DOI: 10.15439/2020F100
ISSN 2300-5963 ACSIS, Vol. 21

Reinforcement Learning Algorithms for Online Single-Machine Scheduling

Yuanyuan Li*, Edoardo Fadda†, Daniele Manerba‡,
Roberto Tadei† and Olivier Terzo*

*LINKS Foundation - Advanced Computing and Applications, 10138 Torino, Italy
Email: {yuanyuan.li, olivier.terzo}@linksfoundation.com
†Department of Control and Computer Engineering, Politecnico di Torino, 10129 Torino, Italy
Email: {edoardo.fadda, roberto.tadei}@polito.it
‡Department of Information Engineering, University of Brescia, 25123 Brescia, Italy
Email: daniele.manerba@unibs.it

***Abstract*—Online scheduling has been an attractive field of research for over three decades. Some recent developments suggest that Reinforcement Learning (RL) techniques have the potential to deal with online scheduling issues effectively. Driven by an industrial application, in this paper we apply four of the most important RL techniques, namely *Q-learning*, *Sarsa*, *Watkins's Q(λ)*, and *Sarsa(λ)*, to the online single-machine scheduling problem. Our main goal is to provide insights on how such techniques perform. The numerical results show that Watkins's Q(λ) performs best in minimizing the total tardiness of the scheduling process.**

I. Introduction

PRODUCTION scheduling is one of the most important aspects to address in many manufacturing companies (see [1]). The optimization problems arising within production scheduling can be of *static* or *dynamic* type (see [2]). In contrast with the static case, in which specifications and requirements are fully and deterministically known in advance, in the dynamic one, additional information (e.g., new orders, changes of available resources) may arrive during the production process itself. In this paper, we will consider the latter case, commonly called *online scheduling*, mainly fostered by our experience on an industrial project (Plastic and Rubber 4.0[1]) in which frequent occurrences of unexpected events call for more dynamic and flexible scheduling (see [3]).

In particular, we will focus on online single-machine scheduling problems with release dates and preemption allowed, in which the objective is to minimize the total tardiness. Let us consider a set $\mathcal{J}$ of jobs that are released over time. As soon as a job arrives, it is added to the end of a waiting queue. For each job $j \in \mathcal{J}$, let dt_j be its due time and ct_j its completion time. The goal of the problem is to arrange the jobs of the queue, so to minimize the total tardiness calculated as $\Gamma = \sum_{j\in\mathcal{J}} ta_j$, where $ta_j := \max\{0, ct_j - dt_j\}$. The motivation of studying a single-machine problem relies on the fact that, in the plastic and rubber manufacturing, the process of transforming raw material into a final product just goes through one or two machines. On the other hand, even for those manufacturing requiring multiple-machine scheduling problems, each machine represents a basic block of a chain. Thus improper usage of a machine can slow down the whole production process.

The easiest way to deal with scheduling in a dynamic context is the use of the so-called *dispatching rules*. These rules first prioritize jobs waiting for being processed and then select the job with a greedy evaluation whenever a machine gets free (see Section II for more details). While most dispatching rules simply schedule on a local view basis, other smarter approaches can be used to provide better results in the long run. For instance, Reinforcement Learning (RL) is a continuing and goal-directed learning paradigm, and it represents a promising approach to deal with online scheduling. The potential of RL on online scheduling has been revealed in several works (see, e.g., [4], [5], [6]). However, while most works compare a single RL algorithm with commonly-used dispatching rules, they lack in comparing different RL algorithms. A research question naturally arises: how do different RL algorithms perform on online scheduling?

Motivated by investigating the applicability of RL algorithms on online single-machine scheduling in detail, in this work, we will compare the following approaches' performance:

- a random assignment (*Random*) which simply selects a job randomly;
- one of the most popular dispatching rules, namely the *earliest due date* (*EDD*) rule;
- four RL approaches, namely *Q-learning*, *Sarsa*, *Watkins's Q(λ)*, and *Sarsa(λ)*.

Furthermore, we will test the algorithms under different operating conditions (e.g., the frequency of job arrivals). *Watkins's Q(λ)* seems the most promising method in most of the cases. Therefore, we contribute the literature on two different aspects: getting insights on the compared methods, and giving

[1] Plastic&Rubber 4.0. Piattaforma Tecnologica per la Fabbrica Intelligente (Technological Platform for Smart Factory), URL: https://www.regione.piemonte.it/web/temi/fondi-progetti-europei/fondo-euro\\peo-sviluppo-regionale-fesr/ricerca-sviluppo-tecnologico-innovazione/piatta\\forma-tecnologica-fabbrica-intelligente

practitioners suggestions on selecting the best method against the specific situation. Notice that comparing and evaluating different algorithms against various aspects and performance indicators is a commonly adopted research methodology (see, e.g., [7], [8], [9], [10], [11], and [12]). The specific comparison of RL algorithms can be found, for instance, in the game field. In [13], the authors compared two RL algorithms (*Q-learning* and *Sarsa*) through the simulation of bargaining games. Even though the two algorithms present slight differences, they might have essentially different simulation results, as reflected in our experiment (see Section IV).

Finally, we also propose some preliminary results obtained by the use of *Deep Q Network* (*DQN*), which utilizes the power of neural networks to approximate the value function (see [14] for a review about DQN). However, our experiments will show that *DQN* is better suited for high-dimensional inputs. In contrast, with smaller input settings, *DQN* has a longer training time and obtains results that are far from the performance of *Watkins's Q(λ)*.

The rest of the paper is organized as follows. Section II is dedicated to a general overview of RL techniques, while Section III introduces and reviews some previous works using RL approaches on scheduling problems. Section IV describes the algorithmic framework for the online single-machine problem. Section V defines the simulation procedure, and the simulation results from three different types of experiments (Section VI). Finally, in Section VII, the paper concludes with a summary of the findings and some future lines.

II. Reinforcement Learning

RL is a branch of Machine Learning that improves automatically through experience. It comes from three main research branches: the first relates to learning by trial-and-error, the second relates to optimal control problems, and the last relates to temporal-difference methods (see [15]). The three approaches converged together in the late eighties to produce the modern RL.

RL approaches can be applied to scenarios in which a decision-maker called *agent* interacts with a set of *states* called *environment* by means of a set of possible *actions*. A *reward* is given to the agent in each specific state. In this paper, we consider a discrete time system, i.e. defined over a finite set $\mathcal{T}$ of time steps with its cardinality being called *time horizon*. As shown in Figure 1, at each time step $t \in \mathcal{T}$, an agent in state S_t takes action A_t, then, the environment reacts by changing into state S_{t+1} and by rewarding the agent of R_{t+1}. The interaction starts from an initial state, and it continues until the end of the time horizon. Such a sequence of actions is named an *episode*. In the following, $\mathcal{E}$ will represent the set of episodes.

Each *state* of the system is associated with a *value function* that estimates the expected future reward achievable from that state. Each state-action pair (S_t, A_t) is associated with a so-called *Q-function* $Q(S_t, A_t)$ that measures the future reward achievable by implementing action A_t in state S_t. The agent's goal is to find the best *policy*, which is a function mapping the set of states to the set of actions, maximizing the cumulative *reward*. If exact knowledge of the Q-function is available, the best policy for each state is defined by $\max_a Q(S_t, a)$.

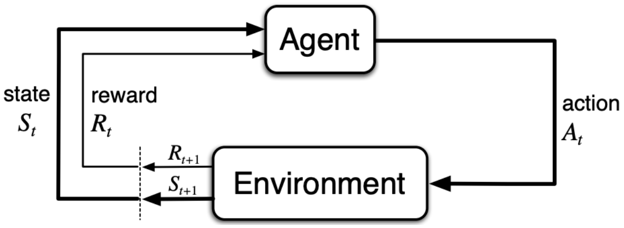


Fig. 1. The agent-environment interaction in RL [15].

To estimate the value functions $Q(s, a)$ and discover the optimal policies, three main classes of RL techniques exist Monte Carlo (MC)-based, Dynamic Programming (DP)-based methods, and temporal-difference (TD)-based methods. Unlike DP-based methods, which require complete knowledge of all the possible transitions, MC-based methods only require some experience and the possibility to sample from the environment randomly. TD-based methods are a sort of combination of MC-based and DP-based ones: they sample from the environment like in MC-based methods and perform updates based on current estimates like DP-based ones. Moreover, TD-based methods are also appreciated for being flexible, easy to implement, and computationally fast. For these reasons, in this paper, we will consider only RL algorithms belonging to the TD-based methods. Even if several TD-based RL algorithms have been introduced in the literature, the most used are *Sarsa* (an acronym for State-Action-Reward-State-Action), *Q-learning* and their variations, e.g. the *Watkins's Q(λ)* method and the *Sarsa(λ)* (see [16]).

III. Literature Review

Since online scheduling has been an active field for several decades, an in-depth analysis of the literature review is out of scope for the present paper. Thus, in this section, we recall some of the most traditional approaches to online scheduling, and we review the main applications of RL to this problem.

Differently from tailored algorithms (heuristic and exact methods), which might require effort in implementation and calibration over a broad set of parameters, dispatching rules are widely adopted for online scheduling for their simplicity (see, e.g., [17]). For instance, the *earliest due date* (*EDD*) dispatching rule is one of the most commonly used ones in practical applications [18]. *EDD* simply schedules first the job with the earliest due date. Again, in [19], the authors propose a deterministic greedy algorithm known as *list scheduling* (*LS*), which simply assigns each job to the machine with the smallest load. For more details, we refer the reader to the work [20] that classified over one hundred dispatching rules. In [21], the authors designed a deterministic algorithm and a randomized one for online machine sequencing problems using Linear Programming techniques. At the same time, in [22], the authors proposed an algorithm to make jobs artificially available to the online scheduler by delaying the release time of jobs.

In online scheduling, a decision-maker is regularly scheduling jobs over time, attempting to reach the overall best performance. Therefore, it is reasonable that RL represents one of the possible techniques able to exploit such a setting.

In [4], the authors interpreted job-shop scheduling problems as sequential decision processes. They try to improve the job dispatching decisions of the agent by employing an RL algorithm. Experimental results on numerous benchmark instances showed the competitiveness of the RL algorithm. More recently, in [6], the authors modeled the scheduling problem as a Markov Decision Process and solved it through a simulation-based value iteration and a simulation-based *Q-learning*. Their results clearly showed that such RL algorithms could achieve better performance concerning several dispatching heuristics, disclosing the potential of RL application in the field. In the context of an online single-machine environment, in [23], the authors compared the performance of *neural fitted Q-learning* techniques using combinations of different states, actions, and rewards. They proved that taking only the necessary inputs of states and actions is more efficient.

While all the discussed works revealed the competitiveness of RL on scheduling problems, a further comparison of the performance among various RL algorithms is still missing in the scheduling literature. With the knowledge of the available studies showing the potential of RL and the demand from the industrial application, we are motivated to compare different RL approaches' performance on online scheduling for getting more insights. In particular, we carry out experimental studies on four of the most commonly used model-free RL algorithms, namely *Q-learning*, *Sarsa*, *Watkins's Q(λ)*, and *Sarsa(λ)*. Our comparison methodology is inspired by [23], in which the best configuration for minimizing maximal lateness is pursued. In our work, instead, we aim at minimizing the total tardiness of the scheduling process. Moreover, another major difference with their work lies in the way we evaluate the results. While they used the result from one run, our results come from 50 runs with different random seeds, and two different time step sizes are tested (the interaction between agent and environment is checked in each step). Also, we further test a neural network-based RL technique showing that it is not necessary to use such a combination when the state space is limited.

IV. Reinforcement Learning Algorithms for Online Scheduling

In this section, we describe the algorithmic framework used to deal with our online single-machine scheduling problem. In particular, we provide several variants based on different RL techniques.

A. States, actions, and rewards

To be approached by RL techniques, we define our problem setting along the lines used in [23]. In particular:

- *state*: a state is associated with each possible length of the jobs in the waiting queue;
- *action*: if not all the jobs are finished, the action is either to select one new job from a specific position of the waiting queue and start processing it (we recall that preemption is allowed), or to continue processing the job which has been already assigned to the machine in the previous step;
- *reward*: since RL techniques aim at maximizing rewards while our problem aims at minimizing the total tardiness, we set the reward of a state as the opposite value of its total tardiness.

When the action implies the selection of a job from a certain position in the waiting queue, it is important to decide the order in which jobs are stored inside the queue. Therefore, we implemented three possible ordering of jobs which provide very different scheduling effects:

- jobs are unsorted (*UNSORT*), i.e., they have the same order as the arrivals;
- jobs are sorted by increasing value of due time (*DT*);
- all unfinished jobs are sorted by increasing the value of the sum of due time and processing time (*DT+PT*).

For instance, by using *DT*, if the action is to select a job in the second position of the queue, the job with the second earliest due time will be processed.

B. RL algorithms adopted

We have decided to implement four different RL algorithms, namely *Q-learning*, *Sarsa*, *Watkins's Q(λ)*, and *Sarsa(λ)*. They are described in the following. Here are some notations used:

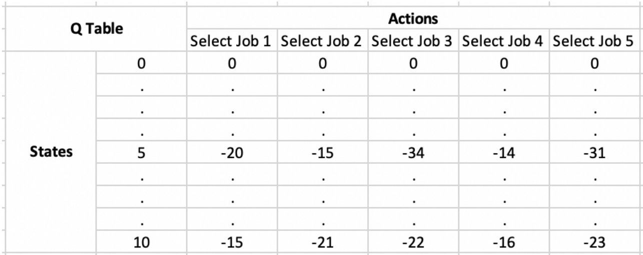
- s state;
- a action;
- $\mathcal{S}$ set of nonterminal states;
- $\mathcal{A}(s)$ set of actions possible in state s;
- S_t state at t;
- A_t action at t;
- R_t reward at t.

1) Q-learning: Q-learning is a technique that learns the value of an optimal policy independently of the agent's action. It is largely adopted for its simplicity in the analysis of the algorithm and for the possibility of early convergence proofs by directly approximating the optimal action-value function (see [16] and [15]). The updating rule for the estimation of the Q-function is:

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha[R_{t+1} + \gamma \max_a Q(S_{t+1}, a) - Q(S_t, A_t)]. \quad (1)$$

The $Q(S_t, A_t)$ function estimates the quality of state-action pair. At each time step t, the reward R_{t+1} from state S_t to S_{t+1} is calculated and $Q(S_t, A_t)$ is updated accordingly. The coefficient α is the learning rate ($0 \leq \alpha \leq 1$); it determines the extent that new information overrides the old information. Furthermore, γ is the discount factor determining the importance of future reward and finally, $\max_a Q(S_{t+1}, a)$ is the estimation of best future value.

The values of the Q-function are stored in a look-up table called *Q table*. Figure 2 displays an example of Q table storing Q-function values for states from 0 to 10 (in row) and actions from selecting *Job 1* to *Job 5* (in column). By overlooking the actual policy being followed in deciding the next action, *Q-learning* simplifies the analysis of the algorithm and enabled early convergence proofs.



Q Table		Actions				
		Select Job 1	Select Job 2	Select Job 3	Select Job 4	Select Job 5
States	0	0	0	0	0	0
	.	.	.	.	.	.
	.	.	.	.	.	.
	.	.	.	.	.	.
	5	-20	-15	-34	-14	-31
	.	.	.	.	.	.
	.	.	.	.	.	.
	.	.	.	.	.	.
	10	-15	-21	-22	-16	-23

Fig. 2. An example of Q table.

2) Sarsa: Sarsa is a technique that updates the estimated Q-function by following the experience gained from executing some policies (see [24] and [15]). The updating rule for the estimation of the Q-function is:

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \\ \alpha[R_{t+1} + \gamma Q(S_{t+1}, A_{t+1}) - Q(S_t, A_t)]. \quad (2)$$

The structure of formula (2) is similar to (1). The only difference is that (2) considers the actual action implemented in the next step A_{t+1}, instead of the generic best action $\max_a Q(S_{t+1}, a)$.

As for *Q-learning*, also in *Sarsa* the values of the Q-function are stored in a *Q table*. Despite the more expensive behaviour with respect to *Q-learning*, *Sarsa* may provide better online performances in some scenarios (as shown by the *Cliff Walking* example in [15]).

3) Watkins's Q(λ): Watkins's Q(λ) is a well-known variant of *Q-learning*. The main difference with respect to classical *Q-learning* is the presence of a so-called *eligibility trace*, i.e. a temporary record of the occurrence of an event, such as the visiting of a state or the taking of an action. The trace marks the memory parameters associated with the event as eligible for undergoing learning changes. A trace is initialized when a state is visited or an action is taken, and then the trace gets decayed over time according to a decaying parameter λ (with $0 \le \lambda \le 1$). Let us call $e_t(s, a)$ the trace for a state-action pair (s, a). Let us also define an indicator parameter $\mathbb{1}_{xy}$ that takes value 1 if and only if x and y are the same, and 0 otherwise. Then, for any (s, a) pair (for all $s \in \mathcal{S}$, $a \in \mathcal{A}$), the updating rule for the estimation of the Q-function is:

$$Q_{t+1}(s, a) = Q_t(s, a) + \alpha \delta_t e_t(s, a) \quad (3)$$

where

$$\delta_t = R_{t+1} + \gamma \max_{a'} Q_t(S_{t+1}, a') - Q_t(S_t, A_t) \quad (4)$$

and

$$e_t(s, a) = \gamma \lambda e_{t-1}(s, a) + \mathbb{1}_{sS_t} \mathbb{1}_{aA_t} \quad (5)$$

if $Q_{t-1}(S_t, A_t) = \max_a Q_{t-1}(S_t, a)$, and $\mathbb{1}_{sS_t} \mathbb{1}_{aA_t}$ otherwise.

As the reader can notice, by plugging Eq. (4) into Eq. (3), we obtain an equation similar to (1) but with the additional eligibility term that increments the value of δ_t if the state and action selected by the algorithm are one of the eligibility states. In the rest of the paper we use *Q(λ)* referring to *Watkins's Q(λ)*.

4) Sarsa(λ): Similarly to *Q(λ)*, the *Sarsa(λ)* algorithm represents a combination between *Sarsa* and eligibility traces to obtain a more general method that may learn more efficiently. Here, for any (s, a) pair (for all $s \in \mathcal{S}$, $a \in \mathcal{A}$), the updating rule for the estimation of the Q-function is:

$$Q_{t+1}(s, a) = Q_t(s, a) + \alpha \delta_t e_t(s, a) \quad (6)$$

where

$$\delta_t = R_{t+1} + \gamma Q_t(S_{t+1}, A_{t+1}) - Q_t(S_t, A_t) \quad (7)$$

and

$$e_t(s, a) = \gamma \lambda e_{t-1}(s, a) + \mathbb{1}_{sS_t} \mathbb{1}_{aA_t} \quad (8)$$

Unlike Eq. (5), there is no other condition (set the eligibility traces to 0 whenever a non-greedy action is taken) added. A deeper discussion about the interpretation of the formulas is given in [15].

V. Simulation Procedure

In order to perform the comparison under interest, we create an online scheduling simulation procedure as described in Algorithm 1.

Algorithm 1 Online scheduling simulation through RL algorithms

```
Require: $|\mathcal{E}|$ number of episodes; $|\mathcal{T}|$ number of time-steps;
Initialize $Q(s, a) = 0, \forall\ s \in \mathcal{S}, a \in \mathcal{A}$;
for $\eta \leftarrow 1$ to $|\mathcal{E}|$ do
    Initialize $S$
    for $t \leftarrow 1$ to $|\mathcal{T}|$ do
        if new jobs arrive then
            Update waiting list $L$
        end if
        if $L$ is not empty then
            Take $A_t$ in $S_t$, observe $R_t$, $S_{t+1}$
            Calculate $A_{t+1}$ and update $Q_t$
            $S_t \leftarrow S_{t+1}$, $A_t \leftarrow A_{t+1}$
        end if
    end for
end for
```

We first update Q tables through a training phase then use the Q tables to select actions in the test phase.

The arrival time of job j are distributed according to an exponential distribution, i.e., $X_j \sim exp(r)$ with the rate parameter valued $r = 0.1$. It is simulated in this way: at the first time step, a random number of jobs (from 1 to 6 jobs) and an interval time (following the exponential distribution) are generated. Once a job is generated (simulating the arrival of the job), it will be put into the waiting queue immediately. Then at the next time step, if the interval time is passed, new jobs will be generated and put into the waiting queue; meanwhile, a new interval time will be created. Otherwise, nothing is created. Then the same procedure repeats till reaching a final state.

For the settings regarding RL algorithms:

- In the policy, $\epsilon = 0.1$ enabling highly greedy actions while keeping some randomness in job selections;
- In the value function, $\alpha = 0.6$, i.e., there is a bit higher tendency to explore more possibilities while a bit lower in keeping exploiting old information, whereas $\gamma = 1.0$, which means it strives for a long-term high reward;
- In the eligibility traces, λ is 0.95, a high decaying value is leading to a longer-lasting trace.

It is worth noting that all the algorithms considered are heuristics. Thus they focus on finding a good solution in a short amount of time by finding a balance between intensified and diversified explorations of the solution space. Nevertheless, the plain implantation of the algorithms above does not ensure enough diversification. For this reason, it is common to use a ϵ-greedy method. Thus, with probability ϵ, exploration is chosen, which means the action is chosen uniformly at random between the available ones. Instead, with probability $1 - \epsilon$, exploitation is chosen by taking the actions with the highest values greedily. After knowing the way to balance exploration and exploitation, we need to define a learning method for finding out policies leading to higher cumulative rewards.

In an episode, we start a new schedule by initializing state S and terminates when either reaching the maximum steps or no jobs to process. To simulate real-time scheduling, for each episode, we check the arrivals of new jobs and update the waiting queue if there are, then we choose the action A, and calculate the reward R and the next state S' accordingly. The Q functions are updated according to the exact RL algorithms used. The same procedure is carried out in both training and test phases except that in the test, the Q table is not initialized with empty values but obtained from the training phase.

Let us see a training example with *Q-learning* to see for the same schedule how the reward is accumulated, and objective value evolves with more episodes passing by. In Fig. 3, the graph on the bottom shows after around 80 episodes, the reward reaches the maximum and holds steady. Accordingly, the objective value - total tardiness drops more slowly after around 80 episodes. While the reward keeps stable, total tardiness continues dropping to around 4,0000. To summarize, using total tardiness as a reward is useful, but it is still challenging to represent the trend of the objective value adequately.

VI. Numerical Experiments

In this section, we propose three different experimental results. Section VI-A compares the performance among random assignment (*Random*), *EDD*, and the four RL approaches implemented. Section VI-B investigates the possible impact of different operating conditions (i.e., frequency of jobs arrivals) on the RL approaches. Finally, Section VI-C compares *Q(λ)* and *DQN*.

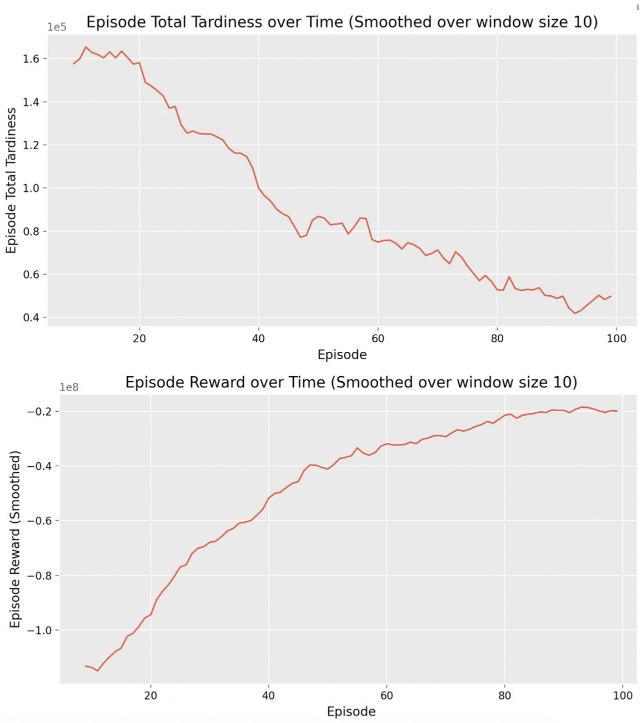


Fig. 3. The changes to reward and the objective value (total tardiness) of 100 episodes.

The algorithms have been implemented in Python 3.6. To avoid possible ambiguities, we locate the related code in a public repository[2]. All the experiments are carried out on an *Intel Core i5* CPU@2.3GHz machine equipped with 8GB RAM and running *MacOS* v10.15.4 operating system.

A. RL algorithms vs Random and EDD

To check if considering different time horizons leads to different results, we consider two experiments in which the time horizon $\mathcal{T}$ is set to 2500 and 5000, respectively. For each of the settings, we ran 50 tests with different random seeds. For each algorithm Θ, we call $\Gamma_{\zeta\Theta}$ the total tardiness achieved in simulation ζ . Furthermore, we define $\rho_{\zeta\Theta}$ to be the percentage gap between the total tardiness achieved by the best algorithm and by algorithm Θ during run ζ, i.e.,

$$\rho_{\zeta\Theta} = \frac{\Gamma_{\zeta\Theta}}{\min_{\zeta\Theta} \Gamma_{\zeta\Theta}}. \tag{9}$$

To compare the different algorithms, we consider the average value of $\rho_{\zeta\Theta}$ concerning all the runs.

The simulation results with the algorithms (under different job orders, time horizons) are displayed in Table I, where

[2] URL: https://github.com/Yuanyuan517/RL_OnlineScheduling.git

avg($\rho_{\zeta\Theta}$), std($\rho_{\zeta\Theta}$) are respectively the mean and standard deviations of $\rho_{\zeta\Theta}$. The best value among all the combinations of algorithms and jobs order policies for each time horizon is highlighted in bold font.

TABLE I: Experiment cases of the algorithms with different settings

		$\lvert\mathcal{T}\rvert$=2500		$\lvert\mathcal{T}\rvert$=5000	
Algorithm	**Jobs order**	avg($\rho_{\zeta\Theta}$)	std($\rho_{\zeta\Theta}$)	avg($\rho_{\zeta\Theta}$)	std($\rho_{\zeta\Theta}$)
Random	-	2.59	0.50	3.06	0.69
EDD	-	7.67	1.76	9.19	1.47
Q-learning	*UNSORT*	2.15	0.43	2.04	0.35
Q-learning	*DT*	1.45	0.28	1.29	0.20
Q-learning	*DT+PT*	1.44	0.30	1.25	0.18
Sarsa	*UNSORT*	2.55	0.53	2.47	0.39
Sarsa	*DT*	1.65	0.40	1.76	0.36
Sarsa	*DT+PT*	1.66	0.47	1.68	0.33
Sarsa(λ)	*UNSORT*	4.42	0.93	5.04	0.93
Sarsa(λ)	*DT*	7.04	1.35	7.73	1.34
Sarsa(λ)	*DT+PT*	3.08	1.03	7.70	1.33
Q(λ)	*UNSORT*	2.04	0.42	2.01	0.40
Q(λ)	*DT*	**1.11**	**0.18**	1.13	0.17
Q(λ)	*DT+PT*	1.19	0.26	**1.09**	**0.14**

While [23] shows *EDD* gets a better result than RL to minimize the maximum tardiness, with the new objective of minimizing total tardiness in our experiments, all RL algorithms get better results than *EDD*.

As shown in Table I, the size of running time steps influenced the result on job order but does not influence the algorithm. And for the case with 2500 steps, the configuration *Q(λ)* plus *DT* gets the best result, instead for 5000 steps, the configuration *Q(λ)* plus *DT+PT* gets the best result.

Besides, we find with the sorting choice *DT+PT* that all algorithms get smaller average values except for the configuration *Q(λ)* with 2500 steps. Comparatively, a randomly sorting job leads to a much worse result.

B. Q(λ) performance against different job arrival rates

Another test is on the operating condition - the frequency of job arrivals for the two best combinations *Q(λ)* plus *DT* and *Q(λ)* plus *DT+PT*, which is controlled by the rate parameter r. To understand whether the value of r affects performance, we experimented with 2 more values, i.e. $r = \{0.05, 0.2\}$ in addition to the previous one $r = 0.1$.

In Table II, the results are also normalized by following Eq. (9) with 50 tests and $|\mathcal{T}| = 2500$ for each test. As shown in the table, with small $r = 0.05$, $r = 0.1$ (indicating jobs arrive much less frequently than the last one), the version with jobs ordered by *DT* performs better. When jobs arrive much more frequently, the version sorted by *DT + PT* wins. Hence a careful selection of algorithms and settings according to the operating conditions matters.

TABLE II: Experiment cases of the rate parameter with best settings from *Q(λ)*.

Jobs order	r	avg($\rho_{\zeta\Theta}$)	std($\rho_{\zeta\Theta}$)
DT	0.05	**1.14**	**0.18**
DT+PT	0.05	1.17	0.55
DT	0.10	**1.10**	**0.17**
DT+PT	0.10	1.17	0.26
DT	0.20	1.17	0.28
DT+PT	0.20	**1.12**	**0.24**

C. Comparison between Q(λ) and DQN

In the third test we compare a four-layer DQN and *Q(λ)* plus *DT+PT*, i.e. the better performing RL algorithm according to Table I. Figure 4 shows such a comparison. The result is from running 50 tests and $|\mathcal{T}| = 5000$ in each test. The horizontal axis represents the total tardiness and the vertical axis shows the probability the objective value falls in. The dark yellow area indicates the overlapping between *Q(λ)* and DQN.

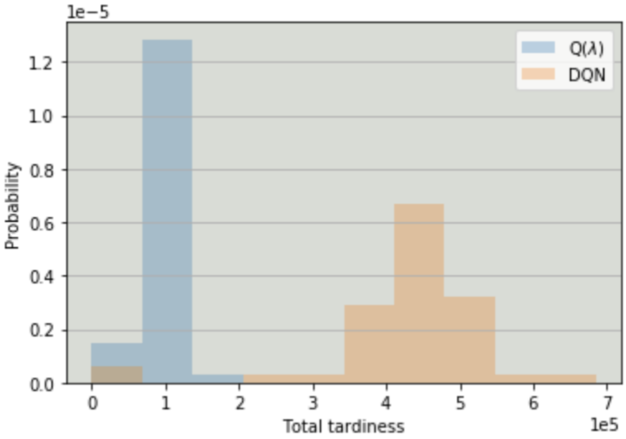


Fig. 4. The comparison between *Q(λ)* and DQN on the total tardiness of 50 runs with different seeds representing different schedules.

We can see *Q(λ)* has much higher probability with smaller objective value, which indicates *Q(λ)* outperforms DQN. Taking into account the time spent in training DQN is almost 10 times of *Q(λ)*, *Q(λ)* is a better option, especially for guaranteeing a flexible and adaptive scheduling in realtime.

VII. Conclusions and Future Work

In this paper, we compared four RL methods, namely *Q-learning*, *Sarsa*, *Watkins's Q(λ)*, and *Sarsa(λ)*, with *EDD* and random assignment on an online single-machine scheduling problem. The experiments show that:

- better scheduling performance is achieved by the RL method *Watkins's Q(λ)*, especially when the action concerns the selection of jobs sorted by due date for the smaller time horizon ($|\mathcal{T}| = 2500$) and the selection of jobs sorted by due date and processing time for bigger time horizon ($|\mathcal{T}| = 5000$).
- the tests on r disclose the combination of *Q(λ)* and job orders have different performances in various operating conditions.

- slight difference in algorithms can profoundly change the results.

Besides, with limited input, using DQN is too costly for extended running time and energy spent in adjusting parameters to guarantee a good result. The results above indicate careful analysis should be done from different angles (running time, operating conditions, average results from multiple experiments) for making a wiser selection of algorithms.

Furthermore, with multiple machines, more transitions must be considered, which need more representational state information. Thus it will be impossible to store values of all state-action pairs in a Q table. DQN may take a leading role then. As indicated by the work [25], unpredictable changes may happen at different places in the state-action space, and more care should be taken to avoid instabilities of DQN. One techniques that can acheive this goal is the usage of kernel function (see [26]), this builds a future research avenue.

Acknowledgement

This research was partially supported by the Plastic and Rubber 4.0 (P&R4.0) research project, POR FESR 2014-2020 - Action I.1b.2.2, funded by Piedmont Region (Italy), Contract No. 319-31. The authors acknowledge all the project partners for their contribution.

References

[1] P. Brucker, *Scheduling Algorithms*, 5th ed. Springer Publishing Company, Incorporated, 2010.

[2] S. C. Graves, “A review of production scheduling,” *Operations Research*, vol. 29, no. 4, pp. 646–675, 1981.

[3] Y. Li, S. Carabelli, E. Fadda, D. Manerba, R. Tadei, and O. Terzo, “Machine learning and optimization for production rescheduling in industry 4.0,” *The International Journal of Advanced Manufacturing Technology*, pp. 1–19, 2020.

[4] T. Gabel and M. Riedmiller, “Adaptive reactive job-shop scheduling with reinforcement learning agents,” *International Journal of Information Technology and Intelligent Computing*, vol. 24, no. 4, pp. 14–18, 2008.

[5] H. Sharma and S. Jain, “Online learning algorithms for dynamic scheduling problems,” in *2011 Second International Conference on Emerging Applications of Information Technology*, 2011, pp. 31–34.

[6] T. Zhang, S. Xie, and O. Rose, “Real-time job shop scheduling based on simulation and markov decision processes,” in *2017 Winter Simulation Conference (WSC)*. IEEE, 2017, pp. 3899–3907.

[7] P. Castrogiovanni, E. Fadda, G. Perboli, and A. Rizzo, “Smartphone data classification technique for detecting the usage of public or private transportation modes,” *IEEE Access*, vol. 8, pp. 58 377–58 391, 2020.

[8] E. Fadda, P. Plebani, and M. Vitali, “Optimizing monitorability of multi-cloud applications,” 06 2016, pp. 411–426.

[9] E. Fadda, G. Perboli, and G. Squillero, “Adaptive batteries exploiting on-line steady-state evolution strategy,” in *Applications of Evolutionary Computation*, G. Squillero and K. Sim, Eds. Cham: Springer International Publishing, 2017, pp. 329–341.

[10] E. Fadda, D. Manerba, R. Tadei, P. Camurati, and G. Cabodi, “KPIs for Optimal Location of charging stations for Electric Vehicles: the Biella case-study,” in *Proceedings of the 2019 Federated Conference on Computer Science and Information Systems*, ser. Annals of Computer Science and Information Systems, M. Ganzha, L. Maciaszek, and M. Paprzycki, Eds., vol. 18. IEEE, 2019, pp. 123–126. [Online]. Available: http://dx.doi.org/10.15439/2019F171

[11] E. Fadda, D. Manerba, G. Cabodi, P. Camurati, and R. Tadei, “Comparative analysis of models and performance indicators for optimal service facility location,” *Transportation Research part E: Logistics and Transportation Reviews (submitted)*, 2020.

[12] R. Giusti, C. Iorfida, Y. Li, D. Manerba, S. Musso, G. Perboli, R. Tadei, and S. Yuan, “Sustainable and de-stressed international supply-chains through the synchro-net approach,” *Sustainability*, vol. 11, p. 1083, 02 2019.

[13] K. Takadama and H. Fujita, “Toward guidelines for modeling learning agents in multiagent-based simulation: Implications from q-learning and sarsa agents,” in *International Workshop on Multi-Agent Systems and Agent-Based Simulation*. Springer, 2004, pp. 159–172.

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

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

[16] C. J. C. H. Watkins, *Learning from delayed rewards*. Thesis Submitted for Ph.D., King’s College, Cambridge, 1989.

[17] A. Kaban, Z. Othman, and D. Rohmah, “Comparison of dispatching rules in job-shop scheduling problem using simulation: a case study,” *International Journal of Simulation Modelling*, vol. 11, no. 3, pp. 129–140, 2012.

[18] H. Suwa and H. Sandoh, *Online scheduling in manufacturing: A cumulative delay approach*. Springer Science & Business Media, 2012.

[19] R. L. Graham, “Bounds for certain multiprocessing anomalies,” *Bell System Technical Journal*, vol. 45, no. 9, pp. 1563–1581, 1966.

[20] S. S. Panwalkar and W. Iskander, “A survey of scheduling rules,” *Operations Research*, vol. 25, no. 1, pp. 45–61, 1977.

[21] J. R. Correa and M. R. Wagner, “Lp-based online scheduling: from single to parallel machines,” *Mathematical Programming*, vol. 119, no. 1, pp. 109–136, 2009.

[22] X. Lu, R. Sitters, and L. Stougie, “A class of on-line scheduling algorithms to minimize total completion time,” *Operations Research Letters*, vol. 31, no. 3, pp. 232–236, 2003.

[23] S. Xie, T. Zhang, and O. Rose, “Online single machine scheduling based on simulation and reinforcement learning,” in *Simulation in Produktion und Logistik 2019*. Simulation in Produktion und Logistik 2019, 2019.

[24] S. Singh, T. Jaakkola, M. L. Littman, and C. Szepesvári, “Convergence results for single-step on-policy reinforcement-learning algorithms,” *Machine learning*, vol. 38, no. 3, pp. 287–308, 2000.

[25] V. François-Lavet, R. Fonteneau, and D. Ernst, “How to discount deep reinforcement learning: Towards new dynamic strategies,” *arXiv preprint arXiv:1512.02011*, 2015.

[26] V. Cerone, E. Fadda, and D. Regruto, “A robust optimization approach to kernel-based nonparametric error-in-variables identification in the presence of bounded noise,” in *2017 American Control Conference (ACC)*. IEEE, may 2017. [Online]. Available: https://doi.org/10.23919