Todo list

This page is intentionally left blank.

Listing of Publications

This dissertation is based on the following publications, listed in chronological order:

Paper I Supervised Learning Linear Priority Dispatch Rules for Job-Shop Scheduling

Paper II Sampling Strategies in Ordinal Regression for Surrogate Assisted Evolutionary Optimization

Paper III Determining the Characteristic of Difficult Job Shop Scheduling Instances for a Heuristic Solution Method

Paper IV Evolutionary Learning of Weighted Linear Composite Dispatching Rules for Scheduling

Paper V Generating Training Data for Learning Linear Composite Dispatching Rules for Scheduling

Paper VI Supervised Learning Linear Composite Dispatch Rules for Scheduling

These publications will be referenced throughout using their Roman numeral. The thesis is divided into two parts: *Prologue*, and *Papers*. Prologue gives a coherent connection for the publications, and elaborates on chosen aspects. Whereas, Papers contains copies of the publications reprinted with permission from the publishers.

Table 1: Summary of experimental designs in Part II.

Paper	Problem	Model	Model parameters	|Model|*
I	JSP	PREF	$\Phi^{\text{OPT}}, \Psi_b$	K
II	$\mathbb{R}$-functions	CMA-ES	surrogate sampling strategies	1
III	JSP	SDR	MWR	1
IV	JSP, FSP	CMA-ES	ES.$C_{\max}$, ES.ρ	1
V	JSP	PREF	$\{\Phi^{\pi} : \pi \in \{\text{OPT}, \text{SDR}, \text{ALL}\}\}$ $\{\Psi_r : r \in \{a, b, f, p\}\}$	K
VI	JSP, FSP	PREF	$\{\Phi^{\pi} : \pi \in \{\text{OPT}, \text{OPT}\varepsilon, \text{DA}i\}\}$ Ψ_p	1

*Models are either stepwise (i.e. total of K models) or fixed throughout the dispatching process.

Mapping between Part I and Part II

The prologue will be addressing the job-shop scheduling problem, detailed in Chapter 2 and correspond to the application in Papers I and III to VI. The problem generators used are subsequently described in Chapter 3. From there, we try do define problem difficulty in Chapter 4, improving upon the ad-hoc definition from Paper III. There will be two algorithms considered: *i)* preference learning in Chapter 8, which is a tailored algorithm, and *ii)* evolutionary search in Chapter 5, which is a general algorithm.

The latter was implemented in Paper IV, which could be improved by incorporating the methodology from Paper II. Preference models on the other hand, are highly dependent on training data, whose collection is addressed in Chapter 6 using passive imitation learning, whereas Paper V included active imitation learning with greatly improved results. Moreover, the training data contains an abundance of information that can be used to determine algorithm's footprint in instance space, which was done for optimal solutions in Paper VI, and in addition to that SDR based trajectories were inspected in Chapter 7 along with tying together the preliminary work in Paper III. Furthermore, Chapter 10 compares to two methodologies, as the preference models had been significantly improved since Paper IV. An overview of experimental settings in Part II is given in Table 1. Finally, the dissertation concludes in Chapter 11 with discussion and addresses future work.

Part I

Prologue

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

The King

1 Introduction

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 problem class (such as scheduling) it is possible to construct problem instances where one heuristic would outperform another.

Each heuristic performs distinctly to others depending on the underlying data distribution of the problem. Because 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 those instances.

Given the ad-hoc nature of the heuristic design process, there is clearly room for improvement. A number of attempts have been made to automate heuristic design, and it is the ultimate goal of this dissertation to automate optimisation heuristics via ordinal regression. The focal point will be based on scheduling processes named job-shop scheduling problem (JSP), and one of its subclasses, the flow-shop scheduling problem (FSP).

There are two main viewpoints on how to approach scheduling problems, namely,

Tailored algorithms or constructive methods,
by building schedules for one problem instance at a time.

General algorithms or iterative methods,
by building schedules for all problem instances at once.

For tailored algorithm construction: *i)* a simple construction heuristic is applied; *ii)* the schedule's features are collected at each dispatch iteration, and *iii)* 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 general algorithm construction, 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 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 can be used for surrogate assisted evolutionary optimisation, where models are ranked – whereas for tailored algorithms, features were ranked.

1.1 Rice's framework for algorithm selection

The aim of this dissertation is to understand what underlying characteristics of the problem instances distinguish *good* and *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 scheduling. Moreover, Smith-Miles and Lopes focused on how varying algorithms perform on different data distributions. Hence, the investigation of heuristic efficiency is closely intertwined with problem generation. The relation between problem structure and heuristic efficiency, called *footprints in instance space*, will be addressed in Chapters 4 and 7. In order to formulate the relationship for footprints, one can utilise Rice's framework for algorithm selection problem from 1976. The framework consists of four fundamental components:

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 $\boldsymbol{\varphi}(\mathbf{x}) = \left[\varphi_1(\mathbf{x}), ..., \varphi_d(\mathbf{x})\right]^T \in \mathcal{F}$ and using algorithm $a \in \mathcal{A}$ the performance is $y = \Upsilon(a, \boldsymbol{\varphi}(\mathbf{x})) \in \mathcal{Y}$, where $\Upsilon : \mathcal{A} \times \mathcal{F} \mapsto \mathcal{Y}$ 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:

Tailored algorithms

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

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.

General algorithms

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

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

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

A schematic flow-chart of the model selection process is illustrated in Fig. 1.1. Meta-data is analysed to investigate problem structure and heuristic effectiveness, i.e., its footprint. Moreover, the schematic details how the preference model, which is a tailored algorithm, from Chapter 8 will come into play in the framework.

1.2 Previous work

The literature in scheduling mainly focuses on different objectives, e.g., Chang (1996) minimised the due-date tightness and Drobouchevitch and Strusevich (2000), Gao et al.

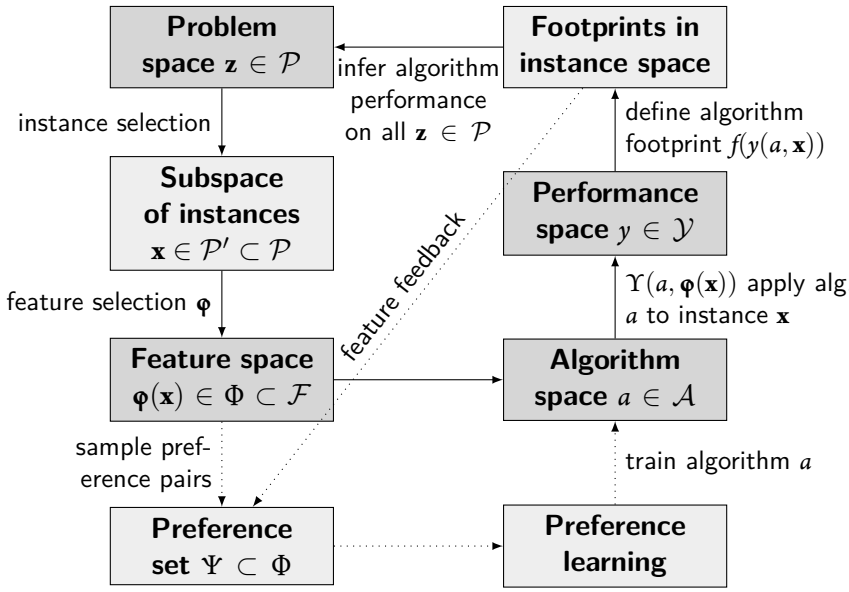


Figure 1.1: Flow-chart for Rice's framework for algorithm selection

(2007) looked into solving for bottleneck machines, or even multi-objective JSP (Tay and Ho, 2008, Vázquez-Rodríguez and Petrovic, 2009, Xia and Wu, 2005). In this dissertation only minimisation of the makespan will be considered, thus ignoring all due-date constraints. Model assumptions (i.e. shop floor constraints) can also vary, e.g., Thiagarajan and Rajendran (2005) incorporate different earliness, tardiness and holding costs. Brandimarte (1993), Pezzella et al. (2008), Xia and Wu (2005) extend the classical JSP set-up, called *flexible* job-shop, by allowing tasks to be processed by any machine from a given set, i.e., adding assignment of operations to the constraints. Moreover, it is possible to reduce JSP to a FSP, since in practice, most jobs in the job-shop use the machines in the same order (Guinet and Legrand, 1998, Ho et al., 2007). Formal mathematical model for JSP is given in Chapter 2.

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, JSP has the reputation of being notoriously difficult to solve. As a result, it's been widely

*NP stands for Non-deterministic Polynomial-time. If P≠NP, then NP-hard problems cannot be solved by a deterministic Turing machine in polynomial time.

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. Figure 1.2 summarise the main techniques applied to solve JSP. The figure is based on Fig. 1 from Jain and Meeran (1999), however, updated to reflect the previous work relevant to this dissertation.

In the field of Artificial Intelligence, Meeran and Morshed (2012) point out that 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 involving 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. Unfortunately, hybridisation of global and local methodologies is non-trivial. In general, combination of the two improves performance. Unfortunately, they often come at a great computational cost.

Various *learning* approaches have been applied to solving job-shop scheduling, such as: *i)* reinforcement learning (Zhang and Dietterich, 1995); *ii)* evolutionary learning (Tay and Ho, 2008), and *iii)* 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 JSP is applying construction heuristics with some priority dispatching rule (DR), e.g., choosing a task corresponding to: *i)* longest or shortest processing time; *ii)* most or least successors (i.e. operation number), or *iii)* ranked positional weight, i.e., sum of processing times of its predecessors or successors. 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 their combinations require expert knowledge or extensive trial-and-error process.

DRs are a very useful approach to dealing with scheduling environments because they are quick to implement (by computers and shop floor operators) and can cope with dynamic changes. Furthermore, DRs are relatively easy to interpret which can be of paramount importance for some end-users. For instance, Keane (2015) used Genetic Programming (GP) to creates features for Case Based Reasoning, which were hard do

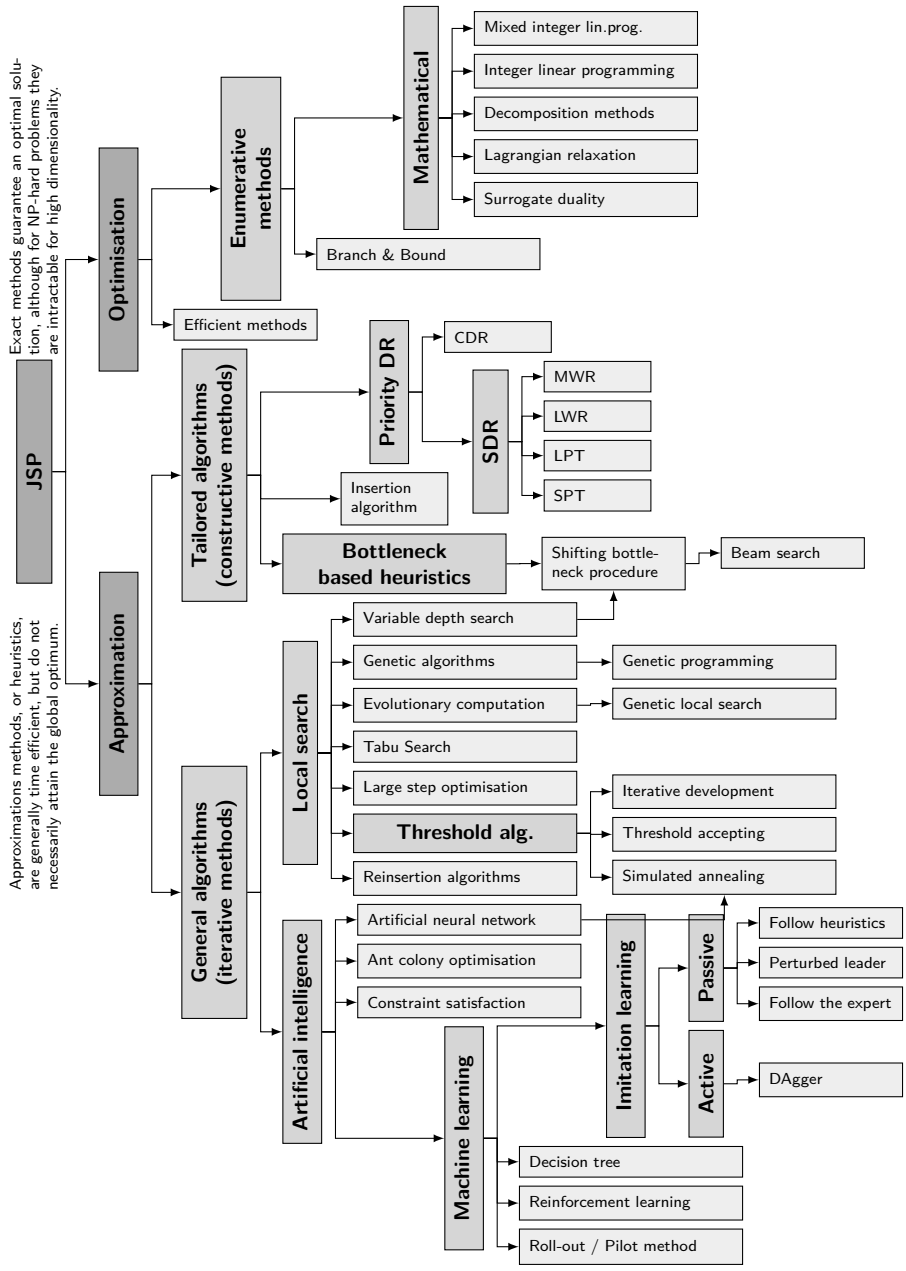


Figure 1.2: Various methods for solving JSP (based on Fig. 1 from Jain and Meeran, 1999)

understand and cumbersome in implementation due to their complexity. In order to mediate the process, the *Espresso Algorithm* from logic circuit design was used for feature selection, as 'espresso' summarises the evolved features obtained by GP, yielding a much simpler form that is more comprehensible for the end-user. The motivation for easily interpretable models, is particularly appealing, even necessary in some cases. Because in some paradigms they become essential for getting them sanctioned, e.g., due to legislation for implementation of uninhabited aerial vehicles (i.e. drones).

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 offline. The optimal solutions are used as training data and a decision tree learning algorithm is 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, where training data is guided by the two best heuristics in the field. Burke et al. point out that in order for their framework to be successful, problem features need to be sufficiently explanatory and training data needs 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 initial goal of the Ph.D. project was to use sophisticated algorithms for preference learning on hard problems, in particular job-shop scheduling, and find ways to mediate the computational effort that they require. After painstaking parameter tuning, and managing to find complex models with high training accuracy. Alas, severely overfitted to the training instances – a simple linear model would suffice with similar performance, and for much less overhead! Also, linear models come with the added benefit of easy interpretability.

Unfortunately, there is not much said about algorithms that fail (Smith-Miles and Bowly, 2015), as the focus tend to be on claiming superiority in performance to some previous approach. So to quote a pioneer in scheduling,

> "The only real mistake is the one from which we learn nothing"
>
> Henry Ford

In order to make the best of a bad situation, this derailment* designed the course of the body of work presented in this dissertation, which is divided into two main phases: *i)* analysis, and *ii)* preference learning based on the analysis.

Analysis

By dwelling on optimal solutions and try to understand their fundamental building blocks, and applying what you learn on *simple* models, before investing valuable time and resources in implementing the current state-of-the-art algorithms. The research questions that are put forth are: *i)* how are optimal solutions *supposed* to behave – what are the key indicators? *ii)* Where and when should there be emphasis on learning? And ultimately, *iii)* what states of our problem are worth investigating further to achieve the desired result?

Hopefully, this preparatory work helps recognising any limitations, and will lead to better algorithm design, or at least improved understanding of *why* the models are performing in the way that they do.

Learning

Furthermore, the learned model should be created in an iterative fashion, such that it the learned state-spaces are representative of the ones the eventual model would likely encounter. This can be achieved using *active* imitation learning, similar to the work of Ross and Bagnell (2010), Ross et al. (2011), in particular their DAgger framework.

*This explains why Paper II is completely different from the other publications.

Algorithm 1 Analysis & Learning Iterative Consecutive Executions (ALICE) framework, given a problem space $\mathcal{P}$, an expert policy $\pi_\star$, and set of benchmark algorithms $\mathcal{A}$.

procedure ALICE($\mathcal{P}, \pi_\star, \mathcal{A}$)
  $Y^{\pi_\star} \leftarrow \{Y(\pi_\star, \boldsymbol{\phi}(\mathbf{x})) \ : \ \mathbf{x} \in \mathcal{P}\}$ ▷ collect optimal solutions
  $\Phi^{\pi_\star} \leftarrow \left\{\boldsymbol{\phi}_{\pi^\star}(\mathbf{x}) \ : \ \mathbf{x} \in \mathcal{P}\right\}$ ▷ collect optimal meta-data
  for all $\pi \in \mathcal{A}$ **do** ▷ for each algorithm
    $Y^{\pi} \leftarrow \{Y(\pi, \boldsymbol{\phi}(\mathbf{x})) \ : \ \mathbf{x} \in \mathcal{P}\}$ ▷ collect solutions
    $\Phi^{\pi} \leftarrow \left\{\boldsymbol{\phi}_{\pi}(\mathbf{x}) \ : \ \mathbf{x} \in \mathcal{P}\right\}$ ▷ collect meta-data
    $\xi_\pi \leftarrow \mathbb{E}\left\{\pi_\star = \pi \ : \ \pi_\star\right\}$ ▷ optimality of π (i.e. when $Y^{\pi_\star} = Y^{\pi}$)
    $\zeta_\pi \leftarrow \textsc{Analyse}(\xi_\pi \mapsto Y^{\pi})$ ▷ relation between optimality and end-result
    $\Phi^{\pi} \leftarrow \textsc{Sample}(\Phi_\pi, \zeta_\pi)$ ▷ adjust set w.r.t. analysis
  end for
  $\Phi \leftarrow \{\Phi^{\pi} \ : \ \pi \in \{\mathcal{A} \cup \pi_\star\}\}$ ▷ training set
  $\hat{\pi} \leftarrow \textsc{Train}(\Phi)$ ▷ apply learning algorithm
  return $\hat{\pi}$ ▷ learned policy
end procedure

ALICE

It's the believe of the author, that the methodology of going about this, which is roughly described in Alg. 1, can be applied to any kind of optimisation problem which involves sequential decision making. As such, then it's suitable to name the framework: *Analysis & Learning Iterative Consecutive Executions*, or ALICE* for short. For demonstration purposes, this dissertation will solely be focusing on applying ALICE to dispatching rules for job-shop scheduling.

The framework mainly involves inspecting the stepwise optimality, ξ_π, for a heuristic policy π and it's relation to its end-result (here the makespan), as it defines its *footprint* in instance space. This is done for a set of benchmark algorithms $\pi \in \mathcal{A}$, during the *analysis* phase, which are then used to guide the training for the eventual the *learned* policy, $\hat{\pi}$. Finally, $\hat{\pi}$, can be post-processed in the same manner as done in the pre-processing phase, i.e., inspect $\xi_{\hat{\pi}}$ and $\zeta_{\hat{\pi}}$.

1.4 Outline

1.4. Update outline once draft is ready...

*The hopefully catchy and very deliberate 'backronym,' pays homage to the wonderful literary character, Alice in Wonderland–a personal favourite of the author.

An approach based on supervised learning, mostly on optimal schedules will be investigated and its effectiveness illustrated by improving upon well known dispatch rules for job-shop scheduling in Chapter 8. The method of generating training data is shown to be critical for the success of the method, as shown in **??**. 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 Paper III 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 8, as they are dependant on features based on those same dispatching rules under investigation.

Due to scarcity of real-world data, we let random problem generators suffice, that are described in Chapter 6. Moreover, the traditional OR-Library benchmark instances are similarly created, although for a greater variety of problem sizes. Smith-Miles and Bowly (2015) warn that general practice in the OR-community is over-tuning of algorithms to a relatively small set of aging* instances. Obviously, the choice of data set has a direct influence of the proposed algorithm. As they are developed with them specifically in mind. This is why robustness towards different problem spaces, than initially trained on, is of so much value, as it indicates how applicable our model is for real-world deployment

1.4. our ability to learn about the strengths and weaknesses of algorithms from empirical evidence

1.5 Supplementary material

The Prologue will mostly focus on traditional job-shop problem instances. However, in Chapter 3 there is a greater variety of problem spaces introduced, and when seen fit some of them will be investigated as well in the subsequent chapters. Since most experiments have been run on all problem spaces, they can be inspected in the supplementary Shiny application written in R. In addition, all source code and data is freely distributed from:

`https://github.com/ALICE-InRu/`

under the permissive creative commons share-alike licence.** Figure 1.3 displays the code's class diagram in relation to the thesis.

*The OR-Library problem instances are mostly from the 1980s and 1990s, or earlier (cf. Table 3.3).

**Attribution-ShareAlike 4.0 International (CC BY-SA 4.0)

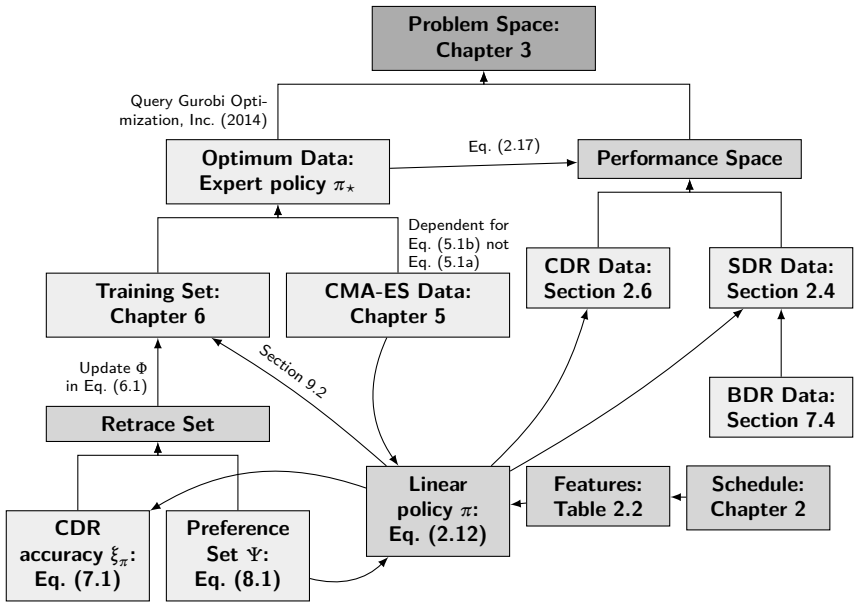


Figure 1.3: Class diagram for ALICE, C# implementation available at github

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

Doorknob

2

Job-shop Scheduling Problem

Scheduling problems, which occur frequently in practice, are a category within combinatorial optimisation problems. A subclass of scheduling problems is job-shop (JSP), which is widely studied in operations research. JSP deals with the allocation of tasks of competing resources where its goal is to optimise one ore more objectives. Job-shop's analogy is from the 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 have a high impact on many manufacturing organisations.

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 *travelling salesman problem** can be contrived as JSP: the salesman as a single machine in use; the cities to be visited are the jobs to be processed, and distance is sequence dependent set-up time. 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 flow-shop scheduling problem (FSP) (Guinet and Legrand, 1998, Tay and Ho, 2008).

*The travelling salesman problem (TSP) was formulated in the 1800s by the mathematicians W.R. Hamilton and Thomas Kirkman (Biggs et al., 1986). The salesman has to visit a set of cities exactly once (i.e. Hamiltonian path), with the objective of minimising the route, in terms of distance, between them.

Therefore, without loss of generality, this dissertation is structured around JSP.

Remark: Throughout the dissertation the FSP variation will *not* be a commonly used permutation flow-shop (PFSP) from the literature,* which has the added constraints of not allowing any jobs to pass one another. Here, the jobs have to be processed in the same machine order. However, machines do not necessarily need to process jobs in the same order, as is implied in PFSP. For PFSP the Manne (1960) model would be more appropriate, rather than the one described in the following section.

2.1 Mathematical formulation

Job-shop considered for this dissertation is when 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 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 notation for scheduling problems (cf. Chapter 2 in Pinedo, 2008) is given by a triplet $\alpha|\beta|\gamma$, where: α describes the machine environment; β details any additional processing characteristics and/or constraints, and finally γ lists the problem's objective. Hence our family of scheduling problems, i.e., a m machine JSP and FSP w.r.t. minimising makespan, is $Jm||C_{\max}$ and $Fm||C_{\max}$, respectively. An additional constraint commonly considered are job release-dates and due-dates, and then the objective is generally minimising the maximum lateness, denoted $Jm|r_j, d_j|L_{\max}$. However, those shop-requirements 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 set-up times.

For any given JSP each job J_j has an indivisible processing time (or cost) on machine M_a, p_{ja}, which is assumed to be integral and finite.

The starting time of job J_j on machine M_a is denoted $x_s(j, a)$ and its completion or end time is denoted $x_e(j, a)$ where,

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

*Paper III wrongly states that it is used PFSP problem instances, it was in fact FSP.

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

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

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.

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

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

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,

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

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 slack,

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

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 attribute in order to measure the quality of the schedule.

Note, from a job-oriented viewpoint, for a job already dispatched $J_j \in \mathcal{J}$ the corresponding set of machines already processed is $\mathcal{M}_j \subset \mathcal{M}$. Similarly from the machine-oriented viewpoint, $M_a \in \mathcal{M}$ with corresponding $\mathcal{J}_a \subset \mathcal{J}$.

2.2 Construction heuristics

Construction heuristics are designed in such a way that it limits the search space in a logical manner, preferably without excluding the true optimum. Here, the construction heuristic, Υ, is to schedule the dispatches as closely together as possible, i.e., minimise the schedule's idle times. More specifically, once an operation (j, a) has been chosen from the job-list, $\mathcal{L}$, by some dispatching rule, it can placed immediately after (but not prior) $x_e(j, \boldsymbol{\sigma}_j(a-1))$

on machine M_a due to Ineq. (2.2). However, to guarantee that Ineq. (2.3) is not violated, idle times M_a are inspected, as they create a slot which in J_j can occupy. Bearing in mind that J_j release time is $x_e(j, \boldsymbol{\sigma}_j(a-1))$ one cannot implement Eq. (2.5) directly, instead it has to be updated as follows,

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

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.,

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

There are several heuristic methods for selecting a slot from Eq. (2.7), 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. Experiments favoured dispatching in the earliest slot,* thus used throughout.

Note that the choice of slot is an intrinsic heuristic within Υ. The focus of this dissertation, however, is on learning the priority of the jobs on the job-list, for a fixed construction heuristic. Hence, there could be some problem instances in which the optimum makespan cannot be achieved, due to the limitations of Υ of not being properly able to differentiate between which slot from Eq. (2.7) 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.

Dispatching rules

Dispatching rules (DR) are an integral part of a construction heuristics, as it determines the priorities of the job-list, i.e., the jobs who still have operations unassigned. Starting with an empty schedule, and sequentially adding one operation (or task) at a time. Then, for each time step k, an operation is dispatched which has the highest priority of the job-list, $\mathcal{L}^{(k)} \subset \mathcal{J}$. If there is a tie, some other priority measure is used. However, let's assume that ties are broken randomly. Algorithm 2 outlines the pseudo code for the entire dispatching process of a JSP problem instance.

*Preliminary experiments of 500 JSP instances where inspected: First slot chosen could always achieve its known optimum by implementing Alg. 2, however, only 97% of instances when choosing the smallest slot.

Algorithm 2 Pseudo code for constructing a JSP sequence using a deterministic scheduling policy (or dispatching rule), π, for a fixed construction heuristic, Υ.

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

Henceforth, we will adopt the following terminology: a *sequence* will refer to the sequential ordering of the dispatches* of tasks to machines, namely,

$$\boldsymbol{\chi} = \{\chi_k\}_{k=1}^K = \left\{(j,a) \ : \ J_j \in \mathcal{L}^{(k)}\right\}_{k=1}^K \tag{2.8}$$

The collective set of allocated tasks to machines, which is interpreted by its sequence, is referred to as a *schedule*; and a *scheduling policy* (or dispatching rule) π will pertain to the manner in which the sequence is manufactured: be it a SDR such as SPT or some other heuristic. Sequence and schedule are often used interchangeably, as they are closely related. A complete schedule is also known as K-solution** (Bertsekas et al., 1997).

2.3 Example

There are many examples of job-shop for real-world application. For demonstration purposes, let's examine a hypothetical problem from the 18th century. Assume we are invited to the Mad Hatter's Tea Party in Wonderland, illustrated in Fig. 2.1. There are four guests attending: J_1) Alice; J_2) March Hare; J_3) Dormouse, and of course our host J_4) Mad Hatter. During these festivities, there are several things each member of the party has to perform. They all have to: M_1) have wine or pour tea; M_2) spread butter; M_3) get a haircut; M_4) check the time of the broken watch for themselves, and M_5) say what they

*Note, only a sequence of J_j is needed, since the corresponding M_a can be obtained by reading $\boldsymbol{\sigma}$.
**A partial schedule, at step k, is called k-solution.

Table 2.1: Example of 4×5 JSP

Guest	Job	Machine ordering $\boldsymbol{\sigma}$					Processing times $\mathbf{p}$				
Alice	J_1	~~1~~	~~2~~	~~3~~	~~4~~	~~5~~	~~26~~	~~25~~	~~40~~	~~15~~	~~42~~
March Hare	J_2	~~1~~	**2**	3	4	5	~~18~~	**86**	86	68	84
Dormouse	J_3	~~1~~	~~3~~	**2**	4	5	~~20~~	**59**	~~23~~	33	96
Mad Hatter	J_4	~~4~~	~~3~~	**1**	5	2	**40**	47	~~55~~	~~13~~	99

mean, e.g., asking a riddle or reciting a poem to the group. The guests are very particular creatures, and would like to do these task in a very specific order, e.g., March Hare insists on doing them alphabetically. Each would rather wait than breaking their habit. They tend to be absent-minded, so each task takes them a different amount of time. Let's assume their processing times and ordering are given in Table 2.1.

Unfortunately, Alice can't stay long. She must leave as soon as possible to play croquet with the Red Queen, and she mustn't be late for that very important date. Otherwise, it's off with someone's head! However, Alice, had a proper upbringing and won't leave the table until everyone has finished their tasks. How should the guests go about their tea-party, in order for Alice to be on-time?

Figure 2.1: The Mad Hatter's Tea Party, from Alice's Adventures in Wonderland by Carroll (1865). Illustration by John Tenniel (1820-1914).

The problem faced by Alice and her new friends is in what order should they rotate their tasks between themselves so that they all finish as soon as possible? This can be considered as is a typical four-job and five-machine job-shop, where: our guests are the jobs; their tasks are the machines, and our objective is to minimise $C_{\max}$, i.e., when Alice can leave.

Let's assume we've come to the party, after 10 operations have already been made (i.e. strikeout entries in Table 2.1), by using the following job sequence,*

$$\boldsymbol{\chi} = \{\chi_i\}_{i=1}^{k-1} = \{J_4, J_2, J_3, J_3, J_1, J_1, J_1, J_1, J_1, J_4\} \tag{2.9}$$

hence currently, at step $k = 11$, the job-list is $\mathcal{L}^{(k)} = \{J_2, J_3, J_4\}$ indicating the 3 potential** jobs (i.e. denoted in bold in Table 2.1) to be dispatched, i.e., $\chi_k \in \mathcal{L}^{(k)}$.

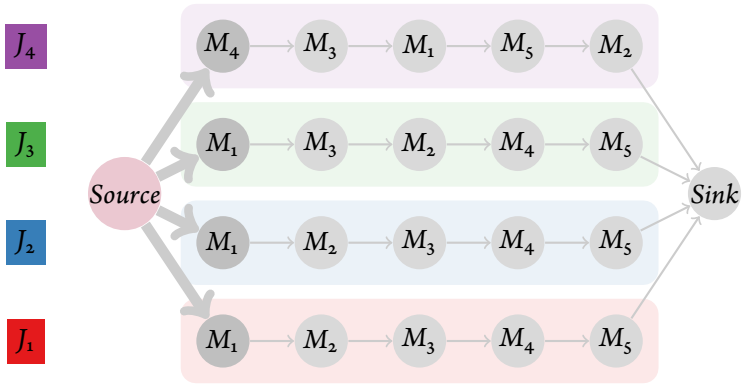
This is a very compact form for the current partial solution, it's easiest to comprehend it via disjunctive graph (Roy and Sussmann, 1964) to model the work-flow of tasks to be scheduled. Let's encode: *i)* the operations as vertices; *ii)* horizontally aligning them w.r.t. each job J_j; *iii)* connect vertices with directed edges according the Ineq. (2.2), and *iv)* by introducing dummy vertices before and after, then the goal is to visit each vertex exactly once, or *Hamiltonian* path: starting at the 'source' (i.e. empty schedule), and finishing at the sink (i.e. complete schedule). The path gives the prescription of the order in which the jobs rotate between machines. Figure 2.2 depicts the path generation at the beginning, midway, and final stages for our Tea Party.: *i)* gray vertices are operations that haven't yet been dispatched; *ii)* pink vertices are the ones that correspond to $\boldsymbol{\chi}$, and *iii)* pink directed edges indicate the current partial Hamiltonian path.

Now we're interested to know when each guest should start their task, i.e., the project schedule. Figure 2.3 illustrates the temporal partial schedule (or k-solution) of Eq. (2.9) as a Gantt-chart: *i)* numbers in the boxes represent the job identification j; *ii)* the width of the box illustrates the processing times for a given job for a particular machine M_a (on the vertical axis); *iii)* the dashed boxes represent the resulting $(k+1)$-solution for when a particular job is scheduled next, and *iv)* the current $C_{\max}$ is denoted with a dotted line. Note, the disjunctive graph from Fig. 2.2b gives the schedule in Fig. 2.3.

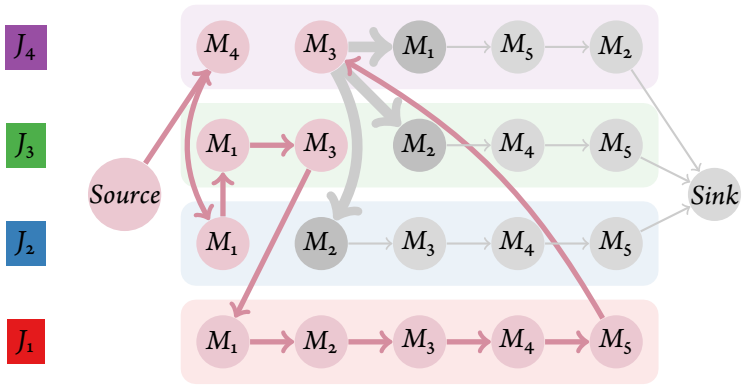
If the job with the shortest processing time were to be scheduled next, i.e., applying SPT-rule, then J_4 would be dispatched. Similarly, for LPT-rule (largest processing time) then J_2 would be dispatched. Other DRs use features not directly observable from looking at the k-solution (but easy to keep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

*In fact this is the sequence resulting from 10 dispatches following the SPT-rule, to be defined shortly.

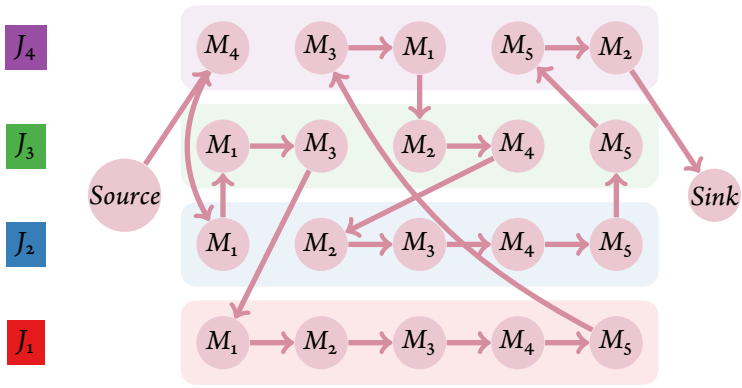
**Alice is quite anxious to leave, so she has already completed everything, and therefore $J_1 \notin \mathcal{L}^{(11)}$.



(a) $k = 0$



(b) $k = K/2$



(c) $k = K$

Figure 2.2: Graph representation of a 4×5 job-shop, where pink vertices are completed tasks, and grey are unassigned. Moreover, grey arrows point to the operations that are next on the job-list, $\mathcal{L}^{(k+1)}$, and pink arrows (traversing from source towards sink) yield the sequence of operations for the schedule, i.e., $\boldsymbol{\chi}$.

2.4 Single priority based dispatching rules

A *single priority dispatching rule* (SDR) is a function of attributes, or features, of the jobs and/or machines of the schedule. The features can be constant or vary throughout the scheduling process. For instance, 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 $\mathcal{L}$,

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 surprising 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 job-list waiting to be scheduled, and are generally only based on few features and simple mathematical operations. Continuing with the example from Section 2.3, the final schedules for these main SDRs (and a possible optimal schedule for reference) are depicted in Fig. 2.4. As we can see, MWR would have been the best strategy for Alice and company, since it has the makespan closest to the optimum.

2.5 Features for job-shop

A DR may need to perform a one-step look-ahead, and observe features of the partial schedule to make a decision. For example by observing the resulting temporal makespan.

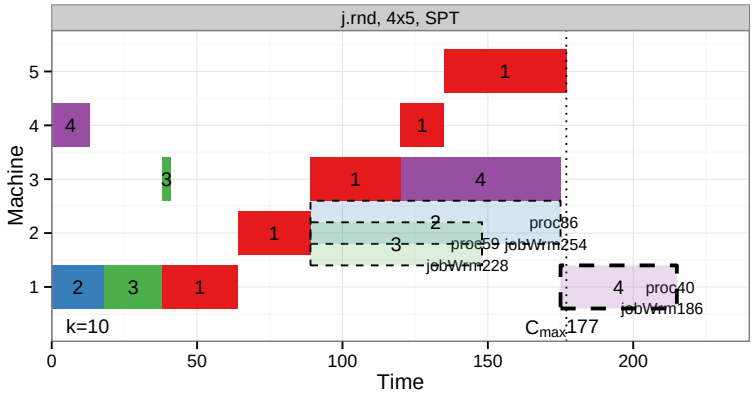


Figure 2.3: Gantt chart of a partial JSP schedule after 10 dispatches: Solid and dashed boxes represent χ and $\mathcal{L}^{(11)}$, respectively. Current C_{max} denoted as dotted line.

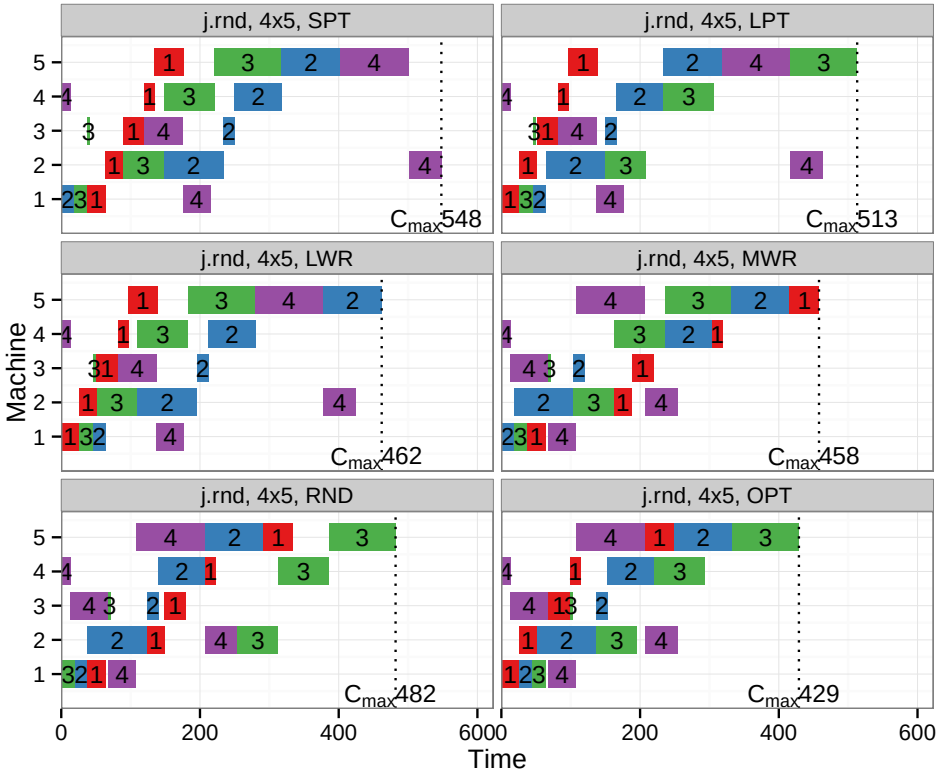


Figure 2.4: SDRs applied to the Tea-party example in Section 2.3. A possible optimal solution is shown in the lower right corner as a reference.

These emanated observed features are sometimes referred to as an *after-state* or *post-decision state*. A k-solution is denoted $\boldsymbol{\chi}^j$ where J_j is the latest dispatch, i.e., $\chi_k = J_j$, and its resulting features is denoted,

$$\boldsymbol{\varphi}^j := \boldsymbol{\varphi}(\boldsymbol{\chi}^j). \tag{2.10}$$

Features are used to grasp the essence of the current state of the schedule. Temporal scheduling features applied in this dissertation are given in Table 2.2.

The features of particular interest were obtained from inspecting the aforementioned SDRs from Section 2.4: namely φ_1 and φ_7. Moreover, φ_1-φ_8 and φ_9-φ_{16} are job-related and machine-related attributes of the current schedule, respectively.

Some features are directly observed from the k-solution, such as the job- and machine-related features, namely, φ_1-φ_{16} and they 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_{17}-φ_{20}, as a benchmark by retrieving what would the resulting $C_{\max}$ be given if that SDR would be implemented from that point forward. Moreover, random completion of the k-solution are implemented, here φ_{21}-φ_{24} corresponds to statistics from 100 random roll-outs, which can be used to identify which features $\boldsymbol{\varphi}$ are promising on a long-term basis. *Roll-out algorithms* (Bertsekas et al., 1997), also known as *Pilot Method* (Duin and Voß, 1999), for combinatorial optimisation aim to improve performance by sequential application of a pilot heuristic, which completes the remaining $K - k$ steps. Roll-outs for JSP have been conducted by Rúnarsson et al. (2012). Continuing with that work, Geirsson (2012) compares several pilot heuristics, e.g., *Randomly Chosen Dispatch Rules* which is similar to φ_{17}-φ_{20} (but here one roll-out per fixed SDR). The motivation being, that a SDR-based roll-out are of higher quality than random ones, requiring less computational budget. However, Geirsson notes that performance w.r.t. traditional random roll-outs is statistically insignificant, and not worth the overhead of implementing various SDRs beforehand.

Geirsson reworks the roll-out algorithm as an $|\mathcal{L}|$-armed bandit,* i.e., each job of the job-list are the levers. Since the best job, j^*, to dispatch at step k, is not known beforehand, therefore all available jobs are evaluated, using roll-outs. As a result, using the features φ_{21}-φ_{24}, the weights $\mathbf{w}$ yield the deterministic pilot heuristic. Although in Geirsson's work, other statistics are used for guidance, e.g., quartile and octile.

*In probability theory, the multi-armed bandit problem (Berry and Fristedt, 1985) describes a gambler at a row of slot machines, who has to decide which machines to play, i.e., pull its lever, in order to maximise his rewards, that are specific to each machine. The gambler also has to decide how many times to play each machine and in which order to play them. The gambler's actions are referred to as *pilot-heuristic*.

It's noted, that the roll-outs considered in this dissertation, are with a very frugal budget, only 100 roll-outs per lever is considered – all evenly distributed between levers. But using the multi-armed bandit paradigm, it's possible to allocate roll-outs originating from the job-list with bias towards more promising levers.

2.6 Composite dispatching rules

Priority dispatching rules were originally introduced in Giffler and Thompson (1960) to resolve conflicts of the job-list, and have made great headway since. They are especially attractive since they are relatively simple to implement, fast and find good schedules. In addition, they are easy to interpret, which makes them desirable for the end-user (i.e. shop floor operators). However, they can also fail unpredictably. 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 DRs.

For instance, optimising $J1||L_{\max}$ (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: *i)* WSPT * (SPT weighted w.r.t. $\mathcal{J}$), and *ii)* minimum slack first (MS),** yields the CDR *Apparent Tardiness Cost*, which can work well on a broader set of problem instances than the original SDRs by themselves.

CDRs can deal with a greater number of more complicated functions constructed from the schedules attributes. In short, a CDR is a combination of several DRs. For instance let π be a CDR comprised of d DRs, then the index I for $J_j \in \mathcal{L}^{(k)}$ using π is,

$$I_j^{\pi} = \sum_{i=1}^{d} w_i \pi_i(\boldsymbol{\chi}^j) \tag{2.11}$$

where $w_i > 0$ and $\sum_{i=0}^{d} w_i = 1$, then w_i gives the *weight* of the influence of π_i (which could be a SDR or another CDR) to π. Note, each π_i is function of J_j's attributes from the k-solution $\boldsymbol{\chi}^j$.

The composite priority dispatching rule presented in Eq. (2.11) can be considered as a special case of a the following general linear value function,

$$\pi(\boldsymbol{\chi}^j) = \sum_{i=1}^{d} w_i \varphi_i(\boldsymbol{\chi}^j) \overset{(2.10)}{=} \left\langle \mathbf{w} \cdot \boldsymbol{\varphi}^j \right\rangle. \tag{2.12}$$

*WSPT is optimal when all release dates and due dates are zero.

**MS is optimal when all due dates are sufficiently loose and spread out.

Table 2.2: 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	$x_e(j,a)$	endTime
φ_4	job arrival time	$x_e(j,a-1)$	arrival
φ_5	time job had to wait	$x_s(j,a) - x_e(j,a-1)$	wait
φ_6	total processing time for job	$\sum_{a\in\mathcal{M}} p_{ja}$	jobTotProcTime
φ_7	total work remaining for job	$\sum_{a'\in\mathcal{M}\setminus\mathcal{M}_j} p_{ja'}$	jobWrm
φ_8	number of assigned operations for job	$\lvert\mathcal{M}_j\rvert$	jobOps
	machine related		
φ_9	when machine is next free	$\max_{j'\in\mathcal{J}_a}\{x_e(j',a)\}$	macFree
φ_{10}	total processing time for machine	$\sum_{j\in\mathcal{J}} p_{ja}$	macTotProcTime
φ_{11}	total work remaining for machine	$\sum_{j'\in\mathcal{J}\setminus\mathcal{J}_a} p_{j'a}$	macWrm
φ_{12}	number of assigned operations for machine	$\lvert\mathcal{J}_a\rvert$	macOps
φ_{13}	change in idle time by assignment	$\Delta s(a,j)$	reducedSlack
φ_{14}	total idle time for machine	$\sum_{j'\in\mathcal{J}_a} s(a,j')$	macSlack
φ_{15}	total idle time for all machines	$\sum_{a'\in\mathcal{M}}\sum_{j'\in\mathcal{J}_{a'}} s(a',j')$	allSlack
φ_{16}	current makespan	$\max_{(j',a')\in\mathcal{J}\times\mathcal{M}_{j'}}\{x_f(j',a')\}$	makespan
	final makespan related		
φ_{17}	final makespan using SPT	$C_{\max}^{\text{SPT}}$	SPT
φ_{18}	final makespan using LPT	$C_{\max}^{\text{LPT}}$	LPT
φ_{19}	final makespan using LWR	$C_{\max}^{\text{LWR}}$	LWR
φ_{20}	final makespan using MWR	$C_{\max}^{\text{MWR}}$	MWR
φ_{RND}	final makespans using 100 random rollouts	$\{C_{\max}^{\text{RND}}\}_{i=1}^{100}$	
φ_{21}	mean for φ_{RND}	$\mathbb{E}\left\{\varphi_{\text{RND}}\right\}$	RNDmean
φ_{22}	standard deviation for φ_{RND}	$\sqrt{\mathbb{E}\left\{\varphi_{\text{RND}}^2\right\} - \mathbb{E}\left\{\varphi_{\text{RND}}\right\}^2}$	RNDstd
φ_{23}	minimum value for φ_{RND}	$\min\{\varphi_{\text{RND}}\}$	RNDmin
φ_{24}	maximum value for φ_{RND}	$\max\{\varphi_{\text{RND}}\}$	RNDmax

when $\pi_i(\cdot) = \varphi_i(\cdot)$, i.e., a composite function of the features from Table 2.2.

Finally, the job to be dispatched, J_{j^*}, corresponds to the one with the highest value, i.e.,

$$J_{j^*} = \underset{J_j \in \mathcal{L}}{\operatorname{argmax}} \ \pi(\boldsymbol{\varphi}^j) \tag{2.13}$$

Since we're using a feature space based on job-attributes, then it's trivial to interpret Eq. (2.12) as the SDRs from Section 2.4. Then for $i \in \{1, \ldots, d\}$, they're simply,

$$\text{SPT:} \qquad w_i = \begin{cases} -1 & \text{if } i = 1 \\ 0 & \text{otherwise} \end{cases} \tag{2.14a}$$

$$\text{LPT:} \qquad w_i = \begin{cases} 1 & \text{if } i = 1 \\ 0 & \text{otherwise} \end{cases} \tag{2.14b}$$

$$\text{MWR:} \qquad w_i = \begin{cases} 1 & \text{if } i = 7 \\ 0 & \text{otherwise} \end{cases} \tag{2.14c}$$

$$\text{LWR:} \qquad w_i = \begin{cases} -1 & \text{if } i = 7 \\ 0 & \text{otherwise} \end{cases} \tag{2.14d}$$

Automated discovery of CDRs

Generally the weights $\mathbf{w}$ in Eq. (2.12) are chosen by the algorithm designer a priori. A more sophisticated approach would have the algorithm discover these weights autonomously. For instance via preference-based imitation learning or evolutionary search, to be discussed in Chapter 8 and Chapter 5, respectively.

Mönch et al. (2013) stress the importance of automated discovery of DRs and named several successful such 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 creation of optimisation heuristics for scheduling.

With meta heuristics one can use existing DRs and use for example portfolio-based algorithm selection either based on a single instance (Gomes and Selman, 2001, Rice, 1976) 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 9 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 11 SDRs for JSP to create a pool of 33 CDRs that strongly outperformed the ones they were based on. The CDRs were created with multi-contextual functions based either on machine idle time or job waiting time (similar to φ_5 and φ_{14} in Table 2.2), creating CDRs thate 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 12 existing DRs from the literature, in their approach they had 48 priority rules combinations, yielding 48 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.

It is intuitive to get a boost in performance by introducing new CDRs, since where one DR might be failing, another could be excelling so combining them together should yield a better CDR. However, these aforementioned approaches introduce fairly ad-hoc solutions and there is no guarantee the optimal combination of dispatching rules were found.

2.7 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. Smith-Miles and Lopes (2011), Smith-Miles et al. (2009) and Paper III), as follows,

Problem space $\mathcal{P}$ is defined as the union of N problem instances consisting of processing time and ordering matrices, $\mathbf{x} = (\mathbf{p}, \boldsymbol{\sigma})$, for n-jobs and m-machines,

$$\mathcal{P} = \{\mathbf{x}_i \ : \ n \times m \}_{i=1}^{N} \tag{2.15}$$

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

Feature space $\mathcal{F}$ which was outlined in Section 2.5. Note, these are not the only possible set of features. However, the local feature, φ_1-φ_{16}, are built on the work by Smith-Miles et al. (2009) and Paper I 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 2.4,

$$\mathcal{A} = \{\text{SPT, LPT, LWR, MWR, RND, } \dots \} . \tag{2.16}$$

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

$$\rho = \frac{C^{\pi}_{\max} - C^{\pi_\star}_{\max}}{C^{\pi_\star}_{\max}} \cdot 100\% \tag{2.17}$$

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

$$\mathcal{Y} = \{\rho_i\}_{i=1}^{N} \tag{2.18}$$

Equation (2.17) measures the discrepancy between predicted value and true outcome, and is commonly referred to as a loss function, which we would like to minimise for π.

The mapping $\Upsilon : \ \mathcal{A} \times \mathcal{F} \mapsto \mathcal{Y}$ is the step-by-step construction heuristic in Alg. 2.

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

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 3.2. Following the approach in Watson et al. (2002), difficult problem instances are not filtered out beforehand. The problem spaces for Part II are summarised in Table 3.1. Note, that the problem generators in Papers IV to VI are the same as described here.

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. They are detailed in Section 3.3.

Table 3.1: JSP and FSP problems spaces used in Part II

Paper	Problem	$I = [u_1, u_2]$*	size ($n \times m$)	name
I	JSP	$[1, 100], [50, 100]$	6×6	*j.rnd*, *j.rndn*
III	JSP	$[1, 200]$	6×6	*j.rnd*
IV	JSP, FSP	$[1, 99], [45, 55]$	$6 \times 5, 10 \times 10$	*j.rnd*, *j.rndn*, *f.rnd*, *f.rndn*, *f.jc*
V	JSP	$[1, 99], [45, 55]$	6×5	*j.rnd*, *j.rndn*
VI	JSP, FSP	$[1, 99], [45, 55]$	10×10	*j.rnd*, *j.rndn*, *f.rnd*

*Processing times are uniformly distributed from an interval $I = [u_1, u_2]$, i.e., $\mathbf{p} \sim \mathcal{U}(u_1, u_2)$.

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 3.2.

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 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.

Hildebrandt et al. (2010) argue that the randomly generated problem instances aren't a proper representative for real-world long-term job-shop applications, e.g., by the narrow choice of release-dates, yielding schedules that are overloading in the beginning phases. However, as stated in Chapter 2, release-dates constraints won't be considered here. 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 $Jm||C_{\max}$ set-up. 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.

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 $\mathcal{P}^{n\times m}_{f.jc}$
where $\mathbf{p}$ is dependent on job index, however, independent of machine index.

FSP machine-correlated $\mathcal{P}^{n\times m}_{f.mc}$
where $\mathbf{p}$ is dependent on machine index, however, independent of job index.

FSP mixed-correlated $\mathcal{P}^{n\times m}_{f.mxc}$
where $\mathbf{p}$ is dependent on machine and job indices.

*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.

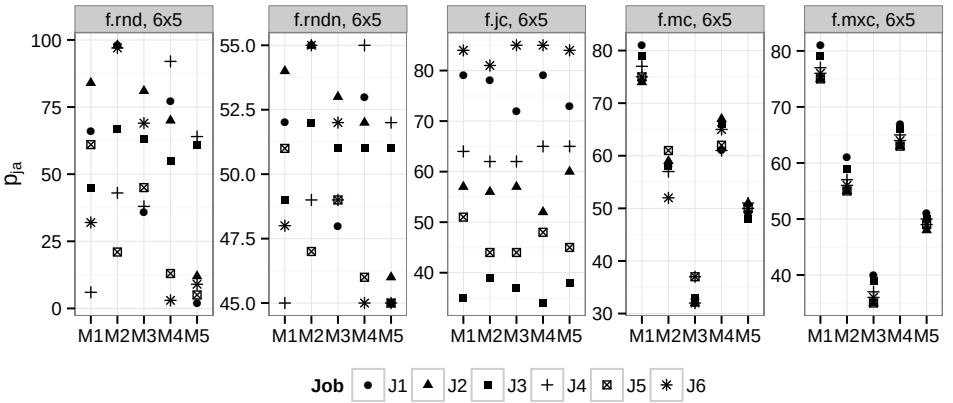


Figure 3.1: Examples of job processing times for 6×5 of different FSP structures.

In all cases, the (job, machine or mixed) correlation can be of degree $0 \leq \alpha \leq 1$. When $\alpha = 0.0$ the problem instances closely correspond to $\mathcal{P}^{n\times m}_{f.rnd}$, hence the degree of α controls the transition of random to structured. Let's assume $\alpha = 1$.

An example of distribution of processing times are depicted in Fig. 3.1, where machine indices are on the horizontal axis, job indices are colour-coded, and their corresponding processing times, p_{ja}, are on the vertical axis.

For each FSP class N_{train} and N_{test} instances were generated for training and testing, respectively. Values for N are given in Table 3.2.

3.3 Benchmark problem suite

A total of 82 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 3.3. Given the high problem dimensions of some problems, the optimum is not known, hence in those instances Eq. (2.17) will be reporting deviation from the latest best known solution (BKS) from the literature, reported by Banharnsakun et al. (2012), Jain and Meeran (1999), and for FSP consult Ancău (2012).

Job-shop 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. Similar to the synthetic JSP problem spaces discussed earlier, Adams et al. (1988) introduce five JSP instances with a random machine ordering and processing times

Table 3.2: Problem space distributions used in experimental studies.

	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
	$\mathcal{P}^{10\times10}_{j.rnd,J_1}$	10×10	300	200	random with job variation
	$\mathcal{P}^{10\times10}_{j.rnd,M_1}$	10×10	300	200	random with machine variation
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

$\mathbf{p} \sim \mathcal{U}(50, 100)$, 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 $\mathbf{p} \sim \mathcal{U}(1, 100)$. There are a total of five problems in four dimension classes: *i)* 20×10; *ii)* 20×15; *iii)* 50×10, and *iv)* 50×10. 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 ten 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.

Table 3.3: Benchmark problems from OR-Library used in experimental studies.

	name	$n \times m$	N_{test}	note	shorthand
JSP	$\mathcal{P}_{ft}$	various	3	Fisher and Thompson (1963)	`ft06,ft10,ft20`
	$\mathcal{P}_{la}$	various	40	Lawrence (1984)	`la01-la40`
	$\mathcal{P}_{abz}$	various	5	Adams et al. (1988)	`abz05-abz09`
	$\mathcal{P}_{orb}$	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-reC42`

*Only odd-numbered instances in `rec01-rec42` 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.

Flow-shop OR-Library

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 in practice. However, only the random instances they used (type C) 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, although Ancău (2012) omitted $\mathcal{P}_{reC}^{75\times 20}$ instances in their comparison.

Sentence first – verdict afterwards.

The Queen

4 Problem difficulty

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*, 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 optimum) 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. This is why we consider the random vs. random-narrow problem spaces in Sections 3.1 and 3.2.

On the other hand, Smith-Miles and Lopes use a completely automated approach. Using timetabling instances, they implement a self-organizing map (SOM) on the feature space to group similar problem instances together, that were both real world instances and synthetic ones using different problem generators. That way it was possible to plot visually the footprints for several algorithms.

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 8 will be heavily based on feature pairs. Thereby making it possible to infer what sort of feature behaviour distinguishes between *good* and *bad* schedules.

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 the subsequent sections is rather on few simple algorithms, namely the SDRs described in Section 2.4, i.e., we will limit the algorithm space to,

$$\mathcal{A} := \{\text{SPT}, \text{LPT}, \text{LWR}, \text{MWR}\} \tag{4.1}$$

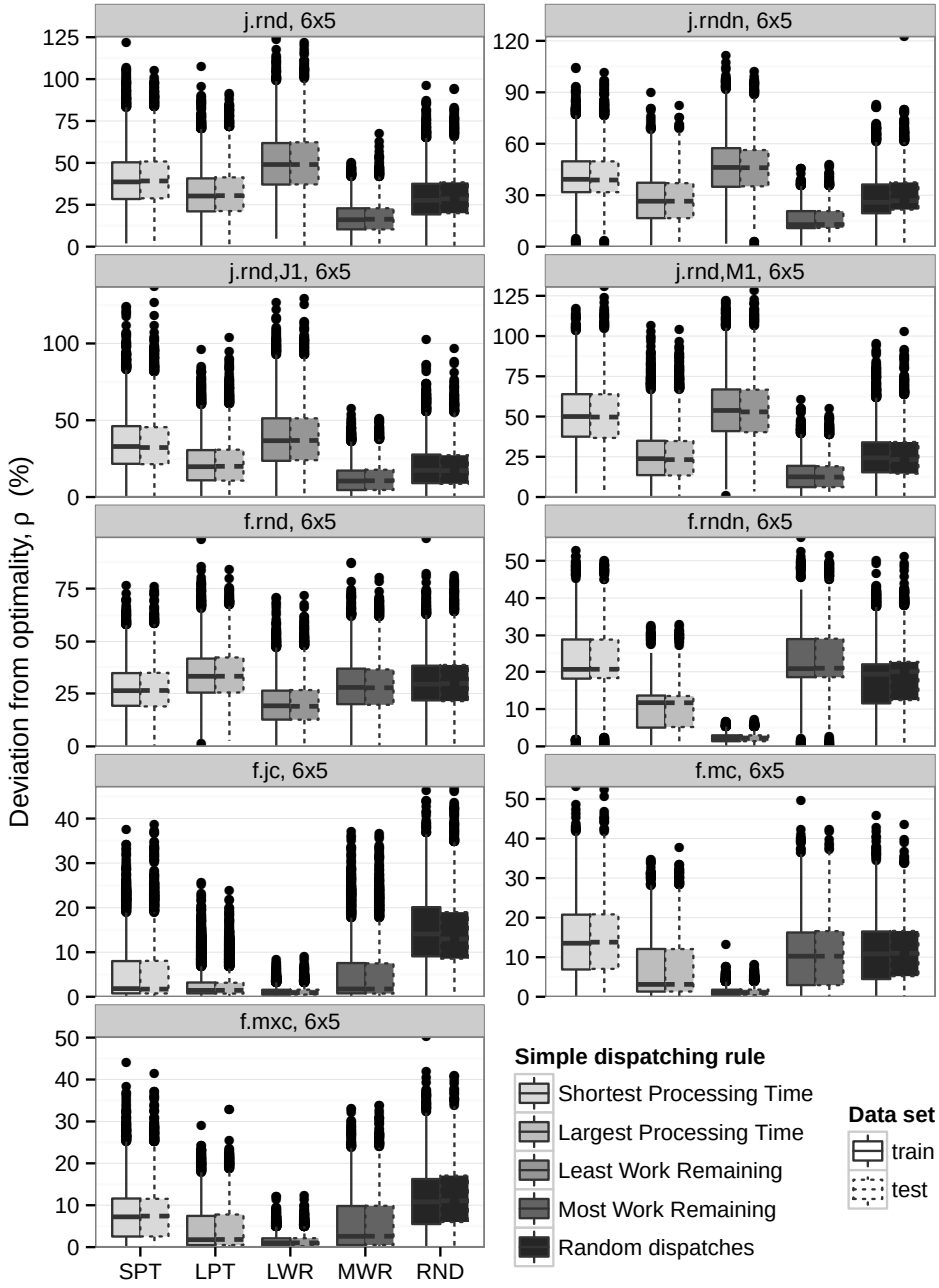
and try to understand *how* they work on the instance space, similar to Watson et al. (2002), who analysed the fitness landscape of several problem classes for a fixed algorithm.

Depending on the data distribution, dispatching rules perform differently. A box-plot for deviation from optimality, ρ, defined by Eq. (2.17), using all problem spaces from Table 3.2 are depicted in Fig. 4.1. 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.

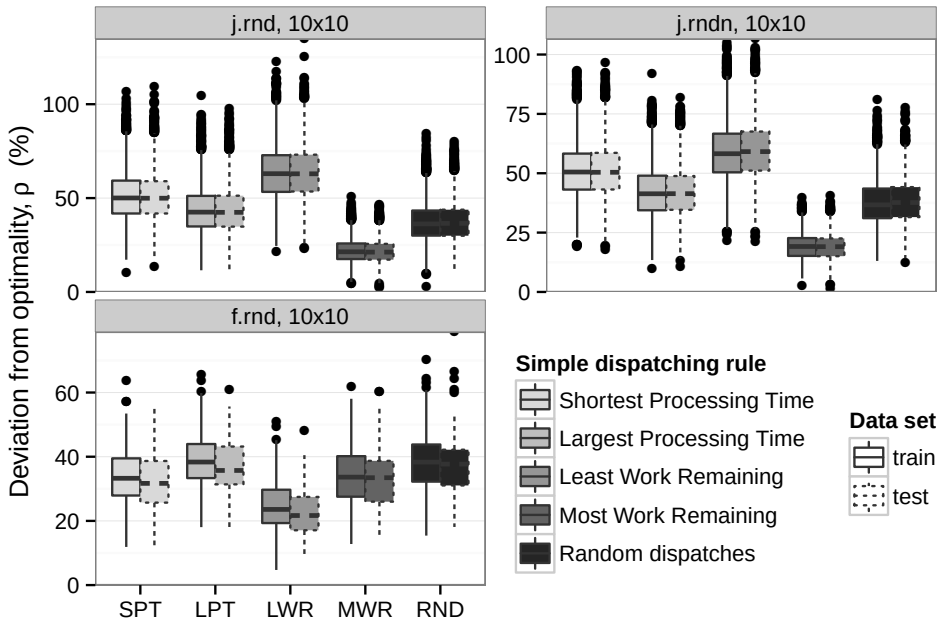
4.1 Distribution difficulty

In Paper III, 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 Section 3.2.



(a) 6×5

Figure 4.1: Box-plots of deviation from optimality, ρ, when applying SDRs for all problem spaces in Chapter 3



(b) 10 × 10

Figure 4.1 (cont.)

4.2 Defining *easy* versus *hard* schedules

It's relatively ad-hoc how to define what makes a schedule 'difficult'. For instance, it could be sensible to define it in terms of how many Simplex iterations are needed to find an optimal schedule, using *Branch and Bound*.* However, preliminary experiments showed that an increased amount of Simplex iterations didn't necessarily transcend to high ρ. If anything, it means there are many optimal (or near-optimal) solutions available, which causes the slow process of pruning branches of the tree, before reaching to a final incumbent solution. If that's the case, than that's promising for our instance, as it's likelier for an arbitrary algorithm to find a good solution.

*Branch and bound (Land and Doig, 1960) is a methodology in integer linear programming, where the original problem is branched into smaller sub-problems until it becomes easily solvable. Each sub-problem has a lower bound on its solution, found with LP-relaxation. Depending on the lower bound, sub-branches are systemically discarded, since they cannot contain the optimal solution.

Table 4.1: Threshold for ρ for easy and hard schedules, i.e., $\rho < \rho^{\text{1st Qu.}}$ and $\rho > \rho^{\text{3rd Qu.}}$ are classified as easy and hard schedules, respectively. Based on Table 3.2 training sets.

(a) 6×5

Problems	Q1	Q3
$\mathcal{P}^{6\times5}_{j.rnd}$	19.91	47.21
$\mathcal{P}^{6\times5}_{j.rndn}$	16.63	45.01
$\mathcal{P}^{6\times5}_{j.rnd,J_1}$	11.85	38.53
$\mathcal{P}^{6\times5}_{j.rnd,M_1}$	16.35	53.19
$\mathcal{P}^{6\times5}_{f.rnd}$	18.46	35.52
$\mathcal{P}^{6\times5}_{f.rndn}$	3.39	21.07
$\mathcal{P}^{6\times5}_{f.jc}$	0.64	3.34
$\mathcal{P}^{6\times5}_{f.mc}$	1.04	13.40
$\mathcal{P}^{6\times5}_{f.mxc}$	0.46	3.67

(b) 10×10

Problems	Q1	Q3
$\mathcal{P}^{10\times10}_{j.rnd}$	29.27	58.45
$\mathcal{P}^{10\times10}_{j.rndn}$	26.74	57.17
$\mathcal{P}^{10\times10}_{j.rnd,J_1}$	17.90	50.29
$\mathcal{P}^{10\times10}_{j.rnd,M_1}$	18.00	65.79
$\mathcal{P}^{10\times10}_{f.rnd}$	26.13	39.27

Intuitively, it's logical to use the schedule's objective to define the difficulty directly, i.e., inspecting deviation from optimality, ρ. Moreover, since the SDRs from Eq. (4.1) will be used throughout as a benchmark for subsequent models, the quartiles for ρ, using the SDRs on their training set will be used to differentiate between easy and hard instances. In particular, the classification is defined as follows,

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

$$\mathcal{E}(a) := \left\{\mathbf{x} \;:\; \rho = \Upsilon(a, \mathbf{x}) < \rho^{\text{1st. Qu.}}\right\} \tag{4.2a}$$

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

$$\mathcal{H}(a) := \left\{\mathbf{x} \;:\; \rho = \Upsilon(a, \mathbf{x}) > \rho^{\text{3rd. Qu.}}\right\} \tag{4.2b}$$

where $\mathbf{x} \in \mathcal{P}_{\text{train}}$ for a given $a \in \mathcal{A}$ from Eq. (4.1). Table 4.1 reports the first and third quartiles 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 Tables 4.2 and 4.3, respectively.

Table 4.2: Percentage (%) of 6×5 training instances classified as easy and hard schedules, defined by Eq. (4.3). Note, each problem space consists of $N_{\text{train}} = 500$.

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

SDR	Easy	Hard
SPT	8.90	30.38
LPT	22.06	15.24
LWR	3.64	54.18
MWR	65.30	0.20

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

SDR	Easy	Hard
SPT	2.88	37.54
LPT	24.42	9.70
LWR	2.10	52.82
MWR	70.70	0.06

(c) $\mathcal{P}^{6\times5}_{j.rnd,J_1}$

SDR	Easy	Hard
SPT	8.22	38.20
LPT	27.92	14.18
LWR	7.80	46.70
MWR	56.00	0.92

(d) $\mathcal{P}^{6\times5}_{j.rnd,M_1}$

SDR	Easy	Hard
SPT	2.28	43.08
LPT	31.68	5.72
LWR	1.10	51.12
MWR	64.96	0.10

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

SDR	Easy	Hard
SPT	23.02	22.90
LPT	8.44	41.82
LWR	47.60	7.50
MWR	20.94	27.82

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

SDR	Easy	Hard
SPT	0.94	44.38
LPT	13.22	7.28
LWR	85.18	0
MWR	0.48	48.42

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

SDR	Easy	Hard
SPT	22.14	36.44
LPT	21.52	24.08
LWR	35.64	2.80
MWR	21.38	36.70

(h) $\mathcal{P}^{6\times5}_{f.mc}$

SDR	Easy	Hard
SPT	10.64	49.20
LPT	18.46	18.98
LWR	49.04	0
MWR	21.46	31.76

(i) $\mathcal{P}^{6\times5}_{f.mxc}$

SDR	Easy	Hard
SPT	12.58	45.16
LPT	26.30	24.78
LWR	31.60	7.68
MWR	29.66	22.48

Table 4.3: Percentage (%) of 10×10 training instances classified as easy and hard schedules, defined by Eq. (4.3). Note, each problem space consists of $N_{\text{train}} = 300$.

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

SDR	Easy	Hard
SPT	2.67	27.00
LPT	10.33	13.67
LWR	0.67	59.33
MWR	86.33	0

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

SDR	Easy	Hard
SPT	1.00	31.67
LPT	6.67	9.33
LWR	0	59.00
MWR	92.33	0

(c) $\mathcal{P}^{10\times10}_{j.rnd,J_1}$

SDR	Easy	Hard
SPT	3.33	40.00
LPT	21.67	11.33
LWR	3.67	48.67
MWR	71.33	0

(d) $\mathcal{P}^{10\times10}_{j.rnd,M_1}$

SDR	Easy	Hard
SPT	0	44.33
LPT	25.33	3.33
LWR	0	52.33
MWR	74.67	0

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

SDR	Easy	Hard
SPT	20.15	20.90
LPT	4.10	49.25
LWR	58.58	5.60
MWR	17.16	24.63

4.3 Consistency of problem instances

The intersection of pairwise SDRs being simultaneously easy or hard are given in Tables 4.4 to 4.7, 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 7.6.

For the corresponding 10×10 training set (cf. Tables 4.6 and 4.7), the intersections between SDRs from 6×5 (cf. Tables 4.4 and 4.5) 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. Table 4.6), 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. Table 4.7), 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?

Table 4.4: Percentage (%) of 6×5 training instances classified as easy simultaneously, defined by Eq. (4.4). Note, each problem space consists of $N_{\text{train}} = 500$.

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

SDR	SPT	LPT	LWR	MWR
SPT	8.90	2.04	1.02	5.44
LPT	2.04	22.06	1.14	17.46
LWR	1.02	1.14	3.64	2.12
MWR	5.44	17.46	2.12	65.30

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

SDR	SPT	LPT	LWR	MWR
SPT	2.88	0.82	0.34	2.12
LPT	0.82	24.42	0.54	18.96
LWR	0.34	0.54	2.10	1.46
MWR	2.12	18.96	1.46	70.70

(c) $\mathcal{P}^{6\times5}_{j.rnd,J_1}$

SDR	SPT	LPT	LWR	MWR
SPT	8.22	3.20	2.46	5.12
LPT	3.20	27.92	3.22	22.10
LWR	2.46	3.22	7.80	4.94
MWR	5.12	22.10	4.94	56.00

(d) $\mathcal{P}^{6\times5}_{j.rnd,M_1}$

SDR	SPT	LPT	LWR	MWR
SPT	2.28	0.60	0.24	1.20
LPT	0.60	31.68	0.36	26.60
LWR	0.24	0.36	1.10	0.64
MWR	1.20	26.60	0.64	64.96

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

SDR	SPT	LPT	LWR	MWR
SPT	23.02	2.76	15.00	4.90
LPT	2.76	8.44	6.12	4.02
LWR	15.00	6.12	47.60	7.46
MWR	4.90	4.02	7.46	20.94

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

SDR	SPT	LPT	LWR	MWR
SPT	0.94	0.30	0.88	0.06
LPT	0.30	13.22	11.74	0.16
LWR	0.88	11.74	85.18	0.36
MWR	0.06	0.16	0.36	0.48

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

SDR	SPT	LPT	LWR	MWR
SPT	22.14	4.24	21.44	3.88
LPT	4.24	21.52	5.78	15.38
LWR	21.44	5.78	35.64	4.62
MWR	3.88	15.38	4.62	21.38

(h) $\mathcal{P}^{6\times5}_{f.mc}$

SDR	SPT	LPT	LWR	MWR
SPT	10.64	5.28	3.74	7.96
LPT	5.28	18.46	8.16	10.08
LWR	3.74	8.16	49.04	4.34
MWR	7.96	10.08	4.34	21.46

(i) $\mathcal{P}^{6\times5}_{f.mxc}$

SDR	SPT	LPT	LWR	MWR
SPT	12.58	0.82	12.42	0.76
LPT	0.82	26.30	1.08	25.10
LWR	12.42	1.08	31.60	0.98
MWR	0.76	25.10	0.98	29.66

Table 4.5: Percentage (%) of 6×5 training instances classified as hard simultaneously, defined by Eq. (4.4). Note, each problem space consists of $N_{\text{train}} = 500$.

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

SDR	SPT	LPT	LWR	MWR
SPT	30.38	5.24	21.08	0.04
LPT	5.24	15.24	9.78	0.10
LWR	21.08	9.78	54.18	0.08
MWR	0.04	0.10	0.08	0.20

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

SDR	SPT	LPT	LWR	MWR
SPT	37.54	4.46	25.56	0.02
LPT	4.46	9.70	6.18	0.04
LWR	25.56	6.18	52.82	0.06
MWR	0.02	0.04	0.06	0.06

(c) $\mathcal{P}^{6\times5}_{j.rnd,J_1}$

SDR	SPT	LPT	LWR	MWR
SPT	38.20	7.34	26.46	0.40
LPT	7.34	14.18	9.10	0.46
LWR	26.46	9.10	46.70	0.48
MWR	0.40	0.46	0.48	0.92

(d) $\mathcal{P}^{6\times5}_{j.rnd,M_1}$

SDR	SPT	LPT	LWR	MWR
SPT	43.08	3.00	31.42	0.04
LPT	3.00	5.72	3.62	0
LWR	31.42	3.62	51.12	0.04
MWR	0.04	0	0.04	0.10

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

SDR	SPT	LPT	LWR	MWR
SPT	22.90	11.70	3.74	6.24
LPT	11.70	41.82	5.64	16.14
LWR	3.74	5.64	7.50	1.16
MWR	6.24	16.14	1.16	27.82

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

SDR	SPT	LPT	LWR	MWR
SPT	44.38	3.48	0	22.20
LPT	3.48	7.28	0	3.90
LWR	0	0	0	0
MWR	22.20	3.90	0	48.42

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

SDR	SPT	LPT	LWR	MWR
SPT	36.44	12.48	2.74	18.22
LPT	12.48	24.08	0.94	14.28
LWR	2.74	0.94	2.80	0.90
MWR	18.22	14.28	0.90	36.70

(h) $\mathcal{P}^{6\times5}_{f.mc}$

SDR	SPT	LPT	LWR	MWR
SPT	49.20	12.94	0	23.16
LPT	12.94	18.98	0	9.76
LWR	0	0	0	0
MWR	23.16	9.76	0	31.76

(i) $\mathcal{P}^{6\times5}_{f.mxc}$

SDR	SPT	LPT	LWR	MWR
SPT	45.16	12.24	7.48	11.34
LPT	12.24	24.78	0.52	14.10
LWR	7.48	0.52	7.68	0.26
MWR	11.34	14.10	0.26	22.48

Table 4.6: Percentage (%) of 10 × 10 training instances classified as easy simultaneously, defined by Eq. (4.4). Note, each problem space consists of $N_{\text{train}} = 300$.

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

SDR	SPT	LPT	LWR	MWR
SPT	2.67	0.33	0	2.33
LPT	0.33	10.33	0	10.33
LWR	0	0	0.67	0.33
MWR	2.33	10.33	0.33	86.33

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

SDR	SPT	LPT	LWR	MWR
SPT	1.00	0.33	0	1.00
LPT	0.33	6.67	0	5.00
LWR	0	0	0	0
MWR	1.00	5.00	0	92.33

(c) $\mathcal{P}^{10\times10}_{j.rnd,J_1}$

SDR	SPT	LPT	LWR	MWR
SPT	3.33	1.00	1.33	3.00
LPT	1.00	21.67	1.67	20.33
LWR	1.33	1.67	3.67	3.67
MWR	3.00	20.33	3.67	71.33

(d) $\mathcal{P}^{10\times10}_{j.rnd,M_1}$

SDR	SPT	LPT	LWR	MWR
SPT	0	0	0	0
LPT	0	25.33	0	25.00
LWR	0	0	0	0
MWR	0	25.00	0	74.67

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

SDR	SPT	LPT	LWR	MWR
SPT	20.15	1.49	15.30	1.87
LPT	1.49	4.10	2.99	0.75
LWR	15.30	2.99	58.58	7.09
MWR	1.87	0.75	7.09	17.16

Table 4.7: Percentage (%) of 10 × 10 training instances classified as hard simultaneously, defined by Eq. (4.4). Note, each problem space consists of $N_{\text{train}} = 300$.

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

SDR	SPT	LPT	LWR	MWR
SPT	27.00	4.67	17.67	0
LPT	4.67	13.67	9.00	0
LWR	17.67	9.00	59.33	0
MWR	0	0	0	0

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

SDR	SPT	LPT	LWR	MWR
SPT	31.67	3.00	23.33	0
LPT	3.00	9.33	5.33	0
LWR	23.33	5.33	59.00	0
MWR	0	0	0	0

(c) $\mathcal{P}^{10\times10}_{j.rnd,J_1}$

SDR	SPT	LPT	LWR	MWR
SPT	40.00	7.00	27.00	0
LPT	7.00	11.33	9.67	0
LWR	27.00	9.67	48.67	0
MWR	0	0	0	0

(d) $\mathcal{P}^{10\times10}_{j.rnd,M_1}$

SDR	SPT	LPT	LWR	MWR
SPT	44.33	1.67	28.00	0
LPT	1.67	3.33	2.00	0
LWR	28.00	2.00	52.33	0
MWR	0	0	0	0

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

SDR	SPT	LPT	LWR	MWR
SPT	20.90	12.31	2.61	4.85
LPT	12.31	49.25	5.22	14.93
LWR	2.61	5.22	5.60	1.49
MWR	4.85	14.93	1.49	24.63

4.4 Discussion and Conclusion

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 quartile (cf. Table 4.1b) 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 7.6 will attempt to explain the evidence show in Tables 4.2 to 4.7.

Note, this section gave the definition of what constitutes an 'easy' and 'hard' schedule. Since these are based on four SDRs (cf. Eq. (4.1)) 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 Eq. (4.2) are based on the first and third quartiles 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.

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.

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

5

Evolutionary Learning of CDRs

Genetic algorithms (GA) are one of the most widely used approaches in JSP literature (Pinedo, 2008). GA is search heuristic that is inspired by the process of natural selection, and is a subclass of evolutionary algorithms (EA), which generate solutions to optimisation problems using techniques based on natural evolution, such as inheritance, mutation, selection, and crossover.

When applying GAs to JSP, an extensive number of schedules need to be evaluated, and even for low dimensional JSP, it can quickly become computationally infeasible. GAs can be used directly on schedules (Ak and Koc, 2012, Cheng et al., 1996, 1999, Qing-dao-er ji and Wang, 2012, Tsai et al., 2007). However, then 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, special care must be taken 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, 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).

The most predominant approach in hyper-heuristics, a framework of creating *new* heuristics from a set of predefined heuristics, is genetic programming (Burke et al., 2013). Dispatching rules based genetic algorithms (DRGA) (Dhingra and Chandna, 2010, Nguyen et al., 2013, Vázquez-Rodríguez and Petrovic, 2009) are a special case of genetic program-

ming (Koza and Poli, 2005), where GAs are applied indirectly to JSP via dispatching rules, i.e., where a solution is no longer a *proper* schedule but a *representation* of a schedule via applying certain DRs consecutively.

As previously discussed in Chapter 1, there are two main viewpoints on how to approach scheduling problems: *i)* tailored algorithms where schedules are built for one problem instance at a time, and *ii)* general algorithms where schedules are built for all problem instances at once. For tailored algorithms 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 Paper I. Expanding on that work, this chapter will explore a general algorithms construction viewpoint where there is no feature set collected beforehand since the learning model is optimised directly via evolutionary search. The framework was first reported in Paper IV, and later used to improve the preference models in Paper V.

Evolutionary search 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. For that reason, it can be mediated with the use of preference learning, as discussed in Paper II, albeit for traditional test functions suite (in particular Rosenbrock's function and Sphere model).

5.1 Experimental setting

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 Paper IV, was to automate that selection similar to DRGA, by translating dispatching rules into measurable features and optimising what their contribution should be via evolutionary search, i.e., optimise the weights $\mathbf{w}$ in Eq. (2.12) directly using covariance matrix adaptation evolution strategy (CMA-ES) by Hansen and Ostermeier (2001), which has been proven to be a very efficient numerical optimisation technique. The framework is straight forward and easy to implement and shows promising results.

Moreover, Section 5.2 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.

Preliminary experiments were first reported in Papers IV and V using: *i)* standard set-up of parameters of the CMA-ES optimisation, and *ii)* runtime was limited to 288 hours on a cluster for each $\mathcal{P}^{6\times5}_{\text{train}}$ problem set given in Table 3.2, where in every case the optimisation reached its maximum walltime. Paper IV had one model for all K time steps, whereas Paper V used a different stepwise model at each dispatch iteration.

Various data distributions from Chapter 3 are investigated. Due to computational cost, only 6×5 instances were initially considered. However, since then the scheduling subroutines have been made more time-efficient, making $\mathcal{P}^{10\times10}_{\text{train}}$ applicable in a reasonable amount of time. CMA-ES models will be mostly trained on the lower dimension, 6×5, and only the general random 10×10 JSP case is explored. Finally, to check robustness, models are validated on benchmark tests sets from the OR-Library (cf. Section 3.3).

5.2 Performance measures

Generally, evolutionary search only needs to minimise the expected fitness value. However, the approach in Paper I was to use the known optimum to correctly label which operations' features were optimal when compared to other possible operations. Therefore, it would be of interest to inspect if there is any performance edge gained by incorporating optimal labelling in evolutionary search. Therefore, two objective functions will be considered, namely,

$$\text{ES}.C_{\max} := \min \mathbb{E}\Big\{ C_{\max} \Big\} \tag{5.1a}$$

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

$$\text{ES}.\rho := \min \mathbb{E}\Big\{ \rho \Big\} \tag{5.1b}$$

which optimises w.r.t. the resulting $C_{\max}$ scaled to its true optimum, i.e., Eq. (2.17).

5.3 Experimental results

Main statistics of the experimental run are given in Table 5.1 and depicted in Fig. 5.1 for both approaches. In addition, evolving decision variables, here weights $\mathbf{w}$ for Eq. (2.12), are depicted in Fig. 5.2.

The evolution of fitness per generation from the CMA-ES optimisation of Eq. (5.1) is depicted in Fig. 5.1. Note, most problem spaces reached their allotted maximum func-

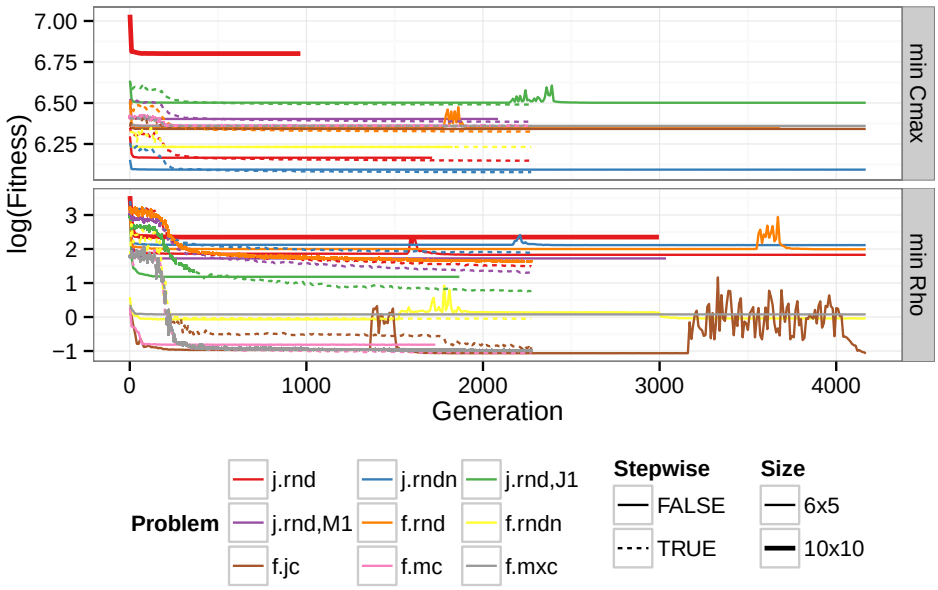


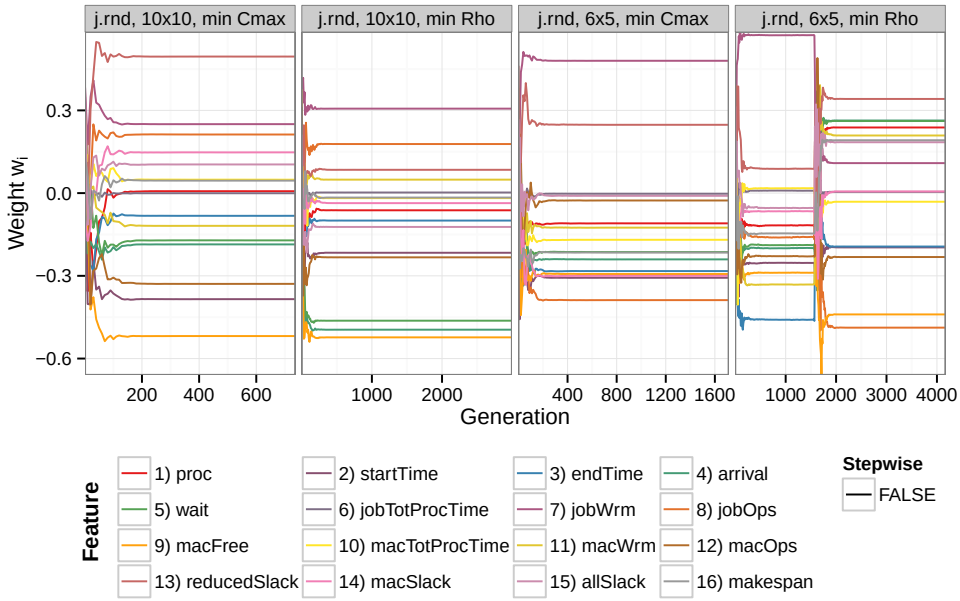
Figure 5.1: Log fitness for optimising w.r.t. Eq. (5.1), per generation of the CMA-ES optimisation.

tion evaluations* without converging. In fact several problem spaces, e.g., $\mathcal{P}^{6\times5}_{f.rnd}$, needed restarting during the optimisation process. Notice $\mathcal{P}^{6\times5}_{j.rnd,J_1}$ especially, then ES.$C_{\max}$ needs more than twice the amount of function evaluations than using ES.ρ as an objective.

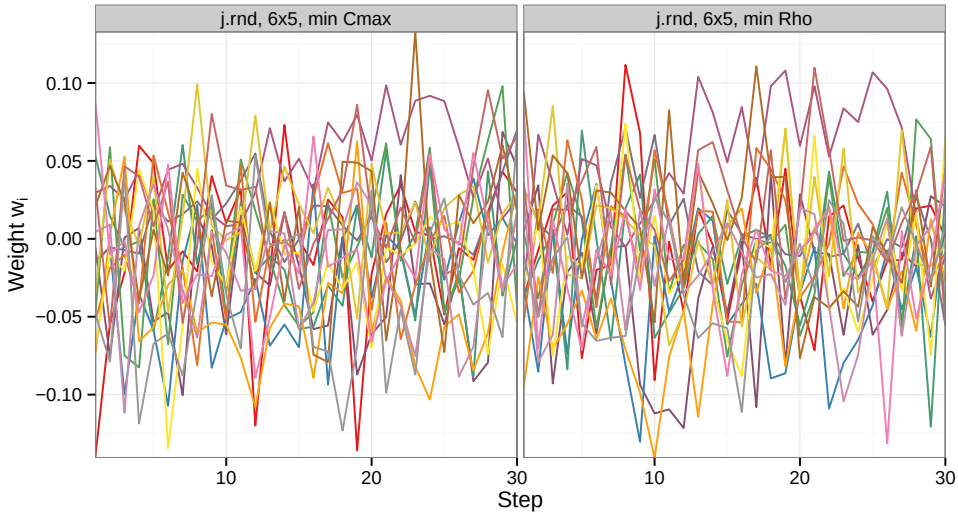
Furthermore, the evolution of the decision variables $\mathbf{w}$ are depicted in Fig. 5.2. As one can see, the relative contribution for each weight clearly differs between problem spaces. Note, that in the case of $\mathcal{P}^{6\times5}_{j.rnd}$ (cf. Figs. 5.1 and 5.2a), CMA-ES restarts around generation 1,600 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. This can be expected due to the fact some features in Table 2.2 are a linear combination of others, e.g. $\varphi_3 = \varphi_1 + \varphi_2$. Moreover, from Fig. 5.2b we see that the evolution of weights w.r.t. each step k is quite erratic and Eq. (5.1a) is somewhat dissimilar to its Eq. (5.1b) counterpart, yet their resulting ρ values are not significantly different. Most likely, this can be explained by feature equivalence.

In order to compare the two objective functions in Eq. (5.1), the best weights reported were used for Eq. (2.12) on the corresponding training and test set. Its box-plot of deviation from optimality, ρ, defined by Eq. (2.17), is depicted in Fig. 5.3a and Table 5.2

*Computational budget was set to 50,000 function evaluations



(a) evolution of weights for $\mathcal{P}^{n\times m}_{j.rnd}$ time independent models



(b) stepwise evolultion of final weights for $\mathcal{P}^{6\times 5}_{j.rnd}$ time dependent models

Figure 5.2: Evolution of weights for features (given in Table 2.2). Note, weights are normalised such that $\|\mathbf{w}\| = 1$.

Table 5.1: Final results for CMA-ES optimisation; total number of generations and function evaluations and its resulting fitness value for both objective functions.

			optimise Eq. (5.1a)			optimise Eq. (5.1b)		
$\mathcal{P}_{\text{train}}$	|Model|*		#gen	#eval	ES.$C_{\max}$	#gen	#eval	ES.ρ
j.rnd	6x5	1	1713	20544	476.34	4168	50004	6.23
j.rnd	6×5	K	2274	50006	467.62	2274	50006	4.38
j.rndn	6×5	1	4168	50004	442.99	4168	50004	8.28
j.rndn	6×5	K	2274	50006	435.87	2274	50006	6.60
$j.rnd, J_1$	6×5	1	4168	50004	666.03	1867	22392	3.26
$j.rnd, J_1$	6×5	K	2274	50006	658.57	2274	50006	2.13
$j.rnd, M_1$	6×5	1	2086	25020	603.46	3037	36432	5.60
$j.rnd, M_1$	6×5	K	2274	50006	592.85	2274	50006	3.66
f.rnd	6×5	1	3683	44184	570.15	4168	50004	7.34
f.rnd	6×5	K	2274	50006	558.37	2274	50006	5.07
f.rndn	6×5	1	1829	21936	508.63	4168	50004	0.92
f.rndn	6×5	K	2274	50006	508.72	2274	50006	0.94
f.jc	6×5	1	4168	50004	567.80	4168	50004	0.34
f.jc	6×5	K	2274	50006	567.74	2274	50006	0.36
f.mc	6×5	1	1796	21540	579.38	1731	20760	0.44
f.mc	6×5	K	2274	50006	578.85	2274	50006	0.34
f.mxc	6×5	1	4168	50004	578.35	4168	50004	1.08
f.mxc	6×5	K	2274	50006	578.09	2274	50006	0.37
j.rnd	10×10	1	966	11592	898.22	2997	35952	10.49

*Models are either stepwise (i.e. total of K models) or fixed throughout the dispatching process.

presents its main statistics: mean, median, standard deviation, minimum and maximum values.

In most cases (except for $\mathcal{P}^{6\times5}_{f.rndn}$, $\mathcal{P}^{6\times5}_{f.jc}$) there was a significant difference w.r.t. lower mean ρ, when using a separate model for each time step, as is to be expected as the optimal dispatching rules used in the beginning of the scheduling process may not necessarily be the same as in the middle, or end of the schedule. Alas, stepwise models aren't appropriate when inspecting robustness towards different problem spaces.** Hence, a single model for all iterations is preferred.

Regarding the choice of objective function in Eq. (5.1), then there is no statistical difference between adopting either objective function with respect to training and test set, save

**Note, time dependant models are inapplicable for OR-Library, since their size $n \times m$ doesn't match any size in the benchmark set. However, this is irrelevant for time independent models.

for time independent $\mathcal{P}^{6\times5}_{f.mxc}$. Now, when applying the time independent models to the OR-Library benchmark data sets, depicted in Fig. 5.3b, then we see a clear performance boost when using Eq. (5.1b) in: *i)* $\mathcal{P}^{6\times5}_{f.jc}$, $\mathcal{P}^{6\times5}_{f.mc}$ and $\mathcal{P}^{6\times5}_{f.mxc}$ for JSP, and *ii)* $\mathcal{P}^{6\times5}_{f.jc}$ and $\mathcal{P}^{6\times5}_{f.mxc}$ for FSP. Therefore, minimisation of expectation of ρ, is preferred over simply using the unscaled resulting makespan. Also it's noted that in Paper IV, then $\mathcal{P}^{6\times5}_{f.jc}$ optimised w.r.t. Eq. (5.1a) gave a considerably worse results, since the optimisation got trapped in a local minima.

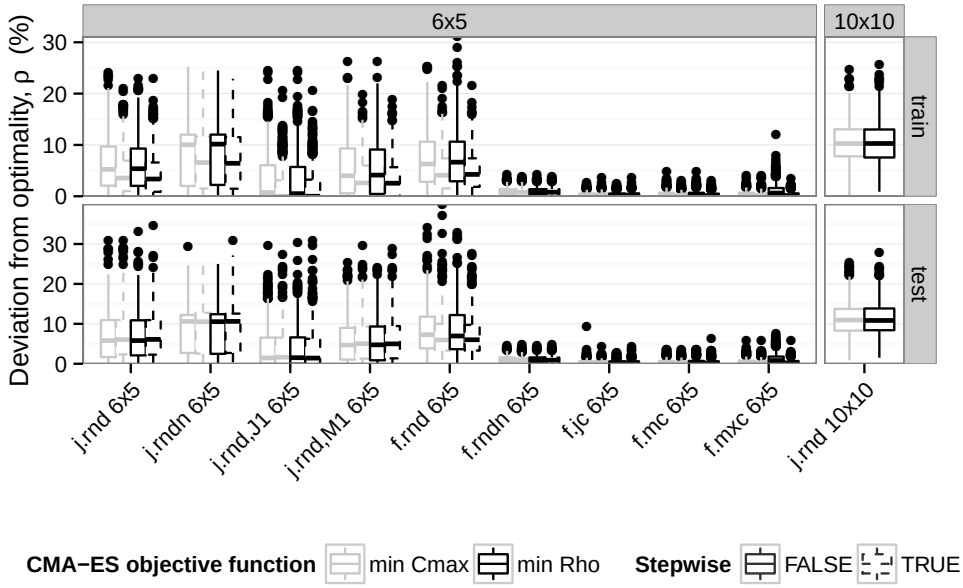
Furthermore, notice how $\mathcal{P}^{6\times5}_{f.mxc}$ obtains a significantly better mean ρ (from 52.8% down to 24.46%) for the JSP (cf. Fig. 5.3b) then it did for it's corresponding problem space, which was the only setting where Eq. (5.1b) was significantly different than Eq. (5.1a) (worsening by $\approx 1\%$).

5.4 Discussion and conclusions

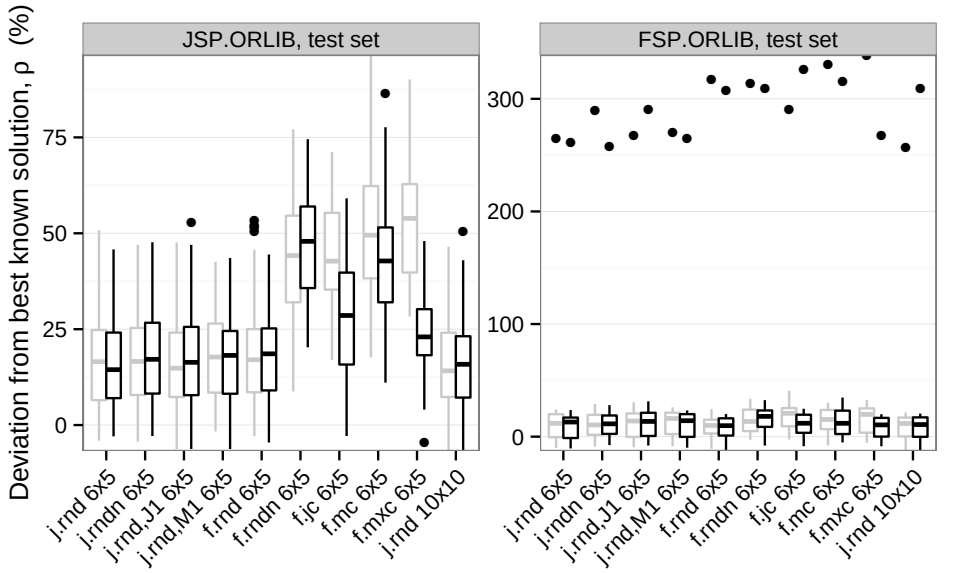
Data distributions considered in this study incorporated more problem spaces than in its initial reports in Paper IV. Furthermore, both time dependent and independent models were optimised with CMA-ES. The former generally obtained lower mean ρ. However, the latter was often equally good, with the added benefit of being applicable for higher (or lower) dimensionality, which was then tested using the benchmark set from the OR-Library.

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. Eq. (5.1)), save for time independent $\mathcal{P}^{6\times5}_{f.mxc}$, when applying the models to their corresponding training and test set. However, preliminary experiments in Paper IV and application on unseen data sets from the OR-Library, showed great improvement when using Eq. (5.1b) over Eq. (5.1a). Implying, even though CMA-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 Paper III. Thus normalising the objective function would help the evolutionary search to deviate from giving too much weight for problematic problem instances.

The main drawback of using evolutionary search for finding optimal weights for Eq. (2.12) is how computationally expensive it is to evaluate the mean expected fitness. Even for a low problem dimension such as 6-job 5-machine JSP, each optimisation run reached their maximum allotted function evaluations without converging. Now, 6×5 JSP requires 30 sequential operations where at each time step there are up to 6 jobs to choose from, in fact its complexity is $O(n!^m)$ (Giffler and Thompson, 1960) making it



(a) Models applied to corresponding training and test set



(b) Models applied to ORLIB test set

Figure 5.3: Box-plot for deviation from optimality, ρ, when implementing the final weights obtained from CMA-ES optimisation, using objective functions from Eq. (5.1).

computationally infeasible to apply this framework for higher dimensions as is. Especially, considering that it's preferred to run these experiments several times – e.g. in Paper IV $\mathcal{P}^{6\times5}_{f.jc}$ got stuck in local minima for ES.$C_{\max}$, which could have been avoided by restarting the optimisation. 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 Paper II. This could reduce the computational cost of the evolutionary search considerably, making it feasible to conduct the experiments from Section 5.3 for problems of higher dimensions, e.g., with these adjustments it is possible to train on 10×10 with greater ease, or even considering even higher dimensions, e.g. 14×14.

Table 5.2: Main statistics for Fig. 5.3a

$\mathcal{P}_{\text{train}}$	|Model|		Eq. (5.1)	Min.	1st Qu.	Median	Mean	3rd Qu.	Max.
j.rnd	6×5	1	ES.$C_{\max}$	0.00	1.96	5.62	6.65	10.38	30.77
j.rnd	6×5	K	ES.$C_{\max}$	0.00	1.47	4.71	5.92	8.48	30.77
j.rnd	6×5	1	ES.ρ	0.00	2.04	5.59	6.55	9.97	33.10
j.rnd	6×5	K	ES.ρ	0.00	1.39	4.64	5.74	8.72	34.75
j.rndn	6×5	1	ES.$C_{\max}$	0.00	2.25	10.36	8.73	12.08	29.38
j.rndn	6×5	K	ES.$C_{\max}$	0.00	1.93	9.54	7.89	12.02	25.67
j.rndn	6×5	1	ES.ρ	0.00	2.25	10.37	8.62	12.22	25.00
j.rndn	6×5	K	ES.ρ	0.00	1.95	9.71	7.88	12.04	30.85
$j.rnd, J_1$	6×5	1	ES.$C_{\max}$	0.00	0.00	1.03	3.64	6.27	29.74
$j.rnd, J_1$	6×5	K	ES.$C_{\max}$	0.00	0.00	0.51	2.96	4.74	27.39
$j.rnd, J_1$	6×5	1	ES.ρ	0.00	0.00	0.94	3.56	6.12	30.43
$j.rnd, J_1$	6×5	K	ES.ρ	0.00	0.00	0.57	2.99	4.53	30.86
$j.rnd, M_1$	6×5	1	ES.$C_{\max}$	0.00	0.71	4.35	5.78	9.12	26.25
$j.rnd, M_1$	6×5	K	ES.$C_{\max}$	0.00	0.38	3.67	4.84	7.51	29.56
$j.rnd, M_1$	6×5	1	ES.ρ	0.00	0.63	4.42	5.76	9.16	26.25
$j.rnd, M_1$	6×5	K	ES.ρ	0.00	0.35	3.77	4.86	7.77	28.87
f.rnd	6×5	1	ES.$C_{\max}$	0.00	3.24	6.76	7.74	11.23	34.24
f.rnd	6×5	K	ES.$C_{\max}$	0.00	2.31	5.13	6.10	8.62	39.95
f.rnd	6×5	1	ES.ρ	0.00	3.30	6.79	7.90	11.45	32.88
f.rnd	6×5	K	ES.ρ	0.00	2.32	5.15	5.97	8.76	27.68
f.rndn	6×5	1	ES.$C_{\max}$	0.00	0.39	0.80	1.00	1.41	4.66
f.rndn	6×5	K	ES.$C_{\max}$	0.00	0.39	0.81	1.00	1.40	4.86
f.rndn	6×5	1	ES.ρ	0.00	0.39	0.80	1.00	1.41	4.66
f.rndn	6×5	K	ES.ρ	0.00	0.39	0.80	0.99	1.40	4.86
f.jc	6×5	1	ES.$C_{\max}$	0.00	0.00	0.26	0.39	0.55	9.41
f.jc	6×5	K	ES.$C_{\max}$	0.00	0.00	0.26	0.37	0.54	4.25
f.jc	6×5	1	ES.ρ	0.00	0.00	0.25	0.36	0.53	2.95
f.jc	6×5	K	ES.ρ	0.00	0.00	0.26	0.38	0.57	4.25
f.mc	6×5	1	ES.$C_{\max}$	0.00	0.00	0.24	0.46	0.69	4.93
f.mc	6×5	K	ES.$C_{\max}$	0.00	0.00	0.16	0.38	0.57	3.69
f.mc	6×5	1	ES.ρ	0.00	0.00	0.25	0.47	0.69	4.93
f.mc	6×5	K	ES.ρ	0.00	0.00	0.17	0.38	0.56	6.29
f.mxc	6×5	1	ES.$C_{\max}$	0.00	0.00	0.20	0.47	0.74	5.84
f.mxc	6×5	K	ES.$C_{\max}$	0.00	0.00	0.17	0.43	0.65	5.84
f.mxc	6×5	1	ES.ρ	0.00	0.19	0.68	1.14	1.68	12.10
f.mxc	6×5	K	ES.ρ	0.00	0.00	0.17	0.40	0.63	5.84
j.rnd	10×10	1	ES.$C_{\max}$	0.13	7.97	10.57	10.86	13.47	25.35
j.rnd	10×10	1	ES.ρ	0.88	7.99	10.64	10.87	13.40	27.81

Well! I've often seen a cat without a grin; but a grin without a cat!
It's the most curious thing I ever say in my life!

Alice

6 Generating Training Data

When building a complete job-shop schedule, $K = n \cdot m$ dispatches must be made sequentially. A job is placed at the earliest available time slot for its next machine, whilst still fulfilling constraints Ineqs. (2.2) and (2.3). Unfinished jobs are dispatched one at a time according to some heuristic, or policy π. After each dispatch* the schedule's current features (cf. Table 2.2) are updated based on the half-finished schedule. Namely, when implementing Alg. 2, a training set will consist of all features from Table 2.2 at every post-decision state visited in line 6. These collected features are denoted Φ, where,

$$\Phi := \bigcup_{i=1}^{N_{\text{train}}} \bigcup_{k=1}^{K} \bigcup_{J_j \in \mathcal{L}^{(k)}} \left\{ \boldsymbol{\varphi}_j \ : \ \mathbf{x}_i \in \mathcal{P}_{\text{train}}^{n \times m} \right\}. \tag{6.1}$$

6.1 Job-shop tree representation

Continuing with the example from Section 2.3, Fig. 6.1 shows how the first two dispatches could be executed for a 4×5 job-shop from Section 2.3. 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.

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

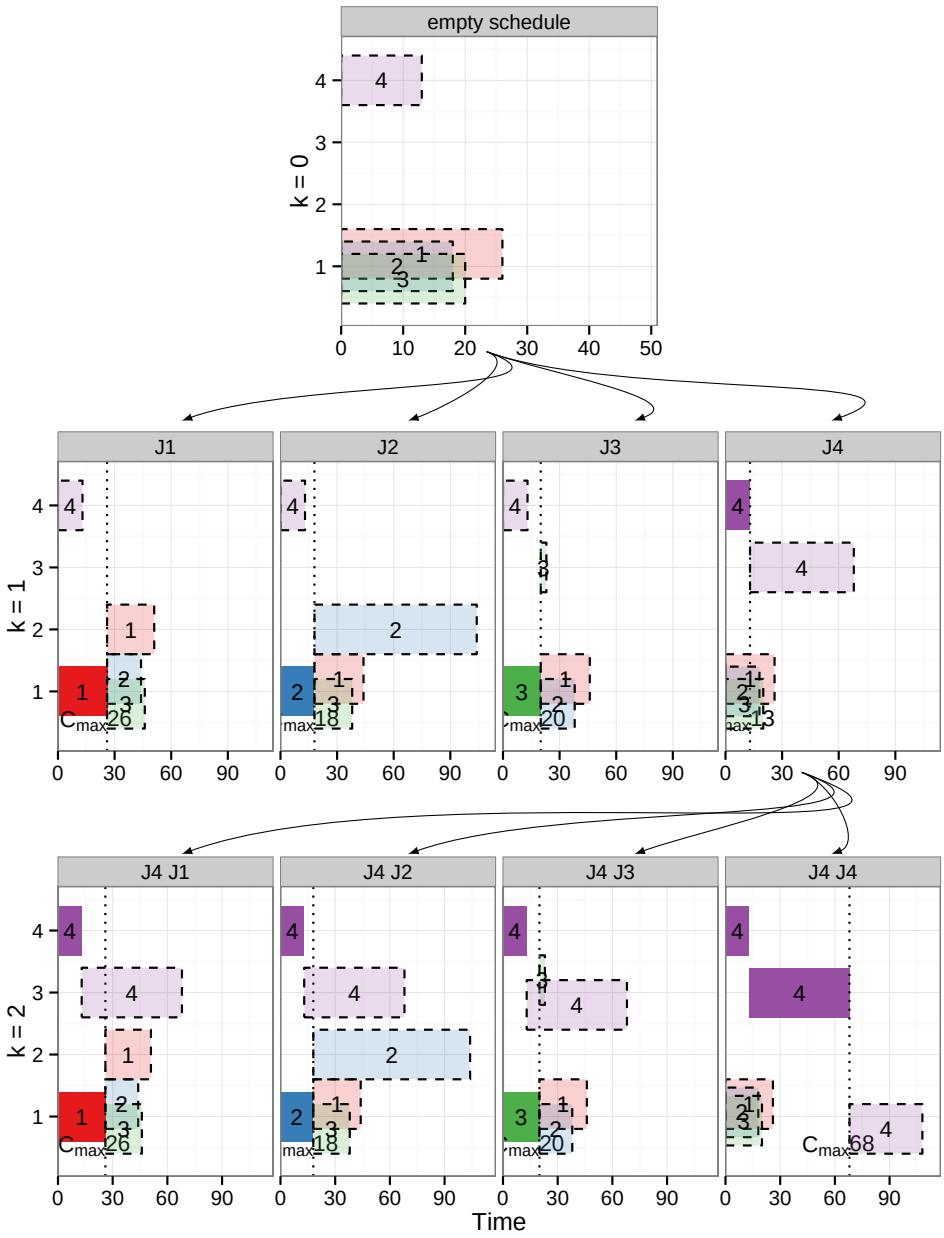


Figure 6.1: Partial Game Tree for job-shop for the first two dispatches. Top layer depicts all possible dispatches (dashed) for an empty schedule. Middle layer depicts all possible dispatches given that one of the dispatches from the layer above has been executed (solid). Bottom layer depicts when job J_4 on machine M_4 has been chosen to be dispatched from the previous layer, moreover it depicts all possible next dispatches from that scenario.

This sort of tree representation is similar to *game trees* (cf. Rosen, 2003) 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 K), 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.

6.2 Labelling schedules w.r.t. optimal decisions

One can easily see that sequence $\boldsymbol{\chi}$ from Eq. (2.8) for task assignments is by no means unique. Inspecting a partial schedule further along in the dispatching process such as in Fig. 2.3, then let's say J_2 would be dispatched next, and in the next iteration J_4. Now this sequence would yield the same schedule as if J_4 would have been dispatched first and then J_2 in the next iteration. This is due to the fact they have non-conflicting machines, which indicates that some of the nodes in game tree can merge. Meanwhile, 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_2 is better and J_4 is worse (or vice versa) since they can both yield the same solution.

Furthermore, 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. (2.17) (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.

6.3 Computational growth

The creation of the game tree for JSP can be done recursively for all possible permutations of dispatches, resulting in a full n-ary tree (since $|\mathcal{L}| \leq n$) of height K. Such an exhaustive search would yield at the most n^K leaf nodes. Worst case scenario being no sub-trees merge.

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

Since the internal vertices (i.e. partial schedules) are only of interest to learn,* the number of those can be at the most $^{n^K-1}/_{n-1}$. Even for small dimensions of n and m the number of internal vertices are quite substantial and thus too computationally expensive to investigate them all. Not to mention that this is done iteratively for all N_{train} problem instances.

Since we know that once a job is processed on all of its machines, then it stops being a contender for future dispatches, therefore the all possible assignments of operations for an $n \times m$ JSP would require an examination of $(n!)^m$ (Giffler and Thompson, 1960), thus a 6×5 problem may have at most $1.93 \cdot 10^{14}$ possible solutions, and for 10×10 problem then it's $3.96 \cdot 10^{65}$ solutions! Thus the factorial growth makes it infeasible for exploring all nodes to completion. However, our training data consist of relatively large N_{train}, so even though we will only pursue one trajectory per instance, then the aggregated training data will give it variety.

6.4 Trajectory sampling strategies

For each feature in Eq. (6.1) we need to keep track of the resulting makespan for its dispatched job. As a result, we obtain the meta-data from Fig. 1.1 as follows,

$$\{\Phi^\pi, \mathcal{Y}^\pi\} := \left\{ \{\boldsymbol{\varphi}_j, C_{\max}^{\pi_\star}(\boldsymbol{\chi}^j)\} \;:\; J_j \in \mathcal{L}^{(k)} \right\}_{k=1}^{K} \in \mathcal{F} \times \mathcal{Y} \tag{6.2}$$

for a single problem instance $\mathbf{x} \in \mathcal{P}_{\text{train}}$, and where $C_{\max}^{\pi_\star}(\boldsymbol{\chi}^j)$ denotes the optimal makespan (i.e. following the expert policy $\pi_\star$) from the resulting post-decision state $\boldsymbol{\chi}^j$.

Due to superabundant possible solutions for a single problem instance, there needs to be some logic based on how to sample the state-space for a valuable outcome. Especially considering the cost of correctly labelling ** each dispatch that is encountered.† Obviously we'd like to inspect optimal solutions as they are what we'd like to mimic. Moreover, since we'd like to infer the footprints in instance space for the SDRs we started doing in Chapter 4, then we will consider them also. Similarly, the weights for Eq. (2.12) that were optimised directly using from evolutionary search (cf. Chapter 5) will also be used.

*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.

**Optimal solutions can be obtained by using a commercial software package by Gurobi Optimization, Inc. (2014), which has a free academic licence. However, GLPK by Free Software Foundation, Inc. (2014) has a free licence. Alas, GLPK has a lacklustre performance w.r.t. speed for solving 10×10 JSP.

†Generally it takes only several hours to collect $N_{\text{train}}^{6\times5} = 500$. Alas, when going to higher dimension, $N_{\text{train}}^{10\times10} = 300$ really becomes an issue, as $\mathcal{P}_{j.rnd}^{10\times10}$ needs a few days, and $\mathcal{P}_{j.rndn}^{10\times10}$ or $\mathcal{P}_{f.rnd}^{10\times10}$ require several weeks!

To clarify, the trajectory sampling strategies for collecting a feature set and its corresponding labelling for Eq. (6.2) are the following:

Optimum trajectory, $\Phi^{\mathbf{OPT}}$ or $\Phi^{\pi_\star}$, at each dispatch some (random) optimal task is dispatched. This is also referred to following the expert policy, $\pi_\star$.

SPT trajectory, $\Phi^{\mathbf{SPT}}$, at each dispatch the task corresponding to shortest processing time is dispatched, i.e., following single priority dispatching rule SPT.

LPT trajectory, $\Phi^{\mathbf{LPT}}$, at each dispatch the task corresponding to largest processing time is dispatched, i.e., following single priority dispatching rule LPT.

LWR trajectory, $\Phi^{\mathbf{LWR}}$, at each dispatch the task corresponding to least work remaining is dispatched, i.e., following single priority dispatching rule LWR.

MWR trajectory, $\Phi^{\mathbf{MWR}}$, at each dispatch the task corresponding to most work remaining is dispatched, i.e., following single priority dispatching rule MWR.

Random trajectory, $\Phi^{\mathbf{RND}}$, at each dispatch some random task is dispatched.

CMA-ES trajectories, $\Phi^{\mathbf{ES}.\rho}$ and $\Phi^{\mathbf{ES}.C_{\max}}$, 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. (2.17), with CMA-ES optimisation from Chapter 5.

All trajectories, $\Phi^{\mathbf{ALL}}$, denotes all aforementioned trajectories were explored, i.e.,

$$\Phi^{\text{ALL}} := \left\{ \Phi^{A} \ : \ \forall A \in \left\{ \pi_\star,\ \text{SPT, LPT, LWR, MWR, RND, ES.}\rho, \text{ES.}C_{\max} \right\} \right\}$$

When following optimal trajectory, then due to the nature of the sequence representation (i.e. $\boldsymbol{\chi}$), the earlier stages for $\mathcal{P}_{j.rnd}$ of the dispatching are more or less equivalent and thus irrelevant (cf. Fig. 7.3). Hence it is appropriate to follow some random optimal path to begin with and then go after some (if not all possible) optimal paths until completion at step K.

In the case of the $\Phi^{\langle \text{SDR} \rangle}$ and $\Phi^{\langle \text{CMA-ES} \rangle}$ trajectories it is sufficient to explore each trajectory exactly once for each problem instance. Whereas, for Φ^{OPT} and Φ^{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 Paper V. However, more SDR-based trajectories are now addressed since, e.g., LWR is considered more favourable for flow-shop rather than MWR (cf. Chapter 4).

The number of features that were collected on a step-by-step basis for $\mathcal{P}_{\text{train}}^{6\times5}$ in Table 3.2 is illustrated in Fig. 6.2. There is an apparent stair-like structure for LWR, in accordance with its motivation (cf. Section 2.4), which is completing jobs advanced in their progress, that is to say minimising $\mathcal{L}$ and from Eq. (6.1) we have $|\Phi| \propto |\mathcal{L}|$. Whereas MWR tries to keep the jobs more balanced, hence more steady $|\mathcal{L}|$, until at $k > (K - n)$ then $|\mathcal{L}| \lesssim (K - k)$, which explains the sharp decent near the end for MWR. Table 6.1 gives the total size for $|\Phi|$, indicating the number of optimisations needed for obtaining $\mathcal{Y}$.

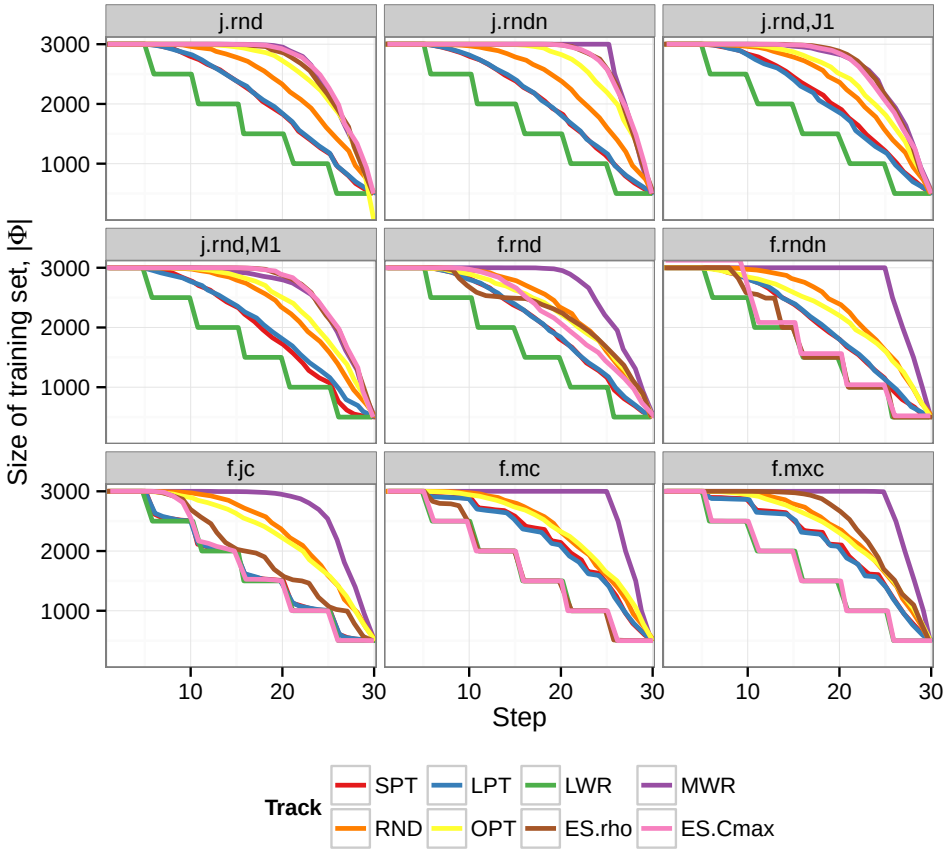


Figure 6.2: Size of 6×5 training set, $|\Phi|$, over different trajectory strategies

Table 6.1: Total number of features in Φ for all K steps. Note '–' denotes not available.

Track	$\mathcal{P}_{\text{train}}^{6\times5}, N_{\text{train}} = 500$									$\mathcal{P}_{\text{train}}^{10\times10}, N_{\text{train}} = 300$		
	j.rnd	*j.rndn*	*j.rnd*, J_1	*j.rnd*, M_1	*f.rnd*	*f.rndn*	*f.jc*	*f.mc*	*f.mxc*	*j.rnd*	*j.rndn*	*f.rnd*
SPT	63197	63074	64560	61320	63287	63123	53678	66995	66216	211351	–	–
LPT	63516	63374	63595	62864	63535	63320	53746	66356	65662	210490	–	–
LWR	52500	52500	52500	52500	52500	52500	52500	52500	5250	165000	–	–
MWR	79230	82500	78327	77934	79288	82500	80546	82498	8245	280739	–	–
RND	71390	71608	71445	71463	71427	71945	71558	71456	7149	252515	–	–
OPT	76592	78176	74109	74069	70037	69180	69716	71602	7102	272858	277717	211763
ES.ρ	78443	81248	78673	78866	68986	55943	60755	53707	74997	277851	–	–
ES.$C_{\max}$	79343	81226	77903	79078	68602	56789	54781	52502	52510	276634	–	–
ALL	564211	573706	561112	558094	537662	515300	497280	517616	537121	1947438	277717	211763

I don't believe there's an atom of meaning in it.

Alice

7

Analysing Solutions

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: *i)* completely contradictory outcomes (i.e. one poor and one good schedule)? Or *ii)* will they more or less follow the their predicted trend? If the latter is the prevalent case, then instances need to be segregated w.r.t. their difficulty, where each has their own learning algorithm implemented, for a more meaningful overall outcome.

Essentially this also 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 for the quality of a solution. If results are contradictory, then 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 to 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: *i)* is there a time of divergence? *ii)* what are 'similar' schedules? *iii)* do similar features yield contradictory outcomes? *iv)* are extra features needed? And *v)* what can be learned from feature behaviour?

Remark: Figures 7.1 and 7.2 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. Paper VI depicts the raw mean as is, albeit only for 10×10 problem spaces, which is also done here for Figs. 7.3 to 7.5 and 7.8.

7.1 Making optimal decisions

In order to create successful dispatching rule, 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 (cf. Φ^{OPT} in Section 6.4), and inspect the evolution of its features (defined in Table 2.2) throughout the dispatching process, which is detailed in Chapter 6. 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. Figure 7.1 depicts how the number of optimal dispatches evolve at each dispatch iteration. Note, that only one optimal trajectory is pursued (chosen at random), hence this is only a lower bound of uniqueness of optimal solutions. As the number of possible dispatches decrease over time, Fig. 7.2 depicts the probability of choosing an optimal dispatch at each iteration.

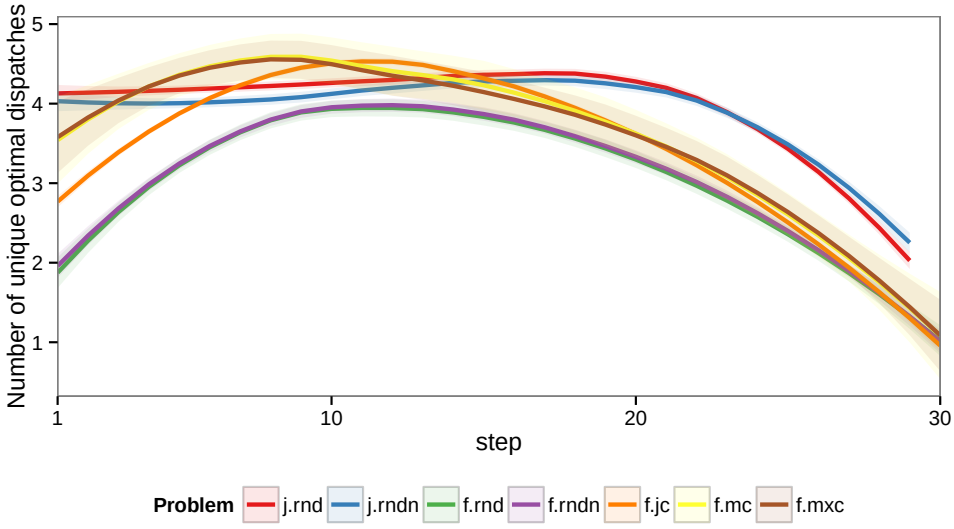
To generalise, we could consider the probability of optimality as a sort of stepwise 'training accuracy.' Then for a given policy π, we'd formalise its optimality (yet still maintaining optimal trajectory) as,

$$\xi^{\star}_{\pi} := \mathbb{E}_{\pi_\star}\Big\{\pi_\star = \pi\Big\} \tag{7.1}$$

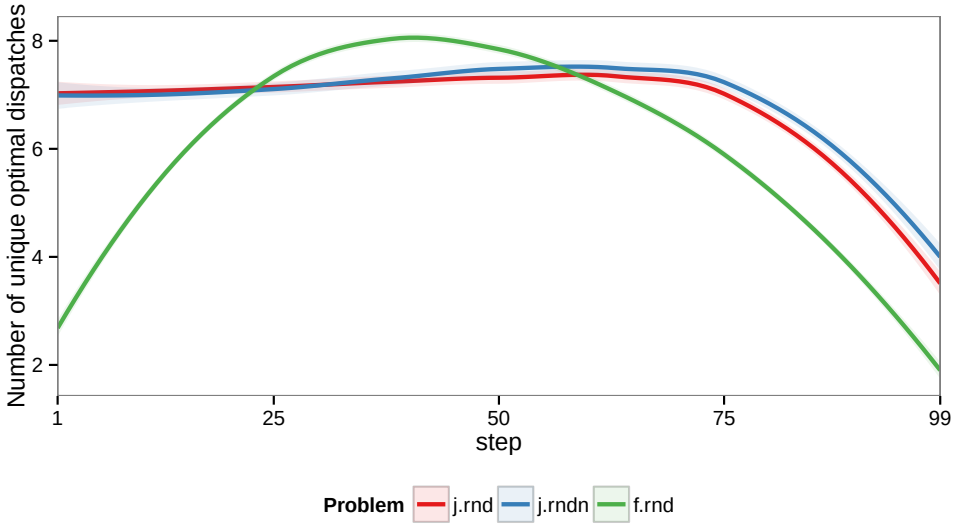
that is to say the mean likelihood of our policy π being equivalent to the expert policy $\pi_\star$, i.e., $Y^{\pi_\star} = Y^{\pi}$. Note, for $\xi^{\star}_{\pi}$ we only need $\{\Phi^{\pi_\star}, \mathcal{Y}^{\pi_\star}\}$ from Eq. (6.2): *i)* retrace $\pi_\star$ as done in Alg. 2, and *ii)* inspect if the job J_{j^*} chosen by π yields the same $C^{\pi_\star(\boldsymbol{\chi}^{j^*})}_{\max}$ as the true optimum, $C^{\pi_\star}_{\max}$.

7.2 Making suboptimal decisions

Looking at Fig. 7.2, $\mathcal{P}^{10\times 10}_{j.rnd}$ 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,

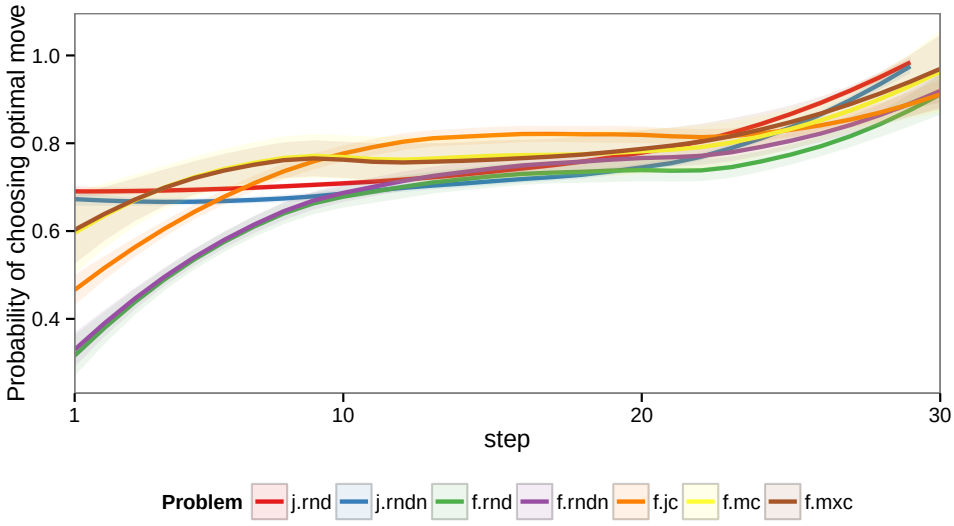


(a) 6×5

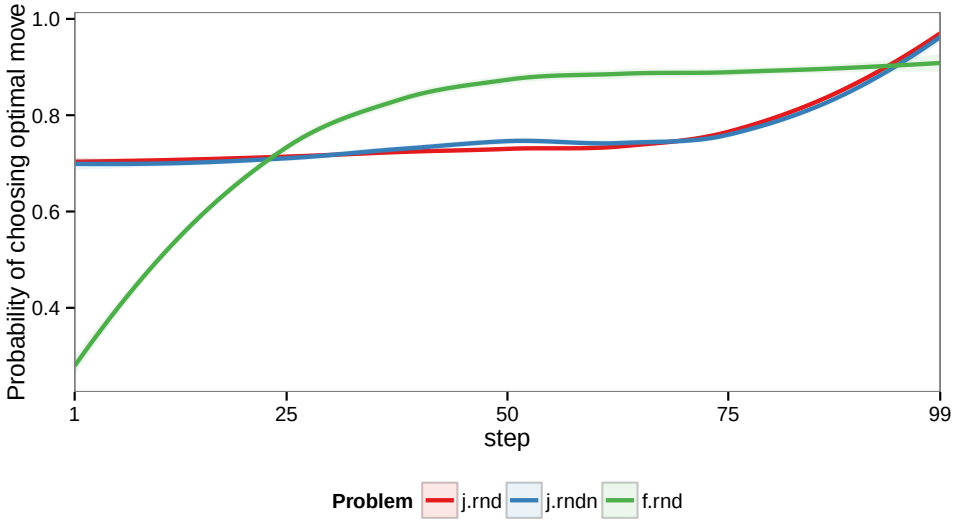


(b) 10×10

Figure 7.1: Number of unique optimal dispatches (lower bound).



(a) 6×5



(b) 10×10

Figure 7.2: Probability of choosing optimal move (at random)

Fig. 7.3 depicts the worst and best case scenario of deviation from optimality, ρ, once you've fallen off the optimal track, defined as follows,

$$\zeta^{\star}_{\min}(k) := \mathbb{E}_{\pi_{\star}}\left\{ \min(\rho) \ : \ \forall C^{j}_{\max} \gtrsim C^{\pi_{\star}}_{\max} \wedge J_j \in \mathcal{L}^{(k)} \right\} \tag{7.2a}$$

$$\zeta^{\star}_{\max}(k) := \mathbb{E}_{\pi_{\star}}\left\{ \max(\rho) \ : \ \forall C^{j}_{\max} \gtrsim C^{\pi_{\star}}_{\max} \wedge J_j \in \mathcal{L}^{(k)} \right\} \tag{7.2b}$$

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. In fact, Fig. 7.5 shows the probability of optimality when following a fixed SDR (i.e. if Eq. (7.1) is conditioned on π instead of $\pi_{\star}$).

It is interesting that for JSP, then making suboptimal decisions makes more of an impact on the resulting makespan as the dispatching process progresses. 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 has 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 get-go. 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.

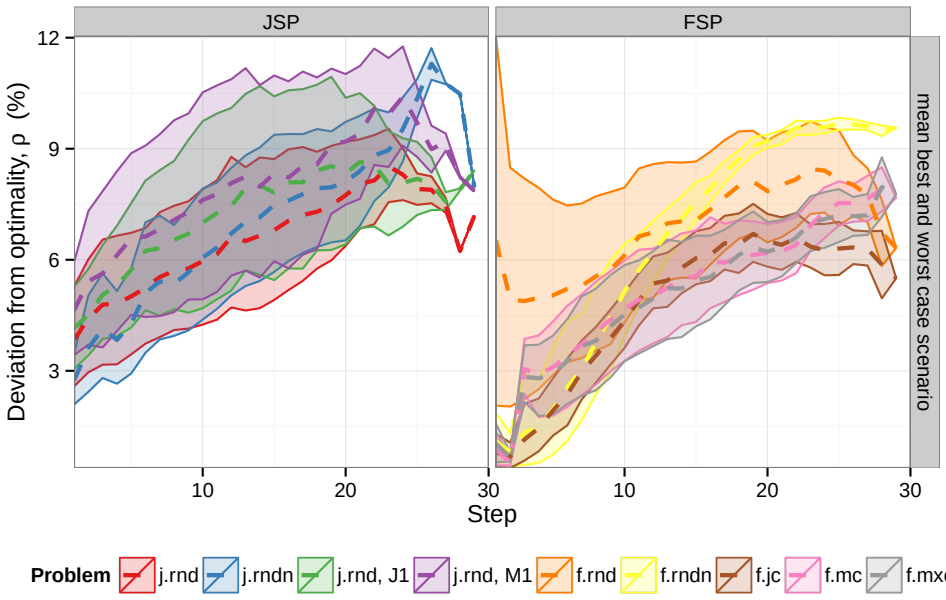
7.3 Optimality of extremal features

The training accuracy from Eq. (7.1) of the aforementioned features from Table 2.2, or probability of a job chosen by an extremal value for a feature being able to yield an optimal makespan on a step-by-step basis, i.e., $\xi^{\star}_{\pm\varphi_i}$, is depicted in Fig. 7.4, for both $\mathcal{P}^{6\times 5}_{j.rnd}$ and $\mathcal{P}^{10\times 10}_{j.rnd}$.* Moreover, the dashed line represents the benchmark of randomly guessing the optimum, $\xi^{\star}_{\text{RND}}$ (cf. Fig. 7.2). Furthermore, the figures are annotated with the corresponding mean deviation from optimality, ρ, for the training set if it were scheduled solely w.r.t. that extremal feature.

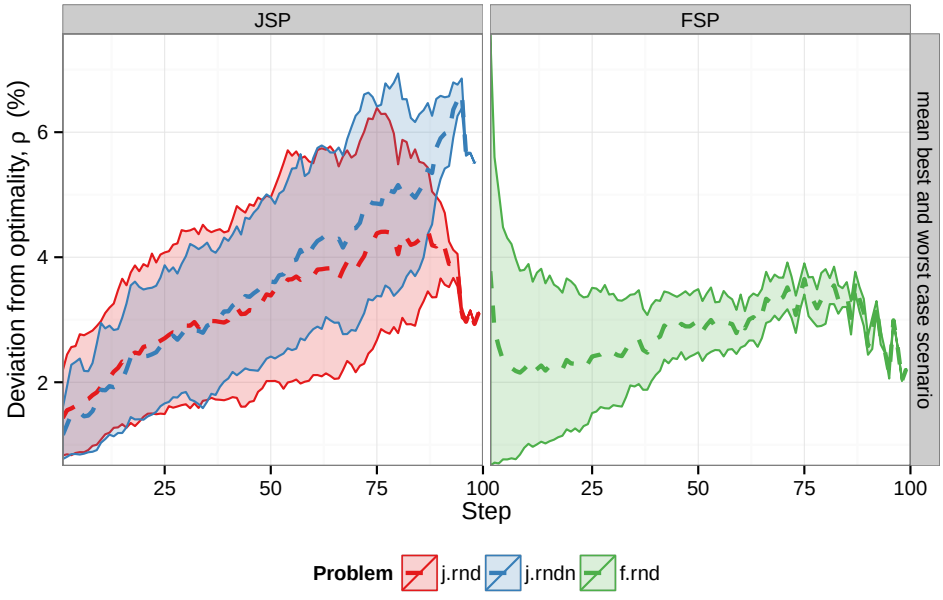
Generally, a high stepwise optimality means a low ρ, e.g., φ_{17}-φ_{24}, save for φ_{22}.** Unfortunately, it's not always so predictable. Take for instance φ_1, then the minimum value gives a better ρ, even though it's unlikelier to be optimal than it's maximum counterpart.

*Additional problem spaces for $\xi^{\star}_{\pm\varphi_i}$ can be found in Shiny application: Features > Extremal.

**Note, φ_{22} is non-informative on its own, as a tight standard deviation implies either consistently high *or* low $C_{\max}$ from the roll-outs.



(a) 6×5



(b) 10×10

Figure 7.3: Mean deviation from optimality, ρ, for best and worst case scenario of when making *one* suboptimal dispatch (i.e. $\zeta^{\star}_{\min}$ and $\zeta^{\star}_{\max}$), depicted as lower and upper bound, respectively. Moreover, mean suboptimal move is given as dashed line.

Before inspecting the local based features further. Notice that the staggering performance edge for φ_{23} is lost when going to a higher dimension (cf. φ_{23} in Fig. 7.4a has $\rho = 1.3\%$ and increases to 8.8% in Fig. 7.4b), implying that 100 random roll-outs for are not sufficient for fully exploring 10×10 state-space, yet highly competitive for 6×5.

Optimality of SDRs

Let's limit ourselves to only features that correspond to SDRs from Section 2.4. Namely, Eq. (2.14) yield: *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. Figure 7.5 depicts the corresponding probabilities from Fig. 7.4 in one graph, for all problem spaces in Table 3.2.

Now, let's bare in mind deviation from optimality, ρ, of applying SDRs throughout the dispatching process (cf. box-plots of which in Fig. 4.1), then there is a some correspondence between high probability of stepwise optimality and low ρ. Alas, this isn't always the case, for $\mathcal{P}^{10\times10}_{j.rnd}$ $\xi^{\star}_{\text{SPT}}$ always outperforms $\xi^{\star}_{\text{LPT}}$ in choosing a dispatch which may result in an optimal schedule. 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.

Since we know that our SDR heuristics aren't perfect, and they're bound to make mistakes at some point. It's interesting to see how that stepwise optimality evolves for its intended trajectory, thereby updating Eq. (7.1) to

$$\xi_{\pi} := \mathbb{E}_{\pi}\Big\{ \pi_{\star} = \pi \Big\} \tag{7.3}$$

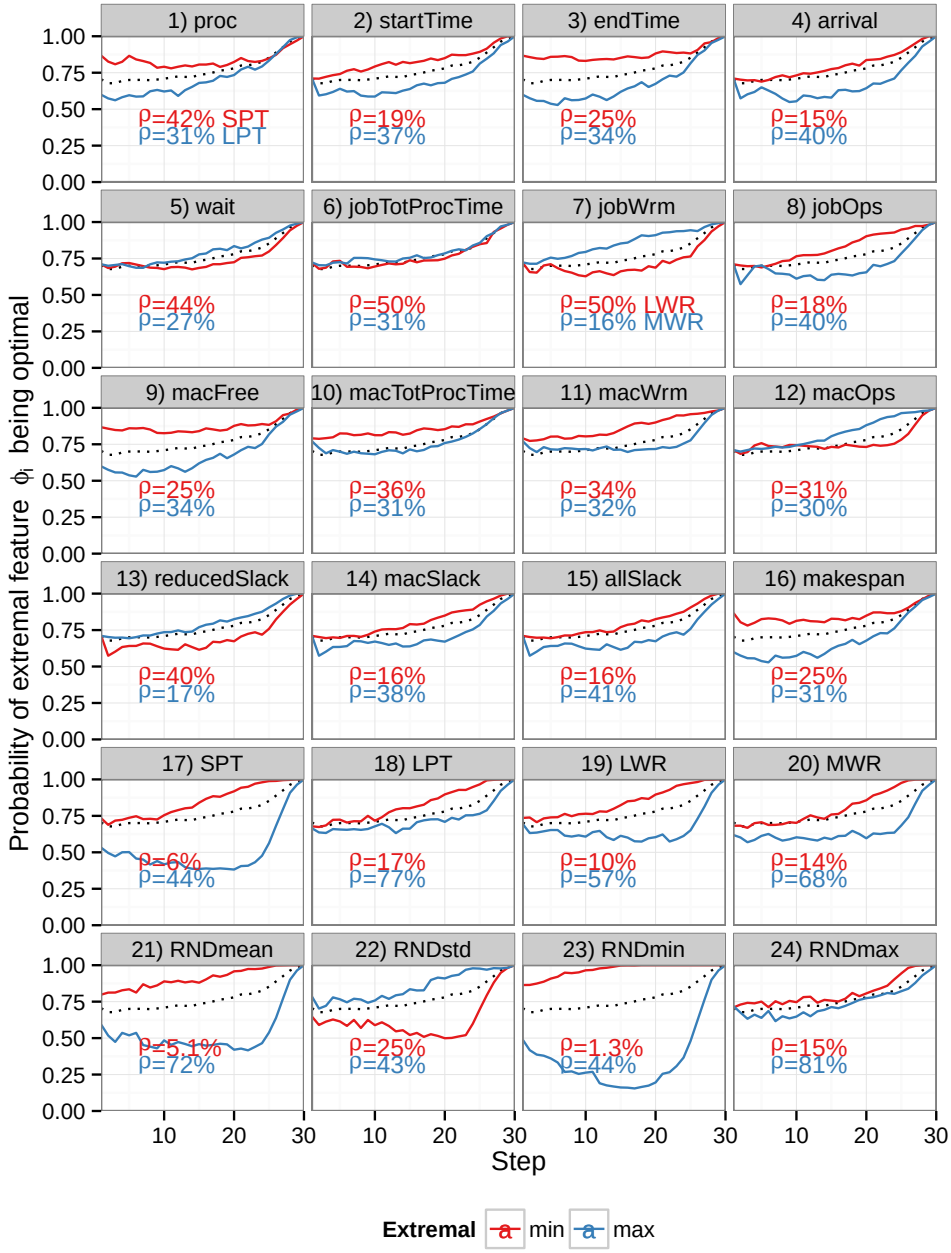
Figure 7.5 shows the difference between $\xi^{\star}_{\langle \text{SDR} \rangle}$ and $\xi_{\langle \text{SDR} \rangle}$. Similarly for Eq. (7.2),

$$\zeta^{\pi}_{\min}(k) := \mathbb{E}_{\pi}\Big\{ \min(\rho) \;:\; \forall C^{\pi_\star}_{\max}(\boldsymbol{\chi}^{j}) \neq C^{\pi_\star}_{\max}(\boldsymbol{\chi}^{j^*}) \wedge j^* = \underset{J_j \in \mathcal{L}^{(k)}}{\operatorname{argmax}}\{\pi(\boldsymbol{\varphi}^{j})\} \Big\} \tag{7.4a}$$

$$\zeta^{\pi}_{\max}(k) := \mathbb{E}_{\pi}\Big\{ \max(\rho) \;:\; \forall C^{\pi_\star}_{\max}(\boldsymbol{\chi}^{j}) \neq C^{\pi_\star}_{\max}(\boldsymbol{\chi}^{j^*}) \wedge j^* = \underset{J_j \in \mathcal{L}^{(k)}}{\operatorname{argmax}}\{\pi(\boldsymbol{\varphi}^{j})\} \Big\} \tag{7.4b}$$

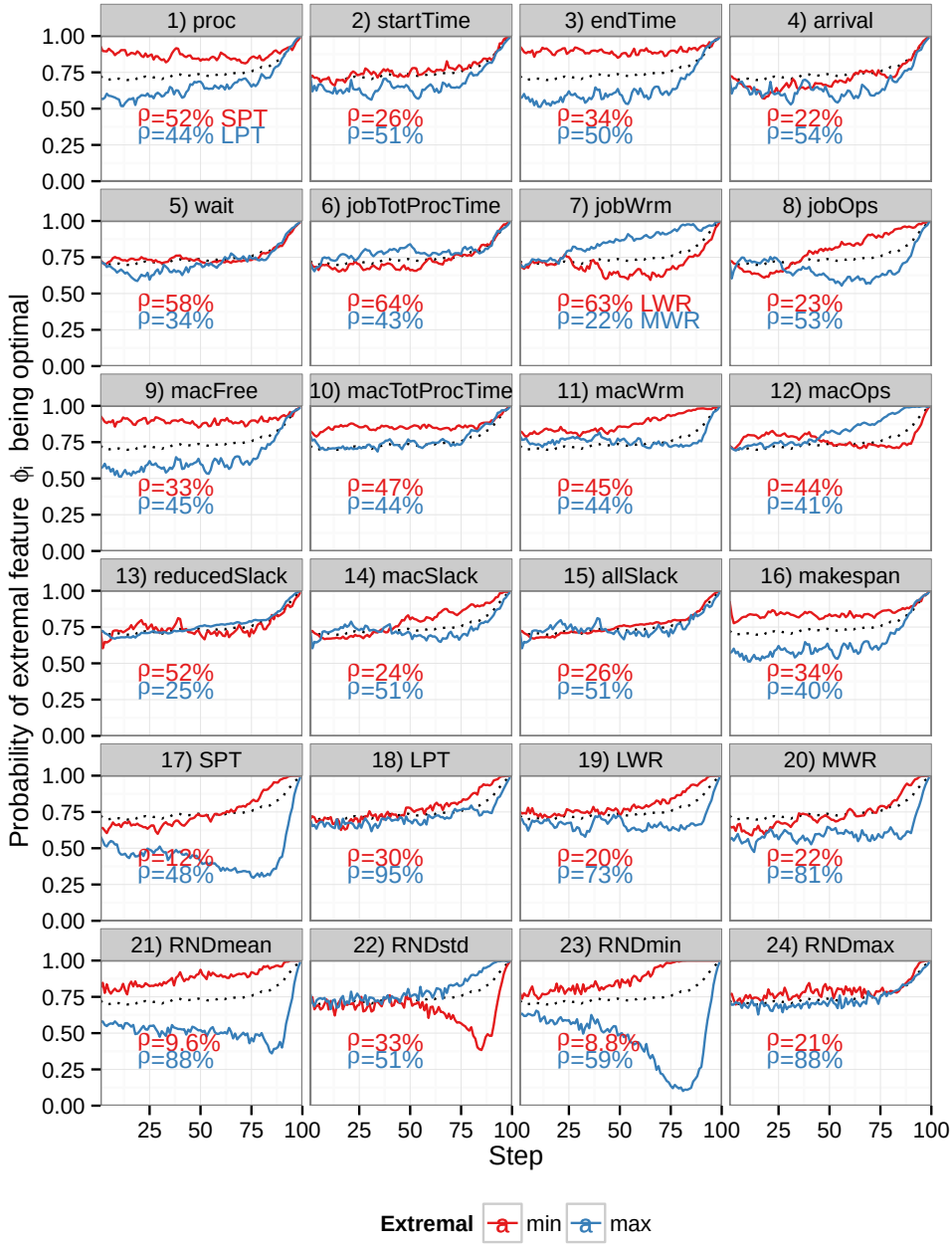
$$\zeta^{\pi}_{\mu}(k) := \mathbb{E}_{\pi}\Big\{ \rho \;:\; C^{\pi_\star}_{\max}(\boldsymbol{\chi}^{j^*}) \wedge j^* = \underset{J_j \in \mathcal{L}^{(k)}}{\operatorname{argmax}}\{\pi(\boldsymbol{\varphi}^{j})\} \Big\} \tag{7.4c}$$

with the additional metric ζ^{π}_{μ}, which gives the mean evolution for deviation from optimality, ρ, when following a fixed policy π. Note, $\zeta^{\pi_\star}_{\min} = \zeta^{\star}_{\min}, \zeta^{\pi_\star}_{\max} = \zeta^{\star}_{\max}$ and $\zeta^{\pi_\star}_{\mu} = 0$. Figure 7.6 depicts Eq. (7.4) for expert policy $\pi_\star$ and SDRs.



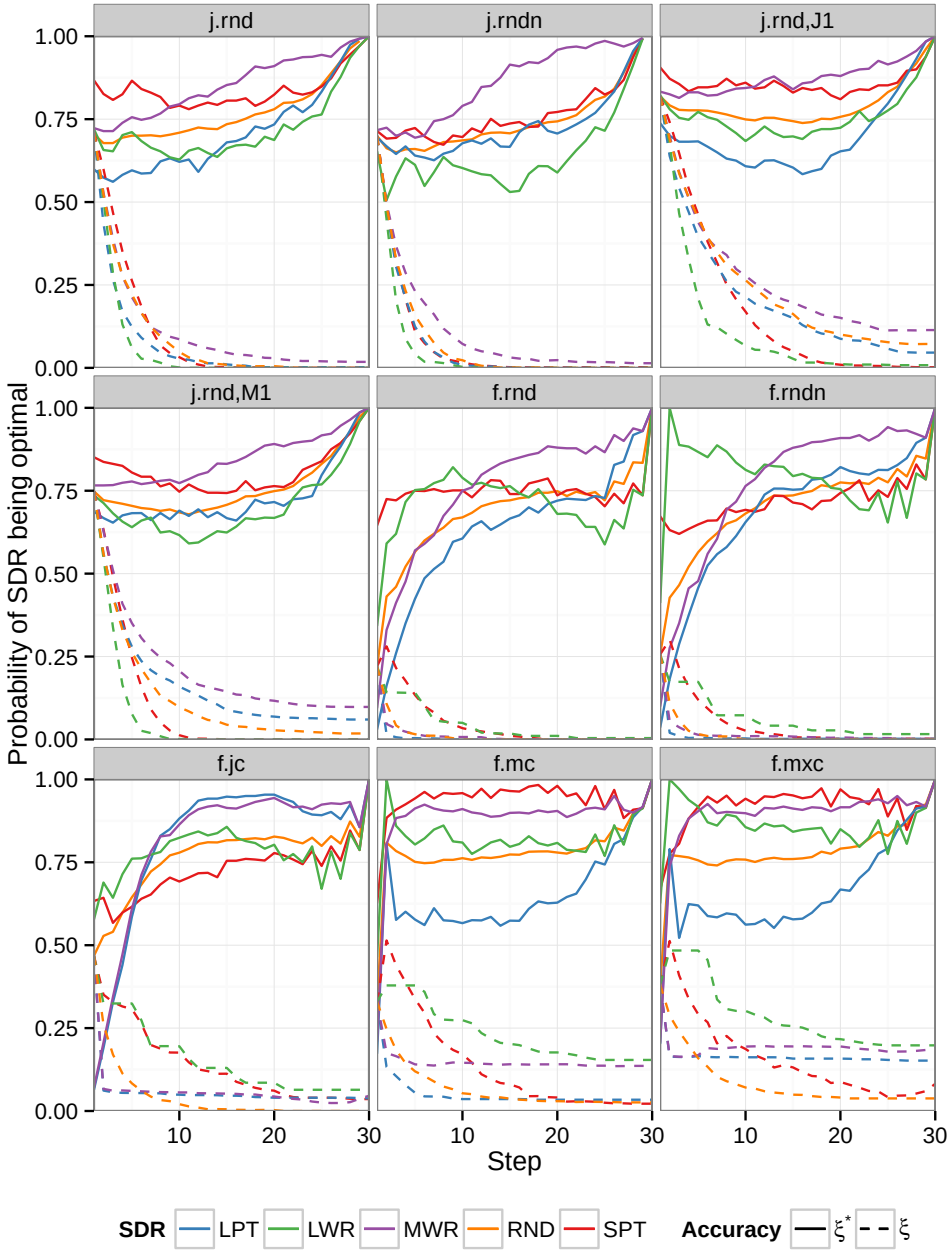
(a) $\mathcal{P}^{6\times5}_{j.rnd}$

Figure 7.4: Probability of extremal feature being optimal



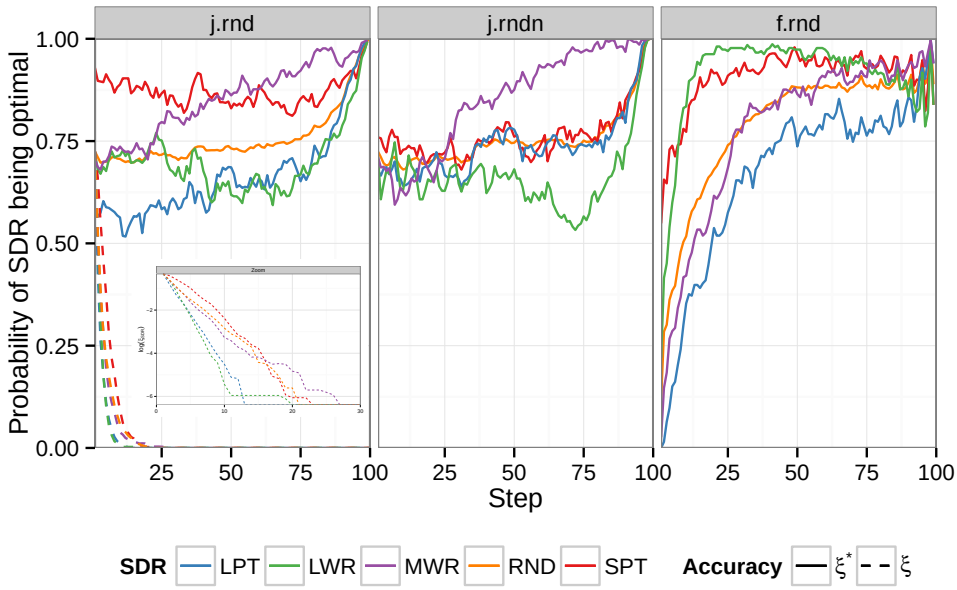
(b) $\mathcal{P}^{10\times10}_{j.rnd}$

Figure 7.4 (cont.)



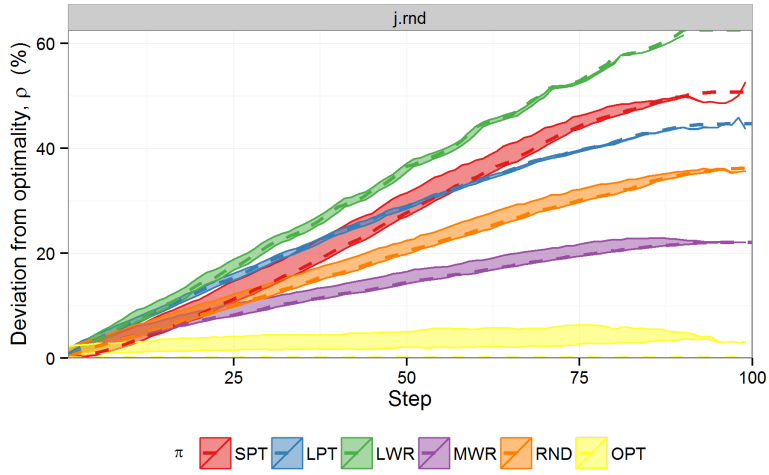
(a) 6×5

Figure 7.5: Probability of SDR yielding optimal move. Both optimal (solid: $\xi^*_{\langle \mathrm{SDR} \rangle}$) and SDR-based (dashed: $\xi_{\langle \mathrm{SDR} \rangle}$) trajectories are inspected.



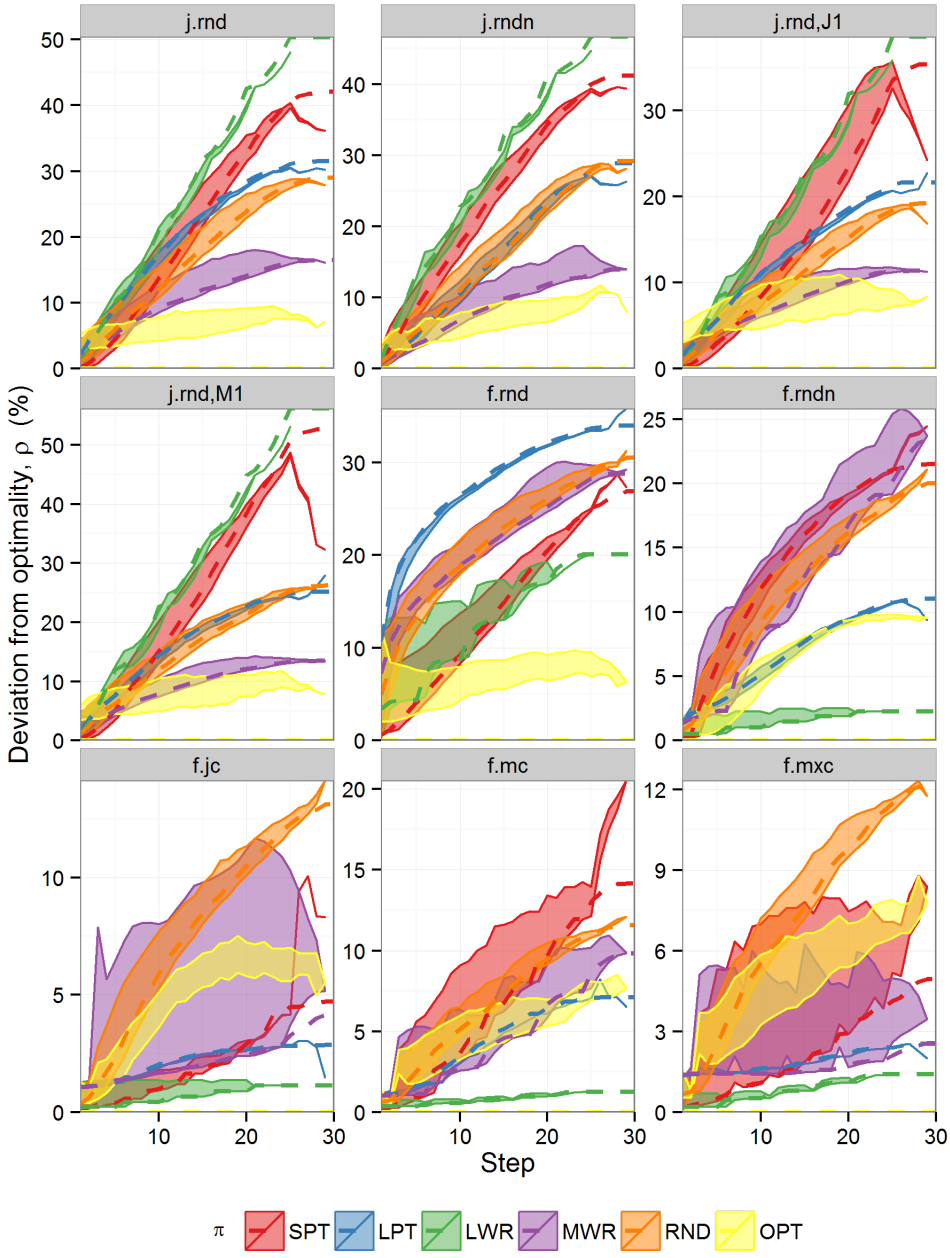
(b) 10 × 10

Figure 7.5 (cont.): Note, due to computational complexity, only $\mathcal{P}^{10\times10}_{j.rnd}$ has SDR-based trajectories also inspected. Otherwise, only optimal is pursued.



(a) 10 × 10

Figure 7.6: Mean deviation from optimality, ρ, for best and worst case scenario when not following a fixed policy π (i.e. $\zeta^{\pi}_{\min}$ and $\zeta^{\pi}_{\max}$), depicted as lower and upper bound, respectively. Moreover, mean evolution of ρ for π (i.e. ζ^{π}_{μ}) is given as a dashed line.



(b) 6×5

Figure 7.6 (cont.): Note, $\{\zeta^{\pi^\star}_{\min}, \zeta^{\pi^\star}_{\max}\}$ are illustrated jointly for $\mathcal{P}_{\text{train}}$ in Fig. 7.3

A means of interpreting Eq. (7.4), is that given a fixed policy π, then $\zeta^{\pi}_{\min}$ describes the potential improvement (iff $\zeta^{\pi}_{\min} < \zeta^{\pi}_{\mu}$) for changing the policy. Whereas, $\zeta^{\pi}_{\max}$ indicates the disadvantages of changing course. When $\zeta^{\pi}_{\min} > \zeta^{\pi}_{\mu}$, then clearly π is not a good policy for said problem space, e.g., for the final dispatches of ζ^{SPT}_{μ} for $\mathcal{P}^{6\times5}_{j.rnd,J_1}$ or $\mathcal{P}^{10\times10}_{j.rnd}$.

Remark: Eqs. (7.3) and (7.4) are based on corresponding meta-data, $\{\Phi^{\pi}, \mathcal{Y}^{\pi}\}$, from Eq. (6.2), whereas Eqs. (7.1) and (7.2) reuse the same expert meta-data, $\{\Phi^{\pi\star}, \mathcal{Y}^{\pi\star}\}$.

7.4 Simple blended dispatching rule

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 (BDR) which would be for instance switching between two SDRs at a predetermined time point.

For instance, MWR and SPT hardly ever coincide for easy or hard schedules (cf. Tables 4.6 and 4.7), so its reasonable to believe they could complement one another. Going back to Fig. 7.5b a presumably good BDR for $\mathcal{P}^{10\times10}_{j.rnd}$ would be starting with $\xi^{\star}_{\text{SPT}}(k)$ and then switching over to $\xi^{\star}_{\text{MWR}}(k)$ at around time step $k = 40$, where the SDRs change places in outperforming one another. In addition, we can see that even though $\xi_{\text{SPT}}(k)$ is generally more likely to find optimal dispatches in the initial steps, shortly after $k = 15$ then $\xi_{\text{MWR}}(k)$ becomes a contender again. A box-plot of deviation from optimality, ρ, for $\mathcal{P}^{10\times10}_{\text{train}}$ is depicted in Fig. 7.7 for a switch between SPT to MWR at time steps $k = \{10, 15, 20, 30, 40\}$. Main statistics are given in Table 7.1.

This little manipulation between SDRs does outperform SPT immensely, yet doesn't manage to gain the performance edge of MWR. 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 DR is needed to improve upon MWR.

A reason for this lack of performance of our proposed BDR at $k = 40$ is perhaps that by starting out with SPT in the beginning, it sets up the schedules in such a way that it's quite greedy and only takes into consideration jobs with shortest immediate processing times. Now, even though it is possible to find optimal schedules from this scenario, as Fig. 7.5 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 7.7 does however show, that by applying MWR instead of SPT in the latter stages, does help the schedule to be more compact w.r.t. SPT. However, the fact remains that the schedules have diverged too far from what MWR would have been able to achieve on its own, i.e., using SPT downgrades the performance of MWR.

Changing to MWR at $k \leq 20$ is not statically significant from MWR (boost in mean ρ is at most 0.5%). However, after $k > 20$ then the BDR starts diverging from MWR. But as pointed in Section 7.2, it's not so fatal to make bad moves in the very first dispatches for $\mathcal{P}^{10\times10}_{j.rnd}$, hence little is gained with improved classification accuracy in that region. But this does tell us that ξ_π is a more reliable indicator than $\xi^\star_\pi$ when it comes to choosing appropriate model parameters. Alas, ξ_π requires collecting the meta-data $\{\Phi^\pi, \mathcal{Y}^\pi\}$ from Eq. (6.2) for its policy π, whereas $\xi^\star_\pi$ reuses $\{\Phi^{\pi_\star}, \mathcal{Y}^{\pi_\star}\}$ for each new policy π.

Revisiting Fig. 7.6a, then we see $\zeta^{\text{SPT}}_\mu(40)$ has already surpassed $\zeta^{\text{MWR}}_\mu(K)$ and there are 60 operations left to dispatch. So a switch for BDR at $k = 40$ never had a chance of improvement. However, at $k \leq 15$ then $\zeta^{\text{SPT}}_\mu(k) < \zeta^{\text{MWR}}_\mu(k)$, which were appropriate turning points for BDR (although not statistically significant).

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 keep on still using the original DR that achieved the best results.

7.5 Feature evolution

In order to put the extremal features from Fig. 7.4 into perspective, it's worth comparing them with how the evolution of the features are over time, depicted in Fig. 7.8 for $\mathcal{P}^{6\times5}_{j.rnd}$ and $\mathcal{P}^{10\times10}_{j.rnd}$.* Note that the optimal trajectory describes how 'good' features should aspire to be like. We can also notice that the relative ranking in φ_{17}-φ_{24} is proportional their expected mean deviation from optimality, ρ (i.e. $\zeta^\pi_\mu(K)$). Although $(K-k)$-step lookahead give consistently the best (single) indicators for finding good solutions. Sadly, they are not practical features for high dimensional data due to computational cost. Nevertheless, bearing Fig. 7.3b in mind then it might be sufficient to do only a few steps lookahead at some key times in the dispatching process. For instance, let the computational budget for $\mathcal{P}^{10\times10}_{f.rnd}$ roll-outs be full K-solutions in the beginning phases, as that's when the problem space is most susceptible to bad moves. Then gradually decrease to only a few step lookahead, as flow-shop is then relatively stable. Conversely for $\mathcal{P}^{10\times10}_{j.rnd}$, start with a few step lookahead, and then expand the horizon as time goes by. Alternatively, when there aren't that many dispatches left, it might be worth developing a hybrid approach where the remaining dispatches from that point are optimised with some exact methods.

*Additional problem spaces can be found in Shiny application: Features > Evolution.

7.6 Emergence of problem difficulty

The main focus now is on knowing *when* during the scheduling process easy and hard problems (using the difficulty definition from Eq. (4.2)) diverge and explore in further detail *why* they diverged. The number of segregated problem instances for Φ^{ALL} and $\Phi^{\langle \text{SDR} \rangle}$ (conditioned on the followed trajectory) for $\mathcal{P}^{6\times5}_{j.rnd}$ are given in Table 7.2.*

Rather than visualising high-dimensional data projected onto two dimensional space (as was the focus in Smith-Miles and Lopes (2011) with SOM), 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 and use a t-test for testing the population correlation coefficient.

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 $\alpha_k = \frac{\alpha}{K}$, and thus maintaining the $\sum_{k=1}^{K} \alpha_k = \alpha$ significance level.

Figure 7.9 indicates the timesteps when easy and hard feature distributions differ. In the initial stages, the features are more or less the same. However, there is a clear time of divergence towards the end of the scheduling process, around $k = 25$ (little sooner for LPT and later for LWR).

7.6. Fig. 7.9

Furthermore, Fig. 7.10 shows when easy or hard features are significantly correlated to the deviation from optimality, ρ. There we can see an apparent difference in correlation between individual features with the resulting schedule depending in what stage it is in the scheduling process, implying that their influence varies over the dispatching sequencing. 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.

7.6. Fig. 7.10

It is possible for a JSP schedule to have more than one sequential dispatching repre-

*Additional problem spaces can be found in Shiny application: Footprints > Stepwise.

sentation. It is especially w.r.t. the initial dispatches. Revisiting Fig. 2.3, if we were to dispatch J_2 f first and then J_4, then that would be the same equivalent temporal schedule if we did it the other way around. This is because they don't create a conflict for one another (as is the case for jobs J_2 and J_3). This drawback of non-uniqueness of sequential dispatching representation explains why there is hardly any significant difference between the difficulties for the initial steps of the scheduling process (cf. Fig. 7.9). As we can see from Table 7.2, the number of problem instances used for statistical testing is quite limited when applying on a single algorithm. Using the non-uniqueness of $\boldsymbol{\chi}$ to our advantage, where there are many jobs that have non-conflicting machines, thereby making subsequent dispatches equivalent to the previous one, i.e., $\boldsymbol{\chi}^k \approx \boldsymbol{\chi}^{(k\pm1)}$. Therefore it's reasonable, when labelled optimal data is scarce, to inspect the stepwise statistical testing based on sliding window of the preceding and subsequent step, i.e., test at time k is based on:

$$\boldsymbol{\varphi}_i^{(k)} := \left\{ \varphi_i^{k'} \;:\; \forall \varphi_i \in \Phi \right\}_{k'=k-1}^{k+1} \tag{7.5}$$

for all individual local features $\varphi_i \in \{1, \ldots, 16\}$ from Table 2.2.

7.7 Summary and conclusions

7.7. From Section 7.3 we noticed that high stepwise optimality generally implies low deviation from optimality, ρ. Moreover, there is clearly an important factor *when* suboptimal moves are made, as Section 7.2 showed. Therefore it's not

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 analysis can undoubtedly be used in better

7.7. Check with best/worst case

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 Chapter 8. The main objective of this work is to illustrate the interaction of a specific algorithm on a given problem structure and its properties.

7.7. In this chapter we will find the footprint from single priority dispatching rules introduced in Section 2.4 on those problem spaces. Presumably, we could use that information to infer the complexity of our synthesised problem spaces summarised in Table 3.2.

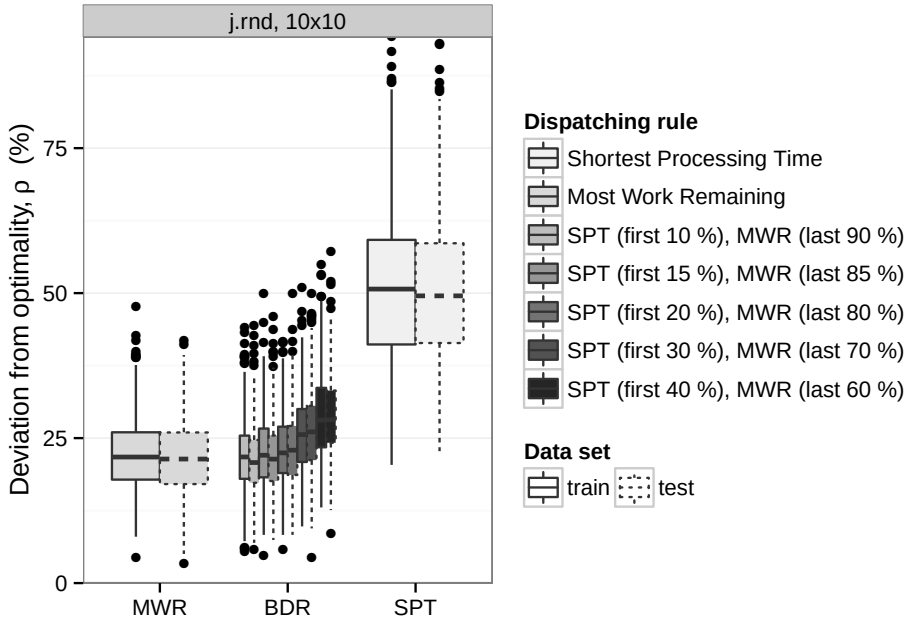
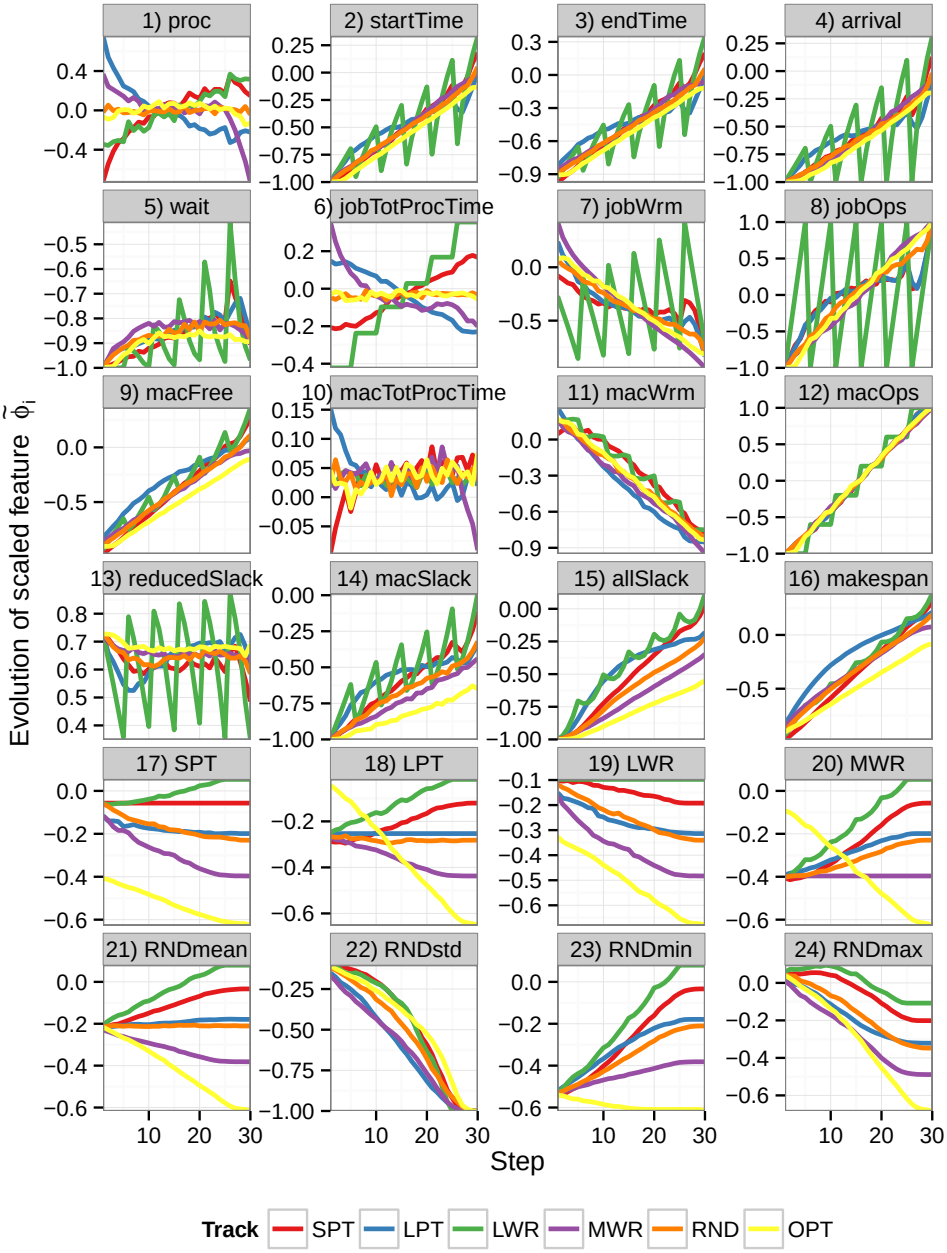


Figure 7.7: Box plot of $\mathcal{P}_{j.rnd}^{10\times10}$ deviation from optimality, ρ, for BDR where SPT is applied for the first 10%, 15%, 20%, 30% or 40% of the dispatches, followed by MWR.

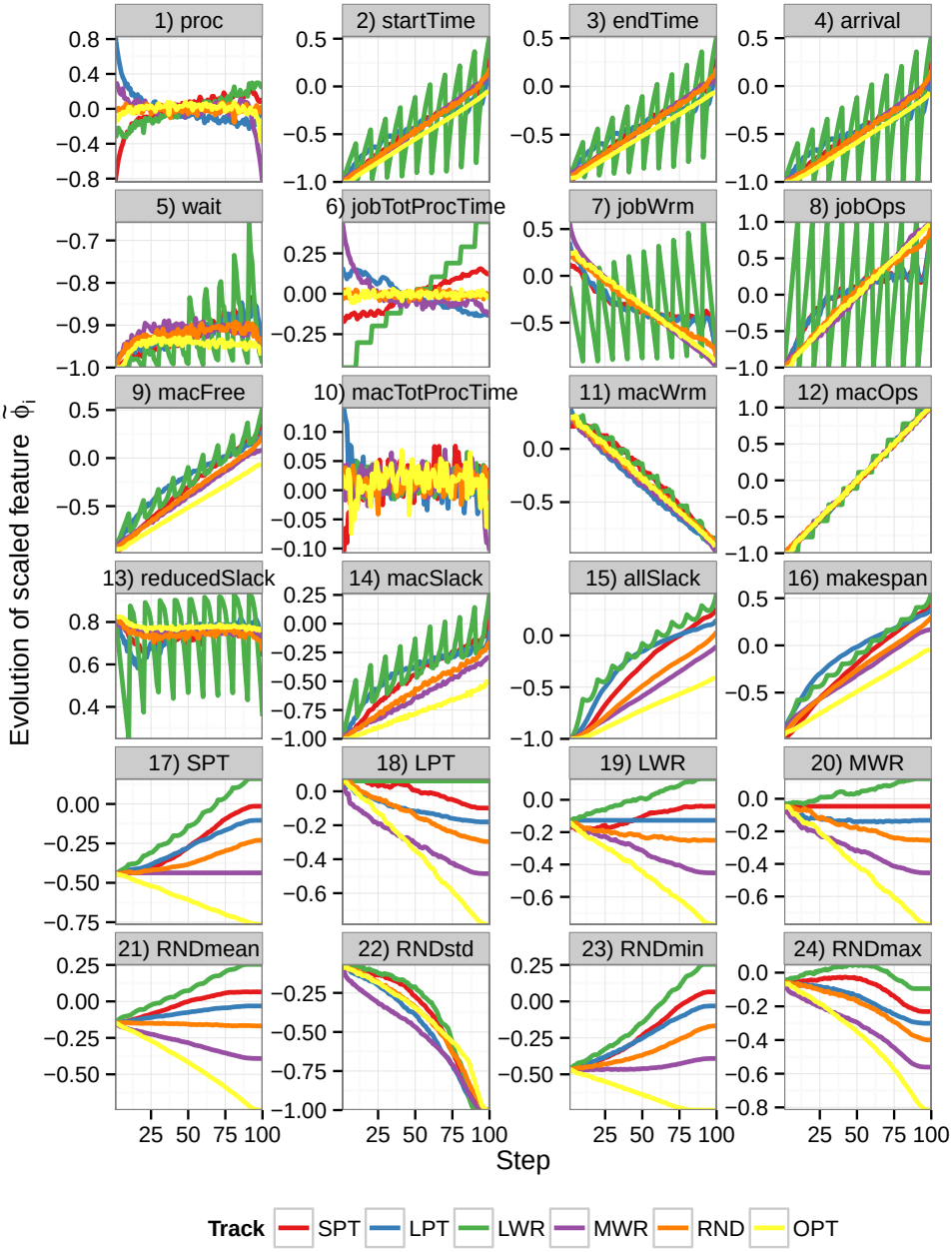
Table 7.1: Main statistics for $\mathcal{P}_{j.rnd}^{10\times10}$ deviation from optimality, ρ, using BDR that changes from SDR at a fixed time step k.

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



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

Figure 7.8: Mean stepwise evolution of $\tilde{\boldsymbol{\varphi}}$, which is scaled according to Eq. (A.16)



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

Figure 7.8 (cont.)

Table 7.2: Number of problem instances after segregating $\mathcal{P}_{j.rnd}^{6\times5}$ w.r.t. difficulty and trajectory.

(a) Used in Fig. 7.9

Track	#Easy	#Hard	#Significant
SPT	37	203	48
LPT	103	81	47
LWR	19	295	31
MWR	328	1	0
Σ	487	580	126

(b) Used in Fig. 7.10

	#Significant	
Track	Easy	Hard
SPT	78	101
LPT	87	45
LWR	48	82
MWR	167	0
ALL	260	138
Σ	640	366

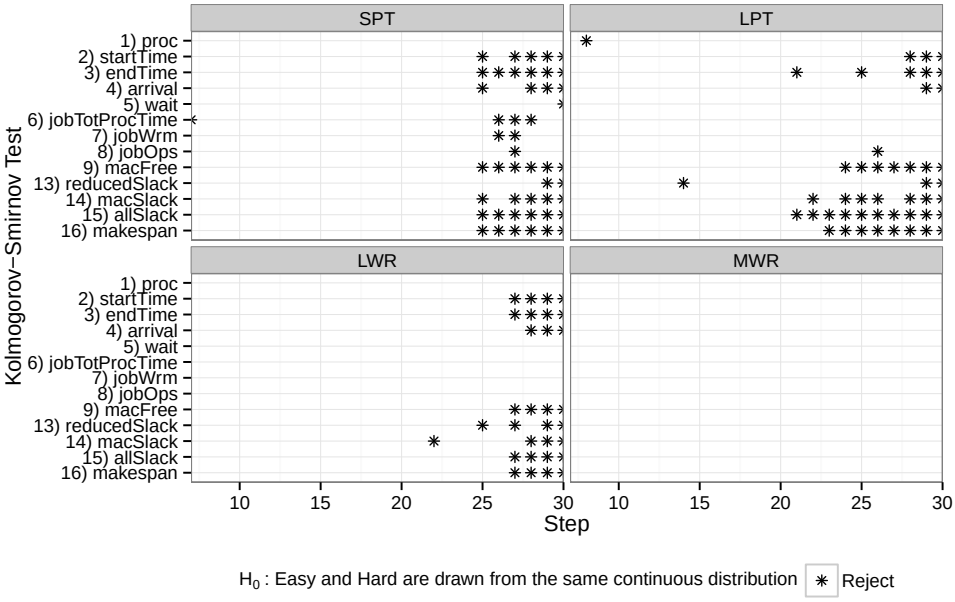


Figure 7.9: Stepwise K-S Test for features $\boldsymbol{\varphi}$ segregated w.r.t. easy and hard problems are drawn from the same data distribution.

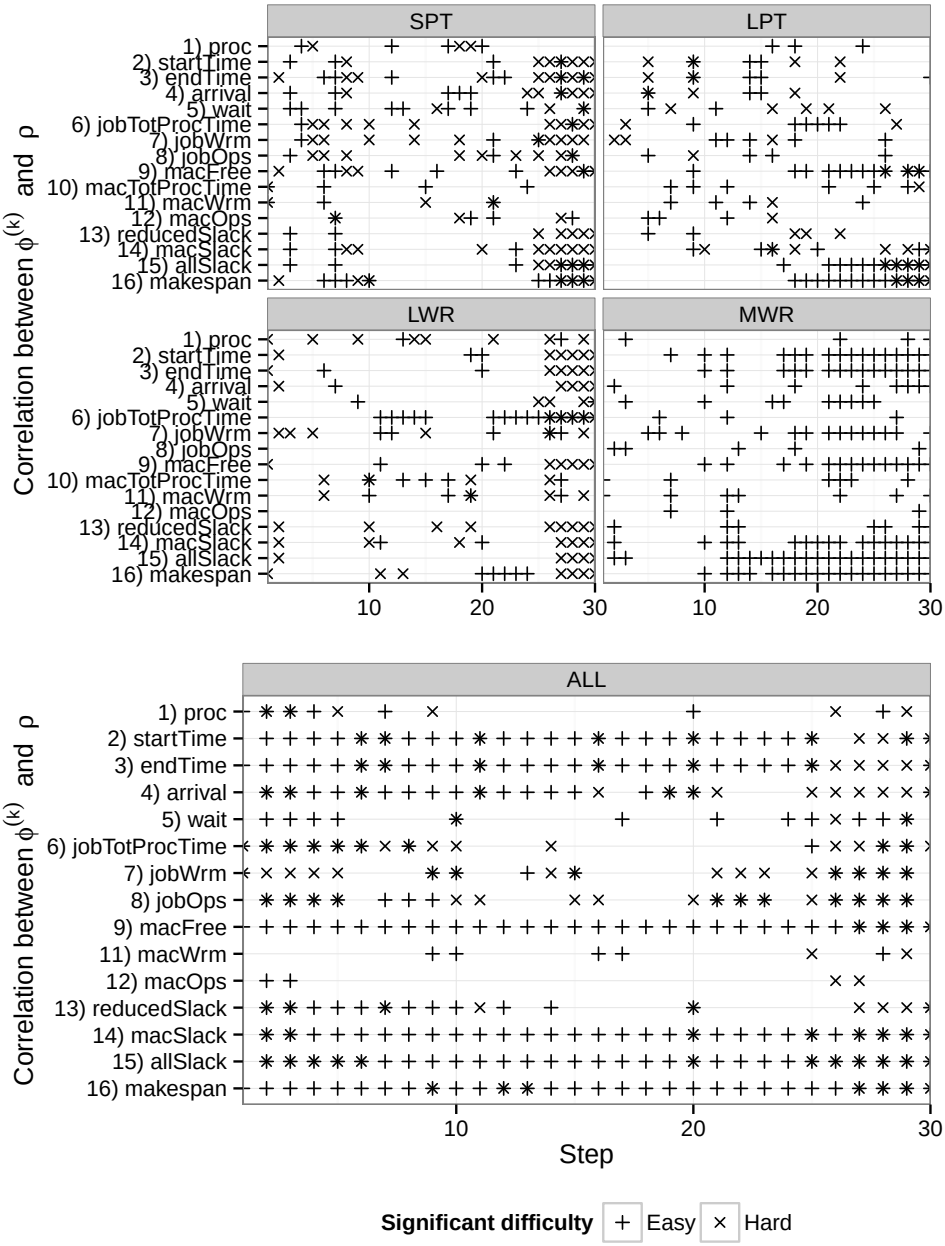


Figure 7.10: Stepwise significance of a correlation coefficient for $\mathcal{P}^{6\times5}_{j.rnd}$ features $\boldsymbol{\varphi}$, segregated w.r.t. easy and hard problems, with resulting deviation from optimality, ρ.

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

Alice

8 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 Rúnarsson (2006), and given in Appendix A for completeness.

8.1 Ordinal regression for job-shop

Using the training set $\{\Phi^{\pi}, \mathcal{Y}^{\pi}\}$, given in Eq. (6.2) by following some policy π, let $\boldsymbol{\varphi}_o \in \Phi^{\pi}$ denote the post-decision state when dispatching job J_o corresponds to an optimal schedule being built. All post-decisions states corresponding to suboptimal dispatches, J_s, are denoted by $\boldsymbol{\varphi}_s \in \Phi^{\pi}$.

Let's label feature sets which 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. The preference learning problem is specified by a set of preference pairs,

$$\Psi := \bigcup_{\{\mathbf{x}_i\}_{i=1}^{N_{\text{train}}}} \left\{ \{\mathbf{z}_o, +1\}, \{\mathbf{z}_s, -1\} \ : \ \forall (J_o, J_s) \in \mathcal{O}^{(k)} \times \mathcal{S}^{(k)} \right\}_{k=1}^{K} \subset \Phi \times Y \quad (8.1)$$

where: *i)* $\Phi \subset \mathcal{F}$ is the training set of $d = 16$ features (cf. the local features from Table 2.2); *ii)* $Y = \{-1, +1\}$ is the outcome space; *iii)* at each dispatch $k \in \{1, \ldots, K\}$,

and *iv)* $J_o \in \mathcal{O}^{(k)}$, $J_s \in \mathcal{S}^{(k)}$ are optimal and suboptimal dispatches, respectively.

A negative example is only created as long as J_s actually results in a worse makespan, i.e., $C_{\max}^{\pi_\star}(\boldsymbol{\chi}^s) \gneq C_{\max}^{\pi_\star}(\boldsymbol{\chi}^o)$, since there can exist situations in which more than one operation can be considered optimal. Hence, $\mathcal{O}^{(k)} \cup \mathcal{S}^{(k)} = \mathcal{L}^{(k)}$, and $\mathcal{O}^{(k)} \cap \mathcal{S}^{(k)} = \emptyset$. If the makespan would be unaltered, the pair is omitted from Υ, 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, i.e., $C_{\max}^{\pi_\star}(\boldsymbol{\chi}^j)$ for all $J_j \in \mathcal{L}^{(k)}$. 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.

Since $Y = \{+1, -1\}$, we can use logistic regression, which makes decisions regarding optimal dispatches and at the same time efficiently estimates a posteriori probabilities. When using linear classification model (cf. Appendix A.2) for Eq. (2.12),then the optimal $\mathbf{w}^*$ obtained from the preference set can be used on any new data point (i.e. partial schedule), $\boldsymbol{\chi}$, and their inner product is proportional to probability estimate Eq. (A.9). Hence, for each job on the job-list, $J_j \in \mathcal{L}$, 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 from Eq. (2.12) where $\pi(\cdot)$ is the classification model obtained by the preference set, Υ, defined by Eq. (8.1).

8.2 Selecting Preference Pairs

Defining the size of the preference set as $l = |\Upsilon|$, then Eq. (8.1) gives the size of the feature training set as $|\Phi| = \frac{1}{2}l$, which is given in Fig. 6.2 and Table 6.1. 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 Paper I was to follow a *single* 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 preference set of size,

$$l = \sum_{i=1}^{N_{\text{train}}} \left(2|\mathcal{O}_i^{(k)}| \cdot |\mathcal{S}_i^{(k)}| \right) \tag{8.2}$$

For the problem spaces considered in Paper I, that sort of simple sampling of the state

space was sufficient for a favourable outcome. However, for a considerably harder problem spaces (cf. Chapter 4) and not to mention increased number of jobs and machines, preliminary experiments were not satisfactory.

A brute force approach was adopted to investigate the feasibility of finding optimal weights $\mathbf{w}$ for Eq. (2.12). By applying CMA-ES (discussed thoroughly in Chapter 5) 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 preference 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 job-list, $\mathcal{L}$, 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 preference 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 offline.

All problem instances are correctly labelled w.r.t. their optimum makespan, found with analytical means.* The main motivation for the data generation of Ψ that will be used in preference learning, will now need to consider the following main aspects:

PREF.1 Which path(s) should be investigated to collect training instances, i.e., Φ. Should they be features gathered resulting in : *i)* optimal solutions (querying expert policy $\pi_\star$)? *ii)* suboptimal solutions when a DR is implemented (following a fixed policy π), or *iii)* combination of both?

PREF.2 What sort of rankings should be compared during each step?

*Optimal solution were found using Gurobi Optimization, Inc. (2014), a commercial software package for solving large-scale linear optimisation and a state-of-the-art solver for mixed integer programming.

PREF.3 What sort of stepwise sampling strategy is needed for a good *single* time independent model?

The collection of the training set Φ in PREF.1 (which is described in Chapter 6) is of paramount of importance, as the subsequent preference pairs in Υ are highly dependent on the quality of Φ. Since the labelling of Φ is quite computationally intensive, its collection should be done parsimoniously in order to not waste valuable time and resources. On the other hand, PREF.2 and PREF.3 are easy to inspect, once Φ has been chosen. The following sections will try to address these research questions.

8.3 Scalability of dispatching rules

In Paper I a separate data set was deliberately created 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 K linear scheduling rules for solving a $n \times m$ 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 Paper I was to take the mean weight for all stepwise linear models, i.e., $\bar{w}_i = \frac{1}{K}\sum_{k=1}^{K} w_i^{(k)}$ where $\mathbf{w}^{(k)}$ is the linear weight resulting from learning preference set $\Upsilon^{(k)}$ at dispatch k.

A more sophisticated way, would be to create a *new* linear model, where the preference set, Υ, is the aggregation of all preference pairs across the K dispatches. This would amount to a substantial training set, and for Υ to be computationally feasible to learn, Υ has to be filtered to size $l_{\max}$. The default set-up will be,

$$l_{\max} := \begin{cases} 5 \cdot 10^5 & \text{for} \quad 10 \times 10 \quad \text{JSP} \\ 10^5 & \text{for} \quad 6 \times 5 \quad \text{JSP} \end{cases} \tag{8.3}$$

which is roughly 60%-70% amount of preferences encountered from one pass of sampling a K-stepped trajectory using a fixed policy $\hat{\pi}$ for the default N_{train} (cf. Table 8.1). Sampling is done randomly, with equal probability.

8.4 Ranking strategies

First let's address PREF.2. The various ranking strategies for adding preference pairs to Υ defined by Eq. (8.1) were first reported in Paper V, and are the following,

> **Basic ranking, Υ_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 Paper I. Note, $|\Upsilon_b|$ is defined in Eq. (8.2).

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

Partial subsequent rankings, Ψ_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 $\Psi_p \subset \Psi_f$.

All rankings, Ψ_a, denotes that all possible rankings were explored, i.e., r_i versus r_j for $i, j \in \{1, \ldots, n'\}$ and $i \neq j$, preference pairs are added.

where $r_1 > r_2 > \ldots > r_{n'}$ ($n' \leq n$) are the rankings of the job-list, $\mathcal{L}^{(k)}$, at time step k. By definition the following property holds:

$$\Psi_p \subset \Psi_f \subset \Psi_b \subset \Psi_a \tag{8.4}$$

To test the validity of different ranking strategies for PREF.2, a training set of $N_{\text{train}} = 500$ problem instances of $\mathcal{P}^{6\times5}_{j.rnd}$ and $\mathcal{P}^{6\times5}_{f.rnd}$is collected for all trajectories described in Section 6.4. The size of the preference set, $|\Psi|$, is depicted in Fig. 8.1 for each iteration k. From which, a linear preference model is created for each preference set, Ψ. A box-plot for deviation from optimality, ρ, defined by Eq. (2.17), is presented in Fig. 8.2. From the figure it is apparent there can be a performance edge gained by implementing a particular trajectory strategy, yet ranking scheme seems to be irrelevant. Moreover, the behaviour is analogous across all other $\mathcal{P}^{6\times5}_{\text{train}}$ in Table 3.2.

First let's restrict the models to $\mathcal{P}_{6\times5}$train. There is no statistical difference between Ψ_f and Ψ_p ranking-schemes across all disciplines, which is expected since Ψ_f is designed to contain the same preference information as Ψ_f (cf. Eq. (8.4)). However, neither of the Pareto ranking-schemes outperform the original Ψ_b set-up from Paper I. The results hold for the test set as well. Any statistical difference between ranking schemes were for Ψ_a, where it was considered slightly lacking than some of its counterparts. Since a smaller preference set is preferred, its opted to use the Ψ_p ranking scheme henceforth as the default set-up for PREF.2.

Moving on to higher dimension, results for $\mathcal{P}^{10\times10}_{j.rnd}$ were similar to $\mathcal{P}^{6\times5}_{\text{train}}$. Only exception begin that ranking schemes showed difference in performance when using Φ^{OPT}, where Ψ^{OPT}_p come on top. Strengthening our previous choice of Ψ_p as standard ranking scheme.

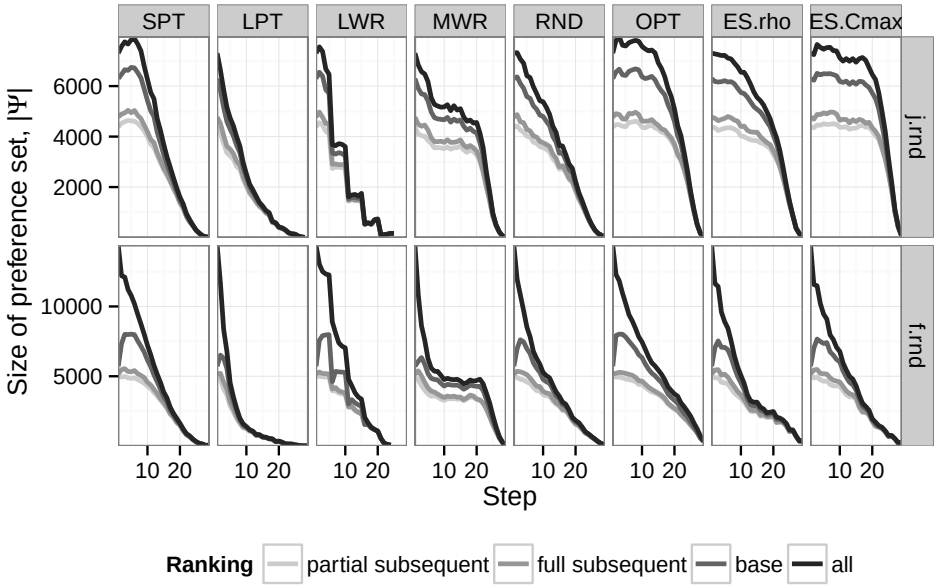


Figure 8.1: Size of $\mathcal{P}^{6\times5}_{\text{train}}$ preference set, $l = |\Psi|$, for different trajectory strategies and ranking schemes (where $N_{\text{train}} = 500$)

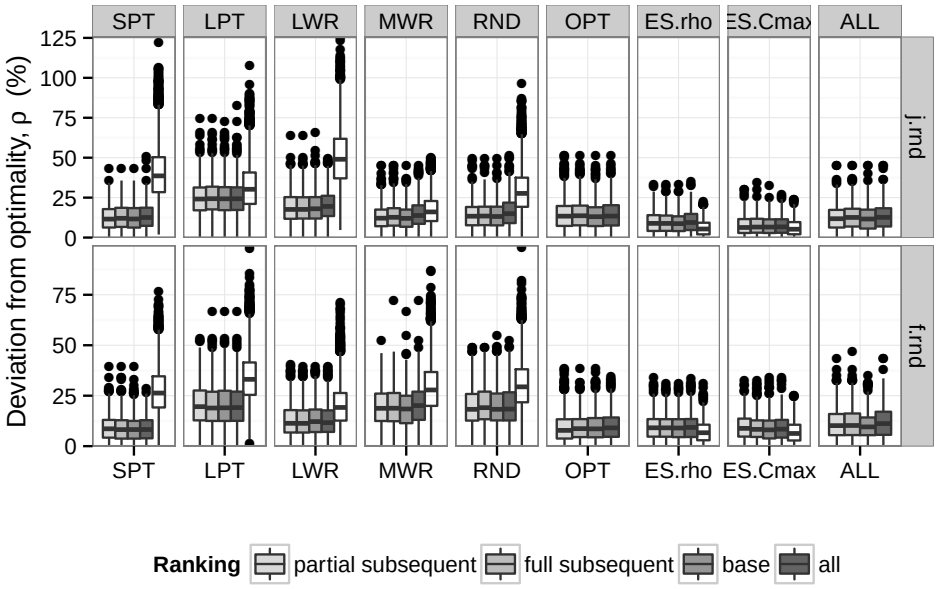


Figure 8.2: Box-plot for various Φ and Ψ set-up using $\mathcal{P}^{6\times5}_{\text{train}}$. The trajectories the models are based on are depicted in white on the far right.

8.5 Trajectory strategies

We'd like to inspect which trajectory is the best to use for Υ. Paper V only considered $\mathcal{P}^{6\times5}_{j.rnd}$ and $\mathcal{P}^{6\times5}_{j.rndn}$, however, results for $\mathcal{P}^{6\times5}_{\text{train}}$ and $\mathcal{P}^{10\times10}_{j.rnd}$ are currently available.* Models from Fig. 8.2 are limited to the ones corresponding to Υ_p. Figure 8.3 jointly illustrates the size of the preference set used, i.e., $|\Upsilon_p|$ from Fig. 8.1. Table 8.1 reports the total amount of preferences for all K dispatches.

Table 8.2 reports the relative ordering of trajectories, ordered w.r.t. their mean deviation from optimality, ρ, and their size of preference set, i.e., $|\Upsilon_p|$. Models that are statistically better are denoted by '$\succ$' otherwise considered equivalent.

For most problem spaces Υ_p^{LPT} was the worst trajectory to pursue. Looking back at Fig. 6.2, then even though Φ^{LPT} was not the trajectory with the least features, the amount of equivalent features w.r.t. $C_{\max}$ are far too many to make a meaningful preference set out of it. It's only for $\mathcal{P}^{6\times5}_{j.rndn}$ that there is another trajectory with fewer preferences, namely Υ_p^{LWR} (cf. Fig. 8.1), and in that case LWR is the worst model. Model that come on top, are those that have a varied Υ. However, aggregating features from all trajectories is not a good idea, as the preference set then becomes too varied for a satisfactory result.

Learning preference pairs from a good scheduling policies, such as $\Phi^{\text{ES}.C_{\max}}$, $\Phi^{\text{ES}.\rho}$ and Φ^{MWR}, gave considerably more favourable results than tracking optimal paths, save for $\mathcal{P}^{6\times5}_{f.jc}$ where the ordering is reversed. Generally, suboptimal routes are preferred. However, even though LWR is a better policy than MWR for FSP, then Φ^{LWR} is a worse candidate than e.g. Φ^{MWR}, but as discussed before, it's due to the lack of varied dispatches for the trajectory.

It is particularly interesting there is statistical difference between Φ^{OPT} and Φ^{RND}, where the latter had improved performance for all JSP problem spaces. In those cases, tracking optimal dispatches gives worse performance as pursuing completely random dispatches. This indicates that exploring only expert policy 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.

Generally, adding suboptimal trajectories with the expert policy, i.e., Φ^{ALL}, gives the learning algorithm a greater variety of preference pairs for getting out of local minima. However, for some problem spaces, e.g., $\mathcal{P}^{6\times5}_{f.rnd}$ and $\mathcal{P}^{6\times5}_{f.mc}$ then additional suboptimal solutions that are too diverse yield a worse outcome than Φ^{OPT} would achieve on its own.

*Additional problem spaces can be found in Shiny application: Preference Models > Trajectories & ranks.

Table 8.1: Total number of preferences in $l = |\Upsilon_p|$ for all K steps. Note '–' denotes not available.

Track	$\mathcal{P}^{6\times5}_{\text{train}}, N_{\text{train}} = 500$									$\mathcal{P}^{10\times10}_{\text{train}}, N_{\text{train}} = 300$		
	j.rnd	*j.rndn*	*j.rnd*, J_1	*j.rnd*, M_1	*f.rnd*	*f.rndn*	*f.jc*	*f.mc*	*f.mxc*	*j.rnd*	*j.rndn*	*f.rnd*
SPT	73926	68410	74416	65150	79388	70808	68956	89788	92036	285912	–	–
LPT	43456	58540	28498	34136	36162	54684	11548	23260	17308	151444	–	–
LWR	46580	46306	32326	41554	64226	68628	69124	40150	40110	163546	–	–
MWR	83756	102092	53246	62056	87376	111708	106226	65882	64692	370104	–	–
RND	72824	80358	52210	61670	77148	77080	64550	55288	55398	313346	–	–
OPT	100910	111736	79404	90948	95388	93036	81306	79836	78440	453662	470522	299952
ES.ρ	93006	111068	64050	89504	77142	63120	45404	36608	74556	427032	–	–
ES.$C_{\max}$	108390	111346	73168	95920	83058	61992	47412	35484	36052	432650	–	–
ALL	622848	689856	457318	540938	599888	601056	494526	426296	458592	2595758	470522	299952

Table 8.2: Relative ordering w.r.t. mean ρ and size of its preference set, $l = |\Upsilon_p|$, for trajectories in Section 6.4

Problem		Ordering of trajectories
$\mathcal{P}^{6\times5}_{j.rnd}$	ρ	$\text{ES.}C_{\max} \succ \text{ES.}\rho \succ \text{ALL} \equiv \text{SPT} \equiv \text{MWR} \equiv \text{RND} \succ \text{OPT} \succ \text{LWR} \succ \text{LPT}$
	l	$\text{ALL} \gg \text{ES.}C_{\max} > \text{OPT} > \text{ES.}\rho > \text{MWR} > \text{SPT} > \text{RND} > \text{LWR} > \text{LPT}$
$\mathcal{P}^{6\times5}_{j.rndn}$	ρ	$\text{ES.}C_{\max} \equiv \text{ES.}\rho \equiv \text{MWR} \equiv \text{RND} \equiv \text{ALL} \equiv \text{SPT} \succ \text{LPT} \succ \text{LWR} \succ \text{OPT}$
	l	$\text{ALL} \gg \text{OPT} > \text{ES.}C_{\max} > \text{ES.}\rho > \text{MWR} > \text{RND} > \text{SPT} > \text{LPT} > \text{LWR}$
$\mathcal{P}^{6\times5}_{j.rnd,J_1}$	ρ	$\text{ES.}C_{\max} \succ \text{ES.}\rho \succ \text{SPT} \succ \text{ALL} \equiv \text{RND} \succ \text{OPT} \equiv \text{LWR} \equiv \text{MWR} \succ \text{LPT}$
	l	$\text{ALL} \gg \text{OPT} > \text{SPT} > \text{ES.}C_{\max} > \text{ES.}\rho > \text{MWR} > \text{RND} > \text{LWR} > \text{LPT}$
$\mathcal{P}^{6\times5}_{j.rnd,M_1}$	ρ	$\text{ES.}C_{\max} \equiv \text{ES.}\rho \succ \text{SPT} \equiv \text{ALL} \equiv \text{RND} \succ \text{LWR} \succ \text{OPT} \succ \text{MWR} \succ \text{LPT}$
	l	$\text{ALL} \gg \text{ES.}C_{\max} > \text{OPT} > \text{ES.}\rho > \text{SPT} > \text{MWR} > \text{RND} > \text{LWR} > \text{LPT}$
$\mathcal{P}^{6\times5}_{f.rnd}$	ρ	$\text{SPT} \equiv \text{ES.}C_{\max} \equiv \text{OPT} \equiv \text{ES.}\rho \succ \text{ALL} \succ \text{LWR} \succ \text{MWR} \equiv \text{RND} \equiv \text{LPT}$
	l	$\text{ALL} \gg \text{OPT} > \text{MWR} > \text{ES.}C_{\max} > \text{SPT} > \text{RND} > \text{ES.}\rho > \text{LWR} > \text{LPT}$
$\mathcal{P}^{6\times5}_{f.rndn}$	ρ	$\text{ES.}\rho \equiv \text{ES.}C_{\max} \equiv \text{RND} \equiv \text{LWR} \equiv \text{ALL} \succ \text{SPT} \equiv \text{LPT} \equiv \text{OPT} \succ \text{MWR}$
	l	$\text{ALL} \gg \text{MWR} > \text{OPT} > \text{RND} > \text{SPT} > \text{LWR} > \text{ES.}\rho > \text{ES.}C_{\max} > \text{LPT}$
$\mathcal{P}^{6\times5}_{f.jc}$	ρ	$\text{OPT} \succ \text{SPT} \equiv \text{LWR} \succ \text{RND} \equiv \text{MWR} \succ \text{LPT} \equiv \text{ES.}C_{\max} \equiv \text{ES.}\rho \succ \text{ALL}$
	l	$\text{ALL} \gg \text{MWR} > \text{OPT} > \text{LWR} > \text{SPT} > \text{RND} > \text{ES.}C_{\max} > \text{ES.}\rho > \text{LPT}$
$\mathcal{P}^{6\times5}_{f.mc}$	ρ	$\text{ES.}\rho \succ \text{LWR} \succ \text{ES.}C_{\max} \equiv \text{MWR} \succ \text{OPT} \equiv \text{LPT} \equiv \text{RND} \equiv \text{SPT} \succ \text{ALL}$
	l	$\text{ALL} \gg \text{SPT} > \text{OPT} > \text{MWR} > \text{RND} > \text{LWR} > \text{ES.}\rho > \text{ES.}C_{\max} > \text{LPT}$
$\mathcal{P}^{6\times5}_{f.mxc}$	ρ	$\text{RND} \equiv \text{OPT} \equiv \text{ES.}\rho \equiv \text{SPT} \equiv \text{ALL} \equiv \text{MWR} \succ \text{LWR} \succ \text{ES.}C_{\max} \succ \text{LPT}$
	l	$\text{ALL} \gg \text{SPT} > \text{OPT} > \text{ES.}\rho > \text{MWR} > \text{RND} > \text{LWR} > \text{ES.}C_{\max} > \text{LPT}$
$\mathcal{P}^{10\times10}_{j.rnd}$	ρ	$\text{ES.}C_{\max} \equiv \text{ES.}\rho \succ \text{MWR} \succ \text{RND} \succ \text{SPT} \equiv \text{OPT} \succ \text{ALL} \succ \text{LPT} \equiv \text{LWR}$
	l	$\text{ALL} \gg \text{OPT} > \text{ES.}C_{\max} > \text{ES.}\rho > \text{MWR} > \text{RND} > \text{SPT} > \text{LWR} > \text{LPT}$

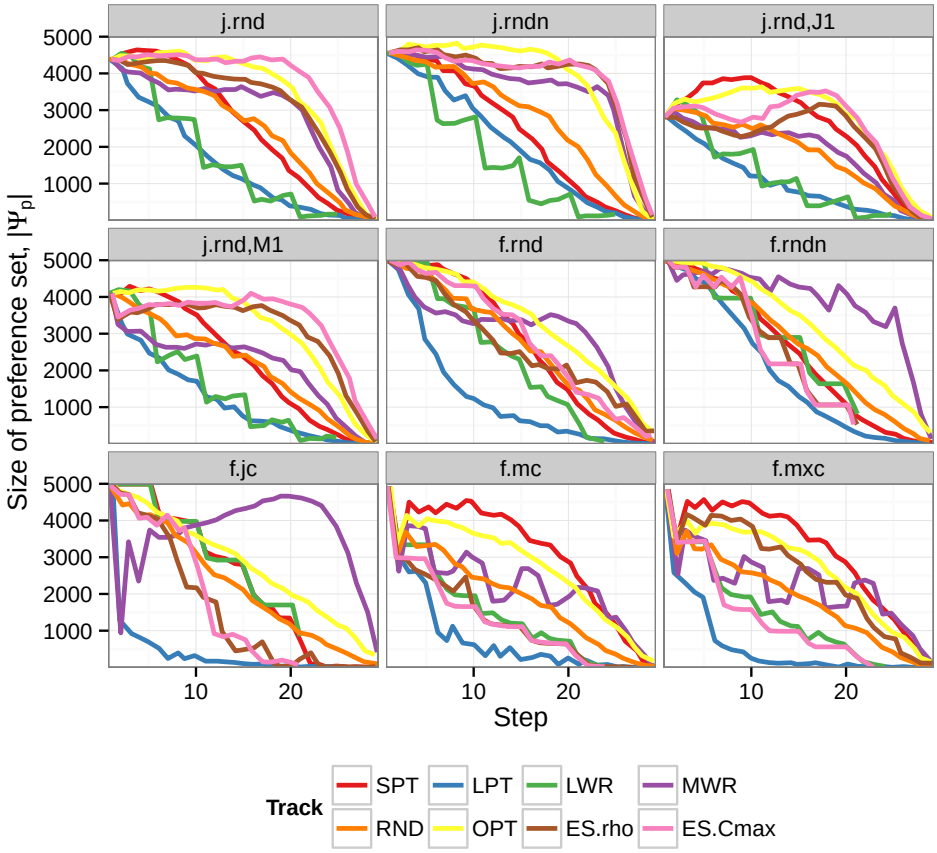


Figure 8.3: Size of $\mathcal{P}^{6\times5}_{\text{train}}$ preference set, $l = |\Psi_p|$, for different trajectory strategies

Comparing $\Psi^{\langle \text{DR} \rangle}$ to its corresponding DR used to guide its collection, then usually the preference model outperformed the DR it was trying to mimic. The exceptions being: *i)* MWR for $\mathcal{P}^{6\times5}_{j.rnd,J_1}$ and $\mathcal{P}^{6\times5}_{j.rnd,M_1}$ (and $\mathcal{P}^{10\times10}_{j.rnd}$ was statistically insignificant); *ii)* LWR for $\mathcal{P}^{6\times5}_{f.mc}$ and $\mathcal{P}^{6\times5}_{f.mxc}$; *iii)* LPT was statistically insignificant for $\mathcal{P}^{6\times5}_{j.rnd,M_1}$, and *iv)* ES.$C_{\max}$ and ES.ρ for all problem spaces, save for $\mathcal{P}^{6\times5}_{j.rndn}$ which was statistically insignificant. Revisiting Fig. 7.6, then when $\Psi^{\langle \pi \rangle}$ succeeds its original policy π, it implies the learning model was able to steer the learned policy towards $\zeta^{\langle \pi \rangle}_{\min}$. In fact, its improvement is proportional to its spread* from $\zeta^{\langle \pi \rangle}_{\mu}$ to $\zeta^{\langle \pi \rangle}_{\mu}$ or $\zeta^{\langle \pi \rangle}_{\min}$. Therefore, a good preference set based on Φ^{π} not only has to have a low ζ^{π}_{μ} to mimic, but also the policy π needs to be sufficiently different from $\zeta^{\pi}_{\min}$ and $\zeta^{\pi}_{\max}$ for adequate learning. That is why $\Phi^{\langle \text{CMA-ES} \rangle}$ strategies were not good enough for preference learning, as their $\zeta^{\pi}_{\langle \cdot \rangle}$ spread was the lowest compared to the other fixed DRs.

*Consult Shiny application: Optimality > Best and worst case scenario.

The rational for using the $\Phi^{\langle \text{CMA-ES} \rangle}$ strategies was mostly due to the fact a linear classifier is creating the training data (using the weights found via CMA-ES optimisation in Eq. (5.1)), hence the training data created should be linearly separable, which in turn should boost the training accuracy for a linear classification learning model. However, these strategies is not outperforming the original DR used in guiding the training data collection.

As the experimental results showed, that 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) gives meaningful preference pairs which the learning algorithm could learn.

8.6 Feature Selection

We know from Chapter 5 there exists linear weights $\mathbf{w}$ for Eq. (2.12) found with evolutionary optimisation that achieve a lower deviation from optimality, ρ, than any of the aforementioned preference models have been able to learn. This goes to show that the $d = 16$ features are 'enough' – meaning there is not a need for defining new ones just yet. However, the optimal weights for Eq. (5.1) were quite erratic (cf. Fig. 5.2). Perhaps the features from $\Phi^{\langle \text{CMA-ES} \rangle}$ are contradictory, and therefore not suitable for preference learning. Furthermore, the SDRs we've inspected so-far are based on two job-attributes from Table 2.2, namely: *i)* φ_1 for SPT and LPT, and *ii)* φ_7 for LWR and MWR, by choosing the lowest value for SPT and LWR, and highest value for LPT and MWR, i.e., the extremal values for those attributes. These SDRs achieve a remarkably low ρ, so perhaps not that many additional features are needed to achieve a competitive result.

For this study we will consider all combinations of feature attributes using either one, two, three or all $d = 16$ of them, for a total of

$$\binom{d}{1} + \binom{d}{2} + \binom{d}{3} + \binom{d}{d} = 697 \tag{8.5}$$

The reason for such a limiting number of active features, are due to the fact we want to keep the models simple enough for improved model interpretability.

For each feature combination, a linear preference model is created, where Ψ_p is limited to the predetermined feature combination. This was done for all $\mathcal{P}_{\text{train}}^{10\times10}$ in Table 3.2, each consisting of $N_{\text{train}} = 300$ problem instances. Moreover, in order to report the validation accuracy, 20% (i.e. $N_{\text{val}} = 60$) of the training set was set aside for reporting the accuracy.

8.6.1 Validation accuracy

As the preference set Υ_p has both preference pairs belonging to optimal ranking, and subsequent rankings, it is not of primary importance to classify *all* rankings correctly, just the optimal ones. Therefore, instead of reporting the validation accuracy based on the classification problem of the correctly labelling the entire problem set Υ_p, it's opted that the validation accuracy is obtained using Eq. (7.1), namely the probability of choosing an optimal decision given the resulting linear weights.* However, in this context, the mean throughout the dispatching process is reported, i.e., $\frac{1}{K}\sum_{k=k'}^{K}\xi_\pi^\star(k')$. Figure 8.4 shows the difference between the two measures of reporting validation accuracy.

Validation accuracy based on $\xi_\pi^\star$ only takes into consideration the likelihood of choosing the optimal move at each time step. However, the classification accuracy is also trying to correctly distinguish all subsequent rankings in addition of choosing the optimal move, as expected that measure is considerably lower.

8.6.2 Pareto front

When training the learning model one wants to keep the validation accuracy high, as that would imply a higher likelihood of making optimal decisions, which would in turn translate into a low final makespan. To test the validity of this assumptions, each of the 697 models is run on the preference set, and its mean ρ is reported against its corresponding validation accuracy in Fig. 8.5. The models are colour-coded w.r.t. the number of active features, and a line is drawn through its Pareto front. Moreover, those solutions are labelled with their corresponding model ID. Moreover, the Pareto front over all 697 models, irrespective of active feature count, is denoted with triangles. Moreover, their values are reported in Table 8.3, where the best objective is given in boldface.

Figure 8.6 depicts $\mathbf{w}$ for all of the learned CDR models reported in Table 8.3. The weights have been normalised for clarity purposes, such that it is scaled to $\|\mathbf{w}\| = 1$, thereby giving each feature their proportional contribution to the preference I_j^π defined by Eq. (2.11). These weights will now be explored further, along with testing whether models are statistically significant to one another, using a Kolmogorov-Smirnov test with $\alpha = 0.05$.

For $\mathcal{P}_{j.rnd}^{10\times10}$ there is no statistical difference between models (2.69, 3.355, 3.358, 3.524), w.r.t. ρ and the latter three w.r.t. accuracy. These models are therefore equivalently 'best' for the problem space. As Fig. 8.6 shows, φ_3, φ_7 and φ_{11} are similar in value, and in the case of 3.358, then φ_9 has similar contribution as φ_3 for the other models. Which, as standalone

*Due to the superabundant number of models then calculating the *preferable* ξ_π from Eq. (7.3) is not viable.

Table 8.3: Mean validation accuracy and mean expected deviation from optimality, ρ, for all CDR models on the Pareto front from Fig. 8.5.

Problem	PREF NrFeat.Model	Accuracy (%) Optimality	Classification	ρ (%)	Pareto
$\mathcal{P}^{10\times10}_{j.rnd}$	**3.524**	91.55	62.57	**12.67**	▲
	3.358	91.82	62.74	12.90	▲
	3.355	91.90	62.71	12.92	▲
	2.69	91.02	61.41	12.92	
	1.11	80.77	55.78	21.63	
	1.13	85.26	57.17	22.79	
	16.1	**92.24**	**63.61**	30.47	▲
	2.111	91.52	59.69	32.68	
	1.6	89.85	58.33	33.08	
	1.3	89.86	58.34	33.41	
	3.300	91.91	60.05	51.87	
$\mathcal{P}^{10\times10}_{j.rndn}$	**3.281**	86.24	60.34	**12.89**	▲
	3.231	86.52	58.92	12.98	▲
	3.222	86.69	58.86	13.23	▲
	2.68	86.19	59.27	13.34	
	3.223	86.73	58.80	13.44	▲
	3.528	86.84	59.49	13.61	▲
	2.52	86.47	59.16	13.65	
	2.73	86.55	59.26	13.67	
	3.159	86.88	58.87	13.91	▲
	3.263	86.95	59.20	14.06	▲
	3.162	86.92	58.97	14.06	▲
	2.51	86.65	58.90	14.06	
	3.147	87.18	58.88	14.29	▲
	3.148	87.45	59.24	14.79	▲
	2.75	87.11	**60.45**	15.30	
	3.418	87.75	59.57	16.22	▲
	1.13	86.22	58.04	19.21	
	2.91	87.12	60.17	19.48	
	3.139	87.81	59.09	29.00	▲
	3.237	88.07	59.40	32.69	▲
	16.1	**88.86**	60.17	42.88	▲
$\mathcal{P}^{10\times10}_{f.rnd}$	**3.539**	95.22	64.97	**16.40**	▲
	3.151	96.06	64.31	16.75	▲
	3.216	96.28	**71.12**	16.78	▲
	2.94	92.79	63.12	16.88	
	3.213	96.30	71.05	17.22	▲
	2.111	94.16	65.07	17.73	
	2.51	95.83	64.21	17.95	
	1.7	87.59	61.74	19.05	
	1.6	92.61	62.91	19.18	
	16.1	**96.67**	70.58	22.50	▲

models are 1.6 and 1.3, respectively, and yield equivalent mean ρ and accuracy. As these features often coincide in job-shop it is justifiable to use only either one, as the it contains the same information as its counterpart.* Most likely, the equivalence of these features is indicating that the schedules are rarely able to dispatch in earlier slots, i.e., $\varphi_3 \approx \varphi_9$.

In addition, (2.111, 3.300) and (16.1, 3.355) are statistically insignificant w.r.t. validation accuracy for $\mathcal{P}^{10\times10}_{j.rnd}$. However, they have considerable performance difference w.r.t. ρ ($\Delta\rho \approx 18\%$). So even looking at stepwise optimality, $\xi^{\star}_{\pi}$ by itself is very fickle, because slight variations can be quite dramatic to the end result.

The solutions on the Pareto front for $\mathcal{P}^{10\times10}_{j.rndn}$ are quite a few models with no (or minimal) statistical difference w.r.t. ρ, and considerably more w.r.t. validation accuracy. Most notably are (3.281, 2.73, 2.75, 1.13) (note, first one has the lowest mean ρ), which are all statistically insignificant w.r.t. validation accuracy yet none w.r.t. ρ, with difference up to $\Delta\rho = 6.32\%$.

For $\mathcal{P}^{10\times10}_{f.rnd}$ almost all models are statistically different w.r.t. ρ, only exception is (1.6, 1.7). Although, w.r.t. validation accuracy, there are a few equivalent models, namely, (3.151, 2.51), (2.94, 1.6) and (3.216, 3.213, 16.1), with 1.2%, 2.3% and 5.75% difference in mean ρ, respectively.

It's interesting to inspect the full model for $\mathcal{P}^{10\times10}_{f.rnd}$, 16.1. Despite having similar contributions, yet statistically significantly different, as all the active features as (3.213, 3.216), then the (slight) interference from of other features, hinders the full model from achieving a low ρ. Only considering φ_8 and φ_{12} with either φ_3 and φ_9, boosts performance by 5.28% and 5.72%, respectively. Thereby stressing the importance of feature selection, to steer clear of over-fitting. Note, unlike $\mathcal{P}^{10\times10}_{j.rnd}$, now φ_3 differs from φ_9, indicating that there are some slots created, which could be better utilised. Moreover, looking at model 2.111 for $\mathcal{P}^{10\times10}_{f.rnd}$, which has similar contributions as the best model, 3.539. Then introducing a third feature, φ_{11}, is the key to the success of the CDR, with a boost of ρ performance by 1.33%.

Note, for both $\mathcal{P}^{10\times10}_{j.rnd}$ and $\mathcal{P}^{10\times10}_{j.rndn}$, model 1.13 is on the Pareto front. The model corresponds to feature φ_7, and in both cases has a weight strictly greater than zero (cf. Fig. 8.6). Revisiting Eq. (2.14), we observe that this implies the learning model was able to discover MWR as one of the Pareto solutions, and as is expected, there is no statistical difference to between 1.13 and MWR.

As one can see from Fig. 8.5, adding additional features to express the linear model boosts performance in both validation accuracy and expected mean for ρ, i.e., the Pareto fronts are cascading towards more desirable outcome with higher number of active features. However, there is a cut-off point for such improvement, as using all features is

*Note, $\varphi_3 \leq \varphi_9$, where $\varphi_3 = \varphi_9$ when J_j is the latest job on M_a, otherwise J_j is placed in a previously created slot on M_a.

generally considerably worse off due to overfitting of classifying the preference set.

Now, let's inspect the models corresponding to the minimum mean ρ and highest mean validation accuracy, highlighted in Table 8.3 and inspect the stepwise optimality for those models in Fig. 8.7, again using probability of randomly guessing an optimal move from **??** (denoted RND) as a benchmark. As one can see for both $\mathcal{P}^{10\times10}_{j.rnd}$ and $\mathcal{P}^{10\times10}_{j.rndn}$, despite having a higher mean validation accuracy overall, the probabilities vary significantly. A lower mean ρ is obtained when the validation accuracy is gradually increasing over time, and especially during the last phase of the scheduling.* Revisiting **??**, this trend indicates that it's likelier for the resulting makespan to be considerably worse off if suboptimal moves are made at later stages, than at earlier stages. Therefore, it's imperative to make the 'best' decision at the 'right' moment, not just look at the overall mean performance. Hence, the measure of validation accuracy as discussed in Section 8.6.1 should take into consideration the impact a suboptimal move yields on a step-by-step basis, e.g., weighted w.r.t. a curve such as depicted in **??**.

Let's revert back to the original SDRs discussed in **??** and compare the best CDR models, a box-plot for ρ is depicted in Fig. 8.8. Firstly, there is a statistical difference between all models, and clearly the CDR model corresponding to minimum mean ρ value, is the clear winner, and outperforms the SDRs substantially. However, the best model w.r.t. maximum validation accuracy, then the CDR model shows a lacklustre performance. In some cases it's better off, e.g., compared to LWR, yet for job-shop it doesn't surpass the performance of MWR. This implies, the learning model is over-fitting the training data. Results hold for the test set.

8.6.3 Conclusions

Current literature still hold single priority dispatching rules in high regard, as they are simple to implement and quite efficient. However, they are generally taken for granted as there is clear lack of investigation of *how* these dispatching rules actually work, and what makes them so successful (or in some cases unsuccessful)? For instance, of the four SDRs this study focuses on, why does MWR outperform so significantly for job-shop yet completely fail for flow-shop? MWR seems to be able to adapt to varying distributions of processing times, however, manipulating the machine ordering causes MWR to break down. By inspecting optimal schedules, and meticulously researching what's going on, every step of the way of the dispatching sequence, some light is shed where these SDRs vary

*It's almost illegible to notice this shift directly from Fig. 8.7, as the difference between the two best models is oscillating up to only 3% at any given step. In fact $\mathcal{P}^{10\times10}_{j.rndn}$ has the most clear difference w.r.t. classification accuracy of indicating when a minimum ρ model excels at choosing the preferred move.

w.r.t. the problem space at hand. Once these simple rules are understood, then it's feasible to extrapolate the knowledge gained and create new composite priority dispatching rules that are likely to be successful.

Creating new dispatching rules is by no means trivial. For job-shop there is the hidden interaction between processing times and machine ordering that's hard to measure. Due to this artefact, feature selection is of paramount importance, and then it becomes the case of not having too many features, as they are likely to hinder generalisation due to over-fitting in training. However, the features need to be explanatory enough to maintain predictive ability. For this reason Eq. (2.12) was limited to up to three active features, as the full feature set was clearly suboptimal w.r.t. its CMA-ES benchmark from Chapter 5. By using features based on the SDRs, along with some additional local features describing the current schedule, it was possible to 'discover' the SDRs when given only one active feature. Although there is not much to be gained by these models, they at least serve as a sanity check for the learning models are on the right track. Furthermore, by adding on additional features, a boost in performance was gained, resulting in a composite priority dispatching rule that outperformed all of the SDR baseline. Although, the best preference model of 3 active features is still not better than the CMA-ES model for $\mathcal{P}^{10\times10}_{j.rnd}$ using 16 features. However, we're starting to close in on the gap, as previously $\Delta\rho \approx 6\%$ (using $\Psi^{\mathrm{ES.C_{max}}}_{p}$) and now $\Delta\rho \approx 2\%$ (using CDR 3.524) in favour of evolutionary search (cf. Table 5.2).

When training the learning model, it's not sufficient to only optimize w.r.t. highest mean validation accuracy. As Section 8.6.2 showed, there is a trade-off between making the over-all best decisions versus making the right decision on crucial time points in the scheduling process, as **??** clearly illustrated. It is for this reason, traditional feature selection such as add1 and drop1 were unsuccessful in preliminary experiments, and thus resorting to having to exhaustively search all feature combinations. This also opens of the question of how should validation accuracy be measured? Since the model is based on learning preferences, both based on optimal versus suboptimal, and then varying degrees of sub-optimality. As we are only looking at the ranks in a black and white fashion, such that the makespans need to be strictly greater to belong to a higher rank, then it can be argued that some ranks should be grouped together if their makespans are sufficiently close. This would simplify the training set, making it (presumably) less of contradictions and more appropriate for linear learning. Or simply the validation accuracy could be weighted w.r.t. the difference in makespan. During the dispatching process, there are some pivotal times which need to be especially taken care off. **??** showed how making suboptimal decisions were more of a factor during the later stages, whereas for flow-shop the case was exact opposite.

Could discuss new sampling strategies, e.g., proportional to best/worst case, optimality, etc. – have done some experiments, but not clear what strategy is best, so only equal probability reported

Despite the abundance of information gathered by following an optimal trajectory, the knowledge obtained is not enough by itself. Since the learning model isn't perfect, it is bound to make a mistake eventually. When it does, the model is in uncharted territory as there is not certainty the samples already collected are able to explain the current situation. For this we propose investigating features from suboptimal trajectories as well, since the future observations depend on previous predictions. A straight forward approach would be to inspect the trajectories of promising SDRs or CDRs. In fact, it would be worth while to try out imitation learning by Ross and Bagnell (2010), Ross et al. (2011), such that the learned policy following an optimal trajectory is used to collect training data, and the learned model is updated. This can be done over several iterations, with the benefit being, that the states that are likely to occur in practice are investigated, and as such used to dissuade the model from making poor choices. Alas, this comes at great computational cost due to the substantial amounts of states that need to be optimised for their correct labelling. Making it only practical for job-shop of a considerable lower dimension.

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

Not done, but possible

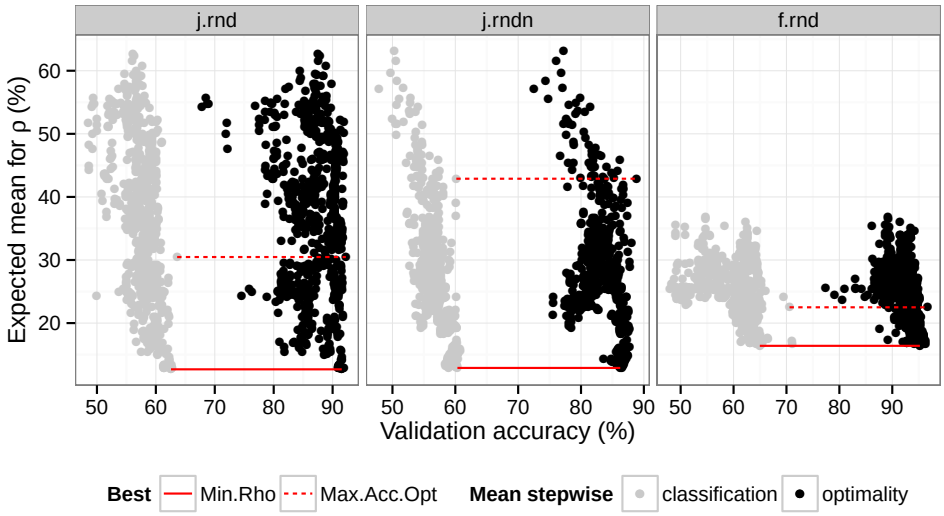


Figure 8.4: Various methods of reporting validation accuracy for preference learning

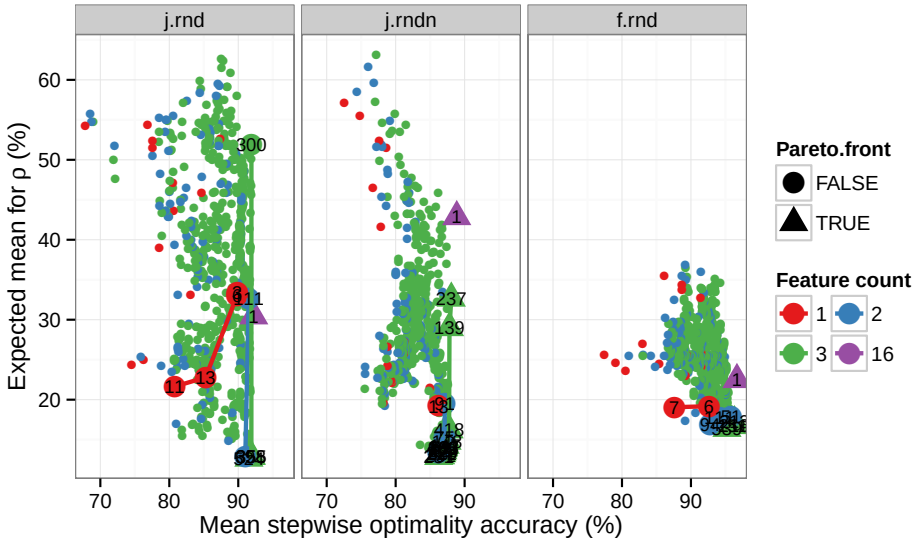


Figure 8.5: Scatter plot for validation accuracy (%) against its corresponding mean expected ρ (%) for all 697 linear models, based on either one, two, three or all d combinations of features. Pareto fronts for each active feature count based on maximum validation accuracy and minimum mean expected ρ (%), and labelled with their model ID. Moreover, actual Pareto front over all models is marked with triangles.

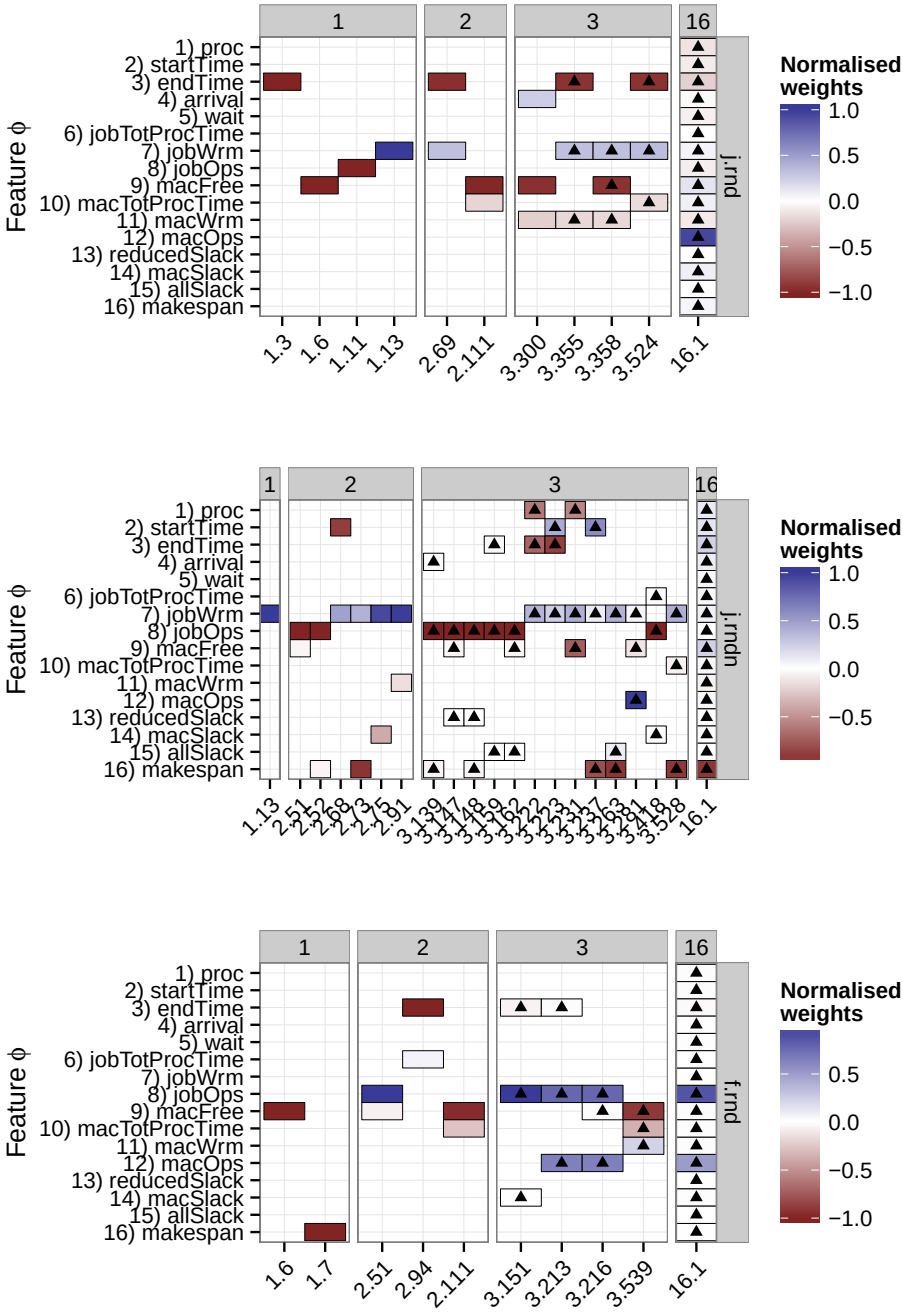


Figure 8.6: Normalised weights for CDR models from Table 8.3, models are grouped w.r.t. its dimensionality, d. Note, a triangle indicates a solution on the Pareto front.

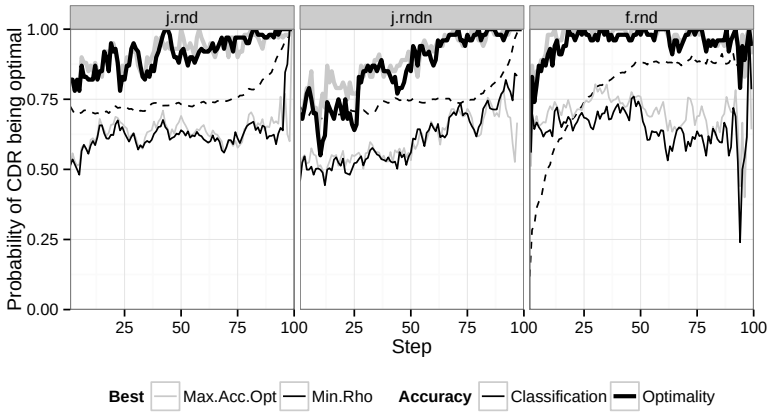


Figure 8.7: Probability of choosing optimal move for models corresponding to highest mean validation accuracy (grey) and lowest mean deviation from optimality, ρ, (black) compared to the baseline of probability of choosing an optimal move at random (dashed).

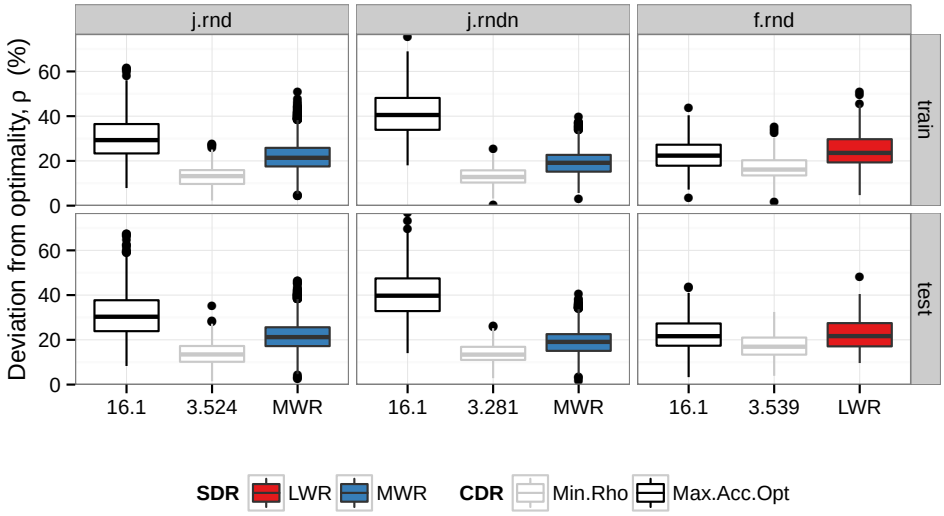


Figure 8.8: Box-plot for deviation from optimality, ρ, (%) for the best CDR models (cf. Table 8.3) and compared against the best SDRs from **??**, both for training and test sets.

9
Imitation Learning

> So a prudent man must always follow in the footsteps of great men and imitate those who have been outstanding. If his own prowess fails to compare with theirs, at least it has an air of greatness about it.
>
> Niccolò di Bernardo dei Machiavelli (1513)

Just as this quote applied to *new principalities acquired with one's own arms and prowess* centuries ago, it equally applies when setting up novel supervised learning algorithms. When it comes to designing algorithms there needs to be emphasis on where to innovate and imitate when visiting state-spaces. This study will show, that when using these guidelines when accumulating training data for supervised learning, it's possible to automate its generation in such a way that the resulting model will be an accurate representative of the instances it will later come across.

Eftirleifar galsa. Mun umorda seinna.

For this purpose, the JSP is used as a case study to illustrate a methodology for generating meaningful training set autonomously, which can be successfully learned using preference-based imitation learning (IL).

The approach was to use supervised learning to determine which feature states are preferable to others.

The training data from Paper I was created from optimal solutions of randomly generated problem instances, i.e., traditional *passive* imitation learning (IL). As JSP is a sequential decision making process, errors are bound to emerge. Due to compound effect

of making suboptimal dispatches, the model leads the schedule astray from learned state-spaces, resulting in the new input being foreign to the learned model.

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

Inspired by the work of Ross and Bagnell (2010), Ross et al. (2011), the methodology of generating training data will now be such that it will iteratively improve upon the model, such that the state-spaces learned will be representative of the state-spaces the eventual model would likely encounter, known as DAgger for imitation learning. Thereby, eliminating the ad-hoc nature of choosing trajectories to learn, by rather letting the model lead its own way in a self-perpetuating manner until it converges.

The outline of the paper is the following, **??** will define formalism for JSP, and **??** and Sections 9.1.1 and 9.1.2 explain the trajectories for sampling meaningful schedule state-spaces used in preference learning, along with some general adjustments for performance boost in **??**. Followed by experimental results in Section 9.3 with comparison for several randomly generated problem spaces. The paper finally concludes in Section 9.4 with discussion and conclusions.

9.1 Passive Imitation Learning

Using the terms from game-theory used in Cesa-Bianchi and Lugosi (2006), then our problem is a basic version of the sequential prediction problem where the predictor (or forecaster), π, observes each element of a sequence $\boldsymbol{\chi}$ of jobs, where at each time step $k \in \{1, \ldots, K\}$, before the k-th job of the sequence is revealed, the predictor guesses its value χ_k on the basis of the previous $k-1$ observations.

9.1.1 Prediction with Expert Advice

Let's assume we know the expert policy $\pi^\star$, which we can query what is the optimal choice of $\chi_k = j^*$ at any given time step k. Now we can use **??** to back-propagate the relationship between post-decision states and $\hat{\pi}$ with preference learning via our collected feature set, denoted Φ^{OPT}, i.e., we collect the features set corresponding following optimal tasks J_{j^*} from $\pi^\star$ in Alg. 2. This baseline trajectory sampling for adding features to the feature set is

a pure strategy where at each dispatch, an optimal task was originally introduced in **?** and explored further in **?**.

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

$$r_1 \succ r_2 \succ \cdots \succ r_{n'} \quad (n' \leq n) \tag{9.1}$$

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

$$\mathcal{O} = \left\{ r_i \ : \ r_i \propto \min_{J_j \in \mathcal{L}} C_{\max}^{\pi_\star}(\boldsymbol{x}^j) \right\} \tag{9.2}$$

found by solving the current partial schedule to optimality using a commercial software package such as Gurobi Optimization, Inc. (2014).

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

Experiments in **?** clearly showed that following the expert policy is not without its faults. There are many obstacles to consider to improve the model. For instance, their experiments Ψ to size l with equal probability. But inspecting the effects of making suboptimal choices varies as a function of times steps, perhaps its stepwise bias should rather be done proportional to the mean cumulative loss to a particular time step?

9.1. The sampling strategy for Ψ in **?** was Bias.1 and serves as a baseline. Bias.2 motivation is to give samples from dispatches that are less than likely to be optimal than simply at random (cf. **??**). On the other hand, Bias.3 and Bias.4 are more focused on sampling w.r.t. the final measure, where the mean ρ is given *one* suboptimal move, otherwise it's assumed expert policy is followed (cf. **??**). Lastly, Bias.5 and Bias.6 are very simplified versions of the aforementioned strategies, depending on the problem space at hand. **??** depicts the stepwise bias strategies for the problem spaces in **??**.

Following strategies for stepwise bias will be proportional to:

Bias.1 equal probability (equal)

Bias.2 inverse optimality for random dispatches (opt)

Bias.3 best case scenario for mean ρ (bcs)

Bias.4 worst case scenario for mean ρ (wcs)

Bias.5 twice as much weight on the first half of the dispatches (dbl1st)

Algorithm 3 Pseudo code for choosing job J_{j^*} following a perturbed leader.

Require: Ranking $r_1 \succ r_2 \succ \cdots > r_{n'}$ $(n' \leq n)$ of the job-list, $\mathcal{L}$ ▷ query $\pi_\star$

procedure PERTURBEDLEADER($\mathcal{L}, \pi_\star$)
  $\varepsilon \leftarrow 0.1$ ▷ likelihood factor
  $p \leftarrow \mathcal{U}(0,1) \in [0,1]$ ▷ uniform probability
  $\mathcal{O} \leftarrow \{j \in \mathcal{L} \ : \ r_j = r_1\}$ ▷ optimal job-list
  $\mathcal{S} \leftarrow \{j \in \mathcal{L} \ : \ r_j > r_1\}$ ▷ suboptimal job-list
  if $p < \varepsilon$ **and** $n' > 1$ **then**
    return $j^* \in \{j \in \mathcal{S} \ : \ r_j = r_2\}$ ▷ any second best job
  else
    return $j^* \in \mathcal{O}$ ▷ any optimal job
  end if
end procedure

Bias.6 twice as much weight on the second half of the dispatches (dbl2nd)

The reader is referred to **?** for a detailed explanation of how to obtain the reference values for Bias.2 to Bias.4. The sampling strategy for Υ in **??** was Bias.1 and serves as a baseline. Bias.2 motivation is to give samples from dispatches that are less than likely to be optimal than simply at random. On the other hand, Bias.3 and Bias.4 are more focused on sampling w.r.t. the final measure, where the mean ρ is given *one* suboptimal move, otherwise it's assumed expert policy is followed. Lastly, Bias.5 and Bias.6 are very simplified versions of the aforementioned strategies, depending on the problem space at hand. **??** depicts the stepwise bias strategies for the problem spaces in **??**.

9.1.2 Follow the Perturbed Leader

By allowing a predictor to randomise it's possible to achieve improved performance Cesa-Bianchi and Lugosi (2006), **?**, which is the inspiration for our new strategy, where we follow the Perturbed Leader (PL). Its pseudo code is given in Alg. 3 and describes how the expert policy (i.e. optimal trajectory) from Section 9.1.1 is subtly "perturbed" with $\varepsilon = 10\%$ likelihood, by choosing a job corresponding to the second best $C_{\max}$ instead of a optimal one with some small probability.

t

9.1.3 Summary

Results showed that the expert policy is a promising starting point. However, since job-shop is a sequential prediction problem, all future observations are dependent on previous

operations. Therefore, learning sampled states that correspond only to optimal or near-optimal schedules isn't of much use when the preference model has diverged too far. This is due to the learner's predictions affects future input observations during its execution, which violates the crucial i.i.d. assumptions of the learning approach, and ignoring this interaction leads to poor performance. In fact, Ross and Bagnell (2010) proves, that assuming the preference model has a training error of ε, then the total compound error (for all K dispatches) the classifier induces itself grows quadratically, $O(\varepsilon K^2)$, for the entire schedule, rather than having linear loss, $O(\varepsilon K)$, if it were i.i.d.

9.2 Active Imitation Learning

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

DAgger can be interpreted as Follow-the-leader algorithm in that at iteration i we pick the best policy $\hat{\pi}_i$ in hindsight, i.e., under all trajectories seen so far over the iterations

"Ross et al. (2011) approach is similar to regularised follow the leader algorithm from sec." In these models the adversary and nature are the same, and the nature chooses a new cost function for each action of the learner at the each iteration of the game. The goal is to minimise the regret that the learner would suffer, compared to the time that if it knew all of the costs imposed by nature in hindsight and had chosen a fixed strategy as the response Ross et al use a regret minimisation setting for learning to drive a computer simulated car, where the output is a sequence of actions in a limited horizon. in their problem the true cost of taking action a in state s, $C(s, a)$ is not known, but they use some expert's knowledge about the loss $l(s, \pi)$ incurred by the policy $a = \pi(s)$ – policy is the function $\pi : \mathcal{S} \to \mathcal{D}(\mathcal{A})$ that maps an state to an action or a distribution over action, and is almost equivalent hypothesis function $h : \mathcal{X} \to \mathcal{Y}$

To automate this process, inspiration from *active* imitation learning presented in Ross et al. (2011) is sought, called *Dataset Aggregation* (DAgger) method, which addresses a no-regret algorithm in an on-line learning setting. The novel meta-algorithm for IL learns a deterministic policy guaranteed to perform well under its induced distribution of states. The method is closely related to Follow-the-leader (cf. Section 9.1.2), however, with a more sophisticated leverage to the expert policy. In short, it entails the model π_i that queries an expert policy (same as in Section 9.1.1), $\pi_\star$, its trying to mimic, but also ensuring the

learned model updates itself in an iterative fashion, until it converges. The benefit of this approach is that the states that are likely to occur in practice are also investigated and as such used to dissuade the model from making poor choices. In fact, the method queries the expert about the desired action at individual post-decision states which are both based on past queries, and the learner's interaction with the *current* environment.

DAgger has been proven successful on a variety of benchmarks, such as: the video games Super Tux Kart and Super Mario Bros. or handwriting recognition – in all cases greatly improving traditional supervised imitation learning approaches Ross et al. (2011), and real-world applications, e.g. autonomous navigation for large unmanned aerial vehicles Ross et al. (2013). To illustrate the effectiveness of DAgger, the Super Mario Bros. experiment gives a very simple and informative understanding of the benefits of the algorithm. In short, Super Mario Bros. is a platform game where the protagonist, Mario, must move across the stage without being hit by enemies or falling through gaps within a certain time limit. One of the reasons the supervised approaches failed, were due to Mario getting stuck up against an obstacle, instead of jumping over it. However, the expert would always jump over them at a greater distance beforehand, and therefore the learned controller would not know of these scenarios. With iterative methods, Mario would encounter these problematic situations and eventually learn how to get himself unstuck.

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

$$\pi_i = \beta_i \pi_\star + (1 - \beta_i)\hat{\pi}_{i-1} \tag{9.3}$$

where $\pi_\star$ is the expert policy and $\hat{\pi}_{i-1}$ is the learned model from the previous iteration. Note, for the initial iteration, $i = 0$, a pure strategy of $\pi_\star$ is followed. Hence, $\hat{\pi}_0$ corresponds to the preference model from Section 9.1.1 (i.e. $\Phi^{\text{IL}0} = \Phi^{\text{OPT}}$).

Equation (9.3) shows that β controls the probability distribution of querying the expert policy $\pi_\star$ instead of the previous imitation model, $\hat{\pi}_{i-1}$. The only requirement for $\{\beta_i\}_i^\infty$ according to Ross et al. (2011) is that $\lim_{T\to\infty} \frac{1}{T} \sum_{i=0}^{T} \beta_i = 0$ to guarantee finding a policy $\hat{\pi}_i$ that achieves ε surrogate loss under its own state distribution limit.

Algorithm 4 explains the pseudo code for how to collect partial training set, $\Phi^{\text{IL}i}$ for i-th iteration of imitation learning. Subsequently, the resulting preference model, $\hat{\pi}_i$, learns on the aggregated datasets from all previous iterations, namely,

$$\Phi^{\text{DA}i} = \bigcup_{i'=0}^{i} \Phi^{\text{IL}i'} \tag{9.4}$$

Algorithm 4 Pseudo code for choosing job J_{j^*} using imitation learning (dependent on iteration i) to collect training set $\Phi^{\text{IL}i}$; either by following optimal trajectory, $\pi_\star$, or preference model from previous iterations, $\hat{\pi}_{i-1}$.

Require: $i \geq 0$
Require: Ranking $r_1 \succ r_2 \succ \cdots > r_{n'}$ $(n' \leq n)$ of the job-list, $\mathcal{L}$ ▷ query $\pi_\star$

```
procedure ImitationLearning(i, π̂_{i−1}, π_⋆)
    p ← U(0, 1) ∈ [0, 1]                                   ▷ uniform probability
    if i > 0 then
        if unsupervised then
            β_i ← 0                                        ▷ always apply imitiation
        else if decreasing supervision then
            β_i ← 0.5^i                                    ▷ liklier to choose imitation with each iteration
        else (fixed supervision)
            β_i ← 0.5                                      ▷ equally likely to choose optimal vs. imitation
        end if
    else (fixed supervision)
        β_i ← 1                                            ▷ always follow expert policy (i.e. optimal)
    end if
    if p > β_i then
        return j* ← argmax_{j∈L} { I_j^{π̂_{i−1}} }         ▷ best job based on π̂_{i−1}, cf. Alg. 2
    else
        O ← { j ∈ L : r_j = r_1 }                          ▷ optimal job-list
        return j* ∈ O                                      ▷ any optimal job
    end if
end procedure
```

and its update procedure is detailed in Alg. 5.

9.3 Experiments

Performance boost

In order to boost training accuracy, two strategies were explored:

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

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

Note, that in preliminary experiments for Boost.1 showed no statistical significance in boost of performance, hence $l_{\max}$ is the same as set in Eq. (8.3). However, Boost.2 strategy showed a considerable change in performance. Summary of N_{train} is given in Table 9.1.

Algorithm 5 DAgger: Dataset Aggregation for JSP

Require: $T \geq 1$

procedure DAGGER($\pi_\star$, Φ^{OPT}, T)
  $\Phi^{\text{IL}_0} \leftarrow \Phi^{\text{OPT}}$ ▷ initialize dataset
  $\hat{\pi}_0 \leftarrow$ TRAIN(Φ^{IL_0}) ▷ initial model, equivalent to Section 9.1.1
  for $i \leftarrow 1$ **to** T **do** ▷ at each imitation learning iteration
    Let $\pi_i = \beta_i \pi_\star + (1 - \beta_i)\hat{\pi}_{i-1}$ ▷ Eq. (9.3)
    Sample a K-solution using π_i ▷ cf. Alg. 4: IMITATIONLEARNING($i, \hat{\pi}_{i-1}, \pi_\star$)
    $\Phi^{\text{IL}i} = \{(s, \pi_\star(s))\}$ ▷ visited states by π_i and actions given by expert
    $\Phi^{\text{DA}i} \leftarrow \Phi^{\text{DA}i-1} \cup \Phi^{\text{IL}i}$ ▷ aggregate datasets, cf. Eq. (9.4)
    $\hat{\pi}_{i+1} \leftarrow$ TRAIN($\Phi^{\text{DA}i}$) ▷ preference model from **??**
  end for
  return best $\hat{\pi}_i$ on validation ▷ best preference model
end procedure

$n \times m$	$N^{\langle\text{default}\rangle}_{\text{train}}$	$N^{\text{OPT}}_{\text{train, EXT}}$	$N^{\text{DA}i}_{\text{train, EXT}}$
6×5	500	5000	$500(i+1)$
10×10	300	1000	$300(i+1)$

Table 9.1: Number of problem instances, $\mathcal{P} = \{\mathbf{p}_i, \boldsymbol{\sigma}_i\}_{i=1}^{N}$, explored for the collection of training set, Φ, in experimental setting.

Note, for the conventional Φ^{OPT} trajectory the extended training set was simply obtained by iterating over more examples. However, for the DAgger trajectories the extended set consisted of each iteration encountering N_{train} *new* problem instances. For a grand total of

$$N^{\text{DA}i}_{\text{train, EXT}} = N_{\text{train}} \cdot (i+1) \tag{9.5}$$

problem instances explored for the aggregated extended training set used for the learning model at iteration i.

DAgger Parameters

Due to time constraints, only $T = 7$ iterations will be inspected. In addition, there will be three mixed strategies for $\{\beta_i\}_{i=0}^{T}$ in Eq. (9.3) considered:

DA.1 *fixed* supervision with $\beta_i = 0.5$ save for $\beta_0 = 1$,

DA.2 *decreasing supervision* with $\beta_i = 0.5^i$,

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

Note, DA.2 starts as DA.1 and decays exponentially towards DA.3. Moreover, DA.3 is a simple parameter-free version of the DAgger algorithm and often performs best in practice Ross et al. (2011).

9.4 Conclusions

"What general lessons might be learnt from this study?"

DAgger for job-shop is not sensitive to choice of β_i in Eq. (9.3).

Place work in wider context

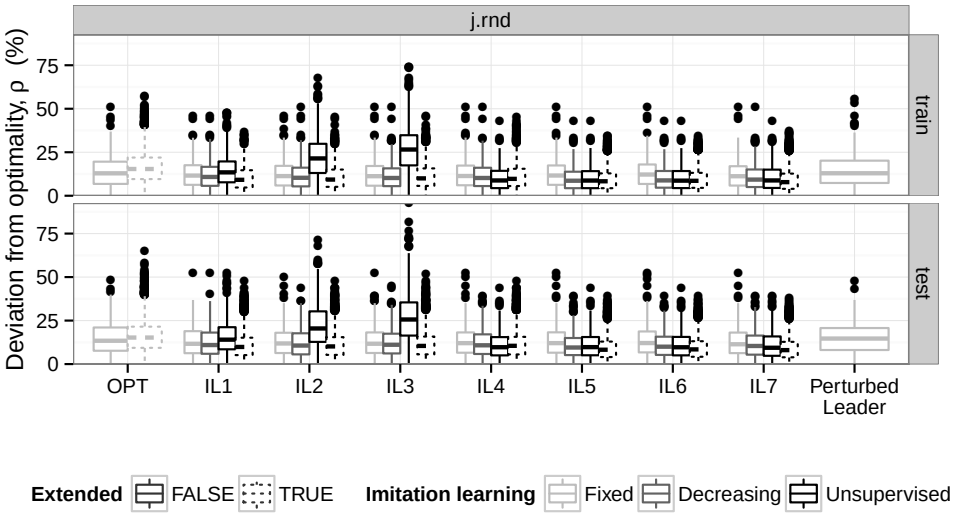
"Training such a predictor, however, is non-trivial as the interdependencies in the sequence of predictions make global optimisation is to leverage information local to modules to aid learning ... To provide good guarentees and performance in practice in this non-i.i.d. (as predictions are interdependent), we also leverage key iterative training methods developped in prior work for imitation learning and structured prediction"

George Bernard Shaw once said: "Imitation is not just the sincerest form of flattery – it's the sincerest form of learning."

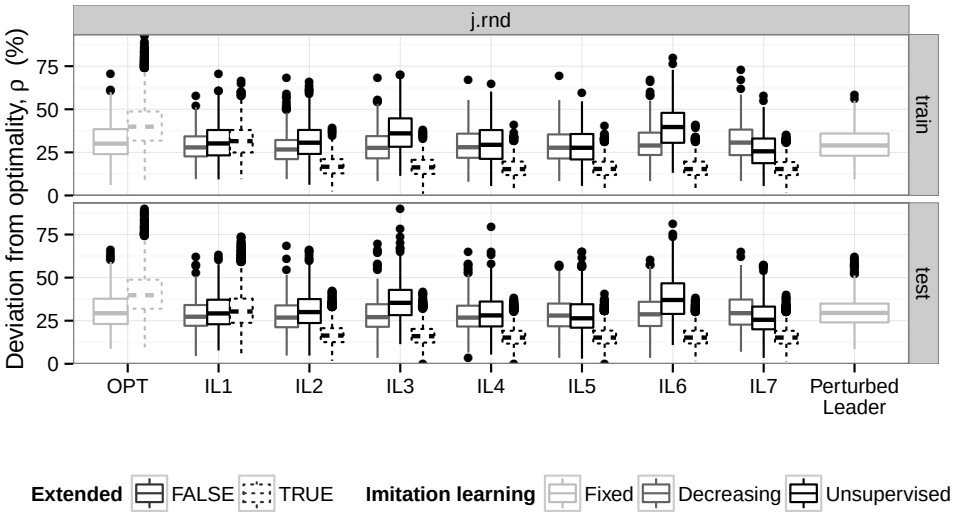
Future Work: Judah et al. (2012) Unfortunately [DAgger] query the expert quite aggressively making them impractical for human experts. In contrast, [Reduction-based Active Imitation Learning (RAIL)] focuses on active querying for the purpose of minimizing the expert's labelling effort. Like our work, they also require a dynamics simulator to help select queries

Maximum Mean Discrepancy (MMD) imitation learning by Kim and Pineau (2013) is an iterative algorithm similar to DAgger. However, the expert policy is only queried when needed in order to reduce computational cost. This occurs when a metric of a new state is sufficiently large enough from a previously queried states (to ensure diversity of learned optimal states). Moreover, in DAgger all data samples are equally important, irrespective of its iteration, which can require great number of iterations to learn how to recover from the mistakes of earlier policies. To address the naivety of the data aggregation, MMD suggests only aggregating a new data point if it is sufficiently different to previously gathered states, *and* if the current policy has made a mistake. Additionally, there are multiple policies, each specializing in a particular region of the state space where previous policies made mistakes. Although MMD has better empirical performance (based on robot applications), it requires defining metrics, which in the case of job-shop is non-trivial (cf. **?**),

*$\beta_0 = 1$ and $\beta_i = 0, \forall i > 0$.



(a) 6×5



(b) 10×10

Figure 9.1: Box plot for deviation from optimality, ρ, (%) using either imitation learning or following perturbed leader strategies.

and fine-tuning thresholds etc., whereas DAgger can be straightforwardly implemented, parameter-free and obtains competitive results, although with some computational overhead due to excess expert queries.

In fact, it's possible to circumvent querying the expert altogether and still have reasonable performance. By applying Locally Optimal Learning to Search (LOLS) Chang et al. (2015) it is possible to use imitation learning (similar to DAgger framework) when the reference policy is poor (i.e. $\pi_\star$ in Eq. (9.3) is suboptimal), although it's noted that the quality (w.r.t near-optimality) of reference policy is in accordance to its performance, as is to be expected.

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

The Gryphon

10

Experiments

10.0. Compare CMA-ES to PREF models

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.

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

10.0. When applying rollout features from **??**, then is sensible to keep track of the best solution found (even though they hadn't been followed), they will referred to as *fortified* solution.

Table 10.1: Results for time dependent models, i.e., each consists of K-models – one for each dispatch step.

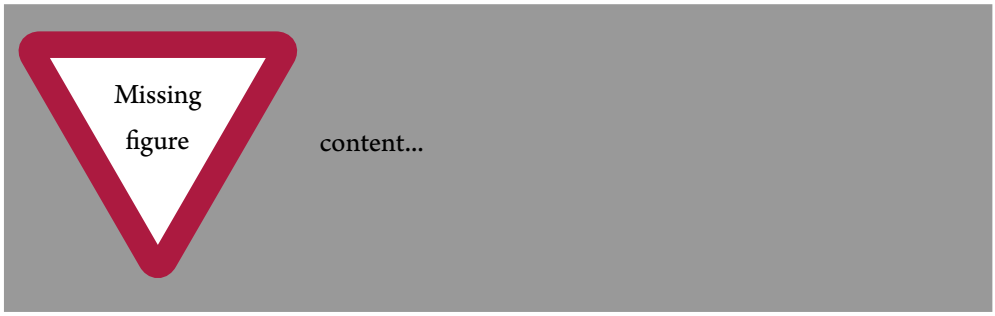
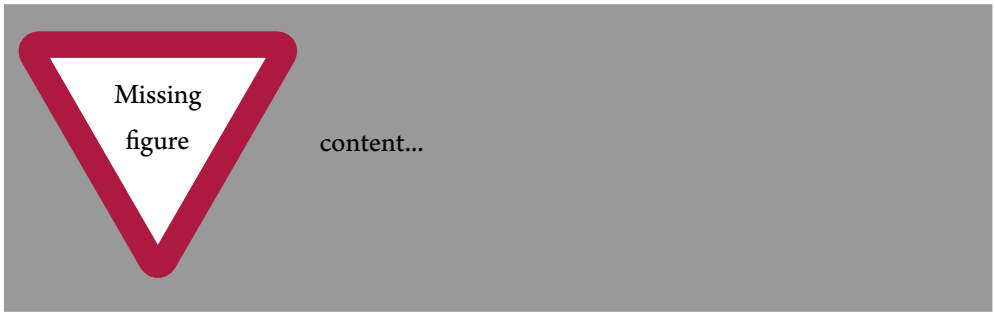


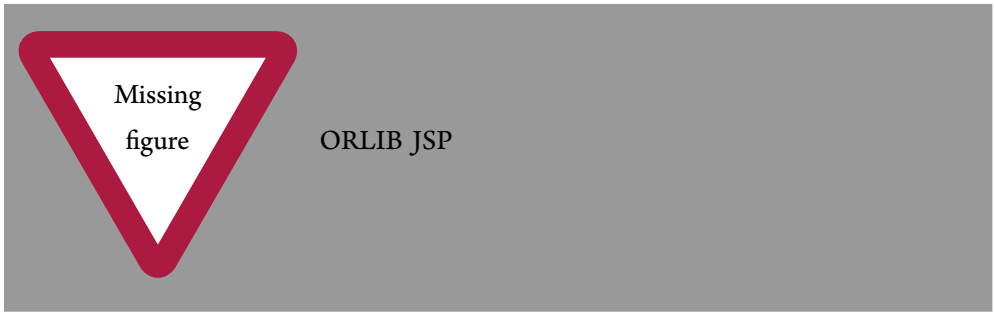
Table 10.2: Results for time independent models

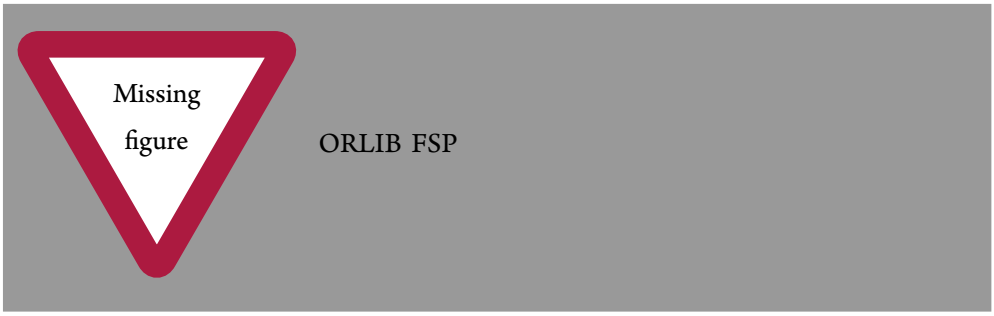


10.1 Time dependent models

10.2 Time independent models

10.3 ORLIB





The weights for Eq. (2.12) in Paper I 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.

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

The Duchess

11 Conclusions

11.0. Write overall conclusions of dissertation!

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

We saw in Section 8.5 that preference models using training data from following SDR policy, i.e., $\Phi^{\langle \text{SDR} \rangle}$, are good for improving its original heuristic. However, this did not transcend for $\Psi^{\langle \text{CMA-ES} \rangle}$. Perhaps, if $\Phi^{\langle \text{CMA-ES} \rangle}$ wasn't based on following the CMA-ES trajectory, but rather using the actual features encountered during its optimisation. Alas, CMA-ES used a computational budget of 50,000 function evaluations, each consisting of the expectation of N_{train} problem instances. So even though Fig. 5.1 becomes relatively stable after a few generations, it would still yield a gigantic feature set that needs to be filtered before going through the optimisation phase of correctly labelling them.

The analysis-phase of ALICE is heavily dependent on having an expert policy one wants to mimic, i.e., knowing the *optimal* solutions for the sake of imitation learning.

Understandably, knowing the true optimum is an unreasonable claim in many situations, especially for high dimensional problem instances. Luckily, there seems to be the possibility to circumvent querying the expert altogether, and still have reasonable perfor-

mance. By applying *locally optimal learning to search* (Chang et al., 2015) it is possible to use imitation learning even when the reference policy is poor. Although it's noted that the quality (w.r.t near-optimality) of reference policy is in accordance to its performance, as is to be expected.

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.

M. Ancău. On solving flowshop scheduling problems. *Proceedings of the Romanian Academy. Series A*, 13(1):71–79, 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`.

D. A. Berry and B. Fristedt. *Bandit Problems: Sequential Allocation of Experiments*. Springer, October 1985.

D. P. Bertsekas, J. N. Tsitsiklis, and C. Wu. Rollout algorithms for combinatorial optimization. *Journal of Heuristics*, 3(3):245–262, 1997.

N. Biggs, E. K. Lloyd, and R. J. Wilson. *Graph Theory, 1736-1936*. Clarendon Press, New York, NY, USA, 1986.

P. Brandimarte. Routing and scheduling in a flexible job shop by tabu search. *Annals of Operations Research*, 41(3):157–183, 1993.

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. *Revue française d'automatique, d'informatique et de recherche opérationnelle. Recherche opérationnelle*, 12(4):333–350, 1978.

L. Carroll. *Alice's Adventures in Wonderland*. Macmillan, 1865.

L. Carroll. *Through the Looking-Glass, and What Alice Found There*. Macmillan, 1871.

N. Cesa-Bianchi and G. Lugosi. *Prediction, Learning, and Games*, chapter 4. Cambridge University Press, New York, NY, USA, 2006. ISBN 0521841089.

C.-C. Chang and C.-J. Lin. LIBSVM: A library for support vector machines. *ACM Transactions on Intelligent Systems and Technology*, 2:27:1–27:27, 2011. Software available at `http://www.csie.ntu.edu.tw/~cjlin/libsvm`.

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.

K. Chang, A. Krishnamurthy, A. Agarwal, H. D. III, and J. Langford. Learning to search better than your teacher. In *Proceedings of The 32nd International Conference on Machine Learning*, pages 2058–2066, 2015.

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.

N. di Bernardo dei Machiavelli. *The Prince*, chapter 6. Penguin Books, 1513. Trans. George Bull, 1961.

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.

C. Duin and S. Voß. The pilot method: A strategy for heuristic repetition with application to the steiner problem in graphs. *Networks*, 34:181–191, 1999.

R.-E. Fan, K.-W. Chang, C.-J. Hsieh, X.-R. Wang, and C.-J. Lin. LIBLINEAR: A library for large linear classification. *Journal of Machine Learning Research*, 9:1871–1874, 2008. URL `http://www.csie.ntu.edu.tw/~cjlin/liblinear`.

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

Free Software Foundation, Inc. GLPK (gnu linear programming kit) (version 4.55) [software], 2014. URL `http://www.gnu.org/software/glpk/`.

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.

E. Geirsson. Rollout algorithms for job-shop scheduling. Master's thesis, University of Iceland, Reykjavík, Iceland, May 2012.

B. Giffler and G. L. Thompson. Algorithms for solving production-scheduling problems. *Operations Research*, 8(4):487–503, 1960.

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.

Gurobi Optimization, Inc. Gurobi optimization (version 6.0.0) [software], 2014. URL `http://www.gurobi.com/`.

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.

T. Helleputte. *LiblineaR: Linear Predictive Models Based on the LIBLINEAR C/C++ Library*, 2015. R package version 1.94-2.

J. Heller. Some numerical experiments for an m× j flow shop and its decision-theoretical aspects. *Operations Research*, 8(2):pp. 178–184, 1960.

R. Herbrich, T. Graepel, and K. Obermayer. *Large Margin Rank Boundaries for Ordinal Regression*, chapter 7. MIT Press, Mar. 2000.

T. Hildebrandt, J. Heger, and B. Scholz-Reiter. Towards improved dispatching rules for complex shop floor scenarios: a genetic programming approach. *GECCO '10: Proceedings of the 12th annual conference on Genetic and evolutionary computation*, pages 257–264, 2010.

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.

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.

T. Joachims. Optimizing search engines using clickthrough data. In *Proceedings of the eighth ACM SIGKDD international conference on Knowledge discovery and data mining*, KDD '02, pages 133–142, New York, NY, USA, 2002. ACM.

K. Judah, A. Fern, and T. G. Dietterich. Active imitation learning via reduction to I.I.D. active learning. *CoRR*, abs/1210.4876, 2012. URL `http://arxiv.org/abs/1210.4876`.

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

A. Keane. Genetic programming, logic design and case-based reasoning for obstacle avoidance. In C. Dhaenens, L. Jourdan, and M.-E. Marmion, editors, *Learning and Intelligent Optimization*, volume 8994 of *Lecture Notes in Computer Science*, pages 104–118. Springer International Publishing, 2015. ISBN 978-3-319-19083-9.

B. Kim and J. Pineau. Maximum mean discrepancy imitation learning. In *Robotics: Science and Systems*, 2013.

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.

A. H. Land and A. G. Doig. An automatic method of solving discrete programming problems. *Econometrica*, 28(3):497–520, 1960.

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.

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.

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.

A. S. Manne. On the job-shop scheduling problem. *Operations Research*, 8(2):219–223, 1960.

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.

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.

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

F. Pezzella, G. Morganti, and G. Ciaschetti. A genetic algorithm for the flexible job-shop scheduling problem. *Computers & Operations Research*, 35(10):3202–3212, 2008. Part Special Issue: Search-based Software Engineering.

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.

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.

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.

K. H. Rosen. *Discrete Mathematics and Its Applications*, chapter 9, pages 631–700. McGraw-Hill, Inc., New York, NY, USA, 5 edition, 2003.

S. Ross and D. Bagnell. Efficient reductions for imitation learning. In Y. W. Teh and D. M. Titterington, editors, *Proceedings of the Thirteenth International Conference on Artificial Intelligence and Statistics (AISTATS-10)*, volume 9, pages 661–668, 2010.

S. Ross, G. J. Gordon, and D. Bagnell. A reduction of imitation learning and structured prediction to no-regret online learning. In G. J. Gordon and D. B. Dunson, editors, *Proceedings of the Fourteenth International Conference on Artificial Intelligence and Statistics (AISTATS-11)*, volume 15, pages 627–635. Journal of Machine Learning Research - Workshop and Conference Proceedings, 2011.

S. Ross, N. Melik-Barkhudarov, K. Shankar, A. Wendel, D. Dey, J. Bagnell, and M. Hebert. Learning monocular reactive uav control in cluttered natural environments. In *Robotics and Automation (ICRA), 2013 IEEE International Conference on*, pages 1765–1772, May 2013.

B. Roy and B. Sussmann. Les problemes d'ordonnancement avec contraintes disjonctives. *Note D.S.*, 9, 1964.

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.

T. P. Rúnarsson. Ordinal regression in evolutionary computation. In T. P. Rúnarsson, 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. P. Rúnarsson, M. Schoenauer, and M. Sebag. Pilot, rollout and monte carlo tree search methods for job shop scheduling. In Y. Hamadi and M. Schoenauer, editors, *Learning and Intelligent Optimization*, Lecture Notes in Computer Science, pages 160–174. Springer Berlin Heidelberg, 2012.

J. Shawe-Taylor and N. Cristianini. *Kernel Methods for Pattern Analysis*. Cambridge University Press, New York, NY, USA, 2004.

K. Smith-Miles and S. Bowly. Generating new test instances by evolving in instance space. *Computers & Operations Research*, 63:102–113, 2015.

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.

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.

W. Xia and Z. Wu. An effective hybrid optimization approach for multi-objective flexible job-shop scheduling problems. *Computers & Industrial Engineering*, 48(2):409–425, 2005.

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.

What is the use of repeating all that stuff, if you don't explain it as you go on? It's by far the most confusing thing I ever heard!

The Mock Turtle

A Ordinal Regression

Ordinal regression has been previously presented in Rúnarsson (2006), but given here for completeness. The preference learning task of linear classification presented there is based on the work proposed in (Fan et al., 2008, Lin et al., 2008). The modification relates to how the point pairs are selected and the fact that a $L2$-regularized logistic regression is used.

A.1 Preference set

The ranking problem is specified by a *preference set*,

$$\Psi := \{(\mathbf{x}_i, y_i)\}_{i=1}^{N} \subset X \times Y \tag{A.1}$$

consisting of N (solution, rank)-pairs, where $Y = \{r_1, \ldots, r_N\}$ is the outcome space with ordered ranks $r_1 > r_2, > \ldots > r_N$.

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

$$\mathbf{x}_i \succ \mathbf{x}_j \quad \Leftrightarrow \quad h(\mathbf{x}_i) > h(\mathbf{x}_j) \tag{A.2}$$

where the symbol $\succ$ denotes 'is preferred to.'

In ordinal regression the task is to obtain function h that can for a given pair $(\mathbf{x}_i, y_i)$ and $(\mathbf{x}_j, y_j)$ distinguish between two different outcomes: $y_i > y_j$ and $y_j > y_i$. The task is, therefore, transformed into the problem of predicting the relative ordering of all possible pairs of examples (Herbrich et al., 2000, Joachims, 2002). However, it is sufficient to consider only all possible pairs of adjacent ranks (see also Shawe-Taylor and Cristianini (2004) for yet an alternative formulation). The preference set, composed of pairs, is then as follows,

$$\Psi = \left\{ (\mathbf{x}_k^{(1)}, \mathbf{x}_k^{(2)}), t_k = \operatorname{sign}(y_k^{(1)} - y_k^{(2)}) \right\}_{k=1}^{N'} \subset X \times Y \tag{A.3}$$

where $(y_k^{(1)} = r_i) \wedge (y_k^{(2)} = r_{i+1})$, and vice versa $(y_k^{(1)} = r_{i+1}) \wedge (y_k^{(2)} = r_i)$, resulting in $N' = 2(N-1)$ possible adjacently ranked preference pairs. The rank difference is denoted by $t_k \in \{-1, 1\}$.

In order to generalize the technique to different solution data types and model spaces an implicit kernel-defined feature space $\Phi \subset \mathbb{R}^d$ of dimension d, with corresponding feature mapping $\boldsymbol{\varphi} : X \mapsto \Phi$ is applied, i.e., the feature vector $\boldsymbol{\varphi}(\mathbf{x}) = [\varphi_1(\mathbf{x}), \ldots, \varphi_d(\mathbf{x})]^T \in \Phi$. Thus the preference set defined by Eq. (A.3) is redefined as follows,

$$\Psi = \left\{ \left(\boldsymbol{\varphi}(\mathbf{x}_k^{(1)}), \boldsymbol{\varphi}(\mathbf{x}_k^{(2)}) \right), t_k = \operatorname{sign}(y_k^{(1)} - y_k^{(2)}) \right\}_{k=1}^{N'} \subset \Phi \times Y. \tag{A.4}$$

A.2 Ordinal Regression

The function used to induce the preference is defined by a linear function in the kernel-defined feature space,

$$h(\mathbf{x}) = \sum_{i=1}^{d} w_i \varphi_i(\mathbf{x}) = \left\langle \mathbf{w} \cdot \boldsymbol{\varphi}(\mathbf{x}) \right\rangle \tag{A.5}$$

where $\mathbf{w} = [w_1, \ldots, w_d] \in \mathbb{R}^d$ has weight w_i for feature φ_i.

The aim now is to find a function h that encounters as few training errors as possible on Ψ. Applying the method of large margin rank boundaries of ordinal regression described in Herbrich et al. (2000), the optimal $\mathbf{w}^*$ is determined by solving the following task,

$$\min_{\mathbf{w}} \quad \tfrac{1}{2}\left\langle \mathbf{w} \cdot \mathbf{w} \right\rangle + \tfrac{C}{2} \sum_{k=1}^{N'} \xi_k^2 \tag{A.6}$$

subject to $t_k \left\langle \mathbf{w} \cdot (\boldsymbol{\varphi}(\mathbf{x}_k^{(1)}) - \boldsymbol{\varphi}(\mathbf{x}_k^{(2)}) \right\rangle \geq 1 - \xi_k$ and $\xi_k \geq 0$, $k = 1, \ldots, N'$. The degree of constraint violation is given by the margin slack variable ξ_k and when greater than 1 the

corresponding pair are incorrectly ranked. Note that,

$$h(\mathbf{x}_i) - h(\mathbf{x}_j) = \left\langle \mathbf{w} \cdot \boldsymbol{\varphi}(\mathbf{x}_i) - \boldsymbol{\varphi}(\mathbf{x}_j) \right\rangle \tag{A.7}$$

and minimising $\left\langle \mathbf{w} \cdot \mathbf{w} \right\rangle$ in Eq. (A.6) maximises the margin between rank boundaries, i.e., the distance between adjacently ranked pair $h(\mathbf{x}^{(1)})$ and $h(\mathbf{x}^{(2)})$.

A.3 Logistic Regression

Let $\mathbf{z}$ denote either $\boldsymbol{\varphi}(\mathbf{x}_k^{(1)}) - \boldsymbol{\varphi}(\mathbf{x}_k^{(2)})$ with $t_k = +1$ or $\boldsymbol{\varphi}(\mathbf{x}_k^{(2)}) - \boldsymbol{\varphi}(\mathbf{x}_k^{(1)})$ with $t_k = -1$, positive or negative example respectively.

Logistic regression learns the optimal parameters $\mathbf{w} \in \mathbb{R}^d$ determined by solving the following task,

$$\min_{\mathbf{w}} \quad \tfrac{1}{2}\left\langle \mathbf{w} \cdot \mathbf{w} \right\rangle + C \sum_{i=1}^{N'} \log\left(1 + e^{-y_i \left\langle \mathbf{w} \cdot \mathbf{z}_i \right\rangle}\right) \tag{A.8}$$

where $C > 0$ is a penalty parameter, and the negative log-likelihood is due to the fact the given data point $\mathbf{z}_i$ and weights $\mathbf{w}$ are assumed to follow the probability model,

$$\mathcal{P}\big(y = \pm 1 | \mathbf{z}, \mathbf{w}\big) = \frac{1}{1 + e^{-y \left\langle \mathbf{w} \cdot \mathbf{z}_i \right\rangle}}. \tag{A.9}$$

The logistic regression defined in Eq. (A.8) is solved iteratively, in particular using Trust Region Newton method (cf. Lin et al., 2008), which generates a sequence $\{\mathbf{w}^{(k)}\}_{k=1}^{\infty}$ converging to the optimal solution $\mathbf{w}^*$ of Eq. (A.8).

A.4 Non-Linear Preference

In the case that the preference set Υ defined by Eq. (A.4) is not linearly separable, a common way of coping with non-linearity is to apply the 'kernel-trick' to transform Υ onto a higher dimension. In which case, the dot product in Eq. (A.5) is replaced by a kernel function κ.

In terms of training data, the optimal $\mathbf{w}^*$ can be expressed as,

$$\mathbf{w}^* = \sum_{k=1}^{N'} a^* t_k \left(\boldsymbol{\varphi}(\mathbf{x}_k^{(1)}) - \boldsymbol{\varphi}(\mathbf{x}_k^{(2)})\right) \tag{A.10}$$

and the function $h(\cdot)$ from Eq. (A.7) may be reconstructed as follows,

$$\begin{aligned} h(\mathbf{x}) = \left\langle \mathbf{w}^* \cdot \boldsymbol{\varphi}(\mathbf{x}) \right\rangle &= \sum_{k=1}^{N'} a^* t_k \left(\left\langle \boldsymbol{\varphi}(\mathbf{x}_k^{(1)}) \cdot \boldsymbol{\varphi}(\mathbf{x}) \right\rangle - \left\langle \boldsymbol{\varphi}(\mathbf{x}_k^{(2)}) \cdot \boldsymbol{\varphi}(\mathbf{x}) \right\rangle \right) \\ &= \sum_{k=1}^{N'} a^* t_k \left(\kappa(\mathbf{x}_k^{(1)}, \mathbf{x}) - \kappa(\mathbf{x}_k^{(2)}, \mathbf{x}) \right) \end{aligned} \tag{A.11}$$

where $\kappa(\mathbf{x}, \mathbf{z}) = \left\langle \boldsymbol{\varphi}(\mathbf{x}) \cdot \boldsymbol{\varphi}(\mathbf{z}) \right\rangle$ is the chosen kernel and a_k^* are the Lagrangian multipliers for the constraints that can be determined by solving the dual quadratic programming problem,

$$\max_a \sum_{k=1}^{N'} a_k - \frac{1}{2} \sum_{i=1}^{N'} \sum_{j=1}^{N'} a_i a_j t_i t_j \left(K\left(\mathbf{x}_i^{(1)}, \mathbf{x}_i^{(2)}, \mathbf{x}_j^{(1)}, \mathbf{x}_j^{(2)} \right) + \frac{1}{C} \delta_{ij} \right) \tag{A.12}$$

subject to $\sum_{k=1}^{N'} a_k t_k = 0$ and $a_k \geq 0$ for all $k \in \{1, \ldots, N'\}$, and where,

$$\begin{aligned} K\left(\mathbf{x}_i^{(1)}, \mathbf{x}_i^{(2)}, \mathbf{x}_j^{(1)}, \mathbf{x}_j^{(2)} \right) &= \kappa\left(\mathbf{x}_i^{(1)}, \mathbf{x}_j^{(1)} \right) - \kappa\left(\mathbf{x}_i^{(1)}, \mathbf{x}_j^{(2)} \right) \\ &\quad - \kappa\left(\mathbf{x}_i^{(2)}, \mathbf{x}_j^{(1)} \right) + \kappa\left(\mathbf{x}_i^{(2)}, \mathbf{x}_j^{(2)} \right) \end{aligned} \tag{A.13}$$

and δ_{ij} is the Kronecker δ defined to be 1 iff $i = j$ and 0 otherwise.

Kernel functions

There are several choices for a kernel κ, e.g., *polynomial kernel*,

$$\kappa_{\text{poly}}(\mathbf{x}_i, \mathbf{x}_j) = \left(1 + \left\langle \mathbf{x}_i \cdot \mathbf{x}_j \right\rangle\right)^p \tag{A.14}$$

of order p, or the most commonly used kernel in the literature which implements a Gaussian radial basis function, the *rbf kernel*,

$$\kappa_{\text{rbf}}(\mathbf{x}_i, \mathbf{x}_j) = e^{-\gamma \|\mathbf{x}_i - \mathbf{x}_j\|^2} \tag{A.15}$$

for $\gamma > 0$.

A.5 Parameter setting and tuning

The regulation parameter C in Eqs. (A.6), (A.8) and (A.12), controls the balance between model complexity and training errors, and must be chosen appropriately. A high value for C gives greater emphasis on correctly distinguishing between different ranks, whereas a low C value results in maximising the margin between classes.

A.6 Scaling

It some cases it becomes necessary to scale the features $\boldsymbol{\varphi}$ from Section 2.5 first, especially if implementing a kernel method in Eq. (A.5). In the case of JSP, scaling makes the features less sensitive to varying problem instances. Moreover, for surrogate modelling (cf. Paper II), it is important to scale the features $\boldsymbol{\varphi}$ as the evolutionary search zooms in on a particular region of the search space.

A standard method of doing so is by scaling the preference set such that all points are in some range, typically $[-1, 1]$. That is, scaled $\tilde{\boldsymbol{\varphi}}$ is,

$$\tilde{\varphi}_i = 2(\varphi_i - \underline{\varphi}_i)/(\overline{\varphi}_i - \underline{\varphi}_i) - 1 \qquad \forall i \in \{1, \ldots, d\} \tag{A.16}$$

where $\underline{\varphi}_i$, $\overline{\varphi}_i$ are the maximum and minimum i-th component of all the feature variables in Φ, namely,

$$\underline{\varphi}_i = \min\{\varphi_i \mid \forall \boldsymbol{\varphi} \in \Phi\} \quad \text{and} \quad \overline{\varphi}_i = \max\{\varphi_i \mid \forall \boldsymbol{\varphi} \in \Phi\} \tag{A.17}$$

where $i \in \{1 \ldots d\}$.

A.7 Implementation

To use linear ordinal regression, then it's best to use LIBLINEAR: A Library for Large Linear Classification by Fan et al. (2008), which contains implementations in several programming languages. The preferred choice of the author was the R-package `LiblineaR` by Helleputte (2015). However, if more sophisticated kernel methods are sought after, then LIBSVM: A Library for Support Vector Machines by Chang and Lin (2011) is an obvious substitute.

APPENDIX A. ORDINAL REGRESSION

Part II

Papers

But it's no use going back to yesterday, because I was a different person then.

Alice

I

Supervised Learning Linear Priority Dispatch Rules for Job-Shop Scheduling

Helga Ingimundardóttir, Tómas Philip Rúnarsson

School of Engineering and Natural Sciences, University of Iceland, Iceland

Learning and Intelligent Optimization

I

This page is intentionally left blank.

Take care of the sense, and the sounds will take care of themselves.

The Duchess

II

Sampling Strategies in Ordinal Regression for Surrogate Assisted Evolutionary Optimization

Helga Ingimundardóttir, Tómas Philip Rúnarsson

School of Engineering and Natural Sciences, University of Iceland, Iceland

Intelligent Systems Design and Applications (ISDA), 2011 11th International Conference on

II

This page is intentionally left blank.

I wonder if I've been changed in the night? Let me think. Was I the same when I got up this morning? I almost think I can remember feeling a little different. But if I'm not the same, the next question is 'Who in the world am I?' Ah, that's the great puzzle!

Alice

III

Determining the Characteristic of Difficult Job Shop Scheduling Instances for a Heuristic Solution Method

Helga Ingimundardóttir, Tómas Philip Rúnarsson

School of Engineering and Natural Sciences, University of Iceland, Iceland

Learning and Intelligent Optimization

III

This page is intentionally left blank.

It would be so nice if something made sense for a change.

Alice

IV

Evolutionary Learning of Weighted Linear Composite Dispatching Rules for Scheduling

Helga Ingimundardóttir

School of Engineering and Natural Sciences, University of Iceland, Iceland

International Conference on Evolutionary Computation Theory and Applications (ECTA) – authorslist

IV.0. Find second publication for Paper IV – Selected papers ISI?

IV

This page is intentionally left blank.

Now, I give you fair warning, either you or your head must be off, and that in about half no time! Take your choice!

The Queen

V

Generating Training Data for Learning Linear Composite Dispatching Rules for Scheduling

Helga Ingimundardóttir, Tómas Philip Rúnarsson

School of Engineering and Natural Sciences, University of Iceland, Iceland

Learning and Intelligent Optimization – Nominated for best paper award

This page is intentionally left blank.

But then, shall I never get any older than I am now? That'll be a comfort, one way – never to be an old woman – but then – always to have lessons to learn!

Alice

VI

Supervised Learning Linear Composite Dispatch Rules for Scheduling

Helga Ingimundardóttir, Tómas Philip Rúnarsson

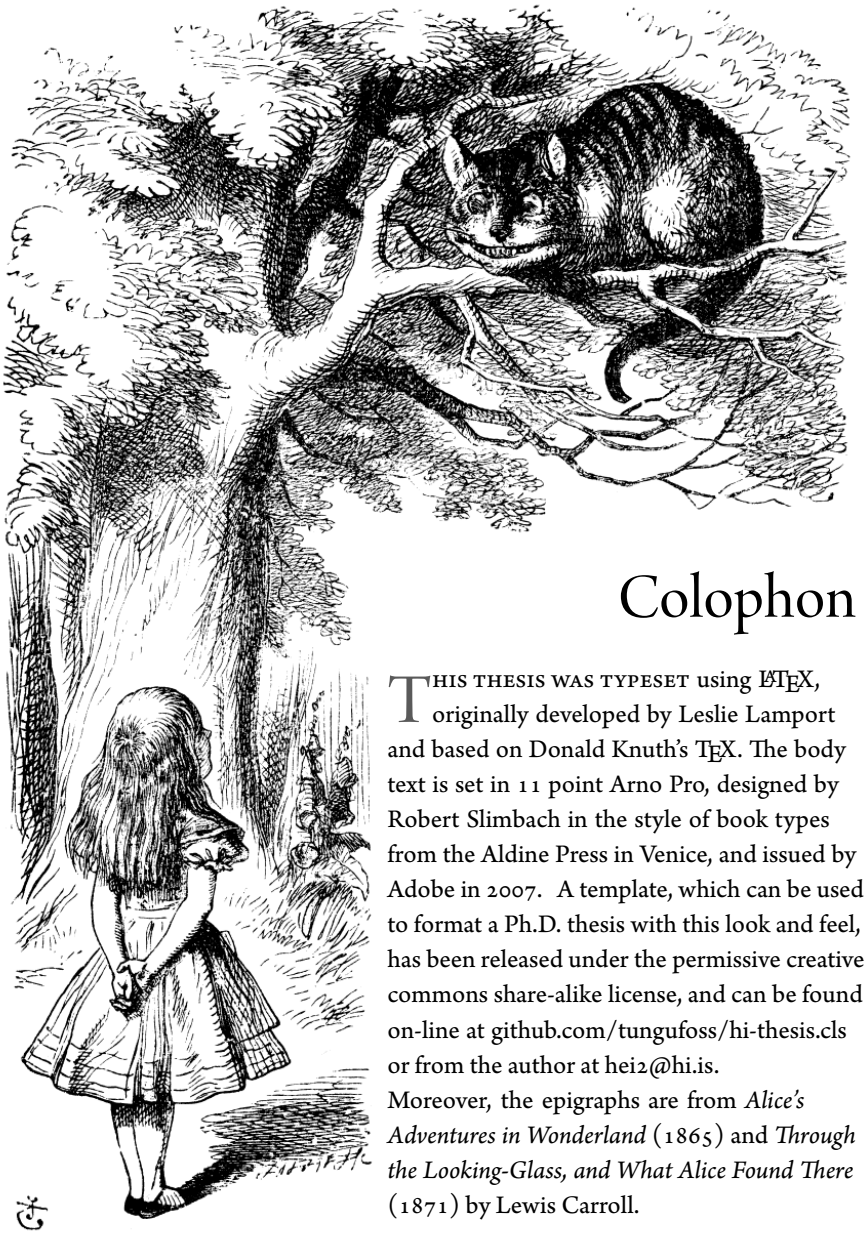
School of Engineering and Natural Sciences, University of Iceland, Iceland

NA – Accepted

VI.0. Paper VI not yet submitted

VI

This page is intentionally left blank.



Colophon

This thesis was typeset using LaTeX, originally developed by Leslie Lamport and based on Donald Knuth's TeX. The body text is set in 11 point Arno Pro, designed by Robert Slimbach in the style of book types from the Aldine Press in Venice, and issued by Adobe in 2007. A template, which can be used to format a Ph.D. thesis with this look and feel, has been released under the permissive creative commons share-alike license, and can be found on-line at github.com/tungufoss/hi-thesis.cls or from the author at hei2@hi.is.

Moreover, the epigraphs are from *Alice's Adventures in Wonderland* (1865) and *Through the Looking-Glass, and What Alice Found There* (1871) by Lewis Carroll.

If you don't know where you are going, any road will take you there.

Lewis Carroll

Alice's Adventures in Wonderland

What is the use of a book, without pictures or conversations? **Alice**

Oh my ears and whiskers, how late it's getting! **Rabbit**

Curiouser and curiouser! **Alice**

Paper III: ~~*I wonder if I've been changed in the night? Let me think. Was I the same when I got up this morning? I almost think I can remember feeling a little different. But if I'm not the same, the next question is 'Who in the world am I?' Ah, that's the great puzzle!*~~ **Alice**

Speak English! I don't know the meaning of half those long words, and I don't believe you do either! **Eaglet**

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

If everybody minded their own business, the world would go around a great deal faster than it does. **The Duchess**

Chapter 3: ~~*If it had grown up, it would have made a dreadfully ugly child; but it makes rather a handsome pig, I think.*~~ **Alice**

We're all mad here. **The Cat**

Why is a raven like a writing desk? **The Hatter**

Twinkle, twinkle, little bat! How I wonder what you're at. **The Hatter**

Off with her head! **The Queen**

Chapter 11: ~~*Tut, tut, child! Everything's got a moral, if only you can find it.*~~ **The Duchess**

Paper II: ~~*Take care of the sense, and the sounds will take care of themselves.*~~ **The Duchess**

We called him Tortoise because he taught us. **The Mock Turtle**

Reeling and Writhing, of course, to begin with, and then the different branches of arithmetic – Ambition, Distraction, Uglification, and Derision. **The Mock Turtle**

Well, I never heard it before, but it sounds uncommon nonsense. **The Mock Turtle**

Chapter 1: ~~*Begin at the beginning and go on till you come to the end: then stop.*~~ **The King**

Chapter 7: ~~*I don't believe there's an atom of meaning in it.*~~ **Alice**

Chapter 4: ~~*Sentence first – verdict afterwards.*~~ **The Queen**

You're nothing but a pack of cards! **Alice**

Paper VI: ~~*But then, shall I never get any older than I am now? That'll be a comfort, one way – never to be an old woman – but then – always to have lessons to learn!*~~ **Alice**

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

I think I should understand that better, if I had it written down: but I can't quite follow it as you say it. **Alice**

That's nothing to what I could say if I chose. **The Duchess**

Paper V: ~~*Now, I give you fair warning, either you or your head must be off, and that in about half no time! Take your choice!*~~ **The Queen**

Paper IV: ~~*It would be so nice if something made sense for a change.*~~ **Alice**

Chapter 2: ~~*Read the directions and directly you will be directed in the right direction.*~~ **Doorknob**

No wonder you're late. Why, this watch is exactly two days slow. **Mad Hatter**

Let me see: four times five is twelve, and four times six is thirteen, and four times seven is – oh dear! I shall never get to twenty at that rate! **Alice**

Chapter 8: ~~*It was much pleasanter at home, when one wasn't always growing larger and smaller, and being ordered about by mice and rabbits.*~~ **Alice**

Chapter 6: ~~*Well! I've often seen a cat without a grin; but a grin without a cat! It's the most curious thing I ever say in my life!*~~ **Alice**

Chapter 5: ~~*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**

Appendix A: ~~*What is the use of repeating all that stuff, if you don't explain it as you go on? It's by far the most confusing thing I ever heard!*~~ **The Mock Turtle**

Ah! Then yours wasn't a really good school. Now at ours they had at the end of the bill. French, music, and washing – extra. **The Mock Turtle**

Oh, how I wish I could shut up like a telescope! I think I could, if I only knew how to begin. **Alice**

Paper I: ~~*But it's no use going back to yesterday, because I was a different person then.*~~ **Alice**

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

Chapter 10: ~~*The adventures first... explanations take such a dreadful time.*~~ **The Gryphon**

Either the well was very deep, or she fell very slowly, for she had plenty of time as she went down to look about her and to wonder what was going to happen next. **Narrator**

Let me see: four times five is twelve, and four times six is thirteen, and four times seven is - oh dear! I shall never get to twenty at that rate! However, the Multiplication Table doesn't signify: let's try Geography. London is the capital of Paris, and Paris is the capital of Rome, and Rome - no, that's all wrong, I'm certain! I must have been changed for Mabel! **Alice**

I have proved by actual trial that a letter, that takes an hour to write, takes only about 3 minutes to read

Lewis Carroll

Through the Looking-Glass

Quotes from *Through the Looking-Glass, and What Alice Found There* (1871) by Lewis Carroll. *It's a poor sort of memory that only works backward.* **The Queen**

Now, here, you see, it takes all the running you can do, to keep in the same place. If you want to get somewhere else, you must run at least twice as fast as that! **The Queen**

Now, Kitty, let's consider who it was that dreamed it all. This is a serious question, my dear, and you should not go on licking your paw like that – as if Dinah hadn't washed you this morning! **Alice**

You see, Kitty, it must have been either me or the Red King. He was part of my dream, of course – but then I was part of his dream, too! Was it the Red King, Kitty? You were his wife, my dear, so you ought to know – oh, Kitty, do help to settle it! I'm sure your paw can wait!" But the provoking kitten only began on the other paw, and pretended it hadn't heard the question. **Alice**

Sometimes I've believed as many as six impossible things before breakfast. **The Queen**

Always speak the truth, think before you speak, and write it down afterwards. **The Queen**