Graph-based Linear Genetic Programming: A Case Study of Dynamic Scheduling

Zhixing Huang, Yi Mei, Fangfang Zhang✉, Mengjie Zhang
{zhixing.huang,yi.mei,fangfang.zhang,mengjie.zhang}@ecs.vuw.ac.nz
School of Engineering and Computer Science
Victoria University of Wellington
Wellington, New Zealand

ABSTRACT

Linear genetic programming (LGP) has been successfully applied to various problems such as classification, symbolic regression and hyper-heuristics for automatic heuristic design. In contrast with the traditional tree-based genetic programming (TGP), LGP uses a sequence of instructions to represent an individual (program), and the data is carried by registers. A common issue of LGP is that LGP is susceptible to introns (i.e., instructions with no effect to the program output), which limits the effectiveness of traditional genetic operators. To address these issues, we propose a new graph-based LGP system. Specifically, graph-based LGP uses graph-based crossover and graph-based mutation to produce offspring. The graph-based crossover operator firstly converts each LGP parent to a directed acyclic graph (DAG), and then swaps the sub-graphs between the DAGs. The graph-based mutation selectively modify the connections in DAGs based on the height of sub graphs. To verify the effectiveness of the new graph-based genetic operators, we take the dynamic job shop scheduling as a case study, which has shown to be a challenging problem for LGP. The experimental results show that the LGP with the new graph-based genetic operators can obtain better scheduling heuristics than the LGP with the traditional operators and TGP.

KEYWORDS

linear genetic programming, directed acyclic graph, hyper-heuristic, dynamic job shop scheduling.

ACM Reference Format:
Zhixing Huang, Yi Mei, Fangfang Zhang✉, Mengjie Zhang. 2022. Graph-based Linear Genetic Programming: A Case Study of Dynamic Scheduling . In *Genetic and Evolutionary Computation Conference (GECCO '22), July 9–13, 2022, Boston, MA, USA.* ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/3512290.3528730

1 INTRODUCTION

Linear genetic programming (LGP) is an important member of the genetic programming (GP) family [4, 28]. Unlike the traditional tree-based genetic programming (TGP), the individuals in LGP are represented linearly as a sequence of instructions, where each instruction is an elementary single-step operation (e.g., $R1 = x + 1$). Compared with TGP, LGP has a number of advantages, such as the ability to reusing building blocks (while TGP requires to copy sub-trees to this end) and better flexibility to evolve different topological structures. As a result, LGP has been successfully applied to various machine learning and optimisation problems such as classification [17], regression [34], and automatic heuristic design [7, 18, 36]. However, contrary to TGP which has been widely applied to scheduling problems [40–42], there are only a few preliminary investigations on LGP for evolving dispatching rules for dynamic scheduling [18]. They show that dynamic scheduling problems are currently challenging problems for LGP.

A major issue of LGP is the effectiveness of genetic operators on the linear representation. In TGP, each sub-tree is connected to the root node, and thus makes effect to the final program output (although TGP may still have redundant sub-trees such as adding zero or multiplying by one). For LGP, however, due to the linear representation, it is more susceptible to no effect instructions than TGP. First, an instruction can be *redundant*. For example, in the program ($R1 = x + 2$; $R1 = x * 3$;), the latter instruction makes the former one redundant. Second, a block of code can be *irrelevant* to the program output. For example, given the program ($R3 = R1 + 1$; $R3 = R3 + R2$; $R0 = R1 * 2$;), where $R0$ is the program output register. In this program, the first two instructions are irrelevant to the program output. The instructions that do not affect the final program output are called *introns*. If we see an LGP program as a directed acyclic graph (DAG), then the DAG can have multiple connected sub-graphs, and the introns are in the sub-graphs that are isolated from the main DAG that contains the final program output. The traditional linear genetic operators directly modify the sequence of instructions without considering whether the modified parts have an effect on the program output or not. Though some variants of genetic operators ensure that at least one effective instruction is modified in breeding, they often destruct useful building blocks (i.e., topological structures of effective instructions). In this case, they may have a too large variation step size, and cannot strike a good balance between exploration and exploitation. In other words, the linear genetic operators are less effective than the tree-based genetic operators for TGP.

To address this issue, we propose two new LGP graph-based genetic operators. Unlike the traditional linear genetic operators that directly consider the sequence of instructions, the proposed graph-based genetic operators consider the DAGs corresponding to the sequence. This way, they can take the relationship (links in the DAG) between instructions into account, and select only

GECCO '22, July 9–13, 2022, Boston, MA, USA

ACM ISBN 978-1-4503-9237-2/22/07…$15.00
https://doi.org/10.1145/3512290.3528730

the instructions contributing to the final program output for modification. Specifically, the graph-based crossover operator selects a sub-graph from the DAGs corresponding to both parents and swaps them. The graph-based mutation mutates registers (i.e., connection in DAGs) based on the height of sub-graphs.

To verify the effectiveness of the LGP with the newly developed graph-based genetic operators, we take the dynamic job shop scheduling (DJSS) as a case study, since previous investigations [18] have shown that this is a challenging problem for LGP (i.e., LGP cannot obtain better dispatching rules than TGP). We expect that the new graph-based genetic operators can improve the search ability of LGP to evolve better dispatching rules for DJSS.

The overall goal of this paper is to develop a new LGP with graph-based genetic operators to achieve better search ability. Specifically, we have the following research objectives:

- To develop a novel graph-based crossover operator and a new graph-based mutation operator for linear representation. The new genetic operators considers the DAGs rather than the raw sequence to select the parts to modify.
- To develop a new LGP system with the graph-based genetic operators.
- To verify the effectiveness of the new LGP system on a number of DJSS scenarios in terms of the performance of the evolved dispatching rules.
- To verify the effectiveness of the new graph-based crossover and mutation operators independently, and analyse their behaviours on modifying the topological structures of the parents.

2 BACKGROUND

2.1 Linear genetic programming

LGP is a GP variant which encodes computer programs or mathematical formula by a sequence of instructions [28]. Each instruction has three main components: destination register, operation/function, and source registers. The operation takes the values from the source registers as inputs, calculates the result, and writes the result to the destination register. For example, in an instruction $R0 = x + 1$, the destination register is $R0$, the operation is +, and the two source registers are x and 1. An LGP individual is denoted as $F = [f_0, f_1, ..., f_{l-1}]$, where l is length of (i.e., the number of instructions in) the individual. For each instruction f_i, its destination register, operation, and source registers are denoted as $des(f_i)$, $op(f_i)$, and $\mathbf{src}(f_i)$, where $\mathbf{src}(f_i) = [src_1(f_i), \ldots, src_k(f_i)]$ is a vector of source registers (e.g., $k = 2$ for typical arithmetic operations).

Fig. 1 is an example of an LGP program with $R0$ as the program output register. It represents a formula that calculates $x_0 \times (x_1 + x_0) - x_2/x_1$. Note that the input features x_i $(i = 0, 1, 2)$ are regarded as a kind of constant registers. All the registers $R0$, $R1$ and $R2$ are initialised to 0. The figure shows both the sequence of instructions and its corresponding DAG. An LGP individual can be decoded into a DAG in $O(n)$ time [1]. In the DAG, the operation vertices have at least one outgoing edge pointing to another function or input feature, while input features have no outgoing edges. Each function and input feature vertex also has incoming edges denoting other vertices that take its results as inputs. These connections from one vertex to another are designated by registers. From the DAG, it can be seen that x_0 and $0 + x_1$ are reused.

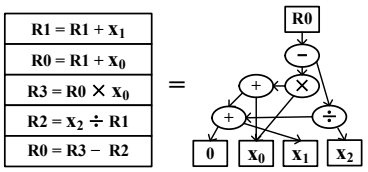

Figure 1: An example of LGP programs and its corresponding DAG

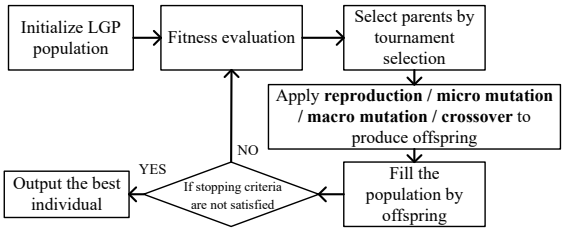

Figure 2: The overall framework of LGP.

Based on the linear representation, LGP has its own genetic operators. There are three basic operators for LGP, linear crossover, macro mutation, and micro mutation. Linear crossover is similar to the crossover in genetic algorithm. It respectively selects a segment of instructions from two parent chromosomes. The two selected segments are then exchanged to produce offspring. For the two types of mutations, they mainly produce offspring by generating new instructions in individuals. The key difference between them is that macro mutation produces offspring by inserting new instructions to a parent individual or removing instructions from it, which will affect the total number of instructions in the individual. Contrarily, micro mutation only mutates the primitives of existing instructions in parent individuals. It will not change the total number of instructions and has a smaller variation step size than macro mutation.

Fig. 2 shows the flowchart of the overall framework of LGP. It follows the standard evolutionary algorithm framework of initialisation, fitness evaluation, parent selection and breeding. The key characteristic of the LGP framework is that it uses the linear crossover, macro mutation and micro mutation, which are designed based on linear representation, to generate offspring.

2.2 Related work

GP methods are conventionally designed based on tree-like structures when it was popularised by Koza [20]. In addition to tree-like structure, several chromosome representations are also proposed to enhance GP methods. For example, Miller et al. [24] proposed a Cartesian GP which encodes a computer program into a grid of nodes. Each node represents a function like a logic gate. The connection among these nodes are designated by a Cartesian co-ordinate system. Cartesian GP has shown a superior performance in circuit design [22, 23], image processing [33], and evolving

neural network architecture [25]. Gene expression programming is another notable variant of GP methods [14]. Different from conventional GP which simply represents a formula by a tree-like structure, gene expression programming encodes a mathematical formula by a sequence of primitives and decodes it back to tree-like structure in execution. Such design leads to a better search efficiency of gene expression programming than tree-based GP. It also relieves the bloat effect in tree-based GP. Gene expression programming has been successfully applied to symbolic regression and classification [43, 44]. Constraints can also be applied to chromosome representation to tailor the search space. For example, grammar-guided GP, which applies a set of grammars to constrain the data types and data structures of tree-based individuals, is proven to be effective in solving program synthesis problems [16]. Besides, the chromosome can be bit strings. For example, O'Neill et al. [31, 32] proposed grammatical evolution which decodes bit strings into complete computer programs based on a set of Backus–Naur form grammars. Besides these GP variants, there are some other GP representations which have shown a prominent performance in certain areas. For example, PushGP which encodes computer programs by Push programming language, is good at solving program synthesis problems [35]. Linear-tree GP which extends GP tree nodes into a linear structure has superior performance to tree-based GP in symbolic regression [19].

LGP is one of prominent variants of GP methods and it has shown superior performance in symbolic regression and classification problems [4, 29]. For example, Downey and Fogelberg respectively applied LGP to solve multi-class image classification problems [12, 13, 15]. Different from tree-like structures which only have one output (i.e., the root), LGP naturally has multiple outputs if defining multiple output registers, each for one sub-class classification. Since these output registers can fully utilise common building blocks for different sub-classes, LGP has advantages over tree-based GP in multi-class classification. LGP also shows superior performance to tree-based GP and artificial neural network in binary classification [2, 34]. Besides, LGP has undergone a good development in solving symbolic regression. For example, Dal Piccol Sotto et al. [7] developed a probability model to learn the distribution of elite LGP individuals and sample offspring based on the probability model in symbolic regression problems. Sotto et al. [6] also verified that LGP has better bloat control than tree-based GP in symbolic regression problems. Additional to these work, LGP is successfully applied to other domains such as automatic algorithm design [8] and parallel computation [10, 11]. LGP has also been applied to dynamic scheduling [18]. However, the preliminary investigation in [18] only validates that LGP is competitive with TGP in solving DJSS problems.

Genetic operator is an important design issue for GP methods [39]. Besides the conventional genetic operators introduced above, some genetic operators of LGP were developed in the last two decades. For example, Banzhaf and Brameier et al. [1, 3] made a lot of comparisons on different types of crossover and mutation. They found that ensuring the effectiveness (i.e., the variation must change at least one LGP effective instructions) and neutrality (i.e., locally search different offspring and ensure that the fitness of offspring must be better or at least equal to the one of its corresponding parent) of LGP mutation can significantly improve the performance of LGP in solving classification and symbolic regression. Besides, based on the domain knowledge, some problem specific operators were proposed for LGP. For example, based on multiple output registers in multi-class classification, Downey et al. [12] designed a class path crossover which swaps the DAGs contributing to a same sub-class. In genetic improvement, some mutation and crossover were also proposed based on a new linear representation of software repair operations [30]. Though these genetic operators successfully improved LGP performance, they are not efficient enough (e.g., multiple fitness evaluation is required), or cannot be extended to dynamic scheduling. To fully utilise the the topological structures of effective instructions to enhance the training and test performance of LGP, a graph-based crossover is developed for LGP in this paper.

3 GRAPH-BASED LGP

This section describes the newly developed LGP with the graph-based genetic operators. The new algorithm follows the generic LGP overall framework as in Fig. 2. The main differences are the newly developed graph-based crossover and mutation operators, which will be described in more detail below.

3.1 Graph-based Crossover

The proposed graph-based crossover for LGP aims to swap LGP instructions based on their topological structures, rather than their raw genome. This way, the topological structures of building blocks are not easily destructed by recombination. Briefly speaking, it first decodes the raw LGP parents into corresponding DAGs. Then, it selects a sub-graph in the main DAGs (producing the final output) in each of the two parents and swaps them. The swapping is done on the sequences of the instructions directly to remove the need of adjusting the links in the resultant DAG.

Algorithm 1 shows the pseudo code of the graph-based crossover operator, where $||\cdot||$ indicates the cardinality (number of elements) of a set/list. Given two LGP parents p_1 and p_2, a set of registers $\mathbb{R}$, as well as three crossover parameters, i.e., the maximal size of sub-graphs $\overline{S}$, size gap limit Δ_S and maximal distance of crossover points D_{cross}, the graph-based crossover first extracts the lists of *effective* instructions (i.e., the main DAG with the final output) p'_1 and p'_2. Then, it selects a sub-graph from each parent subject to the following constraints: (1) the number of effective instructions in both offspring after the swapping is within the graph size range $[\underline{l}, \overline{l}]$, and (2) the gap between the two sub-graph sizes do not exceed Δ_S. Then, it swaps the two sub-graphs to generate two offspring. Note that each parent plays the role of recipient and donor alternatively in the swapping.

The sub-graph of an individual is obtained by backtracking the effective instructions from the selected crossover point. If the current instruction contributes to the target instruction at the crossover point (there is a path from its destination register to the target instruction), the current instruction is added into the sub-graph. The pseudo code is shown in Algorithm 2, where the edge of the paths pointing to the target instruction is stored in $\mathbb{T}$.

The pseudo code of the sub-graph swapping is shown in Algorithm 3. It generates an offspring by replacing the instructions in sub-graph G_1 of the recipient parent p_1 with the sub-graph G_2 of the donor parent p_2. Specifically, it replaces the last instruction

Algorithm 1: GraphCrossover(p_1, p_2, $\mathbb{R}$, $\overline{S}$, Δ_S, D_{cross})

Input: Two LGP parents p_1 and p_2, register set $\mathbb{R}$, maximal size of sub-graphs $\overline{S}$, size gap limit Δ_S, maximal distance of crossover points D_{cross}.
Output: Two LGP offspring c_1 and c_2.

Extract the list of *effective* instructions $p_1' \subseteq p_1$, $p_2' \subseteq p_2$;
/* Select sub-graphs */
repeat
  Randomly select two registers $r_1, r_2 \in \mathbb{R}$;
  repeat
    Randomly sample an instruction index $i_1 \sim U[1, ||p_1'||]$ and $i_2 \sim U[1, ||p_2'||]$;
  until $|i_1 - i_2| \leq D_{cross}$;
  Randomly sample $S_1 \sim U[1, \overline{S}]$, $S_2 \sim U[1, \overline{S}]$;
  G_1 = GetSubGraph(p_1', $\{r_1\}$, i_1, S_1);
  G_2 = GetSubGraph(p_2', $\{r_2\}$, i_2, S_2);
until $||p_1'|| - ||G_1|| + ||G_2|| \in [\underline{l}, \overline{l}]$ *and* $||p_2'|| - ||G_2|| + ||G_1|| \in [\underline{l}, \overline{l}]$ *and* $|||G_1|| - ||G_2||| \leq \Delta_S$;
/* Swap the sub-graphs */
c_1 = GraphSwap(p_1, p_2, G_1, G_2);
c_2 = GraphSwap(p_2, p_1, G_2, G_1);
return c_1, c_2;

Algorithm 2: GetSubGraph(p, $\mathbb{T}$, i, S)

Input: An LGP individual p, a set of target register $\mathbb{T}$, crossover point i, graph size S.
Output: A sub-graph G, represented as a list of instructions.

$G = []$;
for $j = i \rightarrow 0$ **do**
  if $des(p_j) \in \mathbb{T}$ **then**
    $G = [p_j, G]$;
    if $||G|| = S$ **then break**;
    $\mathbb{T} = \mathbb{T} \setminus des(p_j)$;
    for $src \in$ **src**(p_j) **do**
      if *src is a register* **then** $\mathbb{T} = \mathbb{T} \cup src$;
return G;

Algorithm 3: GraphSwap(p_1, p_2, G_1, G_2)

Input: A recipient LGP parent p_1, a donor LGP parent p_2, sub-graphs (lists of instructions) G_1, G_2
Output: An LGP offspring c

$c = p_1$;
Replace $G_1[||G_1|| - 1]$ in c with G_2;
Remove $G_1[0 : ||G_1|| - 2]$ from c;
while $||c|| > \bar{l}$ **do** Randomly remove an intron from c;
return c;

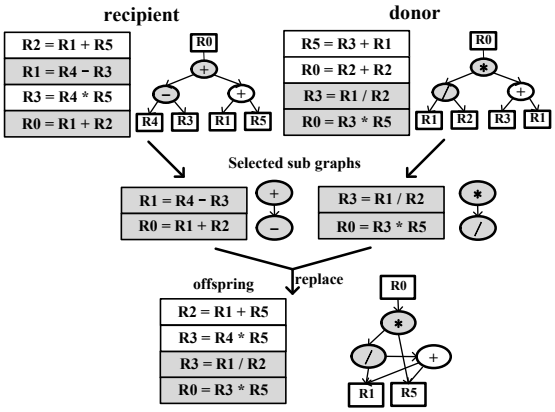

Figure 3: Example of graph-based crossover.

$G_1[||G_1|| - 1]$ in G_1 with G_2. Then, it removes all the other instructions in G_1 from the offspring. Note that the swapping does not replace the instructions at their original positions, but only retains the position of the last instruction of the sub-graphs. This way, we can retain the topological structure of the sub-graph G_2 of the donor parent, which would increase the effectiveness of the swapping/replacement.

An example of the graph-based crossover is shown in Fig. 3. The parent on the left is the recipient and the right one is the donor. We can see that the third instruction $R3 = R4 * R5$ on the left and the second instruction $R0 = R2+R2$ are introns and do not appear in the DAGs. The instructions in the selected sub-graphs are highlighted in gray. Finally, we replace the instruction $R0 = R1 + R2$ on the left with the sub-graph ($R3 = R1/R2$; $R0 = R3 * R5$;) on the right, and remove the remaining instruction $R1 = R4 - R3$ from the left. From this example, we can see that the offspring inherits some building blocks from both the left parent (e.g., $R1 + R5$) and the right parent (e.g., $R0 = \#_1/\#_2 \times \#_3$, where $\#_1$ to $\#_3$ can be any value).

3.2 Graph-based Mutation

The graph-based mutation is mainly used to change the connections in the DAG. For example, mutating the destination register of an instruction is equivalent to transmitting the results of a sub-graph from one receiver to another, and mutating a source register of an instruction is equivalent to activating another sub-graph while deactivating the existing connection. In the existing mutation operator, these variations are performed in a random manner. All the registers in the instructions have the same probability to be mutated. This might be not efficient enough.

To improve the efficiency of LGP mutation, two main ideas are introduced into the design of the graph-based mutation. Firstly, when mutating destination registers, graph-based mutation is more likely to select registers that are less utilised. This way, we can increase the utilisation of all registers and encourage more parallel computation (more intermediate results stored in different registers). As a result, we tend to obtain a "wider" DAG with more parallel branches. Secondly, when mutating source registers, graph-based mutation tends to select new source registers that are the result of more computations (its corresponding sub-graph is deeper). This way, we can increase the utilisation of large sub-graphs and reuse larger building blocks to refine the final output.

Algorithm 4: `GraphMutation`(p, θ_f, θ_c, θ_s)

Input: An LGP parent p, function mutation rate θ_f, constant mutation rate θ_c, source register mutation rate θ_s.
Output: An LGP offspring c.

$c = p$, $rnd \sim U(0, 1)$;
Randomly select an effective instruction with index i_m from c;
if $rnd < \theta_f$ **then**
  Mutate the function of $c[i_m]$ randomly;
else if $rnd < \theta_f + \theta_c$ *and* t *has a constant* **then**
  Mutate the constant of $c[i_m]$ randomly;
else if $rnd < \theta_f + \theta_c + \theta_s$ **then**
  `GraphMutateSourceReg`(c, i_m);
else
  `GraphMutateDestinReg`(c, i_m);
return c;

To implement these two ideas, we calculate the height of a sub-graph rooted at a register in the DAG. If the sub-graph of a certain destination register has a small height, it is likely that the register has not been effectively updated for a while, and it tends to be less utilised. In this case, the destination register has a higher probability to be selected by graph-based mutation to be a new destination register. On the other hand, if the sub-graph of a certain destination register has a large height, it means that the register stores the result that has gone through more computations, and thus is more likely to contain more information. In this case, the destination register is more likely to be selected by the graph-based mutation as a new source register.

The pseudo code of the graph-based mutation is shown in Algorithm 4. In addition to the parent p to be mutated, it has three parameters, i.e., the mutation rates for function θ_f, constant θ_c and source register θ_s. The mutation rate for the destination register can be derived by $\theta_d = 1 - \theta_f - \theta_c - \theta_s$. Specifically, the graph-based mutation operator randomly selects an effective instruction t from p to be mutated, following the mutation rates of the four different components. The mutation of a function and constant (if the instruction contains a constant) is the same as in traditional LGP micro mutation, that is, randomly sampling from the function set or the predefined constant domain. The mutation of the source and destination registers are conducted by the `GraphMutateSourceReg`(c, t) and `GraphMutateDestinReg`(c, t) methods, respectively.

Algorithms 5 and 6 show the mutation of the source and destination registers. They follow similar process. First, for each possible register, the height of the sub-graph before the mutated instruction (i.e., below the mutated instruction in the DAG) is calculated by the `SubGraphHeight`(p, i) method (which is a recursive process shown in Algorithm 7). Then, to encourage manipulating registers with a larger sub-graph height as source registers and using registers with smaller sub-graph height as destination registers, a new register is selected by the roulette wheel selection, where the probability of a register is set proportional

Algorithm 5: `GraphMutationSourceReg`(p, i_m)

Input: An LGP individual p, mutated instruction index i_m.

foreach *register* $r \in \mathbb{R}$ **do**
  Set the height of the sub-graph $h_r = 1$;
  Find the index of preceding instruction index $i' = \max\{j | j < i_m, des(p_j) = r\}$;
  if $i' \geq 0$ **then**
    Calculate the sub-graph height h_r = `SubGraphHeight`(p, i');
foreach *register* $r \in \mathbb{R}$ **do**
  Set the probability $\Pr(r) = \frac{h_r}{\sum_{r \in \mathbb{R}} h_r}$;
Select $r^* \in \mathbb{R}$ by roulette-wheel selection based on $\Pr(r)$;
Replace a source register of $p[i_m]$ with r^*;

Algorithm 6: `GraphMutationDestinReg`(p, i_m)

Input: An LGP individual p, mutated instruction index i_m.

foreach *register* $r \in \mathbb{R}$ **do**
  Set the height of the sub-graph $h_r = 1$;
  Find the index of preceding instruction index $i' = \max\{j | j < i_m, des(p_j) = r\}$;
  if $i' \geq 0$ **then**
    Calculate the sub-graph height h_r = `SubGraphHeight`(p, i');
  Set $score_r = \frac{1}{1+h_r}$;
foreach *register* $r \in \mathbb{R}$ **do**
  Set the probability $\Pr(r) = \frac{score_r}{\sum_{r \in \mathbb{R}} score_r}$;
Select $r^* \in \mathbb{R}$ by roulette-wheel selection based on $\Pr(r)$;
Replace the destination register of $p[i_m]$ with r^*;

Algorithm 7: `SubGraphHeight`(p, i)

Input: An LGP individual p, instruction index i.
Output: The height of the sub-graph rooted at $p[i]$.

$h = 0$;
foreach $src \in$ **src**($p[i]$) **do**
  for $k = i - 1 \rightarrow 0$ **do**
    if $des(p[k]) = r$ **then**
      $h = \max\{$`SubGraphHeight`$(p, k), h\}$;
      break;
return $1 + h$;

to the sub-graph height h_r for source register mutation, while proportional to $1/(1 + h_r)$ for destination register mutation.

Fig. 4 shows an example of graph-based mutation, which calculates the sub-graph and its height for every register of an LGP individual, where the last instruction (in the dashed box) is selected to be mutated. From the figure, we can see that $R0$ and $R4$ have a height of 1, since there is no previous instruction containing them as destination registers. Similarly, we can calculate the sub-graph height of all the registers. In this case, $R1$ and $R2$ will be more

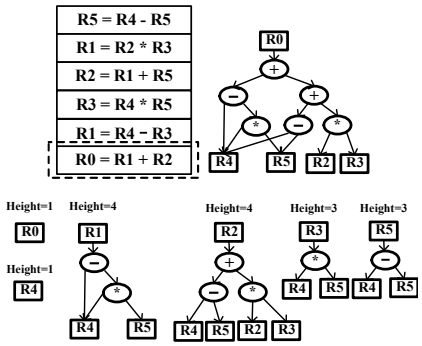

Figure 4: Examples of corresponding DAGs of all registers.

likely to be the new source register, while $R0$ and $R4$ are more likely to be the new destination register.

4 EXPERIMENTS: A CASE STUDY FOR DYNAMIC JOB SHOP SCHEDULING

4.1 Dynamic Job Shop Scheduling

This paper uses the DJSS problem to verify the performance of the proposed graph-based genetic operators. LGP is used as a kind of hyper-heuristic method to evolve dispatching rules for DJSS problems [5, 26]. DJSS is to process a set of jobs $\mathbf{J}$ by a set of machines $\mathbf{M}$. New jobs will come into the job shop over time. Each job $j \in \mathbf{J}$ consists of a sequence of operations $\mathbf{O} = (o_{j1}, ..., o_{jl_j})$, an arrival time α_j, a due date d_j and a weight ω_j. Each operation o_{ji} can be processed by a certain machine $\pi(o_{ji}) \in \mathbf{M}$ with a positive processing time $\sigma(o_{ji})$. o_{ji} cannot be processed until $o_{j,i-1}$ is completed. Each machine can process at most one operation at a time, and the processing cannot be interrupted once started.

In the experiments, we consider three objectives to be minimised, i.e., maximum tardiness for the whole simulation (T_{max}), mean tardiness (T_{mean}), and weighted mean tardiness (WT_{mean}). They are calculated as follows.

- $T_{max} = \max_{j \in J}(\max(c_j - d_j, 0))$
- $T_{mean} = \frac{1}{|J|} \sum_{j \in J}(\max(c_j - d_j, 0))$
- $WT_{mean} = \frac{1}{|J|} \sum_{j \in J}(\max(c_j - d_j, 0) \cdot \omega_j)$

where c_j is the completion time of job j.

The new jobs arrive dynamically, and the arrival time follows a Poisson process. We use a utilisation level parameter to control the frequency of the job arrivals, so that a higher utilisation level indicates that the shop floor is busier. In the experiments, we consider two different utilisation levels of 0.85 (i.e., about 85% of the time the machines are busy) and 0.95 (about 95% of the time the machines are busy). In combination of three objectives and two utilisation levels, we have $3 \times 2 = 6$ different DJSS scenarios. For each DJSS scenario, we create a set of training simulations and an unseen test simulation set. The compared GP algorithms will train a dispatching rule on the training set, and apply it to the test set to assess its test performance.

In the simulation, there are totally 10 machines in the job shop. Each job consists of 2 to 10 operations. The processing time of each operation is sampled from uniform distribution between 1 and 99. The jobs arrival to the job shop based on a Poisson process, with utilisation levels of 0.85 or 0.95. The due-date of job j is defined as 1.5 times the total processing time of j since it is arrived. Among all the jobs, 20% of them have a weight of 1, 60% of the jobs have a weight of 2, while the remaining 20% have a weight of 4. The evaluation of a dispatching rule focuses on the steady-state of the simulation, which means the first 1000 jobs in the simulation will not be counted. The performance metrics only care about the subsequent 5000 jobs after the 1000 warm-up jobs.

We compare the new Graph-based LGP (GLGP) with the standard TGP and LGP. Standard TGP, which has undergone a rapid development as a hyper-heuristic, is a popular baseline for many existing work for dynamic scheduling [27, 37, 38]. It uses its standard genetic operators to evolve dispatching rules in our experiments. For standard LGP, it uses the linear crossover and effective micro and macro mutation [1]. Each compared algorithm is run 30 times independently with different random seeds on each DJSS scenario.

4.2 Parameter Settings

The population size of the three GP methods is set to 256, and the maximal number of generations is set to 50. The other parameters of TGP are set as the recommended ones of existing literature [38]. The common parameters of the two LGP methods are set as follows. The maximal number of instructions in an individual is set to 50 ($\overline{l} = 50$) and the minimal number is set to 1 ($\underline{l} = 1$). The number of registers in LGP is set to 10 (e.g., $\mathbb{R} = \{R0, R1, \ldots, R9\}$. These registers are initialised by different job shop attributes. The rates of reproduction, macro mutation, micro mutation and crossover are set to 10%, 30%, 30% and 30%, respectively. In GLGP, the graph-based crossover takes place of crossover, and the micro-mutation is replaced by the graph-based mutation. Specifically, the parameters of macro mutation and the rates of different components in both the micro mutation and graph-based mutation are set as their recommended values by [4]. The parameters of graph-based crossover are set to $\overline{S} = 20$, $\Delta_S = 5$, $D_{cross} = 30$ respectively. To ensure linear crossover have a similar variation step size with graph-based crossover, the corresponding parameters of linear crossover are set the same as those of graph-based crossover. Specifically, the maximal segment length $L = 30$, maximal difference of selected segments $\Delta_L = 5$, and the maximal distance between crossover points $D_{cross} = 30$. L is set to 30 by assuming the average effective rate of an instruction segment is 60% to 70%. The terminal and function sets of the three GP methods follow the common settings of existing literature of GP for DJSS [18, 21]. All of these three GP methods conduct parent selection by a tournament selection with size 7, and the elitism rate of GP population is set to 10%.

5 RESULTS AND DISCUSSION

5.1 Test Performance

In this sub-section, the test performance of the three GP methods on the six DJSS scenarios are compared. For each DJSS scenario, we conduct the Wilocxon rank sum test with significance level of 0.05 to compare the test performance of the final dispatching rules obtained by the 30 runs.

Table 1: Mean (std.) of the test performance of LGP and GLGP.

Scenarios	TGP	LGP	GLGP
Tmax0.85	2000.25(61.31)	2041.16(84.61)	2011.13(59.66)(≈,≈)
Tmax0.95	4222.82(126.7)	4190.68(143.3)	4107.45(119.53)(+,+)
Tmean0.85	423.53(6.98)	422.53(4.64)	421.59(4.66)(≈,≈)
Tmean0.95	1132.32(15.81)	1129.79(13.24)	1129.74(12.53)(≈,≈)
WTmean0.85	749.65(40.22)	735.41(10.55)	734.51(8.45)(≈,≈)
WTmean0.95	1796.28(132.5)	1780.59(30.68)	1767.85(30.01)(≈,≈)

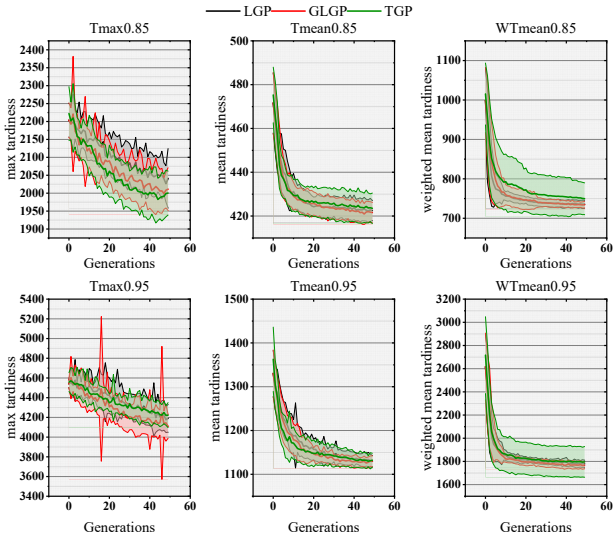

Figure 5: The convergence curves of test performance

Table 1 shows the mean and standard deviation of the test performance over 30 runs of TGP, LGP and GLGP. Under the "GLGP" column, the notations "+", "−", and "≈" indicate that the proposed GLGP performs statistically significantly better than, worse than, similar with TGP and LGP, respectively. From the table, we can see that GLGP significantly outperforms both TGP and LGP on the Tmax0.95 scenario. For the remaining scenarios, there is no statistical difference between the three compared algorithms. However, GLGP obtains slightly better mean values in most cases. Overall, we can see that GLGP shows better test performance than LGP on the six DJSS scenarios.

Fig. 5 shows the convergence curves of the test performance of the best dispatching rule evolved by the three GP methods per generation on the six DJSS scenarios. Specifically, the black and green curves are the convergence curves of LGP and TGP respectively, and the red curves are the ones of GLGP. The shaded area shows the standard deviation around the mean. From the figure, we can see that the convergence curves of GLGP in the Tmax0.85 and Tmax0.95 scenarios drop down faster and deeper than the ones of LGP. Though TGP converges faster than GLGP in Tmax0.85, it becomes inferior to GLGP when it comes to Tmax0.95, which is a more complicated scenario than Tmax0.85. In the two Tmean scenarios, though the three GP methods look quite similar in terms of the average performance, the curves of GLGP have a narrower bias range than the ones of both TGP and LGP. It implies a more stable performance of GLGP than LGP. In the two

Table 2: Mean (std.) of training fitness and training time

Scenarios	TGP	LGP	GLGP
Training fitness			
Tmax0.85	1753.05(54.64)	1784.21(312.83)	1743.44(301.58)(≈,≈)
Tmax0.95	3868.92(162.94)	3848.18(920.95)	3787.19(939.47)(≈,≈)
Tmean0.85	405.14(8.96)	404.17(48.35)	402.69(49.48)(≈,≈)
Tmean0.95	1081.87(47.32)	1084.47(266)	1087.66(281.49)(≈,≈)
WTmean0.85	718.92(16.18)	704.61(77.79)	702.61(78.1)(≈,≈)
WTmean0.95	1737.77(66.6)	1734.63(388.81)	1724.41(383.99)(≈,≈)
Training time (seconds)			
Tmax0.85	769.95(27.09)	680.22(17.63)	903.07(32.92)(−,−)
Tmax0.95	1671.41(63.23)	1511.75(39.43)	2027.68(68.54)(−,−)
Tmean0.85	677.93(17.48)	658.92(14.89)	783.27(18.16)(−,−)
Tmean0.95	1269.56(39.06)	1417.97(51.86)	1383.6(49.6)(−,+)
WTmean0.85	700.53(17.7)	628.29(14.87)	801.54(28.35)(−,−)
WTmean0.95	1423.05(49.91)	1432.01(68.91)	1651.19(67.72)(−,−)

WTmean scenarios, GLGP has a competitive performance with LGP and also more stable than TGP. Overall, the convergence curves validate the superior performance of GLGP on scenario Tmax0.95 and the competitive learning performance on Tmean and WTmean scenarios.

5.2 Training performance

This sub-section investigates the training performance of GLGP, including the average training fitness, training time, and the average program size in evolution. Table 2 compares the training fitness and time of GLGP with TGP and LGP. We can see that the training fitness of all these three GP methods are very similar. But GLGP still has a smaller mean value and standard deviation in terms of training fitness in most cases. On the other hand, GLGP has a much longer training time than TGP and LGP. It implies that, within the same number of simulations, the dispatching rules evolved by GLGP are likely to be more complex than those evolved by TGP and LGP.

Fig. 6 shows the curve of the average program size in the population at each generation of the compared algorithms. For TGP, the program size equals the number of nodes in the tree. To make a fair comparison on the program size, we set the program size of LGP and GLGP to the number of effective instructions multiplying by a factor of 2.0 [9]. As shown in the figure, though the average program size of GLGP in all the six scenarios is relatively small at the beginning, it climbs rapidly during evolution. Especially in later generations (>40), the program size of GLGP even exceeds that of TGP, and ends up with about 50 in most cases. The rapid increase of program size of GLGP leads to much longer decision times than TGP and LGP in simulation. Based on the superior test performance of GLGP, it is reasonable to claim that GLGP can construct more effective and sophisticated dispatching rules than TGP and LGP.

5.3 Component analysis

To further verify the effectiveness of the graph-based crossover and graph-based mutation, we conduct component analysis to compare LGP with the GLGP with the graph-based mutation only (GM-only) and with the graph-based crossover only (GC-only). Table. 3 shows the test performance and standard deviation of LGP, GM-only, and GC-only. The table shows that GM-only performs quite similar with LGP, though the mean value is usually better than that of LGP.

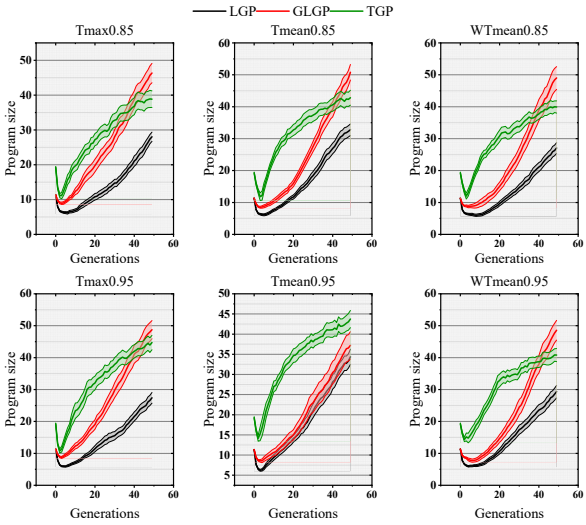

Figure 6: The average program size over generations.

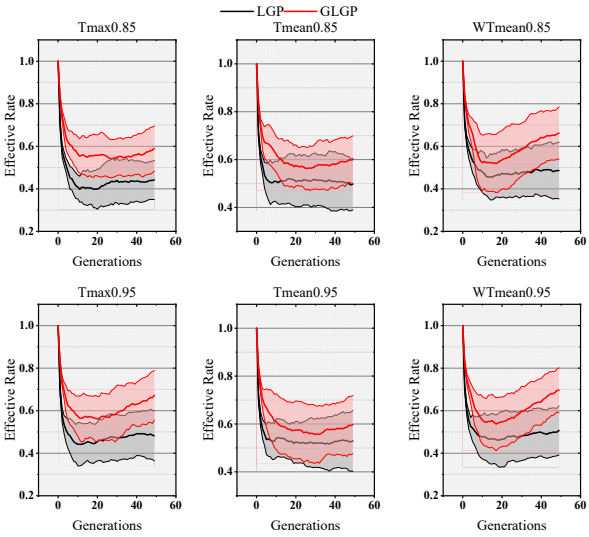

Figure 7: The curves of effective rate

Table 3: Average test performance of graph-base mutation and crossover

Scenarios	LGP	GM-only	GC-only
Tmax0.85	2041.16(84.61)	2017.69(60.47)≈	2000.24(54.21)+
Tmax0.95	4190.68(143.3)	4155.43(124.82)≈	4117.99(155.51)+
Tmean0.85	422.53(4.64)	421.48(4.00)≈	420.09(2.66)+
Tmean0.95	1129.79(13.24)	1132.45(12.43)≈	1130.47(13.27)≈
WTmean0.85	735.41(10.55)	736.3(6.97)≈	732.77(6.29)≈
WTmean0.95	1780.59(30.68)	1775.53(26.79)≈	1767.33(28.87)≈

GC-only, on the other hand, performs much better than LGP. It not only significantly outperforms LGP in three of the six scenarios, but also achieves a very competitive performance in the remaining three scenarios. Furthermore, when looking at the results of GLGP in Table 1, we can see that GC-only can even perform better than GLGP for most scenarios. In summary, we can see that the graph-based crossover plays a major role in the superior performance of GLGP. On the other hand, the graph-based mutation is not so effective, as including it together with the graph-based crossover might even worsen the performance of GLGP.

5.4 Effective Rate

The superior performance of the graph-based genetic operators stems from maintaining more useful building blocks (i.e., the topological structure of effective instructions). As a result, there should be more useful building blocks accumulating in the GLGP individuals. In other words, the proportion of effective instructions (i.e., effective rate) of GLGP should be larger than LGP.

To validate this hypothesis, we examine the effective rate of LGP and GLGP over generations during the evolutionary process in Fig. 7. From the figure, we can see that the red curves (i.e., the effective rate of GLGP) are always above the black curves (i.e., the effective rate of LGP) in all of the six scenarios. Besides, in some specific scenarios such as the two WTmean scenarios, the red curves grow up consistently while the black curves can only maintain at a certain level. Overall, we can see that GLGP successfully reaches a higher effective rate, suggesting that GLGP manages to construct more effective building blocks in the individuals than LGP.

6 CONCLUSIONS

This paper aims to improve the search effectiveness of LGP. This goal has been achieved by proposing a new graph-based LGP, which contains newly developed graph-based crossover and mutation operators. Specifically, the newly proposed crossover firstly converts the LGP individuals from instruction sequences to DAGs, and then swaps sub-graphs in the DAGs between two parents. The graph-based mutation aims to replace the destination registers with less utilised registers, and replace the source registers with more utilised registers. This way, it can utilise intermediate computation results (building blocks) better. We test the proposed GLGP on evolving dispatching rules for DJSS as a case study. The results show that GLGP has significantly better test performance than LGP with conventional linear genetic operators. Further analysis demonstrates that the graph-based crossover plays a major role in the advantage of GLGP, and an important reason of the advantage is the increased complexity and effective rate of the evolved programs.

This paper validates that protecting the topological structure of effective instructions is a simple and effective method to improve LGP performance. There are several future research directions worth to be studied. First, GLGP suffers from a more severe bloat effect (i.e., much larger program size) than other compared algorithms. Encouraging GLGP to evolve compact rules is necessary in further improving its performance. Second, the proposed graph-based mutation is not as effective as expected. A further analysis of the behaviour of graph-based mutation is needed.

REFERENCES

[1] W Banzhaf, M Brameier, M Stautner, and Klaus Weinert. 2003. Genetic Programming and Its Application in Machining Technology. *Advances in Computational Intelligence – Theory and Practice* (2003), 194–242.

[2] M. Brameier and W. Banzhaf. 2001. A comparison of linear genetic programming and neural networks in medical data mining. *IEEE Transactions on Evolutionary Computation* 5, 1 (2001), 17–26.

[3] Markus Brameier and Wolfgang Banzhaf. 2001. *Effective Linear Program Induction.* Technical Report. Technical Report CI-108/01, Collaborative Research Center 531, University of Dortmund.

[4] Markus Brameier and Wolfgang Banzhaf. 2007. *Linear genetic programming.* Vol. 53.

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

[6] Léo Françoso Dal Piccol Sotto and Vinícius Veloso de Melo. 2016. Studying bloat control and maintenance of effective code in linear genetic programming for symbolic regression. *Neurocomputing* 180 (2016), 79–93.

[7] Leo Frangoso Dal Piccol Sotto and Vinicius Veloso de Melo. 2017. A probabilistic linear genetic programming with stochastic context-free grammar for solving symbolic regression problems. *Proceedings of the Genetic and Evolutionary Computation Conference* (2017), 1017–1024.

[8] Léo Françoso Dal Piccol Sotto, Vinícius Veloso de Melo, and Márcio Porto Basgalupp. 2017. λ-LGP: an improved version of linear genetic programming evaluated in the Ant Trail problem. *Knowledge and Information Systems* 52, 2 (2017), 445–465.

[9] Carlton Downey. 2011. *Explorations in Parallel Linear Genetic Programming.* Ph. D. Dissertation.

[10] Carlton Downey and Mengjie Zhang. 2011. Caching for Parallel Linear Genetic Programming Categories and Subject Descriptors. In *Proceedings of the Annual Conference Companion on Genetic and Evolutionary Computation.* 201–202.

[11] Carlton Downey and Mengjie Zhang. 2011. Execution trace caching for Linear Genetic Programming. In *IEEE Congress of Evolutionary Computation,.* IEEE, 1186–1193.

[12] Carlton Downey, Mengjie Zhang, and Will N. Browne. 2010. New crossover operators in linear genetic programming for multiclass object classification. *Proceedings of the Annual Genetic and Evolutionary Computation Conference* (2010), 885–892.

[13] Carlton Downey, Mengjie Zhang, and Jing Liu. 2012. Parallel Linear Genetic Programming for multi-class classification. *Genetic Programming and Evolvable Machines* 13, 3 (2012), 275–304.

[14] Candida Ferreira. 2001. Gene Expression Programming: a New Adaptive Algorithm for Solving Problems. *Complex Systems* 13, 2 (2001), 87–129. arXiv:0102027 [cs]

[15] Christopher Fogelberg. 2005. *Linear Genetic Programming for Multi-class Classification Problems.* Ph. D. Dissertation. Victoria University of Wellington.

[16] Stefan Forstenlechner, David Fagan, Miguel Nicolau, and Michael O'Neill. 2017. A Grammar Design Pattern for Arbitrary Program Synthesis Problems in Genetic Programming. In *European Conference on Genetic Programming*, Vol. 10196. 262–277.

[17] M. I. Heywood and A. N. Zincir-Heywood. 2002. Dynamic page based crossover in linear genetic programming. *IEEE Transactions on Systems, Man, and Cybernetics, Part B: Cybernetics* 32, 3 (2002), 380–388.

[18] Zhixing Huang, Yi Mei, and Mengjie Zhang. 2021. Investigation of Linear Genetic Programming for Dynamic Job Shop Scheduling. In *Proceedings of the IEEE Symposium Series on Computational Intelligence.* IEEE.

[19] Wolfgang Kantschik and Wolfgang Banzhaf. 2001. Linear-tree GP and its comparison with other GP structures. In *European Conference on Genetic Programming*, Vol. 2038. 302–312.

[20] John R. Koza. 1992. *Genetic Programming : On the Programming of Computers By Means of Natural Selection.* Cambridge, MA, USA: MIT Press. 1–836 pages.

[21] Yi Mei, Su Nguyen, and Mengjie Zhang. 2017. Evolving time-invariant dispatching rules in job shop scheduling with genetic programming. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)* 10196 LNCS (2017), 147–163.

[22] Julian Miller, Dominic Job, and Vesselin Vassilev. 2000. Principles in the evolutionary design of digital circuits—Part I. *Genetic Programming and Evolvable Machines,* 1, 1-2 (2000), 7–35.

[23] Julian Miller, Dominic Job, and Vesselin Vassilev. 2000. Principles in the Evolutionary Design of Digital Circuits—Part II. *Genetic Programming and Evolvable Machines* 1, 3 (2000), 259–288.

[24] Jilian F. Miller. 1999. An empirical study of the efficiency of learning boolean functions using a Cartesian Genetic Programming approach. *Proceedings of the Genetic and Evolutionary Computation Conference* 2, December (1999), 1135–1142.

[25] Julian F. Miller, Dennis G. Wilson, and Sylvain Cussat-Blanc. 2019. Evolving Developmental Programs That Build Neural Networks for Solving Multiple Problems. In *Genetic Programming Theory and Practice XVI. Genetic and Evolutionary Computation.* Springer, Cham, 137–178.

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

[27] Su Nguyen, Mengjie Zhang, Mark Johnston, and Kay Chen Tan. 2013. 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 (2013), 621–639.

[28] Peter Nordin. 1994. A compiling genetic programming system that directly manipulates the machine code. *Advances in genetic programming* 1 (1994), 311–331.

[29] Peter Nordin. 1997. *Evolutionary program induction of binary machine code and its applications.* Ph. D. Dissertation. University of Dortmund.

[30] Vinicius Paulo L. Oliveira, Eduardo Faria de Souza, Claire Le Goues, and Celso G. Camilo-Junior. 2018. Improved representation and genetic operators for linear genetic programming for automated program repair. *Empirical Software Engineering* 23, 5 (2018), 2980–3006.

[31] Michael O'Neill and Conor Ryan. 2001. Grammatical evolution. *IEEE Transactions on Evolutionary Computation* 5, 4 (2001), 349–358.

[32] Michael O'Neill and Conor Ryan. 2003. *Grammatical Evolution: Evolutionary Automatic Programming in an Arbitrary Language.* Vol. 4. https://doi.org/10.1007/978-1-4615-0447-4

[33] P. C.D. Paris, E. C. Pedrino, and M. C. Nicoletti. 2015. Automatic learning of image filters using Cartesian genetic programming. *Integrated Computer-Aided Engineering* 22, 2 (2015), 135–151.

[34] Sergejs Provorovs and Arkady Borisov. 2012. Use of Linear Genetic Programming and Artificial Neural Network Methods to Solve Classification Task. *Scientific Journal of Riga Technical University. Computer Sciences* 45, 1 (2012), 133–139.

[35] Lee Spector. 2001. Autoconstructive Evolution: Push, PushGP, and Pushpop. In *Proceedings of the Genetic and Evolutionary Computation Conference.*

[36] Garnett Wilson and Wolfgang Banzhaf. 2008. A comparison of cartesian genetic programming and linear genetic programming. In *Proceedings of European Conference on Genetic Programming.* 182–193.

[37] Fangfang Zhang, Yi Mei, Su Nguyen, Kay Chen Tan, and Mengjie Zhang. 2021. Multitask Genetic Programming-Based Generative Hyperheuristics: A Case Study in Dynamic Scheduling. *IEEE Transactions on Cybernetics* (2021), 1–14. https://doi.org/10.1109/TCYB.2021.3065340

[38] Fangfang Zhang, Yi Mei, Su Nguyen, and Mengjie Zhang. 2021. Collaborative Multifidelity-Based Surrogate Models for Genetic Programming in Dynamic Flexible Job Shop Scheduling. *IEEE Transactions on Cybernetics* (2021), 1–15. https://doi.org/10.1109/TCYB.2021.3050141

[39] Fangfang Zhang, Yi Mei, Su Nguyen, and Mengjie Zhang. 2021. Correlation Coefficient-Based Recombinative Guidance for Genetic Programming Hyper-heuristics in Dynamic Flexible Job Shop Scheduling. *IEEE Transactions on Evolutionary Computation* 25, 3 (2021), 552–566.

[40] Fangfang Zhang, Yi Mei, Su Nguyen, and Mengjie Zhang. 2021. Evolving Scheduling Heuristics via Genetic Programming With Feature Selection in Dynamic Flexible Job-Shop Scheduling. *IEEE Transactions on Cybernetics* 51, 4 (2021), 1797–1811.

[41] Fangfang Zhang, Yi Mei, Su Nguyen, Mengjie Zhang, and Kay Chen Tan. 2021. Surrogate-Assisted Evolutionary Multitasking Genetic Programming for Dynamic Flexible Job Shop Scheduling. *IEEE Transactions on Evolutionary Computation* 25, 4 (2021), 651–665.

[42] Fangfang Zhang, Su Nguyen, Yi Mei, and Mengjie Zhang. 2021. Genetic Programming for Production Scheduling: An Evolutionary Learning Approach. In *Machine Learning: Foundations, Methodologies, and Applications.* Springer, Singapore, XXXIII+338. https://doi.org/10.1007/978-981-16-4859-5

[43] Jinghui Zhong, Liang Feng, and Yew-Soon Soon Ong. 2017. Gene Expression Programming: A Survey [Review Article]. *IEEE Computational Intelligence Magazine* 12, 3 (2017), 54–72.

[44] Jinghui Zhong, Yew-Soon Ong, and Wentong Cai. 2016. Self-Learning Gene Expression Programming. *IEEE Transactions on Evolutionary Computation* 20, 1 (2016), 65–80.