Advanced Linear Genetic Programming and Applications to Dynamic Job Shop Scheduling

by

Zhixing Huang

A thesis
submitted to the Victoria University of Wellington
in fulfilment of the
requirements for the degree of
Doctor of Philosophy
in Computer Science.

Victoria University of Wellington
2024

Abstract

Linear genetic programming (LGP) is an effective evolutionary computation method for searching symbolic solutions. It has been successfully applied to classification and symbolic regression problems and has shown superior performance in these problems because of its linear representation. However, existing studies have not applied LGP to design decision rules for dynamic combinatorial optimization problems. Designing decision rules for dynamic combinatorial optimization problems is substantially different from classification and symbolic regression problems (e.g., no target outputs and limited training instances), which poses new challenges to existing LGP studies. This thesis aims to propose advanced LGP methods and apply them to solve dynamic job shop scheduling (DJSS), a representative dynamic combinatorial optimization problem. More specifically, the overall goal of this thesis is to design LGP methods as a hyper-heuristic (GPHH) method for designing decision rules for DJSS. Our contributions focus on six aspects.

First, this thesis develops an LGP-based hyper-heuristic (LGPHH) framework to effectively train LGP based on DJSS training instances. The LGPHH framework evolves based on a generational framework and initializes registers by diverse features. The results show that the proposed LGPHH method has a superior performance to a basic tree-based GPHH method and can design more compact decision rules than the tree-based one.

Second, this thesis designs new graph-based search mechanisms for enhancing LGP performance. Specifically, we first investigate an effective way to transform the search information in graphs to LGP instructions.

Based on the designed graph-to-instruction transformation, this thesis further proposes a multi-representation GP. The case study of tree-based and linear-based representation shows that the multi-representation GP framework significantly improves the performance of GP methods for DJSS.

Third, this thesis proposes a grammar-guided LGP method to incorporate the domain knowledge of DJSS into LGP search. The grammar-guided LGP method includes a new grammar system, module context-free grammar, for defining grammar rules, and a set of grammar-guided genetic operators for evolving LGP based on the grammar rules. The results show that the proposed grammar-guided LGP can effectively design dispatching rules with IF operations to solve complicated DJSS problems.

Fourth, this thesis proposes a fitness landscape optimization method to automatically optimize the neighborhood structures of LGP solutions to enhance LGP performance. The analyses on the optimized fitness landscape confirm that the proposed method significantly reduces the hardness of LGP fitness landscapes. The empirical results on common DJSS problems further verify that searching against the optimized fitness landscapes has a very competitive performance with advanced methods.

Fifth, this thesis proposes an LGP-based multitask optimization framework based on the multi-output characteristic of LGP to make use of the interplay among similar DJSS problems. The results show that the proposed LGP-based multitask optimization framework has a superior performance to existing multitask GP methods for DJSS problems.

Finally, this thesis further extends two of the advanced LGP methods (i.e., the multi-representation GP and the LGP with fitness landscape optimization) to symbolic regression problems. The superior performance for solving symbolic regression problems implies a good generality of the proposed methods in this thesis to other domains.

List of My Publications

1. **Zhixing Huang**, Yi Mei, Fangfang Zhang, and Mengjie Zhang, "Toward Evolving Dispatching Rules With Flow Control Operations By Grammar-Guided Linear Genetic Programming," *IEEE Transactions on Evolutionary Computation*, pp. 1-15, 2024, doi:10.1109/TEVC.2024.3353207.

2. **Zhixing Huang**, Yi Mei, Fangfang Zhang, Mengjie Zhang, and Wolfgang Banzhaf, "Bridging Directed Acyclic Graphs to Linear Representations in Linear Genetic Programming: A Case Study of Dynamic Scheduling," *Genetic Programming and Evolvable Machines*, vol 25, no 1, article number 5, 2024, doi:10.1007/s10710-023-09478-8.

3. **Zhixing Huang**, Yi Mei, Fangfang Zhang, and Mengjie Zhang, "Multitask Linear Genetic Programming with Shared Individuals and its Application to Dynamic Job Shop Scheduling," *IEEE Transactions on Evolutionary Computation*, pp. 1-15, 2023, doi: 10.1109/TEVC.2023.3263871.

4. **Zhixing Huang**, Yi Mei, Fangfang Zhang, and Mengjie Zhang, "Grammar-guided Linear Genetic Programming for Dynamic Job Shop Scheduling," in *Proceedings of the Genetic and Evolutionary Computation Conference*, pp. 1137–1145, 2023. **[GP Track Best Paper Award]**

5. **Zhixing Huang**, Fangfang Zhang, Yi Mei, and Mengjie Zhang, "An

Investigation of Multitask Linear Genetic Programming for Dynamic Job Shop Scheduling," in *Proceedings of European Conference on Genetic Programming*, pp. 162–178, 2022. **[Best Paper Award]**

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

7. **Zhixing Huang**, Yi Mei, Fangfang Zhang, and Mengjie Zhang, "A Further Investigation to Improve Linear Genetic Programming in Dynamic Job Shop Scheduling," in *Proceedings of IEEE Symposium Series on Computational Intelligence*, 2022, pp. 496–503.

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

9. **Zhixing Huang**, Yi Mei, Fangfang Zhang, Mengjie Zhang, and Wolfgang Banzhaf, "Multi-Representation Genetic Programming: A Case Study on Tree-based and Linear Representations," Submitted to *Evolutionary Computation*.

10. **Zhixing Huang**, Yi Mei, Fangfang Zhang, Mengjie Zhang, and Wolfgang Banzhaf, "Fitness Landscape Optimization Makes Genetic Programming Search Easier," Submitted to *IEEE Transactions on Evolutionary Computation*.

Acknowledgments

I would like to express my great appreciation to my supervisors, A/Prof. Yi Mei, Prof. Mengjie Zhang, and Dr. Fangfang Zhang, for their guidance and support during my PhD study. A/Prof. Mei has spent a lot of time on my research and has shown great tolerance to my strange ideas. He provides many very useful feedback to improve my research skills. Prof. Zhang always trusts me for no reason, which encourages me to explore many unknowns. Dr. Zhang is so nice to me, like an elder sister. She always gives me detailed suggestions from research to life, which is impressive to me. The humbleness, patience, and rigorousness of all my supervisors are invaluable treasures of my life and career. Besides, I would like to thank Prof. Wolfgang Banzhaf who is a big name in LGP. He is always happy to discuss with me and makes our discussion useful and enjoyable. His profound knowledge of LGP always broadened my horizons.

I am grateful to the China Scholarship Council / Victoria University of Wellington Scholarship for their financial support for my PhD study over the past three years. It relieves my financial burden very much.

I would like to thank my friends in the Evolutionary Computation Research Group for the inspiring and open research environment. Many thanks to my two roommates, Meng Xu and Yifan Yang for building a warm atmosphere in our dormitory. Many thanks to Dr. Ruwang Jiao, Dr. Peng Wang, Dr. Qinglan Fan, and Dr. Shaolin Wang who give a lot of guidance and support to me which helps me get through the homesickness. Many thanks to Junhao Huang for sparing time to play badminton

with me. I cannot imagine how dull my PhD can be if you are not here. Many thanks to Jordan MacLachlan for continuously sharing your presentation skills with me. Without these skills, I could not make the two best paper presentations. Many thanks to Hengzhe Zhang for being willing to discuss with me at any time. Thanks Prof. Bing Xue, Dr. Qi Chen, Prof. Hui Ma, and Dr. Aaron Chen. You always recognized my research and presentations, which give me a lot of encouragement. Thanks to Junwei Peng, Yuan Tian, Ruiqi Chen, Bocheng Lin, Qiangqiang Li, Mashfiqul Huq Chowdhury, and so many others unlisted for their jokes and discussions.

Last but not least, I wish to thank my beloved parents, my grandmother, my elder brother, and sister in law for their great support and understanding. You have always been the source of love and company that help me complete my PhD journey. Despite the pandemic at the beginning of my PhD, I finally come to the finish line of my PhD. I am proud to say, I have tried my best and made it!

Contents

List of Tables

List of Figures

Chapter 1

Introduction

From the first life on the earth, to the towering trees and giant beats, then to a man taking his first upright steps, Mother Nature has taught us a vivid lesson of the magical evolution. In artificial intelligence, the power of evolution helps us explore the unknown, and that is evolutionary computation. From designing spacecraft antennas to synthesizing physical laws [80,108], evolutionary computation has advanced for decades in artificial intelligence. In light of these advances, this thesis focuses on applying an evolutionary computation method, linear genetic programming, to help human beings design effective decision rules for dynamic combinatorial optimization problems. Specifically, this thesis proposes advanced linear genetic programming methods to design decision rules for solving dynamic job shop scheduling problems.

This chapter begins by introducing the basic idea of genetic programming and dynamic job shop scheduling. Then, this chapter presents the motivations, research goals, and major contributions of this thesis. Last, the chapter ends with the organization of this thesis.

1.1 Genetic Programming

Genetic programming (GP) is a big family in evolutionary computation that directly searches symbolic solution spaces [106]. GP evolves a population, which consists of a predefined number of GP individuals. In basic GP, a GP individual is a computer program represented by a tree-based structure. GP evolves its individuals to search for competitive computer programs. The evolution of GP is essentially an iteration that competitive individuals consecutively produce offspring by varying computer programs to explore the solution space.

GP has shown to be effective in many domains [11]. For example, GP shows an outstanding performance in designing classifiers for images [16, 54] and documents [63] and synthesizing unknown mathematic formulas [1, 276]. Particularly, GP has shown to be effective in automatically designing decision rules for dynamic combinatorial optimization problems, which has significantly better performance than the decision rules designed by human experts [124,273]. In recent years, GP also played an important role in explainable artificial intelligence [81, 134].

There have been many GP methods in existing studies. For example, cartesian GP [141, 142] and other graph-based GP [213, 214] represent GP individuals as graphs. These graph-based GP methods showed an encouraging performance in designing circuits [143, 144] and searching neural architectures [146]. Gene expression programming encodes the tree structures into fixed-length numerical vectors [57, 122]. PushGP [132, 219] and grammar-guided GP [60,61,131,174] integrate GP algorithms with human programming languages.

1.1.1 Linear Genetic Programming

This thesis focuses on a prominent GP method, linear genetic programming (LGP) [21]. LGP is a GP variant that encodes computer programs or mathematical formulas by a sequence of instructions [9, 163]. The term

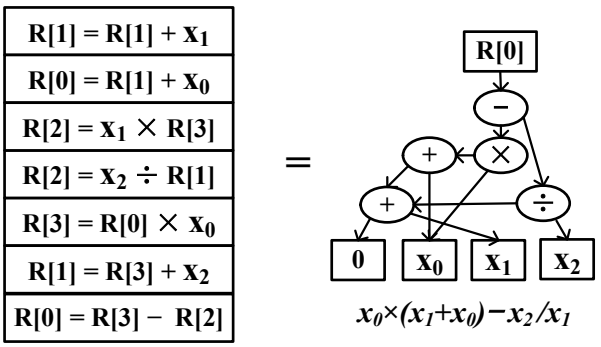

Figure 1.1: An example of LGP programs and its corresponding DAG.

"linear" in LGP has two core meanings. First, an LGP individual is a sequence of register-based instructions (also known as linear representation). Second, an LGP individual executes its instructions sequentially to form a computer program. Fig. 1.1 is an example of an LGP program with $R[0]$ as the program output register. It represents a formula that calculates $x_0 \times (x_1 + x_0) - x_2/x_1$ by sequentially executing the instructions and storing the intermediate results into registers. By connecting the input features (i.e., x_0 to x_2) and functions (e.g., $+$ and $\times$) based on registers, an LGP individual can be represented as a directed acyclic graph (DAG), as shown on the right of Fig. 1.1. LGP evolves individuals within a similar framework to GP but produces offspring in a substantially different way because of the linear representation.

LGP has some advantages over basic tree-based GP (TGP). First, the linear representation facilitates LGP to reuse the intermediate results in a program, which benefits evolving compact computer programs. Second, the linear representation can define multiple outputs easily. The multiple outputs are necessary for applications such as multi-class classification and robot control. Third, LGP individuals are closely related to DAGs. DAGs have a more flexible topology than tree-based structures. Last but not least, the linear representation represents human computer programs naturally (i.e., line-by-line), which facilitates our understanding and anal-

yses of LGP individuals. Nowadays, LGP has been applied to various domains such as classification [49, 196], symbolic regression [87, 212], and airfoil control [182].

1.2 Dynamic Job Shop Scheduling

Job shop scheduling (JSS) problem is a ubiquitous NP-hard combinatorial optimization problem [66]. A job shop processes jobs with a set of machines and outputs products. A job in JSS consists of multiple operations. The key idea of JSS is to arrange different operations to appropriate machines with a certain order and timing so that the performance of the job shop is optimized. The performance of a job shop includes but is not limited to the makespan of processing all the given jobs, the flowtime of each job, and the tardiness of processing [156]. Since JSS is closely interrelated to manufacturing which is a powerful growth engine of the world [277], JSS is becoming a hot topic in both academic and practice [36, 111, 129].

In classic JSS, we know the jobs and machines beforehand. The classic JSS is thus known as static JSS. However, it is common to have new coming jobs or have some machines broken down during processing in real-world production. These cases contribute to a new branch of JSS, dynamic JSS (DJSS). DJSS processes jobs in a dynamic environment in which the information of the scheduling, such as the set of jobs and machines, will change with the processing. These dynamic events cause DJSS a more complex problem than static JSS since we have to adjust schedules timely to improve job shop performance.

1.3 Hyper-heuristics

Hyper-heuristic (HH) methods try to search for suitable scheduling heuristics for a combinatorial optimization problem by selecting or recombining existing scheduling heuristics [25, 44]. Different from heuristic methods

whose search space consists of solutions (i.e., complete schedules), HH methods search in a heuristic space given by users. HH methods have been successfully applied to many applications such as scheduling and routing [205]. Specifically, the heuristics in JSS are known as dispatching rules. It has been shown that HH methods can obtain more sophisticated and effective priority dispatching rules than human-designed ones in many dynamic combinatorial optimization problems such as dynamic routing [153, 239, 278], wireless network [5, 110, 283], and DJSS [3, 22, 273].

Applying GP as a hyper-heuristic method (denoted by GPHH) is an important branch of HH methods [205] (i.e., constructive HH methods). The key idea of GPHH is to apply GP to evolve sophisticated heuristics based on the given functions and simple heuristics. In DJSS, the evolved heuristics play the role of dispatching rules to make instant decisions for dynamic events and construct the final schedule step-by-step. Since GPHH makes a thorough search on the heuristic space, the automatically designed heuristics are likely more effective than those manually designed ones. Furthermore, because of the symbolic representation, the decision rules evolved by GPHH have a great potential for interpretability. In recent years, GPHH has been extensively applied to DJSS problems and has shown promising results [53, 226, 249].

1.4 Motivations

LGP has a number of advantages over basic GP such as easy reuse of intermediate results and natural multi-output [50, 59], and has shown superior performance in benchmark problems. However, its application to combinatorial optimization problems is not fully investigated. This thesis develops advanced LGP methods from four aspects and applies these techniques to solve DJSS problems. The detailed motivations of each issue are introduced below.

1.4.1 Applying LGP to DJSS

Existing studies hardly apply LGP as HH (denoted by LGPHH) to solve dynamic combinatorial optimization problems. To consolidate the foundation of the following studies, this thesis first investigates the effectiveness of basic LGPHH by DJSS. Three reasons motivate the application of LGPHH for solving DJSS problems.

First, LGP has shown encouraging performance in classification and symbolic regression. We expect to apply LGP to further improve the existing GPHH methods for DJSS. Besides, the easy reuse of intermediate results facilitates LGP to evolve compact rules for DJSS.

Second, the training paradigm of LGPHH for solving DJSS gives insights into the practical application of LGP. The training paradigm of LGPHH is greatly different from the one in symbolic regression and classification tasks. For example, the training instances of DJSS are limited and have no expected outputs or labeled data. Many real-world applications suffer similar issues and pose challenges to existing LGP studies.

Third, DJSS is closely related to real-world production and can be extended to many other applications. The wide range of domain knowledge such as the related optimization objectives and their expert-designed heuristics [189, 236] facilitates us to investigate the effectiveness of the LGP-designed heuristics.

1.4.2 Graph-based Characteristics of LGP

One of the important characteristics of LGP is its graph characteristics. By connecting primitives based on registers, an LGP individual can be decoded into a DAG. Presenting LGP individuals (i.e., programs) by DAGs has different advantages from a sequence of instructions. Graphs represent programs in a more compact representation. On the other hand, a sequence of instructions enables neutral search in program spaces and memorizes potential building blocks [145, 217]. Empirical studies have verified

that the two LGP representations, graphs and instruction sequences, are competitive for different tasks [6, 214]. It is valuable to take advantage of both representations in LGP evolution.

However, existing studies did not fully investigate the LGP graph characteristics. More specifically, the utilization of graphs in LGP is *one-way* (i.e., the existing LGP studies mainly consider DAGs as a compact and intuitive way to depict the programs) [21]. Whether there is an effective way to use LGP graph characteristics during evolution is unknown yet. The absence of an effective transformation from DAGs to LGP instructions precludes LGP from fully utilizing the graph information and cooperating with other graph-related techniques such as neural networks. To make full use of LGP graph-based characteristics, this thesis investigates a new graph-based genetic operator for LGP and develops a *two-way* transformation between graphs and LGP programs.

The graph characteristics of LGP further motivate the idea of harnessing multiple GP representations during the evolution, in which GP representations share knowledge via graphs. Generally speaking, a GP representation is expected to be suitable for only a subset of problems. Although some existing studies have investigated the performance of different GP representations in solving different problems based on empirical comparisons [214,244], extending such kind of knowledge to other unseen domains is difficult, and such investigations are often too time-consuming and it is hard to cover all different branches and variants of a problem. When encountering an emerging application or a new problem, users have scarce domain knowledge in selecting a GP representation. To make full use of different GP representations and enhance the search performance of existing GP methods, it would be interesting to investigate whether the different GP representations can cooperate in solving a single task.

1.4.3 Incorporating Domain Knowledge in LGPHH

Human experts already have some domain knowledge in both the LGP and DJSS fields. The domain knowledge consists of both algorithm and problem sides. On the problem side, domain knowledge can be important features for scheduling and the correlation between input features and decisions. On the algorithm side, LGP particularly, domain knowledge can be the suitable number of registers in different parts of programs and some known effective sub-programs. The domain knowledge can be used to shrink the search space and to improve LGP effectiveness.

However, existing studies of LGP and DJSS did not make full use of the domain knowledge during optimization. Introducing more domain knowledge in search dispatching rules is a potential topic to enhance the performance of GPHH methods since it can help reduce the search space and ensure GPHH methods produce meaningful dispatching rules. GPHH has a natural advantage in introducing prior knowledge since it directly constructs a solution by different symbols and these symbols often have different physical meanings. Developing constraints or search mechanisms based on the physical meaning of symbols is a kind of utilization of domain knowledge.

There have been some advanced techniques to fully utilize such kind of knowledge in DJSS such as grammar-guided methods [131, 151] and dimensionality aware methods [136]. However, the existing techniques are mainly designed based on TGP and they cannot be extended to LGP directly. Although LGP has undergone decades of development and has been applied successfully to some problems of classification and symbolic regression [21], few studies endeavor to develop effective methods to fully utilize the problem-specific knowledge for LGP rules.

1.4.4 Fitness Landscape of LGP

A fitness landscape (FL) is a surface that reflects the fitness of all the possible solutions in a search space [105]. A FL plays a crucial role in genetic programming search. A smoother FL with less local optima (e.g., an unimodal landscape) normally implies an easier search problem.

In recent years, many advanced techniques successfully enhanced GP search performance by designing better FLs. For example, multitask GP [267] helps GP jump out from local optima by cooperating with similar landscapes. Feature selection [30] and frequency-based operators [259] change the neighborhood structures (e.g., one-hop mutation) so that GP prefers particular neighbors with a large number of certain features. However, these manually enhanced fitness landscapes need very specific domain knowledge and strong assumptions. For example, in multitask GP, we have to find two (or more) correlated tasks whose fitness landscapes are synergic. In frequency-based mutation, we have to assume that the effective solutions include an effective primitive multiple times, which might not be the case in some applications (e.g., in program synthesis, a program repeats a primitive by looping [238]). It is tedious for human experts to design better FLs. In light of this concern, this thesis intends to propose an approach to automatically design better FLs for LGP, which ultimately enhances LGP performance.

1.4.5 Multi-output LGP for Multitask Optimization

Multitask optimization simultaneously optimizes multiple similar tasks to share the search information [71, 246]. The search information among similar tasks improves the effectiveness of optimization methods. LGP individuals have a great advantage in defining multiple outputs and reusing common building blocks, which are useful in multitask optimization. There have been different multitask optimization techniques developed for GPHH methods [180, 267]. However, many of the existing multitask

techniques are designed based on tree-like structures that only have one output. Although these existing multitask techniques can also be applied to LGP to improve its performance, they cannot fully utilize the advantages of LGP (i.e., the multiple outputs and the easy sharing of common building blocks inside dispatching rules). For example, existing multitask GPHH methods set up multiple sub-populations to solve different DJSS problems and share common building blocks among tasks by duplicating them in different individuals. Contrarily, LGP dispatching rules easily have more than one output, and these outputs share common building blocks within one individual naturally. It is potential for LGP rules to have a concise representation. To fully utilize the characteristics of LGP, specific techniques, such as the training paradigm, the genetic operators, and the chromosome representation, should be re-designed to efficiently evolve LGP individuals in multitask paradigm.

1.5 Research Goals

The overall goal of this thesis is to develop an effective LGPHH method to design effective and trustworthy decision rules for DJSS problems. There are five main research objectives in the thesis which are established respectively based on the motivations mentioned above.

Objective 1: Develop an LGPHH approach based on the training paradigm of HH methods for solving DJSS problems.

This objective aims to develop an LGPHH algorithm for solving DJSS problems. Specifically, this LGPHH algorithm applies a generational evolutionary framework to fit with the training paradigm. The LGPHH is expected to verify the advantages of LGP methods by having a better performance than basic GPHH methods. The LGPHH serves as a baseline for the following objectives.

Objective 2: Develop new search mechanisms to make full use of the graph-based characteristics of LGP.

This objective first proposes a graph-based crossover and mutation to highlight the effective building blocks in LGP parents based on DAGs. It is expected that LGPHH can find effective offspring more efficiently by emphasizing effective building blocks when producing offspring.

Second, this objective proposes a transformation method between graphs and LGP instructions to convert DAGs into LGP building blocks effectively. Specifically, this objective aims to propose three transformation ways which are based on the primitive frequency, the adjacency table, and the adjacency list of a DAG. Then, we investigate these possible transformations to find an effective one from DAGs to LGP building blocks.

Finally, based on the previous findings in this objective, a multi-representation evolutionary framework that simultaneously evolves TGP and LGP individuals is proposed. TGP and LGP have very different program representations and are effective for different problems. The proposed evolutionary framework is expected to obtain more diverse building blocks and have a higher chance of finding better solutions.

Objective 3: Develop LGP-specific grammar-guided techniques for enhancing LGPHH for DJSS by domain knowledge.

The objective aims to develop a grammar-guided LGPHH algorithm for incorporating domain knowledge when solving DJSS problems. First, to extend the grammar-guided techniques to the linear representations of LGP, the proposed algorithm has a new grammar system to specify "legal" instructions and registers. Second, based on the domain knowledge of DJSS problems, a new set of grammar rules is designed within the new grammar system. Third, specific grammar-guided genetic operators are proposed to produce grammatically correct offspring in the new grammar-guided LGPHH algorithm. The proposed grammar-guided LGPHH is expected to have better training efficiency and test performance than the basic LGPHH.

Based on the grammar-guided LGPHH, this objective intends to further introduce flow control operations into LGP primitive sets and apply

grammar-guided techniques to reduce the search space. Given that simply introducing flow control operations into LGP primitive sets brings a large number of redundant solutions, it is necessary to apply grammar-guided techniques to restrict the search space. The introduction of flow control operations are expected to enhance the performance of dispatching rules when solving complicated job shop scenarios, such as the scenarios with floating energy prices.

Objective 4: Develop a fitness landscape optimization method to automatically reduce the hardness of LGP fitness landscapes.

This objective aims to develop an algorithm to automatically optimize FLs of LGP. Specifically, we first index the instructions of LGP and then optimize the indexes to improve FLs. The optimized FL aggregates good solutions and separates good and bad solutions. It is expected that the proposed algorithm can reduce the hardness of FLs automatically by making FLs smoother and reducing their local optima. Based on the optimized fitness landscape, this objective further develops a new search mechanism for LGP which directly searches LGP solutions against the optimized fitness landscape. The search mechanism is expected to enhance LGP performance.

Objective 5: Develop an LGP-based multitask training paradigm to improve the effectiveness of LGP rules in solving multiple DJSS tasks.

This objective aims to develop an LGP-based multitask optimization paradigm based on the multi-output characteristic of LGP for solving multiple DJSS scenarios simultaneously. Specifically, the proposed multitask LGPHH algorithm evolves a sub-population of shared individuals among tasks. The shared individuals among tasks have multiple outputs, each for a task. By inherently sharing common building blocks within LGP individuals with multiple outputs, it is expected that the proposed algorithm can automatically adjust the knowledge-sharing rate among tasks and improve the efficiency of the multitask optimization. To improve the knowledge-sharing effectiveness, new genetic operators that integrate ef-

fective building blocks from different tasks are also designed.

1.6 Major Contributions

This thesis makes the following contributions:

1. This thesis has shown that applying LGP as an HH method to design dispatching rules for DJSS problems has a superior performance to the basic tree-based GPHH method, in terms of the test performance of evolved dispatching rules. The results on program size also verify that the dispatching rules represented by LGP are more compact than those represented by basic tree-base GP because of reusing building blocks. The results show great potential for using LGP to design dispatching rules for DJSS problems.

   To improve LGP performance, this thesis gives several important adaptations of LGP when applying to DJSS problems. First, we apply a generational evolutionary framework to evolve LGP under the training paradigm of HH methods. Second, with the same number of fitness evaluations, we find that a small LGP population and a large number of generations are more suitable for LGPHH than a large LGP population and a small number of generations. Third, we identify a mutation-dominated setting for balancing the genetic operator rates between mutation and crossover to improve the training efficiency of LGPHH. Fourth, we initialize registers by diverse input features of DJSS problems to further improve the training efficiency of LGPHH with the limited number of fitness evaluations.

   Part of this contribution has been published in:

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

Zhixing Huang, Yi Mei, Fangfang Zhang, and Mengjie Zhang, "A Further Investigation to Improve Linear Genetic Programming in Dynamic Job Shop Scheduling," in *Proceedings of IEEE Symposium Series on Computational Intelligence*, 2022, pp. 496–503.

2. This thesis shows how the use of graph information of LGP can enhance DJSS performance, including highlighting effective building blocks by graph-based crossover and mutation, transforming DAGs into LGP instructions based on adjacency lists, and harnessing multiple GP representations based on their graph representations.

   The results show that highlighting effective building blocks based on graphs significantly improves the test performance of LGPHH and has a better training efficiency. The following investigation on the possible transformations from DAGs to LGP instructions verifies that the proposed adjacency list-based crossover is an effective way to convey the search information from graphs to LGP instructions. To fulfill the graph-to-instruction transformation, a register assignment algorithm is developed, which has been shown to be effective in reconstructing the topological structures without significant loss of program effectiveness. The empirical results on unseen data verify that the proposed adjacency list-based crossover has a competitive test performance by directly conveying search information in effective LGP instructions. The investigation also highlights that the effectiveness of making full use of graph-based information becomes more significant when there are more flexible topological structures of graphs (e.g., higher and wider graphs).

   Based on the graph-to-instruction transformation, a multi-representation GP method based on tree-based and linear representation, named MRGP-TL, is developed. The experimental studies on DJSS problems show that the proposed MRGP-TL significantly improves the performance of GP methods without considering the interplay among

different representations. Further analysis shows that MRGP-TL has a very competitive performance with state-of-the-art GPHH in solving DJSS problems. To the best of our knowledge, this work is the first work highlighting that the interplay among different GP representations is useful for improving GP performance.

Part of this contribution has been published in:

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

Zhixing Huang, Yi Mei, Fangfang Zhang, Mengjie Zhang, and Wolfgang Banzhaf, "Bridging Directed Acyclic Graphs to Linear Representations in Linear Genetic Programming: A Case Study of Dynamic Scheduling," *Genetic Programming and Evolvable Machines*, vol 25, no 1, article number 5, 2024, doi:10.1007/s10710-023-09478-8.

Zhixing Huang, Yi Mei, Fangfang Zhang, Mengjie Zhang, and Wolfgang Banzhaf, "Multi-Representation Genetic Programming: A Case Study on Tree-based and Linear Representations," Submitted to *Evolutionary Computation*.

3. This thesis develops a grammar-guided LGP algorithm for solving DJSS problems based on the domain knowledge. Specifically, this thesis develops a module context-free grammar system for defining grammar rules for LGP and gives example grammar rules for solving DJSS problems. A set of grammar-guided genetic operators is developed accordingly to produce offspring based on the grammar.

   The results show that the proposed grammar-guided LGP has better training efficiency than basic LGP, and can produce solutions with good interpretability. Further analyses show that grammar-guided LGP significantly improves the overall test performance when the search space of LGP becomes larger.

Based on the proposed grammar-guided LGPHH, this thesis further introduces flow control operations when designing dispatching rules for DJSS. Specifically, the thesis proposes a new set of normalized terminals for DJSS problems and further proposes a set of grammar rules to restrict the available inputs, the number, and the locations of IF operations. The results show that IF operations are crucial for dispatching rules to solve complex problems and using grammar rules to restrict the usage of IF operations in LGPHH is an effective way to harness IF operations.

Part of this contribution has been published in:

Zhixing Huang, Yi Mei, Fangfang Zhang, and Mengjie Zhang, "Grammar-guided Linear Genetic Programming for Dynamic Job Shop Scheduling," in *Proceedings of the Genetic and Evolutionary Computation Conference*, pp. 1137–1145, 2023. **[GP Track Best Paper Award]**

Zhixing Huang, Yi Mei, Fangfang Zhang, and Mengjie Zhang, "Toward Evolving Dispatching Rules With Flow Control Operations By Grammar-Guided Linear Genetic Programming," *IEEE Transactions on Evolutionary Computation*, pp. 1-15, 2024, doi:10.1109/TEVC.2024.3353207.

4. This thesis proposes a fitness landscape optimization (FLO) method for LGP to automatically reduce the hardness of FLs in DJSS. Specifically, we index the GP symbols and optimize FLs based on the indexes of GP symbols. The symbol indexes define new neighborhood structures of LGP solutions. The proposed method optimizes three aspects of the landscapes, including minimizing the distance between good solutions, maximizing the distance between good and bad solutions, and minimizing the gap between the optimized indexes and the preferred indexes based on domain knowledge.

   The thesis applies four common FL metrics to analyze the hardness of the optimized FLs. The results show that the proposed method

significantly reduces the hardness of fitness landscapes. Our visualization results further confirm some empirically recommended parameter settings in existing LGP literature and discover two interesting patterns of the optimized fitness landscapes. The results on DJSS verify the effectiveness of the FLO method.

To the best of our knowledge, this work is the first attempt to explicitly optimize the FLs of LGP automatically. The proposed FLO method is general enough to apply to other GP methods. Our experiments on four common FL metrics also facilitate further investigations of the correlation of these FL metrics, which is missed by existing FL analysis studies. The two newly discovered patterns of FLs give insights into LGP search. Particularly, the insight of instruction positions implies a potential genetic operator of LGP, that is swapping consecutive instructions, which is missed by existing LGP studies.

Part of this contribution has been published in:

Zhixing Huang, Yi Mei, Fangfang Zhang, Mengjie Zhang, and Wolfgang Banzhaf, "Fitness Landscape Optimization Makes Genetic Programming Search Easier," Submitted to *IEEE Transactions on Pattern Analysis and Machine Intelligence*.

5. This thesis proposes a multitask LGPHH method for simultaneously solving multiple DJSS problems. Specifically, the proposed multitask LGPHH evolves based on a multi-population framework. The first sub-population integrates solutions for different DJSS problems into one LGP individual with multiple outputs, each output for one DJSS problem. The first sub-population evaluates multiple DJSS problems simultaneously. The integrated solutions in the first sub-population share common building blocks and transfer knowledge via multi-output individuals. Based on the newly proposed multitask evolutionary framework, this thesis further proposes a new genetic opera-

tor to merge effective building blocks from different tasks into a new one.

The results show that the proposed method has a significantly better test performance than state-of-the-art multitask GP methods, and the proposed genetic operator has a crucial effect on improving the test performance. Further analyses verify that the new knowledge transfer mechanism can adjust the transfer rate automatically during evolution and thus has superior effectiveness to the knowledge transfer mechanisms with fixed transfer rates. The proposed knowledge transfer mechanism not only enriches the methodologies in transferring knowledge but also provides an effective example of designing graph-based knowledge transfer. The proposed algorithm can be easily extended to other domains such as classification and symbolic regression.

Part of this contribution has been published in:

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

Zhixing Huang, Yi Mei, Fangfang Zhang, and Mengjie Zhang, "Multitask Linear Genetic Programming with Shared Individuals and its Application to Dynamic Job Shop Scheduling," *IEEE Transactions on Evolutionary Computation*, pp. 1-15, 2023,
doi: 10.1109/TEVC.2023.3263871.

1.7 Terminology and Abbreviations

Below are the definitions of the terms commonly used in this thesis:

1. A **genotype** is a sequence of LGP instructions. The genotype of an

LGP individual is inherited from its parents and is changed by genetic operators.

2. A **phenotype** is the effective part of an LGP individual, which essentially determines the final output of an LGP program. It is observable. The decoded DAG of an LGP individual, which represents the core primitives and their connections, is a kind of phenotype.

3. An **intron** is an LGP instruction that does not contribute to the final program output of an LGP individual. Removing the introns from the LGP individual will not affect the outputs.

4. An **exon** is an LGP instruction that contributes to the final program output of an LGP individual. Exons are the complementary part of introns in an LGP individual.

5. A **simulation** is the process that simulates the working environment of a DJSS problem. A simulation of DJSS simulates the job processing in a job shop. The randomly generated jobs come into the job shop and are processed by the machines in the job shop. The simulation performance is regarded as the objective value.

6. An **instance** is a specific simulation instance with a fixed random seed.

7. A **scenario** represents a set of simulation instances with the same problem configuration, e.g., the same objective and utilization level in DJSS. Different scenarios might have different difficulty levels for optimization methods.

8. A **task** is equivalent to a scenario. We use the term "task" to demonstrate an optimization problem for LGPHH. Solving different scenarios simultaneously is a multitask learning problem.

Table 1.1 shows the common abbreviations and their full names or meanings in this thesis.

Table 1.1: Abbreviations and their meanings

Abbreviations	Full Names or meaning
LGP	Linear Genetic Programming
TGP	Tree-based Genetic Programming
DJSS	Dynamic Job Shop Scheduling
HH	Hyper-Heuristics
Tmax	Maximum Tardiness
Tmean	Mean Tardiness
WTmean	Weighted Mean Tardiness
Fmax	Maximum Flowtime
Fmean	Mean Flowtime
WFmean	Weighted Mean Flowtime
PT	The processing time of an operation (refer to Table 3.1)
WKR	The total remaining processing time of the job (refer to Table 3.1)
DAG	Directed Acyclic Graph
ALX	Adjacency List-based Crossover
MRGP	Multi-Representation Genetic Programming
FL	Fitness Landscape
G2LGP	Grammar-guided Linear Genetic Programming
MCFG	Module Context-Free Grammar
FLO	Fitness Landscape Optimization
MTGP	Multitask Genetic Programming
RSE	Relative Square Error
Avg.	Average
std.	Standard deviation

1.8 Organization of Thesis

Fig. 1.2 shows the outline of this thesis, including the main goals (listed with ✓) and involved techniques (listed with •) in each chapter, and the

connections between the chapters in this thesis. The remainder of this thesis is organized as follows. An overview of each chapter is shown in Fig. 1.2. Specifically, the major contribution chapters (i.e., chapters 3 to 7) take advantage of LGP characteristics (e.g., building block reusing and multi-output) and different levels of search information (i.e., genotype, phenotype, and fitness landscapes) to enhance GPHH performance.

Chapter 2 presents the literature review, including the basic concepts of LGP and DJSS problems. Then, this thesis discusses the basic framework of applying LGP-based HH to solve DJSS problems, followed by the related work of the LGP, the approaches for DJSS, and the four objectives.

Chapter 3 describes the application of basic LGPHH to solve DJSS problems. Chapter 3 mainly verified the superior performance of basic LGP based on its advantages. Specifically, chapter 3 first introduces the algorithm components of basic LGPHH in detail. Then, chapter 3 investigates the settings of LGP population size and generation numbers, the variation step size, and the initialization strategies. The results recommend a suit of effective settings of LGP and verify that LGPHH has a better test performance and a more compact representation than tree-based GPHH. This chapter mainly applies the basic LGP and hyper-heuristics techniques.

Chapter 4 describes advanced search mechanisms based on the graph-based characteristics of LGP. Specifically, chapter 4 first describes graph-based genetic operators that highlight the effective building blocks. In addition, chapter 4 investigates three possible transformations to convert graph information into LGP instructions. Based on the investigation, chapter 4 further describes the multi-representation GP method. The results show that making full use of graph-based features improves the training efficiency and test performance of LGP This chapter mainly considers DAGs and adjacency lists of graphs.

Chapter 5 describes a grammar-guided LGPHH method for solving DJSS problems. Specifically, chapter 5 first introduces a grammar-guided

LGPHH with the proposed module context-free grammar system, followed by a set of grammar rules for enhancing LGP performance. Based on the proposed grammar-guided LGPHH, chapter 5 introduces flow control operations into the LGP primitive set and develops corresponding grammar to reduce the redundant solutions caused by flow control operations. The results show that the grammar rules designed based on the domain knowledge of DJSS improve the test effectiveness and training efficiency of LGPHH. Besides, the proposed method improves the interpretability of the output rules. The flow control operations significantly improve LGPHH performance for solving complex DJSS problems. This chapter mainly applies grammar-guided techniques and considers the flow control operations in LGPHH.

Chapter 6 describes a fitness landscape optimization method for genetic programming. Specifically, chapter 6 first introduces a generic fitness landscape optimization algorithm that automatically improves the neighborhood structures. Second, chapter 6 applies LGP as a case study to investigate the effectiveness of the proposed method. The results show that the proposed fitness landscape optimization algorithm significantly reduces the hardness of fitness landscapes. This chapter mainly applies stochastic gradient descent for optimization and fitness landscape metrics for analysis.

Chapter 7 describes an LGP-based multitask optimization method for DJSS problems. Chapter 7 first describes a multi-population evolutionary framework for evolving multi-output LGP individuals. Then, chapter 7 proposes a genetic operator that transfers knowledge among tasks. The empirical results show that the multitask LGPHH significantly improves the training efficiency and test performance of existing multitask GPHH methods.

Chapter 8 extends the proposed advanced LGP methods to symbolic regression problems to investigate their potential generality. Chapter 8 focuses on the proposed multi-representation GP method in chapter 4 and

the fitness landscape optimization method in chapter 6 since the effectiveness of these two methods is less dependent on domain knowledge of DJSS. The results suggest that the two proposed advanced LGP methods have good potential generality to other domains.

Chapter 9 summarizes the achieved objectives and the main conclusions of this thesis. Some discussions and future research directions are also presented in this chapter.

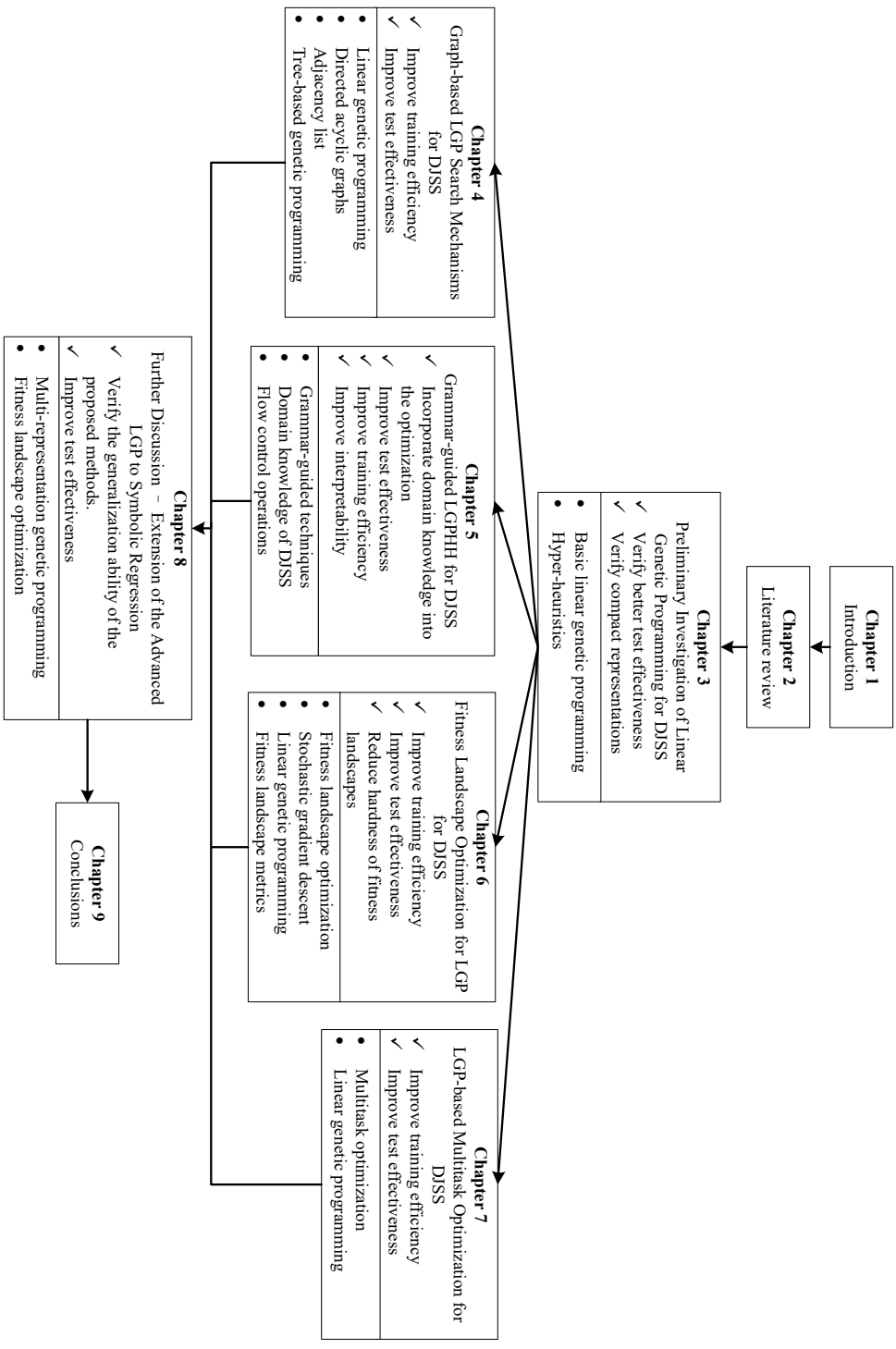

Figure 1.2: The outline of this thesis, including the main goals and involved techniques of each chapter, and the connections between the chapters in this thesis.

Chapter 2

Literature Review

This chapter first introduces the basic concepts in evolutionary algorithms, LGP, dynamic job shop scheduling, and hyper-heuristics. Second, the chapter introduces the existing studies of LGP, JSS, and GPHH for DJSS, respectively. Last, the chapter discussed the related studies for the research goals in this thesis in detail.

2.1 Basic Concepts

2.1.1 Evolutionary Algorithms

The evolutionary algorithm is a kind of artificial intelligence algorithm that borrows the idea of biological evolution to search for problem solutions [7, 147]. Fig. 2.1 shows the overall framework of an evolutionary algorithm. An evolutionary algorithm first initials a population of individuals, each representing a problem solution. In each generation, these individuals are evaluated by a fitness function (i.e., fitness evaluation), and better individuals have more chances to be parents and produce offspring. The production of offspring is based on genetic operators such as mutation and crossover. The algorithm iterates multiple generations until stopping criteria (e.g., reaching the maximum number of generations) are

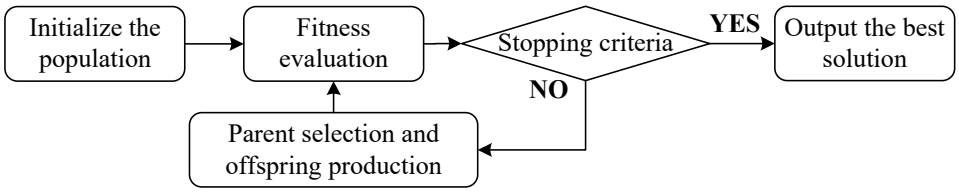

Figure 2.1: The overall framework of evolutionary algorithms.

satisfied and outputs the best individual. This search mechanism makes the evolutionary algorithm a gradient-free optimization method, which is necessary for many real-world problems that hardly have differentiable mathematical models. It has shown that the evolutionary algorithm is an effective stochastic search paradigm for a wide range of problems.

The evolutionary algorithm has two main evolutionary frameworks, generational and steady-state evolutionary algorithms [98]. Generational evolutionary algorithms produce nearly a whole population of offspring to replace the old population at every generation. On the contrary, steady-state evolutionary algorithms produce a small number of offspring each time and only replace the solutions with worse fitness in the population. These two evolutionary frameworks have their pros and cons. For example, generational evolutionary algorithms usually have a higher diversity, while steady-state evolutionary algorithms usually have a quicker convergence.

There are four main branches of evolutionary algorithms. They are genetic algorithm [79], evolution strategy [15], GP [106], and evolutionary programming [58]. This thesis focuses on LGP, a variant of GP.

2.1.2 Linear Genetic Programming

Evolutionary Framework

The evolutionary framework of LGP in this thesis is the generational evolutionary algorithm [214]. LGP starts with randomly initializing the LGP population. If the stopping criteria (e.g., the maximum number of generations and the maximum running time) are not satisfied, all of the LGP individuals will be evaluated based on the predefined fitness function and the training data. To produce offspring, LGP conducts parent selections to pick up parent individuals from the population and choose different genetic operators (e.g., macro mutation, crossover, and micro mutation) to variate the parent individuals based on a certain probability distribution. The distribution is usually predefined manually and the rate of macro mutation and crossover usually take a large proportion. LGP applies these genetic operators exclusively. After generations of evolution, the best-of-run individual (i.e., the individual with the best fitness on training data) will be outputted. The components of the evolutionary framework are introduced in detail below.

Individual Representation

Every LGP individual $\mathbf{f}$ is a sequence of register-based instructions $\mathbf{f} = [f_0, f_1, ..., f_{l-1}], l \in [l_{min}, l_{max}]$, where l is the number of instructions, and l_{min} and l_{max} are the minimum and maximum number of instructions respectively. Every instruction f has three parts: destination register $R_{f,d}$, function $fun_f(\cdot)$, and source registers $R_{f,s}$. All of the destination and source registers come from the same set of registers $\mathcal{R}$. Note that constants (e.g., input features) in LGP programs are treated as a kind of read-only registers, which only serve as source registers. An instruction reads the values from source registers, performs the calculation indicated by the function, and writes the calculation results to the destination registers. Rewritable registers are known as calculation registers. In every program

execution, the registers are first initialized by certain values such as "1.0", and the instructions are executed one by one, from f_0 to f_{l-1}, to represent a complete computer program. The final output of the computer program is stored in a pre-defined output register, which is normally set to the first register by default.

However, not all instructions contribute to the final output. When an instruction is not connected with the program body producing the final output, or the instruction can be removed without affecting the program behavior, the instruction becomes ineffective in calculating the final output. The ineffective instructions are also called "*introns*". Contrarily, the instructions that participate in the calculation of the final output are defined as effective instructions, which are also called "*exons*". Specifically, the term "intron" in this thesis mainly denotes the structural intron whose destination register is not used as source registers by the following instructions[1]. Introns and exons can be identified by an algorithm that reversely checks each instruction in the program based on the output registers [21].

Fig. 2.2 shows an example of an LGP program and its corresponding phenotype in DAG. Specifically, there are eighteen instructions in the program, but only eleven of the instructions are effective (i.e., exons). The introns are highlighted in grey, following a double slash. The final output is returned by the first register R$[0]$. All of the instructions manipulate a register set $\mathcal{R}$ with eight registers. There are eight input features (i.e., constant registers), denoted Input$[0]$ to Input$[7]$. The eight registers are initialized by the eight input features respectively at the beginning of every execution.

An LGP program can be represented as a DAG. Specifically, the LGP program itself and the corresponding DAG are also known as the genotype and phenotype of LGP, respectively. The instruction sequence is transformed into a DAG by connecting functions and constants of exons

[1]We do not consider semantic introns that is a kind of effective instructions but mainly perform meaningless calculation such as $x = x + 0$, which does not affect the final output.

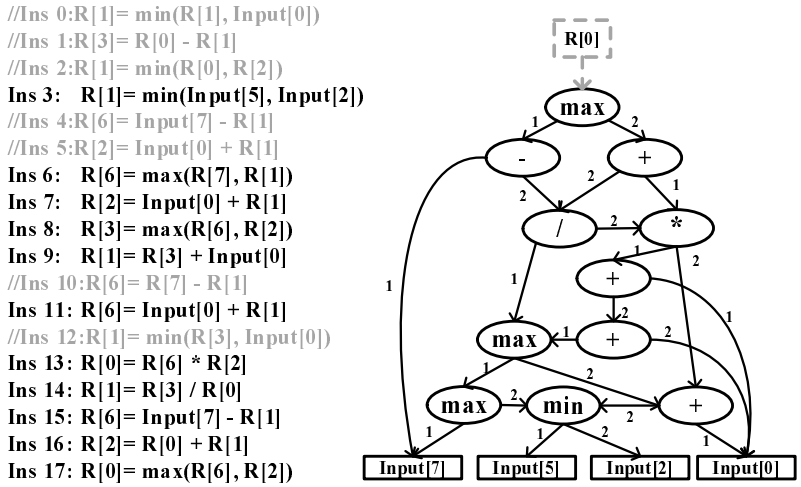

Figure 2.2: An LGP program example and its corresponding DAG. "Input[·]" are read-only registers, and "R[·]" are calculation registers.

based on the registers [21] (i.e., connecting "+" to "×" in DAG if "+" uses the register overwritten by "×" as one of its inputs, see instruction 16 and instruction 13 in Fig. 2.2.). The indexes "1" and "2" indicate the first and second arguments of a specific function. As shown in the right part of Fig. 2.2, the last instruction of the LGP program that overwrites the output register (instruction 17 in this example) is seen as the start node in the DAG. Since the source registers of instruction 17 store the results of instructions 15 and 16 respectively, the start node has two outgoing edges, directing to two graph nodes ("−" and "+") respectively. The indices along the edges indicate the first and second inputs for "$\max(\cdot)$".

Initialization

LGP initializes individuals in a random manner. Both of the program length (i.e., the number of instructions) and the instructions are initialized randomly. Specifically, it is advisable to define an initial maximum and

minimum program length to limit the initial program length of LGP individuals to a certain range. Usually, the initial maximum program length is much smaller than the actual maximum program length in the evolution to encourage LGP to have a thorough search from short programs to long ones.

Fitness Evaluation

Evaluation is a problem-specific step that evaluates the fitness of LGP individuals based on the given training data. Different from basic GP, LGP has an additional step to initialize all the registers before execution. There are several ways for register initialization. For example, the registers can be initialized as some constant values such as "0" and "1". They can also be initialized as different input features. Based on some existing studies, initializing registers into different input features is useful in solving complex problems [21]. Besides, it is also necessary to define the output registers for LGP. The first register is the output register by default without loss of generality.

Parent Selection

The evolutionary framework adopts two selections in LGP evolution: tournament selection and elitism selection. Tournament selection randomly samples a certain number (i.e., tournament size) of individuals from the existing population and only retains the best one among these sampled individuals as the selected individual. The tournament selection is used to select parent individuals for generating offspring in LGP. By this means, some unsatisfactory individuals can also be selected to maintain the diversity of the LGP population.

Elitism selection is mainly used to protect the superior individuals from being eliminated in the evolution. Given an elitism rate such as 1%, the elitism selection picks the best top 1% individuals and retains them

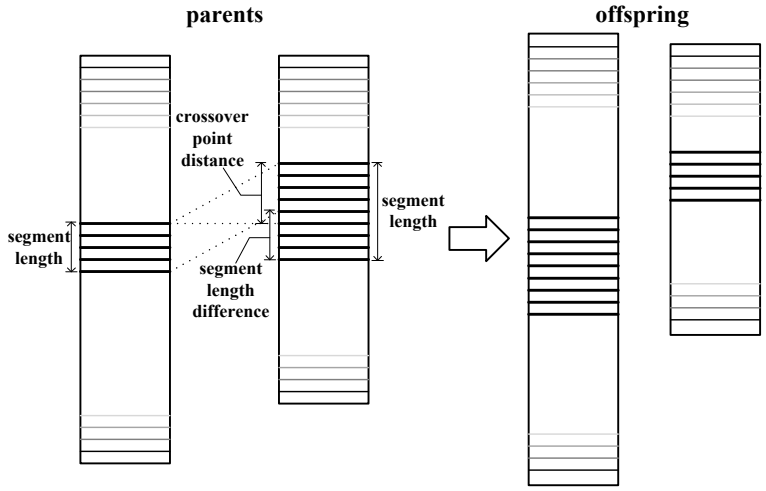

Figure 2.3: Linear crossover.

directly to the offspring population at the next generation.

Offspring Production

LGP applies genetic operators to produce offspring (i.e., new solutions). Genetic operators produce offspring by varying LGP parents. There are three basic genetic operators for LGP, i.e., crossover, macro mutation, and micro mutation.

The main idea of crossover in LGP is to exchange the instruction segments between two parent individuals. Linear crossover (also called two-point crossover) is a typical implementation of crossover [12]. It has three parameters to control the variation, which are the segment length, the length difference of the segments, and the distance between crossover points. Fig. 2.3 shows the relationship among these three parameters. Besides the linear crossover, there are other variants of crossover such as one-point crossover which divides the LGP individual into two parts by identifying a crossover point and exchanging one of the two parts with others, and one-segment crossover which accepts an instruction segment

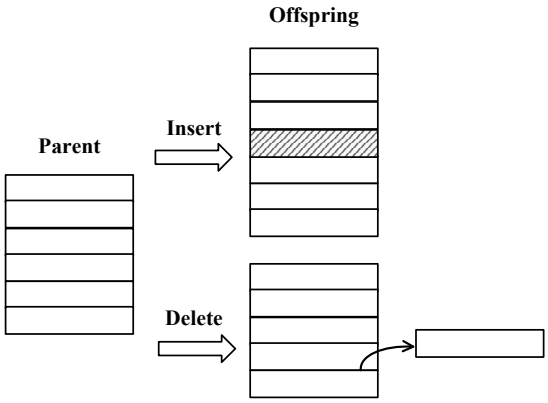

Figure 2.4: Macro mutation.

from another individual or remove a segment of instructions from the individual [21].

The macro mutation mainly updates the parent individual by inserting or deleting the instructions in the LGP individuals. The term "macro" indicates that macro mutation will change the number of instructions in an LGP program. Basically, a macro mutation accepts one parent individual and produces a new one. There are two parameters, the probability of insertion and deletion (denoted as p_{ins} and p_{del} respectively, and $p_{ins} + p_{del} = 1$), to control the behavior of macro mutation (i.e., increasing, maintaining, or reducing the program length). By default, only one instruction will be inserted or deleted from the individual each time. An illustrative diagram of macro mutation is shown in Fig. 2.4. Specifically, a new instruction is inserted to the fourth position in the offspring (i.e., shadow one) or the fifth instruction of the parent individual is removed in the example. There are other variants for macro mutation. For example, Banzhaf et al. [10] proposed an effective version of macro mutation which only inserts or deletes effective instructions into (from) LGP individuals.

The micro mutation only updates the primitives in the instructions such as operations and registers, but does not change the number of instructions directly. To mutate different types of primitives, a probability

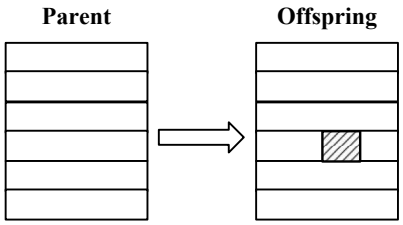

Figure 2.5: Micro mutation.

distribution of different primitive types is designed for micro mutation. Basically, there are three main primitive types in instructions. They are operations, registers, and constants. By default, we mutate these three types of primitives uniformly. Note that the input features and the constant registers are also constants. Fig. 2.5 is an illustrative figure of the micro mutation. One of the instructions in the parent individual is selected randomly, and one of the primitives will be mutated. Specifically, one of the primitives of the fourth instruction (i.e., shadow one) is mutated by the micro mutation. The micro mutation also has its effective version, in which the selected and the new mutated instruction must be an effective instruction [21]. Brameier et al. [21] suggest appending micro mutation after macro mutation and crossover for solving complex problems.

Difference Between TGP and LGP

LGP is different from basic GP which is based on tree structures [106]. Each TGP individual encodes a computer program as a tree. Every tree node represents a function or a terminal (i.e., input feature). Function nodes accept inputs from their sub-trees and deliver results to their parent nodes. Each tree node has up to one parent node. All intermediate results from sub-trees are aggregated at the root, with the root outputting the final result of the program.

A comparison between a linear representation and a tree-based repre-

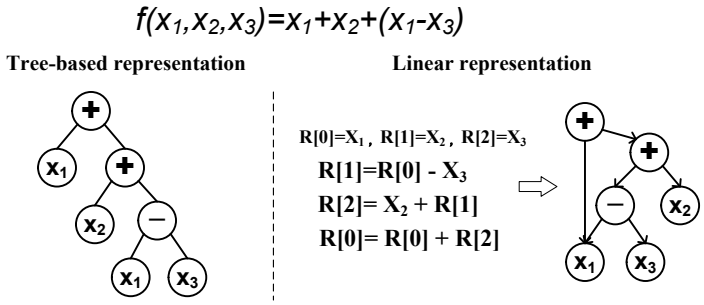

Figure 2.6: An example of GP individuals with tree-based and linear representations for the same mathematical formula.

sentation for the same mathematical formula "$f(x_1, x_2, x_3) = x_1+x_2+(x_1-x_3)$" is shown in Fig. 2.6. Specifically, in the linear representation, x_1 to x_3 are read-only input registers, and the calculation registers $R[0]$ to $R[2]$ are initialized by x_1 to x_3 respectively (e.g., the first instruction is equivalent to "$R[1] = x_1 - x_3$"). The final output of the instruction sequence is stored in $R[0]$.

2.1.3 Dynamic Job Shop Scheduling

Since DJSS was first distinguished from static JSS in 1957 [95], it has undergone a prosperous development and nowadays becomes one of the most popular variants of JSS [150]. Two typical DJSS problems are the one with new job arrival [90, 137] (i.e., the new arriving jobs can only be known and scheduled after they come into the job shop) and the one with machine breakdown [179, 180] (i.e., additional repairing time or redoing the interrupted operations are required). Besides these two types of DJSS, there are also many other dynamic events for DJSS, such as order cancellation [198], due date changes [13], quality rejected and so on [150]. The dynamic events cause DJSS a more complex problem than conventional static JSS. DJSS has a wide range of applications in the real world, such as cloud computing [129], car body assembling factory [206], traffic man-

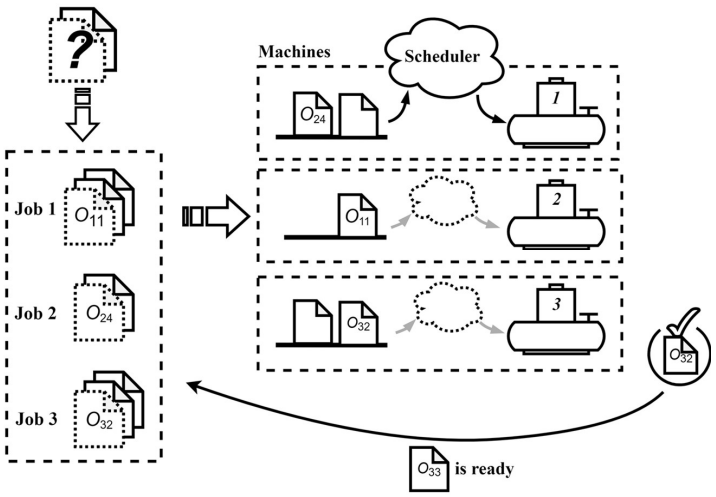

Figure 2.7: The schematic diagram of DJSS.

agement [111], airport runway scheduling [43], and railway control [36]. Because of the enormous business value of DJSS, it has attracted great interest from both academia and industry.

This thesis will mainly focus on the DJSS with new job arrivals since this type of DJSS is representative and quite common in real-world production. The schematic diagram of DJSS with new job arrivals is shown in Fig. 2.7. A job shop has a finite set of machines $\mathbb{M}$, each with an available operation queue $q(m), m \in \mathbb{M}$. There is a job set $\mathbb{J}$ in the job shop. Jobs come into $\mathbb{J}$ during optimization (i.e., new job arrival). Note that the job shop cannot know or schedule jobs until their arrival. For every job $j \in \mathbb{J}$, it has a sequence of operations $\mathcal{O}_j = \{o_{j1}, ..., o_{ji}, ..., o_{jk_j}\}$ where k_j is the number of operations in job j. The operation sequence specifies the execution order of different operations (e.g., o_{j1} must be executed prior to o_{j2}). When an operation o_{ji} is available, it enters the corresponding operation queue $q(m)$ of a certain machine $m(o_{ji}) \in \mathbb{M}$ and is processed based on a scheduler of the machine. o_{ji} is removed from the queue when it is processed, and its next operation $o_{j,i+1}$, if it exists, will enter the corresponding machine queue after o_{ji} is finished (e.g., o_{33} becomes available after o_{32} is

finished in Fig. 2.7). Every operation is processed with a positive processing time $p(o_{ji})$. Every job has its own arrival time α_j, due date d_j, and weight ω_j which describes the importance of a job. Besides, for the sake of simplicity, we simply assume that once an operation starts to be processed by a machine, the process cannot be interrupted by other events, and each machine processes at most one operation at any time. The main task in JSS is to sequence the execution order of operations on each machine so that the job shop performance can be optimized.

$$x(o_{j1}) \geq \alpha_j, \qquad \forall j \in \mathbb{J} \tag{2.1}$$

$$x(o_{ji}) \geq x(o_{j,i-1}) + p(o_{j,i-1}), \qquad \forall j \in \mathbb{J}, i = 2, ..., k_j \tag{2.2}$$

$$p(o_{j,i}) > 0, \qquad \forall j \in \mathbb{J}, i = 1, ..., k_j \tag{2.3}$$

$$x(o_{ji}) \geq x(o_{hi}) + p(o_{hi}) - V \cdot z_{gjh}, \qquad \forall j, h \in \mathbb{J}, j < h, \; g = m(o_{ji}) = m(o_{hi}) \tag{2.4}$$

$$x(o_{hi}) \geq x(o_{ji}) + p(o_{ji}) - V \cdot (1 - z_{gjh}), \qquad \forall j, h \in \mathbb{J}, j < h, \; g = m(o_{ji}) = m(o_{hi}) \tag{2.5}$$

$$z_{gjh} \in \{0, 1\}, \qquad \forall j, h \in \mathbb{J}, g \in \mathbb{M} \tag{2.6}$$

Based on these definitions, we formulate the DJSS problems based on a disjunctive model [109]. Denote the starting time of job j as $x(o_{j1})$, we have constraints (2.1 to 2.6). Constraint (2.1) ensures that all jobs are processed after arrival. Constraints (2.2) and (2.3) guarantee that the operations will be processed sequentially. The V is a large enough variable so that

$$\begin{gathered} x(o_{ji}) \geq x(o_{hi}) + p(o_{hi}) - V \\ x(o_{hi}) \geq x(o_{ji}) + p(o_{ji}) - V \\ \forall j, h \in \mathbb{J}, j < h, g = m(o_{ji}) = m(o_{hi}) \end{gathered}$$

It ensures that every machine can process at most one operation at any time. z_{gjh} is an auxiliary variable to describe which jobs are being processed by machine g at a time. When g decides to process job h prior to

job j, $z_{gjh} = 0$ (so that Eq. (2.4) is active and Eq. (2.5) is inactive). When g decides to process job j prior to h, $z_{gjh} = 1$ (so that Eq. (2.5) is active and Eq. (2.4) is inactive). This DJSS modeling can be extended to many other variants, such as flexible JSS (i.e., an operation can be processed by more than one candidate machines) [115], resource-constrained JSS (i.e., the total number of processing jobs at any time is limited) [155], etc.

There are many optimization objectives in JSS, including minimizing the makespan (i.e., the total processing time of the whole job shop), minimizing the tardiness (i.e., the delay from the due date), and minimizing the flowtime (i.e., the total processing time of a job). Besides, there are some other objectives in literature such as minimizing workload [274], minimizing energy consumption [45, 123], and improving interpretability [136], which facilitates the real-world applications of JSS. These objectives often conflict with each other [128] and can be optimized together as multi-/many-objective problems. This thesis mainly minimizes two popular objectives in DJSS, i.e., tardiness and flowtime.

Simulation Configuration

In this thesis, we evaluate individual fitness by simulation. A GP individual plays the role of dispatching rules in DJSS problems. When a machine becomes idle, the GP individual decides which operation should be processed by prioritizing the available operations. The inputs of the individual are job-related, machine-related, and job-shop-related information. The schedule of a simulation is constructed over the simulation. After the job shop processes a certain number of jobs, the performance of the simulation is output as the fitness of the GP individual.

The configurations of the DJSS simulation are set according to the common settings of existing studies [77, 136]. Specifically, our job shop contains a total of $|\mathbb{M}| = 10$ machines. An operation $o_{ji} \in \mathcal{O}_j$ of a new arrival job j is designated to a certain machine $m(o_{ji})$ randomly and is completed by the designated machine with a continuous processing time

ranging from 1 to 99 (i.e., $p(o_{ji}) \in [1, 99]$). Every job has at least 2 operations and at most 10 operations. The new jobs arrive to the job shop at arrival time α_j. The new jobs have different weights based on a predefined probability. Specifically, 20% have a weight of 1, 60% have a weight of 2, while the rest have a weight of 4. The due date of job j (i.e., d_j) is defined by a due date factor, which is set as 1.5 in the simulation (i.e., $d_j = \alpha_j + 1.5 \times \sum_{o_{ji} \in \mathcal{O}_j} p(o_{ji})$). Our investigation mainly focuses on the steady-state performance of DJSS simulation, in which there are 1000 warm-up jobs in the experiment and only the subsequent 5000 completed jobs are counted in optimization objectives.

In the simulation, jobs come into the job shop based on a Poisson distribution, as shown in Eq. (2.7). t is the time interval before the next job arrival. λ is the mean processing time of a job in the job shop, defined by Eq. (2.8). ν is the average number of operations in the jobs, and μ is the average processing time of operations. The utilization level of machines ρ defines the arrival rate of jobs. A large ρ implies that jobs will be processed by the job shop very quickly (i.e., a small mean actual processing time of jobs) and that new jobs arrive at the job shop in a shorter time. Thus, a large ρ tends to lead to a difficult job shop scenario since a dispatching rule has to make effective schedules to avoid bottlenecks.

$$P(t = \text{next job arrival time}) \sim \exp(-\frac{t}{\lambda}) \tag{2.7}$$

$$\lambda = \frac{\nu \cdot \mu}{\rho \cdot |\mathcal{M}|} \tag{2.8}$$

To comprehensively verify the performance of the proposed method on different difficulty levels, this thesis increases the difficulty levels by increasing the utilization levels of the job shop (i.e., ρ in Eq. (2.8)). The utilization levels ρ are set as 0.85 or 0.95 respectively in our experiment. A higher utilization level implies a busier job shop and more difficulty in finding an effective schedule. To improve the generalizability of LGP-evolved rules, the training instances have different random seeds in different generations, which is also known as instance rotation [78].

Design of Scenarios

In this chapter, we verify LGPHH performance on six example scenarios with different optimization objectives and utilization levels. The six optimization objectives are commonly used in existing literature [156]. The six optimization objectives include maximum tardiness (Tmax), mean tardiness (Tmean), weighted mean tardiness (WTmean), maximum flowtime (Fmax), mean flowtime (Fmean), and weighted mean flowtime (WFmean), where c_j is the actual completed time of job j. The six objectives are job-related objectives rather than job-shop-related objectives because the optimization is an ongoing process during simulation. The job-related objectives would give an insight to the future performance of the job shop.

1. Tmax= $\max_{j\in\mathbb{J}}(\max(c(j)-d(j),0))$
2. Tmean= $\frac{\sum_{j\in\mathbb{J}}\max(c(j)-d(j),0)}{|\mathbb{J}|}$
3. WTmean= $\frac{\sum_{j\in\mathbb{J}}\max(c(j)-d(j),0)\times\omega(j)}{|\mathbb{J}|}$
4. Fmax= $\max_{j\in\mathbb{J}}(c(j)-\alpha(j))$
5. Fmean= $\frac{\sum_{j\in\mathbb{J}}c(j)-\alpha(j)}{|\mathbb{J}|}$
6. WFmean= $\frac{\sum_{j\in\mathbb{J}}(c(j)-\alpha(j))\times\omega(j)}{|\mathbb{J}|}$

The scenarios in this thesis are denoted as "⟨A, B⟩" where "A" is the objective and "B" is the utilization level. For example, ⟨Tmax, 0.95⟩ indicates a DJSS scenario optimizing the maximum tardiness, and jobs arrive to the job shop based on a Poisson distribution with $\rho = 0.95$. The six example scenarios are ⟨Tmax, 0.85⟩, ⟨Tmax, 0.95⟩, ⟨Tmean, 0.85⟩, ⟨Tmean, 0.95⟩, ⟨WTmean, 0.85⟩, and ⟨WTmean, 0.95⟩.

2.2 Related Work

2.2.1 Advances in LGP

LGP is a representative graph-related GP [211, 214]. The linear representations in the GP area showed up as early as the 1980s [38]. After that, Nordin et al. [163, 164] and Wolfgang [9] further developed the linear representation into more general approaches to evolve computer programs in the 1990s. LGP has shown superior performance in symbolic regression and classification problems [21, 164]. For example, Downey and Fogelberg respectively applied LGP to solve multi-class image classification problems [49, 50, 59]. 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 utilize common building blocks for different sub-classes, LGP has advantages over TGP in multi-class classification. LGP also shows a superior performance to TGP and artificial neural network in binary classification [19, 196]. Besides, LGP has undergone a good development in solving symbolic regression. For example, Huang et al. [87] proposed an error back-propagation mechanism for LGP to improve its performance in symbolic regression. Gligorovski and Zhong [68] extended LGP to vectorial functions in symbolic regression. Dal Piccol Sotto et al. [41] developed a probability model to learn the distribution of elite LGP individuals and sample offspring based on the probability model in symbolic regression problems. These improvements for LGP achieved very encouraging results. Sotto et al. [212] also verified that LGP has a better bloat control than TGP in symbolic regression problems. In addition to these works, LGP is successfully applied to other domains such as automatical algorithm design [42] and parallel computation [47, 48]. LGP has also been widely applied in a vast range of industrial applications such as refinery crude oil scheduling [187], objective function synthesis [191], and stall control of airfoils [182].

The genetic operator is an important design issue of LGP. 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. [10, 20] made a lot of comparisons on different types of crossover and mutation operators. They found that ensuring the effectiveness (i.e., the variation must change at least one LGP effective instruction) 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. [49] designed a class path crossover operator that swaps the DAGs contributing to the same subclass. In genetic improvement, some mutation and crossover operators were also proposed based on a new linear representation of software repair operations [170]. 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 because of the lack of target outputs in DJSS.

2.2.2 Dynamic Job Shop Scheduling Approaches

There are roughly three kinds of approaches for solving DJSS, including completely reactive approaches, robust proactive approaches, and predictive-reactive approaches [199]. This thesis mainly applies LGPHH for DJSS, which automatically constructs reactive approaches for DJSS by LGP.

Completely Reactive Approaches

The completely reactive approaches generate new schedules according to ongoing dynamic events without considering the possible dynamic events in the future. In reactive approaches, there have been many heuristic methods that construct schedules for the optimization problem by one-pass. Without the iterative improvement of solutions, heuristic methods can react to dynamic events and make decisions instantly. Specifically, in DJSS domain, the heuristic methods are some human-designed dispatching rules which prioritize operations as soon as the machines become available. There have been many human-designed rules such as earliest due date [94, 189], shortest processing time [210], and weighted apparent tardiness cost [236].

Some studies apply machine learning models to perform instant decisions for DJSS problems. For example, machine learning models such as decision trees, support vector machines, and artificial neural networks are used to learn the behaviors of dispatching rules [152, 169, 195, 207]. These machine learning models mainly treat DJSS problems as classification problems and need optimal schedules as training data. Reinforcement learning is a machine learning technique that learns the dispatching behaviors based on simulations and rewards [39, 112, 178, 197, 204, 279]. It mainly contains an action network that decides the reaction to a certain situation, and a rewarding network that evaluates the value of decisions. Although reinforcement learning gets rid of the dependence on optimal schedules, the design of its rewarding network which guides the learning is tedious.

However, these man-made dispatching rules and machine learning models are highly dependent on the quality of training data and are less effective in solving complicated DJSS problems [248,266]. Contrarily, GPHH has shown its pros and cons. For example, GPHH discovers effective dispatching rules unknown to humans and has shown promising performance in designing effective dispatching rules for large-scale DJSS prob-

lems. However, the trial-and-error search mechanisms of GPHH methods are low efficiency in some cases. Section 2.2.3 gives a detailed literature review of GPHH for DJSS problems. To take advantage of the pros and cons of different methods, combining GPHH methods and other machine learning methods would be a promising direction.

Robust Proactive Approaches

The robust proactive approaches take possible dynamic events in the future into consideration by making a robust initial schedule that is less sensitive to any potential disruption. For example, fuzzy logic and fuzzy arithmetic consider the vague information in the optimization models for solving DJSS with uncertainty [14]. Meta-heuristics such as the genetic algorithm [4, 218] and particle swarm optimization [64, 220, 247], produce robust schedules by simulation optimization methods. They first build a job shop surrogate model based on the given information. The surrogate model imitates the potential dynamic events over simulation. Then the meta-heuristic methods optimize schedules so that they are robust to these dynamic events.

Predicting possible dynamic events in DJSS is a key challenge in proactive approaches. To predict the possible dynamic events, existing studies construct empirical distribution based on historical data to simulate dynamic events such as new job arrival [117]. Some studies formulated robust scheduling problems into a multi-objective optimization problem that optimizes the job shop performance and minimizes the deviation of the schedules with possible dynamic events [114, 166].

Predictive-reactive Approaches

Predictive-reactive approaches perform scheduling within two steps. It first generates a robust schedule in advance by predicting the possible future dynamic events and second repairs the existing schedule triggered by

dynamic events. For example, Liu et al. [117] proposed to solve a dynamic flow shop scheduling problem by a predictive-reactive framework. The predictive-reactive framework includes four meta-heuristics and incorporates a multi-objective optimization technique to perform post-disruption rescheduling. Li and Gao [114] proposed a hybrid rescheduling architecture consisting of the genetic algorithm and tabu search. The hybrid rescheduling architecture performs rescheduling when dynamic events occur, or when the periodic rescheduling point is reached. To react the dynamic events in a short time, Li and Gao proposed to identify the termination criterion of the rescheduling by tabu lists.

2.2.3 Existing GPHH for DJSS

Applying GPHH to DJSS has undergone a profound development [23, 156]. GPHH evolves dispatching rules to react and make instant reactions to the dynamic events of DJSS (i.e., a reactive approach). The existing studies have validated that GPHH can design more effective dispatching rules than human experts in many DJSS scenarios [96, 148]. However, most of the existing studies focus on TGP and neglect LGP. Therefore, this section mainly reviews the techniques for enhancing TGP-based HH. Briefly speaking, these techniques cover primitive design, GP representations, search mechanisms, fitness evaluation, and training instances.

Primitive Design

Primitives compose GP individuals, including terminals and functions. Effective primitives help GP evolve effective dispatching rules. To improve the effectiveness of the primitive set, feature selection techniques and new primitives for GPHH are proposed. For example, Mei et al. [135] and Zhang et al. [261, 271] applied feature selection techniques to enhance the effectiveness of dispatching rules. Specifically, Mei et al. [135] applied a permutation-based method to identify important features, while Zhang

et al. [261, 271] applied a frequency-based method to adaptively learn the important features. Sitahong et al. [208] further improved the frequency-based feature selection method by integrating it into GP evolution and avoiding redundant features.

Some studies designed new primitives to improve the performance of GPHH. For example, Hunt et al. [90] proposed to include the state of the job shop and the stage of jobs' progress into the primitive set to improve GP rules' global perspective. To make the rules designed by GPHH less sensitive to the time of decision situations, Mei et al. [137] proposed to evolve dispatching rules with time-invariant terminals.

GP Representations

GP representations define the connection ways of primitives. Designing effective dispatching rule representations is one of the important ways to enhance the effectiveness of GPHH. A representative work is named "GP-3" proposed by Jakobović [96]. There are three rules for different purposes in GP-3 method. One rule is designed to distinguish bottleneck machines, and the other two rules are the dispatching rules for bottleneck and non-bottleneck machines respectively. It is reported that GP-3 can outperform other human-designed dispatching rules in DJSS. A similar idea of composite rules is also adopted by Nguyen et al. [158]. In [158], various types of composite rules are investigated. For example, the tree structure is divided into a decision tree and man-made dispatching rules or divided into a decision tree and GP-evolved rules. Their computational analysis found that a GP tree composing a decision tree and GP-evolved rules has the best performance. To fully utilize the machines with different properties, Pickardt et al. [190] designed a multi-tree paradigm to evolve a group of dispatching rules and assign them to different machines.

Some studies also attempt to apply other GP representations to encode dispatching rules. For example, Nie et al. [161] applied the gene expression programming for DJSS. Because gene expression programming en-

codes tree-based programs into a list of primitives, it has a good control over the program size without the loss of effectiveness. An investigation by Durasević et al. [51] also validates the similar effectiveness and higher interpretability of gene expression programming compared with conventional TGP.

The suitable representations vary according to the different characteristics of investigated problems. For example, the multi-tree representations are proposed for solving dynamic flexible JSS problems [161, 269]. The multi-tree representation GP has two trees in each individual, one for a routing rule and the other for a sequencing rule. Nguyen et al. [158] also proposed to use different primitives in different parts of the rules to integrate the decision tree and GP-evolved dispatching rules. For example, in the decision tree part, the thresholds of different percentages are allowed, while only the job shop attributes and random real values are allowed as the terminals of GP-evolved heuristics.

To simplify the tree-based dispatching rules and improve their interpretability, Panda et al. [175, 176] and Planinić et al. [194] respectively proposed a simplification method that consists of algebraic reduction and permutation-based pruning to remove redundant building blocks in the outputted dispatching rules.

Note that the existing studies of GP representation in GPHH mainly focus on TGP. The tree-based representation often leads to redundant and complex rules since it is hard for tree-based representations to reuse intermediate results. They have to duplicate effective building blocks or develop multiple trees for different sub-tasks. This limitation prevents TGP from evolving effective rules within a compact program. Contrarily, the easy reuse of intermediate results and the multi-output of LGP potential address this limitation.

Search Mechanisms

Search mechanisms in this thesis mainly indicate the offspring production methods of GP. GPHH produces offspring by genetic operators. The effectiveness of genetic operators greatly affects the performance of GPHH. Zhang et al. [259] proposed to measure the importance of sub-trees based on the feature importance. Important sub-trees have a higher probability of being shared with other GP individuals. Xu et al. [250] proposed a semantic genetic operator for GPHH in solving flexible DJSS. The semantic genetic operator encourages GP to produce diverse offspring by considering semantically different sub-trees, which improves GPHH effectiveness. To improve the diversity of the population, Planinić et al. [193] developed an ϵ-Lexicase selection for GPHH which successfully improved the solution diversity and training convergence speed.

Adding search constraints in offspring production is an effective way to improve the training efficiency and interpretability of dispatching rules. Designing grammar rules for GPHH is an example of search constraints. Hunt et al. [91] first applied grammar-based GP to DJSS problems and designed a set of DJSS-specific grammar rules. They found that the grammar rules can improve the interpretability of dispatching rules but with an acceptable compromise of poorer mean testing performance. The grammar-based GP for DJSS is also compared with other GP methods such as standard GP and gene expression programming [51]. The investigation validated the higher interpretability of dispatching rules evolved by grammar-based GP. Mei et al. [136] also developed a dimension gap to measure the distance between the physical meaning of DJSS attributes to improve the interpretability of dispatching rules. Compared with the previous work [104], Mei et al. additionally introduced a dimension gap and developed a new fitness function based on the dimension gap to improve the flexibility of the dimensionally aware GP. The grammatical evolution is also applied to JSS problems to design dispatching rules. For example, Teng et al. [222] proposed a hybrid method to utilize the characteristics of

both grammatical evolution and genetic programming and designed a set of grammar rules for grammatical evolution to evolve dispatching rules in intercell scheduling problem (i.e., a variant of JSS problems).

Fitness Evaluation

GPHH methods evaluate the fitness of individuals based on DJSS instances. The fitness essentially guided the GP search. Making full use of the limited training instances of DJSS problems is critical for improving GPHH performance. For example, to fully utilize the GP performance over the simulation, Xu et al. [251] proposed a multi-case fitness evaluation method to evaluate GPHH performance. To improve the efficiency of fitness evaluation, Hildebrandt and Branke [77] and Nguyen et al. [160] developed two surrogate models for applying GPHH to DJSS problems.

In [77], Hildebrandt and Branke developed a phenotypic characterization to describe the decision behaviors of dispatching rules. The phenotypic characterization-based surrogate model firstly samples a set of decision situations and sets up a benchmark dispatching rule for reference. The dispatching rules coming into the surrogate model will make decisions for the given decision situations and compare their decisions with the ones of the benchmark dispatching rule. The behavior difference from the benchmark dispatching rule is transformed into a fixed length vector to represent the behavior of given rules. The phenotypic characterization estimates the fitness of a GP individual by measuring the phenotypic similarity between the GP individual and the elite individuals. The phenotypic characteristic-based surrogate model is further utilized and developed by Zhang et al. [268] and Nguyen et al. [159]. They respectively extended the phenotypic characteristic-based surrogate model to dynamic flexible JSS problems and proposed a new selection scheme to balance the exploration and exploitation.

Nguyen et al. [160] developed a surrogate model based on the simplified job shop simulation. The similar idea is firstly utilized in 2013 [190].

The performance in the simplified simulation has a high correlation with the performance in the full simulation. Nguyen et al. [160] also identify that the "HalfShop" which has a half number of machines and jobs of the full simulation has the best estimation performance. The simplified simulation-based surrogate model is widely applied in DJSS problems. For example, Mei et al. [135] applied the surrogate model to make the feature selection in DJSS problems. Zhang et al. [270] and Zhou et al. [286] respectively applied this surrogate model to dynamic flexible JSS problems. Because the training instances of GPHH are rotated at each generation to improve the generalization ability, the fitness values in this generation cannot compare with the fitness in other generations. To tackle this limitation, Zhang et al. [255] proposed a fitness mapping strategy to make the fitness across different instances comparable. Zeiträg et al. [254] applied surrogate techniques to improve the training efficiency of multi-objective DJSS and reduced more than 70% training time. To improve the effectiveness of DJSS surrogates, Zhu et al. [287] proposed a sampling strategy to sample effective and diverse GP individuals for constructing the surrogate.

Multitask optimization is another technique to improve the effectiveness of fitness evaluation. Multitask optimization evaluates the fitness of individuals on more than one DJSS task. The fitness on multiple tasks guides GP methods to jump out of local optima and reach better solutions. For example, Zhang et al. [256] applied multitask optimization techniques to dynamic flexible JSS which significantly improved the effectiveness and efficiency of GPHH. Zhang et al. [263, 268] further integrated surrogate models with multitask learning and knowledge transfer to improve the training efficiency of GPHH. Multitask optimization is also used to enhance GPHH methods for solving multi-objective DJSS problems [272].

Training Instances

Training GPHH methods by effective training instances is a promising method to improve the generalization ability. For example, some investi-

gations improve the generalization ability of dispatching rules by properly setting the training scenarios [138]. They found that ensuring the same number of jobs and machines or at least a similar ratio value of the number of jobs and machines can improve the effectiveness of GPHH methods on static JSS problems. Hart et al. [73] also tried to apply an artificial immune system network to utilize the relationship between dispatching rules and static JSS instances. They showed that by training dispatching rules for different instances and combining them in an ensemble manner, the effectiveness of the proposed methods can be superior to the standard GPHH method. However, these studies mainly focus on solving static JSS problems. To balance the exploration and exploitation of GPHH methods in solving DJSS problems, Karunakaran et al. [103] proposed an active sampling method to evolve specific rules for different instance clusters. They proposed a metric to measure the similarity of different DJSS instances. Based on the similarity metric, different DJSS instances are clustered into different categories. An ϵ-greedy approach was also developed to further evolve the potential dispatching rules based on certain instance clusters.

2.2.4 Graphs in LGP

Graph-based LGP

Evolving LGP programs based on graphs has some benefits over evolving LGP programs based on instruction sequences. First, graphs allow us to have more precise control over the variation step size of exons. Contrarily, the variation in raw genotype might lead to an unexpectedly large variation step size since it might deactivate and activate different subprograms. Second, swapping LGP building blocks based on graphs protects the useful building blocks from being destroyed. For instance, when we swap the instruction sub-sequences of two LGP programs to produce offspring, the exons (and introns) in the sub-sequences are not guaranteed to be effective (and ineffective) after swapping since they might be

deactivated (or activated) in the new LGP program context. The building blocks in the sub-sequences might be distorted severely, which often leads to destructive variations (i.e., the fitness of the offspring is worse than the fitness of their parents) [165]. But swapping sub-graphs of LGP parents naturally maintains the connections of primitives within sub-graphs and protects effective building blocks from being deactivated. Third, treating LGP programs as graphs can further evolve the program by replacing the introns with effective building blocks, especially when the instruction sequence reaches the maximum program size. Fully utilizing the maximum program size implies a longer effective program length and a better search performance [216, 217]. Last but not least, explicitly considering graph information in LGP programs encourages LGP to have a more compact representation and reduce redundancy.

Explicitly employing graphs in LGP has not been well investigated though it is an effective way to show the program. Brameier and Banzhaf [21] claimed that using graph representations to evolve LGP program does not always lead to better performance than conventional LGP in solving classification and symbolic regression problems, but evolving graph representations needs much more complicated genetic operators than imperative representation. However, the conclusions in [21] are only based on classification and symbolic regression problems where fitness evaluations are cheap, which is different from DJSS problems where there is only a limited number of fitness evaluations because of the time-consuming simulation. Sotto et al. [213, 214] compared a number of graph-based GP methods, including LGP, in their experiment. However, the graph-based genetic operators of the compared graph-based GP methods in [213, 214] mainly manipulate genotype by graph-based mutations and miss graph-based crossover for producing new graphs by swapping sub-graphs. The absence of graph-based crossover limits the variation step of graph-based GP methods and precludes useful building blocks from being shared among the population.

In short, the explicit use of graphs in LGP has not been well investigated from the following aspects: 1) the existing literature misses the graph-based crossover that truly swaps sub-graphs to produce offspring; 2) the existing literature misses the graph-to-instruction transformation.

Relationship among Graphs, Exons, and Instructions

First, fully utilizing graph characteristics of LGP is different from only evolving exons. Graphs are the phenotype of LGP programs, and LGP instructions are the genotype. Exons in LGP instructions are the instructions that are highly related to the graphs since the graphs are decoded based on the exons. However, exons cannot fully stand for the phenotype since graphs carry the essential information of both imperative primitives and their connections, which is a higher-level representation than the exons, while exons are highly dependent on the context of an LGP program. Besides, a graph is an abstract representation that is free from specific genotype design (e.g., register-based instructions in LGP or Cartesian coordinates in Cartesian GP). Representing GP programs by graphs enables GP methods to analyze building blocks effectively and to exchange genetic materials with many other graph-based techniques.

Second, evolving LGP programs based on graphs cannot replace evolving LGP based on instruction sequences. Some existing studies have proposed to directly represent GP programs by graphs and to evolve GP programs by manipulating the graphs [6, 133, 214]. However, their results show that different GP representations are suitable to different tasks. For example, LGP and Cartesian GP have certain advantages in solving the tested digital circuit design problems because of the reuse of intermediate results while TGP is good at solving the tested symbolic regression problems [6, 214]. Evolving programs by genotype instead of graphs also enables LGP and Cartesian GP individuals to perform neutral search (i.e., varying the genotype of an individual but not changing the fitness value) and retain potential building blocks in the individuals [69, 145, 217, 224].

The neutral search and the potential building blocks have been shown to be helpful for LGP and Cartesian GP programs to jump out of local optima and improve the population diversity in a wide range of problems. Further, existing literature on evolving GP programs by graphs directly only considered node and edge mutation on graphs in producing offspring and missed graph-based crossover, which makes these methods inefficient to exchange useful building blocks.

In a nutshell, a graph is an abstract representation of effective LGP instructions, while a sequence of LGP instructions represents underlying codes that include both effective and ineffective instructions. Graphs enable LGP programs to cooperate with other graph-based techniques and explicitly take the topological structures into consideration, while instruction representations enable the neutral search and the memory of potential building blocks. Finding an effective way to bridge these two representations would be useful for LGP evolution. However, to the best of our knowledge, there is no existing literature of graph-based genetic programming investigating suitable ways of utilizing the general graph information (e.g., adjacency list) and effective transformations from graph to LGP instructions. The graphs of LGP programs are not fully utilized yet.

Other Graph-based GP

There are some other graph-based genetic programming methods besides LGP. For example, Cartesian genetic programming [141] is one of the well-known graph-based genetic programming methods. The genotype of Cartesian GP is a list of grid nodes. Each node specifies a function, its connections with other nodes, and its Cartesian coordinate in the grid. By executing these grid nodes based on the connection, the individuals of Cartesian GP can be decoded into DAGs to represent computer programs or digital circuits (i.e., the phenotype). Cartesian GP has shown some superior performance in designing digital circuits [143,144], performing image classification [177], and neural architecture search [146]. Wilson et al. [244]

compared Cartesian GP with LGP and found that the way of restricting the interconnectivity of nodes is the key difference between these two graph-based genetic programming methods. Based on the idea of Cartesian GP, Atkinson et al. [6] proposed the Graph Programming method to evolve graphs (abbr. EGGP). Different from Cartesian GP which strictly requires that the connections must go from the right columns to the left columns (i.e., the levels-back constraint), EGGP allows a graph node $\mathcal{A}$ to connect any other graph node $\mathcal{B}$ in the graph, as long as $\mathcal{A}$ and $\mathcal{B}$ do not form a cycle in the graph. The EGGP individuals have higher flexibility in topological structures than the LGP and Cartesian GP individuals. They further validated the effectiveness of EGGP on the test problems by comparing it with TGP, Cartesian GP, and LGP [213, 214]. The results show that all the three DAG-based GP methods (i.e., LGP, Cartesian GP, and EGGP) have a great advantage over TGP in searching multiple-output computer programs such as digital circuits. However, these studies have not investigated the graph-based crossover operators that truly swap sub-graphs and the graph-to-instruction transformations.

2.2.5 Grammar-based Genetic Programming

Grammar is a popular tool to enforce restrictions on GP search space. GP programs can search for effective solutions faster and get rid of redundant programs by reducing to a smaller yet effective search space. There have been many studies about grammar-based genetic programming [131, 243].

Context-free-grammar-based GP is a typical grammar-guided genetic programming method [243]. A context-free grammar is a set of production rules that define the derivation from high-level concepts to low-level concepts regardless of the program context. To construct a program, grammar-guided GP recursively derives the concepts based on the production rules and forms a tree-based program. A context-free grammar is regularly defined by Backus Naur Form. Grammar-guided GP is widely

applied to program synthesis problems [60, 62], regression problems [93], and automatic algorithm design [89]. Under the umbrella of grammar-guided GP, grammatical evolution is representative of linear grammar-guided GP methods [172, 202]. Unlike tree-based ones, grammatical evolution searches on bit strings (or integer strings) and maps the bit strings into computer programs based on a set of grammar rules by a MOD operator. Grammatical evolution has undergone a lot of improvement. For example, Lourenço et. al. [120, 121] developed a structured grammatical evolution that improves the locality.

LOGENPRO [113, 245] is a tree-based grammar-guided genetic programming that uses a context-sensitive grammar, PROLOG Definite Clause Grammars, to define constraints for GP search space. Due to the context-sensitive grammar, LOGENPRO is more expressive than context-free-grammar-basaed GP. Following LOGENPRO, Ross [200] proposed a logic-based GP system with definite clause translation grammar.

Strongly typed GP is an alternative GP method that imposes data type constraints on GP search spaces [151]. Strongly typed GP specifies all the possible data types of arguments and returns for all the non-terminals. The non-terminals can only have children with specified data types. To make the data type constraints more flexible, strongly typed GP additionally introduces generic functions and generic data types which specify a set of possible data types by algebraic quantities. Due to the flexibility in handling different data types, existing studies applied strongly typed GP to different problems, such as classification [17, 186], finance [34, 125, 140], and software testing [240].

Although there have been grammar-guided techniques for TGP, it is difficult to extend these techniques to LGP since LGP has a substantially different representation. The registers in LGP are also hard to constrain by existing grammar-guided techniques. An LGP-based grammar-guided method is needed.

Grammar-based Genetic Programming in Combinatorial Optimization

Introducing domain bias by grammar is not a new idea to enhance GPHH for solving JSS problems. Nguyen et. al. [158] used grammar to define three representation templates for GP individuals, including 1) selecting simple dispatching rules based on machine attributes, 2) an arithmetic representation, and 3) selecting sub-arithmetic representations based on machine attributes. However, [158] did not show the superior performance of their grammar-like method in solving JSS problems. Hunt et. al. [91] used grammar to categorize input features into different types and defined the available input and output types for each function. However, [91] improved the interpretability of dispatching rules but sacrificed effectiveness although only slightly.

Grammar-guided GP methods have also been applied to many other combinatorial optimization problems. For example, Pawlak and O'Neill [184] used grammatical evolution to synthesize constraints for a diet plan optimization problem. Fenton et. al. [139] and Saber et. al. [203] applied grammar-guided GP for network scheduling. Correa et. al. [37] developed a grammar-guided GPHH method for solving corridor allocation problems. Pereira et al. [187, 188] developed a quantum-inspired grammar-based linear GP to schedule crude oil refinery.

Although these existing studies have applied grammar to enhance GP in solving combinatorial optimization problems, the grammar-guided GP methods for combinatorial optimization are not well investigated. Grammar improves GP interpretability or training efficiency while (not necessarily) sacrificing test effectiveness [92, 93]. Furthermore, existing studies mainly include arithmetic and domain-specific operators in their grammar rules but ignore flow control operations, which are expected to be important primitives for solving many combinatorial optimization problems.

Flow Control Operations in GPHH for DJSS

Flow control operations decide which parts of a dispatching rule can be executed based on the input or program context. There are many different implementations of flow control operations in existing GPHH methods for DJSS. For example, GP-3 [96] fixed IF at the output of dispatching rules and designed an IF-included template to explicitly divide the dispatching rule into three sub-rules, one for scheduling bottleneck machines, one for scheduling non-bottleneck machines, and one for detecting bottleneck machines. If the dispatching rule detects a machine as a bottleneck in the job shop, the job shop uses the bottleneck-machine dispatching rule to make decisions and uses the non-bottleneck-machine dispatching rule otherwise. Đurasević et. al. [51] used a unary flow control operation which returns the operand if the operand is larger than 0 and returns 0 otherwise to control program execution flow. Hildebrandt et.al. [78] and Christopher et al. [67] simultaneously included binary flow control operations (e.g., maximum and minimum) and a ternary IF operation in evolving dispatching rules. The ternary IF operation accepts three arguments, returns the second input argument if the first argument is larger than 0, and returns the third input argument otherwise.

When DJSS scenarios become more complex, flow control operations become more important in designing dispatching rules. For example, Masood et. al. [126–128] used maximum, minimum, and a ternary IF operation in solving multi-objective JSS. Karunakaran et. al. [101, 102] included these operations in solving DJSS problems under uncertainty. Park et. al. [179, 181] also included flow control operations in GP primitive set when solving DJSS with machine breakdown.

In addition to the flow control operations mentioned above, Miyashita [148] developed a four-argument IF operation, which returns the third argument if the first argument is less than or equal to the second argument and returns the fourth argument otherwise. Nguyen et al. [157] showed that flow control operations are effective in designing due-date estimation

models.

However, the mentioned studies have not fully investigated flow control operations. They simply included flow control operations into the function set and treated them equivalently with other arithmetic operations, which inevitably leads to a huge search space and a large number of redundant solutions. Moreover, designing flow control operations for TGP in existing studies is not straightforward since the tree-based representation in TGP is greatly different from human line-by-line programs.

IF Operations in Linear Genetic Programming

The linear representation of LGP programs facilitates the introduction of IF operations. The IF-included LGP programs have a similar representation to human programs. For example, LGP programs can indicate the closure of IF branches (like "{...}" in C language) by defining the number of instructions in an IF branch or designing additional labels or pointers (e.g., `endif`) to specify the end of a branch [21,100]. However, despite the potential advantages of IF operations in LGP, the studies of applying IF operations in LGP are very limited, up to the best of our knowledge.

To summarize, existing GP studies have not effectively evolved IF-included solutions as IF operations inevitably introduce many redundant solutions into search spaces. Grammar-based techniques are effective in removing redundant GP solutions from search spaces. However, existing grammar-based GP methods mainly define the basic format of flow control operations (e.g., IF operations must be followed by a boolean operation) but do not address the dimension inconsistency and inactive and ineffective sub-rules of flow control operations. Moreover, existing grammar-based GP methods are designed based on tree-based representations and missed linear representations which can naturally accept human programming skills.

2.2.6 Fitness Landscape of GP

Fitness Landscape Analysis in Genetic Programming

An FL of GP is an important perspective to understand the hardness of GP search. An FL consists of three components: fitness function, solution space, and the neighborhood structure of solutions [228]. A fitness function measures the effectiveness and quality of all the possible solutions, the possible solutions compose a solution space, and the neighborhood structure defines the distance among the possible solutions in the solution space. The existing studies on GP's FL mainly focus on analyzing the hardness of FLs and describing FL properties. Many efforts endeavored to develop quantity metrics to measure the hardness and describe FL properties.

Fitness distance correlation (FDC) is representative of the FL metrics. FDC measures the problem hardness by estimating the correlation between the distance and the fitness difference from global optima [99]. On an easy landscape, solutions are supposed to have better fitness when they are closer to known global optima, which means the GP fitness has a high correlation with the distance to the optimal solutions. Otherwise, the landscape is misleading. Normally, FDC divides the hardness of a search problem into three levels (suppose it is a minimizing problem) [234]:

1. An easy (or straightforward) fitness landscape: FDC> 0.15, and FDC$= 1$ is the ideal case.

2. A deceptive (or unknown) fitness landscape: $0.15 >$FDC> -0.15

3. A misleading fitness landscape: FDC< 0.15.

FDC has been shown a reliable FL metric in the GP area [35, 227, 230, 231, 233] and an effective way to analyze the parameter configurations [225].

The *local optima network* is another metric that has to enumerate all possible solutions in the search space to identify the local optima and their

transitions [167, 168]. The local optima network visualizes an FL by a graph, in which vertices represent the local optima in the search space, and edges represent the transitions (and their probability) between vertices. The characteristics of the graph such as the number of vertices and edges, and the cliquishness of a cluster (i.e., a connected sub-graph) show the characteristics of the FL (e.g., the connectivity of local optima and their distributions). For example, Đurasevic et al. [52] used the local optima network to analyze the effectiveness of different configurations of dimensionally-aware GP, and He and Neri [75] used the local optima to show the distribution of local optima.

However, it is uneasy to apply the above two metrics to problems with large search spaces since they need to know global information beforehand (e.g., global optima and the fitness of all solutions) [223, 232]. To understand the FLs of GP benchmarks with large search spaces, existing studies developed several metrics that measure the hardness based on neighborhood information.

The metrics based on neighborhood information measure problem hardness and describe FLs based on the neighbors of sampled solutions. For example, *negative scope coefficient* (NSC) is a metric that measures the degree of "bad evolvability" (i.e., moving from good solutions to poor solutions) [229]. NSC first identifies the *fitness cloud* based on the fitness of sampled solutions and their neighbors [237]. The abscissas of the fitness cloud are the fitnesses of sampled solutions, and the ordinates of the fitness cloud are the fitnesses of their neighbors. Then, NSC partitions the fitness cloud into segments [235]. NSC gets the negative scopes among the mean values of segment abscissas and ordinates as its result. Normally, NSC is less than zero, and its absolute value indicates the degree of hardness.

Hu et al. [82, 83, 85, 86] analyzed FLs of GP based on *robustness*, *evolvability*, and *accessibility* of three levels: genotypes, phenotypes, and fitness.

Specifically, genotypic robustness indicates the fraction of neutral moves[2] caused by point mutations for a given genotype, genotypic evolvability indicates the proportion of non-neutral moves from a given genotype (or a phenotype), and phenotypic accessibility indicates the propensity of mutating into a certain phenotype. Their results imply that robustness and evolvability are negatively correlated at the genotypic level. Robust genotypes are normally hard to move into genotypes with another phenotype by a point mutation. Galván-López et al. [65] proposed to use *locality* [201] to measure the consistency between genotypic and phenotypic neighborhood structures. They assume that an FL with better consistency between genotypic and phenotypic neighborhood structures is easier for GP to search for good solutions.

To sample solutions from FLs, several metrics perform a random walk on FLs. For example, Kinnear [105] used a *landscape autocorrelation* to measure the correlation of fitness over the random walk on an FL. Slaný and Sekanina [209] proposed two quantity metrics for *ruggedness* and *smoothness* based on the entropy of fitness over the random walk. However, these random walk-based metrics are not accurate enough to predict algorithm performance since they are highly dependent on the starting point of walking in many problems and they are hard to perform *importance sampling* (more weight to sample good solutions) [228].

Improving Fitness Landscapes by Fitness Function

Although few studies explicitly optimize GP's FLs, many existing GP studies are essentially improving the FLs. Designing better fitness functions is a method that improves FLs. For example, in symbolic regression problems, Haut et al. [74] proposed to use R-square, a measure of the correlation coefficient, as the fitness function to encourage GP to produce more concise and less overfit formulas. Chen et al. [28] further verified the

[2]A neutral move is a movement between two solutions with the same fitness on FLs. A non-neutral move is a movement between two solutions with different fitnesses.

effectiveness of applying linear scaling with R-square in GP for symbolic regression tasks.

However, designing better fitness functions for a specific domain is non-trivial and tedious. To construct potentially better fitness functions, multitask optimization [173] proposes to simultaneously optimize several similar tasks, expecting that these similar tasks have synergistic fitness functions. Multitask optimization mutually exchanges the search information among the fitness landscapes with similar fitness functions to enhance search performance. A detailed review of multitask optimization is given in section 2.2.7.

Simultaneously optimizing several alternative formulations for a single problem is another way to build up similar fitness landscapes, so-called multiform optimization. For example, Da et al. [40] formulated a traveling salesman problem into a single-objective and a multi-objective optimization problem respectively, and solved the two optimization problems via a multitask optimization method. Since multi-objective formulations often introduce plateaus into fitness landscapes, it is expected that the multi-objective optimization task can remove some local optima from the single-objective formulation. Additional formulations can also be constructed by adding or relaxing constraints on the optimization problem [97], which is equivalent to constructing different search spaces (and hence different fitness landscapes) for solving a single task.

Sharing the GP search information among similar tasks is helpful to GP evolution. Existing studies of multitask GP have shown great potential in combinatorial optimization problems [257,265], symbolic regression [281], and classification problems [17,242]. A more detailed literature review on multitask GP is given in section 2.2.7. From a broader view, since fitness functions exert evolution pressure on GP by selection operators, advancing selection operators is an alternative way to improve fitness functions of GP [76,275].

Improving Fitness Landscapes by Solution Spaces

Each GP individual is a solution of the GP search, and GP representations directly determine the solution space of the GP search. There have been many GP representations [57, 141, 214]. For example, with the same primitive set, tree-based and linear representations have very different solution spaces because of the different topological structures of primitives [21,106]. These representations have pros and cons for different problems. TGP is good at paralleling building block computation, while LGP is good at reusing building blocks. To reduce redundant solutions from solution spaces, existing studies also apply feature selection [30, 262] and grammar-guided techniques [61, 188] to GP solution spaces.

Using different GP solution representations to construct related fitness landscapes is a potential method to make use of the advantages of solution representations. However, this research direction is not well investigated. Although some studies of evolutionary multitask optimization have cooperated with different solution representations [55, 56], their solution representations are mainly designed based on decision variables. The symbolic representations of GP are much more complex than the numerical representations in existing evolutionary multitask optimization methods, which are non-trivial to share directly.

Improving Fitness Landscapes by Neighborhood Structures

Neighborhood structures define the neighborhood relationship among solutions. Changing neighborhood functions to reshape the fitness landscape can construct related fitness landscapes in the search for effective solutions. One representative example is variable neighborhood search [149], which switches neighborhood functions in the course of the search. By searching within different neighborhoods, variable neighborhood search can reach distant solutions via local search and has a better chance to jump out from a local optimum. Variable neighborhood search is an effective

strategy to enhance other search techniques [26].

The neighborhood structures of existing GP methods are essentially genetic operators. Two neighboring GP solutions can reach each other by performing the genetic operator once. There are a huge number of existing studies proposing various genetic operators for GP in different domains, such as the genotype-based [259], phenotype-based [84,215], and semantics-based [183, 185], and so on.

To conclude the three sub-sections above, although there have been many studies analyzing and improving the FL of GP, the analyses of FLs of GP are mainly based on TGP and the domains excluding combinatorial optimization problems. Besides, the improvements for FLs all need a lot of domain knowledge and manual tuning (e.g., replacing root mean square error with R-square in symbolic regression and designing new genetic operators). It is hard and expensive to refine FLs and extend these improvements to other domains. This thesis will apply existing FL metrics to analyze the FLs of LGP for DJSS and propose an FLO method to automatically improve the FLs of LGP.

2.2.7 GP in Multitask Optimization

Evolutionary Multitask Optimization

Evolutionary multitask optimization is an emerging topic that enhances the performance of evolutionary computation methods by simultaneously optimizing multiple similar tasks [71]. Existing evolutionary multitask methods can be categorized into two paradigms, implicit and explicit genetic transfer [56, 241]. Methods with implicit genetic transfer exchange knowledge among tasks by applying a suite of genetic operators to perform implicit genetic mating (e.g., applying crossover operators on two parents from different tasks). One of the most typical methods with the implicit genetic transfer is the multifactorial evolutionary algorithm (MFEA) [71]. MFEA designs four new concepts: factorial cost, factorial rank, skill

factor, and scalar fitness. The first two terms are vectors for multiple tasks, denoting the fitness and rank in corresponding tasks. The scalar fitness is the minimum factorial rank among tasks. The skill factor indicates the task with the minimum factorial rank. An assortative mating and a vertical cultural transmission are also proposed in [71] to facilitate knowledge transfer. MFEA integrates multiple solutions into one individual based on the skill factor. However, most MFEA methods are designed based on numerical representation. The idea of integrating multiple numerical solutions in MFEA cannot be easily extended to GP methods whose search space is symbolic. Based on MFEA, many studies developed new techniques to enhance its performance. For example, Bali et al. [8] proposed a linear transformation strategy to transform the decision space among tasks. Ding et al. [46] developed a decision variable translation strategy and decision variable shuffling strategy for MFEA to improve the effectiveness of knowledge sharing. Zheng et al. [280] proposed a self-regulated method to perform knowledge sharing based on the relatedness among tasks. Besides, MFEA has been applied to many applications, such as capacitated vehicle routing problems [285], robot path planning problem [253], and bi-level optimization [70]. Gupta et al. [72] also extended MFEA to multi-objective optimization.

However, the implicit genetic transfer has a key limitation. It unnecessarily limits the information exchange within genetic mating [56]. In practice, genetic mating might not be effective enough to transfer knowledge among different tasks. To address this issue, the explicit genetic transfer methods are proposed to explicitly consider different representations and search mechanisms among tasks in knowledge transfer. For example, Feng et al. [31, 56] proposed to use an artificial neural network (e.g., denoising autoencoder) to perform knowledge transfer among different tasks. The artificial neural network is trained beforehand on uniformly sampled data. When performing knowledge transfer, solutions from one task are mapped to another space by the artificial neural net-

work. This method demonstrates the superior performance of explicit genetic transfer over implicit genetic transfer in the investigated single- and multi-objective optimization problems. To capture the essential features of different tasks, Tang et al. [221] transformed the distribution of sub-populations into task-specific low-dimension spaces, and made pairwise mapping among tasks by well-trained alignment matrices. When the mapping discrepancies among tasks are minimized, individuals from different tasks can be transferred to another task based on low-dimension spaces and alignment matrices. Chen et al. [33] treated decision spaces of different tasks as manifolds and projected the decision spaces of tasks to a joint manifold to represent the task relationship. The knowledge transfer among tasks is performed based on the latent task relationship. Kullback-Leibler divergence [32] and Naive Bayes classifier [116] are also extended as methods of selective knowledge transfer. Nowadays, multitask optimization techniques with explicit genetic transfer have been applied to some real-world applications, such as time series prediction [27] and capacitated vehicle routing problem [55].

Multitask GP

Multitask techniques have shown encouraging results in enhancing the performance of GP methods. For example, Zhong et al. [282] and Wei et al. [242] applied multitask techniques to gene expression programming for solving symbolic regression and multi-class classification. Bi et al. [18] proposed to use multitask techniques to encourage GP to construct effective features for image classification. These GP-constructed features from different tasks are concatenated together based on an ensemble method. Bi et al. [17] proposed to use a common tree structure to construct shared features in similar classification tasks.

Multitask GP has been applied to DJSS. Zhang et al. [257,260] proposed a multi-population-based multitask GP method, each sub-population for a task. These sub-populations share the search information by swapping the

building blocks of individuals. Their results verified the effectiveness of the explicit genetic transfer. To further improve the training convergency, surrogate models are also introduced in the multitask framework to selectively share effective knowledge of JSS [267]. The tasks in [267] only accept individuals that are effective in corresponding surrogate models.

Based on the review, we found that existing multitask GP methods are mostly designed based on TGP. Knowledge is transferred mainly by duplicating elite individuals or sub-trees from one task to another, which is inefficient. On the contrary, the multiple outputs in LGP and the common building blocks shared by these outputs provide another potential way to effectively share knowledge. This thesis will develop an LGP-based multitask method to further enhance the performance of existing multitask GP methods.

2.3 Chapter Summary

This chapter introduces the basic concepts of evolutionary computation, LGP, DJSS problems, and heuristics and hyper-heuristics which are fundamental knowledge of this thesis. The details of LGP and DJSS are provided, and how to apply LGP to learn heuristic rules for DJSS is given with examples. Meanwhile, this chapter reviews the existing advances in the LGP area and the approaches for JSS. The existing studies that use GPHH to evolve scheduling heuristics for DJSS are discussed. Based on the research objectives of this thesis, the related studies are summarized. The limitations of the existing studies are highlighted as follows.

1. LGP is a prominent GP variant. However, existing studies miss an important application of LGP that applies LGP as a hyper-heuristics method. With limited training resources and without labeled data, applying LGP as a hyper-heuristic method is greatly different from existing studies of LGP.

2. Graph-related characteristics of LGP are not fully investigated and utilized. In other words, existing studies of LGP miss the graph-to-instruction transformation and the cooperation among GP representations.

3. Existing LGP studies lack an effective way to incorporate domain knowledge (e.g., the dimensions of problem features and the preferred combinations of primitives) into LGP search. This prevents LGP from evolving sophisticated and interpretable dispatching rules for DJSS.

4. The improvement of LGP fitness landscapes brings performance gain, but existing studies have to manually manipulate the fitness landscapes, which is tedious and uneasy to extend.

5. LGP naturally has multiple outputs that share common building blocks. Although this feature is useful for multitask optimization, existing studies miss the investigation of LGP with multitask optimization.

In the following chapters (3 to 8), we will develop advanced LGP methods to address these limitations.

Chapter 3

Preliminary Investigation of Linear Genetic Programming for DJSS

This chapter develops an LGP-based hyper-heuristics (LGPHH) method for DJSS problems. Based on the LGPHH method, we perform a comprehensive investigation of the design details of LGP on DJSS. This chapter is a fundamental chapter for the following chapters.

3.1 Introduction

Although there have been many GPHH methods for DJSS problems [156, 273], most of the existing studies of GPHH for DJSS are based on TGP. It is non-trivial to extend the existing TGP-based studies to LGP because of the linear representation and corresponding genetic operators in LGP. There are three research questions in designing LGPHH methods for DJSS. First, The proper settings for evolving LGP in the generational evolutionary framework and with limited fitness evaluations are unknown. The recommended evolutionary framework in DJSS is the generational evolutionary framework because of the limited training resources in DJSS

problems [78]. However, most of the existing LGP studies evolve within a steady-state evolutionary framework and set a large number of fitness evaluations. The inconsistent evolutionary frameworks and the number of fitness evaluations likely lead to different suitable settings. Second, the effective register initialization strategy for LGPHH is unknown. The register initialization is a unique step for applying LGP, but existing GPHH studies did not investigate the register initialization strategies for LGPHH. Third, how different performance LGP can achieve compared to existing methods is unknown. The advantages of LGP need to be verified by the comparison with existing tree-based GPHH methods.

To answer the three research questions, we investigate the following four aspects of basic LGPHH. First, we investigate the proper settings of the population size and the number of generations for an LGPHH based on the generational evolutionary framework, given the same total number of fitness evaluations. Since the number of generations has a high correlation with variation step size (i.e., a small number of generations might need a large variation step size), we perform a grid search on the proper settings between variation step size and the number of generations.

Second, we investigate the initialization strategy of registers. Brameier et al. [21] have shown that initializing registers by valuable input features significantly improves LGP performance in classification and symbolic regression problems. However, DJSS problems have a large number of input features such as the processing time of an operation and the remaining workload in a certain machine. It is still an open question how to set the initial value of the registers which are much fewer than the number of features.

Third, we verify the performance of LGPHH by comparing it with basic TGP [106, 107]. Specifically, we verify the effectiveness, efficiency, and training time of LGPHH. To ensure the fairness of the comparison between LGP and TGP, we also carefully tune the parameters of TGP.

Finally, we analyze the interpretability of LGP output rules by com-

paring the dispatching rules evolved by LGP and TGP, respectively. We analyze the interpretability based on the program size and example rules.

3.1.1 Chapter Goals

The goal of this chapter is to *investigate and identify effective settings for LGPHH methods for solving DJSS problems by answering the three research questions mentioned above*. The LGPHH with well-tuned settings is expected to have better effectiveness and efficiency than the tree-based GPHH. Specifically, this chapter has the following research objectives:

1. Apply a basic LGPHH method to DJSS problems based on a generational evolutionary framework.

2. Investigate the proper parameter settings of the variation step size and the number of generations of LGPHH.

3. Investigate the register initialization strategies of LGPHH for solving DJSS problems

4. Verify the performance of LGPHH by comparison with tree-based GPHH.

5. Analyze the interpretability of the evolved dispatching rules.

3.1.2 Chapter Organization

The rest of this chapter is organized as follows. Section 3.2 applies an LGPHH method to DJSS problems. Sections 3.3 and 3.4 give the experiment design and the empirical results, respectively. We mainly investigate the detailed design of LGPHH in sections 3.3 and 3.4. Section 3.5 further analyzes the interpretability of output dispatching rules. Finally, section 3.6 concludes this chapter.

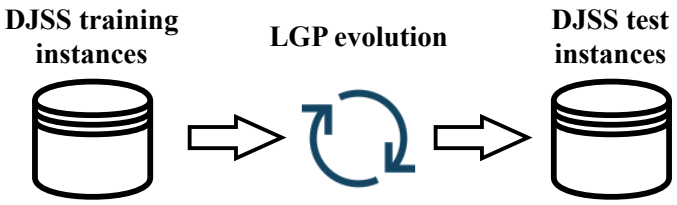

Figure 3.1: The schematic diagram of applying LGPHH to DJSS problems.

3.2 LGPHH for DJSS

3.2.1 Algorithm Description

The overall procedure of applying LGPHH for solving DJSS problems is shown in Fig. 3.1. There are two sets of DJSS instances, one for training and the other for testing. These DJSS instances are generated by unique random seeds of the simulation. LGP evolves dispatching rules based on the training instances. A DJSS instance is a series of jobs that come into the job shop by sampling from a certain distribution. A unique random seed of the distribution ensures the unique DJSS instances. The attributes of these jobs will also follow the settings of the DJSS scenario. The LGP-evolved dispatching rules schedule operations in DJSS instances and treat the overall performance of the job shop as the fitness of LGP individuals. For example, the maximum tardiness and mean flowtime are common job shop performance and fitness functions of LGP individuals in DJSS problems. The best LGP individual after evolution is obtained, and its performance is evaluated on test instances. LGP cannot know the test instances during training, and the test instances are different from training instances.

The pseudo-code of LGPHH based on a generational evolutionary framework for solving DJSS is shown in Alg. 1[1]. First, an LGP popula-

[1] $\texttt{Uniform}(a, b)$ and $\texttt{UniformInt}(a, b)$ return a random floating point number or a random integer between $[a, b)$, respectively. The notations are used in the rest of this

Algorithm 1: The pseudo-code of LGPHH for DJSS

Input: The settings of the DJSS simulation (objective, utilization level, number of machines, and number of jobs and operations.), the parameters of LGPHH (population size, *crossover rate*, *macro mutation rate*, *micro mutation rate*, and tournament selection size).

Output: The best-of-run LGP individual $\mathbf{f}^*$.

```
Initialize the population P with a given size;
Generate a DJSS instance I based on the settings of the simulation;
Evaluate every individual f on I by a DJSS simulation;
while The stopping criteria are not satisfied do
    Parent individuals f1, f2 ← ∅, offspring f1', f2' ← ∅;
    The new population P' ← elite individuals of P;
    while size of P' < size of P do
        f1, f2 ← TournamentSelection(P);
        p ← uniform(0, 1);
        if p ≤ crossover rate then
            Offspring f1', f2' ← Crossover(f1, f2);
        else if p ≤ crossover rate+macro mutation rate then
            f1' ← MacroMutation(f1);
        else if p ≤ macro mutation rate+crossover rate+micro mutation rate then
            f1' ← MicroMutation(f1);
        else
            f1' ← f1;
        P' ← P' ⋃ f1' (or P' ← P' ⋃ {f1', f2'});
    P ← P';
    Evaluate every individual f ∈ P on I by a DJSS simulation;
    Update f* ∈ P;
    Rotate the random seed of I.
Return f*.
```

tion $\mathbf{P}$ is initialized. The fitness of LGP individuals is evaluated by a DJSS simulation. Then based on the fitness, an elitism selection is used to retain elite individuals to the next generation. In breeding, LGP parent individ-

thesis.

uals are selected by a tournament selection. These parents are varied to produce offspring by micro mutation, macro mutation, and crossover respectively based on the given probability. These genetic operators follow the basic operators introduced in Chapter2.1.2. The offspring form the new population and the best-of-run individual is updated. To improve the generalization ability of dispatching rules, DJSS instances are also rotated for every generation (i.e., altering the random seed of the simulation instance). After generations of evolution, the best individual of the population is outputted.

An LGP individual is evaluated by a DJSS simulation. In the simulation, the LGP individual is regarded as a dispatching rule. The dispatching rule is used to prioritize available jobs or machines for making decisions. Specifically, for LGP-based dispatching rules, the registers in LGP individuals are re-initialized before every execution. Then, the instructions in LGP individuals are executed sequentially, and the final output is seen as the priority of a certain candidate decision. Without loss of generality, smaller priority values indicate better priority in this thesis. The decision with the best priority is then executed. After all the decisions have been made, the performance of the generated schedule is measured by a predefined objective function. The value of the objective function is regarded as the fitness of the LGP individual.

3.3 Experiment Design

3.3.1 Parameter Settings

We define the default settings for LGP parameters based on the recommended settings in [21]. Specifically, every LGP individual initially has 1 to 10 instruction(s) and can have at most 64 instructions and at least 1 instruction during evolution. For micro mutation, the mutation rates of different primitive types (i.e., function, constant, destination and source

Table 3.1: The DJSS attributes.

Name	Description
NIQ	the number of operations in the queue
WIQ	the total processing time of operations in the queue
MWT	the waiting time of the machine
PT	the processing time of the operation
NPT	the processing time of the next operation
OWT	the waiting time of the operation
NWT	the waiting time of the next operation
WKR	the total remaining processing time of the job
NOR	the number of remaining operations of the job
WINQ	total processing time of operations in the queue of the machine which specializes in the next operation of the job
NINQ	number of operations in the queue of the machine which specializes in the next operation of the job
rFDD	relative flow due date from the current time
rDD	relative due date from the current time
W	the weight of the job
TIS	time in the system since the arrival
SL	slack

registers) are set as $\mathrm{fun} : \mathrm{destin} : \mathrm{cons} : \mathrm{source} = 50\% : 25\% : 12.5\% : 12.5\%$. For macro mutation, the insertion and deletion rate are set as 67% and 33% respectively. For the linear crossover, the maximal length of the two crossover segments is 30, the maximal length difference of the two segments is 5, and the maximal distance of the two crossover points is 30. The reproduction rate is set to 10%. The tournament size is set as 7 and the elitism rate is 1%. There are a total of 51200 simulations (i.e., fitness evaluation) by default. The function set includes $\{+, -, \times, \div, \max, \min\}$ where $\div$ returns 1.0 if divided by 0.0, and have six calculation registers and sixteen DJSS attributes (i.e., constant registers) as the terminal set. The DJSS attributes are defined based on the existing study [137], as shown in table 3.1.

Every DJSS scenario has 30 independent runs. In the training phase, the individuals in one generation are evaluated by one simulation instance. The simulation instance changes over generations. The outputted dispatching rules from different runs are tested on 50 unseen instances to get the test performance [78].

3.4 Experiment Results

3.4.1 Variation Step Size and Generations

To investigate the correlation between the variation step size and the number of generations, we conduct a grid search on these two settings. Intuitively, when the number of generations is large, the LGP population can have a more thorough search in the neighbor region of existing solutions by small variation step sizes. In this case, elite solutions can be further elaborated by small changes. On the other hand, when there are a small number of generations, LGP population can search more different regions in solution space by large variation step sizes and thus has a high degree of exploration. Here, we select two settings of generations, 50 and 400, to represent small and large numbers of generations, respectively. To retain the same number of total simulations, the population size of LGP is adjusted accordingly, i.e., the total number of simulations divided by the number of generations. To fulfill the different variation step sizes, we design two crossover-dominated and mutation-dominated settings. The crossover-dominated evolution has a large variation step size since the crossover operator changes more than one LGP instruction to produce offspring. Contrarily, the mutation-dominated evolution has a small variation step size since the mutation operator changes at most one instruction each time. Denoting micro mutation rate as θ_{micro}, macro mutation rate as θ_{macro}, and crossover rate as θ_{cross}, we set the genetic operator rate $\theta_{micro} : \theta_{macro} : \theta_{cross} = 10\% : 10\% : 70\%$ for crossover-dominated

settings, and set $\theta_{micro} : \theta_{macro} : \theta_{cross} = 30\% : 30\% : 30\%$ for mutation-dominated settings.

The results of the grid search are shown in table 3.2. A Wilcoxon rank-sum test with a significance level of 0.05 is also applied to these results. The notation "$+$", "$-$", and "$\approx$" respectively denote the compared algorithm is significantly better than, significantly worse than, or similar to the baseline (i.e., generation=50 and crossover-dominated in this experiment). It can be observed that LGP has a significant improvement when exploitation is highlighted. When generations equal to 400 and LGP adopts mutation-dominated evolution, LGP is significantly better than "(crossover-dominated, generations=50)" in four of the six scenarios. Besides, if looking at the "crossover-dominated" row and "generations=50" column respectively, we can see that increasing exploitation, no matter by increasing generations to allocate more training resources for the convergent phase or letting mutation dominate evolution, is helpful to LGP. They significantly improve one or two scenarios respectively.

To identify a proper setting of generations for LGP, we make a further investigation on the number of generations. We conduct two more generation settings, 200 and 800, based on the mutation-dominated setting. Table 3.3 shows that when the number of generations equals to 200 or 400, LGP is significantly better than the baseline setting (i.e., "generations=50") in four of the six scenarios. On the other hand, extremely increasing the number of generations of LGP has a negative impact on the performance. LGP with 800 generations only has a superior performance to the baseline in the two $Tmax$ scenarios. Thus, the number of generations of LGP is suggested to set as 200 or 400 to strike a good exploration-and-exploitation balance.

However, increasing generations increases the training time. As shown in table 3.4, the training time grows up consistently from "generation=50" to "generation=800". The main reason of the increasing training time is the increase in program size. When LGP evolves a large population for a small

Table 3.2: Mean (standard deviation) test performance of LGP with different generations and genetic operator rates.

Scenarios		generations=50 popsize=1024	generations=400 popsize=128
⟨Tmax, 0.85⟩	crossover-dominated	2026.2(86.61)	1928.61(41.68)+
⟨Tmax, 0.95⟩		4179.05(231.24)	4003.52(119.78)+
⟨Tmean, 0.85⟩		419.14(3.22)	419.91(6.11)≈
⟨Tmean, 0.95⟩		1120.39(9.18)	1115.7(8.62)≈
⟨WTmean, 0.85⟩		728.94(7.26)	727.55(7.1)≈
⟨WTmean, 0.95⟩		1746.12(24.47)	1735.12(32.14)≈
⟨Tmax, 0.85⟩	mutation-dominated	1987.46(60.36)+	1953.52(59.54)+
⟨Tmax, 0.95⟩		4109.88(150.49)≈	3999.86(158.62)+
⟨Tmean, 0.85⟩		418.15(2.43)≈	417.16(2.95)+
⟨Tmean, 0.95⟩		1117.89(6.24)≈	1116.51(10.76)≈
⟨WTmean, 0.85⟩		730.6(7.03)≈	726.32(8.02)≈
⟨WTmean, 0.95⟩		1753.39(28.34)≈	1725.95(26.13)+

Table 3.3: Mean (standard deviation) test performance of different generation settings with mutation-dominated LGP.

Scenarios	generations=50 popsize=1024	generations=200 popsize=256	generations=400 popsize=128	generations=800 popsize=64
⟨Tmax, 0.85⟩	2026.2(86.61)	1918.58(39.69)+	1953.52(59.54)+	1943.39(54.17)+
⟨Tmax, 0.95⟩	4179.05(231.24)	3995.55(173.69)+	3999.86(158.62)+	3977.28(120.35)+
⟨Tmean, 0.85⟩	419.14(3.22)	418.14(4.58)+	417.16(2.95)+	419.89(5.16)≈
⟨Tmean, 0.95⟩	1120.39(9.18)	1116.5(11.7)≈	1116.51(10.76)≈	1120.11(16.63)≈
⟨WTmean, 0.85⟩	728.94(7.26)	727.13(6.78)≈	726.32(8.02)≈	728.02(11.15)≈
⟨WTmean, 0.95⟩	1746.12(24.47)	1733.26(26.61)+	1725.95(26.13)+	1735.83(31.43)≈

number of generations, there are few long individuals in the LGP population because of the limited number of generations and limited variation step size. Contrarily, LGP individuals grow long enough after generations of evolution. Long dispatching rules inevitably increase decision time and make DJSS simulation more time-consuming.

Table 3.4: Mean (standard deviation) training time of different generations with mutation-dominated LGP (seconds).

Scenarios	generations=50 popsize=1024	generations=200 popsize=256	generations=400 popsize=128	generations=800 popsize=64
⟨Tmax, 0.85⟩	2483.7(65)	4129.9(115.8)	4360.1(67.8)	4449.9(55.6)
⟨Tmax, 0.95⟩	5641.9(209.9)	10089.4(228.1)	11311.3(246.3)	11710.7(234.2)
⟨Tmean, 0.85⟩	2262.8(41.1)	3870.1(117.3)	4050.2(75.7)	4081.6(73.8)
⟨Tmean, 0.95⟩	4782.7(121.2)	9391.6(341)	9792.1(366.2)	10580.6(443.6)
⟨WTmean, 0.85⟩	2289.9(58.8)	4341.6(138.7)	4258.4(74.3)	4478.2(113)
⟨WTmean, 0.95⟩	4928.4(174.4)	9087.3(310.6)	9550.9(238.8)	9758.1(297)

To verify the program size increase, we compare the average program size of the LGP population over generations with different generation settings. Specifically, we denote the number of effective instructions in an individual as the program size of LGP and use "mut-X", "mut" denoting mutation and "X" for generation settings, to denote evolution settings. As shown in Fig. 3.2, the curves of small generations climb up much slower than the ones of large generations. Based on the results, it is recommended to set the number of generations as 200 and use mutation-dominated settings to balance the training efficiency and test performance.

3.4.2 Register Initialization Strategy

This section investigates the effectiveness of different register initialization strategies. By default, we initialize the six registers by the first six DJSS attributes in table 3.1. However, the first three attributes are machine-related. They are the same for the operations in the same machine, which is helpless in prioritizing operations in the same machine queue. To find an effective register initialization strategy, we compare three strategies in this section. First, the default initialization strategy acts as the baseline. For the second strategy, we use diverse job-related attributes to initialize registers. This strategy encourages dispatching rules to consider differ-

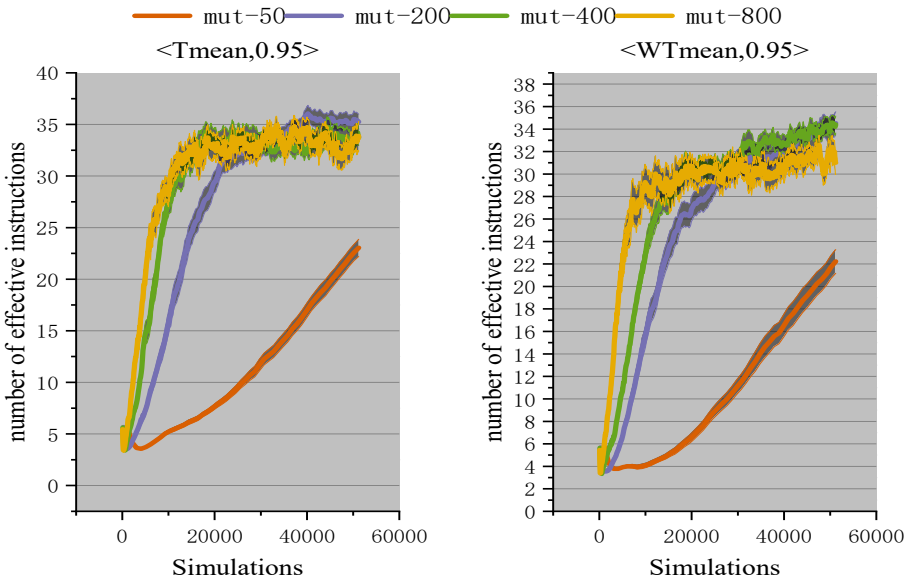

Figure 3.2: Average program sizes of different settings over generations.

ent information about available jobs and ensures that the initial values are at least helpful in distinguishing operations. We identify the diverse attributes based on the domain knowledge. The attribute pairs such as WKR and NOR, rFDD and rDD, WINQ and NINQ, are respectively regarded as similar attributes. So, we simply put one of the attributes in these pairs at the front of the list and leave the other to the end part. The attributes listed in table 3.1 are thus rearranged into

$$\{\mathrm{PT}, \mathrm{NPT}, \mathrm{WINQ}, \mathrm{WKR}, \mathrm{rFDD}, \mathrm{OWT}, \mathrm{NOR}, \mathrm{NINQ},$$
$$\mathrm{W}, \mathrm{rDD}, \mathrm{NWT}, \mathrm{TIS}, \mathrm{SL}, \mathrm{NIQ}, \mathrm{WIQ}, \mathrm{MWT}\}.$$

The first k attributes are used to initialize k registers respectively. This diverse job-related attribute initialization is denoted as "DivJob". Third, we initialize all registers as "1", which is the simplest way for initialization. Other experiment settings follow section 3.3.1 and mutation-dominated

Table 3.5: Mean (standard deviation) test performance of different register initialization strategies.

Scenarios	Default	DivJob	All-ones
⟨Tmax, 0.85⟩	1953.52(59.54)	1914.35(34.99)+	1952.21(45.28)≈
⟨Tmax, 0.95⟩	3999.86(158.62)	3973(135.48)≈	3963.31(81.94)≈
⟨Tmean, 0.85⟩	417.16(2.95)	418.04(3.88)≈	418.01(4.03)≈
⟨Tmean, 0.95⟩	1116.51(10.76)	1112.16(8.65)≈	1117.18(13.95)≈
⟨WTmean, 0.85⟩	726.32(8.02)	727.72(6.04)≈	728.45(8.47)≈
⟨WTmean, 0.95⟩	1725.95(26.13)	1737.68(39.06)≈	1725.63(27.92)≈

settings, evolving 400 generations.

The test performance of the three initialization strategies is shown in table 3.5. Generally speaking, the test performance of the three strategies is very similar, which means LGP is relatively robust to register initialization in terms of test performance. Nevertheless, DivJob still significantly outperforms the other two initialization strategies in one scenario. Besides, DivJob also has a better mean performance than the default initialization on two of the three scenarios with high utilization levels.

To further investigate the effectiveness of initialization strategies, the training performance is also compared. Fig. 3.3 shows the test performance of the best dispatching rule of every generation. As shown in Fig. 3.3, DivJob (i.e., the red curves) averagely drops down faster and deeper than the other two initialization strategies in the two $Tmax$ scenarios. Besides, for the two $Tmean$ scenarios, DivJob has a relatively fast convergence speed at the beginning and has a more stable performance (i.e., narrower bias range) at the final stage of evolution. Though in $WTmean$ scenarios, DivJob has a slower convergence speed than the others at the beginning of evolution, DivJob can still converge to a similar level with other initialization strategies before half of the evolution. Based on these results, we find that initializing registers as various job-related attributes is helpful for LGP in terms of both training and test performance.

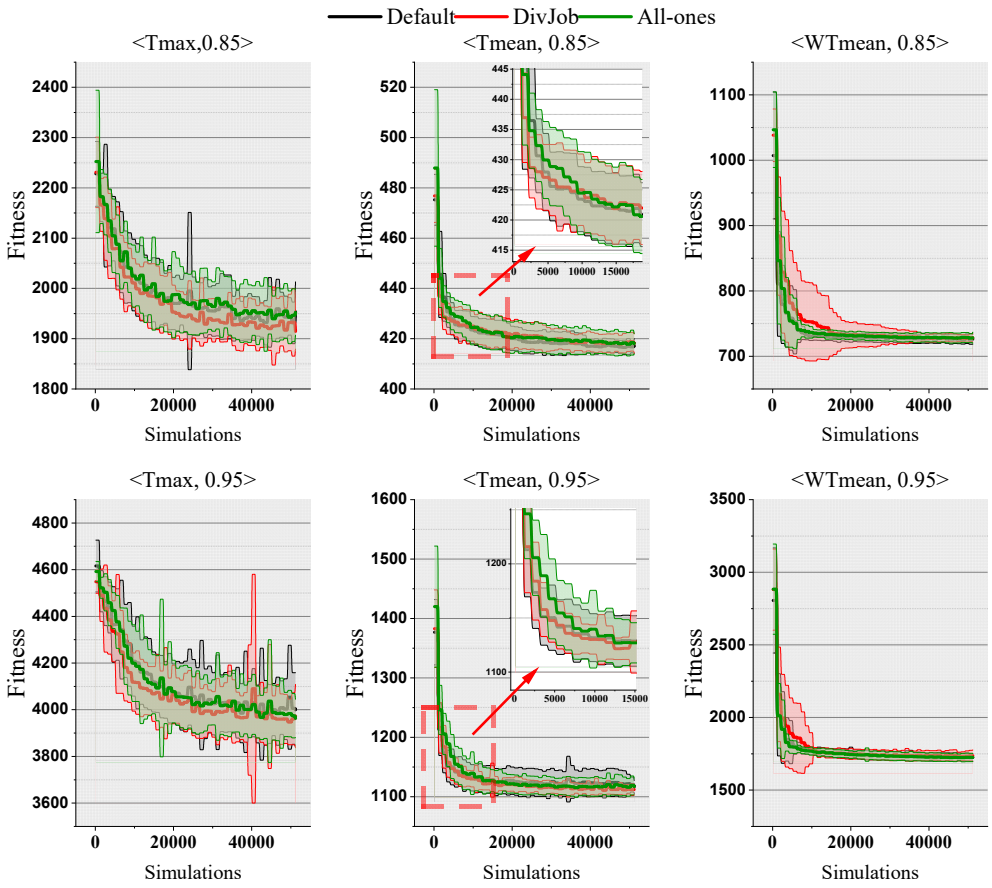

Figure 3.3: Test performance over generations.

Given that the number of registers is often smaller than the number of DJSS attributes, we conduct an investigation on the suitable number of registers based on the newly recommended initialization strategy (i.e., DivJob). The comparison of average test performance and standard deviation is shown in table 3.6. The default setting (i.e., 6 registers) is regarded as the baseline. The test performance of different numbers of registers is similar in most cases. However, when the number of registers is set as 8, LGP outperforms the default setting in ⟨WTmean, 0.85⟩. Besides, LGP with 8 registers can also have relatively small mean test performance in most high-utilization-level scenarios. Based on the results, we recom-

Table 3.6: Mean (standard deviation) test performance of different number of registers.

Scenarios \ #registers	4	8	10	12
⟨Tmax, 0.85⟩	1918.65(29.99)≈	1927.03(50.67)≈	1944.89(53.67)−	1943.65(63.67)≈
⟨Tmax, 0.95⟩	3974.87(154.94)≈	3938.6(117.75)≈	3975.27(102.66)≈	3993.53(168.22)≈
⟨Tmean, 0.85⟩	418.63(3.98)≈	418.06(4.27)≈	416.43(3.18)≈	417.84(2.26)≈
⟨Tmean, 0.95⟩	1115.59(9.97)≈	1115.77(9.37)≈	1115.55(11.13)≈	1116.11(9.92)≈
⟨WTmean, 0.85⟩	724.42(7.1)≈	721.59(6.02)+	726.69(6.51)≈	726.75(6.95)≈
⟨WTmean, 0.95⟩	1725.07(22.07)≈	1729.49(22.2)≈	1731.1(20.1)≈	1742.57(30.17)≈

*The test performance of 6 registers with new register initialization strategy is referred to DivJob in table 3.5.

mended to set the number of registers as 8.

3.4.3 Comparison with TGP

As it is the very beginning of applying LGP in DJSS, we verify the effectiveness of LGPHH by a comparison between basic TGP [137] and LGP in terms of test performance and training efficiency, to show the potential of LGP. The parameters of TGP and LGP are shown in table 3.7. The parameters of TGP and LGP are different because they have their best performance in different settings. We apply their own fine-tuned parameters to ensure a fair comparison. We verify the effectiveness of LGPHH with the recommended settings of generations (i.e., 200) and register initialization strategy (i.e., DivJob and 8 registers). We reduce the maximum program length to 50 in this section and use a smaller variation step size in linear crossover. The parameters of TGP are set based on the recommended parameters of [136]. To retain the same total number of simulations, TGP in the experiment has a population of 1024 individuals and evolves 50 generations.

The results are shown in table 3.8. It can be observed that LGP is sig-

Table 3.7: Default parameter settings of all the compared methods.

Parameters	TGP	LGP
population size	1024	256
generations	50	200
genetic operator rates	crossover 80%,mutation 15%, reproduction 5%	crossover 30%, macro mutation 30%, micro mutation 30%, reproduction 10%
crossover parameters	inner node 90%, leaf node 10%	segment length$\leq$30, segment length difference$\leq$5, crossover point distance$\leq$30
mutation parameters	inner node 90%, leaf node 10%	macro(insertion 67%, deletion 33%), micro ($\theta_{fun} = 50\%$, $\theta_{con} = 12.5\%$ $\theta_{des} = 25\%$, $\theta_{sou} = 12.5\%$,)
initial program size	min depth=2, max depth=6	min instruction=1, max instruction=10
maximum program size	max depth=8	max instruction=50
register number	None	8

Table 3.8: Mean (standard deviation) test performance and training time (in seconds) of TGP and LGP.

Scenarios	TGP		LGP	
	Fitness	Training time	Fitness	Training time
$\langle$Tmax, 0.85$\rangle$	1931.04(44.39)	3036.1(81.9)	1931.14(37.22)$\approx$	3483.5(52.4)
$\langle$Tmax, 0.95$\rangle$	4105.46(193.07)	6386.1(209.4)	3974.2(112)+	8759.4(205.4)
$\langle$Tmean, 0.85$\rangle$	417.34(2.95)	2398.6(67)	417.3(2.15)$\approx$	3480.7(104.9)
$\langle$Tmean, 0.95$\rangle$	1115.64(10.99)	4543.8(131.1)	1115.53(9.45)$\approx$	7727.3(252)
$\langle$WTmean, 0.85$\rangle$	727.93(9.14)	2544.7(80.9)	723.96(6.35)$\approx$	3273.6(72.4)
$\langle$WTmean, 0.95$\rangle$	1745.24(25.5)	4687.7(116)	1728.6(24.1)+	8132.6(279.9)

nificantly better than TGP in two scenarios, in terms of test performance. Besides, LGP has better mean performance than TGP in most of the other scenarios. However, the results show that the training time of LGP is much longer than the ones of TGP. To be more convincing, we make a further comparison to investigate the performance difference between TGP and

LGP within a similar training time. Specifically, we respectively evolve TGP with 70 and 100 generations. Table 3.9 shows the comparison in terms of test performance and training time. TGP with 70 and 100 generations are denoted as "TGP70" and "TGP100" respectively. Table 3.9 shows that increasing the number of generations of TGP can improve the performance of TGP. However, the performance of TGP70 and TGP100 is still similar to the one of LGP. Besides, increasing the number of generations also increases the training time of TGP. The two versions of TGP both have longer training time than LGP in most scenarios.

To verify the training efficiency of LGP, we compare the test performance over generations of TGP, TGP70, TGP100, and LGP. As shown in Fig. 3.4, LGP (i.e., red curves) has a very competitive performance with the other three compared methods. Besides, LGP also drops down much faster than its adversaries in $\langle$Tmax, 0.95$\rangle$.

The results verify that within the same number of simulations, LGP has a significantly better learning ability than TGP (i.e., better effectiveness and efficiency). Given the same training resources, the training efficiency and test performance of LGP are still very competitive with TGP.

3.5 Rule Interpretability

3.5.1 Program Size

We analyze the interpretability of the output dispatching rules by the program size and example rules. Program size of dispatching rules is a common metric to analyze the interpretability. A shorter dispatching rule often has better interpretability. Fig. 3.5 compares the program sizes of output dispatching rules from the compared methods in all the independent runs. The number of tree nodes after simplification is denoted as the program size of TGP rules, and the number of effective instructions multiplied by a factor of 2.0 is denoted as the size of LGP rules. We see that LGP (i.e., the

Table 3.9: Mean (standard deviation) test performance and training time of TGP with 70 and 100 generations and LGP.

Test performance			
Scenarios	TGP70	TGP100	LGP
⟨Tmax, 0.85⟩	1918.87(41.5)≈	1920.31(50.86)≈	1931.14(37.22)
⟨Tmax, 0.95⟩	4012.22(88.63)≈	3985.49(85.69)≈	3974.2(112)
⟨Tmean, 0.85⟩	416.8(3.18)≈	416.03(3.04)≈	417.3(2.15)
⟨Tmean, 0.95⟩	1112.65(11.19)≈	1114.02(15.64)≈	1115.53(9.45)
⟨WTmean, 0.85⟩	726.09(5.52)≈	725.21(6.58)≈	723.96(6.35)
⟨WTmean, 0.95⟩	1730.11(27.46)≈	1726.83(27.84)≈	1728.6(24.1)
Training time (seconds)			
⟨Tmax, 0.85⟩	4868.7(185)	7714.2(260.7)	3483.5(52.4)
⟨Tmax, 0.95⟩	10356.5(313.4)	15809.1(514.5)	8759.4(205.4)
⟨Tmean, 0.85⟩	4014.8(80.9)	5698.9(156.5)	3480.7(104.9)
⟨Tmean, 0.95⟩	7695.1(280.9)	11653.8(391)	7727.3(252)
⟨WTmean, 0.85⟩	3948.5(90.5)	5507.1(124.6)	3273.6(72.4)
⟨WTmean, 0.95⟩	7692.7(294.5)	11637.7(386.3)	8132.6(279.9)

red boxes) has smaller means and medians than the TGP70 and TGP100 (i.e., the yellow and blue boxes) which has similar effectiveness. Although TGP (green boxes) has a similar program size distribution to LGP, the effectiveness of TGP with 50 generations is inferior to LGP. It means that LGP is more likely to evolve more compact and effective programs than TGP methods

3.5.2 Example Rules

To further analyze the interpretability, we analyze an example program of LGP. Fig. 3.6 (and Fig. 3.7) shows an LGP-evolved rule for ⟨Tmax, 0.85⟩, after some manual simplification. We can see that the rule reuses a building block A, which tries to minimize "SL", "PT", and "NIQ". It implies that the operations that are going to be late or have a short pro-

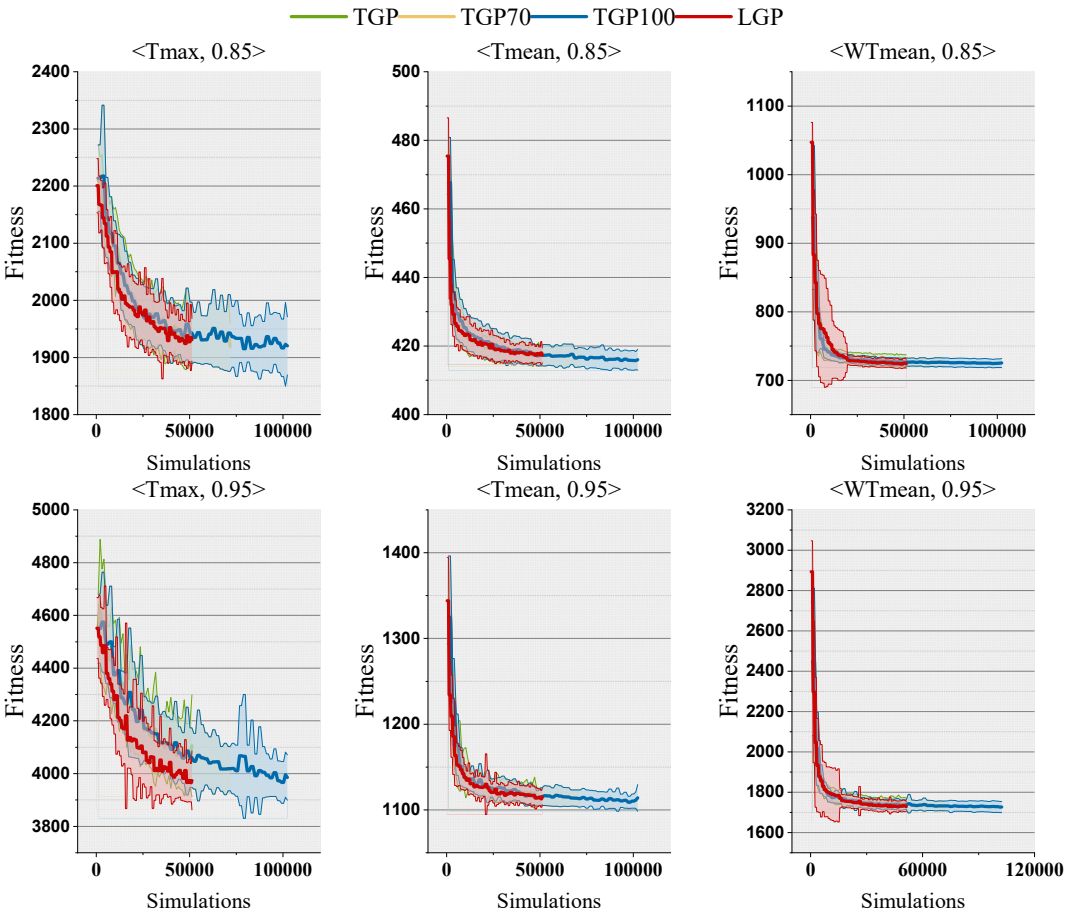

Figure 3.4: Test performance over generations of TGP and LGP.

cessing time will be scheduled first. Besides, the building block tries to maximize the remaining processing time of the job so that the jobs with a large amount of uncompleted work will have a high priority in reducing the maximum tardiness. The building block is reused in a maximum comparison. However, these two terms in the maximum comparison still show favor to "$-$WKR" and "PT", which is consistent with the building block. Given that "$-$WKR" is always non-positive and "PT-2" is positive in most cases, the dispatching rule can be further simplified to the latter element of the maximum comparison.

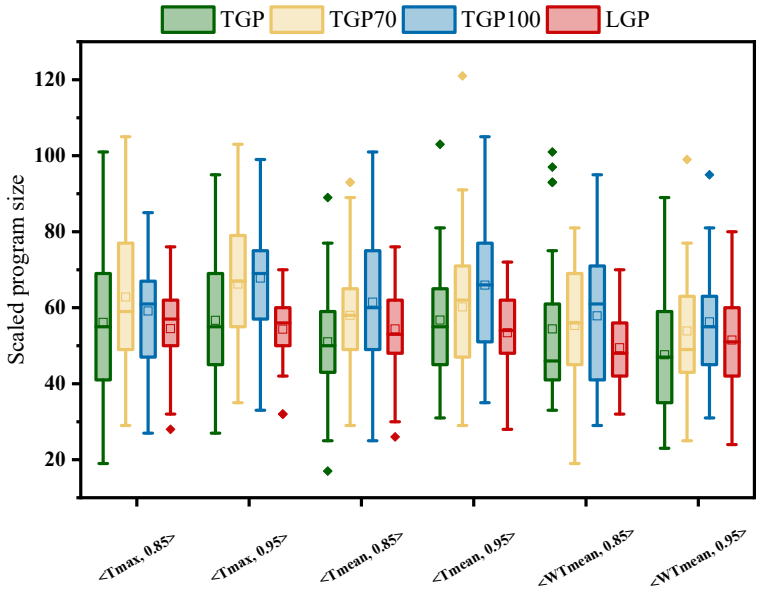

Figure 3.5: Scaled program sizes of TGP with different generations and LGP.

$$A=SL-WKR+3PT+NIQ$$
$$\text{rule}= \max(A-WKR,\ A+PT-2)$$

Figure 3.6: Example program of LGP.

3.6 Chapter Summary

The goal of this chapter is to develop an LGPHH method for DJSS problems. This goal is fulfilled by an LGPHH based on a generational evolutionary framework. To answer the research questions, we make a comprehensive investigation on the variation step size, the number of generations, and the register initialization strategies. Finally, we verify the performance of the LGPHH method by comparison with a tree-based GPHH method.

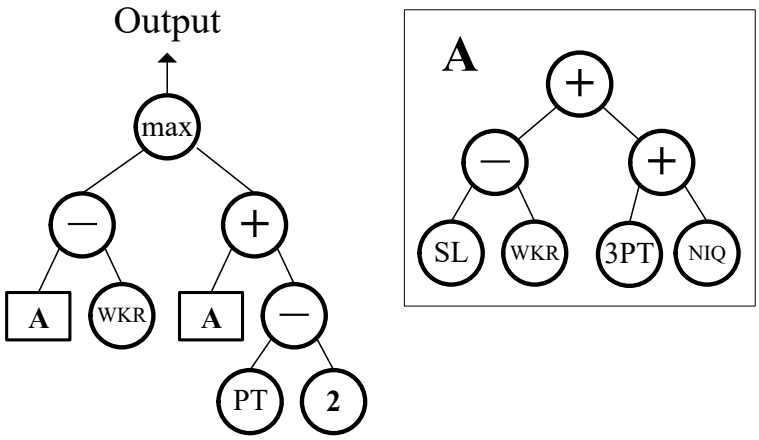

Figure 3.7: The corresponding DAG of the example program in Fig. 3.6

For the first research question, the results recommend that a large number of generations and a small variation step size (i.e., mutation-dominated evolution) are helpful for LGPHH to evolve dispatching rules. For the second research question, it is advisable to use diverse job-related attributes to initialize LGP registers. For the third research question, the LGPHH with the recommended parameter settings has better effectiveness and training efficiency than the tree-based GPHH method within the same number of simulations. The analysis of output dispatching rules from LGPHH verifies that the LGP-evolved dispatching rules are more compact than those tree-based rules. The example rule further implies that the dispatching rules of LGPHH successfully reuse effective building blocks, which is essential for evolving compact dispatching rules. The investigation of basic LGPHH shows the superiority of LGP, which shares a similar conclusion in other LGP applications such as classification and regression. It is likely that the superiority of LGP is extendable to other domains. This chapter makes a foundation (i.e., an effective framework for basic LGPHH for solving DJSS) for the following chapters. In the following chapters, we will develop advanced LGP methods based on this

chapter and apply the advanced methods to solve DJSS problems.

Chapter 4

Graph-based LGP Search Mechanisms for DJSS

LGP is a typical graph-based GP method. However, its graph-based characteristics are not fully investigated. In this chapter, we will develop advanced graph-based search mechanisms for LGP to make full use of the search information of graphs and the synergy between GP representations.

4.1 Introduction

One of the important features of LGP is its graph characteristics. By connecting primitives based on registers, an LGP individual can be decoded into a directed acyclic graph (DAG). Primitives are the basic functions and terminals that compose GP programs. Presenting LGP individuals (i.e., programs) by DAGs has different advantages from a genotype (i.e., a sequence of register-based instructions). For example, graphs represent programs in a more compact representation. On the other hand, a genotype enables neutral search in program spaces and memorizes potential building blocks [145,217]. Empirical studies verified that graphs and LGP genotypes are competitive for different tasks [6,214]. It is valuable to make full use of the graph characteristics of LGP.

There are some studies about utilizing graph information in the course of LGP evolution [6, 21, 214], but they miss three potential improvements based on the LGP graph characteristics. First, due to the linear representation, many instructions in an LGP individual may have no effect on the program output. An LGP instruction can be *redundant*. For example, in the program ($R1 = x + 2;\ R1 = x * 3;$), the latter instruction makes the former one redundant. Besides, a block of code can be *irrelevant* to the program output (i.e., introns). For example, given the program ($R3 = R1 + 1;\ R3 = R3 + R2;\ R0 = R1 * 2;$), where $R0$ is the program output register, the first two instructions are irrelevant to the program output. If we see an LGP program as a 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 effect to the program output or not [21]. 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.

Second, the utilization of graphs in LGP is *one-way* (i.e., the existing LGP studies mainly consider DAGs as a compact and intuitive way to depict the programs). Whether there is an effective way to bridge graphs (i.e., phenotype) and LGP instructions (i.e., genotype) is unknown yet. The absence of an effective transformation from DAGs to instructions precludes LGP from fully utilizing the graph information and cooperating with other graph-related techniques such as neural networks. To have a better understanding of LGP graph-based characteristics, a *two-way* transformation between graphs and LGP programs is necessary.

Third, graphs is an effective medium for conveying search information in different GP representations since most of the GP representations can

be represented as graphs. Existing studies have shown that different GP representations have different pros and cons for solving different problems [214, 244]. However, extending such kind of knowledge to unseen domains is difficult, and such investigations are often too time-consuming and it is hard to cover all different branches and variants of a problem. When encountering an emerging application or a new problem, users have scarce domain knowledge in selecting a GP representation. To make full use of different GP representations and enhance the search performance of existing GP methods, it would be interesting to investigate whether the different GP representations can cooperate in solving a single task.

To fulfill these three improvements, this chapter first proposes four genetic operators for LGP based on its graph-based characteristics and comprehensively investigates the effectiveness of these genetic operators. Then, this chapter further proposes a Multi-Representation GP (MRGP) based on the achievement in the first step. Specifically, we propose a graph-based crossover operator for the first improvement. This new operator takes the relationship (links in the DAG) between the instructions into account and selects only the instructions contributing to the final program output to modify.

To fulfill the two-way transformation between graphs and LGP programs, we design graph-to-instruction transformations for LGP individuals to accept the graph information of DAGs in evolution. The main challenge lies in the fact that DAGs do not contain register information, but LGP individuals are register-based instruction sequences. To overcome this challenge, this chapter proposes two strategies to address the register issue in a graph-to-instruction transformation. In the first transformation strategy, LGP individuals accept graph information without considering the topological information that is represented by registers. We assume that if topological information is not that important, ignoring the register identification is a simple and effective way to convey graph information. In the second strategy, LGP individuals identify the registers (by guess-

ing) for the instructions to reconstruct topological information. However, it is hard to perfectly reconstruct the topological structures by registers because of the absence of register information in DAGs. Specifically, we develop three new genetic operators based on graph node frequency, adjacency matrix, and adjacency list, to fulfill the two transformation strategies.

To make full use of the synergy between different GP representations, this chapter proposes a new MRGP algorithm that simultaneously evolves individuals with more than one representation. This section focuses on the MRGP with two typical GP representations, TGP and LGP, denoted as MRGP-TL. Fig. 4.1 is an example to show the possible impact of switching tree-based and linear GP representations. Given the inputs $X_1 = 3, X_2 = 5$ and the target output 1.5, we apply TGP and LGP to synthesize a mathematical formula. Each pair of TGP and LGP programs with the same fitness represents the same program. Suppose TGP and LGP can only apply mutation operators to produce offspring and start from the same formula $f(X) = X_1 \times X_2$ whose fitness (defined as absolute error) is 13.5 ($|3 \times 5 - 1.5| = 13.5$). The initial values of LGP registers in Fig. 4.1 are 0. To move on to the second step, LGP only needs to mutate the second primitive in the first instruction, but TGP has to perform a subtree mutation. Subtree mutation is a relatively large variation that leads to a large neighborhood. It is more likely (i.e., easier) for LGP to sample the offspring from a small neighborhood (i.e., by one-primitive mutation) than for TGP to sample the exact offspring from the large neighborhood (i.e., by subtree mutation). However, from the second to the third step, TGP only needs to mutate a tree node, while LGP has to mutate a new instruction. Fig. 4.1 shows that switching solutions between LGP and TGP representations can share the search information in TGP and LGP. It is potential for GP individuals to reach better fitness via fewer and smaller variations.

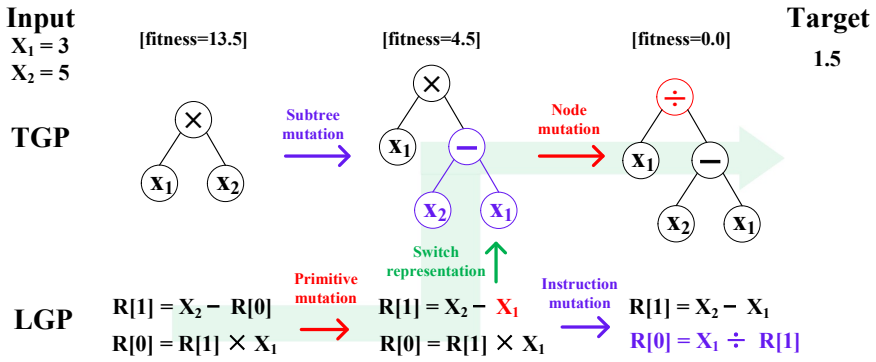

Figure 4.1: Potential impact of switching GP representations. The light green arrow shows a potential easy search trajectory for the LGP individual to reach the target model.

4.1.1 Chapter Goals

The goal of this chapter is to develop advanced graph-based search mechanisms for LGP for DJSS. This chapter proposes and investigates several potential ways of making use of graph-based information. Based on the investigation, this chapter further proposes an MRGP that share search information between GP representations based on graphs. Specifically, this chapter has the following research objectives.

1. Develop a graph-based crossover operator for LGP to highlight the effective building blocks during evolution.

2. Develop three graph-to-instruction genetic operators to enable LGP to accept graph information.

3. Analyze the effectiveness of the four proposed graph-based genetic operators.

4. Based on the investigation of the proposed graph-based operators, propose an MRGP method to make full use of the synergy of different GP representations. Specifically, we take TGP and LGP as examples to verify the effectiveness of the MRGP.

5. Analyze the effectiveness of the proposed MRGP method for solving

DJSS problems.

4.1.2 Chapter Organization

The rest of this chapter is organized as follows. Section 4.2 first proposes the four graph-based genetic operators. Then, section 4.3 makes a comprehensive investigation of the proposed genetic operators. Based on the investigation, section 4.4 further proposes an MRGP method and takes TGP and LGP as examples. Section 4.5 verifies the effectiveness of the MRGP method for solving DJSS problems. Finally, section 4.6 concludes this chapter.

4.2 Proposed Graph-based Operators

This section proposes four graph-based genetic operators. Specifically, we first propose a graph-based crossover (GC) to highlight the effective building blocks in LGP individuals, which are highly related to the corresponding DAGs of the individuals. Second, we propose three graph-to-instruction genetic operators, which are frequency-based crossover (FX), adjacency matrix-based crossover (AMX), and adjacency list-based crossover (ALX). These three operators convey three different types of graph information (i.e., primitive frequency, adjacency matrix, and adjacency list). To investigate the effectiveness of the three graph-to-instruction transformation methods, we explicitly transform LGP instruction segments into DAGs [21], and graph-based genetic operators accept the DAGs as genetic materials to produce offspring. The schematic diagram of conveying graph information to LGP instructions is shown in Fig. 4.2. We apply the three graph-to-instruction genetic operators in the dashed arrow with a question mark. We assume that all the four graph-based genetic operators should effectively evolve LGP programs if they are effective in conveying building block information.

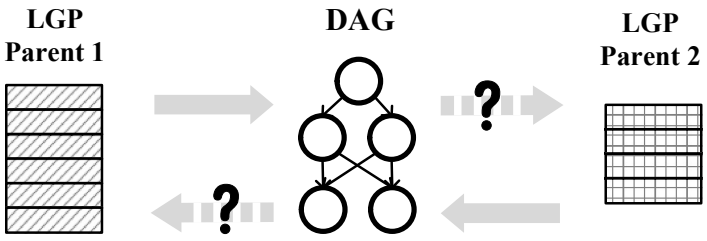

Figure 4.2: The schematic diagram of accepting graphs as LGP genetic materials. The program-to-DAG transformation (i.e., grey solid arrows) is fulfilled by [21]. The DAG-to-program transformation (i.e., dashed arrow) with a question mark is the focus of this section.

4.2.1 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 ignore adjusting the links in the resultant DAG.

Algorithm 2 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 $\mathbf{f}_1$ and $\mathbf{f}_2$, a set of registers $\mathbb{R}$, as well as three crossover parameters, i.e., the maximal size of sub-graphs $\overline{S}$, the maximal size differences of sub-graphs 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) $\mathbf{f}'_1$ and $\mathbf{f}'_2$. 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 $[\underline{l}, \bar{l}]$, and (2) the gap between the two sub-graph sizes do not exceed Δ_S. Then, it swaps the two sub-

Algorithm 2: `GraphCrossover`($\mathbf{f}_1$, $\mathbf{f}_2$, $\mathbb{R}$, $\overline{S}$, Δ_S, D_{cross})

Input: Two LGP parents $\mathbf{f}_1$ and $\mathbf{f}_2$, 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 $\mathbf{c}_1$ and $\mathbf{c}_2$.

Extract the list of *effective* instructions $\mathbf{f}'_1 \subseteq \mathbf{f}_1$, $\mathbf{f}'_2 \subseteq \mathbf{f}_2$;

`/* Select sub-graphs */`

repeat

  Randomly select two registers $r_1, r_2 \in \mathbb{R}$;

  repeat

    Randomly sample an instruction index $i_1 \leftarrow \texttt{UniformInt}(1, ||\mathbf{f}'_1|| + 1)$ and $i_2 \leftarrow \texttt{UniformInt}(1, ||\mathbf{f}'_2|| + 1)$;

  until $|i_1 - i_2| \leq D_{cross}$;

  Randomly sample $S_1 \leftarrow \texttt{UniformInt}(1, \overline{S} + 1)$, $S_2 \leftarrow \texttt{UniformInt}(1, \overline{S} + 1)$;

  $G_1 \leftarrow \texttt{GetSubGraph}(\mathbf{f}'_1, \{r_1\}, i_1, S_1)$;

  $G_2 \leftarrow \texttt{GetSubGraph}(\mathbf{f}'_2, \{r_2\}, i_2, S_2)$;

until $||\mathbf{f}'_1|| - ||G_1|| + ||G_2|| \in [\underline{l}, \overline{l}]$ *and* $||\mathbf{f}'_2|| - ||G_2|| + ||G_1|| \in [\underline{l}, \overline{l}]$ *and* $|||G_1|| - ||G_2||| \leq \Delta_S$;

`/* Swap the sub-graphs */`

$\mathbf{c}_1 \leftarrow \texttt{GraphSwap}(\mathbf{f}_1, \mathbf{f}_2, G_1, G_2)$;

$\mathbf{c}_2 \leftarrow \texttt{GraphSwap}(\mathbf{f}_2, \mathbf{f}_1, G_2, G_1)$;

return $\mathbf{c}_1$, $\mathbf{c}_2$;

graphs to generate two offspring. Note that the swapping is asymmetric, and there is a recipient parent and a donator parent.

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), then the current instruction is added to the sub-graph. The pseudo-code is shown in Algorithm 3, where the fringe 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 4. It generates an offspring by replacing the instructions in sub-graph

Algorithm 3: `GetSubGraph`($\mathbf{f}$, $\mathbb{T}$, i, S)

Input: An LGP individual $\mathbf{f}$, 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 to 0 do
    if des(f_j) ∈ T then
        G ← [f_j, G];
        if ||G|| = S then break;
        T ← T \ des(f_j);
        for src ∈ src(f_j) do
            if src is a register then T ← T ∪ src;
return G;
```

Algorithm 4: `GraphSwap`($\mathbf{f}_1$, $\mathbf{f}_2$, G_1, G_2)

Input: A recipient LGP parent $\mathbf{f}_1$, a donator LGP parent $\mathbf{f}_2$, sub-graphs (lists of instructions) G_1, G_2

Output: An LGP offspring $\mathbf{c}$

```
c ← f_1;
Replace G_1[||G_1|| − 1] in c with G_2;
Remove G_1[0 : ||G_1|| − 2] from c;
while ||c|| > l̄ do Randomly remove an intron from c;
return c;
```

G_1 of the recipient parent $\mathbf{f}_1$ with the sub-graph G_2 of the donator parent $\mathbf{f}_2$. Specifically, it replaces the last instruction $G_1[||G_1|| - 1]$ 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 donator parent, which would increase the effectiveness of the swapping/replacement.

An example of the graph-based crossover is shown in Fig. 4.3. The

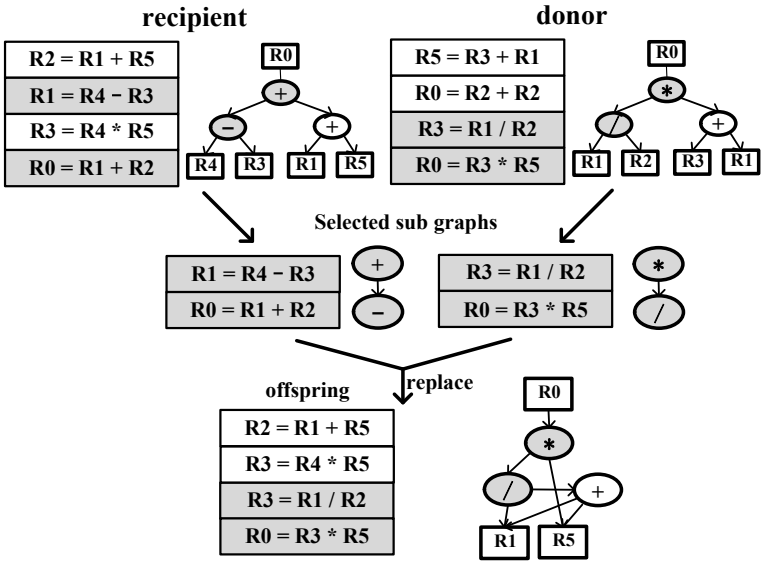

Figure 4.3: Example of graph-based crossover.

parent on the left is the recipient and the one on the right is the donator. 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). This demonstrates the effectiveness of the graph-based crossover.

4.2.2 Frequency-based Crossover

The frequency of primitives is a simple high-level feature of a graph. The frequency of primitives implies the importance of different primi-

tives. Frequency has been widely applied in GP to identify important features [88, 258]. In the FX operator, we utilize the primitive frequency in LGP crossover by seeing the primitive frequency as a kind of distribution.

FX accepts two parent individuals and produces one offspring. The offspring is produced by varying one of the primitives in an instruction of the first parent. When varying the primitive, the old primitive is replaced by a new one based on the frequency of the primitives in the second parent. We hope that varying the primitive based on the frequency of another individual stimulates more useful building blocks. The pseudo-code of FX is shown in Alg. 5. $(\cdot)$ following a set or a list denotes getting an element based on the index. The frequency vector is defined as $\mathbf{F} = [fun_1, fun_2, ..., fun_g, in_1, in_2, ..., in_h]$ where g is the number of functions and h is the number of input features. The function fun_f is sampled by a roulette wheel selection on the function frequency (i.e., $\mathbf{F}[fun_1, fun_2, ..., fun_g]$). A higher frequency implies a larger probability to be selected. If varying constant is triggered (i.e., $\texttt{Uniform}(0, 1) < \theta_{fun} + \theta_{con}$ in line 8 of Alg. 5), one of the source registers is replaced by an input feature which is sampled by a roulette wheel selection on the input feature frequency $\mathbf{F}[in_1, in_2, ..., in_h]$. To ensure that every primitive has a small probability of being selected, we add 1.0 on all the elements of $\mathbf{F}$. The destination register $R_{f',d}$ and the source registers $R_{f',s}$ of the instruction f' are sampled uniformly.

Fig. 4.4 shows an example of producing offspring by FX. First, FX transforms the second parent into a DAG and obtains the frequency of the primitives. Note that the frequency of primitives in a DAG is different from the frequency of primitives in raw effective instructions, as the effective instructions contain register primitives but the DAG does not. Since the DAG merges all the duplicated constants (i.e., input feature $x_{i(i=0,1,2)}$) into one graph node, FX treats the incoming degree as the frequency of constant graph nodes. FX normalizes the frequency of primitives and gets the distribution which further biases the variation on the first parent. We

Algorithm 5: Frequency-based crossover

Input: The two parents $\mathbf{f}_1$ and $\mathbf{f}_2$, function rate θ_{fun}, constant rate θ_{con}, destination register rate θ_{des}, source register rate θ_{sou}

Output: An offspring $\mathbf{c}$

$\mathbf{F} \leftarrow$ get the primitive frequency from $\mathbf{f}_2$;
$\mathbf{c} \leftarrow \mathbf{f}_1$;
$l \leftarrow \texttt{UniformInt}(0, |\mathbf{c}|)$;
$f' \leftarrow \mathbf{c}(l)$;
$r \leftarrow \texttt{Uniform}(0, 1)$;
if $r < \theta_{fun}$ **then**
    `// sample functions`
    $fun_{f'} \leftarrow$ Roulette-wheel selection based on $\mathbf{F}[fun_1, fun_2, ..., fun_g]$
else if $r < \theta_{fun} + \theta_{con}$ **then**
    if *None of* $R_{f',s,1}$ *and* $R_{f',s,2}$ *are constant registers* **then**
        $i \leftarrow \texttt{UniformInt}(1, 3)$;
    else
        $i \leftarrow$ the index of the constant register;
    `// sample constants`
    $R_{f',s,i} \leftarrow$ Roulette-wheel selection based on $\mathbf{F}[in_1, in_2, ..., in_h]$;
else if $r < \theta_{fun} + \theta_{con} + \theta_{des}$ **then**
    $R_{f',d} \leftarrow$ sample a random destination register uniformly;
else
    $i \leftarrow \texttt{UniformInt}(1, 3)$;
    $R_{f',s,i} \leftarrow$ sample a random source register uniformly;
$\mathbf{c}(l) \leftarrow f'$
Return $\mathbf{c}$;

can see that the $R[2]$ in the third instruction of Parent 1 is changed to x_0 based on the distribution.

4.2.3 Adjacency Matrix-based Crossover

The adjacency matrix conveys more graph information than the primitive frequency by highlighting the neighboring relationship of graph nodes.

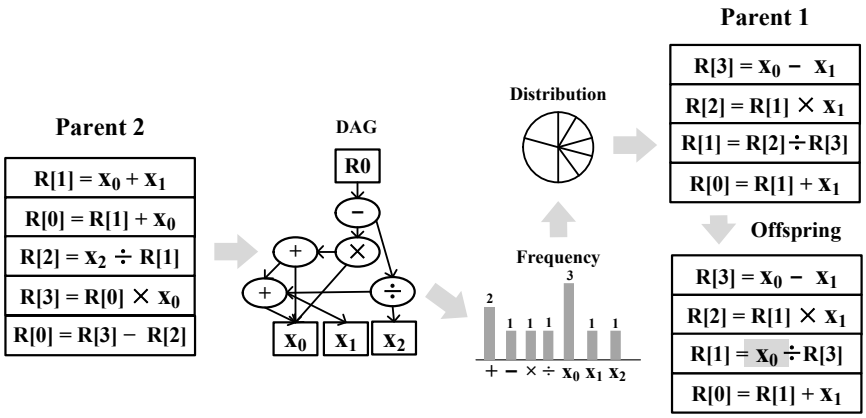

Figure 4.4: An example of FX operator.

The elements in an adjacency matrix are the frequency of a primitive connecting to another primitive in a DAG. We treat the elements in an adjacency matrix as a kind of distribution by normalizing the elements in each row. To this end, we utilize the adjacency matrix in LGP crossover (i.e., AMX).

Similar to FX, AMX accepts two parents and produces one offspring by varying one primitive in one of the instructions. Different from FX, AMX varies functions and constants by a roulette wheel selection based on the adjacency matrix of the graph from the other parent. The pseudo-code of varying a function or constant based on an adjacency matrix is shown in Alg. 6 [1] where the adjacency matrix is defined as

[1] $(x, :)$ and $(:, x)$ following a matrix denote getting the x^{th} row or the x^{th} column of elements respectively.

Algorithm 6: Varying a function or a constant based on the adjacency matrix

Input: Adjacnecy matrix $\mathbf{M}$, to-be-varied individual $\mathbf{f}$, index of the to-be-varied instruction l^*, function-and-constant flag Γ

Output: A new primitive node n

```
if Γ =function then
    out ← a random function;
    for f_out ← f(l* − 1) to f(0) do
        if f_out is an exon and ∃R_{f_out,s,i} = R_{f(l*),d}, i ∈ 1, 2 then
            out ← fun_{f_out}, break;
    n ← Roulette-wheel selection based on M_fun(out, :);
else if Γ =constant then
    out ← fun_{f(l*)};
    n ← Roulette-wheel selection based on M_in(out, :);
Return n;
```

$$\mathbf{M} = \left[\begin{array}{ccc:ccc} fun_{f_1,1} & \cdots & fun_{f_1,g} & in_{f_1,1} & \cdots & in_{f_1,h} \\ \vdots & \ddots & \vdots & \vdots & \ddots & \vdots \\ fun_{f_g,1} & \cdots & fun_{f_g,g} & in_{f_g,1} & \cdots & in_{f_g,h} \\ \hdashline \multicolumn{3}{c:}{0} & \multicolumn{3}{c}{0} \end{array}\right] \begin{array}{l} \left.\vphantom{\begin{array}{c}a\\a\\a\end{array}}\right\} function-out \\ \left.\vphantom{\begin{array}{c}a\\a\end{array}}\right\} constant-out \end{array}$$

$$\underbrace{\qquad\qquad}_{function-in} \quad \underbrace{\qquad\qquad}_{constant-in}$$

$$\Rightarrow \left[\begin{array}{ccc:ccc} fun_{f_1,1} & \cdots & fun_{f_1,g} & in_{f_1,1} & \cdots & in_{f_1,h} \\ \vdots & \ddots & \vdots & \vdots & \ddots & \vdots \\ fun_{f_g,1} & \cdots & fun_{f_g,g} & in_{f_g,1} & \cdots & in_{f_g,h} \end{array}\right] = \left[\mathbf{M}_{fun} \quad \mathbf{M}_{in}\right]$$

Since the constants (i.e., input features) have no outgoing edge in the graph, the two blocks of "constant-out" are two zero matrices. $\mathbf{M}$ is further simplified as $[\mathbf{M}_{fun}\ \mathbf{M}_{in}]$ where $\mathbf{M}_{fun}$ denotes the neighboring relationship from function primitives pointing to function primitives, and $\mathbf{M}_{in}$ denotes

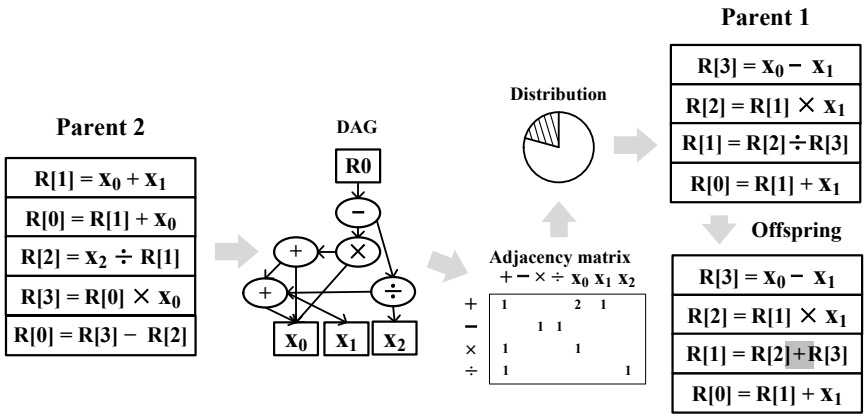

Figure 4.5: An example of AMX operator.

the neighboring relationship from function primitives pointing to input features. When varying the function of the l^*th instruction, AMX first finds the function out that points to the to-be-varied function. From the perspective of LGP genotype, it is equivalent to checking the instructions reversely from $\mathbf{f}(l^* - 1)$ and finding the instruction whose destination register is accepted as inputs of $\mathbf{f}(l^*)$. Then AMX applies the Roulette-wheel selection on $\mathbf{M}_{fun}$ and samples a new primitive based on out. To ensure that every primitive has a small probability of being selected, we add 1.0 on all the elements of $\mathbf{M} = [\mathbf{M}_{fun}\ \mathbf{M}_{in}]$ in the Roulette-wheel selection. When varying the constant of an instruction, AMX applies Roulette-wheel selection based on the function of the instruction and $\mathbf{M}_{in}$.

Fig. 4.5 shows an example of producing offspring by AMX operator. AMX gets a distribution based on the adjacency matrix from the second parent and performs variation based on the distribution. In Fig. 4.5, the "$\div$" in the third instruction of the first parent is varied into "$+$".

4.2.4 Adjacency List-based Crossover

An adjacency list is a graph representation that conveys the connection among graph nodes. To fully utilize the topological information carried by the adjacency list, we design an adjacency list-based crossover to vary instruction segments in the parent individuals. The adjacency list in this thesis is denoted as

$$\mathbf{L} = \Big(\ [fun_1, \mathbf{A}_1] \quad [fun_2, \mathbf{A}_2] \quad \cdots \quad \big[fun_{|\mathbf{L}|}, \mathbf{A}_{|\mathbf{L}|}\big] \ \Big)$$

where each item $[fun_i, \mathbf{A}_i]$ specifies the function fun_i and the list of its neighboring graph nodes $\mathbf{A}_i$. Specifically, $\mathbf{A}_i$ contains one or two nodes in this chapter since we only consider unary and binary functions. For example, we convert the left tree in Fig. 2.6 as

$$\mathbf{L} = \Big(\ [+, [x_1, +]] \quad [+, [x_2, -]] \quad [-, [x_1, x_3]] \ \Big)$$

It is worth noting that the adjacency list in this chapter uses primitive symbols (i.e., functions or terminals) to specify graph nodes to highlight building blocks, which is different from conventional adjacency lists which distinguish graph nodes by the indexes. Based on the adjacency list, this section proposes ALX.

ALX accepts two parent individuals (one as recipient and the other as donor) to produce one offspring. Rather than swapping the instruction sequences like basic linear crossover [12], the donor parent first selects a sub-graph from the DAG (by selecting a sub-sequence of instructions) and obtains the corresponding adjacency list $\mathbf{L}$. The recipient accepts the adjacency list and constructs the new instruction sequence based on the adjacency list. The newly constructed instruction sequence is used to replace another sub-sequence of instructions in the recipient. The pseudo-code of transforming an adjacency list into an instruction sequence is shown in Alg. 7. First, ALX selects a crossover point from the recipient parent and removes a sub-sequence of instructions from the recipient. Based on the

Algorithm 7: Transforming an adjacency list to an instruction sequence

Input: An LGP recipient individual $\mathbf{f}$, an adjacency list $\mathbf{L}$
Output: An offspring $\mathbf{c}$

```
1  c ← f;
2  s ← UniformInt(0, |f|) // randomly select a crossover point
3  Randomly remove a sub-sequence of instructions from c based on s;
4  for j ← 1 to |L| do
5      [fun, A] ← L(j);
6      f ← randomly generate an instruction whose function is fun;
7      Insert f to c(s);
8  c ← RegisterAssignment(c, L, s);
9  if |c| exceeds the maximum and minimum program length then
10     c ← f;
11 Return c;
```

adjacency list, ALX randomly generates a sequence of instructions. Specifically, the functions in the newly generated instructions are coincident with the adjacency list. Since the adjacency list does not convey the information of registers, we propose a register assignment method for ALX to identify the registers in those newly generated instructions, as shown in Alg. 8.

In general, Alg. 8 checks the instruction sequence reversely based on the adjacency list to assign the registers in all the new instructions. There are two main steps in Alg. 8, assigning destination registers and assigning source registers. From the perspective of topological structures, assigning destination registers is equivalent to providing the results of the sub-graph to the upper part of the DAG, while assigning the source registers is equivalent to taking the results from the lower part of the DAG as the inputs of the sub-graph. When assigning destination registers, Alg. 8 ensures the effectiveness of all the newly generated instructions. On the other hand, Alg. 8 assigns source registers based on the neighboring graph nodes (i.e.,

Algorithm 8: `RegisterAssignment`

Input: An LGP individual $\mathbf{f}$, an adjacency list $\mathbf{L}$, crossover point s
Output: An offspring $\mathbf{c}$

for $j \leftarrow s + |\mathbf{L}| - 1$ *to* s **do**
  $[a, \mathbf{A}] \leftarrow \mathbf{L}(s + |\mathbf{L}| - j)$;
  `// assigning destination registers`
  if $\mathbf{c}(j)$ *is not an exon* **then**
    Randomly mutate $R_{\mathbf{c}(j),d}$ until $\mathbf{c}(j)$ is effective;
  `// assigning source registers`
  for $g \leftarrow 1$ *to* $|\mathbf{A}|$ **do**
    $b \leftarrow \mathbf{A}(g)$;
    if b *is a function* **then**
      $\mathbf{L}' \leftarrow$ collect the entity indices from $[j, s]$ where $\mathbf{L}(k).fun = b$ and $k \in [j, s]$;
      if $\mathbf{L}' \neq \emptyset$ **then**
        $l \leftarrow \texttt{UniformInt}(1, |\mathbf{L}'| + 1)$;
        $R_{\mathbf{c}(j),s,g} \leftarrow R_{\mathbf{c}(l),d}$;
      else
        if $j > 0$ *and* $\texttt{UniformInt}(0, j + 1) - 1 > 0$ **then**
          $l \leftarrow \texttt{UniformInt}(1, j + 1)$;
          $R_{\mathbf{c}(j),s,g} \leftarrow R_{\mathbf{c}(l),d}$;
    else if b *is a constant* **then**
      $R_{\mathbf{c}(j),s,g} \leftarrow b$;
Return $\mathbf{c}$;

functions or constants) specified by the adjacency list. Specifically, if the neighboring graph node is a function, Alg. 8 collects the possible instructions whose functions are coincident with the neighboring function in the adjacency list and randomly assigns the destination register from one of the possible instructions as the source register.

Fig. 4.6 shows an example of producing an offspring by ALX. First, ALX selects a sub-graph from the second parent, consisting of "$-, \times, +$", and gets the corresponding adjacency list. Then ALX generates a new in-

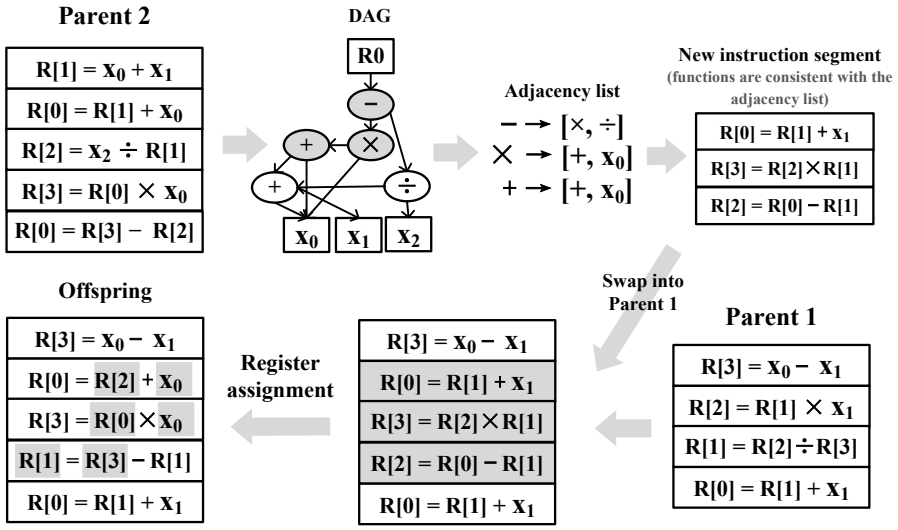

Figure 4.6: An example of ALX operator. The selected graph nodes, the newly generated instructions, and the newly updated primitives are highlighted in gray color.

struction segment based on the adjacency list and swaps it into the first parent (i.e., the 2^{nd} to 4^{th} instructions in the new instruction sequence). To maintain the topological structures among newly inserted instructions, ALX applies the register assignment method (i.e., Alg. 8) to update the registers. Alg. 8 replaces the destination register of the 4^{th} instruction to R[1] to ensure the three swapped-in instructions are effective. Then, based on the adjacency list, Alg. 8 update the source registers in the three swapped-in instructions so that "$-$" accepts the results from "$\times$" and "$\times$" accepts the results from "$+$" and the constant x_0. We can see that "$-, \times, +$" are connected together, and the 2^{nd} and 4^{th} instructions manipulate suggested constants by the adjacency list in the offspring.

Table 4.1: Summary on the pros and cons of different graph information representations.

Graph information	**pros**	**cons**
Frequency	They can bypass the puzzle of identifying registers for instructions.	Frequency only considers functions and constants and does not consider topological information at all, which is susceptible to the number of registers.
Adjacency matrix		Adjacency matrices of LGP programs are sparse in many cases, which cannot provide enough search bias for LGP.
Adjacency list	Adjacency list conveys both frequency and topological structures simultaneously and is less susceptible to graph width.	The effectiveness of adjacency list information might be limited by the effectiveness of reconstructing topological structures based on registers.
Effective instruction	Effective instructions convey all necessary information of sub-graphs and is less susceptible to graph width	Effective instructions cannot transform a graph into an instruction segment.

4.2.5 Summary

We summarize the pros and cons of the four graph information representations, as shown in table 4.1. Transforming graphs into LGP instructions based on graph node frequency and adjacency matrix is a good alternative solution to bypass the puzzle of identifying registers for new instructions. However, frequency-based and adjacency matrix-based information do not explicitly consider the topological information, which might be sus-

ceptible to the number of registers (i.e., graph width). Besides, adjacency matrices of LGP graphs might be often too sparse to effectively guide the search. Adjacency list can convey graph node frequency and their topological structures simultaneously but is dependent on a register assignment method to reconstruct the topological structures. The effectiveness of the register assignment method might limit the effectiveness of adjacency list information. Swapping effective instructions can straightforwardly convey all the necessary search information. However, it cannot fulfill the DAG-to-program transformation since it only manipulates LGP programs.

4.3 Comparison among Graph-based Operators

To verify the effectiveness of the four graph-based genetic operators, this section applies LGP with these four operators to solve DJSS [24, 150, 277]. Specifically, we apply LGP to solve DJSS problems with maximum tardiness, mean tardiness, and weighted mean tardiness with two utilization levels 0.85 and 0.95. The configurations of the simulation follow the ones in chapter 3. Each compared method has 50 independent runs on each scenario.

4.3.1 Comparison Design

To investigate the effectiveness of the graph-based operators, we design seven compared methods. The first two methods are the basic TGP [107] and LGP [21] which are seen as the baseline. We apply these two basic GP methods as baselines to investigate the three graph-based genetic operators. These two basic GP methods are common baselines in existing GPHH for DJSS studies, which help us straightforwardly investigate the effectiveness of the three graph-based operators. The third to sixth methods respectively verify the four newly designed graph-based genetic operators. We replace the micro mutation of the basic LGP with FX and

Table 4.2: Mean test performance (Std.) of all the compared methods.

Sce.	TGP	LGP	LGP+FX	LGP+AMX	LGP+ALX	LGP+GC	LGP+FA
A	1928.4 (40.4) ≈	1956.3 (53.8)	1941.6 (59.4) ≈	1953.1 (115) ≈	1923 (54.3) ≈	1925.9 (55.4) ≈	**1920.1** (46) ≈
B	4060.6 (116) –	3999.2 (90.9)	3948.7 (72.9) ≈	3957.2 (81.7) ≈	3920.7 (86.5) +	3967 (119) ≈	**3901** (88.4) +
C	417.3 (2.5) ≈	417.9 (2.3)	417 (2.6) ≈	417.6 (3.7) ≈	**416.6** (2.4) ≈	417.4 (3) ≈	418.3 (4.3) ≈
D	1116.2 (10) ≈	1118.2 (10.7)	1112.5 (8) ≈	1114.2 (9.6) ≈	1115.5 (12.5) ≈	1113.6 (9) ≈	**1112.2** (8.9) +
E	727.5 (6.5) –	724.3 (5.4)	724 (5.8) ≈	724 (6) ≈	724 (5.7) ≈	**722** (6.3) +	723.8 (6.2) ≈
F	1747.4 (29.6) –	1729.6 (27.7)	1721.4 (31) ≈	1722.2 (29.8) ≈	1730.8 (25.7) ≈	**1718** (21.3) ≈	1725.6 (24.5) ≈
mean rank	6.67	4.67	3.33	4.67	3	**2.83**	**2.83**

A: $\langle Tmax, 0.85\rangle$, B: $\langle Tmax, 0.95\rangle$, C: $\langle Tmean, 0.85\rangle$, D: $\langle Tmean, 0.95\rangle$, E: $\langle WTmean, 0.85\rangle$, F: $\langle WTmean, 0.95\rangle$

AMX in the third and fourth compared methods respectively because the variation step sizes of FX and AMX are similar to LGP micro mutation (i.e., only varying one or a few primitives in the parent but not changing the total number of instructions). The third and fourth methods are denoted as LGP+FX and LGP+AMX respectively. The fifth compared method is denoted as LGP+ALX, in which the linear crossover in the basic LGP is replaced by ALX. The sixth compared method is the LGP with an existing graph-based crossover, denoted as LGP+GC. The other settings in LGP+FX, LGP+AMX, LGP+ALX, and LGP+GC are kept the same as the basic LGP. Finally, we investigate the effectiveness of the cooperation of multiple graph-based genetic operators. Since our prior investigation shows that LGP+FX has better average performance than LGP+AMX, we replace the micro mutation and linear crossover in the basic LGP with FX and ALX simultaneously. The LGP with FX and ALX is denoted as LGP+FA.

The parameters of basic TGP and LGP are defined based on chapter 3. All the compared methods start the search from a small initial program size. All the LGP methods manipulate a register set with 8 registers. The other parameters of LGP methods follow the recommended settings in chapter 3.

4.3.2 Test Performance

The mean test performance of all the compared methods in solving the six DJSS scenarios is shown in table 4.2. We conduct a Friedman test with a significance level of 0.05 on the test performance of all the compared methods. The p-value of the Friedman test is 0.016 which implies there is a significant difference among the compared methods. The notation "$+$", "$-$", and "$\approx$" in table 4.2 denote a method is significantly better than, significantly worse than, or statistically similar to the basic LGP based on Wilcoxon rank-sum test with a significant level of 0.05. The best mean values are highlighted in bold.

As shown in table 4.2, first, three newly proposed graph-based genetic operators (except LGP+AMX) improve the overall performance of basic LGP since the mean ranks of most LGP methods with graph-based genetic operators are better than basic LGP (i.e., smaller is better). Second, the performance of LGP is improved with the amount of graph information overall. Specifically, LGP+FX and LGP+AMX (i.e., distribution and local topological structures) have worse mean ranks than LGP+ALX (i.e., topological structures of sub-graphs), and LGP+ALX has a worse mean rank than LGP+GC which conveys topological structures and register information in exchanging genetic materials. LGP+FA which conveys more graph information by using multiple graph-based genetic operators has the same mean rank as LGP+GC. Table 4.2 also shows that the best mean test performance is mainly achieved by LGP+ALX, LGP+GC, and LGP+FA, which verifies that conveying more graph information (e.g., primitive frequency and topological structures) in the course of exchanging genetic materials is effective in improving LGP performance. Note that although the adjacency matrix is supposed to convey more graph information than graph node frequency, the adjacency matrix does not help LGP+AMX perform better than LGP+FX since adjacency matrices of LGP graphs are often too sparse to provide search bias (i.e., most elements in the adjacency matrix are zero which degenerates AMX to uniform variation). In short, all the

four newly proposed graph-based genetic operators have very competitive performance with basic LGP, which implies the proposed graph-based genetic operators effectively convey the graph information from one parent individual to the other. The improvement in mean ranks implies the potential of utilizing graph information.

4.3.3 Training Performance

This section compares the training performance of different graph-based genetic operators, as shown in Fig. 4.7. Specifically, we compare the test performance of the best individuals from all the compared methods at every generation. Overall, all compared methods perform quite similarly in most problems. But in some problems, we can see gaps among the curves. For example, LGP+FA converges faster than the others in $\langle Tmax, 0.85\rangle$ and $\langle Tmax, 0.95\rangle$, and LGP+GC converges faster than the others in $\langle WTmean, 0.95\rangle$ in the first 20000 simulations. Further, if we look at the lowest convergence curves at different stages, we find that LGP+ALX, LGP+GC, and LGP+FA alternatively take the leading positions in training. For example, in $\langle Tmean, 0.85\rangle$, LGP+ALX is slightly lower than the others in most simulations but is caught up by LGP+GC from 20000 to 40000 simulations. Based on the results, we confirm that conveying as much graph information as possible (e.g., LGP+ALX, LGP+GC, and LGP+FA) can improve LGP performance to some extent.

To conclude, the proposed graph-based genetic operators successfully carry the information from LGP instructions to graph and back to instructions since the training and test performance of the proposed graph-based genetic operators are similar to or better than the performance of conventional genetic operators that directly exchange instructions. We also see that the performance gain of graph-based genetic operators increases with the amount of graph information overall. Furthermore, if we look back at the pros and cons of the graph information (table 4.1), we find that 1) by-

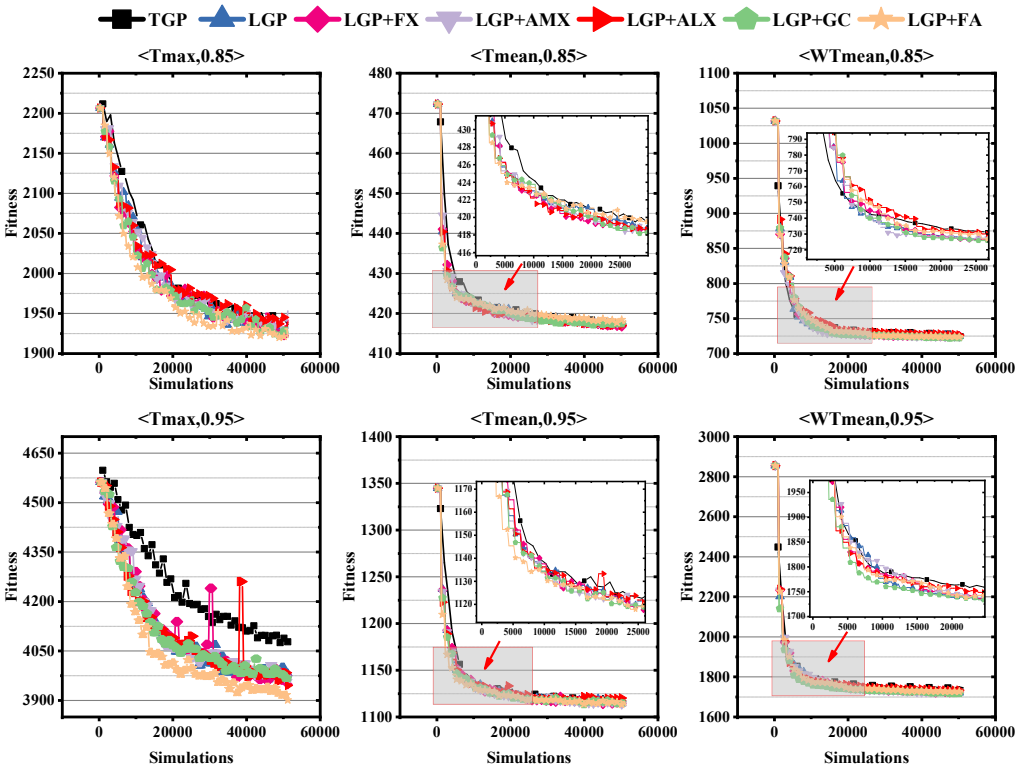

Figure 4.7: The convergence of different graph-based genetic operators on test instances.

passing the issue of identifying registers (i.e., LGP+FX and LGP+AMX) is a feasible way to transform graphs to instructions, but it is not as effective as LGP+ALX because of the loss of graph information 2) LGP+ALX performs as competitively as swapping effective instructions directly, which implies that the register assignment method in LGP+ALX reconstructs the topological structures quite well without much deterioration on effectiveness.

4.3.4 Component Analyses on ALX

Section 4.3.2 verifies that ALX is an effective method for LGP to accept graph information. To investigate the reasons of the superior performance, this section conducts an ablation study on ALX. Given that an adjacency list can convey two kinds of graph information, the frequency of graph nodes and their topological connection, we verify the effectiveness of different graph information separately by four ALX-based methods. We use basic LGP and LGP+ALX as the baseline methods in this section. We develop "ALX/noRegAss" in which we remove the `RegisterAssignment`$(\cdot)$ from LGP+ALX. In this case, LGP+ALX generates the instruction segment only based on each item of the adjacency list and does not further connect these generated instructions by registers. The newly generated instruction segment has a similar graph node frequency to the adjacency list but has very different topological structures, and the effectiveness of the instruction segment cannot be ensured (i.e., might contain a lot of introns after swapping into a parent). Besides, we develop "ALX/randSrc" in which we do not assign source registers for the newly generated instruction in ALX (i.e., removing lines 5-15 in Alg. 8 but ensuring that each newly generated instruction is effective in the offspring). By comparing with ALX/noRegAss, ALX/randSrc eliminates the performance bias caused by the epistases of instructions (i.e., to-be-swapped graph nodes are likely not connected with the parent graph in ALX/noRegAss). Nevertheless, ALX/randSrc does not maintain the topological structures based on the adjacency list either. Note that since all of the parameters in the compared methods follow the settings of the basic LGP which does not maintain the topological structures in its evolution, the compared methods with different components do not show significant performance discrepancy in our prior investigation. Therefore, to highlight the performance discrepancy, we also compare the four compared methods with 12 registers. Other parameters in this section are set the same as section 4.3.2.

Table 4.3 shows the results of the compared methods. We apply the Friedman test and Wilcoxon test to analyze the test performance of the compared methods. The p-values of the Friedman test are 0.284 and 0.012 for 8 and 12 registers respectively, which means there is a significant difference in the test performance with 12 registers. In the comparison with 8 registers, the p-values from a pair-wise Friedman test show that all compared methods are similar. However, the mean ranks and the mean test performance of ALX/noRegAss and ALX/randSrc show that only conveying frequency information of graphs is averagely inferior to conveying both frequency and topological information based on the adjacency list. The results with 12 registers also show a performance reduction when ALX does not maintain topological structures. The basic LGP and ALX/noRegAss have significantly worse performance than LGP+ALX, and ALX/randSrc has a larger (worse) mean rank and worse mean test performance than LGP+ALX in most scenarios.

In summary, the results confirm that ALX effectively uses both frequency and topological information to improve LGP performance. Specifically, first, graph node frequency improves LGP performance since ALX/noRegAss and ALX/randSrc have better test performance than basic LGP overall. Second, connecting every to-be-swapped graph node with the parent graph (i.e., ALX/randSrc and LGP+ALX) and connecting the newly generated instructions among themselves (i.e., LGP+ALX) both enhance LGP performance.

4.4 MRGP Based on Graphs

Section 4.3 shows that ALX is an effective operator for performing the graph-to-instruction transformation. The graph-to-instruction transformation allows us to cooperate with different GP representations based on graphs, that is, MRGP.

The major challenge of designing MRGP is that different GP represen-

Table 4.3: The mean test performance (std.) of LGP with different ALX components. The best mean values and significant p-values are highlighted in bold.

# Reg	Scenario	LGP	ALX/noRegAss	ALX/randSrc	LGP+ALX
8 registers	$\langle Tmax, 0.85\rangle$	1956.3 (53.8) ≈	1925.2 (56.5) ≈	**1922.1** (48) ≈	1923 (54.3)
	$\langle Tmax, 0.95\rangle$	3999.2 (90.9) –	3937.4 (127) ≈	3967.1 (133.6) ≈	**3920.7** (86.5)
	$\langle Tmean, 0.85\rangle$	417.9 (2.3) ≈	417.6 (3.2) ≈	417.7 (3) –	**416.6** (2.4)
	$\langle Tmean, 0.95\rangle$	1118.2 (10.7) ≈	1117.1 (14.6) ≈	**1115.2** (10.8) ≈	1115.5 (12.5)
	$\langle WTmean, 0.85\rangle$	724.3 (5.4) ≈	724.1 (6.6) ≈	724.9 (6.2) ≈	**724** (5.7)
	$\langle WTmean, 0.95\rangle$	**1729.6** (27.7) ≈	1741 (34.8) ≈	1726.8 (23) ≈	1730.8 (25.7)
mean rank		3	2.83	2.5	**1.67**
pair-wise p-value		0.442	0.705	1	
12 registers	$\langle Tmax, 0.85\rangle$	1940.8 (49.9) ≈	1939.3 (52.5) ≈	1936.6 (56.9) ≈	**1932.1** (48.5)
	$\langle Tmax, 0.95\rangle$	3999 (111.8) –	3989.6 (98.3) –	4006.4 (136.3) –	**3941.9** (73.8)
	$\langle Tmean, 0.85\rangle$	417.8 (2.7) ≈	418.1 (2.8) ≈	418.3 (3.5) ≈	**417.5** (2.4)
	$\langle Tmean, 0.95\rangle$	1118.3 (10.8) ≈	1118.5 (10.2) ≈	1117.9 (10.4) ≈	**1115.9** (9.3)
	$\langle WTmean, 0.85\rangle$	726.7 (6.9) ≈	727.1 (7.9) ≈	726.8 (9.1) ≈	**725.2** (5.8)
	$\langle WTmean, 0.95\rangle$	1743.9 (32.5) ≈	1739.8 (29.9) ≈	1737.8 (30.9) ≈	**1737.7** (30.8)
mean rank		3	3.17	2.83	**1**
pair-wise p-value		**0.044**	**0.022**	0.083	

tations cannot be exchanged building blocks directly. For example, tree-based representations in TGP do not contain information about registers in LGP. Furthermore, even within the LGP framework, individuals with different maximum numbers of registers cannot exchange instruction segments directly since it likely produces instructions with invalid registers. In addition, instruction outputs (register values) in LGP individuals can be reused by more than one subsequent instruction because of the graph-based structure, but tree nodes in standard TGP can only be used once. Thus, directly exchanging building blocks from different representations might not always produce valid offspring. To address the above issue, here we propose to unify the building blocks of different GP representations into adjacency lists to effectively exchange building blocks between

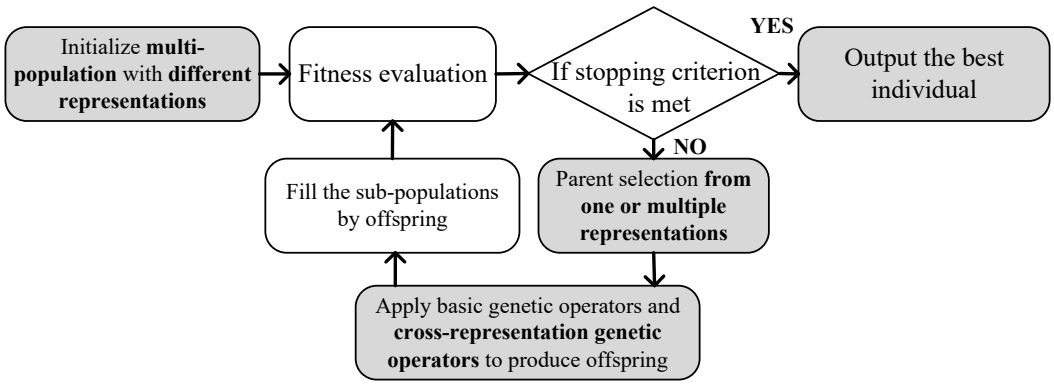

Figure 4.8: Evolutionary framework of MRGP. The novel components are highlighted by the dark boxes.

tree-based and linear GP representations.

Note that although adjacency lists can be an intermediate representation for tree-based and linear representations, this does not mean that an adjacency list is a more effective GP representation than tree-based or linear representations for the two following reasons. First, existing literature shows that evolving computer programs based on graph-based structures are not always better than tree-based and linear representations [214]. Different representations have their own pros and cons for different tasks. Second, a conventional adjacency list relies on graph node indices to distinguish graph nodes. But different graphs (i.e., GP individuals) likely have different indices for the same building blocks (e.g., a three-node building block "$x_1 + x_2$" might be indexed as "$\mathcal{A} \rightarrow [\mathcal{B}, \mathcal{C}]$" and "$\mathcal{D} \rightarrow [\mathcal{E}, \mathcal{F}]$" in two different GP individuals). The indexing mechanisms for adjacency list representations might be too complicated to show obvious advantages.

Algorithm 9: MRGP-TL

Input: cross-representation crossover rate θ_t, tournament selection size s, maximum depth of the tree $\overline{d}$, maximum number of instructions $\overline{L}$, minimum number of instruction $\underline{L}$

Output: best individual $\mathbf{h}$

```
1  Initialize two sub-populations, 𝕊_1 for the tree-based representation and 𝕊_2 for
     the linear representation.
2  while stopping criteria are not satisfied do
       // Evaluation
3      Evaluate fitness of individuals ∀f ∈ 𝕊_1 ⋃ 𝕊_2.
4      Update the best individuals h in 𝕊_1 ⋃ 𝕊_2;
5      for j ← 1 to 2 do
6          𝕊'_j ← ∅;
7          Clone top-1% individuals of 𝕊_j into 𝕊'_j;
8          while |𝕊'_j| < |𝕊_j| do
9              rnd ← Uniform(0, 1);
10             if rnd < θ_t then
11                 p_1 ← TournamentSelection(𝕊_j, s);
12                 i ← UniformInt(1, 3); // randomly pick "1" or "2".
13                 p_2 ← TournamentSelection(𝕊_i, s);
14                 c ← CALX(p_1, p_2, d̄, L̄, L̲);
15             else
16                 Apply corresponding (i.e., TGP or LGP) basic genetic operators
                     on 𝕊_j to produce offspring c (or c_1 and c_2);
17             𝕊'_j ← 𝕊'_j ⋃ {c} (or 𝕊'_j ← 𝕊'_j ⋃ {c_1, c_2});
18         𝕊_j ← 𝕊'_j;
19 Return h.
```

4.4.1 Overall Framework

We propose an overall framework of the MRGP, as shown in Fig. 4.8 (with the new components highlighted in grey). In contrast to the evolutionary framework of basic GP methods, MRGP evolves multiple subpopulations, each for a unique GP representation. When breeding offspring, MRGP selects parents from all the representations and also ap-

plies cross-representation genetic operators to produce offspring. Offspring of a certain representation fill the corresponding sub-population of the next generation. After generations of evolution, the best solution among the sub-populations is output. For example, when there are two sub-populations, one for the tree-based representation and the other for the linear representation, the best solution from the two sub-populations is returned.

This chapter studies MRGP based on the TGP and LGP, denoted as MRGP-TL. The pseudo-code of MRGP-TL is shown in Alg. 9 [2]. First, MRGP-TL initializes two sub-populations, one for evolving tree-based GP individuals and the other for evolving LGP individuals. All individuals in these two sub-populations evolve simultaneously. For each sub-population, we perform elitism selection to retain elite individuals for the next generation (line 7). To fill the sub-population of the next generation, we use tournament selection (i.e., `TournamentSelection`$(\cdot)$) to select individuals as parents and apply different genetic operators based on predefined rates. Specifically, MRGP-TL triggers the cross-representation adjacency-list based crossover (`CALX`$(\cdot)$) based on a predefined rate θ_t (line 10). If the cross-representation adjacency-list-based crossover is triggered, MRGP-TL selects a parent from the current sub-population and selects the other parent from one of the two sub-populations. `CALX`$(\cdot)$ accepts the two parents and produces an offspring. If the operator is not triggered, MRGP-TL applies basic TGP or LGP genetic operators to evolve tree-based and linear representations separately (lines 15-16). The newly generated offspring form the new populations with different representations (line 17). The evolution continues until a stopping criterion is met. The best individual among all the sub-populations with different representations is output as the final result.

[2] $|\cdot|$ denotes the cardinality of a container (e.g., set or list). $(\cdot)$ following a container denotes getting an element from the container based on the index.

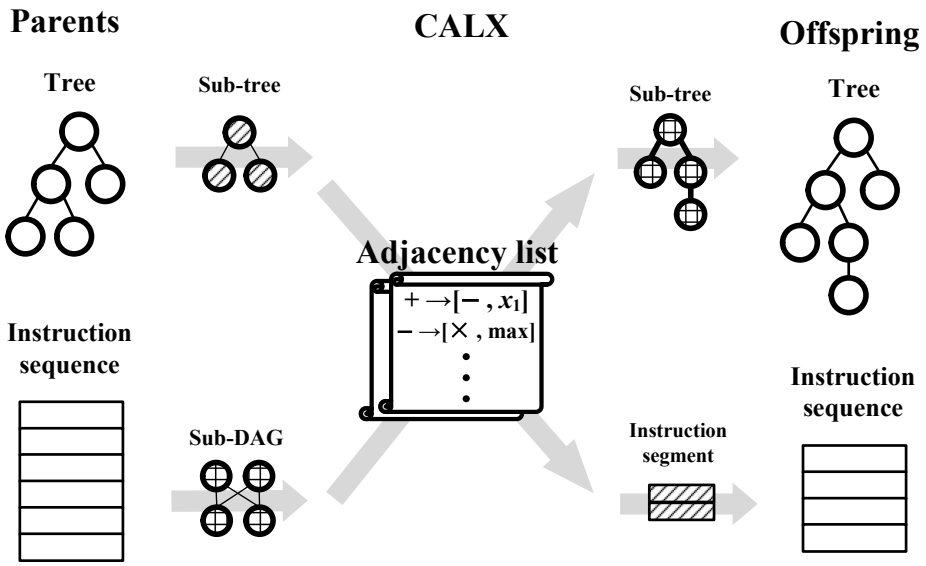

Figure 4.9: The schematic diagram of CALX between trees and instruction sequences.

4.4.2 Cross-representation Adjacency List-based Crossover

Knowledge transfer among representations is implemented by the cross-representation adjacency list-based crossover (CALX), as shown in Fig. 4.9. To swap genetic materials, a tree or an instruction sequence first selects a sub-tree or an instruction segment. The instruction segment is essentially a sub-DAG (or multiple disconnected sub-DAGs). The sub-tree and sub-DAGs are further converted into adjacency lists[3]. Based on the representation of the recipient, a new sub-tree or instruction segment is constructed based on the adjacency list and swapped into the recipient.

Algorithm 10: `CALX`

Input: Parent individuals $\mathbf{p}_1$ and $\mathbf{p}_2$, maximum depth of the tree $\overline{d}$, maximum number of instructions $\overline{L}$, minimum number of instruction $\underline{L}$

Output: An offspring $\mathbf{c}$

Clone $\mathbf{p}_1$ as $\mathbf{c}$;
if $\mathbf{p}_1$ *is a TGP individual* **then**
    `// breeding trees based on adjacency lists`
    Randomly pick an inner tree node t_1 from $\mathbf{c}$;
    if $\mathbf{p}_2$ *is a TGP individual* **then**
        Randomly pick an inner tree node t_2 from $\mathbf{p}_2$;
        $\mathbf{L} \leftarrow$ get the adjacency list of the sub-tree in $\mathbf{p}_2$ whose root is t_2;
    else if $\mathbf{p}_2$ *is an LGP individual* **then**
        Randomly select a crossover point t_2 and select an instruction segment $\mathbf{F}' \subseteq [\mathbf{p}_2(t_2), \mathbf{p}_2(|\mathbf{p}_2|)]$;
        $\mathbf{L} \leftarrow$ get the adjacency list of the sub-graph from $\mathbf{F}'$;
    $t_1' \leftarrow$ `GrowTreeBasedAL`$(\mathbf{L}$, the depth of t_1 in $\mathbf{c}, 1, \overline{d})$;
    Replace the sub-tree with the root of t_1 as the sub-tree with the root of t_1' in $\mathbf{c}$;
else if $\mathbf{p}_1$ *is an LGP individual* **then**
    `// breeding instructions based on adjacency lists`
    if $\mathbf{p}_2$ *is a TGP individual* **then**
        Randomly select a crossover point t_1 and select an instruction segment $\mathbf{F}_1' \subseteq [\mathbf{c}(1), \mathbf{c}(t_1)]$;
        $\mathbf{L}_1 \leftarrow$ get the adjacency list of the sub-graph from $\mathbf{F}_1'$;
        Randomly pick an inner tree node t_2 from $\mathbf{p}_2$;
        $\mathbf{L}_2 \leftarrow$ get the adjacency list of the sub-tree in $\mathbf{p}_2$ whose root is t_2;
    else if $\mathbf{p}_2$ *is an LGP individual* **then**
        Randomly select a crossover point t_1 and select an instruction segment $\mathbf{F}_1' \subseteq [\mathbf{c}(t_1), \mathbf{c}(|\mathbf{c}|)]$;
        $\mathbf{L}_1 \leftarrow$ get the adjacency list of the sub-graph from $\mathbf{F}_1'$;
        Randomly select a crossover point t_2 and select an instruction segment $\mathbf{F}_2' \subseteq [\mathbf{p}_2(t_2), \mathbf{p}_2(|\mathbf{p}_2|)]$;
        $\mathbf{L}_2 \leftarrow$ get the adjacency list of the sub-graph from $\mathbf{F}_2'$;
    $\mathbf{c} \leftarrow$ `GrowInstructionBasedAL`$(\mathbf{p}_1, \mathbf{L}_1, \mathbf{L}_2, t_1, n_1)$
    if $|\mathbf{c}| \notin [\overline{L}, \underline{L}]$ **then**
        $\mathbf{c} \leftarrow \mathbf{p}_1$;
Return $\mathbf{c}$;

Algorithm 11: `GrowTreeBasedAL`

Input: Adjacency list $\mathbf{L}$, current depth d, index of $\mathbf{L}$ I, maximum depth of the tree $\overline{d}$
Output: A tree root r

```
1  if |L| = 0 or d = d̄ then
2      Return r ← a random sub-tree whose depth ≤ d̄ − d + 1;
3  [r, A] ← L(I);
4  if r is a function then
5      for j ← 1 to |A| do
6          c′ ← A(j);
7          if c′ is a function then
8              L′ ← collect the entities from L(k), k ∈ [I, |L|] with L(k).fun = c′;
9              if L′ ≠ ∅ then
10                 c′ ← GrowTreeBasedAL(L, d + 1, UniformInt(1, |L′| + 1), d̄);
11             else
12                 c′ ← a random sub-tree whose depth ≤ d̄ − d − 1;
13         Append c′ as r's child;
14 Return r;
```

Breeding Trees Based on Adjacency Lists

The pseudo-code of `CALX`$(\cdot)$ is shown in Alg.10. If the first and second parents are both TGP individuals, an inner tree node t_2 is randomly selected from the second parent, and an adjacency list $\mathbf{L}$ is generated based on the sub-tree under t_2 (lines 3-6). If the first parent is a TGP individual and the second parent is an LGP individual, we randomly select an instruction segment $\mathbf{F}'$ (lines 7-8) and convert it to sub-DAGs. $\mathbf{L}$ is further constructed based on the selected sub-graphs (line 9). Then, we apply `GrowTreeBasedAL`$(\cdot)$ to build a sub-tree based on $\mathbf{L}$, as shown in Alg. 11.

`GrowTreeBasedAL`$(\cdot)$ is a recursive function to construct tree nodes based on $\mathbf{L}$. Specifically, if `GrowTreeBasedAL`$(\cdot)$ accepts an empty $\mathbf{L}$ or

[3]Disconnected graph nodes are converted into adjacency lists with empty adjacent nodes $\mathbf{A}$

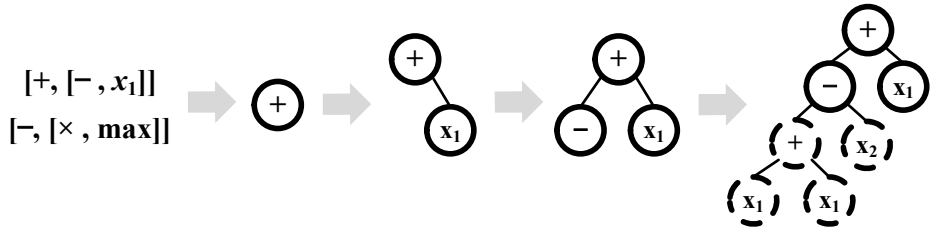

Figure 4.10: An example of constructing a tree by `GrowTreeBasedAL`$(\cdot)$. The dashed tree nodes are randomly generated.

has reached the maximum depth, it returns a random sub-tree to ensure the validity (lines 1-2). Otherwise, `GrowTreeBasedAL`$(\cdot)$ grows a tree node r based on $\mathbf{L}$ (line 3). If r is a function, `GrowTreeBasedAL`$(\cdot)$ checks the adjacency list and recursively applies `GrowTreeBasedAL`$(\cdot)$ to grow the sub-trees of r (lines 4-13). Random sub-trees are constructed if there are no consistent entities in $\mathbf{L}$ (lines 11-12).

Fig. 4.10 is an example of constructing a tree based on an adjacency list. The first item in the adjacency list is "$[+, [-, x_1]]$", and hence the root node of the new sub-tree is "$+$". Since the adjacent nodes of "$+$" are "$-$" (a function) and "x_1" (a terminal), we append "x_1" to the "$+$" and recursively apply `GrowTreeBasedAL`$(\cdot)$ with the second item (i.e., $[-, [\times, \max]]$) in the adjacency list to grow the sub-tree since the function of the second item "$-$" is coincident with the function adjacent node in the first item. Since the adjacent nodes of "$-$" (i.e., "$\times$" and "$\max$") are not included as items in the adjacency list, we randomly generate the sub-trees under "$-$".

Breeding Instructions Based on Adjacency Lists

In `CALX`$(\cdot)$, if the first parent $\mathbf{p}_1$ is an LGP individual, sub-tree and sub-DAGs are respectively selected based on parents' representation, a sub-tree for TGP parent and sub-DAGs for LGP parent (lines 14, 16, 19, and

Algorithm 12: `GrowInstructionBasedAL`

Input: An LGP individual $\mathbf{c}$, adjacency list of the first parent $\mathbf{L}_1$, adjacency list of the second parent $\mathbf{L}_2$, crossover point t_1, instruction range n_1

Output: The LGP offspring $\mathbf{c}$

Randomly remove $|\mathbf{L}_1|$ instructions from $[\mathbf{c}(t_1), \mathbf{c}(t_1 + n_1)]$;
$s \leftarrow t_1 + \texttt{UniformInt}(n_1 - |\mathbf{L}_1| + 1)$;
`// Construct an instruction list`
for $j \leftarrow 1$ *to* $|\mathbf{L}_2|$ **do**
    $[a, \mathbf{A}] \leftarrow \mathbf{L}_2(j)$;
    $f \leftarrow$ randomly generate an instruction whose function is a;
    `// Swap into the program context`
    Insert f to $\mathbf{c}(s)$;
`// Assign registers`
for $j \leftarrow s + |\mathbf{L}_2| - 1$ *to* s **do**
    $[a, \mathbf{A}] \leftarrow \mathbf{L}_2(s + \mathbf{L}_2 - j)$;
    `// Assign destination registers`
    if $\mathbf{c}(j)$ *is not effective to the final output* **then**
        Randomly mutate $\mathbf{c}(j)_d$ until $\mathbf{c}(j)$ is effective;
    `// Assign source registers`
    for $g \leftarrow 1$ *to* $|\mathbf{A}|$ **do**
        $b \leftarrow \mathbf{A}(g)$;
        if b *is a function* **then**
            $\mathbf{L}' \leftarrow$ collect the entity indices from $[j, s]$ where $\mathbf{L}_2(k).fun = b$ and $k \in [j, s]$;
            if $\mathbf{L}' \neq \emptyset$ **then**
                $\mathbf{c}(j)_{s,g} \leftarrow \mathbf{c}(\mathbf{L}'(\texttt{UniformInt}(1, |\mathbf{L}'| + 1)))_d$;
            else
                if $j > 0$ *and* $\texttt{UniformInt}(0, j + 1) - 1 > 0$ **then**
                    $\mathbf{c}(j)_{s,g} \leftarrow \mathbf{c}(\texttt{UniformInt}(1, j + 1))_d$;
        else if b *is a constant* **then**
            $\mathbf{c}(j)_{s,g} \leftarrow b$;
Return $\mathbf{c}$;

21). Specifically, the sub-DAGs are selected by selecting an instruction segment from the LGP parent. Then, adjacency lists $\mathbf{L}_1$ and $\mathbf{L}_2$ are constructed respectively based on the selected sub-tree and sub-graphs (lines 15, 17, 20 and 22). A new instruction sequence is constructed and swapped into $\mathbf{p}_1$ to produce offspring by `GrowInstructionBasedAL`$(\cdot)$ (line 23).

CALX applies `GrowInstructionBasedAL`$(\cdot)$ to construct a new instruction segment for LGP, as shown in Alg. 12. First, $|\mathbf{L}_1|$ instructions are randomly removed (line 1). Then an insertion point s is selected for inserting the new instruction segment (line 2). Instructions are sequentially constructed based on $\mathbf{L}_2$ and swapped into the program context (lines 3-6). To connect the functions and maintain the topological structure of the functions based on $\mathbf{L}_2$, we check the instruction sequence reversely (i.e., from the top of the graph to the bottom) (lines 7-21) so that every newly generated instruction 1) is effective to the final output by altering the destination registers $\mathbf{c}(j)_d$ (lines 9-10), and 2) accepts the inputs from corresponding functions and constants based on $\mathbf{L}_2$ by altering the source registers $\mathbf{c}(j)_{s,g}$ (lines 11-21). Specifically, the effectiveness of an instruction is checked by an $\mathcal{O}(n)$ algorithm [21] (line 9). If the selected instruction is not effective, we randomly mutate the destination register of the instruction until it is effective. `GrowInstructionBasedAL`$(\cdot)$ assigns source registers based on the adjacent node b (line 12). If b is a function, we set the source register of the selected instruction $\mathbf{c}(j)$ as the destination register of a random instruction whose function is coincident with b (lines 14-16). If there is no such an instruction, we set the source register as the destination register of a random instruction precedent to $\mathbf{c}(j)$ (lines 18-19). The constant adjacent nodes replace the source registers directly (lines 20-21).

Fig. 4.11 shows an example of constructing an instruction list based on the adjacency list. First, we construct an instruction list in which the functions (i.e., "$+$" and "$-$") are specified by the adjacency list. Note that the order of functions in the instruction list is reversed to the order in the adjacency list since LGP programs output final results from the bottom.

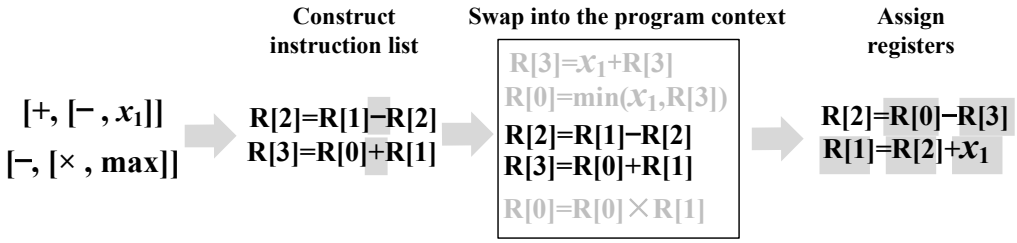

Figure 4.11: An example of constructing instructions by `GrowInstructionBasedAL`$(\cdot)$. Shadowed primitives are the focus of each step.

Second, we swap the newly constructed instruction list into the program context. Third, we adjust the registers in the newly constructed instruction list to maintain the adjacency relationship in the offspring. In this example, we change the destination register R[3] into R[1] to ensure the new instruction list to be effective in the offspring, change R[0] in the second instruction into R[2] and change R[1] into x_1 to fulfill the adjacency relationship "$+ \rightarrow [-, x_1]$". To connect the newly constructed instruction list with the precedent instructions in existing programs, the source registers in the first instruction are also updated.

4.5 Experimental Studies on MRGP for DJSS

4.5.1 Comparison Design

To verify the effectiveness of MRGP-TL, we compare MRGP-TL with three baseline methods. The first two are the basic TGP and LGP. Then, a baseline GP method with two independent sub-populations is developed (denoted as "TLGP"). The two sub-populations independently evolve tree-based and linear representations by the basic genetic operators for each representation and do not exchange genetic materials among representa-

tions. We set the parameters of the compared GP methods based on chapter 3.

For the two algorithms with multiple representations (i.e., TLGP and MRGP-TL), each sub-population has 128 individuals and evolves 200 generations. The parameters of the MRGP-TL are defined based on the baseline method. Specifically, the knowledge transfer rate is defined as 30% by default, without out loss of generality. Since the proposed adjacency list-based operators which are used to transfer knowledge among sub-populations can also exchange the genetic materials for the same representation, the LGP sub-population in MRGP-TL does not apply linear crossover operator, and the TGP sub-population in MRGP-TL reduces the crossover rate from 80% to 50%. The other parameters of TLGP and MRGP-TL are kept the same as in the basic TGP and LGP methods in chapter 3.

We verify the effectiveness of MRGP-TL by twelve DJSS scenarios. The twelve scenarios are $\langle T_{max}, 0.85\rangle$, $\langle T_{max}, 0.95\rangle$, $\langle T_{mean}, 0.85\rangle$, $\langle T_{mean}, 0.95\rangle$, $\langle WT_{mean}, 0.85\rangle$, $\langle WT_{mean}, 0.95\rangle$, $\langle F_{max}, 0.85\rangle$, $\langle F_{max}, 0.95\rangle$, $\langle F_{mean}, 0.85\rangle$, $\langle F_{mean}, 0.95\rangle$, $\langle WF_{mean}, 0.85\rangle$, and $\langle WF_{mean}, 0.95\rangle$.

4.5.2 Experiment Results

Test Performance

Table 4.4 shows the average test performance of the compared methods in solving DJSS problems. We perform a Friedman test ($\alpha = 0.05$) with a Bonferroni correction on the test performance of the compared methods. The null hypothesis of the Friedman test is that there is no significant difference in the test performance of the compared methods.

The p-value of the Friedman test is 0.009, which indicates a significant difference (i.e., alternative hypothesis) among the compared methods. Moreover, MRGP-TL has the best (i.e., smallest) mean rank of test performance among all the compared methods, with very promising pair-

Table 4.4: The mean test performance (std.) of the compared methods.

Datasets or scenarios	TLGP	TGP	LGP	MRGP-TL
$\langle Tmax, 0.85\rangle$	1939.8 (50.4) $\approx$	1928.4 (40.4) $\approx$	1956.3 (53.8) $-$	**1926.9 (79.3)**
$\langle Tmax, 0.95\rangle$	4009.2 (98.9) $-$	4060.6 (116) $-$	3999.2 (90.9) $\approx$	**3964.2 (89)**
$\langle Tmean, 0.85\rangle$	**417.0 (3.2)** $\approx$	417.3 (2.5) $\approx$	417.9 (2.3) $\approx$	417.2 (3.4)
$\langle Tmean, 0.95\rangle$	1116.3 (9.3) $\approx$	**1116.2 (10)** $\approx$	1118.2 (10.7) $\approx$	1116.5 (10.6)
$\langle WTmean, 0.85\rangle$	725.8 (6.1) $\approx$	727.5 (6.5) $-$	724.3 (5.4) $\approx$	**724.0 (5.7)**
$\langle WTmean, 0.95\rangle$	1730.4 (23.6) $\approx$	1747.4 (29.6) $-$	1729.6 (27.7) $\approx$	**1723.5 (19.4)**
$\langle Fmax, 0.85\rangle$	2506.6 (50.3) $\approx$	2494.3 (30) $\approx$	2509.8 (58.8) $\approx$	**2493.0 (33.3)**
$\langle Fmax, 0.95\rangle$	**4544.3 (98.5)** $\approx$	4572.3 (96.5) $\approx$	4585.4 (126.1) $\approx$	4553.0 (110.8)
$\langle Fmean, 0.85\rangle$	864.0 (3.2) $\approx$	863.2 (4.2) $\approx$	864.7 (4) $\approx$	**862.8 (2.9)**
$\langle Fmean, 0.95\rangle$	**1564.9 (10.3)** $\approx$	1565.4 (8.5) $\approx$	1566.8 (10.8) $\approx$	1565.3 (10.6)
$\langle WFmean, 0.85\rangle$	1704.0 (10.2) $\approx$	1705.4 (7.5) $\approx$	**1702.6 (7)** $\approx$	1702.8 (5.6)
$\langle WFmean, 0.95\rangle$	2718.4 (26.4) $\approx$	2730.1 (29.3) $-$	2715.8 (16.4) $\approx$	**2711.0 (20.5)**
win-draw-lose	0-11-1	0-8-4	0-11-1	
Mean rank	2.42	3.0	3.08	**1.5**
p-value (vs. MRGP-TL)	0.492	0.026	0.016	

wise comparison p-values with other compared methods. The results and statistical analyses confirm that the proposed MRGP-TL has a significantly better overall performance than the other three compared methods.

To further investigate the effectiveness of the compared methods on different datasets, table 4.4 shows the results of the Wilcoxon rank-sum test with Bonferroni correction and an α of 0.05 over the test performance of the compared methods. $+$, $-$, and $\approx$ denote that a certain compared method is significantly better than, worse than, or performs similarly to the proposed MRGP-TL respectively, based on the Wilcoxon rank-sum test. The best mean performance is highlighted in bold font. We see that in most datasets and scenarios, MRGP-TL has a very competitive performance with the compared methods. More specifically, MRGP-TL has the best mean performance on 7 of 12 datasets and scenarios. The results confirm that sharing knowledge between tree-based and linear representation

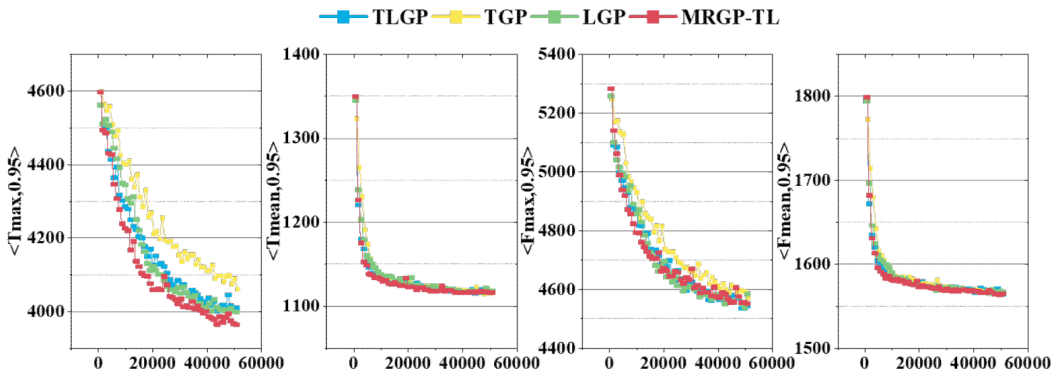

Figure 4.12: Test performance of the compared methods over generations in the four example DJSS problems. X-axis: fitness evaluations. Y-axis: average test objective values for DJSS problems.

successfully improves the effectiveness of GP methods.

Training Performance

To analyze the learning ability of the compared methods, Fig. 4.12 shows the test performance of the compared methods over generations in four example problems. Specifically, we select four DJSS scenarios with a high utilization level (i.e., 0.95) as the example problems since the DJSS scenarios with a high utilization level have better real-world practical value.

MRGP-TL (i.e., red curves) has smaller test performances within fewer fitness evaluations than the others in many cases, such as $\langle Tmax, 0.95\rangle$. In some other cases, though MRGP-TL levels off at a similar test performance with the other compared methods, MRGP-TL achieves the test performance earlier than the compared methods at the early stage of the evolution. The results imply that MRGP-TL has a very competitive training performance with other compared methods in DJSS problems and can find solutions with better effectiveness within fewer simulation times in some

specific cases.

Program Size

To further understand the evolution of MRGP-TL, we analyze the average program size of the population in all the compared methods for solving four example problems, as shown in Fig. 4.13. Specifically, we show the program size of tree-based and linear programs respectively in TLGP and MRGP-TL, denoted by "-T" and "-L" (e.g., tree-based programs in TLGP are denoted as "TLGP-T"). We use the tree nodes to denote the program size of tree-based programs and use the number of effective instructions multiplied by a factor of 2.0 to denote the program size of linear programs. We can see that the average program size from the same representation grows similarly in all the tested problems. For example, TLGP-L, LGP, and MRGP-L all grow from about 20 to about 50. The similar growing curves of the same representation confirm that the proposed cross-representation adjacency list-based crossover operator has a similar variation step size with basic genetic operators and does not significantly change the average program size of the population. Fully utilizing the interplay between tree-based and linear representations improves the effectiveness of solutions without enlarging the program size of the solutions.

Parameter Sensitivity Analyses

The knowledge transfer rate among representations θ_t is a newly introduced parameter. To investigate the influence of θ_t on performance, MRGP-TL with different transfer rates are compared in this section. Specifically, we investigate the performance of MRGP-TL with a θ_t of 0%, 10%, 30%, 50%, and 70% respectively, which are denoted as TL0 (i.e., TLGP), TL10, TL30, TL50, and TL70.

The test performances of MRGP-TL with different θ_ts are shown in Fig. 4.14. We see that MRGP-TL methods with $\theta_t > 0$ on average have smaller

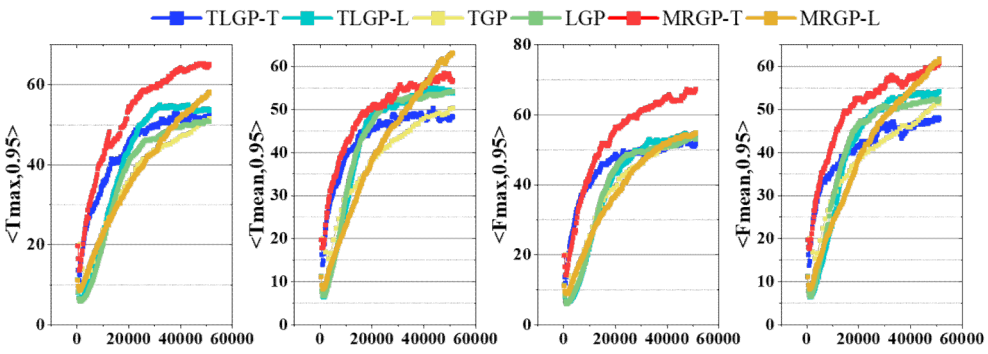

Figure 4.13: The average program size of the population from the compared methods over generations over 50 independent runs. X-axis: fitness evaluations, Y-axis: the average program size of the population.

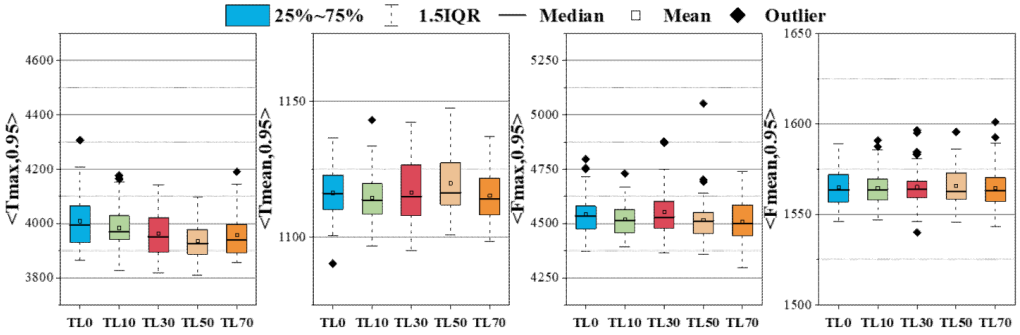

Figure 4.14: The box plots on the test performance of MRGP-TL with different θ_t values over 50 independent runs.

(i.e., better) objective values than MRGP without any knowledge sharing (i.e., TL0 or TLGP). Besides, TL10, TL30, TL50, and TL70 have statistically similar test performance in most cases. But in some cases such as $\langle T_{max}, 0.95\rangle$ and $\langle F_{max}, 0.95\rangle$, the increase of θ value improves the performance of MRGP-TL on average. To conclude, θ_t, the knowledge transfer rate among representations shows robust performance in principle, but tuning on specific scenarios has the potential to further improve MRGP-

Table 4.5: The average test performance (std.) of exchanging search information by basic crossover operators and CALX.

Datasets and scenarios	MP-TGP	MP-LGP	MRGP-TL
⟨Tmean,0.85⟩	418 (4.8) ≈	417.2 (3.5) ≈	**417.2 (3.4)**
⟨Tmean,0.95⟩	1118 (8.6) ≈	**1115.3 (9.6)** ≈	1116.5 (10.6)
⟨WTmean,0.85⟩	727.3 (6) −	724.4 (6.2) ≈	**724 (5.7)**
⟨WTmean,0.95⟩	1737.9 (25.2) −	1724.1 (21.6) ≈	**1723.5 (19.4)**
⟨WFmean,0.85⟩	1706.5 (6.4) −	**1702.4 (6.5)** ≈	1702.8 (5.6)
⟨WFmean,0.95⟩	2722.9 (30.2) ≈	2717.6 (23.9) ≈	**2711 (20.5)**
mean rank	2.92	1.67	**1.42**
p-values	0.024	1.0	

TL performance.

Effectiveness of CALX

The superior performance of MRGP-TL stems from the proposed CALX operator that exchanges building blocks between the two GP representations. To further verify the effectiveness of CALX, we implement two multi-population GP methods.

Specifically, each of the two multi-population GP methods has two sub-populations using the same GP representation (i.e., tree-based or linear representations), denoted as MP-TGP and MP-LGP respectively. The two sub-populations exchange building blocks via basic crossover operators with the same exchanging rate (i.e., $\theta_t = 30\%$). Comparing the effectiveness of MP-T(L)GP and MRGP-TL validates the performance gain caused by the proposed crossover operator.

Table 4.5 shows the test performance of the three compared methods for solving six example problems. We apply the Friedman test ($\alpha = 0.05$) with a Bonferroni correction to analyze the overall performance. The p-value of the Friedman test is 0.018, indicating a significant difference in the test performance of the compared methods (i.e., alternative hypothesis).

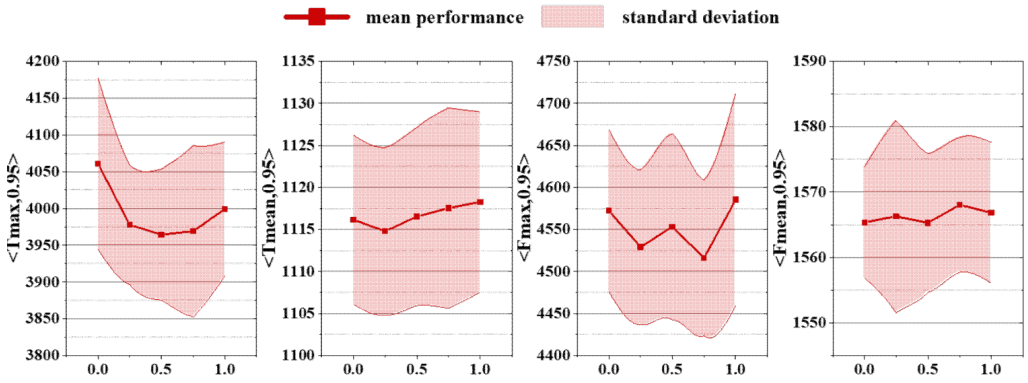

Figure 4.15: Test performance of different population ratios in MRGP-TL. X-axis: LGP population proportion. Y-axis: test performance of MRGP-TL.

The mean ranks given by the Friedman test verify that MRGP-TL has the best test performance among the three compared methods. Specifically, based on the p-values of a pair-wise comparison, we know that MRGP-TL is significantly better than MP-TGP. Although the performance gain of MRGP-TL is not significant compared with MP-LGP, MRGP-TL has better mean performance than MP-LGP on four of the six problems. We believe that the proposed CALX effectively improves the performance of GP methods.

Representations With Various Computation Budgets

Different problems often have their own suitable GP representations, implying that allocating different computation resources to different representations in MRGP-TL might be helpful to the performance of MRGP-TL. To investigate the impact of computation budgets on different GP representations, we adjust the allocation of computation resources by increasing the LGP population proportion from 0% to 100% (and decreasing the TGP population proportion from 100% to 0%). Specifically, we investigate five

settings of LGP proportions, which are 0%, 25%, 50%, 75%, and 100%. The average test performance and standard deviation of MRGP-TL are shown in Fig. 4.15.

In the four example problems, we can see a "V" shape roughly on the mean test performance. It implies that MRGP-TL achieves a relatively good mean test performance and standard deviation when LGP and TGP share a similar proportion of computation resources (i.e., similarly large sub-populations). Although the performance of MRGP-TL can be further improved by adjusting the proportion of TGP and LGP population for a certain problem, uniformly allocating the training resources to different representations is a relatively good and robust setting for MRGP-TL.

Benefit of Cross-representation Knowledge Sharing

To verify that the knowledge sharing among representations is effective in MRGP-TL evolution, this section investigates the ratio that each GP representation produces the best-of-run individuals at every generation over 50 independent runs. Fig. 4.16 shows the average ratio of tree-based and linear representations producing the best-of-run individuals over generations in MRGP-TL. For comparison, Fig. 4.17 shows the ratio that different representations produce the best-of-run individuals without knowledge sharing (i.e., tree-based and linear representations in TLGP).

The two figures show that the two representations in MRGP are much less sensitive than in TLGP. In MRGP (i.e., Fig. 4.16, both tree-based and linear representations produce the best-of-run individuals with a similar ratio (i.e., 0.4$\sim$0.6) in all the selected problems. But in TLGP (i.e., Fig. 4.17), the two representations produce the best-of-run individuals with an imbalanced ratio (e.g., linear representation produces the best-of-run individuals with nearly 90% of runs in the course of evolution in Tower). It confirms that the superior representation in MRGP (e.g., linear representation in Tower benchmark) successfully improves the effectiveness of the other representation (e.g., tree-based representation), which might reduce

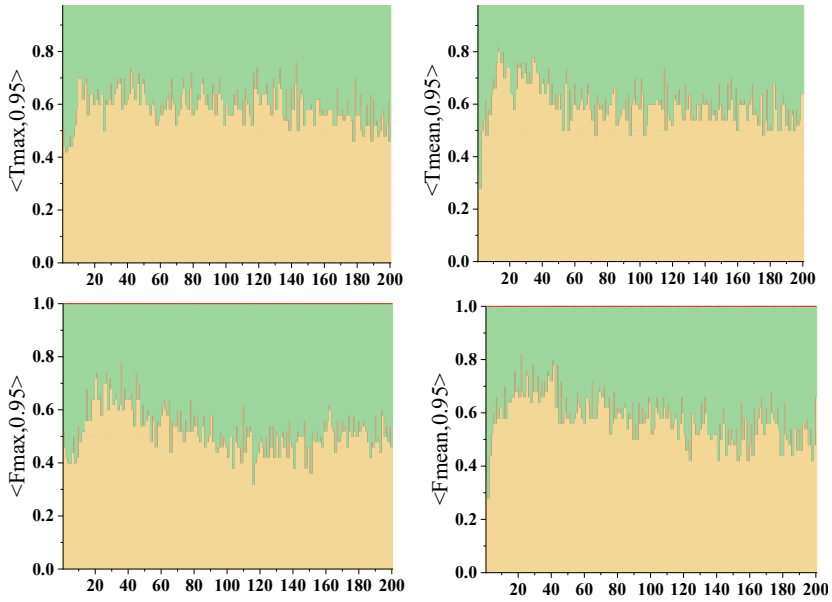

Figure 4.16: The average ratio of tree-based and linear representations producing the best-of-run individuals over generations in MRGP-TL. X-axis: generations, Y-axis: ratio of producing the best-of-run individuals. The green (i.e., upper) area denotes the ratio of linear representation, and the yellow (i.e., lower) area denotes the ratio of tree-based representation.

the dependency on the domain knowledge of GP representations. Furthermore, the improvement of the inferior representation confirms that the knowledge sharing between representations effectively helps both GP representations to find more effective solutions.

MRGP effectively shares search information between representations. When a representation finds better solutions, the other representation in MRGP can be efficiently improved. For example, in $\langle Fmax, 0.95\rangle$, LGP has better solutions at the beginning of evolution. We can see that the tree-based representation finds more effective solutions with the help of LGP search information from generations 10 to 60. After 60 generations, effective solutions in the tree-based representation in turn help the linear rep-

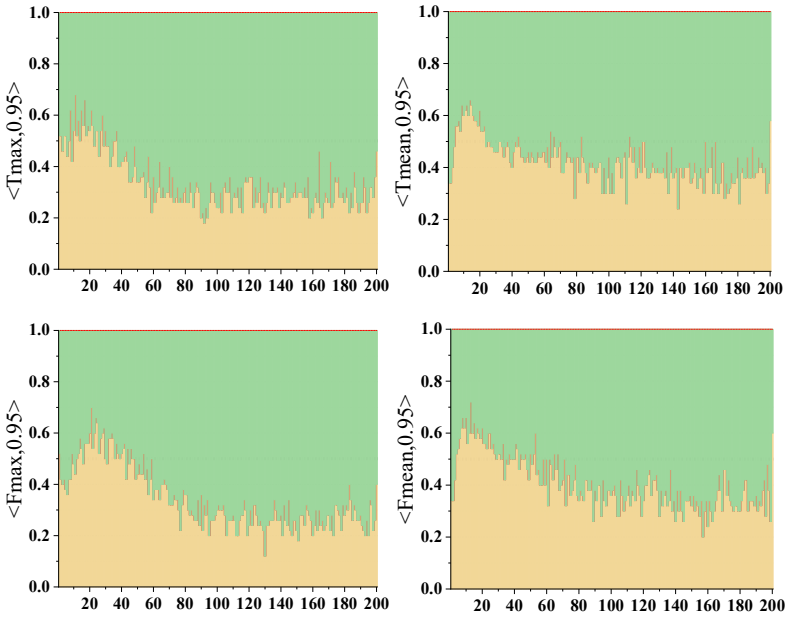

Figure 4.17: The average ratio of tree-based and linear representations producing the best-of-run individuals over generations in TLGP (i.e., **without knowledge sharing**). X-axis: generations, Y-axis: ratio of producing the best-of-run individuals.

resentation find effective solutions and catch up with the tree-based representation at about generation 80. However, in TLGP for $\langle Fmax, 0.95\rangle$, the best-of-run individuals are mainly produced by the linear representation during most of the evolution (i.e., the green area covers over 60% at each generation). Although we see a wave of the ratio between the tree-based and linear representation in $\langle Fmax, 0.95\rangle$ of TLGP, the linear representation cannot help tree-based representation to find more effective solutions in the latter evolution. The results confirm that knowledge sharing between tree-based and linear representations improves the performance of both tree-based and linear GP.

$\mathbf{L}_{\text{TGP}}$ = ([×, [max, PT]] [max, [×, +]] [×, [WIQ, +]] [+, [+, PT]] [+, [+, −]] [+, [×, PT]] [×, [NPT, NINQ]] [−, [NOR, NPT]] [+, [+, min]] [+, [×, PT]] [×, [NPT, NINQ]] [min, [÷, +]] [÷, [max, −]] [max, [max, min]] [max, [NWT, NINQ]] [min, [rFDD, PT]] [−, [max,×]] [max, [WIQ, WKR]] [×, [OWT, WIQ]] [+, [WKR, rFDD]])

$\mathbf{L}_{\text{LGP}}$ = ([×, [min, max]] [min, [−, −]] [max, [−, −]] [−, [max, min]] [min, [min, min]] [min, [max, +]] [max, [max, max]] [max, [−, −]] [−, [max, max]] [−, [×,×]] [×, [max, NINQ]] [max, [×, NPT]] [×, [max, PT]] [max, [+,+]] [min, [÷, ÷]] [+, [÷, +]] [+, [+, ÷]] [÷, [÷, NINQ]] [÷, [×, −]] [×, [÷, min]] [÷, [÷, min]] [max, [+, max]] [max, [min,×]] [×, [PT −]] [min, [max, max]] [max, [+,+]] [−, [+,NPT]] [+, [+,+]] [+, [NPT, min]] [min, [TIS,÷]] [÷, [−,+]] [−, [+, max]] [max, [×, ÷]] [÷, [WIQ, +]] [+, [+, PT]] [+, [+, max]] [+, [×, PT]] [max, [W, min]] [min, [×, NPT]] [×, [NPT, NINQ]])

Figure 4.18: The adjacency lists of the outputted TGP and LGP heuristics from a run in $\langle Fmean, 0.95\rangle$. The dark shadow highlights the shared adjacency of primitives between the two adjacency lists.

Example Analyses on Adjacency Lists

The proposed adjacency list-based crossover shares the search information between tree-based and linear representations by the adjacency of primitives. To have a better understanding of knowledge sharing via adjacency lists, this section analyzes the shared knowledge (i.e., primitive adjacency) in two example adjacency lists, where each item in the adjacency lists contains two pairs of primitive connections. Fig. 4.18 shows two adjacency lists of the best-of-run individuals from the two representations, respectively, of the same run for solving the $\langle Fmean, 0.95\rangle$ DJSS problem. If a primitive connection can be seen in both of the adjacency lists, we highlight the connection with a dark shadow. For example, as the first shadowed item in $\mathbf{L}_{TGP}$ shows the adjacency from "$\times$" to "max", the adjacency items with the same connection in $\mathbf{L}_{LGP}$ are shadowed (e.g., the last item at the second line of $\mathbf{L}_{LGP}$). We can see that the adjacency lists of the output heuristics with tree-based and linear representations have a large number of shared members. For example, both of them prefer concatenating "PT", "NPT", and "NINQ" with "$\times$" and "$+$", which further form the shared building blocks such as "$[+, [\times, PT]]$" and "$[\times, [NPT, NINQ]]$".

Furthermore, the adjacency lists from different representations have distinct characteristics. Because of short and wide tree structures, the adjacency list of the tree-based representation considers more distinct input features, such as "WKR" and "rFDD". In contrast, the adjacency list of the linear representation uses a large number of "max" and "min" to assemble the final result.

Overall, by exchanging adjacency lists, tree-based and linear representations can 1) learn the shared adjacency of effective solutions and 2) learn the distinct characteristics of the other representation.

4.6 Chapter Summary

The main goal of this chapter is to develop graph-based search mechanisms based on the graph characteristics of LGP. We first developed and investigated four possible graph-based operators, including frequency-based, adjacency matrix-based, adjacency list-based, and effective instruction-based operators. The results show that adjacency lists are effective representations of the graph characteristics since they can represent both primitive frequency and topological information.

In light of this finding, we propose to utilize the interplay of different GP representations to automatically identify the most suitable representation for the problem at hand, that is, the multi-representation GP method. We implemented a multi-representation GP method based on tree-based and linear GP representations, denoted as MRGP-TL. The MRGP-TL includes a novel cross-representation adjacency list-based crossover operator to exchange building blocks between tree-based and linear GP representations. To the best of our knowledge, it is the first work highlighting that the interplay among different GP representations is useful for improving GP performance.

The experimental studies on DJSS problems show that the proposed MRGP-TL significantly improves the performance of baseline GP meth-

ods without considering the interplay among different representations. MRGP-TL can take advantage of a suitable GP representation in solving a certain problem, leading to a wider application spectrum. The results also confirm that the performance gain of MRGP-TL stems from the proposed crossover operator which makes full use of the interplay between GP representations. Fully utilizing different GP representations to enhance the search on a single task is a potential direction, that is worthy to be further investigated in other domains as well.

This chapter shows the potential of developing more effective search mechanisms (i.e., graph-based search mechanisms) to enhance LGPHH performance. However, the graph-based search mechanisms do not make full use of the domain knowledge of DJSS. To further improve the effectiveness of LGP search for DJSS, the next chapter will incorporate domain knowledge into the LGP search via grammar-guided techniques.

Chapter 5

Grammar-guided LGPHH for DJSS

Incorporating domain knowledge into LGP search is a potential way to improve the effectiveness and efficiency of LGPHH. However, existing LGPHH studies do not make full use of the domain knowledge. Without the help of domain knowledge, LGP might have to explore the entire search space, which is inefficient. To incorporate the domain knowledge in LGPHH methods for DJSS problems, this chapter proposes a grammar-guided LGPHH method. Specifically, we design grammar rules based on the domain knowledge to reduce redundant building blocks and less effective LGP programs.

5.1 Introduction

DJSS has undergone years of development, and we have accumulated some domain knowledge. For example, the priority of jobs is likely correlated to their processing time (e.g., the heuristic rule of shortest processing time first). In different scenarios, it is also advisable to apply different heuristics (e.g., applying energy efficiency heuristics when the energy price is high.). However, existing LGPHH studies do not make full use

of the domain knowledge of DJSS. LGPHH has to search the entire search space which includes a large number of redundant solutions.

In this chapter, we propose to use grammar rules to constrain LGP search space based on domain knowledge. Grammar rules are used to describe the potential effective structures of all possible dispatching rules. Rather than searching in the entire 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. It is easier for humans to make a more detailed analysis 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. It is necessary to design an LGP-based grammar system that allows humans to describe their constraints in grammar and 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. We need grammar-guided genetic operators to maintain the grammar rules in breeding. Last but not least, we need to carefully design proper grammar rules for DJSS problems based on our domain knowledge to reduce the search space as much as possible without losing promising solutions.

This chapter focuses on incorporating the domain knowledge of flow control operations into LGP search. Flow control operations are important in designing dispatching rules [96, 158, 236]. For example, DJSS problems need IF operations to prioritize energy-efficient jobs if the power cost rate is high and prioritize other jobs otherwise. Moreover, many human pro-

grams and expertise knowledge need flow control operations to describe their complex procedure. Finding an effective way to evolve flow control operations in dispatching rules facilitates humans to make use of domain knowledge and improve the flexibility of dispatching rules.

However, existing studies of LGPHH for DJSS did not effectively evolve flow control operations, mainly due to the following three challenges.

1. *Dimension inconsistency*: flow control operations likely use input features with different physical dimensions (e.g., if "3 meters" is larger than "2 seconds"). The inconsistent dimension makes dispatching rules difficult to be understood.

2. *Inactive sub-rules*: flow control operations easily lead to inactive sub-rules (i.e., introns [21]). For example, the contradictory conditions (i.e., conditions that are always false) of IF operations easily skip a large number of instructions, which makes a dispatching rule quite naive. The variation on flow control operations also likely makes a huge difference in the behaviors of dispatching rules since the variation often substantially changes the data flow in dispatching rules.

3. *Ineffective sub-rules*: the effectiveness of flow control operations is highly dependent on their sub-rules. A flow control operation is useful only when their sub-rules are effective. To ensure the effectiveness of flow control operations, we have to take the effectiveness of sub-rules into consideration.

Existing studies of LGP do not fully consider these three challenges when evolving flow control operations. They simply introduce flow control operations into their primitive set and neglect the characteristics of flow control operations. This inevitably leads to many redundant and obscure solutions in the LGP search space, which impairs the search effectiveness and efficiency of LGP.

5.1.1 Chapter Goals

The goal of this chapter is to *develop an effective grammar-guided LGPHH and design effective grammar rules based on domain knowledge for DJSS problems*. First, we design a grammar system to enable users to define their own grammar flexibly by extending the Backus-Naur Form (BNF) [130]. Different from basic BNF, the proposed grammar system allows users to 1) see their constraints as modules and reuse them in the definition and 2) have a more precise restriction (e.g., program length) on different program parts. Second, we propose three grammar-guided genetic operators for LGP to evolve based on grammar rules. Third, we design a set of LGP-based grammar rules to introduce an example of flow control operations, IF operations, for solving DJSS problems. Specifically, this chapter has the following research objectives:

1. Develop an LGP-based grammar system to define constraints on LGP search spaces based on the domain knowledge.

2. Develop grammar-guided genetic operators to search for LGP solutions based on the defined grammar rules.

3. Incorporate domain knowledge of DJSS to evolve flow control operations in the grammar-guided LGPHH. More specifically, we focus on IF operations as a step towards evolving flow control operations in dispatching rules. IF operations are the basis of many other flow control operations.

4. Verify the effectiveness of the grammar-guided LGPHH with IF operations for solving different DJSS scenarios.

5. Analyse the interpretability of the output rules of grammar-guided LGPHH with IF operations.

5.1.2 Chapter Organization

The rest of this chapter is organized as follows. Section 5.2 first proposes the grammar-guided LGP in detail. Based on the framework of the

grammar-guided LGP, we incorporate the domain knowledge of IF operations in LGP search by designing grammar rules based on three principles in section 5.3. Section 5.4 introduces the experiment design in this chapter. Section 5.5 analyzes the performance of the proposed method, including the test effectiveness, training efficiency, and program interpretability. Section 5.6 further performs an ablation analysis and analyzes the terminal patterns with IF operations. Finally, section 5.7 draws the conclusions of this chapter.

5.2 Proposed Method

This chapter proposes a new grammar-guided LGP (G2LGP) for DJSS problems. Specifically, we first 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 MCFG and the grammar-guided genetic operators are introduced as follows.

5.2.1 Module Context-free Grammar

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

Table 5.1: Keywords in MCFG.

`PROGRAM`	The starting point when producing programs based on grammar.
`defset`	Specify the definition of a set.
"{" 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.
$\sim$	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.

ations like union and intersection on the primitive sets. Second, MCFG allows *derivation rule*s 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 5.1 illustrates the keywords in MCFG and their corresponding meaning, and Fig. 5.1 is an example of MCFG that restricts the search space of synthesizing an LGP program to calculate the values of three basic statistical metrics (i.e., sum, average, and variance).[1] In the beginning, Fig. 5.1 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`".

[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 calculate the values of the three metrics.

```
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 5.1: A complete example of MCFG. "add, sub, mul, div" denote the four basic arithmetic operations.

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.

There are four derivation rules in Fig. 5.1. 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".

```
< O \ {add}\ I \ O > *5
```

destination register set	function set	1st source register set	2nd source register set	maximum repeating time

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 module*s. For instance, in Fig. 5.1, 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.

5.2.2 Evolutionary Framework

The evolutionary framework of G2LGP follows the evolutionary framework of basic LGP for solving DJSS problems in chapter 3. 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. 5.2. 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 grammar-guided genetic operators are necessary to produce offspring conforming the given grammar constraints. These new components are introduced in detail as follows.

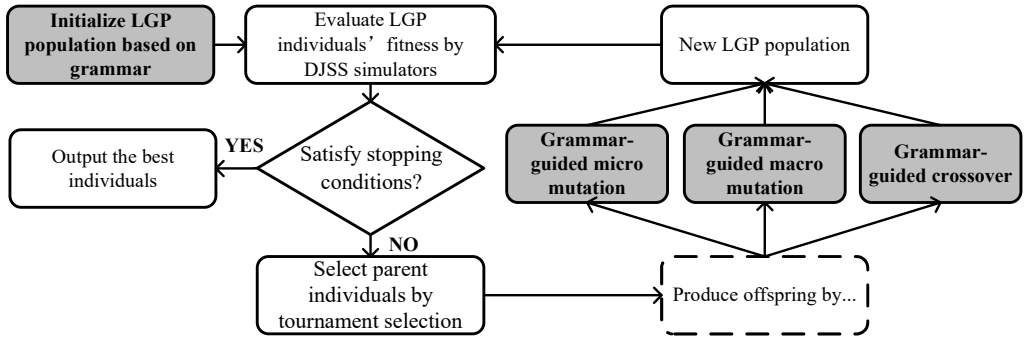

Figure 5.2: The evolutionary framework of G2LGP. The dark steps are the newly proposed steps.

LGP Program Initialization

Based on the grammar rules, we generate LGP individuals by first constructing a derivation tree and second stochastically generating a list of instructions based on the leaf nodes of the derivation tree. Fig. 5.3 shows an example of the initialization of an LGP individual based on the grammar rules in Fig. 5.7.

We construct the derivation tree in a top-down way. `PROGRAM` is the starting symbol in the derivation. Each tree node is a module, remembering the input arguments specified in the grammar rules and specifying the repeating time of sub-modules. The input arguments are actually feasible primitives in different positions of the program. We derive the tree nodes to sub-trees based on the grammar rules in a recursive manner and finally end up with instruction modules (abbreviated as "instr<...>" in Fig. 5.3). All the leaf nodes of derivation trees must be instruction modules. Each instruction module specifies the feasible primitives for its four components, including a destination register, a function, the first source register, and the second source register. For example, the leftmost leaf node specifies that 1) the destination register is one of the eight registers (i.e., `{REG}`), 2) the

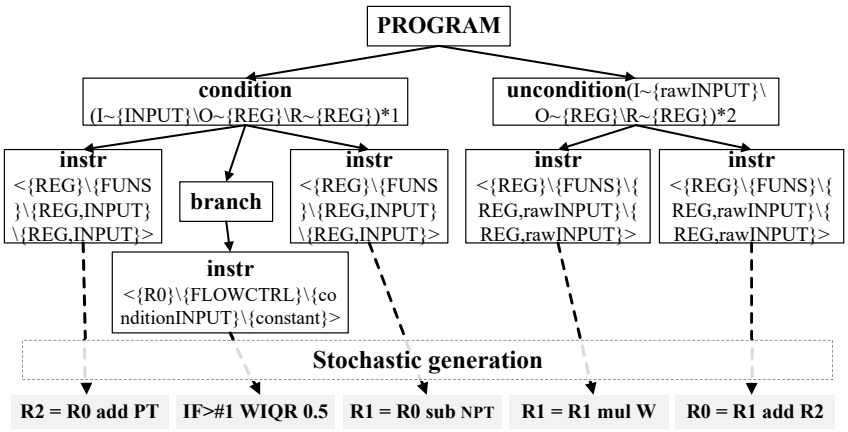

Figure 5.3: An example of initializing an LGP individual based on the proposed grammar rules. The upper part is the derivation tree, and the lower part is the LGP individual.

function is one of the six arithmetic functions (i.e., `{FUNS}`), and 3) the two source registers are registers, normalized terminals, or raw job shop features (i.e., `{REG,INPUT}`). We generate one instruction for each leaf node by randomly selecting one of the primitives given by each component to form an LGP instruction. Specifically, there are five leaf nodes in Fig. 5.3, which leads to an LGP individual with five instructions. The LGP individual is obtained by sequentially arranging generated instructions from the left to the right.

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

```
<{REG}\{FUNS}\{REG,INPUT}\{REG,INPUT}>.
```

Thus, the `R0` is likely changed to other registers and input features.

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 $\mathbf{f}$ consists of a derivation tree $\mathbf{T_f}$ and a list of instructions $\mathbf{I_f}$. 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. 13[2].

When grammar-guided macro mutation accepts an individual $\mathbf{f}$, 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 encour-

[2] 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.

Algorithm 13: `Grammar-guided macro mutation`

Input: An LGP individual $\mathbf{f}$, add instruction rate θ_a, growing node rate θ_g
Output: An offspring $\mathbf{f}'$

$\mathbf{f}' \leftarrow \mathbf{f}$;
for $j \leftarrow 1$ *to* 50 **do**
  $\mathbf{C} \leftarrow$ clone $\mathbf{f}$;
  if $\texttt{Uniform}(0,1) < \theta_g$ **then**
    $m \leftarrow$ randomly pick a tree node from $\mathbf{T_C}$, $\mathbf{s.t.}$ $\overline{|\mathbf{D}(m)|} > 1$;
    $d \leftarrow$ randomly pick a derivation from $\mathbf{D}(m)$;
    if $(\texttt{Uniform}(0,1) < \theta_a$ *and* $|\mathbf{D}(m)| < \overline{|\mathbf{D}(m)|})$ *or* $|\mathbf{D}(m)| = 1$ **then**
      Grow a new derivation d' from m and recursively derive d' if d' does not only contains instruction modules;
      Add d' to $\mathbf{D}(m)$ right before d;
      $\mathbf{I}' \leftarrow$ construct instructions based on d' and its derivation;
      Add $\mathbf{I}'$ to $\mathbf{I_C}$, right before the first instruction derived from d;
    else if $|\mathbf{D}(m)| = \overline{|\mathbf{D}(m)|}$ *or* $|\mathbf{D}(m)| > 1$ **then**
      Remove the instructions derived from d from $\mathbf{I_C}$;
      Remove d from $\mathbf{D}(m)$;
  else
    $m \leftarrow$ randomly pick a tree node from $\mathbf{T_C}$;
    $d \leftarrow$ randomly pick a derivation from $\mathbf{D}(m)$;
    Grow a new derivation d' from m and recursively derive d' if d' does not only contains instruction modules;
    Add d' to $\mathbf{D}(m)$ right before d;
    $\mathbf{I}' \leftarrow$ construct instructions based on d' and its derivation;
    Add $\mathbf{I}'$ to $\mathbf{I_C}$, right before the first instruction derived from d;
    Remove the instructions derived from d from $\mathbf{I_C}$;
    Remove d from $\mathbf{D}(m)$;
  for $i' \leftarrow$ *reversely read from* $\mathbf{I}'$ **do**
    if i' *is an intron in* $\mathbf{C}$ **then**
      Alter the destination register of i' and try to make i' effective.
  if $|\mathbf{I_C}| \in [|\mathbf{I_C}|, \overline{|\mathbf{I_C}|}]$ **then**
    $\mathbf{f}' \leftarrow \mathbf{C}$;
    break;
Return $\mathbf{f}'$;

ages 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 5.2.2 and insert $\mathbf{I}'$ to $\mathbf{I_f}$ right before the first instruction derived from d. If grammar-guided macro mutation decides to remove instructions, the mutation operator 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. 5.1, 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. 5.4 is an example of the grammar-guided macro mutation. First, the grammar-guided macro mutation selects `getSum` derivation tree node (i.e., the red node in Fig. 5.4) and its second derivation (i.e., the blue child nodes). To add instructions, the grammar-guided macro mutation derives the `getSum` node based on the grammar. In this example, the grammar-guided macro mutation grows one new instruction module and inserts it 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 pro-

[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'.

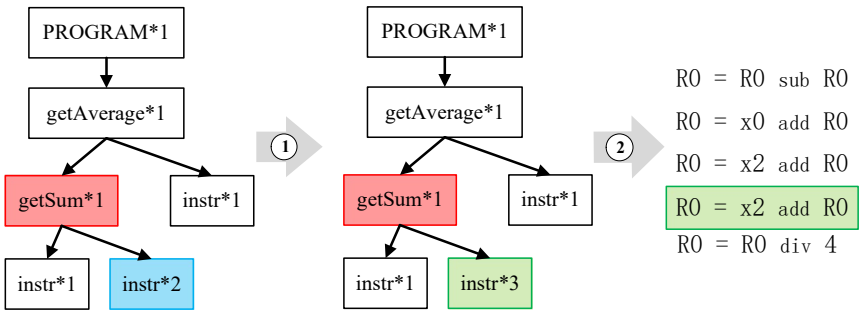

Figure 5.4: An example of adding instructions by the grammar-guided macro mutation. The macro mutation operator first modifies the derivation tree and second mutates the instruction list.

gram, as shown on the right of Fig. 5.4.

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. 14. To avoid the variation step size is limited by poorly defined grammars (e.g., every pair of the same type of derivation nodes, i.e., crossover points, only have one instruction, which makes the crossover operator only swap one instruction each time.), the grammar-guided crossover selects more than one crossover point for each breeding.

First, the grammar-guided crossover selects several 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 10). 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 parameter values). For example, the two `getAverage` in the derivation rule of `getVariance` in Fig. 5.1 are not the same type. They have the

Algorithm 14: `Grammar-guided crossover`

Input: An LGP individual donor $\mathbf{f}_d$, an LGP individual receiver $\mathbf{f}_r$, the maximum number of crossover points N, growing node rate θ_g
Output: An LGP offspring $\mathbf{f}_r$

1 $\mathbf{L}_d \leftarrow \emptyset$, $\mathbf{L}_r \leftarrow \emptyset$;
2 $n \leftarrow$ `UniformInt(1,N+1);`
3 **for** $j \leftarrow 1$ *to* 50 **do**
4 $m_d, m_r \leftarrow null$;
5 **if** $Uniform(0,1) < \theta_g$ **then**
6 $m_d \leftarrow$ randomly pick a tree node from $\mathbf{T}_{\mathbf{f}_d}$, $\mathbf{s.t.}$ $\overline{|\mathbf{D}(m_d)|} > 1$;
7 **else**
8 $m_d \leftarrow$ randomly pick a tree node from $\mathbf{T}_{\mathbf{f}_d}$;
9 **if** $m_d \neq null$ **then**
10 $m_r \leftarrow$ randomly pick a tree node from $\mathbf{T}_{\mathbf{f}_r}$, $\mathbf{s.t.}$ $m_r = m_d$;
11 **if** $m_r \neq null$ **then**
12 **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**
13 Add m_d into $\mathbf{L}_d$;
14 Add m_r into $\mathbf{L}_r$;
15 **if** $|\mathbf{L}_d| \geq n$ **then**
16 **break**;
17 **for** $j \leftarrow 0$ *to* $|\mathbf{L}_d| - 1$ **do**
18 $m_d \leftarrow \mathbf{L}_d[j]$, $m_r \leftarrow \mathbf{L}_r[j]$;
19 $[\mathbf{d}_d, \mathbf{I}_d] \leftarrow$ randomly pick a sub-list of derivations and their corresponding instructions from $\mathbf{D}(m_d)$;
20 $[\mathbf{d}_r, \mathbf{I}_r] \leftarrow$ randomly pick a sub-list of derivations and their corresponding instructions from $\mathbf{D}(m_r)$;
21 **if** $|I_{\mathbf{f}_r}| - |\mathbf{I}_r| + |\mathbf{I}_d| \in [\underline{|\mathbf{I_C}|}, \overline{|\mathbf{I_C}|}]$ **then**
22 Replace $\mathbf{d}_r$ and $\mathbf{I}_r$ with the clone of $\mathbf{d}_d$ and $\mathbf{I}_d$ respectively.
23 Update argument assignment to $\mathbf{d}_r$ and update $\mathbf{I}_r$ if the primitives are not consistent with the derivation tree node.
24 **Return** $\mathbf{f}_r$;

same module name 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, the grammar-guided crossover collects the pair of crossover points into the lists $\mathbf{L}_d$ and $\mathbf{L}_r$ respectively (lines 12 to 16). 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 $\mathcal{A}$.

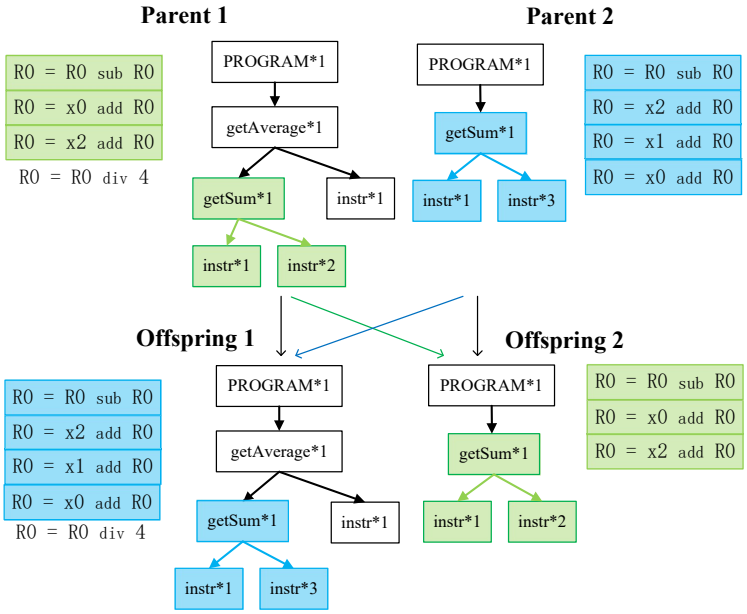

Figure 5.5: An example of the grammar-guided crossover. The blue and green derivation tree nodes and instructions are swapped by the crossover operator.

Second, for each pair of crossover points, the grammar-guided crossover picks a sub-list of derivations from $\mathbf{D}(m_d)$ and $\mathbf{D}(m_r)$ respectively (lines 18-20). If the program size of the offspring is consistent with the predefined minimum and maximum LGP program size, the crossover operator replaces the selected derivation tree nodes and corresponding instructions in the receiver individual with the selected derivation tree nodes and related instructions in the donor individual. Because the input arguments might have different assignments from different roots after swapping, the crossover operator updates the input arguments of the newly replaced sub-derivation tree and instructions accordingly in the receiver individual.

Fig. 5.5 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. 5.5) from the two parent individuals. Based on the derivation tree nodes, the crossover operator selects the related instructions. The crossover operator only selects one crossover point because there are no other non-overlapping nodes. Second, the crossover operator replaces the sub-derivation tree and the instruction list in the receiver and gets offspring. Note that the grammar-guided crossover operator produces two offspring each time by exchanging the role of donor and receiver individuals.

5.3 Evolving Dispatching Rules with Branch Flow Control Operations by G2LGP

To evolve effective and interpretable dispatching rules with branch flow control operations (IF operations particularly), this section proposes three main principles, each for a main challenge mentioned in section 5.1, for incorporating the domain knowledge of IF operations. The three principles are listed below.

1. To address the dimension inconsistency, we first design a set of normalized terminals for DJSS and restrict that IF operations can only compare the proposed normalized terminals with constants. By this means, information in all different dimensions is used in a normalized form, which is easier for humans to understand.

2. To address the inactive sub-rules of IF operations, we design a set of grammar rules to constrain the number and possible positions of IF operations. We restrict dispatching rules to only use IF operations at the beginning of rules with a limited number. This limits the negative impact caused by inactive sub-rules of IF operations.

3. To address the ineffective sub-rules, we coordinate the grammar rules for different parts of a dispatching rule, including IF-included

parts and the rest of it. By designing grammar rules for different parts in a coordinating manner, we improve the effectiveness of flow control operations.

5.3.1 Normalized Terminals

The newly proposed normalized terminal set transforms the existing terminals into a normalized form. Based on the terminal sets of GPHH for solving DJSS problems (see chapter 3), we design twenty normalized terminals (four of them are introduced in sections 5.4.1 and 5.4.2), as shown below. We mainly normalized terminals by their maximum values at corresponding decision situations. For example, we normalize the processing time of the operations on a machine by dividing the processing time of operations over the maximum processing time on the current machine. $||\cdot||_0$ denotes the cardinality of a set or a list (e.g., the number of available operations in the queue of machine m). $||\cdot||_1$ denotes 1-norm regularization of a set or a list (e.g., $||q(m)||_1$ denotes the workload of a machine m, equivalent to $\sum_{o\in q(m)} p(o)$). $\delta(o)$ denotes the waiting time of an operation o, equivalent to the difference between the system time and the ready time of the operation o. $q_{next}(o)$ denotes the corresponding machine queue that processes the next operation of o. $\tau(o_{ji})$ denotes the list of remaining operations in job j after finishing o_{ji}.

To facilitate understanding, we categorize these proposed terminals into three groups: job-related, machine-related, and job shop-related terminals, as shown in table 5.2 to 5.4. These twenty normalized terminals depict various information in making decisions, which is supposed to be comprehensive enough to let IF-included dispatching rules understand decision situations.

Table 5.2: Proposed normalized terminals - Job-related normalized terminals.

Name	Formula	Description
Processing time ratio	PTR$(o_{ji}, m) = \frac{p(o_{ji})}{\max_{o' \in q(m)} p(o')}$	The processing time of an available operation in machine m over the maximum processing time in the current queue.
The number of remaining operations ratio	NORR$(o_{ji}, m) = \frac{\|\|\tau(o_{ji})\|\|_0}{\max_{o' \in q(m)} \|\|\tau(o')\|\|_0}$	The number of remaining operations of job j after processing o_{ji}, divided by the maximum number of remaining operations among all available operations in $q(m)$.
The remaining workload ratio	WKRR$(o_{ji}, m) = \frac{\|\|\tau(o_{ji})\|\|_1}{\max_{o' \in q(m)} \|\|\tau(o')\|\|_1}$	The remaining workload of job j after processing o_{ji}, divided by the maximum remaining workload among all available operations in $q(m)$.
The ratio of the number of operations in the next machine	NNQR$(o_{ji}, m) = \frac{\|\|q_{next}(o_{ji})\|\|_0}{\max_{o' \in q(m)} \|\|q_{next}(o')\|\|_0}$	The number of operations in $q_{next}(o_{ji})$, divided by the maximum number of operations in $q_{next}(o), \forall o \in q(m)$.
The ratio of the workload of the next machine	WNQR$(o_{ji}, m) = \frac{\|\|q_{next}(o_{ji})\|\|_1}{\max_{o' \in q(m)} \|\|q_{next}(o')\|\|_1}$	The total workload of $q_{next}(o_{ji})$, divided by the maximum workload in $q_{next}(o), \forall o \in q(m)$.
The operation waiting time ratio	OWTR$(o_{ji}, m) = \frac{\delta(o_{ji})}{\max_{o' \in q(m)} \delta(o')}$	The waiting time of o_{ji}, divided by the maximum waiting time among all operations in the current queue $q(m)$.
The weight ratio	WR$(o_{ji}, m) = \frac{\omega(o_{ji})}{\max_{\omega(o') \in q(m)} \omega(o')}$	The weight value of o_{ji} divided by the maximum weight value among all operations in $q(m)$
The relative flow due date ratio	rFDR$(o_{ji}, m) = \frac{\alpha_j + \sum_0^i p(o_{ji}) - t}{\max_{o'_{ji} \in q(m)} \alpha_j + \sum_0^i p(o'_{ji}) - t}$	The relative flow due date of o_{ji}, divided by the maximum relative flow due date among all operations in $q(m)$, where t is the system time [137].
The ratio of energy cost rate of a job	JERO$(j) = \frac{r_e(j)}{\max r_e}$	$r_e(j)$ denotes the energy cost rate of a job j. $\max r_e = 3$ in our simulation. Refer to Sections 5.4.1 and 5.4.2.

In our experiment, the data ranges are: PTR$\in [0.01, 1]$, WR$\in [0.25, 1]$, rFDR$\in (-\infty, \infty)$, JERO$\in [0.4, 1]$, NIQR$\in (0, 1]$, WIQR$\in (0, 1]$, DPT$\in [0.01, 1]$, DNPT$\in [0.01, 1]$, MER$\in [0.236, 1]$, BWR$\in (0, 1]$, EPR$\in [0.33, 1]$, and SFR$\in [0.087, 1]$. The data ranges of the other normalized terminals (i.e., NORR, WKRR, NNQR, WNQR, OWTR, DOWT, DNNQ, and DWNQ) are $[0, 1]$.

Table 5.3: Proposed normalized terminals - Machine-related normalized terminals.

Name	Formula	Description
The number of operations in the machine queue ratio	NIQR$(m) = \frac{\|\|q(m)\|\|_0}{\sum_{m' \in \mathbb{M}} \|\|q(m')\|\|_0}$	It indicates the burden of machine m by comparing the number of operations in the machine queue $q(m)$ with the overall number of available operations in the job shop.
The workload in the machine queue ratio	WIQR$(m) = \frac{\|\|q(m)\|\|_1}{\sum_{m' \in \mathbb{M}} \|\|q(m')\|\|_1}$	It indicates the burden of machine m by comparing the total workload of in the machine queue $q(m)$ with the overall workload in the job shop.
Deviation of processing time	DPT$(m) = \frac{\min_{o \in q(m)} p(o)}{\max_{o' \in q(m)} p(o')}$	A simple index to show the processing time discrepancy of the available operations in machine m by comparing minimum processing time with maximum processing time among available operations in $q(m)$.
Deviation of operation waiting time	DOWT$(m) = \frac{\min_{o \in q(m)} \delta(o)}{\max_{o' \in q(m)} \delta(o')}$	A simple index to show the waiting time discrepancy of the available operations in machine m by comparing minimum waiting time with maximum wait time among available operations in $q(m)$.
Deviation of the processing time of the next operation	DNPT$(m) = \frac{\min_{o_{ji} \in q(m)} p(o_{j,i+1})}{\max_{o'_{ji} \in q(m)} p(o'_{j,i+1})}$	A simple index to show the discrepancy of one-step-further decision situations of available operations in $q(m)$ by comparing the minimum processing time of the next operation and the maximum processing time of the next operation.
Deviation of the number of operations in the next machine	DNNQ$(m) = \frac{\min_{o \in q(m)} \|\|q_{next}(o)\|\|_0}{\max_{o' \in q(m)} \|\|q_{next}(o')\|\|_0}$	A simple index to show the discrepancy of one-step-further decision situations of available operations in $q(m)$ by comparing the minimum number of available operations in the next machine with the maximum number of available operations in the next machine.
Deviation of the workload of the next machine	DWNQ$(m) = \frac{\min_{o \in q(m)} \|\|q_{next}(o)\|\|_1}{\max_{o' \in q(m)} \|\|q_{next}(o')\|\|_1}$	A simple index to show the discrepancy of one-step-further decision situations of available operations in $q(m)$ by comparing the minimum workload in the next machine with the maximum workload in the next machine.
The idle energy consumption rate	MER$(m) = \frac{r_m}{\max r_m}$	The normalized energy consumption rate of an idle machine over time. $\max r_m = 7500$ in our simulation. Refer to Sections 5.4.1 and 5.4.2.

In our experiment, the data ranges are: PTR$\in [0.01, 1]$, WR$\in [0.25, 1]$, rFDR$\in (-\infty, \infty)$, JERO$\in [0.4, 1]$, NIQR$\in (0, 1]$, WIQR$\in (0, 1]$, DPT$\in [0.01, 1]$, DNPT$\in [0.01, 1]$, MER$\in [0.236, 1]$, BWR$\in (0, 1]$, EPR$\in [0.33, 1]$, and SFR$\in [0.087, 1]$. The data ranges of the other normalized terminals (i.e., NORR, WKRR, NNQR, WNQR, OWTR, DOWT, DNNQ, and DWNQ) are $[0, 1]$.

Table 5.4: Proposed normalized terminals - Job shop-related normalized terminals.

Name	Formula	Description
Bottleneck workload ratio [158]	BWR= $\max_{m\in\mathbb{M}}$ WIQR(m)	An index of bottleneck by comparing the workload of bottleneck machines with overall workload. Bottleneck machines are the machines with the largest workload at a particular time [158].
Energy price rate	EPR= $\frac{p_e}{\max p_e}$	The normalized job shop-wide energy price. $\max p_e = 0.015$ in our simulation. Refer to Sections 5.4.1 and 5.4.2.
The response cost rate ratio	SFR= $\frac{\varphi}{\max \varphi}$	The normalized job shop-wide cost rate for job response time. $\max \varphi = 2.3$ in our simulation. Refer to Sections 5.4.1 and 5.4.2.

In our experiment, the data ranges are: PTR$\in [0.01, 1]$, WR$\in [0.25, 1]$, rFDR$\in (-\infty, \infty)$, JERO$\in [0.4, 1]$, NIQR$\in (0, 1]$, WIQR$\in (0, 1]$, DPT$\in [0.01, 1]$, DNPT$\in [0.01, 1]$, MER$\in [0.236, 1]$, BWR$\in (0, 1]$, EPR$\in [0.33, 1]$, and SFR$\in [0.087, 1]$. The data ranges of the other normalized terminals (i.e., NORR, WKRR, NNQR, WNQR, OWTR, DOWT, DNNQ, and DWNQ) are $[0, 1]$.

5.3.2 Proposed Grammar Rules

Based on the normalized terminals, we design a set of grammar rules to restrict the use of these normalized terminals in IF operations. To improve the effectiveness and interpretability of IF-included dispatching rules, the proposed grammar rules fulfill three main restrictions of IF operations:

1. Input restriction: IF operations should use normalized terminals and constants as inputs to improve the interpretability of IF conditions.
2. Location restriction: IF branches should locate in the beginning of programs and disjoint with each other (i.e., no nested IF branches) to avoid meaningless program output caused by IF conditions.
3. Number restriction: Dispatching rules should only use a limited number of IF operations to reduce redundant IF branches.

Fig. 5.6 is an example to illustrate the three restrictions. The raw and

	Raw rule	Restricted rule	Comments
Line 0	**R0=R1=R2=0**	**R0=R1=R2=0**	//Initialize registers.
Line 1	**R2= PT + NPT**	**R2= PT + NPT**	
Line 2	**IF>#1 WIQ WINQ**	**IF>#1 WIQR 0.5**	//If the machine is busy
Line 3	**R0= R2 - NPT**	**R1= R2 - NPT**	//Shortest processing time
Line 4	**IF<#1 rFDD R1**	**IF<#1 rFDR 0.1**	//Lines 4-7: If an operation
Line 5	**IF>#2 W 2**	**R1= R2 + rFDD**	delays from a due date,
Line 6	**R1= R2 + rFDD**	**IF>#1 WR 0.6**	prioritize it by "+rFDD".
Line 7	**R0= R1 / W**	**R1= R2 / W**	If a job is important,
Line 8		**R0= R2 + R1**	prioritize it by "/W".

Figure 5.6: Examples of non-restricted and restricted dispatching rules in the LGP representation. The meanings of PT, WIQ, WINQ, rFDD, and W refer to table 3.1.

restricted LGP-based dispatching rules both manipulate three registers, R0, R1, and R2. The final outputs of dispatching rules are stored in the first register R0. Each dispatching rule in Fig. 5.6 contains three IF conditions. "IF$>$ $\#N$ *a b*" denotes that if *a* is larger than *b*, the program executes N subsequent instructions. Otherwise, the program skips the next N instructions.

The two dispatching rules show a similar scheduling pattern. When the workload in a machine queue is heavy (Line 2 for both rules), the two dispatching rules encourage the machine to finish its operations as soon as possible to improve the pipeline level of the job shop, that is, prioritizing operations mainly based on their processing time (i.e., shortest-processing-time-first rule). If an operation is already delayed from its given due date, we prioritize it by adding rFDD (rFDD$<$0 if an operation is delayed). If a job (and its operations) is important (i.e., with a high weight value), we prioritize the operations by dividing them by W. In summary, when the machine is busy, the operation is delayed, and the job is important, the corresponding decision will be most preferred (i.e., smallest dispatching rule value).

However, the two dispatching rules have different interpretability and effectiveness. With the input restriction, the restricted rule uses a normal-

```
defset FUNS {add,sub,mul,div,max,min};
defset FLOWCTRL {IfLarge1,IfLessEq1};
defset rawINPUT
{PT,NPT,WINQ,NINQ,rFDD,rDD,SL,W,OWT,NWT,TIS,WKR,NOR};
defset conditionINPUT {NIQR,WIQR,DPT,DOWT,DNNQ,DWNQ,DNPT,BWR,
PTR,NORR,WKRR,NNQR,WNQR,OWTR,WR,rFDR};
defset INPUT {conditionINPUT,rawINPUT};
defset REG {R0,R1,R2,R3,R4,R5,R6,R7};
defset constant {0.1,0.2,0.3,0.4,0.5,0.6,0.7,0.8,0.9};

begin modulec_0
uncondition(I\O\R) ::= <O\{FUNS}\R+I\R+I>;

branch ::= <{R0}\{FLOWCTRL}\{conditionINPUT}\{constant}>;

condition(I\O\R) ::= <O\{FUNS}\R+I\R+I> :: branch :: <O\{FUNS}\
R+I\R+I>;

PROGRAM ::= condition(I~{INPUT}\O~{REG}\
R~{REG})*5::uncondition(I~{rawINPUT}\O~{REG}\R~{REG})*;
end modulec_0
```

Figure 5.7: The proposed grammar rules for evolving IF-included dispatching rules for DJSS.

ized terminal WIQR to indicate the workload burden of a machine and uses a constant 0.5 to define that the workload is heavy if the workload of this machine accounts for more than half of the overall workload. Contrarily, the raw rule has to compare the features with different physical meanings (i.e., WIQ and WINQ) to approximate the heavy workload (i.e., large enough WIQ), which is not comprehensive enough. With the location restriction, the restricted rule adds one unconditional instruction (i.e., line 8) to assemble register values. But the raw rule overwrites the output register R0 in IF branches, which might lead to meaningless R0 (i.e., returning the initial value of R0) if all the branches are skipped (i.e., all the IF conditions are not satisfied). The nested IF conditions in the raw rule (i.e., lines 4 and 5) also increase the probability that the raw rule skips lines 6 and 7. Although the two rules in Fig. 5.6 use the same number of IF branches, we advocate that unlimited IF conditions easily lead to redundant or contradictory building blocks.

Based on the three restrictions and the domain knowledge of DJSS problems, we design the following grammar rules based on MCFG, as

shown in Fig. 5.7. We first categorize the input features into different concepts (i.e., primitive sets), including arithmetic functions (FUNS), IF operations (FLOWCTRL), raw job shop features (rawINPUT), normalized terminals (conditionINPUT), registers (REG), and constants (constant). We denote the IF operations ("IF> #1" and "IF<= #1") by IfLarge1 and IfLessEq1 respectively. The settings of the primitive set follow chapter 3.

Then, we define the derivation rules to divide LGP programs into subprograms. We divide an LGP program into conditional (i.e., condition (I\O\R)) and non-conditional sub programs (i.e., uncondition(I\O\R)), defined by PROGRAM. The conditional sub-program accepts normalized terminals and raw job shop features as inputs (i.e., I∼{INPUT}) and outputs the results to any of the eight registers (i.e., O∼{REG}). The conditional subprogram includes three instructions, two for arithmetic instructions (i.e., <O\{FUNS}\R+I\R+I>) and one for logical instruction (i.e., branch). To implement the IF-ELSE structure, the conditional subprogram simply first executes an arithmetic instruction unconditionally and then executes the other arithmetic instruction based on the IF condition.

To fulfill the three proposed restrictions, we have three designs in Fig. 5.7. 1) The IF condition (branch) only uses normalized terminals (conditionINPUT) and constants as inputs based on the input restriction. Note that the predefined output register R0 in the IF condition is useless since IF operations do not overwrite registers. 2) Based on the number restriction, the conditional sub program repeats at most five times (i.e., "*5" after condition(I∼{INPUT}\O∼{REG}\R∼{REG}) in PROGRAM) to limit the number of logical operations. 3) The unconditional subprogram is executed after the conditional sub-program to fulfill the location restriction. For the sake of simplicity, the unconditional sub-program only accepts raw job shop features as inputs and outputs the results to any of the eight registers. Note that the design details in Fig. 5.7 are based on

our preliminary investigation and existing studies [21]. For example, the compared method without limiting the number of conditional sub programs averagely has five conditional sub programs. Thus, we limit the conditional sub programs to at most repeating five times in Fig. 5.7 to further reduce the search space.

5.4 Experiment Design

To verify the effectiveness of IF-included dispatching rules and G2LGP, we design three scenario sets with different complexities. Specifically, the first scenario set is the basic one, only optimizing the tardiness or flowtime. The second scenario set increases the complexity of the first scenario set by additionally considering energy cost in the optimization. The third scenario set further increases the complexity by additionally considering energy cost and the response time of operations.

5.4.1 Simulation Design

Basic Scenario Set

The basic scenario set follows the settings in chapter 3. Specifically, the basic scenario set mainly optimizes tardiness or flowtime. the basic scenario set totally contains twelve DJSS scenarios, which are $\langle T_{max}, 0.85\rangle$, $\langle T_{max}, 0.95\rangle$, $\langle T_{mean}, 0.85\rangle$, $\langle T_{mean}, 0.95\rangle$, $\langle WT_{mean}, 0.85\rangle$, $\langle WT_{mean}, 0.95\rangle$, $\langle F_{max}, 0.85\rangle$, $\langle F_{max}, 0.95\rangle$, $\langle F_{mean}, 0.85\rangle$, $\langle F_{mean}, 0.95\rangle$, $\langle WF_{mean}, 0.85\rangle$, and $\langle WF_{mean}, 0.95\rangle$.

Second Scenario Set

To increase the complexity of the problems, we introduce energy cost into optimization objectives of the second scenario set. The energy cost model and the energy price follow [119]. Specifically, each job in the second sce-

nario set has an energy cost rate $r_e \sim U(1.2, 3)$. The machines consume energy in both idle and working time. Each machine has an idle energy consumption rate r_m and a working energy consumption rate $r_m \times r_e$ (r_e is the energy cost rate of on-going jobs). The settings of r_m follow [119] (i.e., machines have different energy consumption rates). To simulate the floating power prices in daily life, the energy price rate p_e in our simulation changes every 10 arrival jobs. The energy price is sampled from three values 0.005, 0.01, and 0.015, based on a uniform distribution. The average energy consumption per job per machine $\overline{E}$ is obtained as follows.

$$\overline{E} = \frac{\sum_{m \in \mathbb{M}} E(m)}{|\mathbb{M}| \times |\mathbb{J}|}$$

$$E(m) = \sum_{t} (\tau_{idle}(t) \times r_m \times p_e(t))$$

$$+ \sum_{t} \sum_{j \in \Theta(m)} \tau_{work}(t) \times r_m \times r_e(j) \times p_e(t)$$

where $E(m)$ is the total energy consumption of machine m. $\tau_{idle}(t)$ and $\tau_{work}(t)$ are the idle and working running time for a machine in a time period t respectively. $\Theta(m)$ is all the processed jobs by machine m.

Based on $\overline{E}$, we extend the six tardiness and flowtime optimization objectives by simply averaging $\overline{E}$ and tardiness and flowtime objective values. For example, T_{max} is transformed to $T^E_{max} = 0.5T_{max} + 0.5\overline{E}$, and T_{mean} is transformed to $T^E_{mean} = 0.5T_{mean} + 0.5\overline{E}$, etc. Note that the two objective values in the linear combination have a similar magnitude based on our preliminary investigation. Together with the two utilization level settings, the second scenario set also has twelve scenarios.

Third Scenario Set

The third scenario set further increases the complexity of DJSS problems by additionally considering job response time in the second scenario. The design of the job response time in this thesis borrows the idea from

[118, 162]. Job response time is an important performance metric in many controlling systems, such as real-time operating systems on computers. To optimize the job response time, we define a response cost R by multiplying the job response time R_t with a response cost rate φ. The response time in this thesis is equivalent to the difference between the start-to-be-processed time and the arrival time of a particular job. There are three levels of φ, 0.2, 1, and 2.3. We reset φ values among the three values every 10 arrival jobs based on a probability of 40%:40%:20%. The average response cost $\overline{R}$ is obtained by

$$\overline{R} = \frac{\sum_{j \in \mathbb{J}} R(j)}{|\mathbb{J}|}$$

$$R(j) = \sum_{t} \sum_{o \in \mathcal{O}_j} R_t(o) \times \varphi(t)$$

where $R(j)$ is the response cost of job j, $R_t(o)$ is the response time for operations, and $\varphi(t)$ is the response cost rate during the waiting time of the operation o. We integrate the response cost R with tardiness, flowtime, and energy cost by the same linear combination to develop six optimization objectives T^{ER}_{max}, T^{ER}_{mean}, WT^{ER}_{mean}, F^{ER}_{max}, F^{ER}_{mean}, and WF^{ER}_{mean}. For example, $T^{ER}_{max} = 0.4T_{max} + 0.3\overline{E} + 0.3\overline{R}$, and $T^{ER}_{mean} = 0.4T_{mean} + 0.3\overline{E} + 0.3\overline{R}$. The three objective values here also have similar magnitudes.

5.4.2 Comparison Design

We use five compared methods to verify the effectiveness of the proposed grammar rules. 1) The first algorithm is the grammar-guided LGPHH without IF operations. It applies grammar rules to constrain the search space in chapter 3. This compared method has shown promising performance for solving DJSS problems in our prior investigation, denoted as G2LGP. 2) The second compared algorithm extends the basic LGPHH by including IF operations and the proposed normalized terminals into its primitive set directly but without grammar-guided evolutionary framework and the proposed grammar rules, denoted as LGP+. 3) and 4)

The third and fourth compared methods are variants of the proposed method which extend the proposed grammar rules by removing some restrictions from Fig. 5.7. Specifically, the third compared method (denoted as G2LGP/input) removes the input restriction and evolves based on a set of grammar rules shown in Fig. 5.8. The source registers of IF operations can be any of the terminals, including all the input features, registers, and constants. The fourth compared method (denoted as G2LGP/locnum) removes the location restriction and the number restriction and evolves based on a set of grammar rules shown in Fig. 5.9. The "$*$" after the `PROGRAM` derivation indicates that there can be any number of conditional sub-programs (`condition`). The `condition` sub-program can derive to either one logical instruction (`branch`) and one arithmetic instruction (`<O\{FUNS}\R+I\R+I>`), or unconditional sub-programs (`uncondition(I∼I\O∼O\R∼R)`) whose input arguments are the same as the parent of the derivation. Thus, there can be a large number of IF operations in a program, and the IF operations can also be used at any position in a program. The last compared method is the proposed algorithm, which evolves G2LGP based on the normalized terminals and the proposed grammar rules, denoted as G2LGP-IF. The primitive set of all the compared methods includes the raw input features, which are designed based on chapter 3. Except for G2LGP, the rest of the compared methods also include the proposed normalized terminals in their primitive sets. The function set for G2LGP is $\{+, -, \times, \div, \max, \min\}$, and the function set for the other four compared methods is $\{+, -, \times, \div, \max, \min, \text{IF} > \#1, \text{IF} <= \#1\}$. The rest of the settings for the compared methods are set as those in chapter 3.

```
... /* other rules are the same as the proposed rules */

branch ::= <{RO}\{FLOWCTRL}\{conditionINPUT}\{constant}>;
branch ::= <{RO}\{FLOWCTRL}\{INPUT,REG,constant}\{INPUT,REG,constant}>;

... /* other rules are the same as the proposed rules */
```

Figure 5.8: The grammar rules of G2LGP/input.

```
... /* other rules are the same as the proposed rules */

condition(I\O\R) ::= <O\{FUNS}\R+I\R+I> :: branch :: <O\{FUNS}\R+I\R+I>;
condition(I\O\R) ::= branch :: <O\{FUNS}\R+I\R+I> |
uncondition(I~I\O~O\R~R);

PROGRAM ::= condition(I~{INPUT}\O~{REG}\
R~{REG})*5::uncondition(I~{rawINPUT}\O~{REG}\R~{REG})*;
PROGRAM ::= condition(I~{rawINPUT}\O~{REG}\R~{REG})*;
end modulec_0
```

Figure 5.9: The grammar rules of G2LGP/locnum.

Table 5.5: Average test objective values (std.) in the basic scenario set.

Scenarios	G2LGP	LGP+	G2LGP/input	G2LGP/locnum	G2LGP-IF
⟨Tmax,0.85⟩	**1922.1 (42.9)** ≈	1978.1 (162.1) −	1927.5 (52) ≈	1939.2 (51.4) ≈	1931 (44.6)
⟨Tmax,0.95⟩	**3943.1 (84)** ≈	4040.5 (218.9) −	3946.7 (79.5) ≈	4045.6 (123.3) −	3968.8 (112.3)
⟨Tmean,0.85⟩	417.7 (2.6) ≈	428.8 (40) ≈	416.9 (2.2) ≈	418 (2.8) −	**416.8 (2.9)**
⟨Tmean,0.95⟩	1116.7 (8.7) ≈	1195.6 (167.4) ≈	1116.4 (11.2) ≈	1122.2 (12.1) ≈	**1116.3 (12.1)**
⟨WTmean,0.85⟩	723.6 (7.5) ≈	770.4 (112.7) ≈	**723.1 (5.4)** ≈	728.4 (7.3) ≈	726.5 (7.3)
⟨WTmean,0.95⟩	1724.4 (26.6) ≈	2103.9 (1739.3) −	**1722.1 (25.5)** ≈	1740.2 (27.7) ≈	1733 (33.9)
⟨Fmax,0.85⟩	2534.6 (74.1) −	2529.2 (63.8) −	**2490 (65.2)** ≈	2535.7 (53.1) −	2503.1 (79.7)
⟨Fmax,0.95⟩	4599.7 (80.6) −	4760.9 (623.5) −	4505.6 (73.4) ≈	4638.4 (120) −	**4501 (74.8)**
⟨Fmean,0.85⟩	864.6 (3.2) ≈	910 (193.4) ≈	**862.4 (2.6)** ≈	865.1 (3.5) −	863.7 (2.6)
⟨Fmean,0.95⟩	1565.3 (10.9) ≈	1649 (245.8) ≈	**1561.7 (9.3)** ≈	1571.3 (16.5) ≈	1565.9 (12.6)
⟨WFmean,0.85⟩	1701.7 (6.1) ≈	1826.4 (661) ≈	**1701.4 (6.5)** ≈	1706.5 (7) ≈	1703.4 (7.5)
⟨WFmean,0.95⟩	2722.8 (25.4) ≈	3610.6 (3943.9) −	**2708 (24.7)** ≈	2724.1 (24.5) −	2711.7 (21.5)
win/draw/lose	0-10-2	0-6-6	0-12-0	0-6-6	
mean rank	2.46	4.25	**1.5**	4.58	2.21
p-values	1.000	0.015	1.000	0.002	

Table 5.6: Average test objective values (std.) in the second scenario set.

Scenarios	G2LGP	LGP+	G2LGP/input	G2LGP/locnum	G2LGP-IF
⟨TmaxE,0.85⟩	2093.8 (24.1) −	2106.1 (38) −	2082.9 (26.5) ≈	2096 (25.4) −	**2078.4 (23.1)**
⟨TmaxE,0.95⟩	3207.8 (86.1) −	3231.4 (302.8) −	3153.2 (63.5) ≈	3233 (128.6) −	**3152.8 (65.9)**
⟨TmeanE,0.85⟩	1331.1 (2.2) ≈	1342.7 (44.8) ≈	**1330 (1.3)** ≈	1330.6 (1.8) ≈	1330.4 (1.5)
⟨TmeanE,0.95⟩	**1651 (6.4)** ≈	1676.6 (67.5) ≈	1652.9 (10.6) ≈	1652.9 (7.9) ≈	1651.7 (9.6)
⟨WTmeanE,0.85⟩	1487.2 (3.6) ≈	1526.4 (82.3) −	**1485.6 (3.1)** +	1486.8 (3.6) ≈	1487.6 (3.7)
⟨WTmeanE,0.95⟩	1985.8 (17.3) ≈	2196 (828) −	**1982 (15.3)** ≈	1993.7 (17.1) ≈	1987.2 (15.2)
⟨FmaxE,0.85⟩	2385.3 (26.2) −	2513.3 (895.8) −	2378.6 (53.8) ≈	2397.7 (45.5) −	**2369.7 (23.1)**
⟨FmaxE,0.95⟩	3465.3 (54.9) ≈	3474.5 (77) −	**3431.8 (42.4)** ≈	3494.2 (58.4) −	3443.7 (56.3)
⟨FmeanE,0.85⟩	1554.1 (2.2) −	1599 (198.4) −	1553.4 (1.9) ≈	1553.9 (1.8) −	**1553 (1.4)**
⟨FmeanE,0.95⟩	1875.6 (5.9) ≈	1913.2 (95.7) ≈	**1873.4 (5.9)** ≈	1878.7 (9.1) ≈	1876.3 (5.7)
⟨WFmeanE,0.85⟩	1975.6 (3.5) ≈	1999.8 (65.3) ≈	**1974.4 (3.7)** ≈	1977.7 (4) −	1975.2 (4.1)
⟨WFmeanE,0.95⟩	2484.1 (17.4) ≈	2592.8 (436.1) ≈	2483.1 (17.6) ≈	**2480.4 (14.8)** ≈	2481.1 (14.7)
win/draw/lose	0-8-4	0-5-7	1-11-0	0-6-6	
mean rank	2.92	4.38	**1.58**	4.13	2.00
p-values	1.000	0.002	1.000	0.010	

Table 5.7: Average test objective values (std.) in the third scenario set.

Scenarios	G2LGP	LGP+	G2LGP/input	G2LGP/locnum	G2LGP-IF
⟨TmaxER,0.85⟩	1643.9 (26.3) −	1640.2 (36.7) −	**1621.5 (18)** ≈	1637.3 (25.4) −	1624.9 (20)
⟨TmaxER,0.95⟩	2807.6 (52.5) −	2838 (112.7) −	**2790.4 (61.5)** ≈	2833.9 (66.9) −	2794.5 (142.5)
⟨TmeanER,0.85⟩	995 (2.4) −	1007.9 (35.1) ≈	992.9 (2.6) ≈	**990.1 (4.4)** +	992.7 (3.4)
⟨TmeanER,0.95⟩	1457.5 (9.1) −	1542.5 (235.7) −	1444.7 (25.4) −	**1405.3 (21.1)** +	1424 (29.2)
⟨WTmeanER,0.85⟩	1132.2 (9.6) ≈	1174.1 (81.5) −	**1128.6 (3.4)** ≈	1131.9 (4.6) ≈	1130.4 (4.7)
⟨WTmeanER,0.95⟩	1774.5 (21.6) ≈	2003.5 (806) −	1767.9 (18.9) ≈	**1754.2 (29.4)** ≈	1761.2 (23.8)
⟨FmaxER,0.85⟩	1882.5 (28.5) −	1889.2 (31) −	**1858.5 (18.9)** ≈	1882.3 (61) ≈	1866.2 (22.8)
⟨FmaxER,0.95⟩	3053.9 (49) −	3293 (1052.1) −	3014.4 (46.1) ≈	3054.2 (59.5) −	**3000.6 (47.5)**
⟨FmeanER,0.85⟩	1173.2 (2.6) −	1206.7 (72.7) −	1171.4 (2.9) ≈	**1167.3 (4.7)** +	1170.4 (3.7)
⟨FmeanER,0.95⟩	1636.2 (9.7) −	1665.1 (95.8) −	1625.7 (20.5) −	**1583.9 (22.3)** +	1602.8 (29.8)
⟨WFmeanER,0.85⟩	1523 (3.8) −	1566.3 (82.5) −	**1520.6 (3.6)** ≈	1522.8 (4.5) −	1521 (3.8)
⟨WFmeanER,0.95⟩	2174 (17.9) −	2276.6 (342) −	2162.6 (14.6) ≈	**2142.5 (29.6)** +	2154.5 (22.3)
win/draw/lose	0-2-10	0-1-11	0-10-2	5-3-4	
mean rank	4.25	4.42	2.17	2.33	**1.83**
p-values	0.002	0.000	1.000	1.000	

5.5 Experiment Results

5.5.1 Test Performance

This section compares the test performance of the five compared methods on the three scenario sets, as shown in Table 5.5 to 5.7. The test per-

formance indicates the actual tardiness and flowtime (by time units) of the compared methods for solving unseen instances in different scenarios. For each scenario set, we first analyze the overall performance of the compared methods by the Friedman's test and then analyze the performance on each scenario based on the Wilcoxon rank-sum test with Bonferroni correction. The significance levels of the Friedman's test and the Wilcoxon test are 0.05. Specifically, the "+" in Table 5.5 to 5.7 indicates that a certain method is significantly better (i.e., having a smaller objective value) than the proposed G2LGP-IF, the "$\approx$" indicates that a method is statistically similar to G2LGP-IF, and the "$-$" indicates that a method is significantly worse than G2LGP-IF. The p-values at the last row indicate the pair-wise comparison between a certain method and G2LGP-IF, with a null hypothesis that the performances of the two compared methods belong to the same distribution and an alternative hypothesis of different distributions.

For the basic scenario set, the p-value of the Friedman's test is 4.25E-07, which indicates a significant difference among the compared methods. Based on the pair-wise comparison and the mean rank, we confirm that the proposed G2LGP-IF has a significantly better overall test performance than directly evolving IF-included dispatching rules by basic LGPHH (i.e., LGP+) and G2LGP without location and number restrictions. On the other hand, we cannot see significant performance differences between state-of-the-art LGP (i.e., G2LGP) whose primitive set has been well designed and G2LGP-IF, and between G2LGP/input and G2LGP-IF. It is likely that the existing primitive set (i.e., excluding the IF operations and normalized terminals) is large enough to compose effective dispatching rules for the basic scenario set, and the grammar of G2LGP-IF and G2LGP/input effectively reduces the search space to a similar size with G2LGP. The Wilcoxon test confirms our observations on the inferior performance of LGP+ and G2LGP/locnum.

The second scenario set which concerns energy cost shows a similar pattern to the basic scenario set, with a Friedman's test p-value of

5.53E-06. In the second scenario, G2LGP-IF is also superior to LGP+ and G2LGP/locnum and performs similarly to G2LGP and G2LGP/input. It is worth mentioning that, despite the insignificant overall performance discrepancy between G2LGP and G2LGP-IF, G2LGP-IF performs significantly better than G2LGP on four scenarios and has better mean performance on eight scenarios. Together with the results in Table 5.5, the results confirm that G2LGP-IF has a very competitive performance with the state-of-the-art LGPHH methods and is superior to basic LGPHH when solving relatively simple scenarios.

The third scenario set which considers tardiness (or flowtime), energy cost, and job response time, shows a substantial difference. The p-value of the Friedman's test is 7.34E-06, indicating a significant difference among the compared methods. Based on the pair-wise comparison, we see that G2LGP-IF significantly outperforms G2LGP and LGP+ in terms of test performance. The Wilcoxon test further verify the superior performance of the proposed G2LGP-IF. On the other hand, the other two G2LGP variants G2LGP/input and G2LGP/locnum have a very competitive performance with G2LGP-IF, but with worse mean ranks (i.e., 2.17 for G2LGP/input and 2.33 for G2LGP/locnum are worse than 1.83 for G2LGP-IF). The results confirm that G2LGP with the proposed grammar restrictions is very effective in solving the complex scenario set which simultaneously optimizes multiple performance metrics.

Based on the results from the basic scenario set to the more complicated ones, we have the following observations:

1) When scenario sets become more and more complicated, evolving IF operations by grammar rules becomes more and more important. The evidences are twofold. First, the performance gap between non-grammar-guided LGP and grammar-guided LGP becomes larger and larger when the scenarios have to optimize more and more performance metrics (e.g., the mean rank of LGP+ increases with scenario complexity). Second, G2LGP, which has no IF primitives and necessary grammar rules, can-

not handle complex scenarios effectively. It performs inferior to G2LGP-IF on more scenarios when scenario sets become more complex (i.e., from 2 significantly worse scenarios to 10 worse scenarios).

2) Directly evolving IF-included dispatching rules is too difficult for existing LGPHH methods since IF operations introduce a large number of redundant solutions into their search spaces. For example, LGP might waste a lot of time searching the contradictory and tautological IF operations that do not contribute to the final output. As shown in Table 5.5 to 5.7, LGP+ which directly includes IF operations in its primitive set always has the worst performance among the compared methods. However, without IF operations, it is hard for existing LGPHH methods to solve complicated scenarios.

3) Tables 5.5 to 5.7 give some insights into the number and location of IF operations. For example, limiting the maximum number of IF operations to five (i.e., G2LGP-IF and G2LGP/input) is effective since G2LGP-IF and G2LGP/input show a good performance in all the three scenario sets. Setting the number of IF operations too small (i.e., G2LGP and LGP+) or too large (i.e., G2LGP/locnum) likely reduces the effectiveness in complex scenarios.

To further analyze the test performance of the compared methods, we show the average test performance of all the compared methods over generations, as shown in Fig. 5.10. Specifically, we select $Tmax$, $WTmean$, $Fmax$, and $Fmean$ with a high utilization level of 0.95 in the three scenario sets as the example scenarios.

We can see that the proposed G2LGP-IF (i.e., the red curves) shows a very competitive performance with other compared methods. On the contrary, basic LGPHH cannot find stable IF-included dispatching rules (i.e., LGP+, the green curves). In some certain generations, the test performance of LGP+ soars up to an extremely poor level, implying that basic LGPHH fails to evolve IF-included dispatching rules with a good generalization ability.

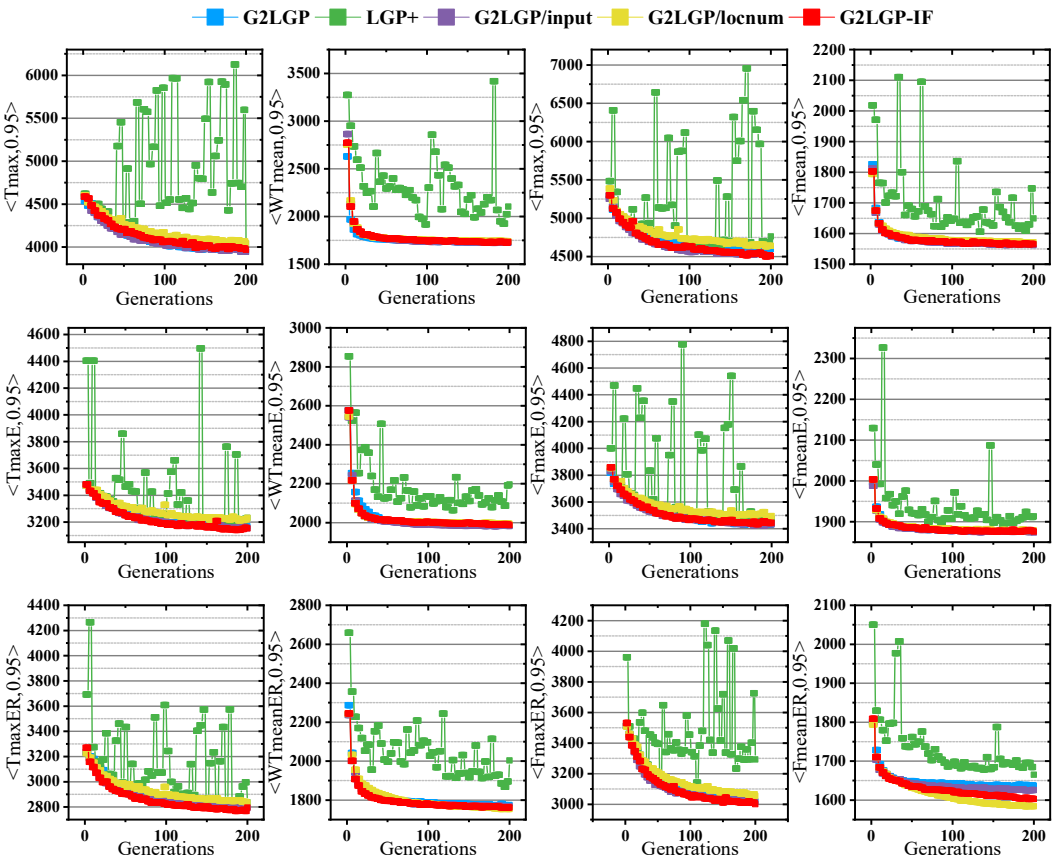

Figure 5.10: The test performance over generations in example scenarios.

5.5.2 Program Size

To have a brief understanding of the interpretability of output rules, Fig. 5.11 shows the average effective program size (i.e., the average number of effective instructions) of best-of-run individuals for each compared method for solving six example scenarios over 50 independent runs. We simply assume that a concise rule (i.e., a smaller program) has good interpretability. The curves of mean values and the shadows of standard deviation show that the proposed G2LGP-IF and G2LGP/input averagely

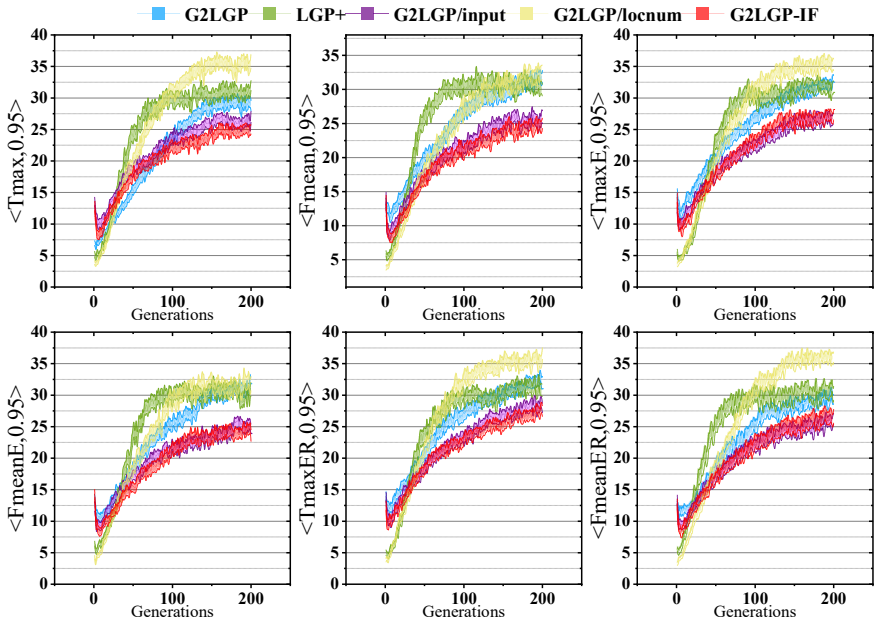

Figure 5.11: The average effective program size (± std.) of best-of-run individuals of the compared methods over generations and 50 independent runs.

achieves significantly smaller program size than the others. For example, in the first example scenario ⟨Tmax, 0.95⟩, the red and purple curves nearly have no overlap with the others at the end of their evolution, indicating a significant program size difference of output programs. Given that both G2LGP-IF and G2LGP/input have restrictions on the number and locations of IF operations by grammar rules, we believe the two proposed restrictions are essential reasons for producing concise programs.

5.5.3 Dimension Consistency

This section analyzes the example dispatching rules of G2LGP-IF and G2LGP/input to demonstrate the dimension consistency achieved by the proposed normalized terminals and input restriction. Specifically, we ran-

domly select two best rules of G2LGP-IF and G2LGP/input from 50 independent runs respectively, for solving ⟨Tmean, 0.95⟩ and ⟨WFmeanER, 0.95⟩. Each pair of the selected rules for the same scenario has a similar test performance. All the four rules have been manually simplified by replacing registers with intermediate results and removing contradictory and tautological IF operations.

⟨**Tmean, 0.95**⟩

The example rule from G2LGP-IF for ⟨Tmean, 0.95⟩ is shown in Eq. (5.1). We can see that when the bottleneck situation is not too severe (i.e., $BWR \leq 0.9$), the dispatching rule prioritizes the operations with a large processing time of the next operation (i.e., "$-NPT$" in a). It implies that the rule intends to use a long-term strategy to process some tough operations before the bottleneck comes. When there is a severe bottleneck in the job shop, the dispatching rule prefers operations with a small processing time to finish more operations in a shorter time (i.e., both "PT" in the main rule and "NPT" in the conditional part are positively correlated to the heuristic value). By this means, the job shop improves the pipeline of the job shop as soon as possible.

$$\begin{aligned} RULE_{IF} &= \max(PT, \frac{5a + 4NOR}{PT}) + (5a + 4NOR) \times PT \\ a &= \begin{cases} WINQ - NPT + PT, & if\ BWR \leq 0.9 \\ NPT \times (NPT + NINQ), & otherwise \end{cases} \end{aligned} \tag{5.1}$$

The example rule from G2LGP/input for ⟨Tmean, 0.95⟩ is shown in Eq. (5.2). The main rule is relatively simple, adding three simple terms together. However, we can see dimension inconsistency in its conditional part a. The first condition $rFDR \leq WKRR\ and\ WINQ \leq 0.4$ compares the terminals with different physical meanings (i.e., $rFDR$ and $WKRR$) and compares the workload in the next machine queue ($WINQ$) with a meaningless constant 0.4. Note that $WINQ$ is larger than 2 (the minimum

workload unit for a single operation is 2) in most cases and is equivalent to 0 only at the beginning of the simulation where most machines are idle. Thus, the second condition is almost a tautological condition, and the third branch of a can be neglected in many cases.

$$
\begin{aligned}
RULE_{input} &= NINQ + 2PT + \min(a, NPT) \\
a &= \begin{cases} NOR, & if\ rFDR \leq WKRR \\ & and\ WINQ \leq 0.4 \\ NINQ^3, & else\ if\ WINQ > 0.4 \\ rFDD, & otherwise \end{cases}
\end{aligned} \tag{5.2}
$$

⟨**WFmeanER, 0.95**⟩

When the problem considers three performance metrics, the example rule from G2LGP-IF still maintains a good interpretability. The rules from G2LGP-IF and G2LGP/input are shown in Eq. (5.3) and Eq. (5.4) respectively. For the rule from G2LGP-IF, we can see that if the user response time is important (i.e., $SFR > 0.2$ implying $SFR = 1$ or $SFR = 2.3$) or the number of operations in the next queue is relatively large ($NNQR > 0.3$), the LGP rule emphasizes the next processing time NPT. Since a small NPT also implies that the next operation will be prioritized when it is available (i.e., replacing PT by NPT in Eq. (5.3)), the job shop prefers finishing those easy-to-process jobs so that it can response more jobs to reduce the overall response time. Otherwise, the job shop focuses on WKR to reduce the overall flowtime.

$$
\begin{aligned}
RULE_{IF} &= ((4PT + a) * a)^2 \\
a &= \frac{b + PT + NINQ}{W} \\
b &= \begin{cases} NPT, & if\ SFR > 0.2\ or\ NNQR > 0.3 \\ WKR, & otherwise \end{cases}
\end{aligned} \tag{5.3}
$$

Contrarily, the example rule from G2LGP/input compares terminals

with different physical meanings in its IF operation. The conditional part c compares the deviation of processing time (DPT) with the number of remaining operations NOR and compares the number of operations in the next machine queue ($NINQ$) with the processing time (PT). These comparisons make the decisions from Eq. (5.4) hard to be interpreted and might lead to unexpected behaviors during optimization.

$$\begin{aligned} RULE_{input} = & \frac{W \cdot PT \cdot a \cdot TIS}{b} + \\ & \frac{W \cdot PT \cdot b \cdot PT + WINQ \cdot NPT \cdot PT}{W^2} + \\ & \frac{2b}{W \cdot PT} + a + c + TIS + PT + SL \\ a = & \, c + NOR \\ b = & \, c + PT \\ c = & \begin{cases} \max(NINQ, PT), & if\ DPT > NOR \\ PT, & otherwise \end{cases} \end{aligned} \tag{5.4}$$

5.5.4 Training Time

Our experiments run on Intel Broadwell (E5-2695v4, 2.1 GHz). With the same total number of fitness evaluations (i.e., 51200), the average training time of the five compared methods are 3.26, 5.39, 8.08, 9.06, and 9.18 hours respectively. They show a pattern of G2LGP $<$ LGP+ $<$ G2LGP/input $\approx$ G2LGP/locnum $\approx$ G2LGP-IF in terms of training time. G2LGP has the shortest training time since it uses neither the normalized terminals nor IF operations. Although LGP+ includes the normalized terminals and IF operations in its primitive set, there are no grammar rules to enforce each LGP rule to use these primitives. Contrarily, the last three compared methods that use grammar to enforce the use of the normalized terminals and IF operations, increase the computation time of each rule. The results show that the use of the normalized terminals and IF operations increases the computation time of dispatching rules.

However, we advocate that the increase in the computation time of dispatching rules here is acceptable in practice since the training of GP methods is off-line and the decision time (i.e., the computation time of all candidate operations) of LGP rules is much smaller than the processing time in reality. Take the longest training time of G2LGP-IF (i.e., 9.18 hours) as an example. The 9.18 hours are composed of $256 \times 200 = 51200$ fitness evaluations, each evaluation processes more than 6000 jobs, and each job with 6 operations on average. Thus, the average decision time for each operation is about $\frac{9.18hours}{51200 \times 6000 \times 6} = 1.8E^{-5}$ seconds, which is much smaller than the operation processing time in many real-world applications. Once we obtain a dispatching rule from the off-line training, the rule can make decisions for unseen problem instances in a very short time. Furthermore, the normalized terminals and IF operation bring a significant performance gain in many DJSS scenarios.

5.5.5 Summary on Main Results

This section investigates the effectiveness and interpretability of the five compared methods. We found that existing LGPHH methods cannot effectively evolve rules with IF operations based on the inferior performance of LGP+. But without IF operations, state-of-the-art LGPHH cannot effectively solve complicated DJSS scenarios (see the inferior performance of G2LGP in the third scenario set). To make effective use of IF operations, we proposed to restrict dispatching rules to only use a limited number of IF operations at the beginning of the rules, which substantially improves the effectiveness of LGPHH in solving complicated scenarios. Moreover, the proposed normalized terminals and input restriction improve the dimension consistency of IF operations in output rules, which can further improve interpretability. This suggests that evolving dispatching rules with IF operations by restricting the input features, number, and location of IF operations has a promising performance in terms of both effective-

```
... /* other rules are the same as the proposed rules */

defset FLOWCTRL {IfLarge3,IfLessEq3};
...
condition(I\O\R) ::= <O\{FUNS}\R+I\R+I> :: branch :: <O\{FUNS}\R+I\R+I>
:: <O\{FUNS}\R+I\R+I> :: <O\{FUNS}\R+I\R+I>;

... /* other rules are the same as the proposed rules */
```

Figure 5.12: The grammar rules of G2LGP-body3.

ness and interoperability.

5.6 Further Analyses

5.6.1 Effectiveness of Simple IF Operations

LGP can naturally implement human-like programming styles of IF operations, including IF branches with different lengths and nested IF branches. In our proposed grammar rules, we restrict IF branches to only contain one instruction and avoid nested IF branches for simplicity, but have not verified the effectiveness of these simple designs. Therefore, this section investigates the effectiveness of long IF branches and nested IF branches.

The first variant of G2LGP, denoted as G2LGP-body3, forces IF branches to include three instructions. The grammar of G2LGP-body3 is shown in Fig. 5.12, where dispatching rules use IF operations with three instructions (i.e., `IfLarge3` and `IfLessEq3`), and the derivation rule of `condition` includes more instruction modules.

The second G2LGP variant is G2LGP-nested, which allows an IF branch to include other IF branches (i.e., nested IF branches). The grammar rules for G2LGP-nested are shown in Fig. 5.13, where dispatching rules use IF operations with one to three instructions, and each logical condition is followed by up to three instruction modules (see `condition` in Fig. 5.13).

We verify the effectiveness of these two G2LGP variants by comparing

```
... /* other rules are the same as the proposed rules */

defset FLOWCTRL {IfLarge1,IfLessEq1,IfLarge2,IfLessEq2,
IfLarge3,IfLessEq3};
...
condition(I\O\R) ::= <O\{FUNS}\R+I\R+I> :: branch :: <O\{FUNS}\R+I\
R+I>*3;

... /* other rules are the same as the proposed rules */
```

Figure 5.13: The grammar rules of G2LGP-nested.

Table 5.8: The test performance of the G2LGP variants in the second scenario set.

Scenarios	G2LGP-IF	G2LGP-body3	G2LGP-nested
⟨TmaxE,0.85⟩	2078.4 (23.1)	2085.8 (22.9) ≈	2083.7 (23.9) ≈
⟨TmaxE,0.95⟩	3152.8 (65.9)	3170.7 (61.5) ≈	3164 (56.9) ≈
⟨TmeanE,0.85⟩	1330.4 (1.5)	1330.4 (1.4) ≈	1330.1 (1.7) ≈
⟨TmeanE,0.95⟩	1651.7 (9.6)	1651.8 (6.6) ≈	1650.4 (5.3) ≈
⟨WTmeanE,0.85⟩	1487.6 (3.7)	1487.6 (3.6) ≈	1487.6 (4.4) ≈
⟨WTmeanE,0.95⟩	1987.2 (15.2)	1988.8 (16.4) ≈	1989.6 (14.1) ≈
⟨FmaxE,0.85⟩	2369.7 (23.1)	2374.3 (19.9) ≈	2369.5 (18.5) ≈
⟨FmaxE,0.95⟩	3443.7 (56.3)	3440.8 (55.2) ≈	3454.4 (69.1) ≈
⟨FmeanE,0.85⟩	1553 (1.4)	1552.9 (1.5) ≈	1553.2 (1.5) ≈
⟨FmeanE,0.95⟩	1876.3 (5.7)	1874.9 (6.2) ≈	1876.2 (6.8) ≈
⟨WFmeanE,0.85⟩	1975.2 (4.1)	1976.2 (4.3) ≈	1976.7 (3.9) ≈
⟨WFmeanE,0.95⟩	2481.1 (14.7)	2483.8 (21.5) ≈	2486 (21.3) ≈
win/draw/lose		0-12-0	0-12-0
mean rank	1.71	2.08	2.21
p-values		1	0.633

their test performance with G2LGP-IF in the second scenario set, as shown in Table 5.8. We use the second scenario set for verification because its configurations follow the popularly used settings of existing studies [119], which makes it comparable to existing studies. In addition, it is more challenging than the basic scenario set, which gives a better understanding of

the proposed algorithm for readers. The Friedman test on the test performance of the three compared methods returns a p-value of 0.428, indicating a null hypothesis of no significant difference among these test performances. We can also see that both G2LGP-body3 and G2LGP-nested are very competitive with G2LGP-IF in Table 5.8. The results confirm that the proposed grammar rules for IF operations are effective enough to produce concise and effective rules. Increasing the complexity of IF branches by increasing the length of IF branches or nesting IF branches does not improve the effectiveness of IF-included dispatching rules in our problem.

5.6.2 Patterns of Normalized Terminals

This section moves a step forward in effectively evolving IF-included dispatching rules for DJSS problems. To understand the relationship between IF operations and decision situations and inspire the future design of IF-included dispatching rules, this section investigates the distributions of input features of IF operations in the produced rules. Since we restrict that G2LGP-IF only uses the proposed normalized terminals as the input features of IF operations, we mainly analyze the frequency of normalized terminals in the best rules over 50 independent runs, as shown in Fig. 5.14. The frequently used normalized terminals imply crucial information in different decision situations. We select four scenarios respectively from the three scenario sets.

In the basic scenario set (Fig. 5.14-(a)), all the four example scenarios switch behaviors based on long-term information, such as the remaining operations and workload of a job (i.e., NORR and WKRR) and the machine that processes the next operation (i.e., NNQR and WNQR). Besides, the maximum objectives like Tmax and Fmax, clearly consider processing time (i.e., PTR and DPT) more than the mean objectives when switching behaviors. PTR and DPT are frequently used in their logical operations. To reduce the maximum values, maximum objectives also switch behaviors

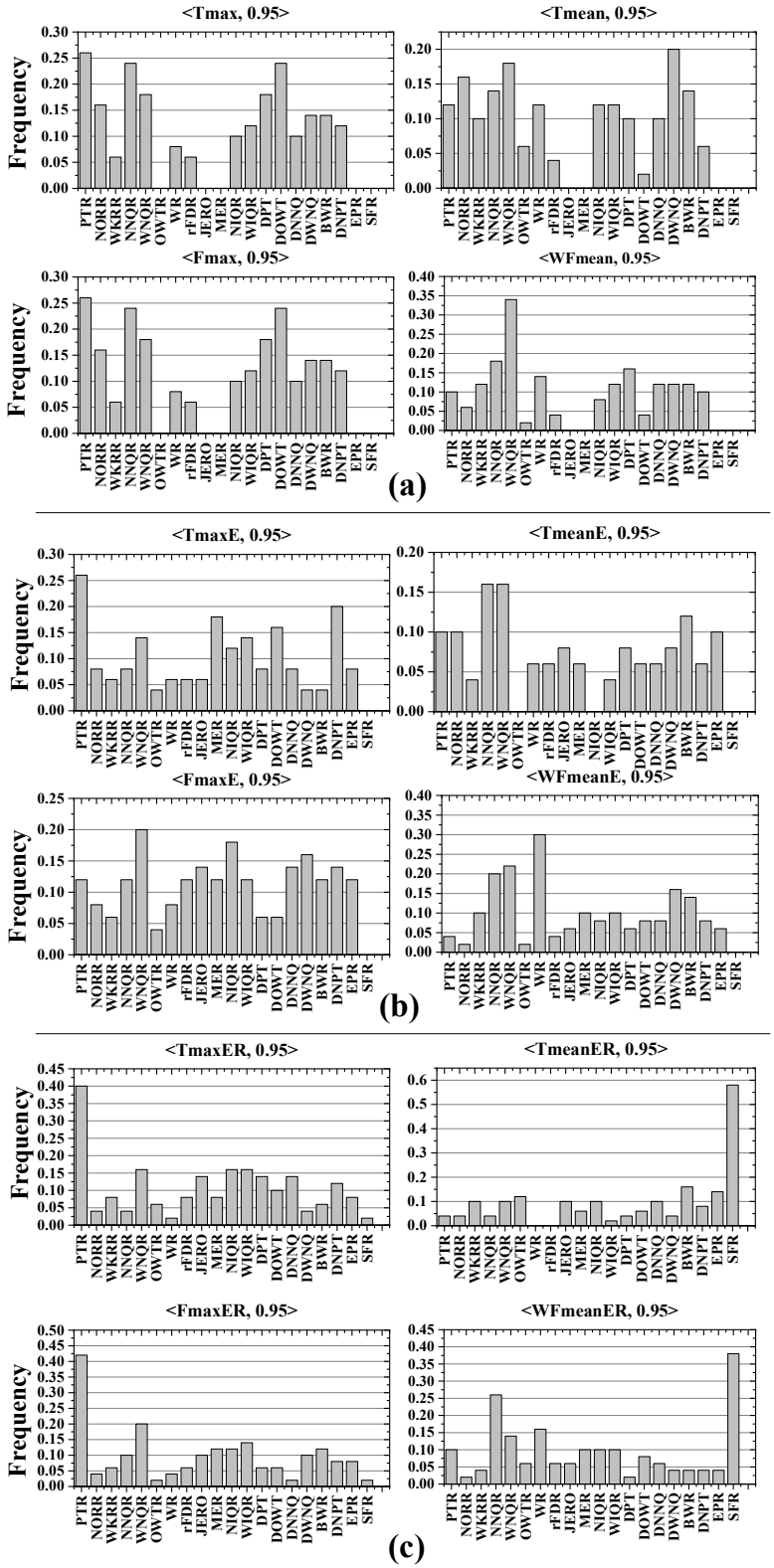

Figure 5.14: Frequency of the normalized terminals over 50 independent runs in the example scenarios.

based on the deviation of operation waiting time (DOWT). When some operations have a large waiting time, those operations are likely to be delayed or have a long flowtime. Dispatching rules should prioritize these operations before it is too late. Contrarily, to improve global performance, Tmean and Fmean do not consider DOWT but consider more machines in the job shop. For example, Tmean emphasizes the deviation of workload in the next machine (DWNQ) and bottleneck workload ratio (BWR). Furthermore, WFmean has the highest rate of using the job weights (WR), with a frequency of nearly 0.15. It implies that switching dispatching behaviors based on the job weights can effectively improve the performance of weighted objectives.

In the energy-considered scenario set (Fig. 5.14-(b)), we have two main observations. First, energy-related terminals are all highlighted in the second scenario set. For example, ⟨TmaxE, 0.95⟩ frequently considers MER, and ⟨TmeanE, 0.95⟩ frequently considers EPR. This implies that different energy prices need different dispatching behaviors. Second, the distributions of other normalized terminals are similar to those in the basic scenario set. For example, NNQR and WNQR are also frequently used in all four example scenarios, ⟨TmaxE, 0.95⟩, and ⟨FmaxE, 0.95⟩ prefer processing time-related terminals such as PTR, and ⟨WFmeanE, 0.95⟩ changes behaviors based on WR. The two observations confirm that the G2LGP-IF dispatching rules simultaneously optimize tardiness- or flowtime-related objectives, and energy cost by dynamically adjusting the dispatching behaviors based on decision situations.

In the third scenario set (Fig. 5.14-(c)) that simultaneously optimizes tardiness/flowtime, energy cost, and response time, the response cost rate ratio SFR is extensively considered in TmeanER and WFmeanER, which means different response cost rates need different dispatching rules. However, the results show that changing behaviors based on SFR is not a good choice for optimizing maximum tardiness and maximum flowtime. ⟨TmaxER, 0.95⟩ and ⟨FmaxER, 0.95⟩ mainly change behaviors based

on processing time (PTR) and the workload of the next machine queue (WNQR).

Based on the twelve example scenarios, we find that WNQR (and NNQR) and WIQR (and NIQR) are frequently used information in all different scenarios. WNQR and NNQR represent similar information indicating how large a work burden the next machine has. WIQR and NIQR represent similar information indicating how large a work burden the current machine has. Although these four normalized terminals might not be the most frequently used ones in the example scenarios, WNQR (and NNQR) and WIQR (and NIQR) have a relatively high frequency (i.e., more than 0.1) in nearly all the cases. The observation implies that different machine situations likely need different dispatching rules.

The analyses for Fig. 5.14 have some new findings compared to existing feature analyses of DJSS [135, 261]. Some input features that did not show their importance in existing studies are highlighted by IF operations. For example, existing studies seldom see WIQ and NIQ as important features. But WIQR and NIQR which represent the same information as WIQ and NIQ are frequently used by IF operations. Existing studies seldom use the operation waiting time (OWT) in output rules. But our analyses show that operation waiting time (i.e., DOWT) is very useful in the IF operations of $\langle$Tmax, 0.95$\rangle$, $\langle$Fmax, 0.95$\rangle$, and $\langle$TmaxE, 0.95$\rangle$.

5.7 Chapter Summary

This chapter aims to find a way to effectively evolve dispatching rules based on domain knowledge for solving complicated DJSS problems. Specifically, we propose G2LGP which provides a way to define constraints on LGP search spaces. Then we identify three key domain knowledge of IF operations and further propose a new set of normalized terminals for DJSS problems and a set of grammar rules to restrict the available inputs, the number, and the locations of IF operations for evolving

IF-included dispatching rules.

We comprehensively investigate the effectiveness of dispatching rules and our proposed method on three scenario sets. The empirical results confirm that IF operations are crucial for dispatching rules in complex problems. The results have verified that using grammar rules to restrict the usage of IF operations in LGPHH is an effective way to harness IF operations. Armed with the proposed normalized terminals and grammar rules, the proposed method outperforms the state-of-the-art LGPHH method in terms of both effectiveness and interpretability. Further analyses highlight that IF operations greatly improve the flexibility of dispatching rules, performing different dispatching behaviors based on different decision situations. The analyses also show a better interpretability of dispatching rules evolved by G2LGP than basic LGP. The analysis on the output dispatching rules finds three important DJSS features that are missed by existing IF-excluded dispatching rules. By properly restricting the use of IF operations, this chapter shows great potential for dispatching rules with IF operations in solving complex problems. This chapter also gives an example for DJSS users to incorporate domain knowledge into GP search via grammar-guided techniques.

In the previous chapters, we have improved the performance of GPHH by 1) replacing tree-based representation with linear representation, 2) designing graph-based search mechanisms for LGPHH, and 3) incorporating domain knowledge to restrict LGP search spaces. These improvements are essentially designing better FLs for solving DJSS problems. However, designing better FLs is tedious and highly dependent on specific domain knowledge. In the next chapter, we will propose a fitness landscape optimization method to automatically optimize FLs.

Chapter 6

Fitness Landscape Optimization for LGP for DJSS

A fitness landscape (FL) plays a crucial role in genetic programming search. In this chapter, we propose an FL optimization method to automatically design easier FLs by optimizing the neighborhood structure of LGP solutions.

6.1 Introduction

An FL is a surface that reflects the fitness of all the possible solutions in a search space [105]. A smoother FL with less local optima (e.g., an unimodal landscape) normally implies an easier search problem. An FL consists of three components: fitness function, solution space, and the neighborhood structure of solutions [228]. A fitness function measures the effectiveness and quality of all the possible solutions, the possible solutions compose a solution space, and the neighborhood structure defines the distance among the possible solutions in the solution space. However, the FL of GP is normally extremely rugged because of the low causality among symbolic solutions (i.e., a small change in a computer program might lead to a huge change in the final output). The rugged FLs make GP search

very challenging.

In recent years, some advanced techniques successfully enhanced GP search performance by designing better FLs. For example, frequency-based [259] and semantic-based operators [183] change the neighborhood structures (e.g., one-hop mutation) so that GP prefers particular neighbors with a large possibility of good solutions. MRGP in chapter 4 helps GP jump out from local optima by cooperating synergy between GP representations. However, these manually enhanced fitness landscapes need very specific domain knowledge and strong assumptions. For example, MRGP has to find two (or more) helpful GP representations and design their graph-to-representation transformations. Frequency-based mutation has to assume that the effective solutions include an effective primitive multiple times, which might not be the case in some applications (e.g., in program synthesis, a program repeats a primitive by looping [238]). Moreover, it is tedious for human experts to design better FLs.

Given the performance gain brought by better fitness landscapes, this chapter aims to study two research questions:

1. Are there any other better fitness landscapes than the existing ones?

2. How can we find these better fitness landscapes automatically?

To answer these two research questions, this chapter proposes a fitness landscape optimization (FLO) method that aggregates good solutions and separates good and poor solutions in the search space by changing their neighborhood structures automatically. We take LGP as an example to verify the effectiveness of the proposed method. The linear representation of LGP individuals facilitates our demonstration of the proposed method.

6.1.1 Chapter Goals

The main goal of this chapter is to *develop an FLO method to automatically design better FLs for LGP*. This chapter first proposes an FLO method based

on LGP. We model the fitness landscape optimization into a minimization problem whose decision variables are symbol indexes. Then, this chapter verifies the hardness reduction caused by the proposed methods on simple benchmark problems. We also visualize the FLs to make a further discussion. Based on the achievement on the simple benchmarks, this chapter extends the proposed methods to common DJSS problems to verify its effectiveness. Specifically, this chapter has the following goals:

1. Develop an FLO method and illustrate the proposed method based on LGP. The proposed FLO method optimizes the FL by changing the neighborhood structures of LGP solutions.

2. Analyze the optimized FLs by visualizing the FLs of a simple DJSS problem. The simple DJSS problem facilitates us to enumerate and evaluate all the possible solutions for visualization.

3. Apply the proposed FLO to solve common DJSS problems and verify the performance gain of LGP methods if searching against the optimized FLs.

6.1.2 Chapter Organization

The rest of this chapter is organized as follows. First, section 6.2 proposes the FLO method based on LGP. Section 6.3 investigates the hardness reduction of FLs and makes a discussion on the visualized FLs. Section 6.3.2 further extends the proposed method to common DJSS problems to verify the superior performance of the proposed method. Section 6.4 concludes this chapter.

6.2 Proposed Method

6.2.1 Main Idea

The main idea of fitness landscape optimization is first indexing symbols into integers and second optimizing the fitness landscape based on symbol

indexes. In this chapter, a symbol is an LGP instruction, and LGP individuals are fixed-length index vectors. The Euclidean distance of the index vectors defines the neighborhood structures of LGP solutions on FLs. Different from the neighborhood structures that are defined by basic genetic operators (e.g., macro mutation), the neighborhood structures based on indexes are flexible to be optimized. We optimize the fitness landscape, essentially the neighborhood structure, by changing the indexes of LGP instructions. We do not optimize the fitness function and the solution space here since they are usually given by specific problems. We denote the indexes of symbols as $\mathbb{I} = [\mathbb{I}_1, \mathbb{I}_2, ..., \mathbb{I}_l, ..., \mathbb{I}_n]^T$ where n is the number of symbols. We define an optimization objective function $F(\mathbb{I})$ for the fitness landscape. FLO is virtually an optimizing problem

$$\mathbb{I} = \arg_{\mathbb{I}} \min F(\mathbb{I}).$$

The main idea of the objective $F(\mathbb{I})$ is to minimize the distance between good solutions (i.e., good GP individuals), maximize the distance between good and poor solutions, and encourage the optimized indexes to be consistent with domain knowledge. By this means, the local optima between optimal solutions would be reduced, and the FL becomes less rugged. The symbol indexes $\mathbb{I}$ represent an LGP genotype G_i by a mapping matrix θ_i, $G_i = \theta_i \mathbb{I}$, where

$$G_i = [G_{i1}, G_{i2}, ..., G_{ik}, ..., G_{im}]^T$$

and

$$\theta_i = \mathbb{R}^{m \times n} = \begin{bmatrix} \theta_{i1} \\ \theta_{i2} \\ \vdots \\ \theta_{im} \end{bmatrix} = \begin{bmatrix} \theta_{i,1,1} & \theta_{i,1,2} & \cdots & \theta_{i,1,n} \\ \theta_{i,2,1} & \ddots & & \vdots \\ \vdots & & \theta_{i,k,l} & \vdots \\ \theta_{i,m,1} & \cdots & \cdots & \theta_{i,m,n} \end{bmatrix}.$$

m is the maximum length of G_i. Each row of θ_i has at most one "1", and the rest of its elements are "0". Since LGP individuals unnecessarily have the maximum length m, G_i uses a placeholder "-1" to index those empty symbols.

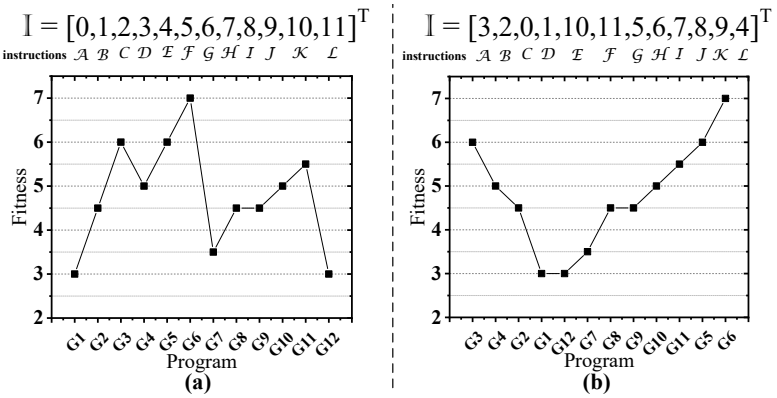

Figure 6.1: A simple example of FLO. (a) the initial FL is rugged; (b) the optimized FL is cone-like. The coordinate of a program is equivalent to its index vector.

To illustrate the idea, Fig. 6.1 shows a simple example of FLO. Suppose there are 12 possible instructions in the LGP search space, denoted from $\mathcal{A}$ to $\mathcal{L}$. These instructions are indexed by 0 to 11 initially (i.e., $\mathbb{I} = [0, 1, ..., 11]^T$). Each program in the search space has one instruction. We thus have 12 unique programs, denoted from G_1 to G_{12}. Specifically, $G_1 = [\mathcal{A}]$, $G_2 = [\mathcal{B}]$, $G_3 = [\mathcal{C}]$, etc. Fig. 6.1-(a) shows the initial FL of these programs. We can see that the initial FL is rugged. There are two optimal programs (supposing it is a minimizing problem) on the FL, "$G_1 = [\mathcal{A}]$" and "$G_{12} = [\mathcal{L}]$", indexed by "$[0]$" and "$[11]$" respectively. To smoothen the FL and reduce the local optima, we aggregate good programs (e.g., G_1 and G_2) and separate good and poor programs (e.g., G_1 and G_6). For example, we let $\mathcal{A}$ and $\mathcal{L}$ have indexes of 3 and 4 respectively. Consequently, G_1 has an index vector of "$[3]$", and G_2 has an index vector of "$[4]$", Fig. 6.1-(b) shows the optimized FL, where the instruction index $\mathbb{I} = [3, 2, 0, 1, ..., 9, 4]^T$, each instruction with a different index from the initial $\mathbb{I}$. The new $\mathbb{I}$ defines new neighborhood structures of the LGP search space. We can see that by aggregating good solutions and separating good and poor solutions, the FL becomes smoother and more "cone-like" (i.e.,

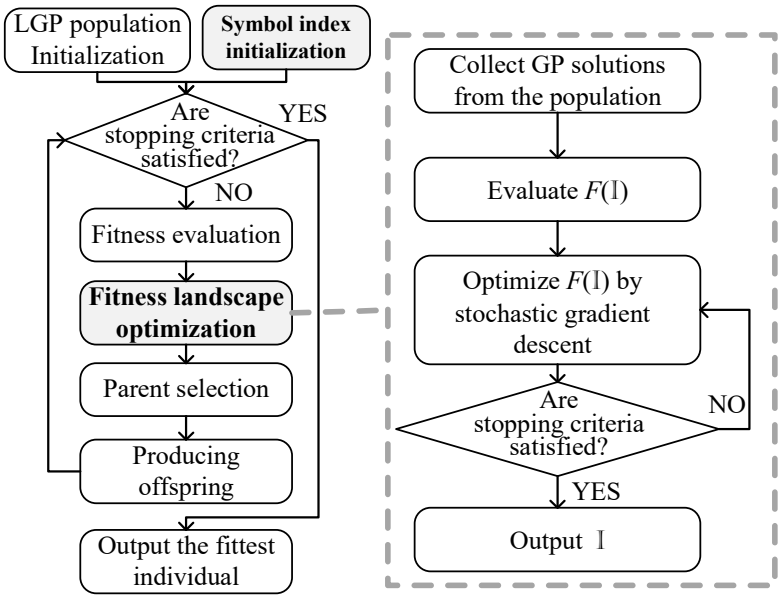

Figure 6.2: The overall framework of FLO (along with the evolutionary framework of LGP on the left).

the distance to the optimal solutions is highly correlated to the fitness discrepancy) than the initial one. The number of local optima is also reduced. It is easier for GP to search for optimal solutions on the optimized FL than on the initial one.

6.2.2 Overall Framework

We optimize the fitness landscape over GP evolution. Fig. 6.2 shows the overall framework of FLO along with the evolutionary framework of basic LGP. We first enumerate all the possible instructions based on the given primitive set and index an initial symbol index $\mathbb{I}$. Each LGP instruction is a symbol. To improve efficiency, we treat the alphabetically equivalent instructions as one instruction. For example, instruction "R[0]$= x_0 + x_1$" is equivalent to "R[0]$= x_1 + x_0$". We randomly index the instructions initially.

At each generation, we optimize FL after the fitness evaluation of LGP individuals. FLO collects LGP individuals from the current LGP population. FLO optimizes the symbol indexes based on the collected LGP individuals in the population since it is impractical to consider all possible LGP individuals, and the evolved individuals in the population often have better fitness (i.e., more important) than the randomly sampled individuals (i.e., importance sampling) [228]. We denote the set of collected LGP individuals as $\mathbb{B}$. To collect a diverse set of good LGP individuals, $\mathbb{B}$ contains the individuals of the top B fitness in the population, each fitness value sampling one individual.

FLO then evaluates $F(\mathbb{I})$ based on the collected LGP individuals. FLO applies a stochastic gradient descent method to optimize the indexes. FLO repeats the optimization until it reaches the stopping criteria. In this work, the stopping criterion is the maximal iteration number of the stochastic gradient descent.

6.2.3 Optimization Objectives

This section formulates our optimization objectives $F(\mathbb{I})$. Based on the main idea of optimization objectives, there are three sub-objectives: inner distance between good individuals, inter distance between good and poor individuals, and the consistency with domain knowledge. They are denoted as $D(\mathbb{I})$, $D_{lose}(\mathbb{I})$, and $E(N(\mathbb{I}))$, respectively. $F(\mathbb{I})$ combines these three objectives linearly, as shown in Eq. (6.1) where α_1 to α_3 are 1.0 by default.

$$F(\mathbb{I}) = \frac{\alpha_1}{D(\mathbb{I}_0)} D(\mathbb{I}) + \frac{\alpha_2 \times D_{lose}(\mathbb{I}_0)}{D_{lose}(\mathbb{I})} + \frac{\alpha_3}{E(N(\mathbb{I}_0))} E(N(\mathbb{I})) \tag{6.1}$$

subject to

$$\mathbb{I}_l \in N; \mathbb{I}_l < n; \mathbb{I}_i \neq \mathbb{I}_j \text{ if } i \neq j$$

The three constraints ensure that the range and the uniqueness of the indexes. Because these sub-objectives have different data scales, they are

normalized to $[0, 1]$ by $D(\mathbb{I}_0)$, $D_{lose}(\mathbb{I}_0)$, and $E(N(\mathbb{I}_0))$, where $\mathbb{I}_0$ are the initial indexes at each generation before optimization.

Inner Distance between Good Individuals

$$D(\mathbb{I}) = \frac{1}{|\mathbb{B}|^2} \sum_{a=1}^{|\mathbb{B}|} \sum_{b=1}^{|\mathbb{B}|} ||G_a - G_b||_2^2 \tag{6.2}$$

$D(\mathbb{I})$ formulates the Euclidean distance between good LGP individuals, as shown in Eq. (6.2). To simplify the derivation, we omit the square root in the Euclidean distance (i.e., squaring the L^2 norm). For any pairs of good LGP individuals in $\mathbb{B}$, $D(\mathbb{I})$ summarizes the squaring Euclidean distance and gets their average. Here, we optimize the inner distance of the best-of-the-run individuals to avoid the distraction from less effective solutions at the beginning of the evolution.

To clarify the relationship between $\mathbb{I}$ and $D(\mathbb{I})$, we denote

$$\begin{aligned} Q_{ab}(\mathbb{I}) = ||G_a - G_b||_2^2 &= \sum_{k}^{m} (G_{ak} - G_{bk})^2 \\ &= \sum_{k}^{m} (\theta_{a,k}\mathbb{I} - \theta_{b,k}\mathbb{I})^2 \\ &= \sum_{k}^{m} (\sum_{l}^{n} (\theta_{a,k,l} - \theta_{b,k,l})\mathbb{I})^2 \end{aligned} \tag{6.3}$$

and $D(\mathbb{I}) = \frac{1}{|\mathbb{B}|^2} \sum_{a=1}^{|\mathbb{B}|} \sum_{b=1}^{|\mathbb{B}|} Q_{ab}(\mathbb{I})$.

Note that since G_{ak} and G_{bk} might be a placeholder "-1". The subtraction of $G_{ak} - G_{bk}$ is redefined as:

$$G_{ak} - G_{bk} = \begin{cases} G_{ak} - G_{bk} & G_{ak} \geq 0 \text{ and } G_{bk} \geq 0 \\ 0 & G_{ak} < 0 \text{ and } G_{bk} < 0 \\ 1 & \text{otherwise} \end{cases}$$

Inter Distance between Good and Bad Individuals

$$D_{lose}(\mathbb{I}) = \frac{1}{|\mathbb{B}||\mathbb{B}_{lose}|}\sum_{a\in\mathbb{B}}^{|\mathbb{B}|}\sum_{b\in\mathbb{B}_{lose}}^{|\mathbb{B}_{lose}|}||G_a - G_b||_2^2$$
$$= \frac{1}{|\mathbb{B}||\mathbb{B}_{lose}|}\sum_{a\in\mathbb{B}}^{|\mathbb{B}|}\sum_{b\in\mathbb{B}_{lose}}^{|\mathbb{B}_{lose}|}Q_{ab}(\mathbb{I}) \tag{6.4}$$

In contrast to $D(\mathbb{I})$, $D_{lose}(\mathbb{I})$ formulates the Euclidean distance between good and poor LGP individuals, as shown in Eq. (6.4). $D_{lose}(\mathbb{I})$ considers poor LGP individuals (i.e., "less competitive" LGP individuals) in the calculation (i.e., $\mathbb{B}_{lose}$). Because at the late stage of GP evolution, the poor individuals in the population also likely contain useful building blocks, $\mathbb{B}_{lose}$ are collected from both the current population and historical poor individuals to make $D_{lose}(\mathbb{I})$ more comprehensive. Specifically, the less competitive LGP individuals from the current population are sampled from the half population with worse fitness. The historical poor LGP individuals are sampled from an archive that records the poor LGP individuals over the generations. $\mathbb{B}_{lose}$ have the same size with $\mathbb{B}$.

Domain Knowledge Consistency

$$E(N(\mathbb{I})) = ||N_0 \otimes N_0 - \frac{1}{n^2}N(\mathbb{I}) \otimes N(\mathbb{I})||_2^2 \tag{6.5}$$

$E(N(\mathbb{I}))$ is defined as the differences between the domain knowledge and the existing indexes, as shown in Eq. (6.5). Smaller differences imply that the symbol indexes are more consistent with the domain knowledge. Eq. (6.5) squares the L^2 norm to simplify the derivation, and applies the element-wise production (denoted as $\otimes$). N_0 is a predefined distance matrix whose elements are the distance between symbols based on the domain knowledge. For example, character $\mathcal{A}$ is closer to $\mathcal{B}$ than to $\mathcal{Z}$ based

on the alphabetical order. We denote N_0 as

$$N_0 = \mathbb{R}^{n\times n} = [d_{N,1}, \cdots, d_{N,l}, \cdots, d_{N,n}]$$
$$= \begin{bmatrix} d_{N,1,1} & \cdots & d_{N,l,1} & \cdots & d_{N,n,1} \\ d_{N,1,2} & \ddots & & & \vdots \\ \vdots & & & \ddots & \vdots \\ d_{N,1,n} & & \cdots & & d_{N,n,n} \end{bmatrix}.$$

In this chapter, we define $d_{N,a,b}$ as the product of the editing distance ($D_{g,a,b}$) and the semantic distance ($D_{s,a,b}$) of instructions a and b. $d_{N,a,b} = \frac{D_{g,a,b}}{D_{g,max}} \cdot \frac{D_{s,a,b}}{D_{s,max}}$ where $D_{g,max}$ and $D_{s,max}$ are the maximum D_g and D_s among the possible instructions. The semantic distance is defined as the Euclidean distance between instruction outputs. The instructions get their outputs based on ten randomly sampled input instances. As these instruction outputs can be reused for different problems, they are not counted in the total fitness evaluation.

$N(\mathbb{I})$ is a distance matrix between the current indexes.

$$N(\mathbb{I}) = \mathbb{R}^{n\times n} = [\Delta_1\mathbb{I}, \cdots, \Delta_l\mathbb{I}, \cdots, \Delta_n\mathbb{I}]$$

where

$$\Delta_l = \mathbb{R}^{n\times n} = \begin{matrix} \\ \\ \\ \\ \\ \\ \end{matrix} \overset{l^{th}column}{\begin{bmatrix} -1 & & & 1 & \\ & -1 & & 1 & \\ & & \ddots & 1 & \\ & & & 0 & \\ & & & 1 & -1 \end{bmatrix}}$$

Because $d_{N,i,j} \in [0, 1]$, we normalize $N(\mathbb{I}) \otimes N(\mathbb{I})$ by $\frac{1}{n^2}$. Based on N_0 and $N(\mathbb{I})$, Eq. (6.5) can be extended as $E(N(\mathbb{I})) = \sum_j^n \sum_i^n \left(d_{N,i,j}^2 - \frac{1}{n^2}(\mathbb{I}_j - \mathbb{I}_i)^2\right)^2$.

6.2.4 Stochastic Gradient Descent

We optimize $F(\mathbb{I})$ by a stochastic gradient descent method. Specifically, the new indexes

$$\mathbb{I}' = \lfloor \mathbb{I} - sign(\frac{\partial F}{\partial \mathbb{I}}) - U(\sigma n) \times \frac{\partial F}{\partial \mathbb{I}} \rfloor \tag{6.6}$$

where σ is the maximum step size and $U(\sigma n)$ returns a uniformly random float value in $[0, \sigma n]$. σ has a range of $(0, 1]$ and multiplies by n to represent actual step size. $sign(\cdot)$ returns 1 for positive inputs, 0 for zero inputs, and -1 otherwise. $-sign(\frac{\partial F}{\partial \mathbb{I}})$ ensures that a symbol index at least moves one unit along the negative gradient. If $F(\mathbb{I}') < F(\mathbb{I})$, $\mathbb{I}$ is updated by $\mathbb{I}'$ (i.e., $\mathbb{I} = \mathbb{I}'$). The stochastic gradient descent iterates until it reaches the maximum number of iterations. Based on Eq. (6.1),

$$\frac{\partial F}{\partial \mathbb{I}} = \frac{\alpha_1}{D(\mathbb{I}_0)} \frac{\partial D}{\partial \mathbb{I}} + \frac{\alpha_2 \times D_{lose}(\mathbb{I}_0)}{-D_{lose}(\mathbb{I})^2} \frac{\partial D_{lose}}{\partial \mathbb{I}} + \frac{\alpha_3}{E(N(\mathbb{I}_0))} \frac{\partial E}{\partial \mathbb{I}} \tag{6.7}$$

The partial derivation of $D(\mathbb{I})$ is shown as Eq. (6.8).

$$\frac{\partial D}{\partial \mathbb{I}} = \left[\frac{\partial D}{\partial \mathbb{I}_1}, \cdots, \frac{\partial D}{\partial \mathbb{I}_l}, \cdots, \frac{\partial D}{\partial \mathbb{I}_n} \right]^T \tag{6.8}$$

where

$$\begin{aligned} \frac{\partial D}{\partial \mathbb{I}_l} &= \frac{1}{|\mathbb{B}|^2} \sum_{a=1}^{|\mathbb{B}|} \sum_{b=1}^{|\mathbb{B}|} \frac{\partial Q_{ab}}{\partial \mathbb{I}_l} \\ &= \frac{1}{|\mathbb{B}|^2} \sum_{a=1}^{|\mathbb{B}|} \sum_{b=1}^{|\mathbb{B}|} (\sum_{k}^{m} 2(G_{ak} - G_{bk})(\theta_{a,k,l} - \theta_{b,k,l})) \end{aligned} \tag{6.9}$$

Based on Eq. (6.9), we can know that when a and b are duplicated individuals, the gradient of $\frac{\partial D}{\partial \mathbb{I}_l}$ will vanish. To avoid the gradient vanishing, we should eliminate duplicated individuals in $\mathbb{B}$ and consider diverse good GP individuals.

Similarly, the partial derivation of $D_{lose}(\mathbb{I})$ is shown as Eq. (6.10).

$$\frac{\partial D_{lose}}{\partial \mathbb{I}} = \left[\frac{\partial D_{lose}}{\partial \mathbb{I}_1}, \cdots, \frac{\partial D_{lose}}{\partial \mathbb{I}_l}, \cdots, \frac{\partial D_{lose}}{\partial \mathbb{I}_n} \right]^T \tag{6.10}$$

where

$$\begin{aligned}\frac{\partial D_{lose}}{\partial \mathbb{I}_l} &= \frac{1}{|\mathbb{B}|}\frac{1}{|\mathbb{B}_{lose}|}\sum_{a\in\mathbb{B}}^{|\mathbb{B}|}\sum_{b\in\mathbb{B}_{lose}}^{|\mathbb{B}_{lose}|}\frac{\partial Q_{ab}}{\partial \mathbb{I}_l} \\ &= \frac{1}{|\mathbb{B}|}\frac{1}{|\mathbb{B}_{lose}|}\sum_{a\in\mathbb{B}}^{|\mathbb{B}|}\sum_{b\in\mathbb{B}_{lose}}^{|\mathbb{B}_{lose}|} \\ &\quad \left[\sum_{k}^{m} 2(G_{ak}-G_{bk})(\theta_{a,k,l}-\theta_{b,k,l})\right] \end{aligned} \tag{6.11}$$

As $\mathbb{B}$ and $\mathbb{B}_{lose}$ are unlikely overlapped (if they are overlapped, just simply remove the overlapped part), Eq. (6.11) is non-zero.

The partial derivation of $E(N(\mathbb{I}))$ is shown as Eq. (6.12).

$$\frac{\partial E}{\partial \mathbb{I}} = \left[\frac{\partial E}{\partial \mathbb{I}_1}, \cdots, \frac{\partial E}{\partial \mathbb{I}_l}, \cdots, \frac{\partial E}{\partial \mathbb{I}_n}\right]^T \tag{6.12}$$

where

$$\begin{aligned}\frac{\partial E}{\partial \mathbb{I}_l} &= \sum_{j}^{n} 2\left[d_{N,l,j}^2 - \frac{1}{n^2}(\mathbb{I}_j - \mathbb{I}_l)^2\right]\frac{2}{n^2}(\mathbb{I}_j - \mathbb{I}_l) \\ &\quad + \sum_{i}^{n} 2\left[d_{N,i,l}^2 - \frac{1}{n^2}(\mathbb{I}_l - \mathbb{I}_i)^2\right]\frac{2}{n^2}(\mathbb{I}_i - \mathbb{I}_l)\end{aligned} \tag{6.13}$$

Note that in Eq. (6.13), the first item is not zero only when $i = l$ and $j \neq l$, and the second item is not zero when $j = l$ and $i \neq l$. We can simplify Eq. (6.13) by assuming $d_{N,i,l} = d_{N,l,i}$ and denoting j in the first item as i without loss of generality, as shown in Eq. (6.14)

$$\begin{aligned}\frac{\partial E}{\partial \mathbb{I}_l} &= \frac{4}{n^2}\left[\sum_{i}^{n}\left[d_{N,l,i}^2 - \frac{1}{n^2}(\mathbb{I}_i - \mathbb{I}_l)^2\right](\mathbb{I}_i - \mathbb{I}_l)\right. \\ &\quad \left. + \sum_{i}^{n}\left[d_{N,i,l}^2 - \frac{1}{n^2}(\mathbb{I}_l - \mathbb{I}_i)^2\right](\mathbb{I}_i - \mathbb{I}_l)\right] \\ &= \frac{8}{n^2}\left[\sum_{i}^{n}\left[d_{N,l,i}^2 - \frac{1}{n^2}(\mathbb{I}_i - \mathbb{I}_l)^2\right](\mathbb{I}_i - \mathbb{I}_l)\right]\end{aligned} \tag{6.14}$$

In practice, we do not have to care the $|\mathbb{I}_i - \mathbb{I}_j|$ if $\mathbb{I}_i$ and $\mathbb{I}_j$ are not effective symbols. So, we further simplify Eq. (6.14) as:

$$\frac{\partial E}{\partial \mathbb{I}_l} = \frac{8}{n^2}\left[\sum_{i\in\mu(\mathbb{I})}^{|\mu(\mathbb{I})|}\left[d_{N,l,i}^2 - \frac{1}{n^2}(\mathbb{I}_i - \mathbb{I}_l)^2\right](\mathbb{I}_i - \mathbb{I}_l)\right] \tag{6.15}$$

where $\mu(\mathbb{I})$ are the used symbols in good individuals $\mathbb{B}$.

$\frac{\partial F}{\partial \mathbb{I}}$ is normally very sparse (i.e., there are a large number of zero or nearly zero elements) when n is large after normalization, which greatly limits the search efficiency of the stochastic gradient descent. To improve the search efficiency, we adjust $\frac{\partial F}{\partial \mathbb{I}}$ by

$$\frac{\partial F}{\partial \mathbb{I}_l} = sign(\frac{\partial F}{\partial \mathbb{I}_l})/(-\ln(\left|\frac{\partial F}{\partial \mathbb{I}_l}\right|))$$

To satisfy the constraints in Eq. (6.1), we perform a flooring operation on $\mathbb{I}_l$ if it is not an integer. To prioritize the effective symbols (indicated by $\mu(\mathbb{I})$), we first update $\mathbb{I}_l(l \in \mu(\mathbb{I}))$ one-by-one in a random order, and then update $\mathbb{I}_l(l \notin \mu(\mathbb{I}))$ one-by-one in a random order based on Eq. (6.6). If $\mathbb{I}'_i = \mathbb{I}'_j(i \neq j)$, we assign $\mathbb{I}'_j$ a random integer ranging between 0 and $n-1$ until $\mathbb{I}'_j$ is unique in $\mathbb{I}'$.

To show the proposed method more clearly, Alg. 15 shows the pseudo-code of FLO. Given an LGP population $\mathbf{P}$ and symbol indexes $\mathbb{I}$, we collect the good and bad individuals ($\mathbb{B}$ and $\mathbb{B}_{lose}$) and the used symbols $\mu(\mathbb{I})$ in good individuals from the population. Then we iterate the stochastic gradient descent up to 20 times. If the new objective function $F(\mathbb{I}')$ is smaller than the original objective value $F(\mathbb{I})$, we use $\mathbb{I}'$ to update $\mathbb{I}$. The new symbol indexes are outputted to form new FLs.

6.3 Experimental Studies of FLO

To verify the effectiveness of the proposed FLO method, this section first analyzes the optimized FL based on a simple DJSS problem and then ap-

Algorithm 15: `Fitness Landscape Optimization`

Input: An LGP population $\mathbf{P}$, symbol indexes $\mathbb{I}$
Output: New symbol indexes $\mathbb{I}_{new}$

$\mathbb{B}$,$\mathbb{B}_{lose} \leftarrow$ collect good and bad individuals from $\mathbf{P}$.
$\mu(\mathbb{I}) \leftarrow$ get the used symbols in $\mathbb{B}$.
Evaluate $F(\mathbb{I})$.
for $j \leftarrow 1$ *to* 20 **do**
    Evaluate $\frac{\partial F}{\partial \mathbb{I}}$ if $\mathbb{I}$ is changed.
    $\mathbb{I}' \leftarrow \lfloor \mathbb{I} - sign(\frac{\partial F}{\partial \mathbb{I}}) - U(\sigma n) \times \frac{\partial F}{\partial \mathbb{I}} \rfloor$
    Evaluate $F(\mathbb{I}')$.
    if $F(\mathbb{I}') < F(\mathbb{I})$ **then**
        /* $\mathbb{I} \leftarrow \mathbb{I}'$ */
        $\mathbb{I}_l \leftarrow \mathbb{I}'_l, l \in \mu(\mathbb{I})$ in a random order.
        $\mathbb{I}_l \leftarrow \mathbb{I}'_l, l \notin \mu(\mathbb{I})$ in a random order.
        $F(\mathbb{I}) \leftarrow F(\mathbb{I}')$.
$\mathbb{I}_{new} \leftarrow \mathbb{I}$.
Return $\mathbb{I}_{new}$;

plies the proposed FLO to enhance LGP for solving common DJSS benchmarks.

6.3.1 Analyses on An Optimized FL

Measures of FL Hardness

We first analyze the hardness of FLs. We expect that after FL optimization, the hardness of the examined problems should be reduced. We apply four common FL analysis metrics to measure the hardness of the test problems, including fitness distance correlation (FDC) [99], negative scope coefficient (NSC) [229], robustness (RBS) [86], and evolvability (EVO) [228]. Specifically, FDC implies the degree that an FL looks like a "cone" (i.e., a high

correlation between the distance to optima and the fitness difference from optima). NSC implies the degree of "bad evolvability". RBS implies the probability of a neutral move. EVO implies the probability of moving toward a better neighbor. The larger values of the four metrics imply an easier FL.

We select $\langle Tmean, 0.85\rangle$ as a case study DJSS problem. To limit the number of possible solutions in $\langle Tmean, 0.85\rangle$, we only retain necessary primitives (i.e., $\{+, -, \max, R_0, R_1, PT, OWT\}$) and constrain the maximum program size of 3 instructions.

Compared Methods

The compared methods in our experiments are essentially the neighborhood structures of FLs, which are the genetic operators of LGP in existing studies. The FLs of these compared methods have problem-specific fitness functions and use LGP search spaces as the solution spaces. With different neighborhood structures, the hardness of LGP FLs might be different.

We compare FLO with two methods, free macro mutation (denoted as "freemut") [21] and frequency-based macro mutation (denoted as "freqmut"). These two genetic operators define the neighborhood structures of compared FLs. The freemut is a basic macro mutation, serving as a baseline. It produces new LGP offspring by inserting or removing a random instruction from an LGP parent. The freqmut is a self-adaptive genetic operator over the generation. It samples new primitives of offspring based on the primitive frequency in the top-K individuals (K=10% in our experiments). The assumption that primitive frequency implies the importance of primitives is also common in advanced GP methods [259, 284].

The neighborhood structure of FLO is defined based on the Euclidean distance of index vectors. If denoting the neighborhood threshold as ϵ, two LGP individuals G_a and G_b are neighbors if $||G_a - G_b||_2 < \epsilon$. Since freemut and freqmut only vary one instruction each time, we also only vary one instruction when finding neighbors in FLO.

The FL analysis metrics analyze the hardness of FLs based on the three compared neighborhood structures respectively (i.e., freemut, freqmut, and the optimized neighborhood structures of FLO). We treat the LGP population at the final generation as the samples of FLs (i.e., importance sampling [228]) when calculating FL metrics. To compare the performance comprehensively, each method runs 50 independent runs on each tested problem.

Parameter Settings

FLO has two main parameters, the number of sampled good solutions from the population B and the maximum step size in the stochastic gradient descent σ. We set $B = 10$ and set $\sigma = 0.1$ by default. Specifically, to sample diverse good solutions in $\mathbb{B}$, we consider the top-B fitness values in the population, each getting one individual, and vice versa for $\mathbb{B}_{lose}$. The threshold of the neighborhood in FLO ϵ is not a parameter in FLO. It is a problem-specific parameter when calculating FL metrics. ϵ is set as approximately 5% of the total number of instructions. The parameters of the basic LGP evolution follow the settings in chapter 3. Each FLO iterates up to 20 times and it performs every two generations.

Experiment Results of FL Hardness

Table 6.1 shows the mean metric values (and their standard deviations) of the compared methods for $\langle Tmean, 0.85 \rangle$. We apply the Wilcoxon rank-sum test with a significant level of 0.05 to analyze these metric values. "$+$" indicates a significantly better metric value, "$-$" indicates a significantly worse metric value, and "$\approx$" indicates a statistically similar metric value with FLO. The best mean metric values are highlighted in bold. We can see that the FLs of FLO have significantly better FDC and EVO values than the compared methods. In terms of NSC, the optimized FLs of FLO show competitive hardness with the compared methods. Although the

Table 6.1: The mean metric values (and standard deviation) on the tested problem

Metrics	freemut	freqmut	FLO
FDC	0.072 (0.086) −	0.046 (0.082) −	**0.184 (0.128)**
NSC	−21.13 (70.12) ≈	−11.83 (42.6) ≈	**−11.23 (37.59)**
RBS	0.699 (0.037) ≈	**0.765 (0.034)** +	0.695 (0.019)
EVO	0.182 (0.043) −	0.092 (0.03) −	**0.407 (0.078)**

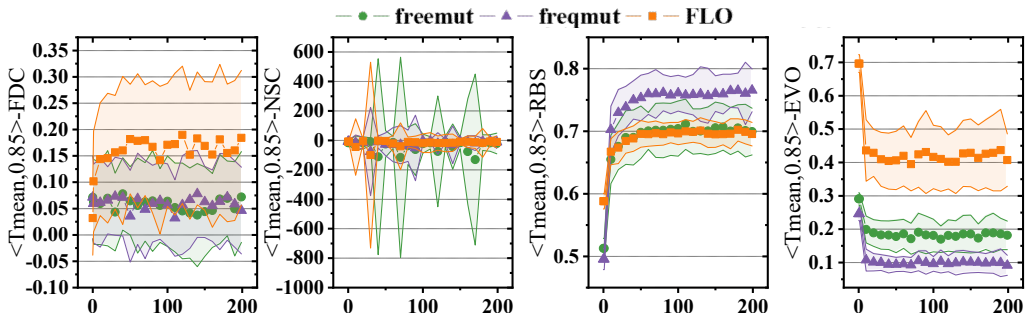

Figure 6.3: FL metrics over generations. X-axis: the number of generations, Y-axis: the metric values. The four sub-figures from the left to right are FDC, NSC, RBS, and EVO.

FLs of freqmut have better RBS values (i.e., a high probability of neutral move), they sacrifice the probability of moving to better neighbors (i.e, significantly worse EVO values than FLO). To conclude, the optimized FLs of FLO are significantly easier than the compared methods in terms of FDC and EVO and at least competitive with the compared methods in terms of NSC. The results show a very encouraging optimization performance of FLO.

To understand the optimization process of FLO, this section investigates the metrics over generations, as shown in Fig. 6.3. Each column in Fig. 6.3 shows a certain metric. In terms of FDC (i.e., the first sub-figure in

Fig. 6.3), FLO substantially improves FDC values over generations and maintains at a significantly higher level than freemut and freqmut. In terms of NSC, the three compared methods all stay at a similar level over the whole evolution. But we can see that FLO on average has a smaller standard deviation than the compared methods at the final stage of evolution, which implies a more stable FL hardness of FLO. The curves of RBS show that FLO is competitive with freemut but worse than freqmut, which is consistent with the results in table 6.1. The FLs of FLO also maintain a significantly higher level of EVO than the compared methods. The curves of FL metrics also confirm that the optimization of FLO is efficient since the curves of FLO often converge at the early stages of evolution.

FL Visualization

To intuitively investigate the FLs of FLO, we visualize the example FLs in this section, as shown in Fig. 6.4. Specifically, we show the initial FLs and the FLs at the 200^{th} generation in the first independent run. Fig. 6.4 shows the scatter plots of the example FLs. Each point on the FLs is a GP solution. The color of the points represents the fitness of the GP solution. To highlight the solutions with good and bad fitness, we only color the solutions with the best 25% fitness and the solutions with the worst 25% fitness, by cool tones and warm tones respectively. We also highlight the optimal solutions by purple stars.

We have three interesting findings based on Fig. 6.4. First, Fig. 6.4 confirms the optimization on FLs. In Fig. 6.4-(a-1) and -(a-2), good and bad solutions are distributed uniformly across the initial FLs. The blue and dark lines spread across the space. However, we can see that the good and bad solutions aggregate respectively after optimization of 100 times (i.e., optimizing FLs every two generations). For example, the good solutions approximately aggregate to the indexes from 15 to 35 on 2^{nd}-3^{rd}-instruction plane in Fig. 6.4-(b-2). More interestingly, the optimal solutions are also located near the cold-tone regions.

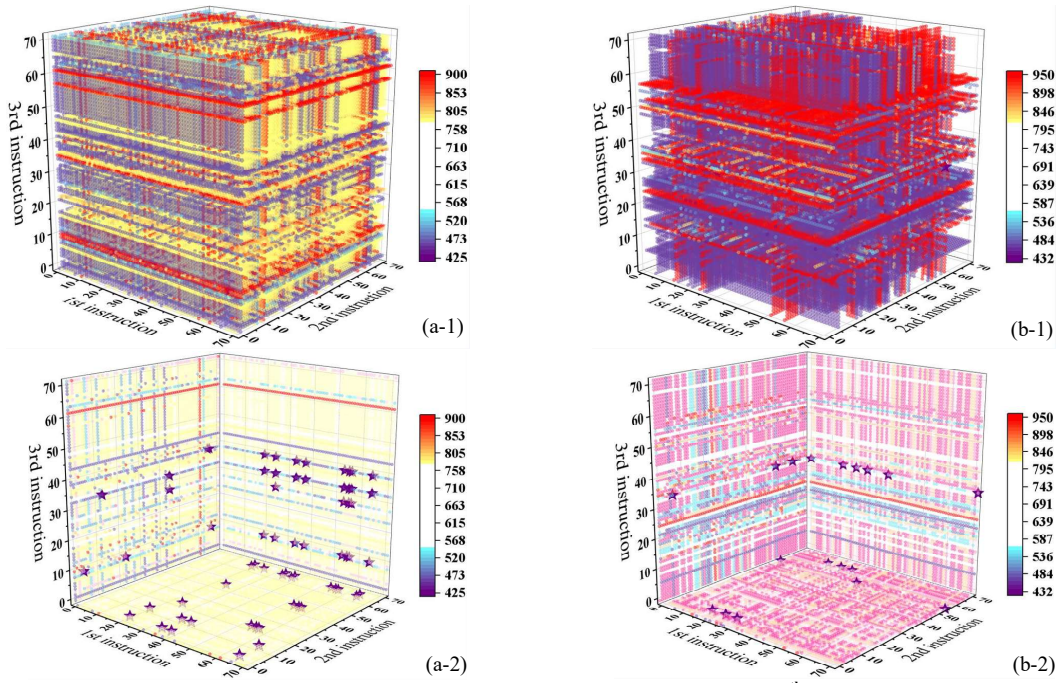

Figure 6.4: The example FLs of ⟨Tmean,0.85⟩. The cool tone indicates good fitness and the warm tone indicates bad fitness. The purple stars indicate the optimal solutions. (a-1) and (a-2) are initial FLs, (b-1) and (b-2) are the FL at the 200th generation. (a-2) and (b-2) are the FLs projected on x-y, x-z, and y-z planes.

Second, we can see a *fitness aligning* phenomenon. The fitness aligning phenomenon means that the solutions with very similar fitness allocate along an axis or a hyperplane (i.e., the fitness is aligned), rather than aggregating as an ellipse or a hypersphere. For example, there are considerable 2-D planes with the same color across Fig. 6.4-(b-1). This is because there are introns in the search space. A GP solution can easily reach another solution with the same fitness by adding, removing, or modifying introns, which is equivalent to moving along axes or against hyperplanes. The fitness aligning phenomenon also implies an empirically recommended

setting in LGP macro mutation, that is only mutating one instruction each time [21]. Based on the fitness aligning phenomenon, only mutating one instruction can 1) perform a neutral move if moving along the aligned fitness (i.e., similar to mutating introns), and 2) jump out of the current fitness if moving orthogonally to the aligned fitness (i.e., similar to mutating exons).

Third, Fig. 6.4 implies the building block theory to some extent. For example, the $2^{nd} - 3^{rd}$-instruction planes in Fig. 6.4-(a-2) and -(b-2) is symmetrical diagonally approximately (named *diagonal symmetry* of FLs). Because the building blocks likely have similar effectiveness in similar places of LGP programs, performing crossover to share effective building blocks is an important way to produce better offspring.

Insights On Fitness Aligning And Diagonal Symmetry

We can see fitness aligning and diagonal symmetry of FLs in Fig. 6.4. To have a better understanding on these two newly found phenomena, this section makes a further discussion. To show the FL of $\langle$Tmean, 0.85$\rangle$ at the 200^{th} generation in a clearer way, we show the cutting planes of Fig. 6.4-(b-1) in Fig. 6.5. Specifically, we draw the cutting planes at the first, second, and third instruction with an index of $25\%n = 18$, $50\%n = 35$, and $75\%n = 53$, respectively. The indexes of $25\%n = 18$, $50\%n = 35$, and $75\%n = 53$ represent the instruction "$R[0] = \max(OWT, R[0])$", "$R[1] = R[0] + R[0]$", and "$R[0] = \max(PT, OWT)$", respectively.

There are two key issues in the two newly found phenomena. First, fitness aligning holds when the search space extends to higher dimensions. We can see that in the third sub-figure in the first row of Fig. 6.5, instruction 53 ($R[0] = \max(PT, OWT)$) is an ineffective exon (i.e., overwriting the final output R[0] by ineffective results) for DJSS. It forces the first two instructions to be introns and leads programs to be less effective. Thus, the 2-D plane (1^{st}-2^{nd}-instruction) has the same poor fitness of 642 (i.e., white color). The 2-D fitness aligning can also be seen in Fig. 6.5, where some

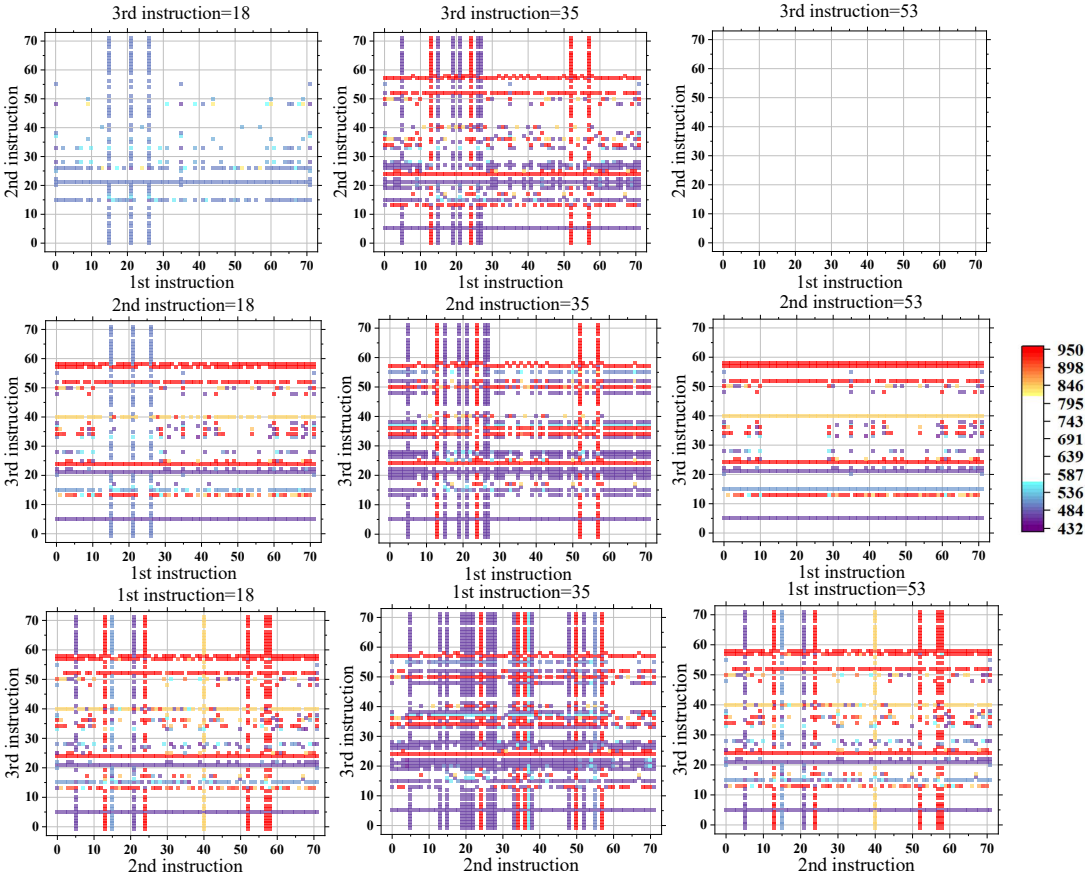

Figure 6.5: The cutting planes of the FL of ⟨Tmean, 0.85⟩ at the 200^{th} generation (i.e., Fig. 6.4-C(b-1)). The 3-D FL is transformed into nine cutting planes by fixing the first, second, and third instruction at an index of 18, 35, and 53, respectively.

2-D fitness planes with the same color go across the cutting planes.

Second, the diagonal symmetry is dependent on two consecutive instructions. In the three rows of Fig. 6.5, the first and third rows have a clear symmetrical layout, while the second row hardly shows diagonal symmetry. Specifically, the second row gets the cutting planes by fixing the second instruction in a program. We believe that because of fixing

Table 6.2: Visualized explanations on (non-)neutral move of instructions and their positions.

Phenomena	Search Operators		Variation on LGP Programs
Fitness aligning	Mutating introns	$\rightsquigarrow$	neutral move on instructions
	Mutating exons	$\rightsquigarrow$	non-neutral move on instructions
Diagonal symmetry	Swapping consecutive instructions	$\rightsquigarrow$	neutral move on instruction positions
	Swapping inconsecutive instructions	$\rightsquigarrow$	non-neutral move on instruction positions

the second instruction, the programs with three instructions (i.e., the programs for $\langle$Tmean, 0.85$\rangle$) are broken into inconsecutive two parts, the first and third instructions. It is likely that swapping inconsecutive instructions greatly affects the effectiveness of programs.

Given the importance of neutral moves in LGP search [21] and the connection between fitness aligning and mutating only one instruction, we highly suspect that the diagonal symmetry implies a missed operator in existing LGP studies, that is, swapping two consecutive instructions. Swapping two consecutive instructions makes use of the diagonal symmetry to perform neutral moves on instruction positions in an LGP solution. Based on the visualized FLs, table 6.2 shows the relationships among the two newly found patterns, the LGP operators, and the (non-)neutral move on instructions (i.e., what instructions should be used) and instruction positions (i.e., where should place the instruction). The "$\rightsquigarrow$" in table 6.2 indicates that the operators perform the neutral and non-neutral moves with a high probability, rather than absolutely.

6.3.2 Test Performance on Common DJSS Problems

The previous section has verified that the proposed method can effectively reduce the hardness of FLs. This section applies LGP to search dispatching rules against the optimized FLs. Specifically, FLO optimizes the FL during LGP evolution, and LGP searches for new solutions based on the up-to-date optimized FLs.

We select four DJSS problems, ⟨Tmax, 0.95⟩, ⟨Tmean, 0.95⟩, ⟨Fmax, 0.95⟩ and ⟨WFmean, 0.95⟩. They are four DJSS problems with different optimization objectives for the overall job shop. The utilization level of 0.95 indicates a much busier and more complex job shop than the utilization level of 0.85.

To verify the effectiveness of the proposed method, there are four compared methods. First, the basic LGP serves as a baseline, denoted as "basicLGP" [21]. The second compared method is a basic LGP method that applies the frequency-based mutation in its evolution (i.e., freqmut). The third compared method is an LGP that applies the operator of swapping consecutive instructions in LGP search. The third method verifies our insight into the neutral move of instruction positions, denoted by "swap". The final compared method is the proposed method which automatically optimizes the FL and applies basic LGP to search against the optimized FL (denoted as LGP-FLO). Specifically, LGP-FLO searches against the optimized FL by moving a symbolic solution toward another better one within the neighborhood. For example, we select two parents by tournament selection. These two parents are represented by index vectors. Then, we produce offspring by moving the index vector of the worse parent toward the index vector of the better parent. In other words, the index vector of the worse parent adds the index vector difference between better and worse parents multiplied by a limited step length. Finally, we construct programs based on the new index vector. LGP-FLO also uses the neutral move on instructions and their positions during the search. The parameters of FLO (i.e., B and σ) are set as 10 and 0.1, respectively. The threshold

Table 6.3: Mean test performance (and standard deviation) on the four DJSS problems. The best mean performance is highlighted in bold.

Problems	basicLGP	freqmut	swap	LGP-FLO
⟨Tmax,0.95⟩	3999.2 (90.9) ≈	3976.3 (190.1) ≈	3969.1 (91.5) ≈	**3967.5 (88.3)**
⟨Tmean,0.95⟩	1118.2 (10.7) ≈	1114.1 (9.1) ≈	**1113.8 (11.4)** ≈	1115 (7.7)
⟨Fmax,0.95⟩	4585.4 (126.1) −	4523.2 (107.3) ≈	4518.4 (70.3) −	**4477.4 (71.4)**
⟨WFmean,0.95⟩	2715.8 (16.4) −	2710.6 (37.7) ≈	**2708.8 (20.6)** ≈	2711.6 (26.4)
mean ranks	4	1.75	2.75	**1.5**
pair-wise p-value	0.037	1.0	1.0	

of neighborhood structures ϵ is approximately 5% of the total number of instructions, that is, 1000 for DJSS problems. The other parameters of the basic LGP follow chapter 3.

Table 6.3 shows the mean test performance (and their standard deviation) over 50 independent runs. The Friedman test on the compared methods shows a p-value of 0.026, which indicates a significant difference in the performance. We further apply the Wilcoxon rank-sum test with a Bonferroni correction and a significance level of 0.05 to analyze the performance of a compared method versus LGP-FLO on each benchmark. The notations of "$+$", "$-$", "$\approx$" have the same meanings as table 6.1.

We can see that by optimizing the FL over the search and searching against the optimized FL, we can significantly improve the performance of LGP methods. Specifically, LGP-FLO has significantly better performance than basic LGP in many benchmarks. Besides, LGP-FLO also has a very competitive performance with the LGP with advanced manually-designed operators (i.e., freqmut). This confirms that searching on the automatically optimized FLs has a very competitive performance with those manually-designed advanced FLs. The mean ranks of the Friedman test also show that LGP-FLO has the best overall performance (i.e., 1.5) amongst the compared methods on the tested benchmark problems. These results imply that the proposed FLO works effectively in much higher dimensions and larger search spaces, given that these benchmarks have a

much larger primitive set and much longer programs in the search spaces.

6.4 Chapter Summary

This chapter shows that there are very likely better FLs than the manually designed ones. To find these landscapes, this chapter proposes an FLO method, which is essentially an optimization problem that optimizes the symbol indexes of symbols to construct better neighborhood structures for symbolic solutions. Specifically, FLO aggregates good solutions, separates good and bad solutions, and encourages the new neighborhood structures to be consistent with the domain knowledge. The experimental studies of FLO draw four main conclusions:

1. The four kinds of FL metrics indicate that the proposed FLO successfully finds significantly more cone-like landscapes than the manually designed landscapes, which reduces the hardness of FLs.

2. The visualization results on LGP landscapes show two important patterns of LGP landscapes. When there are LGP introns in the search space, the solutions with similar fitness likely aggregate to a hyperplane on FL (i.e., fitness aligning), and the hyperplanes spaned by two consecutive LGP instruction positions normally have a diagonally symmetric layout (i.e., diagonal symmetry).

3. The pattern analyses on the visualized FLs further help us find a new operator for LGP, that is, swapping two consecutive instructions, which is missed by existing LGP studies. The results confirm that swapping consecutive instructions implements an essential capability of fine-tuning instruction positions.

4. By simply searching against the optimized FLs, LGP achieves very competitive performance with advanced LGP methods when solving common DJSS problems.

To the best of our knowledge, this work is the first attempt to explicitly optimize FLs of stochastic symbolic search automatically. The proposed FLO method is general enough to apply to other stochastic symbolic search methods once they convert their solutions into a list of symbols. Our experiments use four common FL metrics to evaluate the quality of FLs. This also facilitates further investigations of the correlation of these FL metrics, which is missed by existing FL analysis studies.

Until this chapter, we have focused on optimizing a single DJSS scenario from scratch. However, different DJSS scenarios likely have synergies and share common building blocks. It is interesting to investigate whether these synergies help improve LGPHH performance. To this end, we apply multitask optimization techniques in LGP search in the next chapter, which simultaneously optimizes multiple similar problems to make full use of the problem correlations.

Chapter 7

LGP-based Multitask Optimization for DJSS

7.1 Introduction

Evolutionary multitask optimization is an emerging research area in the last decade [173, 252]. In the light of the outstanding association ability of human brains, evolutionary multitask optimization techniques aim to design evolutionary computation methods that fully exploit the latent synergies among tasks. In contrast to solving every single task independently from scratch, evolutionary multitask optimization techniques solve similar tasks simultaneously and exchange useful information in the course of evolution. Genetic materials such as elite individuals and building blocks are shared among tasks, to enhance convergence speed and search effectiveness. Nowadays, evolutionary multitask techniques have shown remarkable performance on both continuous and discrete optimization problems [71].

Existing literature has applied multitask optimization to enhance tree-based GP in solving machine learning problems and combinatorial optimization and has shown very impressive results [18, 257, 282]. However, conventional tree-based GP is not good at reusing building blocks, since

each node in the tree has at most one parent node. For reusing a sub-tree (building block), tree-based GP has to duplicate that sub-tree or design multiple outputs by complicated tricks [274], which is inefficient in both space and computation and reduces the diversity of genetic materials in the population. On the contrary, the graph characteristics of LGP allow the building blocks of LGP (i.e., sub-graphs) to pass their outputs to multiple graph nodes in the calculation and enable LGP to represent multiple solutions within a single individual naturally, which is very useful in multitask optimization.

However, extending LGP to existing multitask GP methods cannot fully utilize the graph characteristic. The existing multitask GP methods simply see each individual as a solution/heuristic for a specific task and transfer knowledge by duplicating genetic materials (e.g., instruction segments in LGP individuals). Multitask LGP methods have not yet been well investigated.

7.1.1 Chapter Goals

This chapter aims to *propose a new multitask framework based on the graph characteristic of LGP, named **M**ultitask **L**GP with **S**hared **I**ndividuals (MLSI) for DJSS*. MLSI evolves a sub-population of multi-output individuals (i.e., shared individuals). Each shared individual simultaneously encodes more than one solution, each for a specific task and with a specific output, within one directed acyclic graph. These solutions share common building blocks to perform knowledge transfer. The shared individuals then participate in the evolution of all the tasks by shifting their graph outputs, in which way they behave like task-specific individuals, but intrinsically carry common building blocks from the other tasks. Specifically, there are four main goals in this chapter:

1. Develop a new knowledge transfer strategy based on LGP. The proposed strategy fully utilizes the topological structures in LGP by encoding

the solutions from different tasks into a directed acyclic graph.

2. Develop new genetic operators based on the new knowledge transfer strategy. Specifically, we propose a riffle shuffle operator to integrate elite individuals into shared individuals and develop a better-parent reservation strategy to further enhance the effectiveness of the crossover in multitask LGP.

3. Analyze the performance of the proposed method for solving DJSS problems.

4. Verify the reasons for the superior performance of the newly proposed method and the effectiveness of the key components in the proposed method.

7.1.2 Chapter Organization

The rest of this chapter is organized as follows. First, section 7.2 presents the details of the proposed method. Experiment designs including problem formulation and comparison design are illustrated in section 7.3. Section 7.4 and 7.5 show the results and further analyses, respectively. Finally, section 7.6 draws the conclusions.

7.2 Proposed Method

This section demonstrates the proposed MLSI in detail. The chromosome representation and evolutionary framework of MLSI are first described, followed by the selection method. We finally introduce the key new genetic operators in this section.

7.2.1 Program Representation

Fig. 7.1 shows an example of an LGP individual in solving multitask optimization. The registers are initialized by the designated problem features.

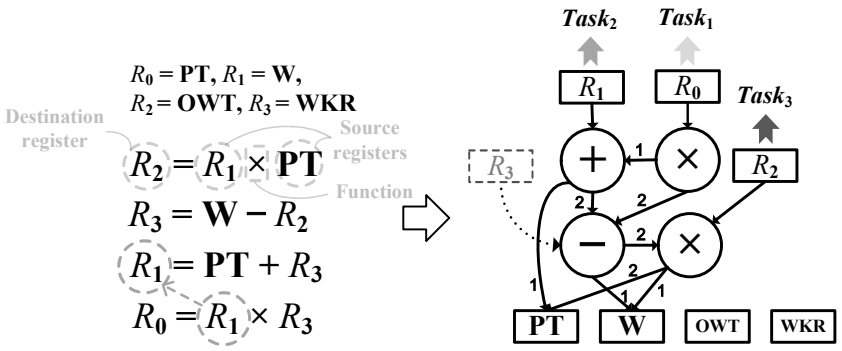

Figure 7.1: An example of LGP individual with three outputs (i.e., R_0, R_1, and R_2). PT: processing time of each operation, W: weight of job, OWT: waiting time of an operation, WKR: remaining processing time of a job.

In Fig. 7.1, R_0 to R_3 are respectively initialized by the input features (i.e., processing time of a job (PT), weight of a job (W), waiting time of an operation (OWT), and the remaining processing time of a job (WKR)). The instructions in the LGP program are executed one by one from the top of the program to the bottom, as single-task LGP individuals. However, to produce the results for different tasks, multiple output registers are defined for LGP. For the sake of simplicity, we designate the first k registers as the output registers for the k tasks (e.g., R_0 to R_2 in Fig. 7.1). We can see that some of the registers are shared among different tasks (i.e., different output registers), such as R_3 in the second instruction. Fig. 7.1 also gives an example of converting LGP instructions into a DAG for multitask optimization. The output of the DAG is stored in output registers R_0, R_1, and R_2. When an LGP individual is shared by multiple tasks, it is evaluated on these tasks respectively and obtains a list of fitness values, each for a specific task.

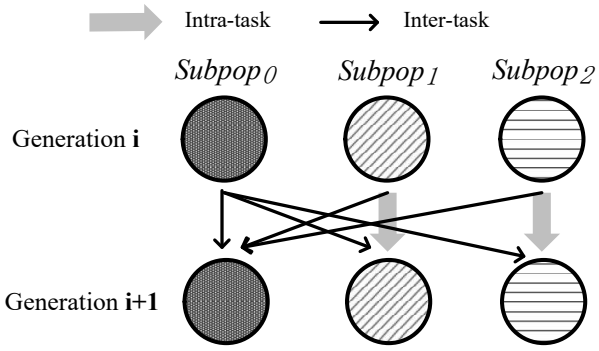

Figure 7.2: The intra- and inter-flow of genetic materials among tasks.

7.2.2 Algorithm Framework

Different from existing multi-population multitask evolutionary frameworks in which each sub-population solves a specific task, the first sub-population $\mathbb{S}_0$ of MLSI is a generalist sub-population that aims to solve all the tasks, while the remaining sub-populations $\mathbb{S}_i(i = 1, ..., k)$ are specialist sub-populations that aim to solve a single task respectively. Fig. 7.2 shows the flow of genetic materials (e.g., building blocks) among tasks. The knowledge of different tasks is shared via $\mathbb{S}_0$. The specialist sub-populations give genetic material to the generalist one and accept knowledge from the generalist sub-population for every generation. Since the individuals in the generalist sub-population $\mathbb{S}_0$ are shared on all tasks, the evolution of the generalist sub-population is seen as another kind of sharing knowledge. By this means, the common building blocks across tasks can be carried in a compact representation in $\mathbb{S}_0$ and flexibly switch representations based on different output registers (i.e., cooperate with task-specific sub-graphs in the individuals).

Each individual $\mathbf{f} \in \mathbb{S}_0$ is evaluated on all the k tasks, and thus has a vector of fitnesses, denoted as $\mathbf{Fit}(\mathbf{f}) = [Fit_1(\mathbf{f}), ..., Fit_k(\mathbf{f})]$. Contrarily, the individuals in $\mathbb{S}_i(i > 0)$ only have a single fitness value for its corresponding task (i.e., $(\mathbf{Fit}(\mathbf{f}) = [Fit_i(\mathbf{f})], i \in \{1, ..., k\})$. Based on

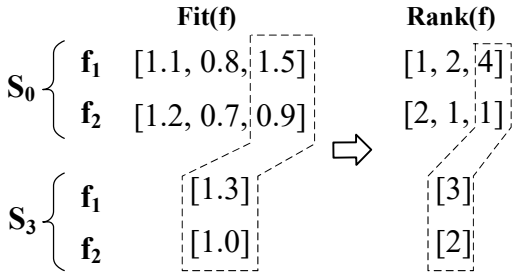

Figure 7.3: An example of determining the rank of the third task (i.e., $Rank_3(\mathbf{f})$) for $\mathbb{S}_0$ and $\mathbb{S}_3$, each sub-population with two individuals.

$\mathbf{Fit}(\mathbf{f})$, the rank of an individual $\mathbf{Rank}(\mathbf{f})$ is designed accordingly. For each generalist individual $\mathbf{f} \in \mathbb{S}_0$, the rank of $\mathbf{f}$ is denoted as $\mathbf{Rank}(\mathbf{f}) = [Rank_1(\mathbf{f}), ..., Rank_k(\mathbf{f})]$. For each specialist individual $\mathbf{f} \in \mathbb{S}_i(i > 0)$, $\mathbf{Rank}(\mathbf{f}) = [Rank_i(\mathbf{f})], i \in \{1, ..., k\}$. To determine the rank of $\mathbf{f}$, i.e., $\mathbf{Rank}(\mathbf{f})$ for solving task t, all the individuals $\mathbf{f} \in \mathbb{S}_0 \bigcup \mathbb{S}_i(t > 0, t = i)$ are combined and sorted together. The example of determining $\mathbf{Rank}(\mathbf{f})$ is shown in Fig. 7.3. There are two individuals in each sub-population. When we are identifying the rank of the third task (i.e., $Rank_3(\mathbf{f})$), those individuals which are evaluated on the third task (i.e., $\mathbf{f} \in \mathbb{S}_0 \bigcup \mathbb{S}_3$) are sorted together based on the corresponding fitness, i.e., $Fit_3(\mathbf{f})$. Smaller fitness has a better rank.

The pseudo-code of MLSI is shown in Alg. 16 where the underlines "$\underset{\sim}{\ }$" highlight the major differences from existing multitask GP methods. To solve k similar tasks simultaneously, MLSI first initializes $k + 1$ sub-populations. At each generation, the individuals in $\mathbf{S}_0$ are evaluated on all the k tasks, and the individuals in $\mathbf{S}_i(i > 0)$ are evaluated on task i respectively. Then, $\mathbf{Fit}(\mathbf{f})$ and $\mathbf{Rank}(\mathbf{f})$ of the individuals are updated based on the new fitness. Elitism selection is applied to retain competent individuals.

New offspring are produced by five genetic operators, including macro mutation, micro mutation, crossover, reproduction, and *riffle shuffle* based

Algorithm 16: Framework of MLSI

Input: k tasks, macro mutation rate θ_{ma}, micro mutation rate θ_{mi}, crossover rate θ_c, reproduction rate θ_r, step size of `RiffleShuffle` η, tournament selection size s.

Output: k best heuristics $h_t(t = 1, ..., k)$, each for a specific task.

Initialize $k + 1$ sub-populations $\mathbb{S}_0$ to $\mathbb{S}_k$.
while *stopping criteria are not satisfied* **do**
  `// Evaluation`
  Evaluate $\mathbf{f} \in \mathbb{S}_0$ on all the tasks. Evaluate $\mathbf{f} \in \mathbb{S}_i(i = 1, .., k)$ on task i respectively;
  Update $\mathbf{Fit}(\mathbf{f})$ and $\mathbf{Rank}(\mathbf{f})$ for $\mathbf{f} \in \{\mathbb{S}_0, ..., \mathbb{S}_k\}$;
  `// Breeding`
  foreach $\mathbb{S}_i(i = 0, ..., k)$ **do**
    $\mathbb{S}'_i \leftarrow \emptyset$;
    Clone elite individuals of $\mathbb{S}_i$ into $\mathbb{S}'_i$;
    while $|\mathbb{S}'_i| < |\mathbb{S}_i|$ **do**
      $rnd \leftarrow$ `Uniform`$(0, 1)$;
      if $rnd < \theta_r$ **then**
        $\mathbf{c} \leftarrow$ `TournamentSpecific`$(\mathbb{S}_i, i, s)$;
      else if $rnd < \theta_r + \theta_c$ **then**
        $\mathbf{c} \leftarrow$ LGP crossover with `BetterParentRes`$(\{\mathbb{S}_0, ..., \mathbb{S}_k\}, i, s)$;
      else
        `// generalist sub-population`
        if $i = 0$ **then**
          $\mathbf{c} \leftarrow$ `RiffleShuffle`$(\{\mathbb{S}_0, ..., \mathbb{S}_k\}, \eta, s)$;
        `// specialist sub-population`
        else
          $\mathbf{p} \leftarrow$ `TournamentSpecific`$(\mathbb{S}_i, i, s)$;
          if $rnd - (\theta_c + \theta_r) < \theta_{ma}$ **then**
            Apply LGP macro mutation on $\mathbf{p}$ to produce offspring $\mathbf{c}$;
          else
            Apply LGP micro mutation on $\mathbf{p}$ to produce offspring $\mathbf{c}$;
      if $\mathbf{c}$ *is produced by macro operators* **then**
        Apply LGP micro mutation on $\mathbf{c}$ to update it;
      $\mathbb{S}'_i \leftarrow \mathbb{S}'_i \bigcup \{\mathbf{c}\}$;
    $\mathbb{S}_i \leftarrow \mathbb{S}'_i$;
  Update the best heuristics $h_t(t = 1, .., k)$ for the k tasks;
Return $h_t(t = 1, .., k)$.

on their rates. The riffle shuffle is the newly proposed genetic operator in this chapter to vary the individuals with multiple outputs, while the other four genetic operators are off-the-shelf operators [21]. Specifically, *effective macro mutation*, *effective micro mutation*, and *linear crossover* in [21] act as mutation and crossover respectively. Linear crossover is enhanced by a newly proposed better-parent reservation strategy (i.e., `BetterParentRes`$(\cdot)$). Given that each individual in $\mathbf{S}_0$ has a list of fitnesses, a new tournament selection `TournamentSpecific`$(\cdot)$ is developed to select parents based on a specific task. More specifically, $\mathbf{S}_0$ applies reproduction, linear crossover, and the riffle shuffle (i.e., `RiffleShuffle`$(\cdot)$) to produce offspring, while $\mathbf{S}_i(i > 0)$ apply reproduction, linear crossover, and mutation to produce offspring. To share knowledge among tasks, the parents of the linear crossover and riffle shuffle are selected from a merged sub-population, formed by $\mathbb{S}_0$ and specialist sub-populations. As suggested by [21], all macro operators (i.e., crossover, riffle shuffle, and macro mutation) are followed by a micro mutation. All the produced offspring replace the parents without further comparison and form the population of the next generation. This evolutionary process is iterated for every generation until the stopping criteria are met. The best individuals of the tasks in all the sub-populations are outputted as the final results.

7.2.3 Selection

The parent selection is implemented by the `TournamentSpecific`$(\cdot)$ method. This method is extended from the standard tournament selection by enabling cross-sub-population selection for a specific task. The pseudo-code of `TournamentSpecific`$(\cdot)$ is shown in Alg. 17, in which $\mathbf{Fit}(\mathbf{f}, t)$ returns the t^{th} element of $\mathbf{Fit}(\mathbf{f})$ if $|\mathbf{Fit}(\mathbf{f})| > 1$, and the only element in $\mathbf{Fit}(\mathbf{f})$ otherwise[1].

Note that in the proposed multitask framework, the parents of linear

[1] same as $\mathbf{Rank}(\mathbf{f}, t)$ in Alg. 19.

Algorithm 17: `TournamentSpecific`

Input: A merged sub-population $\mathbb{S}_{meg}$, task index t, tournament size s
Output: An LGP individual $\mathbf{f}$

```
Randomly select an LGP individual f' from S_meg;
f ← f';
Fit ← Fit(f, t);
for j ← 1 to s − 1 do
    Randomly select an LGP individual f' from S_meg;
    Fit' ← Fit(f', t);
    if Fit' is better than Fit then
        f ← f', Fit = Fit';
Return f;
```

crossover and riffle shuffle for a certain task t are selected from the generalist sub-population (i.e., $\mathbb{S}_0$) and specialist sub-population (i.e., $\mathbb{S}_t$) simultaneously. Specifically, $\mathbb{S}_0$ and $\mathbb{S}_t$ are merged into one sub-population, and the two parents for one mating are selected from the merged sub-population based on the paradigm of tournament selection. Therefore, the parents for linear crossover and riffle shuffle can both come from $\mathbb{S}_0$, or both come from $\mathbb{S}_t$, or come from either $\mathbb{S}_0$ or $\mathbb{S}_t$. The knowledge sharing among different tasks for the linear crossover and riffle shuffle is achieved by selecting the parents from $\mathbb{S}_0$ whose individuals are shared by all the tasks. The parents for task t are selected based on their fitness on that task. When the individuals in $\mathbb{S}_0$ have better fitness on task t than those in $\mathbb{S}_t$, they have a higher probability to be selected by `TournamentSpecific`$(\cdot)$, and vice versa. Such design helps MLSI determine the suitable timing and frequency of knowledge transfer across tasks (i.e., transfer knowledge when the individuals with common building blocks have better performance).

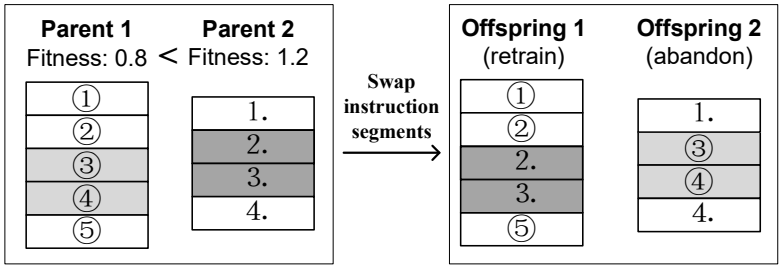

Figure 7.4: An example of linear crossover with better-parent reservation strategy. Smaller fitness is better.

7.2.4 Breeding Offspring

To improve the effectiveness of sharing knowledge, the better-parent reservation is proposed to enhance the crossover operator. The riffle shuffle is proposed to effectively integrate knowledge from different tasks.

Linear Crossover with Better-parent Reservation

Multitask optimization framework promotes effectiveness by sharing knowledge among similar tasks. However, different tasks may have different specialist knowledge. The crossover operator should not only share useful common knowledge among different sub-populations but also retain useful specialist building blocks for each specific task. It is reasonable to assume that the parent with better fitness on a certain task is more likely to have more useful building blocks for that task. Based on this assumption, the crossover operator in this work only retains the child whose corresponding parent (i.e., the parent accepting a new genome to form the offspring) has better fitness. An example of the better-parent reservation is shown in Fig. 7.4 in which the to-be-swapped instructions are highlighted as dark. Since *Parent 1* has better (i.e., smaller) fitness than *Parent 2*, only *offspring 1* which is generated from *Parent 1* by accepting new instructions is retained. The pseudo-code of the linear crossover with better-parent

Algorithm 18: Linear crossover with `BetterParentRes`$(\cdot)$

Input: Set of sub-populations $\{\mathbb{S}_0, ..., \mathbb{S}_k\}$, index of current sub-population i, tournament selection size s

Output: A new offspring $\mathbf{c}$.

```
1  t = i;
2  if i = 0 then
3      t ← UniformInt(1, k);
4  S_meg ← S_0 ⋃ S_t
5  Apply TournamentSpecific(S_meg, t, s) to select p_a and p_b respectively;
6  Apply linear crossover on p_a and p_b to produce two offspring c_a and c_b;
   // Better-parent reservation
7  c ← c_a;
8  if p_b is better than p_a then
9      c ← c_b;
10 Return c;
```

reservation is shown as Alg. 18, in which the linear crossover is followed by the better-parent reservation based on the fitness of the parents.

Riffle Shuffle

Riffle shuffle is a technical term for playing cards. When there are two decks of cards, the riffle shuffle integrates them into a single deck by alternatively interleaving the two decks of cards and maintaining the relative order of cards within every deck. This concept is extended to our work to integrate individuals from different tasks into a new offspring. By this means, heuristics for different tasks cannot only maintain the relative order of their instructions, but also they can use the building blocks from other tasks.

The pseudo-code of the proposed riffle shuffle operator is shown in Alg. 19. First, the riffle shuffle operator samples several LGP parents from different tasks. Then, the effective instructions (i.e., exons) are extracted based on their output registers (lines 5 to 13) (The extraction method of

effective instructions can be referred to [21]). The extracted effective instructions for different tasks are stored in lists respectively. The task index whose sampled parent has the best rank is recorded for later use. For lines 14 to 19, the riffle shuffle operator alternatively interleaves instructions from the lists to form a new instruction sequence. To prevent LGP programs from increasing size too rapidly, a maximum step size η is defined to limit the largest variation of program size. Specifically, the new program size L of the offspring is defined by aggregating the average program size of parents and a random variation size based on η (lines 20 to 21). L is also limited by the maximum and minimum program size of LGP individuals (L_{max} and L_{min}). For lines 22 to 27, when the actual program size after merging exceeds L, instructions are randomly removed until the actual program size is consistent with L. To protect the useful building blocks, the instructions from the parent with a better rank have a higher priority to be kept in the offspring. Specifically, the instructions from the parent with a better rank are extracted based on the output register R_{t^*-1} (line 24). When the program size of the parent with a better rank is smaller than L and the offspring still contains the instructions from other tasks (i.e., $|\mathbf{G}| < |\mathbf{c}|$), the instructions of better parent are protected from being removed (lines 25 and 26).

Fig. 7.5 shows an example of the riffle shuffle. The two LGP parents come from two different tasks and have three and two effective instructions respectively. They first extract the effective instructions, which are the second, fourth, and fifth instructions of the first parent, and the first and second instructions of the second parent. Then, the two sequences of effective instructions are integrated into one sequence alternatively to form an offspring.

Both better-parent reservation and riffle shuffle can be applied to other methods. Specifically, better-parent reservation strategy can be applied together with other crossover operators that accept more than one parent. Riffle shuffle is an LGP-based genetic operator for transferring genetic ma-

Algorithm 19: `RiffleShuffle`

Input: Set of sub-populations $\{\mathbb{S}_0, ..., \mathbb{S}_k\}$, step size η, tournament selection size s
Output: A new offspring $\mathbf{c}$.

$n \leftarrow$ `UniformInt`$(2, k)$;
$\mathbb{T}, \mathbb{P}, \mathbb{G}, \mathbf{c} \leftarrow \emptyset$;
Randomly select n unique task indices to initialize $\mathbb{T}$;
$t^*, Rank^* \leftarrow +\infty$
foreach *t in* $\mathbb{T}$ **do**
    $\mathbf{p}' \leftarrow$ `TournamentSpecific`$(\mathbb{S}_0 \bigcup \mathbb{S}_t, t, s)$;
    $\mathbb{P} \leftarrow \mathbb{P} \bigcup \mathbf{p}'$;
    $\mathbf{G} \leftarrow$ `IdentifyExtrons`$(\mathbf{p}', \{R_{t-1}\})$;
    if $\mathbf{G} \neq \emptyset$ **then**
        $\mathbb{G} \leftarrow \mathbb{G} \bigcup \mathbf{G}$;
    if $\mathbf{Rank}(\mathbf{p}', t) < Rank^*$ **then**
        $Rank^* \leftarrow \mathbf{Rank}(\mathbf{p}', t)$;
        $t^* \leftarrow t$;
`// merge into one individual`
while $\mathbb{G} \neq \emptyset$ **do**
    $\mathbf{G} \leftarrow$ Randomly select an element from $\mathbb{G}$;
    $f \leftarrow$ get and remove the first element from $\mathbf{G}$;
    Append f to $\mathbf{c}$;
    if $\mathbf{G} = \emptyset$ **then**
        remove $\mathbf{G}$ from $\mathbb{G}$;
`// randomly remove instructions`
$L \leftarrow \frac{\Sigma_{\mathbf{p} \in \mathbb{P}} |\mathbf{p}|}{n} +$ `UniformInt`$(-\eta, \eta)$;
$L \leftarrow l_{min}$ if $L < l_{min}$ (or $L \leftarrow l_{max}$ if $L > l_{max}$);
while $|\mathbf{c}| > L$ **do**
    $f_{rmv} \leftarrow$ randomly select an instruction from $\mathbf{c}$;
    $\mathbf{G} \leftarrow$ `IdentifyExtrons`$(\mathbf{c}, R_{t^*-1})$;
    while $|\mathbf{G}| < L$ *and* $|\mathbf{G}| < |\mathbf{c}|$ *and* $f_{rmv} \in \mathbf{G}$ **do**
        $f_{rmv} \leftarrow$ randomly select an instruction from $\mathbf{c}$;
    Remove f_{rmv} from $\mathbf{c}$;
Return $\mathbf{c}$;

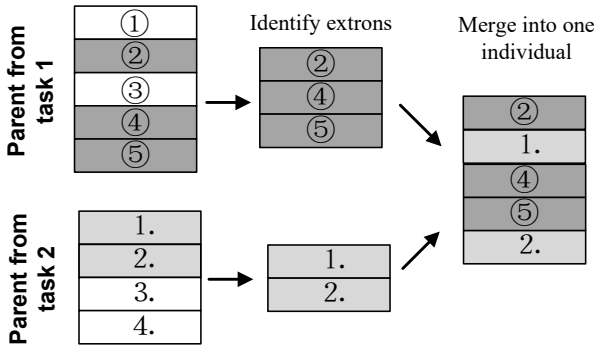

Figure 7.5: An example of the riffle shuffle operator for LGP individuals.

terials. Further, riffle shuffle is designed for merging building blocks from different tasks. Applying riffle shuffle to evolve a single task might produce many redundant instructions since the building blocks from a single task are often too similar. The effectiveness of the two operators and their collaboration with other methods are investigated in section 7.5.1.

7.3 Experiment Design

DJSS often has many similar optimization problems. For example, the peak and off seasons of the same production line might have similar scheduling methods, and the customers might share similar requirements with the manager on the production line. Fully utilizing the common knowledge among these tasks (i.e., multitask optimization) is more efficient than searching from scratch for every single task.

7.3.1 Multitask Scenarios

In multitask optimization of DJSS, six multitask scenarios are designed for verifying the performance of the proposed method according to the above objectives [257]. We regard the tasks with similar settings of the simulation, including the number of machines, the number of operations, and the

Table 7.1: The setting of multitask scenarios represented by optimized objectives and utilization levels.

Scenarios	#tasks (k)	Specific settings of tasks
A	3	$<$ Fmean, $0.95>$,$<$ Fmean, $0.85>$,$<$ Fmean, $0.75>$
B	3	$<$ Tmean, $0.95>$,$<$ Tmean, $0.85>$,$<$ Tmean, $0.75>$
C	3	$<$ WTmean, $0.95>$,$<$ WTmean, $0.85>$,$<$ WTmean, $0.75>$
D	2	$<$ Fmax, $0.95>$,$<$ Tmax, $0.95>$
E	2	$<$ WFmean, $0.95>$,$<$ WTmean, $0.95>$
F	3	$<$ Fmean, $0.95>$,$<$ Tmean, $0.85>$,$<$ WTmean, $0.75>$

processing time of operations, as similar tasks. In these multitask scenarios, each specific optimization problem with a specific optimized objective and a specific utilization level (denoted as ρ) of the simulation is defined as a task, denoted as "$<$ objective, $\rho>$". The settings of the six multitask scenarios are shown in table 7.1. To have a comprehensive investigation, Scenarios A to F cover a wide range of tasks, with respect to the optimized objective and utilization level. It should be noted that the six scenarios cover different job sets which are specified by the utilization level. For example, Sce. A, B, C, and F have the same settings of utilization level for the three tasks (i.e., 0.95, 0.85, and 0.75) and thus have the same job sets with each other. Sce. D and E have different job sets from Sce. A, B, C, and F. Further, the three tasks in Sce. A are configured with different job sets since each task is set with a different utilization level. Other parameters in the simulation are the same as the previous chapters (refer to chapter 3).

7.3.2 Comparison Design

We select four compared methods to verify the performance of MLSI. First, multi-population LGP (MPLGP) is included as the baseline method. It solves multiple tasks simultaneously, each by a sub-population. But these sub-populations do not exchange any information and solve tasks independently. Second, one of the state-of-the-art multitask tree-based GP

methods, M^{2}GP [257], is compared in the experiment. M^{2}GP is a recently proposed method for solving multitask dynamic scheduling. In addition, to comprehensively verify the effectiveness of the proposed mechanism (i.e., transfer knowledge via shared individuals), we directly apply two recent multitask optimization frameworks to LGP, denoted as M^{2}LGP and MFLGP respectively. Specifically, M^{2}LGP replaces the tree-based GP individuals in M^{2}GP [257] with LGP individuals, and MFLGP replaces the individual representation and genetic operators in MFEA [71] (a popular framework for multitask optimization) with that of LGP.

The parameters of the compared methods are designed based on their original paper [257] or chapter 3. All the compared methods have the same total number of simulations to ensure fairness. For MLSI, `TournamentSpecific`$(\cdot)$ replaces the conventional tournament selection. M^{2}GP, which is designed based on tree-based GP, mainly breeds offspring by crossover, mutation, and reproduction. The rates of these three operators are 80%, 15%, and 5% respectively [257]. On the other hand, the LGP-based methods, including MPLGP, M^{2}LGP, MFLGP, and MLSI, employ macro mutation, micro mutation, linear crossover, and reproduction with rates of $\theta_{ma} : \theta_{mi} : \theta_c : \theta_r = 30\% : 30\% : 35\% : 5\%$ to produce offspring. MLSI enhances the linear crossover by the better-parent reservation, and the generalist sub-population in MLSI replaces mutation operators by riffle shuffle.

The transfer rate of M^{2}GP is set as 0.3 as suggested in [257]. Given that their knowledge transfer is mainly implemented based on the crossover, actually $80\% \times 0.3 = 24\%$ of crossover mate parents from different sub-populations. To keep the rate of knowledge transfer the same, 68.6% of linear crossover ($35\% \times 68.6\% = 24\%$) in M^{2}LGP and MFLGP produce offspring based on parents from different sub-populations. Since MLSI selects parents from a merged population, the parameter of transfer rate is not needed for MLSI. Contrarily, MLSI introduces two new parameters, the population size of the generalist sub-population and the step size of

Table 7.2: Parameters of all compared methods.

Parameters	M^{2}GP	MPLGP	M^{2}LGP	MFLGP	MLSI
number of sub-population	k	k	k	1	k+1
(sub) population size	400	100	100	k*100	generalist:30, specialist:70
generations	50	200	200	200	200
elitism selection size for each sub-population	10	3	3	6	3
tournament selection size	5	7	7	7	7
crossover parameters	inner node 90%, leaf node 10%	segment length$\leq$30, segment length difference$\leq$5, crossover point distance$\leq$30			
mutation parameters	inner node 90%, leaf node 10%	macro(insertion 67%, deletion 33%), micro (function 50%, destination register 25%, source register 12.5%, constant 12.5%)			
initial program size	min depth=2, max depth=6	min instruction=1, max instruction=10			
maximum program size	max depth=8	max instruction=50			

riffle shuffle η. Without loss of generality, the size of the generalist sub-population is defined as 30 to approximate the transfer rate of 0.3. η is defined as 15 by default. The parameters are summarized in table 7.2. In the training phase, all the compared methods are trained by k DJSS instances for every generation, each DJSS instance for a specific task. The random seeds of instances are rotated every generation.

7.4 Experiment Results

In this section, we compare the training and test performance of all the compared methods. All the multitask compared methods output the best

solution for each task for comparison. We analyze the objective values on test instances and training convergence curves. Friedman test and Wilcoxon test with a significance level of 0.05 are also applied to make a comprehensive analysis.

7.4.1 Test Performance

Fig. 7.6 shows the test performance of all the compared methods on different multitask scenarios. We can see that the results of MLSI have relatively small objective values in most tasks. For example, in the complex tasks of Scenarios A, B, and D (e.g., A-<Fmean,0.95>, B-<Tmean,0.95>, and D-<Fmax,0.95>), MLSI has better medians and averages of test performance than the other compared methods over 50 independent runs. In other tasks such as C-<WTmean,0.95> and F-<Fmean,0.95>, although we cannot see significant differences between MLSI and the other compared methods, we can confirm that the medians and averages of MLSI test performance are similar to the best medians and best averages in the tested problems.

To have a more comprehensive comparison, we analyze the test performance by the Friedman test, as shown in table 7.3. The p-value of the Friedman is 1.14E^{-4}, which is much smaller than 0.05. It implies that there is a significant difference among these compared methods. Besides, the mean rank from the Friedman test shows that MLSI has the best average ranking on these tasks.

Based on the Friedman test, post-hoc analyses by the Wilcoxon rank sum test with Bonferroni correction and a significance level of 0.05 are conducted to further analyze the performance on different tasks. In table 7.3, "$+$" denotes that a method is significantly better than MLSI, "$-$" denotes that a method is significantly worse than MLSI on a certain task, and "$\approx$" denotes competitive performance with MLSI. It can be seen that the four compared methods are significantly inferior to MLSI on most tasks. Specif-

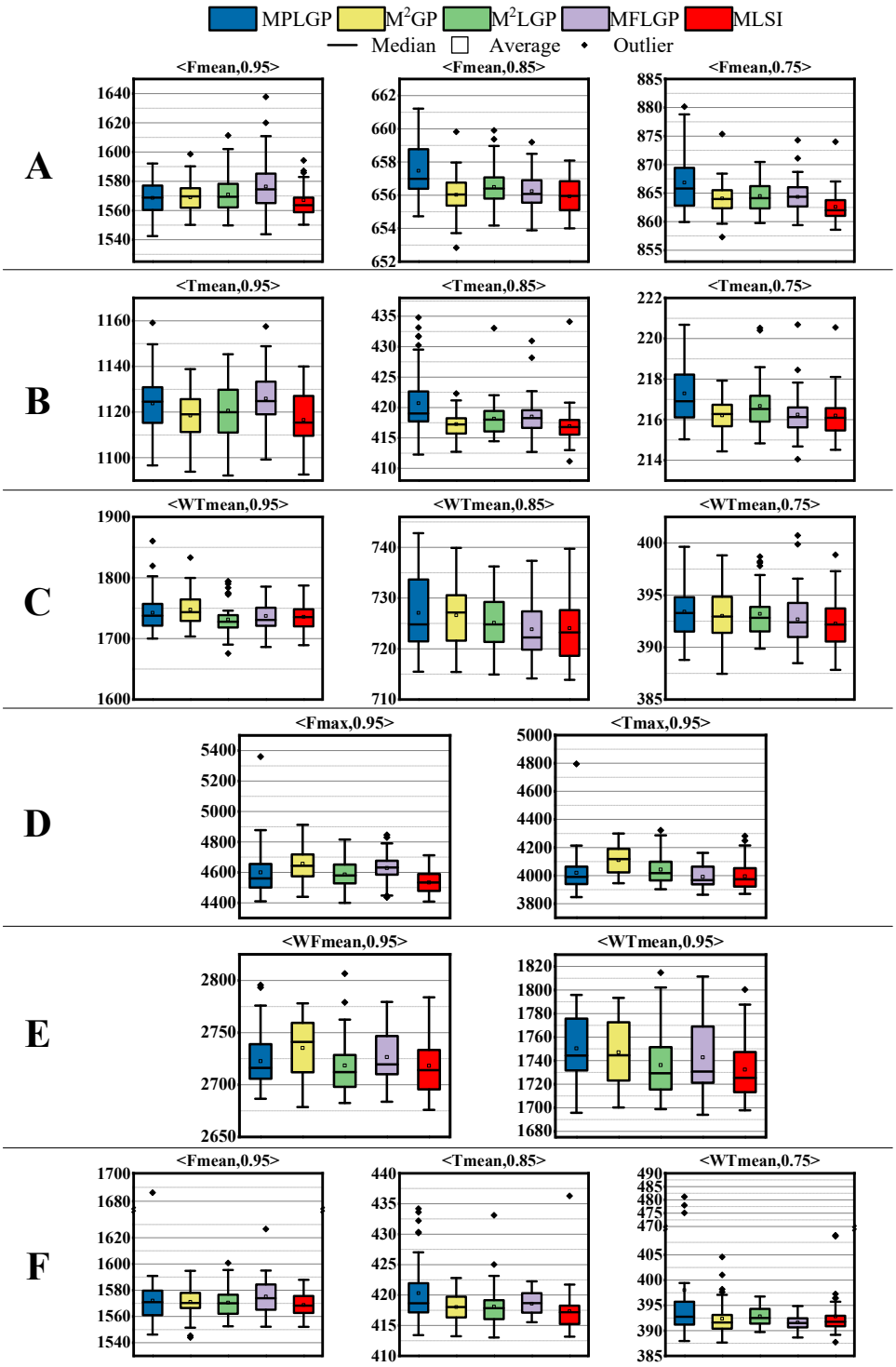

Figure 7.6: The box plots of test performance on all tasks in different multitask scenarios.

Table 7.3: The significance analysis by Friedman test and Wilcoxon test with Bonferroni test (vs. MLSI) with significance level of 0.05.

Scenarios	Tasks	MPLGP	M^{2}GP	M^{2}LGP	MFLGP	MLSI
A	<Fmean,0.95>	$\approx$	$\approx$	$-$	$-$	
	<Fmean,0.85>	$-$	$-$	$-$	$-$	
	<Fmean,0.75>	$-$	$\approx$	$-$	$\approx$	
B	<Tmean,0.95>	$-$	$\approx$	$\approx$	$-$	
	<Tmean,0.85>	$-$	$\approx$	$-$	$-$	
	<Tmean,0.75>	$-$	$\approx$	$-$	$\approx$	
C	<WTmean,0.95>	$\approx$	$-$	$\approx$	$\approx$	
	<WTmean,0.85>	$-$	$-$	$\approx$	$\approx$	
	<WTmean,0.75>	$-$	$\approx$	$\approx$	$\approx$	
D	<Fmax,0.95>	$-$	$-$	$-$	$-$	
	<Tmax,0.95>	$\approx$	$-$	$-$	$\approx$	
E	<WFmean,0.95>	$\approx$	$-$	$\approx$	$\approx$	
	<WTmean,0.95>	$-$	$-$	$\approx$	$\approx$	
F	<Fmean,0.95>	$\approx$	$\approx$	$\approx$	$-$	
	<Tmean,0.85>	$-$	$-$	$\approx$	$-$	
	<WTmean,0.75>	$\approx$	$\approx$	$\approx$	$\approx$	
win-draw-lose		0-6-10	0-8-8	0-9-7	0-9-7	
p-value with Bonferroni correction		$1.53E^{-4}$	$8.24E^{-4}$	0.015	0.012	
mean rank		3.78	3.56	3.13	3.16	**1.38**

ically, MLSI is significantly better than the three state-of-the-art methods (i.e., M^{2}GP, M^{2}LGP, MFLGP) in nearly half of the tasks. The p-values with Bonferroni correction also validate that MLSI has a significantly better overall performance than the others since all of them are much smaller than 0.05. These results verify that the newly proposed multitask mechanism is very effective in improving the performance of multitask LGP.

7.4.2 Training Efficiency

To validate the training efficiency of MLSI, we compare the training convergence. Fig. 7.7 shows the test objectives of all compared methods over evaluation times. We select one of the most complex tasks (i.e., those with a utilization level of 0.95) in each scenario for investigation. It can be seen that MLSI converges faster and lower than the others overall. Specifically, MLSI achieves better performance than the other compared methods over the whole training processing in A-<Fmean,0.95>, B-<Tmean,0.95>, D-<Fmax,0.95> and F-<Fmean,0.95>. Although MLSI performs similarly to other compared methods at the final stage of training in C-<WTmean,0.95> and E-<WFmean,0.95>, MLSI has a faster training efficiency at the early stage of training (e.g., before 10000 simulations) in E-<WFmean,0.95>. The results imply that MLSI has better efficiency in designing effective dispatching rules.

7.5 Further Analyses

In this section, we make further analyses to answer the following research questions:

- What is the effectiveness of the newly proposed riffle shuffle and the better-parent reservation strategy?
- How does performance change with the key parameters of MLSI (i.e., the population size of the generalist population and the step size of riffle shuffle η)?
- How does MLSI control the rate of knowledge transfer?
- What do MLSI individuals look like?

These questions are answered respectively as follows.

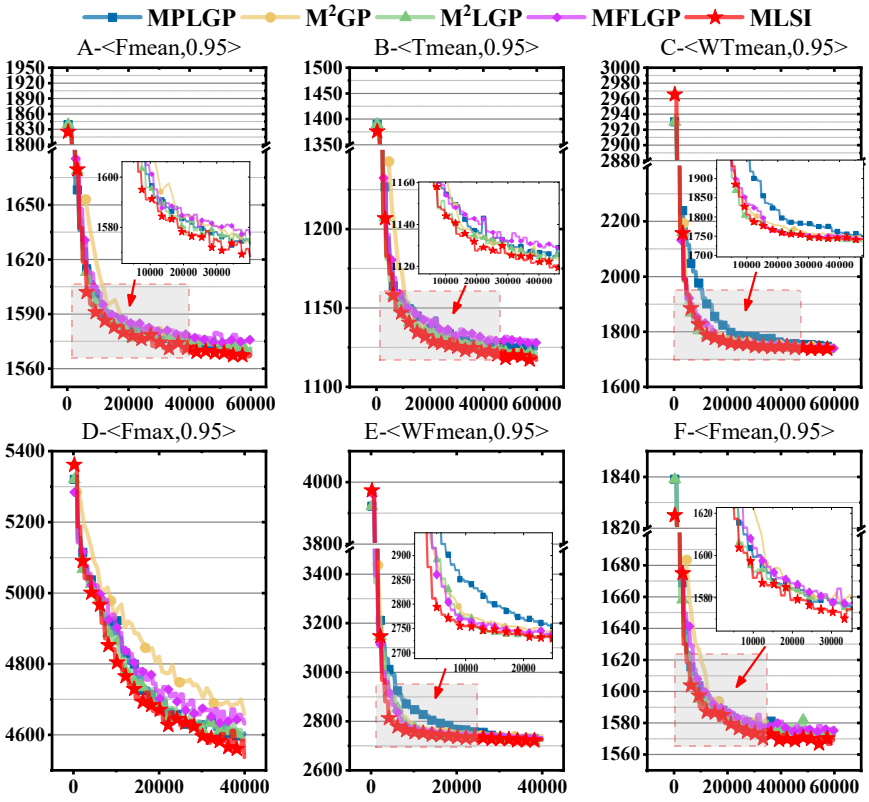

Figure 7.7: Test objectives during evolution. X-axis:evaluation times, Y-axis:objective values.

7.5.1 Component Analysis

To investigate the effectiveness of each key component of MLSI, this section conducts a component analysis. First, to verify the effectiveness of riffle shuffle, an MLSI variant without riffle shuffle, denoted as "MLSI/RS" is developed. Second, to verify the effectiveness of the better-parent reservation, we replace the crossover with the better-parent reservation with conventional crossover, which retains both of the two offspring after mating. This variant of MLSI is denoted as "MLSI/BPR". Further, to verify the performance gain from genetic transfer rather than genetic operator

Table 7.4: Mean (std.) test performance of MLSI with different components and MPLGP+. The best mean values are highlighted by **bold** font.

Tasks	MPLGP+	MLSI/RS	MLSI/BPR	MLSI
A-<Fmean,0.95>	1584.6 (21.4) −	1570 (11.27) ≈	1570.9 (10.85)−	**1567.1** (16.51)
A-<Fmean,0.85>	1118.4 (192.7)−	863.4 (2.67) ≈	863.4 (3.62) ≈	**862.6** (2.53)
A-<Fmean,0.75>	761.3 (100.3) −	656 (1.01) ≈	656.2 (1.25) ≈	**655.9** (1.02)
B-<Tmean,0.95>	1133.8 (23.3) −	1123.3 (18.82) −	1123.9 (13.42)−	**1116.6** (11.25)
B-<Tmean,0.85>	691.2 (164.8) −	417.2 (2.53) ≈	417.2 (2.28) ≈	**417** (3.14)
B-<Tmean,0.75>	332.4 (141.9) −	**216** (1.12) ≈	216.2 (1.12) ≈	216.2 (1.06)
C-<WTmean,0.95>	1925.8 (302.3)−	1743.3 (22.13) ≈	**1732.3** (24.65)≈	1735.8 (23.08)
C-<WTmean,0.85>	1670.2 (691.2)−	726.4 (5.34) −	**724** (5.51) ≈	**724** (6.05)
C-<WTmean,0.75>	682.5 (192.5) −	393.2 (2.8) ≈	**392.3** (1.95) ≈	**392.3** (2.41)
D-<Fmax,0.95>	4628.9 (117.6)−	4624.5 (208.65)−	4559.2 (96.6) ≈	**4533.5** (75.13)
D-<Tmax,0.95>	1.6E4 (23142) −	4024.3 (102.35)≈	**3968** (73.01) ≈	3994.7 (92.4)
E-<WFmean,0.95>	2856.4 (233.2)−	2722.8 (25.9) ≈	**2713.5** (21.78)≈	2718.2 (26.33)
E-<WTmean,0.95>	4799.1 (1517) −	1734.2 (24.55) ≈	**1727.1** (26.63)≈	1732.4 (25.78)
F-<Fmean,0.95>	1587.3 (16.5) −	1569.7 (11.88) ≈	**1568** (11.26) ≈	1568.8 (9.34)
F-<Tmean,0.85>	671.5 (180.1) −	418.1 (2.45) −	418 (2.62) −	**417.3** (3.3)
F-<WTmean,0.75>	786.6 (211.5) −	394.1 (2.65) −	393 (2.66) ≈	**392.8** (3.86)
win-draw-lose	0-0-16	0-11-5	0-13-3	
mean rank	3.94	2.75	1.81	**1.50**
p-value	5.0E-07	0.03	0.90	

bias, we apply riffle shuffle and better-parent reservation to MPLGP, but has no knowledge transfer among tasks, denoted as "MPLGP+". Other settings of these two compared methods are the same as MLSI by default. The comparative results are shown in table 7.4.

First, table 7.4 shows that simply applying the riffle shuffle and better-parent reservation to MPLGP is harmful to MPLGP performance. MPLGP+ performs much worse than MLSI on all the tasks. Since riffle shuffle is specialized for merging various building blocks from different tasks, the similar building blocks in the parents from a single task might be unnecessarily duplicated by riffle shuffle, which produces a large number of redundant instructions in offspring. The results further verify that the performance

gain of MLSI is due to genetic transfer. Second, removing riffle shuffle from MLSI reduces the effectiveness since MLSI/RS is significantly worse than MLSI on five tasks and has a significantly worse (higher) mean rank than MLSI based on the Friedman rank sum test with the Bonferroni correction. The results imply that the riffle shuffle operator is prominent for MLSI. Third, MLSI without the better-parent reservation performs similarly to MLSI on average, implying that selectively retaining offspring based on superior parents is not so crucial for MLSI.

7.5.2 Parameter Sensitivity Analysis

There are two new parameters in MLSI, which are the population size of the generalist sub-population and step size η of the riffle shuffle. To investigate the parameter sensitivity, we compare the performance of MLSI with different parameter settings. Four different MLSI versions are developed. Specifically, "MLSI-pop15", "MLSI-pop20", and "MLSI-pop45" set the population size of the shared population as 15, 20, and 45 respectively. "MLSI-η5", "MLSI-η10" and "MLSI-η20" set η as 5, 10 and 20 respectively. When the population size of the shared population is changed, to be fair, the population size of other specialized sub-populations is updated accordingly (e.g., MLSI-pop15 has specialized sub-populations with 85 individuals). Other parameters of the four methods are set the same as MLSI.

The test performance of different settings is shown in table 7.5. The results show that the performance of all different settings is quite similar to the default ones in most cases based on the Wilcoxon rank sum test. Though significant differences can be seen on a few tasks, the mean rank by the Friedman test and the p-values with Bonferroni correction verify that different parameters have no significant difference in terms of overall performance on these tasks. The results verify the performance of MLSI is robust to the two newly introduced parameters.

Table 7.5: Mean (std.) test performance of MLSI with different parameter settings.

Tasks	MLSI-pop15		MLSI-pop20		MLSI-pop45		MLSI-η5		MLSI-η10		MLSI-η20		MLSI
A-<Fmean,0.95>	1565.8 (11.6)	$\approx$	1569.7 (12.25)	$-$	1569.1 (11.61)	$\approx$	**1564.4** (8.61)	$\approx$	1570.7 (11.3)	$-$	1570.3 (17.2)	$\approx$	1567.1 (16.51)
A-<Fmean,0.85>	**862.6** (2.27)	$\approx$	863.5 (2.48)	$-$	863.3 (2.41)	$-$	863 (2.71)	$\approx$	863.5 (1.84)	$-$	862.9 (2.42)	$\approx$	**862.6** (2.53)
A-<Fmean,0.75>	656.2 (1)	$\approx$	656 (0.96)	$\approx$	655.9 (1.12)	$\approx$	655.8 (1.08)	$\approx$	656.1 (0.97)	$\approx$	**655.7** (1.12)	$\approx$	655.9 (1.02)
B-<Tmean,0.95>	**1115.9** (12.63)	$\approx$	1119.5 (19.76)	$\approx$	1121.3 (11.19)	$-$	1117.9 (10.44)	$\approx$	1120.7 (13.14)	$\approx$	1120.9 (11.47)	$\approx$	1116.6 (11.25)
B-<Tmean,0.85>	**416.5** (2.29)	$\approx$	416.7 (2.72)	$\approx$	417.1 (2.46)	$\approx$	417.4 (2.64)	$\approx$	417 (2.62)	$\approx$	417.3 (2.63)	$\approx$	417 (3.14)
B-<Tmean,0.75>	**216.1** (0.83)	$\approx$	216.2 (1.05)	$\approx$	216.3 (1.01)	$\approx$	216.2 (1.02)	$\approx$	216.3 (1.25)	$\approx$	216.3 (1.12)	$\approx$	216.2 (1.06)
C-<WTmean,0.95>	1727.5 (21.91)	$+$	**1726.9** (22.4)	$+$	1731.2 (24.44)	$\approx$	1729.4 (23.85)	$\approx$	1730.1 (23.79)	$\approx$	1728 (22.87)	$\approx$	1735.8 (23.08)
C-<WTmean,0.85>	**722.4** (6.35)	$\approx$	723.1 (4.63)	$\approx$	724.4 (7.19)	$\approx$	723.6 (6.27)	$\approx$	723.8 (6.14)	$\approx$	723.1 (5.64)	$\approx$	724 (6.05)
C-<WTmean,0.75>	392.2 (2.07)	$\approx$	392.1 (2.3)	$\approx$	392.1 (2.1)	$\approx$	392.1 (2.54)	$\approx$	392.2 (2.34)	$\approx$	**391.9** (2.09)	$\approx$	392.3 (2.41)
D-<Fmax,0.95>	4542.1 (74.6)	$\approx$	4541.8 (83.84)	$\approx$	4557.7 (85.67)	$\approx$	4559 (89.01)	$\approx$	4538.8 (93.97)	$\approx$	4549.3 (89.26)	$\approx$	**4533.5** (75.13)
D-<Tmax,0.95>	3984.6 (94.79)	$\approx$	3970.2 (81.27)	$\approx$	**3956** (85.99)	$+$	3961.9 (93.57)	$\approx$	3984.2 (95.44)	$\approx$	3963.6 (83.82)	$\approx$	3994.7 (92.4)
E-<WFmean,0.95>	2714.3 (25.23)	$\approx$	2713.9 (25.96)	$\approx$	2717.7 (25.08)	$\approx$	2714.4 (23.89)	$\approx$	**2712.5** (28.03)	$\approx$	2713.5 (20.68)	$\approx$	2718.2 (26.33)
E-<WTmean,0.95>	1728.9 (23.36)	$\approx$	1728.9 (24.42)	$\approx$	1728.6 (21.82)	$\approx$	1728.7 (22.46)	$\approx$	1726.8 (27.66)	$\approx$	**1725.3** (23.76)	$\approx$	1732.4 (25.78)
F-<Fmean,0.95>	1569.6 (18.02)	$\approx$	1566.2 (11.44)	$\approx$	1568.4 (15.68)	$\approx$	1569.9 (11.14)	$\approx$	**1564.8** (11.46)	$\approx$	1568.5 (11.32)	$\approx$	1568.8 (9.34)
F-<Tmean,0.85>	417.4 (3.23)	$\approx$	**416.5** (2.32)	$\approx$	417.2 (2.5)	$\approx$	417.7 (2.58)	$\approx$	416.9 (2.37)	$\approx$	417.5 (3.1)	$\approx$	417.3 (3.3)
F-<WTmean,0.75>	392.6 (1.98)	$\approx$	**391.8** (1.96)	$\approx$	392.2 (2.53)	$\approx$	392.3 (1.87)	$\approx$	392.6 (3.72)	$\approx$	394.1 (12.07)	$\approx$	392.8 (3.86)
win-draw-lose	1-15-0		1-13-2		1-13-2		0-16-0		0-14-2		0-16-0		
mean rank	4.09		**3.13**		4.75		3.94		3.97		3.91		4.22
p-value	1.000		0.782		0.993		1.000		1.000		1.000		

7.5.3 Rate of Knowledge Transfer

When selecting parents from a merged set of individuals that consists of generalist and specialist sub-populations, MLSI always selects parents that are good at a certain task, no matter where they come from. It helps MLSI flexibly adjust the rate of knowledge transfer (i.e., selecting parents from the generalist sub-population) in the course of evolution. To validate the necessity of selecting a suitable rate of knowledge transfer, we compare MLSI with different fixed transfer rates and analyze the ratio of knowledge transfer over generations in this section.

First, we explicitly distinguish generalist and specialist sub-populations in the crossover. We select parents from generalist and specialist sub-population based on a fixed rate. The fixed rate of selecting parents from the generalist sub-population is set as 0.1, 0.2, and 0.35 respectively. When the rate is small, the sub-populations will mainly mate within their individuals for crossover, and thus the knowledge transfer among tasks is weak. These compared methods are denoted as "MLSI-fix0.1", "MLSI-fix0.2", and "MLSI-fix0.35" respectively. As shown in table 7.6, when fixing the knowledge transfer rate, the performance of all fixed-rate MLSI variants is decreased. They have a larger mean rank than MLSI and perform significantly worse on two tasks. Furthermore, MLSI-fix0.1 has a significantly worse overall performance than MLSI based on the p-value with Bonferroni correction. Besides, we can see that these fixed-rate variants have different performances on different tasks. For example, while MLSI-fix0.35 has worse mean performance on tasks of Scenario B, MLSI-fix0.2 is not so effective on tasks of Scenario D. It implies that to further improve the performance of specific tasks, a suitable transfer rate should be deliberated.

Second, we investigate the adaptation ability of the knowledge transfer rate in MLSI, which is defined as the mean rate of mating with generalist sub-population parents. The curves of mating rate over generations are drawn. If the mating rate is high, MLSI prefers to produce offspring

Table 7.6: Mean (std.) test performance of MLSI with fixed transfer rates.

Tasks	MLSI-fix0.1	MLSI-fix0.2	MLSI-fix0.35	MLSI
A-<Fmean,0.95>	1569.5 (11.7) ≈	1568.6 (11.0) ≈	1570.9 (12.3) −	**1567.1** (16.5)
A-<Fmean,0.85>	864 (2.1) −	863.3 (2.4) −	863.6 (2.7) −	**862.6** (2.5)
A-<Fmean,0.75>	656.4 (1.1) ≈	656.1 (1.1) ≈	656.2 (1.1) ≈	**655.9** (1.0)
B-<Tmean,0.95>	1120.4 (12.8) ≈	1118.8 (12.1) ≈	1121.7 (12.1) ≈	**1116.6** (11.3)
B-<Tmean,0.85>	417.7 (3.3) ≈	417.3 (3.3) ≈	417.4 (2.3) ≈	**417** (3.1)
B-<Tmean,0.75>	216.3 (0.9) ≈	216.3 (1.3) ≈	216.4 (1.1) ≈	**216.2** (1.1)
C-<WTmean,0.95>	1730.8 (22.5) ≈	1739.1 (30.7) ≈	**1723.4** (24.8) +	1735.8 (23.1)
C-<WTmean,0.85>	724.6 (5.4) ≈	725.1 (6.6) ≈	**724** (6.9) ≈	**724** (6.1)
C-<WTmean,0.75>	**391.7** (2.0) ≈	392.7 (2.4) ≈	392.4 (2.1) ≈	392.3 (2.4)
D-<Fmax,0.95>	4558.5 (91.1) ≈	4586.9 (106.8) −	**4532.4** (83.8) ≈	4533.5 (75.1)
D-<Tmax,0.95>	3985.6 (89.1) ≈	4004.9 (93.4) ≈	**3970.6** (86.2) ≈	3994.7 (92.4)
E-<WFmean,0.95>	2722.2 (27.9) ≈	**2714.8** (25.3) ≈	2719 (28.1) ≈	2718.2 (26.3)
E-<WTmean,0.95>	1734.4 (25.9) ≈	1733.6 (28.1) ≈	**1731.7** (27.5) ≈	1732.4 (25.8)
F-<Fmean,0.95>	1573.4 (12.9) ≈	1567.9 (12.0) ≈	**1567.6** (11.2) ≈	1568.8 (9.3)
F-<Tmean,0.85>	418.4 (2.6) −	417.4 (2.5) ≈	**417.2** (2.3) ≈	417.3 (3.3)
F-<WTmean,0.75>	393.9 (12.4) ≈	393.8 (12.2) ≈	**391.8** (1.9) ≈	392.8 (3.9)
win-draw-lose	0-14-2	0-14-2	1-13-2	
mean rank	3.31	2.63	2.22	**1.84**
p-value	0.01	0.31	0.84	

based on the parents from the generalist sub-population. Thus, the knowledge transfer among tasks is frequent. Contrarily, the individuals from the generalist sub-population are hardly selected by MLSI. The specialist sub-populations mainly evolve independently.

As shown in Fig. 7.8, the decline in mating rate can be seen in all scenarios. At the beginning of evolution, sharing knowledge among tasks is very useful. Individuals in generalist sub-populations often have superior performance, which is more likely to produce effective offspring. However, the mating rate decreases with generations. When heuristics become more sophisticated, evolving them specifically is more effective than sharing knowledge (e.g., building blocks) with other similar tasks. Besides, if we look at the level of the mating rate, we can also find that problems with

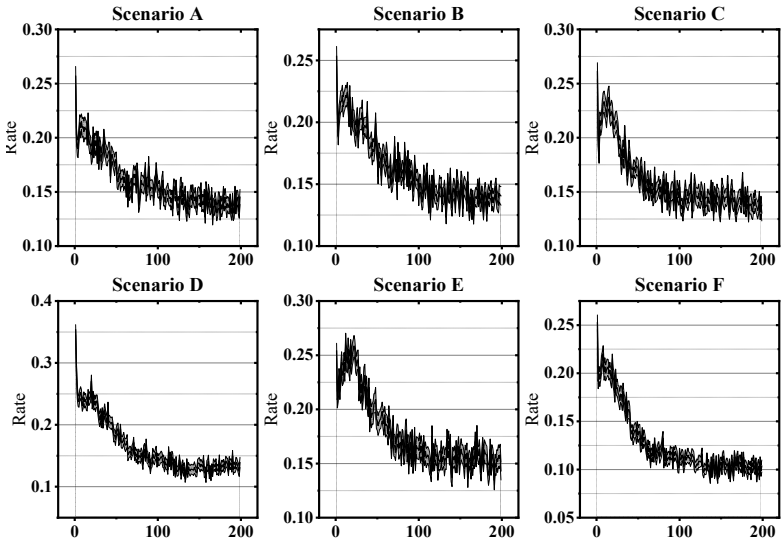

Figure 7.8: The mean rate of knowledge transfer over generations of MLSI. X-axis: generations, Y-axis: the mean rate of mating with the generalist sub-population.

less similar tasks have lower mating rates. For example, Scenario F whose both optimization objectives and utilization levels are different has a mating rate of 0.1 at the final stage of evolution. Contrarily, other problems in which only the objective or utilization level is different, have a mating rate of 0.15 at the end. Based on these results, we believe MLSI not only adjusts the transfer rate in different stages of evolution but also decides the rate based on similarity among tasks.

7.5.4 Example Program Analysis

To further analyze the behavior of MLSI, we investigate the obtained programs by MLSI. Specifically, we pick the final obtained programs for the three tasks (i.e., individuals with the best training fitness for each task) from an independent run in Scenario A as examples, which are shown in Fig. 7.9 to 7.11. In these figures, ovals denote functions while rectangles

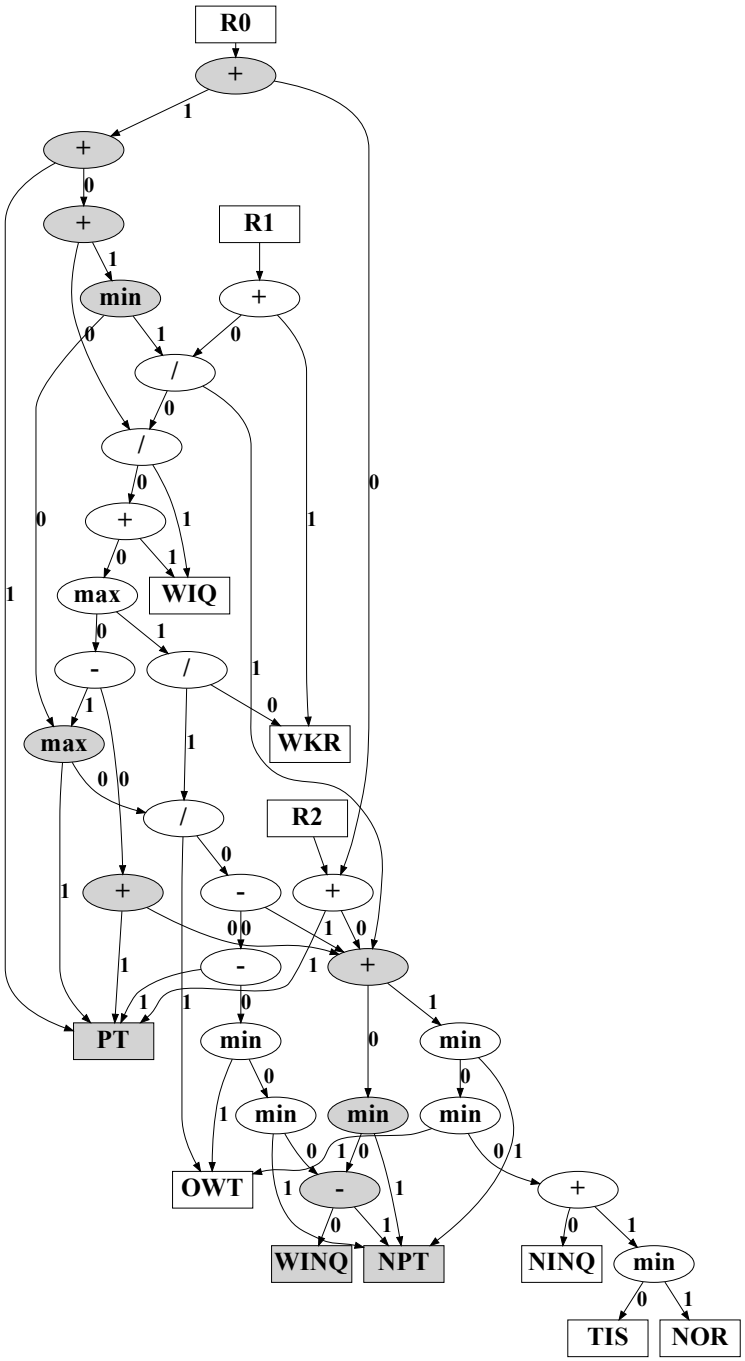

Figure 7.9: The final outputted programs for <Fmean,0.95> from an independent run in Scenario A. The common building blocks are highlighted in dark nodes.

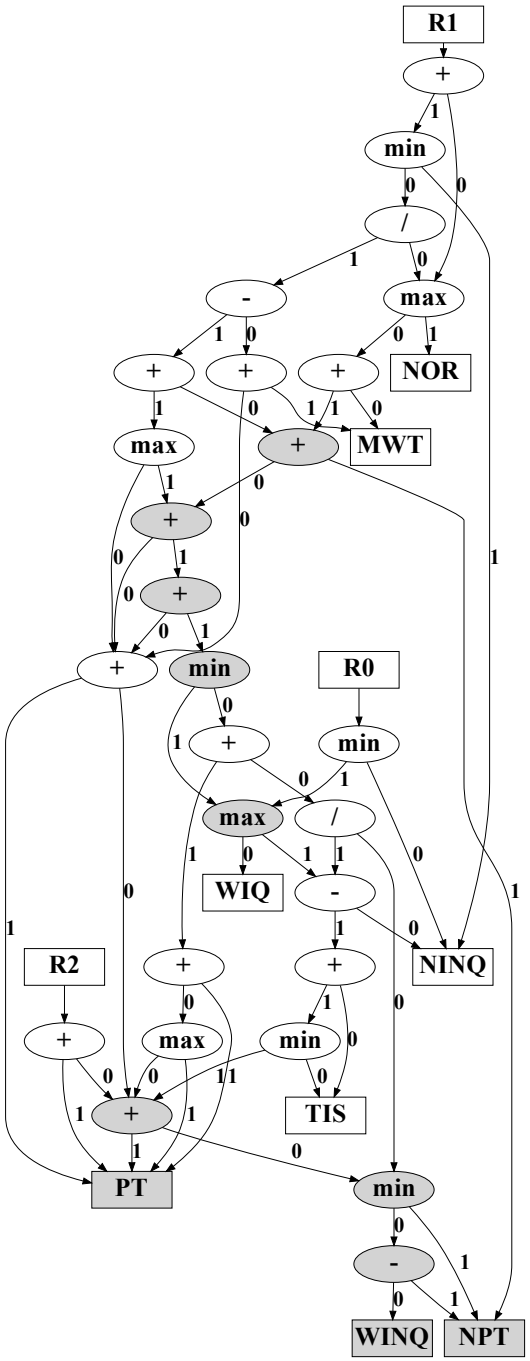

Figure 7.10: The final outputted programs for <Fmean,0.85> from an independent run in Scenario A. The common building blocks are highlighted in dark nodes.

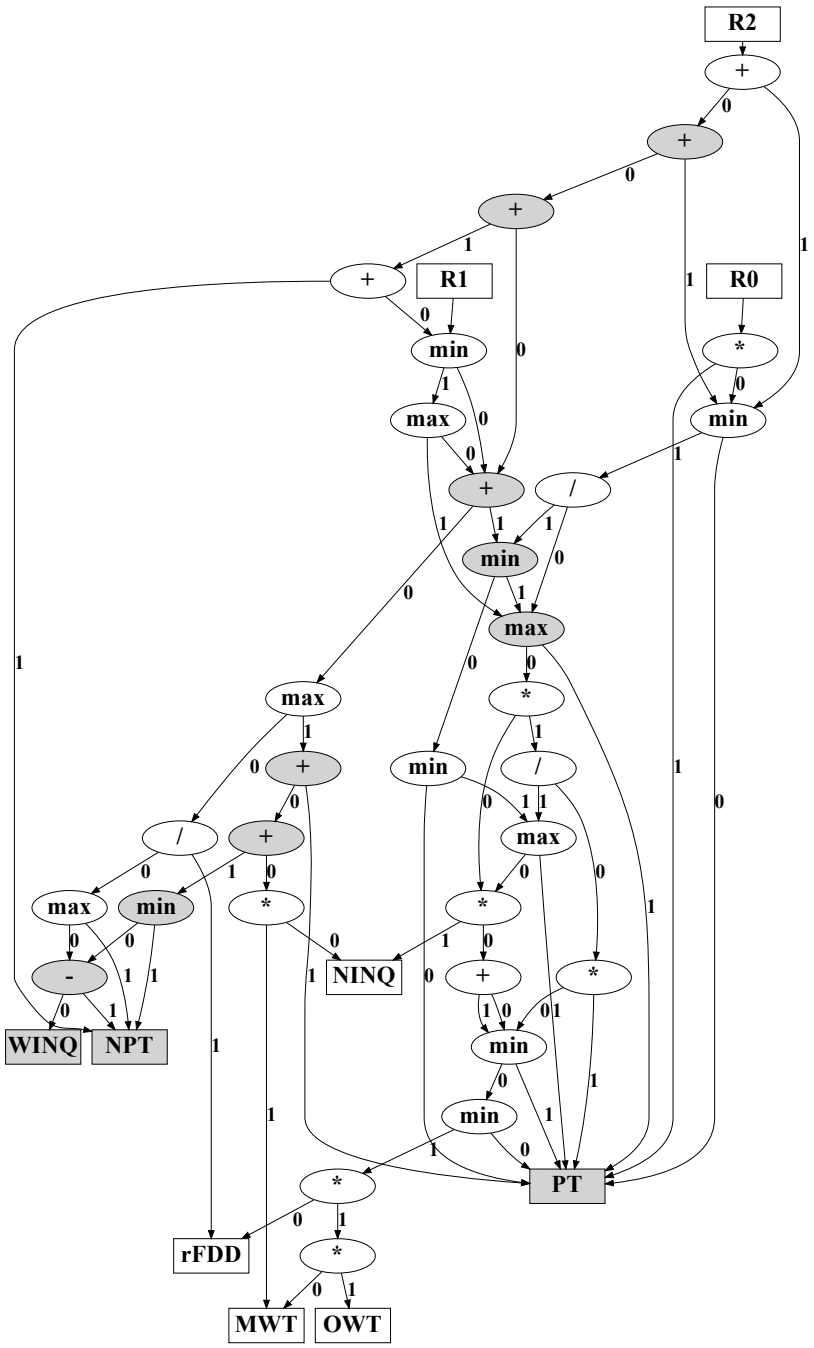

Figure 7.11: The final outputted programs for <Fmean,0.75> from an independent run in Scenario A. The common building blocks are highlighted in dark nodes.

denote terminals. The directed edges specify the inputs of functions. The numbers (i.e., 0 and 1) beside the edges respectively indicate the first or second argument for the function.

It can be seen that these outputted programs for the three similar tasks share some similarities. First, some common building blocks can be found in these outputted programs. For example, these three programs share a building block of "$\min(\mathrm{WINQ}-\mathrm{NPT}, \mathrm{NPT})+\mathrm{PT}$". It implies that minimizing the processing time (PT) and the smaller value between the processing time of the next operation (i.e., NPT) and the remaining total processing time in the next corresponding machine buffer (i.e., WINQ-NPT), is a useful strategy to minimize mean flowtime. Second, they have a similar distribution of terminals. All of them utilize the processing time of operations and the next operation (i.e., PT and NPT), and the total processing time and the number of operations in the next corresponding machine buffer (i.e., WINQ and NINQ). Additionally, at least two of the three programs include a number of remaining operations (NOR), waiting time of an operation (OWT), and other machine-related terminals (e.g., WIQ and MWT) as primitives.

Despite the similarity, we can find that the outputted programs are specialized for different tasks respectively. For example, outputted programs for <Fmean,0.95> (Fig. 7.9) and <Fmean,0.85> (Fig. 7.10) mainly apply "PT" together with addition. Contrarily, the one for <Fmean,0.75> (Fig. 7.11) directly minimizes "PT" (i.e., $\min(\mathrm{PT}, *)$, "*" denotes any input) in most cases. It shows that the three programs have different building blocks when using a terminal. Besides, the output register for specific tasks (e.g., R0 for <Fmean,0.95>) aggregates the results from most of the graph nodes, standing for the most sophisticated program in this LGP individual. On the contrary, the other output registers only store the intermediate results from part of the computer program. It implies that these programs are specifically designed for a certain task. Simply seeing the whole individual from one task as the parents of another is likely to be

ineffective.

By comparing the learnt heuristics from multitask tree-based GP and transfer learning methods for tree-based GP [171, 264], we find that the common building blocks in graph-based structures can be flexibly distributed in different parts of the graphs and can be reused multiple times without duplication, which is different from tree-based GP whose common building blocks appear in particular parts of a tree, and each can be typically used by a single part of tree only. To summarise, graph-based structures show a flexible and concise way of reusing shared knowledge among tasks.

7.6 Chapter Summary

The main goal of this chapter is to improve the effectiveness and training efficiency of LGPHH for DJSS problems by multitask optimization. It has been successfully achieved by designing a new knowledge transfer mechanism that evolves shared individuals with multiple outputs, each for one task, based on the graph-based structure. We also propose corresponding genetic operators to evolve the multitask LGP method.

The experiment results show that the proposed method (i.e., MLSI) is significantly better than the baseline method and three state-of-the-art multitask GP methods. Further analyses verify that the adaptation ability of the transfer rate based on the evolutionary process and the similarity among tasks is the essential reason for the superior performance. These results fully imply the great potential of graph-based structures in multitask optimization. The proposed knowledge transfer mechanism enriches the methodologies in transferring knowledge, but also provides an effective example of designing graph-based knowledge transfer.

Based on this chapter and the previous chapters, we have proposed four advanced LGP methods (i.e., LGP with graph-based search mechanisms, grammar-guided LGP, LGP with FLO, and multitask LGP). These

advanced LGP methods show superior performance in solving DJSS. In the next chapter, we will extend two of the proposed advanced LGP methods to other problem domains, symbolic regression more specifically, to investigate the potential of their generality.

Chapter 8

Further Discussions — Extension of the Advanced LGP to Symbolic Regression

8.1 Introduction

The proposed advanced LGP methods in this thesis have a good potential generality to other domains besides DJSS. To investigate the potential generality of the proposed methods, this chapter extends multi-representation GP (MRGP, see chapter 4) and fitness landscape optimization (FLO, see chapter 6) to symbolic regression (SR). MRGP makes use of the synergy between GP representations to enhance GP performance, and FLO helps discover effective solutions by optimizing the neighborhood structures on the fitness landscape. We choose these two proposed methods for SR problems mainly because they are generic and are less dependent on the characteristics of investigated DJSS problems.

8.1.1 Chapter Goals

The goal of this chapter is to *investigate the potential generality of the two proposed advanced LGP methods by extending them to symbolic regression problems*. Specifically, this chapter has the following research objectives:

1. Develop the experiments of SR problems, including the synthetic and real-world benchmarks in SR and the comparison of other GP methods.

2. Verify the performance of MRGP in SR problems, including the test effectiveness and training efficiency.

3. Verify the performance of FLO in SR problems, including the test effectiveness and training efficiency.

8.1.2 Chapter Organization

The rest of this chapter is organized as follows. Section 8.2 first introduces the SR problems. Sections 8.3 and 8.4 apply MRGP and FLO to SR problems to verify their performance, respectively. Finally, section 8.5 summarizes this chapter.

8.2 Problem Description

SR is a supervised learning problem, in which GP learns regression models to map the input features to given target outputs without presuming the model structure [154, 192, 276]. The effectiveness of solving SR problems gives insights into the generality of the proposed algorithms in this thesis.

We verify the proposed methods based on three synthetic benchmarks and five real-world benchmarks, as shown in table 8.1. The benchmarks are selected from recently published papers for solving symbolic regression [2,87], which are downloaded from the UCI machine learning datasets. The ground truth functions of the synthetic benchmarks cover a wide

Table 8.1: The symbolic regression problems.

Benchmarks	Function	#Features	Data range	#Points (Train,Test)
Synthetic benchmarks				
Nguyen4	$f(x) = x^6 + x^5 + x^4 + x^3 + x^2 + x$	1	[-1,1]	(20,1000)
Keijzer11	$f(x, y) = xy + \sin((x-1)(y-1))$	2	[-1,1]	(100,900)
R1	$f(x) = \frac{(x+1)^3}{x^2-x+1}$	1	[-2,2]	(20,1000)
Real-world benchmarks				
Airfoil	unknown	5	-	(1127,376)
BHouse	unknown	13	-	(380,126)
Tower	unknown	25	-	(3749,1250)
Concrete	unknown	8	-	(772,258)
Redwine	unknown	11	-	(1199, 400)

range of functions (e.g., $\times$ and $\sin$), and the real-world benchmarks have various numbers of features and data ranges.

This chapter applies relative square error (RSE) as the fitness function to measure the performance of GP methods, as shown in Eq. 8.1.

$$\text{RSE} = \frac{\text{MSE}(\mathbf{y}, \hat{\mathbf{y}})}{\text{VAR}(\mathbf{y})} = \frac{\sum_i^n (y_i - \hat{y_i})^2}{\sum_i^n (y_i - \overline{y})^2} \tag{8.1}$$

where MSE is the mean square error, VAR is the variance, and $\mathbf{y}$ and $\hat{\mathbf{y}}$ are the target output and estimated output respectively. $\overline{y}$ is the average of the target output. A small RSE value implies that a regression model has a good fitting performance with the given data.

8.3 Multi-representation GP for SR

8.3.1 Comparison Design

This section verifies the effectiveness of MRGP for solving SR problems. The comparison design in this section follows the one in section 4.5. We take MRGP with tree-based and linear representation (MRGP-TL) as an example to verify the effectiveness of MRGP. We compare MRGP-TL with three baseline methods. They are the basic TGP, basic LGP, and the TLGP which evolves tree-based and linear representations independently by two sub-populations.

We set the parameters of the compared GP methods based on the popular settings in existing literature [29, 106] and section 4.5. In symbolic regression problems, each LGP individual has at most 100 instructions, and each TGP individual has a maximum tree depth of 10. All the compared GP methods use the same function set and terminal set (LGP methods have registers in the terminal set additionally). The function set includes 8 functions, which are $\{+, -, \times, \div, \sin, \cos, \ln(|\cdot|), \sqrt{|\cdot|}\}$[1]. The input feature set is defined based on the inputs of benchmark problems.

8.3.2 Test Performance

Table 8.2 shows that MRGP-TL has superior test effectiveness to the other compared methods. The best mean performance among the compared methods is highlighted in bold. The Friedman test with the Bonferroni correction and a significance level of 0.05 shows a p-value of 0.041, which indicates a significant difference among the compared methods. The further pair-wise comparison shows that MRGP-TL has the best mean rank over

[1] $\div$ returns 1.0 if the dividend equals to 0.0. $\ln(|\cdot|)$ returns the operand if the raw output is smaller than -50.

Table 8.2: The mean test performance (std.) of the compared methods.

Datasets or scenarios	TLGP	TGP	LGP	MRGP-TL
Nguyen4	0.069 (0.059) $-$	0.053 (0.091) $\approx$	0.149 (0.248) $-$	**0.051** (0.087)
Keijzer11	0.365 (0.240) $\approx$	**0.273** (0.121) $\approx$	0.339 (0.142) $\approx$	0.323 (0.133)
R1	0.035 (0.029) $\approx$	**0.022** (0.023) $\approx$	0.034 (0.035) $\approx$	0.025 (0.025)
Airfoil	0.667 (0.091) $\approx$	**0.638** (0.117) $\approx$	0.643 (0.132) $\approx$	0.643 (0.098)
BHouse	0.384 (0.100) $-$	0.392 (0.131) $\approx$	0.404 (0.126) $-$	**0.325** (0.076)
Tower	0.358 (0.052) $-$	0.364 (0.053) $-$	0.345 (0.046) $-$	**0.325** (0.037)
Concrete	0.496 (0.096) $-$	0.438 (0.107) $\approx$	0.471 (0.099) $-$	**0.39** (0.078)
Redwine	0.745 (0.042) $\approx$	0.761 (0.036) $\approx$	0.759 (0.034) $\approx$	**0.757** (0.035)
win-draw-lose	0-4-4	0-7-1	0-4-4	
Mean rank	3.0	2.13	3.25	**1.63**
p-value (vs. MRGP-TL)	0.20	1.0	0.071	

the others. The Wilcoxon rank-sum test with a significance level of 0.05[2] confirms that MRGP-TL is significantly better than the compared methods in many cases. The comparison of the test performance confirms the good effectiveness of MRGP-TL for solving SR problems.

8.3.3 Training Performance

Fig. 8.1 shows the training performance of MRGP-TL and other compared methods on four example SR benchmarks. We can see that the red curves (i.e., MRGP-TL) reach better fitness (i.e., lower) than the other within fewer fitness evaluations. For example, in BHouse, the red curves reach the fitness of 0.4 at about 10000 fitness evaluations, while the other compared methods reach a similar fitness at about 30000 fitness evaluations. The results show a high training efficiency of MRGP-TL.

[2] "$+$" indicates a method is significantly better than MRGP-TL, "$-$" indicates a method is significantly worse than MRGP-TL, and "$\approx$" indicates a statistical similarity.

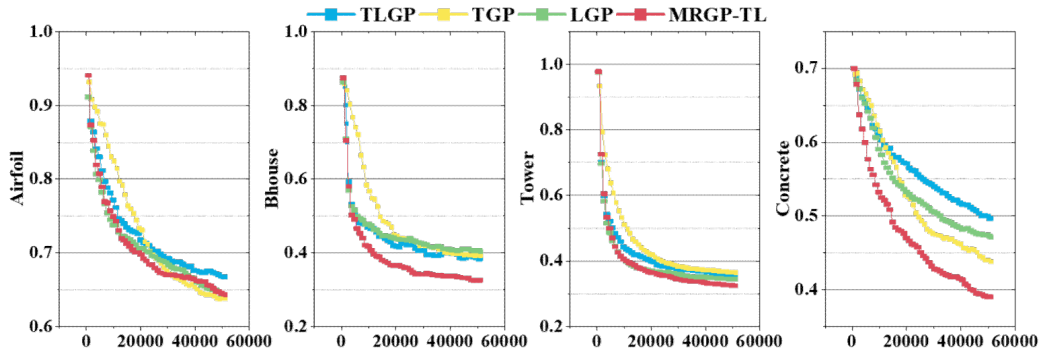

Figure 8.1: Test performance of the compared methods over generations in the four symbolic regression benchmarks. X-axis: fitness evaluations. Y-axis: average test RSE for symbolic regression problems.

8.4 Searching against Optimized Fitness Landscapes of SR

This section verifies the generality of FLO. We verify the effectiveness of FLO by applying LGP to search against the optimized FLs by FLO, denoted as LGP-FLO.

8.4.1 Comparison Design

We select six benchmark problems from symbolic regression problems, including Nguyen4, Keijzer11, R1, Airfoil, BHouse, and Redwine. We apply relative square error (RSE) to measure the performance in symbolic regression. We verify the effectiveness of LGP-FLO by the compared methods in chapter 6. Specifically, "basicLGP" indicates the basic LGP. "freqmut" indicates a basic LGP with a frequency-based mutation. "swap" indicates an LGP that applies the operator of swapping consecutive instructions in LGP search. We also select a state-of-the-art method (denoted as SOTA) in

Table 8.3: Mean test performance (and standard deviation) on the six SR benchmark problems. The best mean performance is highlighted in bold.

Problems	basicLGP	freqmut	swap	SOTA	LGP-FLO
Nguyen4	0.149 (0.248) −	0.069 (0.052) ≈	0.071 (0.064) −	0.052 (0.064) ≈	**0.048 (0.045)**
Keijzer11	0.339 (0.142) −	0.299 (0.103) −	0.288 (0.121) −	**0.213 (0.091)** ≈	0.22 (0.104)
R1	0.034 (0.035) −	0.02 (0.026) −	0.016 (0.027) −	0.011 (0.034) −	**0.01 (0.02)**
Airfoil	0.643 (0.132) −	0.557 (0.113) ≈	0.559 (0.114) ≈	0.524 (0.049) ≈	**0.521 (0.095)**
BHouse	0.404 (0.126) ≈	0.443 (0.11) ≈	0.374 (0.114) +	**0.312 (0.063)** +	0.417 (0.088)
Redwine	0.759 (0.034) −	0.74 (0.037) ≈	0.741 (0.033) ≈	**0.699 (0.025)** +	0.735 (0.031)
mean ranks	4.67	4.17	2.83	**1.5**	1.83
pair-wise p-value	0.019	0.106	1	1	

symbolic regression problems as one of the compared methods. The SOTA in this section represents a semantic LGP [87] in SR problems. The parameters of SOTA methods are set as their recommendation. The primitive set of the compared methods is the same as section 8.3.

8.4.2 Test Performance

Table 8.3 shows the test RSE of the compared methods. The Friedman test with the Bonferroni correction and a significance level of 0.05 shows a p-value of 9.14e-4, indicating a significant difference. The pair-wise analyses further show that LGP-FLO is competitive with SOTA methods and has significantly better test effectiveness than basic LGP. The Wilcoxon rank-sum test confirms that LGP-FLO has superior performance to basic LGP, freqmut, and swap in many cases, which indicates a promising performance of LGP-FLO in SR problems.

8.4.3 Training Performance

We analyze the training performance of the compared methods to have a better understanding of the efficiency of FLO, as shown in Fig. 8.2. We can see that the red curves (i.e., LGP-FLO) achieve better fitness (i.e., lower) than most of the compared methods and have competitive performance in

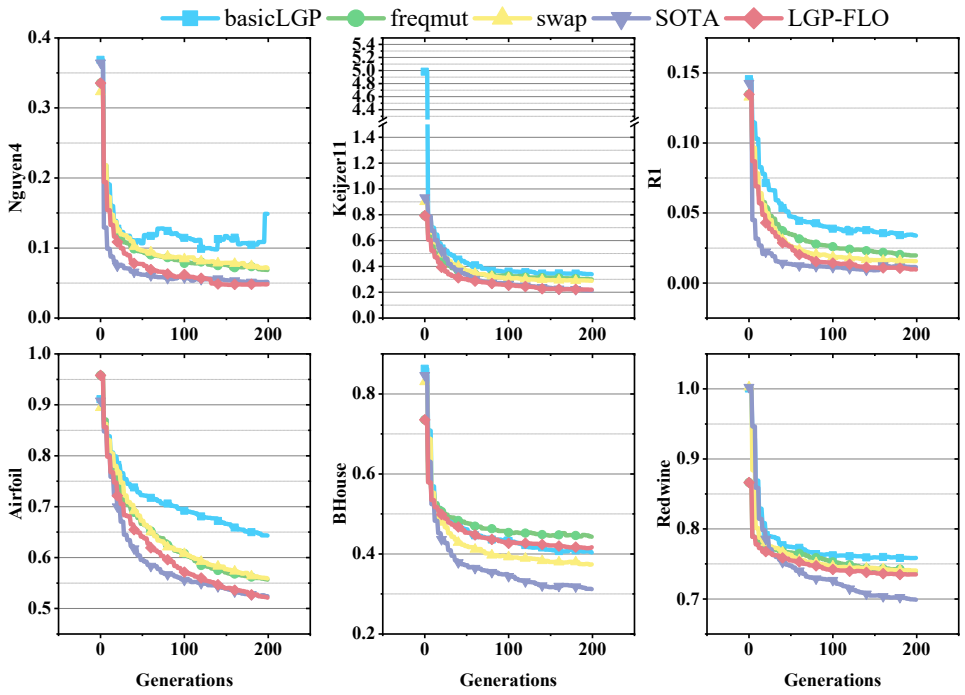

Figure 8.2: The training performance over generations of the compared methods for SR problems. X-axis: the generations, Y-axis: the training RSE.

most cases. Although the SOTA method for SR problems has better training efficiency than LGP-FLO in the last two problems, it is too specific too SR problems and cannot be applied to DJSS problems. In light of the competitive performance of LGP-FLO in both SR problems and DJSS problems (see chapter 6), we believe that FLO is a potential research direction.

8.5 Chapter Summary

The main goal of this chapter is to investigate the potential generality of the proposed advanced LGP methods in this thesis. Specifically, we extend

two of the proposed LGP methods, MRGP and FLO, to SR problems. We extend these two methods by replacing the primitive set and using the common parameter settings of SR problems.

The results of these two methods both show better or at least competitive performance with the compared methods, which is consistent with our findings in DJSS problems. By taking advantage of different GP representations, MRGP is less dependent on the domain knowledge of the suitable representations for different problems. By optimizing the neighborhood structures of FLs based on elite solutions, LGP-FLO achieves encouraging test effectiveness and training efficiency. We believe that both MRGP and FLO are effective in enhancing GP performance and have good generality. It would be interesting to further investigate and make full use of these advanced LGP methods in other domains.

Chapter 9

Conclusions

The overall goal of this thesis is to enhance the performance of LGP methods for DJSS problems. This goal has been successfully achieved by developing four advanced LGP methods, including the graph-based search mechanism, grammar-guided LGP, the fitness landscape optimization technique, and the LGP-based multitask optimization techniques. We apply these advanced techniques to solve DJSS problems to verify the effectiveness of these techniques. We further extend two proposed methods, the graph-based search mechanism and fitness landscape optimization techniques, to symbolic regression problems. The performance on symbolic regression problems verifies that the two advanced LGP methods have a very good generalization ability in other domains. The rest of this chapter highlights the achieved objectives and their main contributions in this thesis. Then, this chapter discusses the potential research directions motivated by the achievement of this thesis.

9.1 Achieved Objectives

The following research objectives have been fulfilled by this thesis.

1. Chapter 3 developed an LGP-based hyper-heuristic method for solving DJSS problems. We identified key adaptations when applying LGPHH

for DJSS. Specifically, 1) LGP should evolve with a small population and long generations, 2) mutation-dominated settings of genetic operator rates are effective for LGPHH, and 3) LGP should fine-tune the number of registers and initialize them by diverse and useful attributes of DJSS problems. The results verify that the LGP is promising in evolving dispatching rules for DJSS and can produce more compact dispatching rules than TGP.

2. Chapter 4 developed a series of graph-based search mechanism to make full use of the graph characteristics of LGP for DJSS. The graph-based search mechanism includes three achievements. First, we verified the effectiveness of the building blocks in graphs by developing graph-based crossover. Second, we proposed an effective graph-to-instruction transformation for LGP programs based on the adjacency list. Converting graphs into LGP instructions based on the adjacency list effectively carries the search information between the graph and instruction representations. Third, based on the graph-to-instruction transformation, this thesis further proposed a multi-representation GP which makes use of the synergy between different GP representations to improve LGP performance in DJSS problems.

3. Chapter 5 developed a grammar-guided LGP framework to incorporate the domain knowledge of DJSS into LGP search. The grammar-guided LGP framework includes a module context-free grammar system and a set of grammar-guided genetic operators. The module context-free grammar system defines the grammar rules, and the grammar-guided genetic operators evolve LGP programs based on the grammar rules. Based on this method, we further incorporate the domain knowledge of the IF operation, a representative flow control operation, into the LGP search for DJSS dispatching rules. The results show that the introduction of IF operations and grammar rules significantly enhances dispatching rule performance for solving complicated DJSS problems.

4. Chapter 6 developed a fitness landscape optimization method to automatically design better fitness landscapes for LGP search. The fit-

ness landscape optimization method improves the fitness landscape by optimizing the neighborhood structures of LGP solutions. We design a stochastic gradient descent method to optimize the neighborhood structures based on the symbol indexes. After the optimization, we visualize the fitness landscape of LGP and discover two patterns of the fitness landscape, that is, fitness aligning and diagonal symmetry.

5. Chapter 7 proposed a multitask LGP and developed an LGP-based multitask evolutionary framework. We designed a graph-based knowledge transfer mechanism (i.e., shared individuals) in the LGP-based multitask optimization framework to make use of the natural sharing of building blocks in LGP. The results show that the proposed method can automatically adjust the transfer rate over the evolution to enhance effectiveness and evolve more compact rules than state-of-the-art methods.

6. Chapter 8 extended two advanced LGP methods, multi-representation GP and LGP with fitness landscape optimization, to symbolic regression problems. The superior performance in symbolic regression problems suggests the good potential generality of the two proposed methods.

9.2 Main Conclusions

This section describes the main conclusions for this thesis drawn from the six major contribution chapters, i.e., chapters 3 to 8.

1. To the best of our knowledge, it is the first time to apply LGP as an HH method for solving dynamic combinatorial optimization problems. Specifically, this thesis proposes three key adaptations of LGP to overcome the limited training instances. The results verify that LGP is an effective method to evolve dispatching rules for DJSS.

2. The performance gain by the graph-based search mechanism verifies that making full use of the graph-based characteristics of LGP is effective in improving LGP performance for DJSS. The proposed graph-to-instruction transformation can be seen as a bridge from DAGs to LGP in-

structions, to make up the missing part of the graph-based characteristics in existing LGP literature. This transformation greatly facilitates future cooperation between LGP and many other graph-based techniques, such as neural networks. In the thesis, we give an example of cooperating the tree-based and linear representation for GP evolution, that is multi-representation GP. To the best of our knowledge, multi-representation GP is the first work highlighting that the interplay among different GP representations is useful for improving GP performance.

3. The proposed grammar-guided LGP is the first work that directly imposes grammar rules on LGP. Aimed with the grammar-guided techniques, we can restrict the search space based on our domain knowledge by grammar rules and enhance the effectiveness of solving DJSS. Furthermore, we introduce the domain knowledge of IF operations into LGPHH for DJSS based on the grammar-guided techniques. It is a very first step in evolving flow control operations in LGPHH. The thesis summarizes three key principles for designing grammar rules of flow control operations. The results show that our proposed methods and principles successfully evolve effective and interpretable dispatching rules with IF operations, and the IF operations significantly improve the performance of dispatching rules for complicated DJSS problems. The achievement of grammar-guided LGP implies the great potential of flow control operations in DJSS problems.

4. The proposed fitness landscape optimization method is the very first work that explicitly optimizes the fitness landscape of LGP in an automatic manner. The proposed method is general enough to improve the fitness landscape for a specific task. The results on DJSS confirm that the proposed fitness landscape optimization method successfully optimizes the landscape to be significantly more cone-like. The visualization of the optimized landscapes further discovers two patterns, which successfully insight a missed effective genetic operator for LGP (i.e., swapping consecutive instructions). The proposed method can significantly improve the

effectiveness of LGP. The achievement shows great potential for the research direction of fitness landscape optimization.

5. The proposed LGP-based multitask optimization framework verifies the performance gain by the multitask optimization techniques for LGPHH methods. The multitask optimization technique improves the effectiveness and efficiency of LGPHH for DJSS. Besides, the proposed multitask LGP provides a new way of sharing search knowledge within one LGP individual. By sharing common building blocks within a multi-output LGP individual, the proposed multitask LGP has a concise representation and efficiently transfers effective knowledge among tasks. The superior test performance of the proposed method verifies the potential of the multi-output characteristic of LGP.

6. Further investigation of LGP on SR problems suggests the good potential generality of the proposed advanced LGP methods. This shows that the proposed advanced LGP methods have great potential in other domains.

9.3 Applications of Our Proposed Methods

This thesis proposed advanced LGP methods for designing dispatching rules in DJSS problems. These methods take advantage of different levels of search information of dispatching rules to enhance LGP performance. Specifically, the graph-based search mechanism improves the search efficiency of LGP based on the DAGs of dispatching rules. The grammar-guided technique facilitates the incorporation of the DJSS domain knowledge into LGP search. The fitness landscape optimization enhances the effectiveness of dispatch rules by optimizing the neighborhood structures of LGP fitness landscapes. The multitask optimization technique further improves LGP performance when users have multiple similar DJSS tasks. DJSS users and researchers can choose the proposed methods based on their needs. When they have limited training time, graph-based mecha-

nism will help. When they have a lot of domain knowledge, grammar-guided techniques will help. When they have multiple similar tasks to solve, the multitask optimization technique will help.

Furthermore, combining these proposed methods into one powerful method is another potential direction. The reason is two-fold. First, these proposed methods enhance LGP search from different perspectives, which can cooperate with each other easily. Second, the proposed multi-representation GP framework facilitates different GP methods with different representations and different search mechanism to evolve together and share search information.

9.4 Future Work

9.4.1 Feature Engineering for DJSS

Existing GPHH methods for DJSS evolve dispatching rules based on myopic features, which limit the overall performance. To further improve the effectiveness of solving DJSS problems, a dispatching rule should consider more foresighted features, such as estimating the arrival time of coming jobs. Besides, designing effective features for DJSS is a tedious but less effective job currently. The features for DJSS are mainly designed manually, and the expert knowledge likely limits GPHH to discover more effective rules. Performing feature engineering for DJSS problems is a potential research direction.

9.4.2 Evolving Large LGP Programs

Based on the achieved objectives in this thesis, a very important future research direction is to evolve large LGP programs. The term "large" indicates that the maximum program size is large, and the primitive set of LGP is large. Existing LGP methods suffer from the bloat effect and lack

effective methods to handle those large programs. When problems become complicated and need more sophisticated programs, short and simple programs from the existing LGP methods cannot effectively tackle the problems.

The proposed methods are potential tools to evolve large LGP programs. For example, we would apply grammar-guided LGP to remove the redundant search spaces in large programs and evolve more advanced flow control operations such as FOR operations. We would apply fitness landscape optimization techniques to optimize the neighborhood structures of effective solutions and make a better estimation on the optimal solutions. Furthermore, we would transfer optimized symbol indexes from simple tasks to difficult tasks so that we improve the efficiency and effectiveness of LGP search.

9.4.3 LGP Computation Hardware

Existing LGP studies only apply multiple threads to parallelize the fitness evaluation of individuals. However, this design ignores the potential of pipelining the execution of LGP instructions. To efficiently evaluate large LGP programs, powerful LGP-efficient computation hardware is necessary. LGP represents programs in the form of assembly programs (i.e., register-based instruction sequences), which is straightforward for computer execution. Pipelining the execution of LGP instructions is a potential direction to further improve the computation efficiency of LGP.

9.4.4 Applications to Other Domains

The results on DJSS and SR problems suggest the good potential generality of the proposed advanced LGP methods. It is interesting to further extend the proposed methods to other domains such as classification and clustering. Besides, the investigated DJSS does not consider some realistic constraints during optimization. It is meaningful to consider more realistic

constraints such as resource constraints in DJSS.

Bibliography

[1] AL-HELALI, B., CHEN, Q., XUE, B., AND ZHANG, M. Multitree Genetic Programming with New Operators for Transfer Learning in Symbolic Regression with Incomplete Data. *IEEE Transactions on Evolutionary Computation 25*, 6 (2021), 1049–1063.

[2] AL-HELALI, B., CHEN, Q., XUE, B., AND ZHANG, M. Multitree genetic programming with new operators for transfer learning in symbolic regression with incomplete data. *IEEE Transactions on Evolutionary Computation 25* (2021), 1049–1063.

[3] AL-SAHAF, H., BI, Y., CHEN, Q., LENSEN, A., MEI, Y., SUN, Y., TRAN, B., XUE, B., AND ZHANG, M. A survey on evolutionary machine learning. *Journal of the Royal Society of New Zealand 49*, 2 (2019), 205–228.

[4] ALI, M. H., SAIF, A., AND GHASEMI, A. Robust job shop scheduling with machine unavailability due to random breakdowns and condition-based maintenance. *International Journal of Production Research 0*, 0 (2023), 1–22.

[5] AROBA, O. J., NAICKER, N., AND ADELIYI, T. T. Node localization in wireless sensor networks using a hyper-heuristic DEEC-Gaussian gradient distance algorithm. *Scientific African 19* (2023), e01560.

[6] ATKINSON, T., PLUMP, D., AND STEPNEY, S. Evolving graphs by graph programming. In *Proceedings of European Conference on Genetic Programming* (2018), pp. 35–51.

[7] BACK, T., FOGEL, D. B., AND MICHALEWICZ, Z. *Handbook of Evolutionary Computation*, 1st ed. CRC Press, Boca Raton, 1997.

[8] BALI, K. K., GUPTA, A., FENG, L., ONG, Y. S., AND SIEW, T. P. Linearized domain adaptation in evolutionary multitasking. In *Proceedings of IEEE Congress on Evolutionary Computation* (2017), pp. 1295–1302.

[9] BANZHAF, W. Genetic programming for pedestrians. In *Proceedings of the Fifth International Conference on Genetic Algorithms* (1993), p. 628.

[10] BANZHAF, W., BRAMEIER, M., STAUTNER, M., AND WEINERT, K. Genetic Programming and Its Application in Machining Technology. *Advances in Computational Intelligence – Theory and Practice* (2003), 194–242.

[11] BANZHAF, W., MACHADO, P., AND ZHANG, M., Eds. *Handbook of Evolutionary Machine Learning*, 1 ed. Genetic and Evolutionary Computation. Springer Nature, Singapore, 2024.

[12] BANZHAF, W., NORDIN, P., KELLER, R. E., AND FRANCONE, F. D. *Genetic Programming: An Introduction On the Automatic Evolution of Computer Programs and Its Applications*. Morgan Kaufmann, 1998.

[13] BAYKASOĞLU, A., AND KARASLAN, F. S. Solving comprehensive dynamic job shop scheduling problem by using a GRASP-based approach. *International Journal of Production Research 7543* (2017), 1–18.

[14] BEHNAMIAN, J. Survey on fuzzy shop scheduling. *Fuzzy Optimization and Decision Making 15*, 3 (2016), 331–366.

[15] BEYER, H.-G., AND SCHWEFEL, H.-P. Evolution strategies – A comprehensive introduction. *Natural Computing 1*, 1 (2002), 3–52.

[16] BI, Y., XUE, B., AND ZHANG, M. Dual-Tree Genetic Programming for Few-Shot Image Classification. *IEEE Transactions on Evolutionary Computation 26*, 3 (2022), 555–569.

[17] BI, Y., XUE, B., AND ZHANG, M. Genetic programming-based evolutionary deep learning for data-efficient image classification. *IEEE Transactions on Evolutionary Computation* (2022), 1–15.

[18] BI, Y., XUE, B., AND ZHANG, M. Multitask Feature Learning as Multiobjective Optimization: A New Genetic Programming Approach to Image Classification. *IEEE Transactions on Cybernetics* (2022), 1–14. doi:10.1109/TCYB.2022.3174519.

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

[20] BRAMEIER, M., AND BANZHAF, W. Effective Linear Program Induction. Tech. rep., Technical Report CI-108/01, Collaborative Research Center 531, University of Dortmund, 2001.

[21] BRAMEIER, M., AND BANZHAF, W. *Linear Genetic Programming*, 1 ed. Genetic and Evolutionary Computation. Springer New York, NY, 2007.

[22] BRANKE, J., NGUYEN, S., PICKARDT, C. W., AND ZHANG, M. Automated Design of Production Scheduling Heuristics: A Review. *IEEE Transactions on Evolutionary Computation 20*, 1 (2016), 110–124.

[23] BURKE, E. K., HYDE, M. R., KENDALL, G., OCHOA, G., OZCAN, E., AND WOODWARD, J. R. Exploring hyper-heuristic methodologies with genetic programming. *Computational Intelligence 1*, 1 (2009), 177–201.

[24] BURKE, E. K., HYDE, M. R., KENDALL, G., OCHOA, G., ʻOZCAN, E., AND WOODWARD, J. R. *A Classification of Hyper-Heuristic Approaches*. Springer, Cham, 2010.

[25] BURKE, E. K., HYDE, M. R., KENDALL, G., OCHOA, G., ÖZCAN, E., AND WOODWARD, J. R. A Classification of Hyper-Heuristic Approaches: Revisited. In *Handbook of Metaheuristics*, International Series in Operations Research & Management Science. Springer International Publishing, 2018, pp. 453–477.

[26] CAZZARO, D., AND PISINGER, D. Variable neighborhood search for large offshore wind farm layout optimization. *Computers and Operations Research 138* (2022), 105588.

[27] CHANDRA, R., ONG, Y. S., AND GOH, C. K. Co-evolutionary multi-task learning for dynamic time series prediction. *Applied Soft Computing Journal 70* (2018), 576–589.

[28] CHEN, Q., XUE, B., AND BANZHAF, W. Relieving Coefficient Learning in Genetic Programming for Symbolic Regression via Correlation and Linear Scaling. In *Proceedings of Genetic and Evolutionary Computation Conference* (2023), pp. 420–428.

[29] CHEN, Q., XUE, B., AND ZHANG, M. Instance based transfer learning for genetic programming for symbolic regression. In *Proceedings of IEEE Congress on Evolutionary Computation* (2019), pp. 3006–3013.

[30] CHEN, Q., ZHANG, M., AND XUE, B. Feature selection to improve generalization of genetic programming for high-dimensional symbolic regression. *IEEE Transactions on Evolutionary Computation 21*, 5 (2017), 792–806.

[31] CHEN, X., HUANG, Y., ZHOU, W., AND FENG, L. Evolutionary Multitasking via Artificial Neural Networks. In *Proceedings of*

IEEE International Conference on Systems, Man and Cybernetics (2021), pp. 1545–1552.

[32] CHEN, Y., ZHONG, J., FENG, L., AND ZHANG, J. An Adaptive Archive-Based Evolutionary Framework for Many-Task Optimization. *IEEE Transactions on Emerging Topics in Computational Intelligence 4*, 3 (2020), 369–384.

[33] CHEN, Z., ZHOU, Y., HE, X., AND ZHANG, J. Learning Task Relationships in Evolutionary Multitasking for Multiobjective Continuous Optimization. *IEEE Transactions on Cybernetics 52*, 6 (2020), 5278–5289.

[34] CHRISTODOULAKI, E., AND KAMPOURIDIS, M. Using strongly typed genetic programming to combine technical and sentiment analysis for algorithmic trading. In *Proceedings of IEEE Congress on Evolutionary Computation* (2022), pp. 1–8.

[35] CLERGUE, M., COLLARD, P., TOMASSINI, M., AND VANNESCHI, L. Fitness Distance Correlation And Problem Difficulty For Genetic Programming. In *Proceedings of Genetic and Evolutionary Computation Conference* (2002), pp. 724–732.

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

[37] CORREA, R. F. R., BERNARDINO, H. S., DE FREITAS, J. M., SOARES, S. S. R. F., GONÇALVES, L. B., AND MORENO, L. L. O. A Grammar-based Genetic Programming Hyper-Heuristic for Corridor Allocation Problem. In *Proceedings of Brazilian Conference on Intelligent Systems* (2022), pp. 504–519.

[38] CRAMER, N. L. A representation for the adaptive generation of simple sequential programs. In *Proceedings of the 1st International Conference on Genetic Algorithms* (1985), pp. 183–187.

[39] CUNHA, B., MADUREIRA, A. M., FONSECA, B., AND COELHO, D. Deep reinforcement learning as a job shop scheduling solver: A literature review. In *Proceedings of International Conference on Hybrid Intelligent Systems* (2020), pp. 350–359.

[40] DA, B., GUPTA, A., ONG, Y. S., AND FENG, L. Evolutionary multitasking across single and multi-objective formulations for improved problem solving. In *Proceedings of IEEE Congress on Evolutionary Computation* (2016), pp. 1695–1701.

[41] DAL PICCOL SOTTO, L. F., AND DE MELO, V. V. 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.

[42] DAL PICCOL SOTTO, L. F., DE MELO, V. V., AND BASGALUPP, M. P. λ-LGP: an improved version of linear genetic programming evaluated in the Ant Trail problem. *Knowledge and Information Systems 52*, 2 (2017), 445–465.

[43] D'ARIANO, A., PACCIARELLI, D., PISTELLI, M., AND PRANZO, M. Real-time scheduling of aircraft arrivals and departures in a terminal maneuvering area. *Networks 65*, 3 (2015), 212–227.

[44] DENZINGER, J., FUCHS, M., AND FUCHS, M. High performance ATP systems by combining several AI methods. In *Proceedings of IJCAI International Joint Conference on Artificial Intelligence* (1997), pp. 102–107.

[45] DING, J., SCHULZ, S., SHEN, L., BUSCHER, U., AND LÜ, Z. Energy aware scheduling in flexible flow shops with hybrid particle swarm optimization. *Computers and Operations Research 125* (2021), 105088.

[46] DING, J., YANG, C., JIN, Y., AND CHAI, T. Generalized Multitasking for Evolutionary Optimization of Expensive Problems. *IEEE Transactions on Evolutionary Computation 23*, 1 (2019), 44–58.

[47] DOWNEY, C., AND ZHANG, M. Caching for Parallel Linear Genetic Programming Categories and Subject Descriptors. In *Proceedings of the Annual Conference Companion on Genetic and Evolutionary Computation* (2011), no. 2, pp. 201–202.

[48] DOWNEY, C., AND ZHANG, M. Execution trace caching for Linear Genetic Programming. In *Proceedings of IEEE Congress of Evolutionary Computation,* (2011), pp. 1186–1193.

[49] DOWNEY, C., ZHANG, M., AND BROWNE, W. N. New crossover operators in linear genetic programming for multiclass object classification. In *Proceedings of the Genetic and Evolutionary Computation Conference* (2010), pp. 885–892.

[50] DOWNEY, C., ZHANG, M., AND LIU, J. Parallel Linear Genetic Programming for multi-class classification. *Genetic Programming and Evolvable Machines 13*, 3 (2012), 275–304.

[51] DURASEVIĆ, M., JAKOBOVIĆ, D., AND KNEŽEVIĆ, K. Adaptive scheduling on unrelated machines with genetic programming. *Applied Soft Computing Journal 48* (2016), 419–430.

[52] DURASEVIC, M., JAKOBOVIC, D., SCOCZYNSKI RIBEIRO MARTINS, M., PICEK, S., AND WAGNER, M. Fitness Landscape Analysis of Dimensionally-Aware Genetic Programming Featuring Feynman Equations. In *Proceedings of International Conference on Parallel Problem Solving from Nature* (2020), pp. 111–124.

[53] FAN, H., XIONG, H., AND GOH, M. Genetic programming-based hyper-heuristic approach for solving dynamic job shop scheduling problem with extended technical precedence constraints. *Computers & Operations Research 134* (2021), 105401.

[54] FAN, Q., BI, Y., XUE, B., AND ZHANG, M. A genetic programming-based method for image classification with small training data. *Knowledge-Based Systems 283* (2024), 111188.

[55] FENG, L., HUANG, Y., ZHOU, L., ZHONG, J., GUPTA, A., TANG, K., AND TAN, K. C. Explicit Evolutionary Multitasking for Combinatorial Optimization: A Case Study on Capacitated Vehicle Routing Problem. *IEEE Transactions on Cybernetics 51*, 6 (2021), 3143–3156.

[56] FENG, L., ZHOU, L., ZHONG, J., GUPTA, A., ONG, Y. S., TAN, K. C., AND QIN, A. K. Evolutionary Multitasking via Explicit Autoencoding. *IEEE Transactions on Cybernetics 49*, 9 (2019), 3457–3470.

[57] FERREIRA, C. Gene Expression Programming: a New Adaptive Algorithm for Solving Problems. *Complex Systems 13*, 2 (2001), 87–129.

[58] FOGEL, D. B., AND FOGEL, L. J. An introduction to evolutionary programming. In *Artificial Evolution* (1996), pp. 21–33.

[59] FOGELBERG, C. *Linear Genetic Programming for Multi-class Classification Problems*. PhD thesis, Victoria University of Wellington, 2005.

[60] FORSTENLECHNER, S., FAGAN, D., NICOLAU, M., AND O'NEILL, M. A Grammar Design Pattern for Arbitrary Program Synthesis Problems in Genetic Programming. In *European Conference on Genetic Programming* (2017), pp. 262–277.

[61] FORSTENLECHNER, S., FAGAN, D., NICOLAU, M., AND O'NEILL, M. Extending Program Synthesis Grammars for Grammar-Guided

Genetic Programming. In *Parallel Problem Solving from Nature – PPSN XV* (2018), pp. 197–208.

[62] FORSTENLECHNER, S., FAGAN, D., NICOLAU, M., AND O'NEILL, M. Extending program synthesis grammars for grammar-guided genetic programming. In *Proceedings of Parallel Problem Solving from Nature* (2018), pp. 197–208.

[63] FU, W., XUE, B., GAO, X., AND ZHANG, M. Genetic Programming for Document Classification: A Transductive Transfer Learning System. *IEEE Transactions on Cybernetics* (2023), 1–14. doi:10.1109/TCYB.2023.3338266.

[64] FU, X., CHAN, F. T., NIU, B., CHUNG, N. S., AND QU, T. A three-level particle swarm optimization with variable neighbourhood search algorithm for the production scheduling problem with mould maintenance. *Swarm and Evolutionary Computation 50* (2019-11), 100572.

[65] GALVÁN-LÓPEZ, E., MCDERMOTT, J., O'NEILL, M., AND BRABAZON, A. Defining locality as a problem difficulty measure in genetic programming. *Genetic Programming and Evolvable Machines 12*, 4 (2011), 365–401.

[66] GAREY, M. R., JOHNSON, D. S., AND SETHI, R. Complexity of Flowshop and Jobshop Scheduling. *Mathematics of Operations Research 1*, 2 (1976), 117–129.

[67] GEIGER, C. D., UZSOY, R., AND AYTUĞ, H. Rapid modeling and discovery of priority dispatching rules: An autonomous learning approach. *Journal of Scheduling 9* (2006), 7–34.

[68] GLIGOROVSKI, N., AND ZHONG, J. LGP-VEC: A vectorial linear genetic programming for symbolic regression. In *Proceedings of the*

Companion Conference on Genetic and Evolutionary Computation (2023), pp. 579–582.

[69] GOLDMAN, B. W., AND PUNCH, W. F. Analysis of cartesian genetic programming's evolutionary mechanisms. *IEEE Transactions on Evolutionary Computation 19* (2015), 359–373.

[70] GUPTA, A., MAŃDZIUK, J., AND ONG, Y.-S. Evolutionary multitasking in bi-level optimization. *Complex & Intelligent Systems 1*, 1-4 (2015), 83–95.

[71] GUPTA, A., ONG, Y. S., AND FENG, L. Multifactorial evolution: Toward evolutionary multitasking. *IEEE Transactions on Evolutionary Computation 20* (2016), 343–357.

[72] GUPTA, A., ONG, Y. S., FENG, L., AND TAN, K. C. Multiobjective Multifactorial Optimization in Evolutionary Multitasking. *IEEE Transactions on Cybernetics 47*, 7 (2017), 1652–1665.

[73] HART, E., AND SIM, K. A hyper-heuristic ensemble method for static job-shop scheduling. *Evolutionary Computation 24*, 4 (2016), 609–635.

[74] HAUT, N., BANZHAF, W., AND PUNCH, B. Correlation Versus RMSE Loss Functions in Symbolic Regression Tasks. In *Genetic Programming Theory and Practice XIX*, Genetic and Evolutionary Computation. Springer Nature, Singapore, 2023, pp. 31–55.

[75] HE, Y., AND NERI, F. Fitness Landscape Analysis of Genetic Programming Search Spaces with Local Optima Networks. In *Proceedings of the Companion Conference on Genetic and Evolutionary Computation* (2023), pp. 2056–2063.

[76] HELMUTH, T., SPECTOR, L., AND MATHESON, J. Solving Uncompromising Problems With Lexicase Selection. *IEEE Transactions on Evolutionary Computation 19*, 5 (2015), 630–643.

[77] HILDEBRANDT, T., AND BRANKE, J. On using surrogates with genetic programming. *Evolutionary Computation 23*, 3 (2015), 343–367.

[78] HILDEBRANDT, T., HEGER, J., AND SCHOLZ-REITER, B. Towards Improved Dispatching Rules for Complex Shop Floor Scenarios — a Genetic Programming Approach. In *Proceedings of the Annual Conference on Genetic and Evolutionary Computation* (2010), pp. 257–264.

[79] HOLLAND, J. H. Genetic Algorithms. *Scientific American 267*, 1 (1992), 66–73.

[80] HORNBY, G., GLOBUS, A., LINDEN, D., AND LOHN, J. Automated Antenna Design with Evolutionary Algorithms. In *Proceedings of Space 2006* (2006).

[81] HU, T. Genetic Programming for Interpretable and Explainable Machine Learning. In *Genetic Programming Theory and Practice XIX*. Springer Nature Singapore, 2023, pp. 81–90.

[82] HU, T., AND BANZHAF, W. Neutrality, Robustness, and Evolvability in Genetic Programming. In *Genetic Programming Theory and Practice XIV*. Springer, Cham, 2018, pp. 101–117.

[83] HU, T., BANZHAF, W., AND MOORE, J. H. Robustness and evolvability of recombination in linear genetic programming. In *European Conference on Genetic Programming* (2013), pp. 97–108.

[84] HU, T., OCHOA, G., AND BANZHAF, W. Phenotype Search Trajectory Networks for Linear Genetic Programming. In *Proceedings of European Conference on Genetic Programming* (2023), pp. 52–67.

[85] HU, T., PAYNE, J. L., BANZHAF, W., AND MOORE, J. H. Robustness, evolvability, and accessibility in linear genetic programming. In *European Conference on Genetic Programming* (2011), pp. 13–24.

[86] HU, T., PAYNE, J. L., BANZHAF, W., AND MOORE, J. H. Evolutionary dynamics on multiple scales: A quantitative analysis of the interplay between genotype, phenotype, and fitness in linear genetic programming. *Genetic Programming and Evolvable Machines 13*, 3 (2012), 305–337.

[87] HUANG, Z., MEI, Y., AND ZHONG, J. Semantic Linear Genetic Programming for Symbolic Regression. *IEEE Transactions on Cybernetics* (2022), 1–14.

[88] HUANG, Z., ZHONG, J., FENG, L., MEI, Y., AND CAI, W. A fast parallel genetic programming framework with adaptively weighted primitives for symbolic regression. *Soft Computing 24* (2020), 7523–7539.

[89] HUGHES, M., GOERIGK, M., AND DOKKA, T. Automatic generation of algorithms for robust optimisation problems using grammar-guided genetic programming. *Computers and Operations Research 133* (2021).

[90] HUNT, R., JOHNSTON, M., AND ZHANG, M. Evolving "less-myopic" scheduling rules for dynamic job shop scheduling with genetic programming. In *Proceedings of the Genetic and Evolutionary Computation Conference* (2014), pp. 927–934.

[91] HUNT, R., RICHARD, J., AND ZHANG, M. Evolving Dispatching Rules with Greater Understandability for Dynamic Job Shop Scheduling Mark Johnston Mark Johnston. Technical report ECSTR-15-6. Tech. Rep. x, Victoria University of Wellington, 2016.

[92] INGELSE, L., AND FONSECA, A. Domain-Aware Feature Learning with Grammar-Guided Genetic Programming. In *Proceedings of European Conference on Genetic Programming* (2023), pp. 227–243.

[93] INGELSE, L., HIDALGO, J.-I., COLMENAR, J. M., LOURENÇO, N., AND FONSECA, A. Comparing Individual Representations in Grammar-Guided Genetic Programming for Glucose Prediction in People with Diabetes. In *Proceedings of the Companion Conference on Genetic and Evolutionary Computation* (2023), pp. 2013–2021.

[94] JACKSON, J. R. Scheduling A Production Line to Minimize Maximum Tardiness. *Management Science Research Project* (1955).

[95] JACKSON, J. R. Simulation research on job shop production. *Naval Research Logistics Quarterly 4*, 4 (1957), 287–295.

[96] JAKOBOVIĆ, D., AND BUDIN, L. Dynamic scheduling with genetic programming. In *Proceedings of European Conference on Genetic Programming* (2006), pp. 73–84.

[97] JIAO, R., XUE, B., AND ZHANG, M. A Multiform Optimization Framework for Constrained Multiobjective Optimization. *IEEE Transactions on Cybernetics* (2022), 1–13. doi:10.1109/TCYB.2022.3178132.

[98] JONES, J., AND SOULE, T. Comparing genetic robustness in generational vs. steady state evolutionary algorithms. In *Proceedings of the 8th annual conference on Genetic and evolutionary computation* (2006), pp. 143–150.

[99] JONES, T., AND FORREST, S. Fitness Distance Correlation as a Measure of Problem Difficulty for Genetic Algorithms. In *Proceedings of the 6th International Conference on Genetic Algorithms* (1995), pp. 184–192.

[100] KANTSCHIK, W., AND BANZHAF, W. Linear-graph GP – A new GP structure. In *Proceedings of European Conference on Genetic Programming* (2002), pp. 83–92.

[101] KARUNAKARAN, D., CHEN, G., MEI, Y., AND ZHANG, M. Toward evolving dispatching rules for dynamic job shop scheduling under uncertainty. In *Proceedings of the Genetic and Evolutionary Computation Conference* (2017), pp. 282–289.

[102] KARUNAKARAN, D., MEI, Y., CHEN, G., AND ZHANG, M. Dynamic job shop scheduling under uncertainty using genetic programming. In *Intelligent and Evolutionary Systems* (2017), pp. 195–210.

[103] KARUNAKARAN, D., MEI, Y., CHEN, G., AND ZHANG, M. Active Sampling for Dynamic Job Shop Scheduling using Genetic Programming. In *Proceedings of IEEE Congress on Evolutionary Computation* (2019), pp. 434–441.

[104] KEIJZER, M., AND BABOVIC, V. Dimensionally Aware Genetic Programming. In *Proceedings of the Genetic and Evolutionary Computation Conference* (1999), pp. 1069–1076.

[105] KINNEAR, K. Fitness landscapes and difficulty in genetic programming. In *Proceedings of the First IEEE Conference on Evolutionary Computation and IEEE World Congress on Computational Intelligence* (1994), pp. 142–147.

[106] KOZA, J. R. *Genetic Programming : On the Programming of Computers By Means of Natural Selection*. Cambridge, MA, USA: MIT Press, 1992.

[107] KOZA, J. R. Genetic programming as a means for programming computers by natural selection. *Statistics and Computing 4*, 2 (1994), 87–112.

[108] KRONBERGER, G., KABLIMAN, E., KRONSTEINER, J., AND KOMMENDA, M. Extending a Physics-Based Constitutive Model using Genetic Programming, 2021. arXiv:2108.01595 [cs].

[109] KU, W. Y., AND BECK, J. C. Mixed Integer Programming models for job shop scheduling: A computational analysis. *Computers and Operations Research 73* (2016), 165–173.

[110] LABONI, N. M., SAFA, S. J., SHARMIN, S., RAZZAQUE, M. A., RAHMAN, M. M., AND HASSAN, M. M. A Hyper Heuristic Algorithm for Efficient Resource Allocation in 5G Mobile Edge Clouds. *IEEE Transactions on Mobile Computing 23*, 1 (2024), 29–41.

[111] LAMORGESE, L., AND MANNINO, C. A noncompact formulation for job-shop scheduling problems in traffic management. *Operations Research 67*, 6 (2019), 1586–1609.

[112] LARA-CÁRDENAS, E., SÁNCHEZ-DÍAZ, X., AMAYA, I., AND ORTIZ-BAYLISS, J. C. Improving hyper-heuristic performance for job shop scheduling problems using neural networks. In *Proceedings of Mexican International Conference on Artificial Intelligence* (2019), pp. 150–161.

[113] LEUNG, W. M., SAK, L. K., WONG, M. L., AND LEUNG, K. S. Applying logic grammars to induce sub-functions in genetic programming. In *Proceedings of the IEEE Conference on Evolutionary Computation* (1995), pp. 737–740.

[114] LI, X., AND GAO, L. A hybrid genetic algorithm and tabu search for multi-objective dynamic JSP. In *Effective Methods for Integrated Process Planning and Scheduling*, vol. 2. Springer Berlin Heidelberg, 2020, pp. 377–403.

[115] LI, X., AND GAO, L. Review for flexible job shop scheduling. In *Effective Methods for Integrated Process Planning and Scheduling*, vol. 2 of *Engineering Applications of Computational Methods*. Springer, Berlin, Heidelberg, 2020, pp. 17–45.

[116] LIN, J., LIU, H. L., XUE, B., ZHANG, M., AND GU, F. Multiobjective Multitasking Optimization Based on Incremental Learning. *IEEE Transactions on Evolutionary Computation 24*, 5 (2020), 824–838.

[117] LIU, F., WANG, S., HONG, Y., AND YUE, X. On the robust and stable flowshop scheduling under stochastic and dynamic disruptions. *IEEE Transactions on Engineering Management 64*, 4 (2017), 539–553.

[118] LIU, X., SHA, L., DIAO, Y., FROEHLICH, S., HELLERSTEIN, J. L., AND PAREKH, S. Online Response Time Optimization of Apache Web Server. In *Proceedings of Quality of Service* (2003), pp. 461–478.

[119] LIU, Y., DONG, H., LOHSE, N., PETROVIC, S., AND GINDY, N. An investigation into minimising total energy consumption and total weighted tardiness in job shops. *Journal of Cleaner Production 65* (2014), 87–96.

[120] LOURENÇO, N., PEREIRA, F., AND COSTA, E. SGE: A Structured Representation for Grammatical Evolution. In *Proceedings of International Conference on Artificial Evolution* (2015), pp. 136–148.

[121] LOURENÇO, N., PEREIRA, F. B., AND COSTA, E. Unveiling the properties of structured grammatical evolution. *Genetic Programming and Evolvable Machines 17*, 3 (2016), 251–289.

[122] LU, Q., XU, C., LUO, J., AND WANG, Z. AB-GEP: Adversarial bandit gene expression programming for symbolic regression. *Swarm and Evolutionary Computation 75* (2022).

[123] LUO, J., EL BAZ, D., XUE, R., AND HU, J. Solving the dynamic energy aware job shop scheduling problem with the heterogeneous parallel genetic algorithm. *Future Generation Computer Systems 108* (2020), 119–134.

[124] MACLACHLAN, J., MEI, Y., BRANKE, J., AND ZHANG, M. Genetic programming hyper-heuristics with vehicle collaboration for uncertain capacitated arc routing problems. *Evolutionary Computation 28*, 4 (2020), 563–593.

[125] MANAHOV, V., HUDSON, R., AND LINSLEY, P. New evidence about the profitability of small and large stocks and the role of volume obtained using strongly typed genetic programming. *Journal of International Financial Markets, Institutions and Money 33* (2014), 299–316.

[126] MASOOD, A., CHEN, G., MEI, Y., AL-SAHAF, H., AND ZHANG, M. Genetic Programming with Pareto Local Search for Many-Objective Job Shop Scheduling. In *Proceedings of Australasian Joint Conference on Artificial Intelligence* (2019), pp. 536–548.

[127] MASOOD, A., CHEN, G., MEI, Y., AL-SAHAF, H., AND ZHANG, M. A Fitness-based Selection Method for Pareto Local Search for Many-Objective Job Shop Scheduling. In *Proceedings of IEEE Congress on Evolutionary Computation* (2020), pp. 1–8.

[128] MASOOD, A., MEI, Y., CHEN, G., AND ZHANG, M. Many-Objective Genetic Programming for Job-Shop Scheduling. In *Proceedings of IEEE Congress on Evolutionary Computation* (2016), pp. 209–216.

[129] MASTELIC, T., OLEKSIAK, A., CLAUSSEN, H., BRANDIC, I., PIERSON, J. M., AND VASILAKOS, A. V. Cloud computing: Survey on energy efficiency. *ACM Computing Surveys 47*, 2 (2015), 1–36.

[130] MCCRACKEN, D. D., AND REILLY, E. D. *Backus-Naur form (BNF)*. John Wiley and Sons Ltd., GBR, 2003, p. 129–131.

[131] MCKAY, R. I., HOAI, N. X., WHIGHAM, P. A., SHAN, Y., AND O'NEILL, M. Grammar-based Genetic programming: A survey. *Genetic Programming and Evolvable Machines 11*, 3-4 (2010), 365–396.

[132] MCPHEE, N. F., FINZEL, M. D., CASALE, M. M., HELMUTH, T., AND SPECTOR, L. A detailed analysis of a PushGP run. In *Genetic Programming Theory and Practice XIV*, Genetic and Evolutionary Computation. Springer, Cham, 2018, pp. 65–83.

[133] MEDVET, E., AND BARTOLI, A. Evolutionary optimization of graphs with graphea. In *Proceedings of International Conference of the Italian Association for Artificial Intelligence* (2020), pp. 83–98.

[134] MEI, Y., CHEN, Q., LENSEN, A., XUE, B., AND ZHANG, M. Explainable Artificial Intelligence by Genetic Programming: A Survey. *IEEE Transactions on Evolutionary Computation* (2022), 1–1. doi: 10.1109/TEVC.2022.3225509.

[135] MEI, Y., NGUYEN, S., XUE, B., AND ZHANG, M. An Efficient Feature Selection Algorithm for Evolving Job Shop Scheduling Rules with Genetic Programming. *IEEE Transactions on Emerging Topics in Computational Intelligence 1*, 5 (2017), 339–353.

[136] MEI, Y., NGUYEN, S., AND ZHANG, M. Constrained dimensionally aware genetic programming for evolving interpretable dispatching rules in dynamic job shop scheduling. In *Proceedings of Asia-Pacific Conference on Simulated Evolution and Learning* (2017), pp. 435–447.

[137] MEI, Y., NGUYEN, S., AND ZHANG, M. Evolving time-invariant dispatching rules in job shop scheduling with genetic programming. In *Proceedings of the European Conference on Genetic Programming* (2017), pp. 147–163.

[138] MEI, Y., AND ZHANG, M. A comprehensive analysis on reusability of GP-evolved job shop dispatching rules. In *Proceedings of IEEE Congress on Evolutionary Computation* (2016), pp. 3590–3597.

[139] MICHAEL, F., DAVID, L., STEPAN, K., HOLGER, C., AND MICHAEL, O. Evolving coverage optimisation functions for heterogeneous net-

works using grammatical genetic programming. In *Proceedings of Applications of Evolutionary Computation* (2016), pp. 219–234.

[140] MICHELL, K., AND KRISTJANPOLLER, W. Generating trading rules on us stock market using strongly typed genetic programming. *Soft Computing 24* (2020), 3257–3274.

[141] MILLER, J. F. An empirical study of the efficiency of learning boolean functions using a Cartesian Genetic Programming approach. In *Proceedings of the Genetic and Evolutionary Computation Conference* (1999), pp. 1135–1142.

[142] MILLER, J. F. Cartesian genetic programming: its status and future. *Genetic Programming and Evolvable Machines 21*, 1-2 (2020), 129–168.

[143] MILLER, J. F., JOB, D., AND VASSILEV, V. K. Principles in the Evolutionary Design of Digital Circuits–Part I. *Genetic Programming and Evolvable Machines, 1*, 1 (2000), 7–35.

[144] MILLER, J. F., JOB, D., AND VASSILEV, V. K. Principles in the Evolutionary Design of Digital Circuits–Part II. *Genetic Programming and Evolvable Machines 1*, 3 (2000), 259–288.

[145] MILLER, J. F., AND SMITH, S. L. Redundancy and computational efficiency in cartesian genetic programming. *IEEE Transactions on Evolutionary Computation 10* (2006), 167–174.

[146] MILLER, J. F., WILSON, D. G., AND CUSSAT-BLANC, S. Evolving Developmental Programs That Build Neural Networks for Solving Multiple Problems. In *Genetic Programming Theory and Practice XVI. Genetic and Evolutionary Computation*. Springer, Cham, 2019, pp. 137–178.

[147] MITCHELL, M., AND TAYLOR, C. E. Evolutionary Computation: An Overview. *Annual Review of Ecology and Systematics 30*, 1 (1999), 593–616.

[148] MIYASHITA, K. Job-shop scheduling with genetic programming. In *Proceedings of the Genetic and Evolutionary Computation Conference* (2000), pp. 505–512.

[149] MLADENOVIĆ, N., AND HANSEN, P. Variable neighborhood search. *Computers & Operations Research 24* (1997), 1097–1100.

[150] MOHAN, J., LANKA, K., AND RAO, A. N. A review of dynamic job shop scheduling techniques. *Procedia Manufacturing 30* (2019), 34–39.

[151] MONTANA, D. J. Strongly typed genetic programming. *Evolutionary Computation 3*, 2 (1995), 199–230.

[152] MOUELHI-CHIBANI, W., AND PIERREVAL, H. Training a neural network to select dispatching rules in real time. *Computers & Industrial Engineering 58*, 2 (2010), 249–256.

[153] MÜLLER, F. M., AND BONILHA, I. S. Hyper-Heuristic Based on ACO and Local Search for Dynamic Optimization Problems. *Algorithms 15*, 1 (2022), 9.

[154] MUNDHENK, T. N., LANDAJUELA, M., GLATT, R., SANTIAGO, C. P., FAISSOL, D. M., AND PETERSEN, B. K. Symbolic regression via neural-guided genetic programming population seeding. In *Proceedings of Advances in Neural Information Processing Systems* (2021), pp. 24912–24923.

[155] NGUYEN, S., ERNST, A., THIRUVADY, D., AND ALAHAKOON, D. Genetic programming approach to learning multi-pass heuristics for resource constrained job scheduling. In *Proceedings of the Genetic and Evolutionary Computation Conference* (2018), pp. 1167–1174.

[156] NGUYEN, S., MEI, Y., AND ZHANG, M. Genetic programming for production scheduling: a survey with a unified framework. *Complex & Intelligent Systems 3*, 1 (2017), 41–66.

[157] NGUYEN, S., ZHANG, M., JOHNSTON, M., AND TAN, K. C. Evolving reusable operation-based due-date assignment models for job shop scheduling with genetic programming. In *Proceedings of European Conference on Genetic Programming* (2012), pp. 121–133.

[158] NGUYEN, S., ZHANG, M., JOHNSTON, M., AND TAN, K. C. 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.

[159] NGUYEN, S., ZHANG, M., JOHNSTON, M., AND TAN, K. C. Selection schemes in surrogate-assisted genetic programming for job shop scheduling. In *Proceedings of Simulated Evolution and Learning* (2014), pp. 656–667.

[160] NGUYEN, S., ZHANG, M., AND TAN, K. C. Surrogate-Assisted Genetic Programming with Simplified Models for Automated Design of Dispatching Rules. *IEEE Transactions on Cybernetics 47*, 9 (2017), 2951–2965.

[161] NIE, L., GAO, L., LI, P., AND LI, X. A GEP-based reactive scheduling policies constructing approach for dynamic flexible job shop scheduling problem with job release dates. *Journal of Intelligent Manufacturing 24*, 4 (2013), 763–774.

[162] NOON, A., KALAKECH, A., AND KADRY, S. A New Round Robin Based Scheduling Algorithm for Operating Systems: Dynamic Quantum Using the Mean Average. *International Journal of Computer Science Issues 8*, 3 (2011), 224–229.

[163] NORDIN, P. A compiling genetic programming system that directly manipulates the machine code. *Advances in genetic programming 1* (1994), 311—-331.

[164] NORDIN, P. *Evolutionary program induction of binary machine code and its applications*. PhD thesis, University of Dortmund, 1997.

[165] NORDIN, P., FRANCONE, F., AND BANZHAF, W. Explicitly Defined Introns and Destructive Crossover in Genetic Programming. In *Advances in Genetic Programming*. MIT Press, Cambridge, MA, USA, 1996, pp. 111–134.

[166] NOUIRI, M., BEKRAR, A., JEMAI, A., TRENTESAUX, D., AMMARI, A. C., AND NIAR, S. Two stage particle swarm optimization to solve the flexible job shop predictive scheduling problem considering possible machine breakdowns. *Computers & Industrial Engineering 112* (2017), 595–606.

[167] OCHOA, G., TOMASSINI, M., VÉREL, S., AND DARABOS, C. A study of NK landscapes' basins and local optima networks. In *Proceedings of the Genetic and evolutionary computation Conference* (2008), pp. 555–562.

[168] OCHOA, G., VEREL, S., DAOLIO, F., AND TOMASSINI, M. Local Optima Networks: A New Model of Combinatorial Fitness Landscapes. In *Recent Advances in the Theory and Application of Fitness Landscapes*, vol. 6. Springer Berlin Heidelberg, 2014, pp. 233–262.

[169] OLAFSSON, S., AND LI, X. Learning effective new single machine dispatching rules from optimal scheduling data. *International Journal of Production Economics 128*, 1 (2010), 118–126.

[170] OLIVEIRA, V. P. L., DE SOUZA, E. F., GOUES, C. L., AND CAMILO-JUNIOR, C. G. Improved representation and genetic operators for linear genetic programming for automated program repair. *Empirical Software Engineering 23*, 5 (2018), 2980–3006.

[171] O'NEILL, D., AL-SAHAF, H., XUE, B., AND ZHANG, M. Common subtrees in related problems: A novel transfer learning approach for

genetic programming. In *Proceedings of IEEE Congress on Evolutionary Computation* (2017), pp. 1287–1294.

[172] O'NEILL, M., AND RYAN, C. Grammatical evolution. *IEEE Transactions on Evolutionary Computation 5*, 4 (2001), 349–358.

[173] ONG, Y. S. Towards evolutionary multitasking: A new paradigm in evolutionary computation. In *Proceedings of Computational Intelligence, Cyber Security and Computational Models* (2015), pp. 25–26.

[174] PADILLO, F., LUNA, J. M., AND VENTURA, S. A Grammar-Guided Genetic Programing Algorithm for Associative Classification in Big Data. *Cognitive Computation 11*, 3 (2019), 331–346.

[175] PANDA, S., AND MEI, Y. Genetic programming with algebraic simplification for dynamic job shop scheduling. In *Proceedings of IEEE Congress on Evolutionary Computation* (2021), pp. 1848–1855.

[176] PANDA, S., MEI, Y., AND ZHANG, M. Simplifying dispatching rules in genetic programming for dynamic job shop scheduling. In *Proceedings of Evolutionary Computation in Combinatorial Optimization* (2022), pp. 95–110.

[177] PARIS, P. C., PEDRINO, E. C., AND NICOLETTI, M. C. Automatic learning of image filters using Cartesian genetic programming. *Integrated Computer-Aided Engineering 22*, 2 (2015), 135–151.

[178] PARK, J., CHUN, J., KIM, S. H., KIM, Y., AND PARK, J. Learning to schedule job-shop problems: representation and policy learning using graph neural network and reinforcement learning. *International Journal of Production Research 0*, 0 (2021), 1–18.

[179] PARK, J., MEI, Y., NGUYEN, S., CHEN, G., AND ZHANG, M. Investigating the generality of genetic programming based hyper-heuristic approach to dynamic job shop scheduling with machine breakdown.

In *Proceedings of Australasian Conference on Artificial Life and Computational Intelligence* (2017), pp. 301–313.

[180] PARK, J., MEI, Y., NGUYEN, S., CHEN, G., AND ZHANG, M. Evolutionary multitask optimisation for dynamic job shop scheduling using niched genetic programming. In *Proceedings of Australasian Joint Conference on Artificial Intelligence* (2018), pp. 739–751.

[181] PARK, J., MEI, Y., NGUYEN, S., CHEN, G., AND ZHANG, M. Investigating a machine breakdown genetic programming approach for dynamic job shop scheduling. In *Proceedings of European Conference on Genetic Programming* (2018), pp. 253–270.

[182] PASSAGGIA, P. Y., QUANSAH, A., MAZELLIER, N., MACEDA, G. Y., AND KOURTA, A. Real-time feedback stall control of an airfoil at large Reynolds numbers using linear genetic programming. *Physics of Fluids 34*, 4 (2022).

[183] PAWLAK, T. P., AND KRAWIEC, K. Competent Geometric Semantic Genetic Programming for Symbolic Regression and Boolean Function Synthesis. *Evolutionary Computation 26*, 2 (2018), 177–212.

[184] PAWLAK, T. P., AND O'NEILL, M. Grammatical evolution for constraint synthesis for mixed-integer linear programming. *Swarm and Evolutionary Computation 64* (2021), 100896.

[185] PAWLAK, T. P., WIELOCH, B., AND KRAWIEC, K. Semantic Backpropagation for Designing Search Operators in Genetic Programming. *IEEE Transactions on Evolutionary Computation 19*, 3 (2015), 326–340.

[186] PEI, W., XUE, B., SHANG, L., AND ZHANG, M. A Threshold-free Classification Mechanism in Genetic Programming for High-dimensional Unbalanced Classification. In *Proceedings of IEEE Congress on Evolutionary Computation* (2020), pp. 1–8.

[187] PEREIRA, C. S., DIAS, D. M., MARTÍ, L., AND VELLASCO, M. A multi-objective decomposition optimization method for refinery crude oil scheduling through genetic programming. In *Proceedings of the Companion Conference on Genetic and Evolutionary Computation* (2023), pp. 1972–1980.

[188] PEREIRA, C. S., DIAS, D. M., PACHECO, M. A. C., VELLASCO, M. M., ABS DA CRUZ, A. V., AND HOLLMANN, E. H. Quantum-Inspired Genetic Programming Algorithm for the Crude Oil Scheduling of a Real-World Refinery. *IEEE Systems Journal 14*, 3 (2020), 3926–3937.

[189] PFUND, M., FOWLER, J. W., GADKARI, A., AND CHEN, Y. Scheduling jobs on parallel machines with setup times and ready times. *Computers and Industrial Engineering 54*, 4 (2008), 764–782.

[190] PICKARDT, C. W., HILDEBRANDT, T., BRANKE, J., HEGER, J., AND SCHOLZ-REITER, B. Evolutionary generation of dispatching rule sets for complex dynamic scheduling problems. *International Journal of Production Economics 145*, 1 (2013), 67–77.

[191] PIETROPOLLI, G., MENARA, G., AND CASTELLI, M. A genetic programming based heuristic to simplify rugged landscapes exploration. *Emerging Science Journal 7*, 4 (2023), 1037–1051.

[192] PIRINGER, D., WAGNER, S., HAIDER, C., FOHLER, A., SILBER, S., AND AFFENZELLER, M. Improving the flexibility of shape-constrained symbolic regression with extended constraints. In *Proceedings of Computer Aided Systems Theory* (2022), pp. 155–163.

[193] PLANINIĆ, L., DURASEVIĆ, M., AND JAKOBOVIĆ, D. On the Application of ϵ-Lexicase Selection in the Generation of Dispatching Rules. In *Proceedings of IEEE Congress on Evolutionary Computation* (2021), pp. 2125–2132.

[194] PLANINIĆ, L., DURASEVIĆ, M., AND JAKOBOVIĆ, D. Towards Interpretable Dispatching Rules: Application of Expression Simplification Methods. In *Proceedings of IEEE Symposium Series on Computational Intelligence* (2021).

[195] PRIORE, P., PARREÑO, J., PINO, R., GÓMEZ, A., AND PUENTE, J. LEARNING-BASED SCHEDULING OF FLEXIBLE MANUFACTURING SYSTEMS USING SUPPORT VECTOR MACHINES. *Applied Artificial Intelligence 24*, 3 (2010), 194–209.

[196] PROVOROVS, S., AND BORISOV, A. 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.

[197] QU, S., WANG, J., AND JASPERNEITE, J. Dynamic scheduling in large-scale stochastic processing networks for demand-driven manufacturing using distributed reinforcement learning. In *Proceedings of IEEE International Conference on Emerging Technologies and Factory Automation* (2018), pp. 433–440.

[198] RAHMANI, D., AND RAMEZANIAN, R. A stable reactive approach in dynamic flexible flow shop scheduling with unexpected disruptions: A case study. *Computers and Industrial Engineering 98* (2016), 360–372.

[199] RENKE, L., PIPLANI, R., AND TORO, C. A review of dynamic scheduling: Context, techniques and prospects. In *Implementing Industry 4.0*, vol. 202. Springer International Publishing, 2021, pp. 229–258.

[200] ROSS, B. J. Logic-based genetic programming with definite clause translation grammars. *New Generation Computing 19*, 4 (2001), 313–337.

[201] ROTHLAUF, F., AND OETZEL, M. On the Locality of Grammatical Evolution. In *Proceedings of European Conference on Genetic Programming* (2006), pp. 320–330.

[202] RYAN, C., COLLINS, J., AND NEILL, M. O. Grammatical evolution: Evolving programs for an arbitrary language. In *Proceedings of European Conference on Genetic Programming* (1998), pp. 83–96.

[203] SABER, T., LYNCH, D., FAGAN, D., KUCERA, S., CLAUSSEN, H., AND O'NEILL, M. Hierarchical Grammar-Guided Genetic Programming Techniques for Scheduling in Heterogeneous Networks. In *Proceedings of IEEE Congress on Evolutionary Computation* (2020), pp. 1–8.

[204] SALLAM, K. M., CHAKRABORTTY, R. K., AND RYAN, M. J. A reinforcement learning based multi-method approach for stochastic resource constrained project scheduling problems. *Expert Systems with Applications 169*, November 2020 (2021), 114479.

[205] SANCHEZ, M., CRUZ-DUARTE, J. M., ORTIZ-BAYLISS, J. C., CEBALLOS, H., TERASHIMA-MARIN, H., AND AMAYA, I. A Systematic Review of Hyper-Heuristics on Combinatorial Optimization Problems. *IEEE Access 8* (2020), 128068–128095.

[206] SCHAUER, J., AND SCHWARZ, C. Job-shop scheduling in a body shop. *Journal of Scheduling 16*, 2 (2013), 215–229.

[207] SHIUE, Y.-R. Data-mining-based dynamic dispatching rule selection mechanism for shop floor control systems using a support vector machine approach. *International Journal of Production Research 47*, 13 (2009-07), 3669–3690.

[208] SITAHONG, A., YUAN, Y., LI, M., MA, J., BA, Z., AND LU, Y. Learning dispatching rules via novel genetic programming with feature

selection in energy-aware dynamic job-shop scheduling. *Scientific Reports 13*, 1 (2023), 8558.

[209] SLANÝ, K., AND SEKANINA, L. Fitness Landscape Analysis and Image Filter Evolution Using Functional-Level CGP. In *Proceedings of European Conference on Genetic Programming* (2007), pp. 311–320.

[210] SMITH, W. E. Various optimizers for single-stage production. *Naval Research Logistics Quarterly 3*, 1-2 (1956), 59–66.

[211] SOARES, L. C. R., AND CARVALHO, M. A. M. Biased random-key genetic algorithm for scheduling identical parallel machines with tooling constraints. *European Journal of Operational Research 285*, 3 (2020), 955–964.

[212] SOTTO, L. F. D. P., AND DE MELO, V. V. Studying bloat control and maintenance of effective code in linear genetic programming for symbolic regression. *Neurocomputing 180* (2016), 79–93.

[213] SOTTO, L. F. D. P., KAUFMANN, P., ATKINSON, T., KALKREUTH, R., AND BASGALUPP, M. P. A study on graph representations for genetic programming. In *Proceedings of Genetic and Evolutionary Computation Conference* (2020), pp. 931–939.

[214] SOTTO, L. F. D. P., KAUFMANN, P., ATKINSON, T., KALKREUTH, R., AND PORTO BASGALUPP, M. Graph representations in genetic programming. *Genetic Programming and Evolvable Machines 22*, 4 (2021), 607–636.

[215] SOTTO, L. F. D. P., AND MELO, V. V. D. Studying bloat control and maintenance of effective code in linear genetic programming for symbolic regression. *Neurocomputing 180* (2016), 79–93.

[216] SOTTO, L. F. D. P., AND ROTHLAUF, F. On the role of non-effective code in linear genetic programming. In *Proceedings of the Genetic*

and Evolutionary Computation Conference (New York, NY, USA, 2019), pp. 1075–1083.

[217] SOTTO, L. F. D. P., ROTHLAUF, F., DE MELO, V. V., AND BASGALUPP, M. P. An analysis of the influence of noneffective instructions in linear genetic programming. *Evolutionary Computation 30* (2022), 51–74.

[218] SOUZA, R. L. C., GHASEMI, A., SAIF, A., AND GHARAEI, A. Robust job-shop scheduling under deterministic and stochastic unavailability constraints due to preventive and corrective maintenance. *Computers & Industrial Engineering 168* (2022), 108130.

[219] SPECTOR, L. Autoconstructive Evolution: Push, PushGP, and Pushpop. In *Proceedings of the Genetic and Evolutionary Computation Conference* (2001), vol. 137.

[220] TANG, D., DAI, M., SALIDO, M. A., AND GIRET, A. Energy-efficient dynamic scheduling for a flexible flow shop using an improved particle swarm optimization. *Computers in Industry 81* (2016-09), 82–95.

[221] TANG, Z., GONG, M., WU, Y., LIU, W., AND XIE, Y. Regularized Evolutionary Multitask Optimization: Learning to Intertask Transfer in Aligned Subspace. *IEEE Transactions on Evolutionary Computation 25*, 2 (2021), 262–276.

[222] TENG, Y., DU, S., HONG, Z., WU, X., TIAN, Y., AND LI, D. A novel grammatical evolution algorithm for automatic design of scheduling heuristics. In *Proceedings of IEEE International Conference on Automation Science and Engineering* (2019), pp. 579–584.

[223] TOMASSINI, M., VANNESCHI, L., COLLARD, P., AND CLERGUE, M. A Study of Fitness Distance Correlation as a Difficulty Measure in Genetic Programming. *Evolutionary Computation 13*, 2 (2005), 213–239.

[224] TURNER, A. J., AND MILLER, J. F. Neutral genetic drift: an investigation using cartesian genetic programming. *Genetic Programming and Evolvable Machines 16* (2015), 531–558.

[225] ČORIĆ, R., ĐUMIĆ, M., AND JAKOBOVIĆ, D. Genetic programming hyperheuristic parameter configuration using fitness landscape analysis. *Applied Intelligence 51*, 10 (2021), 7402–7426.

[226] ĐURASEVIĆ, M., GIL-GALA, F. J., AND JAKOBOVIĆ, D. To Bias or Not to Bias: Probabilistic Initialisation for Evolving Dispatching Rules. In *Proceedings of European Conference on Genetic Programming* (2023), pp. 308–323.

[227] VANNESCHI, L. *Theory and practice for efficient genetic programming*. phdthesis, Université de Lausanne, Faculté des sciences, 2004.

[228] VANNESCHI, L. Fitness landscapes and problem hardness in genetic programming. In *Proceedings of the 12th annual conference companion on Genetic and evolutionary computation* (2010), pp. 2711–2738.

[229] VANNESCHI, L., CLERGUE, M., COLLARD, P., TOMASSINI, M., AND VÉREL, S. Fitness Clouds and Problem Hardness in Genetic Programming. In *Proceedings of Genetic and Evolutionary Computation Conference* (2004), pp. 690–701.

[230] VANNESCHI, L., AND TOMASSINI, M. Pros and Cons of Fitness Distance Correlation in Genetic Programming. In *Proceedings of Genetic and Evolutionary Computation Conference Workshop Program* (2003), pp. 284–287.

[231] VANNESCHI, L., TOMASSINI, M., CLERGUE, M., AND COLLARD, P. Difficulty of Unimodal and Multimodal Landscapes in Genetic Programming. In *Genetic and Evolutionary Computation* (2003), pp. 1788–1799.

[232] VANNESCHI, L., TOMASSINI, M., COLLARD, P., AND CLERGUE, M. Fitness distance correlation in genetic programming: a constructive counterexample. In *Proceedings of the Congress on Evolutionary Computation* (2003), pp. 289–296.

[233] VANNESCHI, L., TOMASSINI, M., COLLARD, P., AND CLERGUE, M. Fitness Distance Correlation in Structural Mutation Genetic Programming. In *Proceedings of European Conference on Genetic Programming* (2003), pp. 455–464.

[234] VANNESCHI, L., TOMASSINI, M., COLLARD, P., AND CLERGUE, M. A Survey of Problem Difficulty in Genetic Programming. In *Proceedings of the Congress of the Italian Association for Artificial Intelligence* (2005), pp. 66–77.

[235] VANNESCHI, L., TOMASSINI, M., COLLARD, P., AND VÉREL, S. Negative Slope Coefficient: A Measure to Characterize Genetic Programming Fitness Landscapes. In *Proceedings of European Conference on Genetic Programming* (2006), pp. 178–189.

[236] VEPSALAINEN, A. P., AND MORTON, T. E. Priority Rules for Job Shops With Weighted Tardiness Costs. *Management Science 33*, 8 (1987), 1035–1047.

[237] VEREL, S., COLLARD, P., AND CLERGUE, M. Where are bottlenecks in NK fitness landscapes? In *Proceedings of The 2003 Congress on Evolutionary Computation* (2003), pp. 273–280.

[238] WAN, M., WEISE, T., AND TANG, K. Novel loop structures and the evolution of mathematical algorithms. In *Proceedings of European Conference on Genetic Programming* (2011), pp. 49–60.

[239] WANG, S., MEI, Y., AND ZHANG, M. A Multi-Objective Genetic Programming Algorithm With α Dominance and Archive for Un-

certain Capacitated Arc Routing Problem. *IEEE Transactions on Evolutionary Computation 27*, 6 (2023), 1633–1647.

[240] WAPPLER, S., AND WEGENER, J. Evolutionary unit testing of object-oriented software using strongly-typed genetic programming. In *Proceedings of Annual Conference on Genetic and Evolutionary Computation* (2006), pp. 1925–1932.

[241] WEI, T., WANG, S., ZHONG, J., LIU, D., AND ZHANG, J. A Review on Evolutionary Multi-Task Optimization: Trends and Challenges. *IEEE Transactions on Evolutionary Computation 26*, 5 (2021), 941–960.

[242] WEI, T., AND ZHONG, J. A Preliminary Study of Knowledge Transfer in Multi-Classification Using Gene Expression Programming. *Frontiers in Neuroscience 13*, January (2020), 1–14.

[243] WHIGHAM, P. Grammatically-based Genetic Programming. In *Proceedings of the Workshop on Genetic Programming: From Theory to Real-World Applications* (1995), pp. 33–41.

[244] WILSON, G., AND BANZHAF, W. A comparison of cartesian genetic programming and linear genetic programming. In *Proceedings of European Conference on Genetic Programming* (2008), pp. 182–193.

[245] WONG, M. L., AND LEUNG, K. S. *Data Mining Using Grammar-Based Genetic Programming and Applications*. Kluwer Academic Publishers, USA, 2000.

[246] WU, Y., DING, H., XIANG, B., SHENG, J., AND MA, W. Evolutionary multitask optimization in real-world applications: A survey. *Journal of Artificial Intelligence and Technology* (2023).

[247] XIONG, J., XING, L., AND CHEN, Y. Robust scheduling for multi-objective flexible job-shop problems with random machine breakdowns. *International Journal of Production Economics 141*, 1 (2013), 112–126.

[248] XU, M., MEI, Y., ZHANG, F., AND ZHANG, M. Genetic programming and reinforcement learning on learning heuristics for dynamic scheduling: A preliminary comparison. *IEEE Computational Intelligence Magazine* (2023). doi:10.1109/MCI.2024.3363970.

[249] XU, M., MEI, Y., ZHANG, F., AND ZHANG, M. Genetic Programming for Dynamic Flexible Job Shop Scheduling: Evolution With Single Individuals and Ensembles. *IEEE Transactions on Evolutionary Computation* (2023), 1–15.

[250] XU, M., MEI, Y., ZHANG, F., AND ZHANG, M. A Semantic Genetic Programming Approach to Evolving Heuristics for Multi-objective Dynamic Scheduling. In *Proceedings of Advances in Artificial Intelligence* (2024), pp. 403–415.

[251] XU, M., ZHANG, F., MEI, Y., AND ZHANG, M. Genetic Programming with Multi-case Fitness for Dynamic Flexible Job Shop Scheduling. In *Proceedings of IEEE Congress on Evolutionary Computation* (2022), pp. 01–08.

[252] XU, Q., WANG, N., WANG, L., LI, W., AND SUN, Q. Multi-task optimization and multi-task evolutionary computation in the past five years: A brief review. *Mathematics 9*, 8 (2021), 1–44.

[253] YI, J., BAI, J., HE, H., ZHOU, W., AND YAO, L. A Multifactorial Evolutionary Algorithm for Multitasking under Interval Uncertainties. *IEEE Transactions on Evolutionary Computation 24*, 5 (2020), 908–922.

[254] ZEITRÄG, Y., FIGUEIRA, J. R., HORTA, N., AND NEVES, R. Surrogate-assisted automatic evolving of dispatching rules for multi-objective dynamic job shop scheduling using genetic programming. *Expert Systems with Applications 209* (2022), 118194.

[255] ZHANG, F., MEI, Y., NGUYEN, S., TAN, K. C., AND ZHANG, M. Instance rotation based surrogate in genetic programming with brood recombination for dynamic job shop scheduling. *IEEE Transactions on Evolutionary Computation* (2022), 1–15.

[256] ZHANG, F., MEI, Y., NGUYEN, S., TAN, K. C., AND ZHANG, M. Multitask genetic programming-based generative hyperheuristics: A case study in dynamic scheduling. *IEEE Transactions on Cybernetics 52*, 10 (2022), 10515–10528.

[257] ZHANG, F., MEI, Y., NGUYEN, S., TAN, K. C., AND ZHANG, M. Multitask genetic programming-based generative hyperheuristics: A case study in dynamic scheduling. *IEEE Transactions on Cybernetics 52* (2022), 10515–10528.

[258] ZHANG, F., MEI, Y., NGUYEN, S., AND ZHANG, M. Genetic programming with adaptive search based on the frequency of features for dynamic flexible job shop scheduling. In *Proceedings of European Conference on Evolutionary Computation in Combinatorial Optimization* (2020), pp. 214–230.

[259] ZHANG, F., MEI, Y., NGUYEN, S., AND ZHANG, M. Guided Subtree Selection for Genetic Operators in Genetic Programming for Dynamic Flexible Job Shop Scheduling. In *Proceedings of European Conference on Genetic Programming* (2020), pp. 262–278.

[260] ZHANG, F., MEI, Y., NGUYEN, S., AND ZHANG, M. A preliminary approach to evolutionary multitasking for dynamic flexible job shop scheduling via genetic programming. In *Proceedings of the Genetic and Evolutionary Computation Conference Companion* (2020), pp. 107–108.

[261] ZHANG, F., MEI, Y., NGUYEN, S., AND ZHANG, M. Evolving Scheduling Heuristics via Genetic Programming With Feature Se-

lection in Dynamic Flexible Job-Shop Scheduling. *IEEE Transactions on Cybernetics 51*, 4 (2021), 1797–1811.

[262] ZHANG, F., MEI, Y., NGUYEN, S., AND ZHANG, M. Evolving Scheduling Heuristics via Genetic Programming with Feature Selection in Dynamic Flexible Job-Shop Scheduling. *IEEE Transactions on Cybernetics 51*, 4 (2021), 1797–1811.

[263] ZHANG, F., MEI, Y., NGUYEN, S., AND ZHANG, M. Collaborative Multifidelity-Based Surrogate Models for Genetic Programming in Dynamic Flexible Job Shop Scheduling. *IEEE Transactions on Cybernetics 52*, 8 (2022), 8142–8156.

[264] ZHANG, F., MEI, Y., NGUYEN, S., AND ZHANG, M. Multitask Multi-objective Genetic Programming for Automated Scheduling Heuristic Learning in Dynamic Flexible Job Shop Scheduling. *IEEE Transactions on Cybernetics* (2022). doi:10.1109/TCYB.2022.3196887.

[265] ZHANG, F., MEI, Y., NGUYEN, S., AND ZHANG, M. Multitask Multiobjective Genetic Programming for Automated Scheduling Heuristic Learning in Dynamic Flexible Job-Shop Scheduling. *IEEE Transactions on Cybernetics* (2022), 1–14. doi:10.1109/TEVC.2023.3263871.

[266] ZHANG, F., MEI, Y., NGUYEN, S., AND ZHANG, M. Survey on genetic programming and machine learning techniques for heuristic design in job shop scheduling. *IEEE Transactions on Evolutionary Computation 28*, 1 (2023), 147–167.

[267] ZHANG, F., MEI, Y., NGUYEN, S., ZHANG, M., AND TAN, K. C. Surrogate-Assisted Evolutionary Multitasking Genetic Programming for Dynamic Flexible Job Shop Scheduling. *IEEE Transactions on Evolutionary Computation 25*, 4 (2021), 651–665.

[268] ZHANG, F., MEI, Y., NGUYEN, S., ZHANG, M., AND TAN, K. C. Surrogate-Assisted Evolutionary Multitasking Genetic Programming for Dynamic Flexible Job Shop Scheduling. *IEEE Transactions on Evolutionary Computation 25*, 4 (2021), 651–665.

[269] ZHANG, F., MEI, Y., AND ZHANG, M. Genetic programming with multi-tree representation for dynamic flexible job shop scheduling. In *Proceedings of Australasian Joint Conference on Artificial Intelligence* (2018), pp. 472–484.

[270] ZHANG, F., MEI, Y., AND ZHANG, M. Surrogate-assisted genetic programming for dynamic flexible job shop scheduling. In *Proceedings of Advances in Artificial Intelligence* (2018), pp. 766–772.

[271] ZHANG, F., MEI, Y., AND ZHANG, M. A two-stage genetic programming hyper-heuristic approach with feature selection for dynamic flexible job shop scheduling. In *Proceedings of the 2019 Genetic and Evolutionary Computation Conference* (2019), pp. 347–355.

[272] ZHANG, F., MEI, Y., AND ZHANG, M. An Investigation of Terminal Settings on Multitask Multi-objective Dynamic Flexible Job Shop Scheduling with Genetic Programming. In *Proceedings of the Companion Conference on Genetic and Evolutionary Computation* (2023), pp. 259–262.

[273] ZHANG, F., NGUYEN, S., MEI, Y., AND ZHANG, M. *Genetic Programming for Production Scheduling - An Evolutionary Learning Approach*. Springer, Singapore, 2021.

[274] ZHANG, G., SHAO, X., LI, P., AND GAO, L. An effective hybrid particle swarm optimization algorithm for multi-objective flexible job-shop scheduling problem. *Computers and Industrial Engineering 56*, 4 (2009), 1309–1318.

[275] ZHANG, H., CHEN, Q., XUE, B., BANZHAF, W., AND ZHANG, M. A Double Lexicase Selection Operator for Bloat Control in Evolutionary Feature Construction for Regression. In *Proceedings of the Genetic and Evolutionary Computation Conference* (2023), pp. 1194–1202.

[276] ZHANG, H., ZHOU, A., CHEN, Q., XUE, B., AND ZHANG, M. SR-Forest: A Genetic Programming based Heterogeneous Ensemble Learning Method. *IEEE Transactions on Evolutionary Computation* (2023), 1–1. doi: 10.1109/TEVC.2023.3243172.

[277] ZHANG, J., DING, G., ZOU, Y., QIN, S., AND FU, J. Review of job shop scheduling research and its new perspectives under Industry 4.0. *Journal of Intelligent Manufacturing 30*, 4 (2019), 1809–1830.

[278] ZHANG, Y., BAI, R., QU, R., TU, C., AND JIN, J. A deep reinforcement learning based hyper-heuristic for combinatorial optimisation with uncertainties. *European Journal of Operational Research 300*, 2 (2022), 418–427.

[279] ZHAO, M., LI, X., GAO, L., WANG, L., AND XIAO, M. An improved q-learning based rescheduling method for flexible job-shops with machine failures. In *Proceedings of IEEE International Conference on Automation Science and Engineering* (2019), pp. 331–337.

[280] ZHENG, X., QIN, A. K., GONG, M., AND ZHOU, D. Self-Regulated Evolutionary Multitask Optimization. *IEEE Transactions on Evolutionary Computation 24*, 1 (2020), 16–28.

[281] ZHONG, J., FENG, L., CAI, W., AND ONG, Y. S. Multifactorial Genetic Programming for Symbolic Regression Problems. *IEEE Transactions on Systems, Man, and Cybernetics: Systems 50*, 11 (2020), 4492–4505.

[282] ZHONG, J., FENG, L., CAI, W., AND ONG, Y. S. Multifactorial Genetic Programming for Symbolic Regression Problems. *IEEE Trans-*

actions on Systems, Man, and Cybernetics: Systems 50, 11 (2020), 4492–4505.

[283] ZHONG, J., HUANG, Z., FENG, L., DU, W., AND LI, Y. A hyper-heuristic framework for lifetime maximization in wireless sensor networks with a mobile sink. *IEEE/CAA Journal of Automatica Sinica 7*, 1 (2020), 223–236.

[284] ZHONG, J., ONG, Y.-S., AND CAI, W. Self-Learning Gene Expression Programming. *IEEE Transactions on Evolutionary Computation 20*, 1 (2016), 65–80.

[285] ZHOU, L., FENG, L., ZHONG, J., ONG, Y. S., ZHU, Z., AND SHA, E. Evolutionary multitasking in combinatorial search spaces: A case study in capacitated vehicle routing problem. In *Proceedings of IEEE Symposium Series on Computational Intelligence* (2016), pp. 1–8.

[286] ZHOU, Y., JUN YANG, J., AND HUANG, Z. Automatic design of scheduling policies for dynamic flexible job shop scheduling via surrogate-assisted cooperative co-evolution genetic programming. *International Journal of Production Research 58*, 9 (2020), 2561–2580.

[287] ZHU, L., ZHANG, F., ZHU, X., CHEN, K., AND ZHANG, M. Sample-Aware Surrogate-Assisted Genetic Programming for Scheduling Heuristics Learning in Dynamic Flexible Job Shop Scheduling. In *Proceedings of the Genetic and Evolutionary Computation Conference* (2023), pp. 384–392.