été réglés via l'observation d'expériences préliminaires.

Dans cette table, pour chaque ensemble d'instances (`M`, `S` et `D`) on rappelle le nombre total d'instances (NB), et une borne inférieure sur le nombre total de camions (LB). Pour chacune des méthodes et par ensemble d'instances, on fournit comme résultat, le nombre total de camions (#C), la somme des espaces inutilisés dans les camions (W), (sous le métrage linéaire pour le dernier camion comme défini dans l'équation 4.13). Cet indicateur est normalisé pour chaque instance par la surface du camion, ce qui fait que le nombre affiché dans la table peut être interprété comme un nombre de camions qui circuleraient à vide, chacun des autres camions étant parfaitement chargés sans aucun espace vide. #M indique la valeur des deux objectifs agrégés (eq. 4.7), c'est à dire la longueur totale des camions mis bout à bout, coupé au métrage linéaire du dernier. On donne la somme sur toutes les instances de la ligne en mètres. Ces deux derniers indicateurs donnent deux regards différents sur le taux de remplissage des camions (en dessous du métrage linéaire pour le dernier camion). Enfin, on donne, pour chaque ensemble d'instances, le temps total de résolution (CPU).

On remarque globalement que les trois versions de l'algorithme de recherche arborescente surpassent la méthode utilisée dans l'entreprise, dans un temps comparable. Également, de manière surprenante `DFS` domine légèrement `DFS-*` en nombre de camions, cette différence est cependant négligeable sur le nombre d'instances considérées. L'ordre d'exploration des différentes branches a tout de même un impact, bien que modéré, au vu des résultats systématiquement moins bons obtenus par l'ordre aléatoire (`DFS-0`). Enfin, on peut chiffrer l'amélioration obtenue par notre meilleure méthode par ensemble d'instances. Pour les instances de `M`, le nombre de camions a baissé de 18%, et le taux de de remplissage des camions ($1 - W/\#C$) est passé de 74% à 83% (avec `DFS-*`). Pour les instances de `S`, le nombre de camions a baissé de 5% et le taux de remplissage est passé de 93% à 96%. Enfin pour `D`, le nombre de camions a baissé de 2% et le taux de remplissage est passé de 94% à 96%. On remarque alors une amélioration significative sur les instances `M` et une amélioration seulement modeste pour les deux autres jeux d'instances. Cela s'explique par le fait que l'ensemble d'instances `M` est difficile du point de vue de la contrainte sur le chargement aux essieux, et l'algorithme `BinLoading` mis en place dans l'entreprise ne gère pas très bien cette contrainte. On rappelle les valeurs des bornes inférieures sur le nombre de camions pour ces ensembles d'instances qui sont de 1244, 9556 et 1573 pour `M`, `S` et `D` respectivement, ne nous laissant finalement qu'une marge de progression peu importante, excepté sur `M`.

Instance alternative. On compare maintenant notre recherche arborescente avec la méthode de l'entreprise sur le jeux d'instances générées `Son`. Les conditions d'expérimentation sont identiques, on montre dans la table 4.8 les résultats de notre

Cl	NB	LB	BinLoading				DFS-*				DFS				DFS-0			
			#C	W	#M	CPU	#C	W	#M	CPU	#C	W	#M	CPU	#C	W	#M	CPU
M	1171	1244	1865	493	15127	38	1525	**266**	**12067**	48	**1521**	268	12089	47	1558	272	12151	50
S	7130	9556	10237	762	94629	136	9747	388	89578	195	**9736**	**387**	**89563**	189	9867	420	90015	200
D	1190	1573	1643	98	13893	34	1605	**60**	13379	46	**1604**	**60**	**13374**	47	1622	70	13511	47

TABLE 4.7 – Comparaison des méthodes sur les instances industrielles

Cl	NB	LB	BinLoading				DFS			
			#C	W	#M	CPU	#C	W	#M	CPU
Son	1166	4269	4592	332	54514	2046	**4412**	**171**	**52343**	2003

TABLE 4.8 – Comparaison des méthodes sur l'ensemble d'instances Son

algorithme avec la valeur de $\boldsymbol{\theta}$ apprise dans la section 4.4.3.2, avec un budget de 150000 conflits par camion. Les indicateurs utilisés sont les mêmes que pour la table 4.10. On peut voir que notre approche surpasse la méthode employée par Renault. On y voit une amélioration de 4% sur le nombre de camions, et le taux de remplissage moyen passe de 93% à 96% tout en restant proche en terme de temps de calcul. On rappelle que la borne inférieure sur cet ensemble d'instances est de 4269 camions, laissant alors une marge de progression.

4.5 Recherche arborescente de Monte Carlo

Dans cette section, nous intégrons l'heuristique issue de l'apprentissage par renforcement dans une recherche arborescente de Monte Carlo (MCTS). On décrit l'algorithme utilisé dans une première partie, avant de présenter des résultats expérimentaux dans une seconde partie.

4.5.1 Algorithme MCTS pour le problème de chargement de camions

Quelques adaptations et précisions par rapport au cadre présenté dans le chapitre 2 sont à apporter pour utiliser la méthode MCTS pour ce problème de chargement de camions. On se place dans le même cadre que pour l'apprentissage 4.4, où une solution est une séquence de rectangles σ. Cependant, les mouvements de *pont* ne sont pas ignorés. On rappelle que l'objectif considéré dans l'arbre $\mathcal{T}$ est l'espace inutile d'une solution W (eq. 4.13) normalisé par la surface du camion, et qu'on veut alors minimiser la somme des pénalités obtenues.

Utilisation du solveur DFS. On souhaite utiliser l'algorithme de recherche en profondeur développé dans la section 4.3.2 pour la phase de simulation. Cette algorithme minimise l'objectif de surface des objets exclus (eq. 4.8), puis l'objectif de métrage linéaire (eq. 4.8) lorsque le premier devient nul. Cet algorithme utilise

alors les bornes 4.10 et 4.11 pour couper des branches de l'arbre. Cependant, comme précisé l'arbre $\mathcal{T}$ se base lui sur l'espace inutile W (eq. 4.13). Il utilise alors la borne inférieure $R(\sigma)$ pour le calcul des récompenses, qui est la surface des rectangles vides de σ, lorsque σ n'est pas une solution complète et $W(\sigma)$ lorsque l'épisode est fini. Cette valeur est normalisée par la surface du camion. On a donc la phase de sélection du MCTS qui ne se base que sur cette valeur pour évaluer les noeuds, mais dans les DFS de la phase de simulation, c'est bien les premiers objectifs qui sont utilisés, ainsi que les bornes associées. De plus, d'un point de vue de l'architecture du programme utilisé, c'est le code du DFS qui fournit les actions disponibles à chaque état. Il utilise alors ses propres bornes pour décider des actions qui ne provoquent pas de conflit, et sont à ajouter à l'arbre. Enfin, avant toute expansion, la partie MCTS fait appel au DFS pour savoir si le noeud courant est cohérent, et peut donc élaguer l'arbre en tenant compte des bornes du DFS.

Sélection. Pour la sélection, on force la sélection de chacune des actions une fois, avant d'utiliser l'équation 1.48. Contrairement au problème de tournées de chariots, les épisodes sont courts, et on veut forcer l'exploration et ne pas exploiter trop vite. Pour cette raison, nous n'utiliserons pas le compromis dynamique introduit dans le chapitre 2. Nous proposons de centrer les valeurs du terme d'exploitation autour de 1 pour chaque arc A de s : $2 - \frac{Q(s,A)}{V(s)}$, le dénominateur étant la valeur de l'état s (dans notre cas, elle est égale à la valeur de l'action Q qui mène à s). Cette valeur est alors supérieure à 1, si l'évaluation de l'action est meilleure que la moyenne de toutes les actions prises dans cet état, et inférieure à 1 dans le cas contraire.

Expansion. Soit v le noeud sélectionné dans la première phase. On passe à la phase d'expansion seulement si $N(v) > 0$. Si c'est le cas, alors le prochain noeud courant est sélectionné aléatoirement parmi les enfants de v. Lors de l'initialisation de la probabilité à priori, nous ne disposons pas de probabilité pour les actions de *pont* lorsque nous les utilisons. On doit alors se passer de cette connaissance à priori et utiliser une distribution uniforme.

Simulation. La phase de simulation est similaire à celle décrite dans le chapitre 2. La simulation commence par une procédure gloutonne en suivant la politique apprise dans la section 4.4. On évalue ensuite le *gain* (eq. 1.18) de l'épisode correspondant à cette simulation pour la racine, c'est à dire à $t = 0$, que l'on note $G(0)$. Le budget de conflit pour le DFS est accordé en comparant le paramètre de seuil Γ au ratio de $G(0)$ et de $V(0)$, qui est la moyenne des *gains* de tout les épisodes évalué à la racine :

$$\beta = \begin{cases} \beta^+ & \text{si solution améliorante} \\ \beta^- + (\beta^+ - \beta^-) * (\Gamma - G(0)/V(0)) & \text{si } G(0)/V(0) < \Gamma \\ 0 & \text{sinon} \end{cases}$$

avec β^- et β^+ respectivement le budget minimum et le budget maximum accordé. Enfin, lorsque le budget de conflits est consommé, la meilleure solution est retournée par la phase de simulation, c'est à dire la solution de la procédure gloutonne ou une solution améliorante. En effet, la procédure n'utilise aucune borne pour éviter des branches non améliorantes, contrairement au DFS.

Rétropropagation. La phase de rétropropagation se déroule comme celle décrite dans la section 1.3. C'est à dire que le *gain* $G(t)$ est calculé pour chacun des arc e traversé à partir du noeud v à l'étape t de la sélection, puis les valeurs de $Q(v,e)$ et $N(v,e)$ sont mises à jour comme suit :

$$Q(v,e) \leftarrow Q(v,e) + \frac{G(t) - Q(v,e)}{N(v,e)+1}$$
$$N(v,e) \leftarrow N(v,e) + 1$$

4.5.2 Résultats expérimentaux

Nous présentons dans cette section, le résultat des expérimentations numériques faites avec la recherche arborescente de Monte Carlo pour le problème de chargement de camions.

Instances industrielles. On présente deux versions du MCTS : une version sans DFS (que l'on note `MCTS`) et une version avec DFS (que l'on note `MCTS+DFS`). Nous avons vu précédemment que l'utilisation d'une valeur de $\boldsymbol{\theta}$ dédiée n'apportait aucun avantage pour une recherche en profondeur et décidons alors d'utiliser une unique valeur de $\boldsymbol{\theta}$ pour toutes les expérimentations. On utilise celle apprise sur les instances de `Son` comme décrit dans la section 4.4.3.2.

On donne pour chacune des versions un budget de conflits par camion. On compte une itération de la méthode comme un conflit, auquel s'ajoute les conflits de la phase de DFS. Le raisonnement est qu'à chaque itération soit une simulation est exécutée (et donc une feuille de l'arbre de recherche est atteinte), soit une branche est supprimée (si aucun fils n'est ajouté ou lors de la phase de sélection à cause des bornes). Lors du calcul du budget du DFS, on tronque le budget accordé s'il fait dépasser le nombre maximum de conflits de la résolution. Pour `MCTS`, ce budget est de 2000 conflits (donc 2000 itérations), et il est de 5000 pour `MCTS+DFS`. Ces valeurs ont été réglées pour s'approcher du temps de calcul de la méthode utilisée dans l'entreprise, et les conflits du DFS se font beaucoup plus rapidement qu'une itération du MCTS.

La table 4.9 donne les paramètres restants : c est le compromis d'exploitation/exploration, β est le paramètre de température, Γ est le paramètre de seuil pour l'activation du DFS, β^- et β^+ sont respectivement le nombre minimal et maximal de conflits pour le DFS. Enfin γ est le coefficient de décroissance dans le calcul du *gain*. Ce dernier est réglé a 1 ce qui veut dire que l'algorithme accordera

autant d'importance aux récompenses futures qu'à la récompense directe. Pour le problème de tournées de chariots, la profondeur des solutions était très importante (environ 1800 opérations pour l'horizon `Jour` donc les résultats étaient impactés par ce paramètre). Pour le problème de chargement de camions, les solutions ne sont pas très profondes, et les résultats intermédiaires ont montré de meilleurs résultats avec $\gamma = 1$.

c	2
β	0.1
Γ	0.5
β^+	100
β^-	500
γ	1

TABLE 4.9 – Valeurs des paramètres des méthodes MCTS

Les résultats de nos variantes MCTS sont fournis dans la table 4.10. Dans cette table, sont également ajoutés les résultats obtenus par l'algorithme utilisé dans l'entreprise (`BinLoading`), ainsi que les résultats obtenus par la recherche arborescente simple présentée à la section précédente (`DFS`), avec la même valeur de $\boldsymbol{\theta}$ que pour les variantes de MCTS. On observe une amélioration globale des résultats par l'utilisation de la méthode `MCTS+DFS` proposée. Plus précisément, on voit une nette amélioration apportée par les méthodes de Monte Carlo sur les instances de `M`, améliorant au mieux de 16% le nombre de camions par rapport à la recherche en profondeur classique (31% par rapport à l'algorithme de l'entreprise). Cela s'explique par le fait que l'heuristique que nous utilisons classe les actions de *pont* en dernière position, alors que la sélection du MCTS traite ce type d'actions de la même manière que les autres. Étant donné que les instances de `M` sont plus difficiles du point de vue de la contrainte de chargement aux essieux(eq. 4.6), il est alors normal d'obtenir une amélioration nette. Pour cet ensemble d'instances, on passe alors d'un taux de remplissage $(1 - W/\#C)$ de 82% pour la méthode `DFS` à 94% pour la méthode `MCTS+DFS`, ce qui est alors proche des taux de remplissage obtenus pour les ensembles `S` et `D` de 96% pour la méthode `MCTS+DFS`. Pour les instances `S` et `D`, on constate une légère diminution du nombre de camions rapporté au nombre total, par rapport à la recherche arborescente avec redémarrages, peu importante au vu du nombre de camions considéré.

Si l'on compare maintenant les deux variantes MCTS proposées, on constate que la version avec DFS domine légèrement l'autre, aussi bien du point des indicateurs de la qualité des solutions, que du temps total de résolution. La méthode permet de trouver des meilleures solutions pour un temps plus proche de celui de l'algorithme utilisé dans l'entreprise.

Instance alternative. On compare ensuite les deux variantes de la recherche arborescente de Monte Carlo avec les méthodes précédentes sur l'instance `Son` sur

Cl	NB	LB	BinLoading				DFS				MCTS				MCTS+DFS			
			#C	W	#M	CPU	#C	W	#M	CPU	#C	W	#M	CPU	#C	W	#M	CPU
M	1171	1244	1865	493	15127	38	1521	268	12089	47	1314	97	9797	73	**1280**	**82**	**9584**	70
S	7130	9556	10237	762	94629	136	9736	387	89563	189	9670	**344**	89027	240	**9663**	345	**88994**	142
D	1190	1573	1643	98	13893	34	1604	60	13374	47	1597	**56**	13332	89	**1596**	**56**	**13325**	40

TABLE 4.10 – Comparaison des méthodes sur les instances industrielles

Cl	NB	LB	BinLoading				DFS				MCTS				MCTS+DFS			
			#C	W	#M	CPU	#C	W	#M	CPU	#C	W	#M	CPU	#C	W	#M	CPU
Son	1166	4269	4592	332	54514	2046	4412	171	52343	2003	4360	126	51741	3561	**4356**	**125**	**51724**	1905

TABLE 4.11 – Comparaison des méthodes sur l'ensemble d'instances Son

un temps de calcul court, avec le même paramétrage excepté la limite de conflits qui est fixée ici à 4000 et 10000 pour MCTS et MCTS+DFS respectivement. Les résultats sont donnés dans la table 4.11. Le constat est similaire à celui effectué sur les instances industrielles peu contraintes sur la charge aux essieux. On observe une amélioration légère de 5% sur le nombre de camions entre MCTS+DFS et la méthode BinLoading pour un temps de calcul similaire, et une amélioration de seulement 1% par rapport à la méthode DFS, passant d'un taux de remplissage de 96% à 97%. On voit également qu'il n'y a que très peu de différences entre les deux variantes de la recherche arborescente de Monte Carlo, excepté le temps de calcul. On obtient alors des solutions similaires en presque deux fois moins de temps en ajoutant des DFS au sein même du MCTS tout en obtenant de meilleurs résultats qu'un DFS classique avec redémarrages.

On donne également dans la table 4.12 l'espace inutile normalisé total pour tous les premiers camions des instances (W) pour nos trois méthodes sur cette même expérimentation, ainsi que le taux de remplissage (Tx). En effet, ces algorithmes ont été conçus pour résoudre le problème de *rectangle packing* dans lequel on minimise le nombre d'objets exclus du camion. Toutes les sous-instances suivant le premier camion sont ainsi dépendantes du résultat sur ce premier camion. On souhaite alors comparer ces méthodes sur le premier camion uniquement. Dans cette table, on voit qu'une fois encore les deux algorithmes de recherche arborescente de Monte Carlo proposent des solutions similaires, et une amélioration légère par rapport à la méthode DFS d'environ 2% en terme de taux de remplissage.

Cl	NB	DFS		MCTS		MCTS+DFS	
		W	Tx	W	Tx	W	Tx
Son	1166	65.71	94.36%	43.85	96.34%	**42.19**	**96.38%**

TABLE 4.12 – Comparaison de l'espace inutile pour le premier camion

Impact des paramètres du DFS. Pour un temps court, nous avons trouvé via des expériences préliminaires qu'il vallait mieux faire peu de DFS avec un budget faible. En effet, il ne semble pas surprenant que dans un court laps de temps, on ne peut pas se permettre de passer trop de temps à faire des DFS, et le temps accordé à ce mécanisme peut rapidement prendre le dessus sur la partie MCTS. On se pose alors la question de l'impact des paramètres que sont Γ, β^- et β^+, le paramètre de seuil, et les paramètres de budget (minimal et maximal) respectivement sur un temps de résolution plus important.

Pour cette expérimentation, on cherche à expliciter le comportement de l'algorithme pour un temps fixe, en faisant varier ces paramètres sur les instances de `Son`. Pour pouvoir fixer un temps de résolution par instance, on se limite alors au premier camion, que l'on essaye de remplir au mieux pendant 15 minutes. On regarde trois valeurs de Γ : 0.5, 0.8 et 1, et, pour ces trois valeurs, on se donne trois intervalles de budget de DFS $[\beta^-, \beta^+]$: $[100 - 500]$, $[1000 - 5000]$, $[2500 - 1000]$.

La table 4.13 reporte les résultats de cette expérimentation. Chaque colonne correspond à un intervalle de budget de DFS, et chaque ligne à une valeur de Γ. Pour chaque paramétrage, on donne le total de l'espace vide normalisé par la surface des camions (W), le nombre total d'itérations en millions (iter) et le pourcentage d'itérations pour lesquelles un DFS est lancé. Pour chacune des lignes est indiqué en gras le meilleur résultat obtenu.

On observe globalement qu'aucun de ces paramétrages ne se démarque réellement. La différence d'espace inutile entre le pire paramétrage et le meilleur est de 0.435, ce qui rapporté au nombre de camions considérés est négligeable. Cependant, on peut observer une légère tendance, si l'on fait très peu de DFS ($\Gamma = 0.5$), alors augmenter le budget semble être bénéfique. A l'inverse, si l'on fait un nombre important de DFS ($\Gamma = 1$), alors il semblerait que diminuer le budget par DFS améliore le résultat final.

Plus précisément, cette table nous permet également de quantifier le nombre de DFS en fonction de la valeur de Γ, allant de 1 à 2% pour $\Gamma = 0.5$ jusqu'à 57% pour $\Gamma = 1$. Pour un temps fixé, le nombre d'itérations diminue logiquement lorsque le budget accordé aux DFS augmente. Plus curieux, il semble y avoir une différence dans le taux de DFS lorsque le budget augmente. En effet, plus les DFS sont longs, et plus le ratio de DFS sur le nombre d'itérations diminue. Une explication serait liée au fait qu'un budget plus important permet alors faire baisser la borne supérieure plus vite, orientant l'arbre plus fortement vers une zone où toutes les simulations retournent de bonnes solutions. La moyenne des valeurs du *gain* des épisodes retournés à la racine de l'arbre est alors biaisée par cette zone, faisant baisser la valeur du *gain* maximale nécessaire d'un épisode pour pouvoir lancer un DFS. De plus amples expérimentations sont néanmoins nécessaires pour confirmer cette hypothèse, ainsi que les liens et impacts de ces deux paramètres sur la qualité de la solution.

	$\beta = [100-500]$			$\beta = [1000-5000]$			$\beta = [2500-10000]$		
	W	Iter	DFS	W	Iter	DFS	W	Iter	DFS
$\Gamma = 0.5$	24.499	15008M	2.0%	24.356	12119M	1.3%	**24.328**	10939M	1.2%
$\Gamma = 0.8$	24.241	6231M	19.7%	**24.064**	1648M	13.3%	24.205	904M	11.8%
$\Gamma = 1$	**24.190**	2421M	57.2%	24.239	445M	40.3%	24.361	216M	39.2%

TABLE 4.13 – Étude des paramètres de la phase de DFS

4.6 Synthèse

Dans ce chapitre, nous avons proposé une approche pour résoudre le problème de chargement de camions dans laquelle nous résolvons successivement un problème à un seul camion. A partir de la procédure gloutonne *best-fit* utilisée dans l'entreprise, nous avons proposé une exploration de l'espace des solutions accessibles par cette méthode par une recherche arborescente. Nous y avons ajouté un mouvement de *pont* permettant d'accéder à un plus grand nombre de solutions du problème. Nous avons ensuite caractérisé quelles sont les solutions que nous pouvons obtenir avec une telle méthode. Nous avons développé une heuristique basée sur la combinaison linéaire de quatre critères pour sélectionner un rectangle à chaque étape, les pondérations de ces critères ont été obtenues avec un algorithme d'apprentissage par renforcement. Cette heuristique a alors été intégrée dans la méthode arborescente. Ceci permet, dans un budget temporel très court, de surpasser les résultats antérieurs sur trois jeux d'instances industrielles ainsi que sur un nouveau jeu d'instances générées. Nous avons ensuite appliqué la recherche arborescente de Monte Carlo à ce problème avec succès, montrant alors un intérêt à l'utilisation d'une recherche en profondeur d'abord dans les phases de simulation, permettant de retourner de meilleures solutions plus rapidement qu'avec la phase de simulation classique. Néanmoins, l'impact des paramètres permettant de définir le temps passé dans un tel mécanisme d'intensification nécessite de plus amples investigations.

Conclusion et perspectives

Durant cette thèse, nous avons proposé et appliqué une méthodologie basée sur l'apprentissage par renforcement pour résoudre deux problèmes d'optimisation combinatoire issus de besoins industriels. Cette méthodologie repose sur deux axes. Le premier axe consiste à assembler plusieurs critères relatifs au problème étudié dans une combinaison linéaire pour former une évaluation de choix possibles à chacun des points de décision d'un algorithme de résolution du problème. Les poids de cette combinaison linéaire sont appris sur un ensemble d'instances par un algorithme d'apprentissage par renforcement. L'heuristique de choix de valeurs peut alors être intégrée dans une méthode de recherche arborescente. Le second axe consiste à utiliser la recherche arborescente de Monte Carlo en intégrant cette heuristique de choix de valeurs. Nous avons alors proposé d'hybrider les phases de simulation en y intégrant de la recherche en profondeur comme un mécanisme d'intensification. Nous avons également proposé d'utiliser l'évolution des bornes comme récompenses, et d'appliquer un coefficient dynamique pour régler le compromis entre exploitation et exploration durant la phase de sélection. Au travers l'étude de problèmes industriels de tournées de chariots et de chargement de camions, nous avons vu l'applicabilité de la méthodologie proposée et sa relative généricité, ainsi que ses performances par rapport aux méthodes existantes dans l'entreprise.

Pour le **problème de tournées de chariots**, nous avons proposé une approche de résolution basée sur la programmation par contraintes. Un des modèles proposés présente de bons résultats sur les instances industrielles mais pas sur des instances plus contraintes. L'approche a été modifiée pour pouvoir intégrer une heuristique de choix de valeurs issue d'un apprentissage par renforcement. Cette heuristique basée sur quatre critères pour chacune des décisions possibles a permis de résoudre un plus grand nombre d'instances difficiles, en plus des instances industrielles. Nous avons montré qu'en intégrant ensuite l'heuristique apprise dans une recherche arborescente de Monte Carlo nous pouvons améliorer ces résultats.

Pour le problème de **chargement de camions**, nous avons proposé une approche itérative de prise en compte des camions. Nous avons proposé une recherche arborescente s'appuyant sur une partie de l'algorithme utilisé dans l'entreprise pour remplir un camion. Nous avons montré que notre approche était compétitive avec l'existant même sans heuristique de choix de valeurs. En intégrant une heuristique de choix de valeurs s'appuyant sur quatre critères, notre méthodes surpasse l'algorithme existant dans l'entreprise. La recherche arborescente de Monte Carlo a ensuite été proposée pour ce problème également réduisant encore le nombre de camion nécessaires.

Nous avons donc montré par ces deux études de cas sur des problèmes très différents, que notre méthodologie était compétitive par rapport aux méthodes utilisées dans l'entreprise. Nous avons montré que la recherche arborescente de Monte Carlo

peut être plus performante qu'une recherche en profondeur d'abord avec retours-arrière utilisant la même heuristique de choix de valeurs.

Il y a deux niveaux de perspective sur les travaux présentés, les perspectives que nous appelons métiers qui concerne directement chacun des problèmes étudiés, et les perspectives méthodologiques applicables aux approches de résolution proposées.

Perspectives métiers

Problèmes intégrés. Les deux problèmes présentés sont en réalité des composants d'un problème plus vaste dont la résolution a été segmentée dans l'entreprise. Pour le problème de tournées de chariots, le problème général consiste à dimensionner l'équipe d'opérateurs d'un atelier donné. L'affectation au préalable des opérateurs aux machines est effectuée avant la réalisation des tournées. Pour le problème de chargement de camions, le problème général comporte la création des piles d'objets. Ces piles sont contraintes par le type des objets qui les composent, leurs poids et les fournisseurs de ces objets. Le problème comporte également la résolution de l'affectation des piles aux camions qui est effectuée par le calcul d'un plan logistique prenant en considération les dates de disponibilités et de demandes des objets. Une des perspectives de la thèse est donc d'élargir le champ de décision pour traiter les problèmes de façon intégrée.

Tournées de chariots. Concernant le problème de tournées de chariots, deux points pourront être intégrés dans le futur. Le premier point concerne certains postes de production qui doivent livrer à plusieurs postes de consommation différents, chacun ayant des cadences de consommation différentes. Cela ajoute alors un point de décision supplémentaire, lorsqu'un chariot est collecté sur un tel poste, il faut alors décider quelle destination doit être livrée. Le second point consiste à intégrer un objectif de schémas récurrents dans les tournées. En effet, les routes sont calculées dans une but de validation, les opérateurs en pratique ne suivront pas à la lettre le plan proposé. Cet objectif, qui a aujourd'hui un contour flou et qui nécessite d'être défini formellement, a pour but de proposer des solutions qui contiendraient le plus possibles des schémas récurrents dans la route, des cycles de trajets dont on dévierait le moins possible. De telles solutions pourraient alors être plus facilement suivies en pratique, et pourraient alors servir réellement de guide plus que de validation.

Chargement de camions. Concernant le second problème, plusieurs aspects métiers qui n'étaient pas dans le cahier des charges au moment de la thèse sont à l'étude :

- La contrainte de charges aux essieux que nous respectons en fin de chargement de camion devrait être respectée à chaque départ de fournisseur, c'est à dire pour chaque rang de piles d'objets ;

— Les chargements proposés par nos algorithmes peuvent parfois ne pas convenir aux opérateurs logistiques, car ils laissent des espaces vides dans le camion dans le but de satisfaire l'équilibre de la charge aux essieux ;
— Les contraintes sur la charge aux essieux avant et arrière peuvent s'appliquer également sur l'équilibre de charge latérale ;

Le premier point peut facilement être pris en compte par notre algorithme, car il suffit de vérifier la contrainte à chaque changement de rang des objets. Concernant le second point la contrainte exact sur les règles de chargement doit être définie clairement, il semblerait qu'ajuster les solutions soit suffisant la plupart du temps. Cependant nos solutions mettent en lumière un principe non utilisé : mettre du vide entre les piles dans le camion permet de plus le remplir. Enfin pour le dernier point, la contrainte n'est pas encore formellement définie et il est alors trop tôt pour se prononcer sur son intégration.

Perspectives méthodologiques

Concernant la méthodologie proposée le long de ce manuscrit plusieurs pistes sont à l'étude.

Complexifier le modèle d'apprentissage Par rapport au modèle d'apprentissage proposé, nous pensons qu'un modèle simple tel que la combinaison linéaire de quelques critères présente l'avantage de ne représenter qu'un coût de calcul négligeable, surtout par rapport aux modèles souvent employés dans la littérature, basés sur des réseaux de neurones. Cependant, nous sommes conscient que notre approche n'encode pas vraiment l'état du problème au moment de la prise de décision. De plus, cette approche n'est pas générique et nécessite un travail non négligeable pour déterminer des critères avant même de pouvoir valider leur pertinence. Si nous pensons qu'il est important de pouvoir intégrer des connaissances métiers dans des heuristiques de branchement, une perspective de ces travaux est d'élargir notre modèle d'apprentissage, notamment en prenant en compte les interactions entre les différentes décisions. Une des directions probables dans cette voie converge alors avec les travaux déjà réalisés sur ce point notamment par Cappart et al. [30] et Chalumeau et al. [35] avec leur intégration dans un solveur de programmation par contraintes, d'heuristiques de choix de valeurs basées sur des réseaux de neurones en graphe dont les poids sont réglés avec l'apprentissage par renforcement. En effet, les interactions entre les différentes décisions peuvent être prises en compte dans ce type de modèle d'apprentissage, et l'utilisation d'un solveur générique permet d'y intégrer des données d'entrées issues du solveur en plus de critères métiers. Notre inquiétude concerne cependant le passage à l'échelle d'une telle méthode, et le temps de calcul nécessaire à chaque ouverture de noeud dans la recherche arborescente, ou à chaque décision dans les simulations d'une recherche arborescente de Monte Carlo. Est-il alors possible d'utiliser une telle approche dans un contexte industriel ?

MCTS : Nouveaux mécanismes Concernant la recherche arborescente de Monte Carlo, si nous avons montré que l'utilisation de recherche en profondeur d'abord au sein des phases de simulation pouvait être bénéfique, l'impact des paramètres utilisés reste à clarifier. Le moyen même d'évaluation de la pertinence de lancer ou non un DFS mérite également de plus amples investigations. Nous aimerions de plus étudier d'autres mécanismes à intégrer dans cet algorithme :

- Remplacer la phase de DFS par une recherche à divergences limitées ;
- Ouvrir les noeuds lors des simulations ;
- Utiliser les simulations pour mettre à jour la politique ;

La recherche à divergences limitées permettrait une exploration moins locale qu'un DFS tout en s'appuyant fortement sur l'heuristique. Cependant, l'utilisation d'une telle méthode pose question par son aspect déterministe. En effet, la phase de simulation ne peut pas être réalisée de manière déterministe car de nombreuses solutions identiques seraient visitées un grand nombre de fois. Une façon d'empêcher de visiter les mêmes solutions plusieurs fois est de laisser les noeuds ouverts lors des simulations. Loth et al. propose un mécanisme similaire dans leur intégration du solveur Gecode [61] avec la recherche arborescente de Monte Carlo [114]. Les noeuds ouverts ne sont pas intégrés dans l'arbre du MCTS, mais l'état du solveur à la fin de la simulation est gardé en mémoire pour pouvoir reprendre la recherche là ou elle s'était arrêtée. Des expérimentations préliminaires d'un tel mécanisme ont déjà été menées sans grand succès, mais mériterait d'être étudiées à nouveau. Enfin, la recherche arborescente de Monte Carlo exécute un grand nombre de simulations, et donc génère un grand nombre de données utilisables comme dans le cadre de l'algorithme REINFORCE. L'intégration de la mise à jour des poids après un certain nombre de simulations permettrait soit de réaliser l'apprentissage directement en ligne, soit de spécialiser l'heuristique pour l'instance considérée.

MCTS : Hybridation avec un solveur de PPC Enfin, nous souhaitons, à l'image des travaux de Loth et al. [114], pouvoir hybrider notre recherche arborescente de Monte Carlo avec un solveur de programmation par contraintes existant. Lors de l'étude du problème de chargement de camions, une bibliothèque à visé générique a été développée. Elle permet d'interfacer une recherche arborescente de Monte Carlo avec un autre algorithme exécutant les simulations. Cette bibliothèque permet ainsi d'utiliser un solveur de PPC pour les phases de simulation. Cette intégration pose cependant la question des récompenses à accorder aux résultats des simulations. En effet, dans l'hybridation envisagée, la procédure gloutonne de la phase de simulation s'arrêtera au premier conflit rencontré, sans retourner de solution complète. Des expériences complémentaires ont été menées sur le problème d'ordonnancement d'atelier à cheminement multiple (*Job-shop Scheduling Problem*) pour prendre en compte la profondeur du conflit en plus de l'accroissement marginal de la borne inférieure. Cependant, cette approche n'a pas donné de résultats satisfaisants pour l'instant et des travaux supplémentaires sont envisagés.

Bibliographie

[1] Kenshin Abe, Zijian Xu, Issei Sato, and Masashi Sugiyama. Solving np-hard problems on graphs by reinforcement learning without domain knowledge. *arXiv*, 1905.11623, 2019. (Cité en pages 32 et 33.)

[2] Tobias Achterberg, Thorsten Koch, and Alexander Martin. Branching rules revisited. *Operations Research Letters*, 33(1) :42–54, 2005. (Cité en page 36.)

[3] Alejandro Marcos Alvarez, Louis Wehenkel, and Quentin Louveaux. Online learning for strong branching approximation in branch-and-bound. Technical report, Department of Electrical Engineering and Computer Science, Universite de Liege, 2016. (Cité en page 36.)

[4] Alejandro Marcos Alvarez, Quentin Louveaux, and Louis Wehenkel. A machine learning-based approximation of strong branching. *INFORMS Journal on Computing*, 29(1) :185–195, 2017. (Cité en page 36.)

[5] Ramón Alvarez-Valdés, Francisco Parreño, and José Manuel Tamarit. Reactive GRASP for the strip-packing problem. *Computers & Operations Research*, 35(4) :1065–1083, 2008. (Cité en page 110.)

[6] Ramón Alvarez-Valdés, Francisco Parreño, and José Manuel Tamarit. A branch and bound algorithm for the strip packing problem. *OR Spectrum*, 31 (2) :431–459, 2009. (Cité en page 99.)

[7] David L. Applegate, Robert Bixby, Václav Chvatal, and William Cook. Finding cuts in the tsp (a preliminary report). Technical report, Center for Discrete Mathematics & Theoretical Computer Science, 1995. (Cité en page 36.)

[8] David L. Applegate, Robert Bixby, Václav Chvatal, and William Cook. *The traveling salesman problem : a computational study*. Princeton university press, 2006. (Cité en page 36.)

[9] Florian Arnold, Ítalo Santana, Kenneth Sörensen, and Thibaut Vidal. PILS : exploring high-order neighborhoods by pattern mining and injection. *Pattern Recognition*, 116 :107957, 2021. (Cité en page 38.)

[10] Peter Auer, Nicolò Cesa-Bianchi, and Paul Fischer. Finite-time analysis of the multiarmed bandit problem. *Machine Learning*, 47(2-3) :235–256, 2002. (Cité en page 29.)

[11] Brenda S. Baker, E. G. Coffman, Jr., and Ronald L. Rivest. Orthogonal packings in two dimensions. *SIAM Journal on Computing*, 9(4) :846–855, 1980. (Cité en pages 100, 111 et 112.)

[12] J. Christopher Beck. Solution-guided multi-point constructive search for job shop scheduling. *Journal of Artificial Intelligence Research*, 29 :49–77, 2007. (Cité en page 13.)

[13] J. Christopher Beck, Patrick Prosser, and Evgeny Selensky. Vehicle routing and job shop scheduling : What's the difference ? In *Proceedings of the 13th International Conference on Automated Planning and Scheduling*, pages 267–276, 2003. (Cité en page 66.)

[14] Richard Bellman. A markovian decision process. *Journal of Mathematics and Mechanics*, 6(5) :679–684, 1957. (Cité en page 17.)

[15] Irwan Bello, Hieu Pham, Quoc V. Le, Mohammad Norouzi, and Samy Bengio. Neural combinatorial optimization with reinforcement learning. In *Proceedings of the 5th International Conference on Learning Representations*, 2017. (Cité en pages 27 et 35.)

[16] Gleb Belov, Guntram Scheithauer, and E. A. Mukhacheva. One-dimensional heuristics adapted for two-dimensional rectangular strip packing. *Journal of the Operational Research Society*, 59(6) :823–832, 2008. (Cité en page 100.)

[17] Yoshua Bengio, Andrea Lodi, and Antoine Prouvost. Machine learning for combinatorial optimization : A methodological tour d'horizon. *European Journal of Operational Research*, 290(2) :405–421, 2021. (Cité en pages 34 et 35.)

[18] Gerardo Berbeglia, Jean-François Cordeau, Irina Gribkovskaia, and Gilbert Laporte. Static pickup and delivery problems : a classification scheme and survey. *TOP*, 15(1) :1–31, 2007. (Cité en page 62.)

[19] David Bergman, André A. Ciré, Willem-Jan van Hoeve, and John N. Hooker. *Decision Diagrams for Optimization.* Artificial Intelligence : Foundations, Theory, and Algorithms. Springer, 2016. (Cité en page 37.)

[20] Judith O Berkey and Pearl Y Wang. Two-dimensional finite bin-packing algorithms. *Journal of the Operational Research Society*, 38(5) :423–429, 1987. (Cité en page 100.)

[21] Dimitri P. Bertsekas. *Dynamic Programming : Deterministic and Stochastic Models.* Prentice-Hall, Inc., USA, 1987. ISBN 0132215810. (Cité en page 21.)

[22] Dimitri P. Bertsekas and John N. Tsitsiklis. *Parallel and Distributed Computation : Numerical Methods.* Prentice-Hall, Inc., USA, 1989. ISBN 0136487009. (Cité en page 21.)

[23] Dimitris Bertsimas, J. Daniel Griffith, Vishal Gupta, Mykel J. Kochenderfer, and Velibor V. Misic. A comparison of monte carlo tree search and rolling horizon optimization for large-scale dynamic resource allocation problems. *European Journal of Operational Research*, 263(2) :664–678, 2017. (Cité en page 32.)

[24] Christian Bessiere. Constraint propagation. In *Handbook of Constraint Programming*, pages 29–83. 2006. (Cité en page 11.)

[25] Marco A. Boschetti and Aristide Mingozzi. The two-dimensional finite bin packing problem. part II : new lower and upper bounds. *4OR*, 1(2) :135–147, 2003. (Cité en pages 99 et 120.)

[26] Frédéric Boussemart, Fred Hemery, Christophe Lecoutre, and Lakhdar Sais. Boosting systematic search by weighting constraints. In *Proceedings of the 16th Eureopean Conference on Artificial Intelligence*, pages 146–150, 2004. (Cité en pages 12 et 72.)

[27] Edmund K. Burke, Graham Kendall, and Glenn Whitwell. A new placement heuristic for the orthogonal stock-cutting problem. *Operational Research*, 52 (4) :655–671, 2004. (Cité en pages 100, 101, 104 et 105.)

[28] Quentin Cappart, Emmanuel Goutierre, David Bergman, and Louis-Martin Rousseau. Improving optimization bounds using machine learning : Decision diagrams meet deep reinforcement learning. In *Proceedings of the 33rd AAAI Conference on Artificial Intelligence*, pages 1443–1451, 2019. (Cité en pages 37 et 51.)

[29] Quentin Cappart, Didier Chételat, Elias B. Khalil, Andrea Lodi, Christopher Morris, and Petar Velickovic. Combinatorial optimization and reasoning with graph neural networks. In *Proceedings of the 30th International Joint Conference on Artificial Intelligence*, pages 4348–4355, 2021. (Cité en page 34.)

[30] Quentin Cappart, Thierry Moisan, Louis-Martin Rousseau, Isabeau Prémont-Schwarz, and André A. Ciré. Combining reinforcement learning and constraint programming for combinatorial optimization. In *Proceedings of the 35th AAAI Conference on Artificial Intelligence*, pages 3677–3687, 2021. (Cité en pages 27, 37, 51 et 139.)

[31] Quentin Cappart, David Bergman, Louis-Martin Rousseau, Isabeau Prémont-Schwarz, and Augustin Parjadis. Improving variable orderings of approximate decision diagrams using reinforcement learning. *INFORMS Journal of Computing*, 0(0), 2022. (Cité en page 37.)

[32] Jacques Carlier, François Clautiaux, and Aziz Moukrim. New reduction procedures and lower bounds for the two-dimensional bin packing problem with fixed orientation. *Computers & Operations Research*, 34(8) :2223–2250, 2007. (Cité en page 99.)

[33] Francesco Carrabs, Jean-François Cordeau, and Gilbert Laporte. Variable neighborhood search for the pickup and delivery traveling salesman problem with LIFO loading. *INFORMS Journal on Computing*, 19(4) :618–632, 2007. (Cité en page 63.)

[34] Tristan Cazenave. Nested monte-carlo search. In Craig Boutilier, editor, *Proceedings of the 21st International Joint Conference on Artificial Intelligence*, pages 456–461, 2009. (Cité en page 63.)

[35] Félix Chalumeau, Ilan Coulon, Quentin Cappart, and Louis-Martin Rousseau. Seapearl : A constraint programming solver guided by reinforcement learning. In *Proceedings of the 18th International Conference on Integration of Constraint Programming, Artificial Intelligence, and Operations Research*, pages 392–409, 2021. (Cité en pages 27, 37, 42 et 139.)

[36] Guillaume Chaslot, Jos Uiterwijk, Bruno Bouzy, and H. Herik. Monte-carlo strategies for computer go. In *Proceedings of the 18th BeNeLux Conference on Artificial Intelligence*, pages 110–117, 01 2006. (Cité en page 27.)

[37] Ronghua Chen, Bo Yang, Shi Li, and Shilong Wang. A self-learning genetic algorithm based on reinforcement learning for flexible job-shop scheduling problem. *Computers & Industrial Engineering*, 149 :106778, 2020. (Cité en page 38.)

[38] Xinyun Chen and Yuandong Tian. Learning to perform local rewriting for combinatorial optimization. In *Proceedings of the 32nd International Conference on Advances in Neural Information Processing Systems*, pages 6278–6289, 2019. (Cité en page 38.)

[39] F. R. K. Chung, M. R. Garey, and D. S. Johnson. On packing two-dimensional bins. *SIAM Journal on Algebraic Discrete Methods*, 3(1) :66–76, 1982. (Cité en page 100.)

[40] François Clautiaux, Jacques Carlier, and Aziz Moukrim. A new exact method for the two-dimensional orthogonal packing problem. *European Journal of Operational Research*, 183(3) :1196–1211, 2007. (Cité en pages 99, 100, 105 et 116.)

[41] François Clautiaux, Antoine Jouglet, Jacques Carlier, and Aziz Moukrim. A new constraint programming approach for the orthogonal packing problem. *Computers & Operations Research*, 35(3) :944–959, 2008. (Cité en page 99.)

[42] E. G. Coffman, Jr., M. R. Garey, D. S. Johnson, and R. E. Tarjan. Performance bounds for level-oriented two-dimensional packing algorithms. *SIAM Journal on Computing*, 9(4) :808–826, 1980. (Cité en page 100.)

[43] Jean-François Cordeau and Gilbert Laporte. The dial-a-ride problem : models and algorithms. *Annals of Operations Research*, 153(1) :29–46, 2007. (Cité en page 64.)

[44] Jean-François Côté, Mauro Dell'Amico, and Manuel Iori. Combinatorial benders' cuts for the strip packing problem. *Operational Research*, 62(3) :643–661, 2014. (Cité en page 100.)

[45] Jean-François Côté, Michel Gendreau, and Jean-Yves Potvin. An exact algorithm for the two-dimensional orthogonal packing problem with unloading constraints. *Operational Research*, 62(5) :1126–1141, 2014. (Cité en pages 98 et 99.)

[46] Rémi Coulom. Efficient selectivity and backup operators in monte-carlo tree search. In *Proceedings of the 5th International Confer- ence on Computers and Games*, pages 72–83, 2006. (Cité en pages 27 et 31.)

[47] Yaodong Cui, Liu Yang, and Qiulian Chen. Heuristic for the rectangular strip packing problem with rotation of items. *Computers & Operations Research*, 40(4) :1094–1099, 2013. (Cité en page 100.)

[48] Yi-Ping Cui, Yaodong Cui, and Tianbing Tang. Sequential heuristic for the two-dimensional bin-packing problem. *European Journal of Operational Research*, 240(1) :43–53, 2015. (Cité en page 100.)

[49] Mauro Dell'Amico, Silvano Martello, and Daniele Vigo. A lower bound for the non-oriented two-dimensional bin packing problem. *Discrete Applied Mathematics*, 118(1-2) :13–24, 2002. (Cité en pages 99 et 110.)

[50] Jacques Desrosiers, Yvan Dumas, and François Soumis. A dynamic programming solution of the large-scale single-vehicle dial-a-ride problem with time windows. *American Journal of Mathematical and Management Sciences*, 6 (3-4) :301–325, 1986. (Cité en page 63.)

[51] Michel Deudon, Pierre Cournut, Alexandre Lacoste, Yossiri Adulyasak, and Louis-Martin Rousseau. Learning heuristics for the TSP by policy gradient. In *Proceedings of the 15th International Conference on Integration of Constraint Programming, Artificial Intelligence, and Operations Research*, pages 170–181, 2018. (Cité en pages 27 et 35.)

[52] Sylvain Ducomman, Hadrien Cambazard, and Bernard Penz. Alternative filtering for the weighted circuit constraint : Comparing lower bounds for the TSP and solving TSPTW. In *Proceedings of the 30th AAAI Conference on Artificial Intelligence*, pages 3390–3396, 2016. (Cité en pages 67 et 68.)

[53] Irina Dumitrescu, Stefan Ropke, Jean-François Cordeau, and Gilbert Laporte. The traveling salesman problem with pickup and delivery : polyhedral results and a branch-and-cut algorithm. *Mathematical Programming*, 121(2) :269–305, 2010. (Cité en page 63.)

[54] Stefan Edelkamp and Max Gath. Solving single vehicle pickup and delivery problems with time windows and capacity constraints using nested monte-carlo search. In *Proceedings of the 6th International Conference on Agents and Artificial Intelligence, Volume 1*, pages 22–33, 2014. (Cité en page 63.)

[55] Marc Etheve, Zacharie Alès, Côme Bissuel, Olivier Juan, and Safia Kedad-Sidhoum. Reinforcement learning for variable selection in a branch and bound algorithm. In *Proceedings of the 17th International Conference on Integration of Constraint Programming, Artificial Intelligence, and Operations Research*, pages 176–185, 2020. (Cité en page 42.)

[56] Sándor P. Fekete and Jörg Schepers. A combinatorial characterization of higher-dimensional orthogonal packing. *Mathematics of Operations Research*, 29(2) :353–368, 2004. (Cité en page 99.)

[57] Sándor P. Fekete, Jörg Schepers, and Jan van der Veen. An exact algorithm for higher-dimensional orthogonal packing. *Operational Research*, 55(3) :569–587, 2007. (Cité en page 99.)

[58] Ryota Furuoka and Shimpei Matsumoto. Worker's knowledge evaluation with single-player monte carlo tree search for a practical reentrant scheduling problem. *Artificial Life and Robotics*, 22(1) :130–138, 2017. (Cité en page 27.)

[59] Maxime Gasse, Didier Chételat, Nicola Ferroni, Laurent Charlin, and Andrea Lodi. Exact combinatorial optimization with graph convolutional neural networks. In *Proceedings of the 33rd International Conference on Neural Information Processing Systems*, pages 15554–15566, 2019. (Cité en page 36.)

[60] Maxime Gasse, Quentin Cappart, Jonas Charfreitag, Laurent Charlin, Didier Chételat, Antonia Chmiela, Justin Dumouchelle, Ambros M. Gleixner, Aleksandr M. Kazachkov, Elias B. Khalil, Pawel Lichocki, Andrea Lodi, Miles Lubin, Chris J. Maddison, Christopher Morris, Dimitri J. Papageorgiou, Augustin Parjadis, Sebastian Pokutta, Antoine Prouvost, Lara Scavuzzo, Giulia Zarpellon, Linxin Yang, Sha Lai, Akang Wang, Xiaodong Luo, Xiang Zhou, Haohan Huang, Sheng Cheng Shao, Yuanming Zhu, Dong Zhang, Tao Quan, Zixuan Cao, Yang Xu, Zhewei Huang, Shuchang Zhou, Chen Binbin, He Minggui, Hao Hao, Zhang Zhiyu, An Zhiwu, and Mao Kun. The machine learning for combinatorial optimization competition (ML4CO) : results and insights. *arXiv*, 2203.02433, 2022. (Cité en page 34.)

[61] Gecode Team. Gecode : Generic constraint development environment, 2006. Available from `http://www.gecode.org`. (Cité en page 140.)

[62] Sylvain Gelly and David Silver. Combining online and offline knowledge in UCT. In *Proceedings of the Twenty-Fourth International Conference on Machine Learning*, pages 273–280, 2007. (Cité en pages 30 et 34.)

[63] Michel Gendreau, Gilbert Laporte, and Daniele Vigo. Heuristics for the traveling salesman problem with pickup and delivery. *Computers & Operations Research*, 26(7) :699–714, 1999. (Cité en page 63.)

[64] Fred Glover and Manuel Laguna. *Tabu Search*, pages 2093–2229. 1998. (Cité en page 16.)

[65] Jack Goffinet and Raghuram Ramanujan. Monte-carlo tree search for the maximum satisfiability problem. In *Proceedings of the 22nd International Conference on Principles and Practice of Constraint Programming*, pages 251–267, 2016. (Cité en pages 31, 32 et 34.)

[66] Bruce L. Golden, Larry Levy, and Rakesh Vohra. The orienteering problem. *Naval Research Logistics*, 34(3) :307–318, 1987. (Cité en pages 7 et 46.)

[67] Carla P. Gomes, Bart Selman, and Henry A. Kautz. Boosting combinatorial search through randomization. In *Proceedings of the 15th National Conference on Artificial Intelligence and 10th Conference on Innovative Applications of Artificial Intelligence,*, pages 431–437, 1998. (Cité en page 13.)

[68] Prateek Gupta, Maxime Gasse, Elias B. Khalil, Pawan Kumar Mudigonda, Andrea Lodi, and Yoshua Bengio. Hybrid models for learning to branch. In *Proceedings of the 33rd International Conference on Advances in Neural Information Processing Systems*, 2020. (Cité en pages 36 et 37.)

[69] Eleni Hadjiconstantinou and Nicos Christofides. An exact algorithm for general, orthogonal, two-dimensional knapsack problems. *European Journal of Operational Research*, 83(1) :39–56, 1995. (Cité en page 99.)

[70] Christoph Hansknecht, Imke Joormann, and Sebastian Stiller. Cuts, primal heuristics, and learning to branch for the time-dependent traveling salesman problem. *arXiv*, 1805.01415, 2018. (Cité en page 36.)

[71] Robert M. Haralick and Gordon L. Elliott. Increasing tree search efficiency for constraint satisfaction problems. *Artificial Intelligence*, 14(3) :263–313, 1980. (Cité en page 12.)

[72] William D. Harvey and Matthew L. Ginsberg. Limited discrepancy search. In *Proceedings of the 14th International Joint Conference on Artificial Intelligence*, pages 607–615, 1995. (Cité en page 77.)

[73] Joseph El Hayek, Aziz Moukrim, and Stéphane Nègre. New resolution algorithm and pretreatments for the two-dimensional bin-packing problem. *Computers & Operations Research*, 35(10) :3184–3201, 2008. (Cité en page 100.)

[74] He He, Hal Daumé III, and Jason Eisner. Learning to search in branch and bound algorithms. In *Proceedings of the 27th International Conference on Advances in Neural Information Processing Systems*, pages 3293–3301, 2014. (Cité en pages 37 et 42.)

[75] Pascal Van Hentenryck and Jean-Philippe Carillon. Generality versus specificity : An experience with AI and OR techniques. In *Proceedings of the 7th National Conference on Artificial Intelligence*, pages 660–664, 1988. (Cité en pages 6 et 68.)

[76] Hipólito Hernández-Pérez and Juan José Salazar González. A branch-and-cut algorithm for a traveling salesman problem with pickup and delivery. *Discrete Applied Mathematics*, 145(1) :126–139, 2004. (Cité en page 63.)

[77] Holger H. Hoos. Automated algorithm configuration and parameter tuning. In *Autonomous Search*, pages 37–71. 2012. (Cité en page 39.)

[78] John J. Hopfield and David W. Tank. "neural" computation of decisions in optimization problems. *Biological Cybernetics*, 52 :141—1552, 1985. (Cité en page 34.)

[79] André Hottung and Kevin Tierney. Neural large neighborhood search for the capacitated vehicle routing problem. In *Proceedings of the 24th European Conference on Artificial Intelligence*, pages 443–450, 2020. (Cité en page 35.)

[80] André Hottung, Shunji Tanaka, and Kevin Tierney. Deep learning assisted heuristic tree search for the container pre-marshalling problem. *Computers & Operations Research*, 113, 2020. (Cité en page 37.)

[81] Jiayi Huang, Md. Mostofa Ali Patwary, and Gregory F. Diamos. Coloring big graphs with alphagozero. *arXiv*, 1902.10162, 2019. (Cité en pages 32 et 51.)

[82] Zeren Huang, Kerong Wang, Furui Liu, Hui-Ling Zhen, Weinan Zhang, Mingxuan Yuan, Jianye Hao, Yong Yu, and Jun Wang. Learning to select cuts for efficient mixed-integer programming. *Pattern Recognition*, 123 :108353, 2022. (Cité en page 37.)

[83] Marc Huber and Günther R. Raidl. Learning beam search : Utilizing machine learning to guide beam search for solving combinatorial optimization problems. In *Proceedings of the 7th International Conference on Machine Learning, Optimization, and Data Science, Part II*, pages 283–298, 2021. (Cité en page 37.)

[84] Ahmed Hussein, Mohamed Medhat Gaber, Eyad Elyan, and Chrisina Jayne. Imitation learning : A survey of learning methods. *ACM Computing Survey*, 50(2) :21 :1–21 :35, 2017. (Cité en page 36.)

[85] Shinji Imahori and Mutsunori Yagiura. The best-fit heuristic for the rectangular strip packing problem : An efficient implementation and the worst-case approximation ratio. *Computers & Operations Research*, 37(2) :325–333, 2010. (Cité en page 105.)

[86] Klaus Jansen and Rob van Stee. On strip packing with rotations. In *Proceedings of the 37th ACM Symposium on Theory of Computing*, pages 755–761, 2005. (Cité en page 100.)

[87] Jorik Jooken, Pieter Leyman, Patrick De Causmaecker, and Tony Wauters. Exploring search space trees using an adapted version of monte carlo tree

search for a combinatorial optimization problem. *arXiv*, 2010.11523, 2020. (Cité en pages 30, 31 et 32.)

[88] Chaitanya K. Joshi, Thomas Laurent, and Xavier Bresson. An efficient graph convolutional network technique for the travelling salesman problem. *arXiv*, 1906.01227, 2019. (Cité en page 35.)

[89] Chaitanya K. Joshi, Quentin Cappart, Louis-Martin Rousseau, and Thomas Laurent. Learning TSP requires rethinking generalization. In *Proceedings of the 27th International Conference on Principles and Practice of Constraint Programming*, pages 33 :1–33 :21, 2021. (Cité en page 35.)

[90] Bahman Kalantari, Arthur V. Hill, and Sant R. Arora. An algorithm for the traveling salesman problem with pickup and delivery customers. *European Journal of Operational Research*, 22(3) :377–386, 1985. (Cité en page 63.)

[91] Oliver Keszocze, Kenneth Schmitz, Jens Schloeter, and Rolf Drechsler. *Improving SAT Solving Using Monte Carlo Tree Search-Based Clause Learning*, pages 107–133. 2020. (Cité en pages 32, 34 et 52.)

[92] Elias B. Khalil, Hanjun Dai, Yuyu Zhang, Bistra Dilkina, and Le Song. Learning combinatorial optimization algorithms over graphs. In *Proceedings of the 30th International Conference on Advances in Neural Information Processing Systems*, pages 6348–6358, 2017. (Cité en pages 35, 36 et 51.)

[93] Elias B. Khalil, Bistra Dilkina, George L. Nemhauser, Shabbir Ahmed, and Yufen Shao. Learning to run heuristics in tree search. In *Proceedings of the 26th International Joint Conference on Artificial Intelligence*, pages 659–666, 2017. (Cité en page 37.)

[94] Elias B. Khalil, Pashootan Vaezipoor, and Bistra Dilkina. Finding backdoors to integer programs : A monte carlo tree search framework. *arXiv*, 2110.08423, 2021. (Cité en pages 32 et 33.)

[95] Elias Boutros Khalil, Pierre Le Bodic, Le Song, George L. Nemhauser, and Bistra Dilkina. Learning to branch in mixed integer programming. In *Proceedings of the 30th AAAI Conference on Artificial Intelligence*, pages 724–731, 2016. (Cité en page 36.)

[96] Scott Kirkpatrick, D. Gelatt Jr., and Mario P. Vecchi. Optimization by simmulated annealing. *Science*, 220(4598) :671–680, 1983. (Cité en page 16.)

[97] Levente Kocsis and Csaba Szepesvári. Bandit based monte-carlo planning. In *Proceedings of the 17th European Conference on Machine Learning*, pages 282–293, 2006. (Cité en pages 27 et 30.)

[98] Wouter Kool and Max Welling. Attention solves your TSP. *arXiv*, 1803.08475, 2018. (Cité en page 35.)

[99] Richard E. Korf, Michael D. Moffitt, and Martha E. Pollack. Optimal rectangle packing. *Annals of Operations Research*, 179(1) :261–295, 2010. (Cité en pages 98, 99 et 100.)

[100] James Kotary, Ferdinando Fioretto, Pascal Van Hentenryck, and Bryan Wilder. End-to-end constrained optimization learning : A survey. In *Proceedings of the 30th International Joint Conference on Artificial Intelligence*, pages 4475–4482, 2021. (Cité en pages 34 et 35.)

[101] Lars Kotthoff. Algorithm selection for combinatorial search problems : A survey. In *Data Mining and Constraint Programming - Foundations of a Cross-Disciplinary Approach*, pages 149–190. 2016. (Cité en page 39.)

[102] Markus Kruber, Marco E. Lübbecke, and Axel Parmentier. Learning when to use a decomposition. In *Proceedings of the 14th International Conference on Integration of Artificial Intelligence and Operation Research Techniques in Contraint Programming*, pages 202–210, 2017. (Cité en page 39.)

[103] Hok-Leung Kung, Shu-Jun Yang, and Kuo-Chan Huang. An improved monte carlo tree search approach to workflow scheduling. *Connection Science*, 34 (1) :1221–1251, 2022. (Cité en pages 32 et 33.)

[104] Philippe Laborie. Algorithms for propagating resource constraints in AI planning and scheduling : Existing approaches and new results. *Artificial Intelligence*, 143(2) :151–188, 2003. (Cité en page 66.)

[105] Antoine Landrieu, Yazid Mati, and Zdenek Binder. A tabu search heuristic for the single vehicle pickup and delivery problem with time windows. *Journal of Intelligent Manufacturing*, 12(5-6) :497–508, 2001. (Cité en page 63.)

[106] Alexandre Laterre, Yunguan Fu, Mohamed Khalil Jabri, Alain-Sam Cohen, David Kas, Karl Hajjar, Torbjorn S. Dahl, Amine Kerkeni, and Karim Beguir. Ranked reward : Enabling self-play reinforcement learning for combinatorial optimization. *arXiv*, 1807.01672, 2018. (Cité en page 32.)

[107] Jean-Louis Laurière. A language and a program for stating and solving combinatorial problems. *Artificial Intelligence*, 10(1) :29–127, 1978. (Cité en pages 6, 65 et 68.)

[108] Hailiang Li, Yan Wang, DanPeng Ma, Yang Fang, and Zhibin Lei. Quasi-monte-carlo tree search for 3d bin packing. In *Proceeding of the 1st Chinese Conference on Pattern Recognition and Computer Vision, Part I*, pages 384–396, 2018. (Cité en pages 32 et 33.)

[109] Zhuwen Li, Qifeng Chen, and Vladlen Koltun. Combinatorial optimization with graph convolutional networks and guided tree search. In *Proceedings of the 31st International Conference on Advances in Neural Information Processing Systems 31*, pages 537–546, 2018. (Cité en pages 35 et 36.)

[110] Giovanni Di Liberto, Serdar Kadioglu, Kevin Leo, and Yuri Malitsky. DASH : dynamic approach for switching heuristics. *European Journal of Operational Research*, 248(3) :943–953, 2016. (Cité en page 39.)

[111] Kui Liu, Zhiwei Wu, Qing Wu, and Yuxia Cheng. Smart DAG task scheduling with efficient pruning-based MCTS method. In *Proceedings of the 9th IEEE International Conference on Parallel & Distributed Processing with Applications, Big Data & Cloud Computing, Sustainable Computing & Communications, Social Computing & Networking,*, pages 348–355, 2019. (Cité en page 32.)

[112] Andrea Lodi and Giulia Zarpellon. On learning and branching : a survey. *TOP*, 25 :207—236, 2017. (Cité en pages 34 et 36.)

[113] Andrea Lodi, Silvano Martello, and Daniele Vigo. Heuristic and metaheuristic approaches for a class of two-dimensional bin packing problems. *INFORMS Journal on Computing*, 11(4) :345–357, 1999. (Cité en pages 98 et 100.)

[114] Manuel Loth, Michèle Sebag, Youssef Hamadi, and Marc Schoenauer. Bandit-based search for constraint programming. In *Proceedings of the 19th International Conference on Principles and Practice of Constraint Programming*, pages 464–480, 2013. (Cité en pages 31, 32, 33 et 140.)

[115] Hao Lu, Xingwen Zhang, and Shuang Yang. A learning-based iterative method for solving vehicle routing problems. In *Proceedings of the 8th International Conference on Learning Representations*, 2020. (Cité en page 38.)

[116] Michael Luby, Alistair Sinclair, and David Zuckerman. Optimal speedup of las vegas algorithms. *Information Processing Letters*, 47(4) :173–180, 1993. (Cité en pages 13, 71 et 77.)

[117] Flavien Lucas. *Résolution de problèmes réalistes de tournées à flotte hétérogène en milieu urbain : vers un solveur adaptatif mêlant recherche opérationnelle et apprentissage automatique.* PhD thesis, University Bretagne Sud, Vannes, Morbihan, France, 2020. (Cité en page 38.)

[118] Alan K. Mackworth. Consistency in networks of relations. *Artificial Intelligence*, 8(1) :99–118, 1977. (Cité en page 11.)

[119] Maryam Karimi Mamaghan, Mehrdad Mohammadi, Patrick Meyer, Amir Mohammad Karimi-Mamaghan, and El-Ghazali Talbi. Machine learning at the service of meta-heuristics for solving combinatorial optimization problems : A state-of-the-art. *European Journal of Operational Research*, 296(2) :393–422, 2022. (Cité en pages 34, 37 et 38.)

[120] Jacek Mandziuk and Cezary Nejman. Uct-based approach to capacitated vehicle routing problem. In *Proceedings of the 14th International Conference*

on Artificial Intelligence and Soft Computing, Part II, pages 679–690, 2015. (Cité en pages 27, 30, 31, 32 et 33.)

[121] Peter Marbach and John N. Tsitsiklis. Simulation-based optimization of markov reward processes. *IEEE Transactions on Automatic Control*, 46(2) :191–209, 2001. (Cité en page 26.)

[122] Silvano Martello and Daniele Vigo. Exact solution of the two-dimensional finite bin packing problem. *Management Science*, 44(3) :388–399, 1998. (Cité en page 99.)

[123] Silvano Martello, Michele Monaci, and Daniele Vigo. An exact approach to the strip-packing problem. *INFORMS Journal on Computing*, 15(3) :310–319, 2003. (Cité en pages 98, 99 et 110.)

[124] Shimpei Matsumoto, Noriaki Hirosue, Kyohei Itonaga, Nobuyuki Ueno, and Hiroaki Ishii. Monte-carlo tree search for a reentrant scheduling problem. In *Proceedings of the 40th International Conference on Computers & Indutrial Engineering*, 2010. (Cité en pages 32 et 33.)

[125] Nina Mazyavkina, Sergey Sviridov, Sergei Ivanov, and Evgeny Burnaev. Reinforcement learning for combinatorial optimization : A survey. *Computers & Operations Research*, 134 :105400, 2021. (Cité en page 34.)

[126] Nenad Mladenovic and Pierre Hansen. Variable neighborhood search. *Computers & Operations Research*, 24(11) :1097–1100, 1997. (Cité en page 16.)

[127] Volodymyr Mnih, Adrià Puigdomènech Badia, Mehdi Mirza, Alex Graves, Timothy P. Lillicrap, Tim Harley, David Silver, and Koray Kavukcuoglu. Asynchronous methods for deep reinforcement learning. In *Proceedings of the 33nd International Conference on Machine Learning*, pages 1928–1937, 2016. (Cité en page 27.)

[128] Vinod Nair, Sergey Bartunov, Felix Gimeno, Ingrid von Glehn, Pawel Lichocki, Ivan Lobov, Brendan O'Donoghue, Nicolas Sonnerat, Christian Tjandraatmadja, Pengming Wang, Ravichandra Addanki, Tharindi Hapuarachchi, Thomas Keck, James Keeling, Pushmeet Kohli, Ira Ktena, Yujia Li, Oriol Vinyals, and Yori Zwols. Solving mixed integer programs using neural networks. *arXiv*, 2012.13349, 2020. (Cité en pages 36 et 38.)

[129] MohammadReza Nazari, Afshin Oroojlooy, Lawrence V. Snyder, and Martin Takác. Reinforcement learning for solving the vehicle routing problem. In *Proceedings of the 31st International Conference on Advances in Neural Information Processing Systems*, pages 9861–9871, 2018. (Cité en page 35.)

[130] Teresa Neto, Miguel Constantino, Isabel Martins, and João Pedro Pedroso. A multi-objective monte carlo tree search for forest harvest scheduling. *European Journal of Operational Research*, 282(3) :1115–1126, 2020. (Cité en page 32.)

[131] Minh Anh Nguyen, Kazushi Sano, and Vu Tu Tran. A monte carlo tree search for traveling salesman problem with drone. *Asian Transport Studies*, 6 :100028, 2020. (Cité en pages 27, 30, 31, 32 et 33.)

[132] Chris Nota and Philip S. Thomas. Is the policy gradient a gradient ? In *Proceedings of the 19th International Conference on Autonomous Agents and Multiagent Systems*, pages 939–947, 2020. (Cité en page 26.)

[133] Hande Öztop, Mehmet Fatih Tasgetiren, Levent Kandiller, and Quan-Ke Pan. A novel general variable neighborhood search through q-learning for no-idle flowshop scheduling. In *Proceedings of the IEEE Congress on Evolutionary Computation*, pages 1–8, 2020. (Cité en page 38.)

[134] Anastasia Paparrizou and Hugues Wattez. Perturbing branching heuristics in constraint solving. In *Proceedings of the 26th International Conference on Principles and Practice of Constraint Programming*, pages 496–513, 2020. (Cité en page 38.)

[135] Augustin Parjadis, Quentin Cappart, Louis-Martin Rousseau, and David Bergman. Improving branch-and-bound using decision diagrams and reinforcement learning. In *Proceedings of the 18th International Conference on Integration of Constraint Programming, Artificial Intelligence, and Operations Research*, pages 446–455, 2021. (Cité en page 37.)

[136] Axel Parmentier. Learning structured approximations of operations research problems. *arXiv*, 2107.04323, 2021. (Cité en page 36.)

[137] Axel Parmentier and Vincent T'kindt. Learning to solve the single machine scheduling problem with release times and sum of completion times. *arXiv*, 2101.01082, 2021. (Cité en page 36.)

[138] Bo Peng, Jiahai Wang, and Zizhen Zhang. A deep reinforcement learning algorithm using dynamic attention model for vehicle routing problems. *arXiv*, 2002.03282, 2020. (Cité en page 35.)

[139] Marcelo Prais and Celso C. Ribeiro. Reactive GRASP : an application to a matrix decomposition problem in TDMA traffic assignment. *INFORMS Journal of Computing*, 12(3) :164–176, 2000. (Cité en page 38.)

[140] Deniz Preil and Michael Krapp. Artificial intelligence-based inventory management : a monte carlo tree search approach. *Annals of Operations Research*, 308(1) :415–439, 2022. (Cité en page 32.)

[141] Alessandro Previti, Raghuram Ramanujan, Marco Schaerf, and Bart Selman. Monte-carlo style UCT search for boolean satisfiability. In *Proceedings of the 12th International Conference of the Italian Association for Artificial Intelligence*, pages 177–188, 2011. (Cité en pages 31, 32, 34 et 51.)

[142] Charles Prud'homme, Jean-Guillaume Fages, and Xavier Lorca. *Choco Documentation*. TASC - LS2N CNRS UMR 6241, COSLING S.A.S., 2017. (Cité en pages 71 et 83.)

[143] Martin L. Puterman and Moon Chirl Shin. Modified policy iteration algorithms for discounted markov decision problems. *Management Science*, 24 (11) :1127–1137, 1978. (Cité en page 21.)

[144] Raghuram Ramanujan and Bart Selman. Trade-offs in sampling-based adversarial planning. In *Proceedings of the 21st International Conference on Automated Planning and Scheduling*, 2011. (Cité en page 32.)

[145] Philippe Refalo. Impact-based search strategies for constraint programming. In *Proceedings of the 10th International Conference on Principles and Practice of Constraint Programming*, pages 557–571, 2004. (Cité en page 12.)

[146] Jacques Renaud, Fayez F. Boctor, and Jamal Ouenniche. A heuristic for the pickup and delivery traveling salesman problem. *Computers & Operations Research*, 27(9) :905–916, 2000. (Cité en page 63.)

[147] Jacques Renaud, Fayez F. Boctor, and Gilbert Laporte. Perturbation heuristics for the pickup and delivery traveling salesman problem. *Computers & Operations Research*, 29(9) :1129–1141, 2002. (Cité en page 63.)

[148] Adrien Rimélé, Philippe Grangier, Michel Gamache, Michel Gendreau, and Louis-Martin Rousseau. Supervised learning and tree search for real-time storage allocation in robotic mobile fulfillment systems. *arXiv*, 2106.02450, 2021. (Cité en pages 32 et 33.)

[149] Stefan Ropke and David Pisinger. An adaptive large neighborhood search heuristic for the pickup and delivery problem with time windows. *Transportation Science*, 40(4) :455–472, 2006. (Cité en pages 38, 39 et 64.)

[150] K.S. Ruland and E.Y. Rodin. The pickup and delivery problem : Faces and branch-and-cut algorithm. *Computers & Mathematics with Applications*, 33 (12) :1–13, 1997. (Cité en page 63.)

[151] Gavin Adrian Rummery and Mahesan Niranjan. On-line q-learning using connectionist systems. Technical report, Cambridge University, 1994. (Cité en page 21.)

[152] Thomas Philip Runarsson, Marc Schoenauer, and Michèle Sebag. Pilot, rollout and monte carlo tree search methods for job shop scheduling. In *Revised Selected Papers of the 6th International Conference on Learning and Intelligent Optimization*, pages 160–174, 2012. (Cité en pages 27, 31, 32, 33, 51 et 54.)

[153] Ashish Sabharwal, Horst Samulowitz, and Chandra Reddy. Guiding combinatorial optimization with UCT. In *Proceedings of the 9th International Conference on Integration of Artificial Intelligence and Operation Research Techniques in Constraint Programming for Combinatorial Optimzation Problems*, pages 356–361, 2012. (Cité en pages 30, 31 et 33.)

[154] Martin W.P. Savelsbergh. Local Search in Routing Problems with Time Windows. *Annals of Operations Research*, 4 :285–305, 1985. (Cité en page 43.)

[155] John Schulman, Sergey Levine, Pieter Abbeel, Michael I. Jordan, and Philipp Moritz. Trust region policy optimization. In *Proceedings of the 32nd International Conference on Machine Learning*, pages 1889–1897, 2015. (Cité en page 27.)

[156] David Silver, Guy Lever, Nicolas Heess, Thomas Degris, Daan Wierstra, and Martin A. Riedmiller. Deterministic policy gradient algorithms. In *Proceedings of the 31th International Conference on Machine Learning*, pages 387–395, 2014. (Cité en page 27.)

[157] David Silver, Aja Huang, Chris J. Maddison, Arthur Guez, Laurent Sifre, George van den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Vedavyas Panneershelvam, Marc Lanctot, Sander Dieleman, Dominik Grewe, John Nham, Nal Kalchbrenner, Ilya Sutskever, Timothy P. Lillicrap, Madeleine Leach, Koray Kavukcuoglu, Thore Graepel, and Demis Hassabis. Mastering the game of Go with deep neural networks and tree search. *Nature*, 529(7587) : 484–489, 2016. (Cité en pages 29 et 30.)

[158] David Silver, Julian Schrittwieser, Karen Simonyan, Ioannis Antonoglou, Aja Huang, Arthur Guez, Thomas Hubert, Lucas Baker, Matthew Lai, Adrian Bolton, Yutian Chen, Timothy P. Lillicrap, Fan Hui, Laurent Sifre, George van den Driessche, Thore Graepel, and Demis Hassabis. Mastering the game of go without human knowledge. *Nature*, 550(7676) :354–359, 2017. (Cité en page 32.)

[159] Helmut Simonis and Barry O'Sullivan. Search strategies for rectangle packing. In Peter J. Stuckey, editor, *Proceedings of the 14th International Conference on Principles and Practice of Constraint Programming*, pages 52–66, 2008. (Cité en page 99.)

[160] Kostas Stergiou. Adaptive constraint propagation in constraint satisfaction : review and evaluation. *Artificial Intelligence Review*, 54(7) :5055–5093, 2021. (Cité en page 38.)

[161] Haoran Sun, Wenbo Chen, Hui Li, and Le Song. Improving learning to branch via reinforcement learning. In *Learning Meets Combinatorial Algorithms at NeurIPS2020*, 2020. (Cité en page 36.)

[162] Richard S. Sutton. Generalization in reinforcement learning : Successful examples using sparse coarse coding. In *Proceedings of the 8th International Conference on Advances in Neural Information*, pages 1038–1044, 1995. (Cité en page 21.)

[163] Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning : An Introduction*. The MIT Press, second edition, 2018. (Cité en pages 17 et 18.)

[164] Richard S. Sutton, David A. McAllester, Satinder Singh, and Yishay Mansour. Policy gradient methods for reinforcement learning with function approximation. In *Proceedings of the 12th International Conference on Neural Information Processing Systems*, pages 1057–1063, 1999. (Cité en page 26.)

[165] El-Ghazali Talbi. Automated design of deep neural networks : A survey and unified taxonomy. *ACM Computing Surveys*, 54(2) :34 :1–34 :37, 2021. (Cité en page 34.)

[166] Yunhao Tang, Shipra Agrawal, and Yuri Faenza. Reinforcement learning for integer programming : Learning to cut. In *Proceedings of the 37th International Conference on Machine Learning*, pages 9367–9376, 2020. (Cité en page 37.)

[167] William R. Thompson. On the likelihood that one unknown probability exceeds another in view of the evidence of two samples. *Biometrika*, 25(3/4) : 285–294, 1933. (Cité en pages 27 et 28.)

[168] Renata Turkes, Kenneth Sörensen, and Lars Magnus Hvattum. Meta-analysis of metaheuristics : Quantifying the effect of adaptiveness in adaptive large neighborhood search. *European Journal of Operational Research*, 292(2) : 423–442, 2021. (Cité en page 39.)

[169] L. J. J. van der Bruggen, Jan Karel Lenstra, and P. C. Schuur. Variable-depth search for the single-vehicle pickup and delivery problem with time windows. *Transportation Science*, 27(3) :298–311, 1993. (Cité en page 63.)

[170] Jannes Verstichel, Patrick De Causmaecker, and Greet Vanden Berghe. An improved best-fit heuristic for the orthogonal strip packing problem. *International Transactions in Operational Research*, 20(5) :711–730, 2013. (Cité en pages 100, 101 et 104.)

[171] Natalia Vesselinova, Rebecca Steinert, Daniel F. Perez-Ramirez, and Magnus Boman. Learning combinatorial optimization on graphs : A survey with applications to networking. *arXiv*, 2005.11081, 2020. (Cité en page 34.)

[172] Oriol Vinyals, Meire Fortunato, and Navdeep Jaitly. Pointer networks. In *Proceedings of the 28th International Conference on Advances in Neural Information Processing Systems*, pages 2692–2700, 2015. (Cité en page 35.)

[173] Toby Walsh. Search in a small world. In *Proceedings of the 16th International Joint Conference on Artificial Intelligence*, pages 1172–1177, 1999. (Cité en page 13.)

[174] Qi Wang and Chunlei Tang. Deep reinforcement learning for transportation network combinatorial optimization : A survey. *Knowledge-Based Systems*, 233 :107526, 2021. (Cité en page 34.)

[175] Weijia Wang and Michèle Sebag. Multi-objective monte-carlo tree search. In *Proceedings of the 4th Asian Conference on Machine Learning*, pages 507–522, 2012. (Cité en page 32.)

[176] Hugues Wattez, Frédéric Koriche, Christophe Lecoutre, Anastasia Paparrizou, and Sébastien Tabary. Learning variable ordering heuristics with multi-armed bandits and restarts. In *Proceedings of the 24th European Conference on Artificial Intelligence*, pages 371–378, 2020. (Cité en page 38.)

[177] Lijun Wei, Wee-Chong Oon, Wenbin Zhu, and Andrew Lim. A skyline heuristic for the 2d rectangular packing and strip packing problems. *European Journal of Operational Research*, 215(2) :337–346, 2011. (Cité en page 100.)

[178] Ronald J. Williams. Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Machine Learning*, 8 :229–256, 1992. (Cité en page 26.)

[179] Wei Xia and Roland H. C. Yap. Learning robust search strategies using a bandit-based approach. In *Proceedings of the Thirty-Second AAAI Conference on Artificial Intelligence*, pages 6657–6665, 2018. (Cité en page 38.)

[180] Zhihao Xing and Shikui Tu. A graph neural network assisted monte carlo tree search approach to traveling salesman problem. *IEEE Access*, 8 :108418–108428, 2020. (Cité en pages 32 et 33.)

[181] Lin Xu, Frank Hutter, Holger H. Hoos, and Kevin Leyton-Brown. Satzilla : Portfolio-based algorithm selection for SAT. *arXiv*, 1111.2249, 2011. (Cité en page 39.)

[182] Ruiyang Xu, Prashank Kadam, and Karl J. Lieberherr. First-order problem solving through neural MCTS based reinforcement learning. *arXiv*, 2101.04167, 2021. (Cité en page 32.)

[183] Yu Yang, Natashia Boland, Bistra Dilkina, and Martin Savelsbergh. Learning generalized strong branching for set covering, set packing, and 0–1 knapsack problems. *European Journal of Operational Research*, 301(3) :828–840, 2022. (Cité en page 36.)

[184] Yunhao Yang and Andrew B. Whinston. A survey on reinforcement learning for combinatorial optimization. *arXiv*, 2008.12248, 2020. (Cité en page 34.)

[185] Giulia Zarpellon, Jason Jo, Andrea Lodi, and Yoshua Bengio. Parameterizing branch-and-bound search trees to learn branching policies. In *Proceedings of the 35th AAAI Conference on Artificial Intelligence*, pages 3931–3939, 2021. (Cité en page 37.)

[186] Zhaoyi Zhang, Songshan Guo, Wenbin Zhu, Wee-Chong Oon, and Andrew Lim. Space defragmentation heuristic for 2d and 3d bin packing problems. In *Proceedings of the 22nd International Joint Conference on Artificial Intelligence*, pages 699–704, 2011. (Cité en page 100.)

[187] Yangming Zhou, Jin-Kao Hao, and Béatrice Duval. Reinforcement learning based local search for grouping problems : A case study on graph coloring. *Expert Systems with Applications*, 64 :412–422, 2016. (Cité en page 38.)