Todo list

Contents

Listing of figures

Listing of tables

Nomenclature

Job-shop Scheduling

n	number of jobs in shop
m	number of machines in shop
$\mathcal{J}$	set of jobs, $\{J_1, \ldots, J_j, \ldots, J_n\}$
$\mathcal{M}$	set of machines, $\{M_1, \ldots, M_a, \ldots, M_m\}$
p_{ja}	processing time for job J_j on machine M_a
σ_j	machine ordering for job J_j
$x_s(j, a)$	starting time for job J_j on machine M_a
$x_f(j, a)$	finishing time for job J_j on machine M_a
$s(a, j)$	flow between current and previous task on machine M_a
$C_{\max}$	makespan, i.e. maximum completion times for all tasks
$\boldsymbol{\chi}$	sequence of dispatches J_j to create (partial) schedule/solution
$\mathcal{R}^{(k)}$	ready-list of jobs at step k, $\mathcal{R} \subset \mathcal{J}$
ρ	percentage relative deviation from optimality
$\tilde{S}_{ja}^{(k)}$	set of applicable flow times on machine M_a that job J_j can be processed at step k
$\mathcal{U}(u_1, u_2)$	uniform distribution from the interval $I = [u_1, u_2] \subset \mathbb{R}$

Ordinal Regression

d	number of distinct features, i.e. dimension of $\boldsymbol{\varphi}$
ℓ	number of dispatches needed for a complete schedule, $\ell = n \cdot m$
Φ	training set
$\boldsymbol{\varphi}_k$	feature set, i.e. post-decision state, of a (partial) schedule at time k
S	preference set
$\mathcal{O}^{(k)}$	set of optimal dispatches at time k
$\mathcal{S}^{(k)}$	set of suboptimal dispatches at time k
τ	Kendall's τ statistic: normalised difference in number of concordant and discordant pairs.

Surrogate Modelling

j	refers to job J_j
a	refers to machine M_a
k	refers to the k-th operation, (j, a)
k	refers to dispatch/time step k for a schedule
opt	(known) optimum
bks	best known solution
sub	sub-optimum

Acronyms

BDR	blended dispatching rule
BKS	Best known solution
CDR	composite dispatching rule
DR	dispatching rule

CMA	covariance matrix adaptation
ES	evolution strategy
PFSP	permutation flow-shop scheduling problem
JSP	job-shop scheduling problem
LPT	largest processing time
LWR	least work remaining
MWR	most work remaining
SDR	simple priority dispatching rule
SPT	shortest processing time

Begin at the beginning and go on till you come to the end: then stop.

The King

1 Introduction

Chapter 1 Unfinished

Hand crafting heuristics for NP-hard problems is a time consuming trial-and-error process, requiring inductive reasoning or problem specific insights from their human designers. Furthermore, within a problems class, such as scheduling, it is possible to construct problem instances where one heuristic would outperform another.

Depending on the underlying data distribution of the problem instances, different heuristics perform differently. This is due to the fact that any algorithm which has superior performance in one class of problems is inevitably inferior over another class, i.e., *no free lunch* theorem (Wolpert and Macready, 1997). The success of a heuristic is how it manages to deal with and manipulate the characteristics of its given problem instance. Thus in order to understand more fully how a heuristic will eventually perform, one needs to look into what kind of problem instances are being introduced to the system. What defines a problem instance, e.g., what are its key features? And how can they help with designing better heuristics? Once the problem instances are fully understood, an appropriate learning algorithm can be implemented in order to create heuristics that are self-adapting to its

problem instances.

Given the ad-hoc nature of the heuristic design process there is clearly room for improving the process. Recently a number of attempts have been made to automate the heuristic design process. The ultimate goal of this dissertation is to automate optimisation heuristics via ordinal regression. The focal point will be based on a scheduling processes named job-shop scheduling problem, and some of its subclass, e.g., flow-shop scheduling problem. There are two main viewpoints on how to approach scheduling problems,

Local level by building schedules for one problem instance at a time.

Global level by building schedules for all problem instances at once.

For local level construction, a simple construction heuristic is applied, the schedule's features are collected at each dispatch iteration, from which a learning model will inspect the feature set to discriminate which operations are preferred to others via ordinal regression. The focus is essentially on creating a meaningful preference set composed of features and their ranks, as the learning algorithm is only run once to find suitable operators for the value function. However, for global level construction, there is no feature set collected beforehand since the learning model is optimised directly via evolutionary search. This required numerous costly value function evaluations. In fact it involves an indirect method of evaluation whether one learning model is preferable to another, w.r.t. which one yields the better expected mean. Evolutionary search only requires the rank of the candidates, and therefore it is appropriate to retain a sufficiently accurate surrogate for the value function during evolution in order to reduce the number of costly true value function evaluations. In this paradigm, ordinal regression is used for surrogate assisted evolutionary optimisation, where models are ranked – whereas on local level features were ranked.

1.1 Rice's framework for algorithm selection

The problem is to understand what underlying characteristics of the problem instances distinguishes *good* and on the other hand *bad* solutions when implementing a particular algorithm. Smith-Miles and Lopes (2011) were interested in discovering whether synthetic instances were in fact similar to real-world instances for timetabling problems. Moreover Smith-Miles and Lopes were interested in how varying algorithms perform on different data distributions. Hence the investigation of heuristic efficiency is closely intertwined to problem generation.

In order to formulate the relationship between problem structure and heuristic efficiency one can utilise Rice's framework for algorithm selection problem from 1976. The framework consists of four fundamental components, namely,

Problem space or instance space $\mathcal{P}$,
: set of problem instances;

Feature space $\mathcal{F}$,
: measurable properties of the instances in $\mathcal{P}$;

Algorithm space $\mathcal{A}$,
: set of all algorithms under inspection;

Performance space $\mathcal{Y}$,
: the outcome for $\mathcal{P}$ using an algorithm from $\mathcal{A}$.

For a given problem instance $\mathbf{x} \in \mathcal{P}$ with d features $\boldsymbol{\varphi}(\mathbf{x}) = \left[\varphi_1(\mathbf{x}), \ldots, \varphi_d(\mathbf{x})\right]^T \in \mathcal{F}$ and using algorithm $a \in \mathcal{A}$ the performance is $y = Y(a, \boldsymbol{\varphi}(\mathbf{x})) \in \mathcal{Y}$, where $Y: \mathcal{A} \times \mathcal{F} \mapsto \mathcal{Y}$ is the mapping for algorithm and feature space onto the performance space. This data collection is often referred to as meta-data.

In the context of Rice's framework, the aforementioned approaches to scheduling problems are to maximise its expected performance:

Local level

$$\max_{\mathcal{F}' \subset \mathcal{F}} \mathbb{E}\left[Y(a, \boldsymbol{\varphi}(\mathbf{x}))\right] \tag{1.1}$$

: The focal point is only using problem instances that represent the problem space, $\mathbf{x} \in \mathcal{P}' \subset \mathcal{P}$, in addition finding a suitable subset of the feature space, $\mathcal{F}' \subset \mathcal{F}|_{\mathcal{P}'}$. If done effectively, then the resulting learning model $a \in \mathcal{A}$ needs only be run once via ordinal regression.

Global level

$$\max_{a \in \mathcal{A}} \mathbb{E}\left[Y(a, \boldsymbol{\varphi}(\mathbf{x}))\right] \tag{1.2}$$

: This is straightforward approach as the algorithm $a \in \mathcal{A}$ is optimised directly given the entire training data $\mathbf{x} \in \mathcal{P}$. Alas, this comes at a great computational cost.

Note, the mappings $\boldsymbol{\varphi}: \mathcal{P} \mapsto \mathcal{F}$ and $Y: \mathcal{A} \mapsto \mathcal{Y}$ are the same for both paradigms.

1.2 Previous work

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 (Garey et al., 1976). 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. Heuristics are generally more time efficient but do not necessarily attain the global optimum. Therefore job-shop scheduling 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 (Meeran and Morshed, 2012), all from simple and straight forward dispatching rules to highly sophisticated frameworks.

MORE CITATIONS OF PREVIOUS TECHNIQUES

As Meeran and Morshed (2012) point out, in the field of Artificial Intelligence, then despite their 'intelligent' solutions the effectiveness of finding the optimum has been rather limited. However, combined with local-search methodologies, they can be improved upon significantly, as Meeran and Morshed showed with the use of a hybrid method using Genetic Algorithms (GA) and Tabu Search (TS). Therefore getting the best of both worlds, namely, the diverse global search obtained from GA and being complemented with the intensified local search capabilities of TS. Now, hybridisation of global and local methodologies is non-trivial. In general combination of the two improves performance, however, they often come at a great computational cost.

Various *learning* approaches have been applied to solving job-shop, such as reinforcement learning (Zhang and Dietterich, 1995), evolutionary learning (Tay and Ho, 2008), and supervised learning (Li and Olafsson, 2005, Malik et al., 2008). The approach taken in this dissertation is a supervised learning classifier using ordinal regression.

A common way of finding a good feasible solution for job-shop scheduling is applying construction heuristics with some dispatching rules, e.g., choosing a task corresponding to longest or shortest operation time; most or least successors; or ranked positional weight, i.e., sum of operation times of its predecessors. Ties are broken in an arbitrary fashion or by another heuristic rule. A summary of over 100 classical dispatching rules for scheduling can be found in Panwalkar and Iskander (1977), and it is noted that these classical dispatching

rules are continually used in research. There is no dominant rule, but the most effective have been single priority dispatching rules based on job processing attributes (Haupt, 1989). Tay and Ho (2008) showed that combining dispatching rules, with the aid of genetic programming, is promising, however, there is large number of rules to choose from, thus its combinations require expert knowledge or extensive trial-and-error process.

The current literature in scheduling focuses on different objectives, e.g., Chang (1996) minimises the due-date tightness and Drobouchevitch and Strusevich (2000), Gao et al. (2007) look into solving for bottleneck machines. In this dissertation only minimisation of the makespan will be considered, thus ignoring all due-date constraints.

This is not 'current'

Model assumptions can also vary, e.g., Thiagarajan and Rajendran (2005) incorporate different earliness, tardiness and holding costs. Moreover, it is possible to reduce job-shop to flow-shop problem, since in practice most jobs in job-shop use the machines in the same order (Guinet and Legrand, 1998, Ho et al., 2007).

Instead of using construction heuristics that creates job-shop schedules by sequentially dispatching one job at a time, one could work with complete feasible schedules and iteratively repairing them for a better result. Such was the approach by Zhang and Dietterich (1995) who studied space shuttle payload processing by using reinforcement learning, 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. This approach of a two phased process of construction and improvement is also implemented in timetable scheduling, e.g., Asmuni et al. (2009) used a fuzzy approach in considering multiple heuristic ordering in the construction process, and only allowed feasible schedules to be passed to the improvement phase.

The alternative to hand-crafting heuristics, is to implement an automatic way of learning heuristics using a data driven approach. Data can be generated using a known heuristic, such an approach is taken in Li and Olafsson (2005) for job-shop where a LPT-heuristic is applied. Afterwards, a decision tree is used to create a dispatching rule with similar logic. However, this method cannot outperform the original LPT-heuristic used to guide the search. For instruction scheduling, this drawback is confronted in Malik et al. (2008), Olafsson and Li (2010), Russell et al. (2009) by using an optimal scheduler, computed off-

line. The optimal solutions are used as training data and a decision tree learning algorithm applied as before. Preferring simple to complex models, the resulting dispatching rules gave significantly better schedules than using popular heuristics in that field, and a lower worst-case factor from optimality. A similar approach is taken for timetable scheduling in Burke et al. (2006), using case based reasoning. Training data is guided by the two best heuristics for timetable scheduling. Burke et al. 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. Again, stressing the importance of meaningful feature selection.

1.3 Contributions

The approach in this dissertation differs from previous studies, as it uses a simple linear combination of features found using a linear classifier based on ordinal regression.

Discuss contributions once draft is ready...

1.4 Outline

An approach based on supervised learning on optimal schedules will be investigated and its effectiveness illustrated by improving upon well known dispatch rules for job-shop scheduling in Chapter 5. The method of generating training data is shown to be critical for the success of the method, as shown in Section 5.3. Moreover the choice of problem instances under consideration is worth considering, as discussed in Chapter 3, and will be used throughout in the subsequent chapters.

The preliminary experiments done in Ingimundardottir and Runarsson (2012) investigated the characteristics of difficult job-shop schedules for a single heuristic, continuing

with that research, Chapter 4 compares a set of widely used dispatching rules on different problem spaces in the hopes of extrapolating where an algorithm excels in order to aid its failing aspects, which will be beneficial information for the creation of learning models in Chapter 5, as they are dependant on features based on those same dispatching rules under investigation.

Update outline once draft is ready...

Read the directions and directly you will be directed in the right direction.

Doorknob

2 Scheduling

Scheduling problems, which occur frequently in practice, are a category within combinatorial optimisation problems. A subclass of scheduling problems is the job-shop scheduling problem (JSP), which is widely studied in operations research. Job-shop deals with the allocation of tasks of competing resources where its goal is to optimise a single or multiple objectives. Job-shop's analogy is from manufacturing industry where a set of jobs are broken down into tasks that must be processed on several machines in a workshop. Furthermore, its formulation can be applied on a wide variety of practical problems in real-life applications which involve decision making, therefore its problem-solving capabilities has a high impact on many manufacturing organisation.

Deterministic JSP is the most *general* case for classical scheduling problems (Jain and Meeran, 1999). Many other scheduling problems can be reformulated as JSP. For instance the widely studied *Travelling Salesman Problem* can be contrived as job-shop with the salesman as a single machine in use and the cities to be visited are the jobs to be processed. Moreover, the general form of JSP assumes that each job can have its own distinctive flow pattern through the machines, which is independent of the other jobs. In the case where

all jobs share the same permutation route, job-shop is reduced to a permutation flow-shop scheduling problem (FSP) (Guinet and Legrand, 1998, Tay and Ho, 2008). Therefore, without loss of generality, this dissertation is structured around JSP.

2.1 Job-shop scheduling problem

JSP considered for this dissertation is where n jobs, $\mathcal{J} = \{J_j\}_{j=1}^{n}$, are scheduled on a finite set, $\mathcal{M} = \{M_a\}_{a=1}^{m}$, of m machines, subject to the constraint that each job J_j must follow a predefined machine order (a chain or sequence of m operations, $\boldsymbol{\sigma}_j = [\sigma_{j1}, \sigma_{j2}, \ldots, \sigma_{jm}]$) and that a machine can handle at most one job at a time. The objective is to schedule jobs in such a manner as to minimise the maximum completion times for all tasks, which is also known as the makespan, $C_{\max}$. A common notion for this family of scheduling problems, i.e., a m machine JSP w.r.t. minimising makespan, is $Jm||C_{\max}$ (cf. Pinedo, 2008). In addition, for FSP w.r.t. minimising makespan the notation is $Fm||C_{\max}$. An additional constraint commonly considered are job release-dates and due-dates, and then the objective is generally minimising the maximum lateness, denoted $Jm||L_{\max}$, however, those constraints will not be considered here.

Henceforth, the index j refers to a job $J_j \in \mathcal{J}$ while the index a refers to a machine $M_a \in \mathcal{M}$. 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. Moreover, index k will denote the time step of the operation. Note that once an operation is started, it must be completed uninterrupted, i.e., pre-emption is not allowed. Moreover, there are no sequence dependent setup times.

2.2 Mathematical formulation

For any given JSP, which consists of n jobs for m machines, then 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 completion time is denoted $x_f(j, a)$ where,

$$x_f(j, a) := x_s(j, a) + p_{ja} \tag{2.1}$$

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.,

$$x_s(j, \boldsymbol{\sigma}_j(a)) \geq x_f(j, \boldsymbol{\sigma}_j(a-1)) \tag{2.2}$$

for all $J_j \in \mathcal{J}$ 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 permutation route, JSP is reduced to a FSP (Guinet and Legrand, 1998, Tay and Ho, 2008).

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

$$x_s(j, a) \geq x_f(j', a) \quad \text{or} \quad x_s(j', a) \geq x_f(j, a) \tag{2.3}$$

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

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,

$$C_{\max} := \max\{x_f(j, \boldsymbol{\sigma}_j(m)) \mid J_j \in \mathcal{J}\}. \tag{2.4}$$

Clearly, w.r.t. minimum makespan, it is preferred that schedules are non-delay, i.e., the machines are not kept idle. The time in which machine M_a is idle between consecutive jobs J_j and $J_{j'}$ is called idle time, or flow,

$$s(a, j) := x_s(j, a) - x_f(j', a) \tag{2.5}$$

where J_j is the immediate successor of $J_{j'}$ on M_a. Although this is not a variable directly needed to construct a schedule for JSP, it is a key feature in order to measure the quality of the schedule.

2.3 Construction heuristics

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 is to schedule the dispatches as closely together as possible, i.e., minimise the schedule's flow. More specifically, once an operation (j, a) has been chosen from the ready-list, $\mathcal{R}$, by some dispatching rule, it can placed immediately after (but not prior) $x_f(j, \boldsymbol{\sigma}_j(a-1))$ on machine

M_a due to constraint Eq. (2.2). However to guarantee that constraint Eq. (2.3) 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 $x_f(j, \boldsymbol{\sigma}_j(a-1))$ one cannot implement Eq. (2.5) directly, instead it has to be updated as follows,

$$\tilde{s}(a, j') \quad := \quad x_s(j'', a) - \max\{x_f(j', a), x_f(j, \boldsymbol{\sigma}_j(a-1))\} \tag{2.6}$$

for all already dispatched jobs $J_{j'}, J_{j''} \in \mathcal{J}_a$ 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.,

$$\tilde{S}_{ja} \quad := \quad \{J_{j'} \in \mathcal{J}_a \mid \tilde{s}(a, j') \geq p_{ja}\}. \tag{2.7}$$

Now, there are several heuristic methods for selecting a slot from Section 2.3, e.g., if the main concern were to utilise the slot space, then choosing the slot with the smallest idle time would yield a closer-fitted schedule and leaving greater idle times undiminished for subsequent dispatches on M_a. However dispatching J_j in the first slot would result in its earliest possible release time, which would be beneficial for subsequent dispatches for J_j. Preliminary experiments favoured dispatching in the first (earliest) slot,* and henceforth will be used throughout the dissertation.

Note that the choice of slot is an intrinsic heuristic within the construction heuristic. Construction heuristics are designed in such a manner that they limit the search space. Preferably without excluding the true optimum. The focus of this dissertation, however, is on learning the priority of the jobs on the ready-list, for a fixed construction heuristic. Hence there are some problem instances in which the optimum makespan cannot be achieved due to the limitations of the schedule's construction heuristic of not being properly able to differentiate between which slot from $\tilde{S}_{ja}$ is the most effective. Instead, hopefully, the learning algorithm will be able to spot these problematic situations, should they arise, by inspecting the schedule's features and translate that into the jobs' priorities.

*Preliminary experiments of 500 JSP instances where inspected: First slot chosen could always achieve its known optimum by implementing the pseudo code in Fig. 2.2, however only 97% of the instances when choosing the smallest slot.

2.4 Example

Let's define a six-job and five-machine job-shop problem, with the following $\mathbf{p} \sim \mathcal{U}(1, 99)$ and $\boldsymbol{\sigma}$ matrices,

$$\mathbf{p} = \begin{bmatrix} 91 & 53 & \mathbf{31} & 59 & 84 \\ 15 & \mathbf{22} & 23 & 13 & 92 \\ 54 & 33 & 15 & 62 & 83 \\ 83 & 51 & 80 & 97 & \mathbf{40} \\ 51 & 27 & \mathbf{74} & 85 & 70 \\ 59 & \mathbf{69} & 66 & 46 & 20 \end{bmatrix}, \quad \boldsymbol{\sigma} = \begin{bmatrix} \boldsymbol{\sigma}_1 \\ \boldsymbol{\sigma}_2 \\ \boldsymbol{\sigma}_3 \\ \boldsymbol{\sigma}_4 \\ \boldsymbol{\sigma}_5 \\ \boldsymbol{\sigma}_6 \end{bmatrix} = \begin{bmatrix} 4 & 5 & \mathbf{3} & 2 & 1 \\ 1 & \mathbf{3} & 2 & 4 & 5 \\ 3 & 1 & 2 & 4 & 5 \\ 2 & 3 & 5 & 1 & \mathbf{4} \\ 2 & 5 & \mathbf{4} & 3 & 1 \\ 2 & \mathbf{3} & 5 & 1 & 4 \end{bmatrix}. \tag{2.8}$$

Now assume 15 operations have already dispatched been made, i.e., the red entries, by using the following sequence of jobs,

$$\boldsymbol{\chi} = [J_3, J_3, J_3, J_3, J_4, J_4, J_5, J_1, J_1, J_2, J_4, J_6, J_4, J_5, J_3] \tag{2.9}$$

hence the ready-list is $\mathcal{R} = \{J_1, J_2, J_4, J_5, J_6\}$ (note that J_3 has traversed through all of its machines) indicating the 5 potential jobs to be dispatched at step $k = 16$, denoted in bold. Figure 2.1 illustrates the temporal partial schedule of the dispatching process. 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 with a dotted line.

If the job with the shortest processing time were to be scheduled next, i.e., implementing the SPT heuristic, then J_2 would be dispatched. Similarly, for the LPT (largest processing time) heuristic then J_5 would be dispatched. Other DRs use features not directly observable from looking at the current partial schedule (but easy to keep record of), for example by assigning jobs with most or least total processing time remaining, i.e., MWR and LWR heuristics, who would yield J_5 and J_4, respectively.

To summarise, in order to create a schedule for JSP, a construction heuristic is chosen with some DR to determine the priority of the jobs on the ready-list, $\mathcal{R}$. Figure 2.2 outlines the pseudo code for the dispatching process of a JSP problem instance.

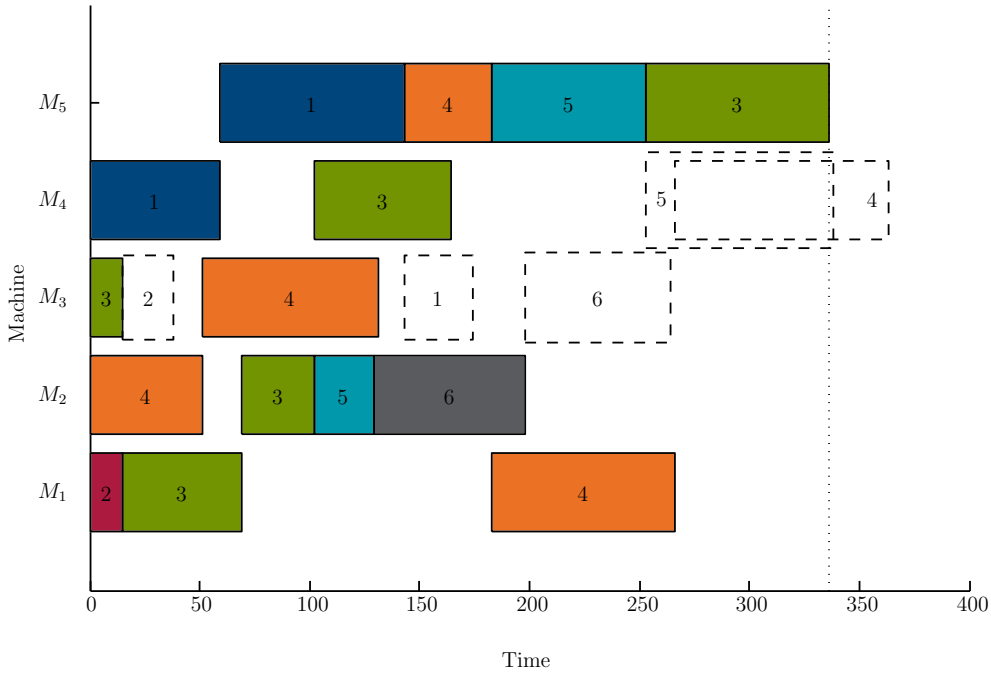


Figure 2.1: Gantt chart of a partial JSP schedule after 15 dispatches: Solid and dashed boxes represent $\boldsymbol{\chi}$ and $\mathcal{R}^{(16)}$, respectively. Current $C_{\max}$ denoted as dotted line.

0 *Initialization*: Let $\boldsymbol{\chi} = \emptyset$ denote
the current dispatching sequence.
1 **for** $k := 1$ to $\ell = n \cdot m$ **do** *(at each dispatch iteration)*
2 **for** $J_j \in \mathcal{R}^{(k)} \subset \mathcal{J}$ **do** *(inspect ready-list)*
3 $I_j^{DR} \leftarrow \mathrm{DR}([\boldsymbol{\chi}, j], Y)$ *(priority J_j)*
4 **od**
5 $j^* \leftarrow \mathrm{argmax}_{j \in \mathcal{R}^{(k)}} \{I_j^{DR}\}$ *(choose highest priority)*
6 $\chi_k \leftarrow j^*$ *(dispatch j^*)*
7 **od**
8 $C_{max} \leftarrow Y(\boldsymbol{\chi})$ *(makespan)*

Figure 2.2: Pseudo code for constructing a JSP sequence using a dispatching rule (DR) for a fixed construction heuristic (CH).

Henceforth, a *sequence* will refer to the sequential ordering of the dispatches of tasks to machines, i.e., (j, a); the collective set of allocated tasks to machines, which is interpreted by its sequence, is referred to as a *schedule*; a *scheduling policy* will pertain to the manner in which the sequence is manufactured for an (near) optimal schedule: be it a SDR such as MWR or via evolutionary search, etc.

2.5 Single-based priority dispatching rules

Dispatching rules (DR) are of a construction heuristics, where one starts with an empty schedule and adds sequentially on one operation (or task) at a time. Namely, at each time step k, an operation is dispatched which has the highest priority of the ready-list, $\mathcal{R}^{(k)} \subset \mathcal{J}$, i.e., the jobs who still have operations unassigned. If there is a tie, some other priority measure is used.

A *single-based priority dispatching rule* (SDR), or simple priority dispatching rule, is a function of features of the jobs and/or machines of the schedule. The features can be constant or vary throughout the scheduling process. For instance, the priority may depend on job processing attributes, such as which job has,

Shortest immediate processing time (SPT)
: greedy approach to finish shortest tasks first,

Longest immediate processing time (LPT)
: greedy approach to finish largest tasks first,

Least work remaining (LWR)
: whose intention is to complete jobs advanced in their progress, i.e., minimising the ready-list $\mathcal{R}$,

Most work remaining (MWR)
: whose intention is to accelerate the processing of jobs that require a great deal of work, yielding a balanced progress for all jobs during dispatching, however in-process inventory can be high.

These rules are the ones most commonly applied in the literature due to their simplicity and efficiency, therefore they will be referenced throughout the dissertation. However there are many more available, e.g., randomly selecting an operation with equal possibility

(RND); minimum slack time (MST); smallest slack per operation (S/OP); and using the aforementioned dispatching rules with predetermined weights. A survey of more than 100 of such rules are presented in Panwalkar and Iskander (1977), however the reader is referred to an in-depth survey for SDRs by Haupt (1989).

To summarise, SDRs assign an index to each job of the ready-list waiting to be scheduled, and are generally only based on few features and simple mathematical operations.

2.6 Features for job-shop

A DR may need to perform a one-step look-ahead and observes features of the partial schedule to make a decision, for example by observing the resulting temporal makespan. These emanated observed features are sometimes referred to as an *after-state* or *post-decision state*.

Features are used to grasp the essence of the current state of the schedule. Temporal scheduling features applied in this dissertation for a job J_j to be dispatched on machine M_a are given in Table 2.1. Note, from a job-oriented viewpoint, for a job already dispatched $J_j \in \mathcal{J}$ the corresponding set of machines now processed is $\mathcal{M}_j \subset \mathcal{M}$. Similarly from the machine-oriented viewpoint, $M_a \in \mathcal{M}$ with corresponding $\mathcal{J}_a \subset \mathcal{J}$.

The features of particular interest were obtained from inspecting the aforementioned SDRs from Section 2.5: φ_1-φ_8 and φ_9-φ_{12} are job-related and machine-related attributes of the current schedule, respectively.

Some features are directly observed from the partial schedule, such as the job- and machine-related features. In addition there are flow-related, φ_{13}-φ_{15}, which measure the influence of idle time on the schedule, and current makespan-related, φ_{16}-φ_{18}.

Note that φ_1-φ_{18} are only based on the current step of the schedule, i.e., schedule's *local features*, and might not give an accurate indication of how it will effect the schedule in the long run. Therefore, a set of features are needed to estimate the schedule's overall performance, referred to as its *global features*. The approach here is to use well known SDRs, φ_{19}-φ_{22}, as a benchmark by retrieving what would the resulting $C_{\max}$ would be given if that SDR would be implemented from that point forward. Moreover, random completion of the partial schedule are implemented, here φ_{23} corresponds to 100 random rollouts, which can be used to identify which features $\boldsymbol{\varphi}$ are promising on a long-term basis.

All of the features vary throughout the scheduling process, w.r.t. operation belonging

to the same time step k, save for φ_5 which varies between jobs; φ_{18} to keep track of features' evolution w.r.t. the scheduling process; and φ_{17} which is static for a given problem instance, but used for normalising other features, such as work-remaining based (φ_7 and φ_{11}) or makespan-based (φ_{22}-φ_{23}) ones. In addition, φ_9, is reported in order to distinguish which features are in conflict with each other.

2.7 Composite dispatching rules

Jayamohan and Rajendran (2004) showed that a careful combination of dispatching rules can perform significantly better. These are referred to as *composite dispatching rules* (CDR), where the priority ranking is an expression of several SDRs. For instance, optimising JSP w.r.t. L_{max} for one machine (Pinedo, 2008, see. chapter 14.2), one can combine SDRs that are optimal for a different criteria of problem instances, which complement each other as a CDR, e.g., combining the SDRs WSPT (SPT weighted w.r.t. $\mathcal{J}$), "which is optimal when all release dates and due dates are zero," and minimum slack first (MS), "which is optimal when all due dates are sufficiently loose and spread out," one gets the CDR apparent tardiness cost (ATC) which can work well on a broader set of problem instances than the original SDRs by themselves.

CDRs can deal with greater number of features and more complicated form, in short, CDR are a combination of several SDRs. For instance let CDR be comprised of d SDRs, then the index I for job J_j using CDR is,

$$I_j^{CDR} = \sum_{i=1}^{d} w_i \cdot \mathrm{DR}^i(\boldsymbol{\varphi}_j) \tag{2.10}$$

where $w_i > 0$ and $\sum_{i=0}^{d} w_i = 1$, then w_i gives the *weight* of the influence of DR^i (which could be SDR or another CDR) to CDR. Note, each DR^i is a function of the job J_j's feature state $\boldsymbol{\varphi}_j$.

2.7.1 Blended dispatching rules

Since each DR yield a priority index I^{DR} then it is easy to translate its index as a performance measure a, i.e., $a : I^{DR} \mapsto \mathcal{Y}$. Then it is possible to combine several performance measures into a single DR, these are referred to as *blended dispatching rules* (BDR), where

Table 2.1: Feature space $\mathcal{F}$ for JSP where job J_j on machine M_a given the resulting temporal schedule after dispatching (j, a).

φ	Feature description	Mathematical formulation	Shorthand
	job related		
φ_1	job processing time	p_{ja}	proc
φ_2	job start-time	$x_s(j, a)$	startTime
φ_3	job end-time	$x_f(j, a)$	endTime
φ_4	job arrival time	$x_f(j, a-1)$	arrival
φ_5	total processing time	$\sum_{a \in \mathcal{M}} p_{ja}$	totalProc
φ_6	time job had to wait	$x_s(j, a) - x_f(j, a-1)$	wait
φ_7	total work remaining for job	$\sum_{a' \in \mathcal{M} \setminus \mathcal{M}_j} p_{ja'}$	wrmJob
φ_8	number of assigned operations for job	$\lvert\mathcal{M}_j\rvert$	jobOps
	machine related		
φ_9	machine ID	a	mac
φ_{10}	when machine is next free	$\max_{j' \in \mathcal{J}_a}\{x_f(j', a)\}$	macFree
φ_{11}	total work remaining for machine	$\sum_{j' \in \mathcal{J} \setminus \mathcal{J}_a} p_{j'a}$	wrmMac
φ_{12}	number of assigned operations for machine	$\lvert\mathcal{J}_a\rvert$	macOps
	flow related		
φ_{13}	change in idle time by assignment	$\Delta s(a, j)$	slotsReduced
φ_{14}	total idle time for machine	$\sum_{j' \in \mathcal{J}_a} s(a, j')$	slots
φ_{15}	total idle time for all machines	$\sum_{a' \in \mathcal{M}} \sum_{j' \in \mathcal{J}_{a'}} s(a', j')$	slotsTotal
	current makespan related		
φ_{16}	current makespan	$\max_{(j', a') \in \mathcal{J} \times \mathcal{M}_{j'}}\{x_f(j', a')\}$	makespan
φ_{17}	total work remaining for all jobs/mac	$\sum_{j' \in \mathcal{J}} \sum_{a' \in \mathcal{M} \setminus \mathcal{M}_{j'}} p_{j'a'}$	wrmTotal
φ_{18}	current step in the dispatching process	$\lvert\boldsymbol{\chi}\rvert$	step
	final makespan related		
φ_{19}	final makespan using SPT	$C_{\max} \lvert \mathrm{DR} = \mathrm{SPT}$	SPT
φ_{20}	final makespan using LPT	$C_{\max} \lvert \mathrm{DR} = \mathrm{LPT}$	LPT
φ_{21}	final makespan using LWR	$C_{\max} \lvert \mathrm{DR} = \mathrm{LWR}$	LWR
φ_{22}	final makespan using MWR	$C_{\max} \lvert \mathrm{DR} = \mathrm{MWR}$	MWR
φ_{23}	final makespan using 100 random rollouts	$\{C_{\max} \lvert \mathrm{DR} = \mathrm{RND}\}_{i=1}^{100}$	RND

an overall blended priority index P is defined as

$$P_j = \sum_{i=1}^{d} w_i \cdot a_i \tag{2.11}$$

where $w_i > 0$ and $\sum_{i=0}^{d} w_i = 1$, then w_i gives the weight of the proportional influence of performance measure a_i (based on some SDR or CDR) to the overall priority.

Generally the weights $\mathbf{w}$ chosen by the algorithm designer a priori. A more sophisticated approach would to learn have the algorithm discover these weights autonomously, for instance via evolutionary search or ordinal regression, to be discussed in Chapter 6 and Chapter 5, respectively.

2.7.2 Automated discovery of dispatching rules

Mönch et al. (2013) stress the importance of automated discovery of DRs and named several of successful implementations in the field of semiconductor wafer fabrication facilities. However, Mönch et al. note that this sort of investigation is still in its infancy and subject for future research.

A recent editorial of the state-of-the-art approaches in advanced dispatching rules for large-scale manufacturing systems by Chen et al. (2013) points out 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.

implying that there has not been much automation in the process of discovering new dispatching rules, which is the ultimate goal of this dissertation, i.e., automate creating optimisation heuristics for scheduling.

With meta heuristics one can use existing DRs and use for example *portfolio-based algorithm selection* (Gomes and Selman, 2001, Rice, 1976), either based on a single instance or class of instances (Xu et al., 2007) to determine which DR to choose from. Instead of optimising which algorithm to use under what data distributions, such as the case of portfolio algorithms, the approach taken in this dissertation is more similar to that of *meta learning* (Vilalta and Drissi, 2002) which is the study of how learning algorithms can be improved, i.e., exploiting their strengths and remedy their failings, in order for a better

algorithm design. Thus creating an adaptable learning algorithm that dynamically finds the appropriate dispatching rule to the data distribution at hand.

Kalyanakrishnan and Stone (2011) point out that meta learning can be very fruitful in reinforcement learning, and in their experiments they 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.

Nguyen et al. (2013) proposed a novel *iterative dispatching rules* for JSP which learns 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 Nguyen et al. stress that there still remains room for improvement.

Korytkowski et al. (2013) implemented *Ant Colony Optimisation* to select the best DR from a selection of nine DRs for JSP and their experiments showed that the choice of DR do affect the results and that for all performance measures considered it was better to have all of the DRs to choose from rather than just a single DR at a time. Similarly, Lu and Romanowski (2013) investigate eleven SDRs for JSP to create a pool of thirty three CDRs that strongly outperformed the ones they were based on, which is intuitive since where one SDR might be failing, another could be excelling, hence combining them should yield a better CDR. Lu and Romanowski create their CDRs with *multi-contextual functions* based either on machine idle time or job waiting time, so one can say that the CDRs are a combination of those two key features of the schedule and then the basic DRs. However, there are no combinations of the basic DR explored, only machine idle time and job waiting time. Yu et al. (2013) used priority rules to combine twelve existing DRs from the literature, in their approach they had forty eight priority rules combinations, yielding forty eight different models to implement and test. This is a fairly ad hoc solution and there is no guarantee the optimal combination of DRs is found.

2.8 Rice's framework for job-shop

Rice's framework for algorithm selection (discussed in Section 1.1) has already been formulated for job-shop (cf. Ingimundardottir and Runarsson, 2012, Smith-Miles and Lopes, 2011, Smith-Miles et al., 2009), as follows,

Problem space $\mathcal{P}$ is defined as the union of N problem instances consisting of processing time and ordering matrices,

$$\mathcal{P} = \left\{ (p_{ja}^{(i)}, \boldsymbol{\sigma}_j^{(i)}) \mid J_j \in \mathcal{J},\ M_a \in \mathcal{M} \right\}_{i=1}^{N} \tag{2.12}$$

Problem generators for $\mathcal{P}$ are given in Chapter 3.

Feature space $\mathcal{F}$ which is outlined in Section 2.6. Note, these are not the only possible set of features, however, the local feature, φ_1-φ_{18}, are built on the work by Ingimundardottir and Runarsson (2011a), Smith-Miles et al. (2009) and deemed successful in capturing the essence of a job-shop data structure;

Algorithm space $\mathcal{A}$ is simply the scheduling policies under consideration, e.g., SDRs from Section 2.5,

$$\mathcal{A} = \{\text{SPT, LPT, LWR, MWR, } \dots\} \tag{2.13}$$

Performance space $\mathcal{Y}$ is based on the resulting $C_{\max}$, defined by Eq. (2.4). The optimum makespan is denoted $C_{\max}^{\text{opt}}$, and the makespan obtained from the scheduling policy $A \in \mathcal{A}$ under inspection by $C_{\max}^{A}$. Since the optimal makespan varies between problem instances the performance measure is the following,

$$\rho = \frac{C_{\max}^{A} - C_{\max}^{opt}}{C_{\max}^{\text{opt}}} \cdot 100\% \tag{2.14}$$

which indicates the deviation from optimality, ρ. Thus $\mathcal{Y}$ is the following,

$$\mathcal{Y} = \{\rho_i\}_{i=1}^{N} \tag{2.15}$$

The mapping $Y : \mathcal{A} \times \mathcal{F} \mapsto \mathcal{Y}$ is the step-by-step construction heuristic introduced in Section 2.3.

If it had grown up, it would have made a dreadfully ugly child; but it makes rather a handsome pig, I think.

Alice

3

Problem generators

Chapter 3 Practically finished

Synthetic problem instances for JSP and FSP will be used throughout this dissertation. The problem spaces are detailed in the Sections 3.1 and 3.2 for JSP and FSP, respectively. Moreover, a brief summary is given in Table 3.1. Following the approach in Watson et al. (2002), difficult problem instances are not filtered out beforehand, although they will be specifically addressed in Chapter 4.

Although real-world instances are desirable, unfortunately they are scarce, hence in some experiments, problem instances from OR-Library maintained by Beasley (1990) will be used as benchmark problems, and detailed in Section 3.3. It is noted, that some of the instances are also simulated, but the majority are based on real-world instances, albeit sometimes simplified.

3.1 Job-shop

Problem instances for JSP are generated stochastically by fixing the number of jobs and machines and discrete processing time are i.i.d. and sampled from a discrete uniform distribution. Two different processing times distributions were explored, namely,

JSP random $\mathcal{P}^{n\times m}_{j.rnd}$
where $\mathbf{p} \sim \mathcal{U}(1, 99)$;

JSP random-narrow $\mathcal{P}^{n\times m}_{j.rndn}$
where $\mathbf{p} \sim \mathcal{U}(45, 55)$.

The machine ordering is a random permutation of all of the machines in the job-shop. For each JSP class N_{train} and N_{test} instances were generated for training and testing, respectively. Values for N are given in Table 3.1.

Although in the case of $\mathcal{P}^{n\times m}_{j.rnd}$ this may be an excessively large range for the uniform distribution, it is however chosen in accordance with the literature (Demirkol et al., 1998) for creating synthesised $Jm||C_{\max}$ problem instances. In addition, w.r.t. the machine ordering, one could look into a subset of JSP where the machines are partitioned into two (or more) sets, where all jobs must be processed on the machines from the first set (in some random order) before being processed on any machine in the second set, commonly denoted as $Jm|2\text{sets}|C_{\max}$ problems, but as discussed in Storer et al. (1992) this family of JSP is considered "hard" (w.r.t. relative error from best known solution) in comparison with the "easy" or "unchallenging" family with the general $Jm||C_{\max}$ setup. This is in stark contrast to Watson et al. (2002) whose findings showed that structured $Fm||C_{\max}$ were much easier to solve than completely random structures. Intuitively, an inherent structure in machine ordering should be exploitable for a better performance. However, for the sake of generality, a random structure is preferred as they correspond to difficult problem instances in the case of JSP. Whereas, structured problem subclasses will be explored for FSP.

Moreover, in order to inspect the impact of any slight change within the problem spaces, two mutated versions were created based on $\mathcal{P}^{n\times m}_{j.rnd}$, namely,

JSP random with job variation $\mathcal{P}^{n\times m}_{j.rnd,J_1}$
where the first job, J_1, is always twice as long as its random counterpart, i.e., $\tilde{p}_{1a} = 2 \cdot p_{1a}$, where $p \in \mathcal{P}^{n\times m}_{j.rnd}$, for all $M_a \in \mathcal{M}$.

JSP random with machine variation $\mathcal{P}^{n\times m}_{j.rnd,M_1}$

where the first machine, M_1, is always twice as long as its random counterpart, i.e., $\tilde{p}_{j1} = 2 \cdot p_{j1}$, where $p \in \mathcal{P}^{n\times m}_{j.rnd}$, for all $J_j \in \mathcal{J}$.

Therefore making job J_1 and machine M_1 bottlenecks for $\mathcal{P}^{n\times m}_{j.rnd,J_1}$ and $\mathcal{P}^{n\times m}_{j.rnd,M_1}$, respectively.

3.2 Flow-shop

Problem instances for FSP are generated using Watson et al. (2002) problem generator*. There are two fundamental types of problem classes: non-structured versus structured.

Firstly, there are two "conventional" random, i.e., non-structured, problem classes for FSP where processing times are i.i.d. and uniformly distributed,

FSP random $\mathcal{P}^{n\times m}_{f.rnd}$

where $\mathbf{p} \sim \mathcal{U}(1, 99)$ whose instances are equivalent to Taillard (1993)†;

FSP random narrow $\mathcal{P}^{n\times m}_{f.rndn}$

where $\mathbf{p} \sim \mathcal{U}(45, 55)$.

In the JSP context $\mathcal{P}^{n\times m}_{f.rnd}$ and $\mathcal{P}^{n\times m}_{f.rndn}$ are analogous to $\mathcal{P}^{n\times m}_{j.rnd}$ and $\mathcal{P}^{n\times m}_{j.rndn}$, respectively.

Secondly, there are three structured problem classes of FSP which are modelled after real-world *characteristics* in flow-shop manufacturing, namely,

FSP job-correlated $\mathcal{P}^{n\times m}_{f.jc}$

job processing times are dependent on job index, however independent of machine index. Job-correlation can be of degree $0 \leq a \leq 1$;

FSP machine-correlated $\mathcal{P}^{n\times m}_{f.mc}$

job processing times are dependent on machine index, however independent of job index. Machine-correlation can be of degree $0 \leq a \leq 1$;

FSP mixed-correlated $\mathcal{P}^{n\times m}_{f.mxc}$

job processing times are dependent on machine and job indices. Mixed-correlation can be of a degree $0 \leq a \leq 1$.

*Both code, written in C++, and problem instances used in their experiments can be found at: `http://www.cs.colostate.edu/sched/generator/`

†Taillard's generator is available from the OR-Library.

Table 3.1: Problem space distributions used in experimental studies. Note, problem instances are synthetic and each problem space is i.i.d. and '–' denotes not available.

type	name	size ($n \times m$)	N_{train}	N_{test}	note
JSP	$\mathcal{P}^{6\times5}_{j.rnd}$	6×5	500	500	random
	$\mathcal{P}^{6\times5}_{j.rndn}$	6×5	500	500	random-narrow
	$\mathcal{P}^{6\times5}_{j.rnd,J_1}$	6×5	500	500	random with job variation
	$\mathcal{P}^{6\times5}_{j.rnd,M_1}$	6×5	500	500	random with machine variation
	$\mathcal{P}^{10\times10}_{j.rnd}$	10×10	300	200	random
	$\mathcal{P}^{10\times10}_{j.rndn}$	10×10	300	200	random-narrow
FSP	$\mathcal{P}^{6\times5}_{f.rnd}$	6×5	500	500	random
	$\mathcal{P}^{6\times5}_{f.rndn}$	6×5	500	500	random-narrow
	$\mathcal{P}^{6\times5}_{f.jc}$	6×5	500	500	job-correlated
	$\mathcal{P}^{6\times5}_{f.mc}$	6×5	500	500	machine-correlated
	$\mathcal{P}^{6\times5}_{f.mxc}$	6×5	500	500	mixed-correlation
	$\mathcal{P}^{10\times10}_{f.rnd}$	10×10	300	200	random

Note, for $a = 0.0$ the problem instances closely correspond to $\mathcal{P}^{n\times m}_{f.rnd}$, hence the degree of a controls the transition of random to structured FSP. However, if not otherwise stipulated, a value of $a = 1$ is assumed.

For each FSP class N_{train} and N_{test} instances were generated for training and testing, respectively. Values for N are given in Table 3.1. Moreover, an example of distribution of processing times are depicted in Fig. 3.1.

3.3 Benchmark problem suite

A total of 62 and 31 benchmark problems for JSP and FSP, respectively, were obtained from the Operations Research Library (OR-Library) maintained by Beasley (1990) and summarised in Table 3.2. Given the high problem dimensions of some problems, the

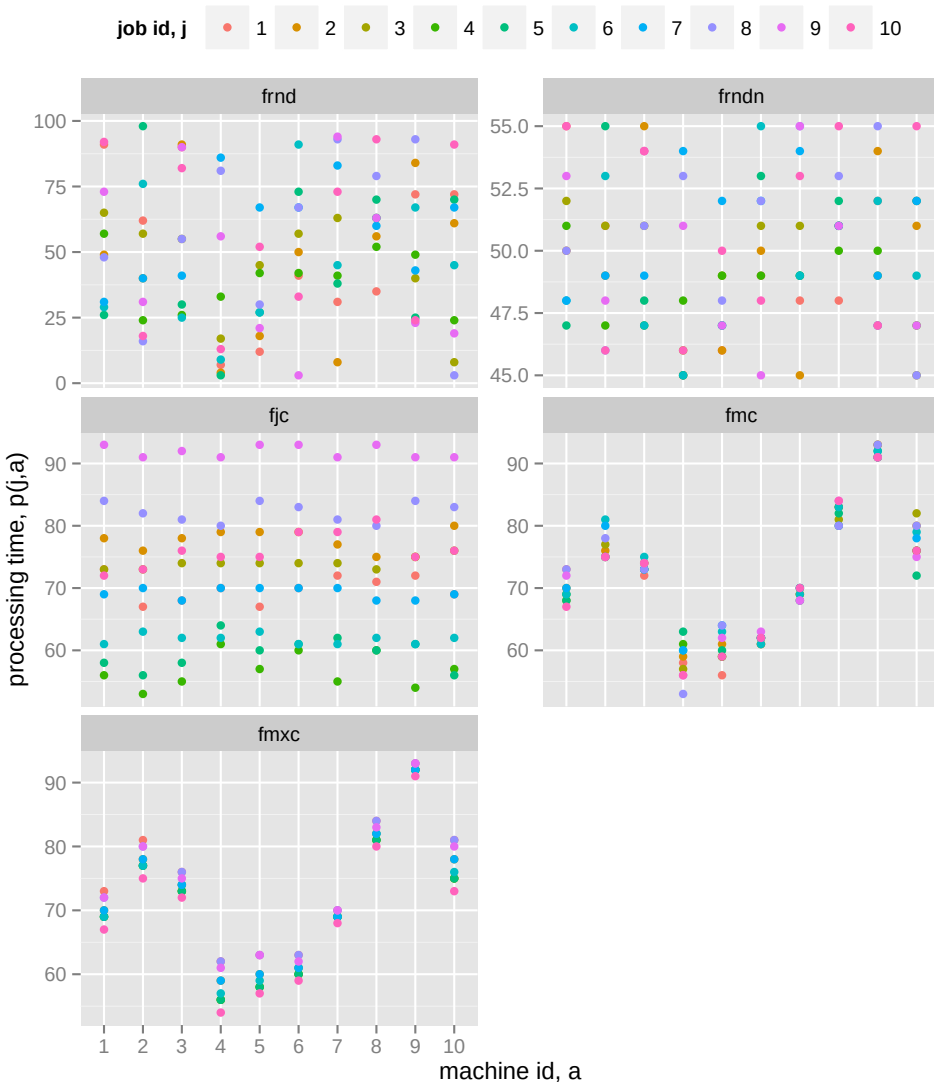


Figure 3.1: Examples of distribution of job processing times for 10 × 10 FSP with different types of structure. Machine indices are on the vertical axis, job indices are colour-coded and their corresponding processing times, p_{ja}, are on the horizontal axis.

optimum is not known, hence in those instances Eq. (2.14) will be reporting deviation from the latest best known solution (BKS) from the literature. For JSP, they are reported by Banharnsakun et al. (2012) save for $\mathcal{P}_{swv}$ which can be found in .

Ekki fundið BKS fyrir jsp swv and fsp OR-LIBRARY

Fisher and Thompson (1963) had one of the more notorious benchmark problems for JSP, and computationally expensive, however, now these instances have been solved to optimality.

Check with gurobi if that's accurate, and figure out which ones haven't yet been solved. Then report BKS.

Lawrence (1984) info – la01–la40

Similar to the synthetic JSP problem spaces discussed earlier, Adams et al. (1988) introduce five JSP instances with a random permutation of machine ordering and processing times following uniform distribution, $\mathbf{p} \sim \mathcal{U}(50, 100)$, for dimensions 10×10 and 20×15. Likewise, Yamada and Nakano (1992) consists of four 20×20 random problem instances, where $\mathbf{p} \sim \mathcal{U}(10, 50)$. Storer et al. (1992) introduce a set of JSP problems where the job processing times following a uniform distribution, $\mathbf{p} \sim \mathcal{U}(1, 100)$. There are a total of five problems in four dimension classes, namely, 20×10 (swv01–swv05), 20×15 (swv06–swv10), 50×10 (swv11–swv15) and 50×10 (swv16–swv20), where the first three classes are considered "hard" and the last one as "easy". Easy problems are ones corresponding to random machine ordering, whereas hard problems are partitioned in such a way the jobs must be processed on the first half of the machines before starting on the second half, i.e., $Jm|2\text{sets}|C_{\max}$.

Applegate and Cook (1991) introduced 10 problem instances of 10×10 JSP where generated such that the machine ordering was chosen by random users in order to make them "difficult." Moreover, the processing times were drawn at random, and the distribution that had the greater gap between its optimal value and standard lower bound was chosen.

For the FSP benchmarks, Heller (1960) introduces two deterministic instances based on "many-machine version of book-printing," where processing times for $n \in \{20, 100\}$ jobs and $m = 10$ machines are relatively short, i.e., $p_{ja} \in \{0, .., 9\}$. Carlier (1978) however comprises of eight problems (of various dimension) where there is high variance in processing times, presumably $\mathbf{p} \sim \mathcal{U}(1, 1000)$.

Reeves (1995) argue that completely random problem instances are unlikely to occur

Table 3.2: Benchmark problems from OR-Library used in experimental studies.

type	name	size $(n \times m)$	N_{test}	note	
JSP	$\mathcal{P}_{abz}$	various	5	Adams et al. (1988)	abz05–abz09 *
	$\mathcal{P}_{ft}$	various	3	Fisher and Thompson (1963)	ft06,ft10,ft20 *
	$\mathcal{P}_{la}$	various	40	Lawrence (1984)	la01–la40 *
	$\mathcal{P}_{orb}$	10×10	10	Applegate and Cook (1991)	orb01–orb10 *
	$\mathcal{P}_{swv}$	various	20	Storer et al. (1992)	swv01–swv20
	$\mathcal{P}_{yn}$	20×20	4	Yamada and Nakano (1992)	yn01–yn04 *
FSP	$\mathcal{P}_{car}$	various	8	Carlier (1978)	car1–car8
	$\mathcal{P}_{hel}$	various	2	Heller (1960)	hel1,hel2
	$\mathcal{P}_{rec}$	various	21	Reeves (1995)	rec01-rec41 †

in practice, however, only the random instances used are reported in the OR-Library; for a total of 42 problem instances with processing times following a uniform distribution, $\mathbf{p} \sim \mathcal{U}(1, 100)$, of dimensions varying from 20×5 to 75×20.

*Best known solutions reported in Banharnsakun et al. (2012).

†Only odd-numbered instances are given, since the even-numbered instances are obtained from the previous instance by just reversing the processing order of each job; the optimal value of each odd-numbered instance and its even-numbered counterpart is the same.

Sentence first – verdict afterwards.

The Queen

4

Problem Structure

Problem structure and heuristic effectiveness are closely intertwined. When investigating the relation between the two, one can research what Corne and Reynolds (2010) call *footprints* in instance space, which is an indicator how an algorithm generalises over a given instance space. This sort of investigation has also been conducted by Pfahringer et al. (2000) under the alias *landmarking*. From experiments performed by Corne and Reynolds, it is evident that one-algorithm-for-all problem instances is not ideal, in accordance with no free lunch theorem (Wolpert and Macready, 1997). An algorithm may be favoured for its best overall performance, however it is rarely the best algorithm available over various subspaces of the instance space. Therefore, when comparing different algorithms one needs to explore how they perform w.r.t. the instance space, i.e., their footprint. That is to say, one can look at it as finding which footprints correspond to a subset of the instance space that works *well* for a given algorithm, and similarly finding which footprints correspond to a subset of the instance space that works *poorly* for a given algorithm.

In the context of job-shop this corresponds to finding *good* (makespan close to its opti-

mum) and *bad* (makespan far off its optimum) schedules. Note, good and bad schedules are interchangeably referred to as *easy* and *hard* schedules (pertaining to the manner they are achieved), respectively.

Smith-Miles and Lopes (2011) also investigate algorithm performance in instance space using footprints. The main difference between Corne and Reynolds and Smith-Miles and Lopes is how they discretise the instance space. In the case of Corne and Reynolds they use job-shop and discretise manually between different problem instances; on one hand w.r.t. processing times, e.g., $\mathbf{p} \sim \mathcal{U}(10, 20)$ versus $\mathbf{p} \sim \mathcal{U}(20, 30)$ etc., and on the other hand w.r.t. number of jobs, n. They warn that footprinting can be uneven, so great care needs to be taken in how to discretise the instance space into subspaces. On the other hand, Smith-Miles and Lopes use a completely automated approach. Using timetabling instances, they implement a self-organizing map to group similar problem instances together, that were both real world instances and synthetic ones using different problem generators.

Going back to the job-shop paradigm, then the interaction between processing time distribution and its permutation is extremely important, because it introduces hidden properties in the data structure making it *easy* or *hard* to schedule for the given algorithm. These underlying characteristics, i.e., features, define its data structure. A more sophisticated way of discretising the instance space is grouping together problem instances that show the same kind of feature behaviour, especially given the fact the learning models in Chapter 5 will be heavily based on feature pairs. Thereby making it possible to infer what sort of feature behaviour distinguishes between *good* and *bad* schedules.

In Ingimundardottir and Runarsson (2012), a single problem generator was used to create $N = 1,500$ synthetic 6×6 job-shop problem instances, where $\mathbf{p} \sim \mathcal{U}(1, 200)$ and $\boldsymbol{\sigma}$ was a random permutation. The experimental study showed that MWR works either well or poorly on a subset of the instances, in fact 18% and 16% of the instances were classified as *easy* and *hard* for MWR, respectively. Since the problem instances were naïvely generated, not to mention given the high variance of the data distribution, it is intuitive that there are some inherent structural qualities that could explain this difference in performance. The experimental study investigated the feature behaviours for these two subsets, namely, the easy and hard problem instances. For some features, the trend was more or less the same, which are explained by the common denominating factor, that all instances were sampled from the same problem generator. Whereas, those features that were highly correlated with the end-result, i.e., the final makespan, which determined if

an instance was labelled easy or hard, then the significant features varied greatly between the two difficulties, which imply the inherent difference in data structure. Moreover, the study in gives support to that random problem instance generators are *too* general and might not suit real-world applications. Watson et al. (2002) argue that problem instance generator should be more structured, since real-world manufacturing environment is not completely random, but rather structured, e.g., job's tasks can be correlated or machines in the shop. Watson et al. propose a problem instance generator that relates to real-world flow-shop attributes, albeit not directly modelled after real-world flow-shop due to the fact that deterministic $Fm||C_{\max}$ is seldom directly applicable in practice (Dudek et al., 1992). This is why $\mathcal{P}^{n\times m}_{f.jc}$, $\mathcal{P}^{n\times m}_{f.mc}$ and $\mathcal{P}^{n\times m}_{f.mxc}$ are also taken into consideration in Chapter 3 as they are an attempt to mimic the real-world characteristics of flow-shop.

It is interesting to know if the difference in the structure of the schedule is time dependent, e.g., is there a clear time of divergence within the scheduling process? Moreover, investigation of how sensitive is the difference between two sets of features, e.g., can two schedules with similar feature values yield completely contradictory outcomes (i.e. one poor and one good schedule)? Or will they more or less follow the their predicted trend? If the latter is the prevalent case, then these instances need to be segregated and each having their own learning algorithm implemented, for a meaningful outcome overall. This also, essentially, answers the question of whether it is in fact feasible to discriminate between *good* and *bad* schedules using the currently selected features as a measure. If results are contradictory, it is an indicator the features selected are not robust enough to capture the essence of the data structure and some key features are missing from the feature set that could be able to discriminate between *good* and *bad* schedules. Additionally, there is also the question of how can one define "similar" schedules, and what measures should be used? This chapter describes some preliminary experiments with the aim of investigating the feasibility of finding distinguishing features corresponding to *good* and *bad* schedules in job-shop. To summarise: (a) How to define problem difficulty? (b) Is there a time of divergence? (c) What are "similar" schedules? (d) Do similar features yield contradictory outcomes? (e) Are extra features needed? And (f) what can be learned from feature behaviour?

Instead of searching through a large set of algorithms and determining which algorithm is the most suitable for a given subset of the instance space, i.e., creating an algorithm portfolio, as is generally the focus in the current literature (Corne and Reynolds, 2010,

Smith-Miles and Lopes, 2011, Smith-Miles et al., 2009), the focus of the experimental study in Sections 4.12.1 to 4.12.7 (each corresponding to a given problem space from Chapter 3) is rather on few simple algorithms, here the SDRs described in Section 2.5, and understanding *how* they work on the instance space, similar to Watson et al. (2002), who analyse the fitness landscape of several problem classes for a fixed algorithm. Note, figures and tables that accompany this chapter are mostly located in Appendix A.

4.1 Distribution difficulty w.r.t. SDRs

Depending on the data distribution, dispatching rules perform differently. Take for instance the common single-based priority dispatching rules; SPT, LPT, LWR and MWR (cf. Section 2.5). A box-plot for deviation from optimality, ρ, defined by Eq. (2.14), for all problem spaces in Chapter 3 are depicted in **??**. As one can see, there is a staggering difference between the interaction of SDRs and their problem space. MWR is by far the best out of the four SDRs inspected for JSP – not only does it reach the known optimum most often but it also has the lowest worst-case factor from optimality. Similarly LWR for FSP. Although the same processing time distribution is used, there are some inherent structure in which MWR and LWR can exploit for JSP and FSP, respectively, whereas the other SDRs cannot. However, *all* of these dispatching rules are considered good and commonly used in practice and no one is better than the rest (Haupt, 1989), it simply depends on the data distribution at hand. This indicates that some distributions are harder than others, and these JSP problem generators simply favours MWR, whereas the FSP problem generators favours LWR.

4.2 Experimental settings

The main focus is on knowing *when* during the scheduling process easy and hard problems diverge and explore in further detail *why* they diverged. Rather than visualising high-dimensional data projected onto two dimensional space (as was the focus in Smith-Miles and Lopes (2011) with self-organising maps), instead appropriate statistical tests with a significance level $\alpha = 0.05$ is applied to determine if there is any difference between different data distributions. For this the two-sample Kolmogorov–Smirnov test (K-S test) is used to determine whether two underlying one-dimensional probability distributions differ. Furthermore, in order to find defining characteristics for easy or hard problems, a

(linear) correlation is computed between features to the resulting deviation from optimality, ρ.

Note, when inspecting any statistical difference between data distribution of the features on a step by step basis, the features at step $k+1$ are of course dependant on all previous k steps. This results in repetitive statistical testing, therefore a Bonferroni adjustment is used to counteract the multiple comparisons, i.e., each stepwise comparison has the significant level $\alpha_k = \frac{\alpha}{\ell}$, and thus maintaining the $\sum_{k=1}^{\ell} \alpha_k = \alpha$ significance level.

4.3 Defining 'easy' versus 'hard' schedules

It's relatively ad-hoc how to define what makes a schedule difficult. Intuitively, it's logical to use the schedule's objective to define it directly, i.e., inspecting deviation from optimality, ρ, defined by Eq. (2.14). Moreover, since the SDRs from Section 2.5 will be used throughout as a benchmark for subsequent models, the quantiles for deviation from optimality, ρ, using the SDRs on their training set will be used to differentiate between easy and hard scheduling problems. In particular, the classification is defined as follows,

Easy schedules belong to the first quantile, i.e.,

$$\mathcal{E}(a) := \{\mathbf{x} \mid \rho = Y(a, \mathbf{x}) < \rho^{\text{1st. Qu.}}\} \tag{4.1}$$

Hard schedules belong to the third quantile, i.e.,

$$\mathcal{H}(a) := \{\mathbf{x} \mid \rho = Y(a, \mathbf{x}) > \rho^{\text{3rd. Qu.}}\} \tag{4.2}$$

where $\mathbf{x} \in \mathcal{P}_{\text{train}}$ for a given dispatching rule $a \in \mathcal{A} := \{\text{SPT, LPT, LWR, MWR}\}$.

?? reports the first and third quantiles for each problem space, i.e., the cut-off values that determine the SDRs difficulty, whose division, defined as percentage of problem instances, i.e.,

$$\frac{|\mathcal{E}(a)|}{N_{\text{train}}} \cdot 100\% \quad \text{and} \quad \frac{|\mathcal{H}(a)|}{N_{\text{train}}} \cdot 100\% \tag{4.3}$$

for each $a \in \mathcal{A}$, are given in **????**, respectively.

4.4 Consistency of problem instances

The intersection of pairwise SDRs being simultaneously easy or hard are given in **????????**, i.e.,

$$\frac{|\mathcal{E}(a_i) \cap \mathcal{E}(a_j)|}{N_{\text{train}}} \cdot 100\% \quad \text{or} \quad \frac{|\mathcal{H}(a_i) \cap \mathcal{H}(a_j)|}{N_{\text{train}}} \cdot 100\% \tag{4.4}$$

where $a_i, a_j \in \mathcal{A}$. Note, when $a_i = a_j$ then Eq. (4.4) is equivalent to Eq. (4.3).

Even though this is a naïve way to inspect difference between varying SDRs, it's does give some initial insight of the potential of improving dispatching rules; a sanity check before going into extensive experiments, as will be done in Section 4.11.

For the corresponding 10×10 training set (cf. **????**), the intersections between SDRs from 6×5 (cf. **????**) seem to hold. However, by going to a higher dimension, the performance edge between SDRs becomes more apparent, e.g., in JSP when there was a slight possibility of LWR being simultaneously easy as other SDRs ($5\% <$ chance), it becomes almost impossible for 10×10. Making LWR a clear underdog. Despite that, for FSP the tables turn; now LWR has the performance edge. For instance, for $\mathcal{P}^{6\times5}_{f.rndn}$ the second best option is to apply LPT (13.22%), however there is a quite high overlap with LWR (11.74%), and since LWR is easier significantly more often (85.18%), the choice of SDR is quite obvious. Although, it goes to show that there is the possibility of improving LWR by sometimes applying LPT-based insight; by seeing what sets apart the intersection of their easy training sets.

Similarly for every 10×10 JSP (cf. **??**), almost all easy LPT schedules are also easy for MWR ($< 1\%$ difference), as is to be expected as MWR is the more sophisticated counterpart for LPT (like LWR is for SPT). However, the greedy approach here is not gaining any new information on how to improve MWR. In fact, MWR is never considered hard for any of the JSP (cf. **??**), therefore no intersection with any hard schedules. But the LPT counterpart has a relatively high occurrence rate (3-14%), so due to the similarity of the dispatching rules, the denominating factor between LPT and MWR can be an indicator for explaining some of MWR's pitfalls. That is to say, why aren't all of the job-shop schedules easy when applying MWR?

These have up until now all been speculations about how SDRs differ. One thing is for certain, the underlying problem space plays a great role on a SDR's success. Even slight variations to one job or machine, i.e., $\mathcal{P}^{10\times10}_{j.rnd,J_1}$ and $\mathcal{P}^{10\times10}_{j.rnd,M_1}$, shows significant change in performance. Due to the presence of bottleneck, MWR is able to detect it and thus

becomes the clear winner. Even outperforming the original $\mathcal{P}^{10\times10}_{j.rnd}$ which they're based on, despite having processing times doubled for a single job or machine, with approximately 10% lower first quantile (cf. **??**) in both cases.

As the objective of this dissertation is not to choose which DR is best to use for each problem instance. The focus is set on finding what characterises of job-shop overall, are of value (i.e. feature selection), and create a new model that works well for the problem space to a great degree. Namely, by exploiting feature behaviour that is considered more favourable. The hypothesis being that features evolutions of easy schedules greatly differ from features evolutions corresponding to hard schedules, and Section 4.11 will attempt to explain the evidence show in **???????????**.

Note, this section gave the definition of what constitutes an 'easy' and 'hard' schedule. Since these are based on four SDRs, $\mathcal{A}$, the training data for the experiments done in this chapter is based on $4N_{\text{train}}$ problem instances, per problem space, therefore,

$$\sum_{a\in\mathcal{A}}|\mathcal{E}(a)| \approx N_{\text{train}} \quad \text{and} \quad \sum_{a\in\mathcal{A}}|\mathcal{H}(a)| \approx N_{\text{train}} \tag{4.5}$$

due to the fact Eqs. (4.1) and (4.2) are based on the first and third quantiles of the entire training set. Now, as the SDRs vary greatly in performance, the contribution of a SDR to Eq. (4.5) varies, resulting in an unbalanced sample size when restricted to a single SDR.

4.5 Probability of choosing optimal decision

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 such 'good' behaviour. For this, we follow an optimal solution,* and inspect the evolution of its features (defined in Table 2.1) throughout the dispatching process. Moreover, it is noted, that there are 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_{train} independent instances which gives the training data variety.

Firstly, we can observe that on a step by step basis there are several optimal dispatches to choose from. **??** depicts how the number of optimal dispatches evolve at each dispatch iteration. Note, that only one optimal trajectory is pursued (chosen at random), hence

*Optimal solutions are obtained by using a commercial software package **?**.

this is only a lower bound of uniqueness of optimal solutions. As the number of possible dispatches decrease over time, **??** depicts the probability of choosing an optimal dispatch at each iteration.

4.6 Making suboptimal decisions

Looking at **??**, $\mathcal{P}_{j.rnd}^{10\times10}$ has a relatively high probability (70% and above) of choosing an optimal job. However, it is imperative to keep making optimal decisions, because once off the optimal track the consequences can be dire. To demonstrate this interaction **??** depicts the worst and best case scenario of deviation from optimality, ρ, once you've fallen off the optimal track. Note, that this is given that you make *one* wrong turn. Generally, there will be many mistakes made, and then the compound effects of making suboptimal decisions really start adding up.

It is interesting that for JSP, that over time making suboptimal decisions make more of an impact on the resulting makespan. This is most likely due to the fact that if a suboptimal decision is made in the early stages, then there is space to rectify the situation with the subsequent dispatches. However, if done at a later point in time, little is to be done as the damage is already been inflicted upon the schedule. However, for FSP, the case is the exact opposite. Under those circumstances it's imperative to make good decisions right from the beginning. This is due to the major structural differences between job-shop and flow-shop, namely the latter having a homogeneous machine ordering, constricting the solution immensely. Luckily, this does have the added benefit of making flow-shop less vulnerable for suboptimal decisions later in the decision process.

4.7 Optimality of simple priority dispatching rules

The probability of optimality of the aforementioned SDRs from Section 2.5, yet still maintaining our optimal trajectory, i.e., the probability of a job chosen by a SDR being able to yield an optimal makespan on a step by step basis, is depicted in **??**. Moreover, the dashed line represents the benchmark of randomly guessing the optimum (cf. **??**).

Now, let's bare in mind deviation from optimality, ρ, of applying SDRs throughout the dispatching process (cf. box-plots of which in **??**), then there is a some correspondence between high probability of stepwise optimality and low ρ. Alas, this isn't always the case, for $\mathcal{P}_{j.rnd}^{10\times10}$ SPT always outperforms LPT w.r.t. stepwise optimality, however this does not

transcend to SPT having a lower ρ value than LPT. Hence, it's not enough to just learn optimal behaviour, one needs to investigate what happens once we encounter suboptimal state spaces.

4.8 Simple blended dispatching rule

As stated before, the goal of this chapter is to utilise feature behaviour to motivate new, and *hopefully* better, dispatching rules. A naïve approach would be creating a simple blended dispatching rule which would be for instance switch between two SDRs at a predetermined time point. Hence, going back to **??** a presumably good BDR for $\mathcal{P}^{10\times10}_{j.rnd}$ would be starting with SPT and then switching over to MWR at around time step $k = 40$, where the SDRs change places in outperforming one another. A box-plot for deviation from optimality, ρ, for all 10×10 problem spaces is depicted in Fig. 4.1. This little manipulation between SDRs does outperform SPT immensely, yet doesn't manage to gain the performance edge of MWR, save for $\mathcal{P}^{10\times10}_{f.rnd}$. This gives us insight that for job-shop, the attribute based on MWR is quite fruitful for good dispatches, whereas the same cannot be said about SPT – a more sophisticated BDR is needed to improve upon MWR.

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 **??** shows, the inherent structure is already taking place, and might make it hard to come across optimal moves by simple methods. Therefore it's by no means guaranteed that by simply swapping over to MWR will handle the situation that applying SPT has already created. Figure 4.1 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, in the case of $\mathcal{P}^{10\times10}_{j.rnd}$ and $\mathcal{P}^{10\times10}_{j.rndn}$ the fact remains that the schedules have diverged too far from what MWR would have been able to achieve on its own. Preferably the blended dispatching rule should use best of both worlds, and outperform all of its inherited DRs, otherwise it goes without saying, one would simply still use the original DR that achieved the best results.

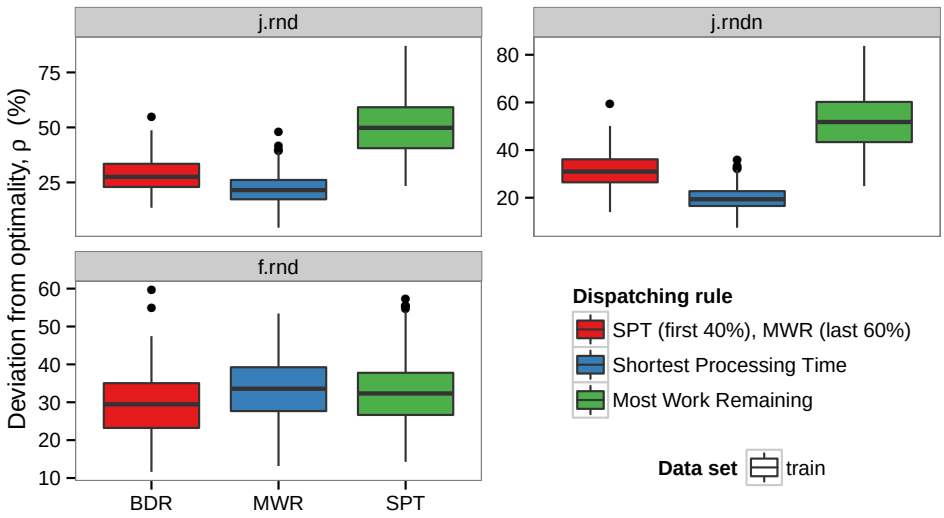


Figure 4.1: Box plot for deviation from optimality, ρ, for BDR where SPT is applied for the first 40% of the dispatches, followed by MWR.

4.9 Extremal feature

The SDRs we've inspected so-far are based on two features from Table 2.1, namely, (i) φ_1 for SPT and LPT, and (ii) φ_7 for LWR and MWR. By choosing the lowest value for the first SDR, and highest value for the latter SDR, i.e., the extremal values for those given features. Let's apply the same methodology from Section 4.7 to all varying features* described in Table 2.1. **??** depict the probability of all extremal features being an optimal dispatch, with random guessing from **??** as a dashed line.

Discuss more?

4.10 Feature evolution

In order to put the extremal features into perspective, it's worth comparing them with how the evolution of the features are over time, depicted in **??**.

*Note, φ_{18}, φ_9 and φ_{17} describe the features, not the schedule. For instance, φ_{18} gives us no new information, as that feature is homogeneous for each time step, making it equivalent to random guessing.

4.11 Emergence of problem difficulty

4.12 Structure of problem spaces

Up till now the discussion has been general and covering many problem spaces simultaneously. The subsequent sections will go into more depth what is going on per individual problem space.

4.12.1 JSP random

4.12.2 JSP random-narrow

4.12.3 FSP random

4.12.4 FSP random narrow

4.12.5 FSP job-correlated

4.12.6 FSP machine-correlated

4.12.7 FSP mixed-correlated

4.13 Discussion and Conclusion

Despite problem instances being created by the same problem generator, they vary among one another enough. As a result, all instances are not created equal; some are always hard to solve, others always easy. Since the description of the problem space isn't enough to predict its performance, we need a measure to understand what's going on. Why are some instances easier to find their optimum (or close enough)? That is to say, what's their secret? This is where their feature evolution comes into play. By using schedules obtained by applying SDRs we have the ability to get some insight into the matter.

From the experimental study it is apparent that features have different correlation with the resulting schedule depending in what stage it is in the scheduling process, implying that their influence varies over the dispatching sequencing. Moreover, features constant throughout the scheduling process are not correlated with the end-result. There are some common features for both difficulties considered which define job-shop on a whole. However the significant features are quite different across the two difficulties, implying there is

a clear difference in their data structure. The amount of significant features were considerably more for easy problems, indicating their key elements had been found. However, the features distinguishing hard problems were scarce. Most likely due to their more complex data structure their key features are of a more composite nature. As a result, new 'global' features were introduced.

It is possible for a JSP schedule to have more than one sequential dispatching representation. It is especially w.r.t. the initial dispatches. Visiting Fig. 2.1 again, if jobs $J_j \in \{J_1, J_2, J_6\}$ were to be dispatched first, then all permutations yield the same equivalent temporal schedule, this is because they don't create a conflict for one another (as is the case for jobs J_4 and J_5). This drawback of non-uniqueness of sequential dispatching representation explains why there is hardly any significant feature for the initial steps of the scheduling process (cf. **??** and **??**).

Since feature selection is of paramount importance in order for algorithms to become successful, one needs to give great thought to how features are selected. What kind of features yield *bad* schedules? And can they be steered onto the path of more promising feature characteristics. This sort of investigation can be an indicator how to create meaningful problem generators. On the account that real-world problem instances are scarce, their hidden properties need be drawn forth in order to generate artificial problem instances from the same data distribution.

The feature attributes need to be based on statistical or theoretical grounds. Scrutiny in understanding the nature of problem instances therefore becomes of paramount importance in feature engineering for learning, as it yields feedback into what features are important to devote more attention to, i.e., features that result in a failing algorithm. For instance, in **??** the slack features have the same distribution in the initial stages of the scheduling process, however there is a clear point of divergence which needs to be investigate why the sudden change? In general, this sort of investigation can undoubtedly be used in better algorithm design which is more equipped to deal with varying problem instances and tailor to individual problem instance's needs, i.e., a footprint-oriented algorithm.

Although this methodology was only implemented on a set of simple single-priority dispatching rules, the methodology is easily adaptable for more complex algorithms, such as the learned preference models in **??**. The main objective of this work is to illustrate the interaction of a specific algorithm on a given problem structure and its properties.

It was much pleasanter at home, when one wasn't always growing larger and smaller, and being ordered about by mice and rabbits.

Alice

5 Preference Learning of CDRs

Learning models considered in this dissertation 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 Runarsson (2006), and given in **??** for completeness.

5.1 Ordinal regression for job-shop

Let $\boldsymbol{\varphi}_o \in \mathcal{F}$ denote the post-decision state when dispatching J_o corresponds to an optimal schedule being built. All post-decisions states corresponding to suboptimal dispatches, J_s, are denoted by $\boldsymbol{\varphi}_s \in \mathcal{F}$. One could label which feature sets were considered optimal, $\mathbf{z}_o = \boldsymbol{\varphi}_o - \boldsymbol{\varphi}_s$, and suboptimal, $\mathbf{z}_s = \boldsymbol{\varphi}_s - \boldsymbol{\varphi}_o$ by $y_o = +1$ and $y_s = -1$ respectively. Note, a negative example is only created as long as J_s actually results in a worse makespan, i.e., $C_{\max}^{(s)} \gneq C_{\max}^{(o)}$, since there can exist situations in which more than one operation can be considered optimal.

The preference learning problem is specified by a set of preference pairs,

$$S := \left\{ \{\mathbf{z}_o, +1\}, \{\mathbf{z}_s, -1\} \mid \forall (o,s) \in \mathcal{O}^{(k)} \times \mathcal{S}^{(k)} \right\}_{k=1}^{\ell} \subset \Phi \times Y \tag{5.1}$$

where $\Phi \subset \mathcal{F}$ is the training set of d features, $Y = \{-1, +1\}$ is the outcome space, ℓ is the total number of optimal dispatches, $o \in \mathcal{O}^{(k)}$, and suboptimal dispatches $s \in \mathcal{S}^{(k)}$, at dispatch k. Note, $\mathcal{O}^{(k)} \cup \mathcal{S}^{(k)} = \mathcal{R}^{(k)}$, and $\mathcal{O}^{(k)} \cap \mathcal{S}^{(k)} = \emptyset$.

For job-shop there are $d = 23$ features (cf. the step-by-step varying features from Table 2.1), and the training set is created in the manner described in Section 5.3.

Which is being used, ordinal or logistic regression?

Logistic regression makes decisions regarding optimal dispatches and at the same time efficiently estimates a posteriori probabilities. When using linear classification model (cf. **??**), i.e.,

$$h(\mathbf{x}) = \sum_{i=1}^{d} w_i \varphi_i(\mathbf{x}) = \left\langle \mathbf{w} \cdot \boldsymbol{\varphi}(\mathbf{x}) \right\rangle, \tag{5.2}$$

the optimal $\mathbf{w}^*$ obtained from the preference set can be used on any new data point, $\boldsymbol{\varphi}$, and their inner product is proportional to probability estimate **??**. Hence, for each job on the ready-list, $J_j \in \mathcal{R}$, let $\boldsymbol{\varphi}_j$ 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.,

$$J_{j^*} = \underset{J_j \in \mathcal{R}}{\operatorname{argmax}} \; h(\boldsymbol{\varphi}_j) \tag{5.3}$$

where $h(\cdot)$ is the classification model obtained by the preference set, S, defined by Eq. (5.1).

5.2 Interpreting linear classification models

Looking at the features description in Table 2.1 it is possible for the ordinal regression to 'discover' the weights $\mathbf{w}$ in order for Eq. (5.2) corresponding applying a single priority

dispatching rules from Section 2.5. For instance,

$$\text{SPT:} \qquad w_i = \begin{cases} -1 & \text{if } i = 1 \\ 0 & \text{otherwise} \end{cases} \tag{5.4}$$

$$\text{LPT:} \qquad w_i = \begin{cases} 1 & \text{if } i = 1 \\ 0 & \text{otherwise} \end{cases} \tag{5.5}$$

$$\text{MWR:} \qquad w_i = \begin{cases} 1 & \text{if } i = 7 \\ 0 & \text{otherwise} \end{cases} \tag{5.6}$$

$$\text{LWR:} \qquad w_i = \begin{cases} -1 & \text{if } i = 7 \\ 0 & \text{otherwise} \end{cases} \tag{5.7}$$

where $i \in \{1, \ldots, d\}$. When using a feature space based on single priority dispatching rules, the linear classification models can very easily be interpreted as composite dispatching rules with predetermined weights.

5.3 Generating training data

For job-shop there are N_{train} problem instances generated using n jobs and m machines for processing times, $\mathbf{p}$, following the same data distribution and a random $\boldsymbol{\sigma}$ permutations, summarised in Table 2.1.

5.3.1 Job-shop tree representation

When building a complete job-shop schedule, $\ell = n \cdot m$ dispatches must be made sequentially. A job is placed at the earliest available time slot for its next machine, whilst still fulfilling constraints Eqs. (2.2) and (2.3). Unfinished jobs are dispatched one at a time according to some heuristic. After each dispatch* the schedule's current features (cf. Table 2.1) are updated based on the half-finished schedule. These collected features are denoted Φ, where,

$$\Phi := \bigcup_{i=1}^{N_{\text{train}}} \bigcup_{k=1}^{\ell} \bigcup_{J_j \in \mathcal{R}^{(k)}} \left\{ \boldsymbol{\varphi}_j \;\middle|\; (\mathbf{p}_i, \boldsymbol{\sigma}_i) \in \mathcal{P}^{n \times m} \right\}. \tag{5.8}$$

*The terms dispatch (iteration) and time step are used interchangeably.

Continuing with the example from Section 2.4, Fig. 5.1 shows how the first two dispatches could be executed for a six-job five-machine job-shop scheduling problem, with the machines, $a \in \{M_1, \ldots, M_5\}$, on the vertical axis and the horizontal axis yields the current makespan. The next possible dispatches are denoted as dashed boxes with the job index j within and its length corresponding to p_{ja}. In the top layer one can see an empty schedule. In the middle layer one of the possible dispatches from the layer above is fixed, and one can see the resulting schedule, i.e., what are the next possible dispatches given this scenario? Assuming J_4 would be dispatched first, the bottom layer depicts all the next possible partial schedules.

This sort of tree representation is similar to *game trees* (cf. **?**) where the root node denotes the initial (i.e. empty) schedule and the leaf nodes denote the complete schedule (resulting after $n \cdot m$ dispatches, thus height of the tree is ℓ), therefore the distance k from an internal node to the root yields the number of operations already dispatched. Traversing from root to leaf node one can obtain a sequence of dispatches that yielded the resulting schedule, i.e., the sequence indicates in which order the tasks should be dispatched for that particular schedule.

However one can easily see that this sequence of task assignments is by no means unique. Inspecting a partial schedule further along in the dispatching process such as in Fig. 2.1, 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. This is due to the fact these are non-conflicting jobs, which indicates that some of the nodes in game tree can merge. In the meantime the states of the schedule are different and thus their features, although they manage to yield with the same (partial) schedule at a later date. In this particular instance one can not infer that choosing J_1 is better and J_2 is worse (or vice versa) since they can both yield the same solution.

In some cases there can 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. In addition, varying permutations of the dispatching sequence (however given the same partial initial sequence) can result in very different complete schedules but can still achieve the same makespan, and thus same deviation from optimality, ρ, defined by Eq. (2.14) (which is the measure under consideration). Care must be taken in this case that neither resulting features are labelled as undesirable. Only the features from a dispatch yielding a truly suboptimal

solution should be labelled undesirable.

The creation of the game tree for job-shop can be done recursively for all possible permutation of dispatches, in the manner described above, resulting in a full n-ary tree (since $|\mathcal{R}| \leq n$) of height ℓ. Such an exhaustive search would yield at the most n^ℓ leaf nodes (worst case scenario being that no sub-trees merge). Now, since the internal vertices, i.e., partial schedules, are only of interest to learn,* the number of those can be at the most $^{n^\ell - 1}/_{n-1}$. Even for small dimensions of n and m the number of internal vertices are quite substantial and thus computationally expensive to investigate them all. Not to mention that this is done iteratively for all N_{train} problem instances.

5.3.2 Labelling schedules w.r.t. optimal decisions

The optimum makespan is known for each problem instance. At each time step (i.e. layer of the game tree) a number of feature pairs are created. They consist of the features $\boldsymbol{\varphi}_o$ resulting from optimal dispatches $o \in \mathcal{O}^{(k)}$, versus features $\boldsymbol{\varphi}_s$ resulting from suboptimal dispatches $s \in \mathcal{S}^{(k)}$ at time k. In particular, each job is compared against another job of the ready-list, $\mathcal{R}^{(k)}$, and if the makespan differs, i.e., $C_{\max}^{(s)} \gneq C_{\max}^{(o)}$, an optimal/suboptimal pair is created. However if the makespan would be unaltered, 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.

The approach taken here is to verify analytically, at each time step, by retaining 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 data generation takes into consideration when there are multiple optimal solutions to the same problem instance.

5.4 Time dependant dispatching rules

At each dispatch iteration k, a number of preference pairs are created, which is then repeated for all the N_{train} problem instances created. A separate data set is deliberately cre-

*The root is the empty initial schedule and for the last dispatch there is only one option left to choose from, so there is no preferred 'choice' to learn.

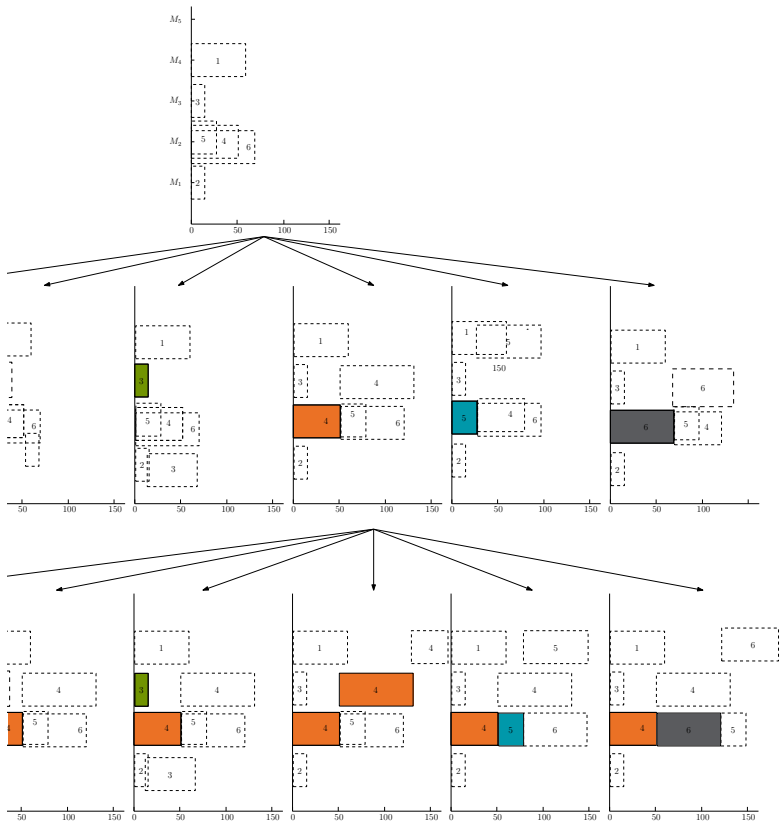


Figure 5.1: Partial Game Tree for job-shop for the first two dispatches. Top layer depicts all possible dispatches (dashed) for an empty schedule. Middle layer depicts all possible dispatches given that one of the dispatches from the layer above has been executed (solid). Bottom layer depicts when job J_4 on machine M_2 has been chosen to be dispatched from the previous layer, moreover it depicts all possible next dispatches from that scenario.

ated for each dispatch iterations, as the initial feeling is that dispatch rules used in the beginning of the schedule building process may not necessarily be the same as in the middle or end of the schedule. As a result there are ℓ linear scheduling rules for solving a $n \times m$ job-shop.

No longer the case, one model for all steps

5.5 Selecting preference pairs

Defining the size of the preference set as $l = |S|$, then Eq. (5.1) gives the size of the feature training set as $|\Phi| = \frac{1}{2}l$. If l is too large, than sampling needs to be done in order for the ordinal regression to be computationally feasible.

The strategy approached in Ingimundardottir and Runarsson (2011a) was to follow some optimal job $J_j \in \mathcal{O}^{(k)}$ (chosen at random), thus creating $|\mathcal{O}^{(k)}| \cdot |\mathcal{S}^{(k)}|$ feature pairs at each dispatch k, resulting in a training size of,

$$l = \sum_{i=1}^{N_{\text{train}}} \left(2|\mathcal{O}_i^{(k)}| \cdot |\mathcal{S}_i^{(k)}| \right) \tag{5.9}$$

For the problem spaces considered there, that sort of simple sampling of the state space was sufficient for a favourable outcome. However for a considerably harder problem spaces (see Chapter 4), preliminary experiments were not satisfactory.

A brute force approach was adopted to investigate the feasibility of finding optimal weights $\mathbf{w}$ for Eq. (5.2). By applying CMA-ES (discussed thoroughly in Chapter 6) to directly minimize the mean $C_{\max}$ w.r.t. the weights $\mathbf{w}$, gave a considerably more favourable result in predicting optimal versus suboptimal dispatching paths. So the question put forth is, why was the ordinal regression not able to detect it? The nature of the CMA-ES is to explore suboptimal routes until it converges to an optimal one. Implying that the previous approach of only looking into one optimal route is not sufficient information. Suggesting that the training set should incorporate a more complete knowledge about *all* possible preferences, i.e., make also the distinction between suboptimal and sub-suboptimal features, etc. This would require a Pareto ranking for the ready-list, $\mathcal{R}$, which can be used to make the distinction to which feature sets are equivalent, better or worse, and to what degree (i.e. giving a weight to the preference)? By doing so, the training set becomes much greater, which of course would again need to be sampled in order to be computationally

feasible to learn.

For instance Li and Olafsson (2005) used decision trees to 'rediscover' LPT by using the dispatching rule to create its training data. The limitations of using heuristics to label the training data is that the learning algorithm will mimic the original heuristic (both when it works poorly and well on the problem instances) and does not consider the real optimum. In order to learn new heuristics that can outperform existing heuristics then the training data needs to be correctly labelled. This drawback is confronted in (Malik et al., 2008, Olafsson and Li, 2010, Russell et al., 2009) by using an optimal scheduler, computed off-line.

These aspects are the main motivation for the data generation in this dissertation. All problem instances are correctly labelled w.r.t. their optimum makespan, found with analytical means.* In order to create training instances (and subsequently preference pairs) both a features resulting in optimal solutions are gathered (following optimal trajectories) and features that would have been chosen if a dispatching rule had been implemented (following DR trajectories). In the latter case, the trajectories pursued here, will be the SDRs from Section 2.5 as well as randomly dispatching operations.

To summarise, one needs to consider two main aspects of the generation of the training data, (a) what sort of rankings should be compared during each step? (b) Which path(s) should be investigated? Pursuing solely optimal trajectories? Creating random dispatches? Following other means: CMA-ES computed weights, single priority dispatching rules, etc.

5.5.1 Ranking strategies

The following ranking strategies were implemented for adding preference pairs to S defined by Eq. (5.1), they were first reported in **?**,

Basic ranking, S_b**,** i.e., all optimum rankings r_1 versus all possible suboptimum rankings r_i, $i \in \{2, \ldots, n'\}$, preference pairs are added – same basic set-up introduced in Ingimundardottir and Runarsson (2011a). Note, $|S_b|$ is defined in Eq. (5.9).

Full subsequent rankings, S_f**,** i.e., all possible combinations of r_i and r_{i+1} for $i \in \{1, \ldots, n'\}$, preference pairs are added.

*Optimal solution were found using **?**, a commercial software package for solving large-scale linear optimization and a state-of-the-art solver for mixed integer programming.

Partial subsequent rankings, S_p**,** i.e., sufficient set of combinations of r_i and r_{i+1} for $i \in \{1, \ldots, n'\}$, are added to the training set – e.g. in the cases that there are more than one operation with the same ranking, only one of that rank is needed to compared to the subsequent rank. Note that $S_p \subset S_f$.

All rankings, S_a**,** denotes that all aforementioned rankings were explored, i.e.,

$$S_a := S_b \cup S_f \cup S_p. \tag{5.10}$$

where $r_1 > r_2 > \ldots > r_{n'}$ $(n' \leq n)$ are the rankings of the ready-list, $\mathcal{R}^{(k)}$, at time step k.

5.5.2 Trajectory strategies

The following trajectory strategies were explored for adding preference pairs to S defined by Eq. (5.1),

SPT trajectory, S^{SPT}**,** at each dispatch the task corresponding to shortest processing time is dispatched, i.e., following the simple dispatching rule SPT.

LPT trajectory, S^{LPT}**,** at each dispatch the task corresponding to largest processing time is dispatched, i.e., following the simple dispatching rule LPT.

LWR trajectory, S^{LWR}**,** at each dispatch the task corresponding to least work remaining is dispatched, i.e., following the simple dispatching rule LWR.

MWR trajectory, S^{MWR}**,** at each dispatch the task corresponding to most work remaining is dispatched, i.e., following the simple dispatching rule MWR.

Optimum trajectory, S^{OPT}**,** at each dispatch some (random) optimal task is dispatched.

Random trajectory, S^{RND}**,** at each dispatch some random task is dispatched.

CMA-ES trajectory, S^{CMA}**,** at each dispatch the task corresponding to highest priority, computed with fixed weights $\mathbf{w}$, which were obtained by optimising the mean for deviation from optimality, ρ, defined by Eq. (2.14), with CMA-ES.

All trajectories, S^{ALL}**,** denotes all aforementioned trajectories were explored, i.e.,

$$S^{ALL} := S^{OPT} \cup S^{CMA} \cup S^{MWR} \cup S^{LWR} \cup S^{RND}. \tag{5.11}$$

In the case of the SDR-based and S^{CMA-ES} trajectories it is sufficient to explore each trajectory exactly once for each problem instance. Whereas, for S^{OPT} and S^{RND} there can be several trajectories worth exploring, however, only one is chosen (at random). It is noted that since the number of problem instances, N_{train}, is relatively large, it is deemed sufficient to explore one trajectory for each instance, in those cases as well.

These trajectory strategies were initially introduced in **?**, save for S^{CMA} and the SDR-based ones, however the latter are currently addressed since, e.g., LWR is considered more favourable for flow-shop rather than MWR (cf. Chapter 4).

5.5.3 Experimental study

To test the validity of different ranking and strategies from Section 5.5, a training set of N_{train} problem instances of 6×5 job-shop and flow-shop summarised in Table 3.1 (omitting the job and machine variations of job-shop). The size of the preference set, S, for different trajectory and ranking strategies are depicted in Fig. 5.2. Note, for now 10×10 problem spaces will be ignored, due to the extreme computational cost of correctly labelling each trajectory. However, in Section 5.6 an optimum path will be pursued for said dimension.

A linear preference (PREF) model was created for each preference set, S, for every problem space considered. A box-plot with deviation from optimality, ρ, defined by Eq. (2.14), is presented in **??**. From the figures it is apparent there can be a performance edge gained by implementing a particular ranking or trajectory strategy, moreover the behaviour is analogous across different disciplines.

Missing MISTA2013 figures

Ranking strategies

There is no statistical difference between S_f and S_p ranking-schemes across all disciplines (cf. **????**), which is expected since S_f is designed to contain the same preference information as S_f. However neither of the Pareto ranking-schemes outperform the original S_b set-up from Ingimundardottir and Runarsson (2011a). The results hold for the test set as well.

Combining the ranking schemes, S_{all}, improves the individual ranking-schemes across all disciplines, except in the case of $S_b^{opt}\big|_{\mathcal{P}_1}$ and $S_b^{rnd}\big|\mathcal{P}_2$, in which case there were no statis-

Figure 5.2: Size of preference set, l, for different trajectory strategies for given problem spaces, where $N_{\text{train}} = 500$.

tical difference. Now, for the test set, the results hold, however there is no statistical difference between S_b and S_{all} for most trajectories $\{S^{opt}, S^{cma}, S^{rnd}\}\big|\mathcal{P}_1$ and $\{S^{opt}, S^{rnd}\}\big|\mathcal{P}_2$. Now, whereas a smaller preference set is preferred, its opted to use the S^b ranking scheme henceforth.

Moreover, it is noted that the learning algorithm is able to significantly outperform the original heuristics, MWR and CMA-ES (white), used to create the training data S^{mwr} (grey) and S^{cma} (yellow), respectively (cf. **????**). For both $\mathcal{P}_1$ and $\mathcal{P}_2$, linear ordinal regression models based on S^{mwr} are significantly better than *MWR*, irrespective of the ranking schemes. Whereas the fixed weights found via CMA-ES are only outperformed by linear ordinal regression models based on $\{S^{cma}_b, S^{cma}_{all}\}$. This implies that ranking scheme needs to be selected appropriately. Result hold for the test data.

Trajectory strategies

Learning preference pairs from a good scheduling policies, such as S^{cma} and S^{mwr}, gave considerably more favourable results than tracking optimal paths (cf. **????**). Suboptimal routes are preferred when dealing with problem1 (for all ranking schemes), however when encountering problem2 the choice of ranking schemes can yield the exact opposite.

It is particularly interesting there is no statistical difference between S^{opt} and S^{rnd} for both $\{S_b, S_f\}\big|\mathcal{P}_1$ and $\{S_b, S_f, S_p\}\big|\mathcal{P}_2$ ranking-schemes. That is to say, tracking optimal dispatches gives the same performance as completely random dispatches. This indicates that exploring only optimal trajectories can result in a training set which the learning algorithm is inept to determine good dispatches in the circumstances when newly encountered features have diverged from the learned feature set labelled to optimum solutions.

Finally, S^{all} gave the best combination across all disciplines. Adding suboptimal trajectories with the optimal trajectory gives the learning algorithm a greater variety of preference pairs for getting out of local minima.

Following CMA-ES guided trajectory

The rational for using the S^{cma} strategy was mostly due to the fact a linear classifier is creating the training data (using the weights found via CMA-ES optimisation), hence the training data created should be linearly separable, which in turn should boost the training accuracy for a linear classification learning model. However, this strategy is easily out-

Figure 5.3: Linear weights for $\mathcal{P}_1$. Weights found via CMA-ES optimisation (red), and weights found via learning classification model based on S_b^{cma} (blue).

Figure 5.4: Linear weights for $\mathcal{P}_2$. Weights found via CMA-ES optimisation in red, and weights found via learning classification model based on S_b^{cma} in blue.

performed by the single priority based dispatching rule MWR guiding the training data collection, S^{mwr}.

Let's inspect the CMA-ES guided training data more closely, in particular the linear weights for Eq. (5.2). The weights are depicted in Figs. 5.3 and 5.4 for problem space problem1 and problem2, respectively. The original weights found via CMA-ES optimisation, that are used to guide the collection of training data, are depicted in red and weights obtained by the linear classification model for S_b^{cma} are depicted in blue.

5.5.4 Summary and conclusion

As the experimental results showed in Section 5.5.3, the ranking of optimal* and suboptimal features are of paramount importance. The subsequent rankings are not of much value, since they are disregarded anyway. However, the trajectories to create training instances have to be varied.

Unlike (Malik et al., 2008, Olafsson and Li, 2010, Russell et al., 2009), learning only on optimal training data was not fruitful. However, inspired by the original work by Li and Olafsson (2005), having DR guide the generation of training data (except correctly labelling with analytic means) gave meaningful preference pairs which the learning algorithm could learn. In conclusion, henceforth, the training data will be generate with S_b^{all} scheme.

5.6 Time independent dispatching rules

As stated in Section 5.5, a separate data set is deliberately created for each dispatch iteration, as it is initially assumed that dispatch rules used in the schedule building might

*Here the tasks labelled 'optimal' do not necessarily yield the optimum makespan (except in the case of following optimal trajectories), instead these are the optimal dispatches for the given partial schedule.

differ in the beginning, the middle or towards the end of the process. As a result there is a local linear model for each dispatch; a total of ℓ linear models for solving $n \times m$ job-shop. Now, if we were to create a global rule, then there would have to be one model for all dispatches iterations. The approach in Ingimundardottir and Runarsson (2011a) was to take the mean weight for all stepwise linear models, i.e., $\bar{w}_i = \frac{1}{\ell} \sum_{k=1}^{\ell} w_i^{(k)}$ where $\mathbf{w}^{(k)}$ is the linear weight resulting from learning preference set $S^{(k)}$ at dispatch k.

A more sophisticated way, would be to create a *new* linear model, where the preference set, S, is the union of the preference pairs across the ℓ dispatches. This would amount to a substantial training set, and for S to be computationally feasible to learn, S has to be filtered to size $l_{\max}$.

5.7 Exhaustive feature selection

5.7.1 Sampling strategies

Four different probability were implemented to sample S to size $l_{\max}$.

Equal, p^{equal} , all preferences are of equal probability.

Optimum, , p^{opt} , preferences were sampled proportional w.r.t. its stepwise optimality (cf. **??**).

Best case scenario, p^{bcs} , preferences were sampled reciprocally proportional w.r.t. its best case scenario of suboptimal dispatches (cf. **??** lower bounds).

Worst case scenario, , p^{wcs} , preferences were sampled reciprocally proportional w.r.t. its worst case scenario of suboptimal dispatches (cf. **??** upper bounds).

Where the latter three probabilities were sampled with replacement, whereas the first without replacement.

5.7.2 Experimental study

Given $N = 500$ problem instances of 6-jobs 5-machines job-shop, one can see from Fig. 5.5 that $l = |\zeta_k|$ can be quite great for small k. Hence, the size of the preference set ζ_k is limited to $l_{\max} \leq 3.5 \times 10^5$ via random sampling prior implementing the learning algorithm. The training accuracy for the linear ordinal regression model is depicted in Fig. 5.6, for problem

Figure 5.5: Size of preference set, l, for ζ_k defined by **??**, given $N_{\text{train}} = 500$ problem instances for problem spaces problem1 (blue) and problem2 (red).

Figure 5.6: Training accuracy for linear ordinal regression model trained on ζ_k defined by **??**, given $N_{\text{train}} = 500$ problem instances for problem spaces problem1 (blue) and problem2 (red).

space distributions problem1 and problem2, described in Table 3.1. Moreover, a box-plot for deviation from optimality, ρ, defined by Eq. (2.14), for problem spaces considered are depicted in Fig. 5.3. Note, figures utilize training set of size N_{train} in all cases. Main statistics are reported in Table 5.1.

Add S_b^{all} which uses a separate local model for each step k, is shown on the far left for comparison in Fig. 5.3 ?!?

Figure 5.3: Box-plot of results for linear ordinal regression model trained on ζ_k defined by **??**, given $N = 1,000$ problem instances for all considered 6×5 problem spaces.

Figure 5.4: Box-plot of results for global linear ordinal regression model, ζ_{14}, trained on problem spaces problem1 (blue) and problem2 (red). Note, the step-by-step S_b^{all} model for corresponding testing set is shown on the far left for comparison.

Table 5.1: Main statistics for deviation from optimality, ρ, using problem spaces $\mathcal{P}_1$, $\mathcal{P}_2$ and $\mathcal{P}_3$ using several scheduling policies

Table 5.2: Main statistics for deviation from optimality, ρ, for OR-Library job-shop benchmark problem instances using linear ordinal regression scheduling policies

In addition, the global linear regression schedule policies $\zeta_{14}(\mathcal{P}_1)$ and $\zeta_{14}(\mathcal{P}_2)$ were tested on synthetic FSP problems subclasses $\mathcal{P}^{6\times5}_{f.rnd}$, $\mathcal{P}^{6\times5}_{f.rndn}$,$\mathcal{P}^{6\times5}_{f.jc}$, $\mathcal{P}^{6\times5}_{f.mc}$ and $\mathcal{P}^{6\times5}_{f.mxc}$ generated by Watson et al. (2002) (cf. Section 3.2).

5.8 Discussion and conclusions

There's a large mustard-mine near here. And the moral of that is –
The more there is of mine, the less there is of yours.

The Duchess

6

Evolutionary Learning of CDRs

Chapter 6 Unfinished, taken from GECCO submission

Genetic algorithms (GA) are one of the most widely used approaches in JSP literature (Pinedo, 2008). However, in that case an extensive number of schedules need to be evaluated, and even for low dimensional JSP that can quickly become computationally infeasible. GAs can be used directly on schedules (Ak and Koc, 2012, Cheng et al., 1996, 1999, Meeran and Morshed, 2012, Qing-dao-er ji and Wang, 2012, Tsai et al., 2007), however, in that case there are many concerns that need to be dealt with. To begin with there are nine encoding schemes for representing the schedules Cheng et al. (1996), in addition there has to be special care when applying cross-over and mutation operators in order for the schedules, now in the role of 'chromosomes,' to still remain feasible. Moreover in case of JSP the GAs are not adapt for fine-tuning around optima, luckily a subsequent local search can mediate the optimisation (Cheng et al., 1999, Meeran and Morshed, 2012).

Another approach is to apply GAs indirectly to JSP, via dispatching rules, i.e., Dispatching Rules Based Genetic Algorithms (DRGA) (Dhingra and Chandna, 2010, Nguyen et al.,

2013, Vázquez-Rodríguez and Petrovic, 2009) where a solution is no longer a *proper* schedule but a *representation* of a schedule via applying certain dispatching rules consecutively. DRGA are a special case of *genetic programming* (Koza and Poli, 2005) which is the most predominant approach in hyper-heuristics is a framework of creating *new* heuristics from a set of predefined heuristics via GA optimisation (Burke et al., 2013).

A prevalent approach to solving JSP is to combine several relatively simple dispatching rules such that they may benefit each other for a given problem space. Generally, this is done on an ad-hoc basis, requiring expert knowledge from heuristics designer, or extensive exploration of suitable combinations of heuristics. The approach in this Chapter, is to automate that selection, by translating dispatching rules into measurable features and optimising what their contribution should be via evolutionary search. The framework is straight forward and easy to implement and shows promising results. Various data distributions from Chapter 3 are investigated, however only trained on the lower dimension, 6×5, yet, validated on higher dimension, 10×10.

Moreover, Section 6.2 shows that the choice of objective function for evolutionary search is worth investigating. Since the optimisation is based on minimising the expected mean of the fitness function over a large set of problem instances, which can vary within. Then normalising the objective function can stabilise the optimisation process away from local minima.

6.1 Introduction

As previously discussed in Chapter 1, there are two main viewpoints on how to approach scheduling problems, *a*) local level by building schedules for one problem instance at a time; and *b*) global level by building schedules for all problem instances at once. For local level construction a simple construction heuristic is applied, the schedule's features are collected at each dispatch iteration, from which a learning model will inspect the feature set to discriminate which operations are preferred to others via ordinal regression. The focus is essentially on creating a meaningful preference set composed of features and their ranks, as the learning algorithm is only run once to find suitable operators for the value function. This is the approach taken in Ingimundardottir and Runarsson (2011a). Expanding on that work, this study will explore global level construction viewpoint, where there is no feature set collected beforehand since the learning model is optimised directly

via evolutionary search. This requires numerous costly value function evaluations. In fact it involves an indirect method of evaluation whether one learning model is preferable to another, w.r.t. which one yields a better expected mean.

Inspired by DRGA, the approach taken in this study is to optimise the weights **w** in **??** directly, via evolutionary search such as covariance matrix adaptation evolution strategy (CMA-ES) Hansen and Ostermeier (2001), which has been proven to be a very efficient numerical optimisation technique.

Using standard set-up of parameters of the CMA-ES optimisation, the runtime was limited to 288 hours on a cluster for each 6×5 training set given in Sections 3.1 and 3.2, and in every case the optimisation reached its maximum walltime.

6.2 Performance measures

Generally, evolutionary search only needs to minimise the expected fitness value, however the approach in Ingimundardottir and Runarsson (2011a) was to use the known optimum to correctly label which operations' features were indeed optimal compared to other possible operations, then it would be of interest to inspect if there is any performance edge gained in incorporating optimal labelling in evolutionary search. Therefore, two objective functions will be considered, namely,

$$ES_{C_{\max}} := \min \mathbb{E}\left(\left[\right) C_{\max}\right] \tag{6.1}$$

for optimising w.r.t. $C_{\max}$ directly, and on the other hand

$$ES_{\rho} := \min \mathbb{E}\left(\left[\right) \rho\right] \tag{6.2}$$

which optimises w.r.t. the resulting $C_{\max}$ scaled to its true optimum, i.e., Eq. (2.14).

Main statistics of the experimental run are given in Table 6.1 and depicted in Fig. 6.2 for both approaches. In addition, evolving decision variables, here weights **w** for **??**, are depicted in Fig. 6.3.

In order to compare the two objective functions, the best weights reported were used for **??** on the corresponding training data. Its box-plot of percentage relative deviation from optimality, defined by Eq. (2.14), is depicted in Fig. 6.1 and main statistics detailed in Table 6.2.

Table 6.1: Final results for CMA-ES optimisation.

$\mathcal{P}$	minimise w.r.t. C_{max}			minimise w.r.t. ρ		
	#gen	#eval	$ES_{C_{max}}$	#gen	#eval	ES_{ρ}
f.jc	5984	65835	567.688	1625	17886	0.361
f.rnd	5088	55979	571.394	4546	50006	7.479
f.rndn	5557	61138	544.764	2701	29722	0.938
j.rnd	4707	51788	448.612	1944	21395	8.258
j.rndn	4802	52833	449.942	1974	21725	8.691

In the case of $\mathcal{P}^{6\times5}_{f.rndn}$, Eq. (6.2) gave a considerably worse results, since the optimisation got trapped in a local minimum, as the erratic evolution of the weighs in Section 6.2.1 suggest. For other problem spaces, Eq. (6.1) gave slightly better results than Eq. (6.2), however, there was no statical difference between adopting either objective function. Therefore, minimisation of expectation of ρ, is preferred over simply using the unscaled resulting makespan.

6.2.1 Problem difficulty

The evolution of fitness per generation from the CMA-ES optimisation of Eq. (6.2) is depicted in Fig. 6.2, and since all problem spaces reached their allotted computational time, without converging. In fact $\mathcal{P}^{6\times5}_{f.rnd}$ and $\mathcal{P}^{6\times5}_{j.rndn}$ needed restarting during the optimisation process. Furthermore, the evolution of the decision variables, $\mathbf{w}$, are depicted in Fig. 6.3. As one can see, the relative contribution for each weight clearly differs between problem spaces. Note that in the case of $\mathcal{P}^{6\times5}_{j.rndn}$ (cf. Section 6.2.1), CMA-ES restarts around generation 1,000 and quickly converges back to its previous fitness, however lateral relation of weights has completely changed. Implying that there are many optimal combinations of weights to be used, which can be expected due to the fact some features in Table 2.1 are a linear combination of one others, e.g., $\varphi_3 = \varphi_1 + \varphi_2$.

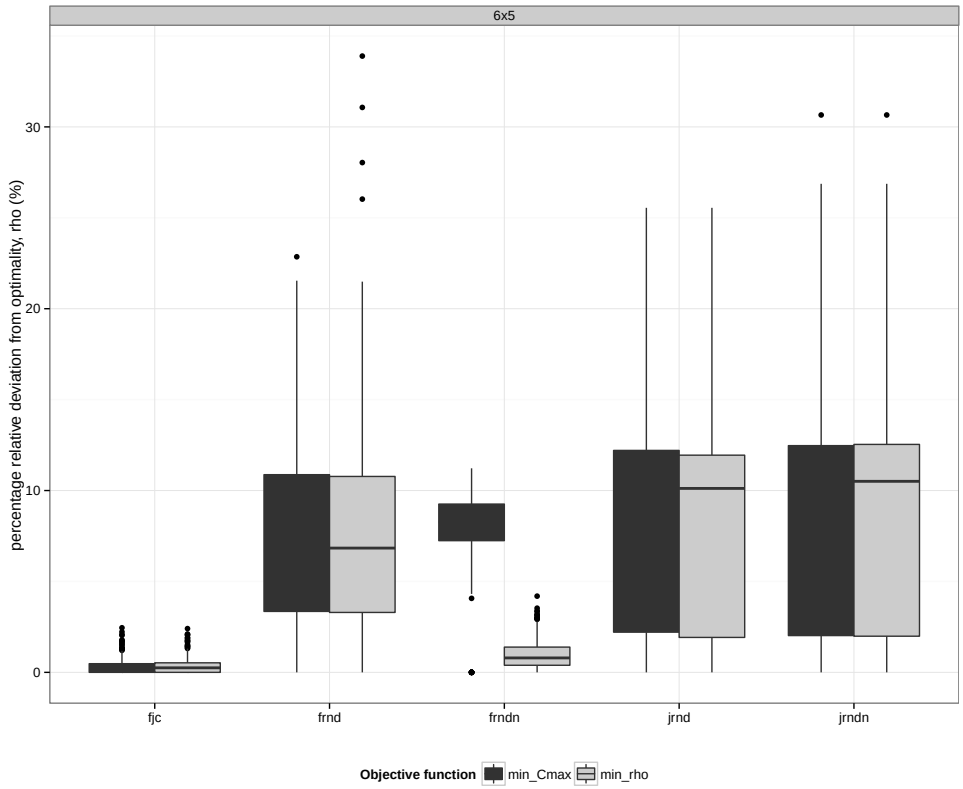


Figure 6.1: Box-plot of training data for percentage relative deviation from optimality, defined by Eq. (2.14), when implementing the final weights obtained from CMA-ES optimisation, using both objective functions from Eqs. (6.1) and (6.2), left and right, respectively.

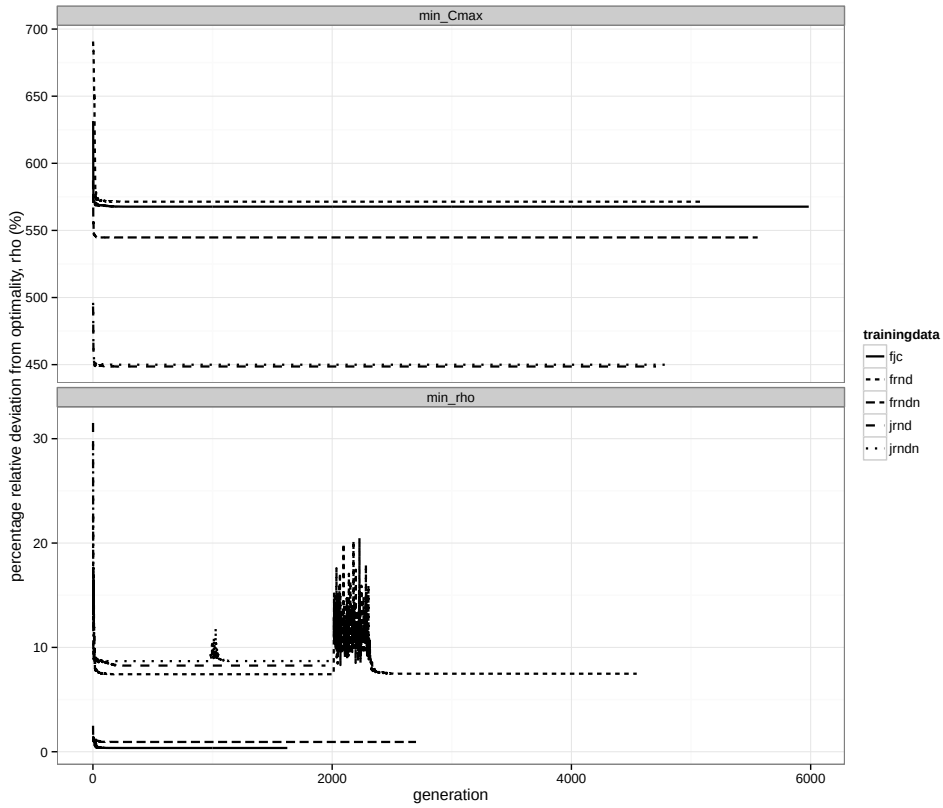
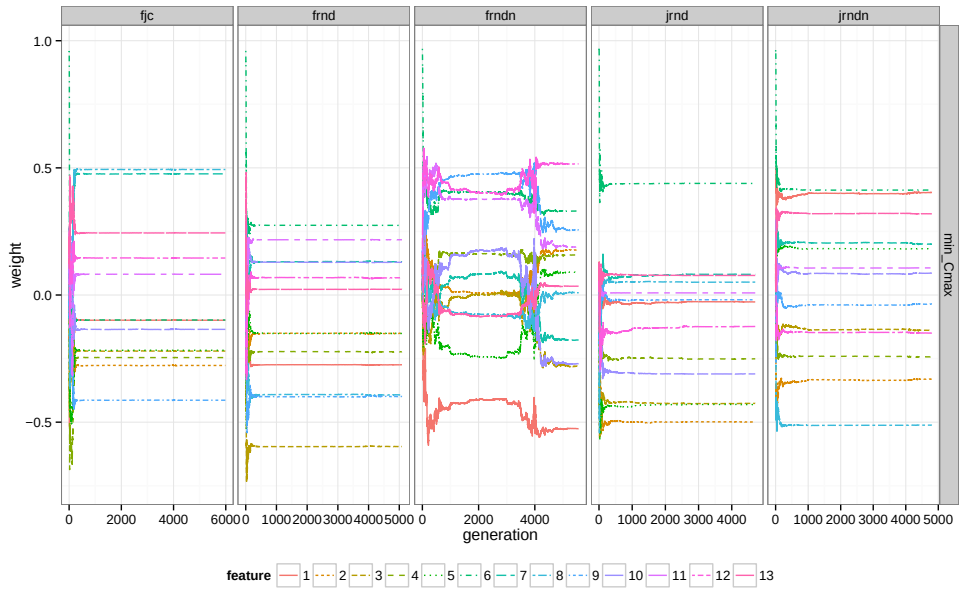
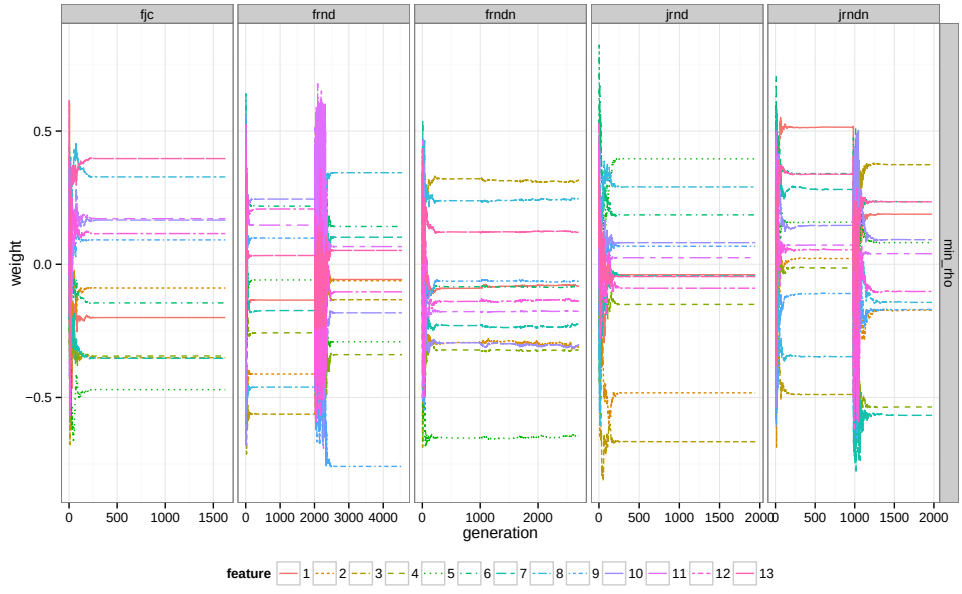


Figure 6.2: Fitness for optimising (w.r.t. Eqs. (6.1) and (6.2) above and below, receptively), per generation of the CMA-ES optimisation.



(**a**) minimise w.r.t. Eq. (6.1)



(**b**) minimise w.r.t. Eq. (6.2)

Figure 6.3: Evolution of weights of features (given in Table 2.1) at each generation of the CMA-ES optimisation. Note, weights are normalised such that $\|\mathbf{w}\| = 1$.

6.2.2 Robustness and scalability

As a benchmark, the linear ordinal regression model (PREF) from Ingimundardottir and Runarsson (2011a) was created. Using the weights obtained from optimising Eq. (6.2) and applying them on their 6×5 training data, their main statistics of Eq. (2.14) are reported in Table 6.2, for all training sets described in Table 3.1. Moreover, the best SDR, from which the features in Table 2.1 were inspired by, are also reported for comparison, i.e., most work remaining (MWR) for all JSP problem spaces, and least work remaining (LWR) for all FSP problem spaces.

To explore the scalability of the learning methods, a similar comparison to Section 6.2.2 is made for the applying the learning models on their corresponding 10×10 testing data, results are reported in Table 6.3. Note that only resulting $C_{\max}$ is reported, as the optimum makespan is not known.

6.3 Discussion and conclusions

Data distributions considered in this study either varied w.r.t. the processing times distributions, continuing the preliminary experiments in Ingimundardottir and Runarsson (2011a) , or w.r.t. the job ordering permutations, i.e., homogeneous σ matrices in FSP versus heterogeneous σ matrices in JSP. From the results based on 6×5 training data, given in Table 6.2, it's obvious that CMA-ES optimisation substantially outperforms the previous PREF methods from Ingimundardottir and Runarsson (2011a), for all problem spaces considered. Furthermore, the results hold when testing on 10×10, (cf. Table 6.3), suggesting the method is indeed scalable for higher dimensions.

Moreover, the study showed that the choice of objective function for evolutionary search is worth investigating. There was no statistical difference from minimising the fitness function directly and its normalisation w.r.t. true optimum (cf. Eqs. (6.1) and (6.2)), save for $\mathcal{P}^{6\times5}_{f.rndn}$. Implying, even though ES doesn't rely on optimal solutions, there are some problem spaces where it can be of great benefit. This is due to the fact that the problem instances can vary greatly within the same problem space Ingimundardottir and Runarsson (2012), thus normalising the objective function would help the evolutionary search to deviate the from giving too much weight for problematic problem instances for the greater good.

Table 6.2: Main statistics of percentage relative deviation from optimality, ρ, defined by Eq. (2.14) for various models, using corresponding 6×5 training data.

(a) $\mathcal{P}^{6\times5}_{j.rnd}$

model	mean	med	sd	min	max
$ES_{C_{max}}$	8.54	10	6	0	26
ES_{ρ}	8.26	10	6	0	26
PREF	10.18	11	7	0	30
MWR	16.48	16	9	0	45

(b) $\mathcal{P}^{6\times5}_{j.rndn}$

model	mean	med	sd	min	max
$ES_{C_{max}}$	8.68	11	6	0	31
ES_{ρ}	8.69	11	6	0	31
PREF	10.00	11	6	0	31
MWR	14.02	13	8	0	37

(c) $\mathcal{P}^{6\times5}_{f.rnd}$

model	mean	med	sd	min	max
$ES_{C_{max}}$	7.44	7	5	0	23
ES_{ρ}	7.48	7	5	0	34
PREF	9.87	9	7	0	38
LWR	20.05	19	10	0	71

(d) $\mathcal{P}^{6\times5}_{f.rndn}$

model	mean	med	sd	min	max
$ES_{C_{max}}$	8.09	8	2	0	11
ES_{ρ}	0.94	1	1	0	4
PREF	2.38	2	1	0	7
LWR	2.25	2	1	0	7

(e) $\mathcal{P}^{6\times5}_{f.jc}$

model	mean	med	sd	min	max
$ES_{C_{max}}$	0.33	0	0	0	2
ES_{ρ}	0.36	0	0	0	2
PREF	1.08	1	1	0	5
LWR	1.13	1	1	0	6

Table 6.3: Main statistics of $C_{\max}$ for various models, using corresponding 10×10 test data.

(a) $\mathcal{P}^{10\times10}_{j.rnd}$

model	mean	med	sd	min	max
$ES_{C_{\max}}$	922.51	914	73	741	1173
ES_{ρ}	931.37	931	71	735	1167
PREF	1011.38	1004	82	809	1281
MWR	997.01	992	81	800	1273

(b) $\mathcal{P}^{10\times10}_{j.rndn}$

model	mean	med	sd	min	max
$ES_{C_{\max}}$	855.85	857	50	719	1010
ES_{ρ}	855.91	856	51	719	1020
PREF	899.94	898	56	769	1130
MWR	897.39	898	56	765	1088

(c) $\mathcal{P}^{10\times10}_{f.rnd}$

model	mean	med	sd	min	max
$ES_{C_{\max}}$	1178.73	1176	80	976	1416
ES_{ρ}	1181.91	1179	80	984	1404
PREF	1215.20	1212	80	1006	1450
LWR	1284.41	1286	85	1042	1495

(d) $\mathcal{P}^{10\times10}_{f.rndn}$

model	mean	med	sd	min	max
$ES_{C_{\max}}$	1065.48	1059	32	992	1222
ES_{ρ}	980.11	980	8	957	1006
PREF	987.49	988	9	958	1011
LWR	986.94	987	9	959	1010

(e) $\mathcal{P}^{10\times10}_{f.jc}$

model	mean	med	sd	min	max
$ES_{C_{\max}}$	1135.44	1134	286	582	1681
ES_{ρ}	1135.47	1134	286	582	1681
PREF	1136.02	1135	286	582	1685
LWR	1136.49	1141	287	581	1690

The weights for **??** in Ingimundardottir and Runarsson (2011a) were found using supervised learning, where the training data was created from optimal solutions of randomly generated problem instances. As an alternative, this study showed that minimising the mean makespan directly using a brute force search via CMA-ES actually results in a better CDRs. The nature of CMA-ES is to explore suboptimal routes until it converges to an optimal one. Implying that the previous approach of only looking into one optimal route may not produce a sufficiently rich training set. That is, the training set should incorporate a more complete knowledge on *all* possible preferences, i.e., make also the distinction between suboptimal and sub-suboptimal features, etc. This would require a Pareto ranking of preferences which can be used to make the distinction to which feature sets are equivalent, better or worse – and to what degree, i.e., by giving a weight to the preference. This would result in a very large training set, which of course could be re-sampled in order to make it computationally feasible to learn.

The main drawback of using evolutionary search for learning optimal weights for **??** is how computationally expensive it is to evaluate the mean expected fitness. Even for a low problem dimension, 6-job 5-machine JSP, each optimisation run reached their walltime of 288hrs, without converging. Now, 6×5 JSP requires 30 sequential dispatches, where at each time step there are up to 6 jobs to choose from, i.e., its complexity is $\mathcal{O}(n^{n \cdot m})$, making it computationally infeasible to apply this framework for higher dimensions as is. However, evolutionary search only requires the rank of the candidates, and therefore it is appropriate to retain a sufficiently accurate surrogate for the value function during evolution in order to reduce the number of costly true value function evaluations, such as the approach in Ingimundardottir and Runarsson (2011b). This could reduce the computational cost of the evolutionary search considerably, making it feasible to conduct the experiments from Section 6.2 for problems of higher dimensions, e.g., with these adjustments it is possible to train on 10×10 and test on for example 14×14 to verify whether scalability holds for even higher dimensions.

The adventures first... explanations take such a dreadful time.

The Gryphon

7 Experiments

Chapter 7 Unfinished

There's something to be said for having a good opening line. Morbi commodo, ipsum sed pharetra gravida, orci $x = 1/a$ magna rhoncus neque, id pulvinar odio lorem non turpis. Nullam sit amet enim. Suspendisse id velit vitae ligula volutpat condimentum. Aliquam erat volutpat. Sed quis velit. Nulla facilisi. Nulla libero. Vivamus pharetra posuere sapien. Nam consectetuer. Sed aliquam, nunc eget euismod ullamcorper, lectus nunc ullamcorper orci, fermentum bibendum enim nibh eget ipsum. Donec porttitor ligula eu dolor. Maecenas vitae nulla consequat libero cursus venenatis. Nam magna enim, accumsan eu, blandit sed, blandit a, eros.

$$\zeta = \frac{1039}{\pi}$$

Tut, tut, child! Everything's got a moral, if only you can find it.

The Duchess

8 Conclusions

Chapter 8
Not started

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Morbi commodo, ipsum sed pharetra gravida, orci magna rhoncus neque, id pulvinar odio lorem non turpis. Nullam sit amet enim. Suspendisse id velit vitae ligula volutpat condimentum. Aliquam erat volutpat. Sed quis velit. Nulla facilisi. Nulla libero. Vivamus pharetra posuere sapien. Nam consectetuer. Sed aliquam, nunc eget euismod ullamcorper, lectus nunc ullamcorper

A cat may look at a king. I've read that in some book, but I don't remember where.

Alice

References

J. Adams, E. Balas, and D. Zawack. The shifting bottleneck procedure for job shop scheduling. *Management Science*, 34(3):391–401, 1988.

B. Ak and E. Koc. A Guide for Genetic Algorithm Based on Parallel Machine Scheduling and Flexible Job-Shop Scheduling. *Procedia - Social and Behavioral Sciences*, 62:817–823, Oct. 2012.

D. Applegate and W. Cook. A computational study of the job-shop scheduling instance. *ORSA Journal on Computing*, 3:149–156, 1991.

H. Asmuni, E. K. Burke, J. M. Garibaldi, B. McCollum, and A. J. Parkes. An investigation of fuzzy multiple heuristic orderings in the construction of university examination timetables. *Computers and Operations Research*, 36(4):981–1001, 2009.

A. Banharnsakun, B. Sirinaovakul, and T. Achalakul. Job shop scheduling with the best-so-far abc. *Engineering Applications of Artificial Intelligence*, 25(3):583–593, 2012.

J. E. Beasley. OR-Library: distributing test problems by electronic mail. *Journal of the Operational Research Society*, 41(11):1069–1072, 1990. URL `http://people.brunel.ac.uk/~mastjjb/jeb/info.html`.

E. Burke, S. Petrovic, and R. Qu. Case-based heuristic selection for timetabling problems. *Journal of Scheduling*, 9:115–132, 2006.

E. K. Burke, M. Gendreau, M. Hyde, G. Kendall, G. Ochoa, E. Ozcan, and R. Qu. Hyper-heuristics: a survey of the state of the art. *Journal of the Operational Research Society*, 64 (12):1695–1724, 2013.

J. Carlier. Ordonnancements a contraintes disjonctives. volume 12, pages 333–351. 1978.

F.-C. R. Chang. A study of due-date assignment rules with constrained tightness in a dynamic job shop. *Computers and Industrial Engineering*, 31(1–2):205–208, 1996.

T. Chen, C. Rajendran, and C.-W. Wu. Advanced dispatching rules for large-scale manufacturing systems. *The International Journal of Advanced Manufacturing Technology*, Feb. 2013.

R. Cheng, M. Gen, and Y. Tsujimura. A tutorial survey of job-shop scheduling problems using genetic algorithms—I. Representation. *Computers & Industrial Engineering*, 30 (4):983–997, 1996.

R. Cheng, M. Gen, and Y. Tsujimura. A tutorial survey of job-shop scheduling problems using genetic algorithms, part II: hybrid genetic search strategies. *Computers & Industrial Engineering*, 36(2):343–364, Apr. 1999.

D. Corne and A. Reynolds. Optimisation and generalisation: Footprints in instance space. In R. Schaefer, C. Cotta, J. Kolodziej, and G. Rudolph, editors, *Parallel Problem Solving from Nature, PPSN XI*, volume 6238 of *Lecture Notes in Computer Science*, pages 22–31. Springer, Berlin, Heidelberg, 2010.

E. Demirkol, S. Mehta, and R. Uzsoy. Benchmarks for shop scheduling problems. *European Journal of Operational Research*, 109(1):137–141, 1998.

A. Dhingra and P. Chandna. A bi-criteria M-machine SDST flow shop scheduling using modified heuristic genetic algorithm. *International Journal of Engineering, Science and Technology*, 2(5):216–225, 2010.

I. Drobouchevitch and V. Strusevich. Heuristics for the two-stage job shop scheduling problem with a bottleneck machine. *European Journal of Operational Research*, 123(2): 229 – 240, 2000. ISSN 0377-2217.

R. Dudek, S. Panwalkar, and M. Smith. The lessons of flowshop scheduling research. *Operations Research*, 40(1):7–13, 1992.

H. Fisher and G. Thompson. *Probabilistic learning combinations of local job-shop scheduling rules*, pages 225–251. Prenice-Hall, Englewood Cliffs, N.J., 1963.

J. Gao, M. Gen, L. Sun, and X. Zhao. A hybrid of genetic algorithm and bottleneck shifting for multiobjective flexible job shop scheduling problems. *Comput. Ind. Eng.*, 53(1): 149–162, Aug. 2007. ISSN 0360-8352.

M. R. Garey, D. S. Johnson, and R. Sethi. The complexity of flowshop and jobshop scheduling. *Mathematics of Operations Research*, 1(2):117–129, 1976.

C. P. Gomes and B. Selman. Algorithm portfolios. *Artificial Intelligence*, 126(1-2):43–62, Feb. 2001.

A. Guinet and M. Legrand. Reduction of job-shop problems to flow-shop problems with precedence constraints. *European Journal of Operational Research*, 109(1):96–110, 1998.

N. Hansen and A. Ostermeier. Completely derandomized self-adaptation in evolution strategies. *Evol. Comput.*, 9(2):159–195, June 2001.

R. Haupt. A survey of priority rule-based scheduling. *OR Spectrum*, 11:3–16, 1989.

J. Heller. Some numerical experiments for an $m \times j$ flow shop and its decision-theoretical aspects,. volume 8, pages 178–184. 1960.

N. B. Ho, J. C. Tay, and E. M. Lai. An effective architecture for learning and evolving flexible job-shop schedules. *European Journal Of Operational Research*, 179:316–333, 2007.

H. Ingimundardottir and T. P. Runarsson. Supervised learning linear priority dispatch rules for job-shop scheduling. In C. Coello, editor, *Learning and Intelligent Optimization*, volume 6683 of *Lecture Notes in Computer Science*, pages 263–277. Springer, Berlin, Heidelberg, 2011a.

H. Ingimundardottir and T. P. Runarsson. Sampling strategies in ordinal regression for surrogate assisted evolutionary optimization. In *Intelligent Systems Design and Applications (ISDA), 2011 11th International Conference on*, pages 1158–1163, nov. 2011b.

H. Ingimundardottir and T. P. Runarsson. Determining the characteristic of difficult job shop scheduling instances for a heuristic solution method. In Y. Hamadi and M. Schoenauer, editors, *Learning and Intelligent Optimization*, Lecture Notes in Computer Science, pages 408–412. Springer, Berlin, Heidelberg, 2012.

A. Jain and S. Meeran. Deterministic job-shop scheduling: Past, present and future. *European Journal of Operational Research*, 113(2):390–434, 1999.

M. Jayamohan and C. Rajendran. Development and analysis of cost-based dispatching rules for job shop scheduling. *European Journal of Operational Research*, 157(2):307–321, 2004.

Y. Jin. A comprehensive survey of fitness approximation in evolutionary computation. *Soft Computing - A Fusion of Foundations, Methodologies and Applications*, 9:3–12, 2005.

Y. Jin, M. Olhofer, and B. Sendhoff. A framework for evolutionary optimization with approximate fitness functions. *IEEE Transactions on Evolutionary Computation*, 6:481–494, 2002.

S. Kalyanakrishnan and P. Stone. Characterizing reinforcement learning methods through parameterized learning problems. *Machine Learning*, 84(1-2):205–247, June 2011.

M. G. Kendall. A new measure of rank correlation. *Biometrika*, 30(1/2):81–93, 1938.

P. Korytkowski, S. Rymaszewski, and T. Wiśniewski. Ant colony optimization for job shop scheduling using multi-attribute dispatching rules. *The International Journal of Advanced Manufacturing Technology*, Feb. 2013.

J. R. Koza and R. Poli. Genetic programming. In E. Burke and G. Kendal, editors, *Introductory Tutorials in Optimization and Decision Support Techniques*, chapter 5. Springer, 2005.

S. Lawrence. Resource constrained project scheduling: an experimental investigation of heuristic scheduling techniques (supplement). Technical report, Graduate School of Industrial Administration, Carnegie-Mellon University, Pittsburgh, Pennsylvania, 1984.

X. Li and S. Olafsson. Discovering dispatching rules using data mining. *Journal of Scheduling*, 8:515–527, 2005.

D. Lim, Y.-S. Ong, Y. Jin, and B. Sendhoff. A study on metamodeling techniques, ensembles, and multi-surrogates in evolutionary computation. In *Proceedings of the 9th annual conference on Genetic and evolutionary computation*, GECCO '07, pages 1288–1295, New York, NY, USA, 2007. ACM.

C.-J. Lin, R. C. Weng, and S. S. Keerthi. Trust region newton method for logistic regression. *J. Mach. Learn. Res.*, 9:627–650, June 2008.

I. Loshchilov, M. Schoenauer, and M. Sebag. Comparison-based optimizers need comparison-based surrogates. In *Proceedings of the 11th international conference on Parallel problem solving from nature: Part I*, PPSN'10, pages 364–373, Berlin, Heidelberg, 2010. Springer.

M.-S. Lu and R. Romanowski. Multicontextual dispatching rules for job shops with dynamic job arrival. *The International Journal of Advanced Manufacturing Technology*, Jan. 2013.

A. M. Malik, T. Russell, M. Chase, and P. Beek. Learning heuristics for basic block instruction scheduling. *Journal of Heuristics*, 14(6):549–569, Dec. 2008.

S. Meeran and M. Morshed. A hybrid genetic tabu search algorithm for solving job shop scheduling problems: a case study. *Journal of intelligent manufacturing*, 23(4):1063–1078, 2012.

B. L. Miller. *Noise, sampling, and efficient genetic algorithms*. PhD thesis, University of Illinois at Urbana-Champaign, Champaign, IL, USA, 1997.

L. Mönch, J. W. Fowler, and S. J. Mason. *Production Planning and Control for Semiconductor Wafer Fabrication Facilities*, volume 52 of *Operations Research/Computer Science Interfaces Series*, chapter 4. Springer, New York, 2013.

S. Nguyen, M. Zhang, M. Johnston, and K. C. Tan. Learning iterative dispatching rules for job shop scheduling with genetic programming. *The International Journal of Advanced Manufacturing Technology*, Feb. 2013.

S. Olafsson and X. Li. Learning effective new single machine dispatching rules from optimal scheduling data. *International Journal of Production Economics*, 128(1):118–126, 2010.

Y. S. Ong, P. B. Nair, A. J. Keane, and K. W. Wong. Surrogate-assisted evolutionary optimization frameworks for high-fidelity engineering design problems. In *In Knowledge Incorporation in Evolutionary Computation*, pages 307–332. Springer Verlag, 2004.

S. S. Panwalkar and W. Iskander. A survey of scheduling rules. *Operations Research*, 25(1): 45–61, 1977.

B. Pfahringer, H. Bensusan, and C. Giraud-carrier. Meta-learning by landmarking various learning algorithms. In *in Proceedings of the 17th International Conference on Machine Learning, ICML'2000*, pages 743–750. Morgan Kaufmann, 2000.

M. L. Pinedo. *Scheduling: Theory, Algorithms, and Systems*. Springer Publishing Company, Incorporated, 3 edition, 2008.

W. Ponweiser, T. Wagner, and M. Vincze. Clustered multiple generalized expected improvement: A novel infill sampling criterion for surrogate models. In *Evolutionary Computation, 2008. CEC 2008. (IEEE World Congress on Computational Intelligence). IEEE Congress on*, pages 3515–3522, june 2008.

R. Qing-dao-er ji and Y. Wang. A new hybrid genetic algorithm for job shop scheduling problem. *Computers & Operations Research*, 39(10):2291–2299, Oct. 2012.

A. Ratle. Optimal sampling strategies for learning a fitness model. In *Evolutionary Computation, 1999. CEC 99. Proceedings of the 1999 Congress on*, volume 3, pages 2078–2085, 1999.

C. Reeves. A genetic algorithm for flowshop sequencing. *Computer Operations Research*, 22:5–13, 1995.

J. R. Rice. The algorithm selection problem. *Advances in Computers*, 15:65–118, 1976.

T. Runarsson. Constrained evolutionary optimization by approximate ranking and surrogate models. In X. Yao, E. Burke, J. Lozano, J. Smith, J. Merelo-Guervós, J. Bullinaria, J. Rowe, P. Tino, A. Kabán, and H.-P. Schwefel, editors, *Parallel Problem Solving from Nature - PPSN VIII*, volume 3242 of *Lecture Notes in Computer Science*, pages 401–410. Springer, Berlin, Heidelberg, 2004.

T. Runarsson. Ordinal regression in evolutionary computation. In T. Runarsson, H.-G. Beyer, E. Burke, J. Merelo-Guervós, L. Whitley, and X. Yao, editors, *Parallel Problem Solving from Nature - PPSN IX*, volume 4193 of *Lecture Notes in Computer Science*, pages 1048–1057. Springer, Berlin, Heidelberg, 2006.

T. Russell, A. M. Malik, M. Chase, and P. van Beek. Learning heuristics for the superblock instruction scheduling problem. *IEEE Trans. on Knowl. and Data Eng.*, 21(10):1489–1502, Oct. 2009.

M. J. Sasena, P. Papalambros, and P. Goovaerts. Exploration of metamodeling sampling criteria for constrained global optimization. *Engineering Optimization*, 34(3):263–278, 2002.

H. Schwefel, editor. *Evolution and optimum seeking*. A Wiley-Interscience publication. Wiley, New Jersey, 1995.

K. Smith-Miles and L. Lopes. Generalising algorithm performance in instance space: A timetabling case study. In C. Coello, editor, *Learning and Intelligent Optimization*, volume 6683 of *Lecture Notes in Computer Science*, pages 524–538. Springer, Berlin, Heidelberg, 2011.

K. Smith-Miles, R. James, J. Giffin, and Y. Tu. A knowledge discovery approach to understanding relationships between scheduling problem structure and heuristic performance. In T. Stützle, editor, *Learning and Intelligent Optimization*, volume 5851 of *Lecture Notes in Computer Science*, pages 89–103. Springer, Berlin, Heidelberg, 2009.

A. Sóbester, S. J. Leary, and A. J. Keane. On the design of optimization strategies based on global response surface approximation models. *J. of Global Optimization*, 33(1):31–59, Sept. 2005.

R. H. Storer, S. D. Wu, and R. Vaccari. New search spaces for sequencing problems with application to job shop scheduling. *Management Science*, 38(10):1495–1509, 1992.

É. D. Taillard. Benchmarks for basic scheduling problems. *European Journal of Operational Research*, pages 1–17, 1993.

J. C. Tay and N. B. Ho. Evolving dispatching rules using genetic programming for solving multi-objective flexible job-shop problems. *Computers and Industrial Engineering*, 54 (3):453–473, 2008.

S. Thiagarajan and C. Rajendran. Scheduling in dynamic assembly job-shops to minimize the sum of weighted earliness, weighted tardiness and weighted flowtime of jobs. *Computers and Industrial Engineering*, 49(4):463–503, 2005.

J.-T. Tsai, T.-K. Liu, W.-H. Ho, and J.-H. Chou. An improved genetic algorithm for job-shop scheduling problems using Taguchi-based crossover. *The International Journal of Advanced Manufacturing Technology*, 38(9-10):987–994, Aug. 2007.

J. A. Vázquez-Rodríguez and S. Petrovic. A new dispatching rule based genetic algorithm for the multi-objective job shop problem. *Journal of Heuristics*, 16(6):771–793, Dec. 2009.

R. Vilalta and Y. Drissi. A perspective view and survey of meta-learning. *Artificial Intelligence Review*, 2002.

J.-P. Watson, L. Barbulescu, L. D. Whitley, and A. E. Howe. Contrasting structured and random permutation flow-shop scheduling problems: Search-space topology and algorithm performance. *INFORMS Journal on Computing*, 14:98–123, 2002.

D. H. Wolpert and W. G. Macready. No free lunch theorems for optimization. *IEEE Transactions on Evolutionary Computation*, 1(1):67–82, 1997.

L. Xu, F. Hutter, H. Hoos, and K. Leyton-Brown. SATzilla-07: The design and analysis of an algorithm portfolio for SAT. *Principles and Practice of ...*, 2007.

T. Yamada and R. Nakano. A genetic algorithm applicable to large-scale job-shop instances. In B. M. R. Manner, editor, *Parallel Problem Solving from Nature - PPSN II*, pages 281–290. Elsevier, 1992.

J.-M. Yu, H.-H. Doh, J.-S. Kim, Y.-J. Kwon, D.-H. Lee, and S.-H. Nam. Input sequencing and scheduling for a reconfigurable manufacturing system with a limited number of fixtures. *The International Journal of Advanced Manufacturing Technology*, Jan. 2013.

W. Zhang and T. G. Dietterich. A reinforcement learning approach to job-shop scheduling. In *Proceedings of the 14th international joint conference on Artificial Intelligence*, volume 2 of *IJCAI'95*, pages 1114–1120, San Francisco, CA, USA, 1995. Morgan Kaufmann Publishers Inc.

f

I can't explain myself, I'm afraid, Sir, because I'm not myself you see.

Alice

A
Problem structure

Figures and tables within this appendix pertain to Chapter 4. **????????** depict the mean over all the training data, which are quite noisy functions.* Thus, for clarity purposes, they are fitted with local polynomial regression, making the boundary points sometimes biased. However, **????** depict the mean as is.

*? depicts the mean as is, albeit only for 10 × 10 problem spaces.