Hyper-heuristic Evolution of Dispatching Rules: 
A Comparison of Rule Representations

Jürgen Branke 
Juergen.Branke@wbs.ac.uk
Warwick Business School, University of Warwick, CV4 7AL Coventry, UK

Torsten Hildebrandt 
hil@biba.uni-bremen.de
Department of Planning and Control of Production Systems, Universität Bremen, Bremen, Germany

Bernd Scholz-Reiter 
bsr@biba.uni-bremen.de
Department of Planning and Control of Production Systems, Universität Bremen, Bremen, Germany

doi:10.1162/EVCO_a_00131

Abstract
Dispatching rules are frequently used for real-time, online scheduling in complex manufacturing systems. Design of such rules is usually done by experts in a time consuming trial-and-error process. Recently, evolutionary algorithms have been proposed to automate the design process. There are several possibilities to represent rules for this hyper-heuristic search. Because the representation determines the search neighborhood and the complexity of the rules that can be evolved, a suitable choice of representation is key for a successful evolutionary algorithm. In this paper we empirically compare three different representations, both numeric and symbolic, for automated rule design: A linear combination of attributes, a representation based on artificial neural networks, and a tree representation. Using appropriate evolutionary algorithms (CMA-ES for the neural network and linear representations, genetic programming for the tree representation), we empirically investigate the suitability of each representation in a dynamic stochastic job shop scenario. We also examine the robustness of the evolved dispatching rules against variations in the underlying job shop scenario, and visualize what the rules do, in order to get an intuitive understanding of their inner workings. Results indicate that the tree representation using an improved version of genetic programming gives the best results if many candidate rules can be evaluated, closely followed by the neural network representation that already leads to good results for small to moderate computational budgets. The linear representation is found to be competitive only for extremely small computational budgets.

Keywords
Job shop scheduling, dispatching rule, representation, genetic programming, CMA-ES, artificial neural network.

1 Introduction
Many practical scheduling problems are dynamically changing over time, for example due to new jobs arriving, stochastic processing times, or machine failures. A promising and widely used approach to tackle such dynamically changing problems is to rely on self-organization via dispatching rules. A dispatching rule is a simple heuristic that derives a priority index for a job from its attributes, and then simply schedules the job with the highest priority.
The challenge is to design local, decentralized dispatching rules which result in a good global performance of a given complex manufacturing system, measured by criteria such as the mean tardiness or mean flowtime of jobs. Usually, dispatching rules are designed by experts in a tedious trial-and-error process, with candidate rules being tested in a simulation environment, modified, and retested until they fulfill the requirements for actual implementation in practice (Geiger et al., 2006). In recent years, several authors have suggested automatically designing suitable dispatching rules by means of evolutionary algorithms (EAs; see, e.g., Geiger et al., 2006; Dimopoulos and Zalzala, 2001; Pickardt et al., 2010). Because it is difficult to decide on the required complexity of a dispatching rule beforehand, many of these approaches use genetic programming (GP; Koza, 1992; Poli et al., 2008) which allows to evolve arbitrarily complex functions. The search space in this case is that of arithmetic expressions for the dispatching rules. Other representations are possible, however, and different search spaces also allow the use of different optimization algorithms. To the best of our knowledge there are no comparisons or guidelines on choosing one over the other.

In this paper, we compare three possible representations for dispatching rules.

1. Expression trees are commonly used in GP as an explicit representation of arbitrarily complex functions. This is a symbolic representation, optimization aims at finding an arithmetic expression.

2. Artificial neural networks (ANNs) also allow arbitrarily complex functions but in an implicit (i.e., sub-symbolic) way. Optimization consists of finding real-valued weights. The weights of the ANN are determined by a standard evolution strategy (CMA-ES; Hansen and Ostermeier, 2001).

3. Simple linear combinations of various job attributes, with weights (i.e., also a real-valued search space) are tuned again by CMA-ES.

We empirically test these approaches on a well researched dynamic, stochastic job shop scheduling problem (JSSP) proposed in Rajendran and Holthaus (1999), and compare results with the best known dispatching rules from the literature. One key contribution of this paper is the comparison of these different representations, analyzing the solution quality obtainable, convergence speed, and search reliability. During our experiments we are able to identity two factors to improve convergence speed and solution quality of the tree representation and GP: normalization of attribute values and removing phenotypic duplicates by an approximate measure of phenotypic equality. Furthermore, solution robustness is investigated, and a visualization of the resulting rules is presented.

The paper is structured as follows. Section 2 describes the underlying problem of learning dispatching rules as a problem of hyper-heuristic search in a simulation optimization setting. The three different representations are presented in Section 3. Our experimental setup is described in Section 4 and the results are presented in Section 5. The paper concludes with a summary and some ideas for future work.

2 Problem Description

2.1 Job Shop Scheduling

Job shop scheduling is concerned with the assignment of operations of a job to machines, subject to technological constraints (operations have to be processed in a pre-determined sequence on specific machines) as well as capacity constraints (each machine can process
Dispatching rules are used to compute priority values for operations based on a given set of attribute values $S$. The attribute set can consist of any scheduling relevant information, including job data and machine data, but also global system information (Haupt, 1989). Attributes frequently used in dispatching rules can be found in reviews on the subject (see, e.g., Haupt, 1989; Blackstone et al., 1982; Holthaus and Rajendran, 1997).

Whenever a machine becomes idle and has waiting operations in its queue, the job with the highest priority value is chosen to be processed next. In the setting we use in this paper, a machine is never kept idle if there are jobs waiting in its queue; that is, we consider only non-delay schedules. Note that dispatching rules are online scheduling mechanisms; that is, decisions are made at the latest possible time (whenever a machine becomes idle) on the basis of the information available at that time. As the computation of priority values is computationally efficient, this method allows real-time scheduling in a dynamic environment.

2.3 Learning Dispatching Rules

We employ a metaheuristic (evolutionary algorithm) to search the space of heuristics (dispatching rules), rather than the space of solutions to the particular scheduling problem. Thus, our approach constitutes a hyper-heuristic approach (Burke et al., 2009, 2013). The derived dispatching rule can later be used for scheduling of many problem instances.

The search space of the hyper-heuristic is determined by at least the following two design decisions.

1. The first design decision is the information that may be used to calculate the priority value, that is, the set of attributes considered (e.g., processing time, due date, work in next queue, etc.). This defines the dimensionality of the attribute space $S$.

2. The second design decision is the representation. This is composed of the function class $P$ (e.g., all linear combinations of the attributes in $S$, or an expression tree with the attributes of $S$ as terminals) and the parameter space $X$ which specifies the parameters that can be modified to optimize the dispatching rule (e.g., weights of the linear combination or nodes and edges of the expression tree).
So for a given function class, an individual with genotype $x \in X$ encodes a dispatching rule $p_x : S \to \mathbb{R}$ that maps a vector $s \in S \subseteq \mathbb{R}^n$ of attribute values from the attribute space $S$ to a real number, the priority value, which in turn defines an order relation among the operations waiting at a machine.

For fitness evaluation, because it is not possible to evaluate a dispatching rule analytically, we follow a simulation optimization approach (Fu, 2001), where an evolutionary algorithm is used to search for a good dispatching rule, and a stochastic simulation is used for evaluation. Note that as the simulation is stochastic, the resulting fitness values are stochastic as well.

The overall approach is depicted in Figure 1. First, the attribute set and representation (function class) need to be specified. This determines the search space on which the optimization algorithm operates. Candidate rules are tested on a shop floor simulator, and the resulting rule performance is used by the optimization algorithm to select the next candidate rules.

In the following, we look at different options for the attribute set (small or large) and the representation (linear combination, neural network, and expression trees).

### 3 Rule Representations

#### 3.1 Choosing a Representation

Evolutionary algorithms usually operate on two levels: The genetic operators are applied on the genotype level, and genotypes are transformed into phenotypes (actual solution) for evaluation. The space of genotypes and the associated mapping from genotype to phenotype are often termed representation (Rothlauf, 2006).

Together with the genetic operators, the genetic representation defines the size and structure of the search space. Representation, genetic operators, and fitness function combine to form the fitness landscape. Because the representation may change the structure of the neighborhood, it has a large impact on the difficulty of the resulting search problem. A suitable choice of representation is thus a key element of a successful and efficient evolutionary algorithm.
Intuitively, the search space should be small, connected, and contain the optimal solution but no infeasible solutions. The fitness landscape should also be smooth in the sense that neighboring solutions (with respect to the genetic operators) have similar fitness, and the optimum ideally should have a large basin of attraction.

A number of publications have formalized characteristics of representations, such as locality (similar genotypes are mapped to similar phenotypes; Raidl and Gottlieb, 2005; Rothlauf, 2003), fitness-distance correlation (Jones and Forrest, 1995), heuristic bias (randomly generated genotypes tend to be mapped to good phenotypes), as well as inheritability (the result of crossover bears a similarity to the parents used; Raidl and Gottlieb, 2005) and redundancy (how many different genotypes map to the same phenotype; Rothlauf and Goldberg, 2003). For a more detailed treatment of the topic of representations, see also Rothlauf (2006).

Although the importance of a proper representation has been known for a long time, designing a good representation is still more an art than a science, and very much depends on the experience and skills of the algorithm developer. A recent paper on open issues in genetic programming (O’Neill et al., 2010) lists representations as one of the key challenges in the area. An empirical comparison of different representations, as in this paper, helps in understanding the advantages and disadvantages of different alternatives, and provides guidelines for algorithm developers in the future.

3.2 Linear Representation

In the linear case the dispatching rule consists of a linear combination of the elements of the attribute set. Thus the genotype $x$ in this representation consists of a weight vector $w$ with as many elements as the attribute set. The priority is computed using the following formula, and the search space is $\mathbb{R}^{|s|}$.

$$p_x(s) = \sum_{i=1}^{|s|} w_i s_i$$

A very early application of heuristic optimization techniques to find dispatching rules is reported in Hershauer and Ebert (1975). Using a heuristic search method, a linear combination of attributes is learned to optimize a cost-related objective function in a job shop environment. This rule representation is very similar to the linear representation investigated in this paper.

3.3 Tree Representation

In the tree representation the dispatching rule $p_x$ is represented as an expression tree forming a valid arithmetic expression. Inner nodes (nonterminal nodes) in the tree are arithmetic operations, and leaf nodes and (terminal nodes) are constants or elements of the attribute set. Figure 2(a) shows an example tree. Each time the priority of a job has to be calculated, the tree gets evaluated, replacing the terminal nodes with their respective current values. Which node types to allow as non-terminal nodes and terminal nodes (besides the elements of the attribute set) is a design choice to be made before the search.

Each non-terminal node has a fixed, predetermined number of child nodes, depending on the operation it encodes. The search space therefore consists of all valid trees, fulfilling the arity-constraints of the non-terminal nodes.

Genetic programming (Koza, 1992; Poli et al., 2008) allows an arbitrarily complex arithmetic expression to be evolved using a variable length tree representation. So, not surprisingly, several authors have used genetic programming to generate dispatching rules that assemble basic job, machine, and system attributes into priority indices.
3.4 Artificial Neural Network Representation

In the neural network representation, the functional mapping between the independent variables (the elements of the attribute set) and the dependent variable (the priority value) is computed by a neural network. We use a very basic structure of the neural network: a fully connected feed-forward neural network with a single hidden layer. Although easy to implement, it is quite powerful as a universal function approximator. Theoretical results show that a feed-forward neural network with a single hidden layer can approximate any continuous function with arbitrary precision (Hornik et al., 1989).

Figure 2(b) shows an example. The network’s input layer consists of all available information from the attribute set plus a bias node that always has the value 1. Values on the input layer are typically normalized to $[-1, +1]$. Every node on the hidden layer is connected with every node from the input layer. Every node from the hidden layer is connected to the single output node, which finally computes the priority of an operation.

Each neuron on the hidden layer computes the function

$$\text{out}(z, w) = \tanh \left( \sum_{i=1}^{n} w_i z_i \right)$$

as its output value. In this formula, $n$ is the number of incoming edges, the vector $z$ contains the activation levels of all nodes with an incoming connection to the neuron, and $w$ is the weight vector specifying a weight for each incoming connection.

Generally speaking, the number of neurons in the hidden layer bounds the complexity of the function which can be approximated. More nodes mean a more flexible
representation, but also more weight values to be tuned, complicating the optimization task. The number of weight values of a fully connected feed-forward network with a single hidden layer as shown in Figure 2(b) can be calculated as \((n_{\text{input}} + 1)n_{\text{hidden}} + n_{\text{hidden}}n_{\text{output}}\), with \(n_{\text{input}}\), \(n_{\text{hidden}}\), and \(n_{\text{output}}\) being the number of nodes on the input, hidden, and output layer, respectively.

Summarizing, the genotype \(x = (w^O, w^H)\) for the neural network representation consists of the weight vector \(w^O\) for the connection weights from the hidden layer to the output neuron, and the weight matrix \(w^H\), specifying the weights from each input to each hidden layer neuron. The priority value is calculated as:

\[
p_x(s) = \sum_{h=1}^{n_{\text{hidden}}} w^O_h \tanh \left( \sum_{i=0}^{n_{\text{input}}} w^H_{h,i} s_i \right).
\]

The formula also illustrates the relationship to the linear representation; a neural network without a hidden layer would be equivalent to the linear representation.

There exist a few papers that use artificial neural networks to select an appropriate dispatching rule for a given problem instance (e.g., Liu and Dong, 1996; Mouelhi-Chibani and Pierreval, 2010), but we found only one paper (Eguchi et al., 2008) using neural networks to represent a dispatching rule. In Eguchi’s paper, a neural network is learned using simulated annealing in an attempt to find a dispatching rule performing well under various shop floor conditions. Since direct learning of such a rule was not successful, a two-stage procedure was employed, first developing specialized networks for different shop conditions and in a second stage simplifying and integrating these networks. Using CMA-ES for optimization, and restricting our input set to attributes commonly used in standard dispatching rules, we can simplify this procedure and directly learn very good dispatching rules.

### 4 Experimental Setup

#### 4.1 Scenario

Our computational experiments use the dynamic job shop scenarios from Rajendran and Holthaus (1999) and try to find good dispatching rules for these scenarios automatically. In total there are 10 machines on the shop floor. Each job entering the system is assigned a random routing; that is, machine visitation order is random with no machine being revisited. Processing times are drawn from a uniform discrete distribution ranging from 1 to 49 min. Job arrival is a Poisson process, that is, interarrival times are exponentially distributed. The mean of this distribution is chosen to reach a desired utilization level on all machines. Unless stated otherwise, all experiments in this paper use a (mean) utilization of 95%.

Following the procedure from Rajendran and Holthaus (1999), we start with an empty shop and simulate the system until we collected data from jobs numbering from 501 to 2,500. The shop is further loaded with jobs, until the completion of these 2,000 jobs to overcome the problem of censored data (Conway, 1965). Data on the first 500 jobs is disregarded to focus on the shop’s steady state behavior.

The objective function is the mean flowtime, that is, the arithmetic mean of the flowtimes of jobs 501 to 2,500: \(FT = \frac{1}{2000} \sum_{j=501}^{2500} (C_j - r_j)\), where \(C_j\) is the completion time of a job \(j\) and \(r_j\) is its release date.

Because the simulation is stochastic, the calculated flowtime is also a stochastic variable. To reduce the impact of stochasticity, we replicate each simulation 10 times and use the average performance for optimization. Furthermore, we use the same 10 random...
number seeds to evaluate all solutions during optimization (common random numbers). All results reported in this paper are based on a different set of 100 independent test runs.

As it turned out in preliminary experiments, rules performing very badly can cause very long simulation times. We therefore terminate a simulation early, if the number of jobs concurrently in the system (work in progress, WIP) becomes larger than 500, which is about five times the maximum WIP of standard rules such as first-in-first-out (FIFO). Such a high WIP is a sure sign of a rule with very bad performance and the system running full. In these cases we assign it a performance computed as $FT + C - numFinished$, where $FT$ is the partial flowtime of those jobs completed, and $C$ is a constant large enough to ensure rules causing the system to run full always receive a worse performance value than well-working rules. We subtract the number of jobs finished from the sum of these two measures, so rules that finish a large number of jobs 501 to 2,500 before the system runs full get a better (i.e., smaller) performance value than those rules completing only a few (or none). The simulation has been implemented in Java and is publicly available (Hildebrandt, 2012).

### 4.2 Benchmark Rules

As benchmark dispatching rules, we use two standard rules (SPT and FIFO) and a rule manually developed by Holthaus and Rajendran (2000) that performs very well for the objective of minimizing mean flow time in job shops (denoted the Holthaus rule). If equal priorities occur in any of the experiments, we prefer the job that is in the system the longest, i.e., we use ERD (earliest release date first) as the tie breaker rule.

**FIFO—First In First Out**

Jobs are processed in the order they entered the queue in front of a machine. This means the priority is calculated using $-AT$, with $AT$ being the arrival time of a job in its current queue.

**ERD—Earliest Release Date First**

Jobs are processed in the order they entered the system. This means priority is calculated using $-RT$, with $RT$ being the release time of a job into the system.

**Random**

This rule is different from the other rules considered in this paper, as job sequences do not follow any deterministic order. If more than one job is ready to be processed, the job started next is determined randomly.

**SPT—Shortest Processing Time First**

The job with the shortest processing time of its current operation is processed first. This rule is usually considered a good choice if the objective is to minimize mean flowtime. Therefore the priority is calculated using $-PT$, where $PT$ is the processing time of a job’s current operation.

**Holthaus**

Using this rule, the priority of a job is calculated using $-(2PT + WINQ + NPT)$, and is thus a linear combination of the current processing time ($PT$), work content in next queue ($WINQ$), and the processing time of a job’s next operation ($NPT$; next processing time). This rule was manually developed and presented by Holthaus and Rajendran (2000) in a follow-up paper to Rajendran and Holthaus (1999). In that paper, the authors
Table 1: Information included in the search if using basic attribute set and additional information used with extended attribute set.

<table>
<thead>
<tr>
<th>Value</th>
<th>Normalization Interval</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Basic and extended attribute sets</td>
<td></td>
<td></td>
</tr>
<tr>
<td>PT</td>
<td>[1,47]</td>
<td>Processing time of current operation</td>
</tr>
<tr>
<td>NPT</td>
<td>[0,47]</td>
<td>Next processing time; that is, processing time of next operation</td>
</tr>
<tr>
<td>WINQ</td>
<td>[0,410]</td>
<td>Work in next queue; that is, sum of processing times of all jobs at the next machine</td>
</tr>
<tr>
<td>Extended attribute set only</td>
<td></td>
<td></td>
</tr>
<tr>
<td>RemProcTime</td>
<td>[1,264]</td>
<td>RPT; that is, sum of processing times of all unfinished operations</td>
</tr>
<tr>
<td>OpsLeft</td>
<td>[1,10]</td>
<td>Operations left; that is, number of all unfinished operations</td>
</tr>
<tr>
<td>TIQ</td>
<td>[0,1500]</td>
<td>Time since arrival in current queue</td>
</tr>
<tr>
<td>TIS</td>
<td>[0,2770]</td>
<td>Time since arrival in system</td>
</tr>
</tbody>
</table>

used slightly different scenarios to assess rule performance; however, this rule is also superior in the original scenarios of Rajendran and Holthaus (1999) used in this paper. This rule is the real benchmark to beat; to the best of our knowledge it is the best manually developed rule for these scenarios.

4.3 Attribute Set and Representation Parameters

A challenge of automatic rule generation is to identify a suitable set of attributes. On the one hand, it is important to include all relevant characteristics of the problem; on the other hand, the search space grows exponentially as a function of the number of attributes provided. For the experiments in this paper, we use two different attribute sets, as shown in Table 1. The basic set consists of those three attributes also used in the Holthaus rule. The extended attribute set additionally contains the remaining processing time, number of unfinished operations, queuing time, and time in system, as additional attributes. These attributes are easily computable and frequently used in standard rules (Haupt, 1989; Blackstone et al., 1982; Holthaus and Rajendran, 1997; Rajendran and Holthaus, 1999).

Table 2 indicates the dimensionality of the search spaces as a measure of search complexity for each representation using basic or extended attribute sets.

For the neural network representation this not only depends on the number of elements in the attribute set, but also on the number of neurons in the hidden layer. In our experiments we use 3 (basic attribute set only), 7, 14, and 28 hidden neurons, resulting in the number of weight values shown in the table. As an example: using 14 neurons on the hidden layer and the basic attribute set results in 70 real-valued weights, which have to be set appropriately by the optimization algorithm.

For the tree representation, the size of the search space depends on the attribute set, the number of additional terminal values (such as constants), as well as the number and arity of nonterminals, that is, operators usable in the tree (see Langdon and Poli, 2002, p. 113). Table 3 lists the terminals and nonterminals used in our experiments. All nonterminals are binary operators, that is, requiring two arguments, with
Table 2: Overview of representations investigated.

<table>
<thead>
<tr>
<th>Attribute Set</th>
<th>Representation</th>
<th>Complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td>Basic</td>
<td>Linear 3d</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Neural network 3 hidden neurons: 15d</td>
<td>7 hidden neurons: 35d</td>
</tr>
<tr>
<td></td>
<td>14 hidden neurons: 70d</td>
<td>28 hidden neurons: 140d</td>
</tr>
<tr>
<td></td>
<td>Tree 6 non-term, (3+2) term</td>
<td></td>
</tr>
<tr>
<td>Extended</td>
<td>Linear 7d</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Neural network 7 hidden neurons: 63d</td>
<td>14 hidden neurons: 126d</td>
</tr>
<tr>
<td></td>
<td>28 hidden neurons: 252d</td>
<td>Tree 6 non-term, (7+2) term</td>
</tr>
</tbody>
</table>

Table 3: GP parameters used.

<table>
<thead>
<tr>
<th>Name</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Population size</td>
<td>500</td>
</tr>
<tr>
<td>Generations</td>
<td>60</td>
</tr>
<tr>
<td>Crossover proportion</td>
<td>90%</td>
</tr>
<tr>
<td>Mutation proportion</td>
<td>10%</td>
</tr>
<tr>
<td>Elitism</td>
<td>5% of population size</td>
</tr>
<tr>
<td>Selection method</td>
<td>Tournament selection (size 7)</td>
</tr>
<tr>
<td></td>
<td>Ramped half-and-half</td>
</tr>
<tr>
<td>Creation type</td>
<td>(Minimum depth 2, maximum depth 6)</td>
</tr>
<tr>
<td>Maximum depth for crossover</td>
<td>17</td>
</tr>
<tr>
<td>Operators</td>
<td>+, -, *, /, max, if-then-else</td>
</tr>
<tr>
<td>Terminals</td>
<td>0, 1, attribute set</td>
</tr>
</tbody>
</table>

the exception of if-then-else, which is a ternary operator, implementing the expression “if \(a \geq 0\) then \(b\) else \(c\),” where \(a, b,\) and \(c\) can be arbitrary valid subexpressions.

4.4 Normalization

Using the neural network representation requires all attributes to be normalized in the range \([-1, +1]\). All attributes used in this paper have a natural lower bound, whereas there is no clear upper bound for some of the attributes, such as TimeInQueue. Value distributions for these attributes also have a long tail, that is, there are few very large values, whereas the majority of values are much smaller. To find the normalization ranges used in this paper, as shown in the third column of Table 1, we therefore analyzed preliminary simulation runs using the Holthaus rule and set the upper bounds for all attributes to the 90% quantiles encountered in these runs.

For the linear and tree representation, such a normalization is optional; therefore, we test these representations with and without normalization. For the tree representation we use \([0, 2]\) as the normalization interval, to avoid problems with a sign change in the
middle of the value range. Especially for the division operation, this could otherwise complicate the search.

4.5 Duplicate Removal
Motivated by our experiments using randomly generated trees (presented in Section 5.1) we implement and test a mechanism to efficiently suppress trees causing duplicate objective values and therefore increase diversity during a GP run. The results of research on genetic programming in Gustafson (2004) indicate that there is a positive correlation between diversity and the performance achieved by GP for many problems, and consequently high diversity is considered beneficial. Therefore Jackson (2011) modified GP eliminating candidate solutions with duplicate objective function values in the initial population as well as during the GP run and reported improved performance. Detecting such duplicates by comparing objective values is quite costly in a simulation-optimization context, as full simulation runs are required to obtain objective function values. The amount of duplicates can be reduced to a large extent without additional simulations using the following approximate objective-value-equivalence.

Two rules are assumed to be equivalent if they produce the same ranking on an artificial decision situation consisting of 100 dummy operations, that is, values for the elements of the attribute set. Attribute values are created randomly within typical ranges, but without considering any dependencies between attributes, for example. Based on approximate equivalence we eliminate all duplicate rules in a population before running any simulations to assess rule performance. These rules are replaced with new rules either created randomly (initial population), or, later on in a GP run, by using the mutation/crossover operators. As a result, with very low additional computational cost, especially without using any additional simulations, we can dramatically reduce the number of duplicates (details are reported later in Section 5.1).

4.6 Parameter Settings—Optimization
For optimization of the numeric representations we use CMA-ES (Auger and Hansen, 2005; Hansen and Ostermeier, 2001) as a state of the art algorithm for real-valued optimization. The symbolic tree representation uses genetic programming (Koza, 1992; Poli et al., 2008) as a widely used technique to optimize variable-length hierarchical representations.

4.6.1 Random Search
As a very basic algorithm we also use simple random search (applicable to all representations) to get an idea of the baseline performance achievable with each of the representations without the influence of different optimization algorithms.

In our random search experiments we randomly choose the values of all variables uniformly distributed in the interval $[-1, +1]$ for all numeric representations. For the tree representation we use the same settings as indicated under creation type in Table 3. In total we create 10,000 random rules for each representation.

4.6.2 Genetic Programming
To implement genetic programming, we use the evolutionary computation in Java (ECJ)-framework.\(^1\) It uses the standard subtree crossover (produces two offspring trees by exchanging a randomly selected subtree between the two parents) and point mutation

(replacing a node of the tree by a new, randomly created subtree) as described in Koza (1992). The ramped half and half initialization method ensures that the generated trees have a variety of sizes and shapes (Koza, 1992).

Based on our previous experience with the use of genetic programming to create dispatching rules in Pickardt et al. (2010) and Hildebrandt et al. (2010), we used the settings shown in Table 3. We also tried to further improve these settings using SPOT (sequential parameter optimization toolkit, Bartz-Beielstein et al., 2005) for automated parameter optimization. We used a (small) budget of 200 algorithm runs to find a better setting of the four parameters: number of generations (i.e., implicitly also population size), mutation rate, elitism fraction, and tournament size. These runs did not lead to improved parameter settings.

4.6.3 CMA-ES

For our experiments we used the Java implementation\(^2\) of the covariance matrix adaptation-evolution strategy (Hansen and Ostermeier, 2001) provided by Nikolaus Hansen, using default parameter settings. As the only parameter to be specified by a user (except for the starting point), we chose \(\sigma_0\) as 0.3 times our initial value range, that is, \(0.3 \times 2 = 0.6\).

We use the IPOP-variant (Auger and Hansen, 2005) of the algorithm, that is, should the algorithm terminate before the overall budget is exhausted, it is restarted at a new starting point with an increased (doubled) population size.

To create the starting points we create a random point just as described under Section 4.6.1, Random Search. To start the algorithm in a more promising region of the search space, we generate up to five candidate starting points, and use the first that does not cause the system to run full. Should all five points result in the system running full, we choose the point with the best objective function value.

5 Results

This section is structured as follows: We first analyze the results of the random search experiments in Section 5.1, before presenting actual optimization results obtained using CMA-ES for the linear (Section 5.2) and neural network (Section 5.4) representations, or GP for the tree representation (Section 5.3). Thereafter, Section 5.5 compares the different representations based on the best settings identified while Section 5.6 investigates search reliability. To gain insight into why the optimized rules perform very well, we visualize exemplary rules in Section 5.7 and discuss findings before looking at rule robustness in Section 5.9.

Unless stated otherwise, we report on mean flowtime relative to the performance of the Holthaus rule, the best manually designed rule for this scenario. For example, a performance of 0.961 means a flowtime reduction of 3.9% over the results of the Holthaus rule. In the tables listing performance differences of different representations/settings, these are given as absolute differences (in minutes), followed by an indication of the statistical significance of the difference shown, based on a nonparametric paired Wilcoxon signed-rank test: “o” indicates no significance, “+” indicates significant (error probability < 5%), “++” indicates high significance (error probability < 1%). Optimization results for all neural network settings in Section 5.4 are based on 100 independent runs. For the linear and tree representations in Sections 5.2 and 5.3 we initially use 50 runs,

and only perform 50 additional runs for the best settings identified. Comparisons between representations in Sections 5.5 and 5.6 are therefore based on 100 independent runs of each representation.

5.1 Random Search

Different search spaces require different optimizers, in our case CMA-ES for linear combinations and neural networks and GP for tree representations. Analyzing randomly generated solutions allows us to gain insight into some of the fundamental characteristics of the search space, rather than the combination of search space and optimizer.

Several properties of randomly generated rules for the different search spaces are listed in Table 4.

The duplicates column lists the fraction of rules yielding duplicate mean flowtime results. As can be seen, no duplicates occurred using linear and neural network representations, that is, each of the 10,000 rules created randomly for each representation resulted in a unique mean flowtime performance. This is very different for the tree representation. Without duplicate suppression, although all trees created are syntactically different, less than half of them resulted in unique flowtime results (numbers in parentheses). One possible explanation is that these trees only differ in subtrees that never have an influence on the priority values. Another possible cause is that the tree representation allows a rule to be represented by various syntactically different trees, such as "\((1 + 1) \times PT\)" being equivalent to "\(PT + PT\)". Finally, only the ordering of jobs is relevant; absolute priority values do not matter. Therefore the two expressions just mentioned both encode a longest processing time first rule, which can also be expressed by PT.

Since the existence of such duplicates may impact the search, we developed the duplicate suppression mechanism described in Section 4.5 to detect and remove duplicate rules based on approximate objective-value-equivalence. This mechanism turns out to be very effective, reducing the fraction of duplicates to less than half a percent (Table 4) and therefore increasing rule diversity. The results for the tree representation in the remaining columns of Table 4, as well as the histograms for the tree representation in Figure 3, are therefore based on rules generated with duplicate suppression enabled.
As described in the last paragraph of Section 4.1, we terminate simulations early during determination of the objective value once a certain threshold of jobs concurrently in the system is exceeded. Long queues in those cases cause particularly long simulation times and would be expected to show poor performance anyway. The column Fraction full in Table 4 lists the percentage of rules which would trigger early stopping. Figure 3 shows histograms of performances relative to the Holthaus rule. The performance of SPT (dotted line), ERD (dashed line), and random (dash-dotted line) rules are also indicated. The black bars in the histogram show the distribution of mean flowtime for the complete set of 10,000 rules created. All of these rules were simulated until the end. The gray bars show only the subset of those rules which would be simulated until the end during optimization. As can be seen, almost no rule performing better than random is stopped prematurely, whereas a high percentage of rules with very bad performance is detected effectively.

The fraction of rules that is better than random is shown in the column Fraction < Random in Table 4. Interestingly, it is around 40%. This means a randomly generated rule is likely to perform worse than a rule that makes random decisions. The reason is probably that a randomly generated rule makes deterministic decisions, while the random rule makes decisions randomly, independently from each other. This avoids systematic errors such as keeping a job waiting forever.

In general, distributions of the objective values are strongly skewed with a long right tail. This can be partly seen from the histograms in Figure 3; although, to improve clarity, they were cut off at about eight times the performance of the Holthaus rule, whereas worse values encountered a range well above 20. This fraction of very bad rules is significantly higher for the basic attribute set, as can be seen in the column Very bad in Table 4. Using the extended attribute set has a moderating effect; rules with extreme values are less frequent. A similar effect can be observed comparing the tree representation with linear and neural network. With tree representation, the fraction of
Figure 4: Convergence curves for linear representation. Basic (empty shapes) or extended attribute set (filled shapes) is indicated by shape. Solid curves represent the use of normalized values, and dashed curves were obtained using raw values.

very good rules (better than SPT) is considerably smaller, but also very bad rules occur less frequently.

Summarizing Table 4 and Figure 3, the results indicate the linear and neural network representations to be similar and the tree representation to be different from these two. The fraction of rules performing better than random, and in particular better than SPT (column Fraction < SPT of Table 4), is considerably smaller for the tree representation than the linear and neural network representations. Obtaining good rules using random search is easier for linear and neural network representations than it is for the tree representation.

5.2 Linear Rule Optimization

The average performance of rules obtained using the linear representation, depending on the number of objective function evaluations, is shown in Figure 4. As can be seen, normalization leads to faster convergence. This is especially true for the extended attribute set, where the differences in scale between the different measures are very large. When the number of objective function evaluations is large, the performance of the different rules become very similar.

Table 5 reports whether performance differences observed at the end of the run (after 30,000 objective function evaluations, rightmost point in Figure 4) are statistically different according to a Wilcoxon signed rank test. Using the extended information set yields a statistically significant improvement in rule performance; however, the difference is just between 0.961 and 0.959 or less than 3 min if expressed in absolute numbers.

For comparison, the Holthaus rule, which is also a (manually designed) linear combination of the input measures, achieves a result 51 min worse than the mean performance of the optimized linear rule using the basic attribute set. Thus, using CMA-ES as an optimization algorithm and a moderate computational budget we can considerably improve rule performance even without using additional information or a different functional form.
Table 5: Absolute differences in mean performance of the linear representation based on 50 runs of each configuration with 30,000 objective function evaluations each. A negative value indicates the setting in the row is better (i.e., smaller mean) than the setting given by the column header. Statistically significant results are highlighted in bold.

<table>
<thead>
<tr>
<th></th>
<th>LIN_BASE_NORM</th>
<th>LIN_EXT</th>
<th>LIN_EXT_NORM</th>
</tr>
</thead>
<tbody>
<tr>
<td>LIN_BASE</td>
<td>0.2 (o)</td>
<td>1.1 (++)</td>
<td>2.5 (++)</td>
</tr>
<tr>
<td>LIN_BASE_NORM</td>
<td>1.3 (++)</td>
<td></td>
<td>2.7 (++)</td>
</tr>
<tr>
<td>LIN_EXT</td>
<td>1.4 (+)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 5: Convergence curves for tree representation using (a) the basic attribute set, and (b) the extended attribute set. In both figures, the solid curve shows the mean performance without normalization or duplicate suppression. The dashed curve results from using no normalization but suppression of duplicates. The dashed-dotted curve uses normalized inputs (no suppressed duplicates). The dotted line uses normalization and suppressed duplicates.

5.3 Tree-Based Rule Optimization

Convergence curves of our experiments using the tree representation are shown in Figure 5(a) for the basic attribute set, and in Figure 5(b) for the extended attribute set. Each graph shows four curves averaged over 50 independent runs each, running the experiments with and without normalization of input values, and using suppression of duplicates or not. The combined use of suppression of duplicates (which enhances the efficiency of the optimization algorithm), as well as normalization (which is a representation parameter), performs best for both attribute sets. Using only normalization or only suppression of duplicates is not as good, but still better than without these enhancements. Table 6 shows absolute performance differences and the results of statistical tests comparing the performances after 30,000 objective function evaluations. Settings using suppression of duplicates are indicated by "_ND" as an abbreviation for "no duplicates" at the end of their names. As an example TREE_BASE_NORM _ND indicates a run using tree representation, basic attribute set, normalization of attributes, and suppression of duplicates.

Using the basic attribute set we can achieve an improvement from 0.922 (worst setting) to 0.907 (best setting); results using the extended attribute set improve from 0.929 to 0.901. It is interesting to note that without using normalized inputs and suppression
Table 6: Absolute differences in mean performance of the tree representation based on 50 runs of each configuration with 30,000 objective function evaluations each. A negative value indicates the setting as given in the first column of each row is better (i.e., smaller mean) than the setting given by the column header. Statistically significant results are highlighted in bold.

<table>
<thead>
<tr>
<th></th>
<th>TREE_BASE</th>
<th>TREE_BASE_ND</th>
<th>TREE_BASE_NORM</th>
<th>TREE_EXT</th>
<th>TREE_EXT_ND</th>
<th>TREE_EXT_NORM</th>
<th>TREE_EXT_NORM_ND</th>
</tr>
</thead>
<tbody>
<tr>
<td>TREE_BASE</td>
<td>13.8 (+++)</td>
<td>12.8 (+++)</td>
<td>19.5 (+++)</td>
<td>14.7 (+++)</td>
<td>16.3 (+++)</td>
<td>28.3 (+++)</td>
<td></td>
</tr>
<tr>
<td>TREE_BASE_ND</td>
<td></td>
<td>-1.1 (o)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>TREE_BASE_NORM</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>TREE_BASE_NORM_ND</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>TREE_EXT</td>
<td>-28.2 (+++)</td>
<td>-4.8 (+++)</td>
<td>-3.3 (o)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>TREE_EXT_ND</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>TREE_EXT_NORM</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>TREE_EXT_NORM_ND</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

of duplicates, the extended attribute set performs worse than the basic terminal set, while the opposite is true when these techniques are used. A part from the apparently higher importance of normalization when combining information from very different attributes in the extended attribute set, it also seems that a more efficient search with duplicate avoidance is important because of the larger search space that comes with the extended set.

5.4 Neural Network Rule Optimization

In order to investigate the effect of network complexity, we tested the neural network representation with different numbers of neurons on the hidden layer. Convergence curves of these experiments are shown in Figure 6 and performance differences as well as results of statistical significance tests comparing the mean performances after 30,000 objective function evaluations are shown in Table 7.

The results indicate that optimization performance depends more on the set of attributes used than on the number of hidden neurons. Especially in the case of the extended attribute set after 30,000 objective function evaluations and even 100 independent optimization runs, performance differences between 7, 14, and 28 hidden neurons are not very significant. Apparently, the possibility of representing priority rules of higher complexity is not needed, or is offset by the resulting higher dimensionality of the search space (cf. Table 2).

The results for the basic attribute set identify 14 hidden nodes as the best setting with a mean performance of 0.908, however also showing little sensitivity to a further increase in the number of hidden neurons to 28 (performance 0.909). Reducing the number of neurons on the hidden layer to seven resulted in a decreased performance to 0.911 and reducing it further to three (not shown in Figure 6) yields a performance of 0.917.

Given these results, we chose 14 as the number of hidden neurons for both the basic as well as extended attribute sets, as it yielded the lowest mean performance over 100 runs and was not significantly worse than the other settings investigated.
Figure 6: Convergence curves for neural network representation. Basic (empty shapes) or extended attribute set (filled shapes) is indicated by shape. In both cases dotted (solid, dashed) lines represent 7 (14, 28) neurons on the hidden layer.

Table 7: Absolute differences in mean performance of the neural network representation based on 100 runs of each configuration with 30,000 objective function evaluations each. A negative value indicates the setting as given in the first column of each row is better (i.e., smaller mean) than the setting given by the column header. Statistically significant results are highlighted in bold.

<table>
<thead>
<tr>
<th></th>
<th>NN_BASE_H7</th>
<th>NN_BASE_H14</th>
<th>NN_BASE_H28</th>
<th>NN_EXT_H7</th>
<th>NN_EXT_H14</th>
</tr>
</thead>
<tbody>
<tr>
<td>NN_BASE_H3</td>
<td>8.5 (+++)</td>
<td>12.2 (+++)</td>
<td>11.6 (+++)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>NN_BASE_H7</td>
<td></td>
<td>3.7 (+++)</td>
<td>3.1 (+++)</td>
<td>NN_EXT_H7</td>
<td>0.6 (o)</td>
</tr>
<tr>
<td>NN_BASE_H14</td>
<td>-0.6 (+++)</td>
<td>NN_EXT_H14</td>
<td>-1.0 (+++)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

5.5 Comparison

Figure 7 compares the performances of the best settings for each representation as identified in Sections 5.2 to 5.4, and shows their convergence curves. Additionally, Figure 8 shows boxplots visualizing the performance distribution after 30,000 objective function evaluations for all settings investigated in the previous sections.

The linear representation clearly results in the worst solutions, with the mean performance not surpassing about 0.96 of the mean flowtime obtained with the Holthaus rule, while the neural network and tree representations obtain solutions of about 0.9. As can be seen in Table 8, comparing the 100 optimization runs for each setting, final performance of the tree representation is significantly better than the performance of the neural network representation for both the extended and the basic attribute set. But this difference, although statistically significant, is small: 0.907 versus 0.908 for the basic attribute set, or 0.903 versus 0.901 for the extended attribute set. In absolute figures this is a difference in mean performance of 1.1 min, or 2.7 min, respectively.
Figure 7: Convergence curves for normalized versions of linear (triangles), tree (squares), and neural network (circles) representation using basic (empty shapes) or extended attribute sets (filled shapes).

Figure 8: Boxplot of performance distributions after 30,000 objective function evaluations. Plots are based on 50 runs for LIN, TREE, and TREE_NORM; all other plots show data from 100 independent runs.

Table 8: Absolute differences in mean performance between tree and neural network representations based on 100 runs of each configuration with 30,000 objective function evaluations each. A negative value indicates the setting as given in the first column of each row is better (i.e., smaller mean) than the setting given by the column header. Statistically significant results are highlighted in bold.

<table>
<thead>
<tr>
<th></th>
<th>TREE_BASE_NORM_ND</th>
<th>NN_EXT_H14</th>
<th>TREE_EXT_NORM_ND</th>
</tr>
</thead>
<tbody>
<tr>
<td>NN_BASE_H14</td>
<td>1.1 (++)</td>
<td>7.2 (++)</td>
<td>9.9 (++)</td>
</tr>
<tr>
<td>TREE_BASE_NORM_ND</td>
<td>6.1 (++)</td>
<td>8.7 (++)</td>
<td></td>
</tr>
<tr>
<td>NN_EXT_H14</td>
<td></td>
<td>2.7 (++)</td>
<td></td>
</tr>
</tbody>
</table>
In terms of convergence, the linear representation clearly is fastest, and after about 100 fitness evaluations for the basic attribute set and about 500 evaluations using the extended attribute set, performance stops improving. While the final quality difference between neural network and tree representation is relatively small, convergence of the neural network representation is much faster and very good rules are found much more quickly than with the tree representation. This is examined more closely in the following section.

5.6 Algorithm Reliability

Apart from mean performance, another important performance measure is reliability, or probability to reach a certain target performance. In this section we assess the reliability of the optimization process, again using the best settings identified for each representation.

One useful definition of success in the context of our paper is achieving a performance better than the Holthaus rule. Analyzing the 100 optimization runs using this measure of success produces the graph shown in Figure 9. After just 100 objective function evaluations all optimization runs with the linear and neural network representation had generated a rule better than the Holthaus rule. As we use a population size of 500 in our genetic programming experiments, the graph for the tree representation shows only the performance of a random search performed when randomly filling the initial population. The fact that GP has a reliability of 99% after 500 objective function evaluations also means that 99 out of 100 genetic programming runs already started with a solution better than the Holthaus rule in their initial population.

To obtain harder target values to define success but without knowing the real optimum of achievable performance, we normalized performance to between 100% for the best mean flowtime achieved by any of the representations, and 0% for the performance of the Holthaus rule. The best rules used to define 100% optimization potential achieved a performance of 0.902 using the basic (improvement of 128 min), and 0.892 for the extended attribute sets (improvement of 141 min). Both of these results were obtained using the tree representation.

Figure 9: Number of runs (out of 100) achieving a performance better than the Holthaus rule using only a small budget (triangles: LIN_BASE_NORM, circles: NN_BASE_H14, squares: TREE_BASE_NORM_ND).
Figure 10 gives an indication of search reliability, showing the performance levels reached by 50 (i.e., median performance) of the 100 runs conducted for linear, neural network, and tree representation (with duplicate avoidance), each based on normalized attribute values. The gray areas in the figure indicate the performance range encountered during the experiments by plotting the 10% and 90% quantiles as well. The conclusions are similar to those derived from the mean performance values: The linear representation yields good results from the start, but quickly stagnates. Neural networks and the tree representation obtain similar but much better solutions at the end, with the neural network representation converging much faster.

While the above comparisons are based on the number of fitness evaluations, it should be noted that evaluating a solution involves simulating the shop floor and calling the priority rule thousands of times. Thus, the computational efficiency for computing the priority value has a significant influence on the overall runtime of the algorithm. Comparing runtimes, however, is not easy, as they depend to a large extent on implementation details. For example, the computational time for a neural network evaluation depends to a large extent on the ability to approximate the tanh function efficiently. The computational time for a tree evaluation depends on the size of the tree, so bloat control (which we did not use) might have a noticeable impact on runtime. For our implementation and setup, we observed that a run using the neural network representation took about 2–3 times as long as a run using the tree representation. So, from a runtime perspective, the faster convergence of the neural network representation would be less pronounced, but still significant. That is, the general conclusions derived above would still hold.

5.7 Visualization

One obstacle to the widespread use of dispatching rules automatically generated with hyper-heuristics is that the resulting rules are often very complicated and cannot be understood intuitively. In order to gain insight into why optimized rules in the different representations perform so well in comparison with standard rules, we plotted the functions encoded by various rules. This has the advantage of being independent of
J. Branke, T. Hildebrandt, and B. Scholz-Reiter

Figure 11: Visualization of priority functions encoded by exemplary rules (one rule for each column). The $x$ axis always shows the processing time ([1, 49]), and the $y$ axis is the processing time on the next machine ([0, 49]), and each row corresponds to a certain level of work in the next queue (WINQ) as indicated on the right. In the online versions, colors show the priorities assigned to each combination of processing time, next processing time, and work in the next queue, ranging from black (lowest priority) to white (highest priority). The lines in the graphs connect points of equal priority.

We therefore analyzed rules using the basic attribute set (the three variables processing time, processing time on the next machine, and level of work in next queue). The results are shown in Figure 11 visualizing the original Holthaus rule, an optimized linear rule (performance 0.959), the best rule found using the TREE_BASE_NORM_ND setting (performance 0.902), and the best neural network rule (performance 0.902).

Comparing the Holthaus rule with the optimized linear rule, the main difference can be identified: priorities decrease much slower with an increasing WINQ-level in case of the optimized rule. Weights in the Holthaus rule are (–2, –1, –1), whereas the optimized rule can be expressed using the weights (–2, –0.27, –0.86).

The performance of this linear rule can be further improved using the tree or neural network representations with the possibility of expressing nonlinear functions. Exemplary rules of these representations are visualized in columns 3 and 4 of Figure 11. As can be seen, the slope of the curves of equal priority changes with an increasing WINQ level. Contrary to the linear rules they have a positive slope in large parts of the images, if the WINQ level is low. Jobs with more work for lightly utilized next machines are preferred, especially if they can be finished quickly on the current machine. With an increasing WINQ level the slope becomes negative as in the linear case, which means shorter (next) processing times are preferred to longer times.

the complexity/representation of the functions involved, but only works for a small attribute set, that is, a small number of variables.
A further point especially visible for the tree as well as the neural network rule is an area of high priority on the left side, and in particular in the lower left corner. This high priority is independent of the WINQ level, that is, for very short jobs on both the current and next machine. This not only prioritizes short jobs of the current machine, but also implicitly takes into account the fact that such a job can quickly be finished on the next machine, as it will also receive a high priority there.

5.8 Rule Analysis

As stated in the previous section, analyzing and comparing rule behavior by visualization is limited to just a few attributes and only a few rules. To gain further insight into rule behavior for the extended attribute set (i.e., a function of seven variables) across representations, we performed an additional analysis regarding the value of each attribute for each representation as well as an analysis of decisions taken by the rules to assess differences and similarities between the different representations.

We first analyzed the importance each attribute has in a generated rule. For this purpose, we took the final rules for all three representations using the extended attribute set and investigated their performance when certain attributes are obscured by replacing their values with random values for each decision situation. The results are shown in Figure 12. As can be seen, performance suffers most if any of the three attributes PT, WINQ, or NPT is obscured. These attributes are also used in the Holthaus rule, therefore it seems the authors of this rule have correctly identified the most important information in their manually developed rule.

Comparing the three different representations, some differences become visible. First, the most important attribute of the linear representation is PT, the processing time of a job’s current operation. While this information is also important for the neural network and tree representations, their performance deteriorates even stronger if information on the work in next queue (WINQ) is missing.
Table 9: Mean number of different decisions averaging pairwise comparisons of each of the 100 rules in each group as well as showing the average number of different decisions of rules in each group compared with the Holthaus rule.

<table>
<thead>
<tr>
<th>TREE_EXT_NORM_ND</th>
<th>LIN_EXT_NORM</th>
<th>NN_EXT_H14</th>
<th>Holthaus</th>
</tr>
</thead>
<tbody>
<tr>
<td>13.4</td>
<td>23.4</td>
<td>18.5</td>
<td>45.7</td>
</tr>
<tr>
<td>6.0</td>
<td>19.2</td>
<td>37.5</td>
<td></td>
</tr>
<tr>
<td>15.0</td>
<td>41.2</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Second, the attribute time in system (TIS) has a high importance for the linear representation (about the same as NPT). This is different from the more flexible representations, which show much less (neural network) or almost no (tree) reaction to the TIS attribute missing.

Third, there is a higher variance of results of the tree representation (see the error bars in Figure 12), especially for the three most important attributes.

In another analysis, we examined the rules’ behavior in a certain fixed set of 100 decision situations with at least three jobs waiting, sampled in runs with the Holthaus rule. We then apply a pair of rules to each of these decision situations and count the number of times they would select different jobs, yielding a number of 0 if all decisions are the same or 100 if the two rules would select different jobs in each situation. This gives us a measure for the behavioral difference between rules in actual decision situations. Now we take 300 rules, 100 generated with each representation, and perform pairwise comparisons between all rules of all groups. Average results of all such groupwise comparisons are shown in Table 9.

These numbers indicate that the group of linear rules is the most homogeneous with on average six decisions taken differently, compared to 15.0 and 13.4 for the ANN and tree representation, respectively. For all groups, differences within each group are smaller than between groups. This seems to indicate that different representations lead to different kinds of rules that differ not only syntactically, but also in behavior. Furthermore, all the rules take very different decisions from the Holthaus rule, as can be seen in Column 5.

5.9 Rule Robustness

Optimizing a rule based on a particular scenario bears the risk of overfitting: The rule works well in this scenario, but not in others. Therefore, all results reported here are based on testing the evolved rules on several additional simulations with different random seeds. We could not observe overfitting as longer running times generally lead to better results also on the test set. Thus, we conclude that during a simulation run, the rules are exposed to many different situations, and have to work well across all of them to survive. Note, however, that so far the main characteristics of the job shop used for training and testing, such as job arrival rate or number of machines, were identical.

Of course, should the scenario change fundamentally, the rules may no longer perform well. But in such cases it would be easy to restart the optimization process and produce a new rule. Some smaller changes, however, should be tolerated by the rules. One such change could be a variation of the system’s utilization level, because the number of jobs increases or decreases in reaction to changed customer demand. Figure 13 shows the results of such experiments. These graphs were obtained by using
Figure 13: Results of robustness experiments. All performances are relative to the values obtained with the Holthaus rule. (a) Lines with triangles stand for the linear representation, circles represent neural networks, and squares mark tree representations. If the symbols are filled, the representation uses the extended terminal set, otherwise the basic set is used. Each line shows the average performance of the best-of-run rules of each representation after 30,000 objective function evaluations, that is, mean performance of the 100 best-of-run rules. (b) The relative performance of the standard rules SPT (solid line), FIFO (dashed line), and ERD (long dashes).

utilization levels between 0.79 and 0.99 (varied in steps of 2%). Figure 13(b) shows the performance of the three standard rules SPT, FIFO, and ERD relative to the Holthaus rule. As can be seen, if utilization is increased toward 99%, the relative performance of SPT deteriorates, whereas the performance of the FIFO and ERD rules actually improves (albeit still being worse than SPT or the Holthaus rule).

The graphs in Figure 13(a) show mean values of the 100 best-of-run rules of the three representations of both extended and basic attribute sets. Please recall that a utilization level of 95% was used to learn these rules. Irrespective of the settings used, optimized rules can maintain a better performance than the Holthaus rule. There is an interesting difference, however, in the rules using the basic and extended attribute sets. Performance of the latter actually improves with increasing utilization, whereas rules using the basic attribute set get worse (or stay at about the same performance level in the case of NN_BASE_H14). As the results of the standard rules indicate, the attributes "time in system" (underlying ERD) and "time in queue" (underlying FIFO) can be used to improve mean flowtime performance for utilization levels above about 95%. While the effect of using this information is only comparably small at 95%, how to use this information is already incorporated in the rules, so some of the performance improvements of FIFO and ERD at very high utilization levels can be gained by the optimized rules. An adverse effect seems to happen at low utilization levels; here the use of the additional information leads to a performance worse than the basic terminal set. This effect is much smaller, however.

6 Conclusion and Outlook

In this paper we empirically compared three different representations and two attribute sets for the automated design of dispatching rules in a dynamic, stochastic job shop scenario.
Looking just at the final solution quality, the tree representation could be identified as the winner, very closely followed by the neural network representation. The linear representation yields clearly inferior results. For linear and tree representation, normalization of attributes improves search performance. For the neural network representation it is mandatory.

Which representation to choose, however, largely depends on the computational budget available, that is, the number of simulation runs. For very small budgets, the linear representation leads to the best results. In our scenario already 100 objective function evaluations were sufficient to reliably obtain rules substantially better than standard and manually developed rules from the literature.

The tree representation is the most flexible representation investigated in this paper. To effectively optimize tree structures, genetic programming requires a relatively large population size, which in turn causes slow convergence to near-optimal rules. Based on preliminary experiments, we could identify a large number of functionally equivalent rules as a source of inefficiency. A new duplicate avoidance mechanism allowed us to eliminate duplicates without requiring time-consuming simulation runs and significantly improved search performance. With this enhancement, genetic programming and the tree representation reached the best performance levels given a sufficiently high computational budget.

Performance of the neural network representation was competitive, however. Although outperformed by the tree representation for very large computational budgets, performance differences were small. More importantly, the neural network representation obtained a very good solution quality much faster. Performance levels very quickly became better than the linear representation, and could substantially improve upon the performance of linear rules. This makes it a very good choice if only a moderate computational budget is available. It would also be the method of choice to learn dispatching rules for very complex scheduling scenarios, for which even a single simulation run is computationally demanding and thus the number of fitness evaluations that can be performed in a reasonable time is very limited.

The generated rules were quite robust against small to moderate changes of the underlying job shop scenario. A visual inspection of some evolved priority functions revealed their inner workings.

There are several ways to extend the work presented in this paper, such as investigating further representations and other optimization algorithms. Promising candidates would be to test Cartesian genetic programming (Miller and Thomson, 2000) as a representation similar to the tree representation, or recent neuroevolution algorithms like CoSyNE (Gomez et al., 2008) to optimize the weights of neural networks. Last but not least, the approach used here can easily be extended to more complex scheduling scenarios and different objective functions.

Given the good performance of the neural network representation, it might be worthwhile to explore their use also in hyper-heuristic approaches for other problem domains. Examples of problem domains frequently solved with genetic programming include bin packing or SAT solving (Burke et al., 2009). A different search space allows the use of additional optimization techniques, which might offer new opportunities to better solve these problems as well.

Another area for future research would be to improve the way of dealing with the stochasticity of the scheduling scenario. Several parameters can be adjusted to influence the accuracy and runtime of a single objective function evaluation: length of a single simulation run, number of simulation runs for each objective function evaluation, and
the proper use of common random numbers (CRN; Law, 2007). Interactions of these parameters with different representations and optimization algorithms could therefore be investigated further.

References


