Journal of Scheduling manuscript No.
(will be inserted by the editor)

Discovering dispatching rules from data and imitation learning

Case study for the job-shop problem

Helga Ingimundardottir · **Thomas Philip Runarsson**

Received: October 13, 2015/ Accepted: date

Abstract Over the years there have been many approaches to create dispatching rules for scheduling. Recent past efforts have focused on direct search methods (e.g. genetic programming) or training on data (e.g. supervised learning). This paper will examine the latter and give a framework on how to do it effectively. Defining training data as $\{\mathbf{x}_i(k), y_i(k)\}_{k=1}^{K} \in \mathscr{D}$ then: *i)* samples $\mathbf{x}_i$ should represent the induced data distribution $\mathscr{D}$. This done by updating the learned model in an active imitation learning fashion; *ii)* y_i is labelled using a solver, and *iii)* data needs to be balanced, as the set is unbalanced w.r.t. dispatching step k.

When querying an expert policy, there is an abundance of valuable information that can be utilised for learning new models. For instance, it's possible to seek out when the scheduling process is most susceptible to failure. Furthermore, generally stepwise optimality (or classification accuracy) implies good end performance, here minimising the final makespan. However, as the impact of suboptimal moves is not fully understood, then the measure needs to be adjusted for its intended trajectory.

Using these guidelines, it becomes easier to create custom dispatching rules for one's particular application. For this paper three different distributions of job-shop will be considered. Moreover, the machine learning approach is based on preference learning which determines what feature states are preferable to others. However, that could easily be substituted for other methods.

Keywords Scheduling · Composite dispatching rules · Performance Analysis · Machine Learning · Imitation Learning · DAgger · Preference Learning

H. Ingimundardottir
Dunhaga 5, IS-107 Reykjavik, Iceland
Tel.: +354-525-4704
Fax: +354-525-4632
E-mail: hei2@hi.is

T.P. Runarsson
Hjardarhagi 2-6, IS-107 Reykjavik, Iceland
Tel.: +354-525-4733
Fax: +354-525-4632
E-mail: tpr@hi.is

1 Introduction

Hand crafting heuristics for scheduling is an ad-hoc approach to finding approximate solutions to the problems. The practice is time-consuming and its performance can even vary dramatically between different problem instances. The aim of this work is to increase our understanding of this process. In particular the learning of new problem specific priority dispatching rules (DR) will be addressed for a subclass of scheduling problems known as the job-shop scheduling problem (JSP).

A recent editorial of the state-of-the-art approaches [6] in advanced dispatching rules for large-scale manufacturing systems reminds us that: "... most traditional dispatching rules are based on historical data. With the emergence of data mining and on-line analytic processing, dispatching rules can now take predictive information into account." The importance of automated discovery of dispatching rules was also emphasised by [27]. Data for learning can also be generated using a known heuristic on a set of problem instances. Such an approach is taken in [23] for single-machine where a decision tree is learned from the data to have similar logic to the dispatching rule. However, this method cannot outperform the original dispatching rule for the data generation. This drawback is confronted in [25, 39, 29] by using an optimal scheduler or policy, computed off-line, for data generation. The resulting dispatching rules, as decision trees, gave significantly better schedules than using popular heuristics in that field, and a lower worst-case factor from optimality. Although, using optimal policies for creating training data gives vital information on how to learn good scheduling rules we will show that this is not sufficient. Once these rules make a suboptimal dispatch the are in uncharted territory and its effects are relatively unknown. This work will illustrate the sensitivity of learned dispatching rule's performance on the way the training data is sampled. For this purpose, JSP is used as a case study to illustrate a methodology for generating meaningful training data, which can be successfully learned using preference-based imitation learning.

The competing alternative to learning dispatching rules from data would be to search the dispatching rule space directly. The prevalent approach in this case would be using an evolutionary algorithm, such as genetic programming (GP). The main drawback, is that the rules from a GP framework can be quite complex, and difficult to interpret. In fact, [13] revisited the experiments from [41] for dynamic job-shop and tested it against some single priority dispatching rules, and found that it only slightly outperformed one rule, and was beat by another. The reason behind this staggering change in performance, may be due to the choice of objective function, and the underlying problem spaces that were used in training. It's argued that the randomly generated problem instances aren't a proper representative for real-world long-term job-shop applications, e.g., by the narrow choice of release times, yielding schedules that are overloading in the beginning phases.

A novel iterative dispatching rules that were evolved with GP for JSP, [28] learned from completed schedules in order to iteratively improve new ones. At each dispatching step, the method can utilise the current feature space to *correctify* some possible *bad* dispatch made previously (sort of reverse lookahead). Their method is straightforward, and thus easy to implement and more importantly computationally inexpensive, although the authors do stress that there is still remains room for improvement.

The most predominant approach in hyper-heuristics is a framework of creating *new* heuristics from a set of predefined heuristics via GA optimisation [3]. Adopting a two-stage hyper-heuristic approach to generate a set of machine-specific DRs for dynamic job-shop, [31] used GP to evolve composite priority dispatching rules (CDR) from basic attributes, along with evolutionary algorithm to assign a CDR to a specific machine. The problem space consists of job-shops in semiconductor manufacturing, with additional shop constraints, as machines are grouped to similar work centres, which can have different set-up time, workload, etc. In fact, the GP emphasised on efficiently dispatching on the work centres with set-up requirements and batching capabilities, which are rules that are non-trivial to determine manually.

Using improvement heuristics, [44] studied space shuttle payload processing by using reinforcement learning (RL), in particular, temporal difference learning. Starting with a relaxed problem, each job was scheduled as early as its temporal partial order would permit, there by initially ignoring any resource constraints on the machines, yielding the schedule's critical path. Then the schedule would be repaired so the resource constraints were satisfied in the minimum amount of iterations.

Using case based reasoning for timetable scheduling, training data in [2] is guided by the two best heuristics in the literature. They point out that in order for their framework to be successful, problem features need to be sufficiently explanatory and training data need to be selected carefully so they can suggest the appropriate solution for a specific range of new cases. Stressing the importance of meaningful feature selection.

With meta heuristics one can use existing DRs, and use for example portfolio-based algorithm selection [33,9], either based on a single instance or class of instances [42] to determine which DR to choose from. Implementing ant colony optimisation to select the best DR from a selection of nine DRs for JSP, experiments from [22] showed that the choice of DR do affect the results and that for all performance measures considered. They showed that it was better to have a all the DRs to choose from rather than just a single DR at a time.

Meta learning can be very fruitful in RL, as experiments from [20] discovered some key discriminants between competing algorithms for their particular problem instances, which provided them with a hybrid algorithm which combines the strengths of the algorithms.

The outline of the paper is the following, Section 2 gives the mathematical formalities of the scheduling problem, and Section 3 describes the main attributes for job-shop, and goes into how to create schedules with dispatching rules. Section 4 sets up the framework for learning from optimal schedules. In particular, the probability of choosing optimal decisions and the effects of making a suboptimal decision. Furthermore, the optimality of common single priority dispatching rules is investigated. With those guidelines, Section 5 goes into detail how to create meaningful composite priority dispatching rules using preference learning, focusing on how to compare operations and collect training data with the importance of good state sampling. Sections 6 and 7 explain the trajectories for sampling meaningful schedule state-spaces used in preference learning, either using passive or active imitation learning. Experimental results are jointly presented in Section 8 with comparison for a single randomly generated problem space. Furthermore, some general adjustments for performance boost is also considered. The paper finally concludes in Section 9 with discussion and conclusions.

2 Job-shop Scheduling

The job-shop problem (JSP) involves the scheduling of jobs on a set of machines. Each job consists of a number of operations which are then processed on the machines in a predetermined order. An optimal solution to the problem will depend on the specific objective.

In this study we will consider the $n \times m$ JSP, where n jobs, $\mathscr{J} = \{J_j\}_{j=1}^n$, are scheduled on a finite set, $\mathscr{M} = \{M_a\}_{a=1}^m$, of m machines. The index j refers to a job $J_j \in \mathscr{J}$ while the index a refers to a machine $M_a \in \mathscr{M}$. If a job requires a number of processing steps or operations, then the pair (j,a) refers to the operation, i.e., processing the task of job J_j on machine M_a.

Each job J_j has an indivisible operation time (or cost) on machine M_a, p_{ja}, which is assumed to be integral and finite. Starting time of job J_j on machine M_a is denoted $x_s(j,a)$ and its end time is denoted $x_e(j,a)$ where,

$$x_e(j,a) := x_s(j,a) + p_{ja} \tag{1}$$

Each job J_j has a specified processing order through the machines, it is a permutation vector, $\boldsymbol{\sigma}_j$, of $\{1,..,m\}$, representing a job J_j can be processed on $M_{\boldsymbol{\sigma}_j(a)}$ only after it has been completely processed on $M_{\boldsymbol{\sigma}_j(a-1)}$, i.e.,

$$x_s(j,\boldsymbol{\sigma}_j(a)) \geq x_e(j,\boldsymbol{\sigma}_j(a-1)) \tag{2}$$

for all $J_j \in \mathscr{J}$ and $a \in \{2,..,m\}$. Note, that each job can have its own distinctive flow pattern through the machines, which is independent of the other jobs. However, in the case that all jobs share the same *fixed* permutation route, referred to as flow-shop (FSP). A commonly used subclass of FSP in the literature is permutation flow-shop, which has the added constraint that the processing order of the jobs on the machines must be identical as well, i.e., no passing of jobs allowed [40].

The disjunctive condition that each machine can handle at most one job at a time is the following,

$$x_s(j,a) \geq x_e(j',a) \quad \text{or} \quad x_s(j',a) \geq x_e(j,a) \tag{3}$$

for all $J_j, J_{j'} \in \mathscr{J}$, $J_j \neq J_{j'}$ and $M_a \in \mathscr{M}$.

The objective function is to minimise its maximum completion times for all tasks, commonly referred to as the makespan, $C_{\max}$, which is defined as follows,

$$C_{\max} := \max\left\{x_e(j,\boldsymbol{\sigma}_j(m)) \,:\, J_j \in \mathscr{J}\right\}. \tag{4}$$

This family of scheduling problems is denoted by $J||C_{\max}$ [32]. Additional constraints commonly considered are job release-dates and due-dates or sequence dependent set-up times, however, these will not be considered here.

In order to find an optimal (or near optimal) solution for scheduling problems one could either use exact methods or heuristics methods. Exact methods guarantee an optimal solution. However, job-shop scheduling is strongly NP-hard [8]. Any exact algorithm generally suffers from the curse of dimensionality, which impedes the application in finding the global optimum in a reasonable amount of time. Using state-of-the-art software for solving scheduling problems, such as LiSA (A Library of Scheduling Algorithms) [1], which includes a specialised version of branch and bound that manages to find optimums for job-shop problems of up to 14×14 [38]. However, problems that are of greater size, become intractable. Heuristics are generally more time efficient but do not necessarily attain the global optimum. Therefore, job-shop has the reputation of being notoriously difficult to solve. As a result, it's been widely studied in deterministic scheduling theory and its class of problems has been tested on a plethora of different solution methodologies from various research fields [26], all from simple and straight forward dispatching rules to highly sophisticated frameworks.

3 Priority Dispatching Rules

Priority dispatching rules determine, from a list of incomplete jobs, $\mathscr{L}$, which job should be dispatched next. This process, where an example of a temporal partial schedule of six-jobs scheduled on five-machines, is illustrated in Figure 1. The numbers in the boxes represent the job identification j. The width of the box illustrates the processing times for a given job for a particular machine M_a (on the vertical axis). The dashed boxes represent the resulting partial schedule for when a particular job is scheduled next. Moreover, the current $C_{\max}$ is denoted by a dotted vertical line. The object is to keep this value as small as possible once all operations are complete. As shown in the example there are 15 operations already scheduled. The *sequence* of dispatches used to create this partial schedule is,

$$\boldsymbol{\chi} = (J_3,J_3,J_3,J_3,J_4,J_4,J_5,J_1,J_1,J_2,J_4,J_6,J_4,J_5,J_3) \tag{5}$$

This refers to the sequential ordering of job dispatches to machines, i.e., (j,a); the collective set of allocated jobs to machines is interpreted by its sequence which is referred to as a *schedule*. A *scheduling policy* will pertain to the manner in which the sequence is determined from the available jobs to be scheduled. In our example, the available jobs are given by the job-list $\mathscr{L}^{(k)} = \{J_1,J_2,J_4,J_5,J_6\}$ with the five potential jobs to be dispatched at step $k = 16$ (note that J_3 is completed).

However, deciding which job to dispatch is not sufficient as one must also know where to place it. In order to build tight schedules it is sensible to place a job as soon as it becomes available and such that the machine idle time is minimal, i.e., schedules are non-delay. There may also be a number of different options for such a placement. In Fig. 1 one observes that J_2, to be scheduled on M_3, could be placed immediately in a slot between J_3 and J_4, or after J_4 on this machine. If J_6 had been placed earlier, a slot would have been created between it and J_4, thus creating a third alternative, namely scheduling J_2 after J_6. The time in which machine M_a is idle between consecutive jobs J_j and $J_{j'}$ is called idle time, or slack,

$$s(a,j) := x_s(j,a) - x_e(j',a) \tag{6}$$

where J_j is the immediate successor of $J_{j'}$ on M_a.

Construction heuristics are designed in such a way that it limits the search space in a logical manner respecting not to exclude the optimum. Here, the construction heuristic, Υ, is to schedule the dispatches as closely together as possible, i.e., minimize the schedule's idle time. More specifically, once an operation (j,a) has been chosen from the job-list $\mathscr{L}$ by some dispatching rule, it can then be placed immediately after (but not prior) to $x_e(j,\boldsymbol{\sigma}_j(a-1))$ on machine M_a due to constraint Eq. (2). However, to guarantee that constraint Eq. (3) is not violated, idle times M_a are inspected as they create flow time which J_j can occupy. Bearing in mind that J_j release time is $x_e(j,\boldsymbol{\sigma}_j(a-1))$ one cannot implement Eq. (6) directly, instead it has to be updated as follows:

$$\tilde{s}(a,j') := x_s(j'',a) - \max\{x_e(j',a), x_e(j,\boldsymbol{\sigma}_j(a-1))\} \tag{7}$$

As all, already dispatched jobs, $J_{j'}, J_{j''} \in \mathscr{J}_a$ where $J_{j''}$ is $J_{j'}$ successor on M_a. Since preemption is not allowed, the only applicable slots are whose idle time can process the entire operation, i.e.

$$\tilde{S}_{ja} := \left\{ J_{j'} \in \mathscr{J}_a \,\middle|\, \tilde{s}(a,j') \geq p_{ja} \right\}. \tag{8}$$

The placement rule applied will decide where to place the job and is intrinsic to the construction heuristic, which is chosen independently of the priority dispatching rule that is applied. Different placement rules could be considered for selecting a slot from Eq. (8), e.g., if the main concern were to utilize the slot space, then choosing the slot with the smallest idle time would yield a closer-fitted schedule and leave greater idle times undiminished for subsequent dispatches on M_a. In our experiments, cases were discovered where such a placement could rule out the possibility of constructing optimal solutions. However, this problem did not occur when jobs are simply placed as early as possible, which is beneficial for subsequent dispatches for J_j. For this reason, it will be the placement rule applied here.

Priority dispatching rules will use attributes of operations, such as processing time, in order to determine the job with the highest priority. Consider again Figure 1, if the job with the shortest processing time (SPT) were to be scheduled next, then J_2 would be dispatched. Similarly, for the longest processing time (LPT) heuristic, J_5 would have the highest priority. Dispatching can also be based on attributes related to the partial schedule. Examples of these are dispatching the job with the most work remaining (MWR) or alternatively the least work remaining (LWR). A survey of more than 100 of such rules are presented in [30]. However, the reader is referred to an in-depth survey for simple or *single priority dispatching rule* (SDR) by [12]. The SDRs assign an index to each job in the job-list and is generally only based on a few attributes and simple mathematical operations.

Designing priority dispatching rules requires recognizing the important attributes of the partial schedules needed to create a reasonable scheduling rule. These attributes attempt to grasp key features of the schedule being constructed. Which attributes are most important will necessarily depend on the objectives of the scheduling problem. At-

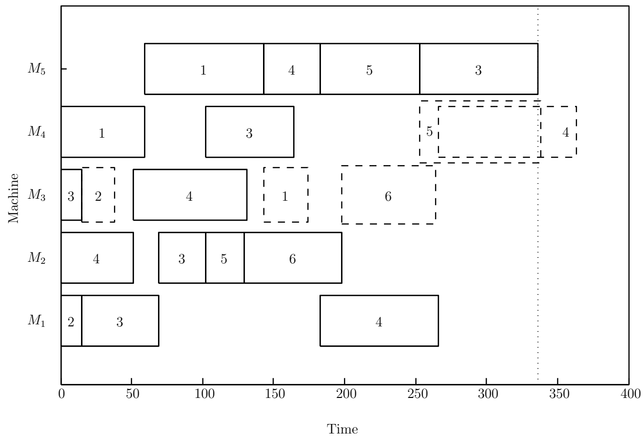

Fig. 1: Gantt chart of a partial JSP schedule after 15 dispatches: Solid and dashed boxes represent $\boldsymbol{\chi}$ and $\mathscr{L}^{(16)}$, respectively. Current $C_{\max}$ denoted as dotted line.

Table 1: Attribute space $\mathscr{A}$ for JSP where job J_j on machine M_a given the resulting temporal schedule after operation (j,a).

$\boldsymbol{\phi}$	Feature description	Mathematical formulation
	job related	
ϕ_1	job processing time	p_{ja}
ϕ_2	job start-time	$x_s(j,a)$
ϕ_3	job end-time	$x_e(j,a)$
ϕ_4	job arrival time	$x_e(j,a-1)$
ϕ_5	time job had to wait	$x_s(j,a) - x_e(j,a-1)$
ϕ_6	total processing time for job	$\sum_{a\in\mathscr{M}} p_{ja}$
ϕ_7	total work remaining for job	$\sum_{a'\in\mathscr{M}\setminus\mathscr{M}_j} p_{ja'}$
ϕ_8	number of assigned operations for job	$\lvert\mathscr{M}_j\rvert$
	machine related	
ϕ_9	when machine is next free	$\max_{j'\in\mathscr{J}_a}\{x_e(j',a)\}$
ϕ_{10}	total processing time for machine	$\sum_{j\in\mathscr{J}} p_{ja}$
ϕ_{11}	total work remaining for machine	$\sum_{j'\in\mathscr{J}\setminus\mathscr{J}_a} p_{j'a}$
ϕ_{12}	number of assigned operations for machine	$\lvert\mathscr{J}_a\rvert$
ϕ_{13}	change in idle time by assignment	$\Delta s(a,j)$
ϕ_{14}	total idle time for machine	$\sum_{j'\in\mathscr{J}_a} s(a,j')$
ϕ_{15}	total idle time for all machines	$\sum_{a'\in\mathscr{M}}\sum_{j'\in\mathscr{J}_{a'}} s(a',j')$
ϕ_{16}	current makespan	$\max_{(j',a')\in\mathscr{J}\times\mathscr{M}_{j'}}\{x_f(j',a')\}$

tributes used in this study applied for each possible operation are given in Table 1, where the set of machines already dispatched for J_j is $\mathscr{M}_j \subset \mathscr{M}$, and similarly, M_a has already had the jobs $\mathscr{J}_a \subset \mathscr{J}$ previously dispatched. The attributes of particular interest were obtained by inspecting the aforementioned SDRs. Attributes ϕ_1-ϕ_8 and ϕ_9-ϕ_{16} are job-related and machine-related, respectively. In fact, [31] note that in the current literature, there is a lack of global perspective in the attribute space, as omitting them won't address the possible negative impact an operation (j,a) might have on other machines at a later time, it is for that reason we consider attributes such as ϕ_{13}-ϕ_{15}, that are slack related and are a means of indicating the current quality of the schedule. All of the attributes, $\boldsymbol{\phi}$, vary throughout the scheduling process, w.r.t. operation belonging to the same time step k, with the exception of ϕ_6 and ϕ_{10} which are static for a given problem instance but varying for each J_j and M_a, respectively.

Priority dispatching rules are attractive since they are relatively easy to implement, perform fast, and find reasonable schedules. In addition, they are relatively easy to interpret, which makes them desirable for the end-user. However, they can also fail unpredictably. A careful combination of dispatching rules has been shown to perform significantly better [18]. These are referred to as *composite priority dispatching rules* (CDR), where the priority ranking is an expression of several dispatching rules. CDRs deal with a greater number of more complicated functions (or features) and are constructed from the schedules attributes. In short, a CDR is a combination of several DRs. For instance let π be a CDR comprised of d DRs, then the index I for $J_j \in \mathscr{L}^{(k)}$ using π is,

$$I_j^{\pi} = \sum_{i=1}^{d} w_i \pi_i(\boldsymbol{\chi}^j) \tag{9}$$

where $w_i > 0$ and $\sum_{i=0}^{d} w_i = 1$ with w_i giving the weight of the influence of π_i (which could be a SDR or another CDR) to π. Note: each π_i is a function of J_j's attributes from the current sequence $\boldsymbol{\chi}$, where $\boldsymbol{\chi}^j$ implies that J_j was the latest dispatch, i.e., the partial schedule given $\chi_k = J_j$.

At each step k, an operation is dispatched which has the highest priority. If there is a tie, some other priority measure is used. Generally the dispatching rules are static during the entire scheduling process. However, ties could also be broken randomly (RND).

While investigating 11 SDRs for JSP, [24] a pool of 33 CDRs was created. This pool strongly outperformed the original CDRs by using multi-contextual functions based on either job waiting time or machine idle time (similar to ϕ_5 and ϕ_{14} in Table 1), i.e., the CDRs are a combination of those two key attributes and then the SDRs. However, there are no combinations of the basic SDRs explored, only those two attributes. Similarly, using priority rules to combine 12 existing DRs from the literature, [43] had 48 CDR combinations which yielded 48 different models to implement and test. It is intuitive to get a boost in performance by introducing new CDRs, since where one DR might be failing, another could be excelling, so combining them together should yield a better CDR. However, these approaches introduce fairly ad-hoc solutions and there is no guarantee the optimal combination of dispatching rules are found.

The composite priority dispatching rule presented in Eq. (9) can be considered as a special case of a the following general linear value function:

$$\pi(\boldsymbol{\chi}^j) = \sum_{i=1}^{d} w_i \phi_i(\boldsymbol{\chi}^j). \tag{10}$$

when $\pi_i(\cdot) = \phi_i(\cdot)$, i.e., a composite function of the features from Table 1. Finally, the job to be dispatched, J_{j^*}, corresponds to the one with the highest value, i.e.,

$$J_{j^*} = \underset{J_j \in \mathscr{L}}{\operatorname{argmax}}\ \pi(\boldsymbol{\chi}^j) \tag{11}$$

Similarly, single priority dispatching rules may be described by this linear model. For instance, let all $w_i = 0$, but with following exceptions: $w_1 = -1$ for SPT, $w_1 = +1$ for LPT, $w_7 = -1$ for LWR and $w_7 = +1$ for MWR. Generally, the weights $\mathbf{w}$ are chosen by the designer or the rule apriori. A more attractive approach would be to learn these weights from problem examples directly. We will now investigate how this may be accomplished.

4 Performance Analysis of Priority Dispatching Rules

In order to create successful dispatching rules, a good starting point is to investigate the properties of optimal solutions and hopefully be able to learn how to mimic the construction of such solutions. For this, we follow optimal solutions, obtained by using a commercial software package [10] and inspect the probability of SDRs being optimal. This serves as an indicator of how hard it is to put our objective up as a machine learning problem. However, we must also take into consideration the end-goal, which is minimising deviation from optimality, ρ, because of it's relationship to stepwise, optimality is not fully understood.

In this section we will describe concerns that must be addressed when learning new priority dispatching rules. At the same time we will describe the experimental set-up used in our study.

4.1 Problem Instances

The class of problem instances used in our studies is the job-shop scheduling problem described in Section 2. Each instance will have different processing times, machine ordering, and dimensions. Each instance will therefore create different challenges for a priority dispatching rule. Dispatching rules learned will be customised for the problems used for their training. For real world application using historical data would be most appropriate. The aim would be to learn a dispatching rule that works well on average for a given distribution of problem instances. To illustrate the performance difference of priority dispatching rules on different problem distributions

within the same class of problems, consider the following three cases. Problem instances for JSP are generated stochastically by fixing the number of jobs and machines to ten. A discrete processing time is sampled independently from a discrete uniform distribution from the interval $I = [u_1, u_2]$, i.e., $\mathbf{p} \sim \mathscr{U}(u_1, u_2)$. The machine order is a random permutation of all of the machines in the job-shop. Two different processing times distributions were explored, namely $\mathscr{P}^{n\times m}_{j.rnd}$ where $I = [1,99]$ and $\mathscr{P}^{n\times m}_{j.rndn}$ where $I = [45,55]$. These instances are referred to as random and random-narrow, respectively. In addition we consider the case where the machine order is fixed and the same for all jobs, i.e. $\boldsymbol{\sigma} = \{1,\ldots,m\}$ where $\mathbf{p} \sim \mathscr{U}(1,99)$. These jobs are denoted by $\mathscr{P}^{n\times m}_{f.rnd}$ and is analogous to $\mathscr{P}^{n\times m}_{j.rnd}$.

The goal is to minimize the makespan, $C_{\max}$. The optimum makespan is denoted $C^{\pi_\star}_{\max}$ (using the expert policy $\pi_\star$), and the makespan obtained from the scheduling policy π under inspection by $C^{\pi}_{\max}$. Since the optimal makespan varies between problem instances the performance measure is the following,

$$\rho = \frac{C^{\pi}_{\max} - C^{\pi_\star}_{\max}}{C^{\pi_\star}_{\max}} \cdot 100\% \tag{12}$$

which indicates the percentage relative deviation from optimality. Note: Eq. (12) measures the discrepancy between predicted value and true outcome, and is commonly referred to as a loss function, which we would like to minimise for π.

Figure 2 depicts the box-plot for Eq. (12) when using the SDRs from Section 3 for all of the problem spaces from Table 2. These box-plots show the difference in performance of the various SDRs. The MWR performs on average the best on the $\mathscr{P}^{n\times m}_{j.rnd}$ and $\mathscr{P}^{n\times m}_{j.rndn}$ problems instances, whereas for $\mathscr{P}^{n\times m}_{f.rnd}$ is is LWR that performs best. It is also interesting to observe that all but the MWR perform statistically worse than random job dispatching on the $\mathscr{P}^{n\times m}_{j.rnd}$ and $\mathscr{P}^{n\times m}_{j.rndn}$ problems instances.

4.2 Reconstructing optimal solutions

When building a complete schedule, $K = n \cdot m$ dispatches must be made sequentially. A job is placed at the earliest available time slot for its next machine, whilst still fulfilling that each machine can handle at most one job at each time, and jobs need to have finished their previous machines according to their machine order. Unfinished jobs, referred to as the job-list denoted $\mathscr{L}$, are dispatched one at a time according to a deterministic scheduling policy (or heuristic), and its pseudo-code is given in Algorithm 1. After each dispatch[1] the schedule's current features (cf. Table 1) are updated based on the half-finished schedule, $\boldsymbol{\chi}$. For each possible post-decision state the temporal features are collected (cf. Line 5) forming the feature set, $\boldsymbol{\Phi}$, based on all N_{train} problem instances available, namely,

$$\boldsymbol{\Phi} := \bigcup_{\{\mathbf{x}_i\}_{i=1}^{N_{\text{train}}}} \left\{ \boldsymbol{\phi}^j \;\middle|\; J_j \in \mathscr{L}^{(k)} \right\}_{k=1}^{K} \subset \mathscr{F} \tag{13}$$

where the feature space $\mathscr{F}$ is described in Table 1, and are based on job- and machine-attributes which are widespread in practice.

It is easy to see that the sequence of task assignments is by no means unique. Inspecting a partial schedule further along in the dispatching process such as in Fig. 1, then let's say J_1 would be dispatched next, and in the next iteration J_2. Now this sequence would yield the same schedule as if J_2 would have been dispatched first and then J_1 in the next iteration, i.e., these are non-conflicting jobs. In this particular

Table 2: Problem space distributions used in experimental studies. Note, problem instances are synthetic and each problem space is i.i.d.

name	size ($n \times m$)	N_{train}	N_{test}	note
$\mathscr{P}^{10\times10}_{j.rnd}$	10×10	300	200	random
$\mathscr{P}^{10\times10}_{j.rndn}$	10×10	300	200	random-narrow
$\mathscr{P}^{10\times10}_{f.rnd}$	10×10	300	200	random

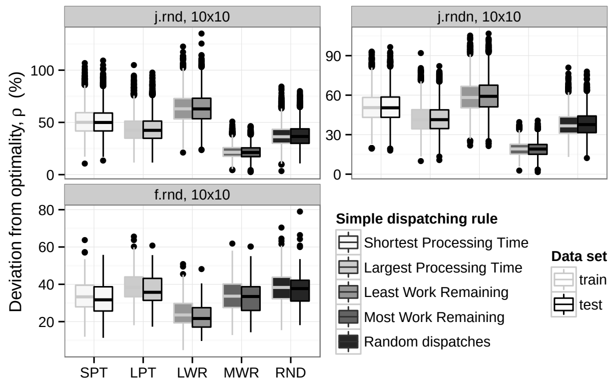

Fig. 2: Box-plot for deviation from optimality, ρ, (%) for SDRs

[1] Dispatch and time step are used interchangeably.

Algorithm 1 Pseudo code for constructing a JSP sequence using a deterministic scheduling policy rule, π, for a fixed construction heuristic, Υ.

procedure SCHEDULEJSP(π, Υ)
  $\boldsymbol{\chi} \leftarrow \emptyset$ ▷ initial current dispatching sequence
  for $k \leftarrow 1$ **to** $K = n \cdot m$ **do** ▷ at each dispatch iteration
    for all $J_j \in \mathscr{L}^{(k)} \subset \mathscr{J}$ **do** ▷ inspect job-list
      $\boldsymbol{\phi}^j \leftarrow \boldsymbol{\phi} \circ \Upsilon\left(\boldsymbol{\chi}^j\right)$ ▷ temporal features for J_j
      $I_j^{\pi} \leftarrow \pi\left(\boldsymbol{\phi}^j\right)$ ▷ priority for J_j
    end for
    $j^* \leftarrow \operatorname{argmax}_{j \in \mathscr{L}^{(k)}} \{I_j^{\pi}\}$ ▷ choose highest priority
    $\chi_k \leftarrow J_{j^*}$ ▷ dispatch j^*
  end for
  return $C_{\max}^{\pi} \leftarrow \Upsilon(\boldsymbol{\chi})$ ▷ makespan and final schedule
end procedure

instance, one cannot infer that choosing J_1 is better and J_2 is worse (or vice versa) since they can both yield the same solution. Furthermore, there may be multiple optimal solutions to the same problem instance. Hence not only is the sequence representation 'flawed' in the sense that slight permutations on the sequence are in fact equivalent w.r.t. the end-result, but very varying permutations on the dispatching sequence (although given the same partial initial sequence) can result in very different complete schedules but can still achieve the same makespan.

The redundancy in building optimal solutions using dispatching rules means that many different dispatches may yield an optimal solution to the problem instance. Let's formalise the probability of optimality (or stepwise classification accuracy) for a given policy π, is defined as,

$$\xi_{\pi}^{\star} := \mathbb{E}_{\pi_\star}\left\{\pi_\star = \pi\right\} \tag{14}$$

that is to say the mean likelihood of our policy π being equivalent to the expert policy $\pi_\star$. The probability that a job chosen by a SDR yields an optimal makespan on a step-by-step basis, i.e., $\xi_{\langle\mathrm{SDR}\rangle}^{\star}$, is depicted in Fig. 3. These probabilities vary quite a bit between the different problem instances distributions studied. From Fig. 3 one observed that $\xi_{\mathrm{MWR}}^{\star}$ has a higher probability than random guessing, in choosing a dispatch which may result in an optimal schedule. This is especially true towards the end of the schedule building process. Similarly, the $\xi_{\mathrm{LWR}}^{\star}$ chooses dispatches resulting in optimal schedules with a higher probability. This would appear to be support the idea that the higher the probability of dispatching jobs that may lead to an optimal schedule, the better the SDRs performance, as illustrated by Fig. 2. However, there is a counter example, $\xi_{\mathrm{SPT}}^{\star}$ has a higher probability than random dispatching of selecting a jobs that may lead to an optimal solution. Nevertheless, the random dispatching performs better than SPT on problem instances $\mathscr{P}_{j.rnd}^{10\times10}$ and $\mathscr{P}_{j.rndn}^{10\times10}$.

Looking at Fig. 6, then $\mathscr{P}_{j.rnd}^{10\times10}$ has a relatively high probability (70% and above) of choosing an optimal job

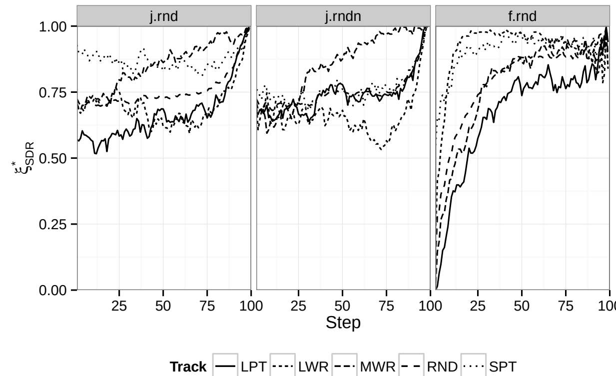

Fig. 3: Probability of SDR being optimal, $\xi_{\langle\mathrm{SDR}\rangle}^{\star}$

at random. However, it is imperative to keep making optimal decisions, because once off the optimal track the consequences are unknown. To demonstrate this Fig. 4 depicts mean worst and best case scenario of the resulting deviation from optimality, ρ, once off the optimal track, defined as follows:

$$\zeta_{\min}^{\star}(k) := \mathbb{E}_{\pi_\star}\left\{\min_{J_j \in \mathscr{L}^{(k)}}(\rho) \;:\; \forall C_{\max}^{\boldsymbol{\chi}^j} \gneq C_{\max}^{\pi_\star}\right\} \tag{15a}$$

$$\zeta_{\max}^{\star}(k) := \mathbb{E}_{\pi_\star}\left\{\max_{J_j \in \mathscr{L}^{(k)}}(\rho) \;:\; \forall C_{\max}^{\boldsymbol{\chi}^j} \gneq C_{\max}^{\pi_\star}\right\} \tag{15b}$$

Note, that this is given that one makes *one* non-optimal dispatch. Generally, there will be more, and then the compound effects of making suboptimal decisions cumulate.

It is interesting to observe that for $\mathscr{P}_{j.rnd}^{10\times10}$ and $\mathscr{P}_{j.rndn}^{10\times10}$ making suboptimal decisions later impacts on the resulting makespan more than doing a mistake early. The opposite seems to be the case for $\mathscr{P}_{f.rnd}^{10\times10}$. In this case it is imperative to make good decisions right from the start. This is due to the major structural differences between JSP and FSP, namely the latter having a homogeneous machine ordering, constricting the solution immensely.

4.3 Blended dispatching rules

A naive approach to create a simple blended dispatching rule (BDR) would be to switch between SDRs at a predetermined time. Observing again Fig. 3, a presumably good BDR for $\mathscr{P}_{j.rnd}^{10\times10}$ would be to start with $\xi_{\mathrm{SPT}}^{\star}$ and then switch over to $\xi_{\mathrm{MWR}}^{\star}$ at around time step $k = 40$, where the SDRs change places in outperforming one another. A box-plot for ρ for the BDR compared with MWR and SPT is depicted in Fig. 5 and its main statistics are reported in Table 3. This simple swap between SDRs does outperform the SPT heuristic, yet doesn't manage to gain the performance edge of MWR. Using SPT downgrades the performance of MWR.

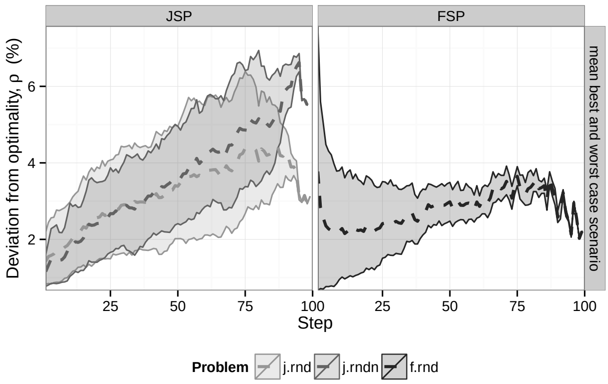

Fig. 4: Mean deviation from optimality, ρ, (%), for best and worst case scenario of making *one* suboptimal dispatch (i.e. $\zeta^\star_{\min}$ and $\zeta^\star_{\max}$), depicted as lower and upper bound, respectively, for $\mathscr{P}^{10\times10}_{j.rnd}$, $\mathscr{P}^{10\times10}_{j.rndn}$ and $\mathscr{P}^{10\times10}_{f.rnd}$. Moreover, mean suboptimal move is given as a dashed line.

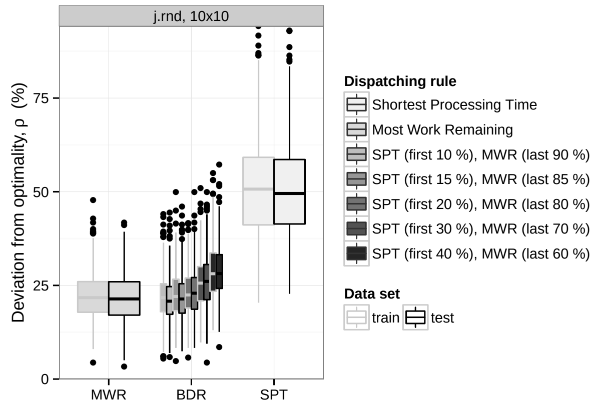

Fig. 5: Box-plot for deviation from optimality, ρ, (%) for BDR where SPT is applied for the first 10%, 15%, 20%, 30% or 40% of the dispatches, followed by MWR

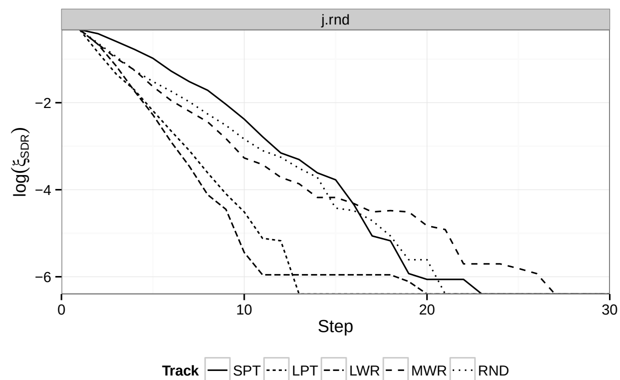

Fig. 6: Log likelihood of SDR being optimal for $\mathscr{P}^{10\times10}_{j.rnd}$, when following its corresponding SDR trajectory, i.e., $\log\left(\xi_{\langle\mathrm{SDR}\rangle}\right)$

A reason for this lack of performance of our proposed BDR is perhaps that by starting out with SPT in the beginning, it sets up the schedules in such a way that it's quite greedy and only takes into consideration jobs with shortest immediate processing times. Now, even though it is possible to find optimal schedules from this scenario, as Fig. 3 shows, the inherent structure that's already taking place, might make it hard to come across by simple methods. Therefore it's by no means guaranteed that by simply swapping over to MWR will handle that situation which applying SPT has already created. Figure 5 does however show, that by applying MWR instead of SPT in the latter stages, does help the schedule to be more compact w.r.t. SPT. However, the fact remains that the schedules have diverged too far from what MWR would have been able to achieve on its own.

In Fig. 3 we inspected the stepwise optimality, given that we were on the optimal trajectory. Since we're bound to make mistakes at some points, it's interesting to see how that stepwise optimality evolves for its intended trajectory, thereby updating Eq. (14) to

$$\xi_\pi := \mathbb{E}_\pi\left\{\pi_\star = \pi\right\} \tag{16}$$

Figure 6 shows the log likelihood for $\xi_{\langle\mathrm{SDR}\rangle}$ using $\mathscr{P}^{10\times10}_{j.rnd}$. There we can see that even though ξ_{SPT} is generally more likely to find optimal dispatches in the initial steps, then shortly after $k = 15$, ξ_{MWR} becomes a contender again. This could explain why our BDR switch at $k = 40$ from Fig. 5 was unsuccessful. However, changing to MWR at $k \leq 20$ is not statically significant from MWR (boost in mean ρ is at most 0.5%). However, after $k > 20$ then the BDR starts diverging from MWR. But as pointed out for Fig. 4, it's not so fatal to make bad moves in the very first dispatches for $\mathscr{P}^{10\times10}_{j.rnd}$, hence little gain with improved classification accuracy in that region.

5 Preference Learning

Section 4.3 demonstrated there is definitely something to be gained by trying out different combinations of DRs, it's just non-trivial how to go about it, and motivates how it's best to go about learning such interaction, which will be addressed in this section.

Learning models considered in this study are based on ordinal regression in which the learning task is formulated as learning preferences. In the case of scheduling, learning which operations are preferred to others. Ordinal regression has been previously presented in [37] and in [14] for JSP, and given here for completeness.

Table 3: Main statistics for $\mathscr{P}_{j.rnd}^{10\times 10}$ deviation from optimality, ρ, using BDR that changes from SDR at a fixed time step k.

SDR #1	SDR #2	k	Set	Min.	1st Qu.	Median	Mean	3rd Qu.	Max.
SPT	–	K	train	20.38	41.15	50.70	51.31	59.18	94.20
SPT	–	K	test	22.75	41.39	49.53	50.52	58.60	93.03
MWR	–	K	train	**4.42**	**17.84**	**21.74**	22.13	26.00	47.78
MWR	–	K	test	**3.37**	**17.07**	21.39	21.65	25.98	**41.80**
SPT	MWR	10	train	5.54	17.98	21.75	**21.99**	**25.43**	**44.02**
SPT	MWR	10	test	5.87	17.29	**20.78**	**21.28**	**24.67**	44.47
SPT	MWR	15	train	4.76	18.24	22.04	22.49	26.65	49.86
SPT	MWR	15	test	7.42	17.60	21.38	21.83	25.45	45.98
SPT	MWR	20	train	5.76	18.98	22.46	23.01	26.97	41.59
SPT	MWR	20	test	8.31	18.64	22.92	23.29	27.10	49.93
SPT	MWR	30	train	9.77	20.89	25.60	25.76	30.01	50.94
SPT	MWR	30	test	4.39	21.20	26.08	26.25	30.58	49.88
SPT	MWR	40	train	13.04	23.42	28.12	28.94	33.67	54.98
SPT	MWR	40	test	8.55	24.20	28.16	28.98	33.20	57.21

The optimum makespan is known for each problem instance. At each time step k, a number of feature pair are created. Let $\boldsymbol{\phi}^o \in \mathbb{R}^d$ denote the post-decision state when dispatching $J_o \in \mathscr{O}^{(k)}$ corresponds to an optimal schedule being built. All post-decisions states corresponding to suboptimal dispatches, $J_s \in \mathscr{S}^{(k)}$, are denoted by $\boldsymbol{\phi}^s \in \mathbb{R}^d$. Note, $\mathscr{O}^{(k)} \cup \mathscr{S}^{(k)} = \mathscr{L}^{(k)}$, and $\mathscr{O}^{(k)} \cap \mathscr{S}^{(k)} = \emptyset$.

The approach taken here is to verify analytically, at each time step, by fixing the current temporal schedule as an initial state, whether it can indeed *somehow* yield an optimal schedule by manipulating the remainder of the sequence. This also takes care of the scenario that having dispatched a job resulting in a different temporal makespan would have resulted in the same final makespan if another optimal dispatching sequence would have been chosen. That is to say the training data generation takes into consideration when there are multiple optimal solutions[2] to the same problem instance.

Let's label features from Eq. (13) that were considered optimal, $\boldsymbol{\psi}^o = \boldsymbol{\phi}^o - \boldsymbol{\phi}^s$, and suboptimal, $\boldsymbol{\psi}^s = \boldsymbol{\phi}^s - \boldsymbol{\phi}^o$ by $y_o = +1$ and $y_s = -1$ respectively. Then, the preference learning problem is specified by a set of preference pairs,

$$\Psi = \left\{ (\boldsymbol{\psi}^o, +1), (\boldsymbol{\psi}^s, -1) \;:\; \forall (J_o, J_s) \in \mathscr{O}^{(k)} \times \mathscr{S}^{(k)} \right\}_{k=1}^{K} \subset \Phi \times Y \tag{17}$$

where $\Phi \subset \mathbb{R}^d$ is the training set of $d = 16$ features (cf. Table 1), $Y = \{+1, -1\}$ is the outcome space from job pairs, $J_o \in \mathscr{O}^{(k)}$ and $J_s \in \mathscr{S}^{(k)}$, for all dispatch steps k.

To summarise, each job is compared against another job of the job-list, $\mathscr{L}^{(k)}$, and if the makespan differs, i.e., $C_{\max}^{(s)} \gneqq C_{\max}^{(o)}$, an optimal/suboptimal pair is created. However, if the makespans are identical the pair is omitted since they give the same optimal makespan. This way, only features from a dispatch resulting in a suboptimal solution is labelled undesirable.

Now let's consider the model space $\mathscr{H} = \{\pi(\cdot) : X \mapsto Y\}$ of mappings from solutions to ranks. Each such function π induces an ordering $\succ$ on the solutions by the following rule,

$$\boldsymbol{\chi}^i \succ \boldsymbol{\chi}^j \quad \Leftrightarrow \quad \pi(\boldsymbol{\chi}^i) > \pi(\boldsymbol{\chi}^j) \tag{18}$$

where the symbol $\succ$ denotes "is preferred to." The function used to induce the preference is defined by a linear function in the feature space,

$$\pi(\boldsymbol{\chi}^j) = \sum_{i=1}^{d} w_i \phi_i(\boldsymbol{\chi}^j) = \left\langle \mathbf{w} \cdot \boldsymbol{\phi}(\boldsymbol{\chi}^j) \right\rangle. \tag{19}$$

Logistic regression learns the optimal parameters $\mathbf{w}^* \in \mathbb{R}^d$. For this study, L2-regularized logistic regression from the LIBLINEAR package [7] without bias is used to learn the preference set Ψ, defined by Section 5. Hence, for each job on the job-list, $J_j \in \mathscr{L}$, let $\boldsymbol{\phi}^j := \boldsymbol{\phi}(\boldsymbol{\chi}^j)$ denote its corresponding post-decision state. Then the job chosen to be dispatched, J_{j^*}, is the one corresponding to the highest preference estimate, i.e., Eq. (11) where $h(\cdot)$ is the classification model obtained by the preference set.

Preliminary experiments for creating step-by-step model was done in [14] where an optimal trajectory was explored, i.e., at each dispatch some (random) optimal task is dispatched, resulting in local linear model for each dispatch; a total of K linear models for solving $n \times m$ JSP. However, the experiments there showed that by fixing the weights to its mean value throughout the dispatching sequence, results remained satisfactory. A more sophisticated way, would be to create a *new* linear model, where the preference set, Ψ, is the union of the preference pairs across the K dispatches, such as described in Section 5. This would amount to a substantial preference set, and for Ψ to be computationally feasible

[2] There can be several optimal solutions available for each problem instance. However, it is deemed sufficient to inspect only one optimal trajectory per problem instance as there are $N_{\text{train}} = 300$ independent instances which gives the training data variety.

to learn, Ψ has to be reduced. For this several ranking strategies were explored in [17], the results there showed that it's sufficient to use partial subsequent rankings, namely, combinations of r_i and r_{i+1} for $i \in \{1,\ldots,n'\}$, are added to the preference set, where $r_1 > r_2 > \ldots > r_{n'}$ ($n' \leq n$) are the rankings of the job-list, in such a manner that in the cases that there are more than one operation with the same ranking, only one of that rank is needed to be compared to the subsequent rank. Moreover, for this study, which deals with 10×10 problem instances, the partial subsequent ranking becomes necessary, as full ranking is computationally infeasible due to its size.

Defining the size of the preference set as $l = |\Psi|$, then if l is too large re-sampling may be needed to be done in order for the ordinal regression to be computationally feasible.

The training data from [14] was created from optimal solutions of randomly generated problem instances, i.e., traditional *passive* imitation learning (IL). As JSP is a sequential decision making process, errors are bound to emerge. Due to compound effect of making suboptimal dispatches, the model leads the schedule astray from learned state-spaces, resulting in the new input being foreign to the learned model.

Alternatively, training data could be generated using suboptimal solution trajectories as well, as was done in [17], where the training data also incorporated following the trajectories obtained by applying successful SDRs from the literature. The reasoning behind it was that they would be beneficial for learning, as they might help the model to escape from local minima once off the coveted optimal path. By simply adding training data obtained by following the trajectories of well-known SDRs, their aggregated training set yielded better models with lower deviation from optimality, ρ, defined by Eq. (12).

Inspired by the work of [34,35], the methodology of generating training data will now be such that it will iteratively improve upon the model, such that the state-spaces learned will be representative of the state-spaces the eventual model would likely encounter, known as DAgger for *active* imitation learning. Thereby, eliminating the ad-hoc nature of choosing trajectories to learn, by rather letting the model lead its own way in a self-perpetuating manner until it converges.

Furthermore, in order to boost training accuracy, two strategies were explored:

Boost.1 increasing number of preferences used in training (i.e. varying $l_{\max} \leq |\Psi|$),

Boost.2 introducing more problem instances (denoted EXT in experimental setting).

Note, that in preliminary experiments for Boost.1 showed no statistical significance in boost of performance. Hence, the default set-up will be, $l_{\max} = 5 \cdot 10^5$, which is roughly the amount of features encountered from one pass of sampling a K-stepped trajectory using a fixed policy π for the default $N_{\text{train}} = 300$.

Another way to adjust training accuracy is to give different weight to various time steps. To address this problem, two different stepwise sampling biases (or data balancing techniques) will be considered:

Bias.1 **(equal)** where each time step has equal probability, same baseline as was used in [16,17].

Bias.2 **(adjdbl2nd)** where each time step is adjusted to the number of preference pairs for that particular step (i.e. each step has equal probability irrespective of quantity of encountered features). This is done with re-sampling. In addition, there is superimposed twice as much likelihood of choosing pairs from the latter half of the dispatching process. Then the final sampled data set is divided as follows: $|\{\Psi(k)\}_{k=0}^{\frac{K}{2}-1}| \approx \frac{1}{3} l_{\max}$ and $|\{\Psi(k)\}_{k=\frac{K}{2}}^{K-1}| \approx \frac{2}{3} l_{\max}$.

Note, as the following sections require repeated collection of training data, and since its labelling is a very time intensive task the remainder of the paper will solely be focusing on $\mathscr{P}_{j.rnd}^{10\times 10}$.

6 Passive Imitation Learning

Using the terms from game-theory used in [4], then our problem is a basic version of the sequential prediction problem where the predictor (or forecaster), π, observes each element of a sequence $\boldsymbol{\chi}$ of jobs, where at each time step $k \in \{1,\ldots,K\}$, before the k-th job of the sequence is revealed, the predictor guesses its value χ_k on the basis of the previous $k-1$ observations.

6.1 Prediction with Expert Advice

Let's assume we know the expert policy $\pi^\star$, which we can query what is the optimal choice of $\chi_k = j^*$ at any given time step k. Now we can use Eq. (11) to back-propagate the relationship between post-decision states and $\hat{\pi}$ with preference learning via our collected feature set, denoted Φ^{OPT}, i.e., we collect the features set corresponding following optimal tasks J_{j^*} from $\pi^\star$ in Algorithm 1. This baseline trajectory sampling for adding features to the feature set is a pure strategy where at each dispatch, an optimal task was originally introduced in [14].

By querying the expert policy, $\pi_\star$, the ranking of the job-list, $\mathscr{L}$, is determined such that,

$$r_1 \succ r_2 \succ \cdots \succ r_{n'} \quad (n' \leq n) \tag{20}$$

Algorithm 2 Pseudo code for choosing job J_{j^*} following a perturbed leader.

Require: Ranking $r_1 \succ r_2 \succ \cdots > r_{n'}$ $(n' \leq n)$ of the job-list, $\mathscr{L}$ ▷ query $\pi_\star$
procedure PERTURBEDLEADER($\mathscr{L}, \pi_\star$)
  $\varepsilon \leftarrow 0.1$ ▷ likelihood factor
  $p \leftarrow \mathscr{U}(0,1) \in [0,1]$ ▷ uniform probability
  $\mathscr{O} \leftarrow \{ j \in \mathscr{L} \mid r_j = r_1 \}$ ▷ optimal job-list
  $\mathscr{S} \leftarrow \{ j \in \mathscr{L} \mid r_j > r_1 \}$ ▷ sub-optimal job-list
  if $p < \varepsilon$ **and** $n' > 1$ **then**
    return $j^* \in \{ j \in \mathscr{S} \mid r_j = r_2 \}$ ▷ any second best job
  else
    return $j^* \in \mathscr{O}$ ▷ any optimal job
  end if
end procedure

implies r_1 is preferable to r_2, and r_2 is preferable to r_3, etc. In our study, we know $r \propto C_{\max}^{\pi_\star}$, hence the optimal job-list is the following,

$$\mathscr{O} = \left\{ r_i \;\middle|\; r_i \propto \min_{J_j \in \mathscr{L}} C_{\max}^{\pi_\star(\boldsymbol{\chi}^j)} \right\} \tag{21}$$

found by solving the current partial schedule to optimality using a commercial software package such as [10].

When $|\mathscr{O}^{(k)}| > 1$, there can be several trajectories worth exploring. However, only one is chosen at random. This is deemed sufficient as the number of problem instances, N_{train}, is relatively large.

6.2 Follow the Perturbed Leader

By allowing a predictor to randomise it's possible to achieve improved performance [4, 11], which is the inspiration for our new strategy, where we follow the Perturbed Leader, denoted OPTε. Its pseudo code is given in Algorithm 2 and describes how the expert policy (i.e. optimal trajectory) from Section 6.1 is subtly "perturbed" with $\varepsilon = 10\%$ likelihood, by choosing a job corresponding to the second best $C_{\max}$ instead of a optimal one with some small probability.

6.3 Results

Results for $\mathscr{P}_{j.rnd}^{10\times10}$ box-plot of deviation from optimality, ρ, is given in Fig. 7 and main statistics are reported in Table 4. To address Boost.2, the extended training set was simply obtained by iterating over more examples, $N_{\text{train, EXT}}^{\text{OPT}} = 1000$. However, we see that the increased number of varied features dissuades the preference models to achieving a good performance w.r.t. ρ. It's preferable to use the default $N_{\text{train}}^{\text{OPT}} = 300$ and allowing slightly perturbing the optimal trajectory, as done for $\Phi^{\text{OPT}\varepsilon}$. Unfortunately, all this overhead has not managed to surpass MWR in performance. The closest to MWR, is using Bias.2 instead of Bias.1, with

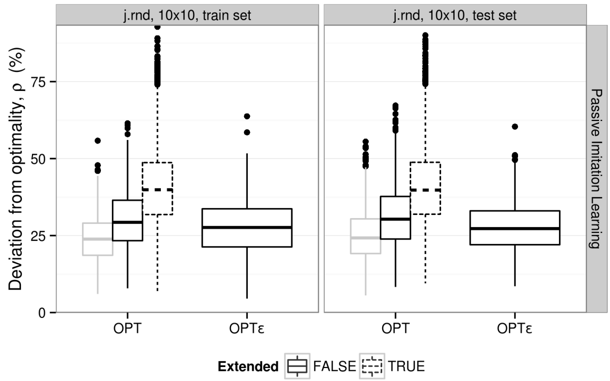

Fig. 7: Box plot for $\mathscr{P}_{j.rnd}^{10\times10}$ deviation from optimality, ρ, using either expert policy and following perturbed leader.

a $\Delta\rho \approx -6.23\%$ boost in mean performance. This is likely due to the fact that if equal probability is used for stepwise sampling, then there are hardly any emphasis given to the final dispatches as there a relatively few (compared to previous steps) preference pairs belonging to those final stages. Revisiting Fig. 4, then the band for $\{\zeta_{\min}^\star, \zeta_{\max}^\star\}$ is quite tight, as the problem is immensely constricted and few operations to choose from. However, the empirical evidence from using Bias.2 shows, is that it is imperative to make right decisions at the very end.

From [14] we know that expert policy is a promising starting point. However, that was for 6×5 dimensionality (i.e. $K = 30$), which is a much simpler problem space. Notice that in Fig. 6 there was virtually no chance for $\xi_\pi(k)$ of choosing a job resulting in optimal makespan after step $k = 28$.

Since job-shop is a sequential prediction problem, all future observations are dependent on previous operations. Therefore, learning sampled states that correspond only to optimal or near-optimal schedules isn't of much use when the preference model has diverged too far. We know from Section 4.3, that good classification accuracy based on $\xi_\pi^\star$ doesn't necessarily mean a low mean deviation from optimality, ρ. This is due to the learner's predictions affects future input observations during its execution, which violates the crucial i.i.d. assumptions of the learning approach, and ignoring this interaction leads to poor performance. In fact, [34] proves, that assuming the preference model has a training error of ε, then the total compound error (for all K dispatches) the classifier induces itself grows quadratically, $O\left(\varepsilon K^2\right)$, for the entire schedule, rather than having linear loss, $O(\varepsilon K)$, if it were i.i.d.

7 Active Imitation Learning

To amend performance from Φ^{OPT}-based models, suboptimal state-spaces were explored in [17] by inspecting the features from successful SDRs, $\Phi^{\langle\text{SDR}\rangle}$, by passively observing a full execution of following the task chosen by the corresponding SDR. This required some trial-and-error as the experiments showed that features obtained by SDR trajectories were not equally useful for learning.

To automate this process, inspiration from *active* imitation learning presented in [35] is sought, called *Dataset Aggregation* (DAgger) method, which addresses a no-regret algorithm in an on-line learning setting. The novel meta-algorithm for IL learns a deterministic policy guaranteed to perform well under its induced distribution of states. The method is closely related to Follow-the-leader (cf. Section 6.2), however, with a more sophisticated leverage to the expert policy. In short, it entails the model π_i that queries an expert policy (same as in Section 6.1), $\pi_\star$, its trying to mimic, but also ensuring the learned model updates itself in an iterative fashion, until it converges. The benefit of this approach is that the states that are likely to occur in practice are also investigated and as such used to dissuade the model from making poor choices. In fact, the method queries the expert about the desired action at individual post-decision states which are both based on past queries, and the learner's interaction with the *current* environment.

DAgger has been proven successful on a variety of benchmarks, such as: the video games Super Tux Kart and Super Mario Bros. or handwriting recognition – in all cases greatly improving traditional supervised imitation learning approaches [35], and real-world applications, e.g. autonomous navigation for large unmanned aerial vehicles [36].

Algorithm 3 Pseudo code for choosing job J_{j^*} using imitation learning (dependent on iteration i) to collect training set $\Phi^{\text{IL}i}$; either by following optimal trajectory, $\pi_\star$, or preference model from previous iterations, $\hat{\pi}_{i-1}$.

Require: $i \geq 0$
Require: Ranking $r_1 \succ r_2 \succ \cdots > r_{n'}$ $(n' \leq n)$ of the job-list, $\mathcal{L}$ ▷ query $\pi_\star$
procedure ACTIVEIL$(i, \hat{\pi}_{i-1}, \pi_\star)$
  $p \leftarrow \mathcal{U}(0,1) \in [0,1]$ ▷ uniform probability
  if $i > 0$ **then** (unsupervised)
    $\beta_i \leftarrow 0$ ▷ always apply imitation
  else (fixed supervision)
    $\beta_i \leftarrow 1$ ▷ always follow expert policy (i.e. optimal)
  end if
  if $p > \beta_i$ **then**
    return $j^* \leftarrow \text{argmax}_{j \in \mathcal{L}} \{I_j^{\hat{\pi}_{i-1}}\}$ ▷ best job based on $\hat{\pi}_{i-1}$, cf. Algorithm 1
  else
    $\mathcal{O} \leftarrow \{ j \in \mathcal{L} \mid r_j = r_1 \}$ ▷ optimal job-list
    return $j^* \in \mathcal{O}$ ▷ any optimal job
  end if
end procedure

Algorithm 4 DAgger: Dataset Aggregation for JSP

Require: $T \geq 1$
procedure DAGGER$(\pi_\star, \Phi^{\text{OPT}}, T)$
  $\Phi^{\text{IL}0} \leftarrow \Phi^{\text{OPT}}$ ▷ initialize dataset
  $\hat{\pi}_0 \leftarrow$ TRAIN$(\Phi^{\text{IL}0})$ ▷ initial model, equivalent to Section 6.1
  for $i \leftarrow 1$ **to** T **do** ▷ at each imitation learning iteration
    Let $\pi_i = \beta_i \pi_\star + (1-\beta_i)\hat{\pi}_{i-1}$ ▷ Eq. (22)
    Sample K-step trajectories using π_i ▷ cf. Algorithm 3: ACTIVEIL$(i, \hat{\pi}_{i-1}, \pi_\star)$
    $\Phi^{\text{IL}i} = \{(s, \pi_\star(s))\}$ ▷ visited states by π_i and actions given by expert
    $\Phi^{\text{DA}i} \leftarrow \Phi^{\text{DA}i-1} \cup \Phi^{\text{IL}i}$ ▷ aggregate datasets, cf. Eq. (23)
    $\hat{\pi}_{i+1} \leftarrow$ TRAIN$(\Phi^{\text{DA}i})$ ▷ preference model from Eq. (10)
  end for
  return best $\hat{\pi}_i$ on validation ▷ best preference model
end procedure

7.1 DAgger

The policy of imitation learning at iteration $i > 0$ is a mixed strategy given as follows,

$$\pi_i = \beta_i \pi_\star + (1-\beta_i)\hat{\pi}_{i-1} \tag{22}$$

where $\pi_\star$ is the expert policy and $\hat{\pi}_{i-1}$ is the learned model from the previous iteration. Note, for the initial iteration, $i = 0$, a pure strategy of $\pi_\star$ is followed. Hence, $\hat{\pi}_0$ corresponds to the preference model from Section 6.1 (i.e. $\Phi^{\text{IL}0} = \Phi^{\text{OPT}}$).

Equation (22) shows that β controls the probability distribution of querying the expert policy $\pi_\star$ instead of the previous imitation model, $\hat{\pi}_{i-1}$. The only requirement for $\{\beta_i\}_i^\infty$ according to [35] is that $\lim_{T\to\infty} \frac{1}{T}\sum_{i=0}^{T} \beta_i = 0$ to guarantee finding a policy $\hat{\pi}_i$ that achieves ε surrogate loss under its own state distribution limit.

Algorithm 3 explains the pseudo code for how to collect partial training set, $\Phi^{\text{IL}i}$ for i-th iteration of active imitation learning. Subsequently, the resulting preference model, $\hat{\pi}_i$, learns on the aggregated datasets from all previous iterations, namely,

$$\Phi^{\text{DA}i} = \bigcup_{i'=0}^{i} \Phi^{\text{IL}i'} \tag{23}$$

and its update procedure is detailed in Algorithm 4.

7.2 Results

Due to time constraints, only $T = 3$ iterations will be inspected. In addition, preliminary experiments showed that DAgger for job-shop is not sensitive to choice of β_i in Eq. (22). Hence, a simple parameter-free version of the DAgger algorithm, which often performs best in practice [35], is chosen. Namely, the mixed strategy for $\{\beta_i\}_{i=0}^{T}$ is

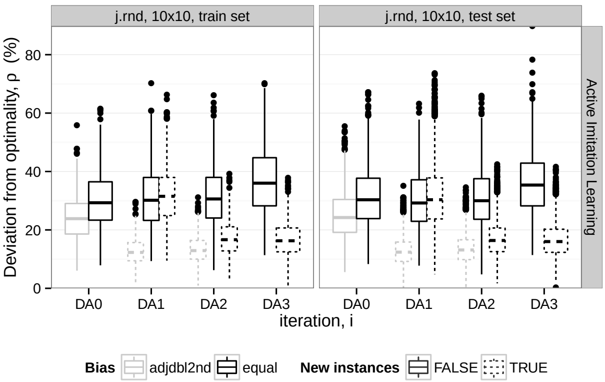

Fig. 8: Box plot for $\mathscr{P}^{10\times10}_{j.rnd}$ deviation from optimality, ρ, using DAgger for JSP

unsupervised with $\beta_i = I(i=0)$, where I is the indicator function.[3]

Regarding Boost.2 strategy, we know from Section 6, that adding new problem instances didn't boost performance for the expert policy (which is equivalent for the initial iteration of DAgger). Hence, for active IL, the extended set is now consisted of each iteration encountering N_{train} *new* problem instances. For a grand total of

$$N^{\text{DA}i}_{\text{train, EXT}} = N_{\text{train}} \cdot (i+1) \tag{24}$$

problem instances explored for the aggregated extended training set used for the learning model at iteration i. This way, we use the extended training data sparingly, as labelling for each problem instances is computationally intensive. As a result, the computational budget for DAgger is same regardless whether there are new problem instances used or not, i.e., $|\Phi^{\text{DA2}}| \approx |\Phi^{\text{DA}i}_{\text{EXT}}|$.

Results for $\mathscr{P}^{10\times10}_{j.rnd}$ box-plot of deviation from optimality, ρ, is given in Fig. 8 and main statistics is reported in Table 4. As we can see DAgger is not fruitful when the same problem instances are continually used. This is due to the fact that there is not enough variance between $\Phi^{\text{IL}i}$, hence the aggregated feature set $\Phi^{\text{DA}i}$ is only slightly perturbed with each iterations. Which from Section 6.3 we saw wasn't a very successful modification for the expert policy. Although, it's noted that by introducing sub-optimal state spaces the preference model is not as drastically bad as the extended optimal policy, even though $|\Phi^{\text{DA}i}| \approx |\Phi^{\text{OPT}}_{\text{EXT}}|$. However, when using new problem instances at each iterations, the feature set becomes varied enough that situations arise that can be learned to achieve a better represented classification problem which yields a lower mean deviation from optimality, ρ.

[3] $\beta_0 = 1$ and $\beta_i = 0, \forall i > 0$.

8 Summary of Imitation Learning

A summary of $\mathscr{P}^{10\times10}_{j.rnd}$ best passive and active imitation learning models w.r.t. deviation from optimality, ρ, from Sections 6.3 and 7.2, respectively, are illustrated in Fig. 9, and main statistics are given in Table 4. To summarise, the following trajectories are used: *i)* expert policy, trained on Φ^{OPT}; *ii)* perturbed leader, trained on $\Phi^{\text{OPT}\varepsilon}$, and *iii)* imitation learning, trained on $\Phi^{\text{DA}i}_{\text{EXT}}$ for iterations $i = \{1,..,3\}$ using extended training set. As a reference, the single priority dispatching rule MWR is shown on at the edges of Fig. 9.

At first we see that the perturbed leader ever so-slightly improves the mean for ρ, rather than using the baseline expert policy. However, active imitation learning is by far the best improvement. With each iteration of DAgger, the models improve upon the previous one with each iteration: *i)* for Bias.1 with Boost.2 then $i=1$ starts with increasing $\Delta\rho \approx +1.39\%$. However, after that first iteration there is a performance boost of $\Delta\rho \approx -15.11\%$ after $i=2$ and $\Delta\rho \approx -0.19\%$ for the final iteration $i=3$, and *ii)* on the other hand when using Bias.2, only one iteration is needed, as $\Delta\rho \approx -11.68$ for $i=1$, and after that it stagnates with $\Delta\rho \approx +0.55\%$ for $i=2$ (therefore $i=3$ was not run). In both cases, DAgger outperforms MWR: *i)* after $i=3$ iterations by $\Delta\rho \approx -5.31\%$ for Bias.1 with Boost.2, and *ii)* after $i=1$ iteration by $\Delta\rho \approx -9.31\%$ for Bias.2. Note, for Bias.1 without Boost.2, then DAgger is unsuccessful, and the aggregated data set downgrades the performance of the previous iterations, making it best to learn solely on the initial expert policy for that model configuration.

Regarding Boost.2, then it's not successful for the expert policy, as ρ increased approximately 10%. This could most likely be counter-acted by increasing $l_{\max}$ to reflect the 700 additional examples. What is interesting though, is that Boost.2 is well suited for active imitation learning, using the same $l_{\max}$ as before. Note, the amount of problems used for $N^{\text{OPT}}_{\text{train, EXT}}$ is equivalent to $T = 2\frac{1}{3}$ iterations of extended DAgger. The *new* varied data gives the aggregated feature set more information of what is important to learn in subsequent iterations, as those new states are more likely to be encountered 'in practice' rather than 'in theory.' Not only does the active imitation learning converge faster, it also consistently improves with each iterations.

9 Discussion and conclusions

Current literature still hold single priority dispatching rules in high regard, as they are simple to implement and quite efficient. However, they are generally taken for granted as there is clear lack of investigation of *how* these dispatching rules actually work, and what makes them so successful (or in some cases unsuccessful)? For instance, of the four SDRs this study focuses on, why does MWR outperform

Table 4: Main statistics for $\mathscr{P}_{j.rnd}^{10\times10}$ deviation from optimality, ρ, using either expert policy, imitation learning or following perturbed leader strategies.

π^a	T^b	Bias	Set	N_{train}	Min.	1st Qu.	Median	Mean	3rd Qu.	Max.
OPT	0	adjdbl2nd	train	300	6.05	18.60	23.85	24.50	29.04	55.81
OPT	0	adjdbl2nd	test	300	5.56	19.16	24.24	25.19	30.42	55.52
OPT	0	equal	train	300	7.87	23.34	29.30	30.73	36.47	61.45
OPT	0	equal	test	300	8.31	23.88	30.32	31.46	37.70	67.24
DA1	1	adjdbl2nd	train	600	2.08	**9.44**	**12.30**	**12.82**	**15.67**	**29.63**
DA1	1	adjdbl2nd	test	300	**0.00**	**9.22**	**12.39**	**12.73**	**15.85**	35.17
DA1	1	equal	train	600	9.47	24.92	31.51	32.12	37.96	66.29
DA1	1	equal	test	300	4.77	23.77	30.34	31.40	37.81	73.73
DA2	2	adjdbl2nd	train	900	**0.93**	10.01	12.91	13.37	16.40	31.19
DA2	2	adjdbl2nd	test	300	0.39	9.84	13.13	13.44	16.62	**34.57**
DA2	2	equal	train	900	2.36	12.82	16.65	17.01	21.06	39.25
DA2	2	equal	test	300	1.72	12.57	16.38	16.89	20.66	42.44
DA3	3	equal	train	1200	0.98	12.50	16.28	16.82	20.67	37.93
DA3	3	equal	test	300	0.26	12.32	16.01	16.52	20.22	41.62
OPTε	0	equal	train	300	4.52	21.31	27.63	28.04	33.69	63.74
OPTε	0	equal	test	300	8.54	22.03	27.26	27.94	33.02	60.38

[a] For DAgger, then $T = 0$ is conventional expert policy (i.e. DA0 = OPT).
[b] If $T = 0$ then *passive* imitation learning. Otherwise, for $T > 0$ it is considered *active* imitation learning.

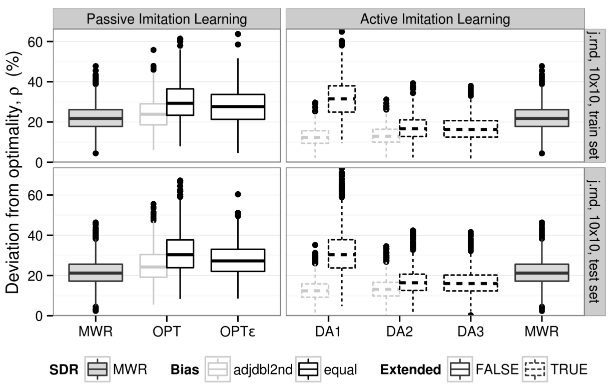

Fig. 9: Box plot for $\mathscr{P}_{j.rnd}^{10\times10}$ deviation from optimality, ρ, using either expert policy, DAgger or following perturbed leader strategies. MWR shown for reference.

so significantly for job-shop yet completely fail for flow-shop? MWR seems to be able to adapt to varying distributions of processing times, however, manipulating the machine ordering causes MWR to break down. By inspecting optimal schedules, and meticulously researching what's going on, every step of the way of the dispatching sequence, some light is shed where these SDRs vary w.r.t. the problem space at hand. Once these simple rules are understood, then it's feasible to extrapolate the knowledge gained and create new composite priority dispatching rules that are likely to be successful.

Creating new dispatching rules is by no means trivial. For job-shop there is the hidden interaction between processing times and machine ordering that's hard to measure. Due to this artefact, feature selection is of paramount importance, and then it becomes the case of not having too many features, as they are likely to hinder generalisation due to over-fitting in training. However, the features need to be explanatory enough to maintain predictive ability.

When training the learning model, it's not sufficient to only optimise w.r.t. highest mean validation accuracy. As there is a trade-off between making the over-all best decisions versus making the right decision on crucial time points in the scheduling process, as Fig. 4 clearly illustrated. This also opens of the question of how should validation accuracy be measured? Since the model is based on learning preferences, both based on optimal versus suboptimal, and then varying degrees of sub-optimality. As we are only looking at the ranks in a black and white fashion, such that the makespans need to be strictly greater to belong to a higher rank, then it can be argued that some ranks should be grouped together if their makespans are sufficiently close. This would simplify the training set, making it (presumably) less of contradictions and more appropriate for linear learning. Or simply the validation accuracy could be weighted w.r.t. the difference in makespan. During the dispatching process, there are some pivotal times which need to be especially taken care off. Figure 4 showed how making suboptimal decisions were more of a factor during the later stages, whereas for flow-shop the case was exact opposite. Experiments in Section 6.3 clearly showed that following the expert policy is not without its faults. There are many obstacles to consider in order to improve model configurations. For instance, the before training the learned model, the preference set Ψ needs to be re-sampled to size $l_{\max}$. As the effects of making suboptimal choices varies as a function of times steps, the stepwise bias should rather be done proportional to the mean cumulative loss to a particular time step. As the experimental studies in Sections 6.3, 7.2 and 8 showed, instead of equal probability (i.e. Bias.1) it was much more fruitful to adjust the set to its number of preference and doubling the emphasis on the second half (i.e. Bias.2). However, there are many other stepwise sampling strategies based on our analysis that could have been chosen instead, as here only a simplification of the trend from Fig. 4 was chosen.

Despite the abundance of information gathered by following an optimal trajectory, the knowledge obtained is not enough by itself. Since the learning model isn't perfect, it is bound to make a mistake eventually. When it does, the model is in uncharted territory as there is not certainty the samples already collected are able to explain the current situation. For this we propose investigating features from suboptimal trajectories as well, since the future observations depend on previous predictions. A straight forward approach would be to inspect the trajectories of promising SDRs or CDRs. However, more information is gained when applying active imitation learning inspired by work of [34,35], such that the learned policy following an optimal trajectory is used to collect training data, and the learned model is updated. This can be done over several iterations, with the benefit being, that the states that are likely to occur in practice are investigated, and as such used to dissuade the model from making poor choices. Alas, this comes at great computational cost due to the substantial amounts of states that need to be optimised for their correct labelling. Making it only practical for job-shop of a considerable lower dimension.

Maximum Mean Discrepancy (MMD) imitation learning by [21] is an iterative algorithm similar to DAgger. However, the expert policy is only queried when needed in order to reduce computational cost. This occurs when a metric of a new state is sufficiently large enough from a previously queried states (to ensure diversity of learned optimal states). Moreover, in DAgger all data samples are equally important, irrespective of its iteration, which can require great number of iterations to learn how to recover from the mistakes of earlier policies. To address the naivety of the data aggregation, MMD suggests only aggregating a new data point if it is sufficiently different to previously gathered states, *and* if the current policy has made a mistake. Additionally, there are multiple policies, each specializing in a particular region of the state space where previous policies made mistakes. Although MMD has better empirical performance (based on robot applications), it requires defining metrics, which in the case of job-shop is non-trivial (cf. [15]), and fine-tuning thresholds etc., whereas DAgger can be straightforwardly implemented, parameter-free and obtains competitive results, although with some computational overhead due to excess expert queries.

Main drawback of DAgger is that it quite aggressively quires the expert, making it impractical for some problems, especially if it involves human experts. To confront that, [19] introduce Reduction-based Active Imitation Learning (RAIL), which involves a dynamic approach similar to DAgger, but more emphasis is used to minimise the expert's labelling effort. In fact, it's possible to circumvent querying the expert altogether and still have reasonable performance. By applying Locally Optimal Learning to Search (LOLS) [5] it is possible to use imitation learning (similar to DAgger framework) when the reference policy is poor (i.e. $\pi_\star$ in Eq. (22) is suboptimal), although it's noted that the quality (w.r.t near-optimality) of reference policy is in accordance to its performance, as is to be expected.

Although this study has been structured around the job-shop scheduling problem, it is easily extended to other types of deterministic optimisation problems that involve sequential decision making. The framework presented here collects snap-shots of the state space by following an optimal trajectory, and verifying the resulting optimal makespan from each possible state. From which the stepwise optimality of individual features can be inspected, which could for instance justify omittance in feature selection. Moreover, by looking at the best and worst case scenario of suboptimal dispatches, it is possible to pinpoint vulnerable times in the scheduling process.

References

1. Andresen, M., Engelhardt, F., Werner, F.: LiSA - A Library of Scheduling Algorithms (version 3.0) [software] (2010). URL `http://www.math.ovgu.de/Lisa.html`
2. Burke, E., Petrovic, S., Qu, R.: Case-based heuristic selection for timetabling problems. Journal of Scheduling **9**, 115–132 (2006)
3. Burke, E.K., Gendreau, M., Hyde, M., Kendall, G., Ochoa, G., Ozcan, E., Qu, R.: Hyper-heuristics: a survey of the state of the art. Journal of the Operational Research Society **64**(12), 1695–1724 (2013)
4. Cesa-Bianchi, N., Lugosi, G.: Prediction, Learning, and Games, chap. 4. Cambridge University Press, New York, NY, USA (2006)
5. Chang, K., Krishnamurthy, A., Agarwal, A., III, H.D., Langford, J.: Learning to search better than your teacher. In: Proceedings of The 32nd International Conference on Machine Learning, pp. 2058–2066 (2015)
6. Chen, T., Rajendran, C., Wu, C.W.: Advanced dispatching rules for large-scale manufacturing systems. The International Journal of Advanced Manufacturing Technology (2013)
7. Fan, R.E., Chang, K.W., Hsieh, C.J., Wang, X.R., Lin, C.J.: LIBLINEAR: A library for large linear classification. Journal of Machine Learning Research **9**, 1871–1874 (2008)
8. Garey, M.R., Johnson, D.S., Sethi, R.: The complexity of flowshop and jobshop scheduling. Mathematics of Operations Research **1**(2), 117–129 (1976)
9. Gomes, C.P., Selman, B.: Algorithm portfolios. Artificial Intelligence **126**(1-2), 43–62 (2001)
10. Gurobi Optimization, Inc.: Gurobi optimization (version 6.0.0) [software] (2014). URL `http://www.gurobi.com/`

11. Hannan, J.: Approximation to bayes risk in repeated play. Contributions to the Theory of Games **3**, 97–139 (1957)
12. Haupt, R.: A survey of priority rule-based scheduling. OR Spectrum **11**, 3–16 (1989)
13. Hildebrandt, T., Heger, J., Scholz-Reiter, B.: Towards improved dispatching rules for complex shop floor scenarios: a genetic programming approach. GECCO '10: Proceedings of the 12th annual conference on Genetic and evolutionary computation pp. 257–264 (2010)
14. Ingimundardottir, H., Runarsson, T.P.: Supervised learning linear priority dispatch rules for job-shop scheduling. In: C.A. Coello (ed.) Learning and Intelligent Optimization, *Lecture Notes in Computer Science*, vol. 6683, pp. 263–277. Springer Berlin Heidelberg (2011)
15. Ingimundardottir, H., Runarsson, T.P.: Determining the characteristic of difficult job shop scheduling instances for a heuristic solution method. In: Y. Hamadi, M. Schoenauer (eds.) Learning and Intelligent Optimization, Lecture Notes in Computer Science, pp. 408–412. Springer Berlin Heidelberg (2012)
16. Ingimundardottir, H., Runarsson, T.P.: Evolutionary learning of weighted linear composite dispatching rules for scheduling. In: International Conference on Evolutionary Computation Theory and Applications (ECTA). SCITEPRESS (2014)
17. Ingimundardttir, H., Philip Rnarsson, T.: Generating training data for learning linear composite dispatching rules for scheduling. In: C. Dhaenens, L. Jourdan, M.E. Marmion (eds.) Learning and Intelligent Optimization, *Lecture Notes in Computer Science*, vol. 8994, pp. 236–248. Springer International Publishing (2015)
18. Jayamohan, M., Rajendran, C.: Development and analysis of cost-based dispatching rules for job shop scheduling. European Journal of Operational Research **157**(2), 307–321 (2004)
19. Judah, K., Fern, A., Dietterich, T.G.: Active imitation learning via reduction to I.I.D. active learning. CoRR **abs/1210.4876** (2012)
20. Kalyanakrishnan, S., Stone, P.: Characterizing reinforcement learning methods through parameterized learning problems. Machine Learning **84**(1-2), 205–247 (2011)
21. Kim, B., Pineau, J.: Maximum mean discrepancy imitation learning. In: Robotics: Science and Systems (2013)
22. Korytkowski, P., Rymaszewski, S., Wiśniewski, T.: Ant colony optimization for job shop scheduling using multi-attribute dispatching rules. The International Journal of Advanced Manufacturing Technology (2013)
23. Li, X., Olafsson, S.: Discovering dispatching rules using data mining. Journal of Scheduling **8**, 515–527 (2005)
24. Lu, M.S., Romanowski, R.: Multicontextual dispatching rules for job shops with dynamic job arrival. The International Journal of Advanced Manufacturing Technology (2013)
25. Malik, A.M., Russell, T., Chase, M., Beek, P.: Learning heuristics for basic block instruction scheduling. Journal of Heuristics **14**(6), 549–569 (2008)
26. Meeran, S., Morshed, M.: A hybrid genetic tabu search algorithm for solving job shop scheduling problems: a case study. Journal of intelligent manufacturing **23**(4), 1063–1078 (2012)
27. Mönch, L., Fowler, J.W., Mason, S.J.: Production Planning and Control for Semiconductor Wafer Fabrication Facilities, *Operations Research/Computer Science Interfaces Series*, vol. 52, chap. 4. Springer, New York (2013)
28. Nguyen, S., Zhang, M., Johnston, M., Tan, K.C.: Learning iterative dispatching rules for job shop scheduling with genetic programming. The International Journal of Advanced Manufacturing Technology (2013)
29. Olafsson, S., Li, X.: Learning effective new single machine dispatching rules from optimal scheduling data. International Journal of Production Economics **128**(1), 118–126 (2010)
30. Panwalkar, S.S., Iskander, W.: A survey of scheduling rules. Operations Research **25**(1), 45–61 (1977)
31. Pickardt, C.W., Hildebrandt, T., Branke, J., Heger, J., Scholz-Reiter, B.: Evolutionary generation of dispatching rule sets for complex dynamic scheduling problems. International Journal of Production Economics **145**(1), 67–77 (2013)
32. Pinedo, M.L.: Scheduling: Theory, Algorithms, and Systems, 3 edn. Springer Publishing Company, Incorporated (2008)
33. Rice, J.R.: The algorithm selection problem. Advances in Computers **15**, 65–118 (1976)
34. Ross, S., Bagnell, D.: Efficient reductions for imitation learning. In: Y.W. Teh, D.M. Titterington (eds.) Proceedings of the Thirteenth International Conference on Artificial Intelligence and Statistics (AISTATS-10), vol. 9, pp. 661–668 (2010)
35. Ross, S., Gordon, G.J., Bagnell, D.: A reduction of imitation learning and structured prediction to no-regret online learning. In: G.J. Gordon, D.B. Dunson (eds.) Proceedings of the Fourteenth International Conference on Artificial Intelligence and Statistics (AISTATS-11), vol. 15, pp. 627–635. Journal of Machine Learning Research - Workshop and Conference Proceedings (2011)
36. Ross, S., Melik-Barkhudarov, N., Shankar, K., Wendel, A., Dey, D., Bagnell, J., Hebert, M.: Learning monocular reactive uav control in cluttered natural environments. In: Robotics and Automation (ICRA), 2013 IEEE International Conference on, pp. 1765–1772 (2013)
37. Runarsson, T.: Ordinal regression in evolutionary computation. In: T. Runarsson, H.G. Beyer, E. Burke, J. Merelo-Guervs, L. Whitley, X. Yao (eds.) Parallel Problem Solving from Nature - PPSN IX, *Lecture Notes in Computer Science*, vol. 4193, pp. 1048–1057. Springer, Berlin, Heidelberg (2006)
38. Runarsson, T., Schoenauer, M., Sebag, M.: Pilot, rollout and monte carlo tree search methods for job shop scheduling. In: Y. Hamadi, M. Schoenauer (eds.) Learning and Intelligent Optimization, Lecture Notes in Computer Science, pp. 160–174. Springer Berlin Heidelberg (2012)
39. Russell, T., Malik, A.M., Chase, M., van Beek, P.: Learning heuristics for the superblock instruction scheduling problem. IEEE Trans. on Knowl. and Data Eng. **21**(10), 1489–1502 (2009)
40. Stafford, E.F.: On the Development of a Mixed-Integer Linear Programming Model for the Flowshop Sequencing Problem. Journal of the Operational Research Society **39**(12), 1163–1174 (1988)
41. Tay, J.C., Ho, N.B.: Evolving dispatching rules using genetic programming for solving multi-objective flexible job-shop problems. Computers and Industrial Engineering **54**(3), 453–473 (2008)
42. Xu, L., Hutter, F., Hoos, H., Leyton-Brown, K.: SATzilla-07: The design and analysis of an algorithm portfolio for SAT. Principles and Practice of ... (2007)
43. Yu, J.M., Doh, H.H., Kim, J.S., Kwon, Y.J., Lee, D.H., Nam, S.H.: Input sequencing and scheduling for a reconfigurable manufacturing system with a limited number of fixtures. The International Journal of Advanced Manufacturing Technology (2013)
44. Zhang, W., Dietterich, T.G.: A reinforcement learning approach to job-shop scheduling. In: Proceedings of the 14th international joint conference on Artificial Intelligence, *IJCAI'95*, vol. 2, pp. 1114–1120. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA (1995)