Résolution des problèmes d'optimisation combinatoire avec une stratégie de retour-arrière basée sur l'apprentissage par renforcement

Mémoire

Ilyess Bachiri

Maîtrise en informatique
Maître ès sciences (M.Sc.)

Québec, Canada

Résumé

Les problèmes d'optimisation combinatoire (*Constraint Optimization Problems* – COP) sont souvent difficiles à résoudre et le choix de la stratégie de recherche a une influence importante sur la performance du solveur. Pour de résoudre un problème d'optimisation combinatoire en explorant un arbre de recherche, il faut choisir une heuristique de choix de variable (qui définit l'ordre dans lequel les variables vont être instanciées), une heuristique de choix de valeur (qui spécifie l'ordre dans lequel les valeurs seront essayées), et une stratégie de retour-arrière (qui détermine vers quel nœud effectuer les retours-arrière lorsqu'une feuille de l'arbre est rencontrée).

Pour les stratégies de retour-arrière, il y a celles dont les retours-arrière sont totalement déterministes (e.g. *Depth-First Search* – DFS) et d'autres qui s'appuient sur des mécanismes d'évaluation de nœuds plus dynamiques (e.g. *Best-First Search*). Certaines (e.g. *Limited Discrepancy Search* – LDS) peuvent être implémentées soit comme un algorithme itératif déterministe ou un évaluateur de nœud.

Une stratégie est dite *adaptative* quand elle s'adapte dynamiquement à la structure du problème et identifie les zones de l'espace de recherche qui contiennent les "bonnes" solutions. Dans ce contexte, des stratégies de branchement adaptatives ont été proposées (e.g. *Impact-Based Search* – IBS) ainsi qu'une stratégie de retour-arrière adaptative (e.g. *Adaptive Discrepancy Search* – ADS), proposée pour les problèmes d'optimisation distribués.

À notre connaissance, aucune stratégie adaptative qui utilise l'apprentissage par renforcement (*Reinforcement Learning* – RL) pour supporter son mécanisme d'apprentissage n'a été proposée dans la littérature. Nous pensons que les techniques de RL permettront un apprentissage plus efficace et qu'une stratégie de retour-arrière munie de ces techniques aura le potentiel de résoudre les problèmes d'optimisation combinatoire plus rapidement.

Dans ce mémoire, nous proposons un algorithme (RLBS) qui “apprend” à faire des retours-arrière de manière efficace lors de l'exploration d'arbres non-binaires. Plus précisément, il s'agit une stratégie de retour-arrière qui se base sur l'apprentissage automatique pour améliorer la performance du solveur. En fait, nous utilisons l'apprentissage par renforcement pour identifier les zones de l'espace de recherche qui contiennent les bonnes solutions. Cette approche a été développée pour les problèmes d'optimisation combinatoire dont l'espace de recherche est encodé dans un arbre non-binaire. Comme les arbres sont non-binaires, on a l'occasion d'effectuer plusieurs retours-arrière vers chaque nœud durant l'exploration. Ceci permet d'apprendre quels nœuds mènent vers les meilleures récompenses en général (c'est-à-dire, vers les feuilles les plus intéressantes). Le branchement est effectué en utilisant une stratégie de choix de variable/valeur quelconque. Toutefois, quand un retour-arrière est nécessaire, la sélection du nœud cible s'appuie sur l'apprentissage par renforcement.

RLBS est évalué sur cinq instances industrielles du problème de la planification des opérations du rabotage du bois et a été comparé à ADS et à LDS sur cette même application. RLBS dépasse LDS et ADS, en termes de temps de calcul nécessaire à la résolution, sur chacune de ces instances-là et trouve la solution optimale plus rapidement. Les expérimentations ont montré que RLBS est en moyenne 4 fois plus rapide que ADS, et 6 fois plus rapide que LDS. RLBS a aussi été évalué sur une instance jouet du même problème et a été comparé à IBS. RLBS surpasse largement IBS. Il est capable de trouver une solution optimale en explorant beaucoup moins de nœuds que le nombre nécessaire à IBS pour trouver une telle solution.

Abstract

Combinatorial optimization problems are often very difficult to solve and the choice of a search strategy has a tremendous influence over the solver's performance. To solve a problem using search, one needs to choose a variable selection strategy (defining the order in which variables will be instantiated), a value selection strategy (that specifies the sequence in which we will try the variable possible values) and a backtracking strategy (that determines to which node we should backtrack/backjump, when a leaf is reached or a dead-end is encountered).

When it comes to backtracking strategies, there are some that are encoded into full deterministic algorithms (e.g. Depth-First Search – DFS), and others that rely on more dynamic node evaluator mechanisms (e.g. Best-First Search). Others (e.g. Limited Discrepancy Search – LDS) can be implemented as a deterministic iterative algorithm or as a node evaluator.

A strategy is said to be *adaptive* when it dynamically adapts to the structure of the problem and identifies the areas of the search space that contain good solutions. Some have proposed adaptive branching strategies (e.g. Impact-based Search – IBS) or a backtracking strategy (e.g. Adaptive Discrepancy Search – ADS) proposed for distributed optimization problems.

To our current knowledge, no adaptive backtracking strategy that relies on Reinforcement Learning (RL) has been proposed yet. We believe that RL techniques could allow a more efficient learning process and that, provided with these techniques, a backtracking strategy has a great potential of solving combinatorial optimization problems in a faster way.

In this thesis, we introduce an algorithm (RLBS) that learns to efficiently backtrack when searching non-binary trees. We consider a machine learning approach which improves the performance of the solver. More specifically, we use reinforcement learning

to identify the areas of the search space that contain good solutions. The approach was developed for optimization problems for which the search space is encoded as a non-binary tree. Since the trees are non-binary, we have the opportunity to backtrack multiple times to each node during the search. This allows learning which nodes generally lead to the best rewards (that is, to the most interesting leaves). Branching can be carried on using any variable/value selection strategy. However, when backtracking is needed, the selection of the target node involves reinforcement learning.

RLBS is evaluated on five instances of the lumber planing problem using real idustrial data, and it is compared to LDS and ADS. It outperforms classic (non-adaptive) search strategies (DFS, LDS), an adaptive branching strategy (IBS), and an adaptive backtracking strategy (ADS) on every instance of this problem. Experiments have shown that RLBS is on average 4 times faster than ADS, and 6 times faster than LDS. RLBS is also evaluated on a toy instance of the lumber planing problem and compared to IBS. RLBS substantially outperforms IBS by solving the problem to optimality much faster.

Table des matières

Résumé **iii**

Abstract **v**

Table des matières **vii**

Liste des tableaux **ix**

Liste des figures **xi**

Remerciements **xvii**

1 Introduction **1**

2 Programmation par contraintes **5**
2.1 Définitions 5
2.2 Résolution de problèmes combinatoires 6
2.3 Algorithmes basés sur l'analyse des déviations 15

3 Apprentissage par renforcement **23**
3.1 Définitions 23
3.2 Exemple : machine à sous 26
3.3 Algorithmes de résolution des problèmes combinatoires basés sur l'apprentissage 28

4 RLBS : Une stratégie de retour-arrière basée sur l'apprentissage par renforcement **43**
4.1 Le retour-arrière formulé comme un problème d'apprentissage par renforcement 44
4.2 Apprentissage par renforcement 46
4.3 Initialisation de l'algorithme 48
4.4 Explication de l'algorithme 48
4.5 Conclusion 49

5 Expérimentations **51**
5.1 Description du problème industriel 51

5.2 Problème de la planification du rabotage sous forme de COP 53
5.3 Heuristiques employées pour la résolution du problème 57
5.4 Configuration des instances industrielles 61
5.5 Résultats et discussion 62

6 Conclusion **73**

Bibliographie **75**

Liste des tableaux

3.1 Résumé des travaux qui combinent l'optimisation combinatoire et l'apprentissage 41

5.1 Temps de calcul nécessaire pour trouver la meilleure solution (RLBS vs. ADS) 66
5.2 Moyenne de temps de calcul pour trouver une solution d'une qualité quelconque (RLBS vs. ADS) 67
5.3 Réduction du temps de calcul pour obtenir la meilleure solution par RLBS vs. LDS pour différentes valeurs du taux d'apprentissage α 69
5.4 Réduction moyenne du temps de calcul pour trouver une solution de qualité quelconque par RLBS vs. LDS pour différentes valeurs du taux d'apprentissage α 69

Liste des figures

2.1 La carte géographique de l'Australie 7
2.2 Une partie de l'arbre de recherche construit pour résoudre le problème de coloriage de la carte de l'Australie 8
2.3 Premier niveau de l'arbre de recherche du problème de coloriage de la carte de l'Australie où la variable W prend les valeurs R, V et B dans l'ordre 10
2.4 Premier niveau de l'arbre de recherche du problème de coloriage de la carte de l'Australie où la variable W prend les valeurs V, R et B dans l'ordre 10
2.5 Premier niveau de l'arbre de recherche du problème de coloriage de la carte de l'Australie où la variable NT prend les valeurs V, R et B dans l'ordre 11
2.6 DFS sur un arbre binaire de profondeur 4. 14
2.7 Nombre total de déviations associées à chaque feuille dans un arbre binaire. 16
2.8 DDS sur un arbre binaire de profondeur 4 20

3.1 Le modèle basique de l'apprentissage par renforcement 24
3.2 Première descente de ADS dans un arbre de recherche 34
3.3 Portion de l'arbre de recherche visitée après un retour-arrière vers chacun des nœuds visités lors de la première descente par ADS 35
3.4 Relation en $ArcValue$ et $bestToDate$ 36

4.1 Éléments de la stratégie de retour-arrière sous forme de problème d'apprentissage par renforcement dans un arbre de recherche 45
4.2 Retour-arrière effectué par RLBS dans un arbre de recherche 47

5.1 Exemple de configuration d'un processus de rabotage. 53
5.2 Plan de production des opérations de rabotage sur six quarts de temps . 54
5.3 La valeur de la fonction objectif en fonction du temps de calcul de LDS, ADS, et RLBS pour le cas #1 63
5.4 La valeur de la fonction objectif en fonction du temps de calcul de LDS, ADS, et RLBS pour le cas #2 64
5.5 La valeur de la fonction objectif en fonction du temps de calcul de LDS, ADS, et RLBS pour le cas #3 64

5.6 La valeur de la fonction objectif en fonction du temps de calcul de LDS, ADS, et RLBS pour le cas #4 65
5.7 La valeur de la fonction objectif en fonction du temps de calcul de LDS, ADS, et RLBS pour le cas #5 65
5.8 La valeur de la fonction objectif en fonction du temps de calcul de RLBS, LDS, IBS et DFS sur un problème jouet 68
5.10 La valeur de la fonction objectif en fonction du temps de calcul de RLBS avec différentes valeurs de alpha pour le cas #2 70
5.9 La valeur de la fonction objectif en fonction du temps de calcul de RLBS avec différentes valeurs de alpha pour le cas #1 70
5.11 La valeur de la fonction objectif en fonction du temps de calcul de RLBS avec différentes valeurs de alpha pour le cas #3 71
5.12 La valeur de la fonction objectif en fonction du temps de calcul de RLBS avec différentes valeurs de alpha pour le cas #4 71
5.13 La valeur de la fonction objectif en fonction du temps de calcul de RLBS avec différentes valeurs de alpha pour le cas #5 72

Liste des algorithmes

1 Pseudo-code d'une itération de LDS 17
2 Pseudo-code d'une itération de ILDS 18
3 Pseudo-code de BLFS 33
4 Reinforcement Learning Backtracking Strategy 46
5 Heuristique de choix de variable processus 59
6 Heuristique de choix de valeur pour les quantités de processus 60

Je dédie ce mémoire à mes parents.
À ma mère, qui a œuvré pour ma réussite, de par son amour, son soutien, tous les sacrifices consentis et ses précieux conseils, pour toute son assistance et sa présence dans ma vie. Reçois à travers ce travail, aussi modeste soit-il, l'expression de mes sentiments et de mon éternelle gratitude.
À mon père, qui peut être fier et trouver ici le résultat de longues années de sacrifices et de privations pour m'aider à avancer dans la vie. Merci pour les valeurs nobles, l'éducation et le soutien permanent venu de toi.

Remerciements

Tout d'abord, j'aimerais remercier le département d'informatique et de génie logiciel pour la qualité de l'enseignement et du matériel académique dispensés. J'aimerais aussi remercier particulierement mon directeur de recherche, le professeur Jonathan Gaudreault, pour son soutien, son encadrement, sa patience, et ses conseils précieux qui m'ont aidé tout au long de ma maîtrise. Je tiens également à remercier mon co-directeur de recherche et directeur de programme, le professeur Brahim Chaib-draa, pour ses directives de qualité, ainsi que le professeur Claude-Guy Quimper pour son aide et ses contributions.

Chapitre 1

Introduction

Les problèmes d'optimisation se divisent en deux catégories : les problèmes avec des variables dont le domaine est continu, et les problèmes avec des variables dont le domaine est discret et que l'on appelle aussi problèmes *combinatoires* [Papadimitriou and Steiglitz, 1998]. La solution pour un problème continu est souvent représentée par un ensemble de nombres réels, tandis que pour un problème combinatoire on se retrouve plutôt à chercher un objet dans un ensemble fini. La solution pour un problème combinatoire se réduit donc à un ensemble de nombres entiers, une permutation, ou un graphe.

L'optimisation combinatoire consiste à chercher la solution à "moindre coût" d'un problème combinatoire. Elle est omniprésente dans l'industrie et on y a souvent recours pour résoudre des problèmes d'ordre opérationnel ou stratégique. Les compagnies aériennes s'en servent pour développer les meilleurs réseaux de transport aérien. Les compagnies de transport l'utilisent pour déterminer quel taxi est le mieux placé pour récupérer un client, dépendamment de plusieurs critères, comme l'emplacement actuel du client, sa destination ou encore l'état actuel des routes. Les compagnies d'expédition et de logistique l'emploient afin de trouver le chemin optimal pour livrer les colis, etc.

Une application directe de l'optimisation combinatoire dans le monde industriel concerne la planification des opérations. Les compagnies peuvent s'appuyer sur l'optimisation combinatoire pour planifier les différentes opérations qui supportent leur exercice et générer des plans de production optimisés. L'industrie du bois est un très bon exemple d'industrie où la planification joue un rôle crucial dans la performance, voire la pérennité, des entreprises qui y opèrent. La production du bois d'œuvre regorge de processus complexes, allant de la récolte des arbres en forêt, le transport liant les zones de ré-

colte aux usines de traitement, en passant par les opérations de sciage, de séchage, et de rabotage, jusqu'à la vente des produits finis à base de bois d'œuvre. Chacun de ces processus devrait être planifié rigoureusement afin de permettre un meilleur rendement.

La planification et l'ordonnancement des opérations est un problème combinatoire classique fort important pour les compagnies industrielles. Il s'agit d'un problème complexe et il représente un grand défi pour l'industrie. La vitesse à laquelle les entreprises sont capables de générer des plans de production de qualité, joue un rôle déterminant pour leur compétitivité sur le marché.

L'arrivée de nouvelles commandes pour des produits manufacturés obligent les entreprises à effectuer une planification de manière périodique. De plus, si un problème d'équipement ou une indisponibilité soudaine de la main-d'œuvre survient, l'entreprise doit être capable de replanifier ses opérations afin de relancer la chaîne de production. Cela peut prendre plusieurs jours à un ordinateur pour calculer la solution optimale. Dès lors, on se voit généralement obligé d'interrompre le calcul avant terme et de procéder avec une solution sous-optimale, c'est-à-dire la meilleure solution trouvée jusqu'à date. Ainsi, l'outil informatique est reconnu performant lorsqu'il est capable de trouver une bonne solution dans une durée de temps très courte.

Ce mémoire s'intéresse à l'optimisation combinatoire avec contraintes, et plus particulièrement à l'amélioration d'un algorithme de résolution, c'est-à-dire un algorithme qui permet de résoudre les problèmes d'optimisation combinatoire. De manière plus précise, ce projet tente de combiner l'optimisation combinatoire à l'apprentissage automatique pour trouver une solution en temps réel. Pour ce faire, des techniques d'apprentissage issues du domaine de l'intelligence artificielle, et qui relèvent plus particulièrement de l'apprentissage par renforcement, sont employées pour guider la résolution des problèmes combinatoires afin d'atteindre de meilleures performances.

Le contexte d'application de ce projet porte sur la production du bois d'œuvre, et plus particulièrement sur la planification et l'ordonnancement des opérations de finition du bois. La production du bois d'œuvre commence en forêt où les arbres sont tronçonnés en billots et transportés à l'usine pour être transformés. La transformation du bois d'œuvre passe par plusieurs étapes dont trois principales : le sciage, le séchage et le rabotage [Gaudreault et al., 2010]. Le processus de sciage transforme les billots en morceaux de bois nommés « sciages ». Une fois séchés à l'unité de séchage, les sciages passent au rabotage où ils sont transformés en différents produits. Chaque produit final est caractérisé par ses dimensions, qu'on appelle « famille », et sa longueur. D'autres

caractéristiques, se reliant à la qualité du bois, sont aussi prises en compte lors de la production.

Les entreprises opérant dans l'industrie du bois reçoivent des commandes de la part de leurs clients. C'est principalement l'élément déclencheur du cycle de production. Chaque commande peut exiger plusieurs types de produits, à des quantités différentes. Pour que les commandes soient satisfaites dans les délais prévus, toute la chaîne de production doit être bien planifiée. La production elle-même est supportée par plusieurs unités interdépendantes, et chacune est amenée à prendre diverses décisions pour accomplir ses tâches. Cette forte interdépendance rend la planification des opérations de production très complexe.

De plus, la production est soumise à plusieurs contraintes liées au processus de fabrication lui-même. Par exemple, la quantité maximale de sciages qui peut être introduite dans le séchoir, le délai de mise en course nécessaire pour les machines de rabotage lors du changement de famille, ou encore la capacité limitée des inventaires. Sous ces conditions, les entreprises se voient obligées de mettre en place des mécanismes d'optimisation qui appuient la planification de la chaîne de production, dans le but d'améliorer leur productivité.

L'approche développée dans ce mémoire consiste en une stratégie de retour-arrière qui se base sur des techniques d'apprentissage par renforcement pour guider l'exploration de l'espace de solutions vers les zones les plus prometteuses en premier. Autrement dit, cette stratégie de retour-arrière permet de concentrer les efforts de la résolution de problème sur les régions de l'espace de recherche jugées plus prometteuses, à savoir celles qui ont le plus de chances de contenir de bonnes solutions. Cette stratégie a été appliquée à des instances industrielles du problème de la planification du rabotage et s'est avérée plus performante que les stratégies auxquelles elle a été comparée.

Le reste du mémoire est organisé comme suit. Tout d'abord, on présente un ensemble de concepts préliminaires, dans les chapitres 2 et 3, et on décrit les travaux qui ont été réalisés dans le passé et qui combinent l'optimisation combinatoire et l'apprentissage. Puis, on décrit en détail l'approche proposée dans le chapitre 4. Enfin, dans le chapitre 5, on présente le problème auquel ce projet s'attaque, en soulignant son importance dans le monde industriel, ainsi que les différentes expérimentations accompagnées des résultats auxquels elles ont abouti. L'approche proposée apporte largement plus d'amélioration et dépasse les approches auxquelles elle a été comparée.

Chapitre 2

Programmation par contraintes

2.1 Définitions

Afin de définir la programmation par contraintes il faudrait tout d'abord expliquer ce qu'est une contrainte. Supposons une liste de variables $\mathcal{X} = \{X_1, \ldots, X_k\}$ avec $k > 0$, où chaque variable se voit associer un domaine $D \in \{D_1, \ldots, D_k\}$. Un *domaine*, dénoté $D_i = dom(X_i)$, représente l'ensemble des valeurs qu'une variable X_i peut prendre. Une *contrainte* représente une relation entre les valeurs des domaines d'un ensemble de variables Ω tel que $\Omega \subseteq \mathcal{X}$ [Apt, 2003]. Cet ensemble de variables Ω se nomme la *portée* de la contrainte, et la cardinalité de la portée s'appelle *l'arité*. Par exemple, une contrainte C_1 qui définit la relation $X_1 \neq X_2$ a une arité de 2 et l'ensemble $\Omega_1 = \{X_1, X_2\}$ constitue sa portée. Les valeurs qui satisfont la contrainte C_1 forment un sous-ensemble de $D_1 \times D_2$, où D_1 est le domaine de la variable X_1 et D_2 le domaine de la variable X_2.

Un *problème de satisfaction de contraintes* (*Constraint Satisfaction Problems – CSP*) est défini par un ensemble de variables $\mathcal{X}$, et un ensemble de contraintes $\mathcal{C}$. Il tient compte de toutes les combinaisons possibles entre les valeurs des domaines de chacune des variables. Une solution au problème est représentée par l'affectation d'une valeur $v_j \in dom(X_i)$ à une variable $X_i \in \mathcal{X}$, et ce pour toutes les variables du problème, de telle manière qu'aucune contrainte ne soit violée.

Un *problème d'optimisation avec contraintes* (*Constraint Optimization Problems – COP*) est un problème combinatoire [1] où l'on veut optimiser (maximiser ou minimiser) une *fonction objectif*. On se retrouve donc dans un contexte d'optimisation où il est non seulement question de satisfaire un ensemble de contraintes, mais aussi d'atteindre une

1. Un problème combinatoire est un problème dont l'espace de solution est discret.

solution dite *optimale*. Une solution optimale est une solution réalisable [2] qui maximise ou minimise une fonction objectif.

L'exemple suivant est une illustration d'un problème de satisfaction de contraintes ainsi qu'un problème d'optimisation par contraintes basiques.

Exemple 1. Soit le problème de satisfaction de contraintes suivant :

$$X \neq Y \tag{2.1}$$

$$X + Y = 10 \tag{2.2}$$

$$dom(X) = \{2, 4, 5, 7, 8\} \tag{2.3}$$

$$dom(Y) = \{2, 3, 5, 6, 8\} \tag{2.4}$$

Dans cet exemple, les contraintes sont représentées par les relations (2.1) et (2.2). L'ensemble des variables est $\{X, Y\}$. On trouve quatre solutions à ce problème : $\{X = 2, Y = 8\}$, $\{X = 8, Y = 2\}$, $\{X = 4, Y = 6\}$ et $\{X = 7, Y = 3\}$.

Supposons que l'on ajoute à ce problème la fonction objectif suivante :

$$\max \quad (2 \times X) + Y \tag{2.5}$$

Ce problème devient aussitôt un problème d'optimisation par contraintes et son espace de solution se réduit à une solution optimale, celle qui maximise $(2 \times X) + Y$. Cette solution optimale est $\{X = 8, Y = 2\}$.

2.2 Résolution de problèmes combinatoires

La résolution de problèmes combinatoires peut se réduire à une exploration globale d'un *arbre de recherche qui décrit l'espace des solutions*. Chaque nœud de l'arbre de recherche représente une affectation d'une valeur à une variable (la représentation de l'espace de recherche sous la forme d'un arbre est expliquée plus en détails dans le paragraphe suivant). Pour résoudre les problèmes combinatoires, il faut donc déterminer deux éléments essentiels : la manière de construire l'arbre de recherche, et la façon

2. Une solution réalisable satisfait toutes les contraintes.

FIGURE 2.1: La carte géographique de l'Australie (tiré de [Russell et al., 2010]).

d'explorer cet arbre. Autrement dit, il faut définir la topologie de l'arbre de recherche en fixant l'ordre d'instanciation des variables et l'ordre de sélection des valeurs, ainsi que la stratégie employée pour visiter les nœuds de l'arbre en quête de solution(s).

2.2.1 Arbres de recherche

Les problèmes combinatoires sont souvent résolus par une fouille dans un arbre [Russell et al., 2010]. Cette fouille explore un arbre de recherche qui décrit l'espace de solutions du problème. Chaque nœud de cet arbre correspond à une assignation partielle des variables du problème. Une assignation est dite partielle lorsqu'au moins une variable n'a pas encore été instanciée. À la racine de l'arbre, aucune variable n'est encore instanciée. Ceci correspond à une assignation dite vide. Chaque nœud de l'arbre correspond à une assignation partielle où une variable additionnelle est instanciée par rapport à son nœud parent. En d'autres mots, à chaque nœud on retrouve la même assignation partielle de son nœud parent à laquelle une nouvelle affectation a été ajoutée. Ainsi, le dernier niveau de l'arbre de recherche renferme des assignations dites complètes, où toutes les variables sont instanciées.

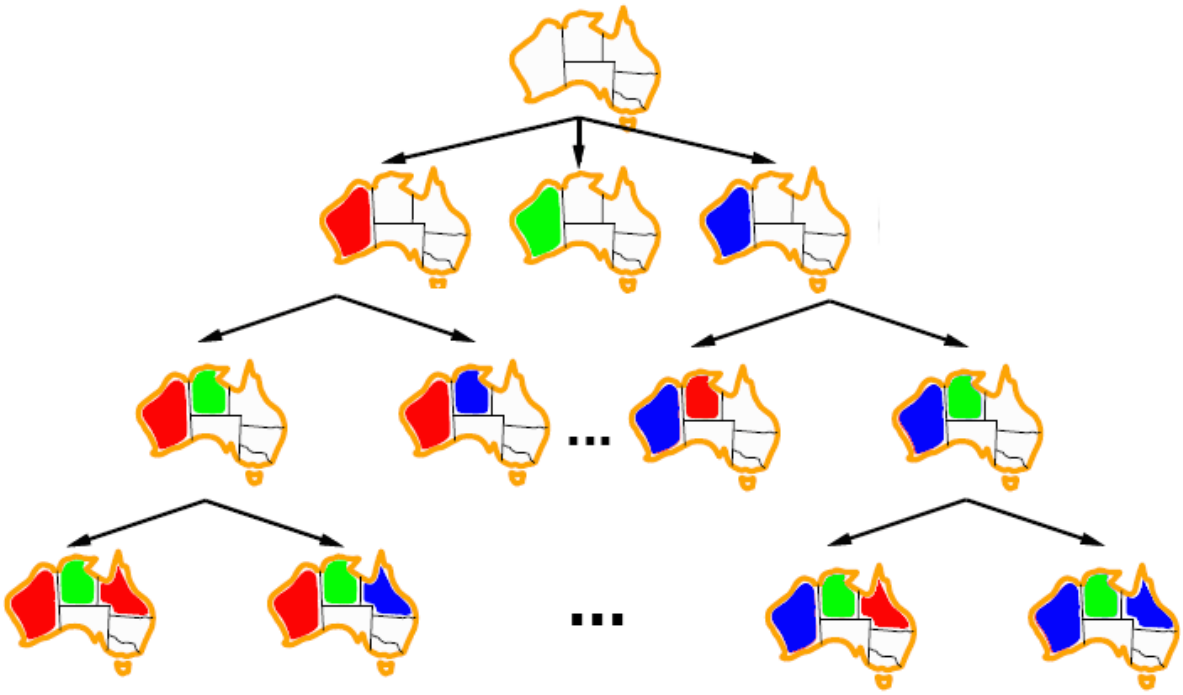

FIGURE 2.2: Une partie de l'arbre de recherche construit pour résoudre le problème de coloriage de la carte de l'Australie (inspiré de [Russell et al., 2010]).

Prenons l'exemple de coloriage d'une carte. Il s'agit d'un problème où il faut colorer chaque zone de la carte d'une couleur différente de toutes les couleurs des zones adjacentes. Ce problème peut être formulé comme un problème combinatoire et résolu en explorant un arbre de recherche. Pour illustrer ce problème, la carte de l'Australie est utilisée (voir figure 2.1). Les variables du modèle pour ce problème représentent les zones de la carte. L'ensemble de toutes les couleurs disponibles constitue les domaines des variables. Les contraintes du modèle sont des contraintes *Diff*[3] sur chaque paire de variables relatives à deux zones adjacentes.

- Variables : W, NT, S, Q, NS, V, T
- Domaines : $dom(X) = \{b, v, r\}$ avec $X = \{W, NT, S, Q, NS, V, T\}$
- Contraintes : $\mathcal{C} = \{W \neq NT, W \neq S, S \neq NT, Q \neq NT, S \neq Q, S \neq NS, Q \neq NS, NS \neq V\}$

Dans cet exemple, la génération de l'arbre de recherche commence par la racine. Il s'agit du premier nœud au sommet de l'arbre qui représente une assignation vide (aucune

3. La contrainte *Diff* s'assure que les variables de sa portée prennent des valeurs différentes

variable n'est instanciée). Chaque nœud fils de la racine représente une instanciation de la même variable, c'est-à-dire l'affection d'une valeur à cette variable. De la même manière, chacun de ces nœuds fils a un ensemble de nœuds fils qui représente l'affection des valeurs du domaine d'une autre variable. Ainsi, chaque niveau de l'arbre contient les nœuds qui représentent toutes les instanciations possibles d'une seule variable (chaque instanciation étant l'affection d'une valeur de son domaine).

Lorsqu'une instanciation viole une contrainte, la génération de l'arbre ne poursuit pas au niveau du nœud responsable de la violation de contrainte (le nœud qui représente l'instanciation en question). Autrement dit, les nœuds fils d'un nœud qui représente une instanciation qui viole une contrainte ne sont jamais inclus dans l'arbre de recherche. Les feuilles de l'arbre de recherche (les nœuds au dernier niveau) représentent les différentes solutions du problème, ainsi que les nœuds relatifs aux assignations qui violent une ou plusieurs contraintes. La figure 2.2 montre une partie de l'arbre de recherche construit pour résoudre le problème de coloriage de la carte de l'Australie [Russell et al., 2010].

La taille de l'arbre de recherche augmente exponentiellement en fonction du nombre de variables. En d'autres termes, pour un problème combinatoire de n variables, avec des domaines de taille d, la taille de l'arbre de recherche serait de taille d^n. Ainsi par exemple, dans le problème du coloriage de la carte d'Australie, on a 7 variables $X = \{W, NT, S, Q, NS, V, T\}$ donc $n = 7$, et des domaines $dom(X) = \{b, v, r\}$ de taille 3 alors $d = 3$. Pour ce problème, la taille de l'arbre de recherche sera donc égale à 3^7.

Pour un même problème CSP, plusieurs arbres de recherche alternatifs sont possibles. Chacun de ces arbres peut être visité selon plusieurs séquences possibles. La topologie de l'arbre de recherche est déterminée par l'ordre d'instanciation des variables, ainsi que l'ordre dans lequel les valeurs sont testées. Donc, la stratégie de choix de variables et la stratégie de choix de valeurs employées ont un rôle déterminant dans l'endroit où se trouvera la solution dans l'arbre de recherche. En d'autres termes, si on change l'ordre dans lequel on teste les valeurs on se retrouve avec un arbre totalement différent. Les figures 2.3 et 2.4 montre un exemple où les valeurs sont essayées dans deux ordres différents pour la variable W (r puis v puis b ; ensuite v puis r puis b). De la même manière, si l'ordre dans lequel les variables sont instanciées est modifié, la topologie de l'arbre de recherche change. Les figures 2.3 et 2.5 montre deux exemples chacun avec un ordre d'instanciation de variables différent. Dans la figure 2.3 la variable W est instanciée en premier, alors que dans la figure 2.5, c'est au contrainte la variable NT qui est instanciée en premier.

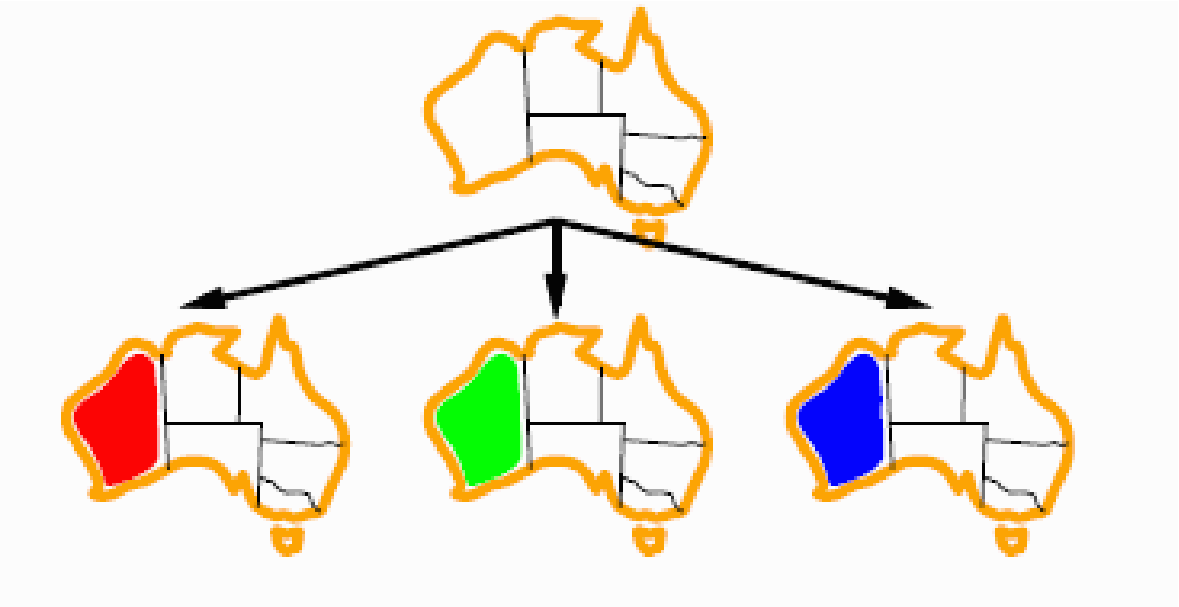

FIGURE 2.3: Premier niveau de l'arbre de recherche du problème de coloriage de la carte de l'Australie où la variable W prend les valeurs R, V et B dans l'ordre (inspiré de [Russell et al., 2010]).

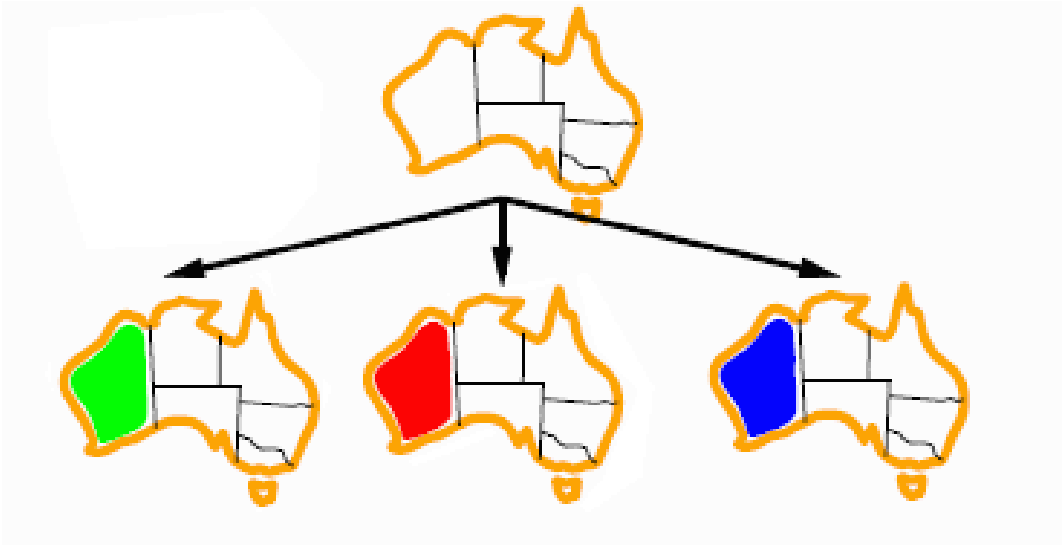

FIGURE 2.4: Premier niveau de l'arbre de recherche du problème de coloriage de la carte de l'Australie où la variable W prend les valeurs V, R et B dans l'ordre (inspiré de [Russell et al., 2010]).

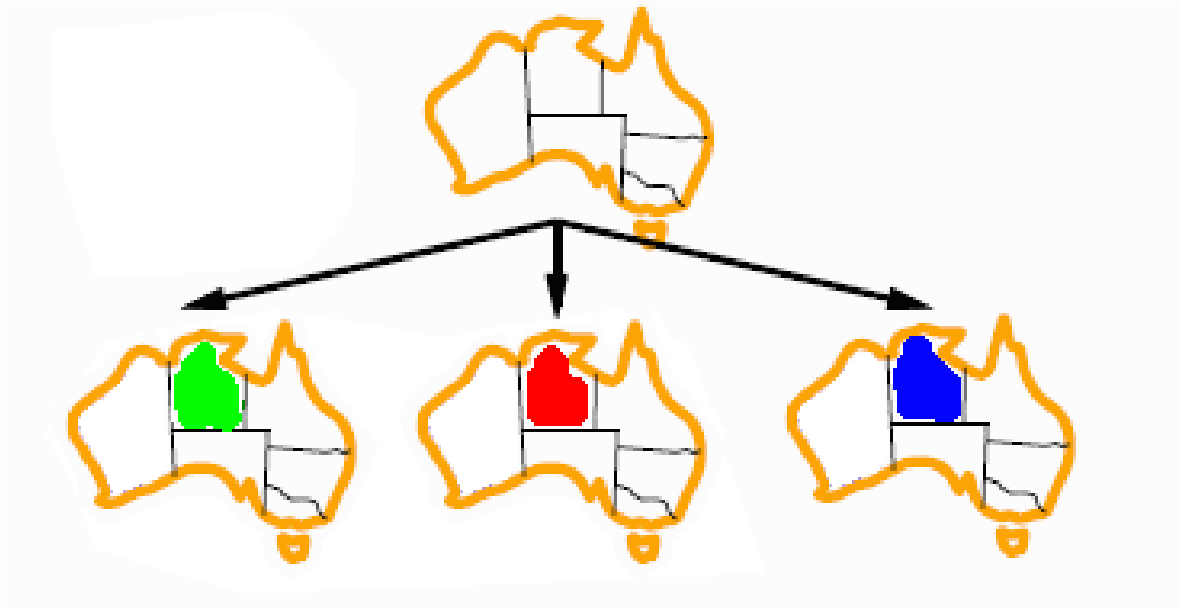

FIGURE 2.5: Premier niveau de l'arbre de recherche du problème de coloriage de la carte de l'Australie où la variable NT prend les valeurs V, R et B dans l'ordre (inspiré de [Russell et al., 2010]).

2.2.2 Stratégies de branchement

Comme nous l'avons expliqué précédemment, la topologie de l'arbre de recherche dépend entièrement de l'ordre d'instanciation des variables, ainsi que l'ordre dans lequel les valeurs sont testées. Par conséquent, le choix de variables et le choix de valeurs ont un impact direct sur l'endroit où se trouvera la solution optimale dans l'arbre de recherche. L'*heuristique de choix de variables* fixe l'ordre dans lequel les variables sont instanciées. L'*heuristique de choix de valeurs*, quant à elle, se charge de fixer l'ordre d'assignation des valeurs.

Principales stratégies de choix de variable

Il existe plusieurs heuristiques standards qui peuvent s'appliquer à n'importe quel problème combinatoire. *Minimum Remaining Value* (MRV) [Smith and Grant, 1997] est une heuristique de choix de variable (aussi appelée *Smallest Domain First*) qui propose de sélectionner, parmi les variables non assignées, celle ayant le moins de valeurs dans son domaine. Un tel choix se base sur le principe que moins on a de choix, plus on a des chances de bien choisir. Aussi, plus le domaine d'une variable est petit, plus il est difficile de l'instancier. Les variables avec le moins de valeurs dans leurs domaines auraient moins de sous-arbres ayant pour racine une de ces valeurs [Mouhoub and Jafari,

2011]. Autrement dit, une telle heuristique donne la priorité aux variables qui ont le plus de chance de provoquer un échec.

Most Constraining Variable (MCV) [Smith and Grant, 1997] est une autre heuristique de choix de variable qui peut s'appliquer à une variété de problèmes. Cette heuristique consiste à sélectionner la variable la plus contrainte. Rappelons que le graphe de contraintes est un graphe où les nœuds représentent les différentes variables du problème, et les arêtes représentent les contraintes. Par exemple, si une arête relie deux variables A et B dans le graphe de contraintes, ceci signifie qu'il existe une contrainte dans le modèle dont la portée est constituée des variables A et B. Le degré d'une variable dans le graphe de contraintes est le nombre d'arêtes auxquelles elle est connectée. MCV ordonne les variables par ordre décroissant de leur degré dans le graphe de contraintes [Dechter and Meiri, 1989]. Autrement dit, l'heuristique MVC choisit en premier la variable qui intervient dans le plus grand nombre de contraintes. Ceci est dicté par le fait que plus une variable intervient dans des contraintes plus elle a d'impact sur la solution.

Aussitôt qu'une variable intervient dans plusieurs contraintes elle devient difficile à instancier, car il faut lui affecter une valeur qui satisfait toutes les contraintes en même temps. De plus, l'instanciation d'une telle variable réduit instantanément la taille de l'arbre de recherche, parce que cette opération déclenche les algorithmes de filtrage de toutes les contraintes liées à la variable en question. Ceci permet de filtrer les domaines de toutes les variables des portées de ces contraintes et ainsi réduire la taille de l'arbre de recherche.

Principales stratégies de choix de valeur

Parmi les heuristiques standards, on trouve *Least Constraining Value* (LCV) [Sycara et al., 1991]. Il s'agit d'une heuristique de choix de valeur qui propose de choisir la valeur la moins contraignante. À chaque fois qu'une variable est instanciée, les algorithmes de filtrage des différentes contraintes filtrent les domaines de toutes les variables non instanciées. Ceci permet d'éliminer toutes les valeurs incohérentes avec l'instanciation nouvellement effectuée. l'heuristique LCV sélectionne la valeur qui entraîne le moins de changement dans les domaines des variables non instanciées. Comme l'exploration ne passe pas en revue toutes les valeurs possibles pour chaque variable avant de trouver la solution optimale, cette heuristique commence par les valeurs qui sont les moins probables de violer une contrainte (car elles déclenchent le moins de changement dans les domaines des variables non instanciées), c'est-à-dire, celles qui ont le plus de chance

de faire partie d'une solution.

Outre les heuristiques standards qui peuvent être employées pour n'importe quel problème combinatoire, il existe des stratégies de branchement développées sur mesure pour des problèmes plus spécifiques et qui, par conséquent, ne peuvent être employées que pour un certain type de problème.

2.2.3 Stratégies de retour-arrière

Un retour-arrière (*backtrack* en anglais) a lieu lorsqu'on atteint un nœud et qu'on est sûr qu'il ne peut mener vers une solution. Dans un contexte de CSP, un retour-arrière est déclenché quand l'assignation partielle du nœud courant ne peut être étendue à une assignation complète, à cause d'une violation de contrainte. Dans un contexte d'optimisation par contraintes (COP), en plus d'être déclenché à cause d'une violation de contrainte, le retour-arrière a également lieu lorsqu'une solution est trouvée. Autrement dit, lorsqu'une assignation complète est générée, un retour-arrière est réalisé pour chercher d'autres solutions et, ultimement, trouver la solution optimale. Le retour-arrière consiste à faire un branchement vers l'un des nœuds candidats au retour-arrière, et de poursuivre l'exploration de l'arbre à partir de ce nœud. Tout nœud dont on n'a pas encore explorer tous les fils est considéré comme un nœud candidat aux retours-arrière.

Vu que l'espace de recherche est représenté sous la forme d'un arbre, la résolution des problèmes combinatoires revient à faire une recherche dans un arbre. Plusieurs algorithmes, ou stratégies de retours-arrière, existent pour résoudre ce genre de problème, certains étant plus performants que d'autres. La recherche en profondeur d'abord (*Depth First Search* – DFS) [Russell et al., 2010] est la stratégie de recherche la plus simple et la plus connue. Elle procède à une fouille en profondeur de l'arbre de recherche. Elle commence à la racine qui devient le premier nœud courant. À partir de ce nœud, elle procède à la visite d'un nœud enfant, qui devient le nouveau nœud courant, jusqu'à atteindre une feuille de l'arbre ou un nœud dont l'assignation correspondante viole une ou plusieurs contraintes. Ensuite, la recherche remonte d'un niveau et se poursuit à partir du nœud parent, qui redevient le nœud courant. La recherche prend fin quand une solution est trouvée dans le cas d'un CSP, ou quand la solution optimale est trouvée, dans le cas d'un COP, ou lorsque l'arbre a été entièrement exploré.

Ainsi, DFS visite l'espace des solutions de gauche à droite, peu importe les caractéristiques du problème traité. Cette stratégie de recherche exécute des retours-arrière, dits *chronologiques*, en commençant par le nœud le plus récemment visité, comme illustré

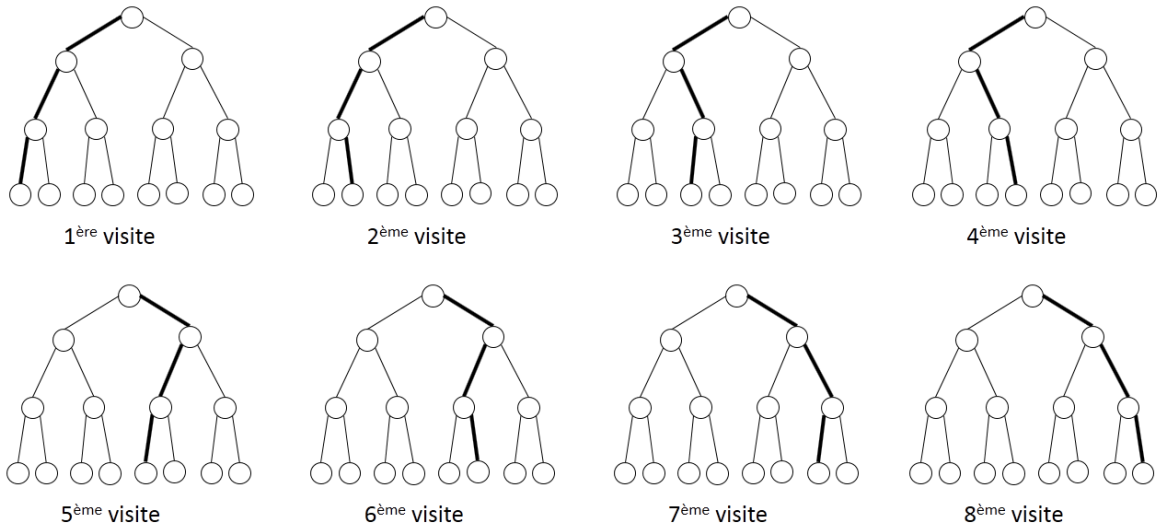

FIGURE 2.6: DFS sur un arbre binaire de profondeur 4.

en figure 2.6. Le fait qu'une solution soit à droite ou à gauche dans l'arbre de recherche repose entièrement sur l'ordre d'instanciation des variables et l'ordre dans lequel les valeurs sont testées. Autrement dit, la topologie de l'arbre de recherche détermine l'emplacement de toutes les solutions. Cependant, la grande majorité des problèmes réels sont complexes et leurs arbres de recherche sont tellement immenses qu'il est impossible de les explorer en entier. Il est alors inutile de s'attendre à ce qu'une stratégie telle que DFS trouve une solution optimale dans le cas d'un COP (ou une solution tout court dans le cas d'un CSP) pour ce genre de problème en un temps raisonnable.

Une autre manière d'effectuer les retours-arrière est de se baser sur une évaluation de nœud pour déterminer le meilleur candidat au retour-arrière. *Best-First Search* (BFS) [Vempaty et al., 1991] fait partie de cette catégorie de stratégies de retour-arrière. En fait, BFS affecte aussi la descente dans l'arbre de recherche. En d'autres termes, l'évaluation de nœuds ne se fait pas uniquement lors des retours-arrière, mais aussi pour décider du prochain nœud à explorer. BFS maintient deux listes de nœuds : une liste des nœuds ouverts (les nœuds dont les fils n'ont pas encore été tous visités) appelée OPEN, et une liste des nœuds fermés (les nœuds dont les fils ont été tous visités), appelée CLOSED [Kumar et al., 1988]. Tout au long de l'exploration de l'arbre de recherche, que ce soit lors des branchements ou au moment d'un retour-arrière, cet algorithme fait appel à un évaluateur de nœud qui sélectionne le meilleur nœud à explo-

rer parmi la liste des nœuds ouverts, d'où le nom *best-first*. Comme BFS doit maintenir les listes OPEN et CLOSED tout au long de l'exploration de l'espace de recherche, la mémoire consommée par ce dernier croît exponentiellement avec la taille du problème à résoudre.

En conclusion, l'exploration d'un arbre pour résoudre un problème combinatoire, nécessite de déterminer une heuristique de choix de variable, une heuristique de choix de valeur, et une stratégie de retour-arrière.

2.3 Algorithmes basés sur l'analyse des déviations

En réalité, rares sont les heuristiques de choix de variable et de choix de valeur qui ne se trompent jamais. En d'autres termes, il n'est pas commun que ces heuristiques ordonnent les variables et les valeurs de telle sorte à placer la solution optimale à l'extrémité gauche de l'arbre de recherche, ce qui permettrait de la trouver après la toute première descente. Les stratégies de branchement ne sont généralement pas parfaites (sinon, la première feuille de l'arbre renfermerait toujours la solution optimale, et le problème ne serait donc plus, par définition, combinatoire). Elles tentent de construire un arbre de recherche où les solutions sont situées de telle manière à les trouver plus rapidement mais elles commettent des erreurs. *Limited Discrepancy Search* (LDS) [Harvey and Ginsberg, 1995a] est un algorithme de recherche qui, justement, tolère les erreurs commises par l'heuristique de choix de valeurs.

Une déviation (*discrepancy* en anglais) est un branchement dans l'arbre de recherche où l'algorithme ne prend pas le chemin le plus favorisé par l'heuristique de choix de valeur [Barták, 2004]. Il s'agit d'une violation du choix préconisé par cette heuristique. Comme cette dernière ordonne les valeurs de gauche à droite, la valeur à l'extrémité gauche étant la meilleure, tout branchement vers un nœud autre que le fils gauche est considéré comme une déviation. Par exemple, dans la figure 2.7, la deuxième feuille à partir de la gauche contient une déviation (commise au dernier niveau de l'arbre) sur le chemin qui la sépare de la racine. De la même manière, pour accéder à la feuille à l'extrémité droite il faut commettre 3 déviations (3 branchements à droite).

LDS permet d'explorer les feuilles de l'arbre avec le plus petit nombre de déviations en premier. Cet algorithme consiste à faire d'abord une recherche en profondeur sans que plus de k choix de l'heuristique ne soient outrepassés, puis, si une solution n'a pas été trouvée, de relancer la recherche en incrémentant la valeur de k. Autrement dit, il

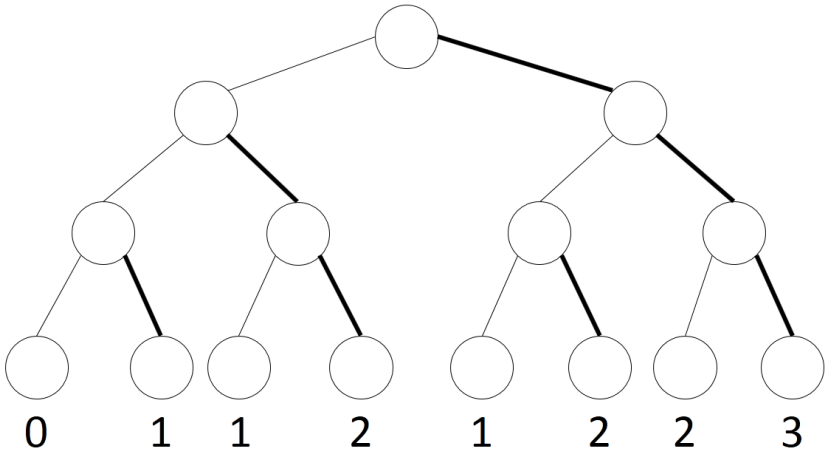

FIGURE 2.7: Nombre total de déviations associées à chaque feuille dans un arbre binaire.

s'agit d'effectuer plusieurs itérations de DFS à partir de la racine de l'arbre, où chaque itération limite l'exploration à une déviation de plus que l'itération précédente.

L'idée fondamentale présupposée par le fait de privilégier les chemins qui respectent le plus souvent l'heuristique est que c'est au bout de ces chemins que l'on devrait avoir le plus de chances de trouver une solution. En fait, ceci est uniquement vrai lorsque l'heuristique de choix de valeur respecte la condition suivante : la probabilité α que l'heuristique fasse le bon choix binaire est constante à tous les niveaux de l'arbre de recherche. Cette condition garantit qu'une feuille de profondeur n dont on a fait k déviations pour l'atteindre ait la probabilité $\alpha^{n-k}(1-\alpha)^k$ d'être la solution [Harvey, 1995]. LDS fait confiance à l'heuristique au début et suit tous ses choix, puis il s'en éloigne progressivement et tente d'autres alternatives. Ceci permet de visiter des branches se trouvant au milieu de l'arbre de recherche sans avoir à visiter toute la moitié de celui-ci (comme le ferait un DFS). Ainsi, LDS visite rapidement des feuilles qui sont davantage différentes les unes des autres que dans le cas d'un retour-arrière chronologique. Ceci est très intéressant quand une bonne heuristique de choix de valeur est employée. Au fait, lorsqu'une bonne heuristique de choix de valeur est utilisée, la probabilité qu'une feuille soit une bonne solution est inversement proportionnelle au nombre de déviations qui existent sur le chemin reliant cette feuille à la racine, à savoir le nombre de branchements à droite effectués à partir de la racine pour atteindre cette feuille. Il est donc plus raisonnable de visiter une feuille au milieu de l'arbre de recherche avec une seule déviation avant de visiter des feuilles se trouvant dans la partie gauche de l'arbre, mais avec plusieurs déviations. Le principe sur lequel se base donc LDS est de ne construire que les branches de l'arbre de recherche qui correspondent à des choix où le nombre

final de déviations est le plus petit possible.

LDS dans sa version originale est évalué sur un arbre binaire [Harvey and Ginsberg, 1995a]. À la première itération, il commence à la racine de l'arbre et effectue une descente en profondeur en branchant sur le fils de gauche jusqu'à la première feuille. Ensuite, il explore tous les chemins avec une seule déviation en commençant par le niveau le plus haut de l'arbre. Pour ce faire, LDS reprend à la racine de l'arbre et effectue une exploration en profondeur à partir de cette dernière et visite uniquement les feuilles dont le chemin vers la racine contient une seule déviation. LDS procède ainsi en autorisant une déviation de plus à chaque itération jusqu'à couverture de l'arbre au complet, comme illustré dans l'algorithme 1.

Algorithm 1 Pseudo-code d'une itération de LDS (tiré de [Korf, 1996])

```
1:  function LDS(node, k)
2:      if node is a leaf then
3:          return node
4:      end if
5:      leftChild ← LeftChild(node)
6:      ILDS(leftChild, k)
7:      if k > 0 then
8:          rightChild ← RightChild(node)
9:          ILDS(rightChild, k-1)
10:     end if
11: end function
```

En fait, en augmentant le nombre de déviations autorisées à chaque itération, LDS revisite plusieurs fois certains nœuds de l'arbre. En effet, les feuilles visitées à l'itération i seront toutes revisitées à l'itération $i + 1$ [Barták, 2004]. Dans le cas extrême, à la toute dernière itération, LDS génère l'arbre au complet même si le chemin de la racine vers la feuille à l'extrémité droite est le seul nouveau chemin à cette itération-ci [Korf, 1996]. *Improved Limited Discrepancy Search* (ILDS) [Korf, 1996] est alors proposé par Korf pour palier à cet inconvénient. En connaissant la profondeur de l'arbre, ILDS peut générer uniquement les chemins de la racine aux feuilles avec exactement un nombre donné k de déviations. Ceci est accompli en retenant la profondeur du sous-arbre en dessous de chaque nœud, et en forçant l'exploration à brancher à droite aussi longtemps que cette profondeur est inférieure ou égale au nombre de déviations k, comme illustré

dans l'algorithme 2. Le paramètre *depth* correspond à la profondeur du sous-arbre à explorer en dessous du nœud courant. Il est égal à la profondeur maximale de l'arbre à chaque appel à partir de la racine. De cette manière, ILDS explore uniquement les feuilles avec un certain nombre de déviations, évitant ainsi de visiter des feuilles de manière redondante. Ceci dit, les feuilles se trouvant à une profondeur au-delà de la profondeur maximum, ainsi que les nœuds internes de l'arbre, sont visités plus d'une seule fois [Korf, 1996]. Ainsi, même en évitant de visiter des feuilles de l'arbre plusieurs fois, ILDS est obligé de visiter des nœuds de manière redondante lors de l'exploration de l'arbre de recherche. Moisan et al. ont réalisé une analyse théorique pour déterminer la charge de travail (mesurée en nombre de visites de nœuds) de ILDS sur un arbre binaire de profondeur n et ont trouvé que ILDS effectue $2^{n+2} - n - 3$ visites de nœuds [Moisan et al., 2013]. Un arbre binaire de profondeur n est constitué de $2^{n+1} - 1$ nœuds. Comme $2^{n+2} - n - 3 > 2^{n+1} - 1$ pour $n \in \{2, 3, 4, \ldots\}$ on a la preuve que ILDS visite des nœuds plus d'une seule fois.

Algorithm 2 Pseudo-code d'une itération de ILDS (tiré de [Korf, 1996])

```
1:  function ILDS(node, k, depth)
2:      if node is a leaf then
3:          return node
4:      end if
5:      if depth > k then
6:          leftChild ← LeftChild(node)
7:          ILDS(leftChild, k, depth-1)
8:      end if
9:      if k > 0 then
10:         rightChild ← RightChild(node)
11:         ILDS(rightChild, k-1, depth-1)
12:     end if
13: end function
```

Dans la suite de ce mémoire, ILDS va être appelé LDS tout court, puisque c'est une version améliorée du même algorithme.

LDS peut aussi être implémenté avec un *évaluateur de nœud* [Beck and Perron, 2000]. À chaque fois qu'un retour-arrière est nécessaire, l'évaluateur de nœud sélectionne le nœud déjà visité pour lequel le prochain fils a le plus petit nombre total de déviations. Dans

cette optique, un autre algorithme basé sur l'analyse des déviations a été introduit par Beck and Perron. Cet algorithme se nomme *Discrepancy Bounded Depth First Search* (DBDFS) [Beck and Perron, 2000] et consiste à effectuer une recherche en profondeur vers toutes les feuilles dont les chemins de la racine contiennent un certain nombre borné de déviations. Étant donné une borne de déviation k, à l'$i$ème itération DBDFS visite toutes les feuilles avec des déviations entre $(i-1)k$ et $ik-1$ inclusivement. DBDFS emploie un évaluateur de nœuds (*node evaluator* en anglais) qui sélectionne les nœuds dont le nombre total de déviations respectent la borne associée à l'itération en cours.

Pour leur part, Prcovic and Neveu introduisent l'algorithme de recherche restreinte [Prcovic and Neveu] qui s'inspire de LDS pour limiter l'exploration de l'espace de recherche à chaque itération, et étendre cette limite au fur et à mesure que la résolution avance. Ces auteurs définissent le graphe de micro-structure comme étant le graphe dont les sommets représentent les différentes valeurs dans les domaines de toutes les variables et dont chaque arête représente la compatibilité de deux affectations de variables. Une solution peut être visualisée comme un ensemble de n nœuds dans ce graphe reliés par des arêtes, n étant le nombre de variables du problème. Tout sous-graphe du graphe de micro-structure peut être vu comme une solution potentielle. Ce concept de solutions potentielles permet d'examiner les compatibilités qu'a une affectation de variable avec chacune des affectations des autres variables. En dénombrant, pour chaque variable liée à un sommet $v_{i,j}$, les compatibilités et en faisant le produit sur toutes ces variables, on obtient le nombre de solutions potentielles relatif à $v_{i,j}$. On peut ainsi définir deux heuristiques, une pour les choix de valeurs et l'autre pour les choix de variables. L'heuristique de choix de valeurs sélectionne la valeur avec la plus grande potentialité. L'heuristique de choix de variable se base sur la somme des potentialités de toutes les valeurs d'un domaine pour sélectionner la variable avec la plus grande potentialité globale.

Le principe des solutions potentielles peut aussi être utilisé pour calculer la probabilité qu'une instanciation complète des variables soit une solution en sachant le potentiel de solutions lié à chaque affection. Dès qu'un ordre de variables est défini et que le potentiel relatif à chaque instanciation est calculé, on peut calculer l'estimation de la probabilité qu'une instanciation soit une solution.

De la même manière que LDS fixe un nombre de déviations à ne pas dépasser pour chaque itération, l'algorithme de recherche restreinte [Prcovic and Neveu] fixe un seuil de probabilité B en dessous duquel, les chemins ne menant qu'à des feuilles, dont l'esti-

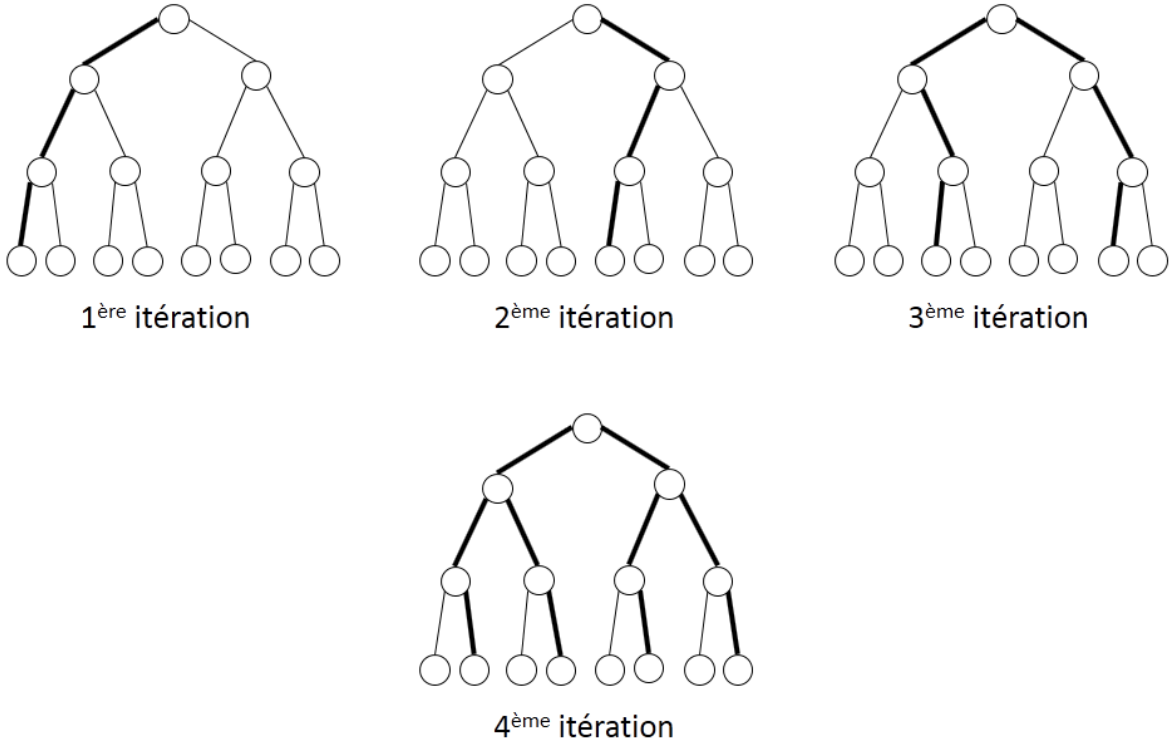

FIGURE 2.8: DDS sur un arbre binaire de profondeur 4 (inspiré de [Walsh, 1997]).

mation de la probabilité d'être une solution, est inférieure à B ne sont pas considérés. En d'autres termes, cet algorithme étudie la probabilité que chaque assignation fasse partie de la solution, et explore les nœuds qui ne dépassent pas la borne de probabilité B.

LDS traite, de la même manière, les feuilles trouvées en suivant des chemins ayant le même nombre de déviations, et ne fait aucune différence entre elles. Or, les feuilles accessibles via des chemins qui ont des déviations au sommet de l'arbre devraient être choisies en premier, car c'est en haut de l'arbre que les heuristiques sont le moins informées et, par conséquent, plus susceptibles de commettre des erreurs. *Depth-bounded Discrepancy Search* (DDS) [Walsh, 1997] a alors été proposé pour tirer profit de cette observation. Cet algorithme permet d'itérer sur les niveaux de l'arbre de recherche en autorisant les déviations à un seul niveau supplémentaire à chaque itération. Il commence par autoriser les déviations du premier niveau, qui se trouve au sommet de l'arbre, et descend les niveaux jusqu'à couverture de l'arbre au complet. Autrement dit, il visite tous les nœuds de l'arbre jusqu'à un niveau $k-1$, puis il autorise les déviations au niveau k, ensuite il suit fidèlement l'heuristique de choix de valeur pour tous les niveaux à partir du niveau $k+1$. À chaque itération, DDS incrémente la valeur de k et recommence la recherche à partir de la racine. DDS se force à commettre

une déviation au niveau k et ne suit pas l'heuristique à ce niveau-ci, pour éviter de revisiter les mêmes branches déjà explorées lors des itérations précédentes. En fait, lors de l'itération k, l'algorithme ne visite que les branches favorisées par l'heuristique au niveau $k + 1$. Par conséquent, ces dernières n'ont pas besoin d'être visitées dans les itérations futures (Figure 2.8).

DDS permet de distinguer les feuilles qui ont le même nombre de déviations selon le niveau de profondeur de l'occurrence de celles-ci. Ainsi, les feuilles avec des déviations en haut de l'arbre sont visitées en premier. Aussi, grâce à cet algorithme, il devient beaucoup plus facile de garantir qu'une branche de l'arbre ne soit pas revisitée, en forçant une déviation à un certain niveau. De plus, DDS non seulement visite toutes les feuilles de l'arbre de recherche mais garantit qu'elles ne soient pas visitées plus d'une seule fois durant toute l'exploration. Quant au nombre total de visites de nœuds, pour un arbre binaire de profondeur n, DDS effectue $2^{n+2} - n - 3$ visites de nœuds [Moisan et al., 2014]. On remarque qu'il s'agit du même nombre de visites de nœuds qu'effectuerait LDS sur un tel arbre [Moisan et al., 2013].

Chapitre 3

Apprentissage par renforcement

L'idée d'apprendre par interaction avec l'environnement est probablement la première qui nous vient à l'esprit quand nous pensons à la nature de l'apprentissage. Interagir avec l'environnement permet aux individus d'exercer leur connexion avec ce dernier, ce qui produit une richesse en information sur les causes et effets, sur les conséquences des actions, et sur ce qu'il faut faire pour atteindre ses objectifs. Les interactions avec l'environnement représentent sans doute une source majeure de connaissances et apprendre à travers ces interactions constitue l'idée fondamentale derrière toutes les théories d'apprentissage et d'intelligence.

3.1 Définitions

L'apprentissage par renforcement (*Reinforcement Learning – RL*) fait partie de la catégorie de l'apprentissage automatique. Il consiste à apprendre comment associer des actions à des situations de façon à maximiser une récompense numérique. L'apprenti, appelé agent d'apprentissage par renforcement ou agent tout court, ne sait pas quelles actions prendre, comme c'est le cas dans la plupart des techniques d'apprentissage automatique. Il doit apprendre quelles actions choisir dans une situation donnée afin d'atteindre un ou plusieurs buts. Le processus d'apprentissage se fait à travers les interactions avec l'environnement (Figure 3.1) et, par conséquent, il est différent des techniques d'apprentissage supervisé qui requièrent les connaissances d'un expert.

L'environnement qui supporte l'apprentissage par renforcement est typiquement formulé sous forme d'un processus décisionnel de Markov (*Markov Decision Prossess* – MDP). Un modèle de MDP se compose d'un ensemble d'états, un ensemble d'actions, une fonction de transition et une fonction de récompense. La fonction de transition

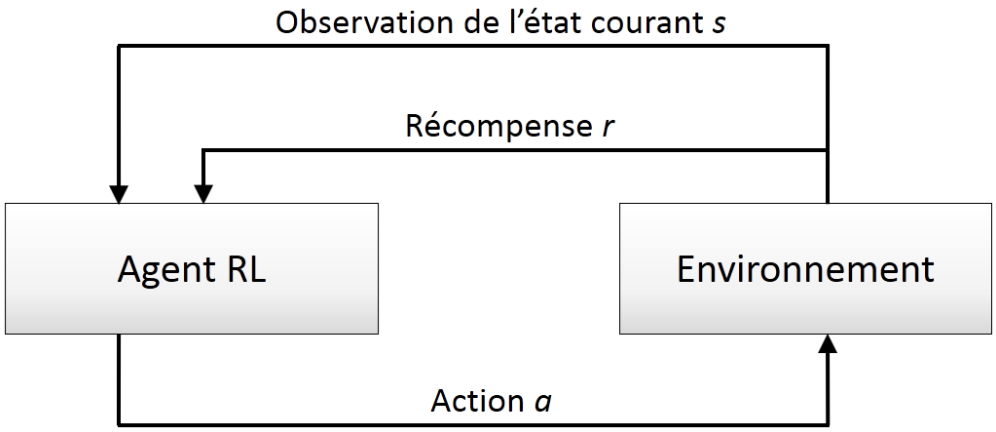

FIGURE 3.1: Le modèle basique de l'apprentissage par renforcement (inspiré de [Alpaydin, 2004]).

$T : S \times A \times S \rightarrow [0; 1]$ décrit toutes les transitions possibles du modèle. Plus précisément, elle retourne la probabilité que le système se retrouve dans un état $s' \in S$ en allant d'un état $s \in S$, et en ayant choisi une action $a \in A$. La fonction de récompense $R : S \times A \times S \rightarrow \mathbb{R}$ retourne une valeur réelle lorsque le système passe d'un état $s \in S$ à un état $s' \in S$ après avoir effectué une action $a \in A$.

Dans cet environnement, l'agent est censé découvrir quelles actions mènent vers les plus grandes récompenses en les essayant. À chaque étape du temps, l'agent observe l'état courant $s \in S$. Dans chaque état s, l'agent doit choisir une action $a \in A$ parmi un ensemble d'actions disponibles à partir de cet état. Une action a peut entraîner une transition de l'état s à un autre état $s' \in S$, selon une fonction de transition probabiliste. Cette fonction représente les différentes probabilités des transitions possibles. Une récompense numérique (*reward* en anglais) r est retournée à l'agent à chaque fois qu'il exécute une action. Cette récompense informe l'agent de la désirabilité d'un état. La plupart du temps, les actions affectent non seulement les récompenses immédiates que l'agent reçoit tout de suite après avoir accompli l'action, mais aussi les récompenses futures.

L'objectif de l'agent est d'apprendre la politique optimale qui lui permettrait de maximiser l'espérance de la récompense cumulée. Une politique (*policy* en anglais) $\pi : S \rightarrow A$ associe les actions aux états. $\pi(s)$ représente l'action que l'agent doit prendre quand il est dans l'état s, selon la politique π. La récompense cumulative peut être exprimée de deux façons. Soit elle prend la forme d'une somme de toutes les récompenses récoltées comme dans l'équation (3.1). Soit elle est représentée par la somme des récompenses

dévaluées comme dans l'équation (3.2).

$$R = r_0 + r_1 + \cdots + r_n \tag{3.1}$$

$$R = \sum_t \gamma^t r_t \tag{3.2}$$

où γ est le facteur de dévaluation.

Pour un horizon infini, le facteur de dévaluation (*discount factor* en anglais) est là pour refléter une dévaluation des récompenses futures. Ce facteur, compris entre 0 et 1, permet de prendre en compte les récompenses plus ou moins loin dans le futur pour le choix des actions de l'agent.

Dans le contexte d'apprentissage par renforcement, dans tout état s, une valeur numérique $Q(s, a)$ peut être attribuée à chaque actions a. Cette valeur est appelée Q-Valeur (Q-*Value* en anglais) et elle représente la désirabilité de prendre l'action a à l'état s. Plus la Q-Valeur est grande, plus l'action associée a des chances de mener vers de bonnes récompenses, selon le "jugement" de l'agent. Chaque fois qu'une récompense est récoltée, l'agent met à jour la Q-Valeur de l'action qui a mené à cette récompense. Cependant, l'ancienne Q-Valeur ne doit pas être complètement écrasée. Sinon, l'agent se retrouverait à baser toutes ses décisions sur la toute dernière expérience uniquement. Le plus approprié serait de garder une partie de l'ancienne Q-Valeur et la mettre à jour avec une partie de la nouvelle Q-Valeur, pour conserver toute l'information collectée durant l'exploration. Lors de la mise à jour de la Q-Valeur, on suppose que l'agent va agir de manière optimale dans le futur. Supposons que s soit l'état courant, s' l'état suivant, a l'action choisie par l'agent et r la récompense qui sera retournée par l'environnement une fois que la transition aura été complétée. La formule de mise à jour de la Q-Valeur sera comme suit :

$$Q_{t+1}(s, a) \leftarrow (1 - \alpha)\, Q_t(s, a) + \alpha \left[r_{t+1} + \gamma \max_{a'} Q_t(s', a') \right] \tag{3.3}$$

où α est le taux d'apprentissage et γ le facteur de dévaluation.

Cette formule (3.3) est à la base de l'apprentissage utilisant la Q-Valeur (Q-*Learning* en anglais). L'apprentissage via la Q-Valeur est une technique d'apprentissage par renforcement qui consiste à trouver la politique optimale en se basant sur les Q-Valeurs. Elle permet d'apprendre la fonction Q qui reflète la récompense cumulative espérée après

avoir pris chaque action, à partir de chaque état. En d'autres termes, l'idée dernière l'apprentissage via la Q-Valeur est d'apprendre une fonction qui prédit la récompense associée à chaque action, dans chacun des états.

3.2 Exemple : machine à sous

Cette sous section va présenter un problème de prise de décision séquentielle qui peut être résolu par l'apprentissage par renforcement. Il s'agit du problème de la "Machine à sous à k bras" [Alpaydin, 2004]. À chaque étape, un nombre fini k d'actions possibles sont disponibles. Chaque action consiste à choisir de tirer sur l'un des bras de la machine à sous, et gagner un certain montant d'argent associé au bras choisi. Donc le choix d'une action mène vers une observation quantifiée par une récompense sous la forme d'une valeur numérique. L'objectif est de décider quels bras tirer pour maximiser l'argent gagné. En d'autres termes, il faut déterminer les actions de telle sorte à maximiser la récompense totale. Il s'agit d'un problème simple d'apprentissage par renforcement car il est mono-état. L'agent n'est pas obligé d'exécuter une séquence d'actions, chacune menant vers un état différent, pour avoir une récompense. Chaque action engendre une récompense de façon immédiate et complète. En d'autres termes, on a affaire à un seul état et il faut uniquement décider des actions à effectuer. Une autre raison pour laquelle ce problème est considéré comme simple est les récompenses immédiates. Les récompenses sont retournées immédiatement après chaque action, contrairement à d'autres problèmes où il peut y avoir un délai entre l'exécution de l'action et le retour de la récompense qui lui est associée.

Les observations sont une source d'information utile pour améliorer la prise de décision future, dans le sens où elles reflètent à quel point chaque action est rentable. L'objectif ultime est de maximiser la somme des récompenses générées. Ceci dit, les observations utilisées pour acquérir l'information sont aussi des sortes de récompenses.

Comme on a un seul état, la valeur $Q(s, a)$ se réduit à $Q(a)$, la valeur de l'action a. Initialement, $Q(a) = 0$ pour toutes les actions a. Quand on essaye une action a, une récompense $r_a > 0$ est retournée. Si les récompenses sont déterministes, la récompense r_a sera toujours retournée après avoir effectué l'action a. Dans ce cas, on peut simplement établir que $Q(a) = r_a$. Si on veut exploiter, une fois qu'une action telle que $Q(a) > 0$ est trouvée, on peut se contenter de la choisir à chaque fois et récolter la récompense r_a. Cependant, il n'y a aucune garantie qu'il n'existe pas une autre action avec une meilleure récompense, à savoir une action a' telle que $Q(a') > Q(a)$. Donc, on

doit explorer.

On peut choisir différentes actions et enregistrer les $Q(a)$ pour toutes les actions. Au moment de l'exploration, l'action avec la plus grande valeur est choisie selon la formule suivante :

$$\text{choisir } a^* \quad \text{si } Q(a^*) = \max_a Q(a) \tag{3.4}$$

Si les récompense sont stochastiques, une récompense différente peut être retournée à chaque fois qu'une même action est effectuée. La valeur de la récompense associée à cette action est donc définie par la probabilité $p(r|a)$ de recevoir la récompense r en ayant choisi l'action a. Dans ce cas, on définit $Q_t(a)$ comme l'estimé de la valeur de l'action a à l'instant t. Il s'agit d'une moyenne de toutes les récompenses qui ont été générées après avoir choisi l'action a avant l'instant t. La formule de mise à jour est définie comme suit :

$$Q_{t+1}(a) \leftarrow (1 - \alpha)\, Q_t(a) + \alpha\, r_{t+1}(a) \tag{3.5}$$

où $r_{t+1}(a)$ est la récompense reçue après avoir pris l'action a au moment $t + 1$.

Le problème de la "Machine à sous à k bras" est un très bon exemple simplifié d'un problème d'apprentissage par renforcement. Il peut être qualifié de simplifié pour plusieurs raisons. Premièrement, il est mono-état. Avoir plusieurs états pour ce problème correspondrait à avoir plusieurs machines à sous avec des distributions de probabilités différentes $p(r|s_i, a_j)$. Dans ce cas, on devrait apprendre $Q(s_i, a_j)$ qui correspond à la valeur associée à l'action a_j dans l'état s_i. Deuxièmement, les actions affectent uniquement les récompenses. Dans le cas contraire, les actions affecteraient aussi l'état prochain. Autrement dit, effectuer une action changerait l'état (se déplacer vers une autre machine) en plus de retourner une récompense. Troisièmement, les récompenses sont immédiates. Dans le cas des récompenses différées, il faudrait être capable d'estimer des valeurs immédiates à partir des récompenses différées.

L'un des principaux problèmes de l'apprentissage par renforcement est d'équilibrer l'exploration et l'exploitation. En d'autres termes, il faudrait savoir quand est-ce qu'il faut exploiter l'expérience déjà acquise dans le passé, et quand est-ce qu'il faut explorer des zones encore inconnues de l'environnement afin d'acquérir de nouvelles expériences. Afin d'obtenir plusieurs récompenses, un agent doit privilégier des actions qu'il avait

déjà essayées dans le passé et qui s'étaient avérées rentables en termes de récompense. Cependant, pour découvrir ces actions-ci, il doit avoir choisi des actions qu'il n'avait pas encore essayées et qui étaient totalement inconnues pour lui. L'agent doit alors exploiter ce qu'il connait déjà pour obtenir des récompenses, mais doit également explorer dans le but de mieux évaluer les actions dans le future. Le dilemme est que ni l'exploitation ni l'exploration ne peut être considérée de manière exclusive sans que ceci mène à un échec. L'agent doit essayer une variété d'actions et progressivement favoriser celles qui semblent rapporter le plus de récompenses. Dans un contexte stochastique, chaque action doit être essayée plusieurs fois pour obtenir une estimation fiable de la récompense espérée.

Un autre problème important que l'on rencontre dans l'apprentissage par renforcement est celui de l'attribution du mérite, où l'on a affaire à des récompenses différées. Les récompenses sont généralement générées après avoir exécuté une séquence d'actions. Or, il se peut que quelques actions dans cette séquence contribuent plus que d'autres dans la récompense retournée. Ainsi, il est dur de déterminer à quelle action parmi toutes les actions passées l'agent doit chaque récompense.

3.3 Algorithmes de résolution des problèmes combinatoires basés sur l'apprentissage

Les problèmes à base de contraintes comme les problèmes d'optimisation par contraintes sont des problèmes combinatoires difficiles. Ces problèmes ont généralement un espace de solutions très grand. La recherche de solutions se fait à travers l'exploration globale de l'arbre de recherche. Jusqu'à maintenant, on s'est plus concentrée sur les méthodes d'exploration globale, mais il existe aussi des méthodes utilisant la recherche locale [1].

Dans les problèmes d'optimisation combinatoire, les retours-arrière ont lieu quand une assignation viole une contrainte ou bien lorsqu'on trouve une solution (dans l'espoir d'en trouver une meilleure). Il arrive souvent qu'on soit amené à faire des retours-arrière vers le même nœud à plusieurs reprises. Si on constate que la solution trouvée à partir du nœud en question ne s'améliore pas, il serait plus judicieux d'ignorer ce dernier au moment des retours-arrière futurs, et explorer un autre nœud potentiellement plus intéressant. En effet, on pourrait brancher vers un nœud plus prometteur, ou un nœud

1. La recherche locale consiste à démarrer la recherche à partir d'une solution initiale, et essayer de l'améliorer en vue d'obtenir la solution optimale. Ceci dit, les techniques de recherche locale sont très sensibles aux minima locaux.

totalement inconnu, en vue d'explorer d'autres portions de l'espace de recherche. On pourrait, dès lors, utiliser l'apprentissage automatique qui servirait à guider la recherche vers les zones de l'espace de recherche les plus prometteuses.

Peu importe le type d'exploration employé (global ou local), il existe deux manières de conduire l'apprentissage : soit de manière en ligne (*online* en anglais), soit hors ligne (*offline* en anglais). L'apprentissage en ligne consiste à apprendre au fur et à mesure que l'exploration progresse sur la même instance de problème, tandis que l'apprentissage hors ligne signifie que l'on apprend sur plusieurs instances d'un problème pour ensuite utiliser les connaissances apprises dans la résolution d'une autre instance du même problème. Les travaux menés sur la résolution de problèmes combinatoires qui emploient l'apprentissage automatique peuvent être structurés en 4 catégories, selon les types de l'exploration et de l'apprentissage utilisés (exploration globale et apprentissage en ligne, exploration globale et apprentissage hors ligne, exploration locale et apprentissage en ligne, et exploration locale et apprentissage hors ligne).

Plusieurs travaux ont tenté d'intégrer des techniques d'apprentissage automatique dans la résolution des problèmes combinatoires [Xu et al., 2008, Epstein and Petrovic, 2007, Samulowitz and Memisevic, 2007]. Ceci a pour but de rendre les heuristiques et les stratégies de recherches plus intelligentes. Celles-ci peuvent alors adapter leur comportement en fonction de la nature du problème à résoudre. Ainsi, les heuristiques et les stratégies de recherche apprennent la structure de l'espace de recherche pour permettre une exploration plus efficace. Ceci dans le but de favoriser les zones les plus prometteuses et, par conséquent, augmenter les chances de trouver la solution optimale plus rapidement.

3.3.1 Exploration globale et apprentissage en ligne

Impact-Based Search (IBS) [Refalo, 2004] propose une approche de résolution qui se base sur l'impact, de chaque affection d'une valeur à une variable, sur la réduction de l'espace de recherche. Cet impact est appris à partir de l'observation de la réduction des domaines durant l'exploration. Cette réduction est due à la propagation des contraintes, menée par leurs algorithmes de filtrage, qui a lieu après chaque instanciation d'une variable. La taille de l'arbre de recherche peut être estimée en calculant la taille du produit cartésien des domaines des variables, comme suit :

$$P = |D_{x_1}| \times |D_{x_2}| \times \cdots \times |D_{x_n}| \tag{3.6}$$

où D_{x_i} représente le domaine de la variable x_i.

En comparant l'estimation de la taille de l'espace de recherche avant (P_{before}) et après (P_{after}) une assignation $x_i = a$ on peut avoir une estimation de la réduction de la taille de l'espace de recherche engendrée par cette assignation. L'impact d'une assignation peut alors être calculé en se basant sur cette réduction, comme illustré dans l'équation (3.7). Il est clair que plus la réduction de l'espace de recherche est importante, plus l'impact est grand. On peut déduire aussi qu'une assignation qui échoue (viole une contrainte) engendre un impact égal à 1. Une assignation qui ne réduit pas la taille de l'espace de recherche ($P_{after} = P_{before}$) entraîne un impact égal à 0.

$$I(x_i = a) = 1 - \frac{P_{after}}{P_{before}} \tag{3.7}$$

L'impact des assignations peut être calculé pour chaque valeur des domaines de toutes les variables non instanciées, mais ceci peut s'avérer très coûteux [Refalo, 2004]. Une alternative serait de calculer, pour chaque assignation, la moyenne des impacts de celle-ci observés jusqu'à date. Soit K l'ensemble des impacts observés jusqu'à date pour une assignation donnée $x_i = a$. L'impact moyen $\bar{I}$ est calculé selon l'équation suivante :

$$\bar{I}(x_i = a) = \frac{\sum_{k \in K} I_k(x_i = a)}{|K|} \tag{3.8}$$

L'impact d'une variable x_i peut être défini par la moyenne des impacts $\bar{I}(x_i = a)$ des valeurs restantes dans le domaine de x_i. Donc, si D'_{x_i} est le domaine de la variable x_i dans son état actuel, l'impact de x_i peut se calculer comme suit :

$$\bar{\bar{I}}(x_i) = \frac{\sum_{a \in D'_{x_i}} \bar{I}(x_i = a)}{|D'_{x_i}|} \tag{3.9}$$

L'approche basée sur le calcul des impacts suppose que le problème est insoluble (il n'a pas de solution) [Refalo, 2004]. Étant donné cette hypothèse, l'objectif est de générer l'arbre de recherche le plus petit possible pour prouver que le problème ne peut pas être résolu. Compte tenu de cette observation, le calcul des impacts selon l'équation (3.9) n'est pas suffisamment précis. On suppose que toutes les valeurs du domaine d'une variable vont être essayées et le but est de déterminer la variable qui aura le plus grand impact quand on lui affecte l'une des valeurs restantes dans son domaine. Par conséquent, pour chaque variable, on doit considérer la réduction de l'espace de

recherche si toutes les valeurs du domaine sont essayées. Soit P la cardinalité du produit cartésien de tous les domaines (voir l'équation (3.6)) et soit x_i une variable avec D'_{x_i} son domaine à son état actuel. L'estimation de la taille de l'espace de recherche après l'assignation $x_i = a_j$ avec $a_j \in D'_{x_i}$ est définie comme suit :

$$P \times (1 - \bar{I}(x_i = a)) \tag{3.10}$$

Donc l'estimation de la taille de l'espace de recherche après avoir essayé d'affecter à la variable x_i toutes les valeurs $a \in D'_{x_i}$ de son domaine serait la somme des estimations pour chacune de ces valeurs, comme illustré dans l'équation suivante :

$$\sum_{a \in D'_{x_i}} P \times (1 - \bar{I}(x_i = a)) \tag{3.11}$$

Comme la valeur de P est constante à chaque nœud, l'impact d'une variable dépend uniquement de son domaine à son état courant, comme illustré dans l'équation qui suit :

$$\mathcal{I}(x_i) = \sum_{a \in D'_{x_i}} (1 - \bar{I}(x_i = a)) \tag{3.12}$$

Impact-Based Search (IBS) s'appuie sur le calcul des impacts pour mieux diriger l'exploration de l'arbre de recherche, et ce, peu importe la formule de l'impact employée [2] (l'impact moyen $\bar{\bar{I}}(x)$ ou la somme des impacts $\mathcal{I}(x)$). Pour ce faire, à chaque fois qu'une variable doit être instanciée, IBS calcule les impacts de toutes les variables et sélectionne celle avec le plus grand impact. Ensuite, il poursuit l'exploration de l'arbre de la même manière à partir de la variable sélectionnée. Donc, IBS apprend quelles variables ont le plus de chances de réduire l'espace de recherche après être instanciées [3] et tente de les instancier en premier. En d'autres termes, il tente de détecter les nœuds de l'arbre de recherche qui sont le plus probables d'être au sommet du plus petit sous-arbre possible, et les privilégie lors de l'exploration de l'arbre. Ainsi, IBS permet de (1) résoudre un problème rapidement ou (2) prouver qu'il n'a pas de solution en générant un petit arbre de recherche [Refalo, 2004].

2. La somme des impacts est en général plus efficace que l'impact moyen [Refalo, 2004].

3. On rappelle que la réduction de l'espace de recherche se fait suite à la propagation des contraintes qui suit chaque instanciation et qui filtre les domaines des variables non instanciées.

Les approches où le système apprend à évaluer la qualité des nœuds sont d'un grand intérêt quand il s'agit de stratégie de retour-arrière. Ruml [Ruml, 2002a] propose une approche intéressante à cet égard. *Best Leaf First Search* (BLFS) [Ruml, 2002a] utilise une régression linéaire afin de prédire le coût de chaque solution du problème à résoudre. L'idée derrière BLFS est de générer en premier les solutions ayant le plus petit coût. Pour ce faire, il génère uniquement une partie de l'arbre de recherche qui contient les feuilles ayant un coût inférieur à une certaine borne *bound*, et explore cet arbre en utilisant un DFS. À chaque itération, la borne *bound* est augmentée et une partie plus grande de l'arbre de recherche est explorée, comme illustré dans l'algorithme 3. En fait, la borne est augmentée de sorte à ce que le nombre de nœuds visités à une itération soit le double du nombre de nœuds visités lors de l'itération précédente (lignes 5 et 6 de l'algorithme 3). À chaque itération, BLFS visite les mêmes nœuds explorés lors de l'itération précédente, plus les nouveaux nœuds autorisés par l'augmentation de la borne *bound*. L'objectif est de s'assurer de visiter un nombre suffisant de nouveaux nœuds à chaque itération pour que ceci vaille le coup de revisiter les nœuds précédemment explorés. Il se trouve que doubler la taille de l'arbre à explorer à chaque itération soit la meilleure façon de procéder [Ruml, 2002a].

Le modèle employé par BLFS ne définit pas vraiment une stratégie de retour-arrière en tant que telle. Au lieu de ceci, BLFS effectue des descentes successives dans des portions de l'arbre de recherche en utilisant DFS. Cet algorithme a abouti à de très bons résultats [Ruml, 2002b], et a inspiré l'algorithme *Adaptive Discrepancy Search* (ADS).

Adaptive Discrepancy Search (ADS) [Gaudreault et al., 2012] est une stratégie de retour-arrière qui apprend au fur et à mesure que l'exploration de l'arbre de recherche progresse. Quand un nœud n est sélectionné pour un retour-arrière, cet algorithme exécute une descente en profondeur dans l'arbre de recherche à partir de son prochain fils non visité jusqu'à rencontrer une feuille. Pour chacun des nœuds m qui se trouvent entre cette feuille et le nœud n, incluant ce dernier, ADS attribue une note qui dépend de (1) la qualité de la solution nouvellement trouvée après cette dernière descente, et (2) des qualités des solutions trouvées précédemment à partir de ce nœud m. Pour chaque nœud, cette note reflète l'amélioration estimée de la qualité de la meilleure solution globale trouvée jusqu'à date qui serait amenée par un retour-arrière vers ce nœud. Un nœud avec une note importante est considéré comme étant prometteur et comme ayant de grandes chances d'améliorer la meilleure solution globale. Pour ce faire, ADS réalise une extrapolation sur le vecteur qui décrit l'historique des qualités des solutions

Algorithm 3 Pseudo-code de BLFS (Tiré de [Ruml, 2002a])

```
1: function BLFS(root)
2:     Visiter quelques feuilles
3:     nodesDesired ← nombre de nœuds visités jusqu'à présent
4:     while l'arbre n'est pas exploré en entier do
5:         nodesDesired ← nodesDesired × 2
6:         bound ← EstimateCost(nodesDesired)
7:         BLFS-expand(root, bound)
8:     end while
9: end function
10: function BLFS-expand(node, bound)
11:     if isLeaf(node) then
12:         Visit(node)
13:     else
14:         children ← GetChildren(node)
15:         n ← Size(children)
16:         for i ← 1 to n do
17:             cost ← BestCompletion(children[i])
18:             if cost ≤ bound then
19:                 BLFS-expand(children[i], bound)
20:             end if
21:         end for
22:     end if
23: end function
```

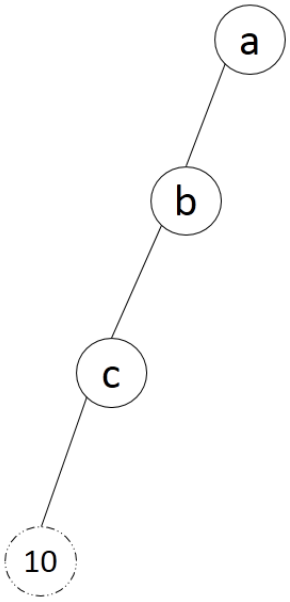

FIGURE 3.2: Première descente de ADS dans un arbre de recherche

trouvées précédemment à partir de chaque nœud. En se basant sur cette extrapolation, l'algorithme prédit la qualité de la prochaine solution qui serait trouvée à partir d'un nœud. Ensuite, il attribue à ce nœud une note qui représente l'amélioration éventuelle de la qualité de la meilleure solution globale qui serait amenée en faisant un retour-arrière vers ce nœud-ci. Autrement dit, ADS analyse la performance (en termes de génération de bonnes solutions) d'un nœud dans le passé, et tente de prédire sa performance dans un futur proche. À chaque fois qu'un retour-arrière est nécessaire, ADS s'appuie sur cette note pour choisir le nœud qui mérite d'être exploré en premier (celui avec la plus grande note). Quant aux autres nœuds, ils finiront par être explorés plus tard.

Supposons que l'on branche une première fois dans un arbre qui décrit l'espace de recherche pour un problème de minimisation, et que l'on trouve la première solution de qualité 10, comme sur la figure 3.2. Les nœuds a, b et c, à ce stade-ci, sont candidats au retour-arrière et sont tous équivalents aux yeux de ADS. Si on fait un retour-arrière vers chacun d'eux une seule fois, il est possible qu'un nœud, voire certains, génère une solution de meilleure qualité que celle trouvée lors de la première descente dans l'arbre de recherche (une qualité inférieure à 10 dans ce cas). En d'autre termes, il est possible que l'un des nœuds a, b et c améliore la qualité de la solution globale. Par exemple, la figure 3.3 montre les nœuds visités après avoir effectué un retour-arrière vers les nœuds a, b, et c. Le nœud b, ayant apporté une amélioration de 4, a plus de potentiel de mener vers de meilleures solutions. Par conséquent, ce nœud va être sélectionné lors du

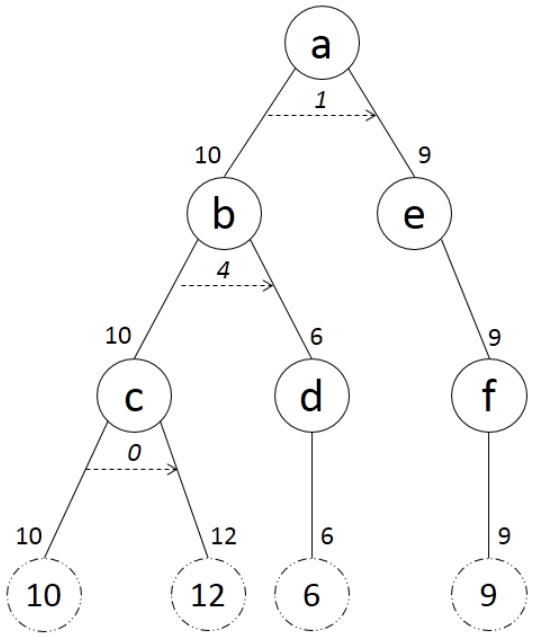

FIGURE 3.3: Portion de l'arbre de recherche visitée après un retour-arrière vers chacun des nœuds visités lors de la première descente par ADS

prochain retour-arrière. On remarque que pour pourvoir évaluer la capacité d'un nœud de générer de bonnes solutions, il faut avoir effectué au moins un retour-arrière vers ce nœud. Si aucun retour-arrière n'a encore été effectué vers un nœud, ADS n'a aucune information par rapport à la capacité de ce nœud d'améliorer la qualité de la solution globale, comme c'est le cas des nœuds d, e et f dans la figure 3.3.

Plus en détails, ADS définit pour chaque nœud deux entités importantes pour supporter le processus d'apprentissage, à savoir $ArcValue$ et $bestToDate$ [Gaudreault, 2009]. Pour chaque nœud n, $ArcValue(i)$ est défini comme étant la qualité de la première solution globale obtenue après une descente en profondeur dans le sous-arbre dont la racine est le fils i de n. Par exemple, sur la figure 3.3 chaque arc i provenant d'un nœud est étiqueté de sa valeur $AcrValue(i)$. De plus, pour chaque nœud, ADS définit un vecteur $bestToDate$ tel que montré dans l'équation (3.13). Chaque valeur à l'index i de ce vecteur est définie comme étant la valeur minimale des toutes les valeurs de $ArcValue$ jusqu'à l'index i.

$$bestToDate[i] = \min_{j=0}^{i} ArcValue(i) \tag{3.13}$$

Le processus de sélection de nœud lors des retours-arrière se base principalement sur

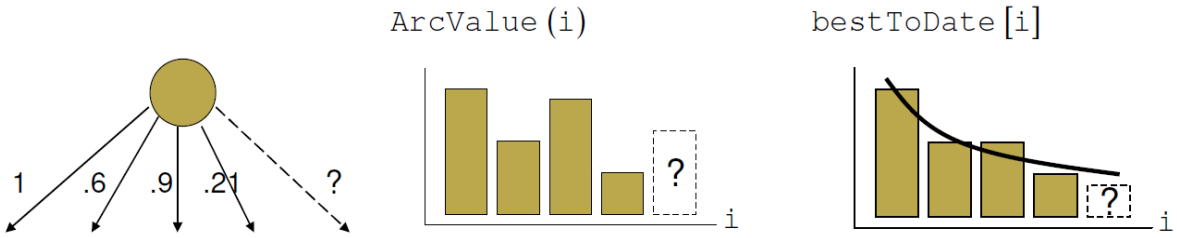

FIGURE 3.4: Relation en $ArcValue$ et $bestToDate$ (Tiré de [Gaudreault, 2009])

les vecteurs $bestToDate$. Chaque vecteur $bestToDate$ a la particularité de représenter une fonction monotone et décroissante, ce qui rend plus facile l'utilisation des points existants pour extrapoler la prochaine valeur du vecteur [Gaudreault, 2009], comme illustré dans la figure 3.4. Cette observation est la base du mécanisme de prédiction qui supporte la sélection de nœud lors des retours-arrière. ADS suppose l'existence d'une fonction continue $F(\mathbb{R}^+) \rightarrow \mathbb{R}^+$ qui approxime $bestToDate$. ADS prédit alors la valeur du prochain point dans $bestToDate$, $F(i)$, i étant la taille de $bestToDate$ au moment de la prédiction. En fait, pour chaque nœud n, la valeur $F(i)$ correspond à un estimé de la qualité de la prochaine solution qui sera générée si un retour-arrière vers n est effectué. Après avoir généré la fonction $F()$, ADS calcule l'amélioration de la qualité de solution estimée, appelée $Improvement()$. Pour ce faire, il calcule la différence entre les valeurs $F(i-1)$ et $F(i)$, i étant la taille de $bestToDate$ au moment de la prédiction, comme illustré dans la formule 3.14. Autrement dit, une fois que les fonctions $F()$ sont générées pour tous les nœuds candidats au retour-arrière, ADS sélectionne celui qui promet, selon $F()$, la plus grande amélioration de la qualité de solution.

$$Improvement(i) = F(i-1) - F(i) \tag{3.14}$$

Donc, on a expliqué que ADS sélectionne les nœuds vers lesquels effectuer les retours-arrière en se basant sur la capacité de ceux-ci d'améliorer la qualité de la solution globale. Autrement dit, ADS regarde à quel point il a été rentable d'effectuer un retour-arrière vers un nœud dans le passé pour évaluer ce nœud. Or, pour avoir effectué un retour-arrière dans le passé vers un nœud, ADS doit l'avoir sélectionné comme cible au retour-arrière. On voit tout de suite qu'il y a un équilibre entre exploration et exploitation qu'il faut respecter. En effet, ADS fonctionne en 2 modes : mode exploration, et mode exploitation. En mode exploitation, ADS exploite l'information collectée sur les différents nœuds internes de l'arbre de recherche ($Improvement(i)$) et se base dessus

pour guider la recherche. En mode exploration, il s'appuie sur l'analyse des déviations et, quand un retour-arrière est nécessaire, il sélectionne le nœud dont le prochain fils non visité a le plus petit nombre total de déviations, tout comme un LDS. Ainsi, l'algorithme doit constamment décider, selon un seuil ϵ, s'il doit se fier à l'information collectée jusqu'à date sur l'espace de recherche et explorer des zones qu'il connaît déjà, ou alors visiter des zones dont il n'a encore aucune information.

ADS est un algorithme complet et n'omet aucune portion de l'espace de recherche. Il apprend une fonction $Improvement(i)$ qui prédit à quel point la solution générée après avoir effectué un retour-arrière vers le nœud i serait bonne. En d'autres mots, il apprend la tendance de chaque nœud à mener vers de bonnes solutions pour concentrer ses efforts sur les nœuds les plus prometteurs en premier. Ceci dit, ADS finit par explorer l'arbre au complet éventuellement. De plus, il a été appliqué au problème de la coordination sciage-séchage-rabotage, qui est un problème distribué, et il s'avère que sa capacité d'apprendre le rend deux fois plus rapide que LDS, sur ce même problème [Gaudreault et al., 2012]. Cependant, le modèle qui supporte le processus d'apprentissage de ADS nécessite des entités et des mécanismes dont le coût ne devrait pas être négligé. D'un côté, les vecteurs *bestToDate* qui grandissent au fur et à mesure que l'exploration avance ajoutent à la complexité en mémoire de ADS. D'un autre, le processus d'approximation de la fonction $F()$, non seulement réalise une extrapolation de *bestToDate* pour chacun des nœuds, mais aussi devient plus coûteux en fonction de la taille des données à extrapoler. En d'autres termes, plus la zone explorée de l'arbre de recherche grandit plus les calculs d'extrapolation requièrent plus de ressources. Il est donc évident que le mécanisme de prédiction ajoute à la complexité de ADS en termes de temps de calcul.

Par ailleurs, Loth et al. ont proposé un algorithme appelé *Bandit Search for Constraint Programming* (BASCOP) [Loth et al., 2013a] qui guide la recherche (en manipulant les décisions de branchement) sur la base des estimations calculées (à travers plusieurs redémarrages) durant l'exploration. BASCOP a été testé sur un problème de *job shop* [Loth et al., 2013b] et a donné des résultats similaires à ce qui existait dans l'état de l'art de la programmation par contrainte.

3.3.2 Exploration globale et apprentissage hors ligne

Xu et al. ont introduit une formulation des CSP sous forme de problème d'apprentissage par renforcement où ils définissent les états comme étant les différentes instances ou sous-instances du CSP et les actions comme étant un ensemble d'heuristiques de

choix de variable [Xu et al., 2009]. Chaque transition correspond à une décision dans la résolution du CSP. Par exemple, dans un état s, une variable est sélectionnée selon l'heuristique a et une valeur lui est affectée. L'instance s est alors changée en une autre instance s', ce qui correspond exactement à une transition. Quant à la fonction de récompense, elle retourne une récompense à chaque fois qu'une solution au CSP est trouvée. L'algorithme Q-Learning est roulé à plusieurs reprises (un nombre fixe d'épisodes) sur une instance du problème pour apprendre la politique optimale. Cette politique spécifie pour chaque point de décision dans l'arbre de recherche quelle heuristique de choix de variable employer, parmi une liste d'heuristiques disponibles, afin de trouver une solution au problème en minimisant le temps de calcul. Autrement dit, l'algorithme possède une liste d'heuristiques de choix de variables qui peuvent toutes être employées pour le problème, et en attribue une à chaque nœud de l'arbre de recherche de sorte à trouver une solution le plus rapidement possible. Cette approche a abouti à des résultats très prometteurs [Xu et al., 2009]. De plus, testé sur une autre instance, cet algorithme a pu détecter qu'une heuristique est supérieure à toutes les autres auxquelles elle a été comparée.

En général, plusieurs paramètres ont besoin d'être réglés minutieusement lors de l'emploi de l'apprentissage par renforcement (comme l'équilibre entre exploitation et exploration, le taux d'apprentissage, et le facteur de dévaluation). Mais davantage d'attention devrait être prêtée lorsqu'il est employé dans un contexte de CSP, car plusieurs problèmes surgissent à ce moment-là. Une instance de CSP peut être résolue en faisant seulement quelques décisions mais peut prendre beaucoup plus de temps qu'une autre instance dont la résolution passe par plusieurs décisions. Ceci est dû aux différents temps de calcul liés à la propagation des contraintes. Donc, l'approche basique de la récompense binaire (1 si on trouve une solution, 0 sinon) n'est pas applicable. Par ailleurs, le fait que les décisions réalisées au sommet de l'arbre de recherche ont généralement un impact essentiel sur la performance de la résolution du problème ne peut pas être bien reflété en employant l'apprentissage par renforcement. En d'autres termes, il serait difficile dans un contexte de CSP d'attribuer des récompenses aux décisions prises proche de la racine de l'arbre. Dans un arbre de recherche, seules les feuilles peuvent indiquer si la séquence de décisions prises à partir de la racine jusqu'à la feuille était bonne ou pas (si elle a abouti à une solution au problème). Par conséquent, les décisions qui mènent directement vers les feuilles de l'arbre sont celles qui génèrent des récompenses.

3.3.3 Exploration locale et apprentissage en ligne

D'autres travaux ont proposé d'employer l'apprentissage par renforcement pour améliorer la recherche locale. Pour les problèmes d'optimisation combinatoire, les algorithmes qui emploient la recherche locale définissent un ensemble de solutions réalisables, une fonction de coût, et une fonction de voisinage. La recherche commence avec une solution réalisable et, à chaque intervalle de temps, se dirige vers une solution voisine pour laquelle la fonction de coût retourne une valeur plus petite. Une trajectoire est définie comme étant une succession de solutions, l'une voisinant l'autre.

Selon Moll et al., l'exploration locale peut être vue comme un processus décisionnel de Markov (MDP) où les états représentent les solutions, et les actions définissent les solutions voisines [Moll et al., 1998]. Les techniques d'apprentissage par renforcement peuvent être utilisées pour apprendre une fonction de coût afin d'améliorer l'exploration locale. Ceci peut être accompli en apprenant une nouvelle fonction de coût sur plusieurs trajectoires de la même instance du problème. L'algorithme STAGE [Boyan and Moore, 1997] suit cette approche et alterne entre l'utilisation de la fonction de coût apprise et l'utilisation de la fonction de coût originale. Au fur et à mesure que la précision de la fonction de coût apprise s'améliore, l'exploration s'exécute mieux sur les nouvelles trajectoires de la même instance.

Miagkikh and Punch proposent une approche de résolution de problèmes d'optimisation combinatoire basée sur une population d'agents d'apprentissage par renforcement [Miagkikh and Punch, 1999b]. Dans cette approche, les affectations des valeurs aux variables représentent les états du problème, c'est-à-dire que chaque assignation [4] représente un état, et les branchements représentent les actions. Chaque paire <état-action> <s,a> se voit attribuer une valeur, appelée *action-valeur*, qui est égale à la récompense moyenne estimée qui serait obtenue si l'action a est choisie à partir de l'état s. Cette approche maintient une population d'agents d'apprentissage par renforcement. Chaque agent de la population est affecté à une partie de l'espace de recherche, où il est censé apprendre les <actions-valeurs> et trouver la meilleure solution locale. Comme les agents explorent des parties du même arbre de recherche, on suppose que les <actions-valeurs> apprises par un agent à l'intérieur d'une zone de l'arbre peuvent être valides dans une autre zone. Cette supposition d'homogénéité permet de combiner les résultats des différents agents. Ainsi, une solution au problème est générée en copiant une partie de la meilleure solution locale trouvée par un agent, et en utilisant les

4. Une assignation peut être vue comme un ensemble de paires <variable-valeur>

<actions-valeurs> d'un autre agent pour compléter les assignations qui restent. Autrement dit, aussitôt que l'information locale provenant de différentes parties de l'espace de recherche est collectée, elle est combinée pour générer une solution globale au problème. Cette approche a été expérimentée sur le problème d'assignation quadratique qui est un problème NP-Difficile, en utilisant 50 agents. Elle s'est avérée compétitive face aux autres techniques de résolution de problèmes d'optimisation [Miagkikh and Punch, 1999b].

3.3.4 Exploration locale et apprentissage hors ligne

Une autre approche qui utilise l'apprentissage par renforcement pour améliorer la recherche locale dans le contexte de l'optimisation combinatoire est d'apprendre une fonction de coût de manière hors ligne (*offline*), et de l'utiliser sur de nouvelles instances du même problème. La publication [Zhang and Dietterich, 1995] peut être classifiée dans cette catégorie-ci.

Moll et al. combinent l'approche de Boyan and Moore [Boyan and Moore, 1997] et celle de Zhang and Dietterich [Zhang and Dietterich, 1995] en vue d'apprendre une fonction de coût qui est indépendante des instances du problème à résoudre [Moll et al., 1998]. Cette fonction de coût, une fois apprise, peut être utilisée pour améliorer l'exploration locale sur toutes les instances du problème. Cette approche consiste en une phase d'apprentissage hors ligne (*offline*), ensuite une alternance entre la fonction de coût apprise et l'originale prend place pour mieux guider l'exploration.

3.3.5 Conclusion

En conclusion, tous ces travaux ont essayé de récolter le maximum d'information sur la structure de l'arbre de recherche pour l'exploiter durant l'exploration. La capacité d'apprendre la structure de l'arbre de recherche peut être considérée comme un atout incontournable pour les algorithmes de résolution de COP. Des problèmes de grande taille peuvent ainsi être résolus, et d'autres peuvent avoir une solution en un temps record.

Le tableau 3.1 résume les travaux importants réalisés pour résoudre les problèmes d'optimisation combinatoire en utilisant de l'apprentissage. Plus précisément, il décrit le type et la technique d'apprentissage employés, ainsi que le type d'exploration effectuée. Certains travaux emploient l'apprentissage par renforcement, tandis que d'autres utilisent d'autres techniques d'apprentissage automatique qui ne relèvent pas nécessai-

rement du domaine de l'apprentissage par renforcement. Tous les travaux sont classifiés selon le type d'exploration (global ou local) et le type d'apprentissage (*online* ou *offline*) utilisés.

Article	**Type d'exploration**	**Type d'apprentissage**	**Ce qui est appris**	**Technique d'apprentissage**
Refalo 2004	Global	*Online*	Impact des assignations	Prédiction de la réduction de la taille de l'espace de recherche associée à chaque assignation
Gaudreault et al. 2012	Global	*Online*	La tendance des nœuds à mener vers de bonnes solutions	Extrapolation de la fonction qui décrit l'historique des meilleures solutions pour chaque nœud
Loth et al. 2013a	Global	*Online*	Heuristiques de choix de variable et de valeur	Apprentissage par renforcement
Xu et al. 2009	Global	*Offline*	L'heuristique de choix de variable optimale	Apprentissage par renforcement
Vasilikos and Lagoudakis 2010	Global	*Offline*	L'algorithme optimal	Apprentissage par renforcement
Zhang and Dietterich 2000	Global	*Offline*	La stratégie d'exploration optimale	Apprentissage par renforcement
Gambardella et al. 1995	Local	*Online*	Q-Values	Ant-Q
Nareyek 2004	Local	*Online*	L'heuristique optimale	Apprentissage par renforcement
Tilak and Mukhopadhyay 2010	Local	*Online*	Un jeu	Learning Automata (RL)
Miagkikh and Punch 1999b	Local	*Online*	Q-Values	Apprentissage par renforcement
Zhang and Dietterich 1995	Local	*Offline*	La fonction d'évaluation d'heuristique optimale	Apprentissage par renforcement
Theodore et al.	Local	*Offline*	La fonction d'évaluation de coût	Apprentissage par renforcement

TABLE 3.1: Résumé des travaux qui combinent l'optimisation combinatoire et l'apprentissage

Chapitre 4

RLBS : Une stratégie de retour-arrière basée sur l'apprentissage par renforcement

Ce chapitre présente un nouvel algorithme qui permet d'effectuer des retours-arrière dans un arbre de recherche et se base sur l'apprentissage par renforcement pour apprendre la structure de l'espace de recherche. Il s'agit d'une stratégie de retour-arrière utilisant les techniques d'apprentissage par renforcement en vue de détecter les zones les plus prometteuses de l'espace de recherche.

L'idée derrière l'algorithme décrit dans ce chapitre est qu'il est plus rentable d'explorer les zones de l'espace de recherche qui sont le plus probables de contenir de bonnes solutions. En d'autres mots, dans le contexte des problèmes d'optimisation avec contraintes, on voudrait effectuer des retours-arrière de manière intelligente, c'est-à-dire en sélectionnant les nœuds de l'arbre de recherche qui ont le plus de chances de mener vers de bonnes solutions. Il s'agit de pouvoir évaluer tous les nœuds de l'arbre de recherche en fonction de leur potentiel à générer de bonnes solutions. Chaque solution est évaluée par la fonction objectif qui lui attribue une valeur selon sa qualité. Après un retour-arrière vers un nœud donné, l'exploration poursuit la recherche jusqu'à trouver une nouvelle solution. La qualité de cette solution indique à quel point il a été rentable de faire ce dernier retour-arrière vers le nœud en question. Dans un contexte d'apprentissage par renforcement, la décision de faire un retour-arrière vers un nœud donné peut être vue comme une action, la solution trouvée représenterait un état, et la qualité de cette solution contribuerait dans la récompense retournée.

4.1 Le retour-arrière formulé comme un problème d'apprentissage par renforcement

Lors de l'exploration d'un arbre de recherche, la stratégie de retour-arrière peut être formulée comme un problème d'apprentissage par renforcement. Les feuilles de l'arbre représentent les états de l'environnement, les nœuds ouverts constituent les actions, et l'amélioration de la meilleure solution trouvée représente la récompense, comme illustré dans la figure 4.1. Dans cette figure, la partie grisée où les arcs sont en pointillés représente la partie encore non explorée de l'arbre de recherche. Les nœuds barrés représentent des nœuds fermés, c'est-à-dire les nœuds dont tous les fils ont été visités. Les nœuds avec un disque de couleur noire correspondent aux actions, à savoir les nœuds candidats au retour-arrière. Tous les nœuds ouverts (qui ont au moins un fils non visité) sont considérés comme des candidats au retour-arrière. Les nœuds représentés avec des cercles au contour doublé correspondent aux états de l'environnement. Il s'agit des feuilles de l'arbre de recherche. Quant aux récompenses, elles sont déduites en fonction de la qualité de chaque solution.

Par exemple, supposons que l'on soit en cours d'exploration pour résoudre un problème de minimisation et que b soit la meilleure solution trouvée jusqu'à présent. Soit q_b la valeur retournée par la fonction objectif pour la solution b. Après avoir trouvé une nouvelle solution x, avec q_x comme valeur retournée par la fonction objectif, la récompense générée sera $r_x = q_b - q_x$. La récompense r_x reflète à quel point la solution x améliore la meilleure solution globale. En d'autres termes, elle donne un idée sur le profit généré par le dernier retour-arrière effectué, pour trouver la solution x. Ainsi, la nouvelle solution est comparée à la meilleure solution trouvée jusqu'à présent, ce qui permet d'éviter les minima locaux.

Le retour-arrière dans le contexte de résolution de problèmes d'optimisation avec contraintes est formulé en tant que problème d'apprentissage par renforcement comme suit :

- États : Les feuilles de l'arbre de recherche
- Actions : Aller aux nœuds ouverts
- Récompenses : L'amélioration de la meilleure solution globale

Nous introduisons une stratégie de retour-arrière adaptative basée sur l'apprentissage par renforcement, que l'on nomme *Reinforcement Learning Backtracking Strategy* (RLBS). Elle permet de concentrer la recherche sur les zones les plus prometteuses de l'espace de recherche en premier. Lors de l'exploration de l'arbre de recherche, à

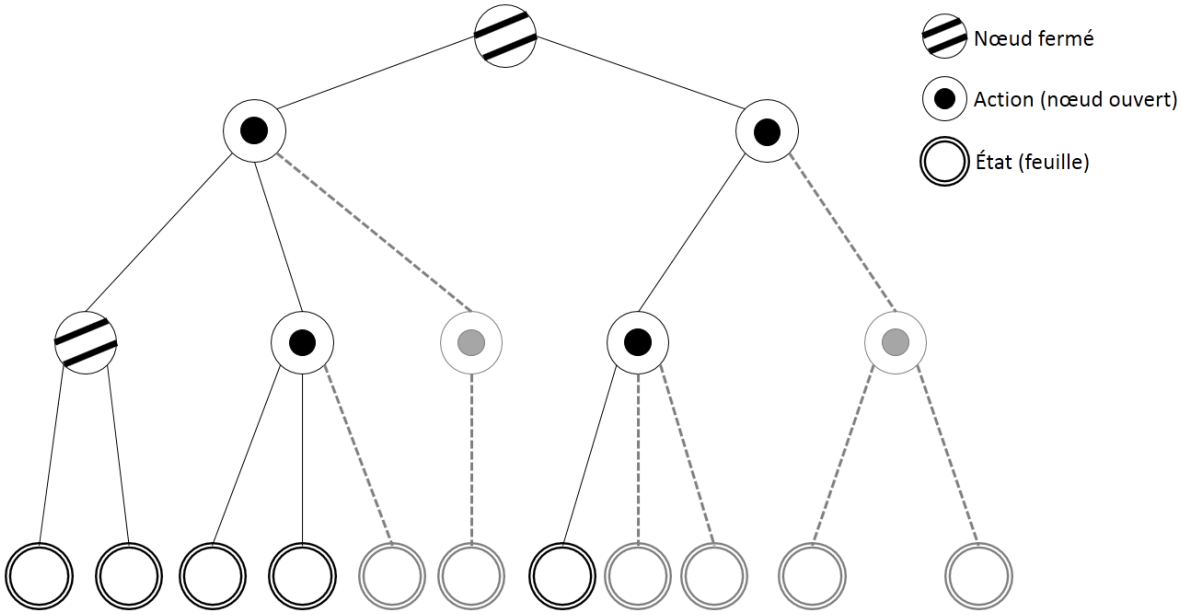

FIGURE 4.1: Éléments de la stratégie de retour-arrière sous forme de problème d'apprentissage par renforcement dans un arbre de recherche

chaque fois qu'une solution est trouvée, un retour-arrière doit être effectué pour poursuivre la recherche. Tous les nœuds ouverts, c'est-à-dire qui ont au moins un seul fils non-exploré, sont des candidats au retour-arrière. Aller à chacun de ces nœuds correspond à une action, à savoir l'action d'effectuer un retour-arrière vers ce nœud. Une fois qu'un nœud est choisi pour le retour-arrière, l'exploration continue à partir de ce nœud-là en effectuant une descente dans l'arbre de recherche à partir de son prochain fils non visité jusqu'à trouver la prochaine solution. La différence entre la qualité de cette nouvelle solution et la meilleure solution trouvée jusqu'à présent représente la récompense retournée pour avoir choisi la dernière action (voir algorithme 4). Autrement dit, la récompense reflète l'apport associé au choix du nœud en question pour le retour-arrière, dans l'amélioration de la meilleure solution globale. Comme les arbres de recherche considérées sont non-binaires, l'algorithme a l'opportunité d'effectuer des retours-arrière à plusieurs reprises vers le même nœud. Ceci permet d'identifier les actions qui payent le plus, c'est-à-dire les nœuds qui mènent vers les meilleures solutions. Par conséquent, ceci offre la possibilité de diriger l'exploration vers les zones les plus prometteuses de l'espace de recherche et, ainsi, optimiser la résolution.

Cette situation nous rappelle le problème de la "Machine à sous à k bras" [Alpaydin, 2004]. Ce problème peut être formulé sous forme d'un problème d'apprentissage par

Algorithm 4 Reinforcement Learning Backtracking Strategy

```
1: repeat
2:     nodes ← GetOpenNodes()
3:     max ← 0
4:     bestNode ← nodes[1]
5:     n ← Size(nodes)
6:     for i ← 2 to n do
7:         qValue ← GetQValue(nodes[i])
8:         if qValue > GetQValue(bestNode) then
9:             bestNode ← nodes[i]
10:            max ← qValue
11:        end if
12:    end for
13:    nodesToUpdate ← NIL
14:    reward ← ExploreUntilFirstSol(bestNode, nodesToUpdate)
15:    n ← Size(nodesToUpdate)
16:    for i ← 1 to n do
17:        oldQValue ← GetQValue(nodesToUpdate[i])
18:        newInfo ← reward − oldQValue
19:        newQValue ← oldQValue + α × newInfo
20:        SetQValue(nodesToUpdate[i], newQValue)
21:    end for
22: until l'arbre de recherche est exploré au complet
```

renforcement mono-état. Plusieurs actions sont disponibles, chacune correspondant à appuyer sur l'un des bras de la machine. Chaque action peut ou pas mener à une récompense de manière stochastique. Dans le cas des retours-arrière, choisir une action peut mener à la découverte de nouveaux nœuds, donc de nouvelles actions, en plus de retourner une récompense. Ceci est complètement stochastique et non-stationnaire.

4.2 Apprentissage par renforcement

L'évaluation de chaque action, représentée par la Q-Value, doit être mise à jour à chaque fois qu'une récompense est collectée après avoir choisi cette action-ci. Mais comme il

s'agit d'un environnement mono-état, l'équation (3.3) se réduit à l'équation suivante :

$$Q_{t+1}(a) \leftarrow (1-\alpha)\, Q_t(a) + \alpha\, r_{t+1}(a) \tag{4.1}$$

où $r_{t+1}(a)$ est la récompense collectée après avoir choisi l'action a au moment $t+1$ et α le taux d'apprentissage.

La prochaine action est sélectionnée en se basant sur les Q-Valeurs. Plus un nœud est payant, plus sa Q-Valeur est importante. Un nœud qui aurait mené à de très bonnes solutions au début mais qui n'aurait jamais mené vers de bonnes solutions par la suite verra sa Q-Value décroître au fil du temps, jusqu'à ce qu'il devienne moins intéressant que d'autres nœuds. Donc, au moment de choisir une action (aller à un nœud ouvert en faisant le retour-arrière) toutes les actions possibles (aller aux nœuds ouverts) sont ordonnées selon leurs Q-Valeurs, et l'action avec la plus grande Q-Valeur est sélectionnée. Une fois qu'un nœud candidat au retour-arrière a est sélectionné, l'algorithme explore son prochain fils non-visité et réalise une descente, comme DFS, à partir de ce dernier jusqu'à la prochaine feuille. Ensuite, l'algorithme évalue la solution nouvellement trouvée, collecte la récompense $r(a)$ et met à jour la Q-Valeur de a selon l'équation (4.1).

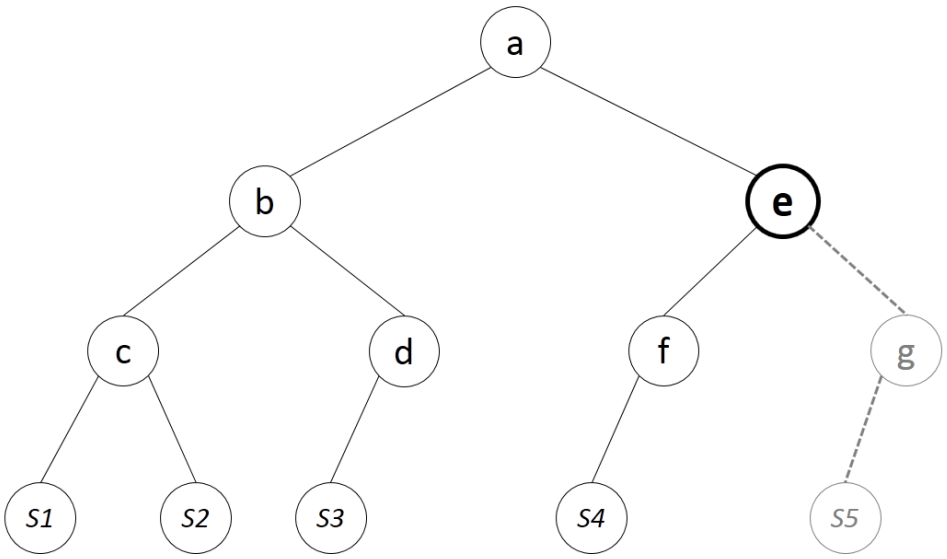

FIGURE 4.2: Retour-arrière effectué par RLBS dans un arbre de recherche

Par exemple, dans la figure 4.2, supposons que l'exploration ait atteint la feuille $S4$ et que le nœud sélectionné pour le retour-arrière soit le nœud e. L'exploration va poursuivre à partir de ce nœud en développant son prochain fils non visité (le nœud g) et en faisant une descente en profondeur dans l'arbre à partir de là jusqu'à trouver la prochaine solution (la feuille $S5$). Ensuite, RLBS va collecter la récompense à partir de la feuille $S5$ et va s'en servir pour mettre à jour les Q-Valeurs des nœuds g et e selon l'équation (4.1).

4.3 Initialisation de l'algorithme

Au début de l'exploration, l'algorithme descend jusqu'à la première feuille de l'arbre comme DFS. Puis, il effectue un retour-arrière vers chacun des nœuds ouverts. À ce moment-là, les nœuds ouverts se trouvent à être tous les nœuds explorés lors de cette première descente sauf la feuille. Le fait d'effectuer ces retours-arrière permet de calculer les Q-Valeurs des nœuds ouverts. Une telle exploration est similaire aux deux premières itérations de LDS, c'est-à-dire que ceci revient à exécuter LDS avec 0 et 1 déviation. Ensuite, l'algorithme commence à se baser sur les Q-Valeurs pour choisir le nœud cible lors des prochains retours-arrière.

Lors des retours-arrière, l'algorithme peut explorer de nouveaux nœuds avant de trouver une feuille de l'arbre, comme dans le cas du nœud g de l'exemple précédent, illustré dans la figure 4.2. Chaque fois qu'un nœud est visité pour la première fois, sa Q-Valeur est initialisée en utilisant celle de son parent.

4.4 Explication de l'algorithme

RLBS (illustré par l'algorithme 4) se divise en trois parties principales : sélection du meilleur nœud, exploration, et mise à jour.

Sélection du meilleur nœud

Lors de l'exploration de l'arbre de recherche, à chaque fois qu'un retour-arrière est nécessaire, RLBS commence par sélectionner le meilleur nœud parmi tous les nœuds ouverts. Pour ce faire, il fait appel à la fonction *GetOpenNodes()* (ligne 2 de l'algorithme 4), ensuite il sélectionne le nœud avec la plus grande Q-Value (ligne 3 à ligne 12 de l'algorithme 4). Pour récupérer les Q-Values, RLBS fait appel à la fonction *GetQValue(node)* qui retourne la Q-Value de *node*.

Exploration à partir du meilleur nœud

Une fois le meilleur nœud sélectionné (appelé *bestNode*), c'est-à-dire le nœud ayant la plus grande Q-Valeur, RLBS poursuit l'exploration à partir de ce nœud jusqu'à ce que la prochaine solution soit trouvée (ligne 14 de l'algorithme 4). Au fait, il s'agit d'une descente dans l'arbre de recherche à partir du prochain fils non visité de *bestNode*. Ceci est réalisé par la fonction *ExploreUntilFirstSol(bestNode, nodesToUpdate)* qui, par la même occasion, garde une trace des nœuds qui devront être mis à jour par la suite (appelés *nodesToUpdate*) et retourne la récompense relative à la nouvelle solution trouvée. *nodesToUpdate* est une liste qui contient le nœud $bestNode$ plus tous les nœuds visités pour la première fois lors de la toute dernière descente. Ces nœuds doivent être mis à jour car ils ont contribué à la découverte de la nouvelle solution après le dernier retour-arrière (dernière action exécutée). Autrement dit, ces nœuds méritent d'être mis à jour après la découverte de la dernière solution car ils se trouvent entre cette solution et le nœud vers lequel le dernier retour-arrière a été effectué (*bestNode*). Ce qui veut dire qu'ils peuvent mener à la même solution que *bestNode*.

Mise à jour des nœuds appropriés

Après que *ExploreUntilFirstSol* ait trouvé une solution et retourné une récompense, RLBS utilise cette dernière pour mettre à jour les Q-Valeurs des nœuds *nodesToUpdate*, incluant *bestNode* (ligne 15 à 21 de l'algorithme 4). Pour ce faire, RLBS calcule, pour chacun des nœuds en question, la nouvelle valeur de Q-Value selon la formule :

$$newQValue \leftarrow oldQValue + \alpha \times (reward - oldQvalue) \tag{4.2}$$

où $newQValue$ est la nouvelle valeur de Q-Value, $oldQValue$ l'ancienne valeur de Q-Value, $reward$ la récompense retournée par *ExploreUntilFirstSol*, et α le taux d'apprentissage (ligne 17 à 19 de l'algorithme 4).

RLBS fait appel ensuite à la fonction *SetQValue(nodeToUpdate, newQValue)* qui met la valeur de Q-Value de *nodeToUpdate* à $newQValue$ (ligne 20 de l'algorithme 4).

4.5 Conclusion

Nous avons vu comment le retour-arrière dans un arbre de recherche peut être formulé en tant que problème d'apprentissage par renforcement. On a aussi expliqué comment

RLBS utilise cette formulation et s'appuie sur des techniques d'apprentissage par renforcement pour exécuter les retours-arrière. Dans le chapitre suivant, on présente les différentes expérimentations menées sur RLBS pour le comparer à d'autres stratégies de retour-arrière, et on discute les résultats de ces expérimentations.

Chapitre 5

Expérimentations

Ce chapitre porte sur des expérimentations menées pour évaluer RLBS et le comparer à d'autres stratégies de retour-arrière, dans le cadre de la résolution des problèmes d'optimisation combinatoire. Il décrit les résultats obtenus sur différentes instances du problème de la planification du rabotage issues directement de l'industrie du bois.

Les données utilisées dans les expérimentations proviennent d'une compagnie canadienne de production de bois d'œuvre.

5.1 Description du problème industriel

Le problème du rabotage s'inscrit dans la chaîne de production, et s'il n'est pas optimisé la productivité de ladite chaîne s'en trouve pénalisée. Ce projet se concentre sur ce problème, et plus particulièrement sur la planification des opérations du rabotage [Gaudreault et al., 2011]. Cette planification signifie la génération d'un plan de production qui permet de répondre à toutes les commandes clients enregistrées, à temps, et en respectant les différentes contraintes liées aux inventaires. Nous nous intéressons particulièrement à un processus de planification en temps réel, qui offre la possibilité de modifier le plan global de production à la demande, pour pouvoir l'adapter aux éventuels imprévus.

La planification des opérations du rabotage du bois en temps réel permet généralement de rendre le plan de production global, au niveau d'une chaîne, très flexible. Dès lors, la chaîne peut s'adapter à des changements de dernière minute. Une illustration de cette dynamique serait l'arrivée d'une nouvelle commande client après la génération du plan de production global. Actuellement, les mécanismes mis en place dans l'industrie

du bois ne peuvent pas supporter ce genre de situation. Par conséquent, les éventuelles commandes futures sont plutôt estimées, suite à des calculs prévisionnels, afin de prédire les quantités à produire pour répondre à toutes les commandes. De telles commandes doivent être satisfaites à temps, et en gardant un niveau d'inventaire raisonnable. Or, ces calculs prévisionnels reposent sur des facteurs souvent instables, comme la tendance du marché, l'historique des ventes, l'historique des commandes, etc., ce qui résulte en des estimations éloignées de la réalité.

Cette incapacité de prédire le futur avec un certain degré de précision, mène à des livraisons en retard et une variation d'inventaire généralement non optimale. Donc, une planification du rabotage du bois à la demande apporterait plus d'optimalité dans la gestion des inventaires et de la matière première, ainsi qu'une réduction remarquable des retards de livraison. Ces deux points se traduisent rapidement par des profits supplémentaires que les compagnies s'empêchent d'avoir à cause des retards, ainsi que des gains considérables d'énergie, dans le sens où le stockage et transfert de matière première entre les différentes unités seraient optimisés.

À l'unité de finition, les sciages provenant de l'unité de séchage sont rabotés pour donner aux pièces la forme souhaitée et arrondir les arêtes. Ensuite, les morceaux de bois sont classés selon leur qualité. Cette qualité dépend des défauts physiques de la pièce et de la teneur résiduelle en eau. Les pièces peuvent être écourtées pour produire des pièces d'une meilleure qualité, et les sciages très longs peuvent être coupés en plusieurs pièces. Généralement, ce processus est automatisé et optimisé de façon à générer la plus grande valeur possible sur le marché, sans tenir compte des commandes des clients.

Le processus de rabotage produit plusieurs types de pièces en sortie (coproduction) à partir d'une même pièce en entrée (divergence) [Gaudreault et al., 2011] (voir la figure 5.1). La coproduction est étroitement lié au processus de transformation lui-même et, par conséquent, ne peut pas être exclue du point de vue de la planification. Un ensemble de processus peuvent être employés afin de transformer un produit en entrée en plusieurs produits en sortie. Les produits sont catégorisés selon leur dimension (par exemple $2'' \times 4''$). La ligne de rabotage peut traiter une seule catégorie de produits durant un quart de travail.

Un temps de mise en course considérable est obligatoire pour passer d'une dimension de produits à l'autre (par exemple du $2'' \times 4''$ au $2'' \times 6''$). Donc, la majorité des entreprises préfèrent de ne permettre le changement de dimension en entrée qu'entre des quarts de travail et privilégient les campagnes qui opèrent sur une même dimension pour plus

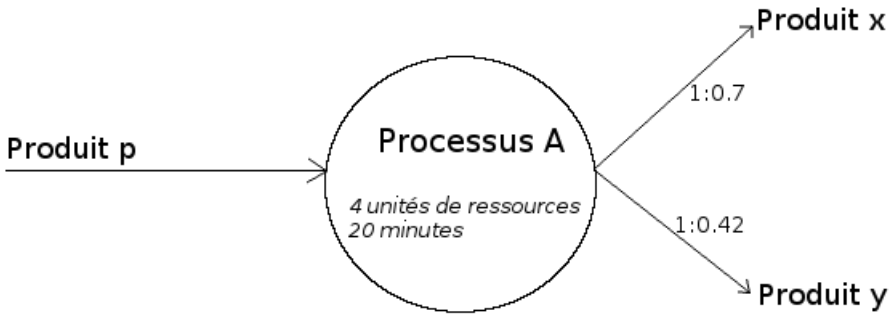

FIGURE 5.1: Exemple de configuration d'un processus de rabotage.

d'un seul quart de travail. Selon cette optique, les sciages de même dimension et même longueur doivent être traités ensemble.

Chaque commande client spécifie une quantité donnée de produits finis qui doit être livrée à un moment précis. La planification des opérations du rabotage consiste en la production d'un plan qui spécifie les campagnes quoi doivent être traitées, et les quantités qui doit être transformées. Plus précisément, elle se résume à prendre 3 décisions :

- Choisir les campagnes à réaliser (i.e. les dimensions) ;
- Le moment de début et de fin de chacune des campagnes ;
- Pour chaque campagne, déterminer la quantité de produits à traiter de chaque longueur.

L'objectif est de générer un plan de production qui minimise les retards de livraison. La figure 5.2 montre un exemple de plan de production pour une ligne de rabotage, qui inclue 3 campagnes ($2'' \times 3''$, $2'' \times 6''$ et $2'' \times 4''$) sur six quarts de travail.

Gaudreault et al. proposent de bonnes heuristiques de choix de variable et de valeur spécifiques au problème de la planification du rabotage [Gaudreault et al., 2010]. Dans [Gaudreault et al., 2012], ces heuristiques ont été employées pour guider la recherche de solutions dans un modèle de programmation par contrainte. Muni de ces heuristiques, LDS a largement surpassé DFS ainsi que l'approche de la programmation mathématique. De la parallélisation a aussi été utilisée pour améliorer la résolution de ce problème [Moisan et al., 2013].

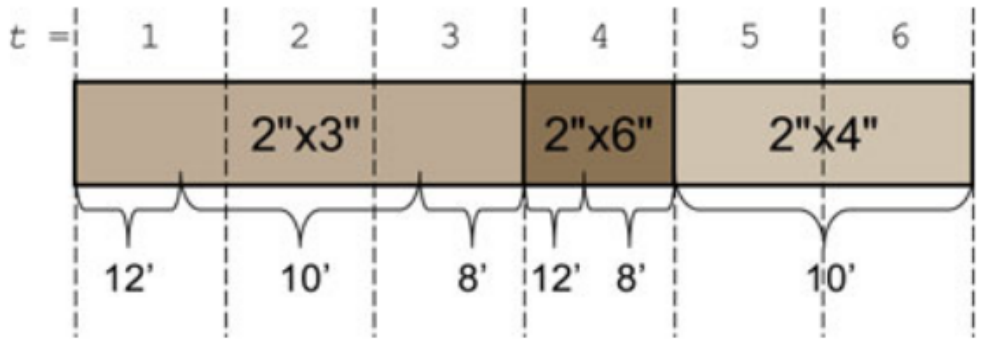

FIGURE 5.2: Plan de production des opérations de rabotage sur six quarts de temps (tiré de [Gaudreault et al., 2012]).

5.2 Problème de la planification du rabotage sous forme de COP

Le problème de la planification du rabotage du bois peut être formulé comme un problème d'optimisation combinatoire avec contraintes [Gaudreault et al., 2011]. Pour ce faire, il faut définir les variables, les contraintes et la fonction objectif du problème d'optimisation. Il faut aussi déterminer l'ensemble des paramètres nécessaires pour résoudre le problème.

Les variables représentent principalement les entités inconnues du problèmes. Dans ce cas, elles représentent les familles à choisir, et les quantités de produits à traiter pour chaque période. Le modèle a aussi besoin de variables supplémentaires qui vont être utilisées dans les contraintes. Quant aux contraintes du problème, elles reflètent les contraintes matérielles propres au processus de transformation, ainsi que des contraintes supplémentaires pour assurer l'intégrité du modèle. Les paramètres, quant à eux, reflètent les propriétés spécifiques à la ligne de rabotage, aux processus de production employés, et au carnet de commandes clients.

Les sous-sections suivantes décrivent plus en détail les différentes entités modélisant le problème de la planification du rabotage sous forme de COP.

5.2.1 Paramètres

On représente le nombre de périodes par T, le nombre de produits par N et le nombre de processus de fabrication par M. Les paramètres employés sont les suivants :

– Un vecteur $Capacite$ de taille T où chaque élément $Capacite[t]$ représente la

capacité disponible à la période t. Ce vecteur reflète la capacité de traitement disponible sur la ligne de rabotage à chaque quart de travail.

- Une matrice $ProcOut$ de taille $M \times N$ où chaque élément $ProcOut[m][n]$ représente le ratio en sortie du produit n pour le processus m.
- Un vecteur $CapConsom$ de taille M où chaque élément $CapConsom[m]$ représente la capacité consommée par le processus m lors d'une seule exécution, c'est-à-dire pour traiter une seule unité en entrée.
- Un vecteur $ProcFamille$ de taille M où chaque élément $ProcFamille[m]$ représente la famille de produit traitée par le processus m.
- Un vecteur $ProcInput$ de taille M où chaque élément $ProcInput[m]$ représente le produit consommé en entrée par le processus m.
- Un vecteur $ComQuantite$ de taille N où chaque élément $ComQuantite[n]$ représente la quantité commandée du produit n.
- Un vecteur $DateLive$ de taille N où chaque élément $DateLive[n]$ représente la période à laquelle le produit doit être fabriqué au plus tard.
- Un vecteur $InvInit$ de taille N où chaque élément $InvInit[n]$ représente la quantité initiale du produit n disponible en inventaire.

5.2.2 Variables

Les variables utilisées dans le modèle sont les suivantes :

- Un vecteur $Famille$ de taille T où chaque élément $Famille[t]$ représente la famille traitée durant la période t.
- Une matrice $ProcPlanPeriode$ de taille $M \times T$ où chaque élément $ProcPlanPeriode[m][t]$ représente le nombre d'exécutions du processus m à la période t.
- Une matrice $Cons$ de taille $N \times T$ où chaque élément $Cons[n][t]$ représente la quantité du produit n consommée en entrée par la ligne de rabotage à la période t.
- Une matrice $Prod$ de taille $N \times T$ où chaque élément $Prod[n][t]$ représente la quantité du produit n produite par la ligne de rabotage à la période t.
- Une matrice $Supp$ de taille $N \times T$ où chaque élément $Prod[n][t]$ représente la quantité du produit n disponible en sortie du séchoir à la période t.
- Une matrice I de taille $N \times T$ où chaque élément $I[n][t]$ représente la quantité du produit n disponible en inventaire à la période t.

– Une matrice BO de taille $N \times T$ où chaque élément $BO[n][t]$ représente la quantité du produit n en arrérage à la période t.
– Une variable *Retard* qui représente le volume d'arrérage total, c'est-à-dire, la somme de tous éléments de la matrice BO.

5.2.3 Contraintes

Voici les contraintes du modèle :

– La capacité totale consommée par tous les processus dans une période ne doit pas dépasser la capacité disponible durant cette période. La contrainte (5.1) s'assure que le plan de production respecte la capacité maximale de la ligne de rabotage et ne surcharge pas cette dernière.

$$\forall t, 1 \leq t \leq T$$
$$\sum_{m=1}^{M} ProcPlanPeriode[m][t] \times CapConsom[m] \leq Capacite[t] \quad (5.1)$$

– Tous les processus exécutés dans une période doivent traiter la même famille de produit que celle choisie pour cette période, comme présenté dans la contrainte (5.2).

$$\forall t, 1 \leq t \leq T \quad \forall m, 1 \leq m \leq M$$
$$Famille[t] \neq ProcFamille[m] \Rightarrow ProcPlanPeriode[m][t] = 0 \quad (5.2)$$

– Variations d'inventaire :

  La quantité de produits dans l'inventaire à une période t doit être égale à celle de la période précédente $t-1$ plus les quantités de produits provenant de l'unité de séchage et celles produites durant cette période t, moins les quantités de produits qui doivent être livrés et celles consommées par les différents processus durant cette même période t, comme décrit dans la contrainte (5.3).

$$\forall t, 1 \leq t \leq T, \quad \forall n, 1 < n \leq N$$
$$I[n][t] = I[n][t-1] + Supp[n][t] - Cons[n][t] + Prod[n][t] - ComQuantite[n][t] \quad (5.3)$$

  Les variables *Cons* et *Prod* doivent refléter l'état des consommations et productions de produits à chaque période. Les contraintes (5.4) et (5.5) assurent cette

intégrité.

$$\forall t, 1 \leq t \leq T, \quad \forall n, 1 \leq n \leq N$$
$$Cons[n][t] = \sum_{m=0, ProcInput[m]=n}^{M} ProcPlanPeriode[m][t] \quad (5.4)$$

$$\forall t, 1 \leq t \leq T, \quad \forall n, 1 \leq n \leq N$$
$$Prod[n]t] = \sum_{m=0}^{M} ProcPlanPeriode[m][t] \times ProcOut[m][n] \quad (5.5)$$

– Arrérage :

L'arrérage constitue les retards dans la production, ce qui peut être représenté conceptuellement par la partie négative de l'inventaire, comme décrit dans la contrainte (5.6).

$$\forall t, 1 \leq t \leq T, \quad \forall n, 1 \leq n \leq N$$
$$BO[n][t] = \begin{cases} -I[n][t] & \text{si } I[n][t] < 0 \\ 0 & \text{sinon} \end{cases} \quad (5.6)$$

– Le volume d'arrérage :

La contrainte (5.7) représente le volume total des retards de production, qui revient à faire la somme de tous les éléments de la matrice BO.

$$Retard = \sum_{n=1}^{N} \sum_{t=1}^{T} BO[n][t] \quad (5.7)$$

5.2.4 Fonction objectif

L'objectif est de produire un plan de production avec le moins de retards possible. Ceci est modélisé par la fonction objectif (5.8) et revient à minimiser la valeur de la variable $Retard$.

$$\min Retard \quad (5.8)$$

5.3 Heuristiques employées pour la résolution du problème

À cause de la complexité du problème de la planification du rabotage, les heuristiques standards qui s'appliquent à toutes sortes de problèmes d'optimisation du genre COP, comme *Most Constraining Variable*, *Least Constraining Value* ou *Minimum Remaining Value*, ne sont pas suffisantes. Il faut donc employer des heuristiques plus spécifiques au problème en question pour mieux guider l'exploration de l'espace de recherche. Il s'agit d'heuristiques qui tiennent compte de tous les paramètres propres au problème pour mieux ordonner l'instanciation des variables.

Les mêmes heuristiques de choix de variable et de valeur qui ont été prouvées bonnes pour ce problème dans [Gaudreault et al., 2012] ont été employées. À savoir, pour chaque période, la famille de produits à traiter est choisie en premier, ensuite les quantités de chaque produit en entrée sont déterminées pour chacun des processus. Le choix de la famille se base principalement sur les commandes clients. La famille pour laquelle le plan actuel présente le plus grand retard de livraison est privilégiée. De la même manière, la quantité est déterminée de façon à satisfaire les commandes clients. Donc, pour chacun des processus, le solveur essaye en premier la quantité de produits en entrée pour satisfaire la demande actuelle, telle que décrite dans le carnet de commandes, en tenant compte de l'état de l'inventaire.

5.3.1 Heuristiques de choix de variable

Les variables $Famille[t]$ sont instanciées dans l'ordre statique, c'est-à-dire qu'on décide des familles à utiliser dans l'ordre croissant des périodes ($Famille[1]$, puis $Famille[2]$ ..., jusqu'à $Famille[N]$, où N est le nombre total des périodes, aussi appelé l'horizon de planification).

Une fois qu'une famille f est choisie pour une période t ($Famille[t] := f$), les quantités des processus compatibles avec la famille f sont déterminées pour la période t. En d'autres termes, les variables $ProcPlanPeriode[m][t]$ sont instanciées, où m représente tout processus compatible avec la famille f. Quant aux processus qui ne sont pas compatibles avec la famille f, leurs quantités sont mises à zéro car ils ne peuvent pas être exécutés. Parmi les processus compatibles avec f, l'heuristique de choix de variable sélectionne en premier celui qui produit la plus grande quantité du produit qui présente le plus grand retard de production, comme illustré dans l'exemple 2.

Algorithm 5 Heuristique de choix de variable processus

```
1: function ProcVariableHeuristic(Model model, Family selectedFamily)
2:     products ← GetProductsOfFamily(model, selectedFamily)
3:     pdMaxBackOrder ← SelectProductMaxBackOrder(model, products)
4:     pcSelectedFamily ← GetProcessesOfFamily(model, selectedFamily)
5:     process ← pcSelectedFamily[1]
6:     maxOutput ← GetOutput(process, pdMaxBackOrder)
7:     n ← size(pcSelectedFamily)
8:     for i ← 2 to n do
9:         output ← GetOutput(pcSelectedFamily[i], pdMaxBackOrder)
10:        if output > maxOutput then
11:            maxOutput ← output
12:            process ← pcSelectedFamily[i]
13:        end if
14:    end for
15:    return process
16: end function
```

Exemple 2. Soit $P = \{m_1, m_2, m_3\}$ l'ensemble des processus compatibles avec la famille f L'heuristique de choix de variable sélectionne les variables $ProcPlanPeriode$ dans cet ordre : $ProcPlanPeriode[m_2][t]$, $ProcPlanPeriode[m_3][t]$, $ProcPlanPeriode[m_1][t]$
Avec

$$BO[n_2][t] > BO[n_3][t] > BO[n_1][t] \tag{5.9}$$

Et

$$\arg\max_n ProcOut[m_1][n] = n_1 \tag{5.10}$$

$$\arg\max_n ProcOut[m_2][n] = n_2 \tag{5.11}$$

$$\arg\max_n ProcOut[m_3][n] = n_3 \tag{5.12}$$

L'algorithme 5 décrit plus en détail le fonctionnement de l'heuristique de choix de variable processus employée pour la résolution du problème. Premièrement, cet algorithme

récupère la liste des produits qui appartiennent à la famille $selectedFamily$ sélectionnée par l'heuristique de choix de valeur pour la variable $Famille[t]$, où t est la période en cours de planification. Pour ce faire, il fait appel à la fonction $GetProduitsOfFamily$ (ligne 2 de l'algorithme 5). Puis, il fait appel à la fonction $SelectProductMaxBackOrder$ pour trouver le produit $pdMaxBackOrder$ avec le plus grand retard de livraison jusqu'à la période t (ligne 3 de l'algorithme 5). Ensuite, il récupère la liste des processus compatibles avec la famille sélectionnée $selectedFamily$ (ligne 4 de l'algorithme 5), et sélectionne celui qui a le plus grand ratio en sortie pour le produit $pdMaxBackOrder$ (lignes 5 à 14 de l'algorithme 5).

5.3.2 Heuristiques de choix de valeur

L'heuristique de choix de valeur pour les variables $Famille$ choisit en premier la famille qui a le plus grand retard de livraison.

Algorithm 6 Heuristique de choix de valeur pour les quantités de processus

```
1:  function ProcValueHeuristic(Model model, Process process)
2:      products ← GetOutputProducts(model, process)
3:      demands ← GetDemands(model, products)
4:      n ← size(demands)
5:      output ← GetOutputQuantity(model, process, products[1])
6:      maxQuantity ← demands[1] / output
7:      for i ← 2 to n do
8:          output ← GetOutputQuantity(model, process, products[i])
9:          quantities[i] ← demands[i] / output
10:         if quantities[i] > maxQuantity then
11:             maxQuantity ← quantities[i]
12:         end if
13:     end for
14:     capacity ← GetCapacity(model)
15:     quantity ← Min(maxQuantity, capacity)
16:     return quantity
17: end function
```

L'heuristique de choix de valeur qui détermine dans quel ordre tester les différentes quantités de processus (instanciation des variables $ProcPlanPeriode$) commence par

la quantité (le nombre d'exécutions du processus) qui garantit de couvrir la demande, limitée par la capacité de la ligne de rabotage. En d'autres termes, la quantité choisie pour un processus donné est celle qui lui permettra de produire suffisamment de produits pour satisfaire la demande, tout en respectant la disponibilité des ressources.

L'algorithme 6 décrit plus en détail le fonctionnement de l'heuristique de choix de valeur pour les quantités de processus. En premier lieu, il récupère la liste des produits *products* qui seront en sortie lorsque le processus dont on veut fixer une quantité sera exécuté. Pour ce faire, il fait appel à la fonction *GetOutputProducts* (ligne 2 de l'algorithme 6). Puis, il appelle la fonction *GetDemands* qui retourne la liste des commandes *demands* des produits *products* (ligne 3 de l'algorithme 6). Ensuite, il calcule quantité *maxQuantity* nécessaire pour couvrir toutes les commandes *demands* (lignes 4 à 13 de l'algorithme 6). Enfin, l'algorithme retourne le minimum entre la quantité *maxQuantity* et la quantité *capacity* autorisée à la période en cours de planification (lignes 14 à 16 de l'algorithme 6).

5.4 Configuration des instances industrielles

Évaluer les performances d'une stratégie de retour-arrière dans l'exploration d'un arbre de recherche n'a de sens que lorsque les heuristiques de choix de variable et de choix de valeur sont bonnes. Autrement dit, il est inutile d'essayer d'évaluer une stratégie de retour-arrière si les heuristiques employées n'ont pas une probabilité de faire le bon choix strictement supérieure à 50%. Prenons le cas extrême où le choix de variable et de valeur se font de manière complètement aléatoire. Dans ce cas, l'arbre de recherche serait construit totalement au hasard et n'aurait aucune structure. Par conséquent, la stratégie de retour-arrière risque d'avoir des performances totalement différentes sur une même instance de problème.

Comme on dispose de très bonnes heuristiques pour le problème de planification des opérations du rabotage, les expérimentations ont été réalisées sur des instances de ce problème issues de l'industrie du bois. Ces données industrielles proviennent de Maibec, une compagnie canadienne de production du bois d'œuvre. Ces instances se constituent des éléments suivants :

- 42 à 46 périodes (quarts de travail)
- 485 à 709 produits
- 113 à 134 processus

– 6 à 9 familles (ou campagnes) de produits

Cependant, afin de pouvoir comparer les algorithmes selon le temps de calcul nécessaire pour atteindre l'optimalité, la taille des instances a dû être réduite. Plus particulièrement, au lieu de planifier sur un horizon de 42 périodes, comme spécifié dans les données industrielles, cet horizon a été réduit à 5 périodes.

5.5 Résultats et discussion

Cette section présente les résultats des différentes expérimentations menées sur des instances du problème de la planification du rabotage du bois. *Reinforcement Learning Backtracking Strategy* (RLBS) est comparé à *Adaptive Discrepancy Search* (ADS) et à *Limited Discrepancy Search* (LDS) sur cinq instances du problème de la planification du rabotage issues de l'industrie. Ensuite, il est comparé à LDS, *Impact-Based Search* (IBS) et *Depth-First Search* (DFS) sur une instance jouet du même problème. Enfin, l'impact du taux d'apprentissage de RLBS sur sa performance est mesuré. Pour ce faire, RLBS est évalué selon différentes valeurs du taux d'apprentissage sur les instances industrielles.

5.5.1 Comparaison de RLBS, ADS, et LDS

RLBS a été évalué en utilisant un taux d'apprentissage ayant pour valeur $\alpha = 0.9$ et a été comparé à ADS et à LDS. Les figures 5.3 à 5.7 présentent les résultats pour les différentes instances du problème. On remarque sur ces figures que pour chacune des instances RLBS arrive a trouver la solution optimale avant LDS. On voit aussi qu'il trouve la meilleure solution avant ADS sur 4 instances (figures 5.3, 5.3, 5.3, et 5.3). Le tableau 5.1 montre la réduction du temps de calcul, mesurée par le nombre de nœuds visités, dont l'exploration a besoin pour trouver la solution optimale. Dans ce tableau, RLBS est comparé à ADS en prenant LDS comme référence. Les 3 premières lignes présentent le temps de calcul nécessaire à LDS, ADS, et RLBS, pour trouver la meilleure solution des différentes instances. La quatrième et la cinquième ligne présentent la réduction du temps de calcul apportée par ADS et RLBS, respectivement, par rapport à LDS. RLBS réduit le temps de calcul pour chacune des instances à une moyenne de 84.45%. Cette réduction peut aller jusqu'à 99.63% (cas 5). Quant à ADS, il apporte une réduction qui peut atteindre 83.2% (cas 5) et qui est en moyenne égale à 31%.

Une réduction d'un certain pourcentage p d'un algorithme a_1 par rapport à un autre algorithme a_2 signifie que a_1 est capable de générer un plan de production qui minimise le plus possible les retards de livraison en une durée de temps p fois plus courte que celle nécessaire à l'algorithme a_2 pour générer un plan équivalent [1]. Autrement dit, cette réduction signifie que l'algorithme a_1 est $\frac{1}{1-p}$ fois plus rapide que l'algorithme a_2. Par exemple, on peut dire qu'en moyenne RLBS est 6.43 fois plus rapide que LDS sur le problème de rabotage du bois étudié (car $\frac{1}{1-0.8445} = 6.43$). De la même manière, on remarque que ADS est en moyenne 1.45 fois plus rapide que LDS (car $\frac{1}{1-0.31} = 1.45$). Par conséquent, on déduit que RLBS est 4.43 fois plus rapide que ADS sur le problème étudié (car $\frac{6.43}{1.45} = 4.43$).

La réduction des retards de livraison peut facilement se traduire en des gains énormes pour les entreprises, que ce soit en termes de profits (chiffre d'affaire) ou de fidélisation de clients (image sur le marché).

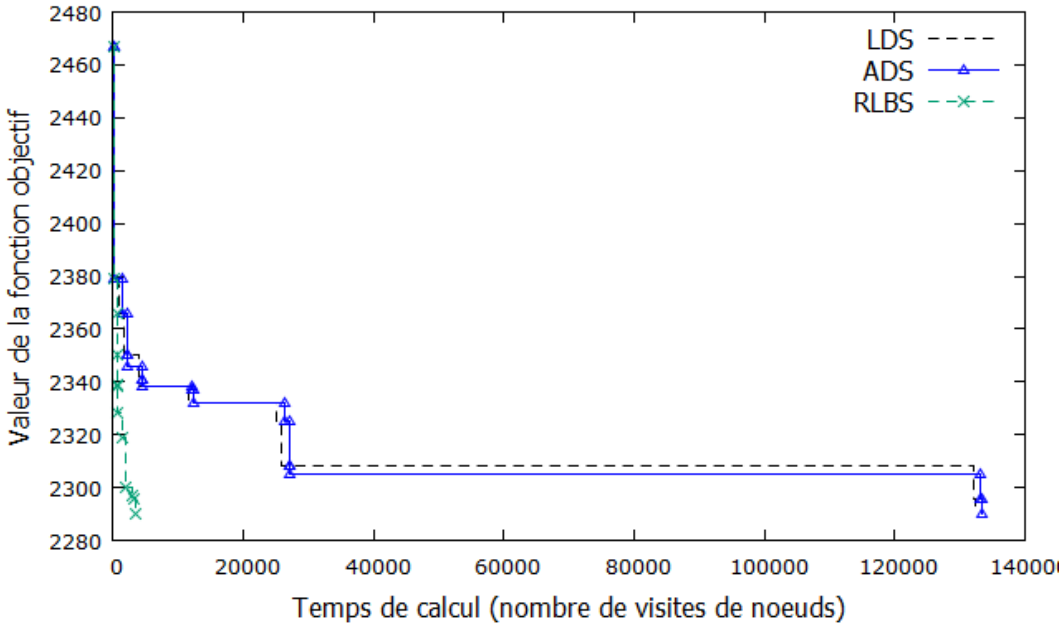

FIGURE 5.3: La valeur de la fonction objectif en fonction du temps de calcul de LDS, ADS, et RLBS pour le cas #1

1. Deux plans sont considérés équivalents quand ils ont la même valeur pour les retards de livraison

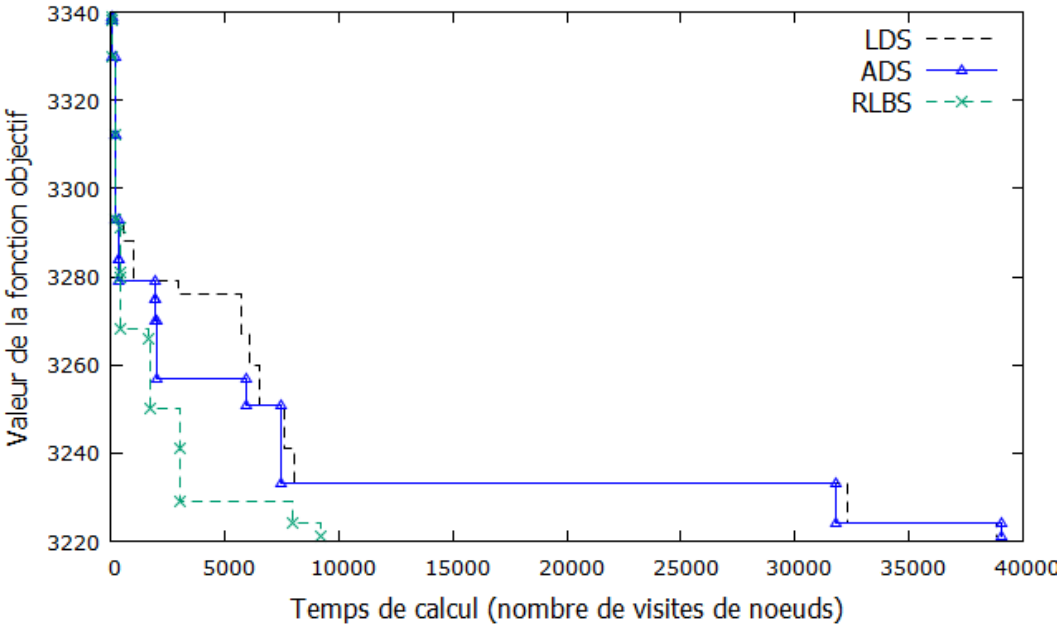

FIGURE 5.4: La valeur de la fonction objectif en fonction du temps de calcul de LDS, ADS, et RLBS pour le cas #2

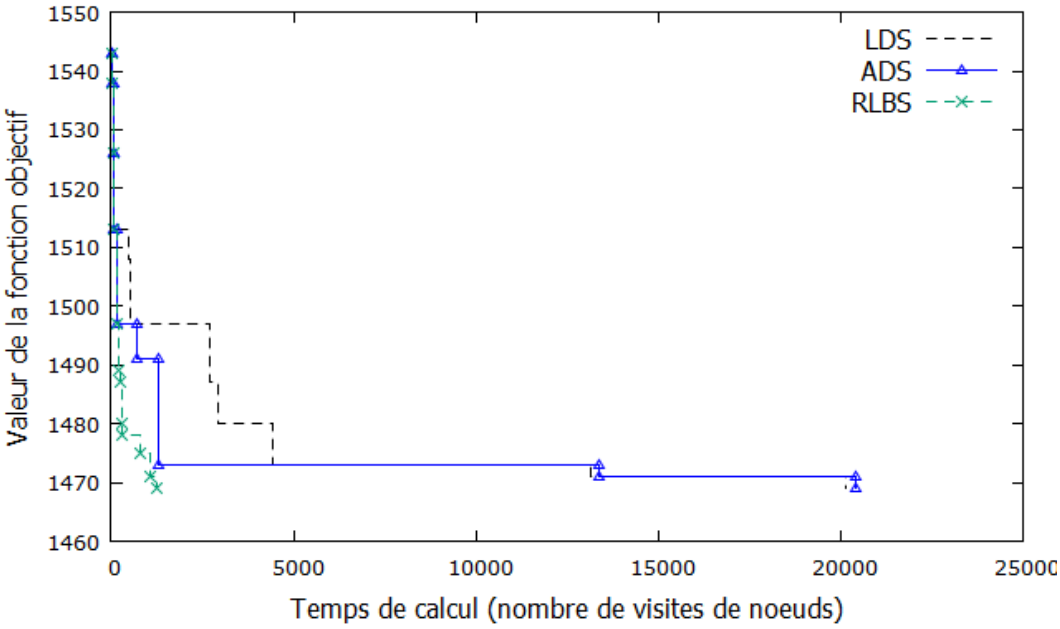

FIGURE 5.5: La valeur de la fonction objectif en fonction du temps de calcul de LDS, ADS, et RLBS pour le cas #3

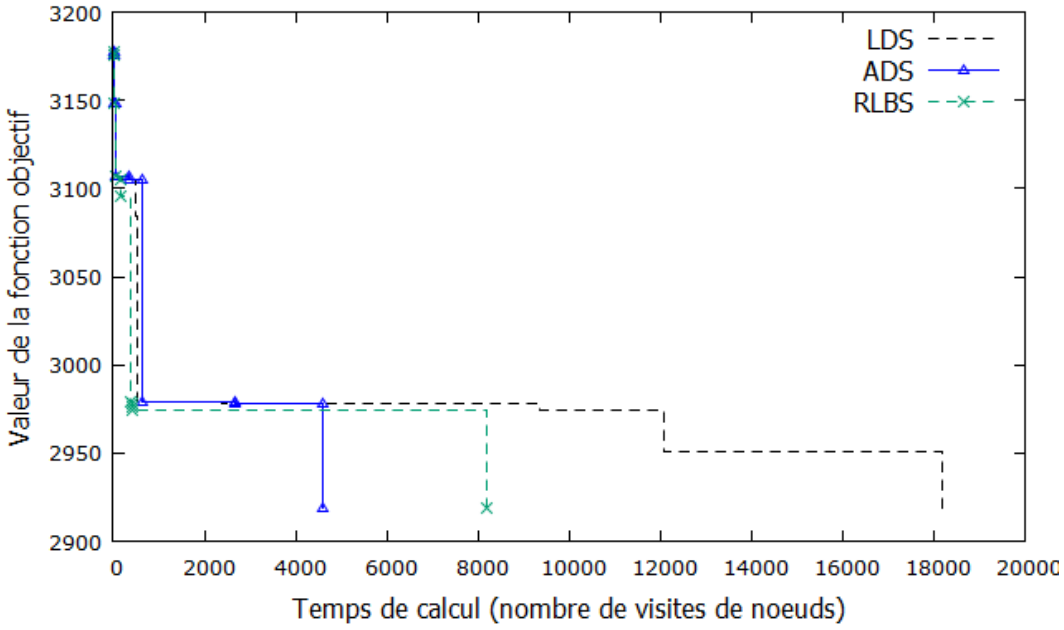

FIGURE 5.6: La valeur de la fonction objectif en fonction du temps de calcul de LDS, ADS, et RLBS pour le cas #4

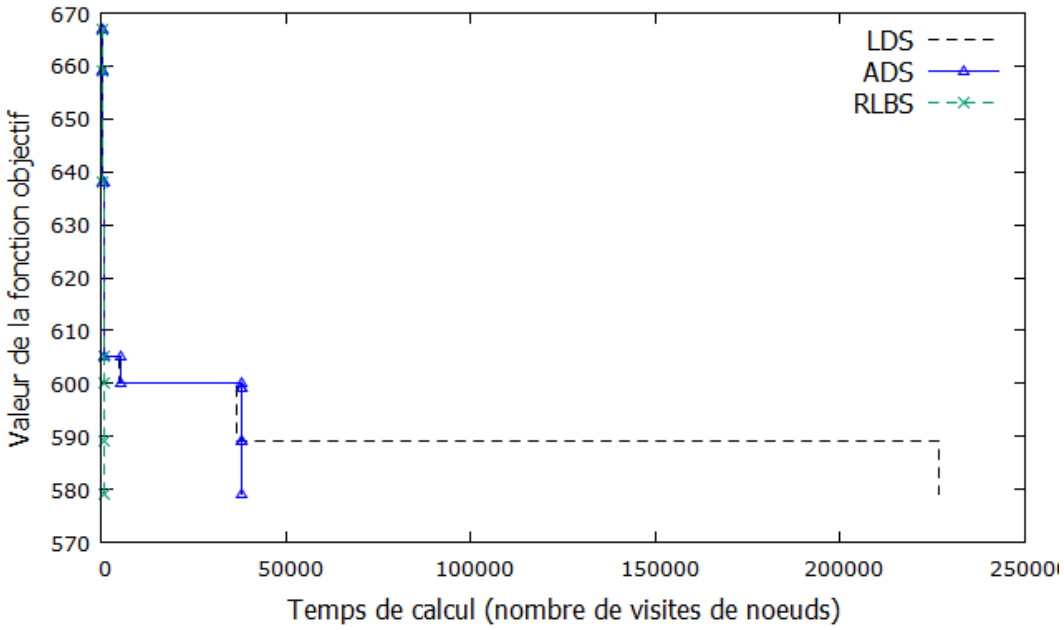

FIGURE 5.7: La valeur de la fonction objectif en fonction du temps de calcul de LDS, ADS, et RLBS pour le cas #5

Dans le contexte industriel, on n'a souvent pas le temps d'attendre que l'exploration

	Cas 1	**Cas 2**	**Cas 3**	**Cas 4**	**Cas 5**	**Moyenne**
LDS	132282	38861	20133	18197	226666	87227.8
ADS	133352	39079	20436	4612	38079	47111.6
RLBS	3334	9173	1233	8213	839	4558.4
Réduction ADS	$\uparrow$ 0.80%	$\uparrow$ 0.56%	$\uparrow$ 1.50%	$\downarrow$ 74.65%	$\downarrow$ 83.20%	$\downarrow$ 31.00%
Réduction RLBS	$\downarrow$ 97.48%	$\downarrow$ 76.40%	$\downarrow$ 93.88%	$\downarrow$ 54.87%	$\downarrow$ 99.63%	$\downarrow$ 84.45%

TABLE 5.1: Temps de calcul nécessaire pour trouver la meilleure solution (RLBS vs. ADS)

trouve la solution optimale. Ceci est dû aux tailles des instances industrielles qui sont beaucoup trop grandes pour que les ordinateurs de nos jours puissent les résoudre à l'optimalité dans un temps raisonnable. Dès lors, il s'avère intéressant de considérer non seulement la réduction de temps de calcul pour trouver la solution optimale, mais aussi la réduction de temps de calcul nécessaire pour obtenir des solutions de qualités intermédiaires. Pour calculer la réduction de temps de calcul pour obtenir une solution de qualité intermédiaire, on compare la durée de temps nécessaire à un algorithme pour trouver une solution d'une qualité quelconque q à celle nécessaire à un autre algorithme pour obtenir une solution de même qualité q, et on déduit la réduction apportée par l'un des algorithme par rapport à l'autre. Le tableau 5.2 montre, pour chaque instance, la moyenne du temps de calcul nécessaire pour obtenir une solution d'une qualité donnée. Les 3 premières lignes présentent les moyennes de temps de calcul pour trouver une solution d'une qualité quelconque de LDS, ADS, et RLBS. La quatrième et la cinquième ligne montrent les réductions apportées par ADS et RLBS, respectivement, par rapport à LDS. On remarque que RLBS apporte une amélioration de 81.53% en moyenne. Cette amélioration peut atteindre 98.35%, comme dans le cas 5. En d'autres termes, RLBS est en moyenne 5.41 fois plus rapide que LDS pour trouver une solution d'une qualité quelconque (car $\frac{1}{1-0.8153} = 5.41$). De même, on remarque que ADS réduit le temps de calcul pour trouver une solution d'une qualité quelconque de 38.03% en moyenne, ce qui le rend 1.61 fois plus rapide que LDS (car $\frac{1}{1-0.3803} = 1.61$). Donc, on peut déduire que RLBS est 3.36 fois plus rapide que ADS pour trouver une solution d'une qualité intermédiaire donnée (car $\frac{5.41}{1.61} = 3.36$).

On rappelle que ces améliorations représentent de considérables opportunités d'augmentation de profit pour les entreprises, qui peuvent se traduire en un meilleur positionnement sur le marché en général.

	Cas 1	**Cas 2**	**Cas 3**	**Cas 4**	**Cas 5**	**Moyenne**
LDS	3154147	724758	154336	964493	2716473	1542841.4
ADS	2897119	625379	98702	352457	848643	964460
RLBS	114795	175934	17146	498167	44561	170120.6
Réduction ADS	↓ 8.15%	↓ 13.71%	↓ 36.05%	↓ 63.46%	↓ 68.76%	↓ 38.03%
Réduction RLBS	↓ 96.36%	↓ 75.73%	↓ 88.89%	↓ 48.34%	↓ 98.35%	↓ 81.53%

TABLE 5.2: Moyenne de temps de calcul pour trouver une solution d'une qualité quelconque (RLBS vs. ADS)

5.5.2 Comparaison de RLBS et IBS

Même en réduisant l'horizon de planification (nombre de périodes), résoudre les instances industrielles à l'optimalité est infaisable en temps raisonnable, à cause de leur grande taille. IBS a été roulé sur ces instances-ci et n'était pas capable de trouver de bonnes solutions même après une très longue durée (plus de 150 heures). Pour cette raison, des problèmes jouets ont été générés pour conduire cette expérimentation. RLBS a été comparé à DFS, LDS, et IBS sur un problème jouet. La raison pour laquelle on a voulu comparé RLBS avec IBS et que ce dernier se base sur de l'apprentissage pour explorer l'arbre de recherche, tout comme RLBS. On rappelle que IBS apprend l'impact que possède chaque affectation d'une valeur à une variable sur la réduction de l'espace de recherche, et choisit à chaque fois le couple (variable,valeur) avec le plus grand impact (qui promet la plus grande réduction de l'espace de recherche).

La figure 5.8 montre que IBS a la pire performance et se fait même dépassé par DFS. Ceci est dû au fait que IBS ne peut pas tirer profit des heuristiques employées (de bonnes heuristiques spécifiques au problème de la planification du rabotage) pour les autres algorithmes. IBS effectue une sélection adaptative des variables et des valeurs, et ne peut utiliser aucune autre heuristique. RLBS présente la meilleure performance et trouve la solution optimale en premier. Il est suivi de LDS, puis DFS, et enfin IBS en dernière position. D'un côté, cette expérimentation nous prouve que les heuristiques employées pour la résolution de ce problème sont très bonnes. Ceci est supporté par le fait que même DFS surpasse largement IBS, principalement grâce à ces heuristiques-là. D'un autre, elle montre qu'on peut produire des performances encore meilleures en combinant de bonnes heuristiques à de l'apprentissage. Cette remarque est soutenue par le fait que RLBS (qui utilise de bonnes heuristiques et se base sur l'apprentissage) est en tête de liste avec la meilleure performance comparé aux autres algorithmes.

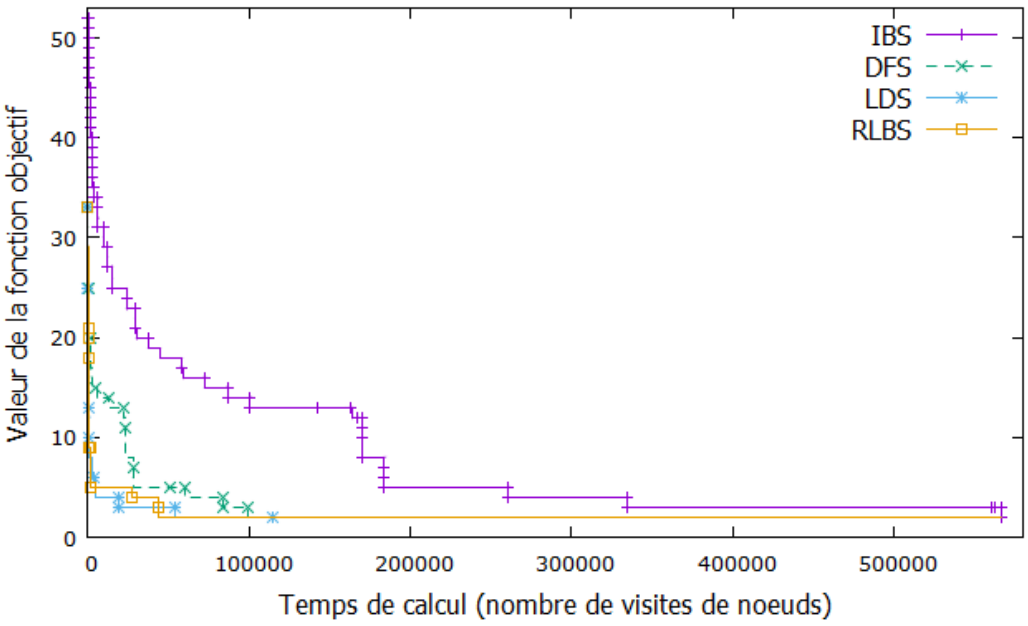

FIGURE 5.8: La valeur de la fonction objectif en fonction du temps de calcul de RLBS, LDS, IBS et DFS sur un problème jouet

5.5.3 Impact du taux d'apprentissage

Le taux d'apprentissage joue un rôle important dans la performance de RLBS. Selon sa valeur, RLBS peut ou peut ne pas explorer certaines zones de l'arbre de recherche avant d'autres. En d'autres termes, dépendamment de la valeur du taux d'apprentissage, l'ordre dans lequel l'algorithme découvre les différentes solutions change. Ceci a un impact direct sur la vitesse à laquelle RLBS peut trouver la solution optimale. Le tableau 5.3 montre la réduction du temps de calcul pour obtenir la solution optimale par RLBS versus LDS, pour différentes valeurs du taux d'apprentissage α. Cette expérimentation a été effectuée sur 5 cas industriels. La moyenne de réduction du temps de calcul pour chaque valeur de α, sur les 5 cas, a été calculée. On aperçoit que RLBS est en moyenne plus rapide à trouver la solution optimale avec un taux d'apprentissage $\alpha = 0.2$ pour le problème de la planification du rabotage étudié. Par contre, en termes de réduction moyenne par rapport à LDS du temps de calcul pour trouver une solution de qualité quelconque, RLBS est en moyenne plus rapide avec une valeur de taux d'apprentissage $\alpha = 0.4$ (tableau 5.4).

Ceci dit, on remarque sur les tableaux 5.3 et 5.4 que même à sa pire performance ($\alpha = 0.9$), RLBS demeure largement meilleure que LDS et apporte une amélioration

α	Cas 1	Cas 2	Cas 3	Cas 4	Cas 5	Moyenne
0.1	↓ 97.32%	↓ 96.56%	↓ 92.97%	↓ 52.96%	↓ 99.63%	↓ 87.89%
0.2	↓ 97.36%	↓ 96.48%	↓ 93.06%	↓ 53.18%	↓ 99.63%	↓ **87.94%**
0.3	↓ 97.38%	↓ 95.44%	↓ 90.26%	↓ 53.64%	↓ 99.63%	↓ 87.27%
0.4	↓ 97.40%	↓ 95.27%	↓ 93.47%	↓ 53.90%	↓ 99.63%	↓ 87.93%
0.5	↓ 97.43%	↓ 86.87%	↓ 93.67%	↓ 54.09%	↓ 99.63%	↓ 86.34%
0.6	↓ 97.47%	↓ 80.54%	↓ 93.68%	↓ 54.28%	↓ 99.63%	↓ 85.12%
0.7	↓ 97.47%	↓ 78.64%	↓ 93.77%	↓ 54.39%	↓ 99.63%	↓ 84.78%
0.8	↓ 97.48%	↓ 77.24%	↓ 93.82%	↓ 54.70%	↓ 99.63%	↓ 84.58%
0.9	↓ 97.48%	↓ 76.40%	↓ 93.88%	↓ 54.87%	↓ 99.63%	↓ 84.45%

TABLE 5.3: Réduction du temps de calcul pour obtenir la meilleure solution par RLBS vs. LDS pour différentes valeurs du taux d'apprentissage α

α	Cas 1	Cas 2	Cas 3	Cas 4	Cas 5	Moyenne
0.1	↓ 96.17%	↓ 90.60%	↓ 91.05%	↓ 45.89%	↓ 98.36%	↓ 84.41%
0.2	↓ 96.20%	↓ 90.54%	↓ 91.07%	↓ 46.29%	↓ 98.36%	↓ 84.49%
0.3	↓ 96.23%	↓ 90.10%	↓ 90.34%	↓ 46.76%	↓ 98.36%	↓ 84.36%
0.4	↓ 96.27%	↓ 90.03%	↓ 91.18%	↓ 47.15%	↓ 98.36%	↓ **84.60%**
0.5	↓ 96.29%	↓ 86.93%	↓ 91.23%	↓ 47.38%	↓ 98.36%	↓ 84.04%
0.6	↓ 96.32%	↓ 84.17%	↓ 91.21%	↓ 47.57%	↓ 98.36%	↓ 83.53%
0.7	↓ 96.34%	↓ 83.07%	↓ 91.23%	↓ 47.86%	↓ 98.36%	↓ 83.37%
0.8	↓ 96.36%	↓ 80.97%	↓ 91.25%	↓ 48.18%	↓ 98.36%	↓ 83.02%
0.9	↓ 96.36%	↓ 75.73%	↓ 88.89%	↓ 48.35%	↓ 98.36%	↓ 81.54%

TABLE 5.4: Réduction moyenne du temps de calcul pour trouver une solution de qualité quelconque par RLBS vs. LDS pour différentes valeurs du taux d'apprentissage α

moyenne de plus de 80%. Autrement dit, peu importe la valeur du taux d'apprentissage choisie, RLBS reste en moyenne au moins 5 fois plus rapide que LDS sur le problème étudié. On constate aussi que pour toutes les instances, sauf le cas 2, la réduction apportée par RLBS par rapport à LDS (aussi bien pour obtenir la solution optimale que pour trouver une solution de qualité quelconque) est proportionnelle à la valeur du taux d'apprentissage α utilisée. Donc, on peut présumer que définir $\alpha = 0.9$ comme valeur par défaut du taux d'apprentissage serait une bonne façon de procéder en général [2].

Les figures de 5.9 à 5.13 montrent la variation de la valeur de la fonction objectif en fonction du temps de calcul de RLBS avec différentes valeurs du taux d'apprentissage pour les cinq cas industriels.

2. Dans le cas d'un problème pour lequel on ne connaît par les performances de RLBS en fonction du taux d'apprentissage α.

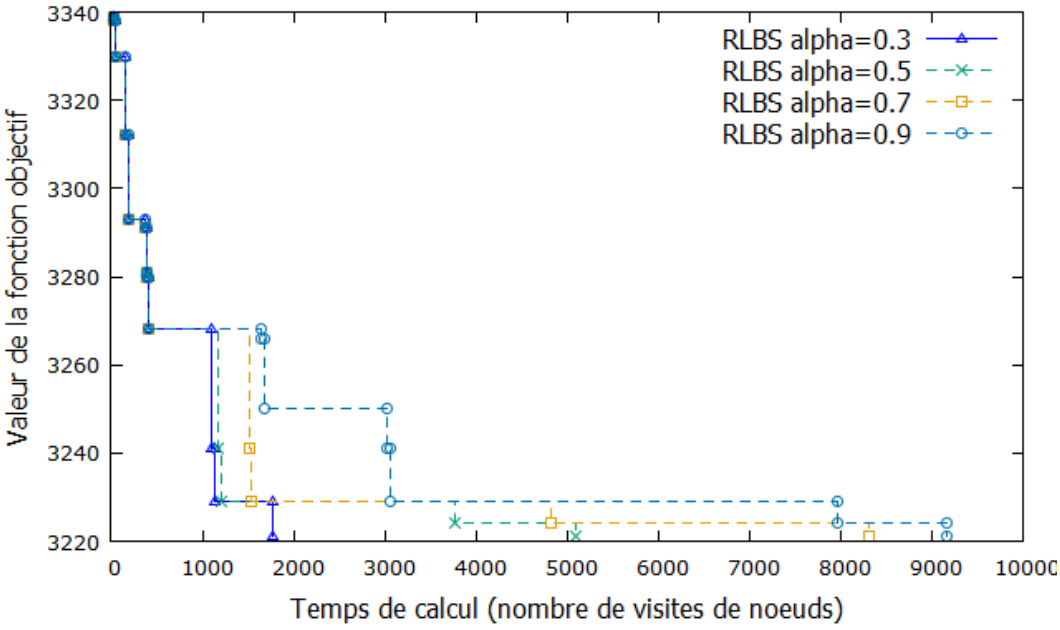

FIGURE 5.10: La valeur de la fonction objectif en fonction du temps de calcul de RLBS avec différentes valeurs de alpha pour le cas #2

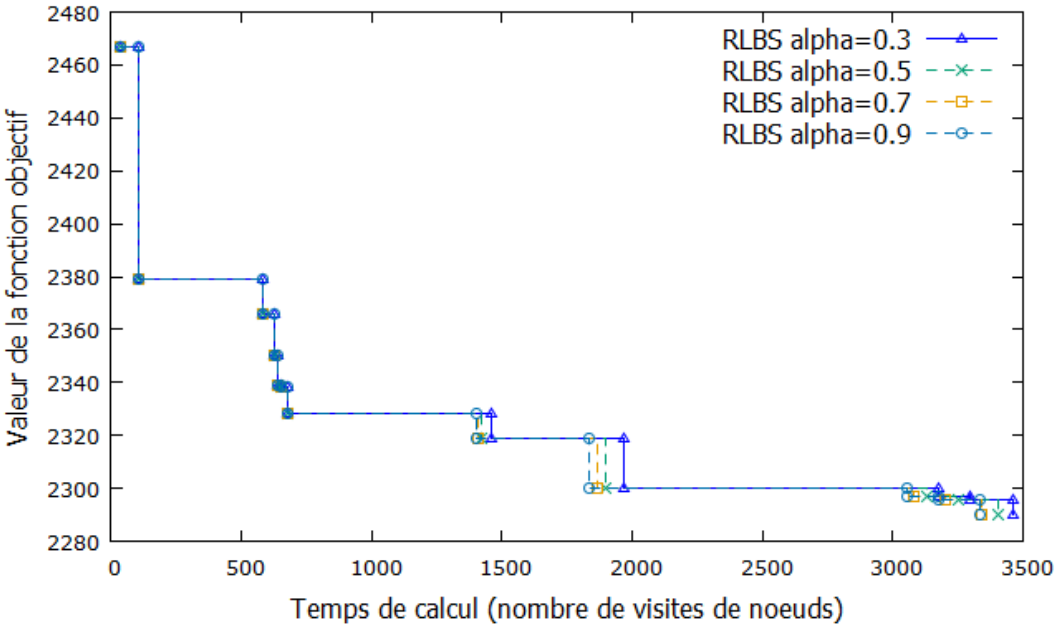

FIGURE 5.9: La valeur de la fonction objectif en fonction du temps de calcul de RLBS avec différentes valeurs de alpha pour le cas #1

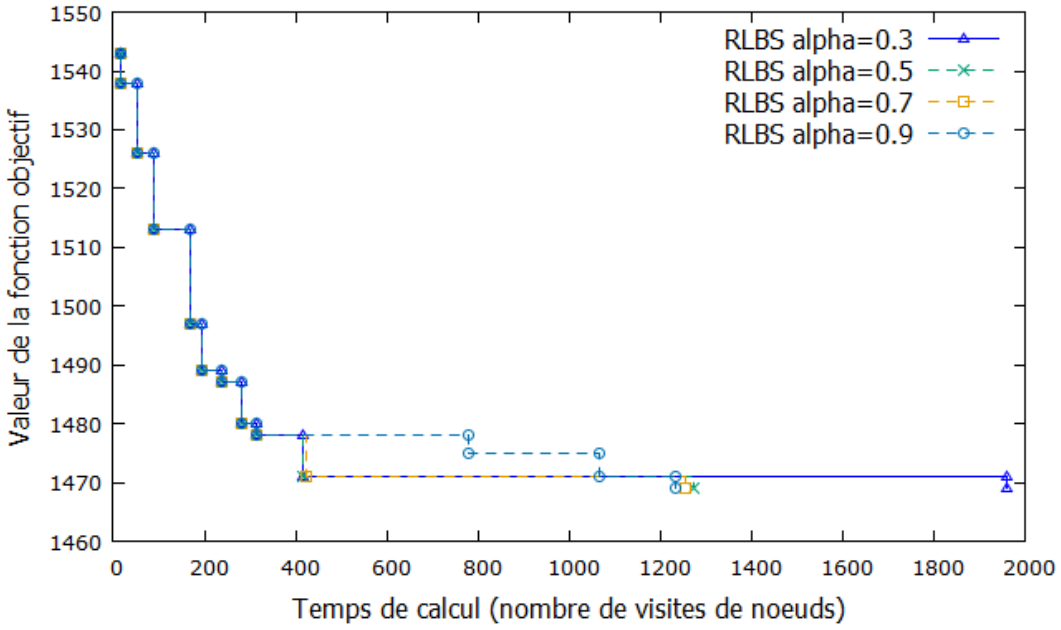

FIGURE 5.11: La valeur de la fonction objectif en fonction du temps de calcul de RLBS avec différentes valeurs de alpha pour le cas #3

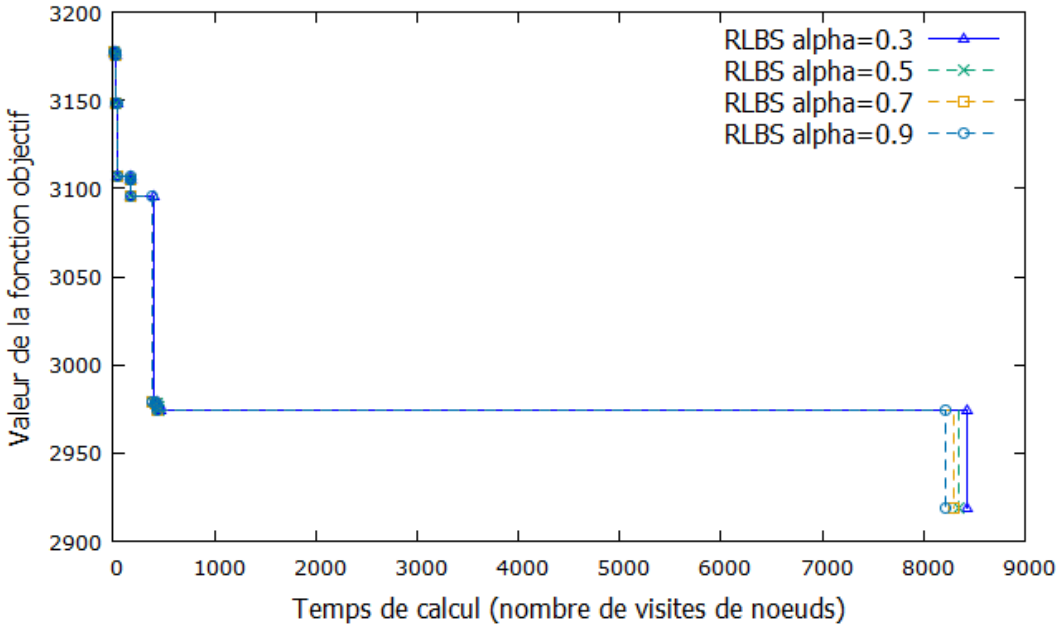

FIGURE 5.12: La valeur de la fonction objectif en fonction du temps de calcul de RLBS avec différentes valeurs de alpha pour le cas #4

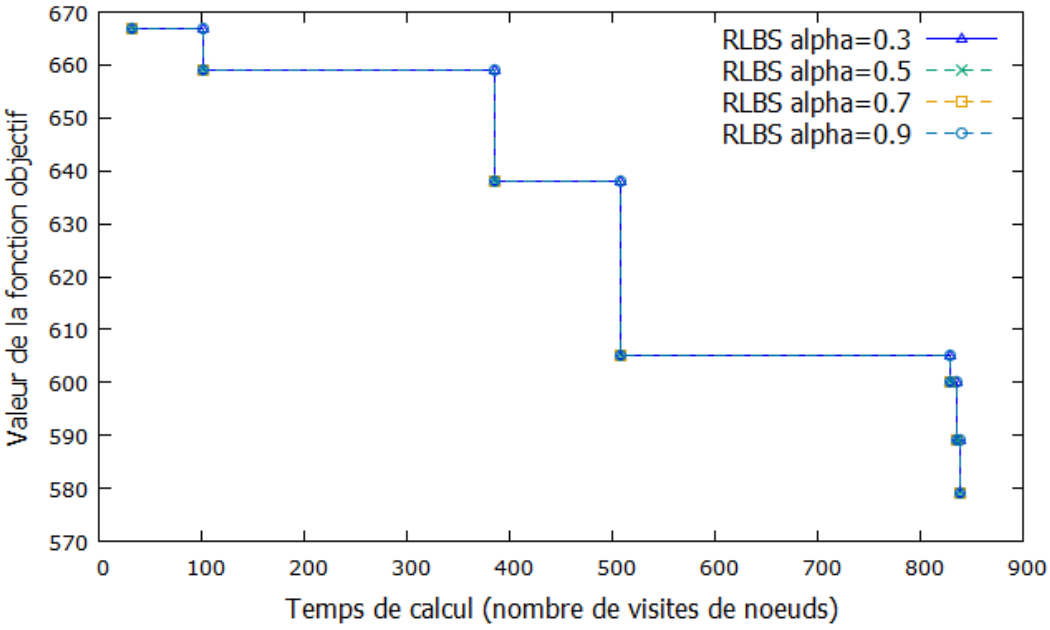

FIGURE 5.13: La valeur de la fonction objectif en fonction du temps de calcul de RLBS avec différentes valeurs de alpha pour le cas #5

Chapitre 6

Conclusion

Ce mémoire propose un mécanisme d'apprentissage basé sur l'apprentissage par renforcement qui permet au solveur d'apprendre dynamiquement à effectuer des retours-arrière de manière efficace. Ceci a été évalué sur plusieurs instances réelles d'un problème industriel difficile de planification et ordonnancement, qui est uniquement résolu en utilisant des heuristiques de branchement spécifiques. La stratégie de retour-arrière proposée améliore grandement la performance de l'exploration en comparaison avec ADS et LDS. Ceci grâce au mécanisme d'apprentissage qui permet d'identifier les nœuds vers lesquels il est le plus profitable d'effectuer des retours-arrière.

L'utilisation de données réelles issues de l'industrie a montré la grande valeur de cette approche. Cependant, des questions demeurent concernant la performance de l'algorithme sur des problèmes pour lesquels on ne connaît pas de bonnes heuristiques de branchement. Dans cette situation, est-ce que ceci vaut la peine d'essayer d'identifier un meilleur nœud candidat au retour-arrière ?

D'autres chercheurs [Xu et al., 2009] ont déjà utilisé l'apprentissage par renforcement pour apprendre à des heuristiques de choix de variable et de choix de valeur. À notre connaissance, RLBS est la première utilisation de l'apprentissage par renforcement pour apprendre une stratégie de retour-arrière. Cet algorithme offre l'avantage d'être utilisable peu importe la stratégie de branchement employée. Ceci le rend très efficace pour les problèmes pour lesquels une bonne stratégie de branchement est connue. Cependant, rien n'empêche de combiner les deux (apprendre une stratégie de branchement et une stratégie de retour-arrière). La combinaison de la stratégie de retour-arrière adaptative et des stratégies de branchement adaptatives ferait l'objet d'une autre opportunité de recherche intéressante.

Bibliographie

Ethem Alpaydin. *Introduction to Machine Learning*. MIT press, 2004.

Krzysztof Apt. *Principles of constraint programming*. Cambridge University Press, 2003.

Roman Barták. Incomplete depth-first search techniques : a short survey. In *Proceedings of the 6th Workshop on Constraint Programming for Decision and Control, Ed. Figwer J*, pages 7–14, 2004.

Andrew G Barto. *Reinforcement Learning : An Introduction*. MIT press, 1998.

J Christopher Beck and Laurent Perron. Discrepancy-bounded depth first search. In *Proceedings of International Conference on Integration of Artificial Intelligence and Operations Research techniques in Constraint Programming (CPAIOR)*, pages 8–10, 2000.

Justin A Boyan and Andrew W Moore. Using prediction to improve combinatorial optimization search. In *Sixth International Workshop on Artificial Intelligence and Statistics*, 1997.

Rina Dechter and Itay Meiri. Experimental evaluation of preprocessing techniques in constraint satisfaction problems. In *Proceedings of the 11th International Joint Conference on Artificial Intelligence-Volume 1*, pages 271–277. Morgan Kaufmann Publishers Inc., 1989.

S Epstein and Smiljana Petrovic. Learning to solve constraint problems. In *Proceedings of ICAPS-07 Workshop on Planning and Learning*. Citeseer, 2007.

Luca Maria Gambardella, Marco Dorigo, et al. Ant-q : A reinforcement learning approach to the traveling salesman problem. In *Proceedings of the International Conference on Machine Learning*, pages 252–260, 1995.

Jonathan Gaudreault. *Algorithmes pour la prise de décision distribuée en contexte hiérarchique*. PhD thesis, École Polytechnique de Montréal, 2009.

Jonathan Gaudreault, Pascal Forget, Jean-Marc Frayret, Alain Rousseau, Sebastien Lemieux, and Sophie D'Amours. Distributed operations planning in the softwood lumber supply chain : models and coordination. *International Journal of Industrial Engineering : Theory Applications and Practice*, 17 :168–189, 2010.

Jonathan Gaudreault, Jean-Marc Frayret, Alain Rousseau, and Sophie D'Amours. Combined planning and scheduling in a divergent production system with co-production : A case study in the lumber industry. *Journal of Computers and Operations Research*, 38(9) :1238–1250, 2011.

Jonathan Gaudreault, Gilles Pesant, J Frayret, and Sophie D'Amours. Supply chain coordination using an adaptive distributed search strategy. *Journal of Systems, Man, and Cybernetics, Part C : Applications and Reviews, IEEE Transactions on*, 42(6) : 1424–1438, 2012.

William D Harvey. *Nonsystematic backtracking search*. PhD thesis, Citeseer, 1995.

William D Harvey and Matthew L Ginsberg. Limited discrepancy search. In *Proceedings of Joint Conference on Artificial Intelligence (1)*, pages 607–615, 1995a.

William D Harvey and Matthew L Ginsberg. Limited discrepancy search. In *Proceedings of International Joint Conference on Artificial Intelligence (1)*, pages 607–615, 1995b.

Leslie Pack Kaelbling, Michael L Littman, and Andrew W Moore. Reinforcement learning : A survey. *Journal of arXiv preprint cs/9605103*, 1996.

Wafa Karoui, Marie-José Huguet, Pierre Lopez, and Wady Naanaa. YIELDS : A yet improved limited discrepancy search for CSPs. In *Proceedings of Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems*, pages 99–111. Springer, 2007.

Richard E Korf. Improved limited discrepancy search. In *Proceedings of Association for the Advancement of Artificial Intelligence/Innovative Applications of Artificial Intelligence, Vol. 1*, pages 286–291, 1996.

Vipin Kumar, K Ramesh, and V Nageshwara Rao. Parallel best-first search of state-space graphs : A summary of results. In *Proceedings of the seventh AAAI Conference on Artificial Intelligence*, volume 88, pages 122–127. Citeseer, 1988.

Manuel Loth, Michele Sebag, Youssef Hamadi, and Marc Schoenauer. Bandit-based search for constraint programming. In *Principles and Practice of Constraint Programming*, pages 464–480. Springer, 2013a.

Manuel Loth, Michele Sebag, Youssef Hamadi, Marc Schoenauer, and Christian Schulte. Hybridizing constraint programming and monte-carlo tree search : Application to the job shop problem. In *Learning and Intelligent Optimization*, pages 315–320. Springer, 2013b.

Victor V Miagkikh and William F Punch. An approach to solving combinatorial optimization problems using a population of reinforcement learning agents. In *Proceedings of the Genetic and Evolutionary Computation Conference*, volume 2, pages 1358–1365, 1999a.

Victor V Miagkikh and William F Punch. Global search in combinatorial optimization using reinforcement learning algorithms. In *Proceedings of Evolutionary Computation, 1999. CEC 99. Proceedings of the 1999 Congress on*, volume 1. IEEE, 1999b.

Pragnesh Jay Modi. *Distributed constraint optimization for multiagent systems*. PhD thesis, University of Southern California, 2003.

Thierry Moisan, Jonathan Gaudreault, and Claude-Guy Quimper. Parallel discrepancy-based search. In *Principles and Practice of Constraint Programming*, pages 30–46. Springer, 2013.

Thierry Moisan, Claude-Guy Quimper, and Jonathan Gaudreault. Parallel depth-bounded discrepancy search. In *Integration of AI and OR Techniques in Constraint Programming*, pages 377–393. Springer, 2014.

Robert Moll, Andrew G Barto, Theodore J Perkins, and Richard S Sutton. Learning instance-independent value functions to enhance local search. In *Advances in Neural Information Processing Systems*. Citeseer, 1998.

Malek Mouhoub and Bahareh Jafari. Heuristic techniques for variable and value ordering in csps. In *Proceedings of the 13th Annual Conference on Genetic and Evolutionary Computation*, pages 457–464. ACM, 2011.

Alexander Nareyek. Choosing search heuristics by non-stationary reinforcement learning. In *Metaheuristics : Computer decision-making*, pages 523–544. Springer, 2004.

Christos H Papadimitriou and Kenneth Steiglitz. *Combinatorial optimization : algorithms and complexity*. Courier Corporation, 1998.

Nicolas Prcovic and Bertrand Neveu. Recherche arborescente restreinte un sous-espace prometteur.

Philippe Refalo. Impact-based search strategies for constraint programming. In *Proceedings of Principles and Practice of Constraint Programming–CP 2004*, pages 557–571. Springer, 2004.

Wheeler Ruml. *Adaptive tree search*. PhD thesis, Citeseer, 2002a.

Wheeler Ruml. Heuristic search in bounded-depth trees : Best-leaf-first search. In *Working Notes of the AAAI-02 Workshop on Probabilistic Approaches in Search*, 2002b.

SJ Russell, P Norvig, and E Davis. Artificial intelligence : a modern approach. *Prentice Hall series in artificial intelligence*, 2010.

Horst Samulowitz and Roland Memisevic. Learning to solve qbf. In *AAAI*, volume 7, pages 255–260, 2007.

Weiming Shen, Lihui Wang, and Qi Hao. Agent-based distributed manufacturing process planning and scheduling : a state-of-the-art survey. *Journal of Systems, Man, and Cybernetics, Part C : Applications and Reviews, IEEE Transactions on*, 36(4) : 563–577, 2006.

Barbara M Smith and Stuart A Grant. Trying harder to fail first. *Research ReportT Series - University of Leeds School of Computer Studies LU SCS RR*, 1997.

Katia Sycara, Steven F Roth, Norman Sadeh, and Mark S Fox. Distributed constrained heuristic search. *Systems, Man and Cybernetics, IEEE Transactions on*, 21(6) :1446–1461, 1991.

Robert Moll Andrew G Barto Theodore, J Perkins, and Richard S Sutton. Learning instance-independent value functions to enhance local search.

Omkar Tilak and Snehasis Mukhopadhyay. Decentralized and partially decentralized reinforcement learning for distributed combinatorial optimization problems. In *Machine Learning and Applications (ICMLA), 2010 Ninth International Conference on*, pages 389–394. IEEE, 2010.

Vasileios Vasilikos and Michail G Lagoudakis. Optimization of heuristic search using recursive algorithm selection and reinforcement learning. *Annals of Mathematics and Artificial Intelligence*, 60(1-2) :119–151, 2010.

Nageshwara Rao Vempaty, Vipin Kumar, and Richard E Korf. Depth-first vs best-first search. In *Proceedings of the ninth National Conference on Artificial Intelligence-Volume 1*, pages 434–440. AAAI Press, 1991.

Toby Walsh. Depth-bounded discrepancy search. In *Proceedings of International Joint Conference on Artificial Intelligence*, volume 97, pages 1388–1393, 1997.

Tony Wauters, Katja Verbeeck, Greet Vanden Berghe, and Patrick De Causmaecker. A multi-agent learning approach for the multi-mode resource-constrained project scheduling problem. In *Proceedings of 8th Int. Conf. on Autonomous Agents and Multiagent Systems (AAMAS 2009)*, pages 1–8, 2009.

Tony Wauters, Katja Verbeeck, G Vanden Berghe, and Patrick De Causmaecker. Learning agents for the multi-mode project scheduling problem. *Journal of the Operational Research Society*, 62(2) :281–290, 2011.

Chelsea C White III and Douglas J White. Markov decision processes. *Journal of European Journal of Operational Research*, 39(1) :1–16, 1989.

Lin Xu, Frank Hutter, Holger H Hoos, and Kevin Leyton-Brown. Satzilla : Portfolio-based algorithm selection for sat. *J. Artif. Intell. Res.(JAIR)*, 32 :565–606, 2008.

Yuehua Xu, David Stern, and Horst Samulowitz. Learning adaptation to solve constraint satisfaction problems. *Journal of Learning and Intelligent Optimization (LION)*, 2009.

Wei Zhang and Thomas G Dietterich. A reinforcement learning approach to job-shop scheduling. In *Proceedings of International Joint Conferences on Artificial Intelligence*, volume 95, pages 1114–1120. Citeseer, 1995.

Wei Zhang and Thomas G Dietterich. Solving combinatorial optimization tasks by reinforcement learning : A general methodology applied to resource-constrained scheduling. *Journal of Artificial Intelligence Reseach*, 1 :1–38, 2000.