IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

Received January 11, 2022, accepted January 31, 2022, date of publication February 14, 2022, date of current version March 4, 2022.

Digital Object Identifier 10.1109/ACCESS.2022.3151346

A Comparative Study of Dispatching Rule Representations in Evolutionary Algorithms for the Dynamic Unrelated Machines Environment

LUCIJA PLANINIĆ[1], (Member, IEEE), HRVOJE BACKOVIĆ[2], MARKO ĐURASEVIĆ[1], (Member, IEEE), AND DOMAGOJ JAKOBOVIĆ[1], (Senior Member, IEEE)

[1]Faculty of Electrical Engineering and Computing, University of Zagreb, 10000 Zagreb, Croatia
[2]Visage Technologies, 10000 Zagreb, Croatia

Corresponding author: Marko Đurasević (marko.durasevic@fer.hr)

This work was supported in part by the Croatian Science Foundation under Project IP-2019-04-4333.

ABSTRACT Dispatching rules are most commonly used to solve scheduling problems under dynamic conditions. Since designing new dispatching rules is a time-consuming process, it can be automated by using various machine learning and evolutionary computation methods. In previous research, genetic programming has been the most commonly used method for automatically designing new dispatching rules. However, there are many other evolutionary methods that use representations other than genetic programming that can be used to create dispatching rules. Some, such as gene expression programming, have already been used successfully, while others, such as Cartesian genetic programming or grammatical evolution, have not yet been used to generate dispatching rules. In this paper, six different methods (genetic programming, gene expression programming, Cartesian genetic programming, grammatical evolution, stack representation, and analytic programming) for generating dispatching rules for the unrelated machines environment are tested and the results for various scheduling criteria are analysed. It is also analysed how different individual sizes in the tested methods affect the performance and average size of the generated dispatching rules. The results show that, with the exception of grammatical evolution and analytic programming, all tested methods perform quite similarly, with results depending on the selected scheduling criterion. The results also show that Cartesian genetic programming is the most resistant to the occurrence of bloat and evolves dispatching rules with the smallest average size.

INDEX TERMS Unrelated machines environment, scheduling, solution representations, dispatching rules.

I. INTRODUCTION

Scheduling can be defined as the process of assigning a set of available jobs to a given set of limited resources in a way that satisfies some user-defined constraints and optimises one or more scheduling criteria [1]. Since most scheduling problems are NP-hard, various metaheuristic algorithms are used to solve scheduling problems [2]. Since these methods search the solution space for a concrete problem, they require that all information of the scheduling problem is available before the system is started. This means that such methods can only be applied for scheduling problems under static conditions, where it is known in advance when certain jobs will be released into the system and what their properties will be. However, in many situations such information is not available. This means that often it is not known in advance when new jobs will arrive into the system. Therefore, scheduling must be done under dynamic conditions, simultaneously with the execution of the system. Since metaheuristic methods cannot be used to solve such scheduling problems, many problem-specific heuristics, called dispatching rules, have been defined in the literature [3], [4].

Dispatching rules (DRs) create the schedule incrementally, which means that each time a job needs to be scheduled on a machine, the DR determines which of the available jobs should be scheduled on which machine. To determine which of the available jobs should be scheduled next, DRs use certain job and system properties to assign a priority to each job, and then select the job with the best priority value. For example, a DR could schedule jobs in the order

The associate editor coordinating the review of this manuscript and approving it for publication was Jamshid Aghaei.

This work is licensed under a Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 License.
For more information, see https://creativecommons.org/licenses/by-nc-nd/4.0/

in which they arrive, meaning that jobs which were released earlier in the system have a higher priority for scheduling. It is important to emphasise that when DRs calculate priorities for jobs, they only use the information that is currently available to them and only calculate priorities for jobs that have already been released into the system. For this reason, and because their execution time is substantially smaller than that of metaheuristic methods, DRs are usually the method of choice when solving scheduling problems under dynamic conditions. An important drawback of using DRs is that a single DR does not perform well for all possible problems and scheduling criteria. This would mean that new DRs have to be developed when no suitable DR exists for a particular criterion or scheduling condition. Unfortunately, designing new DRs is a lengthy and time-consuming process that usually needs to be performed by an expert for the specific domain.

To solve the previously described problem, in the last twenty years, a lot of research concerned with scheduling problems has focused on the automatic design of new DRs [5], [6]. Out of the many machine learning and evolutionary computation approaches, genetic programming (GP) is the most commonly used approach for generating new DRs. Dimopoulos and Zalzala [7], [8] were among the first to use GP to generate new DRs for the single machine environment, while Miyashita was the first to generate new DRs for the job shop environment [9]. The next several studies mostly focused on the application of GP in other machine environments, such as the flexible job shop [10] or the parallel uniform machines environment [11]. Several other studies focused on extending the GP method in different ways, such as adapting GP to detect overloaded machines in the system [12], or generating DRs by GP for problems with additional constraints such as breakdowns [13], batch scheduling [14], setup times and precedence constraints [15], or by using a variety of constraints [16]. Another researched topic is the generation of DRs for simultaneous optimization of multiple criteria, where different multi-objective and many-objective algorithms have been tested for optimising different combinations of scheduling criteria [17]–[20]. GP was also used to design due date assignment rules (DDARs) that approximate the due dates of jobs which arrive into the system. These DDARs were designed either alone [21], [22] or in combination with DRs, which required the development of new procedures for the simultaneous development of DDARs and DRs [23], [24]. The Order and Acceptance Scheduling (OAS) problem, which involves determining which jobs are accepted for scheduling in addition to scheduling jobs on specific machines, has also been studied in detail, and it has been shown that GP produces good DRs for this type of scheduling problem as well [25]–[29]. Recent work has also shifted the focus to some less studied scheduling problems such as the resource-constrained project scheduling problem [30], [31] and the single machine problem with variable capacity [32], [33].

To further improve the performance of the generated DRs, several studies analysed how different ensemble learning methods can be used to create ensembles of DRs which can perform better than a single DR [34]–[38]. Other studies focused on scheduling with uncertainties where some problem parameters were nondeterministic [39]–[41], or on applying surrogate-assisted GP to reduce the computational cost of evolving new DRs [42]–[44]. In [45], an evolution process visualisation framework was proposed to better understand the process of evolving DRs. In [46], the authors propose a new strategy for selecting subtrees in crossover and mutation operators. In it, the probability of selecting the subtree is based on its importance and operator types. The selection of appropriate problem instances for the evolution of DRs was studied in [47]. The study proposes an active sampling method that selects good instances during the evolution process. In [48], different scheduling generation schemes were compared to improve the performance of the generated DR, while in [49], the authors focused on adapting the generated DRs for static scheduling conditions. In [50], [51], a multitask GP model is used to generate heuristics for a broader range of problems to generate more general DRs. In [52], automatically designed DRs were used to generate the initial population of a genetic algorithm, which led to significantly better results compared to randomly generated populations or those initialised by manually designed DRs. In addition to GP, gene expression programming (GEP) also received some attention for generating new DRs [53]–[56], and usually achieved results as good as GP.

Since GP is mostly used in the generation of new DRs, several studies dealt with the comparison of different GP representations or with the comparison of GP with other methods for the automatic design of new DRs. One such study was conducted in [57], in which the authors compared three different representations of GP for generating new DRs. The first representation used certain job and system properties to determine which of the existing manually defined DRs should be applied for the current system conditions, making it quite similar to a decision tree. The second representation generates an entirely new DR using arithmetic expressions. Finally, the third representation is a combination of the previous two, as it allows additional new DRs to be designed and selected in place of the manually designed DRs. In [58], the authors compared DRs generated by GP with those generated by artificial neural networks (ANNs) and a linear representation defined as a weighted linear sum of several job and system attributes. The results show that the linear representation achieved the worst results, while GP and ANNs achieved quite similar results. However, the main advantage of GP over ANNs is that it develops DRs that are easier to interpret since it evolved them in the form of arithmetic expressions. In [56], the authors compared DRs generated by GP, GEP, iterative DRs (IDRs) [59] and dimensionally aware GP [60]. Their study showed that, apart from IDRs which are used under static scheduling conditions, there is usually no difference between the other three methods. Therefore, it mainly

depends on the users which of the tested methods they would apply.

From the literature review described earlier, it is evident that GP is mostly used for generating new DRs, while GEP is used only in some cases. However, no previous study has analysed in depth the impact of the different solution representations used by the different evolutionary computation methods on the quality of the generated DRs. Therefore, the objective of this paper is to compare several GP methods with different solution representations and determine which of these methods achieves the best performance on several scheduling criteria. In addition to the standard GP and GEP methods, both of which have been previously used to generate new DRs, this study additionally uses Cartesian GP (CGP), grammatical evolution (GE), stack representation, and analytic programming (AP) to generate new DRs for the unrelated machines environment. In addition to testing the performance of the six aforementioned methods on several scheduling criteria, this paper also examines the complexity of the expressions produced by the various methods to determine which methods produce the least complex expressions. The main contributions of this paper can be summarised in the following three points:

1) The first application of Cartesian GP, grammatical evolution, stack representation, and analytic programming for evolving new DRs.
2) A comparison of six methods for automatic evolution of new DRs, which complements several previous studies.
3) Analysis of the complexity of expressions generated by the six evolutionary computation methods.

The remainder of the paper is organised as follows. Section II provides a brief introduction to scheduling problems, a description of the solution representations used by the tested evolutionary computation methods, and describes how GP can be used to automatically generate new DRs. Section III describes the experimental design of the problem sets and additionally outlines the chosen parameter values for the different methods. The experimental results obtained by the six selected methods on several scheduling criteria are presented in section IV. Section V provides further analysis of the obtained results, mainly in terms of the average size of the generated DRs. Section VI provides a brief conclusion and identifies opportunities for future work.

II. BACKGROUND AND METHODOLOGY

In this section, the background of scheduling problems and methodology are described. All acronyms used in this paper are listed in Table 1 and notations in Table 2.

A. UNRELATED MACHINES ENVIRONMENT

The unrelated machines environment is defined as a machine environment consisting of n jobs and m machines. Each of the n jobs must be scheduled on one of the m available machines. Once a job is scheduled on a particular machine, it must be executed until it is finished, i.e., no preemption is allowed.

TABLE 1. A list of acronyms.

Acronym	Definition
DR	Dispatching rule
GP	Genetic programming
DDAR	Due date assignment rule
OAS	Order and acceptance scheduling
GEP	Gene expression programming
ANN	Artificial neural network
IDR	Iterative dispatching rule
CGP	Cartesian genetic programming
GE	Grammatical evolution
AP	Analytic programming
GFS	General function set
SGS	Schedule generation scheme
PF	Priority function

Additionally, each machine can only execute one job at any given time. Thus, if all machines are busy, it is necessary to wait until at least one machine becomes available, so that another job can be scheduled. The peculiarity of this environment is that each job j has a different processing time for each machine i, which means that a different processing time p_{ij} is defined for each job-machine pair. Since jobs become available during the execution of the system, the time at which jobs become available (r_j) must also be defined. Depending on which scheduling criteria is optimised, two additional properties can be defined for each job: the due date (d_j) and the weight (w_j). The due date indicates the time by which the job should finish its execution, otherwise a certain penalty will be imposed. On the other hand, the weight specifies the importance of jobs, which indicates that certain jobs should have a higher priority of being scheduled. Finally, C_j is used to denote the time when job j has finished its execution in the constructed schedule.

Although several scheduling criteria are defined in the literature [1], this study will focus on optimising the following four scheduling criteria:

- **Makespan** (C_{max}) - is defined as the largest completion time of all jobs:

$$C_{max} = \max_j(C_j). \tag{1}$$

- **Total flowtime**(Ft) - denotes the sum of flowtimes of all jobs:

$$Ft = \sum_j (C_j - r_j), \tag{2}$$

- **Weighted number of tardy jobs** (Nwt) - denotes the weighted sum of all tardy jobs (the formula is written using the Iverson notation in which the square brackets return 1 if the condition in the square brackets holds, otherwise it returns 0):

$$Nwt = \sum_j w_j[C_j > d_j], \tag{3}$$

TABLE 2. A list of notations.

Notation	Definition
n	Number of jobs
m	Number of machines
p_{ij}	Processing time of job j on machine i
r_j	Time when job j becomes available
d_j	Due date of job j
w_j	Weight of job j
C_j	Time when job j has finished its execution
C_{max}	Makespan, $C_{max} = \max_j C_j$
Ft	Total flowtime, $Ft = \sum_j (C_j - r_j)$
Nwt	Weighted number of tardy jobs, $Nwt = \sum_j w_j [C_j > d_j]$
Twt	Total weighted tardiness, $Twt = \sum_j w_j \max(C_j - d_j, 0)$
h	Head size of genes in GEP
t	Size of the tail of a gene in GEP, $t = h * (n_{max} - 1) + 1$
n_{max}	Maximum number of arguments of all nodes in the function set
n_c	Number of columns in CGP
n_r	Number of rows in CGP
l	Levels-back parameter in CGP
T	Terminal set in GE
N	Non-terminal set in GE
P	Set of production rules in GE
S	The start symbol from N in GE

- **Total weighted tardiness** (*Twt*) - denotes the weighted sum of tardiness values of all jobs:

$$Twt = \sum_j w_j \max(C_j - d_j, 0). \tag{4}$$

The final thing which needs to be specified about scheduling problems are the conditions under which the scheduling process is performed. In this study, scheduling is performed under dynamic conditions, which means that no information about the future of the system is known in advance. Thus, it is not known when jobs will be released into the system, and until they are released, the values of all other job properties are also unknown. This means that the schedule cannot be created before the system is executed because the necessary information is not available. Rather, the schedule must be created in parallel with the execution of the system.

B. SOLUTION REPRESENTATIONS FOR GP

This section contains a brief description of the representations used by the six evolutionary computation methods being compared.

The tree representation of solutions is the most commonly used representation in GP. The inner nodes of the tree represent *function* nodes that take the form of various arithmetic, Boolean, or other kinds of operations. The leaf nodes, on the other hand, are always *terminal* nodes representing certain variables or constants. The size of the expression is usually limited by a parameter that specifies the maximum depth of the tree. The representation is quite easy to interpret and allows GP to evolve expressions of various complexity. However, the representation usually suffers from a serious problem called *bloat* [61]. Bloat represents the uncontrolled growth of expression trees which occurs during evolution, although this growth does not improve their performance. The tree-based GP is very prone to this problem, since using the maximum depth of the trees makes it difficult to limit the size of the expressions precisely. Thus, selecting too large a value for this parameter will result in solutions that are huge and complex, while if too small a tree depth is chosen GP will not be able to generate expressions of the required complexity.

GEP [62] uses an alternative representation that stores the expression not in the form of a tree, but in a linear form, similar to what genetic algorithms do. In this way, GEP attempts to combine the simplicity of the representation and operators from genetic algorithms with the ability to evolve expressions. The individuals in GEP are of constant size, however, the part of the individual used to form the expression depends on its structure. Each GEP individual consists of one or more *genes*, where each gene consists of a constant number of nodes and represents an independent expression tree. Each gene can be divided into two parts, the *head* and the *tail* of the gene. The head of the gene represents the h initial nodes of the gene, where h is a user-specified parameter. This part of the gene can consist of any function and terminal nodes. The remaining nodes belong to the tail of the gene, whose size is calculated as $t = h * (n_{max} - 1) + 1$, where t is the size of the tail and n_{max} is the maximum number of arguments of all nodes in the function set. The tail of the gene consists only of terminal nodes, ensuring that the gene consists of enough terminal nodes to be decoded into a syntactically

correct expression. Each gene is decoded into an expression tree, however, depending on its structure, not all nodes are used to create an expression. Finally, all genes are combined using *linking nodes*, which are usually manually defined function nodes.

CGP [63] uses a graph-based representation to represent solutions, although the individuals are represented as a list of integer numbers which describe the structure of the graph. The representation uses three parameters, namely the *number of columns* (n_c), *number of rows* (n_r), and *levels-back* (l). The first two parameters define the number of nodes in the representation, which are arranged in a grid. The levels-back parameter determines which nodes from previous columns can serve as input for the current node. If levels-back is set to 1, only the nodes from the previous column can be used as input for the current node. Setting levels-back to n_c allows for the current node to connect to any of the nodes in the previous columns. It is often suggested to use a large number of columns and only one row, with levels-back set to the number of columns. For each node, it must be defined which function it represents and which nodes serve as its input. Since each node must be able to represent any of the available functions, the number of inputs is equal to the number of inputs of the function with the largest number of operands. If the node represents a function with a smaller number of operands, the additional inputs are ignored. The inputs of a node are denoted by integer numbers which represent the indices of the nodes that act as inputs. If there are n_i terminal nodes, then the indices $[0, n_i - 1 >$ are used to denote terminal nodes, while the indices $[n_i, n_i + n_c * n_r >$ are used to denote the outputs of the nodes in the grid. The outputs are encoded as additional numbers in the genotype that represent the indices of the nodes whose output is used as the program output.

GE [64] also represents the solution as a linear array of numbers. To decode this array of numbers into a meaningful expression, a predefined grammar is used. The grammar is defined with the tuple $< T, N, P, S >$, where T denotes the terminal set, N the non-terminal set, P the set of production rules, and S the start symbol from N. The goal is to generate an expression which consists only of terminal symbols. To achieve this, the production rules are usually applied in a way that they replace one non-terminal symbol with one or more terminal and/or non-terminal symbols. Non-terminal symbols in the expression are replaced from left to right using production rules, until there are no more non-terminal symbols in the expression. Since multiple production rules can be defined for each non-terminal symbol, the integer number determines which of the available production rules will be used. The benefits of this representation are that the genotype is quite simple and that no new operators need to be defined for this representation, but rather operators for the integer representation can be reused.

The stack representation [65] of solutions is defined by three parameters: the function set, the terminal set, and the maximum individual size. The terminal and function sets are the same as for the tree representation. They contain all the functions, variables, and constants that can make up an individual. The functions and terminals that compose an individual are stored in a linear form. To evaluate the individual, we must generate the mathematical expression that the individual represents. This is done by going through the elements of the individual. Each time a terminal is encountered, it is pushed onto the stack. When a function is encountered, the size of the stack is compared to the number of arguments of the function. If the number of terminals on the stack is greater than or equal to the number of arguments of the encountered function, the required number of elements are popped from the stack and the function is executed. The result obtained by executing the function is then pushed onto the stack. If the number of elements on the stack is less than the number of arguments, the function is simply ignored and the evaluation proceeds to the next element in the individual.

Analytic Programming (AP) [66] represents each individual as a linear array of floating point values from a range defined by the lower bound and upper bound parameters. An important component of AP is the general function set (GFS), which is composed of functions and terminals. The GFS is further divided into subsets based on the number of arguments of functions. When decoding an AP individual, the first step is to convert the floating point values into discrete indices, which represent indices of functions in the GFS. To do this, the original value is converted to a value within the range from 0 to the number of primitives in the GFS. A mathematical expression is then formed by replacing the indices with the functions in the GFS at the corresponding index. The described structure of the general function set is used to avoid the formation of invalid mathematical expressions when replacing indices with elements from the GFS. If the function that is supposed to replace an index has more arguments than there are elements left to the end of the individual, a function with fewer arguments is chosen. This ensures that there are enough elements after the function that can be used as its arguments.

C. GENERATING DRs WITH GP

DRs which will be generated by the previous solution representations can be divided into two parts, the schedule generation scheme (SGS) and the priority function (PF). The SGS defines how the entire schedule is created and which job should be scheduled on which machine. To determine which job should be scheduled on which machine, the SGS uses a priority function that ranks all job-machine pairs, and then selects the best pair and schedules the job on the selected machine. The benefit of such a separation is that a general SGS can be defined for a variety of problems, while the priority function that fits the specific problem or optimization criterion can be selected and used with the SGS. For this reason, the SGS is defined manually, while the PFs are generated using one of the previously described GP methods. Algorithm 1 represents the SGS used to generate schedules in the unrelated machines environment. The intuition behind

this SGS is that each time a job is released or a machine becomes available, the PF is used to determine the priorities for scheduling each of the available jobs on each of the machines, including those that are currently executing other jobs. Based on the calculated priorities, the most appropriate machine for each job is determined, and then all jobs for which the most appropriate machine is available are scheduled in order of their priorities. By calculating priorities even for machines that are busy, it is possible to include idle times into the schedule, as this allows for situations where the best machines may be busy for all jobs, and therefore no job is scheduled on other available machines, but the scheduling decision is postponed to a later time.

Algorithm 1 Schedule Generation Scheme Used by DRs Generated by GP

```
while unscheduled jobs are available do
    Wait until at least one job and one machine are available.
    for all available jobs and all machines do
        Obtain the priority π_ij of scheduling job j on machine i
    end for
    for all available jobs do
        Determine the best machine (the one for which the best value of priority π_ij is achieved).
    end for
    while jobs whose best machine is available exist do
        Determine the best priority of all such jobs
        Schedule the job with the best priority on the corresponding machine
    end while
end while
```

As mentioned earlier, the SGS uses a PF to determine the priority of scheduling a job on a particular machine. Since it is difficult to design these PFs manually, they are generated by the evolutionary computation methods defined earlier. To evolve new PFs, it is mandatory to define elements which will be used for constructing new PFs. Table 3 represents the set of terminal and function nodes used to construct new DRs. The first nine nodes in the table represent terminal nodes which provide certain information about jobs and the current status of the system. The *time* variable used in the definitions of some nodes represents the current system time. Additionally, it must be emphasised that the *dd*, *SL*, and *w* terminal nodes are only used in the development of DRs for optimising the due date related criteria (*Twt* and *Nwt*), since the information provided by these nodes is not meaningful for the other two optimization criteria. The last five nodes in the table represent the function nodes used by the methods to evolve expressions. Although many other function nodes can be used, a previous study has shown that GP produces the best results for this set of function nodes [56].

III. EXPERIMENTAL SETUP

To train and test the PFs, two independent problem sets are used. Both sets consist of 60 problem instances, each containing between 3 and 10 machines and between 10 and 100 jobs. The total fitness of an individual is calculated as the sum of the fitness values for each instance in the problem set. Since problems of different sizes have different values for certain criteria, all fitness values were normalised to remove dependence on the size of the problem instance. For more details on the problem set generation process, see [56].

In addition to defining the problem instances, it is also necessary to obtain the optimal parameters for each of the methods, since the quality of the obtained solutions strongly depends on the parameters used for their generation. Therefore, for each of the previous methods, the parameters were optimised for the *Twt* criterion. They were optimised in such a way that all parameters were fixed at certain predefined values, chosen as a rule of thumb. These values are given in Table 4 in the initial values row. Then, each parameter was tested with several different values, also given in Table 4, while the others were fixed either at the initial value or at the best value found after optimization. For each parameter combination, 30 experiments were performed and the parameter value that gave the best average value of these 30 executions was selected. Thirty runs were performed to obtain statistically accurate results. The parameters that were optimised, the values tested, and the best values obtained after the optimization procedure are shown in Table 4. For the CGP, one row was used with n_r columns and a levels back value equal to n_r. These parameter values were suggested by the author of the approach for problems where an arbitrary directed graph does not need to be implemented [67]. The smaller population values for CGP were intentionally tested, as it is usually suggested to use CGP with smaller population values. The final parameter values for all methods are listed in Table 5. These parameter values will also be used later in the optimization of the remaining three criteria, since optimising the parameters for each criterion individually would be too time consuming.

To ensure that the conclusions drawn based on the obtained results were meaningful, all experiments were performed at least 30 times and the best individual from each run was saved. Based on these 30 individuals, the minimum, median, and maximum values for each experiment were calculated and displayed. In addition, to determine if certain results were better than others, the Mann-Whitney statistical test was used to determine if there was a statistically significant difference between the different experiments.

IV. RESULTS

This section presents the results obtained by the tested methods on the four selected scheduling criteria. Table 6 presents the results obtained by the selected evolutionary computation methods for the optimization of the four scheduling criteria. Additionally, Figure 1 presents the results in the form of boxplots to better illustrate the distribution of the obtained

TABLE 3. Set of primitive nodes used for designing new DRs.

Node name	Description
pt	processing time of job j on the machine i (p_{ij})
pmin	the minimal processing time of job j on all machines: $\min_i(p_{ij})$
pavg	the average processing time of job j on all machines
PAT	patience - the amount of time until the machine with the minimal processing time for the current job will be available
MR	machine ready - the amount of time until the current machine becomes available
age	the time that job j spent in the system: $time - r_j$
dd	due date of job j (d_j)
SL	positive slack of job j: $\max(d_j - p_{ij} - time, 0)$
w_T	weight of job j
$+$	binary addition
$-$	binary subtraction
$*$	binary multiplication
$/$	binary secure division: $/(a,b) = \begin{cases} 1, & \text{if } \lvert b\rvert < 0.000001 \\ \frac{a}{b}, & \text{else} \end{cases}$
POS	$POS(a) = \max(a, 0)$

solutions. From the results, it is immediately apparent that no single method achieved the best results for all four scheduling criteria.

For the *Twt* criterion, the best results were obtained by GP. Although GEP performed worse to a small extent, there was no significant difference between its results and those of GP. However, it can be seen from the boxplot representation that GP achieved less scattered results than GEP, making it slightly more favourable. The Stack representation also performed slightly worse than GP and GEP when comparing by the median. However, it achieved the lowest overall minimum value. According to the box plot, the results were slightly more dispersed than GP and GEP, but less than GE or AP. CGP achieved results significantly worse than GP, but there was no significant difference between it and GEP. AP obtained results which are only better than CGP. The dispersion of the obtained results, which can be seen on the boxplot, is larger than most of the other tested methods. Finally, GE obtained the worst results among all six methods, which can also be seen from the fact that this method usually obtains quite dispersed results.

In the case of the *Nwt* criterion, the situation is a bit more interesting. For this criterion, the best results were again obtained by GP, GEP and Stack, with GEP obtaining a better median value, while Stack obtained the best overall minimum and maximum values. However, there is no significant difference between the results of these methods. CGP, GE, and AP obtained results significantly worse than the other two methods, but there was no significant difference between the results of CGP and GE, and CGP and AP. On the other hand, the results for AP were significantly better than the results achieved by GE.

For the *Ft* criterion, the best results were obtained by Stack, GEP and CGP. The statistical tests showed that there was no significant difference between the results of these three methods. For this criterion, Stack and GEP obtained results that were even significantly better than those of GP. Unfortunately, the results of GE and AP were significantly worse than those of the other three methods. GE obtained scattered results, and many evolved DRs performed poorly. In contrast, AP obtained the least dispersed results, but all of them performed poorly. Unfortunately, it is difficult to determine the reason for this in this criterion. One possibility is that the individual sizes chosen for these two methods were inappropriate for this criterion and that better performance could be obtained with a different individual size.

Finally, for the C_{max} criterion, it is most difficult to determine which of the tested methods performed the best. GEP achieved the best overall median value. However, both CGP and GE were able to develop a DR that performed better than any of the rules generated by GEP. Although GP also achieved a better median value than CGP and GE, the best DR found by GP was inferior to the best DRs generated by the other three methods. The statistical tests show that GP achieved significantly worse results than GEP, while there was no statistically significant difference between the results of GEP and CGP. Both AP and Stack achieved results that were significantly worse compared to all other methods for this criterion and their results were less dispersed than the other methods.

In addition to observing the performance of the different methods, it is also interesting to analyse the size of the decoded expressions of the evolved PFs. Table 7 gives an overview of the average sizes of the PFs generated for the

TABLE 4. Parameter values used for optimisation.

Algorithm	Parameter	Initial value	Tested values	Final (optimal) value
GP	Population size	100	100, 200, 500, 1000, 2000	1000
	Mutation probability	0.5	0.07, 0.1, 0.2, 0.5	0.3
	Tree depth	7	3, 5, 7, 9, 11, 13	5
GEP	Population size	100	100, 200, 500, 1000, 2000	1000
	Mutation probability	0.5	0.07, 0.1, 0.2, 0.5	0.3
	Number of genes	3	2, 3, 4, 5	3
	Head size	10	6, 8, 10, 12, 14	6
CGP	Population size	5	5, 20, 50, 500, 1000	500
	Mutation probability	0.3	0.3, 0.5, 0.7, 0.9	0.3
	Number of columns	100	100, 300, 500	100
GE	Population size	100	100, 200, 500, 1000, 2000	500
	Mutation probability	0.3	0.3, 0.5, 0,7, 0.9	0.7
	Genotype size	100	30, 50, 70, 100, 150, 200, 500, 1000	150
Stack	Population size	100	50, 100, 200, 500, 1000, 2000	2000
	Mutation probability	0.3	0.3, 0.5, 0,7, 0.9	0.5
	Genotype size	50	30, 40, 50, 60, 70, 100, 300	60
AP	Population size	500	50, 100, 200, 500, 1000, 2000	200
	Mutation probability	0.3	0.3, 0.5, 0.7, 0.9	0.3
	Genotype size	50	10, 30, 50, 70, 100	50

four scheduling criteria. The first thing to notice is that the different methods evolved expressions of very different sizes. GP and AP develop the largest expressions, typically comprising about 40 elements. GEP, on the other hand, evolved somewhat simpler expressions, usually with a size of about 30 elements. Stack evolved expressions that typically consisted of about 25 elements. The remaining two methods evolved the smallest expressions, consisting of about 18 nodes. It is obvious that the different methods have a preference for evolving expressions of different sizes. If the size of the evolved expressions is also of importance, it might even make sense to use methods which have a preference to evolve smaller expressions, such as GEP, CGP, and GE, especially since these methods have even been shown to achieve the same or even slightly better performance than GP, e.g., for the Ft and C_{max} criteria. The table also shows that for certain criteria all methods tend to evolve slightly smaller expressions. In addition, it can be seen that for the Nwt and C_{max} criteria, the methods generated expressions that were, on average, several elements smaller than those generated for the Twt and Ft criteria. These results could potentially mean that it might be more beneficial to evolve PFs of smaller sizes for these two criteria.

V. FURTHER ANALYSIS

A. COMPARISON OF DR SIZES FOR DIFFERENT MAXIMUM INDIVIDUAL SIZES

In this section, we will focus on further analysing how the chosen maximum individual size affects the average size and fitness of the DRs generated by the different evolutionary computation methods. Therefore, each of the methods will additionally be tested with several different maximum individual sizes.

Table 8 presents the results obtained for the different maximum tree depths used with GP. In addition to the average sizes of the evolved DRs, the table also contains the theoretical maximum size of an expression for the given depth. From the table, it can be seen that it is quite difficult to precisely control the size of the generated individual using the tree depth parameter. This can be seen from the fact that the maximum expression size grows exponentially with the increase of the tree depth. Naturally, this also affects the average size of the evolved PFs, which is only 13 for the smallest tested depth, while for the largest tested depth the PFs consisted of up to 440 elements on average. It is obvious that the tree size increases significantly with the depth of the tree. Therefore, it can be concluded that with the tree depth parameter it is not easy to control the size of the evolved PFs.

It is also interesting to observe how the different tree depths affect the quality of the generated DRs. Figure 2 shows the boxplot representation of the obtained results. The smallest tested depth produced quite poor results, with most DRs achieving similar performance. GP evolved DRs with the best quality when tree depth was set to 5, which can be seen from the fact that GP achieved the best median value at this tree depth value. As the depth increases, the results begin to deteriorate, as can be seen by the increasing median value of

TABLE 5. Final parameter values.

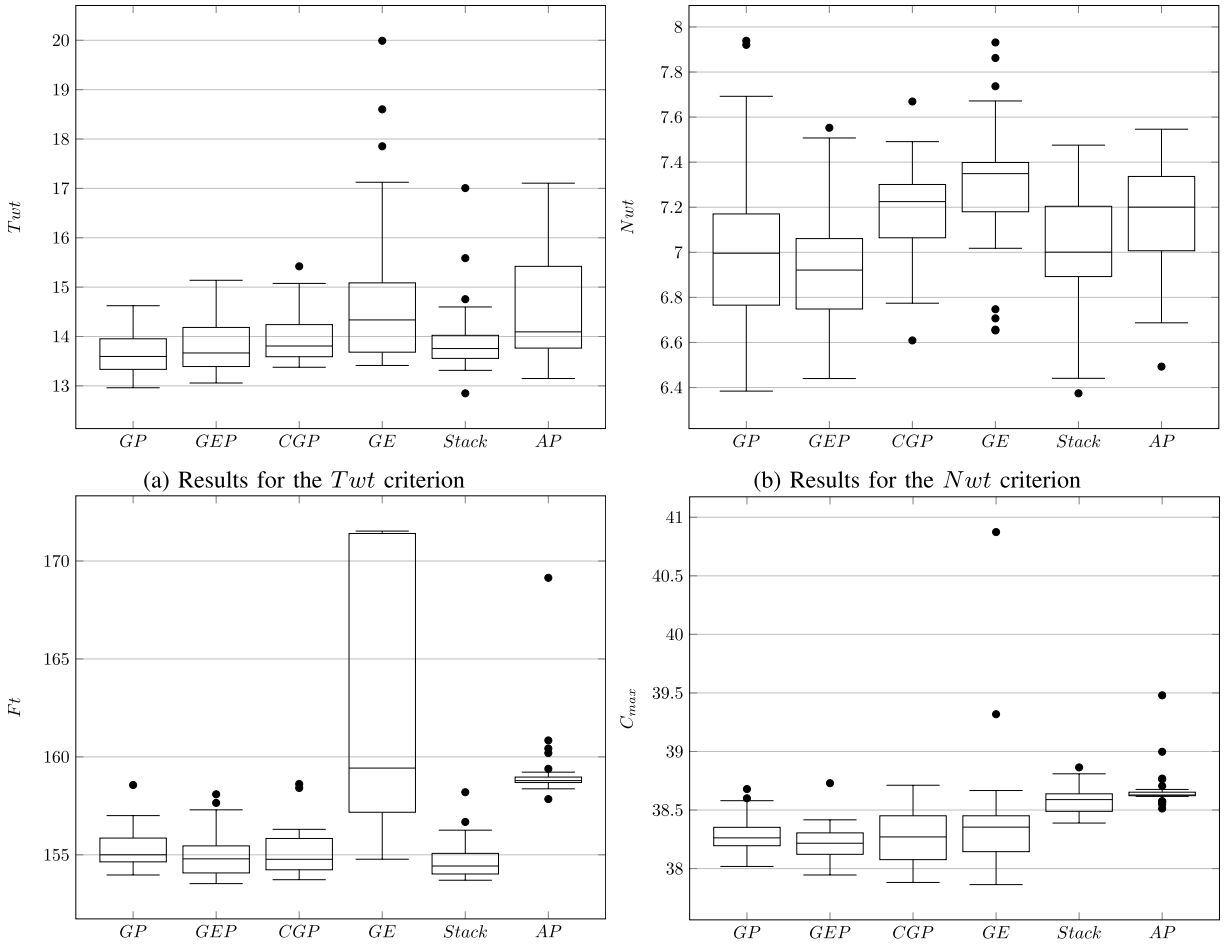
	GP	GEP	CGP	GE	Stack	AP
Population size	1000	1000	500	500	2000	200
Stopping criterion	80 000 function evaluations					
Selection	Steady state tournament selection of size three					Differential Evolution
Initialisation	Ramped half-and-half	Random	Random	Random	Random	Random
Mutation probability	0.3	0.3	0.3	0.7	0.5	0.3
Expression size	Tree depth 5	3 genes and head size 6	100 columns and one row, with levels back equal to 100	150	60	50
Crossover operators	Subtree, uniform, context-preserving, size-fair	One point crossover	Gate one point, gate random, gate uniform, one point, random, uniform	Uniform	Two point crossover	Simple arithmetic, single arithmetic, average, BGA, BLX, BLX-Alpha, BLX-Alpha-Beta, discrete, flat, heuristic, local, one point, random, SBX
Mutation operators	Subtree, Gauss, hoist, node complement, nodereplacement, permutation, shrink	Replacement mutation	One point, one gate	Simple	Node replacement	Simple
Transposition operators	-	IS, RIS, gene transposition	-	-	-	-

TABLE 6. Results of the various GP methods on four scheduling criteria.

	Twt			Nwt			Ft			C_{max}		
	min	med	max	min	med	max	min	med	max	min	med	max
GP	12.96	**13.60**	**14.62**	6.384	7.005	7.939	154.0	155.0	158.6	38.02	38.26	**38.68**
GEP	13.06	13.68	15.14	6.440	**6.925**	7.553	**153.5**	154.8	**158.1**	37.95	**38.22**	38.73
CGP	13.38	13.81	15.42	6.609	7.225	7.669	153.7	154.9	158.6	37.88	38.27	38.71
GE	13.41	14.37	19.99	6.653	7.349	7.931	154.8	159.4	171.5	**37.86**	38.36	40.87
Stack	**12.85**	13.78	17.01	**6.374**	7.003	**7.476**	153.7	**154.5**	158.2	38.39	38.59	38.86
AP	13.15	14.13	17.11	6.493	7.210	7.546	157.8	158.8	169.1	38.51	38.63	39.48

the results. In addition, as the depth increases, the method also produced more widely dispersed results, as can be seen by the many outlier values that were obtained. It is also interesting to note that, for larger depths, GP was able to obtain PFs which performed better than any of the PFs generated when using the tree depth of 5. Thus, by using larger tree depths, GP seems to have a greater possibility of evolving PFs with the absolute best performance. However, these rules were generally quite large, so it would be difficult to interpret them and extract knowledge from them.

Table 9 presents the influence of the number of genes and the head size on the average size of the expressions generated by GEP. In addition to the average expression size of the evolved PFs, the table also contains the maximum



(a) Results for the Twt criterion

(b) Results for the Nwt criterion

(c) Results for the Ft criterion

(d) Results for the C_{max} criterion

FIGURE 1. Box plot representation of results for the different GP representations.

TABLE 7. Average expression sizes of evolved PFs.

	Twt	Nwt	Ft	C_{max}
GP	40.24	38.48	42.40	38.34
GEP	29.66	29.64	28.74	26.64
CGP	18.77	14.63	18.27	18.03
GE	17.40	11.63	15.6	16.13
Stack	25.90	23.03	26.33	22.33
AP	48.17	45.13	31.60	38.27

TABLE 8. Influence of the tree depth in GP.

Tree depth	Size		Fitness		
	th. max	avg	min	med	max
3	15	13.78	13.42	14.21	15.09
5	63	40.24	12.96	13.60	14.62
7	255	89.70	13.05	13.98	17.35
9	1023	153.6	12.80	14.24	18.20
11	4095	200.5	12.92	14.34	18.68
13	16383	443.8	12.58	14.40	18.34

expression size that can be generated by the given parameter value combination. As can be seen, the parameters in GEP allow much finer control over the size of the expressions than was the case with GP. GEP typically evolved expressions with an average size that was about 60% to 70% of the maximum expression size that can be evolved for the given parameter values. For three genes with head size six, GEP evolved PFs with the smallest average size of about 30 elements. On the other hand, the largest PFs with an average size of about 60 elements were generated when using five genes of head size 10.

The performance of PFs generated by using different parameter values for GEP is shown in Figure 3, where the labels denote the number of genes in the individual (denoted by "g") and the head size of each gene (denoted by "h"). Unlike when GP was used, this time it can be observed that the results obtained for different sizes were mostly similar. This is to be expected since the difference in sizes between the

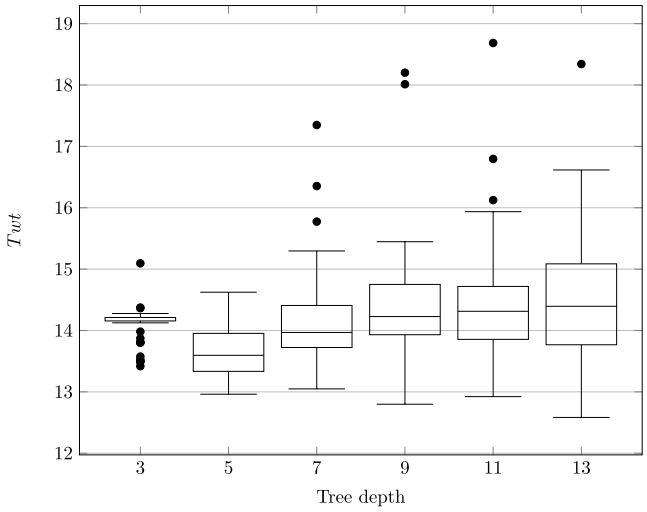


FIGURE 2. Results for different maximum tree depths of GP.

different parameter combinations were not as drastic as when using GP. The best median values were obtained when using individuals consisting of three genes and for smaller head sizes. The best result obtained by GEP was achieved when using the parameter combination that leads to the smallest average size of individuals. It seems that GEP is more suitable for finding good PFs of smaller sizes than GP. It is interesting to observe that in addition to the size of the individual, the structure of the individual also plays an important role. For example, GEP developed PFs with the same average size both when using two genes of head size ten and three genes of head size six. However, PFs which were evolved by GEP with three genes of head size six achieved a better median value. Thus, although both cases result in the same average expression sizes, better performance is achieved when genes with smaller sizes are used.

Table 10 shows the effect of different individual sizes of CGP on the average sizes of generated PFs and their quality. The table shows a rather interesting phenomenon for CGP, namely that although the maximum size of the individual increases, the average size of the generated PFs is only slightly increased. For example, in the case of the smallest tested individual size, CGP evolved expressions that consisted of only 14 nodes on average, while for the largest individual size, CGP generated PFs that consisted of about 24 elements on average. CGP did, on several occasions, evolve DRs of quite large sizes. However, these PFs generally did not perform very well on the test set. Thus, it seems that CGP is more focused on developing PFs with a smaller number of elements. The reason for this could be that the maximum value for the levels-back parameter was used, which means that CGP is able to evolve individuals where a large portion of the nodes are simply skipped and are thus inactive. In this way, CGP can easily evolve PFs of the preferred size.

Figure 4 shows the boxplot representing the results for different individual sizes when using CGP. The figure shows

TABLE 9. Influence of the number of genes and head size in GEP.

Number of genes	Head size	Size th. max	Size avg	Fitness min	Fitness med	Fitness max
2	10	43	30	12.93	13.95	14.76
3	10	65	40.6	13.00	13.75	17.50
4	10	87	53.46	13.15	13.84	15.75
5	10	109	63.22	13.05	13.99	16.52
3	6	41	29.66	13.06	13.68	15.14
3	8	53	35.1	13.27	13.84	14.84
3	12	77	44.7	13.10	13.95	15.92
3	14	89	49.36	12.72	13.90	15.99

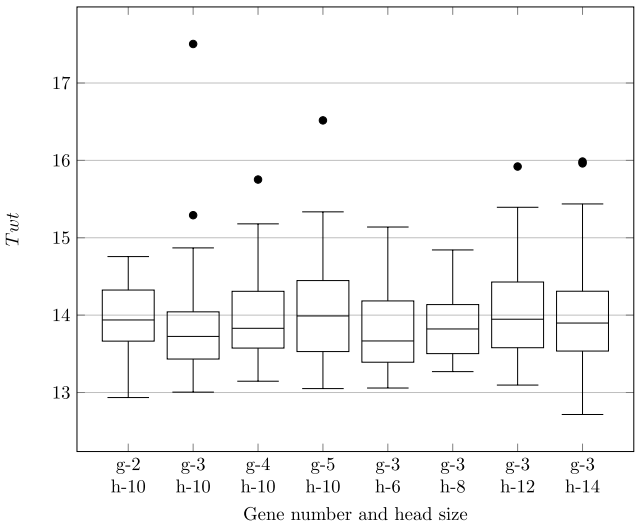


FIGURE 3. Box plot representation of results for different GEP parameter values.

that the size of the individuals has a significant impact on the quality of the results. The best results were obtained when using individuals of size 301, which means that the individual contains 100 nodes, not all of which need to be active. As the size of the individual decreases and increases, the fitness of the individuals deteriorates. For smaller individuals, this is probably due to the fact that the individuals of smaller sizes might not be expressive enough, while the individuals that are too large are probably not quite suitable because mutation is mainly performed on inactive parts, which then does not affect the quality of the individual. This seems to cause even more problems for CGP than when using too small individual sizes, as the algorithm usually performs worse to some degree with larger individual sizes than with smaller individual sizes.

Table 11 presents the results for the different individual sizes of GE. The results from the table show that GE is even more biased towards evolving smaller expressions. This can be seen from the fact that the evolved expressions are relatively small compared to the previous three methods, regardless of the maximum expression size used to evolve the PFs. Therefore, even with larger individual sizes, the average size of PFs will not be more than 18 elements. However, larger PFs are sometimes generated for the larger individual sizes,

TABLE 10. Influence of different maximum individual sizes in CGP.

Individual size	Size		Fitness		
	th. max	avg	min	med	max
91	$2^{31}-1$	14.27	13.36	14.11	15.81
151	$2^{51}-1$	15.13	13.41	14.21	16.07
211	$2^{71}-1$	18.27	12.88	14.19	16.93
301	$2^{101}-1$	18.77	13.38	13.81	15.42
451	$2^{151}-1$	19.53	12.92	14.34	17.04
601	$2^{201}-1$	18.23	13.44	14.31	16.74
901	$2^{301}-1$	20.3	13.50	14.18	15.63
1501	$2^{501}-1$	24.43	12.74	14.34	16.52

but similar to CGP, these individuals did not really perform well and were easily outperformed by smaller PFs.

The performance of PFs generated by different individual sizes using GE is shown in Figure 5. GE did not achieve results that could compete with those of other methods, as can be seen from the fact that the method achieved a rather large median value for all individual sizes. In addition, GE also obtained rather inconsistent results, which can be seen not only in the number of outliers that the method obtained, but also in the fact that a large number of the developed PFs obtained poor results. The use of too small individual sizes leads to the worst results for GE. For the individual size of 150 elements, GE obtained the best median results. As the size increases, the performance of the algorithm deteriorates once again.

The results for different sizes of the Stack representation can be found in Table 12. In this case, the theoretical maximum size of an individual is the same as the maximum size of the individual. However, it is interesting to see that the average size of the individuals for the smallest sizes tested, 30 and 40, is about half of the theoretical maximum. This is to be expected since most evolved expressions will be invalid and therefore major parts of the expression will be discarded when the individual is converted to a PF. The results show that this happens even more for larger individuals. For example, at the maximum size of 300, the average size of individuals is only about 46.

The performances of PFs generated by different individual sizes of the Stack genotype are shown in Figure 6. The boxplot shows that the results are somewhat more dispersed for individual sizes greater than 60. The best results are obtained when the maximum individual size is set to 60. Very similar results are obtained for size 50, while the worst results are obtained for the smallest and largest sizes tested.

The results for different sizes of AP individuals are shown in Table 13. In this case, the theoretical maximum size is also equal to the set maximum individual size. Since every individual created in AP has the maximum size, most evolved individuals will end up with the same size. This explains why the average size of individuals for most sizes tested is almost equal to the theoretical maximum size. The biggest difference

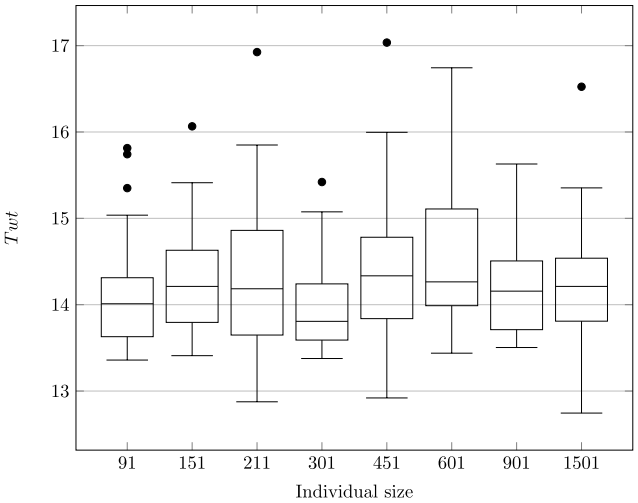


FIGURE 4. Results for different CGP maximum individual sizes.

TABLE 11. Results for the various maximum individual sizes in GE.

Individual size	Size	Fitness		
	avg	min	med	max
30	7	14.65	19.78	27.36
50	10.5	13.10	15.38	22.69
70	11.37	13.58	15.06	20.16
100	13.83	13.24	15.56	27.36
150	17.4	13.41	14.37	19.99
200	15.13	12.80	14.85	25.14
500	14.63	13.40	14.85	27.36
1000	15.17	13.36	14.95	27.36

is seen for individual size 100, where the average individual size is around 90. Therefore, with this method it is quite easy to control the size of the developed individuals.

The performances of PFs generated with different individual sizes are shown in Figure 7. In this case, the worst results are obtained for the smallest individual size, 10, while the best results are obtained for size 50. However, the results for all tested except the smallest one are quite similar and are similarly dispersed. This shows that for the AP method, increasing the individual size does not have a large effect on the quality of the individuals developed.

B. EXAMPLES OF GENERATED DRs

This section briefly reviews the best PFs obtained with each of the six methods tested. It should be mentioned that the rules presented in this section do not represent the very best rules obtained by each of the methods, since the best obtained were quite often very large, but rather the best PFs obtained for the parameter combinations given in Table 5. However, since all of the methods obtained the best median values for these parameters, the PFs presented should still give a good idea of the quality of PFs that can be generated using each of the methods.

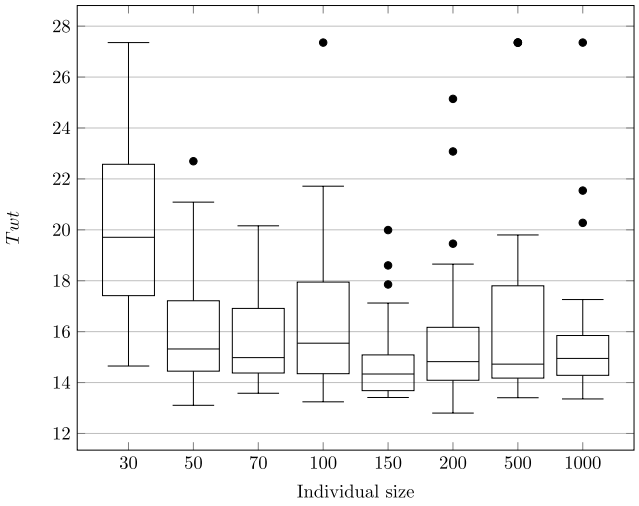


FIGURE 5. Box plot representation of results for different GE maximum individual sizes.

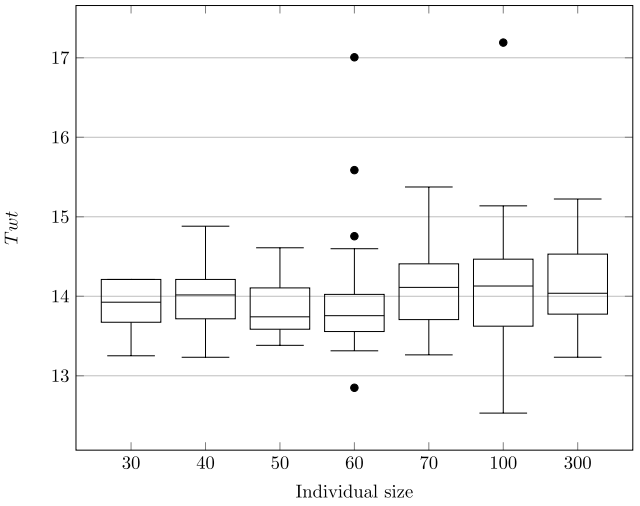


FIGURE 6. Results for different maximum tree depths of Stack genotype.

TABLE 12. Results for the various maximum individual sizes in Stack genotype.

Individual size	Size		Fitness		
	th. max	avg	min	med	max
30	30	16.60	13.25	14.02	14.21
40	40	20.23	13.23	14.02	14.88
50	50	21.77	13.38	13.78	14.61
60	60	25.90	12.85	13.78	17.01
70	70	25.83	13.26	14.13	15.37
100	100	30.73	12.53	14.13	17.19
300	300	46.17	13.23	14.04	15.22

TABLE 13. Results for the various maximum individual sizes in AP.

Individual size	Size		Fitness		
	th.max	avg	min	med	max
10	10	10	13.40	15.45	18.68
30	30	28.17	13.55	14.28	19.70
50	50	49.77	13.15	14.13	17.11
70	70	70	13.28	14.57	16.42
100	100	89.53	13.40	14.38	18.01

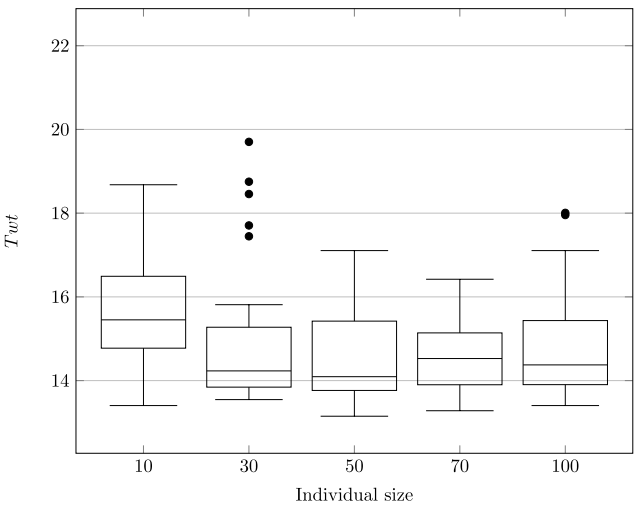


FIGURE 7. Results for different maximum tree depths of AP genotype.

Table 14 shows the PFs evolved by the six methods tested. The PF evolved by Stack is the best performing PF in the table. The size of this PF is 26, which is half that of the next best PF in the table, making it easier to interpret. The PF evolved by GP achieved the second best result, but was also the largest PF among those in the table. By observing the PF generated by GP, it can be seen that it contains several elements that do not have an effect on the value of the priority. For example, it can be seen that the PF applies the *pos* function to several terminal nodes that cannot be negative in themselves. Therefore, removing this function would not change the priority values calculated by the PF, however, the PF would then be somewhat simpler. In addition, it is also common for the expression to contain subexpressions which in most cases do not have a large influence on the priority. An example of such a subexpression in this PF would be $pmin - w$, where w is usually much smaller than the value of the terminal *pmin*. Thus, even if this expression were replaced only by *pmin*, it would probably not have a large influence on the fitness of the PF. Such situations are precisely one of the problems with the tree representation, as it tends to get larger and larger without leading to any significant improvement in the fitness of the PF.

The PF generated by GEP yields a slightly worse result than that of GP, but is almost two times smaller. The PF shows that, with the exception of one unnecessary *pos* function, it does not actually contain any elements that can be immediately classified as redundant. Thus, it appears that GEP is able to control the size of its expressions to a much greater extent,

TABLE 14. PFs generated by the various methods.

Method	Fitness	Size	PF
GP	12.96	52	$\frac{\frac{pmin-w}{w}}{age} - pos(pavg) * (pt + MR) + pos(SL) * \frac{dd}{MR} + \frac{pmin*SL}{pavg-SL} + (\frac{dd}{pt} + pavg + dd) * (pmin - w) + pos(SL) * (SL + pavg) + \frac{dd}{w}$
GEP	13.06	27	$pos\left(pos\left(\frac{pmin}{w}\right) - MR + pavg\right) - pt + \frac{pmin+\frac{SL}{\frac{MR}{SL}}}{MR} + pmin + SL + SL$
CGP	13.38	29	$\left(\frac{pmin}{w} + PAT - MR + SL - pt\right) * pavg + pos(PAT - MR + SL - pt) * pos(MR) + \frac{pmin}{w}$
GE	13.41	20	$dd - pt + MR - pos(w) - pmin + pmin + pos(pos(pos(SL))) - pos(w)$
Stack	12.85	26	$\frac{MR + dd + dd + pt + dd + MR}{PAT} + \frac{pos(pmin)}{w} + dd - (pt + MR) + SL$
AP	13.15	50	$\frac{pos((pos(\frac{\frac{\frac{\frac{PAT}{w}}{dd * dd * dd}}{dd * dd * dd}}{dd * dd * dd * dd * dd * dd}) * \frac{w}{\frac{pos(w)}{MR}}) + pmin + SL)}{pt} * pmin}{w}$

and introduces much less noise into the generated expressions compared to GP. The expression generated by CGP is similar in size to that of GEP, but performs worse than the GEP generated PF to some extent. It is apparent from the PF generated by CGP that it generally does not contain redundant subexpressions, since it contains an unnecessary *pos* function in only one case. Therefore, CGP seems to be suitable for generating PFs of smaller sizes. The PF generated by GE achieves the worst result among the six tested methods, but it also evolved the PF with the smallest size. This one consisted of a large number of subexpressions that ultimately had no effect on the priority value. This can be seen in the multiple redundant *pos* functions applied to the *SL* and *w* terminals, but also in the fact that it generated some expressions such as $pmin - pmin$ that ultimately have no effect on the calculated priority. Finally, the PF generated by AP is the second largest PF in the table. However, its performance is not as good as that of the largest PF. This is because large parts of the evolved expression are redundant. For example, $\frac{dd * dd * dd}{dd * dd * dd}$ could be reduced to only 1 which would reduce the size of the expression by 10 elements. Thus, of all six methods, GE and AP seem to have the most problems with such redundant subexpressions.

VI. CONCLUSION

The objective of this paper was to compare six evolutionary computation methods that can be used to generate new DRs for the unrelated machines scheduling problem. Each of the tested methods uses a different representation for the expressions that serve as PFs and offers different advantages. The tested methods are used for the generation of new DRs for optimising different scheduling criteria. In addition, for each of the tested methods, we analysed how different maximum individual sizes affect the performance of the methods as well as the average size of the PFs they generate.

The results presented in this paper indicate that none of the methods performed the best on all of the tested criteria. With the exception of GE and AP, which performed quite poorly for most criteria, the remaining four methods performed mostly similarly, with their performance largely dependent on the criterion that was optimised. GP and Stack have proven to be the most appropriate when optimising criteria which require more complex PFs, while GEP and CGP were more appropriate for generating DRs for criteria where simpler PFs were preferred. Nevertheless, for most of the tested criteria, the four methods mostly achieve similar results, so there should be no significant difference in the results regardless of which of the four methods is used. Regarding the average size of the generated PFs, CGP and GE generated the expressions with the smallest average size among all tested methods. All methods were found to have redundant parts in the generated expressions. However, the PFs generated by CGP and Stack contained the least amount of redundant subexpressions, suggesting that these methods may be best suited to deal with bloat and redundant subexpressions in PFs.

In future work it is planned to focus more on generating simpler and more interpretable PFs. First, it is planned to simply analyse the generated DRs to better understand which parts of the PFs are redundant and which parts are the most informative. Based on this information, the evolutionary algorithms will be extended with different methods that try to detect redundant parts of PFs during the evolution process and automatically remove them from the expression. This should lead to the generation of simpler and more interpretable PFs. Another research direction would be to use interval arithmetic

to limit the search to those DRs that are valid over the entire domain. In addition, further research will focus on testing different bloat control methods to further reduce the sizes of the generated PFs.

REFERENCES

[1] M. L. Pinedo, *Scheduling: Theory, Algorithms, and Systems*, vol. 9781461423614, 4th ed. Boston, MA, USA: Springer, 2012.

[2] E. Hart, P. Ross, and D. Corne, "Evolutionary scheduling: A review," *Genetic Program. Evolvable Mach.*, vol. 6, no. 2, pp. 191–220, Jun. 2005.

[3] M. Maheswaran, S. Ali, H. J. Siegel, D. Hensgen, and R. F. Freund, "Dynamic mapping of a class of independent tasks onto heterogeneous computing systems," *J. Parallel Distrib. Comput.*, vol. 59, no. 2, pp. 107–131, Nov. 1999.

[4] T. D. Braun, H. J. Siegel, N. Beck, L. L. Bölöni, M. Maheswaran, A. I. Reuther, J. P. Robertson, M. D. Theys, B. Yao, D. Hensgen, and R. F. Freund, "A comparison of eleven static heuristics for mapping a class of independent tasks onto heterogeneous distributed computing systems," *J. Parallel Distrib. Comput.*, vol. 61, no. 6, pp. 810–837, Jun. 2001.

[5] J. Branke, S. Nguyen, C. W. Pickardt, and M. Zhang, "Automated design of production scheduling heuristics: A review," *IEEE Trans. Evol. Comput.*, vol. 20, no. 1, pp. 110–124, Feb. 2016.

[6] S. Nguyen, Y. Mei, and M. Zhang, "Genetic programming for production scheduling: A survey with a unified framework," *Complex Intell. Syst.*, vol. 3, no. 1, pp. 41–66, Mar. 2017.

[7] C. Dimopoulos and A. M. S. Zalzala, "A genetic programming heuristic for the one-machine total tardiness problem," in *Proc. Congr. Evol. Comput.*, Jul. 1999, pp. 2207–2214.

[8] C. Dimopoulos and A. M. S. Zalzala, "Investigating the use of genetic programming for a classic one-machine scheduling problem," *Adv. Eng. Softw.*, vol. 32, no. 6, pp. 489–498, 2001.

[9] K. Miyashita, "Job-shop scheduling with genetic programming," in *Proc. 2nD Annu. Conf. Genetic Evol. Comput.* San Francisco, CA, USA: Morgan Kaufmann Publishers, 2000, pp. 505–512.

[10] N. B. Ho and J. C. Tay, "Evolving dispatching rules for solving the flexible job-shop problem," in *Proc. IEEE Congr. Evol. Comput.*, vol. 3, Sep. 2005, pp. 2848–2855.

[11] D. Jakobović, L. Jelenković, and L. Budin, "Genetic programming heuristics for multiple machine scheduling," in *Genetic Programming*. Berlin, Germany: Springer, 2007, pp. 321–330.

[12] D. Jakobović and L. Budin, "Dynamic scheduling with genetic programming," in *Genetic Programming*, P. Collet, M. Tomassini, M. Ebner, S. Gustafson, and A. Ekárt, Eds. Berlin, Germany: Springer, 2006, pp. 73–84.

[13] W.-J. Yin, M. Liu, and C. Wu, "Learning single-machine scheduling heuristics subject to machine breakdowns with genetic programming," in *Proc. Congr. Evol. Comput. (CEC)*, 2003, pp. 1050–1055.

[14] C. D. Geiger and R. Uzsoy, "Learning effective dispatching rules for batch processor scheduling," *Int. J. Prod. Res.*, vol. 46, no. 6, pp. 1431–1454, Mar. 2008.

[15] D. Jakobović and K. Marasović, "Evolving priority scheduling heuristics with genetic programming," *Appl. Soft Comput.*, vol. 12, no. 9, pp. 2781–2789, Sep. 2012.

[16] K. Jaklinović, M. Đurasević, and D. Jakobović, "Designing dispatching rules with genetic programming for the unrelated machines environment with constraints," *Expert Syst. Appl.*, vol. 172, Jun. 2021, Art. no. 114548.

[17] S. Nguyen, M. Zhang, M. Johnston, and K. C. Tan, "Dynamic multi-objective job shop scheduling: A genetic programming approach," in *Automated Scheduling and Planning: From Theory to Practice*, A. S. Uyar, E. Ozcan, and N. Urquhart, Eds. Berlin, Germany: Springer, 2013, pp. 251–282.

[18] S. Nguyen, M. Zhang, and K. C. Tan, "Enhancing genetic programming based hyper-heuristics for dynamic multi-objective job shop scheduling problems," in *Proc. IEEE Congr. Evol. Comput. (CEC)*, May 2015, pp. 2781–2788.

[19] D. Karunakaran, G. Chen, and M. Zhang, "Parallel multi-objective job shop scheduling using genetic programming," in *Proc. Artif. Life Comput. Intell., 2nd Australas. Conf. (ACALCI)*, T. Ray, R. Sarker, and X. Li, Eds. Canberra, ACT, Australia: Springer, 2016, pp. 234–245.

[20] M. Đurasević and D. Jakobović, "Evolving dispatching rules for optimising many-objective criteria in the unrelated machines environment," *Genetic Program. Evolvable Mach.*, vol. 19, nos. 1–2, pp. 9–51, Sep. 2017.

[21] S. Nguyen, M. Zhang, M. Johnston, and K. C. Tan, "Evolving reusable operation-based due-date assignment models for job shop scheduling with genetic programming," in *Genetic Programming*, A. Moraglio, S. Silva, K. Krawiec, P. Machado, and C. Cotta, Eds. Berlin, Germany: Springer, 2012, pp. 121–133.

[22] S. Nguyen, M. Zhang, M. Johnston, and K. C. Tan, "Genetic programming for evolving due-date assignment models in job shop environments," *Evol. Comput.*, vol. 22, no. 1, pp. 105–138, Mar. 2014.

[23] S. Nguyen, M. Zhang, M. Johnston, and K. C. Tan, "A coevolution genetic programming method to evolve scheduling policies for dynamic multi-objective job shop scheduling problems," in *Proc. IEEE Congr. Evol. Comput.*, Jun. 2012, pp. 1–8.

[24] S. Nguyen, M. Zhang, M. Johnston, and K. C. Tan, "Automatic design of scheduling policies for dynamic multi-objective job shop scheduling via cooperative coevolution genetic programming," *IEEE Trans. Evol. Comput.*, vol. 18, no. 2, pp. 193–208, Apr. 2014.

[25] J. Park, S. Nguyen, M. Zhang, and M. Johnston, "Genetic programming for order acceptance and scheduling," in *Proc. IEEE Congr. Evol. Comput.*, Jun. 2013, pp. 1005–1012.

[26] S. Nguyen, M. Zhang, M. Johnston, and K. C. Tan, "Learning reusable initial solutions for multi-objective order acceptance and scheduling problems with genetic programming," in *Genetic Programming*, K. Krawiec, A. Moraglio, T. Hu, A. Ş. Etaner-Uyar, and B. Hu, Eds. Berlin, Germany: Springer, 2013, pp. 157–168.

[27] S. Nguyen, M. Zhang, and M. Johnston, "A sequential genetic programming method to learn forward construction heuristics for order acceptance and scheduling," in *Proc. IEEE Congr. Evol. Comput. (CEC)*, Jul. 2014, pp. 1824–1831.

[28] S. Nguyen, M. Zhang, and K. C. Tan, "A dispatching rule based genetic algorithm for order acceptance and scheduling," in *Proc. Annu. Conf. Genetic Evol. Comput.*, New York, NY, USA, Jul. 2015, pp. 433–440.

[29] S. Nguyen, "A learning and optimizing system for order acceptance and scheduling," *Int. J. Adv. Manuf. Technol.*, vol. 86, nos. 5–8, pp. 2021–2036, Sep. 2016.

[30] S. Chand, Q. Huynh, H. Singh, T. Ray, and M. Wagner, "On the use of genetic programming to evolve priority rules for resource constrained project scheduling problems," *Inf. Sci.*, vol. 432, pp. 146–163, Mar. 2018.

[31] M. Dumić, D. Šišejković, R. Čorić, and D. Jakobović, "Evolving priority rules for resource constrained project scheduling problem with genetic programming," *Future Gener. Comput. Syst.*, vol. 86, pp. 211–221, Sep. 2018.

[32] F. J. Gil-Gala, C. Mencía, M. R. Sierra, and R. Varela, "Evolving priority rules for on-line scheduling of jobs on a single machine with variable capacity over time," *Appl. Soft Comput.*, vol. 85, Dec. 2019, Art. no. 105782.

[33] F. J. Gil-Gala, M. R. Sierra, C. Mencía, and R. Varela, "Genetic programming with local search to evolve priority rules for scheduling jobs on a machine with time-varying capacity," *Swarm Evol. Comput.*, vol. 66, Oct. 2021, Art. no. 100944.

[34] J. Park, S. Nguyen, M. Zhang, and M. Johnston, "Evolving ensembles of dispatching rules using genetic programming for job shop scheduling," in *Genetic Programming*, vol. 2015, P. Machado, M. I. Heywood, J. McDermott, M. Castelli, P. García-Sánchez, P. Burelli, S. Risi, and K. Sim, Eds. Cham, Switzerland: Springer, 2015, pp. 92–104.

[35] E. Hart and K. Sim, "A hyper-heuristic ensemble method for static job-shop scheduling," *Evol. Comput.*, vol. 24, no. 4, pp. 609–635, Dec. 2016.

[36] M. Đurasević and D. Jakobović, "Comparison of ensemble learning methods for creating ensembles of dispatching rules for the unrelated machines environment," *Genetic Program. Evolvable Mach.*, vol. 19, nos. 1–2, pp. 53–92, Apr. 2017.

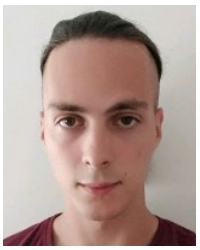
[37] J. Park, Y. Mei, S. Nguyen, G. Chen, and M. Zhang, "An investigation of ensemble combination schemes for genetic programming based hyper-heuristic approaches to dynamic job shop scheduling," *Appl. Soft Comput.*, vol. 63, pp. 72–86, Feb. 2018.

[38] M. Đurasević and D. Jakobović, "Creating dispatching rules by simple ensemble combination," *J. Heuristics*, vol. 25, no. 6, pp. 959–1013, May 2019.

[39] D. Karunakaran, Y. Mei, G. Chen, and M. Zhang, "Dynamic job shop scheduling under uncertainty using genetic programming," in *Intelligent and Evolutionary Systems*, G. Leu, H. K. Singh, and S. Elsayed, Eds. Cham, Switzerland: Springer, 2017, pp. 195–210.

[40] D. Karunakaran, Y. Mei, G. Chen, and M. Zhang, "Evolving dispatching rules for dynamic job shop scheduling with uncertain processing times," in *Proc. IEEE Congr. Evol. Comput. (CEC)*, Jun. 2017, pp. 364–371.
[41] D. Karunakaran, Y. Mei, G. Chen, and M. Zhang, "Toward evolving dispatching rules for dynamic job shop scheduling under uncertainty," in *Proc. Genetic Evol. Comput. Conf.*, New York, NY, USA, Jul. 2017, pp. 282–289.
[42] S. Nguyen, M. Zhang, M. Johnston, and K. C. Tan, "Selection schemes in surrogate-assisted genetic programming for job shop scheduling," in *Simulated Evolution and Learning*, G. Dick, W. N. Browne, P. Whigham, M. Zhang, L. T. Bui, H. Ishibuchi, Y. Jin, X. Li, Y. Shi, P. Singh, K. C. Tan, and K. Tang, Eds. Cham, Switzerland: Springer, 2014, pp. 656–667.
[43] S. Nguyen, M. Zhang, and K. C. Tan, "Surrogate-assisted genetic programming with simplified models for automated design of dispatching rules," *IEEE Trans. Cybern.*, vol. 47, no. 9, pp. 1–15, May 2016.
[44] S. Nguyen, M. Zhang, D. Alahakoon, and K. C. Tan, "Visualizing the evolution of computer programs for genetic programming [research frontier]," *IEEE Comput. Intell. Mag.*, vol. 13, no. 4, pp. 77–94, Nov. 2018.
[45] S. Nguyen, M. Zhang, D. Alahakoon, and K. C. Tan, "People-centric evolutionary system for dynamic production scheduling," *IEEE Trans. Cybern.*, vol. 51, no. 3, pp. 1–14, Mar. 2019.
[46] F. Zhang, Y. Mei, S. Nguyen, and M. Zhang, "Guided subtree selection for genetic operators in genetic programming for dynamic flexible job shop scheduling," in *Genetic Programming*, T. Hu, N. Lourenço, E. Medvet, and F. Divina, Eds. Cham, Switzerland: Springer, 2020, pp. 262–278.
[47] D. Karunakaran, Y. Mei, G. Chen, and M. Zhang, "Active sampling for dynamic job shop scheduling using genetic programming," in *Proc. IEEE Congr. Evol. Comput. (CEC)*, Jun. 2019, pp. 434–441.
[48] M. Đurasević and D. Jakobović, "Comparison of schedule generation schemes for designing dispatching rules with genetic programming in the unrelated machines environment," *Appl. Soft Comput.*, vol. 96, Nov. 2020, Art. no. 106637.
[49] M. Đurasević and D. Jakobović, "Automatic design of dispatching rules for static scheduling conditions," *Neural Comput. Appl.*, vol. 33, no. 10, pp. 5043–5068, Aug. 2020.
[50] F. Zhang, Y. Mei, S. Nguyen, K. C. Tan, and M. Zhang, "Multitask genetic programming-based generative hyperheuristics: A case study in dynamic scheduling," *IEEE Trans. Cybern.*, early access, Mar. 22, 2021, doi: 10.1109/TCYB.2021.3065340.
[51] F. Zhang, Y. Mei, S. Nguyen, M. Zhang, and K. C. Tan, "Surrogate-assisted evolutionary multitask genetic programming for dynamic flexible job shop scheduling," *IEEE Trans. Evol. Comput.*, vol. 25, no. 4, pp. 651–665, Aug. 2021.
[52] I. Vlašić, M. Đurasević, and D. Jakobović, "Improving genetic algorithm performance by population initialisation with dispatching rules," *Comput. Ind. Eng.*, vol. 137, Nov. 2019, Art. no. 106030.
[53] L. Nie, X. Shao, L. Gao, and W. Li, "Evolving scheduling rules with gene expression programming for dynamic single-machine scheduling problems," *Int. J. Adv. Manuf. Technol.*, vol. 50, nos. 5–8, pp. 729–747, Sep. 2010.
[54] L. Nie, L. Gao, P. Li, and L. Zhang, "Application of gene expression programming on dynamic job shop scheduling problem," in *Proc. 15th Int. Conf. Comput. Supported Cooperat. Work Design (CSCWD)*, Jun. 2011, pp. 291–295.
[55] L. Nie, Y. Bai, X. Wang, and K. Liu, "Discover scheduling strategies with gene expression programming for dynamic flexible job shop scheduling problem," in *Advances in Swarm Intelligence*, Y. Tan, Y. Shi, and Z. Ji, Eds. Berlin, Germany: Springer, 2012, pp. 383–390.
[56] M. Durasević, D. Jakobović, and K. Knežević, "Adaptive scheduling on unrelated machines with genetic programming," *Appl. Soft Comput.*, vol. 48, pp. 419–430, Nov. 2016.
[57] S. Nguyen, M. Zhang, M. Johnston, and K. C. Tan, "A computational study of representations in genetic programming to evolve dispatching rules for the job shop scheduling problem," *IEEE Trans. Evol. Comput.*, vol. 17, no. 5, pp. 621–639, Oct. 2013.
[58] J. Branke, T. Hildebrandt, and B. Scholz-Reiter, "Hyper-heuristic evolution of dispatching rules: A comparison of rule representations," *Evol. Comput.*, vol. 23, no. 2, pp. 249–277, Jun. 2015.
[59] S. Nguyen, M. Zhang, M. Johnston, and K. C. Tan, "Learning iterative dispatching rules for job shop scheduling with genetic programming," *Int. J. Adv. Manuf. Technol.*, vol. 67, nos. 1–4, pp. 85–100, Jul. 2013.
[60] M. Keijzer and V. Babovic, "Dimensionally aware genetic programming," *Proc. Genetic Evol. Comput. Conf.*, vol. 2, pp. 1069–1076, 1999.
[61] R. Poli, W. B. Langdon, and N. F. McPhee, *A Field Guide to Genetic Programming*, J. R. Koza, Ed. Lulu.com, 2008. [Online]. Available: http://www.gp-field-guide.org.U.K
[62] C. Ferreira, "Gene Expression programming: A new adaptive algorithm for solving problems," *Complex Syst.*, vol. 13, no. 2, pp. 87–129, Mar. 2001.
[63] J. F. Miller and P. Thomson, *Cartesian Genetic Programming* (Lecture Notes in Computer Science), vol. 1802. Heidelberg, Germany: Springer, 2000, pp. 121–132.
[64] C. Ryan, J. J. Collins, and M. O'Neill, "Grammatical evolution: Evolving programs for an arbitrary language," in *Proc. 1st Eur. Workshop Genetic Program.*, vol. 1391, W. Banzhaf, R. Poli, M. Schoenauer, and T. C. Fogarty, Eds. Paris, France: Springer-Verlag, Apr. 1998, pp. 83–96.
[65] T. Perkis, "Stack-based genetic programming," in *Proc. 1st IEEE Conf. Evol. Comput. IEEE World Congr. Comput. Intell.*, Jun. 1994, pp. 148–153.
[66] I. Zelinka, Z. K. Oplatkova, and L. Nolle, "Analytic programming symbolic regression by means of arbitrary evolutionary algorithm," *J. Simul.*, vol. 6, pp. 1473–8031, Aug. 2005.
[67] J. F. Miller, *Cartesian Genetic Programming*. Berlin, Germany: Springer, 2011, pp. 17–34.

LUCIJA PLANINIĆ (Member, IEEE) received the B.Sc. and M.Sc. degrees in computer science from the Faculty of Electrical Engineering and Computing, University of Zagreb, in 2018 and 2020, respectively, where she is currently pursuing the Ph.D. degree with the Faculty of Electrical Engineering and Computing. She is also a Research Assistant with the Faculty of Electrical Engineering and Computing, University of Zagreb.



HRVOJE BACKOVIĆ received the B.Sc. and M.Sc. degrees in computing from the Faculty of Electrical Engineering and Computing, University of Zagreb, in 2015 and 2017, respectively. He is currently working as a Software Developer at Visage Technologies, Zagreb, Croatia.

MARKO ĐURASEVIĆ (Member, IEEE) received the B.Sc. and M.Sc. degrees in computing from the Faculty of Electrical Engineering and Computing, University of Zagreb, in 2012 and 2014, respectively, and the Ph.D. degree, in February 2018, on the subject of generating dispatching rules for solving scheduling problems in the unrelated machines environment. He is currently an Assistant Professor with the Faculty of Electrical Engineering and Computing, University of Zagreb.

DOMAGOJ JAKOBOVIĆ (Senior Member, IEEE) received the B.Sc. degree, in December 1996, the M.Sc. degree in electrical engineering, in December 2001, and the Ph.D. degree, in December 2005, on the subject of generating scheduling heuristics with genetic programming. Since April 1997, he has been a member of the Research and Teaching Staff with the Department of Electronics, Microelectronics, Computer and Intelligent Systems, Faculty of Electrical Engineering and Computing, University of Zagreb, where he is currently a Full Professor with the Faculty of Electrical Engineering and Computing.