Étude de méthodes arborescentes de Monte-Carlo pour un problème de déplacement de pièces dans un atelier d'assemblage

Valentin Antuori, Emmanuel Hébrard, Marie-José Huguet, S Essodaigui, A Nguyen

Étude de méthodes arborescentes de Monte-Carlo pour un problème de déplacement de pièces dans un atelier d'assemblage

V. Antuori[1,2], E. Hébrard[1,3], MJ. Huguet[1], S. Essodaigui[2], A. Nguyen[2]

[1] LAAS-CNRS, Université de Toulouse, CNRS, INSA, Toulouse, France
[2] Renault, France
[3] ANITI, Université de Toulouse, France

{vantuori,hebrard,huguet}@laas.fr
{siham.essodaigui,alain.nguyen}@renault.com

Résumé

La recherche arborescente Monte-Carlo (MCTS) connaît un développement important pour la résolution de problèmes d'optimisation combinatoire, en particulier, lorsque les mécanismes d'inférence ne passent pas à l'échelle, ou sont trop faibles pour réduire l'espace de recherche. Dans cet article, nous appliquons la méthode MCTS à un problème de voyageur de commerce avec fenêtres de temps et contraintes de capacité, issu d'une chaîne de montage dans la construction automobile. Des adaptations du MCTS de base sont proposées et analysées via une étude expérimentale afin de dégager des pistes génériques pour la résolution de problème d'optimisation combinatoire.

Mots-clés

Recherche arborescente de Monte-Carlo, Ordonnancement, Voyageur de Commerce.

Abstract

The development of the Monte-Carlo Tree Search (MCTS) for combinatorial optimisation has grown significantly, in particular when inference mechanisms do not scale, or are too weak to reduce the search space. In this paper we apply the MCTS to a traveling salesman problem with time windows and a capacity constraint from an assembly line in car manufacturing. Adaptations of the basic MCTS are proposed and analysed via an experimental study in order to identify generic components for the resolution of combinatorial optimisation problems.

Keywords

Monte-Carlo Tree Search, Scheduling, Traveling Salesman Problem

1 Introduction

Pour de nombreux problèmes combinatoires, explorer de façon exhaustive un arbre de recherche serait trop coûteux. En effet, l'espace de recherche peut rarement être suffisamment réduit par des méthodes d'inférence exactes pour que ces méthodes passent à l'échelle.

La méthode Monte-Carlo Tree Search (MCTS) [4, 3] offre une stratégie générique pour s'attaquer à ces problèmes en ne nécessitant que très peu de connaissances spécifiques du problème. Cet algorithme a été proposée dans le contexte des jeux et a contribué aux récents progrès de l'intelligence artificielle, par exemple pour le Go [11].

Dans cette méthode, des simulations guident l'expansion de l'arbre de recherche et maintiennent un compromis entre des phases d'exploitation et d'exploration. Plus précisément, une simulation est l'action d'extension d'un noeud jusqu'à un état final. Cette extension est généralement faite par une heuristique "randomisée". Le résultat obtenu par simulation est ensuite rétro-propagé à ce noeud et à tous ses ancêtres jusqu'à la racine. C'est cette information qui sert de guide pour les itérations futures de l'algorithme via un mécanisme de bandit manchot.

On trouve dans la littérature plusieurs applications de la méthode à des problèmes d'optimisation combinatoire. Dans [7], les auteurs appliquent l'algorithme à un problème de voyageur de commerce avec drone, ils considèrent leur problème comme un jeu où une victoire est l'amélioration de la borne supérieure (amélioration de la meilleure solution déjà obtenue). [2] applique la méthode à deux problèmes stochastiques de gestion des incendies et de contrôle de réseau de files d'attente. Enfin la méthode est aussi appliquée à des problèmes de type ordonnancement [5, 9]. Différentes hybridations de la méthode MCTS avec des méthodes d'optimisation combinatoire ont été proposées. Dans [10], l'algorithme de bandits manchots du MCTS a été utilisé en particulier dans la programmation mixte en nombres entiers (MIP), en remplaçant les simulations de Monte-Carlo par une borne inférieure de la relaxation linéaire. Dans [6], la méthode MCTS a été combinée avec la programmation par contraintes (CP). Pour obtenir une méthode exacte et permettre des redémarrages, les auteurs ont dû modifier de manière significative plusieurs aspects de l'algorithme. La combinaison avec la satisfiabilité booléenne (SAT) proposée par [8] est plus fidèle aux principes de MCTS, cependant, dans ce cas elle n'inclut pas les caractéristiques des solveurs SAT modernes (apprentissage de clause, VSIDS, etc.).

Nous présentons une étude du MCTS appliquée à un problème industriel de type *voyageur de commerce*.
Pour ce problème, étudié dans [1], les méthodes de propagation de contraintes couplé à une heuristique adéquate obtiennent rapidement une solution réalisable pour une grande majorité d'instances industrielles. Néanmoins, sur des instances plus contraintes [1] l'obtention d'une telle heuristique n'est pas triviale
Pour cela une méthode d'apprentissage par renforcement a été utilisée, et l'heuristique ainsi découverte a permis, en conjonction avec d'autres méthodes combinatoires, de surpasser les méthodes employés par l'entreprise. Cependant, pour certains horizons temporels cette approche ne permet pas de résoudre l'ensemble des instances. Au vu de l'essence même de la méthode MCTS qui consiste à guider la recherche par de nombreuse simulations heuristiques, nous nous intéressons à son application pour notre problème, en s'appuysant sur nos précédents travaux.
Nous proposons alors de définir un cadre générique de l'application de la méthode pour les problèmes d'optimisation combinatoires, et d'utiliser ce cadre pour résoudre notre problème. Nous proposons egalement trois *composants additionnels*, que l'on ne retrouve habituellement pas dans cette méthode, qui aident à la résolution du problème étudié tout en restant génériques.

2 Description du problème

Le problème étudié consiste à déplacer un ensemble de m composants de véhicule à travers une chaîne d'assemblage d'un constructeur automobile, de leur point de production vers le lieu de consommation. Chaque composant est produit et consommé par deux machines uniques, et le déplacement de l'une à l'autre machine est effectué au moyen de chariot, spécifique à chaque composant. Lorsqu'un chariot est plein à son lieu de production, un opérateur doit l'amener à son lieu de consommation, et de manière symetrique, lorsqu'un chariot est vide au lieu de consommation, il doit être ramener au lieu de production correspondant. Un cycle de production est le temps c_i nécessaire pour produire (resp. consommer) le composant i, ou plutôt pour remplir (resp. vider) le chariot correspondant. La fin d'un cycle de production marque le début du suivant, et il y a donc $n_i = \frac{H}{c_i}$ jusqu'au à l'horizon temporel H. Pour chacun des cycles k de chaque composant i, il y a deux opérations de collectes, et deux opérations de livraison : la collecte pe_i^k et la livraison de_i^k du chariot vide du point de consommation vers le le point de production, et la collecte pf_i^k et la livraison df_i^k du chariot plein de la production vers la consommation. Chaque opération a prend un temps p_a pour être effectué, et le temps de déplacement entre l'opération a est b prend $D_{a,b}$ unités de temps.
On note A l'ensemble de toute les opérations. Le problème est alors de trouver une permutation σ des $|A| = n$ opérations, respectants les contraintes suivantes [2] :

Routes : Pour tout $1 < j \leq n$, l'operation $\sigma(j)$ doit étant donné une date de début $s_{\sigma,j}$ (et une date de fin $e_{\sigma,j} = s_{\sigma,j} + p_{\sigma(j)}$) prendre en compte la durée et le temps de trajet : $s_{\sigma,j} \geq s_{\sigma,j-1} + p_{\sigma(j-1)} + D_{\sigma(j-1),\sigma(j)}$ (et $s_{\sigma,1} = 0$).

Fenêtre de temps : Une opération a de la période k du composant i a une date de disponibiilité $r_a = (k-1)c_i$ et une date d'échéance $d_a = kc_i$, avec $r_a \leq s_a$ et $e_a \leq d_a$.

Précédences : Les collectes doivent précéder leur livraison.

Longueur du train : L'opérateur peut accrocher les chariots les uns aux autres de manière à former un train de chariot, mais la longueur total du train ne doit pas excéder une longueur donnée $T_{\max}$.

Ce problème est un cas particulier du problème de collecte et livraison à un véhicule avec contrainte de capacité, et fenêtres de temps. Cependant il n'y a pas de fonction objectif à optimiser, et au contraire la faisabilité est difficile. De plus, les cycles de production-consommation impliquent une structure très particulière : les quatres opérations de chaque composant à chaque cycle doivent être effectué durant la même fenêtre de temps, et ceci se répètant à chaque cycle.
Nous travaillons ici avec une relaxation du problème original, en relachant les contraintes sur les dates d'échéances, nous cherchons à minimiser le retard maximum :

$$L(\sigma) = \max(0, \max_{1 \leq j \leq |\sigma|} (e_{\sigma(j)} - d_{\sigma(j)}))$$

Pour ce faire nous devons ajouter une contrainte de précédence additionnelle : pour toute paire d'opération a, b des cycles $k < p$ respectivement, d'un même composant, on a necessairement a qui doit précéder b.

3 MCTS en optimisation combinatoire

3.1 Principes généraux

La méthode MCTS explore un espace de recherche avec une structure arborescente et peut être appliquée à l'optimisation combinatoire avec seulement quelques hypothèses faibles sur le modèle du problème.
Dans l'arbre de recherche développé, un noeud correspond à un état du problème. Chaque noeud est associé à un ensemble de décision menant à des noeuds enfants dans l'arbre. Le but est de trouver un chemin du noeud racine (état initial) à un état final (une solution du problème). A chaque solution, il est possible de calculer la valeur de la fonction objectif et sans perte de généralité, nous supposons ici que nous cherchons une solution de valeur minimale. La méthode MCTS repose sur une succession de quatres phases : (1) une phase de sélection, généralement basé sur un bandit manchot, pour guider l'expansion de l'arbre, et assurer un compromis entre l'exploitation (sélectionner le noeud le plus prometteur) et l'exploration (acquerir de nouvelles connaissances sur d'autres parties de l'arbre) ; (2) une

1. Un jeu de données académiques s'inspirant de données industrielles a été développé pour palier le manque de données réelles.
2. Voir [1] pour une description plus détaillé

phase d'expansion ou des noeuds sont ajoutés dans l'arbre ; (3) une phase de simulation, également appelé "rollout" ou "playout", utilisé pour produire rapidement un chemin d'un nœud donné vers un état final (par exemple basé sur un échantillonnage aléatoire) ; et (4) une mise à jour des informations sur les noeuds pour prendre en compte la nouvelle simulation, informations utiles aux phases de sélections futurs.

Notations. Soit $\mathcal{A}$ un ensemble de décisions, un état $\sigma \in \mathcal{A}^*$ représente une séquence de décisions et $p(\sigma)$ l'état parent de σ. De plus, on note $\sigma; a$ l'état atteint en appliquant la décision a dans l'état σ, et $\mathcal{A}(\sigma)$ l'ensemble des décisions possibles dans l'état σ. A chaque étape de l'algorithme, de nouveaux états sont ajoutés dans un arbre $\mathcal{T}$ qui ne contient initiallement que l'état initial en tant que racine. Pour chacun des états de l'arbre le triplet suivant est mémorisé : $\{N(\sigma), Pr(\sigma), V(\sigma)\}$ où $N(\sigma)$ est le nombre de fois que σ a été visité, $Pr(\sigma)$ est une probabilité a priori (une préférence issue de connaissances spécifique sur le problème) de prendre la décision qui mène à l'état σ à partir de son état parent $p(\sigma)$, et $V(\sigma)$ est une évaluation du noeud calculée à partir des résultats des simulations issues de σ. Habituellement en optimisation combinatoire, l'évaluation du noeud est la moyenne des valeurs de la fonction objectifs des solutions obtenues par les simulations des itérations antérieurs passées par ce noeud. Nous proposons une autre évaluation également basé sur la valeur de la fonction objectif dans la section 3.2

Sélection. La phase de sélection commence au noeud racine, et se termine lorsqu'une feuille de $\mathcal{T}$ est atteinte. À chaque noeud σ, une décision est sélectionnée en fonction des valeurs de chaque triplet des noeuds fils :

$$a^* = \arg\max_{a \in \mathcal{A}(\sigma)} \tilde{V}(\sigma; a) + c * U(\sigma; a) \quad (1)$$

où $\tilde{V}(\sigma; a)$ est le terme d'exploitation, $U(\sigma; a)$ le terme d'exploration, et c est un paramètre qui représente le compromis entre les deux termes. Le terme d'exploitation $\tilde{V}(\sigma; a)$ peut être directement $V(\sigma)$, $V(\sigma)$ étant basé sur la valeur de la fonction objectif, cette valeur peut être très differente d'une instance d'un problème à l'autre, ainsi pour faciliter le réglage du paramètre c, nous proposons de normaliser cette valeur dans $[-1, 1]$:

$$\tilde{V}(\sigma; a) = \begin{cases} 2 * \frac{V^+(\sigma) - V(\sigma;a)}{V^+(\sigma) - V^-(\sigma)} - 1 & \text{if } N(\sigma; a) > 0 \\ 0 & \text{otherwise} \end{cases} \quad (2)$$

Avec :

$$V^+(\sigma) = \max_{a \in \mathcal{A}(\sigma), N(\sigma;a) > 0} V(\sigma; a)$$
$$V^-(\sigma) = \min_{a \in \mathcal{A}(\sigma), N(\sigma;a) > 0} V(\sigma; a)$$

Le terme d'exploration $U(\sigma; a)$ est celui utilisé dans [11] :

$$U(\sigma) = Pr(\sigma) \frac{\sqrt{N(p(\sigma))}}{N(\sigma) + 1} \quad (3)$$

Le compromis effectué vise à choisir l'action qui maximise $\tilde{V}(\sigma; a)$ auquel s'ajoute un bonus qui diminue à chaque visite afin de promouvoir l'exploration. La probabilité a priori biaise l'exploration par des connaissances initiales sur l'état (si elles existent). Ce processus se répète jusqu'à arriver à une feuille de l'arbre $\mathcal{T}$.

Expansion. Soit σ le noeud retourné par la phase de sélection. Pour toute décision $a \in \mathcal{A}(\sigma)$, la phase d'expansion ajoute un noeud enfant à σ et initialise chaque triplet $\{N(\sigma; a), Pr(\sigma; a), V(\sigma; a)\}$. Le nombre de visites $N(\sigma; a)$ et la valeur $V(\sigma; a)$ de chaque enfant de σ sont fixés à 0. Il est également possible d'initialiser les probabilité à priori $Pr(\sigma; a)$, si cette distribution n'est pas connue, la distribution uniforme $1/|\mathcal{A}(\sigma)|$ peut être utilisée. Dans cette phase d'expansion, nous pouvons empêcher la création de certains noeuds menant nécessairement à un échec,par exemple si l'on peut évaluer la borne inférieure d'un noeud que l'on veut ajouter, et que celle ci dépasse la borne supérieure (meilleure solution connue). Il se peut alors que la phase d'expansion supprime un noeud lorsqu'il n'a plus d'enfant possible.

Simulation Le noeud σ obtenu par la phase de sélection est étendu jusqu'à une solution en appliquant généralement un échantillonnage aléatoire des actions possibles jusqu'à un état final. Par exemple, on peut utiliser la distribution de probabilité donnée par $Pr(\sigma; a) \mid a \in \mathcal{A}(\sigma)$ ou toute heuristique gloutonne "randomisée" adaptée au problème traité.

Rétropropagation Pour chaque noeud σ' traversé durant la phase de sélection de la racine à σ, le nombre de visites est incrémenté et la valeur $V(\sigma')$ est alors mise à jour en prenant en compte le résultat de la simulation. Pour des problèmes de type jeux, le résultat d'une simulation est généralement -1 (partie perdue), 1 (partie gagnée) ou 0 (partie nulle). En revanche, pour un problème d'optimisation nous ne disposons pas d'un tel résultat, mais de la valeur de fonction objectif. Dans [7], les auteurs considèrent une amélioration de la meilleur solution courante comme une partie gagnée, pour se rapprocher du MCTS utilisé dans les jeux. Bien souvent, le résultat d'une simulation est la valeur de la fonction objectif elle même [5, 9], et comme évoqué precedemment, dans ce cas, $V(\sigma)$ représente alors l'espérance de l'objectif au noeud σ.

3.2 Adaptations

Dans un contexte d'optimisation combinatoire, différentes adaptations de la méthode MCTS peuvent être considérées.

Evaluation. Dans un problème d'optimisation combinatoire, nous disposons généralement d'une borne inférieure sur la valeur de la fonction objectif qui croit de façon monotone pour chaque décision gloutonne prise. Par conséquent, l'évolution de cette borne pendant la simulation nous permet d'avoir une bonne idée de la qualité de la décision initiale. De plus, dans une procédure gloutonne, on peut conjecturer qu'à mesure que la procédure avance, la corrélation entre la qualité de la décision initiale, et l'accroissement de la borne inférieure diminue. On propose alors

d'évaluer la valeur $V(\sigma)$ d'un noeud σ comme l'espérance de la somme des accroissements marginaux de la borne inférieure obtenus après ce noeud, pondéré par un coefficient γ, décroissant exponentiellement. L'algorithme 1 décrie la procédure d'évaluation et de rétropropagation. Cette procédure prend en entrée la séquence des accroissements marginaux de la borne inférieure obtenue par la simulation R (R_i est alors égal à la difference de borne inférieur entre l'étape i et $i-1$ lors de la simulation), et calcule une évaluation pour le noeud sélectionné par la procédure de sélection. Ensuite tous les noeuds σ jusqu'à la racine sont mis à jour pour que $V(\sigma)$ reflète l'espérance de son impact dans l'accroissement de la borne inférieure future. Dans l'algorithme, $LB(\sigma)$ est l'évaluation de la borne inférieure au noeud σ.

Algorithme 1 : Algorithme de rétropropagation

Données : R : séquence des accroissements marginaux de la borne inférieure obtenu par la simulation, σ : noeud sélectionné par la phase de sélection, γ : facteur de décroissance

1 `// Somme pondérée des accroissement marginaux de la simulation`

2 $val \leftarrow \sum_{i=1}^{|R|} \gamma^{i-1} R_i$ `// Rétropropagation jusqu'à la racine incluse`

3 **répéter**

4 $\quad val \leftarrow \gamma * val + LB(\sigma) - LB(p(\sigma))$

5 $\quad N(\sigma) \leftarrow N(\sigma) + 1$

6 $\quad V(\sigma) \leftarrow V(\sigma) + \frac{val - V(\sigma)}{N(\sigma)}$

7 $\quad \sigma \leftarrow p(\sigma)$

8 **jusqu'à** $\sigma = Nil$;

Compromis dynamique exploitation-exploration. Les expériences préliminaires nous montre Il est possible d'adapter dynamiquement le paramètre contrôlant le compromis entre l'exploration et l'exploitation, en fonction de la profondeur de l'arbre, afin de forcer l'exploitation sur la partie haute de l'arbre au fur et à mesure de sa croissance. En notant $d(\mathcal{T})$ la hauteur de l'arbre $\mathcal{T}$, le coefficient de compromis lors de l'itération t de la phase de sélection s'écrit alors :

$$\beta^{d(\mathcal{T})-t} * c \tag{4}$$

où $\beta < 1$ est un paramètre. Ce compromis dynamique est similaire à une phase de "commit" au noeud racine qui est généralement effectué dans le MCTS, [2, 7] mais faite de manière plus douce : plus la hauteur de l'arbre augmente, moins la décision à la racine est remise en cause.

Approfondissement de simulations. Afin d'améliorer la phase de simulations, nous proposons d'utiliser un Depth First Search (DFS) à budget limité en guise de processus d'intensification. Le principe est de remplacer la procédure gloutonne par un DFS, et de décider d'un budget à accorder lors du premier échec détecté lors de la simulation, par exemple lorsque la borne inférieure dépasse la borne supérieure. Pour cela, on évalue la pertinence de cette première branche (du noeud sélectionné, jusqu'à l'échec). Si cette branche est prometteuse alors nous accordons un budget de recherche plus ou moins important en fonction de la qualité de la branche et un DFS est lancé. Si elle ne l'est pas, alors nous basculons vers la procédure gloutonne pour étendre cette solution partielle jusqu'à une feuille, sans DFS. Lorsque le budget alloué est totalement dépensé deux cas sont possibles. Soit on a obtenu une solution améliorante, et cette solution est retenue. Soit on sélectionne la meilleure solution partielle (selon le même critère que pour définir le budget) et on l'étend jusqu'à une solution par la procédure gloutonne. Plus de détail sur ce mécanisme sont donnés dans la section 4, notre proposition n'étant pas assez générique pour figurer dans cette section.

4 MCTS pour le problème de collecte et livraison

Dans cette section nous donnons des précisions sur la manière dont nous utilisons la méthode pour notre problème de collecte et de livraison

Dans ce problème, la racine de l'arbre consiste en une séquence vide d'opération et une décision consiste à choisir une opération de collecte ou de livraison que l'on ajoute en fin de la séquence d'opérations déjà réalisées. Comme l'objectif *in fine* est d'obtenir une solution sans aucun retard, lors de la phase d'expansion un noeud dont un des enfants est en retard sera supprimé car cette branche ne mènera à aucune solution. Durant les phases de simulation et de rétro-propagation, l'accroissement de la borne inférieure correspond à l'accroissement du retard maximal lorsque l'on ajoute une opération.

Enfin, nous utilisons une heuristique gloutonne stochastique dédiée, configurée par apprentissage par renforcement pour les "rollouts" [1]. Cette procédure détermine une distribution de probabilité sur l'ensemble des opérations disponibles dans un état donné. C'est cette même distribution qui est également utilisé lors de la phase d'expansion, pour obtenir la distribution de probabilité sur les enfants d'un noeud, et qui est ensuite utilisé lors de la phase de sélection (3). Dans un état donné σ, on décrit chaque opération $a \in \mathcal{A}(\sigma)$ par un vecteur de 4 critères :

1. *L'urgence* d'une opération : : $lst(a,\sigma) - \max(r_a, e_{\sigma,|\sigma|} + D_{\sigma(|\sigma|),a})$, avec $lst(a,\sigma)$ la date de début au plus tard de l'opération a pour satisfaire sa date d'échéance, compte tenu des opérations déjà présente, et des contraintes de précédences ;
2. Le *temps de trajet/temps d'attente* de l'opération : $\max(D_{\sigma(|\sigma|),a}, (r_a - e_{\sigma(|\sigma|)}))$;
3. La *longueur* du chariot ;
4. Le *type* de l'opération, 1 pour les opérations de collecte et 0 pour les opérations de livraison.

Chaque critère est ensuite normalisé entre $[0,1]$, et une fonction de *fitness* pour chaque opération est alors obtenu

par une combinaison linéaire de ces critères. On utilise ensuite la fonction `softmax` pour obtenir une distribution de probabilité sur les opérations disponibles, avec un paramètre β qui contrôle la forme de cette distribution. Pour une valeur "basse" de ce paramètre, le probabilité de choisir la meilleur action sera plus grande, quand une valeur plus "haute" permet plus de diversité. Nous utilisons $\beta = 0.005$ pour la simulation ce qui nous permet d'obtenir de bonne solution tout en maintenant la diversité et $\beta = 0.1$ pour la probabilité à priori. Le vecteur de poids de la combinaison linéaire a été calculé par apprentissage par renforcement sur l'ensemble des instances généré aléatoirement, avec un horizon temporel d'une journée (l'horizon temporel intermédiaire de nos catégorie d'instance).

Pour le DFS, nous cherchons à la fois une séquence sans retard, mais également à améliorer notre borne supérieur. Nous prenons en compte ces deux critères pour décider d'un nombre de "backtrack" entre 0 et $\mathcal{B}$ autorisés.

Lors de la procédure gloutonne initiale, nous stoppons l'heuristique lorsque notre retard courant dépasse notre meilleur solution. Si cette condition n'arrive pas, alors nous avons trouvé une solution améliorante, et notre budget est maximum ($\mathcal{B}$) afin de chercher une solution voisine améliorant à nouveau la fonction objectif. Sinon, on se base sur la position ϕ du premier retard détectable dans la séquence d'opérations (i.e., le rang auquel la borne inférieure devient positive) pour définir le budget. Soit ϕ^* la position du premier retard dans la meilleure solution obtenue jusqu'à présent. Le budget est accordé comme suit :

$$\begin{cases} \mathcal{B} & \text{si } \phi \geq \phi^* \\ \mathcal{B}(\frac{\phi^* - \phi}{\phi^* - \alpha * \phi^*})^2 & \text{si } \phi^* > \phi > \alpha * \phi^* \\ 0 & \text{sinon} \end{cases} \quad (5)$$

Avec $\alpha \leq 1$, un paramètre qui permet de définir un seuil. Notre DFS s'appuie sur l'heuristique utilisée dans la procédure gloutonne, à la fois pour l'ordre d'exploration, mais aussi pour limiter le facteur de branchement. En effet, cette procédure détermine une distribution de probabilité sur les actions disponibles, et un paramètre de seuil permet de se limiter aux seules actions les plus probables. Enfin, il est généralement admis qu'une stratégie de *restart* lors du DFS permet d'obtenir de meilleure solution. Suivant ce principe nous permettons alors au DFS (dans la limite de son budget) de restaurer l'état du noeud sélectionné par la phase de sélection au bout d'un certain nombre d'échecs, ce nombre va croître géométriquement à chaque *restart*, et sera réinitialisé pour la prochaine itération de la méthode.

5 Expérimentation

Nous proposons d'évaluer la méthode MCTS sur notre problème de collecte et de livraison de pièce. Nous considérons un ensemble d'instances partitionné en quatre classes (A, B, C, D), générées aléatoirement sur la base des caractéristiques du problème industriel. Chaque classe se différencie par le nombre de composant, allant de 15 composants pour les instances de la classe A, jusqu'à 30 pour celles de la classe D) Pour chacune des classes, nous disposons de 10 instances déclinés en trois horizons temporel ("Shift", "Jour", "Semaine"), pour un total de 120 instances au total [3].

TABLE 1 – Valeur des paramètres

γ	0.9977
c	1
β	0.995
α	0.9
$\mathcal{B}$	50000
redémarrage (base)	100
redémarrage (facteur)	1.2

Nous proposons 5 versions différentes de la méthode MCTS intégrant progressivement les adaptations proposées. Nous appelons `MCTS` la méthode qui utilise la valeur de la fonction objectif directement, sans utiliser l'accroissement marginal pondéré de la borne inférieure, sans DFS, et sans décroissance du paramètre c de compromis exploitation/exploration. La méthode `DSMI` ajoute l'utilisation de l'accroissement marginal de la borne inférieure. Les méthodes `DSMI+DFS` et `DSMI+DFS+DC` ajoutent toutes deux l'intensification de la simulation par DFS, la seconde ajoute également le mécanisme de décroissance du paramètre c comme expliqué dans la section 3.2. Enfin la colonne `DSMI+SAT-DFS+DC` est une variation de `DSMI+DFS+DC`, pour laquelle nous fixons à 1 la borne supérieure pour la partie simulation, ce qui a pour effet de transformer la phase de simulation avec DFS en résolution d'un problème de satisfaction (un échec sera levé dès qu'un retard est rencontré). Cependant une vraie borne supérieure est tout de même disponible, la meilleure branche étant étendue jusqu'à une feuille à la fin du DFS. Les valeurs des paramètres utilisés dans ces méthodes sont donnés dans la table 1. Le paramètre γ est le coefficient de pondération de l'accroissement marginal (algorithme 1). Les paramètres c et β sont respectivement le compromis d'exploitation-exploration et sa variation (équation 4). Les paramètres α et $\mathcal{B}$ sont utilisés pour fixer le budget de "backtrack" de la procédure de DFS (équation 5). Ce choix de valeurs s'appuie sur des expérimentations préliminaires, non détaillées ici.

Nous nous comparons avec deux précédentes méthodes décrites dans [1], l'une basée sur un solveur de programmation par contraintes (`CP`) pour la version satisfaction du problème et l'autre sur une recherche locale multistart (`GRASP`) pour la version en minimisation du retard.

Les résultats sont présentés dans la table 2 où #S représente le nombre d'instances résolues et L_{max} la valeur de l'objectif moyennée sur les 10 instances de la classe lancées 10 fois chacune. Une limite de temps d'une heure a été utilisée pour chacune des méthodes. Enfin l'expérience a été menée sur un cluster de calcul composé de processeurs Xeon E5-2695 v3 @ 2.30GHz et Xeon E5-2695 v4 @ 2.10GHz.

3. Instances disponibles sur `https://gepgitlab.laas.fr/vantuori/trolley-pb`

TABLE 2 – Comparaison des méthodes

Cl	H	CP	GRASP		MCTS		DSMI		DSMI+DFS		DSMI+DFS+DC		DSMI+SAT-DFS+DC	
		#S	#S	L_{max}	#S	L_{max}	#S	L_{max}	#S	L_{max}	#S	L_{max}	#S	L_{max}
A	shift	90	90	10	**100**	**0**	**100**	**0**	**100**	**0**	**100**	**0**	**100**	**0**
	day	90	90	193	90	135	90	115	90	77	**100**	**0**	98	**0**
	week	**80**	70	2433	68	1996	70	**1800**	**80**	1839	77	1840	**80**	1858
B	shift	60	60	467	80	420	**90**	372	82	353	81	**349**	87	439
	day	52	46	3218	50	2954	60	2522	60	2439	63	**2121**	**70**	2134
	week	35	10	26915	10	21070	10	20572	32	20541	31	**20355**	**36**	20635
C	shift	40	40	1941	**49**	**1676**	40	1901	45	1727	40	1824	40	2012
	day	10	10	9498	10	9503	10	8683	26	**8656**	**36**	8747	35	9022
	week	**10**	0	71104	0	**64442**	0	64480	9	64584	9	64474	**10**	64445
D	shift	19	16	2677	**40**	2154	30	2338	33	**2018**	30	2304	30	2621
	day	0	0	13994	0	13659	0	12657	0	12723	**13**	**12225**	11	12340
	week	0	0	107186	0	101474	0	**100533**	0	100760	0	100954	0	100840
Moyenne		40.5	36	19969	41	18290	42	17998	46	17976	48	**17933**	**50**	18029

On peut voir dans ce tableau, que dans sa version de base, le MCTS nous permet de résoudre plus d'instances pour les horizons courts mais dégrade les résultats pour les horizons plus longs. L'utilisation de l'accroissement de la borne inférieure (DSMI) permet d'améliorer légèrement au global le nombre d'instances résolues et la valeur moyenne de l'objectif. Par contre, moins d'instances sont résolues pour les horizons "shift" des classes d'instances C et D. L'ajout du DFS (DSMI+DFS) nous permet surtout de résoudre plus d'instances au global, particulièrement sur les horizons longs ("day" et week"), mais ne domine aucune des méthodes précédentes si l'on regarde ligne par ligne (excepté GRASP, déjà dominée par DSMI). Le mécanisme de décroissance du compromis exploitation/exploration couplé au DFS (DSMI+DFS+DC) permet encore d'améliorer les résultat globaux, et plus particulièrement sur un horizon d'une journée, sans dominer les méthodes précédentes encore une fois. Enfin, la version DSMI+SAT-DFS+DC permet de résoudre plus d'instances au global, contre une dégradation de la fonction objectif, ce qui est attendue, étant donné que le DFS ne cherche plus à améliorer la borne supérieure de l'objectif. Cette dernière version est la plus performante en matière de nombre d'instances résolues : 9.5% d'instances en plus par rapport à la méthode CP que nous avions avant.

6 Conclusion

Nous avons présenté des adaptations de la méthode MCTS pour la résolution d'un problème d'optimisation combinatoire issu de l'industrie automobile. Les évaluations expérimentales montrent un gain de performance par rapport à des méthodes classiques.

Les adaptations que nous proposons sont efficaces pour ce problème, d'autres combinaisons de ces adaptations sont en cours d'évaluation afin d'affiner l'apport de chacune d'entre elles. Les travaux futurs se focaliseront sur l'étude de la généricité de ces adaptations, notamment l'affectation du budget du DFS, en les appliquant à d'autres problèmes académiques et industriels.

Références

[1] Valentin Antuori, Emmanuel Hebrard, Marie-José Huguet, Siham Essodaigui, and Alain Nguyen. Leveraging Reinforcement Learning, Constraint Programming and Local Search : A Case Study in Car Manufacturing. In *Principles and Practice of Constraint Programming - 26th International Conference, CP 2020*, pages 657–672, 2020.

[2] Dimitris Bertsimas, J. Daniel Griffith, Vishal Gupta, Mykel J. Kochenderfer, and Velibor V. Misic. A comparison of Monte Carlo tree search and rolling horizon optimization for large-scale dynamic resource allocation problems. *European Journal Operational Research*, 263(2) :664–678, 2017.

[3] Cameron Browne, Edward Jack Powley, Daniel Whitehouse, Simon M. Lucas, Peter I. Cowling, Philipp Rohlfshagen, Stephen Tavener, Diego Perez Liebana, Spyridon Samothrakis, and Simon Colton. A Survey of Monte Carlo Tree Search Methods. *IEEE Transactions on Computational Intelligence and AI in Games*, 4(1) :1–43, 2012.

[4] Rémi Coulom. Efficient Selectivity and Backup Operators in Monte-Carlo Tree Search. In *Computers and Games, 5th International Conference, CG 2006*, pages 72–83, 2006.

[5] Ryota Furuoka and Shimpei Matsumoto. Worker's knowledge evaluation with single-player monte carlo tree search for a practical reentrant scheduling problem. *Artificial Life and Robotics*, 22(1) :130–138, 2017.

[6] Manuel Loth, Michèle Sebag, Youssef Hamadi, and Marc Schoenauer. Bandit-Based Search for Constraint Programming. In *Principles and Practice of Constraint Programming - 19th International Conference, CP 2013*, pages 464–480, 2013.

[7] Minh Anh Nguyen, Kazushi Sano, and Vu Tu Tran. A monte carlo tree search for traveling salesman problem with drone. *Asian Transport Studies*, 6 :100028, 2020.

[8] Alessandro Previti, Raghuram Ramanujan, Marco Schaerf, and Bart Selman. Monte-Carlo Style UCT Search for Boolean Satisfiability. In *AI*IA 2011 : Artificial Intelligence Around Man and Beyond - XIIth International Conference of the Italian Association for Artificial Intelligence*, pages 177–188, 2011.

[9] Thomas Philip Runarsson, Marc Schoenauer, and Michèle Sebag. Pilot, rollout and monte carlo tree search methods for job shop scheduling. In *Learning and Intelligent Optimization - 6th International Conference, LION 6*, pages 160–174, 2012.

[10] Ashish Sabharwal, Horst Samulowitz, and Chandra Reddy. Guiding combinatorial optimization with UCT. In *Integration of AI and OR Techniques in Contraint Programming for Combinatorial Optimzation Problems - 9th International Conference, CPAIOR 2012*, pages 356–361, 2012.

[11] David Silver, Aja Huang, Chris J. Maddison, Arthur Guez, Laurent Sifre, George van den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Vedavyas Panneershelvam, Marc Lanctot, Sander Dieleman, Dominik Grewe, John Nham, Nal Kalchbrenner, Ilya Sutskever, Timothy P. Lillicrap, Madeleine Leach, Koray Kavukcuoglu, Thore Graepel, and Demis Hassabis. Mastering the game of Go with deep neural networks and tree search. *Nature*, 529(7587) :484–489, 2016.