Exploitation de l'apprentissage par renforcement avec la Programmation par Contraintes ou la Recherche Locale - Cas d'application dans l'industrie automobile

V. Antuori[1,2] **E. Hébrard**[1,3] **M-J. 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

Le travail présenté porte sur un problème de transport de composants dans un atelier d'assemblage de l'industrie automobile. Deux premiers modèles de programmation par contraintes (PPC) ont été proposés et les évaluations expérimentales montrent qu'ils surpassent les performances d'une méthode de recherche locale actuellement utilisée par Renault sur un jeu de données industrielles. De plus, ces expérimentations mettent en évidence qu'une heuristique adéquate permet généralement de guider le solveur vers une solution avec un faible nombre de *backtracks*. Nous proposons alors d'*apprendre* une politique heuristique efficace via l'apprentissage par renforcement, et d'exploiter cette politique dans plusieurs méthodes : PPC avec redémarrage rapide, *recherche à divergences limitées* et recherche locale multi-départ. Ces méthodes sont plus efficaces que que les modèles initiaux de PPC sur les instances industrielles, mais également sur des instances générées aléatoirement.

1 Problème étudié

Le problème considéré dans cet article[1] provient d'un atelier d'assemblage du constructeur automobile Renault. Il consiste à déplacer des composants entre leurs points de production et leurs points de consommation. Ces composants sont transportés à l'aide de chariots dédiés, i.e. spécifiques à chaque composant. Il faut collecter ces chariots lorsqu'ils sont pleins pour les livrer à leurs lieux de consommation, mais également collecter les chariots lorsqu'ils sont vides pour les ramener à leur lieux de production. Les cadences de production et de consommation d'un même composant sont identiques. Ces cadences définissent alors des fenêtres de temps, il s'agit de faire ces requêtes (collecte et livraison du chariot plein, collecte et livraison du chariot vide), avant la fin du prochain cycle de production/consommation, ou ces requêtes devront être effectuées à nouveau, la fin d'une fenêtre de temps marquant le début de la fenêtre de temps des opérations du prochain cycle. De plus, chaque composant dispose de sa propre cadence de production/consommation. Les chariots peuvent s'assembler de manière à former un train de chariots, qui ne doit pas dépasser une certaine taille, chaque chariot ayant une taille différente. Le problème étudié se ramène à un problème de collectes et de livraisons à un véhicule, avec fenêtres de temps et contrainte de capacité, sans objectif à optimiser. La particularité du problème industriel est que les requêtes sont périodiques et que cette périodicité est propre à chaque composant. Le but est de planifier les différentes opérations, en respectant les contraintes de fenêtres de temps et de capacité, jusqu'à un horizon temporel donné (allant d'un *shift* d'une durée de 7h15 jusqu'à une semaine de 6 jours composés chacun de 3 *shifts*).

2 Modèles PPC

Dans l'article nous présentons deux modèles de programmation par contraintes (PPC). Le premier modélise un problème d'ordonnancement disjonctif, avec précédences, fenêtres de temps et contraintes de ré-

1. L'article original a été présenté à la conférence CP 2020[1]

servoir, et est basé sur des variables d'ordre entre les opérations. Le second correspond à un problème de type voyageur de commerce et s'appuie sur la contrainte Circuit. Nous comparons ces deux approches avec une méthode de recherche locale actuellement utilisée par Renault. Les résultats obtenus montrent qu'avec une bonne heuristique, il est possible d'obtenir des solutions quasiment sans *backtrack* pour la plupart des instances industrielles mais, il reste difficile d'obtenir des solutions pour des instances plus contraintes. Aussi, le modèle d'ordonnancement est le plus efficace.

3 Méthodes basées sur l'apprentissage d'une heuristique

Apprentissage par renforcement. Afin d'améliorer la résolution de ce problème, nous proposons d'*apprendre* une heuristique efficace via apprentissage par renforcement. Plus précisément, nous travaillons sur une relaxation du problème (minimisation du retard maximum), qui peut être décrite comme un processus de décision markovien (PDM). Dans ce PDM, un état est une séquence d'opérations (séquence vide à l'état initial), et l'ensemble des actions d'un état est l'ensemble des opérations qui peuvent étendre cette séquence sans violer les contraintes de précédence ni de capacité. On obtient une pénalité lorsqu'une action augmente le retard maximal, la pénalité étant l'accroissement marginal du retard maximum. Nous souhaitons alors apprendre une politique stochastique pour naviguer dans ce PDM, et qui minimise l'espérance de la somme des pénalités (et donc le retard maximum). L'espace des états étant exponentiellement large, nous avons besoin d'un descripteur d'une abstraction d'un état. Pour cela, nous définissons, à chaque état de notre PDM, une liste de 4 critères pour chacune des actions réalisables. Nous agrégeons ces critères par une combinaison linéaire, pour aboutir à un score de *fitness* pour chacune des actions. Ce vecteur de *fitness* est ensuite transformé en distribution de probabilité sur les actions par une fonction `softmax`. La politique stochastique consiste à choisir aléatoirement une action en suivant cette distribution. L'objectif de l'apprentissage par renforcement est alors de trouver les poids de cette combinaison linéaire qui vont minimiser l'espérance de la fonction objectif, à savoir le retard maximum, en suivant cette politique. Pour réaliser cet apprentissage, nous utilisons un algorithme de *policy gradient*, algorithme classique d'apprentissage par renforcement.

Exploitation de l'heuristique apprise. Nous proposons trois méthodes permettant de tirer profit de l'heuristique apprise lors de l'étape d'apprentissage par renforcement. La première méthode est basée sur une adaptation du modèle PPC d'ordonnancement dans lequel nous intégrons une stratégie de redémarrage très agressif, couplé avec l'heuristique stochastique, dans le but d'explorer très rapidement différentes parties de l'arbre. La seconde méthode est basée sur le même modèle PPC mais utilise une stratégie d'exploration de type *Limited Discreancy Search* (LDS) afin de dévier le moins possible de l'heuristique. Pour cela, nous utilisons une version déterministe de l'heuristique, qui consiste à choisir l'opération ayant le meilleur score de *fitness*. La troisième méthode est une recherche locale multi-départ consistant à répéter le processus glouton (basé sur l'heuristique stochastique) suivi d'une recherche locale. Nous avons proposé deux voisinages en $O(nm)$ où n est le nombre total d'opérations et m le nombre de composants différents. Le premier voisinage est un *swap* entre deux opérations de la séquence. Le second voisinage est un "double" *swap*, entre les deux opérations de collecte et de livraison d'un chariot plein, et celles d'un chariot vide.

Résultats expérimentaux. Pour compléter les instances industrielles, nous avons généré un ensemble d'instances aléatoires plus contraintes que les instances réelles. Les expérimentations montrent que les trois méthodes surpassent chacune les performances des premiers modèles de PPC, sur les instances industrielles mais également sur les instances générées aléatoirement. Malgré cela, de nombreuses instances restent non résolues.

À la suite de ces travaux, plusieurs perspectives se dessinent : d'abord nous proposons d'enrichir le modèle d'apprentissage, par l'ajout de critères, ou par le remplacement de la fonction linéaire par une fonction plus complexe comme par exemple un réseau de neurones. Ensuite nous visons à intégrer l'heuristique trouvée dans une approche de type recherche arborescente de Monte-Carlo, car cette approche peut s'appuyer fortement sur une heuristique gloutonne. Enfin le problème présenté est une partie d'un problème plus global, dans lequel on doit affecter les opérateurs aux composants, avant de planifier les routes, et nous projetons de nous attaquer à ce problème.

Références

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

Leveraging Reinforcement Learning, Constraint Programming and Local Search: A Case Study in Car Manufacturing

Valentin Antuori[1,2], Emmanuel Hebrard[1,3], Marie-José Huguet[1], Siham Essodaigui[2], and Alain Nguyen[2]

[1] LAAS-CNRS, Université de Toulouse, CNRS, INSA, Toulouse, France
{vantuori,hebrard,huguet}@laas.fr
[2] Renault, France
{valentin.antuori,siham.essodaigui,alain.nguyen}@renault.com
[3] ANITI, Université de Toulouse, France

Abstract. The problem of transporting vehicle components in a car manufacturer workshop can be seen as a large scale single vehicle pickup and delivery problem with periodic time windows. Our experimental evaluation indicates that a relatively simple constraint model shows some promise and in particular outperforms the local search method currently employed at Renault on industrial data over long time horizon. Interestingly, with an adequate heuristic, constraint propagation is often sufficient to guide the solver toward a solution in a few backtracks on these instances. We therefore propose to learn efficient heuristic policies via reinforcement learning and to leverage this technique in several approaches: rapid-restarts, limited discrepancy search and multi-start local search. Our methods outperform both the current local search approach and the classical CP models on industrial instances as well as on synthetic data.

Keywords: Constraint Programming · Reinforcement Learning · Local Search · Scheduling · Travelling Salesman Problem

1 Introduction

Improving the production line is a constant concern in the industry. The car manufacturer Renault has long been interested in models and techniques from Operations Research and Constraint Programming to tackle the various routing and scheduling problems arising from the production process.

Recent advances in Artificial Intelligence and in particular in Machine Learning (ML) open up many new perspectives for solving large scale combinatorial optimization problems with promising results popularized by the success of AlphaGo and AlphaZero [14, 15]. In particular, several approaches combining reinforcement learning and deep neural networks to guide the reward strategy have been proposed for solving the traveling salesman problem (TSP) [2, 4]. Moreover, the combination of ML and classical combinatorial techniques seems very promising. For instance, the integration of standard TSP heuristics with a neural network heuristic policy outperforms the pure ML approaches [5].

In Section 2 we introduce the problem of planning the flow of vehicle components across the assembly lines. More precisely, given fixed production cycles, logistics operators are in charge of collecting components from, and delivering them to, working stations in such a way that there is no break in the manufacturing process. This problem is in many ways similar to the *Torpedo Scheduling Problem* [7], from the 2016 ACP challenges [13]: pickup and delivery operations are to be scheduled, and these operations are repeated in time because the commodity is being produced constantly at a fixed rate. However, in our problem, components are carried over using trolleys that can be stacked into a “train” that should not exceed a given maximal length. It should be noted that in this paper we consider the problem associated to a single operator whose route is to be optimized. However, the more general problem for Renault is to assign components (or equivalently working stations) to operators as well as to plan the individual routes, and the longer term objective is to proactively design the layout of the assembly line to reduce the cost of logistics operations.

We first evaluate in Section 3 two basic constraint programming (CP) models in `Choco` [11] and compare them to the current method used by Renault: a local search (LS) method implemented in `LocalSolver`[4]. From this preliminary study, we observe that although the problem can be hard for both approaches, CP shows promising results compare to LS. Moreover, if solving the problem via backtracking search seems very unlikely given its size, and if stronger filtering techniques seem to be ineffective, greedy “dives” are often surprisingly successful.

We therefore propose in Section 4 to learn effective stochastic heuristic policies via reinforcement learning. Then, we show how these policies can be used within different tailored approaches: constraint programming with rapid restarts, limited discrepancy search, and multi-start local search approach.

Finally, since industrial benchmarks are easily solved by all new methods introduced in this paper, we generated a synthetic dataset designed to be more challenging. In Section 6 we report experiments on this dataset that further demonstrates the efficiency of the proposed methods.

2 Problem Definition

The Renault assembly line consists of a set of m components to be moved across a workshop, from the point where they are produced to where they are consumed. Each component is produced and consumed by two unique machines, and it is carried from one to the other using a specific trolley. When a trolley is filled at the production point for that component, an operator must bring it to its consumption point. Symmetrically, when a trolley is emptied it must be brought to the corresponding production point. A production cycle is the time c_i taken to produce (resp. consume) component i, that is, to fill (resp. empty) a trolley. The

[4] `https://www.localsolver.com/home.html`

end of a production cycle marks the start of the next, hence there are $n_i = \frac{H}{c_i}$ cycles over a time horizon H. There are two pickups and two deliveries at every cycle k of each component i: the pickup pe_i^k and delivery de_i^k of the empty trolley from consumption to production and the pickup pf_i^k and delivery df_i^k of the full trolley from production to consumption. The processing time of an operation a is denoted p_a and the travel time between operations a and b is denoted $D_{a,b}$.

Let P be the set of pickup operations and D the set of delivery operations: $P = \bigcup_{i=1}^{m} \left(\bigcup_{k=1}^{n_i} \{pe_i^k, pf_i^k\} \right)$, $D = \bigcup_{i=1}^{m} \left(\bigcup_{k=1}^{n_i} \{de_i^k, df_i^k\} \right)$. The problem is to compute the bijection σ (let $\rho = \sigma^{-1}$ be its inverse) between the $|A| = n$ first positive integers to the operations $A = P \cup D$ satisfying the following constraints.

Time windows. As production never stops, all four operations of the k-th cycle of component i must happen during the time window $[(k-1)c_i, kc_i]$. Let $r_{a_i^k}$ (resp. $d_{a_i^k}$) be the release date $(k-1)c_i$ (resp. due date kc_i) of operation a_i^k of the k-th cycle of component i. The start time of operation $\sigma(j)$ is:

$$s_{\sigma,j} = \begin{cases} r_{\sigma(j)} & \text{if } j = 1 \\ \max(r_{\sigma(j)}, s_{\sigma,j-1} + p_{\sigma(j-1)} + D_{\sigma(j-1),\sigma(j)}) & \text{otherwise} \end{cases}$$

Then, the completion time $e_{\sigma,j} = s_{\sigma,j} + p_{\sigma(j)}$ of operation $\sigma(j)$ must be lower than its due date:

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

Precedences. Pickups must precede deliveries.

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

Notice that there are only two possible orderings for the four operations of a production cycle. Indeed, since the first delivery (of either the full or the empty trolley) and the second pickup take place at the same location, doing the second pickup before the first delivery is dominated (w.r.t. the train length and the time windows). Hence Equation 3 is valid, though not necessary:

$$\rho(df_i^k) < \rho(pe_i^k) \vee \rho(de_i^k) < \rho(pf_i^k)\ \forall i \in [1, m]\ \forall k \in [1, n_i] \tag{3}$$

Train length. The operator may assemble trolleys into a train[5], so a pickup need not be directly followed by its delivery. However, the total length of the train of trolleys must not exceed a length $T_{\max}$. Let t_i be the length of the trolley for component i, and let $t_{a_i^k} = t_i$ if $a_i^k \in P$ and $t_{a_i^k} = -t_i$ otherwise, then:

$$\forall j \in [1, n],\ \sum_{l=1}^{j} t_{\sigma(l)} \leq T_{\max} \tag{4}$$

[5] Trolleys are designed so that they can be extracted out of the train in any order

This is a particular case of the *single vehicle pickup and delivery problem with capacity and time windows constraints*. However, there is no objective function, and instead, feasibility is hard. Moreover, the production-consumption cycles entail a very particular structure: the four operations of each component must take place in the same time windows and this is repeated for every cycle. As a result, one of the best method, Large Neighborhood Search [12], is severely hampered since it relies on the objective to evaluate the moves and the insertion of relaxed requests is often very constrained by the specific precedence structure.

3 Baseline Models

We designed two CP models: a variant of a single resource scheduling problem and a variant of a TSP. Then, we describe the current `LocalSolver` model.

Scheduling-based model. The importance of time constraints in the problem studied, suggests that a CP model based on scheduling would be relevant [1]. The problem is a single resource (the operator) scheduling problem with four types of non overlapping operations (pickup and delivery of empty and full trolleys). For each operation $a \in A$, we define the variable $s_a \in [r_a, d_a]$ as the starting date of operation a. Moreover, for each pair of operations a, $b \in A$, we introduce a Boolean variable x_{ab} standing for their relative ordering. In practice, we need much fewer than n^2 Boolean variables as the time windows and Constraint (2) entails many precedences which can be either ignored or directly posted.

$$x_{ab} = \begin{cases} 1 \Leftrightarrow s_b \geq s_a + p_a + D_{a,b} \\ 0 \Leftrightarrow s_a \geq s_b + p_b + D_{b,a} \end{cases} \qquad \forall (a,b) \in A \tag{5}$$

$$x_{df_i^k pe_i^k} \vee x_{de_i^k pf_i^k} \qquad \forall i \in [1,m]\ \forall k \in [1,n_i] \tag{6}$$

Constraint (5) chanel the two sets of variables, and constraint (6) encodes Equation (3). Finally, Constraint (4) can be seen as a reservoir resource with limited capacity, which is filled by pickups, and emptied by deliveries. We use the algorithm from [9] to propagate it on starting date variables using the precedence graph implied by Boolean variables and precedences from Constraint (2).

TSP-based model. The second model is an extension of the first one, to wich we add extra variables and constraints from the model for TSP with time windows proposed in [6]. We need two fake operations, 0 and $n+1$, for the start and the end of the route. For each operation $a \in A \cup \{0, n+1\}$, there is a variable $next_a$ that indicates which operation directly follows a. Also, a variable pos_a indicates the position of the operation a in the sequence of operations. We need another variable $train_a \in [0, T_{\max}]$ which represents the length of the train before the operation a. The following equations express the constraints of the problem:

$$train_0 = 0 \wedge train_{next_a} = train_a + t_a \qquad \forall a \in A \cup \{0\} \tag{7}$$

$$pos_{pf_i^k} < pos_{df_i^k} \wedge pos_{pe_i^k} < pos_{de_i^k} \qquad \forall i \in [1,m]\ \forall k \in [1,n_i] \tag{8}$$

$$s_0 = 0 \wedge s_{next_a} \geq s_a + p_a + D_{a,next_a} \qquad \forall a \in A \cup \{0\} \tag{9}$$

Constraint (7) encodes the train length at every point of the sequence using the ELEMENT constraint and constraint (8) ensures that pickups precede their deliveries. Constraint (9) ensure the accumulation of the time along the tour. Moreover, we use the CIRCUIT constraint to enforce the variables *next* to form an Hamiltonian circuit. Additional redundant constraints are used to make the channeling between variables and improve filtering in addition to the constraint from the first model: $x_{ab} \implies next_b \neq a$, $pos_b > pos_a + 1 \implies next_a \neq b$, $pos_b > pos_a \Leftrightarrow x_{ab}$, $pos_a = \sum_{b \in A \cup \{0,n+1\}} x_{ab}$ and ALLDIFFERENT(pos).

Search strategy. Preliminary experiments revealed that branching on variables in an ordering "consistent" with the sequence of operations was key to solving this problem via CP. In the TSP model, we simply branch on the variables *next* in ascending order, and choose first the operation with least earliest start time. In the scheduling model, we compute the set of pairs of operations a, b such that $\{s_a, s_b\}$ is Pareto-minimal, draw one pair uniformly at random within this set and assign x_{ab} so that the operation with least release date comes first. In conjunction with constraint propagation, this strategy acts as the "nearest city" heuristic in TSP. Indeed, since the sequence of past operations is known, the earliest start time of an operation depends primarily on the distance from the latest operation in the partial sequence (it also depends on the release date).

LocalSolver. The `LocalSolver` (LS) model is similar to the TSP model. It is based on a variable of type list $\boldsymbol{seq}$, a special decision variable type that represent the complete tour: $seq_j = a$ means operation a is performed at position j. This variable is channeled with another list variable $\boldsymbol{pos}$ with $pos_a = j \Leftrightarrow seq_j = a$. We need two other list variables: *train* and s to represent respectively the length of the train and the start time of the operation at a given position.

$$
\begin{aligned}
& s_1 = 0 \wedge \\
& s_j = \max(r_{seq_j}, s_{j-1} + p_{seq_{j-1}} + D_{seq_{j-1},seq_j}) && \forall j \in [2,n] && (10) \\
& train_1 = 0 \ \wedge train_j = train_{j-1} + t_{seq_j} && \forall j \in [2,n] && (11) \\
& s_j + p_{seq_j} \leq d_{seq_j} && \forall j \in [1,n] && (12) \\
& pos_{pf_i^k} < pos_{df_i^k} \wedge pos_{pe_i^k} < pos_{de_i^k} && \forall i \in [1,m]\ \forall k \in [1,n_i] && (13) \\
& pos_{df_i^k} < pos_{pe_i^k} \vee pos_{de_i^k} < pos_{pf_i^k} && \forall i \in [1,m]\ \forall k \in [1,n_i] && (14) \\
& \textsc{Count}(\boldsymbol{seq}) = 0 && && (15)
\end{aligned}
$$

The last constraint (15) acts like the global constraint ALL DIFFERENT and therefore ensures that $\boldsymbol{seq}$ is a permutation. Surprisingly, relaxing the due dates and using the maximal tardiness as objective tends to degrade the performance of `LocalSolver`, hence we kept the satisfaction version.

3.1 Preliminary Experiments

The industrial data we have collected fall into three categories. In the industrial assembly line, each category is associated to one logistic operator who has the charge of a given set of components. In practice, these datasets are relatively underconstrained, with potentially quite large production cycles (time windows) for each component. For each category, denoted by S, L and R, we consider three time horizons: 43 500, 130 500 and 783 000 *hundredths of a minutes* which corresponds to a shift of an operator (7 hours and 15 minutes), a day of work (made up of three shifts) and a week (6 days) respectively.

We then have 9 industrial instances from 400 to more than 10 000 operations. The main differences between those three categories are the number of components, and the synchronicity of the different production cycles. For S instances, there are only 5 components, and their production cycles are almost the same and very short. The other two instances have more than 30 components, with various production cycles (some cycles are short and others are very long).

The CP models were implemented using the constraint solver `Choco` 4.10 [11], and the `LocalSolver` version was 9.0. All experiments of this section were run on Intel(R) Core(TM) i7-8700 CPU @ 3.20GHz with a timeout of 1 hour.

The results are given in Table 1. The first column (Cl) denotes the category of the instance and second column (H) denotes the temporal horizon. For the two CP models (Scheduling and TSP), two indicators are given, the CPU time (in seconds), and the numbers of fails for solved instances. For `LocalSolver`, we give the CPU time when the instances are solved before the time limit.

The CP scheduling-based model solves all of the industrial instances without any fail except for instances R and L on the whole week. Notice, however, that when using generic heuristics such as *Domain over Weighted Degree* [3], only 3 instances could be solved by the same model in less than an hour. Moreover, `LocalSolver` cannot solve the largest instances, and requires much more CPU time in general. Although industrial instances are clearly very underconstrained, they are not trivial. Moreover, even on underconstrained instances, wrong early choices may lead to infeasible subtrees from which we cannot easily escape. In particular, the number of fails for the largest instance of category R shows that the very deep end of the search tree is likely explored in a brute-force manner.

One key factor in solving these instances is for the variable ordering to follow the ordering of the sequence of operations being built. Indeed, the propagation is much more efficient in this way, and in particular, the earliest start times of future operations can be easily computed and, as mentioned in the description of the heuristic, it reflects the distance from the last operation in the route.

We observe that the scheduling-based model is the fastest. Moreover, the TSP-based model contains too many variables and constraints, and run out of memory for the bigger horizon. Instances without any fails show that the first branch can be very slow to explore as propagation in nodes close to the root can be very costly, in particular with the TSP-based model.

We draw two conclusions from these preliminary experiments: First, the basic CP (or LS) models cannot reliably solve hard instances[6]. It is unlikely that stronger propagation would be the answer, as the TSP model (with stronger

[6] There might be too few data points to make that claim on industrial instances, but it is clearly confirmed on the synthetic benchmark (see Table 2).

Table 1: Comparison on the industrial instances

Cl	H	Scheduling		TSP		`LocalSolver`
		CPU	#fail	CPU	#fail	CPU
S	Shift	0.275	0	3.999	0	26
S	Day	0.840	0	48.855	0	566
S	Week	17.747	0	memory out		timeout
L	Shift	7.129	0	36.033	8	121
L	Day	23.468	0	344.338	8	3539
L	Week	318.008	6	memory out		timeout
R	Shift	8.397	0	41.615	14	1065
R	Day	44.215	0	417.116	15	timeout
R	Week	timeout	773738	memory out		timeout

filtering) tends to be less effective, and does not scale very well. Second, building the sequence "chronologically" helps significantly, which explains why CP models outperform local search. As a consequence, greedy runs of the CP solver are surprisingly successful. Therefore, we propose to *learn* efficient heuristic policies and explore methods that can take further advantage of these heuristics.

4 Reinforcement Learning

The search for a feasible sequence can be seen as a Markov Decision Process (MPD) whose *states* are partial sequences σ, and *actions* $\mathcal{A}(\sigma)$ are the operations that can extend σ without breaking precedence (2) nor capacity constraints (4).

In order to apply Reinforcement Learning (RL) to this MDP, it is convenient to relax the due date constraints and replace them by the minimization of the maximum tardiness: $\max\{L(\sigma,j) \mid 1 \leq j \leq |\sigma|\}$ where $L(\sigma,j) = e_{\sigma,j} - d_{\sigma(j)}$ is the tardiness of operation $\sigma(j)$. We also define the maximum tardiness on intervals: $L(\sigma,j,l) = \max\{L(\sigma,q) \mid j \leq q \leq l\}$, and we write $L(\sigma)$ for $L(\sigma,1,|\sigma|)$. Since in this case operations can finish later than their due dates, it is necessary to make explicit the precedence constraints due to production cycles:

$$\max(\rho(df_i^{k-1}), \rho(de_i^{k-1})) < \min(\rho(pe_i^k), \rho(pf_i^k)) \ \forall i \in [1,m] \ \forall k \in [2,n_i] \quad (16)$$

Then we can further restrict the set $\mathcal{A}(\sigma)$ to operations that would not violate Constraints (16), nor (3). As a result, any state of the MDP reachable from the empty state is feasible for the relaxation. Finally, we can define the *penalty* $R(\sigma,j)$ for the j-th decision as the marginal increase of the objective function when the j-th operation is added to σ: $R(\sigma,j) = L(\sigma,1,j) - L(\sigma,1,j-1))$.

Now we can apply standard RL to seek for an effective stochastic heuristic policy for selecting the next operation. Moreover, as the MPD defined above is exponentially large, it is common to abstract a state σ with a small descriptor, namely, $\boldsymbol{\lambda}(\sigma, a)$ a vector of four criteria for operation $a \in A$ in state σ. Let $lst(a, \sigma)$ be the latest starting time of the task a in order to satisfy constraint (1) with respect to the tasks in σ and constraints (2) and (3). For each operation $a \in \mathcal{A}(\sigma)$, we compute $\boldsymbol{\lambda}(\sigma, a)$ as follows:

$$\boldsymbol{\lambda}_1(\sigma, a) = (lst(a, \sigma) - \max(r_a, e_{\sigma,|\sigma|} - L(\sigma) + D_{\sigma(|\sigma|),a}))/\max_{b \in A}\{d_b - r_b\} \tag{17}$$

$$\boldsymbol{\lambda}_2(\sigma, a) = \max(r_a - (e_{\sigma,|\sigma|} - L(\sigma)), D_{\sigma(|\sigma|),a})/\max_{b,c \in A^2}\{D_{b,c}\} \tag{18}$$

$$\boldsymbol{\lambda}_3(\sigma, a) = 1 - |t_a|/T_{\max} \tag{19}$$

$$\boldsymbol{\lambda}_4(\sigma, a) = \begin{cases} 1 & \text{if } a \in P \\ 0 & \text{otherwise} \end{cases} \tag{20}$$

Criterion (17) can be seen as the operation's *emergency*: the distance to the due date of the task. Criterion (18) is the *travel time* from the last task in the sequence. For both of these criterion, we use $e_{\sigma,|\sigma|} - L(\sigma)$ instead of $e_{\sigma,|\sigma|}$ as the end time of the partial sequence σ to offset the impact of previous choices. Criterion (19) is the *length* of the trolley. Indeed, since all operations must eventually be done, doing the operations which have the highest consumption of the "train" resource earlier leaves more freedom for later. Finally, criterion (20) penalizes pickups as leaving a trolley in the train for too long is wasteful.

We want to learn a stochastic policy $\pi_{\boldsymbol{\theta}}$, governed by a set of parameters $\boldsymbol{\theta}$ which gives the probability distribution over the set of actions $\mathcal{A}(\sigma)$ available at state σ. We first define a fitness function $f(\sigma, a)$ as a simple linear combination of the criteria: $f(\sigma, a) = \boldsymbol{\theta}^{\intercal}\boldsymbol{\lambda}(\sigma, a)$. Then, we use the `softmax` function to turn the fitness function into a probability distribution (ignore parameter β for now).

$$\forall a \in \mathcal{A}(\sigma)\ \pi_{\boldsymbol{\theta}}(a \mid \sigma) = \frac{e^{(1-f(\sigma,a))/\beta}}{\sum_{b \in \mathcal{A}(\sigma)} e^{(1-f(\sigma,b))/\beta}} \tag{21}$$

4.1 Policy Gradient

As we look for a stochastic policy, the goal is to find the value of $\boldsymbol{\theta}$ minimizing the expected maximum value (i.e., maximum tardiness) $J(\boldsymbol{\theta})$ of solutions σ produced by the policy $\pi_{\boldsymbol{\theta}}$. The basic idea of policy gradient methods is to minimize $J(\boldsymbol{\theta})$ by gradient descent, that is: iteratively update $\boldsymbol{\theta}$ by subtracting the gradient. We resort to the REINFORCE learning rule [16] to get the gradient of $J(\boldsymbol{\theta})$:

$$\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) = \mathbb{E}_{\sigma \sim \pi_{\boldsymbol{\theta}}(\sigma)}[L(\sigma)\nabla_{\boldsymbol{\theta}} \log \pi_{\boldsymbol{\theta}}(\sigma)] \tag{22}$$

Which can then be approximated via Monte-Carlo sampling over S samples:

$$\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) \approx \frac{1}{S}\sum_{k=1}^{S} L(\sigma_k)\nabla_{\boldsymbol{\theta}} \log \pi_{\boldsymbol{\theta}}(\sigma_k) \tag{23}$$

We can decompose Equation (23) in order to give a specific penalty to each decision (for every position j) of the generated solutions:

$$\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) \approx \frac{1}{S} \sum_{k=1}^{S} \sum_{j=1}^{n} G(\sigma_k, j) \nabla_{\boldsymbol{\theta}} \log \pi_{\boldsymbol{\theta}}(\sigma_k(j) \mid \sigma_k(1, j-1)) \tag{24}$$

The penalty function takes into account that a decision can only affect the future marginal increase of tardiness: we replace the overall tardiness $L(\sigma)$ by:

$$G(\sigma, j) = \begin{cases} R(\sigma, j) & \text{if } j = n \\ R(\sigma, j) + \gamma * G(\sigma, j+1) & \text{otherwise} \end{cases}$$

The value of γ controls how much decisions impact future tardiness. For $\gamma = 0$, we only take into account the immediate penalty. Conversely $\gamma = 1$ means we consider the sum of the penalties from position j.

4.2 Learning algorithm

We learn a value of $\boldsymbol{\theta}$ for the synthetic dataset (Section 6) with the REINFORCE rule. Given a set of instances I, we learn by batch of size $S = q|I|$, in other words, we generate q solutions for each instance. The value of $\boldsymbol{\theta}$ is initialized at random, then we apply the following three steps until convergence or timeout:

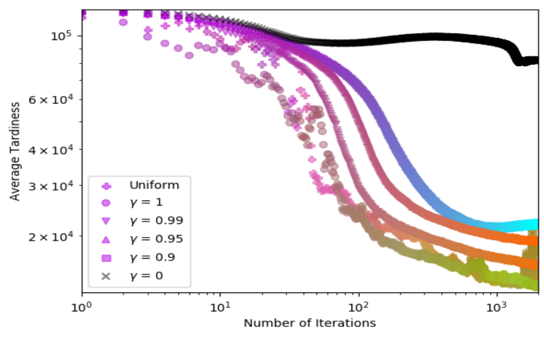
1. Generate S solutions following $\pi_{\boldsymbol{\theta}}$.
2. Compute $\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta})$ according to Equation (24)
3. Update the value of θ as follows: $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \alpha \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta})$

We found out that using a classic `softmax` function did not discriminate enough, and hence acts as random policy. To circumvent this, we use the parameter β in Equation (21) to control the trade off between the quality and the diversity of generated solutions. For a low value of β the policy always chooses the best candidate, whereas a large value yields a more "balanced" policy. It turns out that $\beta = 0.1$ was a good value for learning in our case.

Then, we have evaluated the impact of γ to compute the gradient in Equation (24). Recall that γ controls how much importance we give to the j-th decision in the total penalty: the overall increase of the tardiness from j to n for $\gamma = 1$ or only the instant increase for $\gamma = 0$. Moreover, we also tried to give the penalty $L(\sigma)$ uniformly for every decision j of the policy, instead of giving individual penalties $G(\sigma, j)$. We denote this penalty strategy "Uniform".

For each variant, we plot in Figure (1a) the average performance $L(\sigma)$ of the policy after each iteration (notice the log-scale both for X and Y). Here we learn on all generated instances of a day horizon (40 instances), for 2000 iterations.

High values for γ are always better. For low value of γ, the (local) optimum is higher, and in some cases the method may not even converge, e.g. for $\gamma = 0$. Using uniformly the overall tardiness gives similar results to $\gamma = 1$, however it is less stable, and in Fig. (1a) we observe that it diverges after 1000 iterations.



(a) Convergence for different values of γ

Cl	By Inst.	By Cat.	All
A	1011	1130	1128
B	6417	6766	6797
C	14622	15184	15069
D	20285	20578	20640

(b) Different training sets

Fig. 1: Behavior of reinforcement learning

Datapoints are colored with the vector $\boldsymbol{\theta}$ interpreted as a RGB value ($|\boldsymbol{\theta}| = 4$ but it can be characterized by three values after we normalize so that $\sum_{i=1}^{4} \boldsymbol{\theta}_i = 1$)[7]. We can see for instance that $\gamma = 0.9$ finds a value of $\boldsymbol{\theta}$ that is significantly different from all other methods ($\langle 0, 0.63, 0.29, 0.07 \rangle$). Interestingly, "Uniform" and $\gamma = 1$ not only converge to the same average tardiness, but to similar $\boldsymbol{\theta}$'s (respectively $\langle 0.30, 0.49, 0.16, 0.04 \rangle$ and $\langle 0.25, 0.56, 0.15, 0.04 \rangle$). However, $\gamma = 1$ finds values of $\boldsymbol{\theta}$ that seem closer to the target ("greener") earlier, although it is not really apparent from the value of $L(\sigma)$. These values of θ indicate that a good heuristic is mainly a compromise between the *emergency* and the *travel time*[8]. Moreover, the travel time tends to be more important on larger horizon, because it a longer term impact: all subsequent operations are affected.

Finally, we report in Figure (1b) the average tardiness for each instances following $\pi_{\boldsymbol{\theta}}$. The gain of learning specifically for a given (class of) instance(s) is at best marginal. This is not so surprising as we abstract states with a very simple model using a few criteria. However, it means that the value of $\boldsymbol{\theta}$ learnt on the full dataset is relevant to most instances.

5 Using the Heuristic Policy

We have implemented two types of approaches taking advantage of the stochastic policy learnt via RL: integrating it within CP as a randomized branching heuristic, and using it to generate sequences to be locally optimized via steepest descent in a multi-start local search. Here we describe the necessary modifications of the CP model, and we propose an efficient local search neighborhood.

[7] To highlight the differences we also normalize the RGB values and omit $\gamma = 0$

[8] Although the importance of a criterion also depends on the distribution of the values of λ after normalization, we are confident that the first two criteria are more important than the other two.

5.1 Constraint Programming

In order to use the stochastic policy described in Section 4 in a CP solver, we need to slightly change the scheduling model. We introduce a new set of variables seq_j, one for each position j, standing for the operation at that position, with the following channeling constraint: $x_{seq_j seq_l} = 1 \; \forall j < l \in [1, n]$

We branch only on the variables seq in lexicographic order. Therefore, the propagation for this constraint is straightforward: when branching on $seq_j = a$, as all variables $seq_l \; \forall l < j$ are already instantiated, we set $x_{ab} = 1$, $\forall b \in A \setminus \{seq_l \mid l < j\}$. Conversely, after assigning seq_j we can remove a from the domain of seq_{j+1} if there exists $b \in A \setminus \{seq_l \mid l \leq j\}$ such that the domain of the variable x_{ab} is reduced to $\{0\}$. Moreover, we can easily enforce *Forward Consistency* on $\{seq_1, \ldots, seq_j\}$ with respect to precedence and train size constraints (2) and (4) as well as Constraint (3), i.e., when $\{seq_1, \ldots, seq_{j-1}\}$ are all instantiated, we can remove all values of seq_j that cannot extend the current subsequence. Therefore, we do not need the *Reservoir resources* propagator anymore.

We propose two strategies based on this CP model using the learned policy.

1. *Softmax policy and rapid restart.* In this method we choose randomly the next operation according to the softmax policy (Eq. 21). In order to explore quickly different part of the search tree, we rely on a rapid restart strategy, following a Luby [10] sequence with a factor 15.
2. *Limited Discrepancy Search.* As the key to solve those instances is to follow good heuristics, and to deviate as little as possible from them, *limited discrepancy search* (LDS) [8] fits well with this approach. We run the LDS implementation of `Choco`, which is an iterative version: the discrepancy starts from 0, to a maximum discrepancy parameter incrementally. For this approach we use the deterministic version of the policy $\pi(\sigma) = \arg\min_a f(\sigma, a)$.

5.2 Local Search

The solutions found by the heuristic policy can often be improved by local search moves. Therefore, we also tried a multi-start local search whereby we generate sequences with the heuristic policy, and then improve them via steepest descent. Sequences are generated using the same model used for RL (i.e., with relaxed due dates). Therefore, generated sequences respect all constraints, except (1) and we consider a neighborhood that preserves all other constraints as well. Then we apply the local move that decrease the most the maximum tardiness $L(\sigma)$ until no such move can be found. We use two types of moves and the time complexity of an iteration (i.e., computing, and commiting to, the best move) is in $O(nm)$.

We recall that $L(\sigma, j, l) = \max\{L(\sigma, q) \mid j \leq q \leq l\}$ is the maximum tardiness among all operations between positions j and l in σ.

Swap moves. The first type of moves consists in swapping the values of $\sigma(j)$ and $\sigma(l)$. First, we need to make sure that the ordering of the operations within a given component remains valid, i.e., satifies constraints (2) and (16): a pickup (resp. delivery) operation must stay between its preceding and following deliveries (resp. pickups) for the same component. For every operation a, a valid range between the position of its predecessor $pr(a)$ and of its successor $su(a)$ can be computed in constant time. Then, for all $j \in [1, n]$ we shall consider only the swaps between j and l for $l \in [j+1, su(\sigma(j))]$ and such that $pr(\sigma(l)) \leq j$.

The second condition for the move to be valid is that the swap does not violate constraint (4), i.e., the maximum length of the train. Let $\tau_j = \sum_{l=1}^{j-1} t_{\sigma(l)}$ be the length of the train before the j-th operation. After the swap we have $\tau_{j+1} = \tau_j + t_{\sigma(l)}$ which must be less than $T_{\max}$. At all other ranks until l, the difference will be $t_{\sigma(l)} - t_{\sigma(j)}$, we only need to check the constraint for the maximum train length, that is: $\max\{\tau_q \mid j \leq q \leq l\} + t_{\sigma(l)} - t_{\sigma(j)} \leq T_{\max}$. This can be done in constant (amortized) time for all the swaps of operations a by computing the maximum train length incrementally for each $l \in [j+1, su(\sigma(j))]$.

Then, we need to forecast the maximum tardiness of the sequence σ' where the operations at positions j and l are swapped, i.e., compute the marginal cost of the swap. The tardiness of operations before position j do not change. However, we need to compute the new tardiness $L(\sigma', j)$ and $L(\sigma', l)$ at positions j and l, respectively. Moreover, we need to compute $L(\sigma', j+1, l-1)$ the new maximum tardiness for operations strictly between j and l and $L(\sigma', l+1, n)$ the new maximum tardiness for operations strictly after l.

The new end time $e_{\sigma',j}$ of operation $\sigma'(j) = \sigma(l)$ and hence the tardiness at position j is $L(\sigma', j) = e_{\sigma',j} - d_{\sigma'(j)}$ can be computed in $O(1)$ as follows:

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

Next, operations $\sigma(j+1), \ldots, \sigma(l-1)$ remain in the same order and $\sigma(j+1)$ is shifted by a value $\Delta = e_{\sigma',j} + D_{\sigma'(j),\sigma'(j+1)} - e_{\sigma,j} - D_{\sigma(j),\sigma,j+1}$. However, subsequent operations may not all be equally time-shifted. Indeed, when $\Delta < 0$ there may exist an operation whose release date prevents a shift of Δ.

Let $g_j = r_{\sigma(j)} - s_{\sigma(j)}$ be the *maximum left shift* (negative shift of highest absolute value) for the j-th operation, and let $g_{j,l} = \max\{g_q \mid j \leq q \leq l\}$.

Proposition 1. *If the sequence does not change between positions j and l, a time-shift $\Delta < 0$ at position j yields a time-shift* $\max(\Delta, g_{j,l})$ *at position l.*

Let $L_\Delta(\sigma, j, l)$ be the maximum tardiness on the interval $[j, l]$ of sequence σ time-shifted by Δ from position j. We can define $L_{-\infty}(\sigma, j, l)$ the maximum tardiness on the interval $[j, l]$ for an infinite negative time-shift:

$$L_{-\infty}(\sigma, j, l) = \max\{L(\sigma, q, l) + g_{j,q} \mid j \leq q \leq l\} \tag{25}$$

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

Conversely, when $\Delta > 0$ some of the time-shift may be "absorbed" by the waiting time before an operation. However, there is little point in moving operations coming before a position j with a non-negative waiting time (i.e., where

$s_{\sigma(j)} = r_{\sigma(j)}$), as this operation and all subsequent operations would not profit from the reduction in travel time. Therefore we consider only swaps whose earliest position j is such that $\forall q > j,\ g_q < 0$. As a results, if there is a positive time-shift Δ at $q > j$, we know that $L_\Delta(\sigma, q, l) = \Delta + L(\sigma, q, l)$. Moreover, the values of $L(\sigma, j, l)$, $g_{j,l}$ and $L_{-\infty}(\sigma, j, l)$ can be computed incrementally as:

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

Therefore, when $j < l-1$, we can compute the new tardiness $L(\sigma', j+1, l-1) = L_\Delta(\sigma, j+1, l-1)$ of the operations in the interval $[j+1, l-1]$ in constant (amortized) time since the query of Proposition 2 can be checked in $O(1)$.

The new tardiness $L(\sigma', l)$ at position l is computed in a similar way as for $L(\sigma', j)$ since we know the new start time of $\sigma'(l-1)$ from previous steps.

Finally, in order to compute the new maximum tardiness $L(\sigma', l+1, n)$ over subsequent operations, we precompute $L(\sigma, j, n)$, $g_{j,n}$ and $L_{-\infty}(\sigma, j, n)$ for every position $1 \leq j \leq n$ once after each move in $O(n)$. Then $L(\sigma', l+1, n)$ can be obtained in $O(1)$ for every potential move from Proposition 2.

Therefore, we can check the validity and forecast the marginal cost of a swap in constant amortized time and perform the swap in linear time. The time complexity for an iteration is thus in $O(nm)$ since, for a given component i, the sum of the sizes of the valid ranges for all pickups and deliveries of this component is in $O(n)$. Indeed, let $a_i^1, \dots, a_i^{4n_i}$ be the operations component i ordered as in σ. Then $su(a_i^k) = \rho(a_i^{k+1})$ and $\sum_{k=1}^{4n_i} su(a_i^k) - \rho(a_i^k) = \rho(a_i^{4n_i}) - \rho(a_i^1) \in \Theta(n)$).

Toggle moves. As observed in Section 3, there are only two dominant orderings for the four operations of the k-th production cycle of component i. The second type of moves consists in changing from one to the other of these two orderings, by swapping the values of $\sigma(pf_i^k)$ and $\sigma(pe_i^k)$ and the values of $\sigma(df_i^k)$ and $\sigma(de_i^k)$. This change leaves the size of the train constant, hence all these moves are valid.

Let $j_1 = pf_i^k, j_2 = df_i^k, j_3 = pe_i^k, j_4 = de_i^k$ be the positions of the four operations of component i and cycle k in the current solution, and suppose, wlog, that $j_1 < j_2 < j_3 < j_4$. Let σ' denote the sequence obtained by applying a toggle move on component i and cycle k in σ. In order to forecast the marginal cost of the move, we need to compute the new tardiness at the positions of the four operations involved $L(\sigma', j_1)$, $L(\sigma', j_2)$, $L(\sigma', j_3)$ and $L(\sigma', j_4)$. Moreover, we need the new maximum tardiness on four time-shifted intervals:
$L(\sigma', j_1+1, j_2-1) = L_{\Delta_1}(\sigma, j_1+1, j_2-1)$, $L(\sigma', j_2+1, j_3-1) = L_{\Delta_2}(\sigma, j_2+1, j_3-1)$, $L(\sigma', j_3+1, j_4-1) = L_{\Delta_3}(\sigma, j_3+1, j_4-1)$ and $L(\sigma', j_4+1, n) = L_{\Delta_4}(\sigma, j_4+1, n)$.

Computing the marginal costs can be done via the same formulas as for swaps: we can first compute the new end time for the operation at position j_1, then from it compute the value of Δ_1 that we can use to compute $L_{\Delta_1}(\sigma, j_1+1, j_2-1)$ in $O(j_2-j_1-1)$ time, and so forth. The difference, however, is that there is fewer possible moves ($n/4$), although the computation of the marginal cost cannot be amortized. The resulting time complexity is the same: $O(nm)$.

6 Experimental Results

We generated synthetic instances[9] in order to better assess the approaches. Due to the time windows constraints, it is difficult to generate certifiably feasible instances. Their feasibility has been checked on the shortest possible horizon, i.e., the duration of the longest production cycle of any component, which is about 10 000 time units depending on the instances. There are four categories of instances parameterized by the number of components (15 in category A, 20 in B, 25 in C and 30 in D). In the real dataset, several components have similar production cycles. We replicates this feature: synthetic instances have from 2 or 3 distinct production cycles in category A, to up to 7 in category D. The latter are therefore harder because there are more asynchronous productions cycles. We generated 10 random instances for each category and consider the same three horizons (shift, day and week) for each, as industrial instances.

Experiments for this section were run on a cluster made up of Xeon E5-2695 v3 @ 2.30GHz and Xeon E5-2695 v4 @ 2.10GHz. For the basic CP models, we add randomization and a restart strategy following a Luby sequence, and we ran each of the 120 instances 10 times. We could not carry on experiments on synthetic data with `LocalSolver` because we were not granted a license. However, from the few tests we could do, we expect `LocalSolver` to behave similarly as on industrial benchmarks.

Table 2: Comparison of the methods on generated instances

Cl	H	Scheduling			TSP			CP-softmax			LDS			Multi-start LS		
		#S	CPU	#fail	#S	CPU	#fail	#S	CPU	#fail	#S	CPU	#fail	#S	CPU	L_{max}
A	shift	7.1	418	300K	4.0	56	366	**9.0**	2	15	**9.0**	2	9	**9.0**	0	1m
	day	4.0	29	213	3.6	802	1267	**9.0**	15	815	8.0	21	71	**9.0**	176	19m
	week	3.1	866	1976	0.0	mem. out		**8.0**	118	27	5.0	68	0.0	7.0	155	1h21
B	shift	2.1	389	150K	0.9	844	11K	**6.0**	4	77	**6.0**	15	85	**6.0**	2	11m
	day	1.0	201	15K	0.0	–		**5.2**	341	20K	4.0	12	19	4.6	346	1h
	week	0.0	–		0.0	mem. out		**3.5**	423	715	1.0	99	0.0	1.0	0	4h59
C	shift	0.0	–		0.0	–		**4.0**	103	5366	**4.0**	715	4090	**4.0**	255	32m
	day	0.0	–		0.0	–		**1.0**	12	7	**1.0**	18	27	**1.0**	1	1h45
	week	0.0	–		0.0	mem. out		**1.0**	807	366	0.0	–		0.0	–	11h51
D	shift	0.0	–		0.0	–		**1.9**	697	24K	1.0	442	1058	1.6	1165	31m
	day	0.0	–		0.0	–		0.0	–		0.0	–		0.0	–	2h19
	week	0.0	–		0.0	mem. out		0.0	–		0.0	–		0.0	–	17h52

For each of the methods using the learnt heuristics, we normalize $\boldsymbol{\theta}$ so that $\sum_{i=1}^{4} \boldsymbol{\theta}_i = 1$ and set β to 1/150. We learn the policy by batches of size 240 formed by 6 runs on each of the 40 day-long instances, during 2000 iterations.

[9] Avalaible at `https://gitlab.laas.fr/vantuori/trolley-pb`.

The learning rate depends on the size of the instances: $\alpha = 2^{-12}/\overline{n}$ where $\overline{n}$ is the average number of task in the batch. The rationale is that the magnitude of the gradient depends on the tardiness $L(\sigma)$ which tends to grow with the number of operations. Therefore, we use the learning rate α to offset this growth, which is key to have a stable convergence. For the two methods using the stochastic policy, we made the first run deterministic i.e. the policy becomes stochastic only after the first restart for the CP based one, and after the first iteration for the multi-start local search.

The results are presented in Table 2. We report the number of solved instances (among 10 instances for every time horizon) averaged over 10 randomized runs for each CP model in the column "#S". The synthetic dataset is more constrained than the industrial dataset and the two basic CP models fail to solve most of the instances ("–" in the table indicates a time out). However, the relative performance remains unchanged w.r.t. Table 1: the scheduling-based models shows better performance in terms of number of solved instances and CPU time while scaling better in memory. All three of the RL-based methods significantly outperform previous approaches. The results in Table 2 indicate that the rapid restarts approach dominates the others. However, it may not be as clear-cut as that: for other settings of the hyperparameters (α, β and γ) the relative efficiencies fluctuate and other methods can dominate. Moreover, one advantage of the multi-start local search method is that since due dates are relaxed, imperfect solutions can be produced, even for infeasible instances. We report the average maximum tardiness in column L_{max}.

This global $\boldsymbol{\theta}$ also works well with the industrial dataset. All instances are easily solved by all three methods, except "R" for the week horizon, which is only solved by the rapid restart approach. We learnt a dedicated policy for the industrial dataset with the same settings. It turns out that every instance was solved by the deterministic policy using the new value for $\boldsymbol{\theta}$, except the instance "L" for the week horizon. However, it is easily solved by all three methods.

7 Conclusion

In this paper we have applied reinforcement learning to design simple yet efficient stochastic decision policies for an industrial problem: planning the production process at Renault. Moreover, we have shown how to leverage these heuristic policies within constraint programming and within local search.

The resulting approaches significantly improve over the current local search method used at Renault. However, many instances on synthetic data remain unsolved. We plan on using richer machine learning models, such as neural networks, to represent states. Moreover, we would like to embed this heuristic in a Monte-Carlo Tree Search as it would fit well with our current approach since it relies on many rollouts. Finally, we would like to tackle the more general problem of assigning components to operators and then planning individual routes.

References

1. J. Christopher Beck, Patrick Prosser, and Evgeny Selensky. Vehicle Routing and Job Shop Scheduling: What's the Difference? In *Proceedings of the 13th International Conference on Automated Planning and Scheduling*, ICAPS, pages 267–276, 2003.
2. Irwan Bello, Hieu Pham, Quoc V. Le, Mohammad Norouzi, and Samy Bengio. Neural Combinatorial Optimization with Reinforcement Learning. In *5th International Conference on Learning Representations, Workshop Track Proceedings*, ICLR, 2017.
3. Frederic Boussemart, Fred Hemery, Christophe Lecoutre, and Lakhdar Sais. Boosting systematic search by weighting constraints. In *Proceedings of the 16th European Conference on Artificial Intelligence*, ECAI, pages 146–150, 2004.
4. Hanjun Dai, Elias B. Khalil, Yuyu Zhang, Bistra Dilkina, and Le Song. Learning Combinatorial Optimization Algorithms over Graphs. In *Proceedings of the 31st International Conference on Neural Information Processing Systems*, NIPS, page 6351–6361, 2017.
5. Michel Deudon, Pierre Cournut, Alexandre Lacoste, Yossiri Adulyasak, and Louis-Martin Rousseau. Learning Heuristics for the TSP by Policy Gradient. In *Proceedings of the 15th International Conference on the Integration of Constraint Programming, Artificial Intelligence, and Operations Research*, CPAIOR, pages 170–181, 2018.
6. Sylvain Ducomman, Hadrien Cambazard, and Bernard Penz. Alternative Filtering for the Weighted Circuit Constraint: Comparing Lower Bounds for the TSP and Solving TSPTW. In *Proceedings of the 30th AAAI Conference on Artificial Intelligence*, AAAI, pages 3390–3396, 2016.
7. Martin Josef Geiger, Lucas Kletzander, and Nysret Musliu. Solving the Torpedo Scheduling Problem. *Journal of Artificial Intelligence Research*, 66:1–32, 2019.
8. William D. Harvey and Matthew L. Ginsberg. Limited Discrepancy Search. In *Proceedings of the 14th International Joint Conference on Artificial Intelligence*, IJCAI, page 607–613, 1995.
9. Philippe Laborie. Algorithms for propagating resource constraints in AI planning and scheduling: Existing approaches and new results. *Artificial Intelligence*, 143(2):151–188, 2003.
10. Michael Luby, Alistair Sinclair, and David Zuckerman. Optimal Speedup of Las Vegas Algorithms. *Information Processing Letters*, 47:173–180, 1993.
11. Charles Prud'homme, Jean-Guillaume Fages, and Xavier Lorca. *Choco Documentation*. TASC - LS2N CNRS UMR 6241, COSLING S.A.S., 2017.
12. Stefan Ropke and David Pisinger. An adaptive large neighborhood search heuristic for the pickup and delivery problem with time windows. *Transportation Science*, 40(4):455–472, 2006.
13. Pierre Schaus. The Torpedo Scheduling Problem. `http://cp2016.a4cp.org/program/acp-challenge/`, 2016.
14. David Silver, Aja Huang, Chris J. Maddison, Arthur Guez, Laurent Sifre, George van den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, Sander Dieleman, Dominik Grewe, John Nham, Nal Kalchbrenner, Ilya Sutskever, Timothy 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.

15. David Silver, Julian Schrittwieser, Karen Simonyan, Ioannis Antonoglou, Aja Huang, Arthur Guez, Thomas Hubert, Lucas Baker, Matthew Lai, Adrian Bolton, Yutian Chen, Timothy Lillicrap, Fan Hui, Laurent Sifre, George van den Driessche, Thore Graepel, and Demis Hassabis. Mastering the game of Go without human knowledge. *Nature*, 550(7676):354–359, 2017.
16. Ronald J. Williams. Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Machine Learning*, 8(3):229–256, 1992.