Grammar-Guided Linear Genetic Programming for Dynamic Job Shop 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

Dispatching rules are commonly used to make instant decisions in dynamic scheduling problems. Linear genetic programming (LGP) is one of the effective methods to design dispatching rules automatically. However, the effectiveness and efficiency of LGP methods are limited due to the large search space. Exploring the entire search space of programs is inefficient for LGP since a large number of programs might contain redundant blocks and might be inconsistent with domain knowledge, which would further limit the effectiveness of the produced LGP models. To improve the performance of LGP in dynamic job shop scheduling problems, this paper proposes a grammar-guided LGP to make LGP focus more on promising programs. Our dynamic job shop scheduling simulation results show that the proposed grammar-guided LGP has better training efficiency than basic LGP, and can produce solutions with good explanations. Further analyses show that grammar-guided LGP significantly improves the overall test effectiveness when the number of LGP registers increases.

CCS CONCEPTS

• **Computing methodologies → Planning and scheduling**.

KEYWORDS

Linear Genetic Programming, Grammar, Dynamic Job Shop Scheduling

ACM Reference Format:
Zhixing Huang, Yi Mei, Fangfang Zhang✉, Mengjie Zhang. 2023. Grammar-Guided Linear Genetic Programming for Dynamic Job Shop Scheduling. In *Genetic and Evolutionary Computation Conference (GECCO '23), July 15–19, 2023, Lisbon, Portugal.* ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/3583131.3590394

1 INTRODUCTION

Dynamic combinatorial optimization problems are common in our modern daily life and manufacturing industry [5, 18]. Dynamic job shop scheduling (DJSS) is a typical dynamic combinatorial optimization problem. Different from traditional job shop scheduling problems, DJSS has various dynamic events such as new coming jobs and machine breakdown, in the course of scheduling [37, 38]. Using genetic programming (GP) techniques to design dispatching rules is a promising method for solving DJSS problems [4, 24, 35, 36]. Dispatching rules make decisions by prioritizing operations at decision points, which can react to dynamic events quickly. Compared with manually designed rules, dispatching rules designed by GP are more effective in solving complicated scenarios and free human experts from tedious rule design [9, 17, 34].

Linear genetic programming (LGP) [3, 26] has shown very encouraging results on DJSS problems recently [13, 15]. However, existing LGP methods for designing DJSS dispatching rules have two main limitations. First, the search space is often too large to search effectively and efficiently, especially when the primitive set of LGP contains a large number of registers. Second, LGP-evolved programs are often long and complex, and cannot be easily understood by humans.

In this paper, we propose to use grammar rules to constrain LGP search space to address the above two issues. Grammar rules are used to describe the "legal" structures of all possible dispatching rules. Instead of searching in the whole search space defined by the primitive set, LGP with grammar rules searches dispatching rules within a much smaller search space restricted by grammar rules. With proper grammar rules, we can improve LGP training and test performance. In addition, it is easier for humans to make a detailed analysis of the produced dispatching rules based on the corresponding grammar rules.

Incorporating grammar rules with LGP is non-trivial. To the best of our knowledge, there is no existing literature investigating grammar-guided LGP. There are three main challenges in designing grammar-guided LGP. First, LGP individuals are lists of register-based instructions, which are greatly different from the representations of existing grammar-guided GP methods (e.g., trees and binary strings). It is necessary to design an LGP-based grammar system that allows humans to describe their constraints in grammar that can impose constraints on LGP evolution. Second, existing LGP genetic operators mainly produce offspring by manipulating instruction lists. However, it is hard to maintain the "legitimacy" of offspring if we arbitrarily modify the instruction list. Thus, we need grammar-guided genetic operators to maintain the grammar rules in breeding. Last but not least, we need to design proper grammar rules for DJSS problems to reduce the search space as much as possible without losing promising solutions.

Overall, this paper has three main contributions. First, we design an LGP-specific grammar system, extended from Backus-Naur Form (BNF). Different from basic BNF, the proposed grammar system considers LGP instructions as a basic element and flexibly defines different primitive sets for LGP instructions. Second, we propose

```
//R[2] = (max  x₁ R[0])
R[0] = (−  R[1] x₀)
R[2] = (× R[0] R[0])
R[1] = (min R[2]  x₁)
R[0] = (×  R[1] R[0])
```

Figure 1: A simple example of an LGP program.

three grammar-guided genetic operators to breed LGP offspring based on grammar rules. Third, we design a set of LGP-based grammar rules for solving DJSS problems. The proposed grammar rules improve LGP training efficiency, and significantly reduce the program size without sacrificing the test performance. Further analyses show that the proposed grammar rules significantly improve LGP performance when the number of registers increases.

2 LITERATURE REVIEW

2.1 Dynamic Job Shop Scheduling

This paper focuses on DJSS problems with new job arrival where new jobs come into the job shop during execution. Since the new coming jobs often disturb the original schedule in the job shop, we have to make instant reaction on the new coming jobs by dispatching rules. The job shop processes jobs with a finite set of machines. Each job consists of a list of operations, specifying the execution order of the operations. Each operation is processed by a certain machine with a positive processing time. To measure the performance of the job shop, we consider three common optimization objectives of the DJSS problem in this paper, which are maximum tardiness (T_{max}), mean tardiness (T_{mean}), and weighted mean tardiness (WT_{mean}), as shown as follows.

(1) $T_{max} = \max_{j \in \mathbb{J}}(\max(c(j) - d(j), 0))$

(2) $T_{mean} = \frac{\sum_{j \in \mathbb{J}}(\max(c(j)-d(j),0))}{|\mathbb{J}|}$

(3) $WT_{mean} = \frac{\sum_{j \in \mathbb{J}}(\max(c(j)-d(j),0) \cdot \omega(j))}{|\mathbb{J}|}$

where $c(j)$, $d(j)$, and $\omega(j)$ denote the completion time, due date, and weight of a job j. $\mathbb{J}$ is a set consisting of all jobs.

2.2 Linear Genetic Programming

An LGP individual is a list of register-based instructions, which are executed sequentially. An example of LGP programs is shown in Fig. 1. There are five instructions. In each instruction, a function accepts the two source registers on the right and stores the intermediate calculation results in the destination register on the left. The list of instructions in an LGP individual manipulates the same set of registers to represent a dispatching rule. Specifically, the first instruction in Fig. 1 is an ineffective instruction (i.e., intron, following a "//" notation) that does not contribute to the final output of the program. The output of the program is stored in the first register R[0] by default. Benefiting from the easy reusing of registers, LGP evolves very compact dispatching rules and is proven to be more effective in designing dispatching rules than tree-based GP in some DJSS scenarios [12, 15].

2.3 Grammars in Genetic Programming

Introducing grammar rules to GP is a powerful tool to constrain GP search space based on domain knowledge. In real-world practice, basic GP methods are sometimes too weak to produce legal rules since not all primitives can be put together (e.g., the "IF" operator must be followed by a boolean condition). There have been a lot of different implementations of grammar in GP [21]. For the sake of simplicity, we follow the terminologies used in the survey [21].

Strongly typed GP is one of the well-known GP methods with type constraints [23]. It was included as a special type of grammar-guided GP methods in [21]. Strongly typed GP explicitly considers multiple data types, each with corresponding functions and variables. It has been shown that grammar rules can accelerate GP learning speed and improve the quality of produced programs [19]. Many studies of strongly typed GP show that introducing domain knowledge by type constraints effectively improves the performance and interpretability of GP methods [2, 22].

To further restrict the program structure and the search space, context-free grammar GP (i.e., grammatically-based GP) [6, 33], grammar-guided GP [7, 8], and grammatical evolution [28, 29] apply context-free grammar (e.g., BNF) to define grammar rules. The grammar rules in these methods define the legal extension from a high-level concept to a low-level concept. These GP methods have been successfully applied to many domains such as rule induction algorithms and association rules [30, 31], program synthesis problems [10, 27, 32], and neural network architecture search [1, 20].

The existing studies fully show that introducing grammar rules into GP system is a powerful way to effectively design computer programs for various domains. The grammar rules restrict the search space and force GP methods to produce more explainable programs. However, the investigation of the grammar rules in synthesizing dispatching rules for DJSS problems is very limited. Nguyen et al. [25] used BNF to define the representation structures of dispatching rules. But the comparison in [25] might be largely affected by the different primitive sets defined by the grammars. Hunt et al. [16] applied strongly typed GP to improve the interpretability of the produced dispatching rules. But they sacrificed the effectiveness of the produced dispatching rules when improving the interpretability. Furthermore, all the investigations are based on tree-based GP. To the best of our knowledge, there is no investigation of using grammar rules to enhance LGP performance in DJSS.

3 PROPOSED METHOD

This paper proposes a new grammar-guided LGP (G2LGP) for DJSS problems. The innovations of G2LGP are twofold. First, we propose an LGP-specific grammar system (i.e., module context-free grammar (MCFG)) to introduce DJSS knowledge to the LGP system. Second, we propose a suite of grammar-guided genetic operators for G2LGP. The proposed grammar-guided genetic operators produce offspring based on the grammar rules defined by MCFG. The proposed MCFG and the grammar-guided genetic operators are introduced as follows.

3.1 Module Context-Free Grammar

The main idea of MCFG is to introduce LGP instructions as the basic elements in grammar definition. Specifically, MCFG defines

Table 1: Keywords in MCFG

`PROGRAM`	The starting point when producing programs based on grammar.
`defset`	Specify a set definition.
“{” and “}”	A pair of brackets that define the elements in a set.
::=	Define the possible derivations from a module.
::	Indicate a sequential execution order between two sub-modules.
“<” and “>”	A pair of brackets that define an LGP instruction module. The instruction module is a predefined module in MCFG.
$*$	Define the maximum repeating times of the precedent module. The default value of $*$ is 50.
|	Indicate the “OR” relationship in module derivations.
∼	Set assignment. Assign the primitive set on the right to the left.
\	Separate the assignment of different primitive sets.
;	Termination of a line of grammar.

```
defset FUNS {add,sub,mul,div};
defset REG {R0,R1,R2,R3,R4};
defset INPUT {x0, x1, x2};
defset CONSTANTS {1,2,3,4,5};

getSum(I\O) ::= <O\{sub}\O\O>::<O\{add}\I\O>*5;

getAverage(I\O) ::= getSum(I~I\O~O)::<O\{div}\O\{CONSTANTS}>;

getVariance(I\O) ::=getAverage(I~I\O~{R4})::(<{R1,R2,R3}\{sub}\{INPUT}\{R4}>
::<{R1,R2,R3}\{mul}\{R1,R2,R3}\{R1,R2,R3}>)*3::getAverage(I~{R1,R2,R3}\O~O);

PROGRAM ::= getSum(I~{INPUT}\O~{R0}) | getAverage(I~{INPUT}\O~{R0})
| getVariance(I~{INPUT}\O~{R0});
```

Figure 2: An example of MCFG. “add, sub, mul, div” denote the four basic arithmetic operations.

LGP instructions by *instruction modules*, specified by a pair of “<” and “>”. The instruction modules compose higher-level constraints and finally form a set of constraints on a program. An instruction module defines the feasible primitive sets of the destination register, the function, and the two source registers of an LGP instruction. By giving an instruction module different primitive sets on different positions, MCFG imposes different constraints on LGP programs.

To fulfill the main idea, MCFG has two main extensions from BNF. First, MCFG allows users to define primitive sets and supports set operations like union and intersection on the primitive sets. Second, MCFG allows *derivation rules* to accept primitive sets as input arguments. The derivation rule specifies the derivations from a specific module to sub-modules and passes the primitive sets to sub-modules, and finally to instruction modules.

Table 1 illustrates the keywords in MCFG and their corresponding meaning, and Fig. 2 is an example of MCFG that restricts the search space of synthesizing an LGP program to perform three basic statistical metrics (i.e., sum, average, and variance).[1] In the beginning, Fig. 2 defines four primitive sets by a keyword “`defset`”. For instance, the first line defines the function set including addition, subtraction, multiplication, and division, named “`FUNS`”.

Each derivation rule follows a template

$$\texttt{module name (input arguments)} ::= \texttt{derivation}_1 | \cdots | \texttt{derivation}_n;$$

The module name is defined by users and must be unique. Each derivation consists of at least one sub-module and specifies the argument assignment to the sub-modules. Sub-modules can be instruction modules or the modules defined before.

[1]Note that the example here mainly defines the key primitives in produced programs in order to reduce the search space. LGP still needs to search for a list of instructions to perform the three metrics.

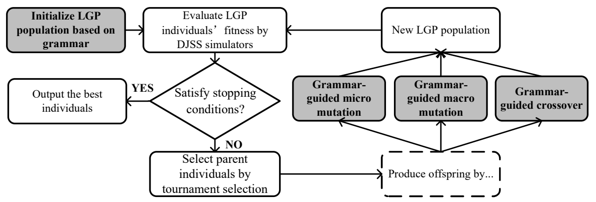

Initialize LGP population based on grammar
Evaluate LGP individuals’ fitness by DJSS simulators
New LGP population
Output the best individuals
YES
Satisfy stopping conditions?
NO
Grammar-guided micro mutation
Grammar-guided macro mutation
Grammar-guided crossover
Select parent individuals by tournament selection
Produce offspring by...

Figure 3: The evolutionary framework of G2LGP. The dark steps are the newly proposed steps.

There are four derivation rules in Fig. 2. The first three derivation rules respectively define the grammar for the three statistical metrics. The last derivation rule defines that the whole LGP program is one of the three metrics. Here, we take `getSum` as an illustrative example. The `getSum` derivation rule has two input arguments “`I`” and “`O`”. The module `getSum` can derive to two instruction modules. The first instruction module maximally repeats 1 time, and the second instruction module maximally repeats 5 times (i.e., “$*5$”). In the second instruction module, the four arguments separated by “\” in the instruction module sequentially define the possible primitives of 1) destination register as “O” from the parent module `getSum`, 2) function as “add”, 3) the first source register as “I” from the parent module `getSum`, and 4) the second source register as “O”.

$$<\ \underset{\substack{\text{destination}\\ \text{register set}}}{\texttt{O}}\ \backslash\ \underset{\substack{\text{function}\\ \text{set}}}{\texttt{\{add\}}}\ \backslash\ \underset{\substack{\text{1st source}\\ \text{register set}}}{\texttt{I}}\ \backslash\ \underset{\substack{\text{2nd source}\\ \text{register set}}}{\texttt{O}}\ >\ \underset{\substack{\text{maximum}\\ \text{repeating time}}}{*5}$$

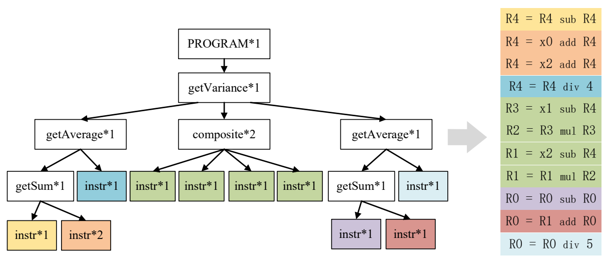
In short, the two instruction modules derived from `getSum` first clear the output register “O”, then use addition to sum up input primitives “I”, and finally store the results into the output register.

Furthermore, MCFG supports using brackets “(” and “)” to implicitly define *composite modules*. For instance, in Fig. 2, the module `getVariance` defines the two instruction modules, `<{R0,R1,R2}\{sub}\{INPUT}\{R3}>` and `<{R0,R1,R2}\{mul}\{R0,R1,R2}\{R0,R1,R2}>`, as a composite module by a pair of “(” and “)” and that the composite module maximally repeats three times.

3.2 Grammar-guided LGP

3.2.1 Framework. The evolutionary framework of G2LGP follows the evolutionary framework of basic LGP for solving DJSS problems [14]. But with grammar, G2LGP has a different program initialization process and applies a suite of grammar-based genetic operators to produce offspring, as shown in Fig. 3. In initialization, G2LGP constructs a population of LGP individuals based on predefined grammar. In offspring breeding, G2LGP applies grammar-guided micro mutation, grammar-guided macro mutation, and grammar-guided crossover to produce offspring. These new components are introduced in detail as follows.

3.2.2 LGP program initialization. G2LGP constructs LGP programs in two steps. First, G2LGP constructs a *derivation tree* based on the grammar. Second, G2LGP generates an LGP program based on the derivation tree. For the sake of simplicity, we use the grammar in Fig. 2 to make a detailed demonstration. Based on Fig. 2, we construct a derivation tree as shown in Fig. 4.



```
R4 = R4 sub R4
R4 = x0 add R4
R4 = x2 add R4
R4 = R4 div 4
R3 = x1 sub R4
R2 = R3 mul R3
R1 = x2 sub R4
R1 = R1 mul R2
R0 = R0 sub R0
R0 = R1 add R0
R0 = R0 div 5
```

Figure 4: An example of a derivation tree and its corresponding LGP program. "instr" is the abbreviation of instruction modules.

In Fig. 4, a derivation tree starts from the starting point `PROGRAM` and selects `getVariance` to derive. Based on the derivation rule, the `getVariance` is fulfilled as

`getVariance({INPUT},{R0})`.

`getVariance` derives to three sub-modules, i.e., two `getAverages` and a composite module repeating two times. By recursively deriving modules, all modules derive to instruction modules (i.e., leaf nodes in the derivation tree).

We construct the program based on the instruction modules. The corresponding example program based on the derivation tree is shown on the right of Fig. 4. Each instruction in the program is related to a derivation tree leaf node with the same color. We can see that the example program shows some potential in synthesizing a variance calculation function, although it is not a correct program for calculating variance. For example, the fifth and sixth instructions calculate the deviation between input `x1` and the (suppose-to-be) mean value, and the fourth and the last instruction respectively divide a constant value, trying to get averages.

3.2.3 *Grammar-guided micro mutation.*

The grammar-guided micro mutation changes one of the primitives in one of the instructions. Specifically, the grammar-guided micro mutation accepts one parent and produces one offspring. It first randomly selects one of the instructions in the program and selects one of the four components (i.e., destination register, function, and two source registers). Based on the feasible primitive set at that component, the grammar-guided micro mutation mutates the primitive.

Take the program in Fig. 4 as an example. If the grammar-guided micro mutation wants to change the first source register of the second instruction, it checks the related instruction module and finds that the instruction module is fulfilled as

`<{R4}\{add}\{x0,x1,x2}\{R4}>`.

Thus, the `x0` is likely changed to `x1` or `x2`.

3.2.4 *Grammar-guided macro mutation.*

Grammar-guided macro mutation has three main operations, i.e., add instructions, remove instructions, and replace instructions. All the three operations change the program by 1) changing the derivation tree and 2) constructing the sub-program based on the new derivation tree. Suppose that an LGP individual $\mathbf{P}$ consists of a derivation tree $\mathbf{T_P}$ and a list of instructions $\mathbf{I_P}$. Each derivation tree node m has a list of child nodes (i.e., fulfilled derivations), denoted as $\mathbf{D}(m)$. $\mathbf{D}(m)$ cannot be empty, except if m is an instruction module (i.e., leaf nodes in derivation trees). The maximum repeating time of m is denoted as $\overline{|\mathbf{D}(m)|}$. The pseudo-code of the grammar-guided macro mutation is shown in Alg. 1[2].

Algorithm 1: `Grammar-guided macro mutation`

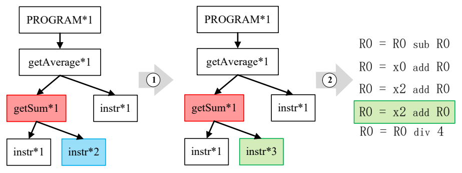
```
   Input: An LGP individual $\mathbf{P}$, add instruction rate $\theta_a$, growing node rate $\theta_g$
   Output: An offspring $\mathbf{P}'$
1  $\mathbf{P}' \leftarrow \mathbf{P}$;
2  for $j \leftarrow 1$ to 50 do
3      $\mathbf{C} \leftarrow$ clone $\mathbf{P}$;
4      if rand(0, 1) $< \theta_g$ then
5          $m \leftarrow$ randomly pick a tree node from $\mathbf{T_C}$, s.t. $\overline{|\mathbf{D}(m)|} > 1$;
6          $d \leftarrow$ randomly pick a derivation from $\mathbf{D}(m)$;
7          if (rand(0, 1) $< \theta_a$ and $|\mathbf{D}(m)| < \overline{|\mathbf{D}(m)|}$) or $|\mathbf{D}(m)| = 1$ then
8              Grow a new derivation $d'$ from $m$ and recursively derive $d'$ if $d'$ does
                 not only contains instruction modules;
9              Add $d'$ to $\mathbf{D}(m)$ right before $d$;
10             $\mathbf{I}' \leftarrow$ construct instructions based on $d'$ and its derivation;
11             Add $\mathbf{I}'$ to $\mathbf{I_C}$, right before the first instruction derived from $d$;
12         else if $|\mathbf{D}(m)| = \overline{|\mathbf{D}(m)|}$ or $|\mathbf{D}(m)| > 1$ then
13             Remove the instructions derived from $d$ from $\mathbf{I_C}$;
14             Remove $d$ from $\mathbf{D}(m)$;
15     else
16         $m \leftarrow$ randomly pick a tree node from $\mathbf{T_C}$;
17         $d \leftarrow$ randomly pick a derivation from $\mathbf{D}(m)$;
18         Grow a new derivation $d'$ from $m$ and recursively derive $d'$ if $d'$ does not
             only contains instruction modules;
19         Add $d'$ to $\mathbf{D}(m)$ right before $d$;
20         $\mathbf{I}' \leftarrow$ construct instructions based on $d'$ and its derivation;
21         Add $\mathbf{I}'$ to $\mathbf{I_C}$, right before the first instruction derived from $d$;
22         Remove the instructions derived from $d$ from $\mathbf{I_C}$;
23         Remove $d$ from $\mathbf{D}(m)$;
24     for $i' \leftarrow$ reversely read from $\mathbf{I}'$ do
25         if $i'$ is an intron in $\mathbf{C}$ then
26             Alter the destination register of $i'$ and try to make $i'$ effective.
27     if $|\mathbf{I_C}| \in [\underline{|\mathbf{I_C}|}, \overline{|\mathbf{I_C}|}]$ then
28         $\mathbf{P}' \leftarrow \mathbf{C}$;
29         break;
30 Return $\mathbf{P}'$;
```

When grammar-guided macro mutation accepts an individual $\mathbf{P}$, the macro mutation randomly selects one of the three operations (i.e., adding, or removing, or replacing instructions) to produce offspring. If grammar-guided macro mutation decides to add or remove instructions, it first randomly selects a derivation tree node m that has a maximum repeating time larger than 1 and a derivation d from the child nodes of m (lines 5 and 6). Selecting a derivation tree node that can repeat more than one time encourages grammar-guided genetic operators to produce offspring by changing the total number of instructions.

If the mutation operator decides to add instructions, it grows a new derivation d' from m and recursively derives d' until the sub-derivation tree under d' reaches instruction modules[3]. d' is added to $\mathbf{D}(m)$ right before d. The mutation operator constructs the instruction list $\mathbf{I}'$ based on Section 3.2.2 and insert $\mathbf{I}'$ to $\mathbf{I_P}$ right before the first instruction derived from d. If grammar-guided macro mutation decides to remove instructions, the mutation operator

[2] `rand`(a, b) returns a random floating point number between a and b. `randint`(a, b) returns a random integer in $[a, b]$. **s.t.** is the abbreviation of "so that", imposing constraints on precedent statement. $|\cdot|$ denotes the cardinality of a list or a set. $\overline{a}$ and $\underline{a}$ denote the maximum and minimum value of a respectively.

[3] if m is a composite module which has no module name in the predefined grammar, the mutation operator simply clones one of the existing child nodes of m as d'.



```
R0 = R0 sub R0
R0 = x0 add R0
R0 = x2 add R0
R0 = x2 add R0
R0 = R0 div 4
```

Figure 5: An example of adding instructions by the grammar-guided macro mutation. The macro mutation operator first modifies the derivation tree and second mutates the instruction list.

removes all the instructions derived from d and removes d from $\mathbf{D}(m)$ (lines 12-14).

If grammar-guided macro mutation decides to replace instructions, it simultaneously performs adding and removing instructions within one breeding (lines 16-23). Allowing the macro mutation to replace instructions enables LGP to jump out local optimum. For example, in Fig. 2, LGP programs can turn to search `getAverage` from searching `getSum` by replacing the derivation tree under `PROGRAM`.

To mimic the effective mutation in basic LGP, the grammar-guided macro mutation checks the effectiveness of all newly generated instructions and tries to make them effective if their corresponding instruction module contains effective destination registers (lines 24-26). To ensure that the program size of the newly generated offspring is in the predefined range (i.e., $[\underline{|\mathbf{I}_C|}, \overline{|\mathbf{I}_C|}]$), we iterate the mutation process for multiple times (e.g., 50 times) until the program size of the offspring satisfies the predefined range (lines 27-29).

Fig. 5 is an example of grammar-guided macro mutation. First, grammar-guided macro mutation selects `getSum` derivation tree node (i.e., the red node in Fig. 5) and its second derivation (i.e., the blue child nodes). To add instructions, grammar-guided macro mutation derives the `getSum` node based on the grammar. In this example, grammar-guided macro mutation grows one new instruction module and inserts them into the derivation list of `getSum` (i.e., the green node has three instruction modules). Then, based on the fulfilled instruction modules, the macro mutation operator constructs a new instruction and inserts it into the program, as shown on the right of Fig. 5.

3.2.5 Grammar-guided crossover. The main idea of the grammar-guided crossover is to swap the instructions belonging to the same type of derivation nodes. The pseudo-code of grammar-guided crossover is shown in Alg. 2. To enable grammar-guided crossover to exchange more genetic materials, the grammar-guided crossover selects more than one crossover point for each breeding.

First, grammar-guided crossover selects a number of pairs of crossover points with the same type from the derivation trees of the two LGP parents (i.e., $m_r = m_d$ at line 12). The = means that 1)m_r and m_d have the same module name, 2)m_r and m_d have the same maximum repeating number $\overline{|\mathbf{D}(m)|}$, and 3) m_r and m_d have the same input argument list (e.g., predefined argument values). For example, the two `getAverage` in the derivation rule of `getVariance` in Fig. 2 are not the same type. They have the same module name

Algorithm 2: `Grammar-guided crossover`

```
   Input: Two LGP individuals $[\mathbf{P}_0, \mathbf{P}_1]$, the maximum number of crossover points $N$,
          growing node rate $\theta_g$
   Output: Two LGP offspring $[\mathbf{P}'_0, \mathbf{P}'_1]$
1  for $i \leftarrow 0$ to 1 do
2      $\mathbf{P}_d \leftarrow \mathbf{P}_{i\%2}, \mathbf{P}_r \leftarrow \mathbf{P}_{(i+1)\%2}$;
3      $\mathbf{L}_d \leftarrow \emptyset, \mathbf{L}_r \leftarrow \emptyset$;
4      $n \leftarrow$ randint(1,N);
5      for $j \leftarrow 1$ to 50 do
6          $m_d, m_r \leftarrow null$;
7          if rand(0,1) $< \theta_g$ then
8              $m_d \leftarrow$ randomly pick a tree node from $\mathbf{T}_{\mathbf{P}_d}$, s.t. $\overline{|\mathbf{D}(m_d)|} > 1$;
9          else
10             $m_d \leftarrow$ randomly pick a tree node from $\mathbf{T}_{\mathbf{P}_d}$;
11         if $m_d \neq null$ then
12             $m_r \leftarrow$ randomly pick a tree node from $\mathbf{T}_{\mathbf{P}_r}$, s.t. $m_r = m_d$;
13         if $m_r \neq null$ then
14             if $m_d$ does not overlap with any elements in $\mathbf{L}_d$ and $m_r$ does not overlap
                 with any elements in $\mathbf{L}_r$ then
15                 Add $m_d$ into $\mathbf{L}_d$;
16                 Add $m_r$ into $\mathbf{L}_r$;
17                 if $|\mathbf{L}_d| \geq n$ then
18                     break;
19     for $j \leftarrow 0$ to $|\mathbf{L}_d| - 1$ do
20         $m_d \leftarrow \mathbf{L}_d[j], m_r \leftarrow \mathbf{L}_r[j]$;
21         $[\mathbf{d}_d, \mathbf{I}_d] \leftarrow$ randomly pick a sub-list of child nodes and their corresponding
             instructions from $\mathbf{D}(m_d)$;
22         $[\mathbf{d}_r, \mathbf{I}_r] \leftarrow$ randomly pick a sub-list of child nodes and their corresponding
             instructions from $\mathbf{D}(m_r)$;
23         if $|I_{\mathbf{P}_r}| - |\mathbf{I}_r| + |\mathbf{I}_d| \in [\underline{|\mathbf{I}_C|}, \overline{|\mathbf{I}_C|}]$ then
24             Replace $\mathbf{d}_r$ and $\mathbf{I}_r$ with the clone of $\mathbf{d}_d$ and $\mathbf{I}_d$ respectively.
25             Update argument assignment to $\mathbf{d}_r$ and update $\mathbf{I}_r$ if the primitives are
                 not consistent with the derivation tree node.
26     $\mathbf{P}'_i \leftarrow \mathbf{P}_r$
27 Return $[\mathbf{P}'_0, \mathbf{P}'_1]$;
```

and the same maximum repeating time but different input argument lists. If the selected crossover points do not overlap with the existing crossover points in the list, grammar-guided crossover collects the pair of crossover points into the lists $\mathbf{L}_d$ and $\mathbf{L}_r$ respectively (lines 14 to 18). The term "overlap" implies that $\mathcal{A}$ overlaps with $\mathcal{B}$ if and only if $\mathcal{A}$ is $\mathcal{B}$ or $\mathcal{A}$ is in the sub-tree of $\mathcal{B}$ or $\mathcal{B}$ is in the sub-tree of $\mathcal{A}$. Since we might not sample a pair of consistent crossover points in an iteration, we simply iterate the sampling processing multiple times (e.g., 50 times) until we collect enough number of crossover points (lines 5-18).

Second, for each pair of crossover points, grammar-guided crossover picks a sub-list of child nodes from $\mathbf{D}(m_d)$ and $\mathbf{D}(m_r)$ respectively (lines 20-22). If the program size of the offspring is consistent with the predefined minimum and maximum LGP program size, the crossover operator exchanges the selected derivation tree nodes and corresponding instructions. Because the input arguments might have different primitive sets from different roots after swapping, the crossover operator updates the instructions accordingly in the receiver individual.

Fig. 6 is an example of the grammar-guided crossover. First, the derivation tree selects two derivation tree nodes with the same type (e.g., `getSum` in Fig. 6) and their related instructions from the two parent individuals. Second, the crossover operator replaces the sub-derivation tree and the instruction list to get offspring.

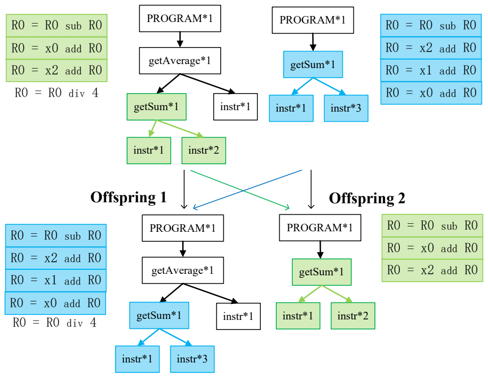


Figure 6: An example of the grammar-guided crossover. The blue and green derivation tree nodes and instructions are swapped by the crossover operator.

4 EXPERIMENT DESIGNS

The simulation settings of the DJSS problems in this paper follow the ones in [12]. Our experiments consider 6 different scenarios, denoted by "⟨Objective, utilization level⟩". Utilization level indicates how busy a job shop is. A higher value of utilization level indicates a busier job shop. Each scenario has a job shop with 10 machines and processes 5000 jobs in its steady state. Specifically, the six scenarios are $\langle T_{max}, 0.85\rangle$, $\langle T_{max}, 0.95\rangle$, $\langle T_{mean}, 0.85\rangle$, $\langle T_{mean}, 0.95\rangle$, $\langle WT_{mean}, 0.85\rangle$, and $\langle WT_{mean}, 0.95\rangle$. For each scenario, we set up a set of training DJSS problem instances, each for a generation, and a set of 50 unseen test DJSS problem instances. All the compared GP methods search a dispatching rule based on the training instances and test the performance of the produced dispatching rule on test instances.

Our experiments mainly compare the performance of LGP with and without grammar (i.e., basic LGP and G2LGP). Specifically, G2LGP replaces the basic micro and macro mutation and linear crossover with grammar-guided micro and macro mutation and grammar-guided crossover. The growing node rates of grammar-guided macro mutation and crossover are both set as 0.8, and the maximum number of crossover points of grammar-guided crossover is set as 5 based on our prior experiments. The compared methods use the same set of terminals, defined as Table 2, and the same set of functions $\{+, -, \times, \div, \max, \min\}$. The other parameter settings are the same as in the existing literature [12].

5 PROPOSED GRAMMARS IN DJSS

Based on the terminal and function sets, we design the grammar rules for LGP in solving DJSS problems, as shown in Fig. 7. The main idea of these grammar rules is to reduce LGP search space by 1) highlighting useful input features and 2) restricting the number of available registers in different parts of LGP programs. We first define six sets of primitives, `FUNS`, `posINPUT`, `negINPUT`, `INPUT`, `REG`, and `HREG`, specifying the functions, different sets of input features, register set, and a subset of registers. The first derivation rule encourages the primitives that imply high priorities in large values (e.g., larger "W" implies a more important job) to follow "sub" and "div". It also encourages the primitives that imply high priorities in small values (e.g., smaller "PT" implies an easier operation) to be the first source register. The module `bias` also implements the similar idea. To avoid excluding effective solutions because of poorly defined grammars, `arith` contains a non-constrained derivation `<O\{FUNS}\R+I\R+I>`. `connectbias` aggregates registers to a specific subset of registers in order to reduce the number of available registers. It is reused in `connectbiasSet`. `compress` constructs subparts of dispatching rules (by `arith`) and store the intermediate results into particular registers by `compressConnect`. By sequentially applying `compress` two times with different register sets, LGP programs consider smaller search spaces in the second part of the programs.

Table 2: The terminal set

Notation	Description
PT	Processing time of an operation in a job
NPT	Processing time of the next operation in a job
WINQ	Total processing time of operations in the buffer of a machine which is the corresponding machine of the next operation in a job
WKR	Total remaining processing time of a job
rFDD	Difference between the expected due date of an operation and the system time
OWT	Waiting time of an operation
NOR	Number of remaining operations of a job
NINQ	Number of operations in the buffer of a machine which is the corresponding machine of the next operation in a job
W	Weight of a job
rDD	Difference between the expected due date of a job and the system time
NWT	Waiting time of the next to-be-ready machine
TIS	Difference between system time and the arrival time of a job
SL	Slack: difference between the expected due date and the sum of the system time and WKR
NIQ	Number of operations in the buffer of a machine
WIQ	Total processing time of operations in the buffer of a machine
MWT	Waiting time of a machine

```
defset FUNS {add,sub,mul,div,max,min};
defset posINPUT {PT,NPT,WINQ,NINQ,rFDD,rDD,SL};
defset negINPUT {W,OWT,NWT,TIS,WKR,NOR}
defset INPUT {posINPUT,negINPUT,WIQ,MWT,NIQ};
defset REG {R0,R1,R2,R3,R4,R5,R6,R7};
defset HREG {R0,R1,R2,R3};

arith(I\O\R) ::= <O\{sub,div,max,min}\R\{negINPUT}+R>
|<O\{FUNS}\{posINPUT}+R\R>|<O\{FUNS}\R+I\R+I>;

bias(O) ::= <O\{sub,div}\O\{negINPUT}>|<O\{add,mul,sub,div}\{posINPUT}\O>;

connectbias(I\O) ::= <O\{FUNS}\I\I>::bias(O~O);

connectbiasSet(O\R) ::= connectbias(I~R\O~{R0})|connectbias(I~R\O~{R1})
|connectbias(I~R\O~{R2})|connectbias(I~R\O~{R3});

compressConnect(R) ::= connectbiasSet(O~{HREG}\R~R)*3;

compress(R) ::= arith(I~{INPUT}+R\O~R\R~R)*::compressConnect(R~R);

PROGRAM ::= compress(R~{REG})::compress(R~{HREG});
```

Figure 7: The grammar rules for LGP in solving DJSS.

6 EXPERIMENT RESULTS

6.1 Test performance

We conduct a Friedman test with a significance level of 0.05 on the test performance. The p-value of the Friedman test is 0.414, which implies that the overall performances of the two compared methods are not significantly different. However, the mean rank of G2LGP in the six scenarios is 1.33, which is better than the mean rank

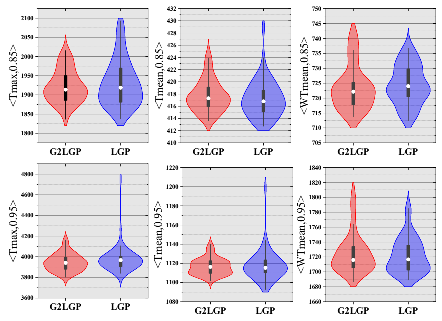


Figure 8: Test performance of the G2LGP and basic LGP on the six scenarios.

of basic LGP (i.e., 1.67). This shows G2LGP achieves competitive performance with basic LGP.

To further investigate the test performance of basic LGP and G2LGP on the six scenarios, we show the test performance by box plot in Fig. 8. Fig. 8 shows that in all six scenarios, the test performances of G2LGP and basic LGP are similar. But in some scenarios such as $\langle T_{mean}, 0.85\rangle$, $\langle T_{max}, 0.95\rangle$, and $\langle T_{mean}, 0.95\rangle$, G2LGP can have more stable performance than basic LGP (i.e., we can see some outliers in the results of basic LGP).

6.2 Training performance

6.2.1 *Training efficiency.*

Grammar rules improve the training efficiency of LGP methods, especially at the early stage of evolution. Fig. 9 shows the training performance of the compared methods[4]. The curves of G2LGP drop down faster than the ones of basic LGP in $\langle T_{max}, 0.85\rangle$, $\langle WT_{mean}, 0.85\rangle$, and $\langle WT_{mean}, 0.95\rangle$, which confirms a faster training efficiency of G2LGP. In contrast, after 10000 evaluations, G2LGP performs similarly to basic LGP. The high training efficiency at the beginning of evolution and the similar performance at the end of evolution imply that the grammar rules effectively reduce the search space. This helps LGP find more useful solutions at the early stage.

6.2.2 *Program size.*

To further understand the impact caused by the proposed grammars, this section compares the average program size and the average effective program size of the LGP population over evolution. Here, we denote the number of (effective) instructions as (effective) program size and select three scenarios with a 0.95 utilization level as examples. We can see that G2LGP averagely has a smaller program size than basic LGP in almost the entire training stage. However, the curves of the average effective program size of G2LGP perform similarly to the ones of basic LGP. It implies that G2LGP averagely has a more compact representation (i.e., less ineffective instructions in the programs) than basic LGP.

6.3 Example programs

Based on the proposed grammar, we compare the interpretability of the produced programs from G2LGP and basic LGP. Here, we

[4]We represent the training performance by the test performance of the best individuals at every generation because the instance rotation in GP training makes training convergence curves very fluctuate [11].

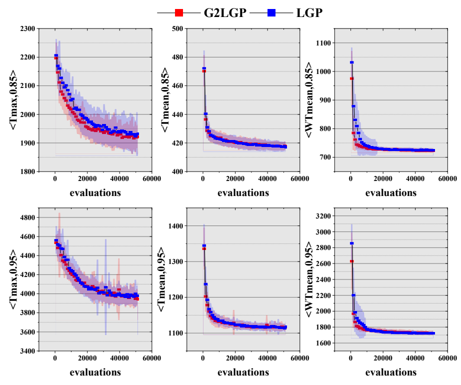


Figure 9: The curves of the training performance of G2LGP and basic LGP.

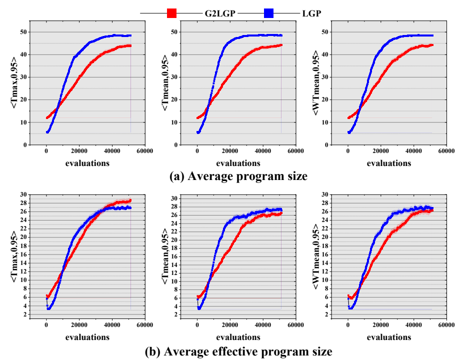
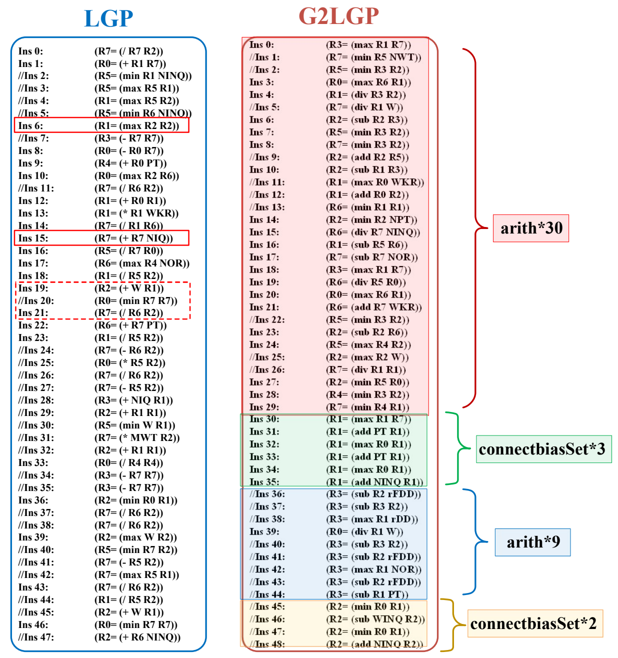


Figure 10: The average program size (± standard deviation) and average effective program size (± standard deviation) over the whole population during evolution.

randomly select two G2LGP-produced and LGP-produced programs from an independent run in $\langle WT_{mean}, 0.95\rangle$, as shown in Fig. 11. The two produced programs from basic LGP and G2LGP have a very similar program size (48 and 49 instructions), but G2LGP has more effective instructions (27 effective instructions) than basic LGP (22 effective instructions). If we look at the effective instructions, we find that the produced program from basic LGP contains some redundant instructions (e.g., Ins 6: (R1=(max R2 R2)) is redundant if the program can use R2 directly) and hard-to-explain instructions (e.g., Ins 15: (R7=(+ R7 NIQ)) considers the machine-related feature NIQ, but NIQ is helpless for prioritizing the operations in the same machine). Besides, the basic LGP program uses complicated structures to do simple things. For example, Ins 19 uses "W" together with addition, and Ins 21 divides R2 to make the final program output negatively correlated with "W". However, complicated structures are easy to be destructed in LGP variation. In contrast, the produced program from G2LGP can hardly see the redundant instructions and structures in the effective instructions.

Furthermore, we can analyze different parts of the produced program from G2LGP based on the derivation tree. Different color blocks in Fig. 11 denote the related instructions and derivation tree nodes. First, the red part of the G2LGP program uses a wide range



```
Ins 0:      (R7= (/ R7 R2))
Ins 1:      (R0= (+ R1 R7))
//Ins 2:    (R5= (min R1 NINQ))
//Ins 3:    (R5= (max R5 R1))
//Ins 4:    (R1= (max R5 R2))
//Ins 5:    (R5= (min R6 NINQ))
Ins 6:      (R1= (max R2 R2))
//Ins 7:    (R3= (- R7 R7))
Ins 8:      (R0= (- R0 R7))
Ins 9:      (R4= (+ R0 PT))
Ins 10:     (R0= (max R2 R6))
//Ins 11:   (R7= (/ R6 R2))
Ins 12:     (R1= (+ R0 R1))
Ins 13:     (R1= (* R1 WKR))
Ins 14:     (R7= (/ R1 R6))
Ins 15:     (R7= (+ R7 NIQ))
Ins 16:     (R5= (/ R7 R0))
Ins 17:     (R6= (max R4 NOR))
Ins 18:     (R1= (/ R5 R2))
Ins 19:     (R2= (+ W R1))
//Ins 20:   (R0= (min R7 R7))
Ins 21:     (R7= (/ R6 R2))
Ins 22:     (R6= (+ R7 PT))
Ins 23:     (R1= (/ R5 R2))
//Ins 24:   (R7= (- R6 R2))
//Ins 25:   (R0= (* R5 R2))
//Ins 26:   (R7= (/ R6 R2))
//Ins 27:   (R7= (- R5 R2))
//Ins 28:   (R3= (+ NIQ R1))
//Ins 29:   (R2= (+ R1 R1))
//Ins 30:   (R5= (min W R1))
//Ins 31:   (R7= (* MWT R2))
//Ins 32:   (R2= (+ R1 R1))
Ins 33:     (R0= (/ R4 R4))
//Ins 34:   (R3= (- R7 R7))
//Ins 35:   (R3= (- R7 R7))
Ins 36:     (R2= (min R0 R1))
//Ins 37:   (R7= (/ R6 R2))
//Ins 38:   (R7= (/ R6 R2))
Ins 39:     (R2= (max W R2))
//Ins 40:   (R5= (min R7 R2))
//Ins 41:   (R7= (- R5 R2))
//Ins 42:   (R7= (max R5 R1))
Ins 43:     (R7= (/ R6 R2))
//Ins 44:   (R1= (/ R5 R2))
//Ins 45:   (R2= (+ W R1))
Ins 46:     (R0= (min R7 R7))
//Ins 47:   (R2= (+ R6 NINQ))
```

```
Ins 0:      (R3= (max R1 R7))
//Ins 1:    (R7= (min R5 NWT))
//Ins 2:    (R5= (min R3 R2))
Ins 3:      (R0= (max R6 R1))
Ins 4:      (R1= (div R3 R2))
//Ins 5:    (R7= (div R1 W))
Ins 6:      (R2= (sub R2 R3))
Ins 7:      (R5= (min R3 R2))
Ins 8:      (R7= (min R3 R2))
//Ins 9:    (R2= (add R2 R5))
Ins 10:     (R2= (sub R1 R3))
//Ins 11:   (R1= (max R0 WKR))
//Ins 12:   (R1= (add R0 R2))
//Ins 13:   (R6= (min R1 R1))
Ins 14:     (R2= (min R2 NPT))
Ins 15:     (R6= (div R7 NINQ))
Ins 16:     (R1= (sub R5 R6))
Ins 17:     (R7= (sub R7 NOR))
Ins 18:     (R3= (max R1 R7))
Ins 19:     (R6= (div R5 R0))
Ins 20:     (R0= (max R6 R1))
Ins 21:     (R6= (add R7 WKR))
//Ins 22:   (R5= (min R3 R2))
Ins 23:     (R2= (sub R2 R6))
Ins 24:     (R5= (max R4 R2))
//Ins 25:   (R2= (max R2 W))
//Ins 26:   (R7= (div R1 R1))
Ins 27:     (R2= (min R5 R0))
Ins 28:     (R4= (min R3 R2))
Ins 29:     (R7= (min R4 R1))
Ins 30:     (R1= (max R1 R7))
Ins 31:     (R1= (add PT R1))
Ins 32:     (R1= (max R0 R1))
Ins 33:     (R1= (add PT R1))
Ins 34:     (R1= (max R0 R1))
Ins 35:     (R1= (add NINQ R1))
//Ins 36:   (R3= (sub R2 rFDD))
//Ins 37:   (R3= (sub R3 R2))
//Ins 38:   (R3= (max R1 rDD))
Ins 39:     (R0= (div R1 W))
//Ins 40:   (R3= (sub R3 R2))
//Ins 41:   (R3= (sub R2 rFDD))
//Ins 42:   (R3= (max R1 NOR))
//Ins 43:   (R3= (sub R2 rFDD))
//Ins 44:   (R3= (sub R1 PT))
//Ins 45:   (R2= (min R0 R1))
//Ins 46:   (R2= (sub WINQ R2))
//Ins 47:   (R2= (min R0 R1))
//Ins 48:   (R2= (add NINQ R2))
```

Figure 11: Example programs of basic LGP (left) and G2LGP (right). "+" is equivalent to "add", and the same to "−, ×, /".

of registers, which implies a high level of parallelization and a wide receptive range on different input features. We can see the red part of the program as a process of feature construction. Second, the green part of the program is related to a `connectbiasSet` module, in which we can explain the instructions as aggregating R0, R1, and R7, and biasing the intermediate results in R1 by PT and NINQ. Based on the division of the G2LGP program, we can conclude that the program intrinsically has different functions in different parts, feature construction at the beginning of the program and aggregating features at the end.

7 FURTHER ANALYSES

One of our main goals is to improve the effectiveness of LGP by restricting LGP program space. But in Section 6, we only see G2LGP has a better training efficiency than basic LGP and ends up with a similar test performance. To investigate whether restricting LGP program space is beneficial for improving LGP effectiveness, we enlarge the program space of the two compared LGP methods by increasing the number of registers and make a further comparison in this section.

We increase the number of available registers of LGP programs from 8 (Section 4) to 12, and the other parameters remain the same as in Section 4. The grammar in Fig. 7 is updated accordingly by including 12 registers in the set REG. The test performance of basic LGP and G2LGP is shown in Table 3. Table 3 shows that G2LGP and basic LGP have significantly different overall performances based on the p-value of the Friedman test with a Bonferroni correction (0.025). The Wilcoxon test with a significance level of 0.05 with a Bonferroni correction on each scenario confirms that G2LGP can be significantly better than LGP in two of six scenarios.

Table 3: Test performance of basic LGP and G2LGP with 12 registers.

Scenario	LGP	G2LGP
	mean (std)	mean (std)
$\langle T_{max}, 0.85\rangle$	1940.8 (49.9)	1921 (41.7) ≈
$\langle T_{max}, 0.95\rangle$	3999 (111.8)	3950.2 (99.9) +
$\langle T_{mean}, 0.85\rangle$	417.8 (2.7)	418.5 (4.9) ≈
$\langle T_{mean}, 0.95\rangle$	1118.3 (10.8)	1115.8 (8.6) ≈
$\langle WT_{mean}, 0.85\rangle$	726.7 (6.9)	727.9 (7.3) ≈
$\langle WT_{mean}, 0.95\rangle$	1743.9 (32.5)	1731.1 (30.2) +
Meanrank	1.917	1.083
p-value	0.025	

Based on the results, we infer that the proposed grammar successfully improves LGP effectiveness in larger program spaces. The reason why G2LGP and basic LGP have a similar test performance in Section 6 is that the program space of LGP methods in Section 6 has been restricted properly by the well-tuned primitive set (e.g., a suitable number of input features and a suitable number of registers [14]). However, the existing studies on restricting the program space by a well-tuned primitive set have not considered some common logical control functions like "IF", and a larger maximum program size also enlarges the program space. It is still beneficial to apply grammar to make a more delicate restriction on the program space.

8 CONCLUSIONS

The main goals of this paper are to improve the effectiveness, efficiency, and interpretability of LGP in solving DJSS problems by reducing the program space. We achieve these goals by a grammar-guided LGP method (i.e., G2LGP). We design an LGP-based grammar system and a suite of grammar-based genetic operators to enable LGP to evolve based on grammar constraints. Furthermore, we design a set of grammar rules based on DJSS problems to verify the effectiveness of G2LGP.

The results show that G2LGP successfully improves the training efficiency in solving the DJSS problems and can produce more easy-to-understand dispatching rules for the DJSS problems. Further analyses show that G2LGP can significantly improve the test effectiveness when the number of LGP registers becomes larger. The results confirm that using grammar to restrict the search space of LGP is an effective method to cooperate the domain knowledge with LGP evolution.

This paper is the first paper that directly imposes grammar rules on LGP, a linear symbolic system. We expect that the introduction of grammar rules gives LGP the potential to consider more complicated primitives like "IF" and search for huge programs within an extremely large search space. We will investigate these directions in the near future.

REFERENCES

[1] Filipe Assunção, Nuno Lourenço, Penousal Machado, and Bernardete Ribeiro. 2019. DENSER: deep evolutionary network structured representation. *Genetic Programming and Evolvable Machines* 20 (2019), 5–35. Issue 1.
[2] Ying Bi, Bing Xue, and Mengjie Zhang. 2022. Genetic Programming-Based Evolutionary Deep Learning for Data-Efficient Image Classification. *IEEE Transactions on Evolutionary Computation* (2022), 1–15. Early Access.
[3] Markus Brameier and Wolfgang Banzhaf. 2007. *Linear Genetic Programming*. Springer US.
[4] Edmund K. Burke, Mathew R. Hyde, Graham Kendall, Gabriela Ochoa, Ender Ozcan, and John R. Woodward. 2009. Exploring hyper-heuristic methodologies

with genetic programming. *Computational Intelligence* 1 (2009), 177–201. Issue 1.

[5] F. Corman and E. Quaglietta. 2015. Closing the loop in real-time railway control: Framework design and impacts on operations. *Transportation Research Part C: Emerging Technologies* 54 (2015), 15–39.

[6] Grant Dick and Peter A. Whigham. 2022. Initialisation and grammar design in grammar-guided evolutionary computation. In *Proceedings of the Genetic and Evolutionary Computation Conference.* 534–537.

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

[8] Stefan Forstenlechner, David Fagan, Miguel Nicolau, and Michael O'Neill. 2018. Extending Program Synthesis Grammars for Grammar-Guided Genetic Programming. In *Proceedings of Parallel Problem Solving from Nature.* 197–208.

[9] Christopher D. Geiger, Reha Uzsoy, and Haldun Aytuğ. 2006. Rapid modeling and discovery of priority dispatching rules: An autonomous learning approach. *Journal of Scheduling* 9 (2006), 7–34. Issue 1.

[10] Erik Hemberg, Jonathan Kelly, and Una May O'Reilly. 2019. On domain knowledge and novelty to improve program synthesis performance with grammatical evolution. In *Proceedings of the 2019 Genetic and Evolutionary Computation Conference.* 1039–1046.

[11] Torsten Hildebrandt, Jens Heger, and Bernd Scholz-reiter. 2010. Towards Improved Dispatching Rules for Complex Shop Floor Scenarios — a Genetic Programming Approach. In *Proceedings of the Annual Conference on Genetic and Evolutionary Computation.* 257–264.

[12] Zhixing Huang, Yi Mei, Fangfang Zhang, and Mengjie Zhang. 2022. Graph-based linear genetic programming: a case study of dynamic scheduling. In *Proceedings of the Genetic and Evolutionary Computation Conference.* 955–963.

[13] Zhixing Huang, Yi Mei, Fangfang Zhang, and Mengjie Zhang. 2023. Multitask Linear Genetic Programming with Shared Individuals and its Application to Dynamic Job Shop Scheduling. *IEEE Transactions on Evolutionary Computation* (2023). https://doi.org/10.1109/TEVC.2023.3263871

[14] Zhixing Huang, Yi Mei, and Mengjie Zhang. 2021. Investigation of Linear Genetic Programming for Dynamic Job Shop Scheduling. *Proceedings of IEEE Symposium Series on Computational Intelligence*, 1–8.

[15] Zhixing Huang, Fangfang Zhang, Yi Mei, and Mengjie Zhang. 2022. An Investigation of Multitask Linear Genetic Programming for Dynamic Job Shop Scheduling. In *Proceedings of European Conference on Genetic Programming.* 162–178.

[16] Rachel Hunt, Johnston Richard, and Mengjie Zhang. 2015. Evolving Dispatching Rules with Greater Understandability for Dynamic Job Shop Scheduling Mark Johnston. Technical report ECSTR-15-6..

[17] Domagoj Jakobović and Leo Budin. 2006. Dynamic Scheduling with Genetic Programming. In *Proceedings of European Conference on Genetic Programming.* 73–84.

[18] Leonardo Lamorgese and Carlo Mannino. 2019. A noncompact formulation for job-shop scheduling problems in traffic management. *Operations Research* 67, 6 (2019), 1586–1609.

[19] Wong Man Leung, Leung Kwong Sak, Man Leung Wong, and Kwong Sak Leung. 1995. Applying logic grammars to induce sub-functions in genetic programming. *Proceedings of the IEEE Conference on Evolutionary Computation* 2, 737–740.

[20] Dimmy Magalhães, Ricardo H.R. Lima, and Aurora Pozo. 2023. Creating deep neural networks for text classification tasks using grammar genetic programming. *Applied Soft Computing* 135 (2023), 110009.

[21] Robert I. McKay, Nguyen Xuan Hoai, Peter Alexander Whigham, Yin Shan, and Michael O'neill. 2010. Grammar-based Genetic programming: A survey. *Genetic Programming and Evolvable Machines* 11 (2010), 365–396. Issue 3-4.

[22] Yi Mei, Qi Chen, Andrew Lensen, Bing Xue, and Mengjie Zhang. 2022. Explainable Artificial Intelligence by Genetic Programming: A Survey. *IEEE Transactions on Evolutionary Computation* (2022), 1–21. https://doi.org/10.1109/TEVC.2022.3225509

[23] David J. Montana. 1995. Strongly typed genetic programming. *Evolutionary Computation* 3 (1995), 199–230. Issue 2.

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

[25] 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 (2013), 621–639. Issue 5.

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

[27] Michael O'Neill, Miguel Nicolau, and Alexandros Agapitos. 2014. Experiments in program synthesis with grammatical evolution: A focus on Integer Sorting. In *Proceedings of the 2014 IEEE Congress on Evolutionary Computation.* 1504–1511.

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

[29] Michael O'Neill and Conor Ryan. 2003. *Grammatical Evolution.* Vol. 4. Springer US.

[30] F. Padillo, J. M. Luna, and S. Ventura. 2019. A Grammar-Guided Genetic Programming Algorithm for Associative Classification in Big Data. *Cognitive Computation* 11 (2019), 331–346. Issue 3.

[31] Gisele L. Pappa and Alex A. Freitas. 2009. Evolving rule induction algorithms with multi-objective grammar-based genetic programming. *Knowledge and Information Systems* 19 (2009), 283–309. Issue 3.

[32] Dominik Sobania and Franz Rothlauf. 2020. Challenges of Program Synthesis with Grammatical Evolution. In *Proceedings of the European Conference on Genetic Programming.* 211–227.

[33] P.A. Whigham. 1995. Grammatically-based Genetic Programming. *Proceedings of the Workshop on Genetic Programming: From Theory to Real-World Applications*, 33–41.

[34] Fangfang Zhang, Yi Mei, Su Nguyen, Kay Chen Tan, and Mengjie Zhang. 2022. Instance Rotation Based Surrogate in Genetic Programming with Brood Recombination for Dynamic Job Shop Scheduling. *IEEE Transactions on Evolutionary Computation* (2022), 1–15. https://doi.org/10.1109/TEVC.2022.3180693

[35] Fangfang Zhang, Yi Mei, Su Nguyen, Kay Chen Tan, and Mengjie Zhang. 2022. Task Relatedness Based Multitask Genetic Programming for Dynamic Flexible Job Shop Scheduling. *IEEE Transactions on Evolutionary Computation* (2022), 1–15. https://doi.org/10.1109/TEVC.2022.3199783

[36] Fangfang Zhang, Yi Mei, Su Nguyen, and Mengjie Zhang. 2022. Multitask Multi-objective Genetic Programming for Automated Scheduling Heuristic Learning in Dynamic Flexible Job-Shop Scheduling. *IEEE Transactions on Cybernetics* (2022), 1–14. https://doi.org/10.1109/TCYB.2022.3196887

[37] Fangfang Zhang, Yi Mei, Su Nguyen, and Mengjie Zhang. 2023. Survey on Genetic Programming and Machine Learning Techniques for Heuristic Design in Job Shop Scheduling. *IEEE Transactions on Evolutionary Computation* (2023), 1–21. https://doi.org/10.1109/TEVC.2023.3255246

[38] Fangfang Zhang, Su Nguyen, Yi Mei, and Mengjie Zhang. 2021. *Genetic Programming for Production Scheduling.* Springer Singapore. XXXIII+338 pages. https://doi.org/10.1007/978-981-16-4859-5