TU WIEN Informatics

Instance Space Analysis for the Job Shop Scheduling Problem

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Software Engineering und Internet Computing

eingereicht von

Simon Strassl, BSc.
Matrikelnummer 01326936

an der Fakultät für Informatik
der Technischen Universität Wien
Betreuung: Priv.-Doz. Dr. Nysret Musliu

Wien, 14. Oktober 2020

Simon Strassl	Nysret Musliu

TU WIEN Informatics

Instance Space Analysis for the Job Shop Scheduling Problem

DIPLOMA THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieur

in

Software Engineering and Internet Computing

by

Simon Strassl, BSc.
Registration Number 01326936

to the Faculty of Informatics

at the TU Wien

Advisor: Priv.-Doz. Dr. Nysret Musliu

Vienna, 14th October, 2020

Simon Strassl	Nysret Musliu

Technische Universität Wien
A-1040 Wien ▪ Karlsplatz 13 ▪ Tel. +43-1-58801-0 ▪ www.tuwien.at

Erklärung zur Verfassung der Arbeit

Simon Strassl, BSc.

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

Wien, 14. Oktober 2020

Simon Strassl

Acknowledgements

First and foremost I would like to express my gratitude to my advisor Dr. Nysret Musliu for his continuous advice and feedback, without which this thesis would not have been possible. I would also like to thank my family, and in particular my parents, whose unconditional support I am eternally grateful for. And finally, I want to give a shout out to all my friends who have been with me along the way, among which Maximilian Moser and Carole Martin deserve a special mention for their assiduous proof-reading of this thesis.

Kurzfassung

Der stetige Anstieg an Größe und Komplexität von Produktionssystemen macht den Einsatz von automatischen Planungssystemen beinahe zu einer Notwendigkeit. Die Wahl des Algorithmuses ist hier ein wichtiger Faktor für die Qualität der Lösung, was wiederum einen direkten Einfluss auf die Effizienz des Systems hat. Es ist daher unerlässlich, dass diese Entscheidung eine informierte ist, die idealerweise von einem automatisierten System getroffen werden kann. Der Fokus dieser Arbeit liegt auf dem Job Shop Scheduling Problem, welches bekanntermaßen $\mathcal{NP}$-schwer ist und in der Literatur bereits vielfach behandelt wurde. Dennoch gibt es bisher relativ wenig Arbeit um ein besseres Verständnis der Landschaft der Instanzen zu schaffen, was sich in der unsystematisch zusammengestellten Sammlung an Testinstanzen niederschlägt. Da diese Instanzen häufig verwendet werden um Algorithmen zu evaluieren, kann ein Bias hier zu falschen Schlüssen und somit schlechteren Ergebnissen führen.

Diese Diplomarbeit enthält eine systematische Analyse des Instanzraums des Job Shop Scheduling Problems mit dem Ziel unser Verständnis des Problems zu verbessern und eine bessere Grundlage für weitere Arbeit zu schaffen. Um dies zu erreichen wurden die in der Literatur verwendeten Testinstanzen analysiert und um neu generierte Instanzen verschiedener Größen und mit Bearbeitungszeiten aus verschiedenen Wahrscheinlichkeitsverteilungen erweitert. Auf Basis dieser Instanzen wurden verschiedene Algorithmen evaluiert, um ihre Verhaltensmuster zu analysieren und Unterschiede zu den existierend Testinstanzen aufzuzeigen. Aufgrund dieser Analyse wurde festgestellt, dass die existierenden Instanzen einen wesentlich kleineren Bereich abdecken als die generierten und zu abweichenden Schlussfolgerungen hinsichtlich der Qualität der Algorithmen führten.

Auf der erweiterte Menge von Instanzen wurden zwei exakte Methoden (CP Optimizer und OR-Tools) und eine Metaheuristik (Tabu-Suche) als die besten Algorithmen ermittelt, wobei jeder in einem anderen Teilbereich hervorstach. Die Metaheuristiken erzielten schlechtere Ergebnisse für Instanzen basierend auf einer konstanten Verteilung oder einer negativen Binomialverteilung, was darauf hinweist, dass die verwendete Nachbarschaftsfunktion für diese Instanzen schlecht geeignet ist. Umgekehrt zeigten die exakten Methoden eine Verschlechterung der Lösungsqualität bei Instanzen mit uniform oder binomial verteilten Bearbeitungszeiten. Auf Basis dieser Eigenschaften wurden Machine Learning Modelle erstellt, welche den besten Algorithmus für eine Instanz vorherzusagen. Das auf dem besten Modell, einem Random Forest, basierende Lösungsverfahren erreichte für 90% der Instanzen die beste Lösung, während der beste individuelle Algorithmus diese nur für 64% erreichte.

Abstract

The continuous increase in size and complexity of production systems makes the use of automated scheduling systems almost a necessity. The choice of algorithm is of course a major factor in the quality of the schedule, which in turn can have a very real impact on the efficiency of the system. It is therefore imperative that this choice is an informed one that can, ideally, be made by an automated system instead of relying on human expertise. The focus of this thesis lies on the job shop scheduling problem, a well-known, $\mathcal{NP}$-hard problem that has been extensively studied in the literature, for which, despite its age and popularity, there has been relatively little work done towards understanding the landscape of instances, which is reflected in the rather haphazard set of commonly used benchmark instances. Since these instances are commonly used to evaluate and compare the performance of algorithms, bias in the benchmark instances may lead to incorrect conclusions about an algorithm's strengths and weaknesses and thus result in a worse solution.

This thesis provides a systematic analysis of the instance space of the job shop scheduling problem with the goal of furthering our understanding of the problem and creating an improved foundation for further work. For this purpose, the benchmark instances commonly used in the literature were analyzed and extended by a set of newly generated instances of various sizes with processing times drawn from different probability distributions. A number of different algorithms were evaluated on the extended instance set to analyze their performance patterns and highlight the differences to the current set of benchmark instances. It was found that the existing instances cover a significantly smaller area than the newly generated ones and did in fact result in different conclusions regarding the algorithms' performances.

On the extended instance set two exact methods (CP Optimizer and OR-Tools) and one metaheuristic (tabu search) were determined to be the best algorithms, however each excelled on a distinct subset of instances. The metaheuristics in particular showed significantly worse performance on instances with processing times drawn from a constant or negative binomial distribution, indicating the neighborhood is ill-suited for this kind of instances. Conversely the exact methods displayed inferior performance on instances with uniformly or binomially distributed processing times. The difference in algorithm performance was utilized to train machine learning models to predict the best algorithm for a given instance. The solver based on the best model, a Random Forest, was able to obtain the best solution for 90% of the instances, whereas the best individual algorithm only obtained the best solution for 64%.

Contents

Kurzfassung **ix**

Abstract **xi**

Contents **xiii**

1 Introduction **1**
1.1 Aims of the Thesis . . . 2
1.2 Contributions . . . 2
1.3 Structure of the Thesis . . . 3

2 State of the Art **5**
2.1 Problem Definition . . . 5
2.2 State of the Art . . . 8

3 Instance Space Analysis for the JSSP **11**
3.1 Statistical Methodology . . . 11
3.2 Problem Instances . . . 12
3.3 Solution Approaches . . . 15
3.4 Performance Data Generation . . . 27
3.5 Features . . . 32
3.6 Performance Data Analysis . . . 36
3.7 Algorithm Selection . . . 37

4 Experimental Evaluation **39**
4.1 Overall Performance . . . 39
4.2 Shape of the Instance Space . . . 43
4.3 Features . . . 45
4.4 Performance Across the Instance Space . . . 50
4.5 Algorithm Selection . . . 57

5 Conclusion **63**

List of Figures **65**

List of Tables **67**

Appendix A **69**

Bibliography **73**

CHAPTER 1

Introduction

As production systems all around the world increase in size, complexity and level of automation, the need for automated scheduling systems also increases. Since the vast majority of underlying problems are $\mathcal{NP}$-hard, one cannot expect to obtain an optimal solution within a reasonable amount of time for larger problem instances. However the quality of the schedule can have a significant impact on the productivity of the overall system, which in turn can lead to massive cost savings. It follows that the choice of scheduling algorithm is a major factor in this process and while there usually exist a wide variety of approaches for any given problem, they vary wildly in the quality of their solutions. It is therefore imperative to choose the right algorithm for any given subset of problem instance and that this choice is an informed one backed by data. Ideally this choice would not have to be made by humans, but rather by a specialized algorithm that is able to determine the ideal approach for any given instance.

However the quality of this choice depends significantly on having correct and representative data regarding the performance of various algorithms on different kinds of instances, i.e. the benchmark instances should paint an accurate picture of the instance space. Historically the data sets used to evaluate the performance of algorithms for a given problem have largely been chosen without an explicit process to ensure they properly cover the space of all possible instances. While it is understandable from a pragmatic standpoint why this path was taken, one can also easily see how this might unfairly bias the results in favor of or against some algorithms, which in turn might lead to incorrect conclusions about the performance of the evaluated algorithm.

In recent years, however, there has been an increased uptake in work towards alleviating this problem, such as the instance space analysis framework proposed by Smith-Miles et al. [SM+14]. The process requires multiple steps: First, one must understand how well the existing benchmark instances cover the instance space. Then the gaps in the instance space can be filled by generating new benchmark instances to ensure proper coverage. Based on these benchmark instances, the performance of various algorithms

can be evaluated and their strengths and weaknesses can be determined, which provides the information necessary to make an informed decision regarding the algorithm to use.

This thesis concerns itself particularly with one of the oldest and best-known scheduling problems – the classical job shop scheduling problem (JSSP) [FP09]. Despite it being a widely known problem, there has been relatively little work done towards a more representative set of test data – an issue that has recently also been raised by Weber et al. [Web+19]. The impact of this issue is magnified by the relatively broad spectrum of solution approaches for the JSSP, including constraint programming [Vil+15], tabu search [Zha+08; NS05; Tai94], simulate annealing [LA87], genetic algorithms [YN92], and numerous others. However, no clear best has yet emerged, indicating that the choice of algorithm can have a significant impact on the quality of the solution.

1.1 Aims of the Thesis

The primary aim of this thesis is to further our understanding of the job shop scheduling instance landscape and to build an improved foundation on which to evaluate the performance of existing and future solution approaches.

In particular, this is achieved by:

- Analyzing the existing benchmark instances found in the literature and generating novel benchmark instances to fill any gaps.
- Defining a set of features used to characterize the instances.
- Gathering performance data from a diverse set of algorithms on which, combined with the feature data, the instance space analysis shall be based.
- Creating and evaluating machine learning models with regard to their suitability for automatically selecting the optimal algorithm for a given instance, with the goal of improving upon the performance of any individual algorithm.

1.2 Contributions

This thesis offers a broad exploration of the job shop scheduling problem, with its main contributions being:

- A novel set of benchmark instances covering a wider range of variations than the ones commonly found in the literature, as well as evidence of their necessity.
- The definition of probing and graph features that have so far not been used as well as a systematic analysis of various instance features, their distribution, their predictive power and importance, but also their similarities.

- The identification of areas of the instance space in which one algorithm outperforms the others and an analysis of the aforementioned areas, as well as an analysis of the algorithms' performance patterns that provides insights into their respective strengths and weaknesses.
- The creation of algorithm-portfolio-based solver by means of automated algorithm selection using various machine learning models, each of which is able to consistently obtain better results than any individual algorithm.

1.3 Structure of the Thesis

The core of this thesis is split into three major chapters as follows:

First up, chapter 2 provides an introduction to the instance space analysis methodology as well as a definition of the job shop scheduling problem. Furthermore, it contains a short survey depicting the state of the art related to instance space analysis for the job shop scheduling problem.

Chapter 3 offers an overview of how the instance space analysis methodology is applied in this thesis. This includes the algorithms used, the instances on which the analysis was performed, the features used to describe the instances, and the experiment setup. A discussion regarding different performance measurements that could be used to evaluate an algorithm's performance and their respective advantages and drawbacks is also included.

Finally chapter 4 presents the experimental results as well as their analysis. The analysis starts with a high-level view of the various algorithms' performance, followed by an in-depth investigation into the shape of the instance space, the feature distribution and the performance characteristics. Finally the data obtained from the instance space analysis is used to build machine learning models with the goal of predicting the best algorithm for a given instance.

CHAPTER 2

State of the Art

2.1 Problem Definition

While the family of problems related to job shop scheduling is vast, this thesis is limited to the basic job shop scheduling problem with the makespan objective function, i.e. $J||C_{max}$ in $\alpha|\beta|\gamma$ notation [Gra+79]. There exist various slightly different definitions (e.g. some permitting the repetition of machines [FP09]), however the following, based on [AC91], is the one most commonly used in the literature.

Given a finite set J of jobs and a finite set M of machines, let $n = |J|$ be the number of jobs and $m = |M|$ be the number of machines. The processing order of each job j is given by a permutation $(o_{j,1}, \ldots, o_{j,m})$ of M. Additionally each job j and machine k is associated with a non-negative, integer-valued processing time $p_{j,k}$. The steps a job has to complete on different machines shall be referred to as operations and $o_{j,i}$ shall be read as 'the machine of operation i of job j', while $p_{j,o_{j,i}}$ shall be read as 'the processing time of operation i of job j'. Furthermore the notation $n{\times}m$ is used to describe an instance with n jobs and m machines.

A schedule is then given by assigning each operation a start time $s_{j,k}$ for all $j \in J$, $k \in M$. In order for a schedule to be feasible, it has to satisfy the following conditions:

- All start times must be positive.
  $s_{j,k} \geq 0$ for all $j \in J$, $k \in M$.
- The operations of a job must be processed sequentially.
  $s_{j,o_{j,i}} + p_{j,o_{j,i}} \leq s_{j,o_{j,i+1}}$ for all $j \in J$, $1 \leq i < |M|$.
- There is no overlap between operations processed on the same machine.
  $s_{j,k} \geq s_{i,k} + p_{i,k} \vee s_{i,k} \geq s_{j,k} + p_{j,k}$ for all $i, j \in J$, $k \in M$, $i \neq j$.

The makespan C_{max}, that is the time of completion of the last job is given as $C_{max} = \max(\{s_{j,k} + p_{j,k} \mid j \in J, k \in M\})$. The goal then is to find a feasible schedule that minimizes C_{max}, the makespan of which shall be referred to as C^*_{max}.

A schedule is called *semi-active* if no operation can be rescheduled to finish earlier without changing the order of operations per machine or violating the constraints. Furthermore a schedule is referred to as *active* if it is not possible to reschedule an operation, even permitting changes in the order of operations per machine, so that some operation completes earlier. Clearly every active schedule must also be semi-active and there exists an equivalent active schedule for every feasible schedule [GT60]. It also follows that there must be at least one active schedule that is optimal. Furthermore, the makespan of all semi-active schedules on a given instance must be bounded by the makespan of a trivial sequential schedule $C^{seq}_{max} = \sum_{j,k} p_{j,k}$. From this point on, unless explicitly specified otherwise, only semi-active schedules will be considered. While there of course exist infinitely many schedules that can be arbitrarily worse (by introducing superfluous idle times between operations), they can trivially be improved and are therefore of no real relevance.

A simple example definition of a problem instance with 3 jobs and 2 machines, as well as a valid schedule for the given instance, can be seen in listing 2.1. Additionally the schedule from listing 2.1 is visualized as a Gantt chart in figure 2.1.

Another way of modeling a JSSP instance is the disjunctive graph $G = (V, C \cup D)$, where V is the set of vertices with a weight w_v assigned to each vertex v, C the set of conjunctive arcs, and D the set of disjunctive arcs. V corresponds to the set of all operations as well as a start node s and an end node t, i.e. $V = \{v_{j,k} \mid j \in J, k \in M\} \cup \{s, t\}$. The weight of a vertex is equivalent to the corresponding operation's processing time $w_{v_{j,k}} = p_{j,k}$,

$$
\begin{aligned}
J &= \{1, 2, 3\} \\
M &= \{0, 1\} \\
\begin{pmatrix} o_{1,1} & o_{1,2} \\ o_{2,1} & o_{2,2} \\ o_{3,1} & o_{3,2} \end{pmatrix} &= \begin{pmatrix} 0 & 1 \\ 1 & 0 \\ 1 & 0 \end{pmatrix} \\
\begin{pmatrix} p_{1,0} & p_{1,1} \\ p_{2,0} & p_{2,1} \\ p_{3,0} & p_{3,1} \end{pmatrix} &= \begin{pmatrix} 2 & 3 \\ 3 & 2 \\ 1 & 4 \end{pmatrix} \\
\begin{pmatrix} s_{1,0} & s_{1,1} \\ s_{2,0} & s_{2,1} \\ s_{3,0} & s_{3,1} \end{pmatrix} &= \begin{pmatrix} 0 & 6 \\ 2 & 0 \\ 6 & 2 \end{pmatrix}
\end{aligned}
$$

Listing 2.1: Sample JSSP instance and schedule

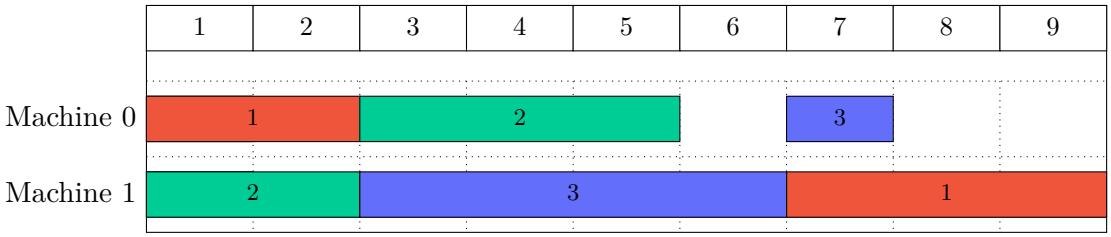


Figure 2.1: Gantt chart for the schedule given in listing 2.1

$w_s = w_t = 0$. C contains an arc for every precedence relation in a job and additionally the arcs connecting s to the first operation and t to the last operation of each job. Thus $C = \{(v_{j,o_{j,k}}, v_{j,o_{j,k+1}}) \,|\, j \in J, 1 \leq k < |M|\} \cup \{(s, v_{j,o_{j,1}}) \,|\, j \in J\} \cup \{(v_{j,o_{j,|M|}}, t) \,|\, j \in J\}$. Finally D contains an undirected edge, expressed as two inverse arcs, between each pair of operations on the same machine, i.e. $D = \{(v_{i,k}, v_{j,k}) | i, j \in J, k \in M, i \neq j\}$. An orientation D' of the disjunctive edges is given by choosing one of the two possible arcs for all pairs of nodes on the same machine. Formally let $D' \subseteq D$ so that for all $(v_1, v_2) \in D$ it holds that $(v_1, v_2) \in D' \leftrightarrow (v_2, v_1) \notin D'$. An example of such an oriented disjunctive graph corresponding to the schedule from listing 2.1 can be seen in figure 2.2. A feasible schedule can be derived from the disjunctive graph by finding an orientation of all disjunctive edges so that the resulting graph is acyclic.[1] Conversely, every feasible schedule defines an orientation that produces an acyclic disjunctive graph. The makespan is then given by the length of the critical path(s), that is the longest path(s) by node weight from s to t, in the oriented conflict graph.

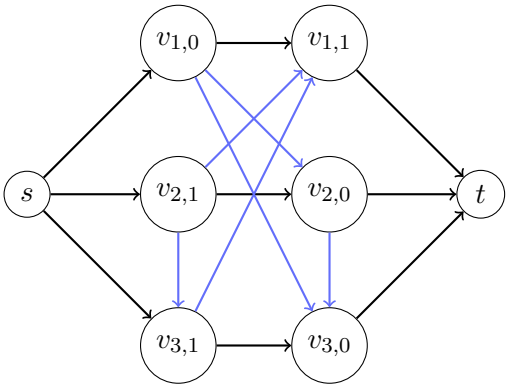


Figure 2.2: Oriented disjunctive graph for the instance and schedule given in listing 2.1 with C in black, D' in blue

[1] Strictly speaking an orientation allows for an infinite set of schedules, however one can always derive a semi-active schedule by topologically sorting the oriented graph and scheduling each operation in this order at the earliest time possible.

2.2 State of the Art

2.2.1 Job Shop Scheduling

The JSSP is one of the oldest, best-known $\mathcal{NP}$-complete problems [Len+77]. It is also known to be among the more difficult problems to solve, having earned the adjectives "extremely hard" [Gra+79], "notoriously difficult" [AC91; JM98] and "notoriously intractable" [Zha+08] – an assessment which rings all too true, given that popular benchmark instances as small as 20×15 do not have a proven optimal solution yet [vHo18]. Due to its age and relative popularity, it has been extensively studied in the literature and there exists a wealth of resources for it. In particular [ÇB15; Vae+96; JM98] provide an overview of the different approaches that have been used. Van Hoorn [vHo18] gives a reasonably up-to-date compilation of relevant benchmark instances for the JSSP, known lower and upper bounds for these instances as well as an overview of state of the art algorithms for the JSSP. In particular constraint programming [Bec+11; Vil+15] as well as metaheuristics [NS05; Zha+08] have both achieved very good solutions.

However, as has recently been recognized by Weber et al. [Web+19], the state of test data for the JSSP is far from optimal, with the set of commonly used benchmark instances being an ad-hoc aggregation of various authors' generated test data. This in turn makes a systematic analysis of the JSSP and the various algorithms' performance characteristics difficult if not downright impossible.

2.2.2 Instance Space Analysis

The aim of the instance space analysis is to alleviate the aforementioned problem and gain a better insight into the various characteristics of the problem and the algorithms' performance patterns. Most of the work has been done by Smith-Miles et al. [SM+09; SM+09; SML12; SM+14] who propose a methodological framework, the high-level process of which is visualized in figure 2.3. It is based on the well-known algorithm selection framework introduced by Rice [Ric76], which has been extended in the following two major ways.

Firstly, by introducing an explicit differentiation between the entire problem space $\mathcal{P}$ and the subset of instances $\mathcal{I}$ for which computational results exist. This distinction is only natural since it is usually not possible to cover the set of all possible instances – particularly since it is not finite for most cases. However, if one obtains a set of instances $\mathcal{I}$ that is representative of $\mathcal{P}$ as a whole, this set can still be used according to Rice's algorithm selection framework. Based on the results on $\mathcal{I}$ in the performance space $\mathcal{Y}$, one can then estimate the algorithm's performance on the entire space $\mathcal{P}$.

Secondly, instead of solely relying on a high dimensional feature space $\mathcal{F}$, the features are projected into $\mathbb{R}^2$ for ease of visualization. This projection has the practical benefit of being easier to visualize and thus analyze than high-dimensional spaces.

With regard to the JSSP, there has been relatively little work done in this direction. There has been some recent work on predicting the optimal makespan of a given JSSP instance

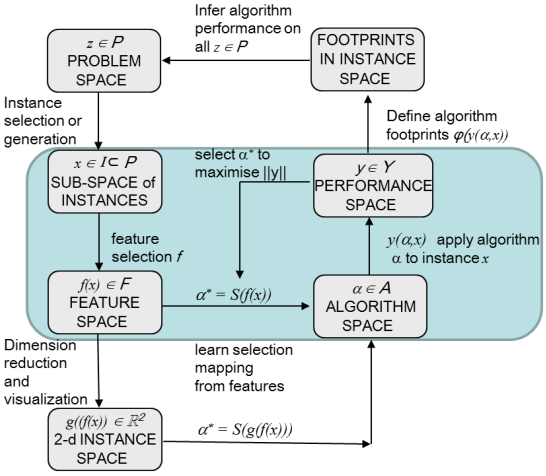


Figure 2.3: Instance space analysis framework from [SM+14]

by Mirshekarian and Šormaz [MŠ16], who introduce a novel performance measure as well as a number of features for JSSP instances, both of which shall be incorporated and evaluated in this thesis. Streeter and Smith [SS06] have also shown that an instance's job to machine ratio has a significant impact on the problem difficulty while Watson et al. [Wat+03; Wat+06] have investigated how different features affect tabu search performance. Corne and Reynolds [CR10] have applied instance space analysis to the single machine job shop scheduling problem and Ingimundardottir and Runarsson [IR12] have performed a basic analysis of the JSSP instance space based on a single dispatching rule heuristic. However there has so far been no systematic, in-depth exploration of the JSSP instance space.

Likewise, while the basic process for algorithm selection has been outlined by Rice [Ric76] and algorithm selection has been applied to a wide variety of problems [Kot16], the results for the JSSP are rather sparse. There has been work on very simple, low knowledge algorithm selection for the JSSP [BF04] as well as on the use of hyperheuristics [HS16; SH14], however there appears to have been no in-depth research on applying algorithm selection to the JSSP.

CHAPTER 3

Instance Space Analysis for the JSSP

This chapter aims to outline the core framework on which this thesis is based and thereby set the stage for the experimental evaluation in chapter 4. Section 3.1 defines the basic statistical methodology and terminology used in this thesis. In section 3.2 the process of collecting existing and generating novel benchmark instances is outlined. Section 3.3 lists the solution approaches used to generate the performance data as described in section 3.4. Along with the features defined in section 3.5, the performance data is analyzed according to the methodology given in section 3.6 in order to obtain a better understanding of the instance space. Finally the methodology used for the algorithm selection is given in section 3.7.

3.1 Statistical Methodology

Unless otherwise specified, statistical tests are performed using the Wilcoxon signed-rank test. Zero-differences are handled by including them but dropping their ranks according to Pratt and Gibbons [PG81]. Furthermore, the level of significance is set at 0.01, i.e. a result is considered statistically significant if $p \leq 0.01$. Where applicable, a visualization of p-values is given in form of a 2D heatmap with the actual numerical p-values included in the cells. For the value $p_{i,j}$ of any cell in such a heatmap one reads: $p_{i,j}$ is the probability of the observed (or more extreme) differences between the values of a_i and b_j, assuming the median difference is ≥ 0 (when testing whether a_i is less than b_i), ≤ 0 (when testing whether a_i is greater than b_i) or 0 (when testing whether a_i is different from b_i).

With regard to correlation, the Pearson correlation coefficient ρ is used unless otherwise specified. It ranges from -1 (a perfect negative linear correlation) over 0 (no linear correlation) up to $+1$ (a perfect positive linear correlation).

3.2 Problem Instances

In order to properly analyze the instance space, a sufficiently large set of benchmark instances that are representative of the instance space as a whole is required to obtain an accurate picture. This is achieved by first collecting the benchmark instances that have commonly been used in the literature and, based on these instances, generating new instances to expand upon the existing ones and fill in potential gaps. In practice, this is an iterative process requiring constant reevaluation and addition/removal of the generated instances, as more gaps in the instance space are found.

3.2.1 Existing Instances

The existing benchmark instances used are taken from the set of instances collected by Van Hoorn [vHo18], which contains all benchmark instances that are commonly used in the literature (242 in total). The instance sizes range from as low as 6×6 (`ft06`) up to 100×20 (`ta71-80`). The different subsets of instances, their sources, the number of instances included as well as the number of jobs and machines used are listed in table 3.1. The distribution of the instances by the number of jobs and machines can be seen in figure 3.1a.

Abbr.	Source	#Inst.	n	m
`ft`	Fisher and Thompson [FT63]	3	6,10,20	5,6,10
`la`	Lawrence [Law84]	40	10,15,20,30	5,10,15
`abz`	Adams et al. [Ada+88]	5	10,20	10,15
`orb`	Applegate and Cook [AC91]	10	10	10
`swv`	Storer et al. [Sto+92]	20	20,50	10,15
`yn`	Yamada and Nakano [YN92]	4	20	20
`ta`	Taillard [Tai93]	80	15,20,30,50,100	15,20
`dmu`	Demirkol et al. [Dem+98]	80	20,30,40,50	15,20

Table 3.1: Existing sets of benchmark instances

3.2.2 Generated Instances

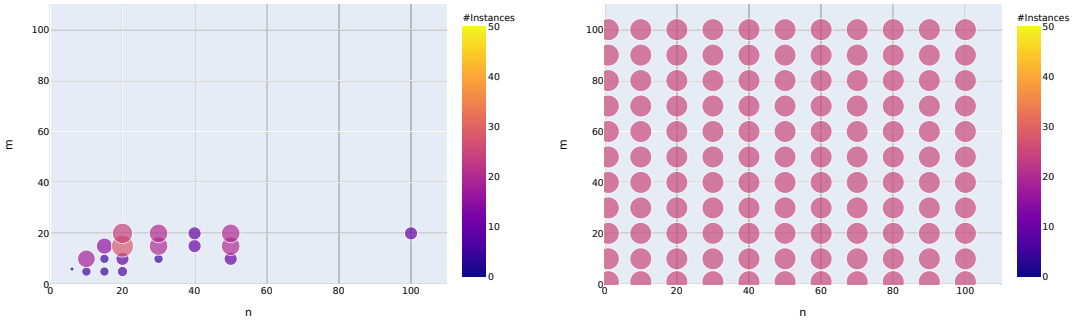
As can be seen in figure 3.1a, even if one only considers the number of jobs and machines, the existing benchmark instances cover only a small portion of all possible configurations. This does not necessarily imply a problem with the benchmark instances – they may very well still be a representative sample – but it does certainly suggest that there are not enough instances to draw conclusions about the shape and properties of the instance space. To remedy this problem, additional instances are generated according to the algorithm given in listing 3.1. The complete set of all generated instances, as well as the source code for the generator, is made available on GitHub and mirrored to Zenodo. [1]

The parameters used are:

- Seed: 42
- Jobs n: $[1, 5, 10, 15, 20, 30, 40, 50, 60, 70, 80, 90, 100]$
- Machines m: $[1, 5, 10, 15, 20, 30, 40, 50, 60, 70, 80, 90, 100]$
- Probability distributions:
  - The constant distribution with value 1. The set of instances derived from this distribution are referred to as `gen-const`. Note that these instances actually correspond to a variant of the JSSP with unit processing times $J|p_{j,k} = 1|C_{max}$, which is nonetheless $\mathcal{NP}$-hard [LRK79].
  - The uniform discrete distribution with values in $[1, 99]$. The set of instances derived from this distribution are referred to as `gen-uniform-99`.
  - The uniform discrete distribution with values in $[1, 200]$. The set of instances derived from this distribution are referred to as `gen-uniform-200`.
  - The binomial distribution with $n = 98$ and $p = 0.5$, shifted up by 1. The set of instances derived from this distribution are referred to as `gen-binom`.
  - The negative binomial distribution with $r = 1$ and $p = 0.5$, shifted up by 1. The set of instances derived from this distribution are referred to as `gen-nbinom`.
- Instances per configuration: 5

In total this results in a set of 4225 instances, however, to ensure a uniform distribution of n and m the instances with $n \in \{5, 15\}$ or $m \in \{5, 15\}$ are removed, leaving 3025 instances on which to perform the analysis. Even though the instances are not used in this analysis, they are included in the instance set due to their usefulness for implementing or analyzing algorithms. Among these 3025 instances are 475 instances with $n = 1$ or $m = 1$, which shall be referred to as *trivial* instances since they only permit trivial sequential

[1] https://doi.org/10.5281/zenodo.4081658



(a) Existing instances (`lit`) (b) Generated instances (`gen`)

Figure 3.1: Number of instances by jobs and machines

solutions. All algorithms are thus expected to be able to solve these instances optimally, but they are included so as to provide a more complete picture of the instance space.

Subsequently the set of generated instances shall be referred to as the `gen` instance set, the set of existing instances from the literature as the `lit` instance set and the union of both as the `all` instance set. The subsets of instances induced by the probability distribution in `gen` are named according to their source probability distribution, with the union of `gen-uniform-99` and `gen-uniform-200` being referred to simply as `gen-uniform`. Similarly the subsets of instances from `lit` are named after their source. The instances from `gen` are numbered consecutively with `gen0001-0845` being from `gen-const`, `gen0846-1690` from `gen-uniform-99`, `gen1691-2535` from `gen-uniform-200`, `gen2536-3380` from `gen-binom` and `gen3381-4225` from `gen-nbinom`.

```
def generate(seed, jobs, machines,
             probability_distributions, instances_per_configuration):
  instances = []
  for d in probability_distributions:
    for j in jobs:
      for m in machines:
        for i in range(0, instances_per_configuration):
          inst_seed = seed + len(instances)
          instance = gen_inst(seed=inst_seed, dist=d, jobs=j, machines=m)
          instances.append(instance)
  return instances

def gen_inst(seed, dist, jobs, machines):
  rand = Random(seed=seed)
  order_rand = Random(seed=rand.randint(0, 9999))
  duration_rand = Random(seed=rand.randint(0, 99999))

  instance = []
  for j in range(0, jobs):
    ops = []
    for m in range(0, machines):
      ops.append(m, dist.sample(duration_rand))
    ops.shuffle(random=order_rand)
    instance.append(ops)
  return instance
```

Listing 3.1: Instance generator (Python-like pseudocode)

3.3 Solution Approaches

The algorithms used can roughly be divided into three major categories: Exact methods, heuristic methods and metaheuristic methods. Each of these categories contains a number of exemplary algorithms that are implemented for the purpose of this thesis. Unless explicitly noted otherwise, the implementation of these algorithms is kept as basic as possible and no particular consideration is given to optimizing any particular algorithm.

3.3.1 Exact Methods

The exact methods used are all based on modeling the problem as a constrained-optimization problem, that is a constraint-satisfaction problem with an objective function. A solver is then applied to the model with the goal of finding a solution that satisfies the constraints and minimizes the objective function. One of the main advantages of exact methods is their ability to find a provably optimal solution, i.e. given enough time the solver will find the optimal solution and prove that no better solution exists. It is worth noting, however, that 'given enough time' may very well turn out to be a lot more time than one is willing to invest, in particular for larger instances. Besides the ability to prove optimality, they also benefit from a relative simplicity in implementation and ease of extensibility since additional constraints can simply be added to the model.

CP Optimizer [IBM20b] (`cpo`)

CP Optimizer, a part of the IBM ILOG CPLEX Optimization Studio, is a constrained-optimization solver with a deliberate focus on scheduling problems, thereby making it uniquely suited for this kind of problem. It has been successfully used by Vilím et al. [Vil+15] to find optimal solutions and lower bounds for previously unsolved JSSP benchmark instances.

Version used: 12.10.0

Model used: Listing 3.2

OR-Tools [Goo20] (`ort`)

OR-Tools is an open-source optimization software suite by Google, which has won the annual MiniZinc challenge [Stu+14] on several occasions.

Version used: 7.7.7810

Model used: Listing 3.3

Chuffed [chu_chuffed_2014] (`chu`)

Chuffed is a constraint programming solver based on lazy clause generation – a combination of SAT solving and finite domain propagation. The model was written in MiniZinc, compiled to FlatZinc and ultimately solved using Chuffed.

Version used: MiniZinc [Net+07] 2.4.2, Chuffed 0.10.4

Model used: Listing 3.4

CPLEX [IBM20a] (`cpl`)

CPLEX Optimizer too is part of the IBM ILOG CPLEX Optimization Studio. While CP Optimizer is deliberately targeted at optimization and scheduling problems, CPLEX is a general-purpose solver for linear, mixed-integer and quadratic programming. As for Chuffed, the model was written in MiniZinc, compiled to FlatZinc and finally solved using CPLEX.

Version used: MiniZinc [Net+07] 2.4.2, CPLEX 12.10.0

Model used: Listing 3.4

```python
def solve(instance):
  mdl = CpoModel()
  machines = [[instance[j][o].machine
               for o in range(0, instance.m)] for j in range(0, instance.n)]
  duration = [[instance[j][o].duration
               for o in range(0, instance.m)] for j in range(0, instance.n)]
  job_operations = [
    [
      mdl.interval_var(size=duration[j][o], name=f'{j}-{o}')
      for o in range(0, instance.m)
    ]
    for j in range(0, instance.n)
  ]

  for j in range(0, instance.n):
    for o in range(1, instance.m):
      mdl.add(mdl.end_before_start(
              job_operations[j][o - 1], job_operations[j][o]))

  machine_operations = [[] for m in range(0, instance.m)]
  for j in range(0, instance.n):
    for o in range(0, instance.m):
      machine_operations[machines[j][o]].append(job_operations[j][o])
  for machine_ops in machine_operations:
    if len(machine_ops) > 0:
      mdl.add(mdl.no_overlap(machine_ops))
  cmax = mdl.max([
    mdl.end_of(job_operations[j][instance.m - 1])
    for j in range(0, instance.n)
  ])
  mdl.add(mdl.minimize(cmax))

  return mdl.solve()
```

Listing 3.2: CP Optimizer model (Python) based on [IBM20b]

```python
def solve(instance):
  model = CpModel()
  all_machines = list(range(0, instance.m))
  horizon = sum(op.duration for job in instance for op in job)
  op_type = collections.namedtuple('op_type', ['start', 'end', 'interval'])

  all_ops = {}
  machine_to_intervals = collections.defaultdict(list)
  for j, job in enumerate(instance):
    for o, op in enumerate(job):
      suffix = f'-{j}-{o}'
      start_var = model.NewIntVar(0, horizon, 'start' + suffix)
      end_var = model.NewIntVar(0, horizon, 'end' + suffix)
      interval_var = model.NewIntervalVar(start_var, op.duration, end_var,
                                          'interval' + suffix)
      all_ops[j, o] = op_type(
          start=start_var, end=end_var, interval=interval_var)
      machine_to_intervals[op.machine].append(interval_var)

  for machine in all_machines:
    model.AddNoOverlap(machine_to_intervals[machine])

  for j, job in enumerate(instance):
    for o in range(0, len(job)-1):
      model.Add(all_ops[j, o+1].start >= all_ops[j, o].end)

  obj_var = model.NewIntVar(0, horizon, 'makespan')
  model.AddMaxEquality(obj_var, [
    all_ops[j, len(job)-1].end
    for j, job in enumerate(instance)
  ])
  model.Minimize(obj_var)

  status = CpSolver().Solve(model)
```

Listing 3.3: OR-Tools model (Python) based on [Goo20]

```
1  include "globals.mzn";
2  int: n;
3  int: m;
4  int: span;
5  array [1..n, 1..2*m] of int: job;
6  array [1..n, 1..m] of var 0..span: t;
7  array [1..n] of int: one = [ 1 | i in 1..n];
8  var 0..span: objective;
9
10 constraint forall(i in 1..n, j in 1..m-1) ( t[i,j] + job[i, 2*j] <= t[i, j+1] );
11
12 constraint
13         forall(k in 1..m) (
14            let {
15               array[1..n] of int: d =
16             [ job[i, 2*j] | i in 1..n, j in 1..m where job[i,2*j-1] = k-1],
17               array[1..n] of var 0..span: s =
18             [ t[i,j] | i in 1..n, j in 1..m where job[i,2*j-1] = k-1]
19             }
20             in
21             cumulative(s, d, one, 1)
22             );
23
24 constraint maximum(objective, [ t[i, m] + job[i, 2*m] | i in 1..n]);
25
26 solve ::
27       int_search([t[i,j] | i in 1..n, j in 1..m], smallest, indomain_min, complete)
28       minimize objective;
29
30 output [
31    "t = ", show(t), "\n",
32    "objective = ", show(objective), "\n"
33 ];
```

Listing 3.4: MiniZinc model based on the jobshop2 model from [Stu+14]

3.3.2 Heuristics

Whereas exact methods aim to find an optimal solution, heuristics focus on finding a good enough solution, usually with a fraction of the computational effort required. They are generally tailor-made for a specific problem, but might also be applicable to related problems.

Dispatching rules

Dispatching rule-based heuristics are among the simplest heuristics available for the JSSP. They work by simply scheduling one operation after another and determining the next operation to schedule by some dispatching rule. They are often preferred over other, more complicated approaches due to their ease of implementation and flexibility. The dispatching rules listed in table 3.2 below use the algorithm given in listing 3.5, which is based on the algorithm by Nguyen et al. [Ngu+13] – a variation of the classic algorithm by Giffler and Thompson [GT60].

Abbr.	Name	Description
`spt`	Shortest Processing Time	Choose $o_{j,i}$ minimizing $p_{j,o_{j,i}}$
`lpt`	Longest Processing Time	Choose $o_{j,i}$ maximizing $p_{j,o_{j,i}}$
`sps`	Shortest Processing Sequence	Choose $o_{j,i}$ minimizing $\sum_{l=i}^{m} 1$
`lps`	Longest Processing Sequence	Choose $o_{j,i}$ maximizing $\sum_{l=i}^{m}$
`lwr`	Least Work Remaining	Choose $o_{j,i}$ minimizing $\sum_{l=i}^{m} p_{j,o_{j,l}}$
`mwr`	Most Work Remaining	Choose $o_{j,i}$ maximizing $\sum_{l=i}^{m} p_{j,o_{j,l}}$

Table 3.2: Dispatching rules

Shifting bottleneck heuristic

The shifting bottleneck heuristic [Ada+88], along with its numerous variations, is generally regarded as one of the best heuristic methods available for the JSSP. Its core mechanic is iteratively creating schedules for each of the machines and combining those into a global schedule. The next machine to be scheduled is chosen by constructing and solving a single machine subproblem $1|r_j|L_{max}$, which is itself $\mathcal{NP}$-hard, for each of the potential candidates and identifying the one with the largest optimal value of L_{max} – the so-called bottleneck machine. This machine is then scheduled according to the optimal schedule for the single machine subproblem, after which all of the already scheduled machines' schedules are adjusted based on the newly introduced machine.

The implementation used is based on one of the variations by Demirkol et al. [Dem+97] and includes six final reoptimization iterations after all machines have been scheduled but incorporates no delayed precedence constraints. The single machine subproblems are solved to optimality using CP Optimizer.

```python
def solve(instance, rule):
  ready = []
  for j in range(0, instance.n):
    ready.append(instance[j][0])

  job_to_next_release = dict()
  machine_to_next_release = dict()

  def release(op):
    return max(job_to_next_release[op.job], machine_to_next_release[op.machine])
  def completion(op):
    return release(op) + op.duration

  release_times = dict()
  while len(ready) > 0:
    earliest_completed = min(ready, key=completion)
    candidates = [op for op in ready
                  if op.machine == earliest_completed.machine
                  and release(op) < completion(earliest_completed)]

    chosen_op = next_by_rule(candidates, rule)
    ready.remove(chosen_op)
    release_times[op] = release(chosen_op)
    job_to_next_release[op.job] = completion(chosen_op)
    machine_to_next_release[op.machine] = completion(chosen_op)

    if chosen_op.order < instance.m - 1:
      ready.append(instance[chosen_op.job][chosen_op.order + 1])

  return release_times
```

Listing 3.5: Generic algorithm for dispatching rule-based scheduling (Python-like pseudocode) based on [Ngu+13; GT60]

3.3.3 Local Search Metaheuristics

While heuristics are generally able to find a reasonable solution relatively fast, there is usually no guarantee that they will ever arrive at an optimal solution, even given enough time. Local search metaheuristics (from this point on referred to simply as metaheuristics) aim to remedy this by systematically exploring the search space of all possible solutions. Since exhaustive enumeration is usually not feasible, the exploration is done by repeatedly moving from a solution to one of its neighbors as defined by some neighborhood function.

For each algorithm, a detailed description in Python-like pseudocode is presented in order to provide as little ambiguity as possible in regard to the implementation. Nevertheless, there will always be implementation details not covered by the pseudocode. The actual implementation written in Rust and compiled with rustc 1.44.1 (c7087fe00 2020-06-17)

and LLVM 9.0 in release mode can be found on GitHub and mirrored to Zenodo.[2]

All metaheuristics use the neighborhood provided by Van Laarhoven et al. [vLa+92], referred to as N_1 by Vaessens et al. [Vae+96]. The basic idea behind this neighborhood is outlined in algorithm 3.1. The moves are generated by swapping two consecutive operations on the same machine along the critical path of the oriented disjunctive graph. This kind of swap will always result in a feasible schedule and that an optimal solution can always eventually be reached. The actual implementation of the neighborhood is slightly more complex in order to provide adequate performance.

Algorithm 3.1: Neighborhood N_1 for metaheuristics [vLa+92]

Input: $G' = (V, C \cup D'_{red})$ the disjunctive graph with oriented disjunctive edges with D'_{red} being the transitive reduction of the oriented edges D'
Output: A set of pairs of vertices along the critical path that may be swapped to obtain new solutions

```
P ← the set of all longest paths from s to t in G';
N ← ∅;
for p ∈ P do
    for (u, v) ∈ p do
        if (u, v) ∈ D'_red then
            N ← N ∪ {(u, v)};
        end
    end
end
return N
```

Random-restart hill climbing (`ran`)

Random-restart hill climbing may quite possibly be one of the simplest metaheuristics. It generates a random solution and improves it by means of a simple hill climbing algorithm, i.e. by repeatedly selecting the neighbor with the best target value. Once the hill climbing gets stuck in a local optimum, the process is restarted from a new random solution.

The high-level layout of random-restart hill climbing algorithm used in this thesis can be seen in listing 3.6.

[2] https://doi.org/10.5281/zenodo.4081660

```
def solve(instance):
  current = random_solution(instance)
  best = current

  def find_best(moves):
    best_move = None
    for move in moves:
      if best_move is None or move.cmax < best_move.cmax:
        best_move = move
    return best_move

  while elapsed_time() < timeout:
    move = find_best(neighbor_moves(current))
    if move is not None and move.cmax < current.cmax:
      current = move.solution
    else:
      current = random_solution()

    if current.cmax < best.cmax:
      best = current

  return best
```

Listing 3.6: Random-restart hill climbing metaheuristic (Python-like pseudocode)

Tabu search (`tab`)

At the core of tabu search is the idea of continuously exploring the search space without visiting formerly explored areas. This is achieved by allowing moves that degrade the current solution and 'blocking the way back', so to speak, by storing the previously made moves and prohibiting their inverse moves in the so-called tabu list.

The tabu search implementation given in listing 3.7 is a slight variation of the one given by Taillard [Tai94]. Its tabu list is implemented by storing the last time a node was swapped with its predecessor in the critical path. It also features a long-term memory mechanism, which penalizes moves that would push a node further towards the end of the critical path, depending on how often the node has already been pushed to the front. The main difference is that resets are permitted (similar to the random-restart hill climbing algorithm) if the tabu search is not able to find any valid move, thereby ensuring the search will continue for the entire allotted time.

```
def solve(instance):
  n, m, N = instance.n, instance.m, instance.n * instance.m

  current = random_solution(instance)
  best = current

  tabu_duration = (n + m/2) * e^(-n/5m) + N/2 * e^(-5m/n)
  last_swap = [-inf for op in range(0, N)]
  push_back_count = [0 for op in range(0, N)]
  max_delta = 0

  iteration = 0

  def find_best_permitted(moves):
    penalty_factor = 0.5 * max_delta * sqrt(N)
    best_move = None
    for move in moves:
      tabu_until = last_swap[move.a] + tabu_duration
      if iteration < tabu_until and move.cmax >= best.cmax: continue

      if best_move is None:
        best_move = move
        continue

      pb_total = sum(push_back_count)
      penalty = penalty_factor * push_back_count[move.b] / pb_total
      best_penalty = penalty_factor * push_back_count[best_move.b] / pb_total

      if move.cmax + penalty < best_move.cmax + best_penalty:
        best_move = move
    return best_move

  while elapsed_time() < timeout:
    move = find_best_permitted(neighbor_moves(current))
    if move is not None:
      max_delta = max(min(move.cmax - current.cmax, 0), max_delta)
      last_swap[move.b] = iteration
      push_back_count[move.b] += 1
      current = move.solution
    else:
      max_delta = 0
      last_swap = [-inf for op in range(0, N)]
      push_back_count = [0 for op in range(0, N)]
      current = random_solution()

    if current.cmax < best.cmax:
      best = current

  return best
```

Listing 3.7: Tabu search metaheuristic (Python-like pseudocode) based on [Tai94]

Simulated annealing (`sim`)

Simulated annealing is a probabilistic local search technique mimicking the physical process of annealing, i.e. the cooling of metals. The idea is to diversify the search early on by probabilistically accepting worsening moves and decreasing this probability as the search continues and the algorithm converges. To achieve this, a temperature parameter is decreased throughout the search. This temperature parameter and the amount by which a move improves/worsens the current solution are then used to determine the probability of accepting the move. The major factors in any simulated annealing implementation are the cooling schedule (how the temperature is decreased), the acceptance function (determining the probability of accepting a move), the initial temperature and the termination criterion (when to stop the search). The implementation is based on the one given by Van Laarhoven et al. [vLa+92] and can be seen in listing 3.8. The variation used in this thesis depends on only two parameters: The initial ratio of accepted solutions χ_0, which is used to estimate the initial temperature and the parameter δ determining the cooling rate. In place of a more elaborate stopping criterion, the search is restarted with a new initial solution once the standard deviation of the accepted moves' costs reaches 0.

Unlike the aforementioned methods, simulated annealing strongly depends on the parameter values used, which usually have to be adjusted for the specific problem instances it is applied to. For this purpose SMAC [Hut+11] is utilized to find a fitting parameter configuration for `all`. 64 parallel SMAC runs with a maximum of 128 evaluations each are performed on the instances from `all` with a 80/20 train/test split. The results confirm the suitability of the parameter values $\chi_0 = 0.95$ and $\delta = 0.001$, which have already been used by Van Laarhoven et al. [vLa+92] and will also be used in this thesis.

```
1  def solve(instance, timeout, χ0, εs, δ):
2    global_best = random_solution(instance)
3    while elapsed_time() < timeout:
4      solution = sa(instance, timeout, χ0, εs, δ):
5      if solution.cmax < global_best.cmax:
6        global_best = solution
7    return global_best
8
9  def sa(instance, timeout, χ0, εs, δ):
10   n, m, N = instance.n, instance.m, instance.n * instance.m
11   L = max(N − n, 1)
12
13   current = best = random_solution(instance)
14   T = estimate_initial_temperature(instance, initial_acceptance_ratio)
15   while elapsed_time() < timeout:
16     move_costs = [current.cmax]
17     for iteration in range(0, L):
18       if elapsed_time() >= timeout: break
19       move = select_random(neighbor_moves(current))
20       if move is None: break
21
22       ΔC = move.cmax − current.cmax
23       acceptance_threshold = 1 if ΔC ≤ 0 else min(1, e^(−ΔC/T))
24       if rand(0,1) < acceptance_threshold:
25         current = move.solution
26         move_costs.append(current.cmax)
27
28     if current.cmax < best.cmax:
29       best = current
30
31     σC = std_dev(move_costs)
32     if σC > 0:
33       T = T / (1+ln(T*(1+ΔC)/3σC))
34     else:
35       return best
36   return best
37
38 def estimate_initial_temperature(instance, χ0)
39   deltas = []
40   for _ in range(0, 30):
41     solution = random_solution(instance)
42     move = select_random(neighbor_moves(solution))
43     if move is not None: deltas.append(move.cmax − solution.cmax)
44
45   m1 = len([delta for delta in deltas if delta <= 0])
46   m2 = len([delta for delta in deltas if delta > 0])
47   ΔC = mean([delta for delta in deltas if delta > 0])
48   return ΔC · (ln(m2 / (m2·χ0 − (1−χ0)·m1)))^−1
```

Listing 3.8: Simulated annealing metaheuristic (Python-like pseudocode) based on [vLa+92]

3.4 Performance Data Generation

In order to obtain the performance data that the analysis is based on, the algorithms described in section 3.3 are applied to each of the instances from section 3.2. However due to the difference in instance sizes and processing time values, the values of C_{max} are vastly different between instances. To alleviate this problem a suitable performance measure must first be found, for the purpose of which various measures are evaluated in section 3.4.1. The methodology used to run the experiments and clean up the resulting data is given in section 3.4.2.

3.4.1 Performance Measures

The actual value of C_{max} is of little use for comparing results across instances, since it varies wildly depending on instance size and processing time values. The truly interesting metric is how far the result is from the optimal solution C^*_{max}, i.e. the ratio $R^* = \frac{C_{max}}{C^*_{max}}$. Unfortunately C^*_{max} is not known for all instances and cannot be obtained within an acceptable amount of time for larger instances. Therefore a reasonable approximation of R^* will have to suffice.

Intuitively the chosen performance measure P should satisfy the following criteria:

1. It should be invariant with respect to scale of processing times p, that is multiplying all processing times by a constant factor should have no effect on M.
2. It should be strongly correlated with R^*.
3. It should not be unduly biased towards or against certain types of instances.

For any performance measure P the Pearson correlation coefficient ρ_{P,R^*} is calculated on the set of all existing benchmark instances with known optimal solutions, using the performance data given in chapter 4. The exact set of instances can be found in table A.2. Instances for which the solver did not find a solution in time were excluded to avoid tainting the data. The results can be seen in figure 3.2. Note however that the selection of instances alone already introduces a bias since there are no optimal solutions available for harder instances and `lit` does not contain any instances with $n < m$. Nonetheless it should at least provide a decent estimate of how strongly P is correlated with R^*.

The following performance measures P_1 to P_6 shall therefore be considered and evaluated regarding their suitability. Note that the biases have only been tested empirically on the aforementioned set of instances, but have not been formally verified. One should therefore only consider them to be indicators of the true biases a certain performance measure may have.

1. $P_1 = C_{max}$. The makespan is the obvious first choice since it is the objective function. For any given instance P_1 is bounded by $[0, C^{seq}_{max}]$ and for the general

case it is bounded by $[0, +\infty[$. However it is not scale-invariant, is affected by both n and m and therefore also strongly biased against larger instances.

2. $P_2 = \frac{C_{max}}{C^{seq}_{max}}$. Considering C_{max} relative to the trivial upper bound C^{seq}_{max} results in a performance measure which is always nicely contained in $[0, 1]$. Additionally it is, per its very definition, scale-invariant.

   Unfortunately the bounds given above are slightly misleading since the true lower bound is different for each instance and can be anything between 0 and 1. For example, the optimal solution for a trivial instance obtains the worst possible score of $P_2 = 1$ since $C^*_{max} = C^{seq}_{max}$. In general, P_2 is biased against instances where $C^*_{max} \approx C^{seq}_{max}$.

3. $P_3 = \frac{C_{max}m}{C^{seq}_{max}}$. This measure was proposed by Mirshekarian and Šormaz [MŠ16], who refer to it as the scheduling efficiency C'. One can clearly see that its value must be bounded by $[1, m]$ for a given instance and thus by $[1, +\infty[$ globally.

   Like P_2 it is scale-invariant but also accounts for the fact that a high ratio $\frac{n}{m}$ tends to increase the value of C^*_{max}. It does however fail to account for the fact that a particularly low ratio $\frac{n}{m}$ can produce the same phenomenon. Additionally it introduces some undesirable asymmetries in regard to the number of jobs and machines. Take for example a sequential schedule for an instance I_1 where $n_1 = 1, m_1 = 2, p_{j,k} = 1 \forall j, k$. Obviously I_1 permits only sequential schedules and therefore $C^*_{max}(I_1) = C^{seq}_{max}(I_1) = p_{1,1} + p_{1,2} = 2$. Exchanging the values n and m produces another instance I_2 where $n_2 = 2, m_2 = 1, p_{j,k} = 1 \forall j, k$. Again I_2 permits only sequential schedules and therefore $C^*_{max}(I_2) = C^{seq}_{max}(I_2) = p_{1,1} + p_{2,1} = 2$. However, when calculating P_3 one obtains $P_3(I_1, s) = \frac{2m_1}{2} = m_1 = 2$ for I_1 and $P_3(I_2, S) = \frac{2m_2}{2} = m_2 = 1$ for I_2. Since I_1 and I_2 are effectively the same instances, one must conclude that P_3 is biased against instances with a particularly low job to machine ratio.

4. $P_4 = \frac{C_{max}nm}{C^{seq}_{max}}$. P_4 can be derived by dividing the makespan by the average processing time. It is therefore scale-invariant and fixes the asymmetry prevalent in P_3 by removing the obvious machine bias. Similarly to P_3 it is bounded globally by $[1, +\infty[$ and by $[1, nm]$ for a given instance since:

$$\frac{C_{max}nm}{C^{seq}_{max}} \leq \frac{C^{seq}_{max}nm}{C^{seq}_{max}}$$

   The bounds already give an indication as to what this measure might be biased against. Consider for example two sequential instances I_1 and I_2 with $n_1 = n_2 = 1$ and $m_1 = 2$ whereas $m_2 = 3$. Since the instances permit only sequential solutions $P_4(I_1) = n_1m_1 = 2$ and $P_4(I_2) = n_2m_2 = 3$, thereby indicating a bias against instance with larger values nm.

5. $P_5 = \frac{C_{max}}{\max(\max_j(\sum_k p_{j,k}), \max_k(\sum_j p_{j,k}))}$. P_5 considers the makespan relative to the lower bound given by Taillard [Tai93]. The lower bound is simply the maximum of

two distinct lower bounds, one of them being the machine with the longest total processing time and the other being the job with the longest total processing time.

P_5 is bounded by $[1, \max(n, m)]$. The lower bound is trivial since C_{max} can never be lower than the job or machine with the longest processing time. The upper bound can be derived as follows:

Per the definition of P_5:

$$P_5 = \frac{C_{max}}{\max(\max_j(\sum_k p_{j,k}), \max_k(\sum_j p_{j,k}))}$$

It follows that:

$$P_5 = \min(\frac{C_{max}}{\max_j(\sum_k p_{j,k})}, \frac{C_{max}}{\max_k(\sum_j p_{j,k})})$$

Considering only one the first fraction it can be deduced:

$$\frac{C_{max}}{\max_j(\sum_k p_{j,k})} \leq \frac{\sum_{j,k} p_{j,k}}{\max_j(\sum_k p_{j,k})} \leq \frac{n \max_j(\sum_k p_{j,k})}{\max_j(\sum_k p_{j,k})} = n$$

The other fraction follows analogously and therefore:

$$P_5 \leq \max(n, m)$$

An unfortunate property of P_5 is that the upper bound is dependent on the size of the problem instance and the lower bound of 1 can only be obtained for some very particular instances. Taillard [Tai93] conjectures that the lower bound becomes tight as $\frac{n}{m}$ goes to ∞. Streeter and Smith [SS06] have shown this to be true both empirically and, to some extent, formally for $\frac{n}{m} \to \infty$ and $\frac{n}{m} \to 0$. Unfortunately this in turn implies that P_5 is biased against instances with $\frac{n}{m} \approx 1$. On the upside, it does not suffer the same biases as P_2 or P_3 and is scale-invariant.

6. $P_6 = \frac{C_{max}}{C^{best}_{max}}$ where C^{best}_{max} is the makespan of the best algorithm on the given instance. This measure is closely related to the relative error, however it avoids the unergonomic situation of having a best score of 0 while still retaining all other benefits. It works well for comparing algorithms and, to a lesser extent, for comparing instances and is of course bounded by $\left[1, \frac{C^{seq}_{max}}{C^{*}_{max}}\right]$ for a given instance, since at best the result will be as good as the best algorithm, but it might be arbitrarily worse. On the upside it is not directly affected by the size of the problem and is also scale-invariant, as long as one assumes that the algorithms too are scale-invariant. However it may be biased in favor of harder instances since all instances will appear to have a good result as long as all algorithms perform similarly (even if they all perform badly). Additionally it is highly dependent on the exact set of algorithms used (and their implementations), making it hard to reproduce. Strictly speaking, in order to truly reproduce P_6 one would have to include exactly the same algorithms with the same experimental setup. However,

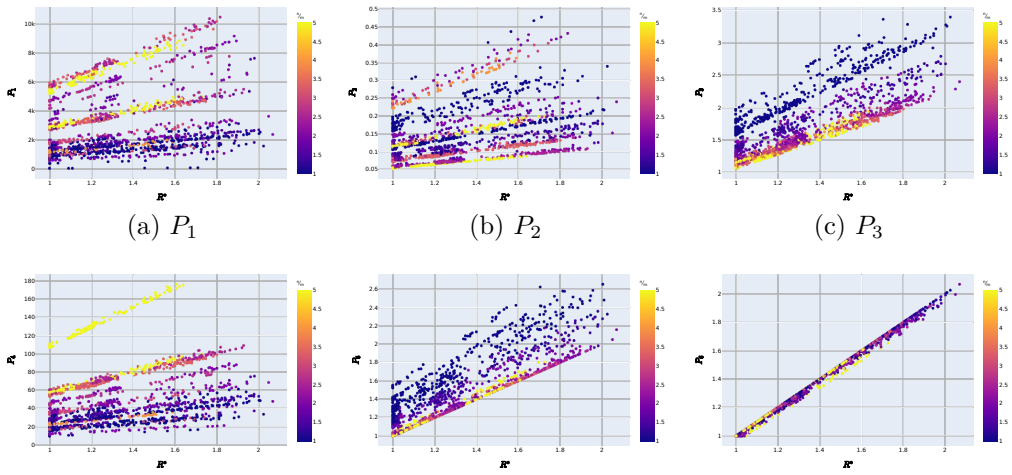
there are some redeeming qualities that may lessen the impact of this in practice. For one, it is most likely not necessary to implement all algorithms but rather only a small subset of particularly good algorithms since the weak algorithms will only have a small impact (if any at all). Furthermore, the quality of the approximation can only be improved as more and better algorithms are added to the portfolio, whereas other performance measures are essentially fixed and do not allow for any improvement.

The correlation between the performance measures and R^* can be seen in figure 3.2. Their relevant properties are summarized in table 3.3. P_1 is quite simply not comparable across instances due to it lacking scale invariance and both P_2 and P_3 suffer from some very obvious and unfortunate biases. While P_4 looks interesting on paper, its correlation with R^* is rather poor compared to the other performance measures. P_5 presents a decent choice, being easy to implement and having a reasonable correlation with R^*. However the bias against instances with particularly high or low values of $\frac{n}{m}$ make it problematic in practice – an issue that is also evident when comparing it to R^*. This leaves only P_6 which has the major drawback of being dependent on the algorithms used and is, at least to some extent, biased against easy instances.

Unfortunately none of the aforementioned performance measures completely statisfy all criteria posed above – certain tradeoffs are therefore required. In this case, P_6 was chosen as the performance measure best suited for the purpose of this thesis. The bias against easy instances does not seem to be much of a problem in practice, or, at the very least, it is significantly less biased than the other performance measures.

Measure	Inst. bounds	Global bounds	Scale inv.	ρ_{P,R^*}	Bias against
P_1	$[0, C^{seq}_{max}]$	$[0, +\infty]$	no	0.3101	high m, n, $p_{j,k}$
P_2	$[0, 1]$	$[0, 1]$	yes	0.2825	$C^*_{max} \approx C^{seq}_{max}$
P_3	$[1, m]$	$[1, +\infty]$	yes	0.7638	low $\frac{n}{m}$
P_4	$[1, nm]$	$[1, +\infty]$	yes	0.3151	high nm
P_5	$[1, \max(n, m)]$	$[1, +\infty]$	yes	0.8544	$\frac{n}{m} \approx 1$
P_6	$\left[1, \frac{C^{seq}_{max}}{C^*_{max}}\right]$	$[1, +\infty]$	yes	0.9988	easy instances

Table 3.3: Relevant properties of performance measures



(a) P_1 (b) P_2 (c) P_3

(d) P_4 (e) P_5 (f) P_6

Figure 3.2: R^* versus performance measure for results of all algorithms on instances with known optimum

3.4.2 Experiment Methodology

Each run is performed on a single Intel Xeon E5-2650 v4 core with a set wall-clock time limit of 5 minutes. Since the algorithms check the elapsed time internally, most runs end up taking slightly longer than the given time limit. However the effect is negligible for the most part with the difference in mean runtime being $\leq 2s$ for all algorithms. The results the analysis in chapter 4 is based on (minus the excluded algorithms), as well as the features for each instance, can be found on GitHub and Zenodo.[3]

All metaheuristics are run 5 times with different seeds (41, 42, 43, 44, 45) so as to control for random factors, from which the median result (by makespan) is chosen for evaluation. The mean absolute percentage error when comparing all results on a given instance to the median result is 0.48% for `sim`, 0.59% for `ran` and 0.5% for `tab`. Exact methods and heuristics are run a single time until the time limit is reached. CPLEX is excluded due to it being unable to solve 1854 ($\sim$57%) instances from `all`, making it unsuitable for further analysis. The same goes for the shifting bottleneck heuristic since, even though it generally produces good results, its runtime scales exceedingly poorly to larger instances and it exceeds the time limit on 2177 ($\sim$67%) instances, thereby disqualifying it. It is worth noting however that it does provide decent solutions on the instances it manages to solve within the time limit. Furthermore, the data for Chuffed requires minor corrections due to it being unable to produce a feasible schedule for `orb07`. This is most likely caused by a bug in the solver or model triggered by `orb07` containing an operation with a processing time of 0. Since this is a singular issue, it is remedied by assigning Chuffed the worst C_{max} of all algorithms for `orb07` only.

[3] https://doi.org/10.5281/zenodo.4081662

3.5 Features

To characterize the instances and determine their location in the instance space, a number of features are derived from each instance. While some features per definition consist of only a single value, most require some kind of aggregation over a set of values. For these, the set of statistical key figures given in table 3.4 is calculated.

Figure	Description
min	Minimum value
max	Maximum value
range	max − min
mean	Mean value
q1	First quartile
median	Median (second quartile)
q3	Third quartile
std-dev	Standard deviation
gini	Gini coefficient

Table 3.4: Statistical key figures

Apart from the basic features (*jobs*, *machines*, *operations*, *job-machine-ratio*), the feature set also includes the lower bound from [Tai94] *maxlb*, the makespan of a sequential schedule *seq-cmax* and the *skewness*, i.e. $max(\frac{n}{m}, \frac{m}{n})$. A number of features related to operation slots and machine load are taken from [MŠ16]. Furthermore additional probing features and features based on the disjunctive graph of the problem instance are proposed, which will be described below in more detail.

3.5.1 Probing Features

The probing features are calculated by running each of the algorithms selected for analysis (see section 3.3, section 3.4) a single time with a time limit of 1 second on the same hardware as is used for the full experiments. The shifting bottleneck heuristic had to be excluded due to it almost never obtaining a solution within the time limit. The makespan of the solution (or C^{seq}_{max} if none was found) is obtained and the other performance measures P_{2-5} are derived from it. P_6 is intentionally left out due to it being too dependent on the exact set of algorithms used. Furthermore for each performance measure statistical key figures are calculated on the subset of all heuristics, the subset of all metaheuristics, the subset of all exact methods, and finally the entire set of algorithms.

3.5.2 Graph Features

The graph features are based on the undirected disjunctive graph of the problem instance, which is derived from the directed graph by replacing each directed arc (u, v) with an undirected edge $\{u, v\}$. The following features are extracted from the graph.

- *density*: The density of the graph $dens(G) = \frac{2|E|}{|V|(|V|-1)}$.
- *vertex-degree*: The number of edges that are incident to v, i.e. $d(v) = |\{e \in E : v \in e\}|$.
- *betweenness-centrality*: The shortest path betweenness centrality measures how often a vertex v occurs as a bridge on the shortest path between two other vertices. No node weights (or edge weights) are considered for this measure and the shortest path is thus always the path with the fewest edges.

  Formally it is given by $bc(v) = \sum_{s \neq t \in V \setminus \{v\}} \frac{\sigma(s,t|v)}{\sigma(s,t)}$ where $\sigma(s,t|v)$ is the number of shortest paths from s to t passing through v, and $\sigma(s,t)$ is the number of shortest paths from s to t. The actual value used is the betweenness centrality normalized by the number of vertex pairs $\frac{(|V|-1)(|V|-2)}{2}$.
- *clustering-coefficient*: The clustering coefficient of a vertex v is the normalized number of triangles passing through v. Formally $cc(v) = \frac{2T(v)}{d(v)(d(v)-1)}$ where $T(v)$ is the number of triangles passing through v.

Ultimately this results in a total of 604 features per instance, which are listed in table 3.5. It should be noted however that some carry no information (e.g. the mean of a set of values that is already normalized by the mean) or are strongly correlated with some other features. Nonetheless, all features are included for the sake of consistency and the task of determining their usefulness is left to the feature selection.

Group	Feature(s)	Description
Overall	*jobs*	number of jobs n
	machines	number of machines m
	operations	number of operations nm
	job-machine-ratio	job to machine ratio $\frac{n}{m}$
	skewness	Maximum of job to machine ratio and machine to job ratio $\max(\frac{n}{m}, \frac{m}{n})$
	seq-cmax	makespan of the sequential schedule C^{seq}_{max}
	maxlb	lower bound from [Tai93]
Operations	*operation-pt*	statistical key figures regarding the processing time of all operations
Jobs	*job-pt*	statistical key figures regarding the total processing time per job (relativ to mean, mean operation processing time)

Group	Feature(s)	Description
Machines	*machine-pt*	statistical key figures regarding the total processing time per machine (relative to mean, mean operation processing time)
Operation slots	*os-pt*	statistical key figures regarding the total processing time per operation slot (relative to mean, mean operation processing time)
	os-mm	statistical key figures regarding the number of missing machines per operation slot (relative to mean, number of machines)
	os-rm	statistical key figures regarding the number of repeated machines per operation slot (relative to mean, number of machines)
	os-rma	statistical key figures regarding the number of repeated machines per operation slot (amplified) (relative to mean, number of machines, mean operation processing time)
	os-rma-pt	statistical key figures regarding the number of repeated machines (amplified) multiplied by the average operation processing time per operation slot (relative to mean, number of machines, mean operation processing time)
Machine load	*mlu*	statistical key figures regarding the machine load uniformity (relative to mean, number of machines)
	mlv	statistical key figures regarding the machine load voids (relative to mean, number of machines)
	mlva	statistical key figures regarding the machine load voids (amplified) (relative to mean, number of machines)
Probing	*priority-spt*	P_{1-5} for shortest processing time dispatching rule heuristic
	priority-lpt	P_{1-5} for longest processing time dispatching rule heuristic
	priority-sps	P_{1-5} for shortest processing sequence dispatching rule heuristic
	priority-lps	P_{1-5} for longest processing sequence dispatching rule heuristic
	priority-lwr	P_{1-5} for least work remaining dispatching rule heuristic
	priority-mwr	P_{1-5} for most work remaining dispatching rule heuristic

Group	Feature(s)	Description
	tab	P_{1-5} for tabu search metaheuristic
	ran	P_{1-5} for random restart hill-climbing metaheuristic
	sim	P_{1-5} for simulated annealing metaheuristic
	cpo	P_{1-5} for CP Optimizer
	ort	P_{1-5} for OR-Tools
	chu	P_{1-5} for Chuffed
	cpl	P_{1-5} for CPlex
	heuristic	statistical key figures regarding P_{1-5} for all heuristics
	metaheuristic	statistical key figures regarding P_{1-5} for all metaheuristics
	exact	statistical key figures regarding P_{1-5} for all exact methods
	all	statistical key figures regarding P_{1-5} for all algorithms
Graph	*density*	density of the undirected disjunctive graph
	vertex-degree	statistical key figures regarding the degree of all vertices in the undirected disjunctive graph
	betweenness-centrality	statistical key figures regarding the normalized betweenness centrality of all vertices in the undirected disjunctive graph
	clustering-coefficient	statistical key figures regarding the clustering coefficient of all vertices in the undirected disjunctive graph

Table 3.5: Instance features

3.5.3 Feature Selection

The following methodology for selecting features so as to obtain an optimal instance space projection is based on the work of Muñoz [Muñ20] on the excellent MATILDA [oMe20] toolkit provided by the University of Melbourne, of which a slightly customized version is used for selection and projection of the features. The feature values are preprocessed by bounding outliers to the median plus/minus five times the interquartile range and normalizing the feature values to be close to normally distributed by applying a box-cox transformation [BC64] and a Z-score standardization. Based on the preprocessed features, an optimal subset is selected to be used for further analysis. First, the top 6 features

by Pearson correlation between feature and algorithm performance are selected for each algorithm. All selected features are clustered by applying k-means clustering (max. 1000 iterations, 100 repeats, max. 6 clusters), based on the pairwise Pearson correlation coefficient. The goal then is to select a single feature from each cluster so as to achieve an optimal visualization later on. To estimate the suitability of a feature set, the features are projected into $\mathbb{R}$ using principal component analysis. On this projection, a Random Forest with 100 trees is applied to predict whether an instance is easy or not for each algorithm. A genetic algorithm is then used to search for the subset of features that results in the most accurate Random Forest predictions. The selected features are finally projected into $\mathbb{R}^2$ using the method described in [Muñ+18] for visualization purposes.

To better understand the relations between features, a clustering of all features is performed using agglomerative clustering with 55 clusters (chosen using silhouette analysis), $1-\rho$ as the distance metric and the average distance between each observation as the linkage criterion. Constant features, that is features with a standard deviation of 0 (plus/minus floating-point errors), do not carry any information and are dropped before clustering. A complete list of all clusters and their respective features is included in table A.1.

3.6 Performance Data Analysis

When analyzing the algorithms' performances, this thesis will primarily focus on the performance measure P_6 for the reasons given in section 3.4.1. Sometimes, however, it is useful to obtain a binary view of an algorithm's performance, for which two additional measures inspired by [SM+14] are used. Firstly, an algorithm shall be considered ϵ-good on a given instance if its performance is less than or equal to $(1+\epsilon)$-times the best performance on that instance. Under the chosen performance measure P_6 this works out to $P_6 \leq 1+\epsilon$ since the best algorithm always obtains a score of 1. If an algorithm is ϵ-good for $\epsilon = 0.05$, it will be referred to as *good* and *bad* otherwise. Secondly, an algorithm will be considered a *winner* for a given instance if it is among the best algorithms for that instance, i.e. it is ϵ-good with $\epsilon = 0$ (or, in terms of P_6, $P_6 = 1$). An algorithm that is not only a winner on an instance, but even the only winner, shall then be referred to as the *unique winner*.

To visualize the performance, the projection of the selected features into $\mathbb{R}^2$, as given in section 3.5, is chosen as the primary tool, besides the usual commonly used plots and visualization methods. It is worth noting that all other feature visualizations use the raw feature values, whereas only the projection itself is based on the preprocessed features. Further insights into the shape of the instance space are obtained by analyzing trained machine learning models (see section 3.7) with regard to the impact a feature has on the quality of the models' predictions. The predictive strength of the feature is estimated via its so-called permutation importance. To calculate the permutation importance of a given feature, a Random Forest classifier is fitted on the dataset and then evaluated on the same dataset, providing a baseline performance value for the classifier. In the next

step, the entire feature column is shuffled and the classifier's performance is reevaluated on the modified dataset, thereby breaking the relationship between the feature and the target. The difference to the baseline thus gives an estimate for the importance of the feature in the classifier. The performance measure used is the MCC (see section 3.7) and the process is repeated 30 times per feature.

3.7 Algorithm Selection

A number of different machine learning models are evaluated with the goal of predicting the optimal algorithm, that is the algorithm with the minimal makespan, for any given instance, based on a subset of the features from section 3.5. The problem is modeled as a multiclass classification problem, with the classes being the previously listed algorithms and the ground truth being the algorithm with the minimal makespan on the instance. Ties are broken in favor of the contender with the largest number of won instances over the entire dataset. It is worth noting that this way of breaking ties does introduce a minor bias against algorithms that perform well but are seldomly the best. The performance of the different machine learning models is evaluated using 20-times iterated stratified 10-fold cross-validation, with the primary performance measure being the multiclass generalization of the Matthews Correlation Coefficient (MCC) [Mat75; Gor04; Jur+12]. The value of the multiclass MCC ranges from somewhere between -1 and 0 (depending on the class distribution) up to 1 with 0 being essentially random and 1 a perfect prediction. As a secondary performance measure, the actual performance of the solver based on the model is calculated. This secondary measure is not included in the training of the models in any way and is only calculated on the validation set.

All of the following models are implemented using scikit-learn (0.23.2) and the default values are used for all parameters not mentioned.

- Most Frequent (`MF`), a model that always predicts the most frequent class, thereby serving as a trivial baseline predictor.
- K-Nearest-Neighbors (`KNN`) with k neighbors, uniform weights and euclidean distance. The value of k is set to 3, which was determined to be optimal by searching all values from 1 to 100 and choosing the one with the best mean MCC over 10 cross-validation splits.
- Decision Tree (`DT`) with Gini impurity as the split criterion.
- Random Forest (`RF`) with n trees, Gini impurity as the split criterion and the maximum number of features to consider for a split set to $\sqrt{k}$, where k is the total number of features. The value of n is set to 170, which was found by the aforementioned search method, with values from 10 to 1000 and a step width of 10.
- Multilayer Perceptron (`MLP`) with n hidden nodes and a rectified linear unit (ReLu) activation function. The value of n is set to 80, which was found by searching all values from 10 to 200 with a step width of 10.

CHAPTER 4

Experimental Evaluation

This chapter starts out with a general overview of the overall performance of different algorithms and the basic shape of the instance space to provide a foundation for the more in-depth parts of the analysis. Next up is an analysis of the selected features, their distribution across the instance space as well as their importance and similarities. This is followed by a more detailed discussion regarding the individual performance characteristics of the evaluated algorithms. Finally the information obtained in the previous steps is used to build machine learning models, which aim to select the best algorithm for a given instance and thereby create a portfolio solver that improves over the performance of any individual algorithm. Woven through the entire chapter is a comparison between `lit` and `gen` to provide a general insight into how well the existing benchmark instances already cover the space.

4.1 Overall Performance

Before considering more fine-grained performance measures, it is worth taking a high-level look at the number of instances from `all` where a given algorithm was the (unique) winner in table 4.1. From this basic analysis alone one can already infer that three algorithms (`cpo`, `ort` and `tab`) are likely to be serious contenders since they are the algorithms with the highest number of wins and the only algorithms that manage to be the unique winner on at least one instance. All other algorithms do find the optimal solution in some cases, however most heuristics (`spt`, `lpt`, `lps`, `lwr`) only obtain a winning solution for the trivial instances, with the exception of `sps` and `mwr`, which manage to win an additional 31 and 15 instances respectively.

This assessment is mostly in agreement with the aggregated performance of all algorithms across `all`. However, there are some new insights to be found too. Consider the results of the Wilcoxon signed-rank test in figure 4.3c and the statistical key figures in table 4.2. When comparing only the mean P_6, `cpo` is actually the best followed by `tab` and then

Algorithm	#Wins	#Unique wins
cpo	1528	293
ort	1742	633
chu	835	
ran	556	
tab	2079	1069
sim	620	
spt	525	
lpt	525	
sps	556	
lps	525	
lwr	525	
mwr	540	

Table 4.1: Number of (unique) wins by algorithm on instances from `all`

`chu`/`ort`, both of which are roughly tied for third place. The dispatching rule-based heuristics generally do not exhibit spectacular performance, except for `sps` and `mwr`. In particular `mwr` stands out by achieving impressively good results, considering its relative simplicity, and beating even `ran`, `sim` as well as all other heuristics. Of course both `ran` and `sim` could easily close this gap by using `mwr` as the initial construction heuristic.

However the mean P_6 does not tell the whole story – in fact comparing the paired differences with the Wilcoxon signed-rank test determines `tab` to be better than all other algorithms, followed by `cpo`, `ort` and then `chu`. The reason for this can already be glimpsed in figure 4.1 in which `tab` shows a far wider range of outliers, which skew the mean but leave the results of the Wilcoxon tests largely unaffected. Furthermore, when considering the standard deviation and maximum values, exact methods, with the exception of `ort`, show a lower standard deviation and a lower maximum value than metaheuristics, suggesting a more consistent performance; an assessment that is also in line with the distribution in figure 4.1.

Algorithm	Mean	Std Dev	Minimum	Q_1	Median	Q_3	Maximum
cpo	1.0203	0.0259	1.0000	1.0000	1.0046	1.0390	1.1094
ort	1.1086	0.1656	1.0000	1.0000	1.0000	1.1969	1.5728
chu	1.1065	0.0935	1.0000	1.0000	1.0955	1.1803	1.7834
ran	1.2651	0.1957	1.0000	1.0638	1.2991	1.3972	2.0000
tab	1.0697	0.1837	1.0000	1.0000	1.0000	1.0126	1.9935
sim	1.2158	0.1851	1.0000	1.0352	1.2330	1.3092	2.1346
spt	1.4894	0.2866	1.0000	1.3259	1.5282	1.6905	2.2707
lpt	1.3440	0.1943	1.0000	1.2318	1.3862	1.4855	1.8792
sps	1.1516	0.1011	1.0000	1.0812	1.1643	1.2164	1.6525
lps	1.5413	0.2900	1.0000	1.3751	1.6370	1.7550	2.1618
lwr	1.5736	0.3064	1.0000	1.4092	1.6761	1.8018	2.3000
mwr	1.1458	0.0963	1.0000	1.0681	1.1584	1.2096	1.5090

Table 4.2: Statistical key figures for aggregated P_6 by algorithm over instances from all

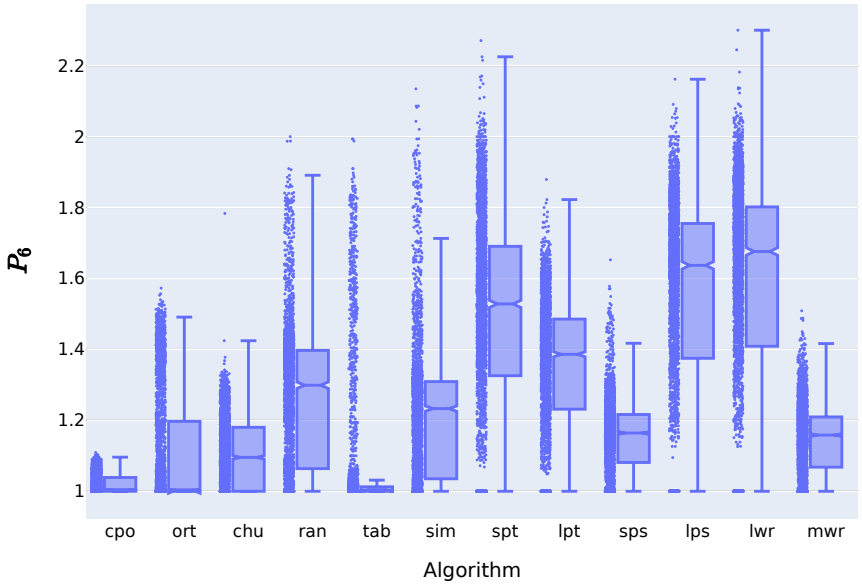


Figure 4.1: P_6 by algorithm for instances from all

When comparing the results from the gen and lit dataset in figure 4.2, one finds the general trends to be the same: tab and cpo are the best algorithms, ort and chu are right behind, and the dispatching rule-based heuristics also show similar behavior. However there are also some notable differences: For one, the gen dataset generally results in a larger range of values, as is to be expected due to it containing both very easy (e.g. $1\times m$ or $n\times 1$) as well as very hard problems (e.g. 100×100). The inclusion of trivial instances may also explain why the dispatching rule-based heuristics perform better on the gen dataset. The exact methods appear to be less affected by this but they too suffer from decreased performance on the harder instances.

Another interesting fact to note is the difference in performance between chu, ort and sim according to the Wilcoxon test, as can be seen in figure 4.3. While chu shows no significantly better performance than ort and better performance than sim on gen, it actually falls behind both on lit. Likewise ran exhibits a statistically significant improvement in performance over sps and mwr on lit but not on gen. Furthermore the difference in performance between tab and cpo is not statistically significant on lit while it is on gen.

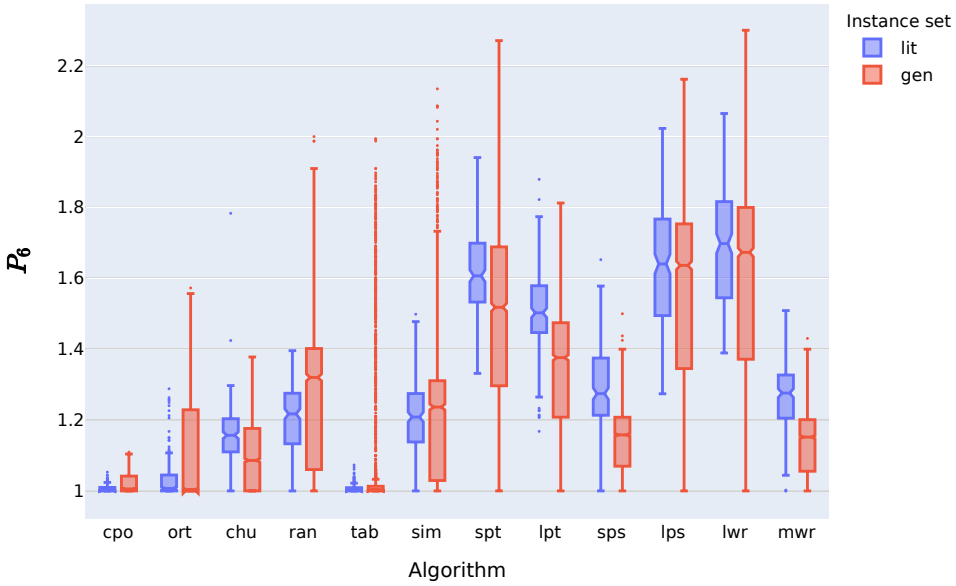
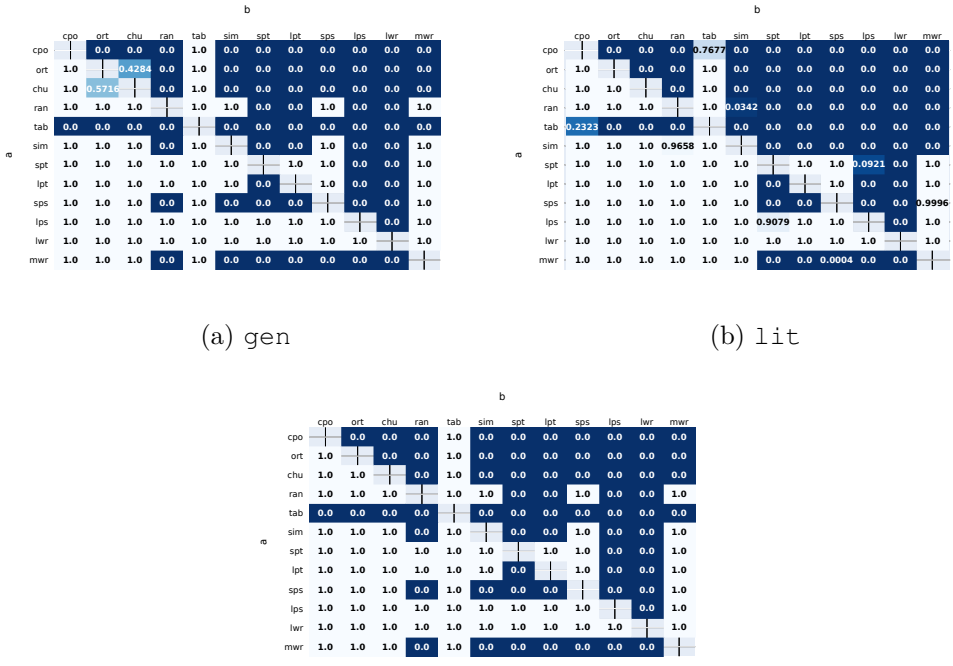


Figure 4.2: P_6 by algorithm for instances from lit and gen



(a) `gen` (b) `lit`

(c) `all`

Figure 4.3: p-values comparing P_6 of algorithms | one-sided, less

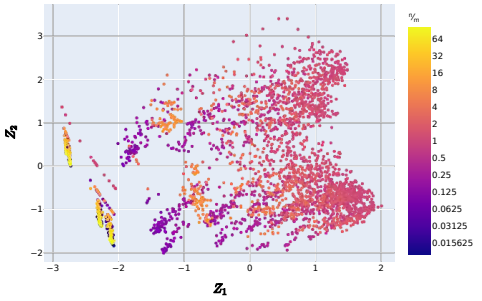
4.2 Shape of the Instance Space

The instances are projected to $\mathbb{R}^2$ by calculating the coordinates Z_1 and Z_2 as a linear combination of the selected features according to section 3.5, the results of which can be seen in figure 4.4. The selected features themselves will be further discussed in section 4.3.

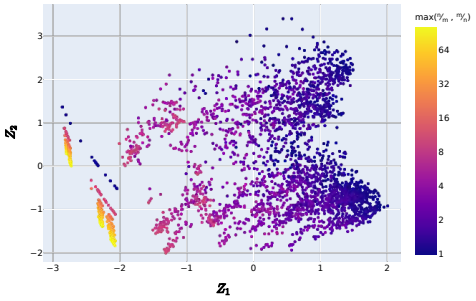
Considering figures 4.4a to 4.4d reveals a highly distinct cluster of trivial instances around $Z_1 \approx -2.5$. One can also identify an approximately linear relationship between the skewness and Z_1. Instances with a high number of jobs are generally found on the right side of the chart ($Z_1 \gtrapprox 0.5$), except for two anomalous clusters near $Z_1 \approx -1$. These clusters also stand out as having a low number of machines and therefore an unusually high job to machine ratio.

When breaking down the instances from `gen` by their respective source (figure 4.5a), one can clearly see that instances based on different probability distributions occupy different areas in the instance space, although they do have some overlap. Particularly `gen-uniform` and `gen-binom` seem to occupy similar areas which are quite distinct from `gen-nbinom` and `gen-const`. Considering the distribution of instances from `lit` (figure 4.5b) reveals that the instances from `lit` are mostly concentrated around the center of the space and do not cover some areas at all. This can, to some extent,

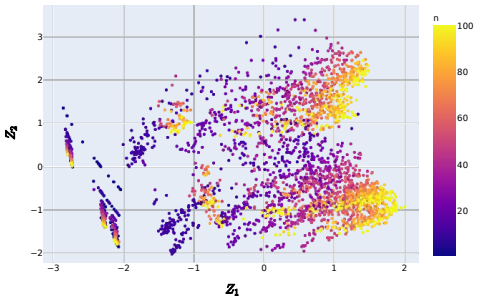
be explained by lack of extreme job to machine ratios and the lack of large numbers of jobs and/or machines in `lit`. However, almost no instances from `lit` are to be found in the area covered by `gen-const` and very few in the area covered by `gen-nbinom`, suggesting the difference goes beyond instance size.



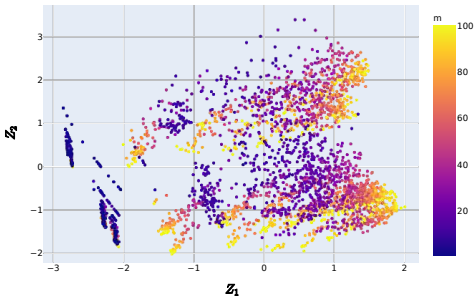
(a) Job to machine ratio $\frac{n}{m}$



(b) Skewness $\max(\frac{n}{m}, \frac{m}{n})$

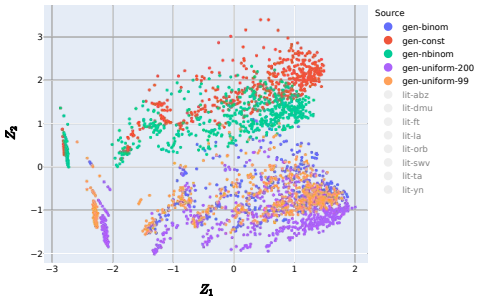


(c) Number of jobs n

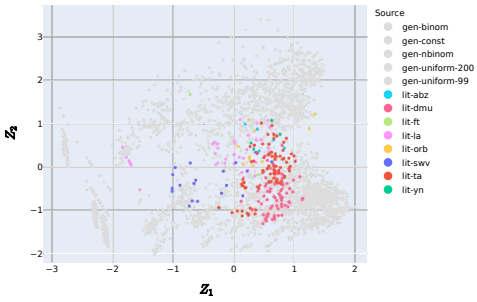


(d) Number of machines m

Figure 4.4: Basic features of projected instances from `all`



(a) `gen` highlighted



(b) `lit` highlighted

Figure 4.5: Source of projected instances from `all`

4.3 Features

The selected features and the coefficients for projection into $\mathbb{R}^2$ are listed in table 4.3, which result in an explained variance of 0.4936. From 6 selected features, a full 3 of them are probing features. The other features are related to the graph clustering coefficient and operation slot missing machines. Their distribution over `all` is visualized in figure 4.8.

Due to the large number of features generated, there will inevitably be some overlap between them, i.e. there exist certain groups of features which basically carry the same, or at least very similar, information – patterns that can be revealed by a clustering analysis. A complete list of all clusters and their respective features is included in table A.1, whereas a high-level visualization of the correlation between features, sorted and labelled by their respective clusters, can be seen in figure 4.9. Due to this there are also a wide range of feature combinations that produce a similar result with only minor variations, making the feature selection highly susceptible to minor changes in the input data.

Feature	Cluster	Z_1	Z_2
graph-clustering-coefficient-range	1	0.5462	0.2915
mlv-rel-m-std-dev	24	0.4127	0.0334
operation-pt-mean	9	0.2374	−0.209
probing-exact-p1-median	2	0.0598	−1.0038
probing-heuristic-p5-std-dev	1	−0.0143	0.4773
probing-sim-p5	11	0.519	0.1715

Table 4.3: Selected features with cluster number and projection coefficients

When ranking the selected features by their permutation importance, as can be seen in figure 4.6, it becomes clear that some features have a significantly higher impact on the model's performance than others. The importance scores may of course vary for different

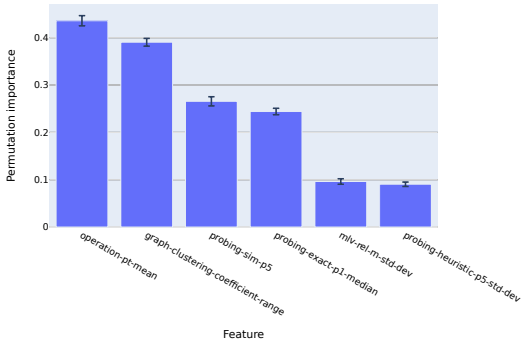


Figure 4.6: Mean and standard deviation of permutation importance of selected features for 30 repetitions

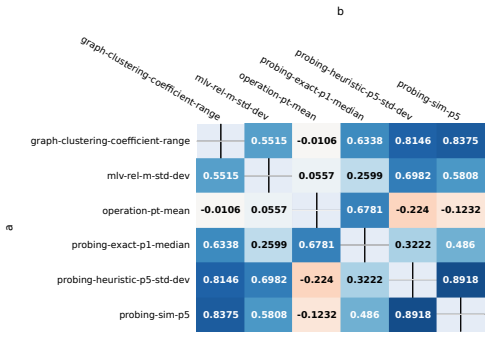


Figure 4.7: Pearson correlation coefficient for pairs of selected features

models, however they serve as a basic indicator of a feature's predictive strength. When taking a closer look at the selected features one finds the following.

1. *graph-clustering-coefficient-range*: The range of the clustering coefficients for each vertex in the merged disjunctive graph achieves a permutation importance score of 0.39. Based on visual comparison, the clustering coefficient seems to follow a similar pattern as the job to machine ratio, while nicely separating the cluster of trivial instances from the others.

   It is contained in cluster #1, which incidentally also includes *probing-heuristic-p5-std-dev*, as well as several other probing features, all of them either the standard deviation, range or Gini coefficient. The only other non-probing features included are *mlu-range* and *mlu-std-dev*.

2. *mlv-rel-m-std-dev*: The standard deviation of the machine load voids relative to the total number of machines achieves a score of 0.1 and shows an interesting pattern. It is relatively uniformly distributed, with larger values for instances with a job to machine ratio close to 1. However it also shows some very curious outliers with particularly high values near the center of the instance space.

   Cluster #24 is a rather small cluster containing only 6 features. These features are the range/standard deviation of *mlv-rel-m*, *os-mm-rel-m* and *os-rm-rel-m* respectively.

3. *operation-pt-mean*: The mean $p_{j,k}$ value achieves a score of 0.44, thereby making it the most important feature by a narrow margin. When viewing the distribution of this feature in figure 4.8c one finds that it, unsurprisingly, has the highest values among instances from `gen-uniform-200`, followed by `gen-uniform-99` and `gen-binom`, whereas `gen-const` and `gen-nbinom` show significantly lower values. Considering its correlation to other selected features in figure 4.7 shows it only being correlated with *probing-exact-p1-median* – an observation that can easily be explained by the fact that none of the other features incorporate the processing times to a similar extent.

   Its cluster, cluster #9, is relatively small and contains only key figures related to *operation-pt*.

4. *probing-exact-p1-median*: The median P_1 of all exact methods achieves a score of 0.24. Following the same logic as the previous feature, this one appears to be useful for identifying instances that are easy/hard for exact methods. However, it is notable that the makespan was chosen, which is dependent on n, m as well as the values of $p_{j,k}$ in place of a metric that is to some extent normalized by problem size. This results in a very clear cluster of high values for the large instances in `gen-uniform-200`.

   This feature is contained in cluster #2, with the majority of features being somehow influenced by the values of $p_{j,k}$ such as *seq-cmax*, *maxlb*, *jop-pt*, *machine-pt*, *os-operation-pt*, *os-rm-pt* and a variety of probing features based on P_1.

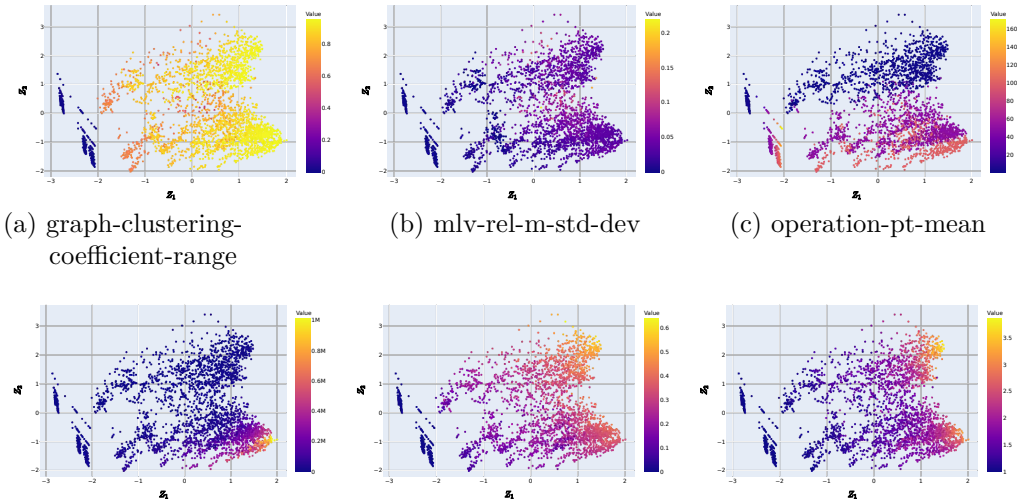
5. *probing-heuristic-p5-std-dev*: This feature is contained in the same cluster as *graph-clustering-coefficient-range* and follows an analogous pattern, even if it might not be immediately obvious when comparing their visualizations.

   They do however show a strong correlation as can be seen in figure 4.7. Unsurprisingly it has a relatively low importance of 0.09.

6. *probing-sim-p5*: The value of P_5 for `sim` achieves an importance of 0.27. Loosely speaking, one might say that this feature serves to separate instances by metaheuristic performance.

   It is contained in cluster #11, which is made up entirely of probing features based on P_5 of heuristics and metaheuristics.

The presence of non-probing features, henceforth referred to as *simple* features, in the clusters of the selected features' clusters as well as the superficially similar distribution of probing and simple features suggests that the probing features may carry relatively little additional information – a hypothesis that will be further explored in section 4.5.



(a) graph-clustering-coefficient-range (b) mlv-rel-m-std-dev (c) operation-pt-mean

(d) probing-exact-p1-median (e) probing-heuristic-p5-std-dev (f) probing-sim-p5

Figure 4.8: Feature values of projected instances from `all`

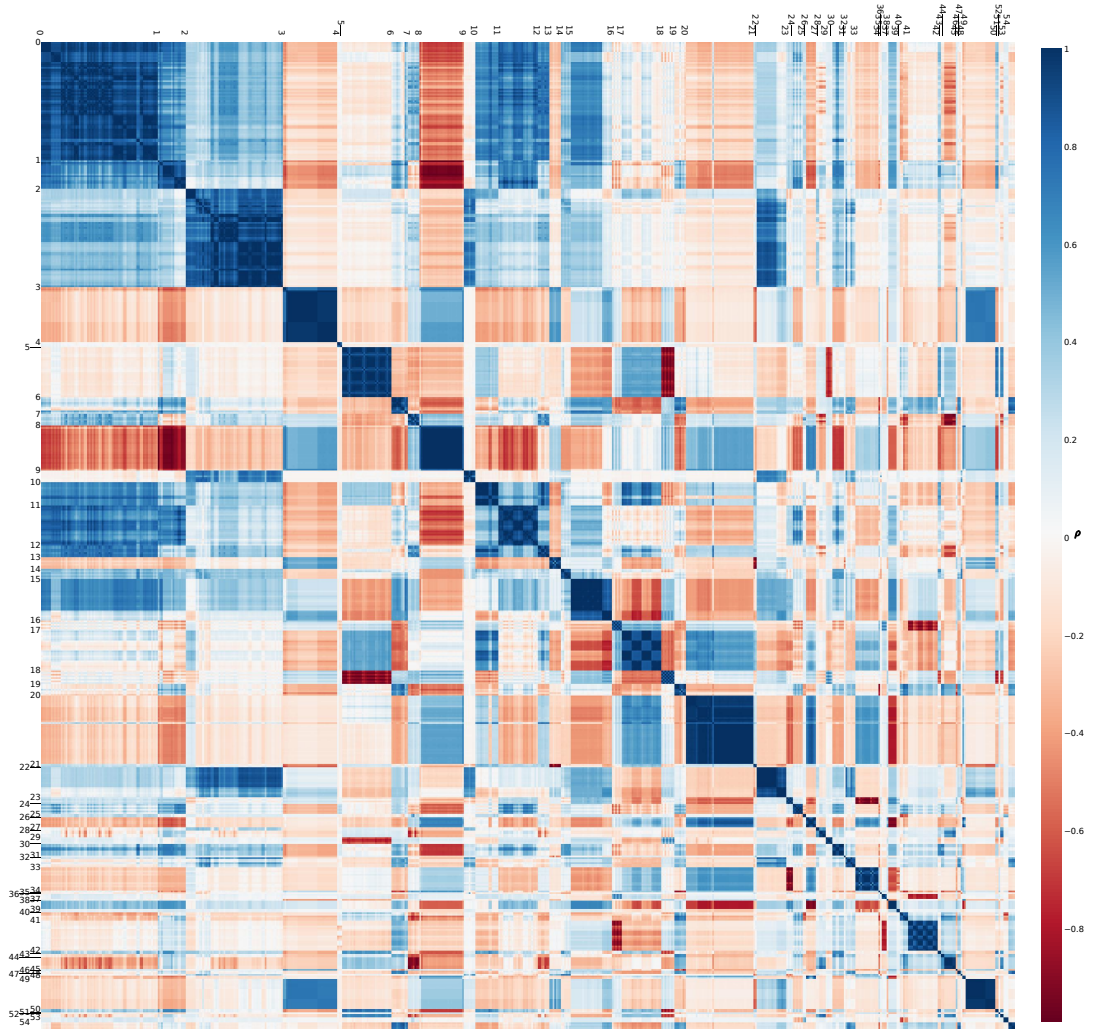
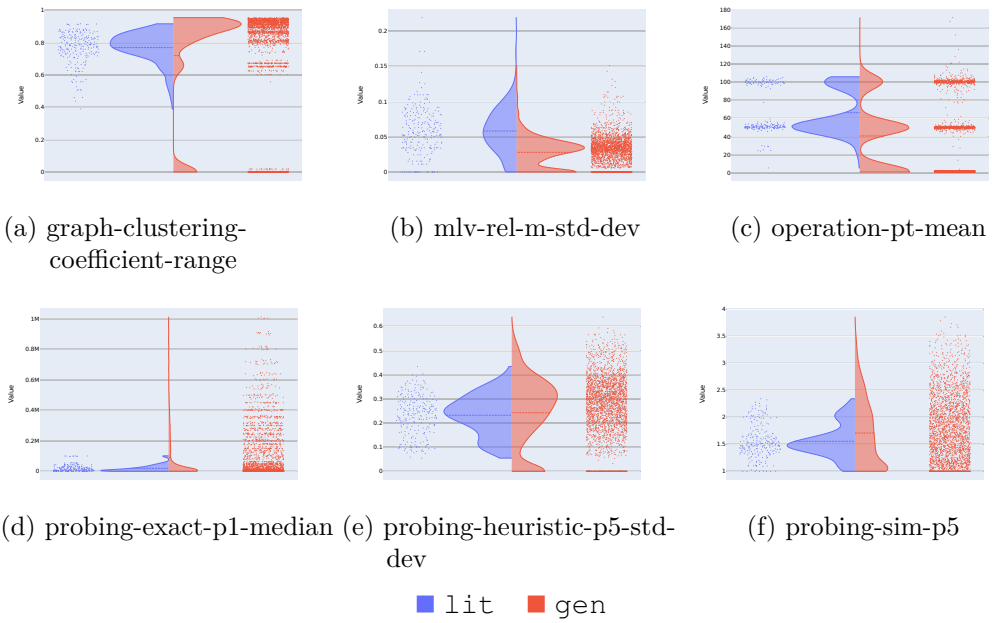


Figure 4.9: Pearson correlation coefficient for pairs of features grouped by clusters. Cluster starts are marked by the corresponding cluster number.

4.3.1 Comparison Between Existing and Generated Instances

Comparing the distribution of feature values in figure 4.10 adds further evidence to the hypothesis that `lit` is not a sufficient representation of the instance space as a whole. Broadly speaking they all follow the same general trend of `gen` covering a wider range of values than `lit`. However, there are four features that stand out and deserve particular consideration:

- *graph-clustering-coefficient-range* (figure 4.10a): The distributions appear to be reasonably similar, except for `gen` showing a very distinct second peak near 0. This peak is the result of the inclusion of trivial instances in `gen`, which are not present in `lit` and can, in a similar form, also be observed in figures 4.10b, 4.10e and 4.10f.
- *mlv-rel-m-std-dev* (figure 4.10b): The distribution provides an interesting contrast to the general trend, with `lit` actually having a wider range of values than `gen`. Its distribution over the projected instance space (figure 4.8b) also stands out as being relatively uniform, with only a few outliers having a value ≥ 0.1. Furthermore, all instances with a value ≥ 0.15 are from `lit`, specifically from `la`, `orb` and `ft`. Unfortunately the information as to how exactly these instances were generated appears to have been lost in time. One may however speculate that these instances were most likely generated by a fundamentally different kind of algorithm and thus contain a particularly unbalanced distribution of machines across operation slots.
- *operation-pt-mean* (figure 4.10c): There are two pronounced peaks for the mean processing time in `lit` with some outliers around them. The peaks correspond exactly to a mean processing time of 50 and 100.5 – the mean processing times for the instances from `ta` and `dmu`. For `gen` one can find a third peak corresponding to the instances from `gen-const` and `gen-nbinom` with significantly lower mean processing times. Furthermore, `gen` includes a larger number of outliers around the peaks due to simply including a larger number of instances.
- *probing-exact-p1-median* (figure 4.10d): For this feature, the dominance of `gen` over `lit` is most pronounced, with `gen` showing a vastly larger range of values, and a much flatter distribution, although with a heavy tail. The main reason for this discrepancy is, once more, the inclusion of larger instances in `gen`, which, by their very nature, are harder to solve, whereas the heavy tail can be explained by `gen-const` and `gen-nbinom`.



(a) graph-clustering-coefficient-range (b) mlv-rel-m-std-dev (c) operation-pt-mean

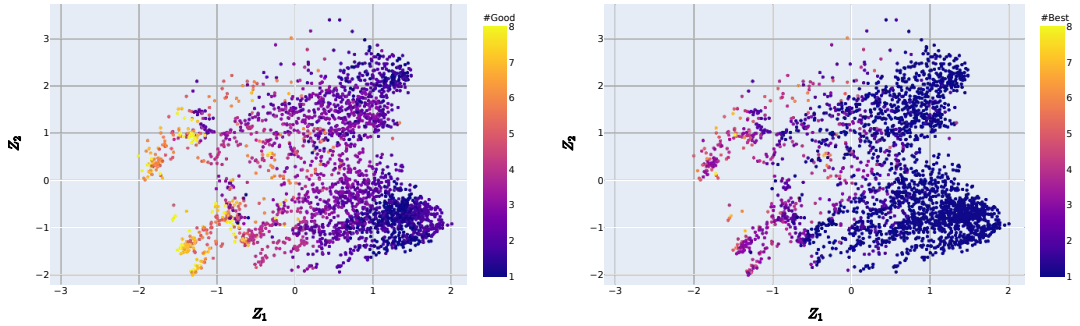
(d) probing-exact-p1-median (e) probing-heuristic-p5-std-dev (f) probing-sim-p5

Figure 4.10: Comparison of feature value distribution over instances from `lit` and `gen`

4.4 Performance Across the Instance Space

Before diving into the specifics of the individual algorithms, it is useful to first obtain a high-level picture of how the difficulty is distributed across the instance space. For this purpose, the number of good and best algorithms per instance is shown in figure 4.11a and figure 4.11b respectively. To obtain a better visualization, the trivial instances have been excluded since every algorithm is expected to, and in fact does, obtain the optimal solution on those instances, as one can see in figure 4.12. The distributions are relatively similar to the skewness as on would expect given the results of Streeter and Smith [SS06]. One can also clearly identify the areas in which (almost) all algorithms obtain good results around $Z_1 \approx -1.5$ and the two clusters that seem to be particularly difficult around $Z_1 \approx 1.5$, $Z_2 \approx -1$ and $Z_1 \approx 1$, $Z_2 \approx 2$.

The performance of all individual algorithms is visualized in figure 4.12 with a global color scale showing P_6. Figure 4.13 shows the same data, but with an adjusted color scale for each algorithm so as to better highlight the performance profile of individual algorithms. In consideration of the exact methods, it is striking that they all exhibit a similar pattern. They show consistently good performance, except for the previously identified cluster around $Z_1 \approx 1.5$, $Z_2 \approx -1$ where their performance is slightly degraded. This cluster consists primarily of instances from `gen-uniform` and `gen-binom` with a job to machine ratio close to 1, as can be seen in figure 4.4. Particularly the performance of `ort` is strongly degraded in this cluster, while `chu` performs only slightly worse. `cpo` on the other hand actually performs impressively well over pretty much all instances,



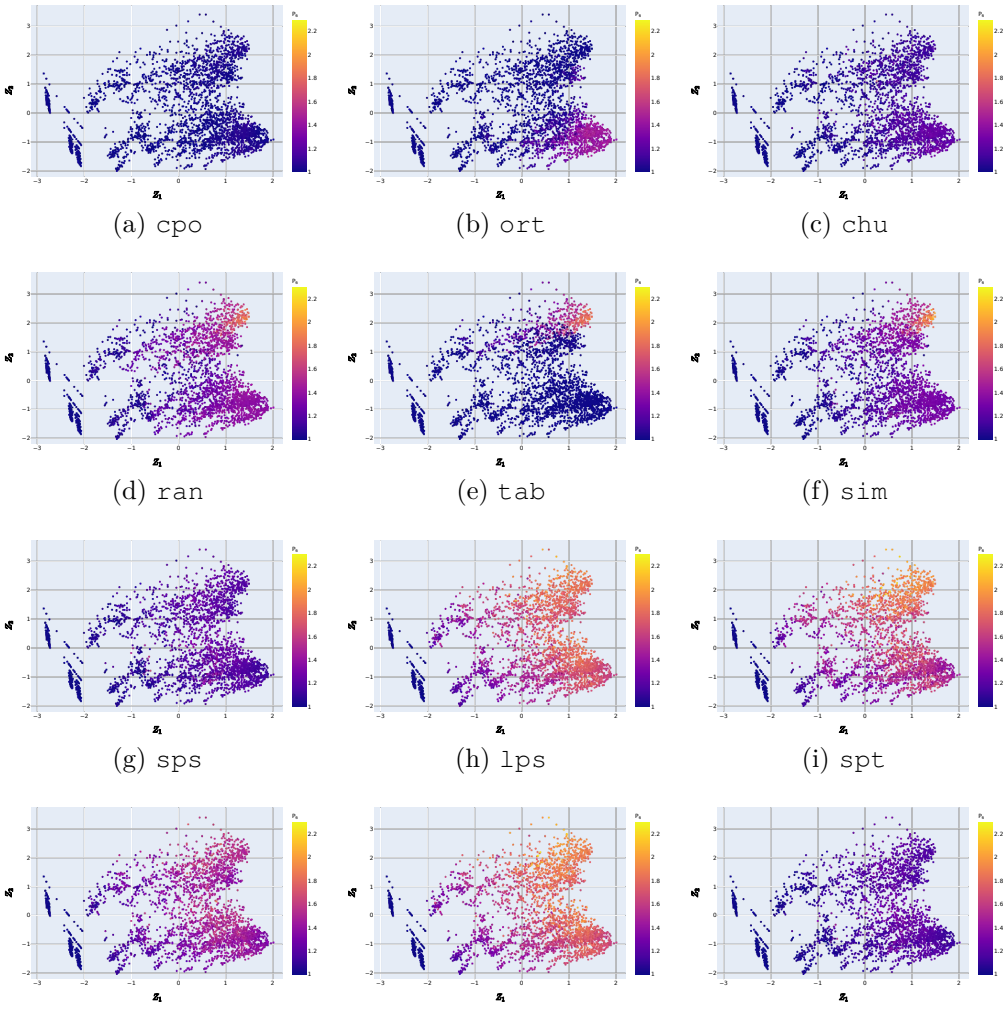
(a) By number of good algorithms (b) By number of best algorithms

Figure 4.11: Instance difficulty for projected instances from `all` (excluding trivial instances)

except for a minor degradation in the aforementioned area. When comparing the performance distribution of metaheuristics, they all show a very distinct cluster with degraded performance near $Z_1 \approx 1$, $Z_2 \approx 2$. This cluster is firmly within `gen-const` and has a job to machine ratio near 1.

Figure 4.14 shows the instances on which a given algorithm is the (unique) winner. As shown earlier in table 4.1, only `cpo`, `ort` and `tab` are the unique winners on distinct subsets of `all`. While `chu` is not the unique winner on any instance and does not have a lot of ties either, it makes up for this by performing decently on all instances, as can be seen in figure 4.12. `ort` on the other hand displays the opposite behavior – it performs very well on a fairly large subset of instances, more than `cpo` in fact, and usually wins on instances from `gen-const` or `gen-nbinom`, but performs significantly worse on the cluster identified earlier. Conversely `tab` clearly dominates the aforementioned cluster of instances from `gen-uniform` and `gen-binom`, but struggles with the instances from `gen-const` and `gen-nbinom`. `sim` achieves a few non-trivial ties, as does `ran`, and even `sps` and `mwr` manage to squeeze out some ties on the easier instances, although this can probably be attributed to chance. It is however notable that `sps` and `mwr` actually manage to each solve one of the benchmark instances from `lit` optimally (la06 and la10 respectively).

The binary view of instances where a given algorithm is good as shown in figure 4.15 further emphasizes trends that have, to some extent, already been pointed out in figure 4.12. `ort` is good on most of the dataset except for the one very distinct cluster. `cpo` stands out as showing good performance over almost all instances, while `tab` shows some weakness on the instances from `gen-const`.



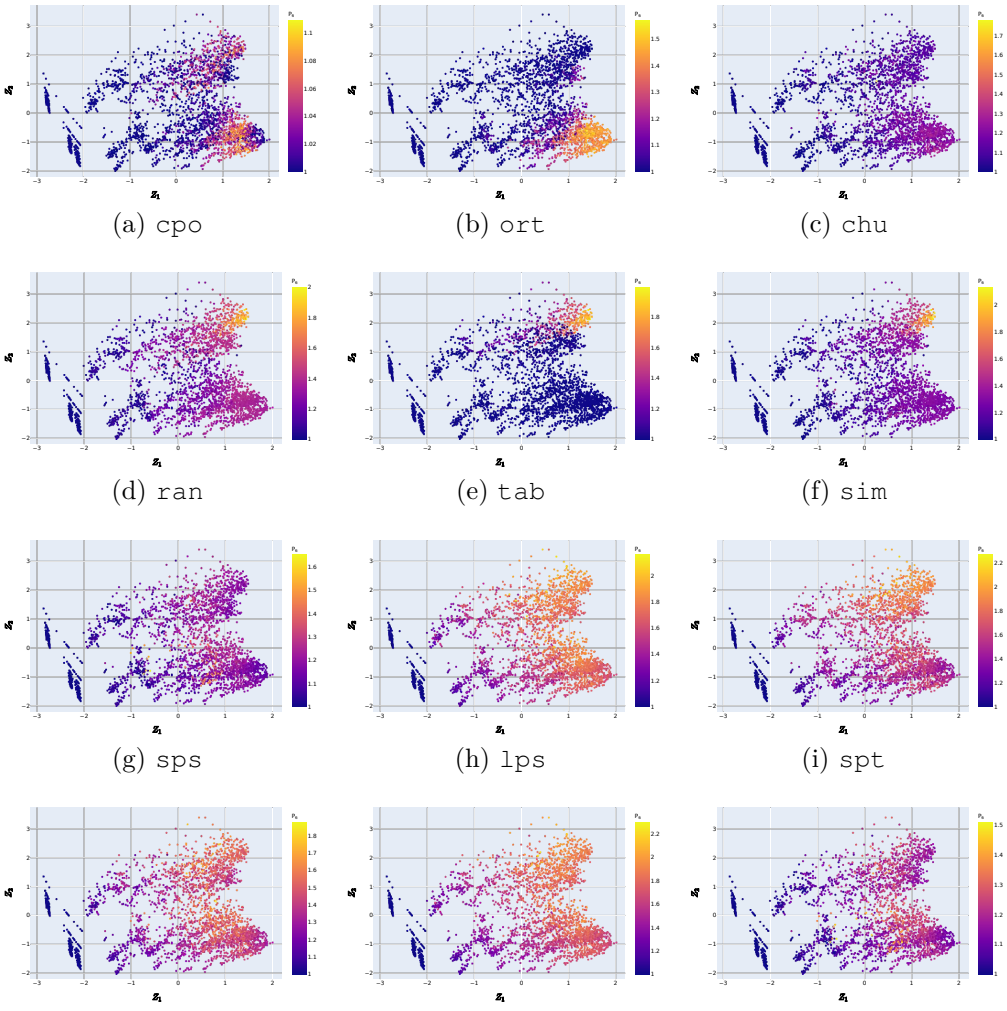
(a) cpo (b) ort (c) chu

(d) ran (e) tab (f) sim

(g) sps (h) lps (i) spt

(j) lpt (k) lwr (l) mwr

Figure 4.12: P_6 of algorithms on projected instances from all (global color scale)



(a) cpo (b) ort (c) chu

(d) ran (e) tab (f) sim

(g) sps (h) lps (i) spt

(j) lpt (k) lwr (l) mwr

Figure 4.13: P_6 of algorithms on projected instances from `all` (individual color scale)

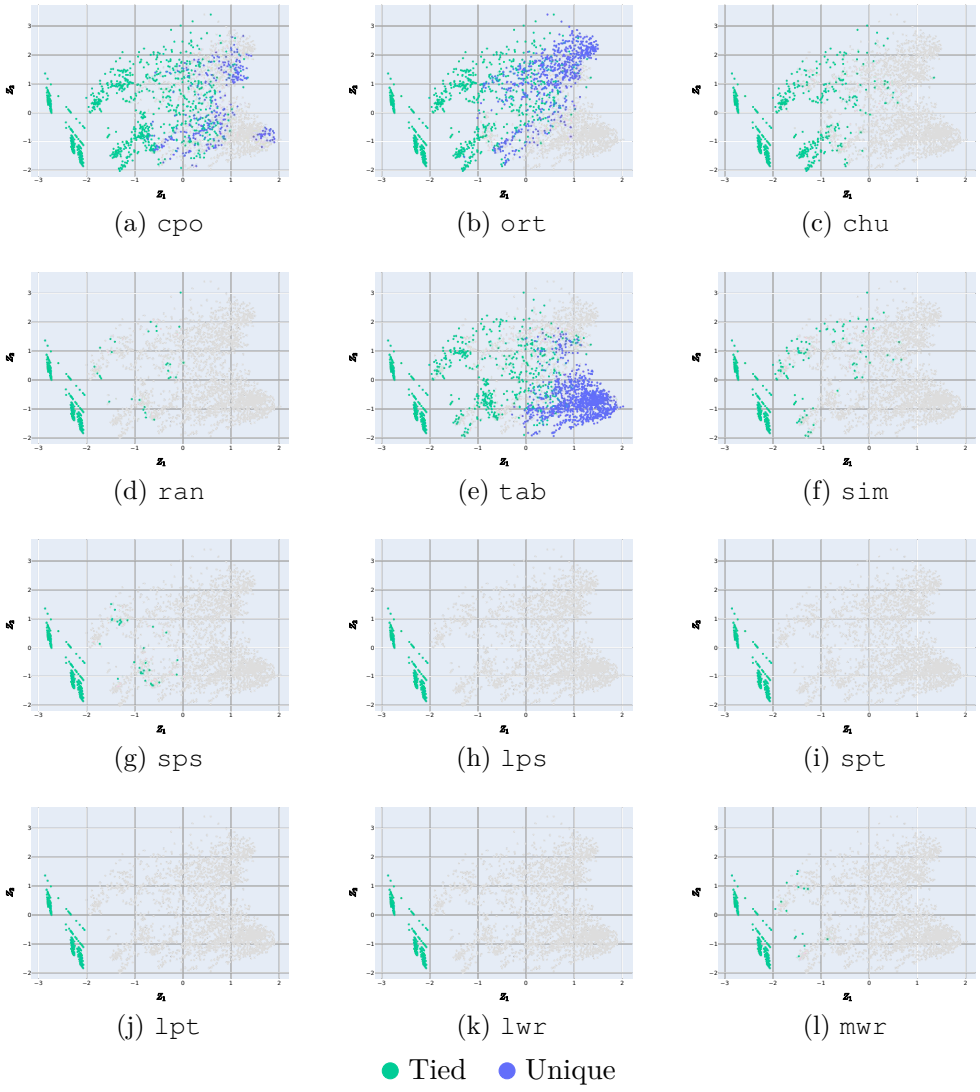
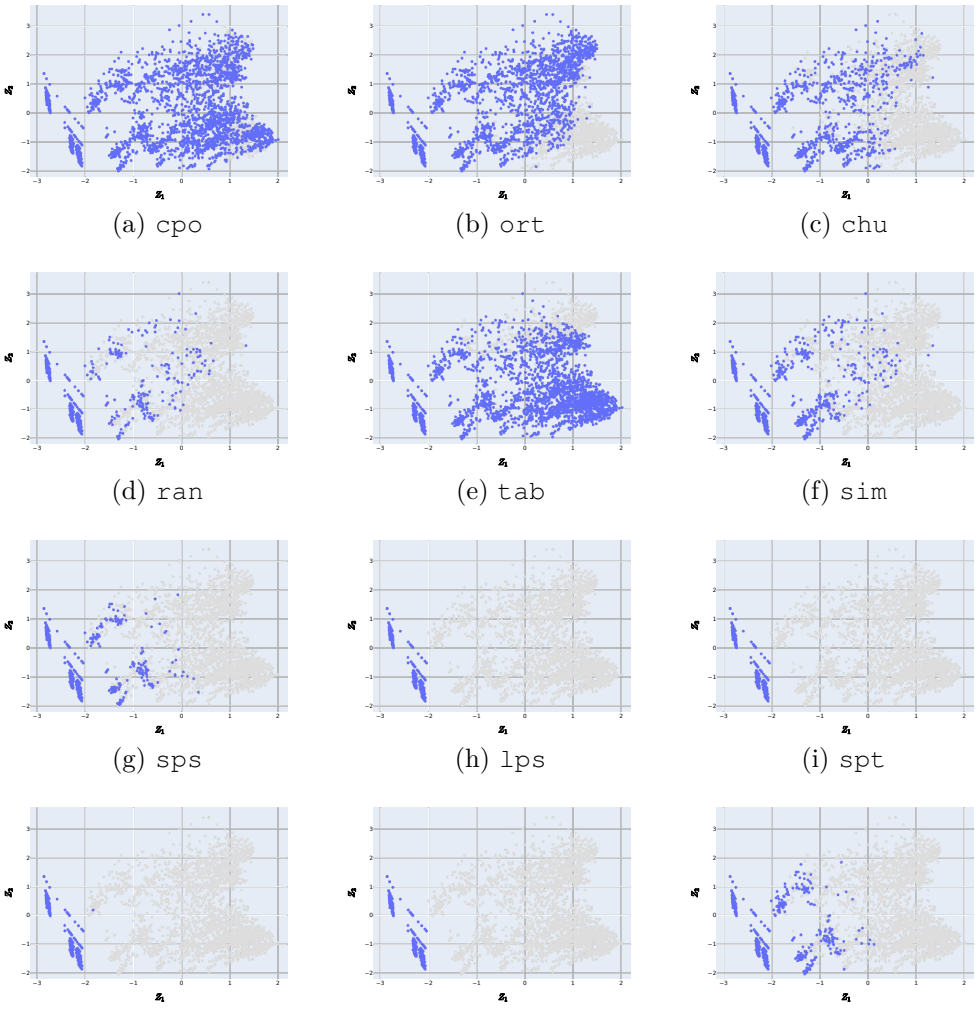


Figure 4.14: Projected instances on which each algorithm is the (unique) winner from `all`



(a) cpo (b) ort (c) chu

(d) ran (e) tab (f) sim

(g) sps (h) lps (i) spt

(j) lpt (k) lwr (l) mwr

Figure 4.15: Projected instances on which each algorithm is good from all

The analysis of individual algorithms' results suggests there is a difference in algorithm performance depending on the probability distribution the processing times are drawn from. This hypothesis is confirmed when considering the algorithms' performances for each subset of instances generated based on a different probability distribution in figure 4.16.
The exact methods all show better performance on `gen-const` and `gen-nbinom` with the effects being most pronounced for `ort` and `cpo` being least affected. Metaheuristics on the other hand struggle most with `gen-const` – most likely due to the neighborhood being ill-suited for this kind of problem. This suggests one of two possible reasons: Either the constant processing times change the topology of the search space in a way that makes it harder to find good solutions, or they make it easier for exact methods to find good solutions. One should also note that the cluster stands out particularly for `tab`, which shows impressively good performance on almost all other instances. Among the dispatching rule-based heuristics, most seem to suffer the inverse effects as the exact methods showing worse performance on `gen-const` and `gen-nbinom`, with the curious exception of `sps` and `mwr`, whose performance is mostly stable across all subsets.

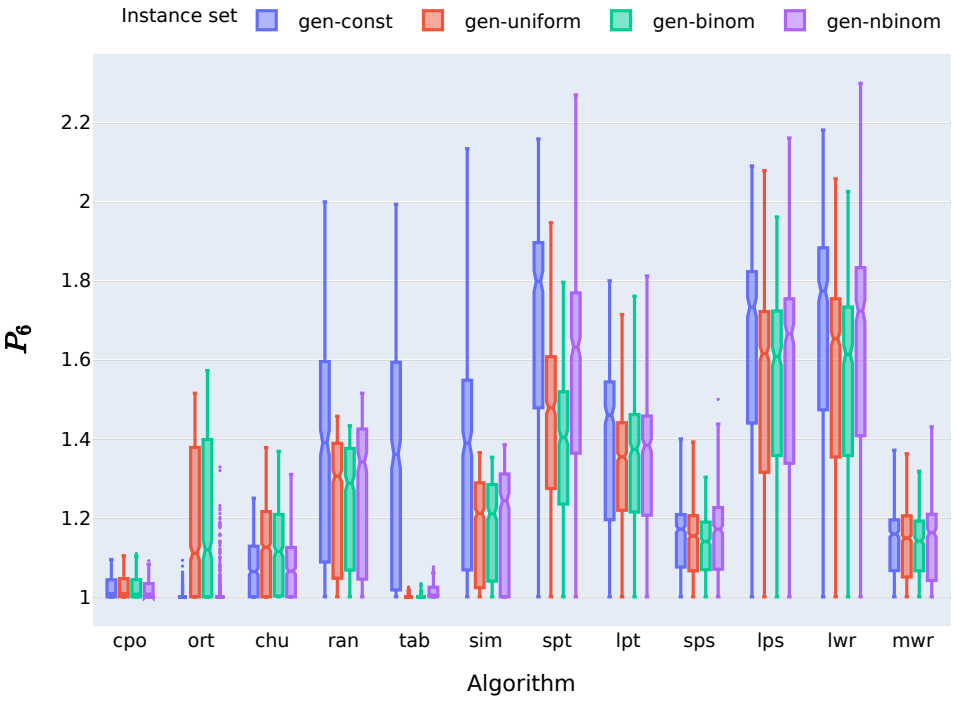
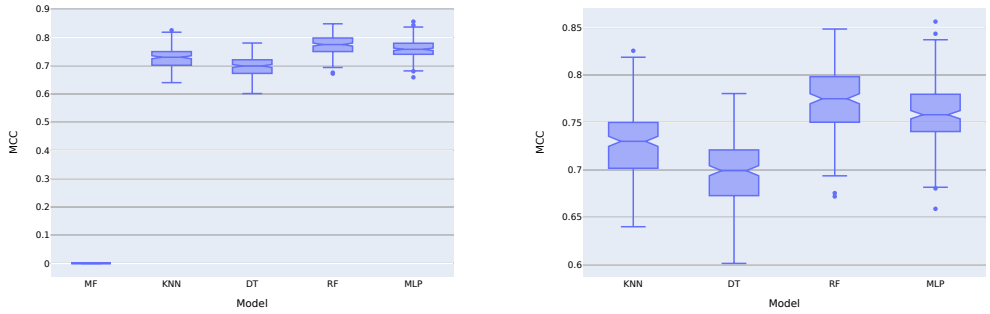


Figure 4.16: P_6 by algorithm on instances from subsets of `gen`

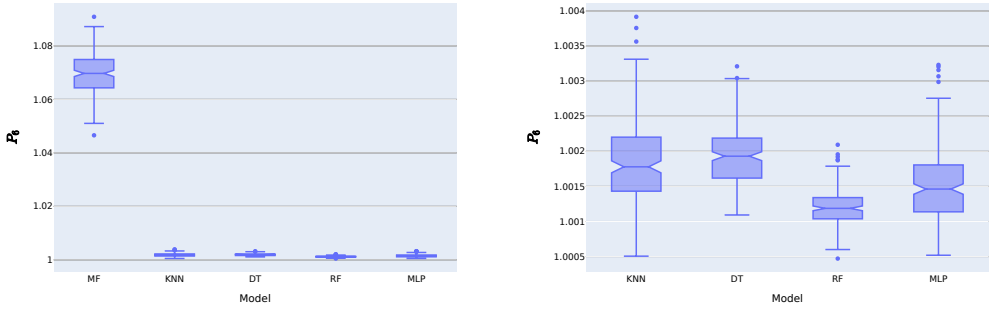
4.5 Algorithm Selection

All models are able to improve upon the prediction of the MF model (i.e. always predicting tab) to a significant extent, as can be seen in figures 4.17 and 4.18. The best model by both metrics turns out to be RF, with a median MCC of 0.7759 and a median P_6 of 1.0012. It should be noted that, even though the differences in model performance are statistically significant, the P_6 of the solver is near perfect for all machine learning models with the overall worst result being 1.0038 for MLP, whereas the best P_6 for MF is 1.0564. The statistical significance levels of the differences in MCC or solver P_6 over the 20×10 cross-validation splits can be seen in figure 4.19. For the most part, they simply confirm what one would expect based on the previous analysis: All machine learning models achieve a statistically significant improvement over the most frequent prediction of tab with the best being RF, which shows a statistically significant difference in performance to all other models in both MCC and P_6.



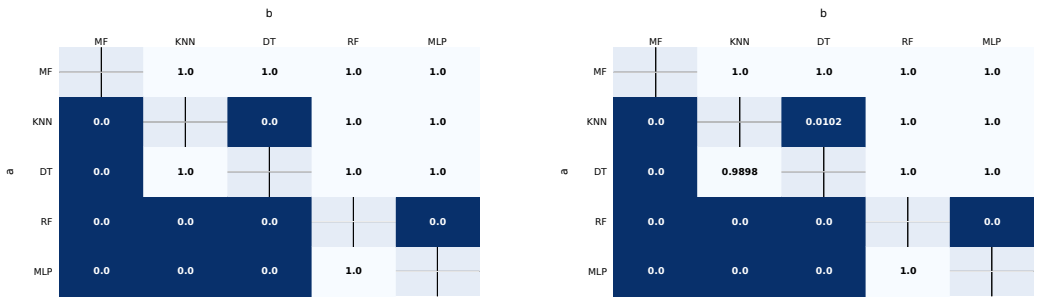
(a) Including MF (b) Excluding MF

Figure 4.17: MCC of machine learning models predicting the best algorithm for 20×10 cross-validation splits



(a) Including MF (b) Excluding MF

Figure 4.18: P_6 of solver induced by machine learning models predicting the best algorithm for 20×10 cross-validation splits



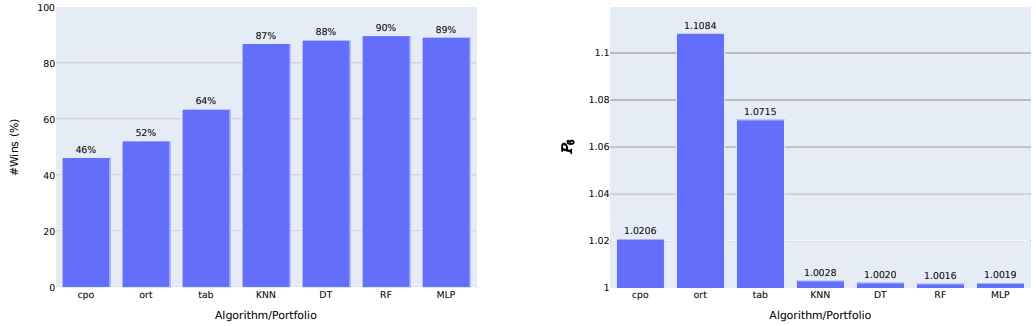
(a) MCC | one-sided, greater

(b) Solver P_6 | one-sided, less

Figure 4.19: p-values comparing performance measures of machine learning models predicting the best algorithm for 20×10 cross-validation splits

To compare the predictions with the actual algorithms' performances, the models are fitted on a stratified training set containing 50% of all instances and applied to generate predictions for the remaining 50% of instances. A comparison of the predictions' quality and the best algorithms' performances is displayed in figure 4.20. For both the number of wins (figure 4.20a) and the mean P_6 (figure 4.20b), the portfolio solvers show a clear improvement over any individual algorithm. The best solver by both measures (`RF`) is able to predict the best algorithm for 90% of the instances and thereby achieves a mean P_6 of 1.0016. Compared to the best individual algorithm by mean P_6 (`cpo`), the solver is able to obtain a 1.9% improvement in mean P_6.

For visualization purposes, the prediction of a `RF` model on 50% of `all` is shown in figure 4.21. Comparing the predictions to the areas where the various algorithms perform best (figure 4.14) shows the general trends found in the instance space analysis reflected in the predictions, i.e. `tab` dominates the bottom-right cluster, except for the very right where `cpo` shows better performance. The top-right area is firmly in the hands of `ort` and the center area, for which all algorithms show similar performance, is split up between `tab` and `ort`, while `cpo` is sprinkled all over the place. Trivial instances fall to `tab` as is to be expected since ties are broken in its favor.



(a) Relative number of wins (b) P_6

Figure 4.20: Performance of algorithms/portfolios on 50% of `all`

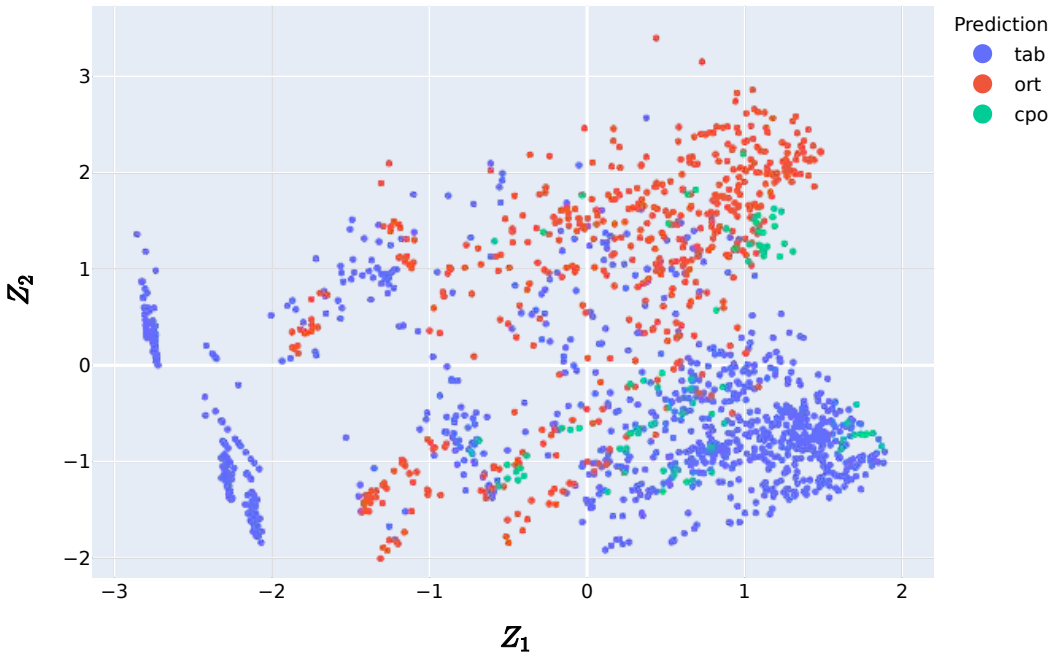


Figure 4.21: Predicted best algorithm by `RF` on 50% of `all`

4.5.1 Impact of Selected Features

The hypothesis that the selected probing features can be substituted by simpler features without excessive loss of information has already been posed in section 4.3. To test it, two alternate sets of features are generated:

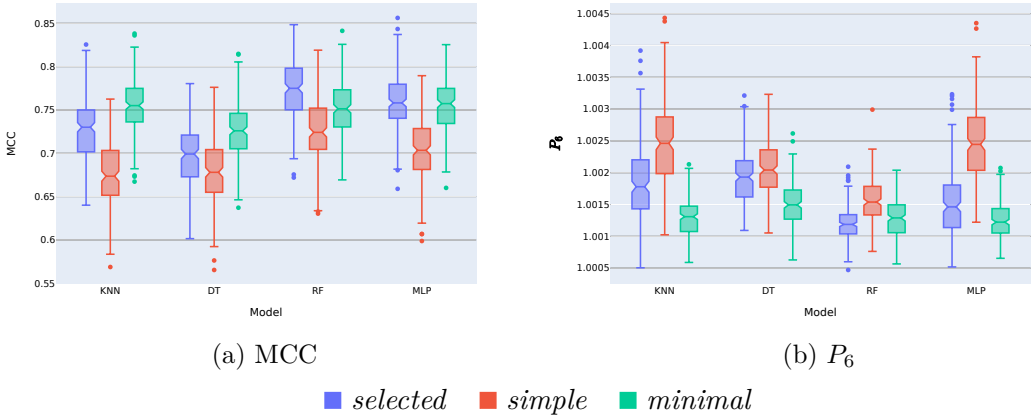
- *simple*: This set of features is generated by iteratively selecting the non-probing feature with the highest correlation from the same cluster for each of the selected features listed in section 4.3. If no non-probing feature is present in the cluster, the feature with the highest correlation over the entire dataset is selected. Features that have already been added to the new feature set are not considered again.

  Ultimately this produces the following set of features: *graph-clustering-coefficient-range*, *mlv-rel-m-std-dev*, *operation-length-mean*, *seq-cmax*, *mlu-range* and *mlu-q1*.

- *minimal*: This set is generated by the same process, however only features from the *Overall* group are permitted.

  The features are: *operations*, *seq-cmax*, *maxlb*, *jobs*, *machines* and *job-machine-ratio*.

The same machine learning models are trained on the alternate sets of features (after feature normalization) and evaluated on the same 20×10 cross-validation splits as for the original set of features, which shall be referred to as *selected*. The results can be seen in figure 4.22 and are very much in accordance with the initial hypothesis: The simplification of features does generally degrade performance, however the effect is relatively minor when considering the absolute differences and only pronounced for `RF`. Nonetheless the differences in both MCC and P_6 by *selected* over *simple* and *minimal* are, to a large part, statistically significant – in particular for `RF` as confirmed by the p-values in table 4.4 Conversely for the `KNN` and `DT` models the *minimal* feature set actually results in better performance than *selected*. Another curious observation is that the models based on *minimal* generally perform better than those using *simple*, although this effect may be explained by an unfortunate selection of features. It may thus be worth including probing features if one aims to obtain the best possible results, but they may be left out as long as a minor decrease in prediction quality is acceptable. However, the probing features are to be left out, one might as well remove the moderately more complicated features too and go straight for the simplest features possible, while still retaining very decent performance.



(a) MCC (b) P_6

Figure 4.22: Performance of machine learning models by feature set for 20×10 cross-validation splits

Model	P_6 *simple*	P_6 *minimal*	MCC *simple*	MCC *minimal*
DT	0.0009	1	0	1
KNN	0	1	0	1
MLP	0	1	0	0.0558
RF	0	0.0002	0	0

Table 4.4: p-values for difference in MCC (one-sided, greater) and P_6 (one-sided, less) of *selected* versus other feature sets for 20×10 cross-validation splits

CHAPTER 5

Conclusion

This thesis presents a broad analysis of the instance space of the job shop scheduling problem to further the current understanding of the problem, pave the way towards a better setting for comparing the performance of algorithms and show the usefulness of automated algorithm selection. The following can be considered to be the main contributions:

- Based on an analysis of the existing instances in the literature, a novel set of benchmark instances has been created. The generated instance set contains 4225 instances of various sizes and processing time distributions, from which 3025 were included for this analysis. They are uniformly distributed with regard to the number of jobs and the number of machines and cover a significantly larger range of variations than the existing benchmark instances. Of particular note are the sets with processing times drawn from a constant or negative binomial distribution, which have so far not been found in the literature. However, even if one were to limit oneself to uniformly distributed processing times, as has commonly been the case in the literature, the new instances still cover a far wider range of instance sizes. While there can be no certainty as to whether any set of instances truly covers all possibilities, this extended set of instances is a step towards benchmarking data that is more representative of the instance space as a whole.
- To characterize the instances, a set of features, 604 in total, was created, which includes features found in the literature as well as additional probing and graph-based features, which were proposed in this thesis. Among these features, several clusters of highly similar features could be identified. An analysis of the features' importance showed probing features to be among the most useful, with their inclusion resulting in a statistically significant improvement in algorithm selection performance over similar sets of simpler features. It has also been shown that the probing features could be replaced by very simple features with only a mild

degradation in performance – a sacrifice one may very well be willing to make in practice.

- In general, the results by Streeter and Smith [SS06] have been confirmed insofar that the difficulty of JSSP instances is, apart from the obvious factor of instance size, strongly influenced by the job to machine ratio. However the analysis also shows clear differences in algorithm performance on specific areas of the instance space. In particular the best-performing algorithms (CP Optimizer, tabu search and OR-Tools) each have their own distinct subset of instances on which they excel. Overall CP Optimizer achieved the best mean and worst-case performance while tabu search took the lead with regard to the number of instances for which it achieved the best solution.

  This result might come as a surprise since one would generally expect the performance of exact methods to degrade on larger instances, while metaheuristics are usually less affected. No such patterns were observed for instances up to 100×100, however no statements can be made about the behavior on even larger instances. There did however emerge one major difference between exact methods and metaheuristics with regard to the distribution of processing times: While exact methods perform better on instances with processing times drawn from a constant or negative binomial distribution, the metaheuristics perform rather poorly on those instances, instead excelling on instances with processing times drawn from uniform or binomial distributions. Particularly for constant processing times this effect is present for all metaheuristics, suggesting the neighborhood is ill-suited for this kind of problem.

- Predicting the best algorithm for any given instance using machine learning models resulted in a statistically significant improvement of performance over any individual algorithm's performance. When trained on 50% of the instances and evaluated on the remaining 50%, the solver based on the best model, a Random Forest, was able to obtain the best solution for 90% of the instances, whereas the best individual algorithm only obtained the best solution on 64%.

Besides the aforementioned contributions, the comparison of different performance measures and their respective advantages and drawbacks is also worth mentioning and may prove useful for similar experiments, although a systematic analysis with a larger set of instances would be preferable before making any definite statements.

The work done in this thesis also opens up a wide variety of avenues for further exploration. This includes extending the set of benchmark instances even further with real-world instances, instances with processing times drawn from different probability distributions, or instances using biased shuffle algorithms. There is also a significant amount of work to be done with regard to understanding why algorithms exhibit certain performance patterns and how they could be improved. It is to be hoped that this thesis will serve as a solid foundation for further research in this direction.

List of Figures

2.1 Gantt chart for the schedule given in listing 2.1 7
2.2 Oriented disjunctive graph for the instance and schedule given in listing 2.1 7
2.3 Instance space analysis framework from [SM+14] 9

3.1 Number of instances by jobs and machines 14
3.2 R^* versus performance measure for results of all algorithms on instances with known optimum 31

4.1 P_6 by algorithm for instances from `all` 41
4.2 P_6 by algorithm for instances from `lit` and `gen` 42
4.3 p-values comparing P_6 of algorithms | one-sided, less 43
4.4 Basic features of projected instances from `all` 44
4.5 Source of projected instances from `all` 44
4.6 Mean and standard deviation of permutation importance of selected features 45
4.7 Pearson correlation coefficient for pairs of selected features 45
4.8 Feature values of projected instances from `all` 47
4.9 Pearson correlation coefficient for pairs of features grouped by clusters . . 48
4.10 Comparison of feature value distribution over instances from `lit` and `gen` 50
4.11 Instance difficulty for projected instances from `all` 51
4.12 P_6 of algorithms on projected instances from `all` (global color scale) . . . 52
4.13 P_6 of algorithms on projected instances from `all` (individual color scale) 53
4.14 Projected instances on which each algorithm is the (unique) winner from `all` 54
4.15 Projected instances on which each algorithm is good from `all` 55
4.16 P_6 by algorithm on instances from subsets of `gen` 56
4.17 MCC of machine learning models predicting the best algorithm 57
4.18 P_6 of solver induced by machine learning models predicting the best algorithm 57
4.19 p-values comparing performance measures of machine learning models predicting the best algorithm 58
4.20 Performance of algorithms/portfolios on 50% of `all` 59
4.21 Predicted best algorithm by `RF` on 50% of `all` 59
4.22 Performance of machine learning models by feature set 61

List of Tables

3.1 Existing sets of benchmark instances 12
3.2 Dispatching rules 20
3.3 Relevant properties of performance measures 30
3.4 Statistical key figures 32
3.5 Instance features 35

4.1 Number of (unique) wins by algorithm on instances from `all` 40
4.2 Statistical key figures for aggregated P_6 by algorithm over instances from `all` 41
4.3 Selected features with cluster number and projection coefficients 45
4.4 p-values for difference in MCC (one-sided, greater) and P_6 (one-sided, less) of *selected* versus other feature sets 61

A.1 Feature clusters 71
A.2 Instances from `lit` with known optimal solution 72

Appendix A

Cluster	Features
0	mlu-max, mlu-mean, mlu-median, mlu-min, mlu-q1, mlu-q3, mlv-range, mlv-std-dev, operations, os-mm-range, os-mm-std-dev, os-rm-range, os-rm-std-dev, probing-all-p4-max, probing-all-p4-mean, probing-all-p4-median, probing-all-p4-min, probing-all-p4-q1, probing-all-p4-q3, probing-all-p4-range, probing-all-p4-std-dev, probing-all-p5-max, probing-all-p5-mean, probing-all-p5-q3, probing-all-p5-range, probing-all-p5-std-dev, probing-chu-p4, probing-chu-p5, probing-cpl-p4, probing-cpl-p5, probing-cpo-p4, probing-cpo-p5, probing-exact-p4-max, probing-exact-p4-mean, probing-exact-p4-median, probing-exact-p4-min, probing-exact-p4-q1, probing-exact-p4-q3, probing-exact-p5-max, probing-exact-p5-mean, probing-exact-p5-median, probing-exact-p5-min, probing-exact-p5-q1, probing-exact-p5-q3, probing-heuristic-p4-max, probing-heuristic-p4-mean, probing-heuristic-p4-median, probing-heuristic-p4-min, probing-heuristic-p4-q1, probing-heuristic-p4-q3, probing-heuristic-p4-range, probing-heuristic-p4-std-dev, probing-metaheuristic-p4-max, probing-metaheuristic-p4-mean, probing-metaheuristic-p4-median, probing-metaheuristic-p4-min, probing-metaheuristic-p4-q1, probing-metaheuristic-p4-q3, probing-metaheuristic-p4-range, probing-metaheuristic-p4-std-dev, probing-ort-p4, probing-ort-p5, probing-priority-lps-p4, probing-priority-lpt-p4, probing-priority-lwr-p4, probing-priority-mwr-p4, probing-priority-sps-p4, probing-priority-spt-p4, probing-ran-p4, probing-sim-p4, probing-tab-p4
1	graph-clustering-coefficient-range, mlu-range, mlu-std-dev, probing-all-p1-gini, probing-all-p2-gini, probing-all-p2-range, probing-all-p2-std-dev, probing-all-p3-gini, probing-all-p4-gini, probing-all-p5-gini, probing-heuristic-p1-gini, probing-heuristic-p2-gini, probing-heuristic-p3-gini, probing-heuristic-p4-gini, probing-heuristic-p5-gini, probing-heuristic-p5-range, probing-heuristic-p5-std-dev
2	job-pt-max, job-pt-mean, job-pt-median, job-pt-min, job-pt-q1, job-pt-q3, job-pt-range, job-pt-std-dev, machine-pt-range, machine-pt-std-dev, maxlb, operation-slot-operation-pt-range, operation-slot-operation-pt-std-dev, os-rm-pt-range, os-rm-pt-std-dev, probing-all-p1-max, probing-all-p1-mean, probing-all-p1-median, probing-all-p1-min, probing-all-p1-q1, probing-all-p1-q3, probing-all-p1-range, probing-all-p1-std-dev, probing-chu-p1, probing-cpl-p1, probing-cpo-p1, probing-exact-p1-max, probing-exact-p1-mean, probing-exact-p1-median, probing-exact-p1-min, probing-exact-p1-q1, probing-exact-p1-q3, probing-heuristic-p1-max, probing-heuristic-p1-mean, probing-heuristic-p1-median, probing-heuristic-p1-min, probing-heuristic-p1-q1, probing-heuristic-p1-q3, probing-heuristic-p1-range, probing-heuristic-p1-std-dev, probing-metaheuristic-p1-max, probing-metaheuristic-p1-mean, probing-metaheuristic-p1-median, probing-metaheuristic-p1-min, probing-metaheuristic-p1-q1, probing-metaheuristic-p1-q3, probing-metaheuristic-p1-range, probing-metaheuristic-p1-std-dev, probing-ort-p1, probing-priority-lps-p1, probing-priority-lpt-p1, probing-priority-lwr-p1, probing-priority-mwr-p1, probing-priority-sps-p1, probing-priority-spt-p1, probing-ran-p1, probing-sim-p1, probing-tab-p1, seq-p1
3	graph-clustering-coefficient-min, graph-density, job-machine-ratio, os-rm-max, os-rm-mean, os-rm-median, os-rm-min, os-rm-pt-rel-m-op-mean-max, os-rm-pt-rel-m-op-mean-mean, os-rm-pt-rel-m-op-mean-median, os-rm-pt-rel-m-op-mean-min, os-rm-pt-rel-m-op-mean-q1, os-rm-pt-rel-m-op-mean-q3, os-rm-q1, os-rm-q3, os-rm-rel-m-max, os-rm-rel-m-max, os-rm-rel-m-mean, os-rm-rel-m-mean, os-rm-rel-m-median, os-rm-rel-m-median, os-rm-rel-m-min, os-rm-rel-m-min, os-rm-rel-m-q1, os-rm-rel-m-q1, os-rm-rel-m-q3, os-rm-rel-m-q3, os-rma-pt-rel-m-op-mean-max, os-rma-pt-rel-m-op-mean-mean, os-rma-pt-rel-m-op-mean-median, os-rma-pt-rel-m-op-mean-min, os-rma-pt-rel-m-op-mean-q1, os-rma-pt-rel-m-op-mean-q3
4	mlv-rel-mean-q3, mlva-rel-mean-q3, os-mm-rel-mean-q3
5	os-rm-gini, os-rm-gini, os-rm-pt-gini, os-rm-pt-rel-m-gini, os-rm-pt-rel-m-op-mean-gini, os-rm-pt-rel-mean-gini, os-rm-pt-rel-mean-max, os-rm-pt-rel-mean-q3, os-rm-pt-rel-mean-range, os-rm-pt-rel-mean-std-dev, os-rm-rel-m-gini, os-rm-rel-m-gini, os-rm-rel-mean-gini, os-rm-rel-mean-gini, os-rm-rel-mean-max, os-rm-rel-mean-max, os-rm-rel-mean-q3, os-rm-rel-mean-q3, os-rm-rel-mean-range, os-rm-rel-mean-range, os-rm-rel-mean-std-dev, os-rm-rel-mean-std-dev, os-rma-pt-gini, os-rma-pt-rel-m-gini, os-rma-pt-rel-m-op-mean-gini, os-rma-pt-rel-mean-gini, os-rma-pt-rel-mean-max, os-rma-pt-rel-mean-q3, os-rma-pt-rel-mean-range, os-rma-pt-rel-mean-std-dev
6	mlu-rel-m-max, mlu-rel-m-mean, mlu-rel-m-median, mlu-rel-m-min, mlu-rel-m-q1, mlu-rel-m-q3, mlu-rel-m-range, mlu-rel-m-std-dev, os-rm-range, os-rm-std-dev

Cluster	Features
7	probing-all-p2-q3, probing-chu-p2, probing-exact-p2-q1, probing-exact-p3ean, probing-exact-p3edian, probing-exact-p3in, probing-ort-p2
8	graph-vertex-degree-std-dev, probing-all-p2-q1, probing-all-p3ean, probing-all-p3edian, probing-all-p3in, probing-heuristic-p2-q1, probing-heuristic-p2-q3, probing-heuristic-p3ax, probing-heuristic-p3ean, probing-heuristic-p3edian, probing-heuristic-p3in, probing-metaheuristic-p2-q1, probing-metaheuristic-p2-q3, probing-metaheuristic-p3ax, probing-metaheuristic-p3ean, probing-metaheuristic-p3edian, probing-metaheuristic-p3in, probing-priority-lps-p2, probing-priority-lpt-p2, probing-priority-lwr-p2, probing-priority-mwr-p2, probing-priority-sps-p2, probing-priority-spt-p2, probing-ran-p2, probing-sim-p2, probing-tab-p2, skewness
9	operation-pt-max, operation-pt-mean, operation-pt-median, operation-pt-q1, operation-pt-q3, operation-pt-range, operation-pt-std-dev
10	job-pt-rel-op-mean-max, job-pt-rel-op-mean-mean, job-pt-rel-op-mean-median, job-pt-rel-op-mean-min, job-pt-rel-op-mean-q1, job-pt-rel-op-mean-q3, machines, probing-all-p3-max, probing-all-p3-range, probing-all-p3-std-dev, probing-cpl-p3, probing-cpo-p3, probing-exact-p3-max, probing-exact-p3-q3
11	probing-all-p5-median, probing-all-p5-min, probing-all-p5-q1, probing-heuristic-p5-max, probing-heuristic-p5-mean, probing-heuristic-p5-median, probing-heuristic-p5-min, probing-heuristic-p5-q1, probing-heuristic-p5-q3, probing-metaheuristic-p5-max, probing-metaheuristic-p5-mean, probing-metaheuristic-p5-median, probing-metaheuristic-p5-min, probing-metaheuristic-p5-q1, probing-metaheuristic-p5-q3, probing-priority-lps-p5, probing-priority-lpt-p5, probing-priority-lwr-p5, probing-priority-mwr-p5, probing-priority-sps-p5, probing-priority-spt-p5, probing-ran-p5, probing-sim-p5, probing-tab-p5
12	probing-all-p3-q3, probing-chu-p3, probing-exact-p3-mean, probing-exact-p3-median, probing-exact-p3-min, probing-exact-p3-q1, probing-ort-p3
13	job-pt-gini, job-pt-rel-mean-gini, job-pt-rel-mean-max, job-pt-rel-mean-q3, job-pt-rel-mean-range, job-pt-rel-mean-std-dev, job-pt-rel-op-mean-gini
14	job-pt-rel-op-mean-range, job-pt-rel-op-mean-std-dev, machine-pt-rel-op-mean-range, machine-pt-rel-op-mean-std-dev, operation-slot-operation-pt-rel-op-mean-range, operation-slot-operation-pt-rel-op-mean-std-dev
15	graph-vertex-degree-max, graph-vertex-degree-mean, graph-vertex-degree-median, graph-vertex-degree-min, graph-vertex-degree-q1, graph-vertex-degree-q3, jobs, machine-pt-rel-op-mean-max, machine-pt-rel-op-mean-mean, machine-pt-rel-op-mean-median, machine-pt-rel-op-mean-min, machine-pt-rel-op-mean-q1, machine-pt-rel-op-mean-q3, operation-slot-operation-pt-rel-op-mean-max, operation-slot-operation-pt-rel-op-mean-mean, operation-slot-operation-pt-rel-op-mean-median, operation-slot-operation-pt-rel-op-mean-min, operation-slot-operation-pt-rel-op-mean-q1, operation-slot-operation-pt-rel-op-mean-q3, os-rm-max, os-rm-mean, os-rm-median, os-rm-min, os-rm-q1, os-rm-q3
16	mlv-rel-mean-min, mlv-rel-mean-q1, mlva-rel-mean-min, mlva-rel-mean-q1, os-mm-rel-mean-min, os-mm-rel-mean-q1
17	mlv-max, mlv-mean, mlv-median, mlv-min, mlv-q1, mlv-q3, mlv-rel-m-max, mlv-rel-m-mean, mlv-rel-m-median, mlv-rel-m-min, mlv-rel-m-q1, mlv-rel-m-q3, os-mm-max, os-mm-mean, os-mm-median, os-mm-min, os-mm-q1, os-mm-q3, os-mm-rel-m-max, os-mm-rel-m-mean, os-mm-rel-m-median, os-mm-rel-m-min, os-mm-rel-m-q1, os-mm-rel-m-q3
18	os-rm-pt-rel-mean-min, os-rm-pt-rel-mean-q1, os-rm-rel-mean-min, os-rm-rel-mean-min, os-rm-rel-mean-q1, os-rm-rel-mean-q1, os-rma-pt-rel-mean-min, os-rma-pt-rel-mean-q1
19	mlu-gini, mlu-rel-m-gini, mlu-rel-mean-gini, mlu-rel-mean-max, mlu-rel-mean-q3, mlu-rel-mean-range, mlu-rel-mean-std-dev
20	mlva-max, mlva-mean, mlva-median, mlva-min, mlva-q1, mlva-q3, mlva-range, mlva-rel-m-max, mlva-rel-m-mean, mlva-rel-m-median, mlva-rel-m-min, mlva-rel-m-q1, mlva-rel-m-q3, mlva-rel-m-range, mlva-rel-m-std-dev, mlva-std-dev, probing-all-p3-mean, probing-all-p3-median, probing-all-p3-min, probing-all-p3-q1, probing-heuristic-p3-max, probing-heuristic-p3-mean, probing-heuristic-p3-median, probing-heuristic-p3-min, probing-heuristic-p3-q1, probing-heuristic-p3-q3, probing-metaheuristic-p3-max, probing-metaheuristic-p3-mean, probing-metaheuristic-p3-median, probing-metaheuristic-p3-min, probing-metaheuristic-p3-q1, probing-metaheuristic-p3-q3, probing-priority-lps-p3, probing-priority-lpt-p3, probing-priority-lwr-p3, probing-priority-mwr-p3, probing-priority-sps-p3, probing-priority-spt-p3, probing-ran-p3, probing-sim-p3, probing-tab-p3
21	job-pt-rel-mean-min, job-pt-rel-mean-q1
22	machine-pt-max, machine-pt-mean, machine-pt-median, machine-pt-min, machine-pt-q1, machine-pt-q3, operation-slot-operation-pt-max, operation-slot-operation-pt-mean, operation-slot-operation-pt-median, operation-slot-operation-pt-min, operation-slot-operation-pt-q1, operation-slot-operation-pt-q3, os-rm-pt-max, os-rm-pt-mean, os-rm-pt-median, os-rm-pt-min, os-rm-pt-q1, os-rm-pt-q3
23	machine-pt-rel-mean-min, machine-pt-rel-mean-q1, operation-slot-operation-pt-rel-mean-min, operation-slot-operation-pt-rel-mean-q1
24	mlv-rel-m-range, mlv-rel-m-std-dev, os-mm-rel-m-range, os-mm-rel-m-std-dev, os-rm-rel-m-range, os-rm-rel-m-std-dev
25	probing-cpo-p2, probing-exact-p2-q3
26	graph-betweenness-centrality-max, graph-betweenness-centrality-mean, graph-betweenness-centrality-median, graph-betweenness-centrality-q1, graph-betweenness-centrality-q3, graph-betweenness-centrality-range

Cluster	Features
27	probing-exact-p1-range, probing-exact-p1-std-dev
28	probing-exact-p4-range, probing-exact-p4-std-dev, probing-exact-p5-range, probing-exact-p5-std-dev
29	os-rm-pt-rel-mean-median, os-rm-rel-mean-median, os-rm-rel-mean-median, os-rma-pt-rel-mean-median
30	probing-metaheuristic-p1-gini, probing-metaheuristic-p2-gini, probing-metaheuristic-p3-gini, probing-metaheuristic-p4-gini, probing-metaheuristic-p5-gini, probing-metaheuristic-p5-range, probing-metaheuristic-p5-std-dev
31	job-pt-rel-mean-median
32	os-rm-pt-rel-m-range, os-rm-pt-rel-m-std-dev, os-rma-pt-range, os-rma-pt-rel-m-range, os-rma-pt-rel-m-std-dev, os-rma-pt-std-dev
33	machine-pt-gini, machine-pt-rel-mean-gini, machine-pt-rel-mean-max, machine-pt-rel-mean-q3, machine-pt-rel-mean-range, machine-pt-rel-mean-std-dev, machine-pt-rel-op-mean-gini, operation-slot-operation-pt-gini, operation-slot-operation-pt-rel-mean-gini, operation-slot-operation-pt-rel-mean-max, operation-slot-operation-pt-rel-mean-q3, operation-slot-operation-pt-rel-mean-range, operation-slot-operation-pt-rel-mean-std-dev, operation-slot-operation-pt-rel-op-mean-gini
34	mlu-rel-mean-q1
35	operation-pt-min
36	mlv-rel-mean-median, mlva-rel-mean-median, os-mm-rel-mean-median
37	graph-betweenness-centrality-gini
38	graph-clustering-coefficient-max, graph-clustering-coefficient-mean, graph-clustering-coefficient-median, graph-clustering-coefficient-q1, graph-clustering-coefficient-q3
39	machine-pt-rel-mean-median, operation-slot-operation-pt-rel-mean-median
40	graph-betweenness-centrality-min, graph-clustering-coefficient-gini, graph-clustering-coefficient-std-dev, probing-heuristic-p2-range, probing-heuristic-p2-std-dev
41	mlv-gini, mlv-rel-m-gini, mlv-rel-mean-gini, mlv-rel-mean-max, mlv-rel-mean-range, mlv-rel-mean-std-dev, mlva-gini, mlva-rel-m-gini, mlva-rel-mean-gini, mlva-rel-mean-max, mlva-rel-mean-range, mlva-rel-mean-std-dev, os-mm-gini, os-mm-rel-m-gini, os-mm-rel-mean-gini, os-mm-rel-mean-max, os-mm-rel-mean-range, os-mm-rel-mean-std-dev
42	probing-metaheuristic-p3-range, probing-metaheuristic-p3-std-dev
43	probing-metaheuristic-p2-range, probing-metaheuristic-p2-std-dev
44	probing-exact-p1-gini, probing-exact-p2-gini, probing-exact-p2-range, probing-exact-p2-std-dev, probing-exact-p3-gini, probing-exact-p4-gini, probing-exact-p5-gini
45	mlu-rel-mean-min
46	os-rm-pt-rel-m-op-mean-range, os-rm-pt-rel-m-op-mean-std-dev
47	operation-pt-gini
48	graph-betweenness-centrality-std-dev, graph-vertex-degree-gini
49	os-rm-pt-rel-m-max, os-rm-pt-rel-m-mean, os-rm-pt-rel-m-median, os-rm-pt-rel-m-min, os-rm-pt-rel-m-q1, os-rm-pt-rel-m-q3, os-rma-pt-max, os-rma-pt-mean, os-rma-pt-median, os-rma-pt-min, os-rma-pt-q1, os-rma-pt-q3, os-rma-pt-rel-m-max, os-rma-pt-rel-m-mean, os-rma-pt-rel-m-median, os-rma-pt-rel-m-min, os-rma-pt-rel-m-q1, os-rma-pt-rel-m-q3
50	probing-heuristic-p3-range, probing-heuristic-p3-std-dev
51	mlu-rel-mean-median
52	probing-exact-p3-range, probing-exact-p3-std-dev
53	probing-all-p3ax, probing-cpl-p2, probing-exact-p3ax
54	os-rm-rel-m-range, os-rm-rel-m-std-dev, os-rma-pt-rel-m-op-mean-range, os-rma-pt-rel-m-op-mean-std-dev

Table A.1: Feature clusters

Instance	n	m	C^*_{max}
abz5	10	10	1,234
abz6	10	10	943
abz7	20	15	656
abz9	20	15	678
dmu03	20	15	2,731
dmu05	20	15	2,749
dmu13	30	15	3,681
dmu14	30	15	3,394
dmu15	30	15	3,343
dmu18	30	20	3,844
dmu21	40	15	4,380
dmu22	40	15	4,725
dmu23	40	15	4,668
dmu24	40	15	4,648
dmu25	40	15	4,164
dmu26	40	20	4,647
dmu27	40	20	4,848
dmu28	40	20	4,692
dmu29	40	20	4,691
dmu30	40	20	4,732
dmu31	50	15	5,640
dmu32	50	15	5,927
dmu33	50	15	5,728
dmu34	50	15	5,385
dmu35	50	15	5,635
dmu36	50	20	5,621
dmu37	50	20	5,851
dmu38	50	20	5,713
dmu39	50	20	5,747
dmu40	50	20	5,577
ft06	6	6	55
ft10	10	10	930
ft20	20	5	1,165
la01	10	5	666
la02	10	5	655
la03	10	5	597
la04	10	5	590
la05	10	5	593
la06	15	5	926
la07	15	5	890
la08	15	5	863
la09	15	5	951
la10	15	5	958
la11	20	5	1,222
la12	20	5	1,039
la13	20	5	1,150
la14	20	5	1,292
la15	20	5	1,207
la16	10	10	945
la17	10	10	784
la18	10	10	848
la19	10	10	842

Instance	n	m	C^*_{max}
la20	10	10	902
la21	15	10	1,046
la22	15	10	927
la23	15	10	1,032
la24	15	10	935
la25	15	10	977
la26	20	10	1,218
la27	20	10	1,235
la28	20	10	1,216
la29	20	10	1,152
la30	20	10	1,355
la31	30	10	1,784
la32	30	10	1,850
la33	30	10	1,719
la34	30	10	1,721
la35	30	10	1,888
la36	15	15	1,268
la37	15	15	1,397
la38	15	15	1,196
la39	15	15	1,233
la40	15	15	1,222
orb01	10	10	1,059
orb02	10	10	888
orb03	10	10	1,005
orb04	10	10	1,005
orb05	10	10	887
orb06	10	10	1,010
orb07	10	10	397
orb08	10	10	899
orb09	10	10	934
orb10	10	10	944
swv01	20	10	1,407
swv02	20	10	1,475
swv03	20	10	1,398
swv04	20	10	1,464
swv05	20	10	1,424
swv11	50	10	2,983
swv13	50	10	3,104
swv14	50	10	2,968
swv15	50	10	2,885
swv16	50	10	2,924
swv17	50	10	2,794
swv18	50	10	2,852
swv19	50	10	2,843
swv20	50	10	2,823
ta01	15	15	1,231
ta02	15	15	1,244
ta03	15	15	1,218
ta04	15	15	1,175
ta05	15	15	1,224
ta06	15	15	1,238
ta07	15	15	1,227

Instance	n	m	C^*_{max}
ta08	15	15	1,217
ta09	15	15	1,274
ta10	15	15	1,241
ta11	20	15	1,357
ta12	20	15	1,367
ta13	20	15	1,342
ta14	20	15	1,345
ta15	20	15	1,339
ta16	20	15	1,360
ta17	20	15	1,462
ta19	20	15	1,332
ta20	20	15	1,348
ta21	20	20	1,642
ta24	20	20	1,644
ta28	20	20	1,603
ta31	30	15	1,764
ta35	30	15	2,007
ta36	30	15	1,819
ta37	30	15	1,771
ta38	30	15	1,673
ta39	30	15	1,795
ta51	50	15	2,760
ta52	50	15	2,756
ta53	50	15	2,717
ta54	50	15	2,839
ta55	50	15	2,679
ta56	50	15	2,781
ta57	50	15	2,943
ta58	50	15	2,885
ta59	50	15	2,655
ta60	50	15	2,723
ta61	50	20	2,868
ta62	50	20	2,869
ta63	50	20	2,755
ta64	50	20	2,702
ta65	50	20	2,725
ta66	50	20	2,845
ta67	50	20	2,825
ta68	50	20	2,784
ta69	50	20	3,071
ta70	50	20	2,995
ta71	100	20	5,464
ta72	100	20	5,181
ta73	100	20	5,568
ta74	100	20	5,339
ta75	100	20	5,392
ta76	100	20	5,342
ta77	100	20	5,436
ta78	100	20	5,394
ta79	100	20	5,358
ta80	100	20	5,183
yn01	20	20	884

Table A.2: Instances from `lit` with known optimal solution

Bibliography

[AC91] David Applegate and William Cook. "A Computational Study of the Job-Shop Scheduling Problem". *ORSA Journal on Computing* 3.2 (1991). DOI: 10.1287/ijoc.3.2.149.

[Ada+88] Joseph Adams et al. "The Shifting Bottleneck Procedure for Job Shop Scheduling". *Management Science* 34.3 (1988). DOI: 10.1287/mnsc.34.3.391.

[BC64] G. E. P. Box and D. R. Cox. "An Analysis of Transformations". *Journal of the Royal Statistical Society: Series B (Methodological)* 26.2 (1964). DOI: 10.1111/j.2517-6161.1964.tb00553.x.

[Bec+11] J. Christopher Beck et al. "Combining Constraint Programming and Local Search for Job-Shop Scheduling". *INFORMS Journal on Computing* 23.1 (2011). DOI: 10.1287/ijoc.1100.0388.

[BF04] J. Christopher Beck and Eugene C. Freuder. "Simple Rules for Low-Knowledge Algorithm Selection". *Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems, First International Conference, CPAIOR 2004, Nice, France, April 20-22, 2004, Proceedings*. Vol. 3011. Lecture Notes in Computer Science. Springer, 2004. DOI: 10.1007/978-3-540-24664-0_4.

[ÇB15] Banu Çaliş and Serol Bulkan. "A research survey: review of AI solution strategies of job shop scheduling problem". *Journal of Intelligent Manufacturing* 26.5 (2015). DOI: 10.1007/s10845-013-0837-8.

[CR10] David W. Corne and Alan P. Reynolds. "Optimisation and Generalisation: Footprints in Instance Space". *Parallel Problem Solving from Nature - PPSN XI, 11th International Conference, Kraków, Poland, September 11-15, 2010, Proceedings, Part I*. Vol. 6238. Lecture Notes in Computer Science. Springer, 2010. DOI: 10.1007/978-3-642-15844-5_3.

[Dem+97] Ebru Demirkol et al. "A Computational Study of Shifting Bottleneck Procedures for Shop Scheduling Problems". *Journal of Heuristics* 3.2 (1997). DOI: 10.1023/A:1009627429878.

[Dem+98] Ebru Demirkol et al. “Benchmarks for shop scheduling problems”. *European Journal of Operational Research* 109.1 (1998). DOI: 10.1016/S0377-2217(97)00019-2.

[FP09] Christodoulos A. Floudas and Panos M. Pardalos. *Encyclopedia of optimization*. Springer Science & Business Media, 2009. DOI: 10.1007/978-0-387-74759-0.

[FT63] H. Fisher and G. L. Thompson. “Probabilistic learning combinations of local job-shop scheduling rules”. *Industrial Scheduling*. Prentice Hall, 1963.

[Goo20] Google. *OR-Tools*. 2020. URL: https://developers.google.com/optimization (visited on 09/01/2020).

[Gor04] J. Gorodkin. “Comparing two K-category assignments by a K-category correlation coefficient”. *Computational Biology and Chemistry* 28.5-6 (2004). DOI: 10.1016/j.compbiolchem.2004.09.006.

[Gra+79] R. L. Graham et al. “Optimization and Approximation in Deterministic Sequencing and Scheduling: a Survey”. *Annals of Discrete Mathematics*. Vol. 5. Discrete Optimization II. Elsevier, 1979. DOI: 10.1016/S0167-5060(08)70356-X.

[GT60] B. Giffler and G. L. Thompson. “Algorithms for Solving Production-Scheduling Problems”. *Operations Research* 8.4 (1960). DOI: 10.1287/opre.8.4.487.

[HS16] Emma Hart and Kevin Sim. “A Hyper-Heuristic Ensemble Method for Static Job-Shop Scheduling”. *Evolutionary Computation* 24.4 (2016). DOI: 10.1162/EVCO_a_00183.

[Hut+11] Frank Hutter et al. “Sequential Model-Based Optimization for General Algorithm Configuration”. *Learning and Intelligent Optimization - 5th International Conference, LION 5, Rome, Italy, January 17-21, 2011. Selected Papers*. Vol. 6683. Lecture Notes in Computer Science. Springer, 2011. DOI: 10.1007/978-3-642-25566-3_40.

[IBM20a] IBM. *IBM CPLEX Optimizer*. 2020. URL: https://www.ibm.com/analytics/cplex-optimizer (visited on 09/01/2020).

[IBM20b] IBM. *IBM ILOG CP Optimizer*. 2020. URL: https://www.ibm.com/analytics/cplex-cp-optimizer (visited on 09/01/2020).

[IR12] Helga Ingimundardottir and Thomas Philip Runarsson. “Determining the Characteristic of Difficult Job Shop Scheduling Instances for a Heuristic Solution Method”. *Learning and Intelligent Optimization - 6th International Conference, LION 6, Paris, France, January 16-20, 2012, Revised Selected Papers*. Vol. 7219. Lecture Notes in Computer Science. Springer, 2012. DOI: 10.1007/978-3-642-34413-8_36.

[JM98] Anant Singh Jain and Sheik Meeran. *A State-Of-The-Art Review Of Job-Shop Scheduling Techniques*. Tech. rep. Department of Applied Physics, Electronic and Mechanical Engineering: University of Dundee, Dundee, Scotland, 1998.

[Jur+12] Giuseppe Jurman et al. “A Comparison of MCC and CEN Error Measures in Multi-Class Prediction”. *PLoS ONE* 7.8 (2012). DOI: 10.1371/journal.pone.0041882.

[Kot16] Lars Kotthoff. “Algorithm Selection for Combinatorial Search Problems: A Survey”. *Data Mining and Constraint Programming - Foundations of a Cross-Disciplinary Approach*. Vol. 10101. Lecture Notes in Computer Science. Springer, 2016. DOI: 10.1007/978-3-319-50137-6_7.

[LA87] P. J. M. van Laarhoven and E. H. L. Aarts. *Simulated Annealing: Theory and Applications*. Mathematics and Its Applications. Springer Netherlands, 1987. DOI: 10.1007/978-94-015-7744-1.

[Law84] S. Lawrence. *Resource constrained project scheduling*. Technical Report. Carnegie-Mellon University, 1984.

[Len+77] J.K. Lenstra et al. “Complexity of Machine Scheduling Problems”. *Annals of Discrete Mathematics*. Vol. 1. Elsevier, 1977. DOI: 10.1016/S0167-5060(08)70743-X.

[LRK79] J.K. Lenstra and A.H.G. Rinnooy Kan. “Computational Complexity of Discrete Optimization Problems”. *Annals of Discrete Mathematics*. Vol. 4. Elsevier, 1979. DOI: 10.1016/S0167-5060(08)70821-5.

[Mat75] B.W. Matthews. “Comparison of the predicted and observed secondary structure of T4 phage lysozyme”. *Biochimica et Biophysica Acta (BBA) - Protein Structure* 405.2 (1975). DOI: 10.1016/0005-2795(75)90109-9.

[MŠ16] Sadegh Mirshekarian and Dušan N. Šormaz. “Correlation of job-shop scheduling problem features with scheduling efficiency”. *Expert Systems with Applications* 62 (2016). DOI: 10.1016/j.eswa.2016.06.014.

[Muñ+18] Mario A. Muñoz et al. “Instance spaces for machine learning classification”. *Machine Learning* 107.1 (2018). DOI: 10.1007/s10994-017-5629-5.

[Muñ20] Mario Andrés Muñoz. *Instance Space*. 2020. URL: https://github.com/andremun/InstanceSpace (visited on 07/27/2020).

[Net+07] Nicholas Nethercote et al. “MiniZinc: Towards a Standard CP Modelling Language”. *Principles and Practice of Constraint Programming - CP 2007, 13th International Conference, CP 2007, Providence, RI, USA, September 23-27, 2007, Proceedings*. Vol. 4741. Lecture Notes in Computer Science. Springer, 2007. DOI: 10.1007/978-3-540-74970-7_38.

[Ngu+13] Su Nguyen et al. “A Computational Study of Representations in Genetic Programming to Evolve Dispatching Rules for the Job Shop Scheduling Problem”. *IEEE Transactions on Evolutionary Computation* 17.5 (2013). DOI: 10.1109/TEVC.2012.2227326.

[NS05] Eugeniusz Nowicki and Czes?aw Smutnicki. “An Advanced Tabu Search Algorithm for the Job Shop Problem”. *Journal of Scheduling* 8.2 (2005). DOI: 10.1007/s10951-005-6364-5.

[oMe20] University of Melbourne. *MATILDA*. 2020. URL: https://matilda.unimelb.edu.au/matilda/ (visited on 07/27/2020).

[PG81] John W. Pratt and Jean D. Gibbons. *Concepts of Nonparametric Theory*. Springer Series in Statistics. New York, NY: Springer New York, 1981. DOI: 10.1007/978-1-4612-5931-2.

[Ric76] John R. Rice. "The Algorithm Selection Problem". *Advances in Computers* 15 (1976). DOI: 10.1016/S0065-2458(08)60520-3.

[SH14] Kevin Sim and Emma Hart. "An improved immune inspired hyper-heuristic for combinatorial optimisation problems". *Proceedings of the 2014 conference on Genetic and evolutionary computation - GECCO '14*. Vancouver, BC, Canada: ACM Press, 2014. DOI: 10.1145/2576768.2598241.

[SM+09] Kate A. Smith-Miles et al. "A Knowledge Discovery Approach to Understanding Relationships between Scheduling Problem Structure and Heuristic Performance". *Learning and Intelligent Optimization*. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2009. DOI: 10.1007/978-3-642-11169-3_7.

[SM+14] Kate Smith-Miles et al. "Towards objective measures of algorithm performance across instance space". *Computers & Operations Research* 45 (2014). DOI: 10.1016/j.cor.2013.11.015.

[SML12] Kate Smith-Miles and Leo Lopes. "Measuring instance difficulty for combinatorial optimization problems". *Computers & Operations Research* 39.5 (2012). DOI: 10.1016/j.cor.2011.07.006.

[SS06] M. J. Streeter and S. F. Smith. "How the Landscape of Random Job Shop Scheduling Instances Depends on the Ratio of Jobs to Machines". *Journal of Artificial Intelligence Research* 26 (2006). DOI: 10.1613/jair.2013.

[Sto+92] Robert H. Storer et al. "New Search Spaces for Sequencing Problems with Application to Job Shop Scheduling". *Management Science* 38.10 (1992). DOI: 10.1287/mnsc.38.10.1495.

[Stu+14] Peter J. Stuckey et al. "The MiniZinc Challenge 2008–2013". *AI Magazine* 35.2 (2014). DOI: 10.1609/aimag.v35i2.2539.

[Tai93] E. Taillard. "Benchmarks for basic scheduling problems". *European Journal of Operational Research* 64.2 (1993). DOI: 10.1016/0377-2217(93)90182-M.

[Tai94] Éric D. Taillard. "Parallel Taboo Search Techniques for the Job Shop Scheduling Problem". *ORSA Journal on Computing* 6.2 (1994). DOI: 10.1287/ijoc.6.2.108.

[Vae+96] R. J. M. Vaessens et al. "Job Shop Scheduling by Local Search". *INFORMS Journal on Computing* 8.3 (1996). DOI: 10.1287/ijoc.8.3.302.

[vHo18] Jelke J. van Hoorn. "The Current state of bounds on benchmark instances of the job-shop scheduling problem". *Journal of Scheduling* 21.1 (2018). DOI: 10.1007/s10951-017-0547-8.

[Vil+15] Petr Vilím et al. "Failure-Directed Search for Constraint-Based Scheduling". *Integration of AI and OR Techniques in Constraint Programming - 12th International Conference, CPAIOR 2015, Barcelona, Spain, May 18-22, 2015, Proceedings*. Vol. 9075. Lecture Notes in Computer Science. Springer, 2015. DOI: 10.1007/978-3-319-18008-3_30.

[vLa+92] Peter J. M. van Laarhoven et al. "Job Shop Scheduling by Simulated Annealing". *Operations Research* 40.1 (1992). DOI: 10.1287/opre.40.1.113.

[Wat+03] Jean-Paul Watson et al. "Problem difficulty for tabu search in job-shop scheduling". *Artificial Intelligence* 143.2 (2003). DOI: 10.1016/S0004-3702(02)00363-6.

[Wat+06] Jean-Paul Watson et al. "Deconstructing Nowicki and Smutnicki's -TSAB tabu search algorithm for the job-shop scheduling problem". *Computers & Operations Research* 33.9 (2006). DOI: 10.1016/j.cor.2005.07.016.

[Web+19] Edzard Weber et al. "Need for Standardization and Systematization of Test Data for Job-Shop Scheduling". *Data* 4.1 (2019). DOI: 10.3390/data4010032.

[YN92] Takeshi Yamada and Ryohei Nakano. "A Genetic Algorithm Applicable to Large-Scale Job-Shop Problems". *Parallel Problem Solving from Nature 2, PPSN-II, Brussels, Belgium, September 28-30, 1992*. Elsevier, 1992.

[Zha+08] Chao Yong Zhang et al. "A very fast TS/SA algorithm for the job shop scheduling problem". *Computers & Operations Research* 35.1 (2008). DOI: 10.1016/j.cor.2006.02.024.