Date of publication xxxx 00, 0000, date of current version xxxx 00, 0000.

Digital Object Identifier 10.1109/ACCESS.2017.DOI

A comparative study of dispatching rule representations in evolutionary algorithms

LUCIJA PLANINIĆ[1], (Member, IEEE), HRVOJE BACKOVIĆ [2], MARKO ĐURASEVIĆ [3], (MEMBER, IEEE), AND DOMAGOJ JAKOBOVIĆ [4], (Member, IEEE).

[1]Faculty of Electrical Engineering and Computing, University of Zagreb, Zagreb, Croatia (e-mail: lucija.planinic@fer.hr)
[2]Visage Technologies, Zagreb, Croatia (e-mail: hrvoje.backovic@fer.hr)
[3]Faculty of Electrical Engineering and Computing, University of Zagreb, Zagreb, Croatia (e-mail: marko.durasevic@fer.hr)
[3]Faculty of Electrical Engineering and Computing, University of Zagreb, Zagreb, Croatia (e-mail: domagoj.jakobovic@fer.hr)

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

This work has been supported in part by the Croatian Science Foundation under the 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 automatised by using various machine learning and evolutionary computation methods. In previous research, genetic programming was the most predominantly used method for the automatic design of new dispatching rules. However, there are many other evolutionary methods which use different representations than genetic programming, that can be used for generating dispatching rules. Some, like gene expression programming, were already successfully applied, while others like Cartesian genetic programming or grammatical evolution, were not used to generate dispatching rules. This paper will test six different methods (genetic programming, gene expression programming, Cartesian genetic programming, grammatical evolution, stack representation and analytic programming) to generate new dispatching rules for the unrelated machines environment, and will analyse how the tested methods perform on various scheduling criteria. The paper also analyses how different individual sizes in the tested methods affect the performance and average size of the generated dispatching rules. The results indicate that, except for the grammatical evolution and analytic programming, all tested methods perform quite similar, with the results depending on selected scheduling criterion. The results also demonstrate that Cartesian genetic programming was the most resistant to the occurrence of bloat, and that it evolved dispatching rules of the smallest average sizes.

INDEX TERMS Unrelated machines environment, Scheduling, Solution representations, Dispatching rules

I. INTRODUCTION

Scheduling can be defined as the process of allocating a set of available jobs to a certain set of limited resources, in a way that some user defined conditions are satisfied and one or more scheduling criteria are optimised [1]. Because 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 start of the system. This means that such methods can be applied only 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, meaning that it is not known in advance when new jobs will arrive into the system. Therefore, scheduling needs to be performed under dynamic conditions, simultaneously with the execution of the system. Since in such situations metaheuristic methods can not be used to solve scheduling problems, many problem specific heuristics, called dispatching rules, were 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, and on which machine. In order to determine which of the available jobs should be scheduled, the DRs use certain job and system properties for assigning a priority to each job, and then select the one with the best priority value. For example, a DR could schedule jobs in the order of their arrival, meaning that jobs which were sooner released into the system will have a higher priority of being scheduled. It is important to outline that when calculating the priorities for jobs, DRs use only the information which is currently available to them, and calculate the priorities only of those jobs which were already released into the system. Because of the previous reason, and also 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. One important disadvantage of DRs is that a single DR does not perform well for all possible situations and scheduling criteria, which would mean that new DRs would need to be designed if no appropriate DR exists for a given criterion, or scheduling condition. Unfortunately, designing new DRs is a long and time consuming process, which usually needs to be performed by a domain expert.

In order to solve the previously outlined problem, over the last twenty years a lot of research concerned with scheduling problems has focused on automatically designing new DRs [5], [6]. Out of the many machine learning and evolutionary computation approaches, genetic programming (GP) is the most commonly applied approach for generating new DRs. Dimopoulos and Zalzala [7], [8] were among the first who used GP to generate new DRs for the single machine environment, while [9] was the first to generate new DRs for the job shop environment. The next several studies mostly focused on applying GP in other machine environments, like in the flexible job shop [10] or the parallel uniform machines environment [11]. Several other studies focused on extending the GP method in various ways, like adapting GP for detecting overloaded machines in the system [12], or generating DRs by GP for problems with additional constraints like 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 optimising multiple criteria simultaneously, where different multi-objective and many-objective algorithms were tested for optimising various combinations of scheduling criteria [17]–[20]. GP was also applied for designing due date assignment rules (DDARs), which approximate the due dates of jobs which arrive into the system. These DDARs were designed either on their own [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, where in addition to scheduling jobs on certain machines it also required to determine which jobs will be accepted for scheduling, was also extensively investigated, and it was shown that GP generated good DRs even for this type of scheduling problem [25]–[29]. Recent papers also shifted the focus on some less investigated scheduling problems like the resource constrained project scheduling problem [30], [31] and the single machine problem with variable capacity [32], [33].

To further enhance the performance of the generated DRs, several studies analysed how different ensemble learning methods could be used to create ensembles of DRs, which can perform better than a single DR [34]–[38]. Other studies focused on scheduling with uncertainties in which some problem parameters were not deterministic [39]–[41], or on applying surrogate assisted GP to reduce the computational cost of evolving new DRs [42]–[44]. A framework for visualisation of the evolutionary process in order to better understand the process of evolving DRs was proposed in [45]. 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. Selecting 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 evolutionary process. In [48] different schedule generation schemes were compared to improve the performance of generated DR, whereas in [49] the authors focused on adapting the generated DRs for static scheduling conditions. A multitask GP model which generates heuristics for a wider range of problems is applied in [50], [51] to generate more general DRs. In [52] automatically designed DRs have been used to generate the initial population of a genetic algorithm, which has lead to significantly improved results when comparing to randomly generated populations or those initialised by manually designed DRs. Beside GP, gene expression programming (GEP) also received a certain amount of attention for the generation of new DRs [53]–[56], and mostly achieved equally good results as GP.

Since GP is predominantly used when generating new DRs, several studies were concerned with the comparison of different GP representations, or with the comparison of GP to other methods for the automatic design of new DRs. One such study was conducted by [57], in which the authors compared three different representations of GP for the creation of 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 a completely new DR by using arithmetic expressions. Finally, the third representation represents a combination of the previous two, since it allows that additional new DRs can be designed which can be selected in place of the manually designed DRs. In [58] the authors compared the DRs generated by GP with those generated by artificial neural networks (ANNs) and a linear representation which is defined as a weighted linear sum of several job and system attributes. The results demonstrate that the linear representation achieved the worst results, while GP and ANNs achieved quite similar results.

However, the main benefit of GP against ANNs is that it evolves DRs which are easier to interpret, since it evolved them in a 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 demonstrated that apart for the IDRs which are used under static scheduling conditions, usually there was no difference among the other three methods. Therefore, it would mostly depend on the users which one of the tested methods they would apply.

From the previously outlined literature overview it is evident that GP is predominantly applied for generating new DRs, with GEP being used only in several occasions. However, no previous study did extensively analyse the effect of different solution representations, which are used by various 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, which were both previously applied for generating new DRs, this study will additionally use Cartesian GP (CGP), grammatical evolution (GE), the stack representation and analytic programming (AP) to generate new DRs for the unrelated machines environment. In addition to testing the performance of the previous four methods on several scheduling criteria, the paper will also investigate the complexity of expressions which are generated by the different methods, to identify which methods generate the least complex expressions. The main contributions of this paper can be summarised with the following three points:

1) The first application of Cartesian GP, grammatical evolution, the 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 rest of the paper is structured as follows. Section II provides a short introduction of scheduling problems, a description of the solution representations used by the tested evolutionary computation methods, and finally describes how GP can be applied for automatic generation of new DRs. In Section III the experimental design of the problem sets is described, and additionally the selected parameter values for the different methods are outlined. The experimental results achieved by the six selected methods on several scheduling criteria are presented in Section IV. Section V provides a further analysis of the obtained results, mostly concerning the average size of the generated DRs. Section VI gives a brief conclusion and outlines possibilities for future work.

II. BACKGROUND AND METHODOLOGY

A. UNRELATED MACHINES ENVIRONMENT

The unrelated machines environment is defined as a machine environment which consists of n jobs and m machines. Each one of the n jobs needs to be scheduled on one of the m available machines. Once a job is scheduled on a certain machine, it needs to be executed until it is finished, meaning that no preemption is allowed. Additionally, each machine can execute only one job at each moment. Thus, if all machines are busy it will be required to wait until at least one 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, meaning that for each job machine pair a different processing time p_{ij} is defined. Since jobs become available during the execution of the system, the time at which jobs become available (r_j) also needs to be defined. Additionally, depending on which scheduling criteria is optimised, each job can have two additional properties defined, the due date (d_j) and the weight w_j. The due date defines the point in time until which the job should finish its execution, or otherwise a certain penalty will be invoked. On the other hand, the weight specifies the importance of jobs, denoting that certain jobs should have a higher priority of being scheduled. Finally, C_j will be used to denote the moment in time when job j finished with its execution in the constructed schedule.

Although various 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):

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

- **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 conducted. In this study, scheduling will be performed under dynamic conditions, which means that no information about the future of the system is known in advance. Therefore, it is unknown when jobs will be released into the system, and until they are released the values of all other job properties will be unknown as well. This means that the schedule cannot be created before the execution of the system, since the required information will be unavailable. Rather, the schedule needs to be constructed in parallel with the execution of the system.

B. SOLUTION REPRESENTATIONS FOR GP

This section will give a short description of the representations that are used by the six evolutionary computation methods which will be compared.

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

GEP [62] uses an alternative representation which does not store the expression in the form of a tree, but rather in a linear form similarly as genetic algorithms do. In this way GEP tries to combine the simplicity of the representation and operators from genetic algorithms, with the possibility to evolve expressions. The individuals in GEP are of constant size, however, the part of the individual which is 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 *tail* of the gene. The head of the gene represents the starting h 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 rest of the nodes belong to the tail of the gene, whose size is calculated as $t = h * (n_{max} - 1) + 1$, where t is the tail size, and n_{max} the maximum number of arguments from all nodes in the function set. The tail of the gene consists only of terminal nodes, which ensures that the gene will consist of enough terminal nodes for it 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 for creating 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, which are 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 act as an input to the current node. If levels-back is set to 1, then only the nodes from the previous column can be used as inputs 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 is required to define which function it represents and which nodes act as its input. Since each node must be able to represent any of the functions, the number of inputs which 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 extra inputs are ignored. The inputs of a node are denoted with integer numbers, which represent the indices of the nodes that act as the input. If there are n_i terminal nodes, then the indices $[0, n_i - 1>$ are used to denote terminal nodes, whereas the indices $[n_i, n_i + n_c * n_r>$ are used to denote outputs of the nodes in the grid. The outputs are encoded as additional numbers in the genotype, that represent the indices of nodes whose output will be used as the program output.

GE [64] also represents the solution as a linear array of numbers. In order 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. In order to do so, 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 by using production rules, until there are no more non-terminal symbols in the expression. Since for each non-terminal symbol several production rules can be defined, 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 there is no need to define new operators 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, terminal set and the maximum individual size. The terminal and function set are the same as in the tree representation. They contain all of the functions, variables and constants which can compose an individual. The functions and terminals that compose an individual are stored in a linear form. To evaluate the individual, we need to generate the mathematical expression that the individual represents. This is done by going through the elements of the individual. Whenever a terminal is encountered, it is pushed on to 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 on to the stack. If the number of elements on the stack is smaller than the number of arguments, the function is simply ignored and the evaluation moves on 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 that is defined by the lower bound and upper bound parameters. A vital component of AP is the general function set (GFS) which is composed of functions and terminals. The GFS is also divided into subsets according to the number of arguments of functions. When decoding an AP individual, the first step is to transform the floating point values to discrete indices which represent indices of functions in the GFS. This is done by transforming the original value to a value within the range from 0 to the number of primitives in the GFS. A mathematical expression is then built by replacing the indices with the functions in GFS at the corresponding index. The described structure of the general function set is used to avoid forming invalid mathematical expressions while replacing indices with elements from the GFS. When the function that is supposed to replace an index has more arguments than there are elements remaining to the end of the individual, a function with less arguments is chosen. This ensures that there are enough elements after the function to use 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. In order to determine which job should be scheduled on which machine, the SGS uses a priority function which 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 wide variety of problems, while the priority function which fits the concrete problem or optimised criterion can be selected and used with the SGS. Because of that the SGS is defined manually, while the PFs will be generated by using one of the previously described GP methods. Algorithm 1 represents the SGS which is used for creating schedules in the unrelated machines environment. The intuition behind this SGS is that every time a job is released or a machine becomes available, the PF is used to determine the priorities of scheduling each of the available jobs on each of the machines, even the ones which are currently executing other jobs. Based on the calculated priorities the most appropriate machine for each job is determined, and then all jobs whose most appropriate machine is available are scheduled in order of their priorities. By calculating priorities even for busy machines, it is possible to insert idle times into the schedule, since this allows for situations where for all jobs the best machines can be busy, and therefore no job would be scheduled on other available machines, but rather the scheduling decision would be postponed to a later moment in time.

Algorithm 1 Schedule generation scheme used by DRs generated by GP

```
1:  while unscheduled jobs are available do
2:      Wait until at least one job and one machine are available
3:      for all available jobs and all machines do
4:          Obtain the priority $\pi_{ij}$ of scheduling job $j$ on machine $i$
5:      end for
6:      for all available jobs do
7:          Determine the best machine (the one for which the best value of priority $\pi_{ij}$ is achieved)
8:      end for
9:      while jobs whose best machine is available exist do
10:         Determine the best priority of all such jobs
11:         Schedule the job with the best priority on the corresponding machine
12:     end while
13: end while
```

As previously denoted, the SGS will use a PF to determine the priority of scheduling a job on a certain machine. Since it is difficult to design these PFs manually, they will be generated by the previously defined evolutionary computation methods. To evolve new PFs it is mandatory to define elements which will be used for constructing new PFs. Table 1 represents the set of terminal and function nodes which are used for constructing 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 that is used in the definitions of some nodes represents the current system time. Additionally, it needs to be stressed that the dd, SL, and w terminal nodes are used only when evolving DRs for optimising the due date related criteria (Twt and Nwt), since the information that these nodes provide is not meaningful for the other two optimised criteria. The last five nodes in the table represent the function nodes which are used by the methods to evolve expressions. Although many other function nodes can be used, a previous study demonstrated that for this set of function nodes GP achieved best results [56].

TABLE 1: 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	$POS(a) = \max(a, 0)$

III. EXPERIMENTAL SETUP

To train and test the PFs, two independent problem sets will be used. Both sets will 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 fitness values for each instance in the problem set. Since problems of different sizes will have different values for certain criteria, all fitness values were normalised in order to remove the dependency on the size of the problem instance. Additional details about the generation process of the problem sets can be found in [56].

Aside from defining the problem instances, it is also required to obtain optimal parameters for each of the methods, since the quality of the obtained solutions will depend heavily upon the parameters which were used for their generation. Therefore, for each of the previous methods the parameters were optimised for the Twt criterion. They were optimised in a way that all parameters were fixed to certain predefined values that were selected as a rule of thumb. These values are denoted in Table 2 in the initial value row. After that, each parameter was tested with several different values, also denoted in Table 2, while the others were kept fixed, either to the initial value, or the best found value after optimisation. For each parameter combination 30 experiments were performed and the parameter value which obtained the best average value of those 30 executions was selected. Thirty runs were performed in order to obtain statistically accurate results. The parameters that were optimised, the tested values, and the best values determined after the optimisation procedure are represented in Table 2. For the CGP it was decided to use one row with n_r number of columns, and a levels back value equal to n_r. These parameter values were suggested by the author of the approach for problems in which an arbitrary directed graph does not need to be implemented [67]. The smaller population values for CGP were intentionally tested since it is usually suggested that CGP is used with smaller population values. The final parameter values of all methods are presented in Table 3. These parameter values will later on be used when optimising the remaining three criteria as well, since optimising the parameters for each criterion individually would be too time consuming.

To ensure that the conclusions which are made based on the obtained results are significant, 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 were calculated and displayed for each experiment. Additionally, to determine whether certain results are better than others, the Mann-Whitney statistical test was used to calculate if there is a statistically significant difference between the different experiments.

IV. RESULTS

This section will represent the results which were achieved by the tested methods on the four selected scheduling criteria. Table 4 represents the results achieved by the selected evolutionary computation methods, for optimising four scheduling criteria. Additionally, Figure 1 represents the results in the form of box plots, to better denote the distribution of the obtained solutions. From the results it is immediately evident that no single method achieved the best results across all four scheduling criteria.

For the Twt criterion the best results were achieved by GP. Although GEP achieved results which were to a small extent worse, there was no significant difference between

TABLE 2: 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

its results and those achieved by GP. However, from the box plot representation it is visible that GP achieved less dispersed results than GEP, which makes it slightly more favourable. The Stack representation also performed slightly worse than GP and GEP when comparing by the median. However, it achieved the overall lowest minimal value. According to the box plot, the results were a bit more dispersed than GP and GEP, but less than GE or AP. CGP achieved results which are significantly worse than those of GP, but there was no significant difference between it and GEP. AP achieved results which are only better than CGP. The dispersion of the achieved results, which can be seen on the box plot, is greater than most other tested methods. Finally, GE achieved the worst results among all six methods, which can also be seen from the fact that this method usually achieved quite dispersed results.

In the case of the Nwt criterion the situation is somewhat more interesting. For this criterion the best results were again achieved by GP, GEP and Stack, with GEP achieving a better median value, while Stack achieved the best overall minimum and maximum values. However, no significant difference exists between the results of these methods. CGP, GE and AP achieved results which were significantly inferior to those of the other two methods, however, there was no significant difference in 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 those three methods. For this criterion Stack and GEP achieved results which were even significantly better than those of GP. Unfortunately, GE and AP achieved results which were significantly worse than any of the previous three methods. GE achieved results which were dispersed, and many evolved DRs performed poorly. However, AP achieved the least dispersed results but all of them performed poorly. Unfortunately, it is difficult to determine the reason why this would happen for this criterion. One possibility is that the chosen individual sizes for these two methods were inappropriate for this criterion, and that with another individual size better performance could be achieved.

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

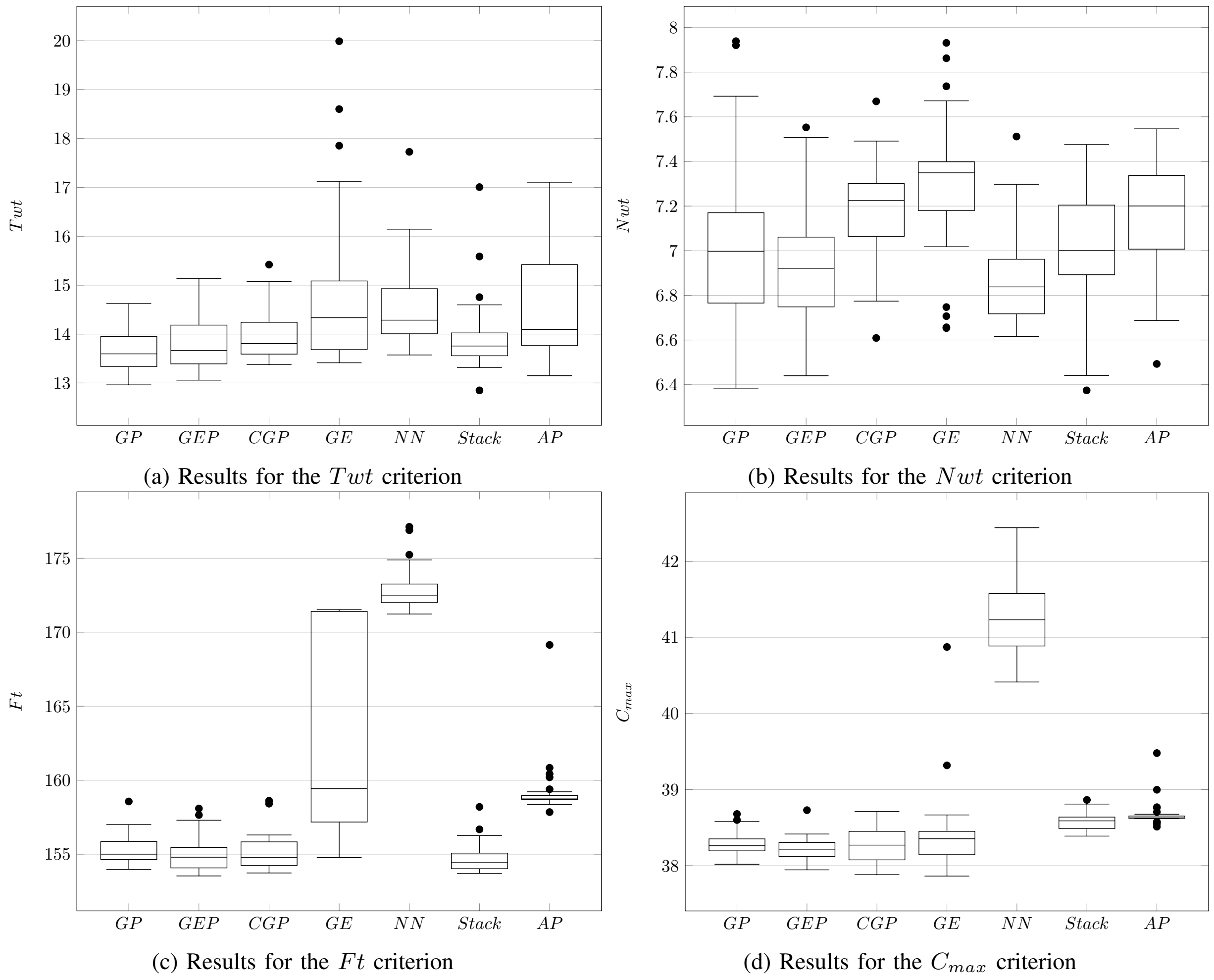
Apart from observing the performance of the different methods, it is also interesting to analyse the size of the

TABLE 3: 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, nodereplace-ment, permutation, shrink	Replacement mutation	One point, one gate	Simple	Node replacement	Simple
Transposition operators	-	IS, RIS, gene transposition	-	-	-	-

TABLE 4: 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



(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

decoded expressions of the evolved PFs. Table 5 outlines the average sizes of PFs generated for the four scheduling criteria. The first thing which can be noticed is that the different methods evolved expressions of substantially different sizes. GP and AP evolved the largest expressions, usually of sizes of around 40 elements. GEP, on the other hand, evolved somewhat simpler expressions, usually of sizes of around 30 elements. Stack evolved expressions which usually consisted of around 25 elements. The remaining two methods evolved the smallest expressions, of around 18 nodes. It is evident that the different methods have a preference to evolve 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, like GEP, CGP, and GE, especially since those methods have even demonstrated to achieve equal, or even slightly better performance than GP, like for the Ft and C_{max} criteria. The table also demonstrates that for certain criteria all methods tend to evolve slightly smaller expressions. Additionally, is is evident that almost consistently for the Nwt and C_{max} criteria, the methods generated expressions of average sizes which were for a few elements smaller than those generated for the Twt and Ft criteria. These results could possibly mean that for those two criteria it could be more beneficial to evolve PFs of smaller sizes.

TABLE 5: 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

V. FURTHER ANALYSIS

TABLE 6: 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

A. COMPARISON OF DR SIZES FOR DIFFERENT MAXIMUM INDIVIDUAL SIZES

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

Table 6 represents the results obtained for the different maximum tree depths used with GP. Apart from the average sizes of the evolved DRs, the table also includes the theoretical maximum size of an expression for the given depth. From the table it is evident that with the tree depth parameter it is quite hard to precisely control the sizes of the evolved individuals. This can be seen from the fact that the maximum expression size grows exponentially with the increase of the tree depth. Naturally, this also has an effect on the average size of the evolved PFs, which can be seen to consist of only 13 elements for the smallest tested depth, while for the largest tested depth the sizes consisted of even 440 elements on average. It is evident that the tree size grows substantially with the depth of the tree. Therefore, it can be concluded that with the tree depth parameter it is not simple to control the sizes of the evolved PFs.

It is also interesting to observe how the different tree depths influence the quality of the generated DRs. Figure 2 shows the box plot representation of the obtained results. The smallest tested depth resulted in quite bad results, with most DRs achieving a similar performance. By using the tree of depth 5, GP evolved DRs of the best quality, which can be seen from the fact that for this value GP achieved the best median value. As the depth is increased further, the results begin to deteriorate which can be seen from the increasing median value of the results. Additionally, the method also achieved more dispersed results as the depth increased, which is evident from the many outlier values which 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 size 5. Therefore, by using larger tree depths GP seems to have a greater possibility of evolving PFs with the absolute best performance. However, these rules were usually of quite large sizes, and it would therefore be hard to interpret them and extract knowledge out of them.

Table 7 represents the influence of the number of genes

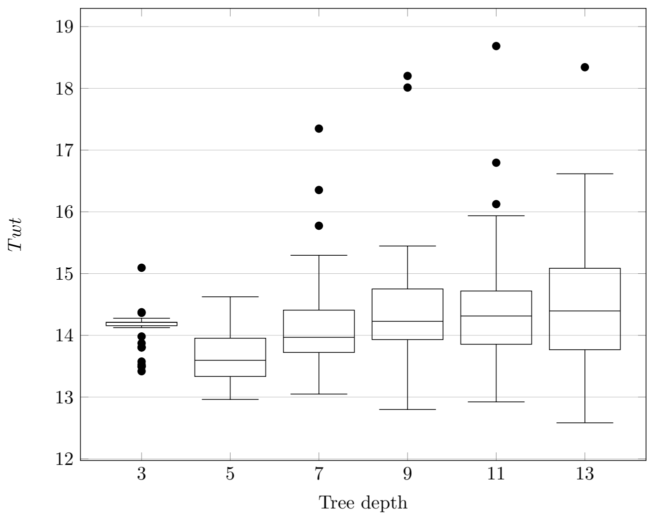


FIGURE 2: Results for different maximum tree depths of GP

TABLE 7: Influence of the number of genes and head size in GEP

Number of genes	Head size	Size		Fitness		
		th. max	avg	min	med	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

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

The performance of the PFs which were 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 with "g"), and the head size of each gene (denoted with "h"). Unlike when using GP, this time it can be observed that the obtained results for the different sizes were mostly similar. This is expected since the difference in

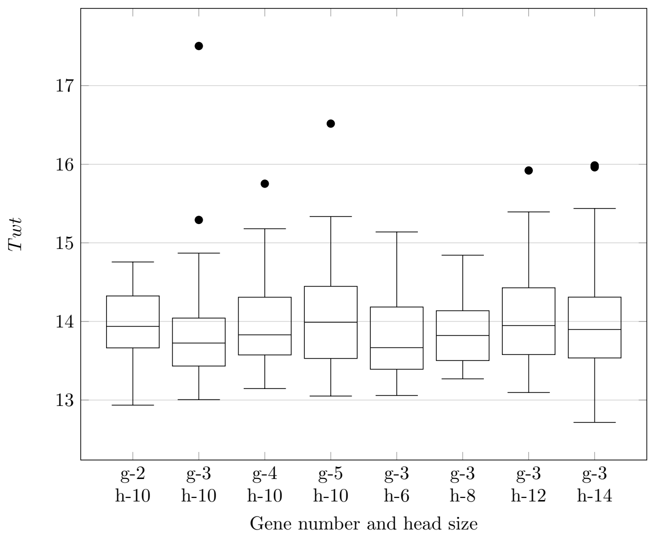


FIGURE 3: Box plot representation of results for different GEP parameter values

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

Table 8 represents the effect of the different individual sizes of CGP on the average sizes of the generated PFs and their quality. The table shows one quite interesting occurrence for CGP, which is that although the maximum size of the individual increases, the average size of the generated PFs is increased only slightly. For example, in the case of the smallest tested individual size, CGP evolved expressions which consisted of only 14 nodes on the average, while for the largest individual size CGP generated PFs consisting of around 24 elements on average. In several occasions CGP did evolve DRs of quite large sizes, however, these PFs usually did not achieve a very good result on the test set. Therefore, it seems that CGP is more focused on evolving PFs containing a smaller number of elements. The reason for this could be that the maximum value for the level-back parameter was used, meaning that CGP will be able to evolve individuals in which a large portion of the nodes will simply be skipped and will thus be inactive. This allows

TABLE 8: 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

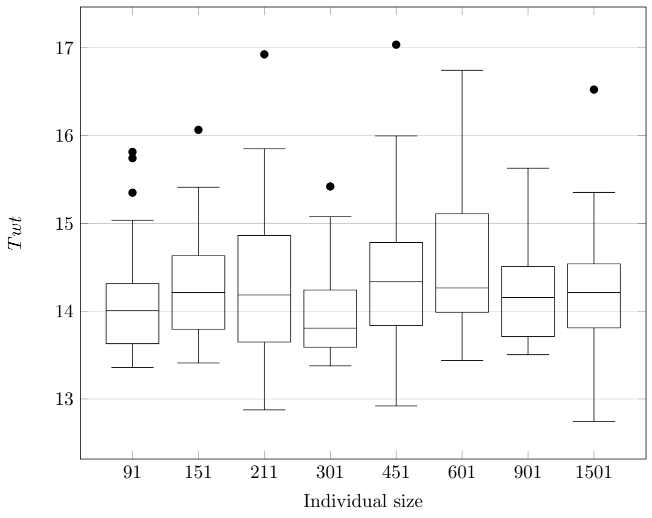


FIGURE 4: Results for different CGP maximum individual sizes

CGP to easily evolve PFs of the preferred sizes.

Figure 4 shows the box plot which represents the results achieved for the different individual sizes when using CGP. The figure shows that the size of the individuals has a significant influence on the quality of the results. The best results were achieved when using individuals of size 301, which would mean that the individual contains 100 nodes, which do not all need to be active. As the size of the individual decreases and increases, the fitness of the individuals deteriorates. For smaller individual sizes this is probably due to the fact that the individuals of smaller sizes might not be expressive enough, while the too large individuals are probably not quite suited due to the fact that the mutation will mostly be performed on inactive parts, which then does not have any effect on the quality of the individual. This seems to cause even more problems for CGP than when using too small individual sizes, since for larger individual sizes the algorithm usually achieved to a certain extent worse results than for smaller individual sizes.

Table 9 represents the results for the various individual sizes of GE. The results from the table denote that GE

TABLE 9: 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 even more biased towards evolving smaller expressions, which can be seen from the fact that the evolved expressions are relatively small when compared to the previous three methods, regardless of the maximum expression size which was used to evolve the PFs. Therefore, even for larger individual sizes, the average PF size will not contain more than 18 elements. However, larger PFs are sometimes generated for the larger individual sizes, but similarly as for the CGP, these individuals did not achieve really good results, and were easily outperformed by smaller PFs.

The performance of the PFs generated by various individual sizes when using GE are shown in Figure 5. GE did not achieve results which were competitive with those of other methods, which can be seen from the fact that for all individual sizes the method obtained a quite large median value. Furthermore, GE also obtained quite dispersed results, which can be seen not only from the number of outliers which were obtained by the method, but also from the fact that a great number of evolved PFs achieved bad results. Using too small individual sizes leads to the worst performance for GE. For the individual size of 150 elements GE achieved the best median values. As the size increases the performance of the algorithm deteriorates once again.

The results for different sizes of the Stack representation are shown in Table 10. In this case the theoretical maximum size of an individual is the same as the maximum individual size. However, it is interesting to see that the average size of individuals for the smallest tested sizes, 30 and 40, are about half of the theoretical maximum. This can be expected since most evolved expressions are going to be invalid, and therefore major parts of the expression are going to be discarded while transforming the individual into the PF. The results show that this happens even more for larger individual sizes. For example, for the maximum size 300, the average size of individuals is only around 46.

The performances of the PFs generated by different individual sizes of the Stack genotype are shown in Figure 6. The box plot shows that the results are somewhat more dispersed for individual sizes larger than 60. The best results are achieved when the maximum individual size is set to 60.

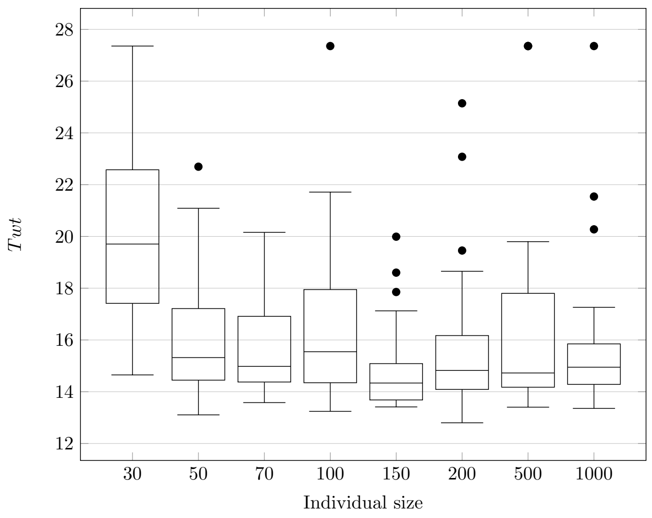


FIGURE 5: Box plot representation of results for different GE maximum individual sizes

TABLE 10: 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

Very similar results are achieved for the size 50, while the worst results are achieved for the smallest and the biggest tested sizes.

The results for different sizes of AP individuals are shown in Table 11. In this case, the theoretical maximum size is also equal to the set maximum individual size. Since each individual that is created in AP is of maximal size, most evolved individuals will end up being the same size. That explains why the average individual size for most tested sizes is almost the same as the theoretical maximum. The biggest difference is seen for individual size 100, where the average individual size is around 90. Therefore, with this method, controlling the sizes of evolved individuals is quite straightforward.

The performances of the PFs generated by different individual sizes are shown in Figure 7. In this case, the worst results are achieved for the smallest individual size, 10, while the best are achieved for size 50. However, the results for all tested sizes except the smallest one are quite similar and are similarly dispersed. This shows that for the AP method, increasing the individual size doesn't have much of an effect on the quality of the evolved individuals.

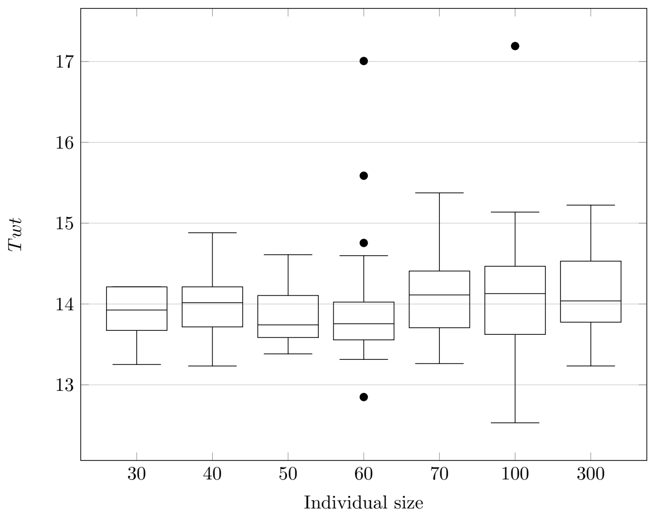


FIGURE 6: Results for different maximum tree depths of Stack genotype

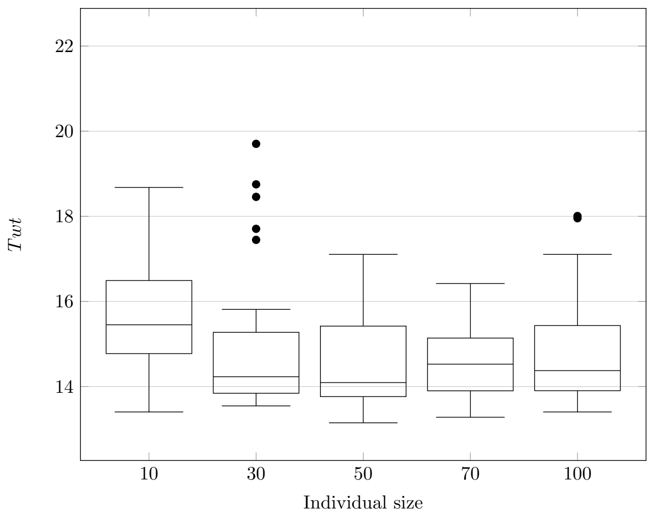


FIGURE 7: Results for different maximum tree depths of AP genotype

TABLE 11: 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

B. EXAMPLES OF GENERATED DRS

This section will give a short overview of the best PFs which were obtained by each of the six tested methods. It should be mentioned that the rules represented in this section do not represent the very best rules obtained by each of the methods, since the best obtained rules were quite often of large sizes, but rather the best PFs which were obtained for the parameter combinations denoted in Table 3. Nevertheless, since for those parameters all the methods achieved the best median values, the presented PFs should still give a good notion on the quality of the PFs which can be evolved by each of the methods.

Table 12 represents the PFs evolved by the four tested methods. The PF evolved by Stack is the best performing PF in the table. The size of that PF is 26 which is half of the size of the next best PF in the table, which makes it easier to interpret. The PF evolved by GP achieved the second best result, but was also the largest PF among the ones presented in the table. By observing the PF generated by GP, it can be seen that it contains several elements which do not have an effect on the value of the priority. For example, it can be seen that the PF applies the pos function on several terminal nodes, which by themselves can not be negative. Therefore, even by removing this function the values determined by the PF would not change, however, the PF would be slightly simpler. Additionally, it also often happens that the expression includes 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 quite smaller than the value of the $pmin$ terminal. Therefore, even if this expression were replaced by only $pmin$, it would probably not have a large influence on the fitness of the PF. Such situations are exactly one of the problems with the tree representation, since it tends to increase in size, without leading to a significant improvement of the fitness of the PF.

The PF generated by GEP is shown to achieve a slightly worse result than that of GP, but is almost two times smaller. The PF shows that for the exception of one unnecessary pos function, it does not really contain any elements which can immediately be classified as redundant. Therefore, it seems that GEP is able to control the size of its expressions to a much greater extent, and introduces much less noise into the generated expressions when compared to GP. The expression generated by CGP is of similar size to that of GEP, but it performs to a certain extent worse than the PF generate by GEP. It can be seen from the PF generated by CGP that it usually does not contain redundant subexpressions, since it contains an unnecessary pos function in only one occasion. Therefore, CGP also seems to be good for creating 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 out of a large number of subexpressions which in the end did not have any effect on the priority value. This can be seen from the several redundant pos functions applied to the SL and w terminals, but also from the fact that it generated some expressions like $pmin-pmin$ which in the end do not have any effect on the calculation of the value of the PF. Finally,

TABLE 12: PFs generated by the various methods

Method	Fitness	Size	PF
GP	12.96	52	$\frac{pmin-w}{\frac{w}{age}} - pos(pavg) * (pt + MR) + pos(SL) * \frac{dd}{MR} + \frac{pmin*SL}{pavg-SL} + (\frac{dd}{pt} + pavg + dd) * (pmin - w) + pos(SL) * (SL + pavg) + \frac{dd}{w}$
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	$\frac{\frac{pos((pos(\frac{\frac{\frac{w}{\frac{PAT}{\frac{\frac{dd*dd*dd}{dd*dd*dd}}{dd*dd*dd*dd*dd*dd}}}}{PAT}}{w}) * \frac{w}{\frac{pos(w)}{MR}}) + pmin + SL)}{pt} * pmin}{w}$

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 comes from the fact that large parts of the evolved expression are redundant. For example, $\frac{dd*dd*dd}{dd*dd*dd}$ could be reduced to just 1 which would reduce the expression size by 10 elements. Thus, out 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 which 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 which will act as the PFs, and offers different benefits. The tested methods are used for the generation of new DRs for optimising different scheduling criteria. Additionally, for each of the tested methods an analysis was performed on how different maximum individual sizes affect the performance of the methods, as well as the average size of the PFs they generate.

The results which were presented in this paper suggest that neither of the methods achieved the best results across all of the tested criteria. With the exception of GE and AP, which achieved quite bad results for most of the criteria, the remaining four methods achieved mostly similar results, with their performance depending largely on the criterion which 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, the four methods achieve mostly similar results for most of the tested criteria, therefore there should not be a significant difference in the results regardless of which of the four methods is used. As for the sizes of the generated PFs, CGP and GE generated expressions of the smallest average size out of all the tested methods. For all methods it was noticed that the generated expressions contain redundant parts. However, PFs generated with CGP and Stack contained the least amount of redundant subexpressions, thus outlining that those methods might be the most appropriate 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 of all it is planned to simply analyse the generated DRs to better understand which parts of PFs are redundant and which parts are the most informative. By using that information, the evolutionary algorithms will be enhanced with different methods which will try to detect redundant parts of PFs during the evolution process, and automatically remove those parts 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 restrict the search to only those DRs which are valid on the entire domain. Additionally, further research will also focus on testing different bloat control methods to further reduce the sizes of the generated PFs.

REFERENCES

[1] Michael L. Pinedo. Scheduling: Theory, algorithms, and systems: Fourth edition, volume 9781461423614. Springer US, Boston, MA, 2012.

[2] Emma Hart, Peter Ross, and David Corne. Evolutionary Scheduling: A Review. Genetic Programming and Evolvable Machines, 6(2):191–220, June 2005.

[3] Muthucumaru Maheswaran, Shoukat Ali, Howard Jay Siegel, Debra Hensgen, and Richard F. Freund. Dynamic Mapping of a Class of Independent Tasks onto Heterogeneous Computing Systems. Journal of Parallel and Distributed Computing, 59(2):107–131, November 1999.

[4] Tracy D Braun, Howard Jay Siegel, Noah Beck, Ladislau L Bölöni, Muthucumaru Maheswaran, Albert I Reuther, James P Robertson, Mitchell D Theys, Bin Yao, Debra Hensgen, and Richard F Freund. A Comparison of Eleven Static Heuristics for Mapping a Class of Independent Tasks onto Heterogeneous Distributed Computing Systems. Journal of Parallel and Distributed Computing, 61(6):810–837, June 2001.

[5] Jurgen Branke, Su Nguyen, Christoph W. Pickardt, and Mengjie Zhang. Automated Design of Production Scheduling Heuristics: A Review. IEEE Transactions on Evolutionary Computation, 20(1):110–124, February 2016.

[6] Su Nguyen, Yi Mei, and Mengjie Zhang. Genetic programming for production scheduling: a survey with a unified framework. Complex & Intelligent Systems, 3(1):41–66, March 2017.

[7] C. Dimopoulos and A.M.S. Zalzala. A genetic programming heuristic for the one-machine total tardiness problem. In Proceedings of the 1999 Congress on Evolutionary Computation-CEC99 (Cat. No. 99TH8406), pages 2207–2214. IEEE, 1999.

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

[9] Kazuo Miyashita. Job-shop scheduling with genetic programming. In Proceedings of the 2Nd Annual Conference on Genetic and Evolutionary Computation, GECCO'00, pages 505–512, San Francisco, CA, USA, 2000. Morgan Kaufmann Publishers Inc.

[10] Nhu Binh Ho and Joc Cing Tay. Evolving Dispatching Rules for solving the Flexible Job-Shop Problem. In 2005 IEEE Congress on Evolutionary Computation, volume 3, pages 2848–2855. IEEE, 2005.

[11] Domagoj Jakobović, Leonardo Jelenković, and Leo Budin. Genetic Programming Heuristics for Multiple Machine Scheduling. In Genetic Programming, pages 321–330. Springer Berlin Heidelberg, Berlin, Heidelberg, 2007.

[12] Domagoj Jakobović and Leo Budin. Dynamic scheduling with genetic programming. In Pierre Collet, Marco Tomassini, Marc Ebner, Steven Gustafson, and Anikó Ekárt, editors, Genetic Programming: 9th European Conference, EuroGP 2006, Budapest, Hungary, April 10-12, 2006. Proceedings, pages 73–84. Springer Berlin Heidelberg, Berlin, Heidelberg, 2006.

[13] Wen-Jun Yin, Min Liu, and Cheng Wu. Learning single-machine scheduling heuristics subject to machine breakdowns with genetic programming. In The 2003 Congress on Evolutionary Computation, 2003. CEC '03., volume 2, pages 1050–1055. IEEE, 2003.

[14] Christopher D. Geiger and Reha Uzsoy. Learning effective dispatching rules for batch processor scheduling. International Journal of Production Research, 46(6):1431–1454, March 2008.

[15] Domagoj Jakobović and Kristina Marasović. Evolving priority scheduling heuristics with genetic programming. Applied Soft Computing, 12(9):2781–2789, September 2012.

[16] Kristijan Jaklinović, Marko Đurasević, and Domagoj Jakobović. Designing dispatching rules with genetic programming for the unrelated machines environment with constraints. Expert Systems with Applications, 172:114548, 2021.

[17] Su Nguyen, Mengjie Zhang, Mark Johnston, and Kay Chen Tan. Dynamic multi-objective job shop scheduling: A genetic programming approach. In A. Sima Uyar, Ender Ozcan, and Neil Urquhart, editors, Automated Scheduling and Planning: From Theory to Practice, pages 251–282. Springer Berlin Heidelberg, Berlin, Heidelberg, 2013.

[18] Su Nguyen, Mengjie Zhang, and Kay Chen Tan. Enhancing genetic programming based hyper-heuristics for dynamic multi-objective job shop scheduling problems. In 2015 IEEE Congress on Evolutionary Computation (CEC), pages 2781–2788. IEEE, May 2015.

[19] Deepak Karunakaran, Gang Chen, and Mengjie Zhang. Parallel Multi-objective Job Shop Scheduling Using Genetic Programming. In Tapabrata Ray, Ruhul Sarker, and Xiaodong Li, editors, Artificial Life and Computational Intelligence: Second Australasian Conference, ACALCI 2016, Canberra, ACT, Australia, February 2-5, 2016, Proceedings, pages 234–245. Springer International Publishing, 2016.

[20] Marko Đurasević and Domagoj Jakobović. Evolving dispatching rules for optimising many-objective criteria in the unrelated machines environment. Genetic Programming and Evolvable Machines, Sep 2017.

[21] Su Nguyen, Mengjie Zhang, Mark Johnston, and Kay Chen Tan. Evolving reusable operation-based due-date assignment models for job shop scheduling with genetic programming. In Alberto Moraglio, Sara Silva, Krzysztof Krawiec, Penousal Machado, and Carlos Cotta, editors, Genetic Programming: 15th European Conference, EuroGP 2012, Málaga, Spain, April 11-13, 2012. Proceedings, pages 121–133. Springer Berlin Heidelberg, Berlin, Heidelberg, 2012.

[22] Su Nguyen, Mengjie Zhang, Mark Johnston, and Kay Chen Tan. Genetic Programming for Evolving Due-Date Assignment Models in Job Shop Environments. Evolutionary Computation, 22(1):105–138, March 2014.

[23] Su Nguyen, Mengjie Zhang, Mark Johnston, and Kay Chen Tan. A coevolution genetic programming method to evolve scheduling policies for dynamic multi-objective job shop scheduling problems. In 2012 IEEE Congress on Evolutionary Computation, pages 1–8. IEEE, June 2012.

[24] Su Nguyen, Mengjie Zhang, Mark Johnston, and Kay Chen Tan. Automatic Design of Scheduling Policies for Dynamic Multi-objective Job Shop Scheduling via Cooperative Coevolution Genetic Programming. IEEE Transactions on Evolutionary Computation, 18(2):193–208, April 2014.

[25] John Park, Su Nguyen, Mengjie Zhang, and Mark Johnston. Genetic programming for order acceptance and scheduling. In 2013 IEEE Congress on Evolutionary Computation, pages 1005–1012. IEEE, June 2013.

[26] Su Nguyen, Mengjie Zhang, Mark Johnston, and Kay Chen Tan. Learning Reusable Initial Solutions for Multi-objective Order Acceptance and Scheduling Problems with Genetic Programming. In Krzysztof Krawiec, Alberto Moraglio, Ting Hu, A. Şima Etaner-Uyar, and Bin Hu, editors, Genetic Programming: 16th European Conference, EuroGP 2013, Vienna, Austria, April 3-5, 2013. Proceedings, pages 157–168. Springer Berlin Heidelberg, Berlin, Heidelberg, 2013.

[27] Su Nguyen, Mengjie Zhang, and Mark Johnston. A sequential genetic programming method to learn forward construction heuristics for order acceptance and scheduling. In 2014 IEEE Congress on Evolutionary Computation (CEC), pages 1824–1831. IEEE, July 2014.

[28] Su Nguyen, Mengjie Zhang, and Kay Chen Tan. A Dispatching rule based Genetic Algorithm for Order Acceptance and Scheduling. In Proceedings of the 2015 on Genetic and Evolutionary Computation Conference - GECCO '15, pages 433–440, New York, New York, USA, 2015. ACM Press.

[29] Su Nguyen. A learning and optimizing system for order acceptance and scheduling. The International Journal of Advanced Manufacturing Technology, 86(5-8):2021–2036, September 2016.

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

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

[32] Francisco J. Gil-Gala, Carlos Mencía, María R. Sierra, and Ramiro Varela. Evolving priority rules for on-line scheduling of jobs on a single machine with variable capacity over time. Applied Soft Computing, 85:105782, 2019.

[33] Francisco J. Gil-Gala, María R. Sierra, Carlos Mencía, and Ramiro Varela. Genetic programming with local search to evolve priority rules for scheduling jobs on a machine with time-varying capacity. Swarm and Evolutionary Computation, 66:100944, 2021.

[34] John Park, Su Nguyen, Mengjie Zhang, and Mark Johnston. Evolving ensembles of dispatching rules using genetic programming for job shop scheduling. In Penousal Machado, Malcolm I. Heywood, James McDermott, Mauro Castelli, Pablo García-Sánchez, Paolo Burelli, Sebastian Risi, and Kevin Sim, editors, Genetic Programming: 18th European Conference, EuroGP 2015, Copenhagen, Denmark, April 8-10, 2015, Proceedings, pages 92–104. Springer International Publishing, Cham, 2015.

[35] Emma Hart and Kevin Sim. A Hyper-Heuristic Ensemble Method for Static Job-Shop Scheduling. Evolutionary Computation, 24(4):609–635, December 2016.

[36] Marko Đurasević and Domagoj Jakobović. Comparison of ensemble learning methods for creating ensembles of dispatching rules for the unrelated machines environment. Genetic Programming and Evolvable Machines, Apr 2017.

[37] John Park, Yi Mei, Su Nguyen, Gang Chen, and Mengjie Zhang. An investigation of ensemble combination schemes for genetic programming based hyper-heuristic approaches to dynamic job shop scheduling. Applied Soft Computing, 63:72 – 86, 2018.

[38] Marko Đurasević and Domagoj Jakobović. Creating dispatching rules by simple ensemble combination. Journal of Heuristics, 25(6):959–1013, may 2019.

[39] Deepak Karunakaran, Yi Mei, Gang Chen, and Mengjie Zhang. Dynamic Job Shop Scheduling Under Uncertainty Using Genetic Programming. In George Leu, Hemant Kumar Singh, and Saber Elsayed, editors, Intelligent and Evolutionary Systems: The 20th Asia Pacific Symposium, IES 2016, Canberra, Australia, November 2016, Proceedings, pages 195–210. Springer International Publishing, Cham, 2017.

[40] Deepak Karunakaran, Yi Mei, Gang Chen, and Mengjie Zhang. Evolving dispatching rules for dynamic Job shop scheduling with uncertain processing times. In 2017 IEEE Congress on Evolutionary Computation (CEC), pages 364–371. IEEE, June 2017.

[41] Deepak Karunakaran, Yi Mei, Gang Chen, and Mengjie Zhang. Toward evolving dispatching rules for dynamic job shop scheduling under uncertainty. In Proceedings of the Genetic and Evolutionary Computation Conference on - GECCO '17, pages 282–289, New York, New York, USA, 2017. ACM Press.

[42] Su Nguyen, Mengjie Zhang, Mark Johnston, and Kay Chen Tan. Selection Schemes in Surrogate-Assisted Genetic Programming for Job Shop Scheduling. In Grant Dick, Will N. Browne, Peter Whigham, Mengjie Zhang, Lam Thu Bui, Hisao Ishibuchi, Yaochu Jin, Xiaodong Li, Yuhui Shi, Pramod Singh, Kay Chen Tan, and Ke Tang, editors, Simulated Evolution and Learning: 10th International Conference, SEAL 2014, Dunedin, New Zealand, December 15-18, 2014. Proceedings, pages 656–667. Springer International Publishing, Cham, 2014.

[43] Su Nguyen, Mengjie Zhang, and Kay Chen Tan. Surrogate-Assisted Genetic Programming With Simplified Models for Automated Design of Dispatching Rules. IEEE Transactions on Cybernetics, pages 1–15, 2016.

[44] S. Nguyen, M. Zhang, D. Alahakoon, and K. C. Tan. Visualizing the evolution of computer programs for genetic programming [research frontier]. IEEE Computational Intelligence Magazine, 13(4):77–94, Nov 2018.

[45] S. Nguyen, M. Zhang, D. Alahakoon, and K. C. Tan. People-centric evolutionary system for dynamic production scheduling. IEEE Transactions on Cybernetics, pages 1–14, 2019.

[46] Fangfang Zhang, Yi Mei, Su Nguyen, and Mengjie Zhang. Guided subtree selection for genetic operators in genetic programming for dynamic flexible job shop scheduling. In Ting Hu, Nuno Lourenço, Eric Medvet, and Federico Divina, editors, Genetic Programming, pages 262–278, Cham, 2020. Springer International Publishing.

[47] D. Karunakaran, Y. Mei, G. Chen, and M. Zhang. Active sampling for dynamic job shop scheduling using genetic programming. In 2019 IEEE Congress on Evolutionary Computation (CEC), pages 434–441, 2019.

[48] Marko Durasević and Domagoj Jakobović. Comparison of schedule generation schemes for designing dispatching rules with genetic programming in the unrelated machines environment. Applied Soft Computing, 96:106637, 2020.

[49] Marko Đurasević and Domagoj Jakobović. Automatic design of dispatching rules for static scheduling conditions. Neural Computing and Applications, August 2020.

[50] Fangfang Zhang, Yi Mei, Su Nguyen, Kay Chen Tan, and Mengjie Zhang. Multitask genetic programming-based generative hyperheuristics: A case study in dynamic scheduling. IEEE Transactions on Cybernetics, pages 1–14, 2021.

[51] Fangfang Zhang, Yi Mei, Su Nguyen, Mengjie Zhang, and Kay Chen Tan. Surrogate-assisted evolutionary multitask genetic programming for dynamic flexible job shop scheduling. IEEE Transactions on Evolutionary Computation, 25(4):651–665, 2021.

[52] Ivan Vlašić, Marko Đurasević, and Domagoj Jakobović. Improving genetic algorithm performance by population initialisation with dispatching rules. Computers & Industrial Engineering, 137:106030, 2019.

[53] Li Nie, Xinyu Shao, Liang Gao, and Weidong Li. Evolving scheduling rules with gene expression programming for dynamic single-machine scheduling problems. The International Journal of Advanced Manufacturing Technology, 50(5-8):729–747, September 2010.

[54] Li Nie, Liang Gao, Peigen Li, and Liping Zhang. Application of gene expression programming on dynamic job shop scheduling problem. In Proceedings of the 2011 15th International Conference on Computer Supported Cooperative Work in Design (CSCWD), pages 291–295. IEEE, June 2011.

[55] Li Nie, Yuewei Bai, Xiaogang Wang, and Kai Liu. Discover Scheduling Strategies with Gene Expression Programming for Dynamic Flexible Job Shop Scheduling Problem. In Ying Tan, Yuhui Shi, and Zhen Ji, editors, Advances in Swarm Intelligence: Third International Conference, ICSI 2012, Shenzhen, China, June 17-20, 2012 Proceedings, Part II, pages 383–390. Springer Berlin Heidelberg, Berlin, Heidelberg, 2012.

[56] Marko Durasević, Domagoj Jakobović, and Karlo Knežević. Adaptive scheduling on unrelated machines with genetic programming. Applied Soft Computing, 48:419–430, November 2016.

[57] Su Nguyen, Mengjie Zhang, Mark Johnston, and Kay Chen Tan. A Computational Study of Representations in Genetic Programming to Evolve Dispatching Rules for the Job Shop Scheduling Problem. IEEE Transactions on Evolutionary Computation, 17(5):621–639, October 2013.

[58] Jürgen Branke, Torsten Hildebrandt, and Bernd Scholz-Reiter. Hyper-heuristic Evolution of Dispatching Rules: A Comparison of Rule Representations. Evolutionary Computation, 23(2):249–277, June 2015.

[59] Su Nguyen, Mengjie Zhang, Mark Johnston, and Kay Chen Tan. Learning iterative dispatching rules for job shop scheduling with genetic programming. The International Journal of Advanced Manufacturing Technology, 67(1-4):85–100, July 2013.

[60] Maarten Keijzer and Vladan Babovic. Dimensionally Aware Genetic Programming. Proceedings of the Genetic and Evolutionary Computation Conference, 2:1069–1076, 1999.

[61] Riccardo Poli, William B. Langdon, and Nicholas Freitag McPhee. A field guide to genetic programming. Published via `http://lulu.com` and freely available at `http://www.gp-field-guide.org.uk`, 2008. (With contributions by J. R. Koza).

[62] Candida Ferreira. Gene expression programming: a new adaptive algorithm for solving problems. Complex Systems, 13(2):87–129, 2001.

[63] Julian F. Miller and Peter Thomson. Cartesian genetic programming. In Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics), volume 1802, pages 121–132, 2000.

[64] Conor Ryan, J. J. Collins, and Michael O'Neill. Grammatical evolution: Evolving programs for an arbitrary language. In Wolfgang Banzhaf, Riccardo Poli, Marc Schoenauer, and Terence C. Fogarty, editors, Proceedings of the First European Workshop on Genetic Programming, volume 1391 of LNCS, pages 83–96, Paris, 14-15 April 1998. Springer-Verlag.

[65] T. Perkis. Stack-based genetic programming. In Proceedings of the First IEEE Conference on Evolutionary Computation. IEEE World Congress on Computational Intelligence, pages 148–153 vol.1, 1994.

[66] Ivan Zelinka, Zuzana Kominkova Oplatkova, and Lars Nolle. Analytic programming symbolic regression by means of arbitrary evolutionary algorithm. J. of SIMULATION, 6:1473–8031, 08 2005.

[67] Julian F. Miller. Cartesian Genetic Programming, pages 17–34. Springer Berlin Heidelberg, Berlin, Heidelberg, 2011.

LUCIJA PLANINIĆ is a research assistant at the Faculty of Electrical Engineering and Computing, University of Zagreb. She received the B.Sc. and M.Sc. degree in computer science from the Faculty of Electrical Engineering and Computing, University of Zagreb in 2018 and 2020 respectively. She is currently pursuing her PhD at the Faculty of Electrical Engineering and Computing at the University of Zagreb.

HRVOJE BACKOVIĆ received the B.Sc. and M.Sc. degree in computing from the Faculty of Electrical Engineering and Computing, University of Zagreb in 2015 and 2017 respectively. Currently, he is working as a software developer at Visage Technologies, Zagreb, Croatia.

MARKO ĐURASEVIĆ is a assistant professor at the Faculty of Electrical Engineering and computing, University of Zagreb. He received the B.Sc. and M.Sc. degree in computing from the Faculty of Electrical Engineering and Computing, University of Zagreb in 2012 and 2014 respectively. Furthermore, he received his PhD degree in February 2018 on the subject of generating dispatching rules for solving scheduling problems in the unrelated machines environment.

DOMAGOJ JAKOBOVIĆ is a Full professor at Faculty of Electrical Engineering and Computing, University of Zagreb. He received B.S. degree in December 1996. and MS degree in December 2001. in Electrical Engineering. Since April 1997. he is a member of the research and teaching staff at the Department of Electronics, Microelectronics, Computer and Intelligent Systems of Faculty of Electrical Engineering and Computing, University of Zagreb. He received Ph.D. degree in December 2005 on the subject of generating scheduling heuristics with genetic programming.

• • •