*Manuscript
Click here to view linked References

Evolving priority rules for on-line scheduling of jobs on a single machine with variable capacity over time

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

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

Abstract

On-line scheduling is often required in a number of real-life settings. This is the case of distributing charging times for a large fleet of electric vehicles arriving stochastically to a charging station working under power constraints. In this paper, we consider a scheduling problem derived from a situation of this type: one machine scheduling with variable capacity and tardiness minimization, denoted $(1, Cap(t)||\sum T_i)$. The goal is to develop new priority rules to improve the results from some classical ones as Earliest Due Date (EDD) or Apparent Tardiness Cost (ATC). To this end, we developed a Genetic Programming (GP) approach. The efficiency of this algorithm relies on some smart representation of the expression trees. Besides, we restrict the search space to that of dimensionally compliant expressions, which allows GP to reach single and clear solutions. We conducted an experimental study showing that GP is able to evolve new rules that outperform ATC and EDD using the same problem attributes and operations.

Keywords: Scheduling, One machine scheduling, Priority Rules, Genetic Programming, Hyperheuristics, Electric Vehicle Charging Scheduling

1. Introduction

One machine scheduling problems are of great interest in the field of scheduling for many reasons. Sometimes, they appear as natural relaxations of more complex problems and so they are useful to obtain lower bounds [4]. In other situations they appear as building blocks in the development of solutions to more complex scheduling problems [2].

This paper deals with a problem of this class introduced in [12] in the context of scheduling the charging times of a large fleet of Electric Vehicles (EVs) in the charging station designed in [29]. In this problem, a number of jobs must be scheduled on a single machine, with the objective of minimizing the *total*

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

tardiness objective function. The unique characteristic of this problem being that the machine has capacity to process more than one job at a time, but this capacity varies over time. So, it may be denoted as $(1, Cap(t)||\sum T_i)$ in the standard $\alpha|\beta|\gamma$ notation proposed in [10].

Solving the Electric Vehicle Charging Scheduling Problem (EVCSP) tackled in [12] amounts to solving a number of instances of the $(1, Cap(t)||\sum T_i)$ problem. Due to the computational intractability of this problem and the tight real-time requirements of the EVCSP, *on-line scheduling* represents the most (if not the only) suitable approach. On-line scheduling is usually performed by means of *schedule generation schemes* guided by *priority rules*. This is a common approach that has been successfully applied to a number of scheduling problems [1, 30]. In [12], the $(1, Cap(t)||\sum T_i)$ problem is solved by means of the *Apparent Tardiness Cost* (ATC) priority rule, commonly used in the context of scheduling with tardiness objectives. The problem was then solved in [21] by means of a genetic algorithm and later in [20] by a memetic algorithm producing better solutions. However, none of these approaches is suitable for on-line scheduling as it is required to solve the EVCSP.

The aim of this paper is the automated development of new, efficient, priority rules specifically adapted to address the $(1, Cap(t)||\sum T_i)$ problem. A natural way to cope with this task is the use of hyper-heuristics, as search needs to be conducted in a space of heuristics rather than in a space of solutions to the scheduling problem. Since priority rules are arithmetic expressions, they can be naturally represented by trees; so, we opted to investigate a Genetic Programming (GP) approach. We start from a conventional GP as it was proposed in [18] and then we propose some enhancements of this algorithm that improve its efficiency and the readability of the obtained expressions, namely, a method to represent expression trees that allows for very efficient implementations of some genetic operators, and the restriction of candidate solutions to well formed expressions from the dimensional point of view. Experimental results indicate that GP is capable of evolving effective priority rules for the $(1, Cap(t)||\sum T_i)$ problem, outperforming ATC and other classical priority rules. The results also provide insights of practical interest that motivate further research.

The remainder of the paper is organized as follows. Section 2 reviews some GP approaches to evolve priority rules for scheduling problems. In Section 3, we give the formal definition of the $(1, Cap(t)||\sum T_i)$ problem. Section 4 introduces the solving method proposed for the $(1, Cap(t)||\sum T_i)$ problem, which consists of two main components: schedule builder and priority rules. In this section we also present some previous results from classical priority rules and raised the hypotheses of this research. Section 5 describes the GP approach proposed to evolve new priority rules. In Section 6, we report the results of the experimental study conducted to evaluate the proposed GP approach. Finally, in Section 7 we summarize the main conclusions and outline some ideas for future work.

2. Evolving priority rules for scheduling problems

The terms *Dispatching Rule* (DR) and *Priority Rule* (PR) are commonly used in the scheduling literature to refer to "a simple heuristic that derives a priority index of a job from its attributes" [3]. Due to their low computational cost, PRs are well suited for *on-line scheduling*: the job with the highest priority among those available at a given time is scheduled next. In this section, we review some existing GP approaches proposed to discover dispatching or priority rules for scheduling problems, such as job shop (JSSP), one machine or unrelated parallel machines scheduling problems, among others. In some cases, the purpose is just to find a *good* priority rule which is then embodied into a schedule builder. Other works notice that a single rule may not suffice and focus on finding sets of rules to be applied collaboratively to solve instances with different characteristics.

Branke, Hildeblant and Schols-Reiter analyze in [3] three representation models for priority rules for the dynamic JSSP: expression trees commonly used in GP, Artificial Neural Networks (ANNs) and weighted linear combination of job properties. Their results show that expression trees evolved by GP perform slightly better than the other approaches.

Hart and Sim propose evolving sets of rules that are used collaboratively to solve problems [11] . They use GP to evolve a set of PRs for the static JSSP. They consider single and composite dispatching rules as terminal nodes, as for example SPT or ATC, in addition to some parameters. The rules are used in combination with several schedule builders, as for example the well-known Giffler and Thompson algorithm [9]. These rules are sequenced into heuristics. To produce a solution to the problem, each rule in the heuristic is applied in turn to schedule a single operation. An ensemble-based approach is also taken by Park et al. in [26], where the authors analyze some voting strategies to decide among the results from an ensemble or rules.

Ingimundardottir and Runarsson consider in [13] composite PRs for the JSSP given by linear combinations of 16 problem features, as for example total remaining work for a job or total idle time for all machines. The weights in the linear function are learned from a set of optimal solutions obtained by a MILP solver. Preference and imitation learning are used for this purpose.

Nguyen, Zhang and Johnston use GP to learn PRs for the Order Acceptance and Scheduling problem (OAS) directly from optimal scheduling decisions [25]. Instead of evolving just a single rule, a set of rules is evolved which is then used in a Forward Construction Heuristic (FCH): at each step the rule that produces the best local improvement is applied. One of the novelties of this model is that the fitness of a rule depends on how well the rule performs at each decision point (i.e., whether or not it takes an optimal decision) rather than on the final objective values of the schedule.

Durasevic, Jakobovi and Kneevi consider on-line scheduling for multiple unrelated parallel machines [8]. They propose evolving new priority rules with GP, incorporating some enhancements as dimension awareness to guarantee semantically correct rules, and some GP variant as gene expression.

The Resource Constrained Project Scheduling Problem (RCPSP) was also considered in some works. Chand et al. [5] and Dumic et al. [7] evolved PRs by GP, which outperform many of the existing ones. The same problem with dynamic resource disruptions was considered in [6].

The above represent just some of the approaches proposed recently to evolve priority rules for some families of scheduling problems and show that priority scheduling is an active line of research. In this paper, we build on some of the ideas proposed in these research works to devise new efficient rules for the $(1, Cap(t)||\sum T_i)$ problem.

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

As pointed out before, the problem of scheduling a set of jobs on a machine with variable capacity, denoted $(1, Cap(t)||\sum T_i)$, comes from the Electric Vehicle Charging Scheduling Problem (EVCSP) considered in [12]. Specifically, solving one instance of the EVCSP requires solving a large number of instances of the $(1, Cap(t)||\sum T_i)$. We refer the interested reader to [12] for further details. For the purpose of this paper it suffices to say that the $(1, Cap(t)||\sum T_i)$ problem needs to be solved in very short time due to the real time requirements of the EVCSP.

3.1. 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 i has duration p_i and due date d_i. The goal is to allocate starting times $st_i, 1 \leq i \leq n$ 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.,

$$C_i = st_i + p_i, \tag{2}$$

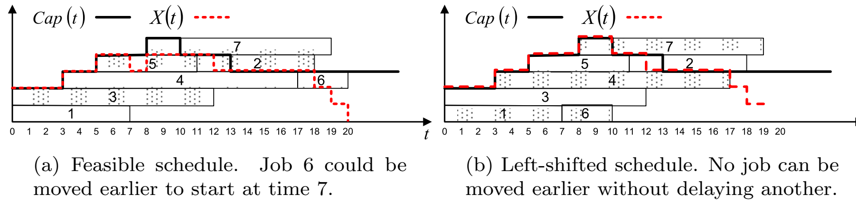
where C_i is the completion time of job i.

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.

Figure 1 (taken from [20]) shows an example of two feasible schedules for a problem with 7 jobs; the capacity of the machine varies between 2 and 5 over



(a) Feasible schedule. Job 6 could be moved earlier to start at time 7.

(b) Left-shifted schedule. No job can be moved earlier without delaying another.

Figure 1: Two feasible schedules 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 over time.

time. Due dates are not represented for the sake of clarity. As we can observe, in both schedules $X(t) \leq Cap(t)$ for all $t \geq 0$.

In order to analyze the complexity of the $(1, Cap(t)||\sum T_i)$ problem, we may look at a different but similar problem as $(m(t)||\sum T_i)$, where there are a number of $m(t)$ machines available over each time interval $[t, t+1)$. One particular case of this problem is when there are P machines that are continuously available, which is referred to as the $(P||\sum T_i)$ problem. This problem was proven to be NP-hard in the ordinary sense in [17]. At the same time, $(P||\sum T_i)$ is a particular case of $(1, Cap(t)||\sum T_i)$ when $Cap(t) = P$ for all $t >= 0$. Therefore, the $(P||\sum T_i)$ problem can be trivially reduced to the $(1, Cap(t)||\sum T_i)$ problem by a specific definition of $Cap(t)$, and so it follows that $(1, Cap(t)||\sum T_i)$ is NP-hard as well.

4. Existing approaches and working hypotheses

Given the real time requirements of the EVCSP, on-line scheduling based on schedule builders guided by priority rules may be an appropriate approach to solve the $(1, Cap(t)||\sum T_i)$ problem. In this section, we describe the schedule builder used by our approach and give some of the classical priority rules that may be used for this problem.

4.1. Schedule builder

Schedule builders (also known as *schedule generation schemes*) are a key element in the development of efficient scheduling algorithms, as they provide a way of computing a subset of the feasible schedules, allowing the definition of a search space. We use a similar schedule builder as the one proposed in [20], which produces *left-shifted schedules*, in which no job can be scheduled earlier without delaying the starting time of some other job. As an example, in Figure 1, the schedule (a) is not left-shifted, while (b) is a left-shifted schedule.

The schedule builder is depicted in Algorithm 1: it maintains a set US with the unscheduled jobs, as well as the consumed capacity $X(t)$ due to the jobs scheduled so far. US is initialized with all the jobs. In each iteration, the algorithm builds the subset US^* containing the jobs in US that can be scheduled at the earliest possible starting time, denoted $\gamma(\alpha)$, and selects one of these jobs to be scheduled.

Throughout the course of the algorithm, the job to be scheduled at each iteration is selected non-determiniscally. Nonetheless, Algorithm 1 always yields a feasible left-shifted schedule; for example, the sequence of choices $(1, 3, 4, 5, 6, 7, 2)$ would lead to building the schedule in Figure 1(b). In addition, any left-shifted schedule may be obtained considering the appropriate choice in each iteration. In other words, the scheduler searches in the whole space of left-shifted schedules, which is *dominant* for the $(1, Cap(t)||\sum T_i)$ problem, i.e., it contains at least one optimal schedule [20].

The schedule builder may be guided by any priority rule or heuristic, as we show in the next section. Besides, it could be embedded as a decoder in a genetic algorithm, for example, as done in [20] with a similar scheduler.

4.2. Priority rules for the $(1, Cap(t)||\sum T_i)$

A schedule builder, as the one shown in Algorithm 1, 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 *on-line 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 *Earliest Due Date* (EDD) or *Shortest Processing Time* (SPT) rules; the first one picks the operation with the smallest due date, while SPT selects the one with the least duration; in other words, they calculate priorities for an eligible job j as $\pi_j = 1/d_j$ and $\pi_j = 1/p_j$ respectively. These two rules are often used for objective functions that are non decreasing with the completion time of the jobs, as for example the makespan, the lateness or even the 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 on 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. [28, 15]); with this rule, the priority of each job $j \in US^*$ is given by

---

Algorithm 1 Schedule Builder

Data: A $(1, Cap(t)||\sum T_i)$ problem instance $\mathcal{P}$.
Result: A feasible schedule S for $\mathcal{P}$.
$US \leftarrow \{1, 2, ..., n\}$;
$X(t) \leftarrow 0; \forall t \geq 0$;
while $US \neq \emptyset$ **do**
    $\gamma(\alpha) = min\{t' | \exists u \in US; X(t) < Cap(t), t' \leq t < t' + p_u\}$;
    $US^* = \{u \in US | X(t) < Cap(t), \gamma(\alpha) \leq t < \gamma(\alpha) + p_u\}$;
    Non-deterministically pick job $u \in US^*$;
    Assign $st_u = \gamma(\alpha)$;
    Update $X(t) \leftarrow X(t) + 1; \forall t$ with $st_u \leq t < st_u + p_u$;
    $US \leftarrow US - \{u\}$;
end
return *The schedule* $S = (st_1, st_2, ..., st_n)$;

Table 1: Summary of results from [20]. Average total tardiness obtained by different priority rules EDD, SPT, ATC with four values of parameter g (0.25, 0.5, 0.75, 1.0) and a Genetic Algorithm (GA).

			ATC				GA		
n	EDD	SPT	0.25	0.5	0.75	1.0	Best	Avg	Time(s)
15	8.28	14.03	7.35	7.17	7.27	7.50	6.53	6.53	13.13
30	26.56	55.84	19.36	18.93	18.81	19.09	17.73	17.76	21.49
45	46.52	137.74	36.25	36.20	35.83	36.75	33.31	33.45	30.22
60	131.31	262.54	90.59	89.86	89.03	89.23	86.84	87.19	38.48
Avg	53.17	117.93	38.39	38.04	37.74	38.14	36.10	36.23	25.83

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

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 is decreasing with the slack time to its due date $d_j - \gamma(\alpha) - p_j$.

4.3. Some previous results and working hypotheses

Table 1 reproduces some results reported in [20] obtained by the rules EDD, SPT and ATC combined with a schedule builder similar to the one described in Algorithm 1, and by a genetic algorithm proposed therein (GA), over a set of 120 instances distributed in four sets having a different number of jobs (15, 30, 45, 60) with 30 instances each. As we can see, ATC produces much better results than both EDD and SPT, the results of the latter being actually poor, as can be expected due to the fact that this rule does not consider any information related to the tardiness objective. Besides, the performance of ATC depends on the value of the parameter g; the best value of g depending in turn on the size of the instances n. Furthermore, the ATC rule yields worse results than GA, which of course takes much longer time than the priority rules. These facts lead us to formulate the following hypotheses:

1. The ATC rule may be outperformed by new rules having a different structure or more detailed information of the problem domain, or just considering other parameters.
2. Given a benchmark containing instances with a similar structure, there may exist priority rules that are well adapted to this particular benchmark.

5. Evolving new priority rules with Genetic Programming

From the hypotheses above, our purpose is to devise new dispatching rules for the $(1, Cap(t)||\sum T_i)$ problem. To this end, we propose using hyper-heuristics,

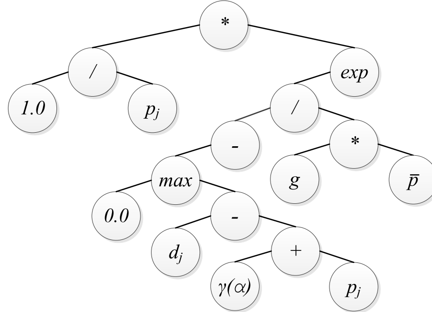


Figure 2: Expression tree representing the ATC rule.

as they provide a natural way of searching over a (sub)space of the heuristics that solve a given problem. As we are interested in devising some arithmetic expression, as that of the ATC rule given in Equation (4), Genetic Programming (GP) [18] represents a good choice as it provides a way of evolving tree structures. We start considering a conventional GP approach as the one proposed in [18] and then introduce some variations and improvements detailed below.

5.1. Functional and terminal symbols

The first step in the design of a GP approach is selecting the sets of terminal and function nodes to build the candidate trees. Terminal symbols represent the elementary properties that are considered relevant to establish jobs' priorities as, for example, processing times, due dates, etc., as well as some constants. Function symbols are the elementary arithmetic operations and some other unary and binary functions.

Table 2: Functional and terminal symbols used to build expression trees.

Functionals		
	Binary	- + / * *max min*
	Unitary	- pow_2 *sqrt exp ln*
Terminals		
	Data	p_i d_i $\gamma(\alpha)$ $\bar{p}$
	Constants	0.0 0.1 ... 0.9 1.0

Our starting decisions are the following. Looking at conventional rules as ATC, EDD or SPT, we have chosen the functional and terminal sets of symbols showed in Table 2. This way, the evolved rules will exploit the same attributes and operations as the conventional hand-made rules; therefore we could make a fair comparison between new and conventional rules, and analyze the extent to which these attributes are relevant for designing priority rules. Figure 2 shows the tree representing the ATC rule. EDD and SPT have much simpler representations.

5.2. Grammars and expression trees

The expression trees that can be generated from the chosen alphabet must be restricted by means of some grammar. Therefore, the grammar is a key element as it defines the search space of the GP. We consider two grammars herein, termed $Gram1$ and $Gram2$ respectively. $Gram1$ is the simplest one and may generate any well formed arithmetic expression without any other restriction. So, we will have to deal with some issues as division by 0 or the generation of some subexpressions as, for example, $0.7 + p_i$ or $\sqrt{p_i} + d_i$, which are not *dimensionally correct* (or *dimensionally compliant*) and so they may not be very natural and may give rise to complex and not too rational priority rules.

As pointed out in [16], evolving dimensionally correct formulae may contribute "to enhance the search efficiency by utilising the knowledge contained in the dimension information and to enhance the interpretability of the produced formulae". Dimensionally correct expressions are, for example, essential if we try to discover models through observations of physical phenomena where data are given together with the units of measurement. Only this kind of formulae may have clear physical semantics. In our problem, we have adimensional data; namely, constants, and data whose dimension is time (t); namely, processing times, due dates and earliest starting time.

For the above reasons, we propose to use another grammar, $Gram2$, that only generates dimensionally correct expressions, which are a subset of the productions that may be generated by $Gram1$. 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 *max* and *min* may also have 0.0 as one argument and any other expression as the other argument, the dimension of the result being that of the non-zero argument. The operations * and / can be applied to any pair of operands with independence of their dimensions, being the dimension of the result 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 adimensional expressions. Some examples of dimensionally compliant expressions are the classic SPT, EDD, and ATC rules, all having dimension t^{-1} at the root of the expression tree.

It is clear that restricting to dimensionally compliant expressions will require more sophisticated initialization, mating and mutation operators to guarantee feasible expressions. Figure 3 shows two expression trees: the first one (a) is dimensionally compliant, while the second one (b) is not dimensionally compliant as it expresses a summation of two terms with dimension t and t^2 respectively.

Regardless of the grammar used, it is usual to restrict the size of the generated trees. This can be done by limiting the number of nodes, the depth of the tree or both. Otherwise, the GP could generate very large expressions of low practical use. These and other restrictions have to be considered in the derivation algorithm. We discuss this issue in further sections.

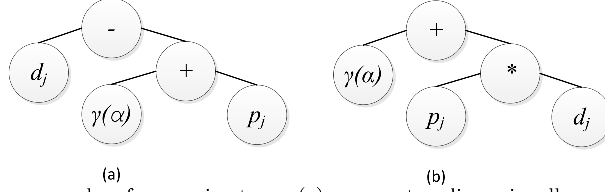


Figure 3: Two examples of expression trees: (a) represents a dimensionally correct expression, while (b) represents an expression that is not dimensionally correct.

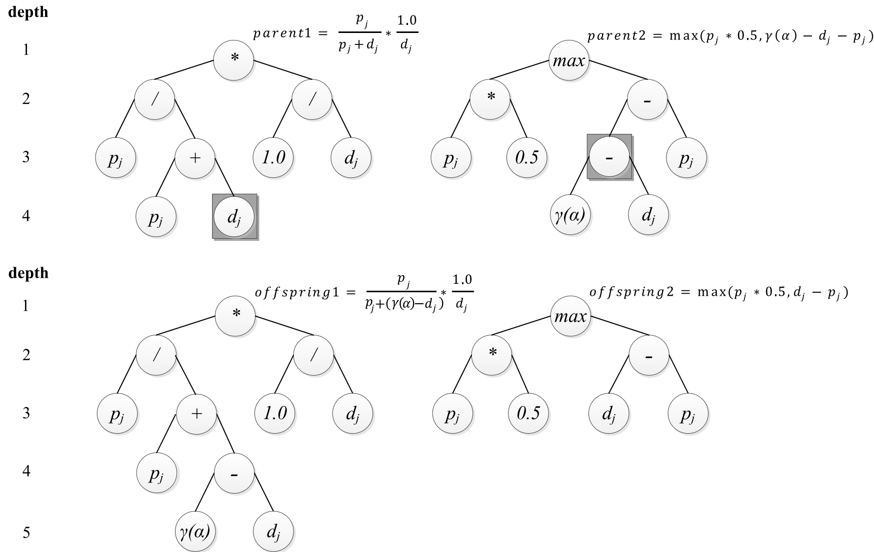


Figure 4: Illustration of single point crossover. Two parents with depth 4 are mated to generate two candidate offsprings. The subtrees rooted at the nodes remarked with a square box in the parents are swapped. The first offspring is discarded due to exceeding the maximum depth of 4 levels, while the second one is feasible.

5.3. Evaluation of chromosomes

The evaluation of chromosomes is the most time consuming component of the algorithm. It requires solving a battery of instances of the $(1, Cap(t)||\sum T_i)$ problem, the *training set*, with Algorithm 1 combined with the priority rule represented by the chromosome. As pointed out before, due to the fact that trees represent arbitrary expressions, it may be the case that some indetermination, as for example a division by 0, occurs during the evaluation process. To deal with this, we propose the following strategy: if in a given iteration a rule produces an indetermination for some job, another rule that cannot produce an indetermination, for example EDD, is chosen to decide in this iteration. This way, an expression producing many indeterminations is penalized as many decisions will be taken by EDD rule. The fitness values are simply calculated as the inverse of the summation of total tardiness values obtained for all the instances

in the training set, breaking ties in favor of rules with less size. This seems to be the most natural fitness value given the objectives of tardiness minimization and small size rules.

5.4. Evolutionary strategy

We consider three different evolutionary schemes, termed *Evol*1, *Evol*2 and *Evol*3 respectively. The first one is that proposed in [18]; namely, a generational schema with tournament selection and unconditional replacement. In *Evol*2 schema, chromosomes are organized into pairs in the selection phase and tournament replacement is done between every two parents and their two offsprings. This schema was successfully used to solve some scheduling problems by means of genetic and memetic algorithms [31, 22]. As we will see, when these strategies are used to solve our problem, they show quite different convergence patterns. In particular, *Evol*1 shows a low convergence rate after a relatively small number of generations, while *Evol*2 tends to produce low diversity on the population. For these reasons, we propose an alternative strategy, *Evol*3, whose rationale is to establish a selective pressure in between those in *Evol*1 and *Evol*2. So, *Evol*3 is defined as *Evol*2 but the chromosomes selected in the replacement phase are the best child and the best of the remaining three chromosomes having different fitness than the best child.

In all cases, elitism is considered; the best chromosome in a generation is passed unchanged to the next one.

5.5. Genetic operators

Genetic operators must guarantee feasibility of the produced expression trees, therefore they strongly depend on the grammar used. With *Gram*1, we could use crossover and mutation operators as proposed in [18]: one point crossover and single mutation. The first one selects one node in each of the two parents and swaps the subtrees rooted at these points. Single mutation just chooses one node in the tree and changes this node for a random subtree. In both cases, if the offspring is not feasible (i.e., it exceeds the maximum depth or size established), it is discarded and another one is tried. This process is repeated for a maximum number of trials. Figure 4 shows an example where two trees are mated to generate two new candidate trees. The second one is valid while the first one is discarded due to exceeding the maximum depth of 4 levels.

When *Gram*2 is used, there are restrictions due to dimensionality. Specifically, in crossover the mating points must represent subexpressions with the same dimension, and in mutation the new subexpression must have the same dimension as the old one.

5.6. Tree representation

Tree representation in memory is an important issue as it conditions the efficiency of the genetic operators. We propose to use an array-based representation of trees commonly used in the implementation of binary heaps. In this

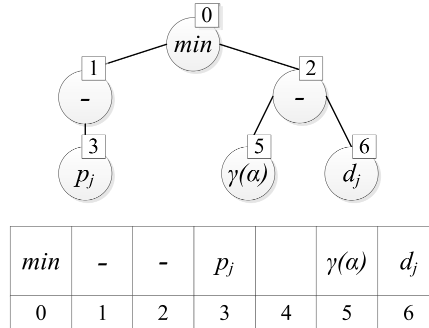


Figure 5: Array representation of an expression tree. The position 4 in the array has null value due to the left node of the root having only one child.

approach, trees are represented as single arrays of nodes organized such that for a node in position i, its parent is in position $(i-1)/2$, and their children are in positions $2i+1$ (left) and $2i+2$ (right) respectively. Besides, the maximum size, $\mathcal{S}$, and the maximum depth, $\mathcal{D}$, are related as $\mathcal{S} = 2^{\mathcal{D}} - 1$. The root node is in $i = 0$ and the nodes in positions $i \geq \mathcal{S}/2$ are leaf nodes, i.e., terminals. The remaining positions, i.e., positions $1 \leq i < \mathcal{S}/2$ may contain terminal or function nodes; if one of these nodes represents an unary function its left child is empty and if it represents a terminal node both children are empty. Figure 5 shows a example of expression tree and its array-based representation. One advantage of this representation is that it naturally allows for the control of maximum size and depth of the evolved trees.

5.7. Initial population

The use of the array-based representation facilitates the construction of feasible random trees, in particular when they must represent dimensionally compliant expressions. To create initial chromosomes representing dimansionally correct expressions, we propose the derivation procedure given in Algorithm 2. This algorithm starts from the maximum depth of the tree $\mathcal{D}$ and a probability $\mathcal{P}$. If $\mathcal{P} = 1$, the tree will be full, otherwise the tree will have branches of different lengths and will be more or less unbalanced depending on the random selection of null nodes. Therefore, taking different values of $\mathcal{P}$ we can obtain different distributions of random trees in the initial population. For example, taking $\mathcal{P} = 1$ for one half of the population and $\mathcal{P} < 1$ for the other half, we will have the so-called half-n-half method proposed by Koza [18].

The algorithm iterates over the array positions, from the last one $\mathcal{S} - 1$, where $\mathcal{S} = 2^{\mathcal{D}} - 1$ is the array size, backwards. In each iteration it fixes the value at position i, denoted $\mathcal{B}(i)$. The operation [.] produces the dimension of the expression inside. Let us analyze each case separately:

1. If $i \geq \mathcal{S}/2$, i.e., $\mathcal{B}(i)$ is a leaf, and $i\%2 = 0$, i.e., i is the position of the right child of the node in position j with $i = 2j + 2$, then the $\mathcal{B}(i)$ value

is chosen a terminal with probability $\mathcal{P}$ and null with probability $1 - \mathcal{P}$ ($\mathcal{U}\{\dots\}$ denotes random uniform selection from a set of items and, with a slight abuse of notation, we denote $\mathcal{P}\{a, b\}$ the probabilistic selection of a or b with probabilities $\mathcal{P}$ and $1 - \mathcal{P}$ respectively). Here it is worth to remark that this is the action taken always in the first iteration, i.e., for $i = \mathcal{S} - 1$.

2. If $i \geq \mathcal{S}/2$ and $i\%2 \neq 0$, i.e., i is the position of the left child of the node in position j with $i = 2j + 1$, then the $\mathcal{B}(i)$ value depends on the value of the right child, i.e., the node in position $i + 1$ (remember that if a non terminal node has only one child, this is the left child, so if the right child is not null the left child must not be null either). So, if $\mathcal{B}(i + 1)$ is null, as before, the $\mathcal{B}(i)$ value may be either a terminal or null. However, if $\mathcal{B}(i + 1)$ is not null, then the $\mathcal{B}(i)$ value must not be null.
3. If $\mathcal{B}(2i + 1)$ is null, then $i < \mathcal{S}/2$ and $\mathcal{B}(2i + 2)$ must be null (see the last sentence in previous paragraph). So, following a similar reasoning as before, $\mathcal{B}(i)$ is a terminal or null depending on i being left or right child of another node.
4. If $i < \mathcal{S}/2$ and $\mathcal{B}(2i + 1)$ is not null, then the $\mathcal{B}(i)$ value must be a function. The arity of this function being 1 if $\mathcal{B}(2i + 2)$ is null and 2 if it is not null. In the last case, if both subexpressions have the same dimension, the operator is chosen uniformly in $\{+, -, *, /, max, min\}$, if they have different dimension the operator is chosen from $\{*, /\}$ and if one of the subexpressions is "0.0" the operator is chosen from $\{max, min\}$. If the right child is null, then the operator is unary and so it is chosen from $\{-, pow_2, sqrt, exp, ln\}$ if the expression in $B(2i + 1)$ is adimensional and from $\{-, pow_2, sqrt\}$ otherwise.

The iteration over the array may terminate with a null chromosome, i.e., $\mathcal{B}(0) = NULL$, with probability $(1 - \mathcal{P})^{\mathcal{S}}$, which is very low for reasonable values of the parameter $\mathcal{P}$. In that case, a new iterative process must be started to generate another random expression tree. Finally the algorithm returns the built expression tree.

The algorithm may be used to generate a tree with maximum depth $\mathcal{D} - 1$, just filling the positions from $\mathcal{S} - 1$ to $\mathcal{S} - 2^{\mathcal{D}-1}$ with null values and then starting the iterative process from $i = \mathcal{S} - 2^{\mathcal{D}-1} - 1$. Analogously, a tree of any depth in $1..(\mathcal{D} - k), 1 \leq k < \mathcal{D}$ may be constructed starting the process in $i = \mathcal{S} - \sum_{1 \leq i \leq k} 2^{\mathcal{D}-k} - 1$.

Algorithm 2 builds expression trees that are dimensionally correct. However, it may be easily transformed into general tree builder just making the two conditions related with the dimension of the expressions to be always true.

6. Experimental Study

We have conducted an experimental study aimed at analyzing the behaviour of the proposed GP and, in particular, at assessing the quality of the obtained rules. To this end, we implemented a prototype in Java, and ran a series of

Algorithm 2 *Gram*2 derivation algorithm

Result: A feasible expression tree $\mathcal{B}$.
Data: Maximum depth $\mathcal{D}$ for the tree and a probability $\mathcal{P}$.
$\mathcal{S} \leftarrow 2^{\mathcal{D}} - 1$;
$i \leftarrow \mathcal{S} - 1$;
repeat
  while $i \geq 0$ **do**
    if $i \geq \mathcal{S}/2 \vee \mathcal{B}(2i+1) = NULL$ **then**
      if $i\%2 = 0$ **then**
        $\mathcal{B}(i) \leftarrow \mathcal{P}\{\mathcal{U}\{p_i,\, d_i,\, \gamma(\alpha),\, \bar{p},\, c\},\, NULL\}$
      else
        if $\mathcal{B}(i+1) = NULL$ **then**
          $\mathcal{B}(i) \leftarrow \mathcal{P}\{\mathcal{U}\{p_i,\, d_i,\, \gamma(\alpha),\, \bar{p},\, c\},\, NULL\}$
        else
          $\mathcal{B}(i) \leftarrow \mathcal{U}\{p_i,\, d_i,\, \gamma(\alpha),\, \bar{p},\, c\}$
        end
      end
    else
      if $\mathcal{B}(2i+2) \neq NULL$ **then**
        if $[\mathcal{B}(2i+1)] = [\mathcal{B}(2i+2)]$ **then**
          $\mathcal{B}(i) \leftarrow \mathcal{U}\{+,\, -,\, max,\, min,\, *,\, /\}$
        else
          if $\mathcal{B}(2i+1) = 0.0 \vee \mathcal{B}(2i+2) = 0.0$ **then**
            $\mathcal{B}(i) \leftarrow \mathcal{U}\{max, min\}$
          else
            $\mathcal{B}(i) \leftarrow \mathcal{U}\{*,\, /\}$
          end
        end
      else
        if $B(2i+1)$ *is adimensional* **then**
          $\mathcal{B}(i) \leftarrow \mathcal{U}\{-,\, pow_2,\, sqrt,\, exp,\, ln\}$
        else
          $\mathcal{B}(i) \leftarrow \mathcal{U}\{-,\, pow_2,\, sqrt\}$
        end
      end
    end
    $i \leftarrow i - 1$
  end
until $\mathcal{B}(0) \neq NULL$;
return The chromosome $\mathcal{B}$

experiments on a Linux cluster (Intel Xeon 2.26 GHz. 128 GB RAM). Due to the stochastic nature of GP, for each input data 30 independent runs were conducted with different initial random seeds and the best and average solutions were recorded.

6.1. The benchmark set

The experiments were carried out over a benchmark set of 1000 instances, generated by means of the procedure introduced in [20]. Each instance is characterized by the number of jobs (n) and the maximum capacity of the machine (MC). Given fixed n and MC, a random instance is generated using uniform distributions as follows (all sampled values are integers):

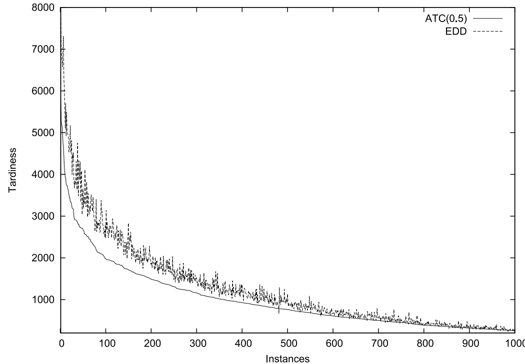


Figure 6: Tardiness distribution for the 1000 instances of the benchmark set obtained by ATC(0.5) and EDD rules

1. Each job $i \in \mathcal{J} = \{1, ..., n\}$ is assigned a random processing time $p_i \in \{1, ..., 100\}$.
2. Once all jobs have a processing time, they are assigned a random due date $d_i \in [p_i, \max(p_i + 2, \sum_{j \in \mathcal{J}} p_j/2)]$.
3. The capacity of the machine ($Cap(t)$) is generated as a unimodal function, with each constant interval taking a random duration in the range $[1, \sum p_j/MC]$. Both the initial and the final capacity of the machine is a random integer in $\{1, 2\}$.

In order to avoid under-constrained instances, we generated instances with $n = 60$ and $MC = 10$ until obtaining 2000 instances whose total tardiness are larger than 0 when solved with the ATC(0.5) rule. Then, the 1000 instances with the largest tardiness were selected. Figure 6 shows the total tardiness values of these instances sorted from the largest (5824) to the lowest (256) values over the x-axis. To appreciate the difference between the rules ATC(0.5) and EDD, the total tardiness values obtained by EDD rule are also showed. As we can observe, EDD rule is worse than ATC(0.5) in most cases.

To select the training set we proceeded as follows: the 1000 instances were sorted from 0 to 999 from the highest to the lowest tardiness produced by the rule ATC(0.5) (see Figure 6); then, they were distributed in 20 subsets of 50 instances each, so that subset i, with $0 \leq i < 20$ contains the instances such that $j\%20 = i$, with $0 \leq j \leq 999$, being j the index of the instance in the above ordering. This way, each candidate set contains instances distributed over the the whole range of tardiness. In the experimental study we considered the set $i = 10$ as the training set and remaining 950 instances for testing.

6.2. Parameter setting

As pointed by Poli et al. in [27] "genetic programming is in practice robust, and it is likely that many different parameter values will work". Therefore, a thorough parametric analysis is not as relevant as it is for other evolutionary

Table 3: Values fixed for some of the GP parameters.

Cross. and Mutation ratio	1.0 and 0.02 resp.
Population size	200
Number of generations	500
Elitism	1
Initial Population	Half full ($\mathcal{P} = 1$), half not full ($\mathcal{P} = 0.95$)

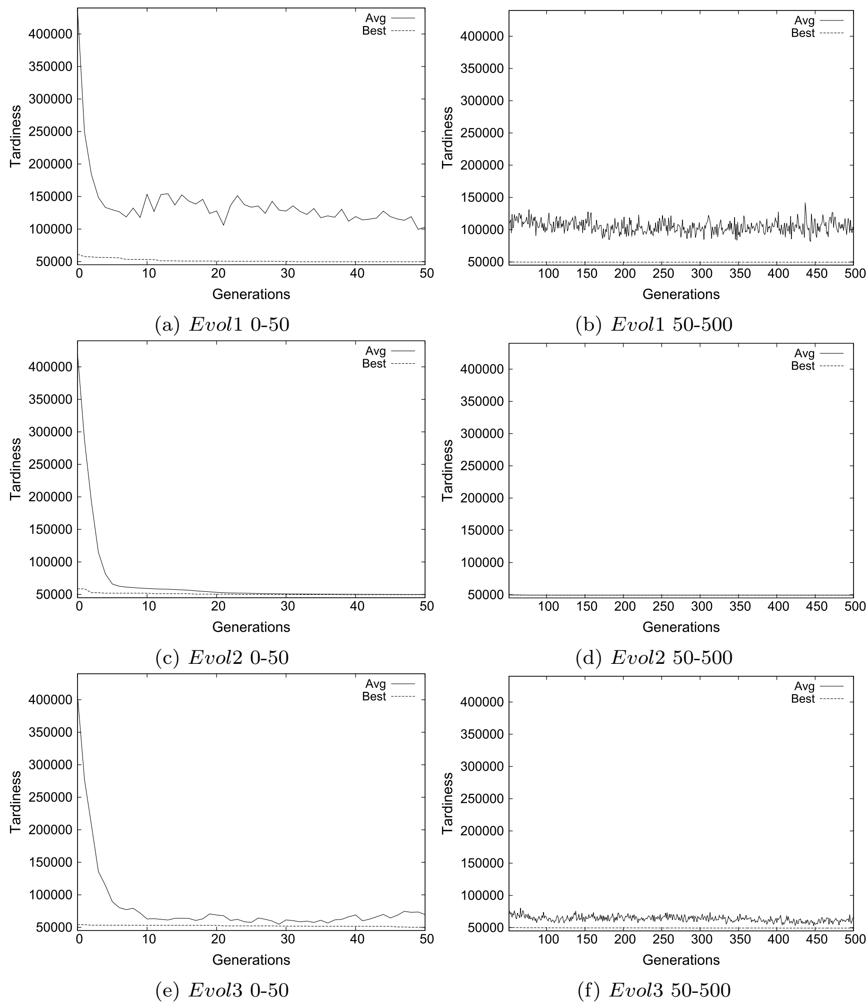
algorithms. In any case, there is general agreement that the population size and the tree size seem to be the most important parameters. Taking this into account, we started from a set of values for parameters as population size, number of generations and probabilities for crossover and mutation, which can be considered standard in the literature. Then, we analyzed how some variations in these parameters affect the behaviour of the GP algorithm. From these results, we fixed some of the parameter values as it is indicated in Table 3. These values are similar to those reported in other studies, for example [14], with the exception of the number of generations that in some cases is fixed to a small value, 30 or 50. We also observed that in the first 50 generations GP makes most of its work; however it is in the subsequent generations were GP is able to improve the rules so that they outperform the classical ones and at the same time the size of the rules is reduced. Under these conditions, the time taken in one execution of GP is about 600 minutes.

Besides, we analyze the three evolutionary schemes described in Section 5.4. Figure 7 shows the evolution pattern from one run of GP when learning from all 50 instances of the training set. As we can observe, the three strategies show quick convergence in average over the first 10 generations. However, after this generation the average value and the differences between average and best values are quite different.

$Evol1$ converges to large tardiness values showing low quality of chromosomes. $Evol2$ shows a clear premature convergence pattern as the average and best solutions have actually the same tardiness values after 50 generations. Finally, $Evol3$ presents the best convergence pattern: the average tardiness is much lower than it is for $Evol1$ and at the same time there are substantial differences between the average and best values; therefore it is the one with the best balance between quality and diversity and so it is able to improve the best solution after 450 generations.

Regarding the tree size and depth, we took ATC rule as initial reference, whose expression tree (see Figure 2) has depth 8 and size 17. In the previous experiments we considered maximum depth of 7 levels, and so maximum size of 32 nodes, but the analysis of the best value for the maximum depth is left to a thorough experimental study in following sections.

As pointed out in [14], the number of training instances is also an important parameter. In this work, we did not make a detailed analysis of this value; instead we have taken a number of 50 instances, which seems to be a reasonable number considering the values taken in other works. Besides, we have performed



(a) $Evol1$ 0-50

(b) $Evol1$ 50-500

(c) $Evol2$ 0-50

(d) $Evol2$ 50-500

(e) $Evol3$ 0-50

(f) $Evol3$ 50-500

Figure 7: GP evolution in one run with three evolutionary strategies $Evol1$, $Evol2$ and $Evol3$. The figures on the left show the average and best solution evolution over generations 0 to 50, while the figures on the right show the evolution from generation 50 to 500.

some complementary experiments in which GP evolved rules learning from only one example, and then analyzed the rules obtained and compared them to the rules evolved from 50 instances.

Table 4: Values considered for the remaining GP parameters (see Table 3).

Evolution strategy	*Evol3*
Grammar	General ($Gram1$), dimensionally compliant ($Gram2$)
Maximum tree depth ($\mathcal{D}$)	3, 4, 5, 6, 7, 8
Maximum tree size ($\mathcal{S}$)	$2^{\mathcal{D}} - 1$
Learning instances	50
Testing instances	950

6.3. GP results

Considering the above parameters, we performed a series of experiments aimed at assessing the performance of GP in obtaining good priority rules for the $(1, Cap(t)||\sum T_i)$ problem; in particular, it is our purpose to clarify the ability of the evolved rules to generalize across a large set of unseen instances. Besides, we tried to establish the differences between the rules obtained from the two grammars ($Gram1$ general and $Gram2$ dimensionally compliant) and the most convenient maximum depth allowed to the evolved trees. For each set of rules, we report tardiness values (best and average) on the training and test sets. Besides, dominance values; i.e, the percentage of times a given rule produces the best results among a given set of rules across a set of problem instances, are given. The size of the rules (average in the case of the evolved rules) is also showed. Additionally, we show results obtained by the Memetic Algorithm (MA) proposed in [20], some complementary results obtained from learning with only one rule and some preliminary results from ensembles of rules.

6.3.1. Evaluation of the grammars and maximum tree depth

The results obtained from the rules evolved by GP with both grammars and different values for the maximum depth are summarized in Table 5. We can observe that the dimensionally compliant rules are not better than the general rules, in terms of tardiness values. In fact, the general rules seem to be slightly better, which is in agreement with some preliminary results reported in [14], where the authors did not find statistical differences between canonical and "analitically correct" expressions. We performed Wilcoxon paired tests to assess the statistical differences between the results from the best rule and the average of the 30 rules on the test set. The results showed that the average values are better for random rules (p-value 2.79×10^{-4}), but the results from the best rule are better for dimensionally compliant rules (p-value 3.0×10^{-4}), which is in agreement with the larger standard deviation showed by these rules. At the same time, the size of the evolved rules is lower for dimensionally compliant ones, which is a clear advantage. Figure 8 shows the expression trees of the best

Table 5: Summary of results obtained by GP with *Gram*1 and *Gram*2 and maximum depth of the evolved trees varying from 3 to 8. Best, average and standard deviation (σ) 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. The columns Dom. refer to the percentage of times a result (Best/Avg.) is better than that from all the 11 best classical rules considered; namely ATC with 10 values of the parameter g and EDD (see Table 6).

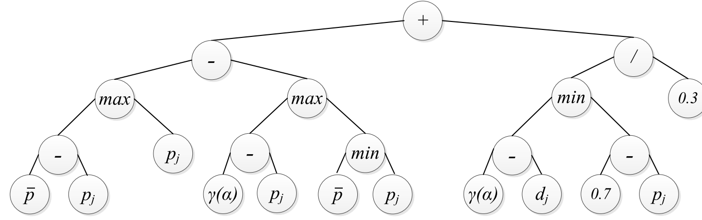
	Max.	Training				Testing				Avg.
Gram.	Depth	Best	Avg.	σ	Dom.	Best	Avg.	σ	Dom.	Size
	3	1027.50	1034.42	14.10	2/2	1021.11	1028.29	15.78	16/14	6
*Gram*1	4	987.56	1003.08	7.50	44/18	996.43	1007.47	7.43	43/25	13
(gen.	5	986.08	996.71	4.59	48/26	996.10	1001.56	3.14	42/31	20
rules)	6	984.28	995.78	4.71	48/28	993.67	1001.24	4.07	43/31	28
	7	985.16	996.43	5.97	48/28	994.85	1001.62	4.09	42/31	34
	8	984.96	995.04	5.78	44/29	994.64	1001.16	3.68	42/32	46
	3	1034.04	1049.89	15.85	2/1	1022.61	1043.09	20.48	15/9	6
*Gram*2	4	997.18	1012.29	23.73	30/15	1000.13	1015.42	23.96	33/22	10
(dim.	5	989.18	1000.68	11.06	42/24	995.37	1005.39	10.65	43/29	19
comp.)	6	986.32	996.80	6.07	52/28	992.14	1001.64	5.00	49/32	21
	7	986.82	998.39	8.37	48/25	994.66	1003.02	6.26	45/30	24
	8	986.92	1000.80	11.91	44/25	995.09	1006.44	10.69	42/29	29

rules obtained by both grammars considering a maximum depth of 5 levels. As we can observe, the dimensionally compliant rule is smaller and more rational as it does not contain any odd expressions, as for example $0.7 - p_j$, which appears in the rule of Figure 8(a). For the above reasons, we consider that dimensionally compliant rules are the best option and that for these instances (with 60 jobs each) 5 or 6 are the most appropriate values for the maximum depth of the evolved trees.

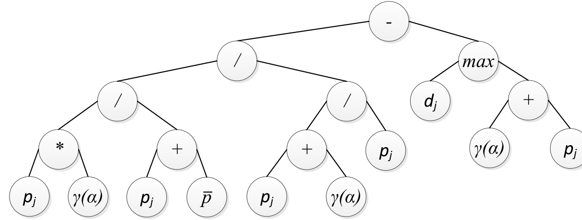
6.3.2. Comparison to classical rules and MA

Regarding the comparison to the classical rules (see Table 6), with the exception of those rules obtained giving the trees very low maximum depth (3 or 4), the average tardiness values of the 30 rules evolved with either grammar and tree depth are better than the values obtained from the classical rules Random, SPT, EDD and ATC with 10 different g-values. The Random rule gives each job a random priority distributed uniformly in $[0, 1]$; hence we report the best and the average values from 30 executions. Besides, as we can see in Table 5, if we look at dominance results (the percentage of instances for which a rule is better than EDD and all the 10 variants of ATC rules), we can observe that the best evolved rule is the best one on about half of the instances and on one among each four considering the average of the 30 rules.

At the same time, the results obtained by MA [20] are much better than those obtained by the evolved rules, showing that there is still room to improve. Here it is important to be aware of the differences in the execution time.



(a) Expression tree generated by $Gram1$. It is not dimensionally compliant as it includes the subexpression $0.7 - p_j$.



(b) Expression tree generated by $Gram2$. It is dimensionally compliant and the dimension at the root node is t.

Figure 8: Expression trees of the best rules generated by GP with grammars $Gram1$ and $Gram2$ and maximum depth of 5 levels.

Table 6: Summary of results obtained by the rules ATC (with different values of the parameter g), SPT, EDD, Random and MA. For the last two ones, results from 30 independent runs are reported.

			Training		Testing	
Classical rules	ATC	g-value				
		0.1	1060.20		1065.24	
		0.2	1033.50		1042.17	
		0.3	1008.60		1024.02	
		0.4	1005.34		1014.75	
		0.5	1000.52		1011.33	
		0.6	1002.80		1011.46	
		0.7	1007.74		1013.11	
		0.8	1010.70		1018.51	
		0.9	1021.98		1024.35	
		1.0	1030.82		1032.28	
	EDD		1256.84		1291.66	
	SPT		5473.18		5319.81	
	Random		Best	Avg.	Best	Avg.
			7980.18	8056.27	7779.10	7790.56
			Training		Testing	
Memetic Algorithm [20]			Best	Avg.	Best	Avg.
			950.18	950.63	959.06	959.51

6.3.3. *Generalization capability of the evolved rules*

It is reasonable to expect that a rule evolved from a set of training instances will perform better on these instances than over another set of (unseen) test instances. However, it is also obvious that there is no reason for some classical rule, as ATC for example, to be better in some set than in other one. Therefore, to evaluate the generalization capability of some evolved rule we could consider the ratios between the average results of some classical rule taken as reference and the evolved rule on both the training and the test sets. If both ratios are similar, we could conclude that the rule generalizes well, however if the ratio in testing are much lower than the ratio in training instances, then the generalization capability would not be satisfactory.

Going to our results, we may take ATC(0.5) as a reference rule, as it performed the best in the benchmark set considered. If we analyze the rule evolved by $Gram1$ and maximum depth 6, the ratios on training and testing are $1000.52/1011.33 = 0.9893$ and $996.80/1001.64 = 0.9952$ respectively; which are quite similar. Analogous results are obtained for the remaining evolved rules; for this reason we can say that the rules evolved by GP exhibit a high generalization capability.

6.3.4. *Training with only one instance*

The interest of these experiments is to clarify that the best rule for one instance may not be the best one for others. Obviously, if only one instance is used in the training phase, one can expect that the evolved rules will achieve very good performance on this instance, hopefully close to the performance of some off-line approaches as genetic algorithms. At the same time, one may expect that this rule will not perform so well on unseen instances.

In these experiments, we considered each of the 50 training instances, made 30 independent runs of GP and registered the 30 rules for each of the 50 instances (1500 rules in all). Then, each one of these rules was used to solve the 1000 instances of the benchmark set. The results are summarized in Table 7. “Best” are the tardiness values from the best rule for each of the 50 instances, specifically the first row “One instance” is the average value from each rule on the corresponding instance (average of 50 values), the row “Test set” is the average value of applying the 50 best rules to the 950 instances of the test set (average of $50 \times 950 = 47500$ values). Analogously, columns “Avg.” and “Worst” show values averaged for all 1500 rules and the 50 worst rules respectively.

Table 7: Summary of results from GP learning from only one instance.

	*Gram*1			*Gram*2		
	Best	Avg.	Worst	Best	Avg.	Worst
One instance	956.20	975.31	1010.32	956.00	974.49	1009.16
Test set	994.84	1125.08	1684.07	995.34	1118.22	1873.38

From these results we may draw the following observations. The value 956.20 in $Gram1$, “Best” and “One instance” compared to 986.32 (the value from the

rule evolved from the training set with $Gram1$ and maximum depth 6) makes it clear that learning from only one instance makes it possible to obtain rules which solve this instance much better than rules evolved from a set of 50 instances. Furthermore, this value, 956.20, is very close to 950.63, the result obtained by MA in average from the 50 instances of the training set, showing that the rules evolved from only one instance are able to solve that instance very effectively. At the same time, the value 994.84 for "Best" and "Test set" versus 992.14 (the value of the same rule as before on the test set) means that rules evolved from 50 instances generalize better than rules evolved from only one instance. Similar observations may be made considering the columns "Avg." and "Worst", as well as the results from $Gram2$.

6.3.5. Considering ensembles of rules

From the above observations, it seems clear that it is not easy to evolve a rule that may generalize well over a large set of unseen instances so that it can compete with off-line algorithms such as MA. Therefore, we propose considering ensembles of rules that will be exploited in the following way: an instance is solved by the schedule builder guided by each one of the rules of the ensemble; and the best of the obtained solutions is considered as the solution produced by the ensemble. This method would be feasible for ensembles of limited size due to the execution time being still suitable for the real time requirements of the EVCSP. To analyze the viability of this method we show the solutions reached by ensembles of various sizes. Firstly, 10 instances of the ATC rule with parameter g varying in $0.1 \ldots 1.0$ and then ensembles formed by the 3, 5, 10 and 15 best of the 30 rules obtained by GP with $Gram2$ and maximum depth of 6 levels for the expression trees. The results are summarized in Table 8.

Table 8: Summary of results from ensembles of rules with different sizes. Best rules of Size x refers to the s best of the 30 rules obtained in the experiments reported in Table 5 for maximum depth 6 and $Gram2$.

Ensemble	Size	Training	Testing
ATC(0.1,...,1.0)	10	984.18	989.57
Best rules	3	979.40	986.87
	5	975.02	982.28
	10	972.92	978.63
	15	972.34	977.09

Looking at the results from ATC, we can see that the values obtained in both training and test sets are rather similar to the values obtained by a single rule learned by GP with any grammar and maximum depth of 6 levels for the evolved trees. This result means that a single rule learned by GP performs similar to 10 classical rules working at the same time.

Moreover, putting some learned rules to work together reduces significantly the average tardiness and, as we can observe, the best trade-off between size and improvement for the considered sizes is 10. So, although they are preliminary,

these results strongly suggest that learning ensembles of rules is an interesting line for future research.

7. Conclusions

This paper studies the one machine scheduling problem with variable capacity, denoted $(1, Cap(t)||\sum T_i)$, and shows that Genetic Programming is a suitable approach to generate new priority rules, improving the best-performing classical ones for total tardiness minimization such as EDD or ATC. In order to make a fair comparison, we considered the same problem attributes and operations as in these rules. At the same time, we have seen that there is still room for improvement, as off-line algorithms, like memetic algorithms, are able to obtain even better solutions, of course running for much longer time than a schedule builder guided by priority rules. Therefore, we conjecture that by using more attributes of the problem, in particular some related to the capacity of the machine, better rules may be still evolved. Besides, it seems clear that no single rule can be the best one in every problem instance. Furthermore, the preliminary results obtained from ensembles of rules suggest that learning sets of rules covering problem instances with different characteristics may be a promising line of research. Another line of future research will be the use of some variants of GP as, for example, Cartesian Genetic Programming (CGP) [24, 23, 19]. CGP may be useful due to its inherent capability to limit the size of the evolved priority rules.

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] Artigues, C., Lopez, P., Ayache, P., 2005. Schedule generation schemes for the job shop problem with sequence-dependent setup times: Dominance properties and computational analysis. Annals of Operations Research 138, 21–52.

[2] Balas, E., Simonetti, N., Vazacopoulos, A., 2008. Job shop scheduling with setup times, deadlines and precedence constraints. Journal of Scheduling 11, 253–262.

[3] Branke, J., Hildebrandt, T., Scholz-Reiter, B., 2015. Hyper-heuristic evolution of dispatching rules: A comparison of rule representations. Evolutionary Computation 23 (2), 249–277.

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

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

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

[7] Dumic, M., Sisejkovic, D., Coric, R., Jakobovic, D., 2018. Evolving priority rules for resource constrained project scheduling problem with genetic programming. Future Generation Computer Systems 86, 211 – 221.

[8] Durasevic, M., Jakobovi, D., Kneevi, K., 2016. Adaptive scheduling on unrelated machines with genetic programming. Applied Soft Computing 48, 419 – 430.

[9] Giffler, B., Thompson, G. L., 1960. Algorithms for solving production scheduling problems. Operations Research 8, 487–503.

[10] Graham, R., Lawler, E., Lenstra, J., Kan, A., 1979. Optimization and approximation in deterministic sequencing and scheduling: a survey. Annals of Discrete Mathematics 5, 287 – 326.

[11] Hart, E., Sim, K., 2016. A hyper-heuristic ensemble method for static job-shop scheduling. Evolutionary Computation 24 (4), 609–635.

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

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

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

[15] Kaplan, S., Rabadi, G., 2012. 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.

[16] Keijzer, M., Babovic, V., 1999. Dimensionally aware genetic programming. In: Proceedings of the 1st Annual Conference on Genetic and Evolutionary Computation - Volume 2. GECCO'99. pp. 1069–1076.

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

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

[19] Manazir, A., Raza, K., 2019. Recent developments in cartesian genetic programming and its variants. ACM Comput. Surv. 51 (6), 122:1–122:29.

[20] Mencía, C., Sierra, M., Mencía, R., Varela, R., 2019. Evolutionary one-machine scheduling in the context of electric vehicles charging. Integrated Computer-Aided Engineering 26 (1), 49–63.

[21] Mencía, C., Sierra, M. R., Mencía, R., Varela, R., 2017. Genetic algorithm for scheduling charging times of electric vehicles subject to time dependent power availability. In: Natural and Artificial Computation for Biomedicine and Neuroscience. Springer International Publishing, Cham, pp. 160–169.

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

[23] Miller, J. F., 2011. Cartesian Genetic Programming. Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 17–34.

[24] Miller, J. F., Smith, S. L., 2006. Redundancy and computational efficiency in cartesian genetic programming. IEEE Transactions on Evolutionary Computation 10 (2), 167–174.

[25] Nguyen, S., Zhang, M., Johnston, M., 2014. A sequential genetic programming method to learn forward construction heuristics for order acceptance and scheduling. In: Proceedings of the 2014 IEEE Congress on Evolutionary Computation, CEC 2014.

[26] Park, J., Mei, Y., Nguyen, S., Chen, G., Zhang, M., 2018. An investigation of ensemble combination schemes for genetic programming based hyper-heuristic approaches to dynamic job shop scheduling. Applied Soft Computing 63, 72 – 86.

[27] Poli, R., Langdon, W. B., McPhee, N. F., 2008. A field guide to genetic programming. Published via `http://lulu.com` and freely available at `http://www.gp-field-guide.org.uk`.

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

[29] Sedano, J., Portal, M., Hernández-Arauzo, A., Villar, J. R., Puente, J., Varela, R., 2013. Intelligent system for electric vehicle charging: Design and operation. DYNA 88 (6), 640–647.

[30] Swamidass, P. M. (Ed.), 2000. Priority scheduling rules. In Encyclopedia of Production and Manufacturing Management. Springer US, Boston, MA, pp. 527–528.

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