Titre: Title:	Apprentissage par renforcement d'heuristiques de branchement en programmation par contraintes
Auteur: Author:	Bilel Omrani
Date:	2021
Type:	Mémoire ou thèse / Dissertation or Thesis
Référence: Citation:	Omrani, B. (2021). Apprentissage par renforcement d'heuristiques de branchement en programmation par contraintes [Mémoire de maîtrise, Polytechnique Montréal]. PolyPublie. https://publications.polymtl.ca/6571/

URL de PolyPublie: PolyPublie URL:	https://publications.polymtl.ca/6571/
Directeurs de recherche: Advisors:	Gilles Pesant, & Quentin Cappart
Programme: Program:	Génie informatique

POLYTECHNIQUE MONTRÉAL
affiliée à l'Université de Montréal

Apprentissage par renforcement d'heuristiques de branchement en Programmation par Contraintes

BILEL OMRANI
Département de génie informatique et génie logiciel

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

Mai 2021

POLYTECHNIQUE MONTRÉAL
affiliée à l'Université de Montréal

Ce mémoire intitulé :

Apprentissage par renforcement d'heuristiques de branchement en Programmation par Contraintes

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

Michel GAGNON, président
Gilles PESANT, membre et directeur de recherche
Quentin CAPPART, membre et codirecteur de recherche
Louis-Martin ROUSSEAU, membre

DÉDICACE

Je dédie ce travail à mes professeurs de classe préparatoire,
Stéphane Komilikis et Serge Dupont.

REMERCIEMENTS

Je tiens à profondément remercier mes directeurs de recherche Gilles Pesant et Quentin Cappart pour leur soutien, leur écoute et la grande flexibilité qu'ils m'ont accordé tout au long de cette expérience de recherche. Nos échanges ont été une grande source d'inspiration et de motivation qui m'a permis de mener ce projet à son terme. Un très grand remerciement à Behrouz Babaki pour toute l'aide prodiguée au début de mon projet lors de ma prise en main des clusters de Compute Canada. Je tiens également à remercier le Fonds de Recherche du Québec Nature et technologie (FRQNT) pour leur soutien financier lors de ce projet de recherche ainsi que pour la flexibilisation des conditions de financement accordée suites aux mesures sanitaires. Un grand remerciement à tous mes collègues du laboratoire Quosséça et en particulier à Fabrice pour nos conversations passionnées et passionnantes. Merci à Compute Canada pour la mise à disposition des ressources de calcul, indispensable à ce projet.

RÉSUMÉ

La résolution de problèmes combinatoires en programmation par contraintes (CP) se fait par énumération des solutions. Cette procédure exhaustive est très coûteuse et la conception d'heuristiques de branchement visant à accélérer l'exploration de l'espace des solutions est souvent indispensable. Néanmoins, la conception d'heuristiques dédiées est un processus long, nécessitant une expertise et ne se généralise pas à un large nombre de problèmes différents. Ce constat a motivé l'utilisation de modèles d'apprentissage machine pour automatiquement apprendre des heuristiques performantes sans expertise humaine. Ces modèles entraînés par apprentissage par renforcement (RL) peuvent alors être intégrés à des solveurs de programmation par contraintes au sein d'une méthode de recherche simple comme la fouille en profondeur pour explorer systématiquement l'espace des solutions.

Inspiré par les récents développement du système AlphaZero, un algorithme générique combinant recherche arborescente et apprentissage par renforcement, nous proposons un nouvel algorithme de résolution de problèmes combinatoires combinant les avantages de la programmation par contraintes avec la possibilité d'apprendre automatiquement une stratégie d'exploration de l'espace des solutions. Une analyse expérimentale préliminaire sur les problèmes du coloriage de graphe et du sac-à-dos multidimensionnel montre que cette méthode permet d'obtenir de meilleures performances (en terme de noeuds explorés) par rapport à des heuristiques classiques mais est également plus performante qu'une méthode hybride intégrant un agent de RL à une recherche en profondeur.

ABSTRACT

Discrete optimization with Constraint Programming (CP) requires an expensive enumeration of the whole solution space. In order to be tractable, CP uses carefully crafted branching heuristics to speed up the search. Designing these heuristics is time-consuming, requires a human expertise and is very problem-specific. This observation has motivated the use of Machine Learning models to automatically learn efficient heuristics from raw data, without human supervision. Models trained by Reinforcement Learning (RL) can then be integrated within CP solvers and used in combination with a simple search algorithm such as a Depth-First Search (DFS).

We propose a new algorithm to solve combinatorial problems within the CP framework, inspired by the recent advent of AlphaZero, a generic hybrid method featuring a tree search algorithm with Reinforcement Learning models. This method combines all the benefits of CP with the ability to directly learn efficient heuristics directly from raw interactions with the problem to solve. A preliminary benchmark on the graph coloring and the multiknapsack problems shows that this method is able to improve performance (*i.e.* to reduce the number of explored nodes in the tree) upon simpler heuristics but also upon another CP+RL hybrid method which features a DFS search.

TABLE DES MATIÈRES

LISTE DES TABLEAUX

LISTE DES FIGURES

LISTE DES SIGLES ET ABRÉVIATIONS

COP	Problème d'optimisation de contraintes
CP	Programmation par contraintes
CSP	Problème de satisfaction de contraintes
DFS	Recherche en profondeur
DQN	Deep-Q-Network
DRL	Apprentissage par renforcement profond
GNN	Réseau de neurones à convolution de graphes
IP	Programmation en nombres entiers
LDS	Limited Discrepancy Search
MDP	Processus de décision markovien
ML	Machine Learning
RL	Apprentissage par renforcement

CHAPITRE 1 INTRODUCTION

Planifier une tournée de véhicules, ordonner une série de tâches, ou encore trouver le meilleur coup au jeu des échecs sont autant d'exemples de problèmes appartenant à la classe des problèmes combinatoires. Un problème combinatoire consiste à trouver la meilleure solution dans un ensemble discret dit ensemble des solutions réalisables. En général, cet ensemble compte un très grand nombre d'éléments, et est décrit de manière implicite, c'est-à-dire par un ensemble de contraintes que doivent satisfaire les solutions réalisables. Omniprésents en recherche opérationnelle, l'étude et la résolution de ces problèmes est un domaine d'une grande importance pratique en planification et en logistique par exemple.

La résolution de tels problèmes passe le plus souvent par une énumération de l'ensemble des solutions possibles. En pratique cette énumération est intractable pour la plupart des problèmes de taille intéressante. La difficulté ainsi que l'importance pratique des problèmes combinatoires ont mené au développement d'une littérature riche et à la création de plusieurs paradigmes de résolution : programmation logique, solveur SAT, programmation en nombre entiers, ou encore programmation par contraintes. La programmation par contraintes (CP) est un paradigme proposant de résoudre les problèmes combinatoires en conservant et en exploitant le plus possible la structure des contraintes du problème pour réduire la taille de l'espace à énumérer et guider intelligemment la recherche.

Pour obtenir une solution en temps raisonnable, la résolution de problèmes combinatoires requiert souvent l'exploitation de connaissances a priori sur le problème. La programmation par contraintes permet d'injecter de tels a priori par le biais d'heuristiques de recherche, qui guident l'exploration de l'espace des solutions. Néanmoins, la conception de telles heuristiques demande une certaine expertise et demeure largement spécifique au problème à l'étude.

De récents travaux [1–4] proposent de déléguer la conception de telles heuristiques à des algorithmes d'apprentissage automatique. Outre la possibilité de se passer d'une expertise humaine, ces méthodes ouvrent la voie à la découverte de nouvelles façons, plus performantes, d'explorer efficacement l'espace des solutions, et donc d'accélérer la résolution des problèmes combinatoires. Basées sur de l'apprentissage par renforcement, ces propositions permettent d'apprendre des stratégies de recherche seulement sur la base d'interactions répétées avec un solveur.

Dans un autre domaine, AlphaZero [5], un algorithme proposant une interaction très étroite entre un algorithme de recherche arborescente et un modèle d'apprentissage machine, a connu un succès prodigieux au jeu de Go, un jeu de plateau très stratégique. Au vu de l'impression-

nant succès empirique de AlphaZero, il semble intéressant d'appliquer des principes similaires à un solveur de programmation par contraintes afin de valider si ces derniers apportent un gain de performance lors de la résolution de problèmes combinatoires.

1.1 Programmation par contraintes

La programmation par contraintes (CP) est un paradigme de programmation né dans les années 80, permettant la résolution de problèmes combinatoires. Ces problèmes combinatoires sont omniprésents dans un grand nombre d'applications et incluent la planification, l'ordonnancement de tâches ou encore l'alignement de séquences d'acides-aminés. La programmation par contraintes a été appliquée avec succès dans de nombreux domaines : en planification d'horaires [6–9], en maintenance [10], en diagnostic logiciel [11], en gestion de réseau [12] ou encore en bio-informatique [13]. Ce paradigme se distingue par son aspect déclaratif. En effet, en CP l'utilisateur déclare le problème qu'il cherche à résoudre par le biais d'un modèle utilisant des primitives de haut niveau appelées contraintes globales, et une grande partie de l'aspect procédural de la résolution est automatisée. L'utilisateur a cependant toujours la possibilité d'exploiter d'éventuels a priori sur le problème en influençant les procédures d'exploration de l'espace des solutions. En cela, la programmation par contraintes cherche à obtenir le meilleur des paradigmes déclaratifs (formulation naturelle et de haut niveau du problème) et procéduraux (flexibilité de l'algorithme de recherche).

Une conséquence directe de ce paradigme est que la modélisation est très facile pour l'utilisateur (bien qu'écrire un bon modèle requiert une certaine expertise) et reste plus expressive que des approches concurrentes comme la programmation en nombres entiers (IP) ou les solveurs SAT. Ces dernières approches utilisent essentiellement un seul type de primitives. Bien qu'un grand nombre de contraintes globales puissent se ramener sous la forme d'un problème IP ou encore SAT, la modélisation avec des contraintes globales permet de conserver en grande partie la structure combinatoire du problème, car elles encodent la sémantique de sous-problèmes plus riches. En plus d'une modélisation plus naturelle, cette structure permet parfois de résoudre certains problèmes beaucoup plus rapidement [14]. Cependant, les efforts de recherche des dernières décennies en programmation mathématique ont permis de considérablement accélérer les méthodes de programmation en nombres entiers [15] : les solveurs commerciaux sont désormais capables de résoudre très efficacement des modèles de l'ordre de la centaine de milliers de variables. Pareillement, les solveurs SAT offrent des stratégies de recherche générique efficaces, rendant possible la résolution de problèmes à grande échelle.

La programmation mathématique reste un outil privilégié en recherche opérationnelle. Une des raisons possibles expliquant pourquoi la CP est parfois plus difficile à appliquer est

la difficulté à concevoir des stratégies de recherche efficaces. Ces stratégies de recherche sont d'une grande importance pour accélérer la résolution mais leur conception requiert une certaine expertise et est loin d'être systématique. La conception et l'évaluation d'heuristiques constituent une partie importante de la recherche académique en CP [16–18].

La programmation par contraintes a historiquement emprunté des outils à de nombreuses autres disciplines. L'hybridation avec la recherche opérationnelle s'est avérée particulièrement fertile, avec de nombreuses méthodes à l'intersection de la programmation par contraintes et de la programmation mathématique : filtrage par coûts réduits [19], CP-based branch-and-price [20], décomposition de Benders [21]. Des méthodes issues d'autres littératures témoignent également de la grande richesse de l'écosystème en programmation par contraintes. Citons notamment les méthodes à l'intersection avec les solveurs SAT [22], la recherche locale [23], ou encore les diagrammes de décision [24, 25].

Plus récemment, l'application du Machine Learning (ML) à la résolution de problèmes combinatoires commence à attirer l'attention de la communauté. [3] propose une méthode hybride utilisant un algorithme d'apprentissage automatique entraîné par Reinforcement Learning (RL) comme heuristique pour guider le recherche en sein d'un solveur CP. Ces travaux ont été récemment généralisés dans [2], qui propose un solveur hybride flexible pouvant s'accommoder d'un grand nombre de problèmes combinatoires.

1.2 Apprentissage par renforcement et AlphaZero

Durant la dernière décennie, l'apprentissage par renforcement a connu un développement rapide et permis des progrès remarquables dans de nombreux domaines (robotique [26], commande optimale [27], jeu vidéo [28], etc.). L'une des applications récentes les plus spectaculaires de ce paradigme est sans doute le succès du système AlphaGo en 2017 face à Lee Sedol, l'un des meilleurs joueurs au monde de Go [29]. Cet algorithme, combinant astucieusement recherche arborescente, réseaux de neurones, self-play et apprentissage par renforcement a permis d'obtenir d'excellents résultats sur ce jeu très stratégique, longtemps considéré comme l'un des grands défis en intelligence artificielle. Ce système a depuis été plusieurs fois amélioré et généralisé : AlphaZero [30], MuZero [31] en sont les versions les plus récentes. Outre le regain d'intérêt pour le jeu de Go que suscitent ces avancées, c'est bien la généralité des techniques d'apprentissage utilisées qui expliquent le succès de tels systèmes.

1.3 Objectifs de recherche

Face au succès du système AlphaZero ainsi qu'à l'intérêt récent que suscite l'apprentissage par renforcement en optimisation combinatoire, il semble naturel de chercher à appliquer ces idées en programmation par contraintes. AlphaZero combine plusieurs idées déjà bien connues en intelligence artificielle y compris en CP (recherche arborescente, randomisation, restarts) mais intègre à ces procédures des modèles d'apprentissage entraînés par apprentissage par renforcement pour naviguer au sein de l'espace gigantesque de solutions possibles sur des problèmes de grandes dimensions. La méthode proposée dans [2] est une première étape prometteuse dans la direction d'une méthode hybride CP et RL pour la résolution de problèmes combinatoires. Néanmoins, cette méthode repose sur des algorithmes de recherche assez simple (recherche de type fouille en profondeur avec backtracking chronologique) et n'explore pas les récentes avancées proposées par les systèmes plus complexes tels qu'AlphaZero. Dans ce mémoire, nous cherchons à étendre les travaux précédents en proposant une nouvelle méthode de recherche librement inspirée d'AlphaZero et adaptée aux besoins de la programmation par contraintes. Nous proposons également une analyse expérimentale préliminaire afin d'en étudier les performances et propriétés et cherchons à la comparer avec des méthodes plus traditionnelles en CP, mais également à la méthode originale proposée dans [2]. L'objectif premier de cette recherche est de proposer une nouvelle méthode hybride CP/RL et d'en valider la pertinence pour de futurs travaux. Cette étude expérimentale, bien que préliminaire, permet de conjecturer que ce nouvel algorithme permet effectivement de réduire le nombre de noeuds explorés par rapport à une méthode hybride plus simple utilisant une recherche en profondeur. Il semble également que la méthode proposée dans ce mémoire réduise significativement la sensibilité de l'apprentissage en la distribution d'entraînement, améliorant ainsi la robustesse et permettant au praticien de réutiliser un modèle entraîné sur certains problèmes et de l'appliquer à des problèmes légèrement différents sans souffrir d'une perte de performance trop importante.

1.4 Plan du mémoire

Ce mémoire est divisé en 5 chapitres, dont la présente introduction forme le chapitre 1. Le chapitre 2 présente les notions de base en programmation par contraintes, en apprentissage par renforcement ainsi qu'une présentation simplifiée du système AlphaZero [30], permettant ainsi de replacer les objectifs de recherche de ce mémoire en contexte. Le chapitre 3 présente plus en détail la méthode hybride CP et RL proposée dans [2], sur laquelle nous nous basons pour construire notre propre méthode. Ce chapitre présente également la contribution princi-

pale de ce mémoire, à savoir le principe de la méthode hybride inspirée d'AlphaZero appliquée en programmation par contraintes. Le chapitre 4 consiste en une évaluation expérimentale préliminaire de cette méthode ainsi qu'une discussion des résultats obtenus. Finalement, le chapitre 5 conclut ce mémoire et donne quelques opportunités pour de futurs travaux sur ce sujet.

CHAPITRE 2 NOTIONS DE BASE ET REVUE DE LITTÉRATURE

2.1 Programmation par contraintes

Cette section présente brièvement les concepts et techniques de base en programmation par contraintes. Ce contenu est librement adapté de [32, Chapitre 2-4].

2.1.1 Problèmes combinatoires et modélisation

Un modèle en programmation par contraintes est donné par un ensemble fini de variables de décision identifiées par l'utilisateur, chacune pouvant prendre un nombre fini de valeurs. Des contraintes sont spécifiées sur les valeurs que peuvent prendre ces variables. Ce modèle définit un *problème de satisfaction de contraintes* (CSP). Formellement, un CSP est donné par un triplet $\langle X, D, C\rangle$ où

— $X = \{x_1, \ldots, x_n\}$ est un ensemble fini de variables,

— $D = \{v_1, \ldots, v_m\}$ est un ensemble fini de valeurs,

— $C = \{c_1, \ldots, c_p\}$ est un ensemble fini de contraintes.

Chaque variable x prend sa valeur dans un sous-ensemble de D noté $D(x)$ appelé son *domaine*. Une contrainte c est une relation sur un sous-ensemble de variables appelé sa *portée*. Le graphe de cette relation spécifie les *solutions* de la contrainte. Le problème de satisfaction de contraintes est de trouver une assignation de chacune de variables telle que toutes les contraintes soient satisfaites (ou de prouver qu'une telle assignation n'existe pas).

Illustrons cette définition avec le problème du *Sudoku*. Étant donnée une grille 9×9 partiellement remplie de chiffres entre 1 et 9, l'objectif est de remplir la grille avec des chiffres de sorte que

1. Tous les chiffres sur une même ligne soient distincts,
2. Tous les chiffres sur une même colonne soient distincts,
3. Tous les chiffres d'un même bloc 3×3 soient distincts.

Une modélisation à l'aide d'un programme par contraintes pour ce problème peut être la suivante : chaque case du Sudoku est associée à une variable $x_{i,j}$ dont le domaine est $\{1, \ldots, 9\}$ sauf certaines variables correspondantes aux cases déjà remplies (auquel cas leur domaine est réduit à un singleton). Les contraintes sont données à l'aide d'une primitive `all-different` spécifiant que les variables dans sa portée doivent prendre des valeurs différentes. Nous spécifions donc les contraintes :

	2		5		1		9	
8			2		3			6
	3			6			7	
		1				6		
5	4						1	9
		2				7		
	9			3			8	
2			8		4			7
	1		9		7		6	

Figure 2.1 Exemple de Sudoku non résolu

1. $\forall i \in \{1, \ldots, 9\}$, `all-different`$(\{x_{i,j} \mid j \in \{1, \ldots, 9\}\})$,
2. $\forall j \in \{1, \ldots, 9\}$, `all-different`$(\{x_{i,j} \mid i \in \{1, \ldots, 9\}\})$,
3. $\forall k, l \in \{0, 1, 2\}$, `all-different`$(\{x_{3k+i,3l+j} \mid i, j \in \{1, 2, 3\}\})$.

Un autre type de problème combinatoire est celui du problème d'*optimisation de contraintes* (COP) dans lequel la solution, en plus de vérifier un CSP, doit minimiser une fonction objectif. Formellement, un COP est un 4-uplet $\langle X, D, C, f\rangle$ tel que $\langle X, D, C\rangle$ est un CSP et $f : D(x_1) \times \cdots \times D(x_n) \to \mathbb{R}$ est une fonction à minimiser sur l'ensemble des solutions réalisables de $\langle X, D, C\rangle$. Un exemple classique de tel problème est celui du problème du voyageur de commerce (*Traveling Salesman Problem*). Étant donné un graphe pondéré (V, E) à n sommets, quel circuit passant par tous les sommets minimise la somme des poids le long de ce dernier ? Ce problème peut apparaître comme une sous-structure d'autres problèmes plus complexes, comme c'est le cas des variantes *Traveling Salesman Problem with Time Windows* [33] ou encore *Multiple Traveling Salesman Problem* [34]. Il est donc possible de créer une contrainte globale pour ce problème (englobant toute la logique d'inférence associée) et de réutiliser cette contrainte comme une primitive pour ces variantes. Cela illustre la flexibilité dont dispose le praticien dans la modélisation. La littérature en programmation par contraintes compte un large catalogue de contraintes globales haut-niveau : contrainte de circuit hamiltonien, *packing*, etc. [35].

Étant donné sa généralité, il est peu surprenant d'apprendre que le formalisme du CSP est généralement utilisé pour modéliser des problèmes NP-difficiles/NP-complets. Une conséquence directe de ce résultat est que l'on ne sait pas formuler de méthodes de résolution avec une garantie d'être plus rapides qu'une recherche exhaustive de l'ensemble des solutions. Cependant une recherche exhaustive est intractable même pour des problèmes relativement petits. La programmation par contraintes utilise des techniques avancées d'inférence logique et des stratégies d'exploration pour considérablement réduire le temps de résolution en évitant une

recherche exhaustive, tout en garantissant un certificat de faisabilité (d'optimalité pour un COP) ou d'infaisabilité.

2.1.2 Filtrage et propagation des contraintes

L'avantage apporté par l'utilisation de primitives de haut niveau lors de la modélisation est que chaque contrainte peut apporter des mécanismes d'inférence permettant d'obtenir un certain niveau de *cohérence locale*. Autrement dit, une sous-routine locale à chaque contrainte permet de retirer un certain nombre de valeurs dans le domaine des variables de sa portée : on parle de *filtrage*. Il existe différents niveaux de cohérence locale pour une contrainte (cohérence d'arc, de domaine, de bornes, etc.), chacune traduisant une condition nécessaire sur les valeurs que peuvent prendre les variables de sa portée. Prenons l'exemple de la cohérence de domaine. Soit une contrainte c sur les variables $x_1, \ldots, x_k$. La cohérence de domaine est atteinte si pour tout $1 \leq i \leq k$, et pour toute valeur v dans $D(x_i)$ on a

$$D(x_1) \times \cdots \times D(x_{i-1}) \times \{v\} \times D(x_{i+1}) \times \cdots \times D(x_k) \cap c \neq \emptyset.$$

Il s'agit bien sûr d'une condition nécessaire à l'existence d'une solution à la contrainte c faisant intervenir $x_i = v$. Autrement, l'assignation $x_i = v$ n'est pas supportée par les autres domaines et cela suffit à démontrer que v ne participe à aucune solution. Sur la Figure 2.1, intéressons-nous à la case en haut à gauche. Initialement le modèle nous donne le domaine $D(x_{1,1}) = \{1, \ldots, 9\}$. Mais au vu des contraintes, il est facile de voir qu'un certain nombre de valeurs ne sont pas supportées. Une première étape de filtrage permet de réduire le domaine à $D(x_{1,1}) = \{4, 6, 7\}$. Le retrait d'une valeur du domaine d'une variable peut entraîner le retrait en cascade de valeurs dans le domaine d'autres variables, car certaines propriétés de cohérence pour d'autres contraintes ne sont plus satisfaites. On parle alors de *propagation*, un puissant mécanisme permettant de réduire la taille du problème à résoudre.

La propagation des contraintes s'effectue par le biais d'un algorithme dit de point-fixe. Cet algorithme commence par déclencher séquentiellement les algorithmes de filtrage de chaque contrainte du problème. Comme vu précédemment, si au moins une valeur d'un domaine est supprimée par une contrainte lors de ce premier passage, il est possible que la cohérence locale ne soit plus respectée pour une autre contrainte. Il est dès lors nécessaire de planifier une deuxième itération de filtrage. Ce processus est itéré jusqu'à l'obtention d'un point fixe, c'est-à-dire jusqu'à ce qu'aucune valeur ne soit supprimée au cours d'une itération de l'algorithme.

2.1.3 Algorithmes de recherche et stratégies de branchement

Il est à noter que le mécanisme de propagation des contraintes est souvent insuffisant pour complètement réduire les domaines soit à des singletons (auquel cas le CSP est résolu) soit à l'ensemble vide (auquel cas le CSP n'admet pas de solution). Après que le processus de filtrage ait été itéré jusqu'à l'obtention d'un point fixe, il faut un autre moyen pour faire progresser la résolution. L'idée consiste à partitionner l'espace des solutions en plusieurs régions et à explorer récursivement ces régions jusqu'à l'obtention d'une solution. En pratique, un algorithme de *backtracking* est utilisé pour construire un arbre de recherche en effectuant des *décisions de branchement*. Si un sous-arbre admet une solution, alors le CSP est résolu, et s'il est démontré comme incohérent alors l'algorithme *backtrack*, c'est-à-dire explore une autre région selon des modalités spécifiées par l'algorithme utilisé. La qualité des décisions de branchement joue un rôle crucial dans l'exploration efficace de l'espace des solutions. De mauvaises décisions de branchement mènent à un nombre important d'échecs et augmentent le temps de résolution. En programmation par contraintes, il est d'usage d'utiliser des branchements binaires pour lesquels le sous-arbre gauche correspond à l'assignation d'une valeur à une variable ($x = v$), et le sous-arbre droit correspond à l'ajout de la négation de cette assignation ($x \neq v$) comme contrainte. Une telle décision de branchement se fait en deux étapes :

1. Une heuristique de choix de variable choisit la variable x du branchement,
2. Une heuristique de choix de valeur choisit la valeur $v \in D(x)$ assignée à x.

Ces deux types d'heuristiques obéissent souvent à deux principes de conception différents.

L'heuristique de choix de variable obéit au *Fail-First Principle* : on choisit la variable la plus susceptible de provoquer un échec. L'explication est la suivante : pour résoudre un CSP, toutes les variables doivent être assignées. Sachant que toutes les variables seront assignées à un moment ou un autre de la recherche, autant commencer par la variable la plus susceptible de provoquer un échec : il vaut mieux échouer tôt dans la recherche que tard, car cela impliquerait d'échouer dans de nombreux sous-arbres ce qui retarde la recherche. Par exemple, l'heuristique `dom` choisit la variable avec le plus petit domaine, l'intuition étant que si le domaine est déjà petit, il est généralement plus probable qu'après filtrage, le domaine devienne vide.

L'heuristique de choix de valeur obéit à un principe opposé : le *Succeed-First Principle*. On choisit la valeur la plus susceptible d'aboutir à une solution. L'explication est que contrairement aux variables, nous ne sommes pas obligés d'explorer toutes les valeurs. Au contraire, nous souhaitons explorer le moins de valeurs possible avant de trouver une solution réalisable. Autant donc choisir la valeur permettant d'aboutir le plus vite à une solution.

L'algorithme de backtracking spécifie la procédure en cas d'échec à un certain nœud de la recherche. Le parcours en profondeur (ou *chronological backtracking*) est le type de backtracking le plus simple. L'algorithme de recherche suit l'heuristique de branchement (sous-arbre gauche) jusqu'à aboutir à une feuille. Si cette feuille constitue une solution, la recherche s'arrête. Dans le cas contraire, l'algorithme fouille récursivement le sous-arbre droit au-dessus du niveau ayant provoqué l'échec. La Figure 2.2 illustre les premières étapes d'un parcours en profondeur (DFS) : à partir d'un état racine, le DFS parcourt récursivement le sous-arbre gauche puis le sous-arbre droit. Les feuilles en rouge représentent un échec (preuve d'insatisfaisabilité). L'algorithme backtrack et cherche une solution dans le prochain sous-arbre. En cas d'exploration complète des deux sous-arbres sans trouver de solution, cela constitue une preuve d'insatisfaisabilité du problème racine. Comme cette procédure revient à effectuer une fouille en profondeur de l'arbre de recherche, elle est garantie d'explorer toutes les solutions.

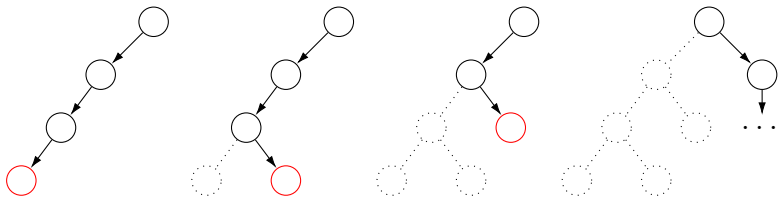


Figure 2.2 Exemple des premières étapes d'un parcours en profondeur.

L'inconvénient majeur de cette méthode est que si l'heuristique de branchement fait une mauvaise décision tôt dans la recherche, l'intégralité du sous-arbre doit être explorée avant de reconsidérer cette décision. Or c'est justement tôt dans la recherche (lorsque peu de variables sont fixées) que les heuristiques de branchement sont les plus susceptibles de prendre une mauvaise décision. Cette observation motive l'utilisation d'autres stratégies.

Le *Limited Discrepancy Search* [36] est une stratégie de backtracking qui consiste à visiter les nœuds dans l'arbre de recherche selon un ordre croissant de déviations (*discrepancies*) avec l'heuristique de branchement. La Figure 2.3 illustre les premières étapes du parcours effectué par le LDS : la recherche s'effectue en autorisant un nombre croissant de déviations par rapport à l'heuristique. Pour un nombre k de déviations sur une profondeur h, il y a $\binom{h+1}{k}$ chemins générés. La première feuille atteinte est celle qui ne dévie jamais de l'heuristique. En cas d'échec, les nœuds atteignables avec une seule déviation sont explorés, généralement en commençant par autoriser la déviation au sommet de l'arbre (car c'est en début de recherche que l'heuristique a plus de chances de s'être trompée) puis en l'autorisant de plus en plus tard. Viennent ensuite les nœuds atteignables avec deux déviations et ainsi de suite. Puisque tous les chemins finissent par être générés, elle est garantie d'explorer toutes les solutions. Lorsque l'heuristique fait de bonnes décisions de branchement, LDS explore généralement

moins de nœuds avant de trouver une solution. En effet, LDS est moins sensible aux erreurs de l'heuristique en début de recherche, et une fois proche des feuilles, elle exploite mieux l'heuristique.

Une autre possibilité consiste en l'utilisation de *restarts* et de randomisation. L'utilisation de randomisation et *restart* au sein de méthodes de backtracking remonte au moins aux travaux de Luby [37]. Il a été observé que redémarrer la recherche arborescente périodiquement permet de réduire l'impact des erreurs de guidage des heuristiques de recherche, ce qui a amené l'introduction des algorithmes de backtracking avec restart : cette méthode de recherche termine soit quand une solution a été trouvée ou que le nombre de backtracks depuis un échec dépasse un certain seuil. Plus formellement, une stratégie de restart $(t_1, t_2, \dots)$ est une suite infinie d'entiers naturels. L'algorithme de recherche est exécuté en autorisant jusqu'à t_1 backtracks. Si aucune solution n'est trouvée, l'algorithme est exécuté en autorisant t_2 backtracks, etc. Luby montre que la séquence

$$(k, k, 2k, k, k, 2k, 4k, k, k, 2k, k, k, 2k, 4k, 8k, k, \dots) \tag{2.1}$$

(où k est une constante) est une stratégie de restart universelle optimale en un certain sens, mais il existe d'autres types de stratégies explorées dans la littérature (seuil fixe, suite géométrique, ...). Il est empiriquement établi que ces stratégies sont particulièrement effectives pour les problèmes à queue longue, c'est-à-dire les problèmes pour lesquels la queue de la distribution de probabilité que la stratégie trouve une solution en moins de t étapes décroît polynomialement [38].

2.2 Apprentissage automatique et réseaux de neurones

Cette section présente une brève introduction à l'apprentissage machine (ML) et plus spécifiquement à l'apprentissage profond, c'est-à-dire à l'utilisation des réseaux de neurones comme approximateurs de fonctions pour résoudre des tâches de régression et de classification.

2.2.1 Réseaux de neurones denses

Les réseaux de neurones sont d'excellents approximateurs de fonctions, utilisés en apprentissage profond pour toutes sortes de tâches (régression, classification, modèles génératifs) dans de nombreuses applications (traitement naturel du langage, vision par ordinateur, etc.). Les réseaux de neurones les plus simples, appelés réseaux denses, sont constitués d'une série de couches de neurones, chaque couche l étant paramétrée par une matrice de poids W_l et d'un vecteur de biais b_l. Dans la suite, nous noterons θ l'ensemble des paramètres du réseau de

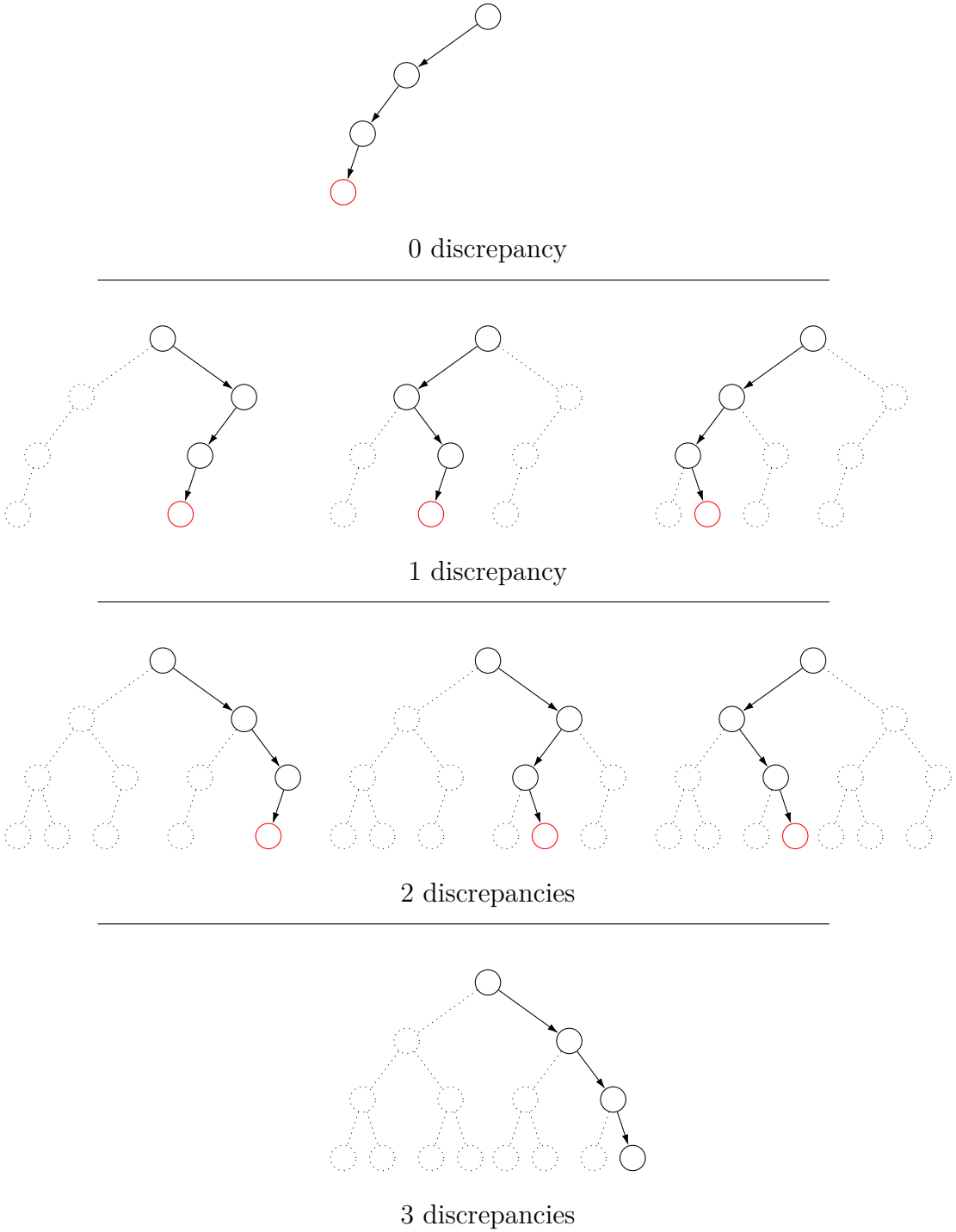


Figure 2.3 Exemple des premières étapes du LDS

neurones, c'est-à-dire l'ensemble des matrices W_l et des vecteurs b_l. Le réseau de neurones consiste en une fonction f_θ qui applique une série de transformations linéaires à un vecteur d'entrée x, suivie de l'application d'une fonction d'activation non-linéaire φ_l :

$$f_\theta(x) = \varphi_L \left(W_L \dots \varphi_1 \left(W_1 \cdot \varphi_0(W_0 x + b_0) + b_1 \right) + \dots + b_L \right) \tag{2.2}$$

Le théorème d'approximation universelle [39] garantie que les fonctions de cette forme peuvent (avec suffisamment de neurones) approximer avec une précision arbitraire n'importe quelle fonction suffisamment régulière.

Les paramètres de ces modèles peuvent être mis à jour efficacement grâce à un algorithme d'optimisation stochastique de type descente de gradient sur un objectif que l'on cherche à optimiser. Par exemple, pour un problème de régression, si l'on dispose d'un ensemble de paires entrées-sorties (x_i, y_i), on peut chercher les paramètres θ d'un réseau de neurones f_θ minimisant l'erreur quadratique

$$L(\theta) = \sum_i \left(y_i - f_\theta(x_i) \right)^2 . \tag{2.3}$$

Le gradient de cet objectif peut être calculé très efficacement par l'algorithme de rétropropagation, qui exploite la structure en couche du réseau

Des librairies d'auto-différentiation telles que Tensorflow ou Pytorch sont capables de calculer automatiquement le gradient $\nabla_\theta L$, et les paramètres du réseau peuvent alors être mis à jour dans la direction contraire afin de minimiser L. Dans sa forme la plus simple, la descente de gradient prend la forme de la règle suivante, dite descente de gradient à pas fixe :

$$\theta \leftarrow \theta - \alpha \nabla_\theta L(\theta) \tag{2.4}$$

où α est un paramètre appelé "pas d'apprentissage" contrôlant la force de la mise à jour. Cette procédure est répétée itérativement jusqu'à convergence.

2.2.2 Apprentissage machine sur les graphes

Bien que les réseaux de neurones simples tels que les réseaux denses vus précédemment soient adaptés à de nombreuses applications, ils ont l'inconvénient de ne pouvoir travailler qu'avec des vecteurs pour entrées et sorties. En pratique il est souvent nécessaire de travailler avec des structures plus générales que de simples vecteurs. Les graphes sont un exemple de structures générales pouvant être utilisées dans de nombreux domaines : chimie, physique, sociologie,

etc. L'apprentissage automatique sur les graphes est un domaine plus récent mais comprenant de nombreuses applications : inférence sur des systèmes physiques [40], chimiques [41], sur des graphes de connaissances [42] ou encore en optimisation combinatoire comme nous le verrons plus en détail en section 2.6 [43].

Le principe des réseaux de neurones à convolution de graphes, ou Graph Neural Network (GNN) [44, 45] est le suivant : soit $G = (V, E)$ un graphe que l'on supposera simple et non-orienté pour les besoins de la présentation. Les sommets sont potentiellement munis d'un vecteur de caractéristiques $\mathbf{f}_v$ pour chaque $v \in V$ et les arêtes sont potentiellement munies d'un vecteur de caractéristiques $\mathbf{h}_{u,v}$ pour chaque arête $(u, v) \in E$. Nous notons $\mathcal{N}(v)$ l'ensemble de sommets adjacents à v. Le GNN cherche à encoder chaque sommet $v \in V$ par un vecteur $\mu(v)$ élément d'un espace latent de dimension finie par une procédure de message-passing paramétrée. Il existe une grande variété de conceptions de GNN dans la littérature [45], nous exposons ci-dessous un schéma itératif unificateur proposé dans [46] appelé *Message Passing Neural Network*. Pour des raisons de clarté, nous illustrons le modèle abstrait avec une instantiation possible d'architecture de GNN introduite dans [47].

1. **Création des messages :** Chaque sommet $w \in \mathcal{N}(v)$ crée un message qui sera envoyé à v grâce à une fonction de messages M_θ pouvant prendre en compte les caractéristiques stockées sur l'arête (w, v) :

$$m_{t-1}(w \to v) = M_\theta \left(\mu_{t-1}(w), \mu_{t-1}(v), \mathbf{h}_{w,v}\right) \tag{2.5}$$

   Par exemple dans l'architecture proposée par [47], le message du sommet w au sommet v est un vecteur donné par

$$m_{t-1}(w \to v) = \theta_1 \mu_{t-1}(v) + \theta_2 \sigma(\theta_3 \mathbf{h}_{w,v}) \tag{2.6}$$

   où θ_1, θ_2 et θ_3 sont des matrices de paramètres devant être appris et σ est une fonction d'activation non-linéaire.

2. **Envoi et agrégation des messages :** Chaque sommet v envoie ses messages à ses voisins, et reçoit un ensemble de messages de ses voisins. Les messages reçus sont agrégés à l'aide d'un opérateur d'agrégation (ou de pooling) $\oplus$:

$$\Delta_t(v) = \bigoplus_{w \in \mathcal{N}(v)} m_{t-1}(w \to v). \tag{2.7}$$

   Cette opération d'agrégation est choisie de façon à être invariante par permutation de l'ordre des messages afin de conserver la propriété d'invariance par re-numérotation

des sommets. Une simple sommation des messages vérifie cette propriété. Dans [47], l'agrégation consiste effectivement en une simple sommation des messages incidents :

$$\Delta_t(v) = \sum_{w \in \mathcal{N}(v)} m_{t-1}(w \to v). \tag{2.8}$$

3. **Mise à jour de l'état latent :** L'état latent du sommet v est mis à jour à l'aide d'une fonction de mise à jour φ_θ pouvant également prendre en compte les caractéristiques du sommet en question :

$$\mu_t(v) = \varphi_\theta\Big(\mu_{t-1}(v), \Delta_t(v), \mathbf{f}_v\Big). \tag{2.9}$$

Dans [47], l'état latent du sommet v est mis à jour par la règle suivante :

$$\mu_t(v) = \sigma\left(\theta_4 \mathbf{f}_v + \Delta_t(v)\right) \tag{2.10}$$

où θ_4 est là encore une matrice de paramètres devant être appris.

Conjointement, les opérations données en exemple ci-dessus forment la règle de mise à jour donnée dans [47]

$$\mu_t(v) = \sigma\left(\theta_1 \sum_{w \in \mathcal{N}(v)} \mu_{t-1}(w) + \theta_2 \sum_{w \in \mathcal{N}(v)} \sigma(\theta_3 \mathbf{h}_{v,w}) + \theta_4 \mathbf{f}_v\right). \tag{2.11}$$

Le modèle abstrait a néanmoins l'avantage d'unifier un grand nombre d'architectures alternatives [46]. Les couches GCN [48] sont un autre type de convolution de graphes utilisant l'opération

$$H^{(l+1)} = \sigma\left(\tilde{D}^{-1/2}\tilde{A}\tilde{D}^{-1/2}H^{(l)}W^{(l)}\right)$$

où $H^{(l)}$ est une matrice représentant les caractéristiques de chaque sommet à chaque itération l, A est la matrice d'adjacence du graphe et

$$\tilde{A} = A + I_n$$
$$\tilde{D}_{ii} = \sum_j \tilde{A}_{ij}.$$

Cette procédure est typiquement répétée pendant un certain nombre d'itérations T afin que chaque sommet v puisse recevoir des messages de sommets qui ne sont pas situés dans son voisinage immédiat. Le nombre d'itérations contrôle le degré de localité de l'inférence : un nombre d'itérations plus élevé autorise des interactions entre sommets éloignés dans le graphe

G.

En fonction de la tâche à effectuer (régression, classification, génération...) les vecteurs latents $\mu_T(v)$ sont post-traités de façon adaptée. Les paramètres de l'architecture sont appris grâce à une méthode d'optimisation comme présenté dans la section précédente.

Le point clé est que cette méthode permet d'obtenir un algorithme d'inférence fonctionnant sur n'importe quel graphe, quelle qu'en soit la topologie, et respecte un certain nombre de symétries et invariances indispensables à l'étude de graphes (invariance par renumérotation des sommets notamment). Nous référons le lecteur à la revue de littérature [45] pour plus de détails sur les réseaux de neurones à convolution de graphes et leurs applications.

2.3 Apprentissage par renforcement

Cette section présente les principes et méthodes de base en apprentissage par renforcement. Pour plus de détails, le lecteur pourra consulter l'ouvrage de référence [49].

L'apprentissage par renforcement est une méthode de programmation dynamique approchée, rendant possible la résolution de problèmes très complexes en robotique [26], en commande optimale [27], en jeu vidéo [28] ou de plateau [30]. Le problème de la prise de décision séquentielle sous incertitude consiste à planifier une série de décisions dans un environnement incertain de façon à optimiser un objectif donné. Face à un problème aussi commun et général, il est peu surprenant d'apprendre que de nombreuses communautés se sont emparées de ce problème, menant à des développements plus ou moins indépendants. Mais c'est à partir des années 50, avec le formalisme des processus de décision markoviens (MDP) et de la programmation dynamique stochastique développée par R. Bellman [50] qu'une base théorique unifiée est proposée. Cette théorie a gagné en popularité dans des domaines variés : économie, télécommunication, commande optimale, recherche opérationnelle...

2.3.1 Programmation dynamique en horizon infini

Un processus de décision markovien (MDP) décrit un système dynamique stochastique contrôlé par un agent. Plus formellement, un MDP est donné par un 5-uplet $\langle \mathcal{S}, \mathcal{A}, \{\mathcal{A}(s) \mid s \in \mathcal{S}\}, P, r \rangle$ où

— $\mathcal{S}$ est un ensemble non-vide d'états (supposé fini dans ce chapitre),

— $\mathcal{A}$ est un ensemble non-vide d'actions (supposé fini dans ce chapitre),

— $\mathcal{A}(s)$ est un sous-ensemble non-vide de $\mathcal{A}$ contenant les actions admissibles depuis l'état s,

— P est un noyau de transition, avec $P(s'|s,a)$ donnant la probabilité de transition vers l'état s' si l'action a est effectuée depuis l'état s,

— r est une fonction réelle de récompense, avec $r(s,a)$ la récompense obtenue si l'action a est effectuée depuis l'état s.

Le processus étant stochastique, à chaque instant t il n'est pas possible de connaître avec certitude vers quel état s' l'environnement va se trouver après l'effet d'une action a. Pour définir un agent, il faut donc spécifier une *politique* prescrivant une action (ou plus généralement une distribution de probabilité sur l'espace d'actions) pour tout état $s \in \mathcal{S}$. Plus formellement, on définit une politique π par la probabilité de sélectionner l'action a depuis l'état s, notée $\pi(a|s)$. Nous noterons Π l'ensemble de toutes les politiques π.

La donnée d'un état initial $s \in \mathcal{S}$ et d'une politique $\pi \in \Pi$ induit une loi de probabilité sur l'espace des trajectoires de la forme $\tau = (s_0, a_0, s_1, a_1, \dots)$. Notons que cette trajectoire est aléatoire du fait des dynamiques (et potentiellement de la politique) stochastiques. Nous noterons $\mathbb{E}_s^\pi$ l'opérateur d'espérance contre cette loi de probabilité. L'objectif du problème de contrôle en horizon infini est de trouver une politique $\pi \in \Pi$ maximisant l'espérance de la somme actualisée des récompenses sur un horizon infini depuis chaque état initial [1] s

$$\max_{\pi \in \Pi} \mathbb{E}_s^\pi \left[\sum_{t=0}^{\infty} \gamma^t r(s_t, a_t) \right] \tag{2.12}$$

où $0 < \gamma \leq 1$ est un facteur d'actualisation.

La programmation dynamique se base sur une observation simple appelée *Principe d'Optimalité de Bellman* : en commençant depuis un état s, une trajectoire optimale peut se décomposer en deux parties, une première action optimale a, puis une sous-trajectoire optimale depuis l'état résultant de la transition $(s,a) \to s'$. Ainsi, si l'on note $V^*(s)$ la valeur du problème en commençant depuis l'état s, c'est-à-dire

$$V^*(s) = \max_{\pi \in \Pi} \mathbb{E}_s^\pi \left[\sum_{t=0}^{\infty} \gamma^t r(s_t, a_t) \right], \tag{2.13}$$

nous avons la relation suivante, dite *équation de Bellman*

$$V^*(s) = \max_{a \in \mathcal{A}(s)} \left[r(s,a) + \gamma \sum_{s' \in \mathcal{S}} P(s'|s,a) V^*(s') \right]. \tag{2.14}$$

1. L'existence d'une politique optimale qui atteint le maximum de récompense depuis **chaque** état est non triviale en général. Cette subtilité dépasse le cadre de ce mémoire, mais le lecteur peut admettre que sous les hypothèses énoncées, le problème est bien posé et admet bien une solution.

Dans cette équation, V^* apparaît comme le point fixe d'un certain opérateur, appelé *opérateur de Bellman*. Cela suggère une méthode de résolution itérative : en itérant cet opérateur sur une fonction initialisée aléatoirement, on peut montrer que l'on génère une suite de fonctions V_k qui converge vers V^*. Une politique optimale déterministe π^* peut être déduite en prenant

$$\pi^*(s) = \arg\max_{a \in \mathcal{A}(s)} \left[r(s,a) + \gamma \sum_{s' \in \mathcal{S}} P(s'|s,a) V^*(s') \right] \tag{2.15}$$

une fois V^* obtenue [2]. Cette méthode est connue sous le nom de *value iteration*. Nous attirons l'attention sur deux grands défis dans l'application de cette méthode :

1. le calcul de l'opérateur de Bellman et la résolution du problème de maximisation (2.14) nécessitent de parfaitement pouvoir modéliser le MDP, et en particulier les dynamiques de transitions $P(s'|s,a)$. En pratique, modéliser explicitement ces transitions est difficile, on parle de *malédiction de la modélisation*,
2. la manipulation et le stockage en mémoire des fonctions de valeurs V_k ainsi de que de la politique optimale π^* deviennent problématiques lorsque l'espace d'états et d'action sont très grands comme c'est le cas dans de nombreuses applications : on parle de *malédiction de la dimensionnalité*.

Face à ces défis, on utilise plutôt des méthodes de programmation dynamique approchées, dont l'apprentissage par renforcement.

En RL, pour contrer la malédiction de la modélisation, on ne suppose plus disposer des dynamiques du MDP sous la forme d'un modèle mathématique explicite, mais plutôt sous la forme implicite d'un modèle génératif (ou simulateur) avec lequel un agent va interagir afin d'échantillonner des trajectoires. Pour contrer la malédiction de la dimensionnalité, les politiques et fonctions de valeurs sont modélisées par des approximateurs de fonctions. Les réseaux de neurones sont un choix privilégié d'approximateurs universels de fonctions. La combinaison de l'apprentissage par renforcement et des réseaux de neurones profonds (*Deep Learning*) est nommé Apprentissage par renforcement profond ou *Deep Reinforcement Learning* (DRL).

2.3.2 Q-learning et Deep Q-Learning

En RL, nous ne disposons plus du modèle mathématique explicite du MDP que l'on cherche à résoudre. Il est donc nécessaire d'adapter la méthode de la *Value Iteration* présentée précédemment. En particulier, il n'est plus possible de représenter la fonction de valeur du

2. En pratique, V^* et π^* sont déterminées conjointement respectivement comme le maximum et le maximiseur du terme entre crochets.

problème par une fonction de valeur $V^*(s)$: en effet, même dans l'éventualité où l'on disposerait de la fonction V^*, on ne peut plus en déduire une politique à l'aide de (2.15) car les dynamiques $P(s'|s,a)$ sont inconnues. En revanche, il est possible de définir un Q-facteur (ou Q-fonction), défini par

$$Q^*(s,a) = \max_{\pi \in \Pi} \mathbb{E}_s^\pi \left[\sum_{t=0}^{\infty} \gamma^t r(s_t, a_t) \;\middle|\; s_0 = s, a_0 = a \right]. \tag{2.16}$$

La Q-fonction et la fonction de valeur V sont liées par les relations fonctionnelles suivantes :

$$V^*(s) = \max_{a \in \mathcal{A}(s)} Q^*(s,a) \tag{2.17}$$

$$Q^*(s,a) = r(s,a) + \gamma \sum_{s' \in \mathcal{S}} P(s'|s,a) V^*(s'). \tag{2.18}$$

En injectant ces relations dans l'équation de Bellman (2.14), on montre que Q^* vérifie elle-même une seconde équation de Bellman

$$Q^*(s,a) = r(s,a) + \gamma \sum_{s' \in \mathcal{S}} P(s'|s,a) \max_{a' \in \mathcal{A}(s')} Q^*(s',a'). \tag{2.19}$$

Cette équation est encore une équation de point fixe, mais contrairement à précédemment, si l'on dispose de Q^* il est possible d'en déduire une politique optimale sans connaître le noyau de transition P (ni même la fonction de récompense r) :

$$\pi^*(s) = \arg\max_{a \in \mathcal{A}(s)} Q^*(s,a). \tag{2.20}$$

Il reste à déterminer comment résoudre (2.19) sans connaître les dynamiques du MDP, seulement à partir d'interactions avec un simulateur. Cette question est résolue par le célèbre algorithme Q-learning [51]. L'idée est de mettre à jour une estimation de la Q-fonction pour chaque transition observée par l'agent au cours de son interaction avec l'environnement. Pour chaque transition (s,a,r,s') observée, l'estimation de la Q-fonction peut être mise à jour par la règle

$$Q_{k+1}(s,a) \leftarrow Q_k(s,a) + \alpha_k \left(r + \gamma \max_{a' \in \mathcal{A}(s')} Q_k(s',a') - Q_k(s,a) \right). \tag{2.21}$$

Cette règle de mise à jour peut être interprétée comme un algorithme stochastique de recherche de racine appliqué à l'équation (2.19). Plus rigoureusement, ce type d'algorithme itératif appartient à une classe de méthodes appelées *algorithmes d'approximation stochastique*, ou encore *algorithmes de Robbins et Monro*. Nous référons à [52, 53] pour plus de détails.

En DRL, pour contrer la malédiction de la dimensionnalité, la Q-fonction est approximée par un réseau de neurones Q_θ paramétré par un ensemble de paramètres θ. L'équation de Bellman (2.19) étant une condition nécessaire et suffisante d'optimalité, on peut chercher à minimiser l'écart entre le membre de gauche et le membre de droite pour construire une fonction objectif pour entrainer le modèle. Après avoir collecté une base de données d'interactions $\mathcal{D} = \{(s_t, a_t, r_t, s_{t+1})\}$, ce réseau est donc entraîné de façon à minimiser le résidu de Bellman

$$J(\theta) = \frac{1}{2}\mathbb{E}_{\mathcal{D}}\left[\left(y(s_t, a_t) - Q_\theta(s_t, a_t)\right)^2\right] \tag{2.22}$$

où $y(s_t, a_t) = r_t + \gamma \max_{a' \in \mathcal{A}(s_{t+1})} Q_\theta(s_{t+1}, a')$ est la cible visée par la régression [3]. L'intuition est que si ce résidu de Bellman est nul pour toute paire état-action (s_t, a_t), alors la condition de consistance temporelle (2.19) est validée et l'on obtient la Q-fonction optimale. La méthode décrite constitue la base de l'algorithme du *Deep-Q-Network* (ou DQN), introduit dans [54] et est l'un des premiers succès de l'application de l'apprentissage profond en RL.

Un élément important dans la conception de ces algorithmes est la stratégie utilisée pour collecter les données $\mathcal{D}$. La politique utilisée lors de l'interaction avec l'environnement doit équilibrer deux principes contradictoires : l'*exploration* et l'*exploitation*. L'exploration consiste à essayer de nouvelles actions de façon à assurer une couverture suffisamment large de l'espace d'états. Autrement, l'algorithme peut ne jamais découvrir une politique optimale car les données observées n'échantillonnent qu'un sous-espace sous-optimal d'états. L'exploitation consiste au contraire à ne pas explorer, et à ne se fier qu'à l'estimation courante de la Q-fonction pour guider la collecte de données vers un sous-ensemble de l'espace d'états que l'on pense être optimal (sur la base d'informations incomplètes). Si l'algorithme n'exploite pas assez, il risque de trop explorer et de ne jamais concentrer ses efforts vers une bonne solution.

Cette discussion sur le dilemme exploration/exploitation sera approfondie dans la section 2.4.

2.3.3 Méthode de gradient de politique

L'algorithme du Deep-Q-Network présenté dans la section précédente appartient à une famille d'algorithmes en RL appelée *Value-based methods*. Ces méthodes cherchent à approximer une fonction de valeur (typiquement une Q-fonction) et d'utiliser cette fonction pour en déduire une politique optimale avec la relation (2.20). Il existe une deuxième famille d'algorithmes, appelée *Policy-based methods*, basée sur l'apprentissage direct d'une politique optimale, sans modéliser de fonctions de valeurs. Ces méthodes partent d'une paramétrisation de l'espace des

3. Cette cible est considérée comme fixe par rapport aux paramètres du réseau de neurones, le gradient doit donc être bloqué à travers ce terme.

politiques Π, typiquement par un réseau de neurones π_θ prenant en entrée un état s, et donnant en sortie un score $\psi_\theta(s, a)$ pour chaque action $a \in \mathcal{A}$. Ces scores sont ensuite convertis en une distribution de probabilité sur $\mathcal{A}(s)$ via une activation softmax par exemple

$$\pi_\theta(a|s) = \frac{\exp \psi_\theta(s, a)}{\sum_{a' \in \mathcal{A}(s)} \exp \psi_\theta(s, a')}. \tag{2.23}$$

En pratique, les actions non-valides depuis l'état s sont filtrées par le mécanisme de masque suivant : le vecteur $\psi_\theta(s, a)$ est additionné avec un vecteur de masque m défini par $m_i = 0$ si la i-ème action est valide depuis s et $m_i = -\infty$ si la i-ème action est invalide. Comme $\lim_{x \to -\infty} \exp(x) = 0$, cela a pour effet d'ignorer les entrées du vecteur $\psi_\theta(s, a)$ qui ne correspondent pas à une action valide. Cette distribution discrète peut ensuite être échantillonnée pour obtenir une décision. Il est dès lors possible de directement optimiser la quantité d'intérêt : la somme des récompenses obtenues par l'agent en partant d'un état initial s,

$$G(\theta) = \mathbb{E}_s^{\pi_\theta} \left[\sum_{t=0}^{\infty} \gamma^t r(s_t, a_t) \right]. \tag{2.24}$$

Comme souvent en apprentissage automatique, cette optimisation se fait par le biais d'une méthode locale de gradient. Cependant, le calcul exact du gradient de G est intractable car il demande la connaissance de la distribution de probabilité sur l'espace des trajectoires du problème. Le *Policy Gradient Theorem* [55] est un résultat remarquable donnant un estimateur de $\nabla_\theta G$ qui ne nécessite pas la connaissance du modèle du MDP, et qui peut être calculé à partir de trajectoires collectées par la politique π_θ. Ce théorème donne la relation suivante :

$$\nabla_\theta G(\theta) = \mathbb{E}_s^{\pi_\theta} \left[\sum_{t=0}^{\infty} \left(\sum_{t'=0}^{\infty} \gamma^{t'} r(s_{t'}, a_{t'}) \right) \nabla_\theta \log \pi_\theta(a_t|s_t) \right], \tag{2.25}$$

l'expression entre crochet peut donc être utilisée comme un estimateur non-biaisé du gradient de l'objectif (2.24). Remarquons que l'estimateur ci-dessus doit être pris contre la distribution de trajectoires induites par la politique π_θ en commençant depuis l'état s, il s'agit d'une méthode dite *on-policy* puisqu'elle requiert que les données collectées soient issues de la politique en cours d'optimisation. Cette propriété contraste avec la méthode du Q-learning, dite *off-policy*, qui est capable d'apprendre la Q-fonction optimale Q^* à partir de données issues de n'importe quelle politique (en particulier une politique exploratrice).

2.4 Recherche arborescente de Monte Carlo

La recherche arborescente de Monte Carlo (MCTS) [56] est une méthode de recherche combinant l'exhaustivité d'une recherche arborescente avec les excellentes propriétés des méthodes de recherche randomisées pour l'exploration d'espaces de grandes dimensions. Le MCTS a reçu un intérêt considérable en intelligence artificielle, notamment dans la conception d'agents pour la résolution de jeux à deux joueurs [57–59]. Cette section propose une brève présentation du fonctionnement de cet algorithme et montre comment le MCTS peut être vu comme une méthode hybride (*value-based* et *policy-based*) pour la résolution de MDP.

2.4.1 Fonctionnement

L'idée de l'algorithme MCTS est de conserver la structure arborescente du MDP pour guider la recherche d'une politique optimale. Calculer exactement $Q^*(s,a)$ pour chaque couple $(s,a) \in \mathcal{S} \times \mathcal{A}$ est intractable pour la plupart des problèmes d'intérêt pratique. Plutôt que de chercher à calculer exactement $Q^*(s,a)$ en étudiant toutes les possibilités à partir d'un nœud, on approxime cette grandeur par un estimateur $Q(s,a)$ obtenu en échantillonnant au hasard un grand nombre de trajectoires dans l'arbre et en cumulant les statistiques obtenues pour chaque nœud. Autrement dit, on approxime le gain théorique par une moyenne des gains obtenus sur un grand nombre de trajectoires simulées aléatoirement. Pour simplifier la présentation, dans cette section nous supposerons que la fonction de récompense est nulle sauf au niveau des états terminaux. La Q-fonction d'une paire (s,a) est donc égale à la récompense terminale obtenue en moyenne depuis l'état s en effectuant l'action a.

Initialement, l'algorithme part de l'état racine s_0, et construit de manière asymétrique et incrémentale l'arbre de recherche, tout en maintenant une estimation $Q(s,a)$ pour chaque paire (s,a) rencontrée. On maintient également le nombre de fois que l'arête (s,a) a été sélectionnée pendant la recherche, que nous noterons $N(s,a)$. L'algorithme procède en quatre étapes, répétées autant de fois que nécessaire (typiquement on alloue un certain temps de calcul à l'algorithme et ce dernier effectue autant d'itérations que possible pendant le temps imparti) :

— *Selection* : Parmi les nœuds initiaux déjà rencontrés, l'algorithme utilise une politique spéciale nommée Tree Policy pour sélectionner une trajectoire jusqu'à tomber sur une *feuille* c'est-à-dire un nœud v dont les fils ne sont pas encore tous connus de l'algorithme i.e. l'algorithme ne stocke pas encore de statistiques sur tous les fils de v.

— *Expansion* : À partir de v, l'algorithme sélectionne un fils u qui n'est pas encore présent dans l'arbre. L'objectif est donc d'obtenir une estimation de la qualité de ce nouveau

nœud u pour améliorer l'estimation globale de la fonction Q.

— *Simulation* : L'algorithme utilise alors à partir de u une politique appelée DEFAULT POLICY pour descendre dans l'arbre jusqu'à atteindre un état terminal et obtenir une récompense terminale z.

— *Rétropropagation* : Les statistiques des ancêtres de u sont mises à jour :

$$Q(s,a) \leftarrow (1-\alpha)Q(s,a) + \alpha z \tag{2.26}$$

$$N(s,a) \leftarrow N(s,a) + 1 \tag{2.27}$$

où α est un paramètre contrôlant la force de la mise à jour.

Une fois le temps alloué dépassé, on choisit l'action qui maximise $N(s,a)$ c'est-à-dire l'action qui a le plus été choisie par le MCTS durant la recherche [4].

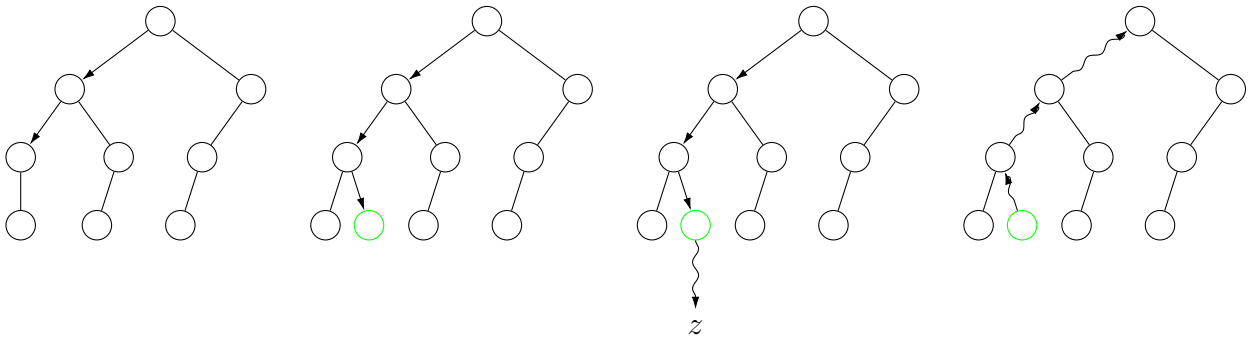


Figure 2.4 Principe de fonctionnement de la recherche arborescente de Monte Carlo

Il est possible de montrer que le MCTS est *correct* [5] sous certaines conditions. En particulier, si les politiques TREE POLICY et DEFAULT POLICY permettent d'explorer un nombre infini de fois tous les nœuds de l'arbre, alors le MCTS construit asymptotiquement l'arbre de recherche dans son intégralité.

La recherche s'est concentrée sur le choix des politiques TREE POLICY et DEFAULTPOLICY afin de guider la recherche de manière efficace, tout en conservant les propriétés de conver-

4. On peut également choisir l'action qui maximise $Q^*(s,a)$ mais ce choix est moins courant dans la littérature

5. On dit d'un algorithme qu'il est *correct* lorsqu'il effectue exactement ce pourquoi il a été conçu. Dans notre cas, la correction correspond au fait que l'algorithme renvoie, dans la limite d'un temps infini, une politique optimale

gence et de correction de l'algorithme. Dans la variante la plus simple, ces deux politiques sont des politiques aléatoires uniformes : l'algorithme choisit aléatoirement les nœuds pour former la trajectoire. Cependant pour obtenir de bonnes performances sur les problèmes d'une certaine complexité, les TREE POLICY et DEFAULTPOLICY doivent être plus sophistiquées. Dans la section suivante, nous présentons l'algorithme de l'UCT, qui propose une TREE POLICY mathématiquement justifiée et optimale en un certain sens.

2.4.2 Règle de l'UCT

L'idée de l'UCT [60] (Upper Confidence Tree) est la suivante : plus l'estimation d'un nœud $Q(s,a)$ est élevée, plus ce nœud est prometteur. La politique TREE POLICY devrait donc choisir plus souvent les nœuds ayant un estimateur élevé plutôt que de les choisir complètement aléatoirement. Cependant, si l'on choisit de procéder comme ceci, on risque de très peu sélectionner des nœuds qui semblent médiocres sur la base d'un petit nombre d'essais. Il est possible qu'un nœud soit en réalité très prometteur mais que le peu de simulations que la méthode ait effectuées ne permette pas de se rendre compte de ce potentiel.

De ce constat émerge une remarque importante : une bonne politique TREE POLICY doit bien équilibrer *l'exploitation* et *l'exploration* :

- *Exploitation* : Une politique exploitante est une politique qui utilise les statistiques déjà obtenues pour guider sa recherche vers des nœuds performants
- *Exploration* : Une politique exploratrice est une politique qui sélectionne en priorité des nœuds par lesquels elle est peu passée, ceci pour éviter l'écueil de la sous-optimalité d'une exploitation trop sévère

Ces deux propriétés sont contradictoires, mais pourtant les deux sont essentielles pour obtenir une sélection de nœud performante sans être sous-optimale. [61] montre qu'il existe une manière optimale de sélectionner un nœud de façon à équilibrer correctement l'exploitation et l'exploration : en partant du nœud v, la politique TREE POLICY doit sélectionner, à chaque niveau dans l'arbre, l'action a qui maximise la quantité suivante, appelée UCB (Upper Confidence Bound) :

$$UCB(s,a) = \underbrace{Q(s,a)}_{\text{exploitation}} + C \underbrace{\sqrt{\frac{\ln \sum_{a'} N(s,a')}{N(s,a)}}}_{\text{exploration}} \tag{2.28}$$

où C est une constante servant à équilibrer les deux termes et $N(s,a)$ est le nombre de fois que la paire (s,a) a été visitée lors des précédentes itérations. Le premier terme est un terme

d'exploitation : plus le noeud en question est estimé comme bon, plus il a de chance de se faire sélectionner. Mais comme vu précédemment, si un noeud n'a été visité qu'un petit nombre de fois, l'estimation de $Q(s,a)$ sera teintée d'une grande incertitude, et il est nécessaire de compenser ce biais. Le second terme est un terme d'exploration qui dépend du nombre de fois que la paire (s,a) a été explorée précédemment. On observe que ce terme tend vers $+\infty$ lorsque $N(s,a) \to 0$, ce qui signifie que si un noeud n'a pas été suffisamment exploré, il finira par être choisi même si sa Q-value $Q(s,a)$ est faible, parvenant ainsi à équilibrer exploration et exploitation.

2.5 AlphaZero

AlphaZero [30] est un système hybride complexe permettant de rendre les problèmes de très grande dimension à la portée du MCTS. En effet, malgré les bonnes propriétés du MCTS, son application à des problèmes de très grande taille reste problématique pour des raisons de scalabilité horizontale et verticale : les problèmes pratiques se caractérisent par des horizons temporels élevés (problème de scalabilité verticale) ainsi que par des espaces d'états et d'actions de grande dimension (problème de scalabilité horizontale). Ces problèmes combinés font que l'échantillonage aléatoire de la Default Policy donne une très mauvaise estimation de la valeur d'un noeud. Plus précisément, cette estimation reste non-biaisée mais souffre d'une très grande variance, ce qui rend l'algorithme inefficace en pratique car un trop grand nombre d'itérations est nécessaire pour contrecarrer la variance élevée de l'estimation.

2.5.1 Principes d'AlphaZero

La contribution principale d'AlphaZero est de combiner un algorithme de recherche arborescente de Monte Carlo avec des réseaux de neurones entraînés par Deep RL afin de résoudre ces problèmes de scalabilité. Les heuristiques *tree-policy* et default-policy sont remplacés par deux réseaux de neurones entraînés par Deep RL. Le premier réseau de neurones, appelé *value network* et noté $v_\theta(s)$ est chargé d'estimer la *value fonction* de l'état s. Le deuxième réseau, appelé *policy network* et noté $\vec{p}_\theta(s)$ est chargé de proposer un vecteur de probabilités sur l'ensemble des actions possibles depuis l'état s.

Lors de l'entraînement du réseau, le MCTS génère une base de données d'entraînement sous la forme $(s_t, \vec{\pi}_t, z_t)$ où s_t est un état ayant été visité lors de la recherche arborescente, $\vec{\pi}_t$ est une estimation de la politique optimale depuis l'état s (dont nous détaillerons l'obtention un peu plus tard) et z_t est la valeur de l'état terminal obtenue à l'issu de la simulation. Les deux réseaux de neurones sont alors entraînés conjointement de façon à minimiser la fonction

de perte

$$l(\theta) = \sum_t \left(v_\theta(s_t) - z_t\right)^2 - \vec{\pi}_t(s_t) \cdot \log(\vec{p}_\theta(s_t)). \tag{2.29}$$

Autrement dit, le *value network* est optimisé de façon à ce qu'il apprenne la valeur de l'état terminal obtenu depuis s_t en optimisant l'erreur quadratique tandis que le *policy network* est optimisé de façon à se rapprocher de la politique optimale en optimisant l'entropie croisée.

2.5.2 Le MCTS comme amélioration de politique

Étant donné un état s, le *policy network* propose une politique $p_\theta(s)$ sous la forme d'un vecteur de probabilités de sélection pour chaque action $a \in \mathcal{A}(s)$. Durant la phase d'entraînement, nous souhaitons améliorer ces estimations.

Plus précisément, pour chaque paire état-action (s, a) dans l'arbre du MCTS, on maintient

— $Q(s, a)$ une estimation de la Q-value de la paire (s, a)

— $N(s, a)$ le nombre de fois que l'action a a été prise depuis l'état s dans les itérations précédentes

— $P(s, \cdot) = \vec{p}_\theta(s)$ l'estimation initiale de la politique optimale depuis l'état s, donnée par le *policy-network*.

Avec ces données, la borne supérieure optimiste sur la Q-value proposée par AlphaZero est donnée par

$$U(s, a) = Q(s, a) + c_{\text{puct}} P(s, a) \frac{\sqrt{\sum_{a'} N(s, a')}}{1 + N(s, a)}. \tag{2.30}$$

La constante c_{puct} est une constante qui contrôle le degré d'exploration. La formule proposée dans l'article original d'AlphaZero est légèrement différente de celle présentée dans la Section 2.4.2. Il s'agit d'une variante de la formule de l'UCT appelée *Polynomial Upper Confidence Trees* (PUCT) [62] possédant des propriétés très similaires à la formule de l'UCT présentée précédemment.

Après avoir initialisé l'arbre du MCTS avec le noeud racine s, une itération d'entraînement procède comme suit :

1. On calcule l'action a^* qui maximise la borne supérieure $U(s, a)$
2. Si l'état s' obtenu après l'action a^* depuis s existe dans l'arbre du MCTS, on appelle récursivement la procédure de recherche depuis s'. S'il n'existe pas, on ajoute s' à l'arbre et on initialise $P(s', \cdot) = \vec{p}_\theta(s')$, $v(s') = v_\theta(s')$ avec les réseaux de neurones, $Q(s', a) = 0$ et $N(s', a) = 0$ pour chaque a. Plutôt que d'effectuer une simulation aléatoire, on propage la valeur $v(s')$ vers les ancêtres rencontrés lors de la simulation en cours et met

à jour les Q-value. En revanche si s' est un état terminal, on propage la récompense terminale.

Après un certain nombre de simulations, les compteurs $N(s, a)$ au sommet racine s convergent vers une meilleure approximation de la politique optimale que celle fournie par le policy-network car cette estimation bénéficie de la capacité de recherche arborescente du MCTS. La politique améliorée $\vec{\pi}(s)$ est donnée par les compteurs normalisés

$$\vec{\pi}(s) = \frac{N(s, \cdot)}{\sum_{a'} N(s, a')}. \tag{2.31}$$

Au moment de l'inférence, le MCTS est utilisé comme précédemment mais en figeant les paramètres des réseaux de neurones.

2.6 Optimisation combinatoire et apprentissage automatique

Dans la Section 2.1, nous avons présenté la notion d'optimisation combinatoire ainsi que quelques-unes de ses nombreuses applications. La programmation par contraintes est une des approches permettant la résolution de ce type de problèmes, mais comme vu précédemment, les seules techniques de la CP ne permettent pas toujours de résoudre certains problèmes. Cela a motivé l'introduction de nouvelles méthodes de résolution de tels problèmes.

Parmi les propositions plus récentes, l'utilisation de l'apprentissage automatique pour la résolution de problèmes combinatoires suscite un intérêt grandissant. En effet, l'apprentissage automatique offre la possibilité de se passer d'expertise humaine dans l'élaboration d'algorithmes ou d'heuristique dédiées. Sur la plupart des problèmes combinatoires, l'utilisation de méthodes spécialement conçues pour exploiter leur structure est critique à l'obtention de bonnes performances. La création de ces méthodes dédiées est à la fois longue, coûteuse, et nécessite une expertise humaine. La promesse de pouvoir exploiter la structure du problème automatiquement est donc très séduisante.

Il existe principalement deux approches permettant d'utiliser l'apprentissage automatique pour la résolution de problèmes combinatoires. La première peut être qualifiée de *méthode directe* : un modèle est directement utilisé pour prendre en entrée un certain problème, et donner une solution en sortie [43, 63]. Ces méthodes utilisent le plus souvent de l'apprentissage par renforcement comme paradigme. En effet, la plupart des problèmes combinatoires d'intérêt sont des problèmes NP-difficiles, et il est donc peu envisageable de créer une base d'apprentissage étiquetée car cela nécessiterait d'obtenir au préalable la solution d'un grand nombre de problèmes NP-difficiles. Le principal inconvénient de ce type de méthode est qu'il

est impossible d'obtenir une garantie d'optimalité sur la solution obtenue, ni même une quantification de l'écart d'optimalité. Un autre désavantage est que ces méthodes ne peuvent pas tirer parti d'un budget computationnel élevé : une fois la solution approchée calculée, elles ne peuvent pas l'améliorer de façon systématique.

Une autre approche, plus indirecte, consiste à utiliser l'apprentissage automatique comme sous-procédure d'un algorithme de résolution exacte [1, 3]. [1] propose d'utiliser une forme d'apprentissage supervisé appelée apprentissage par imitation, consistant à entraîner un modèle d'apprentissage à reproduire les décisions d'heuristiques efficaces mais très coûteuses. De cette façon, le modèle appris permet de guider efficacement la recherche tout en restant plus léger que les heuristiques ayant servies à l'entraîner. [3] propose d'utiliser un solveur CP, et délègue la sélection de valeur lors du branchement à un modèle d'apprentissage par renforcement profond. Ce travail est notamment motivé par le fait que la sélection de valeur en CP est non-triviale sur la plupart des problèmes. La possibilité d'apprendre automatiquement une heuristique de sélection de valeur ouvre donc de nouvelles opportunités. Par rapport aux méthodes directes, celle-ci conserve toutes les bonnes propriétés des solveurs exacts : certificat d'optimalité, quantification de l'écart d'optimalité, possibilité de tirer parti d'un budget computationnel plus long pour améliorer systématiquement la solution. L'une des limites de ces travaux est que l'heuristique de sélection de valeur n'est pas directement entraînée dans le solveur lors de la résolution mais fait intervenir une étape auxiliaire de régression à une solution.

[2] étend le travail de [3] en proposant une méthode hybride unifiée, intégrant une heuristique de sélection de valeur directement entraînée par le solveur lors de la résolution. Une évaluation empirique montre que cette méthode est capable d'apprendre des heuristiques de branchement plus performantes que certaines heuristiques dédiées simples.

[64] propose d'utiliser une recherche MCTS au sein d'un solveur CP pour la résolution du problème Job Shop, et montre empiriquement une amélioration significative des performances en terme de nombre de noeuds explorés par rapport à une recherche en profondeur. Ces travaux ne proposent cependant pas d'utiliser d'apprentissage pour guider la recherche. Malgré les propositions d'hybridations CP+MCTS et CP+RL vus précédemment, il n'existe pas à notre connaissance de méthodes hybrides CP+MCTS+RL. Dans le chapitre suivant, nous présentons quelques éléments techniques derrière l'hybridation proposée par [2] et nous appuyons sur ces précédents travaux pour motiver une nouvelle hybridation CP-RL-MCTS s'inspirant des récents succès d'AlphaZero appliqué aux problèmes markoviens.

CHAPITRE 3 APPRENTISSAGE AUTOMATIQUE DE SELECTION DE VALEUR EN CP

3.1 Modélisation du problème d'apprentissage

3.1.1 Définition du MDP

L'une des premières difficultés rencontrées dans l'utilisation d'un modèle RL pour la sélection de valeur en CP est celle de la modélisation du problème d'apprentissage. [2] propose de modéliser le problème d'apprentissage par un MDP où :

1. L'espace d'états est un triplet contenant l'exemplaire du problème combinatoire en cours de résolution, l'état courant de la recherche au sein du solveur et le modèle CP partiellement résolu.
2. L'espace d'actions est l'ensemble des valeurs du domaine de la variable choisie pour le branchement. Rappelons que ce travail se limite à la sélection de valeurs et que la sélection de variables est donc déléguée à une autre heuristique, indépendante du modèle RL.
3. La fonction de transition met à jour le modèle CP en fonction de l'action choisie. Cela comprend l'étape de propagation des contraintes assurée par le solveur CP.
4. La fonction de récompense est un élément délicat du modèle car elle conditionne le signal d'apprentissage reçu par l'agent, et donc le comportement qui sera appris. [2] propose par défaut une récompense de -1 pour chaque étape, ce qui pousse l'agent à explorer le moins de noeuds possibles avant de résoudre le problème. D'autres termes peuvent s'avérer utiles en fonction du problème et de l'objectif à optimiser.

Techniquement, l'agent RL doit donc pouvoir prendre en entrée une représentation d'état hautement non-euclidienne car un modèle CP partiellement résolu (autrement dit, un CSP ou COP) est un objet mathématique qui ne peut pas se représenter par un vecteur de dimension finie fixée. Cette difficulté est résolue par l'utilisation d'un GNN sur un modèle graphique judicieusement choisi.

3.1.2 Formulation d'un modèle graphique

L'agent RL doit pouvoir s'accommoder de n'importe quel exemplaire de n'importe quel modèle CP possible. Ne serait-ce que parce que le nombre de variables ou encore la topologie des contraintes en présence peut grandement varier entre chaque exemplaire, il est important

d'adopter une représentation d'état flexible et universelle. L'idée consiste à représenter le modèle CP par un graphe et à utiliser un GNN sur ce graphe pour autoriser l'inférence sur n'importe quelle topologie de modèle. Il existe une représentation graphique appelée *graphe de contraintes* également utilisée en programmation par contraintes permettant d'encoder un problème combinatoire dans une structure graphique. Il consiste en un graphe non-orienté $G(V, E)$ où chaque variable $x \in X$ du modèle CP est associée à un noeud-variable $s_x \in V$, chaque contrainte $c \in C$ est associée à un noeud-contrainte $s_c \in V$ et deux noeuds s_x et s_c sont reliés par une arête si et seulement si la variable x apparaît dans la portée de la contrainte c. Ce graphe forme donc un graphe biparti qui encode la structure du problème combinatoire.

Le solveur SeaPearl adopte une représentation graphique similaire, adaptée en ajoutant des noeuds de type *valeur* : une valeur v est associée à un noeud-valeur s_v ; un noeud-valeur s_v et un noeud-variable s_x sont reliés par une arête si et seulement si la valeur v apparaît dans le domaine $D(x)$. Chaque noeud dans ce graphe triparti peut être équipé d'un vecteur de caractéristiques donnant des informations auxiliaires sur ladite contrainte, variable ou valeur (type de contrainte, taille du domaine, ...). Ces caractéristiques sont alors utilisées au sein de la procédure de *message-passing* du GNN, permettant ainsi à l'agent de RL d'exploiter ces informations pour guider le branchement.

3.2 SeaPearl.jl : un solveur hybride

3.2.1 Le langage Julia

La création d'une méthode hybride entre programmation par contraintes et apprentissage par renforcement pose également quelques défis en termes d'implémentation. Les solveurs de programmation par contraintes sont des outils très optimisés, traditionnellement écrits dans des langages orienté-objet. Citons le solveur Gecode [65] écrit en C++, Choco [66] écrit en Java ou encore OscaR [67] écrit en Scala.

À l'inverse, la communauté de l'apprentissage automatique s'est détournée de ces langages et leur préfère des langages plus productifs et hautement interactifs, Python étant le plus populaire. La pratique de l'apprentissage profond nécessite l'utilisation de librairies d'auto-différentiation comme Tensorflow ou Pytorch, pouvant tirer parti de l'accélération matérielle fournie par les GPU ainsi que du calcul distribué. Mais Python est un langage dynamique et interprété, rendant difficile son utilisation pour des applications nécessitant une haute performance. Les routines gourmandes en ressources sont généralement écrites en C et compilées en langage machine puis appelées depuis l'interpréteur, ce qui rend complexe la création d'outils

haute-performance. Le développement d'une méthode hybride à base de RL pour la recherche en CP nécessite de réconcilier ces différents besoins.

`Julia` [68] est un langage de programmation haut-niveau, polyvalent et pensé pour le calcul scientifique. Ce langage combine productivité et performance, avec une syntaxe proche du `MATLAB` permettant un prototypage rapide et productif, tandis que la compilation *just-in-time* (JIT) permet d'obtenir des performances comparables à un code C compilé. Ces propriétés font de `Julia` un excellent langage pour la conception d'un solveur de programmation par contraintes. Mais `Julia` est également un langage moderne, ayant un large écosystème de librairies dédiés à l'apprentissage automatique, à l'apprentissage par renforcement, à la programmation probabiliste, etc. Cet atout par rapport aux langages plus traditionnels rend ce langage particulièrement adapté pour la création de méthodes hybrides comme celle présentée dans ce mémoire. Finalement, `Julia` est libre de droit, mis à disposition sous une licence permissive, rendant le langage très accessible pour la communauté scientifique, à la fois académique et industrielle.

3.2.2 Principes de l'hybridation

SeaPearl utilise un algorithme de recherche de type recherche en profondeur (DFS) en alternant, selon le schéma classique en CP, propagation des contraintes, sélection de variable, sélection de valeur, puis en backtrackant le cas échéant. La différence notable avec un solveur CP classique est le fait que l'heuristique de sélection de valeur est apprise automatiquement par apprentissage par renforcement.

Le protocole d'apprentissage est le suivant :

1. L'utilisateur définit une distribution d'exemplaires qui sera utilisée pour entraîner l'agent. Idéalement, cette distribution est représentative du type de problème qui sera résolu après entraînement.
2. Le solveur résout successivement les exemplaires de la base d'entraînement. Juste après initialisation, l'agent de RL n'a encore rien appris et les branchements seront donc proches de branchements aléatoires. Au fur et à mesure de l'apprentissage, l'agent affine sa stratégie et explore de mieux en mieux l'espace des solutions.
3. Une fois la performance satisfaisante, le modèle peut être gelé et utilisé en mode inférence pour résoudre les problèmes de test.

Dans l'article original [2], l'agent de RL a été entraîné avec un algorithme de Deep-Q-learning, mais d'autres approches peuvent être envisagées. Empiriquement, cette méthode a montré des

résultats encourageants sur les problèmes du coloriage de graphes et du Traveling Salesman Problem with Time Windows (TSPTW).

Néanmoins, ce travail se limite à l'intégration au sein d'une méthode recherche de type DFS. Si la recherche en profondeur est un choix naturel et répandu en CP, elle n'est pas dénuée d'inconvénients. Rappelons que cette méthode est particulièrement sensible aux erreurs de l'heuristique au sommet de l'arbre de recherche. Or c'est précisément au sommet de l'arbre, lorsque le problème est peu contraint, qu'une heuristique a le plus de chance de faire une erreur de sélection de valeur. Ceci explique que la recherche en profondeur peut souffrir de pathologies sur des problèmes complexes comme les problèmes à queue longues (voir Section 2.1.3). De plus, l'utilisation d'apprentissage par renforcement en conjonction avec des méthodes de recherche arborescente ouvre la voie à des technique plus avancées comme celle présentée précédemment avec AlphaZero. AlphaZero propose une intégration beaucoup plus étroite de la boucle d'apprentissage RL et de la recherche arborescente sous la forme d'un MCTS. Dans la section suivante, nous explorons la possibilité de s'inspirer du système proposé par AlphaZero pour créer une méthode hybride de résolution de problème combinatoire selon le paradigme de la programmation par contraintes. Nous présentons une proposition d'extension des travaux de [2] en intégrant au sein du solveur SeaPearl une méthode de recherche combinant un Monte Carlo Tree Search avec deux heuristiques apprises par RL pour guider la recherche en programmation par contraintes. La section suivante décrit plus en détail les principes de cette hybridation.

3.3 Combiner RL et MCTS en recherche CP

Remarquons que la modélisation du problème d'apprentissage proposée par SeaPearl est largement indépendante de la méthode de recherche proposée. Il semble donc judicieux de conserver le même MDP servant à formuler le problème d'apprentissage, mais de proposer une architecture alternative. Plutôt que d'introduire un unique modèle utilisé comme heuristique de sélection de valeur, nous introduisons deux modèles :

— Le *value network* $v_\theta(s)$ est chargé d'estimer la *value function* de l'état s, c'est-à-dire la somme des récompenses obtenue par une politique optimale depuis l'état s. Dans notre contexte, la définition de la *value function* dépend bien sûr de la fonction de récompense spécifiée par l'utilisateur. Pour la suite, nous prendrons l'exemple de la fonction de récompense suivante :

$$r(s,a) = \begin{cases} -r_{\max} & \text{si l'action } a \text{ mène à un échec} \\ -1 & \text{sinon} \end{cases} \tag{3.1}$$

avec $r_{\max}$ choisi de façon à être typiquement grand par rapport au nombre de variables du problème. Cette fonction de récompense pousse l'agent à résoudre le problème le plus rapidement possible en évitant le plus possible de faire échouer la recherche. Avec cette fonction de récompense, et dans l'hypothèse ou le *value network* a une précision de 100%, une simple inférence depuis l'état s permet d'anticiper si l'état s débouche sur une solution ou sur un échec.

— Le *policy network* $p_\theta(s)$ est chargé de proposer un vecteur de probabilité sur l'ensemble des actions disponibles depuis l'état s tel que ce vecteur approxime une politique d'exploration-exploitation optimale depuis l'état s.

Détaillons à présent le fonctionnement de l'algorithme lors de la phase d'apprentissage. Les réseaux de neurones v_θ et p_θ sont initialisés aléatoirement. Une structure de données dédiée est allouée afin de stocker les statistiques suivantes pour chaque paire (s, a) :

1. $Q(s, a)$: une estimation de la Q-value de la paire (s, a)
2. $N(s, a)$: le nombre de fois que l'action a a été prise depuis l'état s dans les itérations précédentes
3. $P(s, \cdot)$: l'estimation de la politique exploration-exploitation optimale depuis l'état s, donnée par le *policy network*. Cette structure sert principalement de cache afin de ne pas avoir à utiliser le GNN plusieurs fois sur la même entrée, car cette inférence est coûteuse.

Ces structures de données sont choisies de façon à ce que $Q(s, a) = 0$ et $N(s, a) = 0$ lorsque la paire (s, a) n'existe pas dans la structure de données (voir Section 3.4 pour plus de détails sur l'implémentation), de sorte que la valeur nulle soit toujours implicite.

Notons s_0 l'état initial, comprenant le modèle CP de l'exemplaire à résoudre. L'état s_0 est ajouté à l'arbre du MCTS. Pour chaque action a disponible depuis l'état s_0, le policy network est utilisé pour estimer la politique $P(s_0, a) = p_\theta(s_0, a)$. Cette quantité est ensuite utilisée pour calculer la borne supérieure optimiste sur la Q-value

$$U(s_0, a) = Q(s_0, a) + c_{\text{puct}} P(s_0, a) \frac{\sqrt{\sum_{a'} N(s_0, a')}}{1 + N(s_0, a)}. \tag{3.2}$$

La méthode choisit alors l'action $a^* = \arg\max_a U(s, a)$. L'état successeur est donné par une fonction de transition $s' = T(s, a)$, qui comprend l'étape propagation des contraintes jusqu'à obtention du point fixe.

Si l'état s' existe dans l'arbre du MCTS, alors la procédure de recherche est appelée récursivement depuis l'état s'. S'il n'existe pas et qu'il n'est pas un sommet terminal, on rajoute

s' à l'arbre du MCTS et initialise $P(s', \cdot) = p_\theta(s', \cdot)$. Les Q-values des ancêtres de s' doivent alors être mises à jour de façon à prendre en compte la *value function* du descendant nouvellement ajouté. En effet, rappelons que les Q-values doivent satisfaire une condition de consistance temporelle le long d'une trajectoire. En notant $(s_0, a_0, s_1, a_1, \ldots, s_N)$ la trajectoire d'états-actions visités par le MCTS, les Q-values sont mises à jour du bas vers le haut par la relation

$$Q(s_t, a_t) \leftarrow (1-\alpha)Q(s_t, a_t) + \alpha\Big[r(s_t, a_t) + v_\theta(s_{t+1})\Big]. \tag{3.3}$$

où α est un pas d'apprentissage, permettant de quantifier l'agressivité de la mise à jour : plus α est proche de 1, plus l'agent a tendance à oublier les précédentes estimations de la Q-function. Si s_N est un état terminal, on propage directement la récompense terminale :

$$Q(s_{N-1}, a_{N-1}) \leftarrow (1-\alpha)Q(s_{N-1}, a_{N-1}) + \alpha r(s_{N-1}, a_{N-1}). \tag{3.4}$$

Après plusieurs itérations, la politique optimale d'exploration-exploitation depuis un état s est mieux approximée par les compteurs normalisés

$$\vec{\pi}(s) = \frac{N(s, \cdot)}{\sum_{a'} N(s, a')}. \tag{3.5}$$

car le MCTS aura lui-même équilibré exploration et exploitation (voir Section 2.4.2).

Cette procédure est répétée jusqu'à la résolution du problème ou jusqu'à ce qu'un nombre maximal d'itérations du MCTS $T_{\max}$ soit dépassé (dans la suite nous prendrons $T_{\max} = \infty$) avant de passer à la résolution de l'exemplaire suivant. Ceci est répété pour résoudre N_e exemplaires de la base d'apprentissage, ce qui forme un *épisode*.

À la fin d'un épisode, on obtient alors une base d'apprentissage composée de plusieurs trajectoires états-récompense de la forme $(s_0, r_0, s_1, r_1, \ldots, s_T, r_T)$ pour chacun des N_e exemplaires (autant qu'il y a eu d'itérations lors de la recherche MCTS). Les deux réseaux de neurones sont alors entraînés de façon à minimiser la fonction de perte

$$l(\theta) = \sum_{t=0}^{T} \Big[v_\theta(s_t) - y_t\Big]^2 - \pi(s_t) \cdot \log p_\theta(s_t) \tag{3.6}$$

avec $y_t = r_t + v_\theta(s_{t+1})$, $y_T = r_T$ et $\pi(s_t)$ est donnée par (3.5) [1]. Notons que cet objectif est différent de l'objectif original d'AlphaZero car nous devons prendre en compte le fait qu'il y a des récompenses intermédiaires, là où AlphaZero se limite au cas où seule la récompense

1. Exactement comme pour le Deep-Q-learning, cette cible est considérée comme fixe par rapport aux paramètres du réseau, on ne propage donc pas de gradient à travers y_t.

terminale est non-nulle. L'algorithme 1 présente la recherche MCTS adaptée à la recherche CP. Ce processus, représenté en Figure 3.1, est répété itérativement jusqu'à convergence ou épuisement du budget computationnel d'entraînement.

La procédure d'entraînement, représentée dans l'algorithme 2, suit la même principe que celui présenté dans la Section 2.5.1.

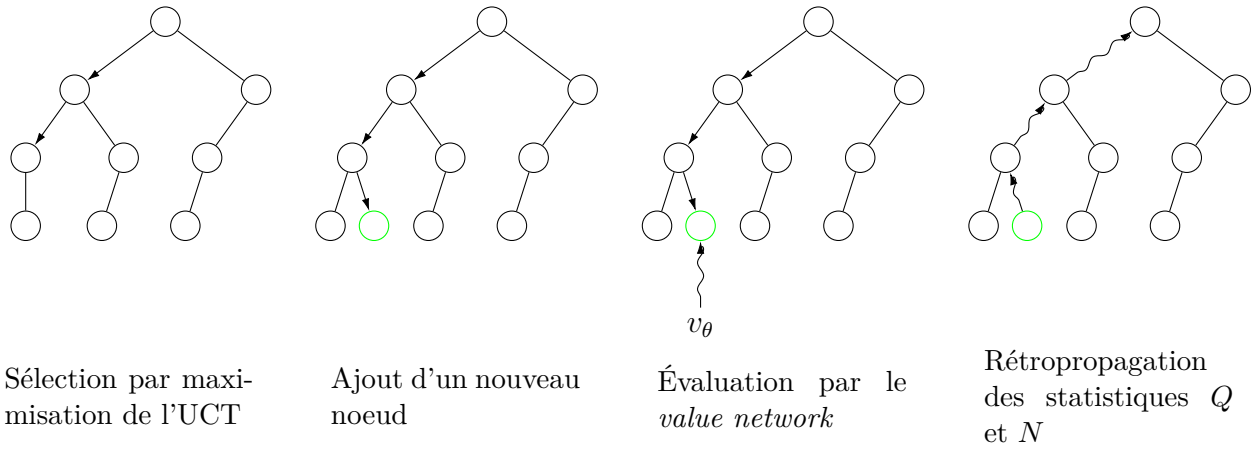


Figure 3.1 Illustration d'une itération du MCTS dans la méthode hybride MCTS+RL

3.4 Implémentation

Dans cette section, nous présentons quelques choix d'implémentation relatifs à l'algorithme décrit précédemment.

La Figure 3.2 présente une schématisation de l'organisation des différentes entités. Une fois la distribution d'entraînement définie, le solveur échantillone des exemplaires de cette base et appelle itérativement l'heuristique pour résoudre ces problèmes. L'agent d'apprentissage par renforcement ajoute les données d'interaction avec le solveur dans une structure de donnée appelée *replay buffer* en apprentissage par renforcement. Ces données sont utilisées lors de l'apprentissage du modèle à la fin de chaque épisode d'interaction avec le solveur.

Contrairement à une recherche DFS simple, la méthode proposée requiert le maintien de structures de données additionnelles pour stocker $Q(s, a)$ et $N(s, a)$ pour chaque paire état-action (s, a). Pour implémenter ces structures de données, nous utilisons une structure d'accumulateur en Julia, c'est-à-dire une structure maintenant un nombre accumulé pour chaque clé. Cette structure renvoie 0 lorsque la clé n'existe pas dans la structure de données. Cela a deux avantages par rapport à une implantation naïve avec des tableaux :

Algorithme 1 Monte Carlo Tree Search

Entrées : $\mathcal{T}$ est l'arbre du MCTS, MCTS(s) renvoie une estimation de la *value function* de l'état s

fonction MCTS(s)
  si s est terminal **alors**
    renvoyer récompense terminale z
  fin si
  si $s \notin \mathcal{T}$ **alors** ▷ Ajout de l'état s dans l'arbre
    $\mathcal{T} \leftarrow \mathcal{T} \cup \{s\}$
    $N(s, \cdot) \leftarrow 0$
    $Q(s, \cdot) \leftarrow 0$
    $P(s, \cdot) \leftarrow p_\theta(s)$
    renvoyer $v_\theta(s)$
  fin si
  $a^* \leftarrow \arg\max_a U(s, a)$
  $s' \leftarrow T(s, a^*)$ ▷ Calcul de l'état successeur
  $v \leftarrow \text{MCTS}(s')$
  $Q(s, a^*) \leftarrow (1 - \alpha)Q(s, a^*) + \alpha\Big[r(s, a^*) + v\Big]$
  $N(s, a^*) \leftarrow N(s, a^*) + 1$
  renvoyer $r(s, a^*) + v$
fin fonction

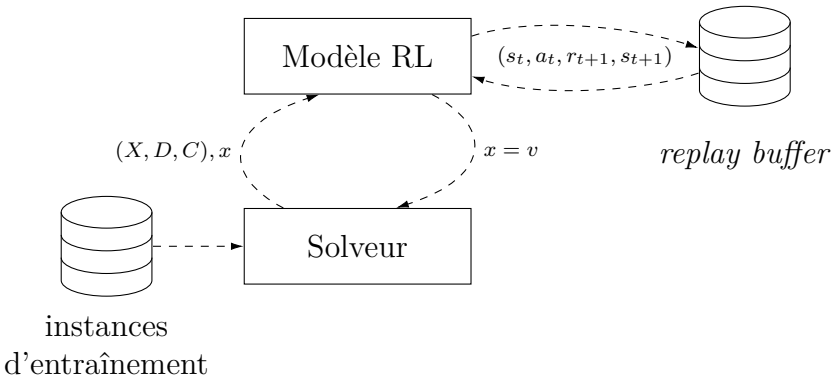


Figure 3.2 Architecture de la méthode hybride

Algorithme 2 Phase d'apprentissage

Entrées : $\mathcal{P}$ est une base d'apprentissage de problèmes combinatoires

fonction TRAIN
  tant que critère d'arrêt non vérifié **faire**

    # Interaction avec le solveur pendant 1 épisode
    pour itération de 1 à N_e **faire**
      Prélever aléatoirement un problème s_0 de $\mathcal{P}$
      $T \leftarrow 0$
      tant que s_0 n'est pas résolu et que $T < T_{\max}$ **faire**
        MCTS(s_0)
        $T \leftarrow T + 1$
        Ajouter la trajectoire $(s_0, r_0, \ldots, s_T, r_T)$ au buffer
      fin tant que
    fin pour

    # Entraînement des réseaux de neurones à la fin de chaque épisode
    pour chaque trajectoire $(s_0, r_0, \ldots, s_T, r_T)$ dans le buffer **faire**
      $\theta \leftarrow \theta - \beta \nabla_\theta l(\theta)$
    fin pour
  fin tant que
fin fonction

1. Cela évite de devoir réallouer de la mémoire lorsque le tableau n'est plus assez grand pour stocker toutes les paires états-actions rencontrées. En effet, du fait que l'arbre du MCTS est construit de manière incrémentale, on ne sait pas *a priori* quelle en sera la taille, on ne peut donc pas pré-allouer un tableau avec une taille suffisante.
2. Le 0 est implicite, et n'est donc pas représenté en mémoire, ce qui évite des initialisations inutiles au moment où un noeud est ajouté au MCTS.

Cette idée ne peut pas être appliquée au cache $P(s, a)$ car on ne souhaite pas qu'il y ait une valeur par défaut implicite : cette structure sert uniquement à mettre en cache les prédictions du policy-network sur les états ayant déjà été rencontrés pendant la recherche. À la place, nous adoptons une structure de type tableau, avec un schéma de réallocation géométrique : on pré-alloue un tableau de taille T_0, et à chaque fois que le tableau devient trop petit, on réalloue un tableau de taille $T_{t+1} = 2T_t$ de façon à réduire exponentiellement le nombre de réallocations nécessaires et à garantir un accès à coût amorti constant.

Une seconde amélioration réside dans le choix d'une structure de données adaptée à la maximisation de $U(s, a)$ donnée par (3.2). En effet, ce problème de maximisation doit être réalisé à chaque fois que l'on navigue dans l'arbre du MCTS ; il est impératif de disposer d'une implantation efficace. Nous choisissons l'usage d'une file de priorité `Priority`$[s]$ pour chaque état s, contenant chaque action a disponible depuis s comme clé, munie d'une priorité égale à $U(s, a)$. Ces priorités sont mises à jour à chaque étape de rétropropagation car les Q-values $Q(s, a)$ et les compteurs $N(s, a)$ sont modifiés, ce qui nécessite de changer la borne $U(s, a)$. Comme cette modification ne concerne que des paires états-actions ayant été visitées lors de la précédente itération du MCTS, cela n'impacte qu'un nombre réduit de files de priorité, et l'essentiel des structures reste intact. Le gain est en revanche très intéressant, car la maximisation de $U(s, a)$ à chaque appel récursif se fait en temps constant.

CHAPITRE 4 EVALUATION EMPIRIQUE

Dans ce chapitre, nous cherchons à évaluer expérimentalement les performances et les propriétés de l'algorithme MCTS présenté dans le chapitre précédent. Ces évaluations ont notamment pour objectif de comparer cette nouvelle méthode de recherche avec l'algorithme initial de SeaPearl présenté dans [2] et d'en isoler quelques propriétés intéressantes pour le praticien.

4.1 Méthodologie

4.1.1 Questions de recherche

Nous commençons par présenter les objectifs de nos expériences. Nous cherchons à répondre à deux questions de recherche relatives à l'algorithme présenté :

1. Est-ce que l'hybridation RL+MCTS présentée précédemment permet d'améliorer les performances de résolution par rapport à la méthode initiale de SeaPearl, basée sur une heuristique de sélection de valeur et d'une recherche en profondeur ? Ces performances peuvent être mesurées en terme de nombre de noeuds explorés dans les arbres de recherche respectifs de chaque méthode avant d'atteindre une solution, ou encore par le temps de résolution mis avant d'aboutir à une solution. Il est également intéressant de mettre en comparaison ces méthodes récentes utilisant de l'apprentissage par renforcement avec des heuristiques de sélection de valeur plus simples, qui serviront de référence.
2. Quelle est la sensibilité de ces méthodes basées sur de l'apprentissage par rapport à la distribution d'entraînement ? Autrement dit, peut-on évaluer et comparer les capacité de *transfer learning* des deux méthodes proposées ?

Dans toutes les expériences qui suivent, le choix de variable est fixé en utilisant une heuristique `min-domain` choisissant la variable avec le domaine le plus petit. Les expériences ont été effectuées sur un noeud de la grappe Cedar fourni par Compute Canada, ayant les caractéristiques suivantes :

— CPU : 2 x Intel E5-2650 v4 Broadwell @ 2.2GHz

— Mémoire : 125Go

— GPU : 4x NVIDIA P100 Pascal (12G HBM2 memory)

4.1.2 Problèmes d'application

Nous proposons d'évaluer les différents algorithmes sur la base de deux problèmes bien connus dans la littérature : le coloriage de graphe et le problème du sac à dos multidimensionnel.

Coloriage de graphe

Étant donné un graphe $G = (V, E)$ et un entier k, un k-coloriage de G est une fonction de l'ensemble des sommets V vers l'ensemble des couleurs numérotées de 1 à k telle que deux sommets adjacents soient coloriés avec deux couleurs différentes. Le problème du coloriage de graphe consiste en la détermination de l'entier minimal k tel qu'il existe un k-coloriage de G. Un tel entier k est alors appelé le *nombre chromatique de* G. Ce problème est NP-difficile, et intervient dans de nombreuses applications en planification par exemple.

Ce problème se prête à de multiples modélisations, et peut être résolu à l'aide de différents paradigmes. Nous utilisons le modèle CP suivant : pour un graphe $G = (V, E)$ à n sommets numéroté de 1 à n, chaque sommet $i = 1, 2, \ldots, n$ se voit attribué une variable v_i ayant pour domaine $\{1, 2, \ldots, n\}$ représentant sa couleur dans un coloriage minimal. Ce domaine vient d'une borne supérieure triviale que l'on peut donner sur le nombre chromatique, donnée par le nombre de sommets dans le graphe (cette borne est atteinte pour le graphe complet d'ordre n). Comme il s'agit d'un problème d'optimisation, on adopte la procédure conventionnelle de transformation d'un COP en une série de CSP : on introduit une variable auxiliaire z ayant pour domaine $\{1, 2, \ldots, n\}$ donnant le nombre chromatique de G. La variable z est utilisée au sein de la recherche en conjonction avec une borne supérieure progressivement resserrée à mesure qu'une solution au problème de satisfaction est trouvée, implémentant ainsi une recherche de type Branch-and-Bound. Le modèle formel est donné par

$$\min z \tag{4.1a}$$

$$v_i \neq v_j \quad \forall (v_i, v_j) \in E \tag{4.1b}$$

$$v_i \leq z \quad \forall v_i \in V \tag{4.1c}$$

Les exemplaires de ce problème sont générés à l'aide de deux paramètres : le nombre de sommets du graphe n, et la densité du graphe ρ donnée par

$$\rho = \frac{2|E|}{n(n-1)}. \tag{4.2}$$

La densité 0 correspond au graphe où tous les sommets sont isolés, et la densité 1 au graphe

complet. Par le lemme des poignées de mains, en notant $\bar{d}$ le degré moyen des sommets de G, on obtient

$$n\bar{d} = 2|E| = n(n-1)\rho \tag{4.3}$$

ce qui nous donne

$$\bar{d} = (n-1)\rho. \tag{4.4}$$

Le degré de chaque sommet est alors choisi aléatoirement en suivant une loi géométrique [1] de paramètre $(n-1)^{-1}\rho^{-1}$ et les arêtes sont créées aléatoirement entre chaque sommet de façon à respecter les degrés.

Problème du sac à dos multiples

Le 0-1 Multiple Knapsack Problem, ou problème du sac à dos multiples est une variation du problème du sac à dos : étant donné un ensemble de n objets et un ensemble de m sac à dos ($m \leq n$), on introduit

- v_i : la valeur de l'objet i
- w_i : le poids de l'objet i (supposé positif)
- c_j : la capacité du sac j (supposée positive).

L'objectif est de sélectionner m sous-ensembles disjoints d'objets de telle sorte que la somme des valeurs de tous les objets sélectionnés soit maximale et que chaque sous-ensemble puisse être assigné à un sac différent en respectant sa capacité. Plus formellement, nous adoptons le modèle suivant : pour chaque objet i et chaque sac j, nous introduisons la variable binaire x_{ij} stipulant si l'objet i est assigné au sac j. Le modèle est alors donné par

$$\max z = \sum_{j=1}^{m}\sum_{i=1}^{n} v_i x_{ij} \tag{4.5a}$$

$$\text{avec } \sum_{i=1}^{n} w_i x_{ij} \leq c_j, \quad \forall j \in \{1,\ldots,m\} \tag{4.5b}$$

$$\sum_{j=1}^{m} x_{ij} \leq 1, \quad \forall i \in \{1,\ldots,n\}\,. \tag{4.5c}$$

Ce modèle est directement traduit en un modèle CP grâce à l'utilisation de contraintes linéaires.

Les instances de ce problème sont générées à l'aide de 3 paramètres : le nombre d'objets n, le nombre de sacs m, le poids maximal d'un objet $w_{\max}$. Les poids sont uniformément

1. On rappelle que l'espérance d'une loi géométrique de paramètre p est $1/p$.

distribués entre 1 et $w_{\max}$ et les valeurs v_i sont uniformément distribuées entre $p_i - w_{\max}/10$ et $p_i + w_{\max}/10$ [69]. Les capacités des sacs sont uniformément réparties entre $\frac{nw_{\max}}{8m}$ et $\frac{nw_{\max}}{2m}$.

4.2 Protocole et architecture

Les modèles RL sont entraînés sur une base d'apprentissage d'exemplaires générées aléatoirement selon les modalités décrites dans la section précédente pendant 500 épisodes. L'architecture utilisée (schématisée en Figure 4.1) est composée de 3 couches de convolution de graphes de type GCN [48] partagées entre le value network et le policy network, suivies de deux sorties de type réseaux denses. La première sortie correspond au value network et donne un scalaire $v_\theta(s)$ correspondant à une estimation de la valeur de l'état s. Les activations de la seconde sortie sont filtrées par un masque pour ne prendre en compte que les activations correspondantes aux valeurs encore présentes après filtrage au moment de l'inférence et sont suivies d'une activation softmax. Cette deuxième sortie donne un vecteur de probabilité $P(s, \cdot)$ dont chaque composante est la probabilité de sélection de l'action correspondante. Cette architecture permet de partager les couches de convolutions de graphes entre le value-network et le policy network, l'intuition étant que ces couches intermédiaires extraient des caractéristiques utiles aux problèmes de prédiction, à la fois pour la fonction de valeur et pour la politique optimale.

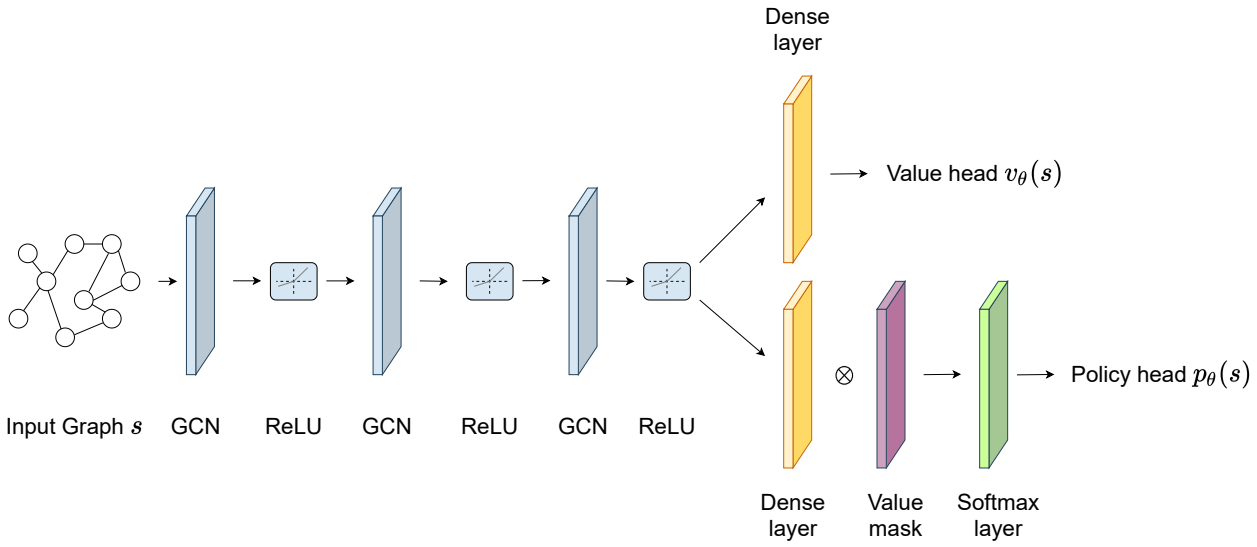


Figure 4.1 Architecture du modèle MCTS+RL

L'évaluation après entraînement se fait sur un ensemble de test comprenant 500 instances issues de la distribution d'entraînement.

Le Tableau 4.1 présente les différents hyperparamètres utilisés pour obtenir les résultats présentés dans les sections suivantes.

Paramètre	**Symbole**	**Valeur**
Pas d'apprentissage pour la Q-fonction	α	1
Nombre d'exemplaires à chaque époque	N_e	10
Nombre d'épisodes	–	500
Nombre d'itérations maximal du MCTS par exemplaire	$T_{\max}$	∞
Nombre maximal de tuples (s, a, r, s') stockés dans le *replay buffer*	–	10^4
Pas d'apprentissage de la descente de gradient stochastique	β	$5 \cdot 10^{-4}$
Constante d'équilibre exploration-exploitation	c_{puct}	$1/\sqrt{2}$
Nombre de neurones dans la *value head*	–	256
Nombre de neurones dans la *policy head*	–	256

Tableau 4.1 Hyperparamètres utilisés pour les expériences

4.3 Dynamiques d'apprentissage et profils de performance

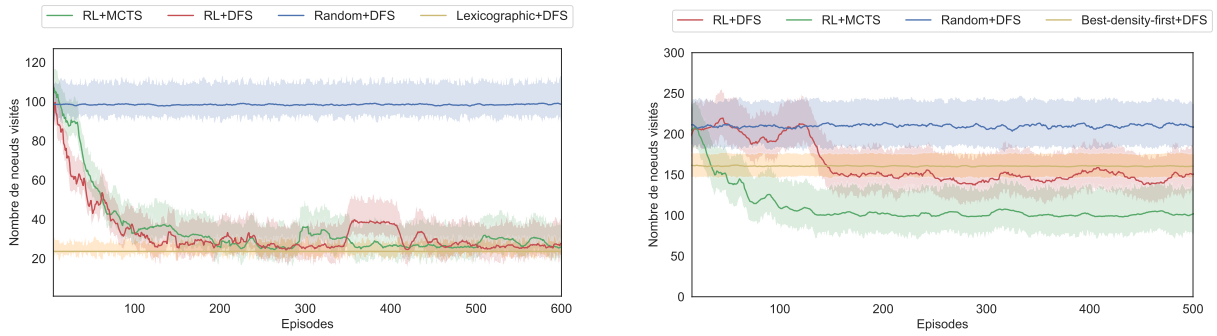
Dans cette section, nous cherchons à répondre à la première question de recherche : l'algorithme hybride MCTS+RL apporte-t-il un gain en terme de performance par rapport à l'hybride DFS+RL ? Cette première expérience consiste en un entraînement de chaque architecture sur une base de problèmes générés selon les modalités spécifiées dans la section précédente. Un nouveau problème est échantilloné à chaque itération. La Figure 4.2 rapporte le nombre de noeuds explorés avant preuve d'optimalité sur une base de validation comprenant 10 nouveaux problèmes, indépendants de la base d'apprentissage en fonction du nombre d'itérations. Pour les deux problèmes à l'étude, nous rapportons également les performances moyennes d'une heuristique de sélection de valeur aléatoire, et pour le problème du coloriage de graphe, nous rapportons également le comportement de l'heuristique lexicographique, qui sélectionne la plus petite valeur du domaine sélectionné.

Ces résultats permettent de constater que les deux modèles parviennent effectivement à apprendre par interactions avec le problème. Pour le problème du coloriage de graphe, nous obtenons des résultats similaires à ceux obtenus dans [2]. Distinguons l'analyse en fonction du problème.

Coloriage de graphe On observe des dynamiques d'apprentissage très similaires entre l'algorithme initial proposé par [2] (RL+MCTS) et la méthode proposée dans ce mémoire

(RL+MCTS). À convergence, les deux heuristiques apprises par RL performent significativement mieux qu'une recherche aléatoire et s'approchent des performances d'un ordonnancement lexicographique. La bonne performance de cette dernière heuristique s'explique simplement : sur ce problème, nous cherchons à minimiser le nombre de couleurs, or un choix de valeur lexicographique est biaisé en faveur du choix d'un nombre limité de couleurs, et oriente donc potentiellement la recherche vers de bonnes solutions.

Sac à dos multiples Pour ce problème, nous ordonnons les variables selon la densité de valeur v_i/w_i décroissante. Nous utilisons comme heuristique de référence l'heuristique consistant à affecter la valeur 1 à la variable sélectionnée, que nous nommerons Best-density-first. Nous observons des dynamiques différentes sur ce second problème. Bien que les deux méthodes finissent par mieux performer qu'une recherche aléatoire et que l'heuristique Best-density-first, la recherche basée sur l'algorithme du MCTS converge plus rapidement vers une meilleure heuristique, comme cela peut être observé par le plus faible nombre de noeuds explorés en moyenne. Cette différence souligne le caractère structurellement différent des deux problèmes à l'étude, ainsi qu'une meilleure performance de la recherche MCTS sur ce problème.

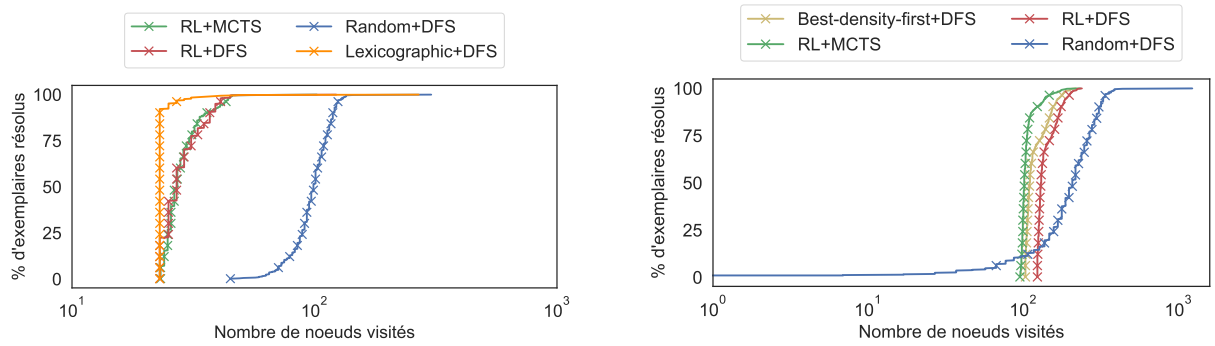


(a) (b)

Figure 4.2 Évolution du nombre de noeuds explorés sur une base de test en fonction du nombre d'itérations d'apprentissage. (a) Coloriage de graphe, $n = 20, \rho = 0.2$. (b) Sac à dos multiple, $n = 60, m = 4, w_{\max} = 10$

La Figure 4.3 représente les profils de performances des différentes heuristiques, comprenant les deux heuristiques basées sur du RL après apprentissage. Ces graphiques permettent de confirmer les deux observations précédentes : les heuristiques basées sur l'apprentissage par renforcement (DFS+RL et MCTS+RL) obtiennent un profil de performance très proche de celui obtenu par l'heuristique lexicographique sur le problème du coloriage de graphe, et la méthode MCTS/RL obtient un profil légèrement meilleur que celui de la méthode DFS+RL

sur le problème du Multiknapsack.



(a) (b)

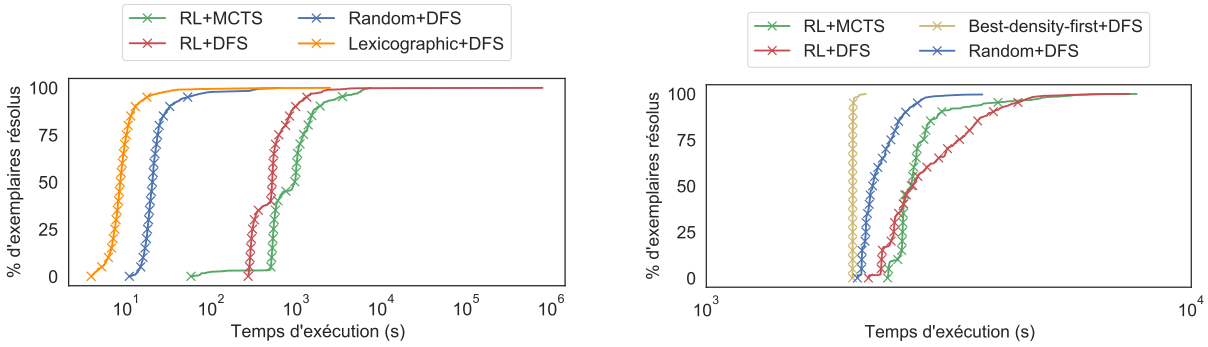
Figure 4.3 Profils de performance. (a) Coloriage de graphe, $n = 20, \rho = 0.2$. (b) Sac à dos multidimensionnel, $n = 60, m = 4, w_{\max} = 10$

La Figure 4.4 présente le temps de résolution de chaque méthode sur chacun des deux problèmes. On observe que du point de vue du temps de résolution effectif pour chaque problème, les deux méthodes basées sur du RL ne sont pas compétitives, et que même une recherche aléatoire obtient de meilleures performances de ce point de vue. Cette observation peut s'expliquer par la lourdeur de l'inférence nécessaire à chaque itération pour les méthodes RL+DFS et RL+MCTS. En effet, chaque itération dans l'arbre de recherche de ces méthodes nécessite une inférence avec un GNN, qui malgré le fait qu'elle soit effectuée sur GPU, reste une opération très lourde. Il est à noter que du fait que l'on utilise un paradigme séquentiel en RL, il est impossible de faire de l'inférence en *batch* c'est-à-dire effectuer plusieurs prédictions en une seule inférence. Or l'inférence en batch est en général très importante en ML pour l'accélération des modèles d'apprentissage automatique. L'accélération des modèles proposés demeure un problème ouvert, et constitue une opportunité de recherche future intéressante, afin de rendre ces approches utilisables en pratique.

4.4 Robustesse et *transfer learning*

La troisième question que l'on cherche à investiguer traite de la robustesse des différentes méthodes au sens du *transfer learning*. Pour introduire cette notion, il est utile de prendre du recul et de schématiser le flux de travail en CP conventionnelle (sans méthodes de learning) et le flux de travail nécessaire lors de l'utilisation de méthode de learning.

Habituellement, pour résoudre un problème en CP (mais également avec d'autres paradigmes non-basés sur du Machine Learning), on procède comme suit :



(a) (b)

Figure 4.4 Profil de temps de résolution. (a) Coloriage de graphe, $n = 20, \rho = 0.2$. (b) Sac à dos multiple, $n = 60, m = 4, w_{\max} = 10$

1. écriture et entrée du modèle dans un solveur,
2. conception et implantation d'une heuristique de recherche adaptée,
3. résolution du problème.

Cette procédure est répétée autant de fois que nécessaire jusqu'à l'obtention d'une solution satisfaisante. Si l'on veut résoudre un nouveau problème, il est nécessaire de reprendre toutes les étapes depuis le début. Les méthodes basées sur de l'apprentissage automatique telle que celle présentée dans [2] ou encore dans ce mémoire nécessite une approche totalement différente :

1. écriture du modèle
2. création d'une distribution de problèmes représentatifs du problème à résoudre
3. entraînement d'un modèle d'apprentissage machine sur la distribution d'entraînement pour apprendre une heuristique
4. résolution du problème de test avec le modèle appris précédemment.

À ce titre, ce type d'approche peut être considérée comme indirecte. Une conséquence notable qui n'est pas sans surprise du point de vue du praticien de la CP traditionnelle est que l'heuristique obtenue à la fin de cette procédure dépend des problèmes utilisés lors de l'apprentissage. Cette dépendance, bien que tout à fait acceptée en apprentissage automatique, peut soulever quelques interrogations : quelle est la nature de cette dépendance, peut-on la quantifier, que se passe-t-il lorsque qu'on entraîne une heuristique sur une distribution $\mathcal{P}$ mais qu'on l'évalue sur un problème appartenant à une autre distribution $\mathcal{P}' \neq \mathcal{P}$? Ces questions sont d'un intérêt pratique pour une raison importante : bien qu'automatique, l'entraînement des modèles d'apprentissage automatique pour la résolution de problèmes combinatoires est une

procédure très coûteuse, en terme de temps et de matériel. Le praticien peut donc s'attendre à ce que ce coût puisse être amorti en utilisant la même heuristique sur un grand nombre de problèmes légèrement différents de ceux utilisés pour l'entraînement. Toutes ces questions relèvent du domaine du *transfer learning*, autrement dit de l'analyse du comportement d'un modèle sur une distribution différente de celle utilisée pour l'entraînement.

Dans cette section, nous étudions deux dépendances relatives à la distribution d'entraînement :

1. quel est l'impact de la taille de la base de données d'apprentissage sur les méthodes RL+DFS et RL+MCTS ?
2. comment évaluer le transfer learning de chacune de ces deux méthodes ?

La Figure 4.5 présente l'évolution du nombre de noeuds explorés après apprentissage en faisant varier la taille de la base d'apprentissage sur le problème du sac à dos multidimensionnel. Nous observons deux phénomènes attendus :

1. Plus le nombre de problèmes vus lors de l'apprentissage est élevé, meilleures sont les performances sur une base de validation.
2. Il y a un phénomène de saturation : à partir d'une certaine taille, les performances cessent de s'améliorer.

On note également que la méthode RL+MCTS semble mieux tirer parti d'un faible nombre d'exemplaires, visible par la décroissance plus rapide du nombre moyen de noeuds explorés.

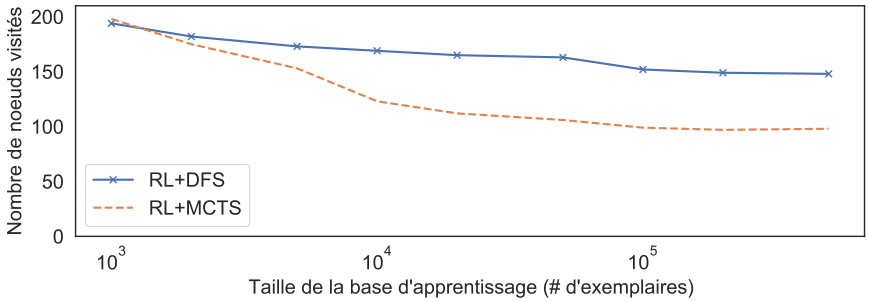


Figure 4.5 Évolution de la performance en fonction du nombre de problèmes dans la base de données d'apprentissage, évaluée sur une base de validation de 500 problèmes

Intéressons-nous maintenant à la quantification du transfer learning de ces deux méthodes. Bien que d'une certaine importance pratique dans le contexte du ML appliqué à l'optimisation discrète, cette question est malheureusement très peu explorée dans la littérature. Nous proposons de quantifier le transfer learning par le protocole suivant, illustré en Figure 4.6 :

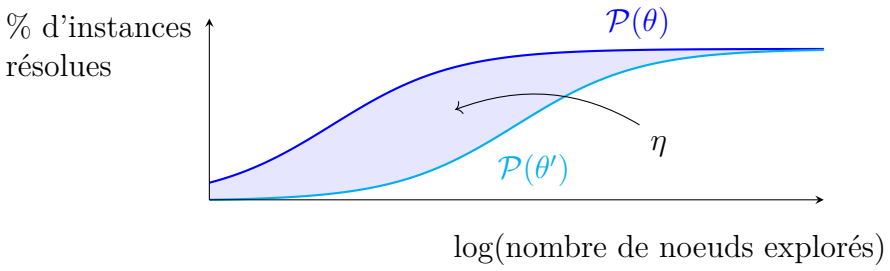


Figure 4.6 Illustration de l'indicateur de robustesse d'une heuristique

1. On entraîne une heuristique par apprentissage par renforcement sur une distribution de problèmes $\mathcal{P}(\theta)$ où θ est un vecteur de paramètres, et on calcule le profil de performance F_θ de cette heuristique sur la dstribution $\mathcal{P}(\theta)$. Autrement dit, $F_\theta(x)$ donne la proportion de problèmes résolus en moins de x échecs parmi une base de validation suivant la distribution $\mathcal{P}(\theta)$.
2. Une fois entraînée sur $\mathcal{P}(\theta)$, on évalue la même heuristique sur une distribution différente $\mathcal{P}(\theta')$ et on calcule le profil de performance $F_{\theta'}$. Intuitivement, nous nous attendons à ce que ce profil $F_{\theta'}$ se décale vers la droite par rapport à F_θ, car l'heuristique sera moins performante sur la distribution $\mathcal{P}(\theta')$.
3. On considère l'indicateur de transfert suivant :

$$\eta = \int_0^{+\infty} F_\theta(\log x) - F_{\theta'}(\log x)\,\mathrm{d}x \tag{4.6}$$

autrement dit, nous quantifions le transfert learning par l'aire située entre les profils F_θ et $F_{\theta'}$ dans le plan logarithmique en le nombre d'échecs. Intuitivement, si cet indicateur vaut 0, cela signifie que les deux profils se superposent [2] et que le transfert est parfait. Plus cet indicateur est élevé, plus le profil $F_{\theta'}$ est décalé vers la droite par rapport à F_θ traduisant ainsi une perte importante de performance sur la distribution $\mathcal{P}(\theta')$.

La Figure 4.7 présente un diagramme de transfert sur le problème du coloriage de graphe, c'est-à-dire l'indicateur de transfert η obtenu en faisant varier les paramètres de la distribution (taille du graphe et densité). Pour toutes les heuristiques, le diagramme de performance de référence F_θ est celui obtenu avec les paramètres $n = 10$ et $\rho = 0.2$ et les heuristiques sans apprentissage (Random et Lexicographic) sont utilisées au sein d'une recherche DFS.

Une première observation attendue est qu'une heuristique de sélection de valeur aléatoire possède un très mauvais transfert : en effet, par rapport au profil de référence obtenu avec

2. Nous supposons que $F_\theta(x) \geq F_{\theta'}(x)$ pour tout x, ce qui signifie que l'heuristique est toujours plus performante sur $\mathcal{P}(\theta)$ que sur $\mathcal{P}(\theta')$.

$n = 10$ et $\rho = 0.2$, l'indicateur η montre une augmentation significative à mesure que la distribution de graphes s'éloigne de la distribution de référence. En réalité, la Figure 4.7 présente une version saturée et normalisée de η car les capacités de transfert de l'heuristique aléatoire sont si mauvaises qu'elles auraient rendue le reste de la figure très peu lisible. L'ordonnancement lexicographique présente au contraire un excellent transfert sur ce problème. Malgré le fait que les problèmes deviennent plus difficiles (ne serait-ce que parce que la taille des problèmes augmentent selon l'axe des abscisses), l'indicateur η n'augmente pas de manière significative, montrant que cette heuristique s'adapte bien à des problèmes de graphes plus grands et plus denses sans significativement augmenter le nombre d'échecs dans l'arbre de recherche relativement aux autres méthodes. Les méthodes basées sur l'apprentissage par renforcement présentent un diagramme intermédiaire entre celui de Random et celui de Lexicographic : le transfert semble être généralement meilleur que celui d'une recherche aléatoire, indiquant qu'il y aurait une forme de transfert vers les problèmes de graphes plus difficiles, mais les performances se dégradent significativement à mesure que l'on s'éloigne de la distribution d'apprentissage. La méthode présentée dans ce mémoire (RL+MCTS) semble avoir un transfert très légèrement meilleur que celui de la méthode présentée dans [2] (RL+DFS) même si cette différence reste faible.

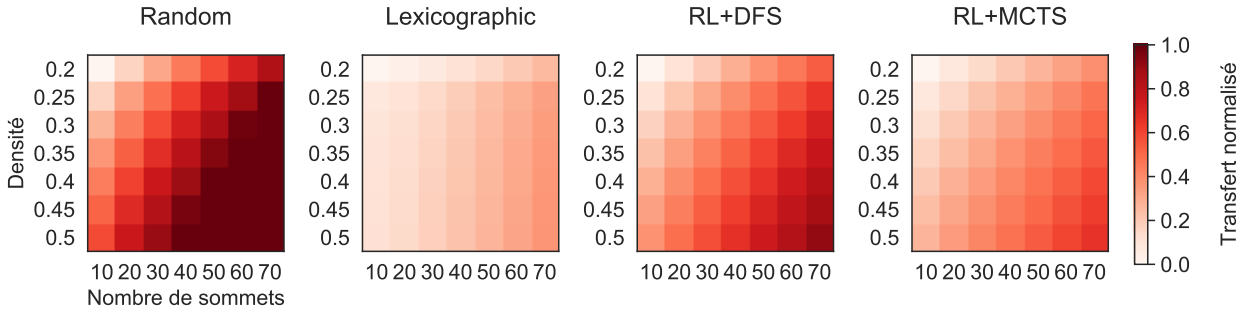


Figure 4.7 Diagramme de transfert normalisé sur le problème du coloriage de graphe avec pour profil de référence $n = 10$ et $\rho = 0.2$

On peut en outre tirer la conclusion suivante : même si les résultats de la Figure 4.3(a) montrent que les méthodes RL+DFS et RL+MCTS obtiennent des performances similaires à Lexicographic lorsqu'elles sont évaluées sur la distribution d'entraînement, cela ne veut pas dire que le comportement appris est similaire à celui de Lexicographic. En réalité, les résultats présentés en Figure 4.7 démontrent que les politiques apprises par les deux méthodes d'apprentissage sont très différentes de Lexicographic car leur transfert est significativement moins bon.

Les résultats précédents montrent que les capacités de transfert des méthodes basées sur du RL sont assez mauvaises lorsqu'elles sont entraînées sur une distribution très peu *étalée* (au sens où son entropie est faible) : en effet, les méthodes ont été entraînées uniquement sur des problèmes de graphes avec $n = 10$ et $\rho = 0.2$. Il est raisonnable de penser que si, lors de l'entraînement, les agents de RL peuvent apprendre sur une base d'apprentissage plus diversifiée (avec une entropie plus élevée), les capacités de transfert peuvent être améliorée. La Figure 4.8 présente une comparaison des diagrammes de transfert précédents avec ceux obtenus lorsque les heuristiques de RL sont entraînées sur une distribution plus diversifiée : au lieu d'être entraînées sur des graphes de taille $n = 10$ et de densité $\rho = 0.2$, les heuristiques sont entraînées sur une distribution bi-variée de graphes $P(n, \rho)$ de sorte que

— la marginale $P(n)$ soit une distribution géométrique d'espérance $n = 10$

— la distribution conditionnelle $P(\rho \mid n)$ suit une loi exponentielle d'espérance $\rho = 0.2$.

Avec cette distribution d'apprentissage, les agents de RL apprennent essentiellement sur la base de graphes de paramètres $n = 10$ et $\rho = 0.2$ mais observent également d'autres types de graphes.

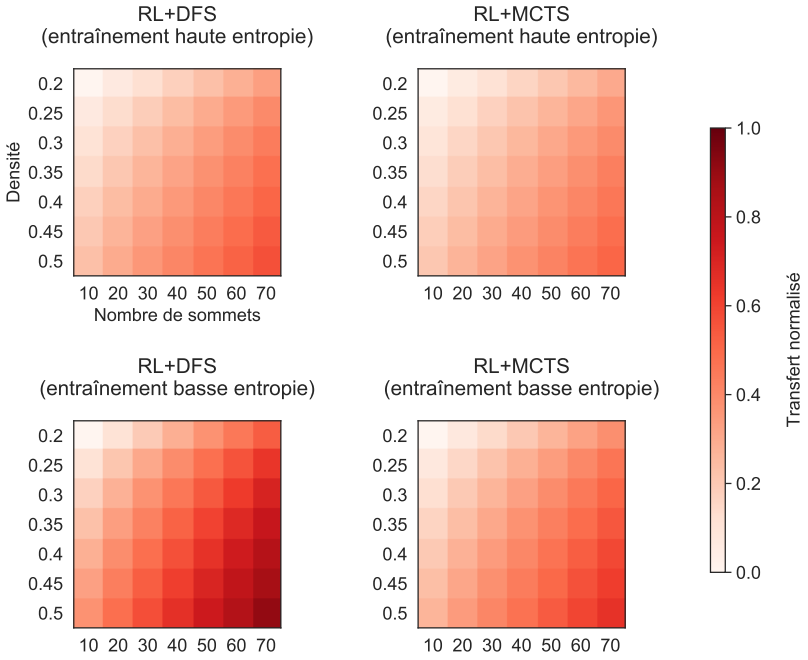


Figure 4.8 Comparaison des diagrammes de transfert avec un entraînement sur une distribution à entropie haute (en haut) et une distribution à entropie basse (en bas)

En diversifiant les types de graphes rencontrés lors de l'entraînement, on observe une amélioration du transfert sur les problèmes plus difficiles, même si cette amélioration demeure bien en deçà des capacités de transfert d'une heuristique comme Lexicographic.

CHAPITRE 5 CONCLUSION ET PERSPECTIVES

5.1 Synthèse des travaux

Dans ce mémoire, nous avons proposé un nouvel algorithme de recherche hybride intégré à un solveur de programmation par contraintes, inspiré du système AlphaZero. Cet algorithme combine de manière très étroite une méthode de recherche arborescente équilibrant exploration et exploration à chaque noeud avec deux réseaux de neurones entraînés par apprentissage par renforcement permettant conjointement de guider la recherche efficacement tout en tirant parti d'expériences passées pour améliorer la recherche au fur et à mesure de la résolution du problème. Expérimentalement, cette méthode montre de bonnes performances en terme de noeuds explorés sur le problème du sac-à-dos multidimensionnel par rapport à l'algorithme de recherche proposé dans [2] Il a également été montré que cette nouvelle méthode a de meilleures propriétés de transfert par rapport à la méthode [2], autrement dit, l'heuristique apprise grâce à notre méthode est moins sensible à un changement de distribution de problèmes ce qui est un atout car il est possible d'amortir le coût élevé d'entraînement de ce modèle en l'utilisant sur des problèmes légèrement différents de ceux vus pendant l'apprentissage.

5.2 Limitations de la solution proposée

Les méthodes basées sur de l'apprentissage par renforcement demeurent néanmoins très gourmandes en ressources et ne sont pas compétitives sur le plan du temps de résolution, ce qui limite considérablement leur intérêt pratique. De plus, malgré l'amélioration des capacités de transfert par rapport à la méthode initiale basée sur la recherche en profondeur, ce transfert reste très faible et les modèles appris souffrent toujours d'une baisse de performance importante lorsqu'ils sont utilisés pour résoudre des problèmes légèrement différents de ceux vus pendant l'apprentissage.

L'analyse empirique préliminaire présentée dans ce mémoire repose sur des instances de petite taille et sur seulement deux problèmes différents. Les conclusions tirées doivent donc être prises avec une certaine précaution et nous ne pouvons pas affirmer qu'elles restent valides à plus grande échelle. Elles permettent néanmoins de formuler des hypothèses de recherche pertinentes pouvant être validées avec une étude expérimentale plus conséquente.

5.3 Améliorations futures

Plusieurs aspects de ce sujet peuvent être explorés pour améliorer la méthode proposée. Outre la question du temps de résolution et du transfer learning évoqué plus haut, il semble raisonnable de vouloir consolider les résultats précédents avec une étude expérimentale plus exhaustive. Nous pouvons également citer trois directions de recherche potentiellement intéressantes pour développer ce sujet.

La première amélioration porte sur la parallélisation de cet algorithme. La parallélisation en programmation par contraintes est un sujet très largement étudié [70, 71]. Le Monte Carlo Tree Search que nous utilisons comme noyau dur de notre méthode a lui-même été adapté à des utilisations en parallèle [72, 73]. La parallélisation de la méthode proposée pourrait permettre de mitiger sa lenteur sur des problèmes suffisamment larges, et pourrait également permettre d'utiliser les réseaux de neurones en *batch* en parallélisant l'inférence sur plusieurs *workers* ce qui réduirait le goulot d'étranglement de performance qui se trouve actuellement du côté de l'inférence par les modèles de GNN.

Une deuxième direction de recherche intéressante est celle de l'exploitation de caractéristiques des valeurs au sein du GNN. En effet, dans ce mémoire, nous n'avons pas cherché à exploiter d'informations particulières sur les valeurs du domaine de la variable sélectionnée. L'exploitation d'informations supplémentaires comme par exemple une estimation de la *densité de solution* locale à chaque assignation variable-valeur à la manière de ce qui est proposé dans [74] peut donner plus d'informations aux réseaux de neurones pour guider la recherche et potentiellement réduire le nombre de noeuds explorés.

Une troisième direction de recherche pourrait être d'explorer de nouvelles architectures plus légères et plus adaptées à une utilisation séquentielle au sein d'un arbre de recherche. Actuellement, les performances sont principalement limitées par l'utilisation de modèles d'apprentissage automatique assez lourds tels que les GNN. Dans notre cas, comme il est difficile de paralléliser l'inférence de ces modèles, l'exécution s'en retrouve fortement impactée. L'utilisation de modèles plus légers pourrait apporter un gain en terme de temps d'exécution.

RÉFÉRENCES

[1] M. Gasse, D. Chételat, N. Ferroni, L. Charlin et A. Lodi, "Exact combinatorial optimization with graph convolutional neural networks," *arXiv preprint arXiv :1906.01629*, 2019.

[2] F. Chalumeau, I. Coulon, Q. Cappart et L.-M. Rousseau, "Seapearl : A constraint programming solver guided by reinforcement learning," *arXiv preprint arXiv :2102.09193*, 2021.

[3] Q. Cappart, T. Moisan, L.-M. Rousseau, I. Prémont-Schwarz et A. Cire, "Combining reinforcement learning and constraint programming for combinatorial optimization," *arXiv preprint arXiv :2006.01610*, 2020.

[4] E. Khalil, P. Le Bodic, L. Song, G. Nemhauser et B. Dilkina, "Learning to branch in mixed integer programming," dans *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 30, n°. 1, 2016.

[5] D. Silver, T. Hubert, J. Schrittwieser, I. Antonoglou, M. Lai, A. Guez, M. Lanctot, L. Sifre, D. Kumaran, T. Graepel *et al.*, "A general reinforcement learning algorithm that masters chess, shogi, and go through self-play," *Science*, vol. 362, n°. 6419, p. 1140–1144, 2018.

[6] P. Boizumault et Y. Delon, "L. p eridy. planning exams using constraint logic programming," dans *Proc. 2nd International Conference on the Practical Applications of Prolog*, 1994.

[7] G. Weil, K. Heus, P. Francois et M. Poujade, "Constraint programming for nurse scheduling," *IEEE Engineering in medicine and biology magazine*, vol. 14, n°. 4, p. 417–422, 1995.

[8] S. Bourdais, P. Galinier et G. Pesant, "Hibiscus : A constraint programming application to staff scheduling in health care," dans *International Conference on Principles and Practice of Constraint Programming*. Springer, 2003, p. 153–167.

[9] N. Guerinik et M. Van Caneghem, "Solving crew scheduling problems by constraint programming," dans *International Conference on Principles and Practice of Constraint Programming*. Springer, 1995, p. 481–498.

[10] T. Creemers, L. R. Giralt, J. Riera, C. Ferrarons, J. Rocca et X. Corbella, "Constraint-based maintenance scheduling on an electric power-distribution network," dans *Proc. of Practical Application of Prolog (PAP95), Paris, France*, 1995.

[11] D. Sabin, M. Sabin, R. D. Russell et E. C. Freuder, "A constraint-based approach to diagnosing software problems in computer networks," dans *International Conference on Principles and Practice of Constraint Programming*. Springer, 1995, p. 463–480.

[12] C. Chiopris et M. Fabris, "Optimal management of a large computer network with chip," dans *2nd Conf Practical Applications of Prolog*, 1994.

[13] P. Barahona et L. Krippahl, "Constraint programming in structural bioinformatics," *Constraints*, vol. 13, n°. 1-2, p. 3–20, 2008.

[14] R. Oliveira et M. Ribeiro, "Comparing mixed & integer programming vs. constraint programming by solving job-shop scheduling problems," vol. 6, p. 211–238, 03 2015.

[15] H. D. Sherali et P. J. Driscoll, "Evolution and state-of-the-art in integer programming," *Journal of Computational and Applied Mathematics*, vol. 124, n°. 1, p. 319 – 340, 2000, numerical Analysis 2000. Vol. IV : Optimization and Nonlinear Equations. [En ligne]. Disponible : http://www.sciencedirect.com/science/article/pii/S0377042700004313

[16] F. Boussemart, F. Hemery, C. Lecoutre et L. Sais, "Boosting systematic search by weighting constraints," dans *ECAI*, vol. 16, 2004, p. 146.

[17] P. Refalo, "Impact-based search strategies for constraint programming," dans *International Conference on Principles and Practice of Constraint Programming*. Springer, 2004, p. 557–571.

[18] A. Zanarini et G. Pesant, "Solution counting algorithms for constraint-centered search heuristics," *Constraints*, vol. 14, n°. 3, p. 392–413, 2009.

[19] M. Milano et W. J. van Hoeve, "Reduced cost-based ranking for generating promising subproblems," dans *Principles and Practice of Constraint Programming - CP 2002*, P. Van Hentenryck, édit. Berlin, Heidelberg : Springer Berlin Heidelberg, 2002, p. 1–16.

[20] T. Fahle, U. Junker, S. E. Karisch, N. Kohl, M. Sellmann et B. Vaaben, "Constraint programming based column generation for crew assignment," *Journal of Heuristics*, vol. 8, n°. 1, p. 59–81, 2002.

[21] J. N. Hooker, "A hybrid method for planning and scheduling," dans *International Conference on Principles and Practice of Constraint Programming*. Springer, 2004, p. 305–316.

[22] P. J. Stuckey, "Lazy clause generation : Combining the power of sat and cp (and mip ?) solving," dans *Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems*, A. Lodi, M. Milano et P. Toth, édit. Berlin, Heidelberg : Springer Berlin Heidelberg, 2010, p. 5–9.

[23] H. Hojabri, M. Gendreau, J.-Y. Potvin et L.-M. Rousseau, “Large neighborhood search with constraint programming for a vehicle routing problem with synchronization constraints,” *Computers & Operations Research*, vol. 92, p. 87 – 97, 2018. [En ligne]. Disponible : http://www.sciencedirect.com/science/article/pii/S0305054817302885

[24] D. Bergman, A. A. Cire, W.-J. Van Hoeve et J. Hooker, *Decision diagrams for optimization*. Springer, 2016, vol. 1.

[25] R. Gentzel, L. Michel et W.-J. van Hoeve, “Haddock : A language and architecture for decision diagram compilation,” dans *International Conference on Principles and Practice of Constraint Programming*. Springer, 2020, p. 531–547.

[26] J. Kober, J. A. Bagnell et J. Peters, “Reinforcement learning in robotics : A survey,” *The International Journal of Robotics Research*, vol. 32, n°. 11, p. 1238–1274, 2013.

[27] Y. Li, Y. Wen, D. Tao et K. Guan, “Transforming cooling optimization for green data center via deep reinforcement learning,” *IEEE transactions on cybernetics*, vol. 50, n°. 5, p. 2002–2013, 2019.

[28] O. Vinyals, I. Babuschkin, W. M. Czarnecki, M. Mathieu, A. Dudzik, J. Chung, D. H. Choi, R. Powell, T. Ewalds, P. Georgiev *et al.*, “Grandmaster level in starcraft ii using multi-agent reinforcement learning,” *Nature*, vol. 575, n°. 7782, p. 350–354, 2019.

[29] D. Silver, A. Huang, C. J. Maddison, A. Guez, L. Sifre, G. Van Den Driessche, J. Schrittwieser, I. Antonoglou, V. Panneershelvam, M. Lanctot *et al.*, “Mastering the game of go with deep neural networks and tree search,” *nature*, vol. 529, n°. 7587, p. 484–489, 2016.

[30] D. Silver, T. Hubert, J. Schrittwieser, I. Antonoglou, M. Lai, A. Guez, M. Lanctot, L. Sifre, D. Kumaran, T. Graepel *et al.*, “Mastering chess and shogi by self-play with a general reinforcement learning algorithm,” *arXiv preprint arXiv :1712.01815*, 2017.

[31] J. Schrittwieser, I. Antonoglou, T. Hubert, K. Simonyan, L. Sifre, S. Schmitt, A. Guez, E. Lockhart, D. Hassabis, T. Graepel *et al.*, “Mastering atari, go, chess and shogi by planning with a learned model,” *Nature*, vol. 588, n°. 7839, p. 604–609, 2020.

[32] F. Rossi, P. Van Beek et T. Walsh, *Handbook of constraint programming*. Elsevier, 2006.

[33] N. Ascheuer, M. Fischetti et M. Grötschel, “Solving the asymmetric travelling salesman problem with time windows by branch-and-cut,” *Mathematical Programming*, vol. 90, n°. 3, p. 475–506, 2001.

[34] T. Bektas, “The multiple traveling salesman problem : an overview of formulations and solution procedures,” *Omega*, vol. 34, n°. 3, p. 209–219, 2006.

[35] N. Beldiceanu, M. Carlsson et J.-X. Rampon, “Global constraint catalog,” 2010.

[36] W. D. Harvey et M. L. Ginsberg, “Limited discrepancy search,” dans *IJCAI (1)*, 1995, p. 607–615.

[37] M. L. Alistair, A. Sinclair et D. Zuckerman, “Optimal speedup of las vegas algorithms,” *Information Processing Letters*, vol. 47, p. 173–180, 1993.

[38] C. P. Gomes, B. Selman, N. Crato et H. Kautz, “Heavy-tailed phenomena in satisfiability and constraint satisfaction problems,” *Journal of automated reasoning*, vol. 24, n°. 1, p. 67–100, 2000.

[39] B. C. Csáji *et al.*, “Approximation with artificial neural networks,” *Faculty of Sciences, Etvs Lornd University, Hungary*, vol. 24, n°. 48, p. 7, 2001.

[40] P. W. Battaglia, R. Pascanu, M. Lai, D. Rezende et K. Kavukcuoglu, “Interaction networks for learning about objects, relations and physics,” *arXiv preprint arXiv :1612.00222*, 2016.

[41] D. Duvenaud, D. Maclaurin, J. Aguilera-Iparraguirre, R. Gómez-Bombarelli, T. Hirzel, A. Aspuru-Guzik et R. P. Adams, “Convolutional networks on graphs for learning molecular fingerprints,” *arXiv preprint arXiv :1509.09292*, 2015.

[42] T. Hamaguchi, H. Oiwa, M. Shimbo et Y. Matsumoto, “Knowledge transfer for out-of-knowledge-base entities : A graph neural network approach,” *arXiv preprint arXiv :1706.05674*, 2017.

[43] H. Dai, E. B. Khalil, Y. Zhang, B. Dilkina et L. Song, “Learning combinatorial optimization algorithms over graphs,” *arXiv preprint arXiv :1704.01665*, 2017.

[44] F. Scarselli, M. Gori, A. C. Tsoi, M. Hagenbuchner et G. Monfardini, “The graph neural network model,” *IEEE transactions on neural networks*, vol. 20, n°. 1, p. 61–80, 2008.

[45] J. Zhou, G. Cui, Z. Zhang, C. Yang, Z. Liu, L. Wang, C. Li et M. Sun, “Graph neural networks : A review of methods and applications,” *arXiv preprint arXiv :1812.08434*, 2018.

[46] J. Gilmer, S. S. Schoenholz, P. F. Riley, O. Vinyals et G. E. Dahl, “Neural message passing for quantum chemistry,” dans *International Conference on Machine Learning*. PMLR, 2017, p. 1263–1272.

[47] H. Dai, B. Dai et L. Song, “Discriminative embeddings of latent variable models for structured data,” dans *International conference on machine learning*. PMLR, 2016, p. 2702–2711.

[48] T. N. Kipf et M. Welling, “Semi-supervised classification with graph convolutional networks,” *arXiv preprint arXiv :1609.02907*, 2016.

[49] R. S. Sutton et A. G. Barto, *Reinforcement learning : An introduction*. MIT press, 2018.

[50] R. Bellman, "Dynamic programming," *Science*, vol. 153, n°. 3731, p. 34–37, 1966.

[51] C. J. Watkins et P. Dayan, "Q-learning," *Machine learning*, vol. 8, n°. 3-4, p. 279–292, 1992.

[52] V. S. Borkar, *Stochastic approximation : a dynamical systems viewpoint*. Springer, 2009, vol. 48.

[53] B. Bharath et V. S. Borkar, "Stochastic approximation algorithms : Overview and recent trends," *Sadhana*, vol. 24, n°. 4-5, p. 425–452, 1999.

[54] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra et M. Riedmiller, "Playing atari with deep reinforcement learning," *arXiv preprint arXiv :1312.5602*, 2013.

[55] R. S. Sutton, D. McAllester, S. Singh et Y. Mansour, "Policy gradient methods for reinforcement learning with function approximation," *Advances in neural information processing systems*, vol. 12, p. 1057–1063, 1999.

[56] C. B. Browne, E. Powley, D. Whitehouse, S. M. Lucas, P. I. Cowling, P. Rohlfshagen, S. Tavener, D. Perez, S. Samothrakis et S. Colton, "A survey of monte carlo tree search methods," *IEEE Transactions on Computational Intelligence and AI in games*, vol. 4, n°. 1, p. 1–43, 2012.

[57] C.-S. Lee, M.-H. Wang, G. Chaslot, J.-B. Hoock, A. Rimmel, O. Teytaud, S.-R. Tsai, S.-C. Hsu et T.-P. Hong, "The computational intelligence of mogo revealed in taiwan's computer go tournaments," *IEEE Transactions on Computational Intelligence and AI in games*, vol. 1, n°. 1, p. 73–89, 2009.

[58] T. Raiko et J. Peltonen, "Application of uct search to the connection games of hex, y,* star, and renkula !" *AI and Machine Consciousness*, 2008.

[59] J. Nijssen, "Playing othello using monte carlo," *Strategies*, p. 1–9, 2007.

[60] L. Kocsis et C. Szepesvári, "Bandit based monte-carlo planning," dans *European conference on machine learning*. Springer, 2006, p. 282–293.

[61] P. Auer, N. Cesa-Bianchi et P. Fischer, "Finite-time analysis of the multiarmed bandit problem," *Machine learning*, vol. 47, n°. 2, p. 235–256, 2002.

[62] D. Auger, A. Couetoux et O. Teytaud, "Continuous upper confidence trees with polynomial exploration–consistency," dans *Joint European Conference on Machine Learning and Knowledge Discovery in Databases*. Springer, 2013, p. 194–209.

[63] I. Bello, H. Pham, Q. V. Le, M. Norouzi et S. Bengio, "Neural combinatorial optimization with reinforcement learning," *arXiv preprint arXiv :1611.09940*, 2016.

[64] M. Loth, M. Sebag, Y. Hamadi, M. Schoenauer et C. Schulte, "Hybridizing constraint programming and monte-carlo tree search : Application to the job shop problem," dans *International Conference on Learning and Intelligent Optimization*. Springer, 2013, p. 315–320.

[65] Gecode Team, "Gecode : Generic constraint development environment," 2006, available from `http://www.gecode.org`.

[66] N. Jussien, G. Rochart et X. Lorca, "Choco : an open source java constraint programming library," 2008.

[67] O. Team, "Oscar : Scala in or, 2012."

[68] J. Bezanson, A. Edelman, S. Karpinski et V. B. Shah, "Julia : A fresh approach to numerical computing," *SIAM review*, vol. 59, n°. 1, p. 65–98, 2017.

[69] D. Pisinger, "Core problems in knapsack algorithms," *Operations Research*, vol. 47, n°. 4, p. 570–575, 1999.

[70] J.-C. Régin et A. Malapert, "Parallel constraint programming," dans *Handbook of Parallel Constraint Reasoning*. Springer, 2018, p. 337–379.

[71] J.-C. Régin, M. Rezgui et A. Malapert, "Embarrassingly parallel search," dans *International Conference on Principles and Practice of Constraint Programming*. Springer, 2013, p. 596–610.

[72] G. M.-B. Chaslot, M. H. Winands et H. J. van Den Herik, "Parallel monte-carlo tree search," dans *International Conference on Computers and Games*. Springer, 2008, p. 60–71.

[73] K. Rocki et R. Suda, "Large-scale parallel monte carlo tree search on gpu," dans *2011 IEEE International Symposium on Parallel and Distributed Processing Workshops and Phd Forum*. IEEE, 2011, p. 2034–2037.

[74] G. Pesant, "From support propagation to belief propagation in constraint programming," *Journal of Artificial Intelligence Research*, vol. 66, p. 123–150, 2019.