Genetic Programming with Local Search to evolve priority rules for scheduling jobs on a machine with time-varying capacity

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

[a]*Department of Computer Science, University of Oviedo, Gijón 33204, Spain*

Abstract

Priority rules combined with schedule generation schemes are a usual approach to online scheduling. These rules are commonly designed by experts on the problem domain. However, some automatic method may be better as it could capture some characteristics of the problem that are not evident to the human eye. Furthermore, automatic methods could devise priority rules adapted to particular sets of instances of the problem at hand. In this paper we propose a Memetic Algorithm, which combines a Genetic Program and a Local Search algorithm, to evolve priority rules for the problem of scheduling a set of jobs on a machine with time-varying capacity. We propose a number of neighbourhood structures that are specifically designed to this problem. These structures were analyzed theoretically and also experimentally on the version of the problem with tardiness minimization, which provided interesting insights on this problem. The results of the experimental study show that a proper selection and combination of neighbourhood structures allows the Memetic Algorithm to outperform previous approaches to the same problem.

Keywords: One Machine Scheduling, Priority Rules, Local Search, Genetic Programming, Memetic Algorithm

1. Introduction

Priority rules, also called *dispatching rules*, have been used for decades to solve combinatorial problems, in particular to solve complex scheduling problems. In this field, these rules are usually exploited in combination with *schedule builders*, or *schedule generation schemes* [1], which provide a way to enumerate and build schedules in a given space. A schedule builder is a constructive algorithm that in each step selects non-deterministically the next job or operation to be scheduled. This selection may be performed by some priority rule, which

*Corresponding author.
Email address: ramiro@uniovi.es (Ramiro Varela)
URL: http://www.di.uniovi.es/iscop (Ramiro Varela)

establishes an ordering on the available options, preferably exploiting knowledge from the problem domain. One of the main features of these kind of solvers is that they are able to produce solutions very quickly, in comparison with other exact or even approximate approaches, such as state space search [2, 3, 4] or metaheuristics [5, 6], at the cost of producing solutions of lower quality. In spite of that, schedule builders may reach solutions of reasonable quality if the guiding rule is tailored to the specific characteristics of the problem; in this case they are suitable when a schedule must be obtained by a time limit, which is known as *online scheduling*.

Online scheduling problems arise in many real scenarios. One example is the Electric Vehicle Charging Scheduling Problem (EVCSP) presented in [7]. In this problem, the charging times of a number of Electric Vehicles (EV) must be scheduled in a charging station organized into three lines, each one connected to one of the phases of a three-phase feeder [8]. Due to technological restrictions, the number of EVs charging at the same time in each of the three lines must be similar and so, in general, the power that may be consumed in a line varies over time. The EVCSP is dynamic and the solving procedure proposed in [7] decomposes this problem into a large number of instances of the problem of scheduling a set of jobs on a single machine with time-varying capacity, denoted $(1, Cap(t)||\sum T_i)$, which must be solved over the scheduling horizon. For the proposed system to be effective, tens or even hundreds of instances of this problem must be solved in just a few seconds.

As showed in [9, 10], evolutionary algorithms may reach good solutions to the $(1, Cap(t)||\sum T_i)$ problem, although taking a prohibitive time for the online requirements of the EVCSP. So, in [7], a schedule builder guided by the well-known *Apparent Tardiness Cost* (ATC) rule was exploited.

Priority rules can be defined manually by experts on the problem domain, as it is the case of the ATC rule [11], although it is clear that automatic methods could capture some characteristics of the scheduling problem that are not clear to human experts, and in this way the priority rules may be adapted to instances with some particular characteristics. In this regard, it is notable the recent interest of researchers for these methods, most of whom are adopting some hyperheuristic strategy, i.e., a method that searches across some space of heuristics [12] to solve a problem, instead of searching on the space of solutions. Given the structure of priority rules, Genetic Programming (GP) [13] arises as the most common and natural choice.

Some examples of scheduling problems for which different authors developed GP approaches to evolve priority rules are the job shop scheduling problem [14, 15, 16, 17], some versions of the one machine sequencing problem [18, 19, 20], unrelated parallel machines scheduling problems [21], bin packing [22] or resource constrained project scheduling [23, 24, 25], among others. Burke et al. [12] classify these approaches as *heuristic generation*, specifically these algorithms are named *genetic programming based hyper-heuristics*.

GP was also applied to dynamic scheduling problems. Branke et al. [26] provide an exhaustive study on the dynamic flexible job shop scheduling problem, in which they study some representations and algorithms to evolve priority

rules. They conclude that evolving expression trees with genetic programming is the best choice. This work was later extended in [27], in which they provide a nice description of the state of the art, as well as a taxonomy of hyper-heuristics and some guidelines for designing them. The same problem was recently considered by Zhang et al. [28], who explored the idea of evolutionary multitask learning. This paradigm was later extended in [29] by including a surrogate model to reduce the high time consumed in the training process. Durasević and Jakobović [30] also provide a survey of dispatching rules, specifically designed for the dynamic unrelated machines environment. In this work, they collected a large set of rules that were tested under a number of scheduling criteria to analyze which rule is the most suitable depending on the situation.

The combination of an evolutionary algorithm with a local search is the most typical form of a Memetic Algorithm (MA) as defined in [31, 32, 33]. MAs are among the most outstanding hybrid metaheuristics and have a long track record of success in solving famous hard combinatorial problems as, for example, the Traveling Salesman Problem [34, 35], Job Shop Scheduling [36, 37], the classic One Machine Sequencing Problem [38] or the Tool Switching Problem [39], to name just a few.

In contrast to other evolutionary algorithms, one of the limitations of the existing GP approaches is that they usually do not exploit Local Search (LS) to intensify the search in the neighbourhood of the evolved solutions. This fact is noticed in [40], where the authors also remark the low capacity of the GP operators to intensify the search in the most promising areas of the search space. As far as we know, there are just a few attempts to exploit LS in combination with GP; one of them is proposed in [41], where the authors exploit two LS operators in a GP that evolves decision trees for classification. Regarding generation of dispatching rules, in [42] the authors proposed an iterated local search algorithm, which was applied to the dynamic Job Shop Scheduling problem. They exploited a single mutation of the expression trees as neighbourhood structure. This structure was later exploited in [43] in combination with GP. Some of the inconveniences of this structure are its high cardinality and that it generates many equivalent expressions.

In this paper, we propose combining an extension of the GP proposed in [20] with LS to solve the $(1, Cap(t)||\sum T_i)$ problem; the algorithm is termed MGP (Memetic Genetic Program) herein. The LS incorporates some neighbourhood structures designed specifically for this problem. The main contribution of the paper is the definition and analysis of these structures, both theoretical and experimental, as well as their incorporation into MGP. The results of the conducted experimental study show that the priority rules evolved by MGP outperform the rules obtained by previous methods.

The remainder of the paper is organized as follows. The next section introduces the $(1, Cap(t)||\sum T_i)$ problem and how this problem may be solved online by a schedule builder guided by a priority rule. Section 3 summarizes the methods proposed so far to build new priority rules for this problem. Section 4 describes the structure of the LS algorithm. The definition and analysis of the proposed neighbourhood structures are given in Section 5. MGP is described

in Section 6. Section 7 reports the results of the conducted experimental study. Finally, Section 8 summarizes the main conclusions of the paper and proposes some new lines for further study.

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

In this section, we introduce the formal definition of the problem of sequencing jobs on a machine with variable capacity over time and how it may be solved online by means of schedule builders guided by priority rules.

2.1. Definition of the problem

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 on the machine, $X(t)$, 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:

$$\sum_{j=1,\ldots,n} \max(0, C_j - d_j) \tag{3}$$

which should be minimized.

As pointed out in [9], one particular case of this problem is when the capacity of the machine is constant over time. This is the parallel identical machines problem [11], denoted $(P||\sum T_i)$, which is NP-hard. Thus, it follows that the $(1, Cap(t)||\sum T_i)$ problem is NP-hard as well.

2.2. Solving the problem online

To solve the $(1, Cap(t)||\sum T_i)$ problem online, we consider a method with two main components: the schedule builder and the priority rules. The schedule builder is depicted in Algorithm 1; US denotes the current unscheduled jobs and $X(t)$ is the consumed capacity of the machine by the jobs scheduled so far. US is initialized with all the jobs (line 1). In each iteration, the algorithm builds the subset US^* containing the jobs in US that can be scheduled at the earliest possible starting time (line 4), denoted $\gamma(\alpha)$, and then selects one of these

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: $US \leftarrow \{1, 2, ..., n\}$; */* Initializes unscheduled jobs */*
2: $X(t) \leftarrow 0, t \geq 0$; */* and consumed capacity*/*
3: **while** $US \neq \emptyset$ **do**
4: $US^* \leftarrow \{u \in US | X(t) < Cap(t), \gamma(\alpha) \leq t < \gamma(\alpha) + p_u\}$; */* Candidate jobs to be scheduled next */*
5: Non-deterministically pick job $u \in US^*$; */* Job selection */*
6: $st_u \leftarrow \gamma(\alpha)$; */* Schedule the selected job */*
7: $X(t) \leftarrow X(t) + 1, st_u \leq t < st_u + p_u$;*/* Update consumed capacity */*
8: $US \leftarrow US - \{u\}$; */* Update unscheduled jobs */*
9: **return** The schedule $S = (st_1, st_2, ..., st_n)$;

jobs (lines 5) non-deterministically. The selected job, u, is scheduled at time $\gamma(\alpha)$ (line 6). After that, the consumed capacity and the unscheduled jobs are updated (lines 7 and 8). The algorithm finishes when all the jobs are scheduled (line 3) and returns the built schedule (line 9).

The schedule builder may be used in combination with some priority rule to make the non-deterministic choice in each iteration: the job having the highest priority in US^* is chosen to be scheduled. This paradigm is called *priority scheduling*, which is particularly appropriate for *online scheduling*, where decisions must be made quickly. In the literature there are a number of rules that could be adapted to the $(1, Cap(t)||\sum T_i)$ problem. Among them, we may consider the *Earliest Due Date* (EDD) or *Shortest Processing Time* (SPT) rules. These two rules are often used for objective functions that are non decreasing with the completion time of the jobs, as for example makespan, lateness or even tardiness. As they are quite simple rules, they often produce rather moderate results. In contrast, more sophisticated rules are usually able to produce (much) better results as they take into account more knowledge of the problem. This is the case of the *Apparent Tardiness Cost* (ATC) rule, which was used with success to solve some scheduling problems with tardiness objectives (e.g. [44, 45]); with this rule, the priority of each job $j \in US^*$ is given by

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

In Equation (4), $\gamma(\alpha)$ denotes the earliest starting time for a job in US, $\bar{p}$ is the average processing time of the jobs in US and g is a look-ahead parameter to be introduced by the user. As we can see, the ATC rule combines the information exploited by SPT and EDD as the priority of a job j is in inverse ratio with its duration p_j and it decreases with the slack time to its due date $d_j - \gamma(\alpha) - p_j$.

3. Structure, representation and construction of priority rules

In this section we introduce the set of symbols and the grammar established to build up new priority rules. These two elements, together with the maximum

size and depth allowed to the rules, establish the *search space* in which the algorithms may search for new rules. Besides, we describe an efficient generation procedure, which strongly relies on a specific array representation of rules. This procedure was exploited in [46] in the context of heuristic state space search.

3.1. Set of symbols and generative grammar

Table 1 shows the set of terminal symbols and operators considered herein. In addition to a set of dimensionless constants, the terminal symbols represent single attributes of a problem instance as job durations, due dates and earliest starting time of a job, whose dimension is time, denoted T, in all cases. Besides, we consider a reduced number of arithmetic operators which are commonly used in hand made priority rules as ATC, for example.

In this work, we propose to restrict the search to the set of rules represented by *dimensionally compliant* expressions. In these expressions, operations as +, -, *max* and *min* can only be applied to operands with the same dimension, being the dimension of their result the same as that of the operands. The operations * and / can be applied to any pair of operands with independence of their dimensions, being the dimension of the result of the product or quotient, respectively for * and /, of the operands' dimensions. Analogously, operations pow_2 and *sqrt* can be applied to any operand. Besides, we consider that operations as *exp* or *ln* can only be applied to dimensionless expressions. Some examples of dimensionally compliant expressions are the classic human-designed SPT, EDD, and ATC rules, all having dimension T^{-1}. One evident feature of dimensionally compliant rules is that they are more rational and understandable by humans than other rules not having this property. Furthermore, the results reported in [20] from the proposed Genetic Program show that restricting the search to the space of dimensionally compliant rules makes the Genetic Program to evolve rules of similar quality and lower size than searching across the whole space of feasible arithmetic expressions.

3.2. Rule representation

We consider the array representation for priority rules proposed in [20], whose interpretation, borrowed from binary heaps implementation, is as follows. Let $\mathcal{B}$ denote the array representing a priority rule; for convenience the indices of their components are denoted $0, \ldots, \mathcal{S}$. So, $\mathcal{S} + 1$ is the size of the array, which fulfils $\mathcal{S} + 1 = 2^{\mathcal{D}} - 1$, $\mathcal{D}$ being the maximum depth of a tree represented by the array. $\mathcal{B}_0$ is the root node, and the remaining positions may

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

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

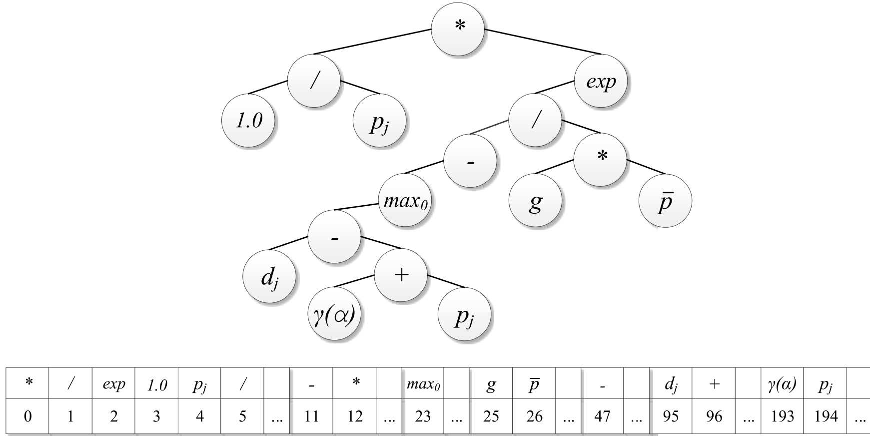


Figure 1: Expression tree and array representation of the tree for the ATC rule.

either be $NULL$ or contain a terminal or function symbol. If $\mathcal{B}_i$ is not $NULL$ its parent is $\mathcal{B}_{(i-1)/2}$ and its children (if they exist) 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.

Figure 1 shows the representation of the ATC rule. As it has depth 8 it requires an array with at least 255 positions, even though it only has a size of 17 nodes.

3.3. Building new rules from scratch

The use of arrays to represent priority rules facilitates the process of enumerating and building feasible rules having maximum depth $\mathcal{D}$. To build dimensionally compliant expressions, the expression trees may be generated by filling the array from right to left starting in position $\mathcal{S}$. In this way, one operator is always inserted after its operands, which facilitates building dimensionally compliant expressions.

Algorithm 2 shows the generation procedure for the grammar briefly described in Section 3.1 and the function and terminal symbols given in Table 1. In the algorithm, $\mathcal{C}$ denotes the set of constants. $[B_k]$ denotes the dimension of the expression tree under position k; in particular $[B_k] = 1$ if the expression is dimensionless. The algorithm iterates from the last position of the array ($i = \mathcal{S}$) to the first one ($i = 0$), and in each iteration it chooses non-deterministically the symbol to be inserted in position i. The notation $\mathcal{U}(X)$ means that one element of the set X is chosen uniformly, and abusing the language $\mathcal{P}[a, b]$ means that a is chosen with probability $\mathcal{P}$ and b with probability $1 - \mathcal{P}$ respectively. The value of $\mathcal{P}$ determines the structure of the tree. If $\mathcal{P}$=1, then no symbol $NULL$ is chosen in any iteration (see line 11) and so we will have a *full* tree. For $\mathcal{P} < 1$, the expression tree is expected to be non full. The symbol $<_{lex}$ expresses a lexicographical ordering between subtrees. It is defined from a total ordering on the symbols in the alphabet (see Table 1), so that two subtrees B_i

Algorithm 2 Grammar Derivation Algorithm

Data: Maximum depth $\mathcal{D}$ for the tree and a probability $\mathcal{P}$.
Result: A feasible expression tree $\mathcal{B}$.

```
1: S ← 2^D − 1;
2: for all i = 0, ..., S do
3:     B_i = NULL;
4: i ← S;
5: while B_0 = NULL do
6:     while i ≥ 0 do
7:         if i ≥ S/2 ∨ B_{2i+1} = NULL then
8:             if i%2 ≠ 0 ∧ B_{i+1} ≠ NULL then
9:                 B_i ← U({p_j, d_j, γ(α), p̄, C});
10:            else
11:                B_i ← P[U({p_j, d_j, γ(α), p̄, C}), NULL];
12:        else
13:            if B_{2i+2} ≠ NULL then
14:                if [B_{2i+1}] = [B_{2i+2}] then
15:                    if B_{2i+1} <_lex B_{2i+2} then
16:                        B_i ← U({+, −, max, min, ×, /});
17:                    else
18:                        B_i ← U({−, /});
19:                else
20:                    if B_{2i+1} <_lex B_{2i+2} then
21:                        B(i) ← U({×, /});
22:                    else
23:                        B(i) ← U({/});
24:            else
25:                if [B_{2i+1}] = 1 then
26:                    B(i) ← U({−, pow_2, sqrt, max_0, min_0, exp, ln});
27:                else
28:                    B(i) ← U({−, pow_2, sqrt, max_0, min_0});
29:        i ← i − 1;
30: return The rule B;
```

and B_j, rooted at the same level within an expression tree, fulfil $B_i \leq_{lex} B_j$ if the chain of symbols obtained from B_i following a pre-order traversal is lower than the chain obtained from B_j following the same procedure. The utility of this operator is to avoid the generation of some equivalent subexpressions rooted at commutative operators as $+$, $\times$, max and min.

Going into more detail

- The condition $i \geq \mathcal{S}/2 \vee \mathcal{B}_{2i+1} = NULL$ (line 7) 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 $i\%2 \neq 0 \wedge \mathcal{B}_{i+1} \neq NULL$ (line 8), 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 (line 10) i could contain $NULL$ as well.
- However, if $i < \mathcal{S}/2 \wedge \mathcal{B}_{2i+1} \neq NULL$ (line 12), the position i must contain a function symbol as at least one of their children is not null. The

function will be unitary or binary depending on the left child being null (line 13) or not (line 24) respectively.

- If it is not null (line 13), we have to consider binary operands (lines 14-23). All operands may be applied if both subtrees have the same dimension (line 14), but the commutative operands, i.e., $+$, max, min and $*$ are considered only if the left subtree is lower than the right one (line 15), to avoid duplicated solutions. If the subtrees have different dimension (line 19) only the operators $*$ and $/$ can be used, the first one is considered only if the left subtree is lower than the right one to avoid duplications as before.
- If it is null (line 24), any of the unitary operators may be put in B_i, unless the subtree represents an expression having dimension other than 1, i.e., is not dimensionless, in which case exp and ln cannot be considered.

4. The Local Search Algorithm

A local search algorithm starts from a candidate solution and then iteratively moves to a neighbouring solution selected by some criteria. It commonly finishes either after a number of iterations or when no improvement is possible from the current solution. In doing so, the local search algorithm turns a given solution into a nearby local optimum. In order to devise an effective local search algorithm, the key point is the *neighbourhood structure* (also called neighbourhood rule). Besides, two more decisions must be taken: the acceptance criterion to chose one of the neighbouring solutions to move to, and the termination condition.

Algorithm 3 Local Search Algorithm

Data: Initial rule $\mathcal{B}$. Neighbourhood structure $\mathcal{N}$. Acceptance Criterion: Hill Climbing (HC) or Gradient Descent (GD).
Result: An (hopefully) improved rule $\mathcal{B}^*$.

```
1: B* ← B; /* Starts the best so far solution */
2: termination_condition ← false;
3: while not termination_condition do
4:    B' ← B*; /* Starts the best neighbour of B* */
5:    for all B'' ∈ N(B*) do /* Explore the neighbors of B* */
6:       if B'' is better than B' then
7:          B' ← B'';/* Updates the best neighbour of B* */
8:          if Acceptation Criterion is HC then
9:             break; /* Some neighbour improved the current best */
10:   if B' is better than B* then
11:      B* ← B';/* Updates the best so far solution */
12:   else
13:      termination_condition ← true;
14: return The rule B*;
```

Algorithm 3 shows the main structure of the Local Search Algorithm (LSA) we consider herein. The algorithm starts from a rule $\mathcal{B}$, a neighbourhood structure $\mathcal{N}$, and the acceptance criterion, which may be hill climbing or gradient descent. In each iteration, the algorithm obtains the neighbouring solutions $\mathcal{N}(B^*)$ (line 5) of the current solution B^* (initially $B^* = B$ in line 4) and selects B' (line 6-9) to be the first improving neighbour (if hill climbing) or the best of all neighbours (gradient descent). Only if B' is better than B^* (line 10) then the local search continues from B' (line 11), otherwise the termination condition is met, i.e., none of the neighbours is better than the current solution (line 13), and the local search algorithm finishes and returns the best solution reached B^* (line 14).

5. Neighbourhood structures

As mentioned, the neighbourhood structure is the key element of a local search algorithm. In general, this structure must produce a number of neighbouring solutions from a given solution just doing some small variations on the given solution. In this way, it is expected that the new solutions are not too different from the current one. Furthermore, the variation operation should be designed so that the new solutions have some chance to be better than the current one, which is not generally easy. In order to keep the algorithm within a reasonable time consumption, the number of neighbours must be limited and, if the evaluation of them is too expensive, some surrogate method is sometimes used for approximate evaluation, at the risk of discarding a number of improving solutions.

Another appealing feature of a neighbourhood is the *connectivity property*. This property holds if every solution is reachable from any other in the search space by repeatedly applying the neighbourhood operator. This property is relevant as it guarantees that the algorithms may eventually reach an optimal solution. However, it does not always have practical importance and there are many high performing neighbourhoods not holding this property. If a neighbourhood does not have this property, it can still reach many, if not all, the solutions, if it is applied repeatedly starting from many different initial solutions. In any case, it may be interesting to analyze the subspace that a given neighbourhood can reach from a given solution, namely its *connectivity*, as this information may help to predict its performance.

All of the above issues were successfully tackled, for example, in the design of local searches for scheduling problems as the classic job shop and many of its variants [37, 36, 47, 48]. In these cases, the neighbourhood structures consist in doing small changes on the schedules, so the new schedules are in general similar to the original ones.

However, dealing with expression trees is rather different as doing small changes in an expression tree may give rise to quite different schedules produced by the new priority rules. In spite of that, we can find in the literature some successful studies on neighbourhood structures for priority rules, some of them in the field of scheduling. In [49], the authors review a number of local search

Table 2: Subsets of symbols that may be exchanged by the neighbourhood structure $\mathcal{N}^1$. Each symbol may be exchanged by any other in the same row.

p_j	d_j	$\gamma(\alpha)$	$\bar{p}$
0.1	0.2	...	0.9
exp	ln		
max_0	min_0	$-$	
max	min	$+$	$-$

methods that are used in combination with Genetic Programming, and classify these methods into two categories: *tuning of numerical coefficients* and *subtree fine-tuning*. In [43], the authors deal with the job shop scheduling problem and use methods of the second category; they exploit a restricted subtree mutation (RSM) operator that inserts a small subtree, of depth at most 2, in a random node of the expression tree. They also use another kind of neighbourhood termed attribute mutation. This method is based on a tree representation that includes a flag for each node of the tree so that if the flag in a node is set to 0 the subtree rooted at this node vanishes and it is taken as the constant 1 in the expression. In both cases, the structure of a neighbouring priority rule may be quite different from the original one. Regarding the category of coefficients tuning, in [50] the authors exploit Genetic Programming to solve symbolic regression problems. The evolved expression trees are enhanced with multiplication factors in the nodes, which are adjusted by means of a local search whose neighbourhood structure relies on numerical optimization.

In this paper, we consider the space of priority rules defined in Section 2 and propose some neighbourhood structures tailored to the particular features of these rules. We consider methods of type subtree fine-tuning and leave other methods as tuning of numerical coefficients for a further study as they will require some changes in the structure of the rules. The proposed structures rely on the array representation of expression trees and perform changes on a given subset I of their components $\{0, \ldots, S\}$. In this way, the cardinality of a neighbourhood may be controlled by the value of $|I|$. For convenience, we will use the notation $\mathcal{N}(I, E, \mathcal{B})$ to denote the set of neighbouring rules obtained by the structure $\mathcal{N}$ from the rule represented by the array $\mathcal{B}$ making feasible changes in all the positions of the array in the set I, using the elements of the set E.

Definition 1. ($\mathcal{N}^1$). A single move swaps the symbol in position $\mathcal{B}_i$, $i \in I$, by another one in the set E, which in this case is a set of single symbols. The change is only permitted if the new symbol maintains the dimension of the subtree rooted at i. The subsets of symbols that are exchangeable by $\mathcal{N}^1$ are shown in Table 2, one subset in each row. In the case of constants, only the previous and posterior values are considered in order to reduce the cardinality of $\mathcal{N}^1(I, E, \mathcal{B})$. Notice that the symbols $/$, $*$, pow_2 and $sqrt$ cannot be exchanged by any other. One of the properties of this structure is that the size and shape of the trees in $\mathcal{N}^1(I, E, \mathcal{B})$ are the same as those of $\mathcal{B}$.

Table 3: Number of expressions in the set E_3 exploited in the neighbourhood structure $\mathcal{N}^2$, eliminating (*Reduced*) or not (*Full*) equivalent expressions.

	Number of expressions	
Dimension	*Reduced*	*Full*
T^{-1}	43	44
T^{0}	78	830
$T^{0.5}$	4	4
T^{1}	124	212
T^{2}	10	20

Definition 2. ($\mathcal{N}^2$). A move in $\mathcal{N}^2$ changes the subtree rooted at $\mathcal{B}_i$, $i \in I$, by one of the candidate expressions with maximum size n given in the set E, which in this case is denoted E_n. As in $\mathcal{N}^1$, the dimension of the tree rooted at $\mathcal{B}_i$ must be the same as in the original one. Besides, the depth of the neighbouring tree must not exceed the maximum depth $\mathcal{D}$. In this case, it is clear that the size and shape of the tree may change. In order to reduce the number of neighbours, some equivalent expressions are avoided by symmetry breaking considering the commutativity of operations as $+$, $*$, min and max, which could give rise to equivalent expressions as $max(p_j, d_j)$ and $max(d_j, p_j)$. For the same reason, the value of n must be limited. Besides, we do not include in E_n binary operations between constant symbols.

In our experimental study we only used the sets E_1, E_2 and E_3 formed by all the trees of size 1, 2 and 3, respectively. E_1 only contains the terminal symbols, which have dimension T^1 or are dimensionless constants. However, E_2 and E_3 contain a larger number of expressions with dimensions varying from T^{-1} to T^2. The reduction on the number of neighbours produced by symmetry breaking from these sets may be appreciated in Table 3. Notice that $E_1 \subset E_2 \subset E_3$.

$\mathcal{N}^2$ is similar to the structure considered in [42], which is termed restricted subtree mutation (RSM). The main differences being that $\mathcal{N}^2$ only exchanges subtrees having the same dimension and that it avoids some neighbours by symmetry breaking; in turn, RSM may only insert subtrees of maximum depth 2. As a result, the cardinality of RSM is much larger and its neighbours are generally much more different from the original solution than they are for $\mathcal{N}^2$.

If we analyze the structures $\mathcal{N}^1$ and $\mathcal{N}^2$ we can see that in general $|\mathcal{N}^1(I, E, \mathcal{B})|$ is much lower than $|\mathcal{N}^2(I, E_n, \mathcal{B})|$, but they may have some elements in common, so it makes sense considering $\mathcal{N}^1 \cup \mathcal{N}^2$ as a new structure.

Figure 2 shows the rule $max(p_j, d_j) * \gamma(\alpha)$, its array representation and some neighbours from the two structures considering $I = \{1\}$. The trees in a), b) and c) are generated by $\mathcal{N}^1$ changing the symbol max by symbols $-$, $+$ and min, respectively. While the rules in d), e) and f) are generated by $\mathcal{N}^2$ from the expressions d_j, $-p_j$ and $max(p_j, d_j)$ respectively.

Neither of the structures $\mathcal{N}^1$ and $\mathcal{N}^2$ have the connectivity property. This is clear as their connectivities are reduced to different subsets of expressions having the same dimension as the original solution. This feature could be exploited

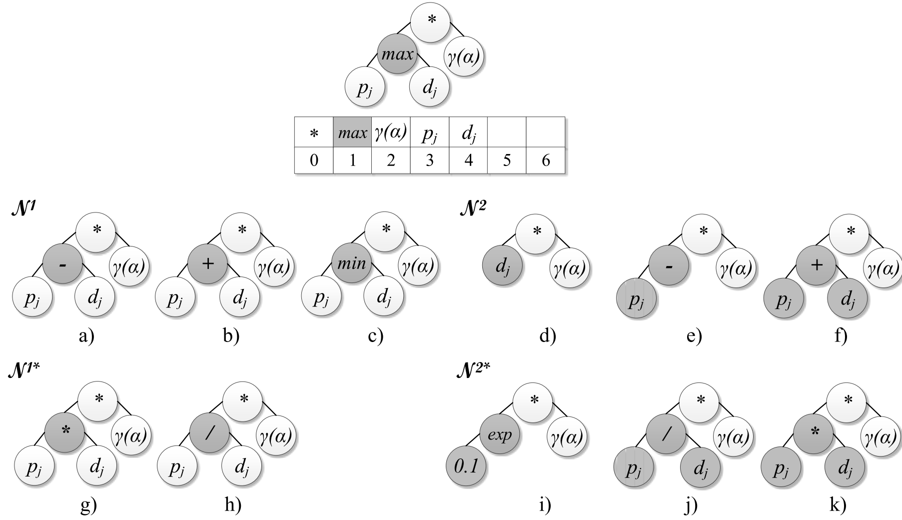


Figure 2: Some neighbors of the expression $max(p_j, d_j) * \gamma(\alpha)$, generated using $\mathcal{N}^1$, $\mathcal{N}^2$ and their extensions $\mathcal{N}^{1*}$ and $\mathcal{N}^{2*}$.

to restrict the search to a subspace of expressions having the same dimension, but at the same time it is inconvenient if our purpose is to devise an algorithm searching for solutions in the whole space. From the above, it would be interesting to devise some neighbourhood that could give rise to expression trees having different dimensionality. To do that, we have to look at the operators admitting operands with different dimension so that they may generate expressions having different dimensions as well. These are the operators *, /, - (unitary), min_0, max_0, $sqrt$ and pow_2. Bearing this in mind, we can extend, for example, $\mathcal{N}^1$ and $\mathcal{N}^2$ as follows:

Definition 3. ($\mathcal{N}^{1*}$). This neighbourhood extends $\mathcal{N}^1$ in the following way: if each of the nodes from the parent node of B_i to the root of the tree contains one operator in $\{*, /, -, min_0, max_0, sqrt, pow_2\}$, then the symbol in B_i may be exchanged by any other symbol in E producing a feasible expression in the subtree rooted at B_i. This means that a dimensionless constant may be exchanged by a dimensional attribute, and that a binary operator as + may be exchanged by *, for example; this way producing a neighbouring expression tree having different dimension than the original one.

Definition 4. ($\mathcal{N}^{2*}$). Analogously, under the same conditions of B_i as in the previous definition, this neighbourhood extends $\mathcal{N}^2$ so that any subtree in E may be inserted in a node B_i as long as the resulting subtree under B_i is feasible and the whole tree does not exceed the allowed depth $\mathcal{D}$.

Clearly, these two extended neighbourhoods allow the algorithms to search across a space containing expression trees having different dimensions. Regarding their connectivity properties, certainly $\mathcal{N}^{1*}$ does not have this property, as

the neighbouring expression tree has the same shape as the original one. $\mathcal{N}^{2*}$ has not this property either but, if the set E is exhaustive enough, $\mathcal{N}^{1*} \cup \mathcal{N}^{2*}$ does have this property, as stated in the following result.

Proposition 1 (Connectivity property of $\mathcal{N}^{1*} \cup \mathcal{N}^{2*}$)**.** *Let E be the set of all feasible subtrees of sizes 1, 2 and 3 that may be built from the given set of symbols, and let $\mathcal{T}_1$ and $\mathcal{T}_2$ be two feasible expression trees in the space limited by $\mathcal{S}$, namely having maximum size $\mathcal{S}+1$ and maximum depth $\mathcal{D} = log_2(\mathcal{S}+2)$. Then, $\mathcal{T}_2$ is reachable from $\mathcal{T}_1$ through a finite sequence of $\mathcal{N}^{1*} \cup \mathcal{N}^{2*}$ moves.*

Proof. We may give a constructive procedure as follows. From any expression tree as $\mathcal{T}_1$ we can take a trivial neighbour $\mathcal{T}$ given by a terminal symbol from E in its root. Then, from this initial tree $\mathcal{T}$, another tree $\mathcal{T}'$ having the same structure and the same symbols in the leaf nodes as $\mathcal{T}_2$ may be built through a finite number of $\mathcal{N}^{2*}$ moves as follows: in each step, one of the leaves in the tree having a different symbol than the node in the same array position i in $\mathcal{T}_2$ is exchanged by an expression $e \in E$ so that:

- If i corresponds to a leaf in $\mathcal{T}_2$, then the same terminal symbol in $\mathcal{T}_2$ is inserted in the position i in the tree.
- If i corresponds to a binary subtree in $\mathcal{T}_2$, then any expression $e \in E$ representing a binary tree rooted in one of the operators in $\{*, /\}$ is inserted in the same position in S.
- If i corresponds to a subtree with only one child (the left one) in $\mathcal{T}_2$, then any expression $e \in E$ representing a degenerated subtree of depth 2 rooted in any symbol from $\{-, min_0, max_0, sqrt, pow_2\}$ is inserted in the same position i in S.

After the above process, the built tree $\mathcal{T}'$ has the same number of nodes and shape as $\mathcal{T}_2$, and their leaves are the same. The only differences between these trees are in the operators in the inner nodes; in $\mathcal{T}'$ all of them belong to the set $\{*, /, -, min_0, max_0, sqrt, pow_2\}$. Therefore, the operator in each one of the inner nodes in $\mathcal{T}'$ may be exchanged by the operator in the same position in $\mathcal{T}_2$ by a single move of $\mathcal{N}^{1*}$, starting from the deepest nodes to the root. The resulting tree is then $\mathcal{T}_2$.

□

From all the above, in the experimental study we will consider the structures $\mathcal{N}^{1*}$, $\mathcal{N}^{2*}$ and the union of both that will be denoted $\mathcal{N}^{12*}$.

6. The Memetic Genetic Program

We propose a Memetic Genetic Program (MGP) that combines a Genetic Program (GP) similar to that proposed in [20] with the Local Search Algorithm (LSA) described in the previous section. Algorithm 4 shows its main structure. It is a generational algorithm that starts from an initial population of *#popsize*

Algorithm 4 Memetic Genetic Program

Data: A set $\mathcal{M}$ of instances of the $(1, Cap(t)||\sum T_i)$ problem. Parameters: crossover probability p_c, mutation probability p_m, number of generations $\#gen$, population size $\#popsize$, local search probability p_{LS}.

Result: An expression tree representing a priority rule for the $(1, Cap(t)||\sum T_i)$ problem.

1: Generate the initial population $\mathcal{P}(0)$ with #popsize expression trees (Algorithm 2);
2: Evaluate $\mathcal{P}(0)$ on the set $\mathcal{M}$;
3: **for all** t=1 to #gen-1 **do**
4: **Selection**: organize the expression trees in $\mathcal{P}(t-1)$ into pairs of parents at random;
5: **Recombination**: mate each pair of parent expression trees and mutate the two offsprings in accordance with p_c and p_m;
6: **Evaluation**: evaluate the resulting expression trees on the set $\mathcal{M}$;
7: **LocalSearch**: apply LSA (Algorithm 3) to the offspring in accordance with p_{LS};
8: **Replacement**: make a tournament selection of two expression trees from every two parents and their offsprings to build the population in the next generation $\mathcal{P}(t)$;
9: **return** The best expression tree reached;

individuals (expressions trees) generated by Algorithm 2 (lines 1-2). Then, it iterates over a number of generations ($\#gen$). Each iteration (line 3) starts with a selection procedure (line 4) in which the individuals are organized into pairs at random. Then, in the recombination step (lines 5-6), in accordance with the mating and mutation probabilities p_c and p_m, each pair undergoes crossover and the resulting offspring are mutated. These operators are adapted from the classical ones described in [13], so that they always generate feasible expression trees, i.e., they have maximum depth $\mathcal{D}$ and represent dimensional compliant expressions. After these steps, each individual is improved by local search (Algorithm 3) with probability p_{LS} (line 7). Finally, the replacement operator (line 8) passes the best two individuals from each pair of parents and their offspring to the next generation, which confers the algorithm an implicit form of elitism.

7. Experimental study

We conducted an experimental study aimed at analyzing the components of the proposed MGP, i.e., the LSA and the GP working alone, and to compare MGP to the GP giving both of them the same time. Remember that the GP considered here is quite similar to that proposed in [51], which represents the state of the art to evolve priority rules for the $(1, Cap(t)||\sum T_i)$ problem. To this end, we implemented a prototype in Java and ran a series of experiments on a Linux cluster (Intel Xeon 2.26 GHz. 128 GB RAM). This cluster distributes the workload into 28 processing nodes, so we opted to perform 28 independent runs for each problem instance.

The experiments were carried out across the set of instances proposed in [51]. This set includes instances of 60 jobs each and a maximum capacity of the machine of 10, and were generated with a procedure that tries to mimic the generation of instances produced in the actual EVCSP when all three lines of the charging station are working under the highest demand (60 EVs) and a

contracted power limited for at most 10 EVs charging simultaneously in each line. The procedure is as follows, MC is the maximum capacity of the machine, $U(a, b)$ is an integer sampled uniformly in the interval $[a, b]$, and $N(\mu, \sigma)$ is an integer sampled from a normal distribution with mean μ and standard deviation σ:

1. For each operation i, its processing time is set as $p_i = U(20, 100)$. Based on these values, we define $min_p_i = min\{p_i, i = 1, \ldots, n\}$ and $sum_p_i = \sum_{i=1}^{n} p_i$.
2. The initial capacity of the machine is set as $IC = U(1, MC)$, whereas its final capacity is $FC = 2$. Then, the capacity of the machine is defined by different intervals, first increasing the capacity one by one from IC to MC, and then decreasing it one by one until FC.
3. The duration of each capacity interval is set as $max\{min_p_i/4, N(R, 0.2 \times R)\}$, where $R = sum_p_i/S$ and $S = \sum_{j=IC}^{MC-1} j + \sum_{j=FC}^{MC} j$. This aims at enforcing the operations to be distributed over all the capacity intervals.
4. Finally, for each operation i, its due date is set as $d_i = U(p_i, B)$, where $B = R \times (2 \times MC - IC - 1)$ approximates the completion time of all the operations.

With this procedure, we generated 50 the purpose of training and 1000 more instances for the test. These instances are such that both the EDD rule and the ATC rule with $g \in \{0.25, 0.5, 0.75, 1.0\}$ produce schedules with total tardiness greater than 0.

We conducted the experimental study across these instances, but we also generated another set of instances with lower size to assess the performance of the algorithms depending on the problem size. These instances represent situations of the charging station working with about half contracted power, i.e., $MC = 5$ and half of the maximum demand, i.e., with 30 EVs.

7.1. Evaluation of LSA

To analyze the effectiveness of the LSA, we performed some preliminary experiments in which LSA was considered alone and applied to various sets of rules. The goal was to assess the capacity of the neighbourhood structures to produce improving rules, and which of the control strategies, namely hill climbing (HC) or gradient descent (GD), may be more appropriate. We have taken 6 sets of 28 rules each with different characteristics depending on the maximum depth and quality. Specifically, we considered random rules and rules evolved by the GP (GP rules), and in each case we considered rules with different depths D_i of 4, 6 and 8. In any case, we fixed the final maximum depth D_f to 8. All rules were tried to be improved by different versions of the LSA combining the two control strategies, HC and GD, and the three mentioned neighbourhoods, $\mathcal{N}^{1*}$, $\mathcal{N}^{2*}$ and $\mathcal{N}^{12*}$. Here it is important to remark that to apply $\mathcal{N}^{12*}$ the neighbours from $\mathcal{N}^{1*}$ and $\mathcal{N}^{2*}$ are shuffled at random to avoid bias towards one of the structures.

The results of these experiments are summarized in Tables 4 and 5, which show the results of the LSA starting from rules generated by the GP and random rules respectively. Regarding the first ones, we can observe that the LSA is able to produce moderate improvements in all cases; this is not surprising as the GP rules are generally good. There are not strong differences between the control strategies, HC and GD. The main differences observed are due to the neighbourhood structures; $\mathcal{N}^{1*}$ produces the worst results, but it takes much less time as it performs less iterations ($\#Ite$) and consequently it has to evaluate fewer different rules ($\#Dif$). In any case, the improvement of the rules comes at the cost of larger sizes.

The results produced by the LSA from random rules (Table 5) are quite different. As these are generally bad rules, the LSA was able to produce significant improvements in all cases. Now, there are strong differences between HC and GD in favor of the second, especially when they are combined with $\mathcal{N}^{2*}$ and $\mathcal{N}^{12*}$. Maybe, the most surprising results in these experiments are those from HC. We can observe that the best results are obtained in combination with $\mathcal{N}^{1*}$, even in this case the results are better than those from GD for the largest instances. Furthermore, when HC is combined with $\mathcal{N}^{2*}$ and $\mathcal{N}^{12*}$ the results are clearly worse than when these neighbourhoods are combined with GD. In that case, both the small size of the rules and the short time taken by LSA seem odd. The reason for this is that random rules are often quickly improved by neighbouring rules of small size produced by $\mathcal{N}^{2*}$, as for example $-d_j$ that is in fact equivalent to the EDD rule. Then these rules are not improved by any neighbour and so LSA terminates too soon.

From these results, the combination LSA+GD+$\mathcal{N}^{12*}$ seems to be the most appropriate. Therefore it is the option chosen to be combined with the GP in the proposed MGP.

7.2. Evaluation of GP

As pointed out, the GP considered is quite similar to that proposed and analyzed in [20]. For this reason, we do not perform here an exhaustive analysis, but only provide some results aimed at assessing its behaviour depending on the size of the rules (D_i) in the initial population and the maximum size (D_f) allowed to the rules evolved. We considered D_i and D_f values of 4, 6 and 8; and combinations of them so that $D_f \geq D_i$. Table 6 summarizes the results of these experiments. For each pair (D_i,D_f) the table reports results from 28 independent runs: the best and average results of the 28 rules on the training and test sets, the value of the best rule in training on the test set ($Best_T$), the average size and depth of the rules, the average number of generations reached by the time limit (300 minutes per run) and the number of different rules actually evaluated. At this point it is important to remark that the evaluated rules are stored so that if they appear in subsequent generations they do not have to be evaluated again.

From Table 6, we may draw the following conclusions: first, the quality, size and depth of the evolved rules are in direct ratio with D_f and they are almost independent from D_i. The number of generations reached is in inverse

Table 4: Analysis of the performance of LSA on rules evolved by GP considering different control strategies, GD and HC, different neighbourhood structures, $\mathcal{N}^{1*}$, $\mathcal{N}^{2*}$ and $\mathcal{N}^{12*}$, and different maximum initial depth, D_i. In any case, the maximum depth allowed for the rules modified by LSA is D_m=8. The average total tardiness produced by the initial rules for D_i 4, 6 and 8 are 1722.20, 1648.02 and 1625.31 on the training set and 1745.10, 1682.63 and 1664.41 on the test set respectively. For each combination of parameters, we report, on average, the total tardiness produced by the improved rules in the Training and Test sets, the Size and Depth of the improved rules, the number of iterations of LSA($\#Ite$), the number of generated ($\#Gen$) and feasible ($\#Fea$) rules, how many of these rules were different ($\#Dif$), and finally the time in seconds taken to improve a rule.

D_i	LSA	$\mathcal{N}$	Training	Test	Size	Depth	$\#Ite$	$\#Gen$	$\#Fea$	$\#Dif$	Time
4	GD	1*	1722.09	1745.20	12.71	4.00	1.07	48.46	48.46	44.96	4.07
		2*	1634.99	1671.27	22.71	6.32	7.21	20570.86	20264.89	13695.79	1179.46
		12*	1634.99	1671.27	22.71	6.32	7.21	20987.86	20681.89	13852.00	1200.57
	HC	1*	1722.09	1745.20	12.71	4.00	1.07	48.46	46.96	43.93	3.96
		2*	1666.14	1701.48	22.93	6.50	16.96	46637.00	29820.04	15113.71	1290.79
		12*	1660.91	1696.48	23.32	6.43	18.86	51830.61	33103.61	16503.54	1551.25
6	GD	1*	1647.29	1682.06	30.32	5.96	1.11	103.04	103.04	97.00	12.57
		2*	1632.01	1677.85	38.54	7.46	6.79	35519.86	34503.43	26019.82	3292.04
		12*	1631.98	1677.83	38.36	7.46	6.75	35932.14	34878.46	26038.79	3303.96
	HC	1*	1647.29	1682.06	30.32	5.96	1.11	103.04	97.18	91.57	11.36
		2*	1635.83	1677.44	33.61	7.11	12.64	57937.14	35595.71	22097.00	2410.93
		12*	1636.14	1678.07	33.71	7.04	11.89	56193.75	34244.11	21048.61	2296.04
8	GD	1*	1625.03	1664.50	48.25	7.71	1.25	165.36	165.36	157.75	24.14
		2*	1617.33	1661.26	54.04	7.93	6.61	45040.57	41172.68	32338.64	4751.21
		12*	1617.33	1661.26	54.04	7.93	6.61	45978.93	42111.04	32656.71	4794.11
	HC	1*	1625.03	1664.50	48.25	7.71	1.25	165.36	148.68	143.00	22.00
		2*	1618.13	1662.50	50.04	7.79	10.96	66486.00	40888.39	27977.96	3733.61

Table 5: Analysis of the performance of LSA on random rules. The average total tardiness produced by the initial rules for D_i 4, 6 and 8 are 5577.56, 6088.82 and 5189.72 on the training set and 5609.42, 6114.84 and 5236.84 on the test set respectively.

D_i	LSA	$\mathcal{N}$	Training	Test	Size	Depth	$\#Ite$	$\#Gen$	$\#Fea$	$\#Dif$	Time
4	GD	1*	1950.06	1970.03	7.89	3.96	3.11	91.75	91.75	71.14	5.25
		2*	1730.46	1761.94	16.32	5.61	7.29	17951.79	17536.29	10820.43	883.89
		12*	**1731.27**	**1762.59**	**15.89**	**5.61**	**7.14**	**17592.04**	**17182.07**	**10599.86**	**866.54**
	HC	1*	2144.88	2158.89	7.89	3.96	4.32	143.46	50.57	42.43	3.39
		2*	3507.07	3516.18	2.18	1.86	1.93	2966.86	834.25	286.61	19.21
		12*	3507.07	3516.18	2.18	1.86	1.93	3021.61	844.46	290.61	20.36
6	GD	1*	1911.84	1939.03	16.43	5.96	3.43	244.00	244.00	193.39	18.25
		2*	1750.04	1776.71	16.21	5.14	7.29	27050.29	26291.07	17178.36	1586.64
		12*	**1766.29**	**1795.40**	**15.82**	**5.14**	**6.39**	**23870.32**	**23417.93**	**15480.04**	**1444.11**
	HC	1*	1903.22	1927.84	16.43	5.96	7.00	595.57	151.18	132.04	13.14
		2*	3671.07	3679.78	2.36	1.89	2.00	5233.29	877.43	303.71	24.50
		12*	3671.07	3679.78	2.36	1.89	2.00	5332.71	888.75	308.07	24.36
8	GD	1*	1910.20	1943.99	60.50	8.00	2.93	722.54	722.54	586.50	107.79
		2*	1829.55	1859.12	19.00	4.68	4.68	54711.00	47982.68	34080.96	5439.11
		12*	**1824.27**	**1854.80**	**21.21**	**4.86**	**5.00**	**59094.54**	**51728.64**	**36607.39**	**5972.79**
	HC	1*	1898.72	1932.98	60.50	8.00	13.82	3626.50	667.00	606.07	113.79
		2*	3031.90	3042.73	2.43	2.04	2.00	15413.00	885.86	316.71	21.68

ratio with D_f showing that the genetic operators (crossover and mutation) and the evaluation of chromosomes take a longer and longer time with the size of the encoded rules. There is only one exception for $D_i = D_f = 4$, where

we can see that the number of generations is quite similar to the number of different chromosomes evaluated, which is an odd behaviour for an evolutionary algorithm. The reason for this is that restricting the depth of the expression trees to a very low value, such as 4, makes the search space so low that many feasible trees appear repeatedly along the search process.

All in all, we consider $D_i = 4$ and $D_f = 8$ in the comparison of the GP and MGP carried out in the next section.

Table 6: Results from GP with different combinations of D_i and D_f; for each combination 28 independent runs were done with 300 minutes of time limit in each one.

		Training		Test						
D_i	D_f	Best	Avg.	Best	Avg.	$Best_T$	Size	Depth	$\#Gen$	$\#Dif$
4	4	1653.30	1763.50	1649.93	1790.16	1649.93	13.11	4.00	42618.64	45871.96
	6	1609.80	1676.32	1629.40	1707.07	1637.38	28.04	5.93	3809.21	133605.11
	8	**1590.38**	**1622.93**	**1636.29**	**1659.18**	**1636.29**	**47.71**	**7.71**	**881.07**	**97894.71**
6	6	1605.58	1667.18	1636.99	1699.41	1650.41	27.64	5.96	4441.75	122488.07
	8	1599.54	1652.99	1636.80	1688.92	1639.60	47.71	7.82	1061.82	95362.11
8	8	1592.80	1666.87	1629.34	1709.66	1641.96	50.36	7.93	851.21	99581.96

7.3. Evaluation of MGP

To evaluate MGP, we considered a number of variants of the LSA combining the neighbourhood structures and establishing reasonable limits to the number of chromosomes selected to undergo local search in each generation. Moreover, when $\mathcal{N}^{2*}$ is applied, we had to restrict the number of neighbours evaluated as the cardinality of this structure is huge. The results of these experiments are summarized in Table 7. The first remarkable result is that the number of chromosomes evaluated by MGP is about twice the number of chromosomes evaluated by the GP in the same time limit (300 minutes). The main reason for this may be the easiness of the neighbourhood structures to generate a variety of feasible chromosomes in contrast to the fact that the crossover and mutation operators generate many infeasible chromosomes. Furthermore, the average size of the chromosomes grow more slowly due to the LSA generating chromosomes with similar size in opposition to crossover and mutation that may generate chromosomes of quite different sizes.

We can see that to allow MGP to complete a reasonable number of generations (50 or more), we have to restrict both the fraction of chromosomes that undergo local search and the number of neighbours considered. If we look at the average results, both in test and training, we can see that any version of MGP is better than the GP regarding both tardiness and the size of the rules, with only one exception in this case. Anyway, it seems that the options with $\mathcal{N}^{12*}$ produce the best results.

However, if we look at the best values, including $Best_T$, which refers to the application of the best rule in training to the test set, the results are not so clear. This may be due to the highly stochastic nature of the GP.

The boxplot in Figure 3(a) summarizes the average tardiness of the considered algorithms in the training set. To assess their differences we performed a

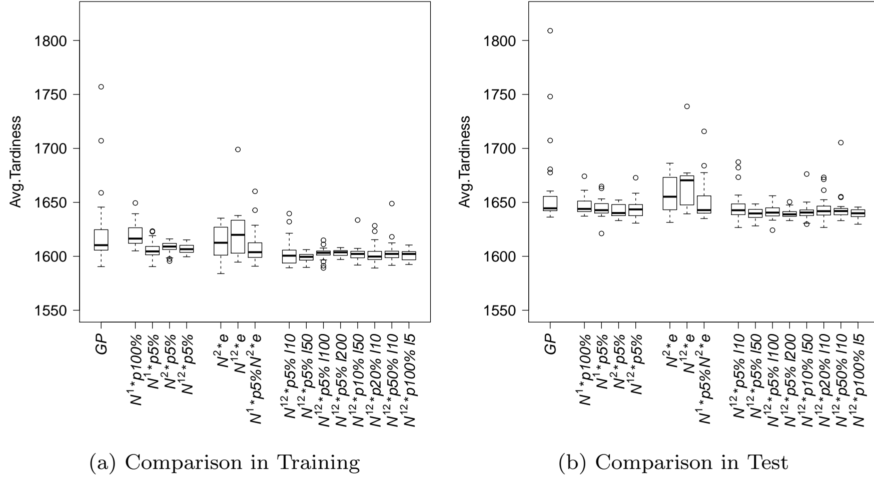
Table 7: Summary of results from the GP (first row) and MGP(the remaining rows) with different LS strategies. The notation $\mathcal{N}^X pY\%\ lZ$ means that the LSA exploits the neighbourhood $\mathcal{N}^X$, which is applied to a fraction Y of the population in each generation and only a number of at most Z neighbours are selected at random. The symbol e means that the LSA is only applied to the best chromosome in each generation. In all cases $D_i = 4$ and D_f=8.

	Training		Test						
Algorithm	Best	Avg.	Best	Avg.	$Best_T$	Size	Depth	$\#Gen$	$\#Dif$
GP	1590.38	1622.93	1636.29	1659.18	1636.29	47.71	7.71	881.07	97894.71
$\mathcal{N}^{1*}p100\%$	1605.02	1619.36	1637.17	1646.86	1647.87	27.71	7.18	17.46	179334.14
$\mathcal{N}^{1*}p5\%$	1590.42	1606.16	1621.12	1644.28	1637.08	44.29	7.93	113.61	157664.79
$\mathcal{N}^{2*}p5\%$	1595.56	1608.94	1633.07	1641.71	1634.62	25.36	6.96	4.50	175422.32
$\mathcal{N}^{12*}p5\%$	1599.54	1606.88	1630.71	1643.90	1634.70	30.14	7.25	4.75	175439.36
$\mathcal{N}^{2*}e$	1583.92	1613.69	1631.43	1658.58	1631.43	43.11	7.82	471.64	131534.96
$\mathcal{N}^{12*}e$	1594.58	1620.23	1639.34	1663.86	1642.95	42.46	7.93	629.21	122856.71
$\mathcal{N}^{1*}p5\%\ \mathcal{N}^{2*}e$	1590.80	1609.17	1634.93	1651.07	1635.21	41.36	7.93	94.29	162865.68
$\mathcal{N}^{12*}p5\%\ l10$	1589.34	1602.47	1626.68	1646.57	1635.72	47.89	7.96	385.18	139925.46
$\mathcal{N}^{12*}p5\%\ l50$	**1589.72**	**1599.01**	**1628.17**	**1639.57**	**1629.47**	**44.96**	**7.86**	**87.75**	**166036.29**
$\mathcal{N}^{12*}p5\%\ l200$	1597.04	1603.10	1633.05	1639.48	1638.93	38.07	7.86	29.82	200745.07
$\mathcal{N}^{12*}p10\%\ l50$	1591.86	1602.35	1629.79	1641.39	1638.62	39.64	7.89	60.32	191032.14
$\mathcal{N}^{12*}p20\%\ l10$	1589.12	1601.78	1626.74	1643.92	1630.87	46.07	7.93	168.14	158521.11
$\mathcal{N}^{12*}p50\%\ l10$	1591.66	1603.58	1632.99	1644.01	1644.55	43.32	7.96	97.82	187401.18

statistical test; as some of the results do not follow a normal distribution, we performed a Kruskal-Wallis test. The results of these tests show that there are statistical differences among the methods (*p-value*=4.14E-23) and that all versions of MGP that apply LS to a percentage of the population and that limit the number of neighbours at the same time take the first positions, being $\mathcal{N}^{12*}p5\%$ $l50$ the first one. These are the versions of MGP that reach a reasonable number of generations (50 or more) by the time limit. To analyze the differences between $\mathcal{N}^{12*}p5\%\ l50$ and each one of the remaining methods, we performed a number of post-hoc procedures using the Dunn test. Table 8 shows significant differences (✓) on the training set between $\mathcal{N}^{12*}p5\%\ l50$ and the methods that do not limit the number of neighbours; while there are not differences with the methods that apply LS to a percentage of the population and limit the number of neighbours at the same time. In both cases, these are sharp results as they are independent on the adjustment.

The boxplot in Figure 3(b) summarizes the average tardiness of the algorithms on the test set. There are also statistical differences between them (*p-value*=1.03E-14); even though the best method is $\mathcal{N}^{12*}p5\%\ l200$, while the best in training, $\mathcal{N}^{12*}p5\%\ l50$, is third in the test. The results of the Dunn test also show similar results between $\mathcal{N}^{12*}p5\%\ l50$ (the best in training) and the remaining methods, as we can see in Table 9.

In addition to the best and average results reached by the algorithms, it is interesting to analyze their convergence patterns and the distribution of the final solutions in the 28 independent runs. For this purpose, we selected 4 of the versions analyzed in Table 7, namely the GP and MGP considering $\mathcal{N}^{12*}p5\%$, $\mathcal{N}^{12*}e$ and $\mathcal{N}^{12*}p5\%\ l50$. These versions are denoted GP, MGP_p, MGP_e and MGP_{pl} in Figure 4. Figures 4 (a) and (c) show the evolution over time (not over



(a) Comparison in Training

(b) Comparison in Test

Figure 3: Boxplots execution.

Table 8: N×1 comparisons of training control algorithm ($\mathcal{N}^{12*}p5\%\ l50$) with the different algorithms for the Dunn‘s post-hoc test on training executions.

Adjustement / Algorithm	Bonferroni	Holm	Hochberg	Hommel	Hochberg & Benjamini	Hochberg & Yekutieli
GP	✓	✓	✓	✓	✓	✓
$\mathcal{N}^{1*}p100\%$	✓	✓	✓	✓	✓	✓
$\mathcal{N}^{1*}p5\%$	✓	✓	✓	✓	✓	✓
$\mathcal{N}^{1*}p5\%\ \mathcal{N}^{2*}e$	✓	✓	✓	✓	✓	✓
$\mathcal{N}^{12*}p100\%\ l5$						
$\mathcal{N}^{12*}p10\%\ l50$						
$\mathcal{N}^{12*}p20\%\ l10$						
$\mathcal{N}^{12*}p50\%\ l10$						
$\mathcal{N}^{12*}p5\%$	✓	✓	✓	✓	✓	✓
$\mathcal{N}^{12*}p5\%\ l10$						
$\mathcal{N}^{12*}p5\%\ l100$					✓	
$\mathcal{N}^{12*}p5\%\ l200$					✓	
$\mathcal{N}^{12*}e$	✓	✓	✓	✓	✓	✓
$\mathcal{N}^{2*}p5\%$	✓	✓	✓	✓	✓	✓
$\mathcal{N}^{2*}e$	✓	✓	✓	✓	✓	✓

generations) of the average value of the 28 rules on the training and test sets respectively. The GP is the worst algorithm in both cases, and $\mathcal{N}^{12*}e$, which applies LS to only the best chromosome in each generation, converges prematurely. It is MGP with $\mathcal{N}^{12*}p5\%\ l50$ the one that shows the best convergence pattern. In this case, the average tardiness in the test set seems to be stabilized after about 100 minutes, while it continues improving in the training set, what suggests overffiting on the training set, but without the rules getting worse on the test set.

Figure 4 (e) shows the evolution of the average size of the rules over time; again the GP produces the largest rate, but it is $\mathcal{N}^{12*}p5\%$ the one that produces the lowest rate.

Table 9: N×1 comparisons of training control algorithm ($\mathcal{N}^{12*}p5\%\ l50$) with different algorithms for the Dunn's test on test executions.

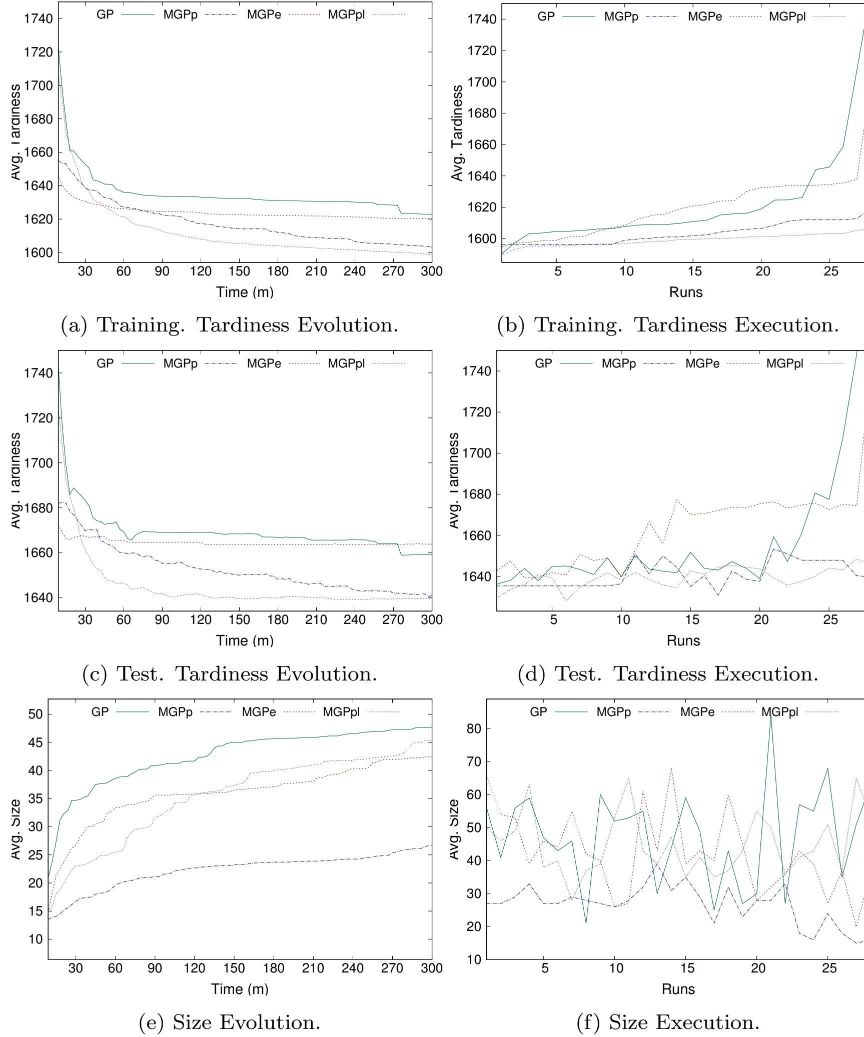
Algorithm \ Adjustment	Bonferroni	Holm	Hochberg	Hommel	Hochberg & Benjamini	Hochberg & Yekutieli
GP	✓	✓	✓	✓	✓	✓
$\mathcal{N}^{1*}p100\%$	✓	✓	✓	✓	✓	✓
$\mathcal{N}^{1*}p5\%$					✓	
$\mathcal{N}^{1*}p5\%\ \mathcal{N}^{2*}e$		✓	✓	✓	✓	✓
$\mathcal{N}^{12*}p100\%\ l5$						
$\mathcal{N}^{12*}p10\%\ l50$						
$\mathcal{N}^{12*}p20\%\ l10$						
$\mathcal{N}^{12*}p50\%\ l10$						
$\mathcal{N}^{12*}p5\%$						
$\mathcal{N}^{12*}p5\%\ l10$						
$\mathcal{N}^{12*}p5\%\ l100$						
$\mathcal{N}^{12*}p5\%\ l200$						
$\mathcal{N}^{12*}e$	✓	✓	✓	✓	✓	✓
$\mathcal{N}^{2*}p5\%$						
$\mathcal{N}^{2*}e$	✓	✓	✓	✓	✓	✓

Regarding the distribution of tardiness and sizes from the 28 independent runs, Figures 4 (b) and (d) show the values for the training and test sets respectively sorted in increasing order on the training set. The GP shows the largest dispersion, while $\mathcal{N}^{12*}p5\%$ and $\mathcal{N}^{12*}p5\%\ l50$ are the most stable, being the last one better in average as showed in Table 7. Finally, Figure 4 (f) shows the size of the 28 rules evolved following the same order; in this case $\mathcal{N}^{12*}p5\%$ stands out as the most stable and the one that produces the smallest rules, in accordance with the values reported in Table 7.

To evaluate MGP in searching for priority rules to solve instances with different size, we performed some experiments on the second set of instances, those with 30 jobs and a maximum capacity of the machine for 5 jobs at a time. We executed the GP alone and with the neighbourhood structures $\mathcal{N}^{12*}p5\%$ and $\mathcal{N}^{12*}p5\%\ l50$. The results are reported in Table 10, where we can see that MGP with $\mathcal{N}^{12*}p5\%$ produces quite similar results to the GP, but MGP with $\mathcal{N}^{12*}p5\%\ l50$ performs better. In the last case, the evolved rules are much better on the training set and also they are clearly better on the test set in average. However, the best rule in training is quite similar in test to the best rules evolved by the two other methods. In any case, the expected performance for a rule evolved by MGP with $\mathcal{N}^{12*}p5\%\ l50$ is much better than the expected rules evolved by both the GP and MGP with $\mathcal{N}^{12*}p5\%$. Furthermore, the results are better than those produced by the best version of the ATC rule, in this case with $g = 0.3$, which produced average tardiness of 444.28 and 445.87 on the training and test sets respectively.

8. Conclusions and future work

In this paper, we consider a Genetic Program (GP) that evolves priority rules for a scheduling problem and propose some neighbourhood structures that allow the GP to make local improvements on the evolved rules.



(a) Training. Tardiness Evolution.

(b) Training. Tardiness Execution.

(c) Test. Tardiness Evolution.

(d) Test. Tardiness Execution.

(e) Size Evolution.

(f) Size Execution.

Figure 4: Left: evolution of the best rule over time (0 - 300 minutes); (a) and (c) show the average tardiness of the 28 rules on training and test instances, (e) shows the average size of the evolved rules over time. Right: final rules; (b) and (d) show the tardiness values of the best rules reached in the 28 independent runs, (f) shows the size of the rules evolved by each algorithm in each run. The symbols MGP_p, MGP_e and MGP_{pl} denote $\mathcal{N}^{12*}p5\%$, $\mathcal{N}^{12*}e$ and $\mathcal{N}^{12*}p5\%\ l50$ respectively.

Taking as case of study the one machine scheduling problem with variable capacity over time and tardiness minimization, denoted $(1, Cap(t)||\sum T_i)$, the goal was to evolve new priority rules adapted to the features of a given benchmark set. Combining the proposed neighbourhood structures with the GP, we developed a Memetic Genetic Program (MGP) whose expected solutions, i.e., priority rules, are better and smaller than the rules expected from the original

Table 10: Summary of results from GP and MGP with two LSA strategies on the small instances. The running conditions are the same as in the results reported in Table 7.

	Training		Test				
	Best	Avg.	Best	Avg.	$Best_T$	Size	Depth
GP	428.08	467.87	440.25	475.27	441.83	43.21	7.75
$\mathcal{N}^{12*}p5\%$	430.04	466.68	441.33	474.51	441.46	42.61	7.86
$\mathcal{N}^{12*}p5\%\ l50$	420.60	427.37	439.85	443.58	442.60	46.68	7.96

GP.

Therefore, the major conclusion we can draw from this work is that it is possible to design local search operators, adapted to the problem domain, to improve the performance of a GP for evolving scheduling rules. We have seen that establishing neighbourhood structures well adapted to the problem is the hardest task, and that the evaluation of the neighbouring solutions may be very time consuming due to the huge cardinality of these structures.

This work leaves open some lines for future research. Undoubtedly, the development of simplified or surrogate models to evaluate the huge amount of neighbours is one of them. Besides, the combination of symbolic and numerical local search strategies, as suggested in [40], could help to make a fine tuning of the rules and to decrease their size. And, finally, considering other scheduling problems would be necessary to assess the performance of the proposed rule-generation paradigm.

Acknowledgements

This research has been supported by the Spanish Government under research projects TIN2016-79190-R and PID2019-106263RB-I00, and by the Principality of Asturias under grant IDI/2018/000176. Particularly, Francisco Gil-Gala is supported by the scholarship FPI17 / BES-2017-08203.

References

[1] C. Artigues, P. Lopez, P. Ayache, Schedule Generation Schemes for the Job Shop Problem with Sequence-Dependent Setup Times: Dominance Properties and Computational Analysis, Annals of Operations Research 138 (2005) 21–52.

[2] P. Brucker, B. Jurisch, B. Sievers, A branch and bound algorithm for the job-shop scheduling problem, Discrete applied mathematics 49 (1-3) (1994) 107–127.

[3] M. R. Sierra, R. Varela, Pruning by dominance in best-first search for the job shop Scheduling problem with total flow time, Journal of Intelligent Manufacturing 21 (2010) 111–119.

[4] C. Mencía, M. R. Sierra, R. Varela, Partially Informed Depth-First Search for the Job Shop Problem, Proceedings of the International Conference on Automated Planning and Scheduling 20 (1) (2021) 113–120.

[5] E. Talbi, Metaheuristics - From Design to Implementation, Wiley, ISBN 978-0-470-27858-1, 2009.

[6] G. R. Raidl, J. Puchinger, C. Blum, Metaheuristic Hybrids, Springer International Publishing, 385–417. 2019.

[7] A. Hernández-Arauzo, J. Puente, R. Varela, J. Sedano, Electric vehicle charging under power and balance constraints as dynamic scheduling, Computers & Industrial Engineering 85 (2015) 306 – 315.

[8] A. Hernández-Arauzo, J. Puente, M. A. González, R. Varela, J. Sedano, Dynamic Scheduling of Electric Vehicle Charging under Limited Power and Phase Balance Constraints, in: ICAPS'13: Proceedings of SPARK'13. Scheduling and Planning Applications workshop, 1–8, 2013.

[9] C. Mencía, M. R. Sierra, R. Mencía, R. Varela, Genetic Algorithm for Scheduling Charging Times of Electric Vehicles Subject to Time Dependent Power Availability, in: J. M. Ferrández Vicente, J. R. Álvarez-Sánchez, F. de la Paz López, J. Toledo Moreno, H. Adeli (Eds.), Natural and Artificial Computation for Biomedicine and Neuroscience, Springer International Publishing, Cham, 160–169, 2017.

[10] C. Mencía, M. R. Sierra, R. Mencía, R. Varela, Evolutionary one-machine scheduling in the context of electric vehicles charging, Integrated Computer-Aided Engineering 26 (1) (2019) 1–15.

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

[12] E. K. Burke, M. R. Hyde, G. Kendall, G. Ochoa, E. Ozcan, J. R. Woodward, A Classification of Hyper-Heuristic Approaches: Revisited, vol. 272 of *International Series in Operations Research & Management Science*, Springer International Publishing, 453–477, 2019.

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

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

[15] S. Nguyen, M. Zhang, M. Johnston, K. Tan, Dynamic Multi-objective Job Shop Scheduling: A Genetic Programming Approach, vol. 505 of *Automated Scheduling and Planning. Studies in Computational Intelligence*, 251–282, 2013.

[16] R. Hunt, M. Johnston, M. Zhang, Evolving "Less-myopic" Scheduling Rules for Dynamic Job Shop Scheduling with Genetic Programming, in: GECCO'14: Proceedings of the 2014 Annual Conference on Genetic and Evolutionary Computation, 927–934, 2014.

[17] H. Ingimundardottir, T. P. Runarsson, Discovering dispatching rules from data using imitation learning: A case study for the job-shop problem, Journal of Scheduling 21 (4) (2018) 413–428.

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

[19] D. Jakobović, K. Marasović, Evolving priority scheduling heuristics with genetic programming, Applied Soft Computing 12 (9) (2012) 2781 – 2789.

[20] F. J. Gil-Gala, C. Mencía, M. R. Sierra, R. Varela, Evolving priority rules for on-line scheduling of jobs on a single machine with variable capacity over time, Applied Soft Computing 85 (2019) 105782.

[21] M. Durasević, D. Jakobović, K. Knežević, Adaptive scheduling on unrelated machines with genetic programming, Applied Soft Computing 48 (2016) 419–430.

[22] E. K. Burke, M. R. Hyde, G. Kendall, J. Woodward, Automating the Packing Heuristic Design Process with Genetic Programming, Evolutionary Computation 20 (1) (2012) 63–89.

[23] S. Chand, Q. Huynh, H. Singh, T. Ray, M. Wagner, On the use of genetic programming to evolve priority rules for resource constrained project scheduling problems, Information Sciences 432 (2018) 146–163.

[24] S. Chand, H. Singh, T. Ray, Evolving heuristics for the resource constrained project scheduling problem with dynamic resource disruptions, Swarm and Evolutionary Computation 44 (2019) 897–912.

[25] M. Dumić, D. Šišejkovic, R. Čorić, D. Jakobović, Evolving priority rules for resource constrained project scheduling problem with genetic programming, Future Generation Computer Systems 86 (2018) 211–221.

[26] J. Branke, T. Hildebrandt, B. Scholz-Reiter, Hyper-heuristic Evolution of Dispatching Rules: A Comparison of Rule Representations, Evolutionary Computation 23 (2) (2015) 249–277.

[27] J. Branke, S. Nguyen, C. W. Pickardt, M. Zhang, Automated Design of Production Scheduling Heuristics: A Review, IEEE Transactions on Evolutionary Computation 20 (1) (2016) 110–124.

[28] F. Zhang, Y. Mei, S. Nguyen, K. C. Tan, M. Zhang, Multitask Genetic Programming-Based Generative Hyperheuristics: A Case Study in Dynamic Scheduling, IEEE Transactions on Cybernetics (2021) 1–14.

[29] F. Zhang, Y. Mei, S. Nguyen, M. Zhang, K. C. Tan, Surrogate-Assisted Evolutionary Multitask Genetic Programming for Dynamic Flexible Job Shop Scheduling, IEEE Transactions on Evolutionary Computation (2021) 1–15.

[30] M. Durasevi, D. Jakobovi, A survey of dispatching rules for the dynamic unrelated machines environment, Expert Systems with Applications 113 (2018) 555–569.

[31] P. Moscato, On evolution, search, optimization, genetic algorithms and martial arts: Towards memetic algorithms, Caltech concurrent computation program, C3P Report 826.

[32] F. Neri, C. Cotta, Memetic algorithms and memetic computing optimization: A literature review, Swarm and Evolutionary Computation 2 (2012) 1–14.

[33] J. Del Ser, E. Osaba, D. Molina, X.-S. Yang, S. Salcedo-Sanz, D. Camacho, S. Das, P. N. Suganthan, C. A. Coello Coello, F. Herrera, Bio-inspired computation: Where we stand and what's next, Swarm and Evolutionary Computation 48 (2019) 220–250.

[34] P. Merz, B. Freisleben, Memetic Algorithms for the Traveling Salesman Problem, Complex Systems 13 (4) (2002) 297345.

[35] L. Buriol, P. Moscato, P. França, A New Memetic Algorithm for the Asymmetric Traveling Salesman Problem, Journal of Heuristics 10 (2004) 483–506.

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

[37] D. Mattfeld, Evolutionary Search and the Job Shop. Investigations on Genetic Algorithms for Production Scheduling, Springer-Verlag, 1995.

[38] P. M. França, A. Mendes, P. Moscato, A memetic algorithm for the total tardiness single machine scheduling problem, European Journal of Operational Research 132 (1) (2001) 224–242.

[39] J. Amaya, C. Cotta, A. Fernández-Leiva, P. García-Sánchez, Deep memetic models for combinatorial optimization problems: application to the tool switching problem, Memetic Computing (2019) 1–20.

[40] L. Trujillo, E. Z-Flores, P. Juárez-Smith, P. Legrand, S. Silva, M. Castelli, L. Vanneschi, O. Schütze, L. Munoz, Local Search is Underused in Genetic Programming, Springer International Publishing, 119–137, 2018.

[41] P. Wang, K. Tang, E. P. K. Tsang, X. Yao, A Memetic Genetic Programming with decision tree-based local search for classification problems, in: CEC'11: IEEE Congress of Evolutionary Computation, 917–924, 2011.

[42] S. Nguyen, M. Zhang, M. Johnston, K. C. Tan, Automatic Programming via Iterated Local Search for Dynamic Job Shop Scheduling, IEEE Transactions on Cybernetics 45 (1) (2015) 1–14.

[43] S. Nguyen, Y. Mei, B. Xue, M. Zhang, A Hybrid Genetic Programming Algorithm for Automated Design of Dispatching Rules, Evolutionary Computation 27 (3) (2019) 467–496.

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

[45] S. Kaplan, G. 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) (2012) 276–285.

[46] F. J. Gil-Gala, C. Mencía, M. R. Sierra, R. Varela, Exhaustive search of priority rules for on-line scheduling, in: Proceedings of the 2020 Conference on ECAI 2020: 24th European Conference on Artificial Intelligence, 2020.

[47] R. Mencía, M. R. Sierra, C. Mencía, R. Varela, Memetic algorithms for the job shop scheduling problem with operators, Applied Soft Computing 34 (2015) 94–105.

[48] R. Mencía, M. R. Sierra, C. Mencía, R. Varela, Genetic algorithms for the scheduling problem with arbitrary precedence relations and skilled operators, Integrated Computer-Aided Engineering 23 (3) (2016) 269–285.

[49] T. Dou, P. Rockett, Comparison of semantic-based local search methods for multiobjective genetic programming, Genetic Programming and Evolvable Machines 19 (4) (2018) 535–563.

[50] M. Kommenda, B. Burlacu, G. Kronberger, M. Affenzeller, Parameter identification for symbolic regression using nonlinear least squares, Genetic Programming and Evolvable Machines 21 (2019) 471–501.

[51] F. J. Gil-Gala, M. R. Sierra, C. Mencía, R. Varela, Combining hyperheuristics to evolve ensembles of priority rules for on-line scheduling, Natural Computing, doi:10.1007/s11047-020-09793-4.