24th European Conference on Artificial Intelligence - ECAI 2020
Santiago de Compostela, Spain

Exhaustive search of priority rules for on-line scheduling

Francisco J. Gil-Gala[2] and **Carlos Mencía**[2] and **María R. Sierra**[2] and **Ramiro Varela**[1,2]

Abstract. Real-life scheduling problems often require computing solutions on-line, due to real-time requirements. In this scenario, greedy algorithms guided by priority rules are of common use. This is the case of the problem of scheduling jobs on a machine with variable capacity and total tardiness minimization, denoted $(1, Cap(t)||\sum T_i)$. Recent work proposed a Genetic Programming (GP) approach for evolving priority rules for this problem, outperforming several well-known classical rules. In this paper, we consider state space search as an alternative framework and propose a new method based on a refined exhaustive enumeration of priority rules. Our approach, termed Systematic Search and Heuristic Evaluation (SSHE), integrates powerful pruning techniques and an efficient heuristic procedure used to evaluate candidate rules. Experimental results indicate that SSHE represents a valuable alternative to GP.

1 Introduction

One-machine scheduling problems play a central role in scheduling. Besides modeling a variety of real-life settings, these problems often arise as relaxations of other problems [4] or may appear as a result of a decomposition process when solving other, more complex, problems [1]. This paper deals with a problem of the last class introduced in [13] in the context of scheduling the charging times of a fleet of Electric Vehicles (EV). In this problem, a number of jobs must be scheduled on a single machine, whose capacity varies over time, and with the goal of minimizing the *total tardiness* objective function. According to the standard $\alpha|\beta|\gamma$ notation proposed in [11], it is denoted as $(1, Cap(t)||\sum T_i)$. Due to real-time requirements, this problem needs to be solved quickly, in an on-line fashion. In this setting, the use of a schedule builder guided by efficient priority rules represents a suitable approach.

The aim of this paper is the automated development of priority rules for the $(1, Cap(t)||\sum T_i)$ problem; more concretely, priority rules adapted to specific problem instance distributions. Recent work proposed a Genetic Programming (GP) approach [9, 10] which was shown able to evolve priority rules outperforming several classical ones from the literature. However, due to its stochastic nature, GP may not always converge to high-quality solutions. As an alternative, we consider systematic search as a means to generating effective priority rules. Our approach, termed *Systematic Search and Heuristic Evaluation* (SSHE), is based on a refined exhaustive enumeration of priority rules. The efficiency of the proposed method relies on the use of powerful pruning techniques, that allow for reducing the search space by discarding redundant and useless rules, as well as on a heuristic procedure used to evaluate rules efficiently. Experimental results indicate the practical suitability of SSHE, constituting a valuable alternative to GP on search spaces of reasonable size.

The remainder of the paper is organized as follows. In the next section the problem $(1, Cap(t)||\sum T_i)$ is defined. Section 3 reviews existing methods proposed in the literature to solve this problem and establishes the working hypotheses. In Section 4, we describe the proposed SSHE method. Section 5 reports the results of an experimental study. Finally, Section 6 summarizes the main conclusions of the paper and outlines some lines for future research.

[1] Contact Author

[2] Department of Computer Science, University of Oviedo, Gijón 33204, Spain, email:{giljavier, menciacarlos, sierramaria, ramiro}@uniovi.es

2 The $(1, Cap(t)||\sum T_i)$ problem

The $(1, Cap(t)||\sum T_i)$ problem derives from the Electric Vehicle Charging Scheduling Problem (EVCSP) introduced in [13]. However, this problem could model any other setting where a single machine is able to perform a number of tasks at a time, being its capacity variable over time. In [13], a number of electric vehicles are distributed over three charging lines and at each scheduling point a number of instances of the $(1, Cap(t)||\sum T_i)$ problem may arise in each line, which must be solved on-line. For the sake of space, and to keep the focus on the $(1, Cap(t)||\sum T_i)$ problem, we do not describe the EVCSP herein and refer the interested reader to [13] for further details.

2.1 Problem definition

The $(1, Cap(t)||\sum T_i)$ problem is defined as follows. We are given a number of n jobs $\{1, \ldots, n\}$, all of them available at time $t = 0$, which have to be scheduled on a machine whose capacity varies over time, such that $Cap(t) \geq 0, t \geq 0$, is the capacity of the machine in the interval $[t, t+1)$. Job j has duration p_j and due date d_j. The goal is to allocate starting times $st_j, 1 \leq j \leq n$ to the jobs on the machine such that the following constraints are satisfied:

i. At any time $t \geq 0$ the number of jobs that are processed in parallel, $X(t)$, cannot exceed the capacity of the machine, i.e.,
$$X(t) \leq Cap(t). \tag{1}$$

ii. The processing of jobs on the machine cannot be preempted, i.e.,
$$C_j = st_j + p_j, \tag{2}$$
where C_j is the completion time of job j.

The objective function is the total tardiness, defined as:
$$\sum_{i=1,\ldots,n} \max(0, C_i - d_i) \tag{3}$$
which should be minimized.

As an example, Figure 1 shows a feasible schedule for a problem instance with 7 jobs.

The identical parallel machines problem, denoted $(P||\sum T_i)$ in [17], known to be NP-hard, can be reduced to the $(1, Cap(t)||\sum T_i)$ problem, so this problem is NP-hard as well.

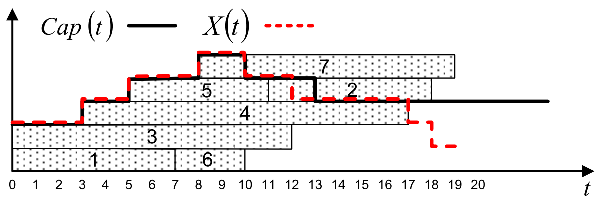


Figure 1: A feasible schedule for an instance of the $(1, Cap(t)||\sum T_i)$ problem with 7 jobs and a machine with capacity varying between 2 and 5. For clarity, due dates are omitted.

3 Review of the current solving methods and working hypotheses

In [13], the $(1, Cap(t)||\sum T_i)$ problem was solved on-line by means of a non-deterministic schedule builder guided by priority rules adapted from the literature. More recently, an off-line solution was proposed in [19] by means of a memetic algorithm that, as expected, reaches much better solutions at the expense of taking much more time. The differences between the solutions obtained by the methods above motivated the Genetic Programming (GP) approach proposed in [10] to obtain new priority rules, which were shown to perform much better than the classical ones. In this section, we review the mentioned schedule builder and the priority rules, both the classical ones and those evolved by GP.

3.1 Schedule builder

A schedule builder is a non-deterministic algorithm that provides a way for enumerating a subset of the feasible schedules, thus enabling the definition of a search space for a given scheduling problem. Algorithm 1 shows a schedule builder for the $(1, Cap(t)||\sum T_i)$ problem taken from [10]. In this algorithm US represents the set of unscheduled jobs at a given time, and $X(t)$ denotes the consumed capacity of the machine due to the jobs scheduled so far. In each iteration, the algorithm schedules one unscheduled job among the ones that can start at the earliest possible time, denoted $\gamma(\alpha)$.

As an example, the schedule in Figure 1 can be built by this algorithm following the sequence of choices $(1, 3, 4, 5, 6, 7, 2)$. The search space defined by this scheduler is dominant, i.e., there always is a sequence of choices that produces an optimal solution.

3.2 Classic priority rules

A *Priority Rule* (PR) is a "a simple heuristic that derives a priority index of a job from its attributes" [2]. PRs may be combined with schedule builders as Algorithm 1 so the job with the highest priority in US^* is scheduled at each iteration. This way, we obtain an algorithm suitable for on-line scheduling.

Several well-known rules in the literature may be adapted to the $(1, Cap(t)||\sum T_i)$ problem; among others, the simple *Earliest Due Date* (EDD) and *Shortest Processing Time* (SPT) rules; which calculate priorities for an eligible job j as $\pi_j = 1/d_j$ and $\pi_j = 1/p_j$ respectively, as well as more sophisticated rules as the *Apparent Tardiness Cost* (ATC) rule [21, 15], which calculates

$$\pi_j = \frac{1}{p_j} exp\left[\frac{-max(0, d_j - \gamma(\alpha) - p_j)}{g\bar{p}}\right], \quad (4)$$

where $\bar{p}$ is the average processing time of the unscheduled jobs and g is a look-ahead parameter to be introduced by the user.

Algorithm 1 Schedule Builder

```
**Data:** A $(1, Cap(t)||\sum T_i)$ problem instance $\mathcal{P}$.
**Result:** A feasible schedule $S$ for $\mathcal{P}$.
 1: $US \leftarrow \{1, 2, ..., n\}$;
 2: $X(t) \leftarrow 0, t \geq 0$;
 3: while $US \neq \emptyset$ do
 4:     $\gamma(\alpha) = min\{t' | X(t) < Cap(t), t \in [t', t' + p_u), u \in US\}$
 5:     $US^* \leftarrow \{u \in US | X(t) < Cap(t), t \in [\gamma(\alpha), \gamma(\alpha) + p_u)\}$
 6:     Non-deterministically pick job $u \in US^*$;
 7:     $st_u \leftarrow \gamma(\alpha)$;
 8:     $X(t) \leftarrow X(t) + 1, t \in [st_u, st_u + p_u)$;
 9:     $US \leftarrow US - \{u\}$;
10: return The schedule $S = (st_1, st_2, ..., st_n)$;
```

3.3 Learning priority rules

Classical PRs are usually defined by hand by experts from the knowledge or intuitions they may have about a given problem. For this reason, they are often easily understandable, but they may not capture non-trivial characteristics of the problem domain that are not evident to the expert eye. Therefore, learning arises as an alternative method.

In this context, GP is a common technique to learning rules for problems such as job shop [12, 14], one machine [6], unrelated parallel machines [8], or resource constrained project scheduling problems [5, 7], among others. These approaches are based on the framework proposed by Koza in [18], adapting it to the particular problem, which requires taking two main decisions; establishing first a set of terminal and function symbols, and then defining a grammar to build expression trees. Hybridizations between GP and other algorithms have also been considered. For example, Nguyen et al. [20] proposed a hybrid genetic programming algorithm based on applying a local improvement algorithm to the rules calculated by GP. Burke et al. in [3] classify all these approaches as *heuristic generation*, specifically these methods are named *GP-based hyper-heuristics*.

To improve readability, Keijzer and Babovic introduced the notion of dimensionally aware rules [16], which were used in other works such as [8] and [10]. In the latter, GP was proposed to evolve priority rules for the $(1, Cap(t)||\sum T_i)$ problem. The set of symbols used was similar to that given in Table 1, which is the one we consider herein. This set includes the basic operators and some arithmetic functions, four attributes from the problem domain and 9 constants. Regarding the grammar, the approach in [10] considered the possibility of generating any dimensionally compliant well-formed arithmetic expression, also considered in this work.

Table 1: Functional and terminal sets used to build expression trees. Symbol "-" is considered in unary and binary versions. max_0 and min_0 return the maximum and minimum of an expression and 0.

Binary functions	-	+	/	$\times$	max	min	
Unary functions	-	pow_2	$sqrt$	exp	ln	max_0	min_0
Terminals	p_j	d_j	$\gamma(\alpha)$	$\bar{p}$	0.1	...	0.9

The GP presented in [10] was designed from all the considerations above. In an experimental study, each evolved rule was evaluated over a set of 50 training instances and then the best rules from 30 independent runs were evaluated over a large set of unseen testing instances. The evolved rules were shown to outperform SPT, EDD and ATC with any value of g in $\{0.1, \ldots, 1.0\}$. Even when the depth of the trees is restricted to a small value as 4, not only the best but also the average rule outperform ATC with any value of g.

Table 2: Summary of results obtained by GP proposed in [10] with different values of maximum depth and size. Best and average results from 30 runs are reported. Avg. Size is the average size of the best rules obtained in 30 independent runs. The Best value in testing is that obtained by the best rule in training.

Max.	Max.	Training		Testing		Avg.
Depth	Size	Best	Avg.	Best	Avg.	Size
3	7	1034.04	1049.89	1022.61	1043.09	6
4	15	997.18	1012.29	1000.13	1015.42	10
5	31	989.18	1000.68	995.37	1005.39	19
6	63	986.32	996.80	992.14	1001.64	21
7	127	986.82	998.39	994.66	1003.02	24
8	255	986.92	1000.80	995.09	1006.44	29

Table 2 shows a summary of the results obtained by GP [10] considering different values of maximum size and depth of the evolved trees. As we can see, the best values correspond to a maximum depth of 6 levels; being worse for both smaller and larger values. In particular, the quality of the rules gets slightly worse as long as the allowed depth increases, and the size of the rules grows at the same time. These results indicate that in these cases the search space is so large that GP is not able to converge to better solutions.

On the other hand, when the search is restricted to rules of maximum depth of 3 or 4 levels the reached rules may be worse due to the search space not containing better rules. However, given the reasonable size of the search space for these depth values, exhaustive search could be an alternative to a stochastic method as GP and so the last conjecture could be clarified. This is the proposal of this work, which is developed in the next section.

4 Calculating priority rules as state space search

We propose to systematically generate all the priority rules that can be built from a given set of symbols, a grammar and some constraints on the size and depth of the expression trees. This method can be expected to be of practical use for search spaces of reasonable size. Limiting the size of the search space brings a number of advantages, as guaranteeing small expression trees. In addition, the best rule in this space may be better than the rules evolved by GP searching over larger spaces. In any case, if the number of candidate rules is large, evaluating all of them on a set of 50 instances of the $(1, Cap(t)||\sum T_i)$ problem, as done by GP in [10], could be impractical. In this case, the use of a surrogate method may be suitable to speed up the evaluation. Furthermore, to keep the effective search space as reduced as possible, we can use strategies to avoid redundant rules or even to discard some rules using knowledge from the problem domain. These techniques could improve the efficiency of the method at the risk of losing optimality in some cases.

As done in [10], we restrict the search to dimensionally compliant expressions. This is reasonable when the expression trees represent arithmetic expressions involving dimensional magnitudes, as it does happen in equations representing physical phenomena. The dimension of the four characteristics considered in Table 1: d_j, p_j, $\bar{p}$ and $\gamma(\alpha)$ is time, denoted t, whereas constants are considered adimensional. This way, dimension compliant rules require that operations $+$, $-$, max or min are only applied to expressions having the same dimension; while operations as $\times$, $/$, pow_2, $sqrt$, max_0 and min_0 may be applied to expressions of any dimension; and exp and ln can only be applied to adimensional expressions. It is easy to see that the expression tree of the ATC rule is dimensionally compliant and that the dimension of the whole expression is t^{-1}, the same as the dimension of both EDD and SPT rules.

The proposed method is termed herein *Systematic Search and Heuristic Evaluation* (SSHE) and it is described in the following subsections. We first describe a representation of priority rules that facilitates the enumeration procedure. Then, we devise an algorithm that generates the set of feasible priority rules restricted to a given maximum size and depth of the expression trees. Besides, we introduce a number of pruning techniques that prevent the algorithm from generating a number of equivalent and useless rules. Finally, we describe the search algorithm and the way generated rules are evaluated.

4.1 Representation of priority rules

Expression trees representing rules are mapped into arrays, whose size is established from the maximum number of elements, $\mathcal{P}$, and the maximum depth, $\mathcal{D}$, allowed to trees. Let $\mathcal{B}$ denote the array representing a priority rule, for convenience we denote the indices of their components as $0, \ldots, \mathcal{S}$. So, $\mathcal{S} + 1$ is the size of the array, which must fulfill $\mathcal{P} \leq \mathcal{S} + 1 = 2^{\mathcal{D}} - 1$.

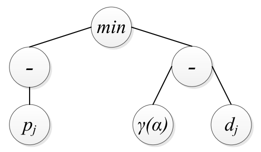
The interpretation of the array content is borrowed from binary heaps implementation. $\mathcal{B}_0$ is the root node, and the remaining positions may either be $NULL$ or contain a terminal or function symbol. If $\mathcal{B}_i$ is not $NULL$ its parent is $\mathcal{B}_{(i-1)/2}$; if it has children, they are $\mathcal{B}_{2i+1}$ (left child) and $\mathcal{B}_{2i+2}$ (right child). If a node only has one child, it is the left one. Therefore, the nodes of the tree are distributed in the array so that nodes at depth $1 \leq k \leq \mathcal{D}$ are in positions $2^{k-1}-1, \ldots, 2^k-2$. In particular, the positions $\mathcal{S}/2, \ldots, \mathcal{S}$ can only contain terminal or $NULL$ symbols, while positions $1, \ldots, \mathcal{S}/2-1$ may contain any terminal or function symbol, or $NULL$; and position 0 can only contain terminal or function symbols.

Figure 2 shows the array representation of an expression tree with size 6 and depth 3.

4.2 The search space of priority rules

The use of arrays to represent priority rules facilitates the process of enumerating and building all feasible rules given $\mathcal{D}$ and $\mathcal{P}$. The expression trees are generated by filling the array from right to left starting in position $\mathcal{S}$. In this process, the terminal, function or $NULL$ symbol inserted in position i must be coherent with the symbols previously inserted in positions $[i+1..\mathcal{S}]$, and it must guarantee not to exceed the maximum size of the rule. Besides, each time an operator, either unary of binary, is inserted in a position of the array, we have to be aware of the dimension of its operands; in fact, this is the reason why the array is filled from right to left.

Algorithm 2 shows the generation procedure for the grammar described in Section 3.3 and the function and terminal symbols given in Table 1 (in the algorithm $\mathcal{C}$ denotes the set of constants). Besides, $[B_k]$ denotes the dimension of the expression tree under position k, in particular if $[B_k] = 1$ the expression is adimensional. To produce a feasible priority rule, this procedure is called initially on an empty



min	-	-	p_j		$\gamma(\alpha)$	d_j
0	1	2	3	4	5	6

Figure 2: Array representation of a priority rule.

Algorithm 2 Grammar generation procedure

```
Data: A state $\mathcal{B}_{i+1..\mathcal{S}}$, $0 \le i \le \mathcal{S}$ and the number $p \le \mathcal{P}$ of non
      null positions in $\mathcal{B}_{i+1..\mathcal{S}}$.
Result: A set $B(i)$ of successor states of $\mathcal{B}_{i+1..\mathcal{S}}$.
 1: if $p = \mathcal{P}$ then
 2:   return $\emptyset$;
 3: if $i \ge \mathcal{S}/2 \vee \mathcal{B}_{2i+1} = NULL$ then
 4:   if $i\%2 \ne 0 \wedge \mathcal{B}_{i+1} \ne NULL$ then
 5:     $\mathcal{B}(i) \leftarrow \{p_i, d_i, \gamma(\alpha), \bar{p}\} \cup \mathcal{C}$;
 6:   else
 7:     $\mathcal{B}(i) \leftarrow \{p_i, d_i, \gamma(\alpha), \bar{p}, NULL\} \cup \mathcal{C}$;
 8: else
 9:   if $\mathcal{B}_{2i+2} \ne NULL$ then
10:     if $[\mathcal{B}_{2i+1}] = [\mathcal{B}_{2i+2}]$ then
11:       $\mathcal{B}(i) \leftarrow \{+, -, max, min, \times, /\}$;
12:     else
13:       $\mathcal{B}(i) \leftarrow \{\times, /\}$;
14:   else
15:     if $[B_{2i+1}] = 1$ then
16:       $\mathcal{B}(i) \leftarrow \{-, pow_2, sqrt, max_0, min_0, exp, ln\}$;
17:     else
18:       $\mathcal{B}(i) \leftarrow \{-, pow_2, sqrt, max_0, min_0\}$;
19: return the set of states $\mathcal{B}(i)$;
```

array, i.e., $i = \mathcal{S}$, and then iteratively from the state returned in $\mathcal{B}(i)$ as long as $i \ge 0$ and the number of non null positions in the state is lower than $\mathcal{P}$. The notation $\mathcal{B}(i) \leftarrow X$ means that one of the symbols in the set X is chosen non deterministically and assigned to the position i of the array $\mathcal{B}$, then the new state $\mathcal{B}_{i..\mathcal{S}}$ is assigned to the set $\mathcal{B}(i)$. This way, we have a greedy algorithm that may finish either in a dead end, if the number of non null symbols in the array reaches $\mathcal{P}$ and $i > 0$, or with $i = 0$ and the array representing a full expression tree. This greedy algorithm may be naturally extended to build a full search tree by just considering that $\mathcal{B}(i) \leftarrow X$ produces a set of successor states of $\mathcal{B}_{i+1..\mathcal{S}}$, one for each of the symbols in X.

With the above, we explain some details of Algorithm 2. The condition $i \ge \mathcal{S}/2 \vee \mathcal{B}_{2i+1} = NULL$ in line 3 expresses that either i is a position in the second half of the array or i is in the first half and its left child is $NULL$ and so its right child must be $NULL$ as well. In either case, the value inserted in i cannot be a functional symbol. Furthermore, if $i\%2 \ne 0 \wedge \mathcal{B}_{i+1} \ne NULL$ in line 4 holds, then i is the left child of node j, with $i = 2j + 1$, and the right child of j is not null, so the position i cannot be $NULL$ and must contain any terminal symbol; otherwise i could contain $NULL$ as well. Otherwise, if $i < \mathcal{S}/2 \wedge \mathcal{B}_{2i+1} \ne NULL$ (line 8), the position i must contain a function symbol as at least one of their children is not null. The function will be unary or binary depending on the left child being null or not. Furthermore, we have to take into account that the operations $+$, $-$, max and min require that both operands have the same dimension (line 11) and that the functions exp and ln can only be applied to adimensional expressions (line 16).

Figure 3 shows a fraction of the search tree that Algorithm 2 would generate for $\mathcal{S} = 2$. With this value, priority rules of at most three symbols could be generated, among others SPT and EDD.

4.3 Pruning the search space

The search space generated by Algorithm 2 may be reduced to a great extent from the observation that it contains many equivalent rules. We consider that two rules are equivalent if they induce the same order of priorities on the operations in US^* in Algorithm 1, for any problem instance and iteration of the algorithm.

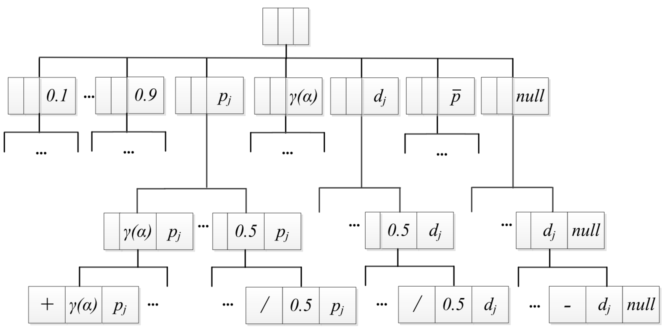


Figure 3: Part of the tree generated by Algorithm 2 for $\mathcal{S} = 2$. The priority rules may contain 1, 2 or at most three symbols, and the depth of the expression trees can be at most 2.

Some of these equivalences can be efficiently identified along the construction process due to symmetries in subexpressions rooted at commutative operators, namely $+$, $\times$, min and max. To do that, we define a total ordering on the terminal and function symbols. This ordering may be used to induce a total ordering on subtrees whose roots are at the same level in a given state. To this aim, we perform a preorder traversal of both subtrees, comparing them lexicographically w.r.t. the total ordering defined. Then, for a subtree rooted at a commutative operator, we enforce that the left subtree must be less than the right one. In particular, this strategy also enforces the operands of a commutative operator to be different. The control of symmetries may be done in line 11 of Algorithm 2, so that if the left child (in position $2i + 1$) is not less than the right child (in position $2i + 2$), no successors are generated from the commutative operators. This mechanism preserves completeness of the search space, which follows from the next result.

Proposition 1. *Let $\mathcal{R}$ be a priority rule containing one subexpression $\langle L \text{ op } R\rangle$ with op a commutative operator and $L \ge R$. There exists a rule $\mathcal{R}'$ equivalent to $\mathcal{R}$ either not using a commutative operator in the subexpression or fulfilling the condition $L < R$.*

Proof. *Case 1 ($L > R$):* Define $\mathcal{R}'$ by swapping L and R in $\mathcal{R}$. $\mathcal{R}'$ fulfils the condition and is equivalent to $\mathcal{R}$, since op is commutative.

Case 2 ($L = R$): If op $= +$, define $\mathcal{R}'$ by substituting the subexpression $\langle L \text{ op } R\rangle$ in $\mathcal{R}$ by $L/0.5$. If op $= \times$, substitute the subexpression in $\mathcal{R}$ by $pow_2(L)$. Otherwhise, if op $\in \{min, max\}$ substitute the subexpression by L. In all cases $\mathcal{R}'$ is equivalent to $\mathcal{R}$, and no commutative operator is used in the resulting subexpression. □

In addition, the number of constants in the set of terminal symbols may have a significant impact on the size of the search space, mainly if there is no restriction on how these can appear in expressions. So, another way to reduce the search space is to limit the number of constants, e.g., to a smaller subset as $\{0.25, 0.5, 0.75\}$, at the expense of losing some rules, and also to restrict the allowed operations with them. In this regard, we opted to avoid binary operations among constants. To this aim, the states generated in line 5 will not include a constant if there is already a constant in position $i + 1$ (note that in this case i and $i + 1$ are the positions of the roots of the left and right subtrees of the tree rooted at $(i - 1)/2$). Besides, in lines 16 and 18 the operators max_0 and min_0 will not be applied to constants. Some other restrictions may be imposed to avoid useless subexpressions, for example not applying the operators max_0 and min_0 to subexpressions rooted at operators max_0 and min_0; and applying the unary operator $-$ only to constants (lines 16 and 18).

4.4 Evaluation of the states and search algorithm

Conventional state space search requires an evaluation function on the states that is often defined from problem domain knowledge. For an intermediate state, the evaluation function returns an estimation on the cost to reach a solution from this state, while for a goal state it gives the actual cost of the solution represented by such node. In our setting, goal states are valid priority rules, whereas intermediate states are partial expression trees. To evaluate goal states we may follow the same strategy as in the GP proposed in [10]; i.e., the candidate priority rule is evaluated on a number of instances of the $(1, Cap(t)||\sum T_i)$ problem and the inverse of the average total tardiness is taken as its cost. However, this method cannot be extended to evaluate intermediate states, as they do not represent actual priority rules. In this respect, we have not devised any method that allows for discriminating in favor of promising states to guide the search.

As a consequence of the observations above, in order to get a complete algorithm, we have to perform an exhaustive search and evaluate each one of the goals reached. As the number of goals may be very large, even for small values of parameters $\mathcal{P}$ and $\mathcal{D}$, evaluating all of them on a number of 50 non-trivial training instances may be prohibitive. Therefore, we have to think of some surrogate or simplified method. The use of surrogates is common in scheduling, for example to estimate the fitness value of neighboring solutions within local search [22]. Additionally, we may exploit knowledge from the problem domain to discard some of the goal states; in our case, we could take into account the problem attributes and, for example, discard the rules not containing some of the most relevant ones.

From the above ideas, we propose to use a simplified evaluation consisting in a preliminary evaluation of the goal states on a few small instances. The rules performing poorly on these instances are discarded without full evaluation on the training instances. The rationale behind this is that a rule performing poorly at solving small, easy, instances could be expected to perform poorly on larger ones. So, hopefully, the number of good rules discarded in this process will be small. At the same time, if we look at the problem attributes in Table 1, p_i, d_i, $\gamma(\alpha)$ and $\bar{p}$, the first two are specific of the job i, and so it is natural to think that a rule not containing them would not be good. Notice that SPT contains only p_i, EDD contains d_i and ATC contains both. Besides, it seems reasonable to think that $\gamma(\alpha)$ and $\bar{p}$ have to be included in good rules as well. To clarify this, we will analyze this issue in the experimental study.

Regarding the search strategy, Depth First Search (DFS) seems the most reasonable option. We recall that we have to visit and evaluate all goal states and we have no way to guide the search towards the most promising ones. So, the efficiency of the search algorithm SSHE relies on the pruning strategies and the efficient evaluation of the goal states.

5 Experimental study

We conducted an experimental study aimed at analyzing the components of the proposed method and comparing it to the state-of-the-art Genetic Programming (GP) approach proposed in [10]. To this aim, we implemented a prototype in Java and ran a series of experiments on a Linux cluster (Intel Xeon 2.26 GHz. 128 GB RAM), as in [10].

5.1 The benchmark set

We used two sets of instances; the first one is that proposed in [9] to evaluate the performance of GP. This set includes 1,000 instances

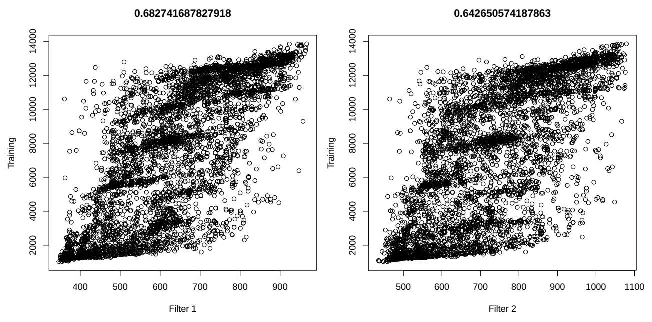


Figure 4: Relation on the average total tardiness obtained by 10,000 random rules on the training and filter sets (two filters with 5 instances each are considered, Filter 1 and Filter 2). The numbers at the top show the Kendall's tau-b coefficient.

.

of 60 jobs each and a maximum capacity of the machine of 10. As done in [10], 50 instances are used for training and the remaining 950 for testing. The second set consists of small instances and it is used for filtering, as described in Section 4.4. The instances in this set have 10 jobs and a maximum capacity of the machine of 3. We generated 2,000 instances and solved all of them using the ATC rule with $g = 0.5$, selecting the 1,000 instances with the greatest tardiness values. We recall that the use of filters is aimed at discarding low-performing rules while keeping good ones in short time. This way, the filters used in the experiments consist each of 5 of these small instances taken at random. The filtering mechanism discards any rule performing worse than ATC with $g = 0.5$ on more than one instance in the filter, avoiding its full evaluation on the training set.

5.2 Preliminary analysis

In this section, we show the results from a preliminary study aimed at analyzing the expected utility of the filtering mechanism, the relevance of the symbols used to generate rules, the influence of the dimension of the expression trees, and finally the effectiveness of the symmetry breaking mechanism at reducing the search space.

5.2.1 Filtering mechanism

To analyze the filtering strategy and to visualize the behavior of random rules, we performed a preliminary experiment: 10,000 rules were generated at random in the space defined by $\mathcal{D}$=4, $\mathcal{P}$=15 and $\mathcal{C}$=$\{0.25, 0.5, 0.75\}$, following the same procedure as that used to generate the initial population of the GP proposed in [10]. With each rule we solved the instances of the training and two filter sets (termed *Filter 1* and *Filter 2* throughout).

Figure 4 shows the average total tardiness obtained; x values correspond to filter sets and y values to the training set. In the latter, the best rule produces an average total tardiness of 1,030.94 and the average value over all rules is 7,686.11, which is similar to the average value obtained by random search, i.e., by selecting one job in line 6 of Algorithm 1 uniformly. We can observe that there is a positive correlation between the values obtained on the filter and training sets. At the same time, there are some differences between the filters. Besides, it is observed that there is no rule that being bad in the filters is good in the training set. These results suggest that the filtering mechanism may be useful, but at the same time that the results may be dependent on the filter used.

Table 3: Frequency of symbols in the rules passing the filters. For each symbol in Table 1 and filter, we show the frequency of the symbol in the passing rules w.r.t. its frecuency in the total set of rules, and the average total tardiness of the rules passing the filter in the training set of 50 instances.

	$\frac{\%filtered}{\%total}$		Avg. tardiness	
Symbol	$Filter1$	$Filter2$	$Filter1$	$Filter2$
0.25	93.27	69.59	1089.55	1097.75
0.5	49.53	85.78	1037.00	1059.77
0.75	58.30	88.01	1034.62	1111.18
p_i	126.58	126.58	1063.12	1084.76
d_i	126.58	126.58	1063.12	1084.76
$\gamma(\alpha)$	126.58	126.58	1063.12	1084.76
$\bar{p}$	89.97	101.23	1077.58	1083.11
/	53.11	79.45	1104.27	1087.84
$\times$	92.16	109.19	1038.87	1088.95
min	192.93	136.90	1051.21	1066.75
max	162.08	78.05	1077.60	1102.90
+	163.63	150.29	1046.82	1082.94
-	157.14	159.28	1063.18	1084.75
$sqrt$	27.97	40.99	1029.22	1049.02
pow_2	52.26	41.00	1030.46	1084.54
-	158.29	122.19	1044.62	1063.51
max_0	23.09	17.15	1013.42	1068.23
min_0	23.09	30.26	1038.25	1041.87
ln	75.80	111.90	1037.05	1070.28
exp	39.01	48.31	1021.52	1044.26

5.2.2 *Relevance of the symbols*

To assess the relevance of the symbols (see Table 1) we generated the whole set of priority rules in the space defined by $\mathcal{D}$=4, $\mathcal{P}$=15, and $\mathcal{C}=\{0.25, 0.5, 0.75\}$ (2.44×10^9 rules in all). Then, we analyzed the frequency of each symbol in these rules and also in the rules passing the filters. For this purpose, we considered the same two filters as in Figure 4 (Filter 1 and Filter 2).

The results are summarized in Table 3. For each symbol and filter, we show the quotient $\frac{\%filtered}{\%total}$ in percentage terms and the average total tardiness obtained by the rules passing the filter. $\%total$ is the percentage of rules that contain a given symbol and $\%filtered$ is the percentage of the rules containing the symbol in the subset of rules that passed the filter (in these experiments 6,275 and 14,362 rules passed Filter 1 and 2 respectively). This way, values of $\frac{\%filtered}{\%total}$ in percentage greater than 100 mean that the symbol is relevant as the ratio of rules containing it is greater in the filtered set than in the original set.

We can observe that all the symbols have a significant presence in the rules passing the filters, even though there are differences depending on the filter used. Symbols as min, max, $+$ and $-$ have a greater presence in the filtered sets than they have in the original set. Importantly, the frequency values of the symbols p_i, d_i and $\gamma(\alpha)$ are the same for the two considered filters, although Filter 2 is passed by more than twice the number of rules that pass Filter 1. This is not due to a coincidence; indeed, all the rules passing the filters contain all these three symbols. This is in agreement with the conjecture made in Section 4.4. This experimental evidence motivates discarding candidate rules not having the three symbols p_i, d_i and $\gamma(\alpha)$. This includes more than half of the rules in the search space.

5.2.3 *Dimension of the expression trees*

We restrict the study to the rules containing the three symbols p_i, d_i and $\gamma(\alpha)$ in the set of rules considered in Section 5.2.2. There are rules with 47 different dimensions (from $t^{-7.0}$ to $t^{8.0}$). However, only the dimensions t^{-2}, t^{-1}, t^0, t^1 and t^2 have a significant presence after filtering. Table 4 shows the frequency and average total tardiness values of these rules. It is remarkable the high proportion of rules with dimension t^1. Given their tardiness values, we could restrict the search to this space. However, this is only a preliminary result and additional experiments would be necessary for drawing more meaninful conclusions.

Table 4: Frequency of dimensions on the set of rules containing the three symbols p_i, d_i and $\gamma(\alpha)$. For each dimension t^{-2}, t^{-1}, t^0, t^1 and t^2 and filter, we show the frequency of the symbol in the passing rules w.r.t. its frequency on the total set of rules, and the average tardiness of the rules passing the filter in the training set.

	$\frac{\%filtered}{\%total}$		Avg. tardiness	
Dimension	$Filter1$	$Filter2$	$Filter1$	$Filter2$
t^{-2}	59.47	21.19	1053.74	1058.83
t^{-1}	31.17	20.89	1072.84	1062.13
t^0	26.32	43.98	1559.39	1087.08
t^1	209.06	184.67	1027.35	1075.71
t^2	73.37	132.48	1045.90	1127.07

Table 5: Summary of results from different combinations of $\mathcal{D}$, $\mathcal{P}$, $\mathcal{C}$ and time limit of 2 hours, with and without symmetry breaking in the generation of states. The generated goals are not evaluated.

			With symmetry breaking			Without symmetry breaking		
			Time	Generated		Time	Generated	
$\mathcal{D}$	$\mathcal{P}$	$\lvert\mathcal{C}\rvert$	hh:mm:ss	Nodes	Goals	hh:mm:ss	Nodes	Goals
3	7	9	00:00:00	2.06E5	5.80E3	00:00:00	4.17E5	1.83E4
4	12	0	00:00:56	1.41E8	2.41E7	00:04:40	7.33E8	1.57E8
4	12	3	00:10:04	1.53E9	1.08E8	00:40:45	6.37E9	6.44E8
4	12	9	02:00:00	1.83E10	3.72E8	02:00:00	1.82E10	4.67E8
4	15	0	00:03:06	4.16E8	1.74E8	00:30:42	4.19E9	2.23E9
4	15	3	00:31:53	4.34E9	1.15E9	02:00:00	1.65E10	6.09E9
4	15	9	02:00:00	1.57E10	2.58E9	02:00:00	1.65E10	4.00E9
5	12	0	02:00:00	1.75E10	3.52E6	02:00:00	1.82E10	4.67E8

5.2.4 *Symmetry breaking*

We performed a number of experiments considering different values of $\mathcal{D}$, $\mathcal{P}$, $\mathcal{C}$, with and without symmetry breaking, and a time limit of 2 hours. The results are summarized in Table 5. For $\mathcal{D} = 3$ the whole search space can be generated with and without symmetry breaking in less than 1s. However, symmetry breaking is able to reduce the number of goal states in almost one order of magnitude. On the other hand, for $\mathcal{D} = 5$, the search space is so large that the time limit is always reached. In both cases the number of generated states is rather similar, showing that the symmetry breaking mechanism represents a low overhead. However, symmetry breaking allows for generating around two orders of magnitude less goals due to avoiding equivalent rules. For $\mathcal{D} = 4$, the results strongly depend on $\mathcal{P}$ and $\mathcal{C}$, with three different cases:

- The algorithm terminates in both cases; but by using symmetry breaking it takes shorter time and produces much fewer goals.
- The algorithm terminates with symmetry breaking but not without it; in these cases the number of generated states and goals is lower in the first case, making clear the usefulness of symmetry breaking to avoid redundant rules.
- The algorithm terminates in no case; in these situations both algorithms generate a similar number of states and goals, but without symmetry breaking many of the goals are equivalent.

5.3 Performance of SSHE

From the results above, we choose $\mathcal{D} = 4$, $\mathcal{P} = 15$ and discard any rule not containing the 3 symbols p_i, d_i and $\gamma(\alpha)$. Besides, we considered three values for the set $\mathcal{C}$: $\emptyset$, $\{0.25, 0.5, 0.75\}$ and $\{0.1, \ldots, 0.9\}$; 4 options for dimension: t^{-1}, t^0, t^1 and any dimension; and two filters: Filter 1 and Filter 2. The time limit was 10 hours (the same time given to GP in [10]). The results are summarized in Table 6; from them, we can make the following observations:

Table 6: Results by SSHE with $\mathcal{D}$=4 and $\mathcal{P}$=15, considering different sets of constants, dimensions and filters. The time limit was 10 hours.

				Training		Test	
$\lvert\mathcal{C}\rvert$	Dim.	Fi.	hh:mm:ss	Best	Avg.	Best	Avg.
0	t^{-1}	1	00:15:56	1024.36	1072.74	1035.13	1064.50
0	t^{-1}	2	00:15:59	1018.80	1066.79	1016.12	1079.41
3	t^{-1}	1	02:05:13	1011.34	1072.84	1007.69	1061.99
3	t^{-1}	2	02:06:00	999.86	1062.13	1009.78	1070.42
9	t^{-1}	1	10:00:00	1006.96	1056.37	1005.50	1045.65
9	t^{-1}	2	10:00:00	997.64	1007.15	1012.28	1017.07
0	t^0	1	00:30:14	1002.88	1177.48	1014.42	1166.93
0	t^0	2	00:30:51	1000.30	1087.91	1010.29	1100.87
3	t^0	1	03:42:41	990.76	1559.39	997.02	1528.67
3	t^0	2	03:39:51	998.88	1087.08	1011.94	1099.65
9	t^0	1	10:00:00	999.84	4369.41	1015.95	4178.28
9	t^0	2	10:00:00	996.94	1611.72	1005.03	1583.04
0	t^1	1	00:53:22	1000.96	1039.75	1008.19	1031.78
0	t^1	2	00:54:36	998.82	1073.44	1008.78	1088.29
3	t^1	1	05:46:19	998.48	1027.35	1010.51	1023.36
3	t^1	2	05:54:48	995.22	1075.71	1012.70	1079.67
9	t^1	1	10:00:00	996.40	1043.34	1000.33	1037.19
9	t^1	2	10:00:00	998.34	1083.66	1006.96	1077.40
0	any	1	02:13:50	1000.96	1053.70	1008.19	1046.00
0	any	2	02:15:17	998.82	1078.98	1008.78	1091.82
3	any	1	10:00:00	990.76	1074.40	997.02	1067.74
3	any	2	10:00:00	995.22	1086.52	1012.70	1090.51
9	any	1	10:00:00	996.40	1095.18	1000.33	1086.51
9	any	2	10:00:00	996.10	1115.10	1008.41	1106.29

- In order for the algorithm to terminate by 10 hours, we have to restrict to $|\mathcal{C}| \leq 3$ and to rules of the same dimension.
- Although the number of rules that undergo full evaluation depends on the filter used, as shown in Section 5.2.2, the time taken by the algorithm is very similar, suggesting that most of the time is spent in applying the filters. Notice that with $\mathcal{D} = 4$, $\mathcal{P} = 15$ and $|\mathcal{C}| \leq 3$, the time spent in generating priority rules represents a small fraction of the total running time (see Table 5).
- Regarding total tardiness values, it is remarkable the bad average results from some rules of dimension t^0. To draw sharper conclussions, we summarize in Table 7 the total tardiness values obtained by the best rules averaged for each value of the parameters. We can observe that $\mathcal{C} = \{0.25, 0.5, 0.75\}$ is the best option, t^{-1} is the worst dimension and the chosen filter is not too relevant. Even though the results in Table 7 suggest that *any* dimension is the best option, it may also be too time consuming, as indicated in Table 6. On the other hand, rules of dimension t^1 yield solutions of similar quality in much shorter time, and the proportion of rules having dimension t^1 that pass any of the filters in Table 4 is greater than with any other dimension. So, dimension t^1 is to be preferred.

5.4 Comparison to GP

As pointed out, the GP proposed in [10] is the only method available to evolve priority rules for the $(1, Cap(t)||\sum T_i)$ problem. This algorithm was evaluated on the same benchmark we consider herein. Different experiments were carried out with the maximum depth of the expression trees $\mathcal{D}$ varying from 3 to 8 and maximum size $\mathcal{P} = 2^{\mathcal{D}} - 1$. Due to its stochastic nature, 30 independent runs were performed. Table 2 summarizes the total tardiness values produced by the best and average of the 30 rules, averaged for the 50 instances of the training set and the 950 of the test set. Given the limitations of SSHE mentioned in previous sections, we compare GP and SSHE with $\mathcal{D} = 4$. For smaller values the search space is small and SSHE is able to reach an optimal solution in very short time, while for values of $\mathcal{D}$ greater than 4, GP is expected to perform better due to the combinatorial explosion of states in SSHE.

For a fair comparison, we gave SSHE a time limit of 10 hours (as given to GP for one single run), and performed 30 runs using 30 different filters (generated as Filter 1 and Filter 2). In doing so, we may compare the best and average of the 30 rules obtained with those evolved by GP. Besides, in these experiments, SSHE restricts the search to expressions containing the three symbols p_i, d_i and $\gamma(\alpha)$, and with dimension t^1. The results are reported in Table 8. Noticeably, regarding total tardiness values, the rules calculated by SSHE outperform the rules evolved by GP by a wide margin. Indeed, despite restricting the search to small rules, SSHE is able to find better rules in average than GP with larger values of $\mathcal{D}$ and $\mathcal{P}$ (see Table 2). On the other hand, the rules evolved by GP are smaller in average than those produced by SSHE. This does not come as a surprise, since GP introduces a bias in the evaluation function in favor of small expression trees. Anyway, SSHE could be restricted to a lower value of $\mathcal{P}$ to search for smaller rules if necessary.

Table 7: Summary of results from Table 6 averaged for each $\mathcal{C}$, dimension and filter.

	Avg. tardiness			Avg. tardiness			Avg. tardiness	
$\lvert\mathcal{C}\rvert$	Training	Test	Dim.	Training	Test	Fi.	Training	Test
0	1005.7	1013.7	t^{-1}	1009.8	1014.4	1	1001.7	1008.4
3	997.6	1007.4	t^0	998.3	1009.1	2	999.6	1010.3
9	998.6	1006.9	t^1	998.0	1007.9			
			any	996.4	1005.9			

Table 8: Summary of results obtained by GP proposed in [10] and SSHE with $\mathcal{D}$=4, $\mathcal{P}$=15, $|\mathcal{C}|$=3 and 30 different filters. Best, average and standard deviation results from 30 runs are reported. Avg. Size is the average size of the best rules obtained in 30 independent runs. The Best value in testing is that obtained by the best rule in training.

	Training			Testing			Avg.
	Best	Avg.	SD	Best	Avg.	SD	Size
GP	997.18	1012.29	23.73	1000.13	1015.42	23.96	10
SSHE	993.90	995.62	1.90	997.05	1001.17	4.71	13

Another advantage that SSHE shows over GP is that it is more stable, as can be observed from the standard deviation values. In the experiments, the average time taken by SSHE was 7 hours and 47 minutes, reaching the time limit of 10 hours in 7 out of the 30 runs.

6 Conclusions

The automated discovery of priority rules is an important and challenging task. In this paper, we propose a new method for generating rules for the $(1, Cap(t)||\sum T_i)$ problem. Our approach is based on systematically searching over the space of possible expression trees of a given maximum depth and size, and integrates a number of effective optimizations to reduce the search space and improve efficiency. From the experimental study, we can conclude that state space search represents a useful alternative to hyper-heuristics as Genetic Programming on search spaces of reasonable size. The experimental results also encourage further research, as devising new optimizations or applying the proposed approach to other scheduling problems. In addition, the use of constraint reasoning frameworks, such as Boolean satisfiability (SAT) or Constraint Programming (CP), represents a promising line for the future, which could allow for exploiting additional pruning techniques. Finally, we will investigate the development of parameterized priority rules using abstract constants, such as the ATC rule.

ACKNOWLEDGEMENTS

This research has been supported by the Spanish Government under research project TIN2016-79190-R and by the Principality of Asturias under grant IDI/2018/000176.

REFERENCES

[1] Egon. Balas, Neil. Simonetti, and Alkis. Vazacopoulos, 'Job shop scheduling with setup times, deadlines and precedence constraints', *Journal of Scheduling*, **11**, 253–262, (2008).

[2] Juergen Branke, Torsten Hildebrandt, and Bernd Scholz-Reiter, 'Hyper-heuristic evolution of dispatching rules: A comparison of rule representations', *Evolutionary Computation*, **23**(2), 249–277, (2015).

[3] Edmund K. Burke, Matthew R. Hyde, Graham Kendall, Gabriela Ochoa, Ender Özcan, and John R. Woodward, *A Classification of Hyper-Heuristic Approaches: Revisited*, 453–477, Springer International Publishing, Cham, 2019.

[4] J. Carlier, 'The one-machine sequencing problem', *European Journal of Operational Research*, **11**, 42–47, (1982).

[5] Shelvin Chand, Quang Huynh, Hemant Singh, Tapabrata Ray, and Markus Wagner, 'On the use of genetic programming to evolve priority rules for resource constrained project scheduling problems', *Information Sciences*, **432**, 146 – 163, (2018).

[6] C. Dimopoulos and A.M.S. Zalzala, 'Investigating the use of genetic programming for a classic one-machine scheduling problem', *Advances in Engineering Software*, **32**(6), 489 – 498, (2001).

[7] Mateja Dumić, Dominik Šišejković, Rebeka Čorić, and Domagoj Jakobović, 'Evolving priority rules for resource constrained project scheduling problem with genetic programming', *Future Generation Computer Systems*, **86**, 211 – 221, (2018).

[8] Marko Durasevic, Domagoj Jakobovi, and Karlo Kneževi, 'Adaptive scheduling on unrelated machines with genetic programming', *Applied Soft Computing*, **48**, 419 – 430, (2016).

[9] Francisco J. Gil-Gala, Carlos Mencía, María Sierra, and Ramiro Varela, 'Genetic programming to evolve priority rules for on-line scheduling on single machine with variable capacity', *XVIII Conferencia de la Asociación Española para la Inteligencia Artificial, MAEB*, (2018).

[10] Francisco J. Gil-Gala, Carlos Mencía, María Sierra, and Ramiro Varela, 'Evolving priority rules for on-line scheduling of jobs on a single machine with variable capacity over time', *Applied Soft Computing*, **85**, (2019).

[11] R.L. Graham, E.L. Lawler, J.K. Lenstra, and A.H.G.Rinnooy Kan, 'Optimization and approximation in deterministic sequencing and scheduling: a survey', *Annals of Discrete Mathematics*, **5**, 287 – 326, (1979).

[12] Emma Hart and Kevin Sim, 'A hyper-heuristic ensemble method for static job-shop scheduling', *Evolutionary Computation*, **24**(4), 609–635, (2016).

[13] Alejandro Hernández-Arauzo, Jorge Puente, Ramiro Varela, and Javier Sedano, 'Electric vehicle charging under power and balance constraints as dynamic scheduling', *Computers & Industrial Engineering*, **85**, 306 – 315, (2015).

[14] Helga Ingimundardottir and Thomas Philip Runarsson, 'Discovering dispatching rules from data using imitation learning: A case study for the job-shop problem', *Journal of Scheduling*, **21**(4), 413–428, (2018).

[15] Sezgin Kaplan and Ghaith Rabadi, 'Exact and heuristic algorithms for the aerial refueling parallel machine scheduling problem with due date-to-deadline window and ready times', *Computers & Industrial Engineering*, **62**(1), 276–285, (2012).

[16] Maarten Keijzer and Vladan Babovic, 'Dimensionally aware genetic programming', in *Proceedings of the 1st Annual Conference on Genetic and Evolutionary Computation - Volume 2*, GECCO'99, pp. 1069–1076, (1999).

[17] C. Koulamas, 'The total tardiness problem: Review and extensions', *Operations Research*, **42**, 1025–1041, (1994).

[18] John R. Koza, *Genetic Programming: On the Programming of Computers by Means of Natural Selection*, MIT Press, 1992.

[19] Carlos Mencía, María Sierra, Raúl Mencía, and Ramiro Varela, 'Evolutionary one-machine scheduling in the context of electric vehicles charging', *Integrated Computer-Aided Engineering*, **26**(1), 49–63, (2019).

[20] Su Nguyen, Yi Mei, Bing Xue, and Mengjie Zhang, 'A hybrid genetic programming algorithm for automated design of dispatching rules', *Evolutionary Computation*, **27**(3), 467–496, (2019).

[21] S-O. Sang-Oh Shim and Y-D Kim, 'Scheduling on parallel identical machines to minimize total tardiness', *European Journal of Operational Research*, **177**(1), 135–146, (2007).

[22] Camino R. Vela, Ramiro Varela, and Miguel A. González, 'Local search and genetic algorithm for the job shop scheduling problem with sequence dependent setup times', *Journal of Heuristics*, **16**(2), 139–165, (2010).