symmetry

MDPI

Article

Automatic Design of Efficient Heuristics for Two-Stage Hybrid Flow Shop Scheduling

Lingxuan Liu [1,*] and Leyuan Shi [2]

[1] Academy for Advanced Interdisciplinary Studies, Peking University, Beijing 100871, China
[2] Department of Industrial & Systems Engineering, University of Wisconsin-Madison, Madison, WI 53706, USA; leyuan@engr.wisc.edu
* Correspondence: liulingxuan@pku.edu.cn

Abstract: This paper addresses the two-stage hybrid flow shop scheduling problem with a batch processor in the first stage and a discrete processor in the second stage. Incompatible job families and limited buffer size are considered. This hybrid flow shop configuration commonly appears in manufacturing operations and the batch processor is always the bottleneck which breaks the symmetry of processing time. Since making a real-time high-quality schedule is challenging, we focus on the automatic design of efficient heuristics for this two-stage problem based on the genetic programming method. We develop a hyper-heuristic approach to automate the tedious trial-and-error design process of heuristics. The goal is to generate efficient dispatching rules for identifying complete schedules to minimize the total completion time. A genetic programming with cooperative co-evolution approach is proposed to evolve the schedule policy automatically. Numerical results demonstrate that the proposed approach outperforms both the constructive heuristic and meta-heuristic algorithms, and is capable of producing high-quality schedules within seconds.

Keywords: batch processor; cooperative co-evolution; dispatching rules; genetic programming; hybrid flow shop scheduling

Citation: Liu, L.; Shi, L. Automatic Design of Efficient Heuristics for Two-Stage Hybrid Flow Shop Scheduling. *Symmetry* **2022**, *14*, 632. https://doi.org/10.3390/sym14040632

Academic Editors: Deming Lei and Ming Li

Received: 7 March 2022
Accepted: 17 March 2022
Published: 22 March 2022

1. Introduction

Most studies on job scheduling problems have addressed manufacturing systems that only consist of various discrete processors. However, there are many existing manufacturing systems that are comprised of both a discrete processor and a batch processor, where the batch processor can process several jobs simultaneously. This can be seen in many industries, such as the semiconductor industry, the aerospace industry, the auto-mobile industry, the steel industry, and the ceramics industry. For example, semiconductor manufacturing involves numerous batch processing operations such as diffusion, etching and burn-in operation of devices, followed by many non-batching operations such as planarization and ion implantation.

In these manufacturing systems, the batch processor is always the bottleneck due to its expensive and time-consuming characteristics [1], which break the processing time symmetry of successive operations of a flow shop. Identifying effective production schedules can be quite beneficial for such systems.

However, owing to the distinguishing characteristic of the batch processor and the cooperation with the discrete processor, making a production schedule of this system should address at least three aspects, namely, the batch formation, batch sequencing and job sequencing of the discrete processor. Multiple decision points significantly increase the complexity of the decision-making logic. Hence, an efficient and effective scheduling algorithm is urged to help with better decision-making.

In this paper, we investigate a two-stage hybrid flow shop scheduling problem with a batch processor and a uni-capacity discrete processor in the first and second stage,

respectively. This configuration is derived from an aerospace equipment manufacturer, but is a general production unit of many complex manufacturing environments.

A size-limited buffer (LB) is considered between two processors; this is more practical because many factories have limited space in their manufacturing room and minimal storage space. We also consider the incompatible job families (IF), which disallow jobs from the different job families to be formed into a batch because of different chemical or temperature requirements or other technique parameters. The objective is to minimize the total completion time. We denote this problem as $\beta \rightarrow \delta|LB, IF|\sum C_i$ based on the three-field notation of [2], where β and δ denote a batch processor and a discrete processor, respectively, and $\rightarrow$ represents the sequence of processors. This paper studies exactly the same problem as [3], but proposes a more effective and efficient algorithm which solves the original problem better and is capable of solving large scale problems for real-time scheduling.

Since the $\beta \rightarrow \delta|LB, IF|\sum C_i$ problem reduces to the $\beta \rightarrow \delta|\sum C_i$ problem, which is shown to be NP-complete [4], the problem studied in this paper is also NP-complete and exact methods are hardly able to solve the problem with large instances in a reasonable time. This usually leads to the development of heuristics, which can provide not necessarily optimal but practical acceptable solutions in a short time. However, the design of a heuristic could be a tedious trial-and-error process and the manually designed heuristic may become no longer efficient when a problem changes, e.g., the shortest processing time (SPT) rule is efficient for the criterion of C_{max} but less efficient for T_{max}. Fortunately, this process can be automated by a hyper-heuristic which is an optimization method searching on the space of heuristic.

Genetic Programming (GP) is one of the most prevalent approaches in the domain of evolutionary computing. Although it was initially proposed for searching optimal small computer programs, many recent studies have demonstrated that GP is highly suitable for automatically evolving heuristics in the form of the dispatching rule [5].

Motivated by these, this paper aims to develop a GP-based hyper-heuristic to automate the process of designing reusable dispatching rules, which can be applied for solving new problem instances once they are generated, for solving the problem of $\beta \rightarrow \delta|LB, IF|\sum C_i$.

The main contributions of this paper are summarized as follows:

- We formally formulate and decompose the $\beta \rightarrow \delta|LB, IF|\sum C_i$ problem into three parts, and design an efficient greedy dealing dispatching rules (GDDR) heuristic of which the time complexity is $\mathcal{O}(n^2)$.
- On the basis of GDDR, we firstly design a cooperative co-evolutionary genetic programming (GDDR-CCGP) to characterize different decision points of the problem with three sub-populations, and generate dispatching rules for each decision point while simultaneously exploring the synergistic relationship between dispatching rules by adopting the proposed two-trial evaluation scheme.
- Numerical results demonstrate that the proposed GDDR-CCGP algorithm significantly outperforms the state-of-the-art meta-heuristic algorithms in terms of both solution quality and efficiency.
- We also discuss the importance of three decision points of the two-stage hybrid flow shop scheduling problem, which provides new insights for the design of dispatching rules.

The problem studied in this paper originated from an aerospace equipment manufacturer, where heat-treatment is one of the most important bottleneck operations. Our proposed algorithm can be applied to manufacturers suffering from the bottleneck of batch processing operations such as gear manufacture and semiconductor manufacture. The complete scheduling policy generated by the proposed GDDR-CCGP algorithm can be embedded into the ERP or APS system of manufacturing enterprises to directly guide production. High-quality dispatching rule based scheduling policies can quickly respond to complex dynamic changes of production environment by enabling real-time online decision-making. Moreover, managers can learn useful knowledge from the dispatching

rules generated by our algorithm and incorporate the knowledge into manually designed heuristics, i.e., learn from machine learning.

The rest of this paper is organized as follows. A literature review is conducted in Section 2. Section 3 introduces the description and mathematics model of the problem. In Section 4, the greedy dealing dispatching rules (GDDR) heuristic scheme is proposed. In Section 5, we develop a cooperative co-evolution genetic programming based hyper-heuristic to generate dispatching rule combinations which are utilized by the GDDR heuristic. Computational experiments are conducted in Section 6. Variants of the proposed hyper-heuristic are compared and discussed in Section 7. Finally, we provide our conclusions and future works in Section 8.

2. Literature Review

In this section, we conduct a literature review from two perspectives. One perspective is the existing research on the two-stage hybrid flow-shop scheduling problem. The other perspective is the related works about applying GP to generate a production schedule policy.

Ahmadi et al. [4] studied two-stage hybrid flow-shop scheduling problems with variants of configurations including $\beta \rightarrow \delta$, $\delta \rightarrow \beta$ and $\beta_1 \rightarrow \beta_2$, where both objectives of C_{max} and $\sum C_i$ were considered. In the context of $\beta \rightarrow \delta | \sum C_i$, they proposed a Full Batch-Dealing-SPT Policy to find near optimal schedules. However, their research assumed an identical family and an infinite buffer size. In addition, Su et al. [6] considered the size-limited buffer into the $\beta \rightarrow \delta$ problem, and proposed a heuristic to solve the problem of $\beta \rightarrow \delta | C_{max}$, thus, a branch-and-bound algorithm was developed as a benchmark. Computational results showed that the heuristic can provide high quality solutions for large instances up to 400 jobs. In their research, processing times of the batching operations were considered as identical. Afterwards, Fu et al. [7] considered both incompatible families and size-limited buffer in the problem of $\beta \rightarrow \delta | \overline{C}$, where buffer size h was equal to the physical capacity of batch processor b. They proposed a differential evolution-based algorithm for the batch sequencing phase, the full batch dealing policy of [4] and the first-in-first-out rule were applied in the batch formation phase and job sequencing phase, respectively. In addition, two heuristic algorithms and a lower bound were developed for performance evaluation. Shi et al. [8] considered the same problem with a larger buffer and proposed a greedy shortest processing time heuristic algorithm. Zhang et al. [3] extended the work of [8] and developed a recycled differential evolution approach (RDE) for the batch sequencing phase. A composite lower bound was proposed to evaluate the efficiency of RDE, and results showed that, in almost all instances, RDE performs significantly better than the differential evolution-based algorithm proposed by [7]. However, RDE requires much more time for large-scale problem instances. Mauluddin [9] addressed a three-stage flow shop scheduling problem of $\delta \rightarrow \beta \rightarrow \delta | \sum C_i$, they developed a mixed integer linear programming model with analysis, but algorithms were not yet proposed.

Hyper-heuristic is essentially a search methodology upon the space of heuristic, it selects or generates a heuristic for computational search problems rather than solving problems directly. In this sense, a hyper-heuristic usually incorporates a meta-heuristic or learning strategy to enhance the generation or selection efficiency during the searching process. For example, genetic algorithm [10,11] and tabu search [12] have been applied for searching a heuristic. A well-designed hyper-heuristic can yield more reusable and generally applicable heuristics than many meta-heuristics, such hyper-heuristics have been successfully applied to many problem domains, such as production scheduling, cutting and packing [13], vehicle routing [14] and timetabling [15]. The most recent and comprehensive review related to hyper-heuristics can be found in [5]. In this paper, we employ the genetic programming as a hyper-heuristic to generate dispatching rules for production scheduling problems.

In the context of GP, most work discussed in this paper was based on [16] and employed the expression tree representation. Miyashita et al. [17] employed GP to evolve

dispatching rules for the first time, they evolved dispatching rules for two predetermined classes (bottlenecks and non-bottlenecks) of machines separately. Then, Dimopoulos and Zalzala [18] investigated a single machine total tardiness problem and experiment results showed that the GP-generated heuristic is superior to three manually designed heuristics. In addition, Geiger et al. [19] developed a scheduling rule discovery and parallel learning system to evolve dispatching rules for a variety of single machine problems. It has been shown that the proposed approach can evolve competitive or better dispatching rules than benchmark rules. They also extended their work to a two-machine flow shop problem, where they simultaneously evolved dispatching rules for each machine. As a result, two separate dispatching rules composed a scheduling policy. By employing the same approach, Geiger and Uzsoy [20] evolved dispatching rules for the single batch processor problem with incompatible job families. Instead of evolving dispatching rules for the batch formation phase and batch sequencing phase separately, they dealt with the two phases with a single dispatching rule. Hildebrandt et al. [21] studied a complex dynamic job shop scheduling problem with the objective function of minimizing cycle time. They evolved a unique sequencing rule to use on every machine and results showed that the rule evolved by GP outperformed the manually designed rules. However, the batch formation rule was not considered. Most recently, Shi et al. [22] proposed a ranking-and-selection-based genetic programming for the dynamic job shop scheduling problem. They formulated the evaluation problem as a ranking and selection problem and proposed a novel evaluation strategy to enhance the search efficiency of GP. A comprehensive review of GP can be found in [23].

In terms of applying GP to problems with multiple decision points, related works are rare. Park et al. [24] studied the order acceptance and scheduling problem, they proposed five GPs, in which dual genetic programming was included. The dual genetic programming consists of two specialized GPs to handle the order acceptance decision and the job sequencing decision separately. However, it performs the worst among the five GPs, which suggests that there is neither significant advantage nor necessity to separate the training of the two decision points. Nguyen et al. [25] proposed a diversified multi-objective cooperative co-evolution (DMOCC) to evolve scheduling policies for the dynamic multi-objective job shop scheduling problem. The DMOCC comprises two GPs, one for evolving due date assignment rules and the other evolves job sequencing rules. They showed that DMOCC performs better than other search strategies in their research. Yska et al. [26] investigated the dynamic flexible job shop scheduling problem by adopting the genetic programming with cooperative co-evolution. They co-evolved the routing rule and sequencing rule simultaneously. Experiment results showed that the proposed algorithm could evolve promising schedule policies that performed much better than the GP only sequencing rule with a fixed routing rule.

To the best of our knowledge, there is no existing research simultaneously evolving heuristics for both the batch processor and single capacity processor, as well as the problem of $\beta \rightarrow \delta|LB, IF|\sum C_i$.

3. Problem Description

In this section, some notations are introduced and the two-stage hybrid flow shop scheduling problem of $\beta \rightarrow \delta|LB, IF|\sum C_i$ is described.

Figure 1 describes the manufacturing configuration of the problem we studied. In our problem, there are n jobs belonging to m job families. All jobs are of the same size and each job only occupies one capacity unit of the batch or the buffer. Each batch of family k can only contain up to b jobs from the same family k and has a family dependent batch processing time q_k. Following the full batch property proposed by [27], we assume that the number of jobs of each family, as well as the buffer size, is an integral multiple of batch capacity. This assumption is also based on the reality that, as the batch processor is always the bottleneck, there are enough jobs for full batches of a family and schedulers are trying to make full batches as possible as they can. Whenever a job needs to be processed, it goes

through the batch and discrete processors sequentially. Before a job is processed by the batch processor, it has to be formed into a batch, then the batch is processed by the batch processor. After that, if the empty space of the buffer is greater than batch capacity b, the batch is released to the buffer; otherwise, the batch is blocked in the batch processor until there is enough buffer space for it. The job can be processed by the discrete processor in arbitrary sequence without preemption. The goal is to find a complete schedule which consists of batch formation, batch sequencing and job sequencing decisions, to minimize the total completion time of all jobs. Based on the description, the decision variables and problem formulation are given below.

Constraint (1) defines the objective function of minimizing total completion time. Constraints (2) and (3) guarantee a job can be assigned to only one batch and a batch can handle up to b jobs. Constraints (4) and (5) ensure a batch belongs to only one job family and the number of jobs of a family is an integral multiple of the batch capacity. The constraint of incompatible job family is captured by Constraint (6). Constraints (7) and (8) restrict the start time and release time of a batch to ensure that it is reasonable. Constraint (9) denotes the batch processing time. Constraints (10)–(13) disallow a batch to be released from the batch processor until there are at least b spaces in the buffer. Constraints (14) and (15) ensure that jobs can be processed by the discrete processor only when it has been released from the batch processor and all jobs in front of it are completed. Constraints (16)–(19) define the range of decision variables. During our preliminary experiments, only small instances of this model can be solved by the CPLEX 12.6.3 within one hour, we do not report these results in this paper. In the rest of this paper, we aim to propose a genetic programming-based hyper-heuristic for automatically generating a heuristic to solve this problem efficiently.

$$\min \sum_{j\in N} C_j \tag{1}$$

$$\text{s.t.} \quad \sum_{l\in B} x_{il} = 1, \qquad \forall i \in N, \tag{2}$$

$$\sum_{i\in N} x_{il} = b, \qquad \forall l \in B, \tag{3}$$

$$\sum_{k\in F} y_{kl} = 1, \qquad \forall l \in B, \tag{4}$$

$$\sum_{l\in B} y_{kl} = \frac{n_k}{b}, \qquad \forall k \in F, \tag{5}$$

$$x_{il} \le y_{kl}, \qquad \forall k \in F, i \in F_k, l \in B, \tag{6}$$

$$r_l - t_l \ge 0, \qquad \forall l \in B, \tag{7}$$

$$r_{l+1} - t_{l+1} \ge r_l, \qquad \forall l \in B, l \ne a, \tag{8}$$

$$t_l \ge q_k y_{kl}, \qquad \forall k \in F, l \in B, \tag{9}$$

$$u_{ij} + u_{ji} = 1, \qquad \forall i, j \in N, i < j, \tag{10}$$

$$\sum_{j\in N, j\ne i} u_{ji} + 1 \le \beta + M_1(1 - z_{li}), \qquad \forall l \in B, l \ge h+1, i \in N, \tag{11}$$

$$\sum_{j\in N, j\ne i} u_{ji} + 1 \ge \beta + \varepsilon + [1 - \beta - \varepsilon] z_{li}, \qquad \forall l \in B, l > h, i \in N, \tag{12}$$

$$r_l \ge C_i - p_i - M_2(1 - z_{li}), \qquad l \in B, l > h, i \in N \tag{13}$$

$$C_i - p_i \ge C_j - M_2 u_{ij}, \qquad \forall i, j \in N, i \ne j, \tag{14}$$

$$C_i - p_i \ge r_l - M_3(1 - x_{il}), \qquad \forall i \in N, l \in B, \tag{15}$$

$$u_{ij} \in \{0, 1\}, \qquad \forall i, j \in N, i \ne j, \tag{16}$$

$$y_{kl} \in \{0, 1\}, \qquad \forall k \in F, l \in B, \tag{17}$$

$$x_{il} \in \{0, 1\}, \qquad \forall i \in N, l \in B, \tag{18}$$

$$z_{li} \in \{0, 1\}, \qquad \forall i \in N, l \in B. \tag{19}$$

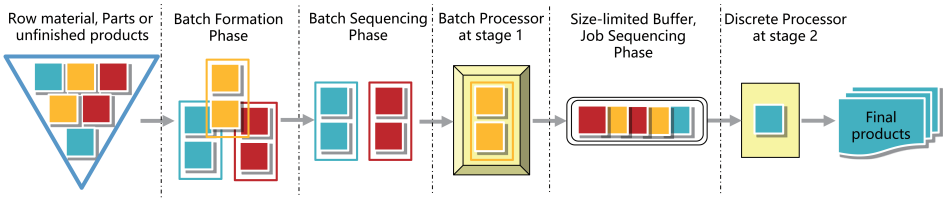

Figure 1. An illustration of the two-stage hybrid flow shop with incompatible job families and size limited buffer constraints.

4. Heuristic Design

4.1. Problem Decomposition

The dispatching rule is a simple heuristic whereby, whenever a machine is idle or a decision needs to be made, it calculates a priority for each queueing job and selects the job with the highest priority for processing. To take advantage of the dispatching rules, we decompose the problem into three sub-parts by the nature of the problem, namely, the batch formation, batch sequencing and job sequencing of the discrete processor. Thus, three dispatching rules are required for each decision point.

Specifically, the batch formation dispatching rule (BFDR) is applied to form jobs into batches, then a batch sequencing dispatching rule (BSDR) is utilized whenever the batch processor is idle, to decide the next batch to be processed on the batch processor. After that, whenever the discrete processor is idle, the job sequencing dispatching rule (JSDR) takes the responsibility of deciding the next job to be processed on the discrete processor.

For the sake of convenience, a schedule policy of our problem can be written as the combination of BFDR-BSDR-JSDR to represent the separate dispatching rules of three decision points clearly. For instance, FB-BSDR-SPT is a schedule policy that employs the full batch policy (FB) for batch formation, a dispatching rule (BSDR) for batch sequencing and the shortest processing time (SPT) rule for job sequencing.

4.2. Greedy Dealing Dispatching Rules Heuristic

In this section, we propose the greedy dealing dispatching rules (GDDR) heuristic for the problem of $\beta \rightarrow \delta|LB, IF|\sum C_i$, in which, three dispatching rules are required for a decision point. The GDDR algorithm essentially provides the heuristic structure for solving the two-stage hybrid flow shop problems and illustrates the utilization of each dispatching rule. The detailed procedure of the GDDR heuristic is described as follows:

The time complexity of GDDR is $\mathcal{O}(n^2)$ and it generates a set of batches, a schedule of these batches on the batch processor and a schedule (with the completion time) of jobs processed on the discrete processor. Then we can calculate the objective (in this study, the total completion time) of the schedule.

Since the three dispatching rules in the GDDR can be any suitable existing dispatching rules, which dispatching rules should be used and how they cooperate with each other become difficult problems. In the next section, we deal with these problems by proposing a genetic programming-based hyper-heuristic to automatically generate dispatching rules for the GDDR algorithm.

Greedy Dealing Dispatching Rules (GDDR)
Step 0. Input three dispatching rules (BFDR, BSDR and JSDR) for each decision point; set job family index $k = 1$.
Step 1. For job family k, sort jobs $j \in F_k$ in the non-decreasing order of processing time on the discrete machine p_j, i.e., $p_{[1]} \leq p_{[2]} \leq \ldots \leq p_{[n_k]}$. Denote this job sequence as π_k and initialize a_k empty batches. Set job index $i = 1$ and batch index $b = 1$, where $i = 1, 2, \ldots, n_k$ and $b = 1, 2, \ldots, a_k$.
Step 2. Greedily try to assign the ith job of π_k to each of a_k, batches that still have space remain, and calculate a priority for each possible batch assignment by applying BFDR;
Step 3. Assign the ith job of π_k to the batch with the highest priority, break ties arbitrarily. If $i < n_k$, set $i = i + 1$ and go to *Step 2*; otherwise, go to *Step 4*.
Step 4. If $k < m$, set $k = k + 1$ and go to *Step 1*; otherwise, go to *Step 5*;
Step 5. Whenever the batch processor is idle, calculate priorities for all batches queueing in front of the batch processor by applying BSDR;
Step 6. Select the batch with the highest priority to process on the batch processor and remove this batch from the queue.
Step 7. Whenever the discrete processor is idle, calculate priorities for all queueing jobs in the buffer by applying JSDR.
Step 8. Select the job with the highest priority to process on the discrete processor and remove this job from the buffer.
Step 9. Repeat *Steps 5–8* until all jobs are finished

5. Genetic Programming-Based Hyper-Heuristic

The three dispatching rules of GDDR can be generated by any problem-independent hyper-heuristic, such as GP, gene expression programming or neural network. Owing to the variable length encoding of expression tree representation, manually designed heuristics and domain knowledge can be easily expressed by GP [28], indicating that experts' existing years of experience are available for GP. In this section, we propose a GP-based hyper-heuristic, where dispatching rules are utilized according to the GDDR algorithm and evaluated by a cooperative co-evolution strategy. Intuitively, the proposed algorithm is named as the GDDR-based cooperative co-evolutionary genetic programming (GDDR-CCGP). We first briefly introduce the basic GP used in the GDDR-CCGP, and then propose the GDDR-CCGP algorithm.

5.1. Genetic Programming

Genetic programming is an evolutionary computing method based on population evolution. A population of GP consists of a number of individuals, each individual is represented as an expression tree. As a hyper-heuristic, each individual corresponds to a dispatching rule. The leaf nodes of the expression tree are filled up with job attributes (terminals), e.g., job processing time, whereas other non-leaf nodes are filled up with functions including, but not limited to, arithmetic operators. As a simple example, the SPT rule can be represented as an expression tree with a negative operator in the root node and the processing time (PT) in the leaf node, resulting in an expression of $0 - PT$.

Individuals of GP are evolved by employing genetic operators, which include the sub-tree crossover, point mutation and ramped-half-and-half tree builder [16]. The sub-tree crossover generates new individuals by randomly selecting two nodes from two parents, then swapping the sub-trees of these two nodes while guaranteeing the maximum depth constraint is not violated; otherwise, it tries other nodes. Figure 2 presents an illustration of the sub-tree crossover operator, where the SPT rule and the minimum slack rule are selected as parents. As for point mutation, similar to sub-tree crossover, two random nodes are selected from parents, but replace the sub-tree of these two nodes with new generated trees by performing the ramped-half-and-half tree builder. Similarly, the maximum depth constraint cannot be violated; otherwise, it will try to generate another sub-tree. The parent individuals are randomly selected by applying the tournament selection.

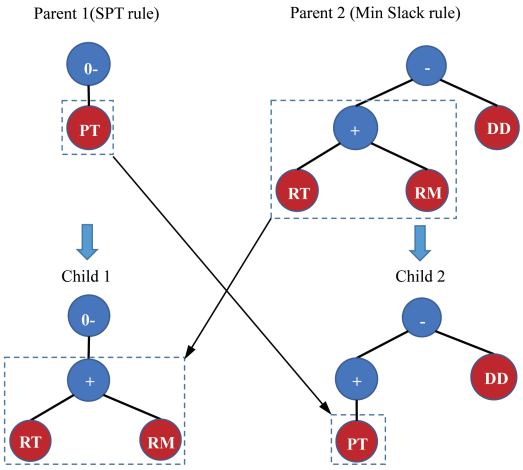

Figure 2. An illustration of the sub-tree crossover operator.

5.2. GDDR-CCGP Algorithm

In order to deal with the multiple decision points of the studied problem, the proposed GDDR-CCGP adopts the cooperative co-evolution strategy, which is an efficient approach for solving problems with co-adapted sub-parts and strives to search for individuals that perform well together. It breaks a high-dimensional optimization space into several much simpler sub-spaces for sub-populations to search [29].

In the GDDR-CCGP algorithm, we divide the heuristic space into three sub-spaces each of which corresponds to a heuristic space of a decision point described previously. Then, three independent GPs are employed for each search on a sub-space. The cooperative co-evolution of the three GPs is realized during individual evaluation.

In the GDDR-CCGP algorithm, we employ three basic GPs to evolve the corresponding three dispatching rules required by the GDDR algorithm. Each GP has its own sub-population and thus, results in three sub-populations; $P_1 = \{p_{1,1}, p_{1,2}, \ldots\}$, $P_2 = \{p_{2,1}, p_{2,2}, \ldots\}$, and $P_3 = \{p_{3,1}, p_{3,2}, \ldots\}$ represent the sub-populations of BFDR, BSDR, and JSDR, respectively. Thus, a schedule policy s consists of three rules from each sub-population.

The pseudo-code of the GDDR-CCGP algorithm is given in Algorithm 1. The GDDR-CCGP starts with the initialization of sub-populations by implementing the ramped-half-and-half tree builder. Then in each generation, every individual (dispatching rule) in the three sub-populations is evaluated in the form of schedule policy to obtain a fitness value (see next sub-section), which in turn governs the search behavior of the GDDR-CCGP. With the guidance of fitness values, the elites of each sub-population are selected to enter the next generation directly. Then, each sub-population is evolved independently by applying genetic operators from the sub-tree crossover and point mutation to generate the new sub-population of the next generation. The GDDR-CCGP moves to the next generation when there are sufficient number (popsize) of individuals for each sub-population. During the evaluation of individuals, each evaluated schedule policy s is recorded in a set S, and the best schedule policy is returned after the GDDR-CCGP is terminated. Figure 3 also graphically illustrates the procedure of the GDDR-CCGP algorithm.

Algorithm 1: GDDR-CCGP

$T \leftarrow$ a set of training instances $T_t, t = 1, 2, \ldots, |T|$;
$S \leftarrow$ a set of evaluated schedule policy;
Initialize P_1, P_2, and P_3 with the ramped-half-and-half tree builder;
$dr_1 \leftarrow p_{1,1}$, $dr_2 \leftarrow p_{2,1}$, $dr_3 \leftarrow p_{3,1}$;
while *maxGeneration is not reached* **do**
    `// evaluate batch formation rules`
    for *i from 1 to popsize* **do**
        $r_2 \leftarrow$ random individual from P_2;
        $r_3 \leftarrow$ random individual from P_3;
        $s \leftarrow (p_{1,i}, dr_2, dr_3)$, $s' \leftarrow (p_{1,i}, r_2, r_3)$;
        $\text{fit}(p_{1,i}) \leftarrow \min\{eval(s, T), eval(s', T)\}$;
        Add s and s' to **S**;
    end
    $dr_1 \leftarrow \arg\min_{p_{1,i} \in P_1} \text{fit}(p_{1,i})$;
    `// evaluate batch sequencing rules`
    for *i from 1 to popsize* **do**
        $r_1 \leftarrow$ random individual from P_1;
        $r_3 \leftarrow$ random individual from P_3;
        $s \leftarrow (dr_1, p_{2,i}, dr_3)$, $s' \leftarrow (r_1, p_{2,i}, r_3)$;
        $\text{fit}(p_{2,i}) \leftarrow \min\{eval(s, T), eval(s', T)\}$;
        Add s and s' to S;
    end
    $dr_2 \leftarrow \arg\min_{p_{2,i} \in P_2} \text{fit}(p_{2,i})$;
    `// evaluate job sequencing rules`
    for *i from 1 to popsize* **do**
        $r_1 \leftarrow$ random individual from P_1;
        $r_2 \leftarrow$ random individual from P_2;
        $s \leftarrow (dr_1, dr_2, p_{3,i})$, $s' \leftarrow (r_1, r_2, p_{3,i})$;
        $\text{fit}(p_{3,i}) \leftarrow \min\{eval(s, T), eval(s', T)\}$;
        Add s and s' to S;
    end
    $dr_3 \leftarrow \arg\min_{p_{3,i} \in P_3} \text{fit}(p_{3,i})$;
    `// breed sub-populations`
    for *r from 1 to 3* **do**
        $P_r \leftarrow$ generated offspring by applying the genetic operators of reproduction, sub-tree crossover, and point mutation;
    end
end
return $s = \arg\min_{s \in S} \text{fit}(s)$;

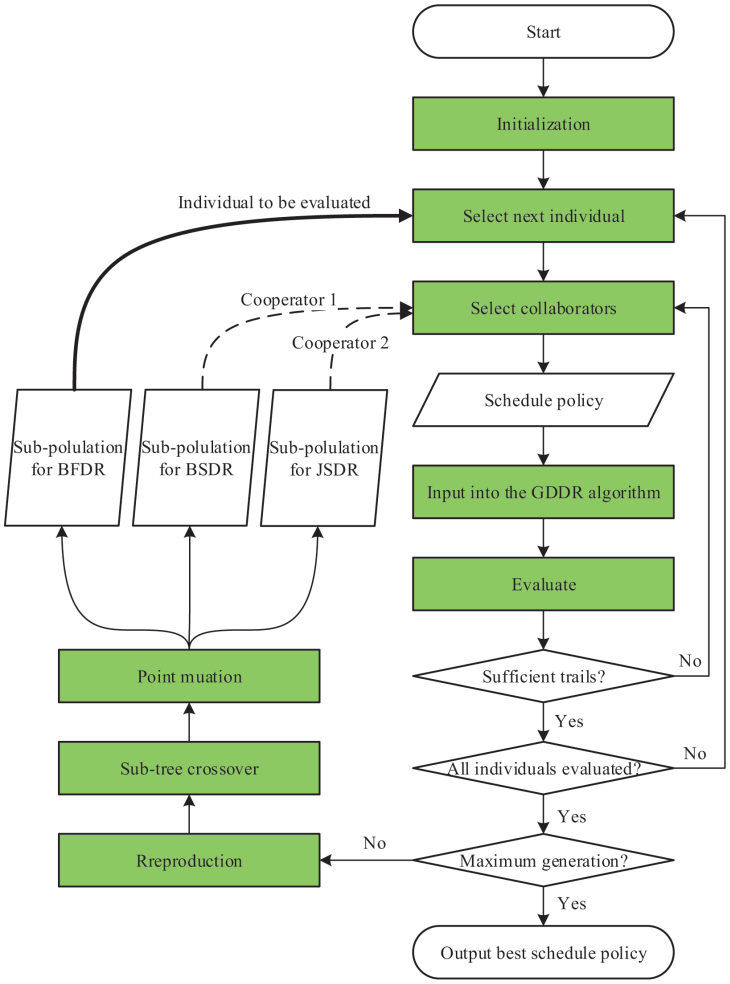

Figure 3. Procedure of the GDDR-CCGP algorithm. An individual from the sub-population for BFDR is being evaluated.

5.3. Fitness Evaluation

Unlike conventional GP, the GDDR-CCGP evaluates individuals in groups, i.e., the fitness of an individual is decided by its performance in the context of individuals from other sub-populations. As a result, an individual is usually evaluated multiple times, each time with a different group of cooperative individuals. We denote each cooperation as a trial and the final fitness is calculated based on these multiple trials.

More specifically, for each trial an individual undertakes, the GDDR-CCGP combines the candidate individual of a certain sub-population with two cooperators selected from the other two sub-populations to form a schedule policy. Figure 3 presents an example of individual evaluation, where an individual of sub-population for BSDR is being evaluated.

Moreover, the selection of cooperators must address the problems of selection pressure, number of trials and trial credit assignment [30]. In this paper, we propose a two-trial evaluation scheme as described in Algorithm 1 (line 5–28). For the first trial, a candidate individual is cooperated with the best individuals of each other sub-population to form a schedule policy s. In the second trial, candidate individual is placed with two random individuals selected from each sub-population to form a schedule policy s'. Then, the fitness of the candidate individual is defined as the best fitness values reached by s and s'. In this way, we prevent the GDDR-CCGP algorithm from being too greedy while sustaining the ability of exploration.

Whenever a schedule policy is ready for evaluation, it is applied to the training instance set T to obtain a fitness. Algorithm 2 describes the procedure of fitness evaluation. For each instance $T_t \in T$, we build a simulation based on the configuration of the instance.

At the beginning of the simulation, all processors are idle and jobs are initialized according to instance T_t. Then dispatching rules of the schedule policy are applied according to the procedure of the GDDR to make online decisions in the simulation. An objective value $O_t(s)$ is obtained when the simulation is finished. After all simulations of the training set T are finished, the fitness of the schedule policy is calculated by

$$fit(s) = \frac{1}{|T|} \sum_{t=1}^{|T|} \frac{O_t(s) - O_t(ref)}{O_t(ref)} \times 100,$$

where $O_t(ref)$ is the objective value reached by applying a benchmark algorithm or calculated by a lower bound. $fit(s)$ is essentially the average gap between evaluated schedule policy and benchmark algorithm. Thus, a lower $fit(s)$ indicates a better performance of the evaluated schedule policy.

Algorithm 2: eval(s, T)

Input: a schedule policy s and the training set T
Output: the objective value reached by applying s

$fit(s) \leftarrow 0$;
for *t from 1 to* $|T|$ **do**
  $O_t(s) \leftarrow$ apply schedule policy s in one replication run of simulation based on instance T_t;
  $O_t(ref) \leftarrow$ apply the reference algorithm in one replication run of simulation based on instance T_t;
  $fit(s) \leftarrow fit(s) + \frac{O_t(S) - O_t(ref)}{O_t(ref)} \times 100$;
end
return $fit(s)/|T|$

6. Computational Experiments

In this section, computational experiments are conducted to evaluate the performance of the proposed GDDR-CCGP algorithm. The problem specified terminal sets and parameter settings are designed, and then the GDDR-CCGP algorithm is compared with the state-of-the-art algorithms of this two-stage hybrid flow shop scheduling problem.

6.1. Terminal Sets and Parameter Settings

The GPs' terminals consist of basic attributes about the current job or batch and the "less-myopic" terminals which take into account information regarding environmental attributes associated with the current job or batch [31]. In order to solve the problem of $\beta \rightarrow \delta|LB, IF|\sum C_i$ efficiently, we customize the terminal sets of GPs based on the utilization of dispatching rules in the GDDR algorithm. Tables 1 and 2 present the problem-specified terminals and the parameter settings of each GP. To provide more environment information for BFDR to assign a job to a correct batch, we involve $MinBWL$ and $MaxBWL$ terminals to provide information about other batches. As for the terminal sets of GPs for BSDR and JSDR, the "less-myopic" terminals are utilized, which provides information on the other processor associated with the state of current processor. These "less-myopic" terminals consist of $PTQB$, $NJQB$, and BBT. By involving these three terminals, the generation of dispatching rules are allowed to adapt to the changing conditions of the upstream or downstream processor.

The function sets include four basic arithmetic operators, namely, $+, -, \times$, protected division $\div$ which returns 1 when divided by 0, and a max operator which takes two

arguments and returns the maximum value of them. Moreover, a ternary if-less-then-else (IfLT) is included and defined as follows:

$$IfLT(x_1, x_2, x_3) = \begin{cases} x_2, & x_1 < 0 \\ x_3, & otherwise \end{cases},$$

where x_1, x_2, x_3 are sub-expressions evolved by GP.

Function sets of the three GPs are of the same, except we extend the function set of GP for JSDR by adding the operator Neg $(0 - x_1)$, which returns the negative value of its argument. Since the scheduling problem of the job sequencing phase is a single machine total completion time minimization problem with a job release time, the SPT rule has been shown to be near optimal in most cases [32]. Thus, the addition of the Neg operation is based on the hope of providing opportunities for GP to evolve a dispatching rule that performs at least as good as the SPT rule $(0 - PT)$ by combining operator Neg with the terminal PT.

Table 1. Terminals of GDDR-CCGP for the problem of $\beta \rightarrow \delta|LB, IF|\sum C_i$.

Notation	Description
PT	The processing time of a job on the discrete processor
BPT	The processing time of a batch on the batch processor
BWL	Work load of a batch, calculated by sum of PTs of all jobs in a batch
MinBWL	When a job is assigned to a batch, the minimum BWL of the remaining batches
MaxBWL	When a job is assigned to a batch, the maximum BWL of the remaining batches
BSR	The number of space units remaining for a batch
BBT	The time of batch blocking in the batch processor
FPTR	Sum of PTs remaining for a job family
PTQB	Sum of PTs queueing in the buffer
NJQB	Number of jobs queueing in the buffer
ERC	A random constant from -1 to 1

Furthermore, as given in Table 2, the evolution parameter settings of three GPs are the same, where a high crossover rate is adopted with the hope of exploring more individuals. These evolution parameter settings are commonly used without paying much effort to parameter tuning, so that the contribution of the GDDR-CCGP algorithm is mainly owed to the well-designed heuristic structure, evaluation scheme, terminal set and function set.

Table 2. Parameter settings of GDDR-CCGP for the problem of $\beta \rightarrow \delta|LB, IF|\sum C_i$.

Parameter	Value
Terminal set of GP for BFDR	PT, BPT, BWL, MinBWL, MaxBWL, FPTR, BSR, 0, 1
Terminal set of GP for BSDR	BPT, BWL, PTQB, NJQB,0, 1
Terminal set of GP for JSDR	PT, PTQB, NJQB, BBT, ERC
Function set	$+$, $-$, $\times$, $\div$, Max, IfLT, (Neg)
Initialization	Ramped-half-and-half tree builder
Population size	500
Generation	50
Crossover rate	95%
Mutation rate	5%
Elitism	20
Max. tree depth	17
Individual selection	Tournament selection (size 7)
Cooperators	1 elite of last generation, 1 random

6.2. Problem Instances Generation

Problem instances were generated randomly by adopting the idea of [3,7], and we extended their instances with additional large-scale instances. As a result, three instance sets were generated, namely, T_s, T_m, and T_l, to test the performance of algorithms in small-scale, medium-scale and large-scale instances. A configuration of the problem is represented by the combination $n - m - b - h$, which describes the total number of jobs and job families, batch capacity and buffer size of a problem configuration. For each problem configuration $n - m - b - h$, we randomly generated 50 instances by applying the following parameters:

- job processing time of job i: $p_i \sim U[1, 10]$;
- batch processing time of job family k: $q_k \sim U\left(\frac{b}{n}\sum_{i=1}^{n} p_i - 10, \frac{b}{n}\sum_{i=1}^{n} p_i + 10\right)$;
- total number of jobs in job family k: $n_k = \lfloor \frac{n}{mb} \rfloor, k = 1, 2, \ldots, m-1$, and $n_m = n - \sum_{k-1}^{m-1} n_k$.

The parameters of each instance set are listed as below:

- T_s: $n = \{40, 60, 80\}$, $m = \{2, 4\}$, $b = \{5, 10\}$, $h = \{2, 3\}$;
- T_m: $n = \{100, 200, 300, 400\}$, $m = \{2, 4, 10\}$, $b = \{10, 20\}$, $h = \{2, 3\}$;
- T_l: $n = \{600, 900, 1200\}$, $m = \{10, 20\}$, $b = \{10, 20\}$, $h = 2$.

T_l is mainly distinguished by its larger number of jobs and job families. Furthermore, the buffer size is restricted to two as a smaller buffer size generally leads to a more complex problem. T_l was designed with the aim of testing the effectiveness and robustness of the evolved schedule policy in unseen and more complex problem configurations. Moreover, an additional filter of $\frac{n}{mb} < 1$ is applied to get rid of simple configurations and results in 82 configurations (24, 46 and 12 configurations for T_s, T_m, and T_l, respectively) consisting of 4100 instances.

6.3. Comparison Results

In order to evaluate the performance of the proposed GDDR-CCGP problem, three algorithms proposed by [3] were selected as benchmarks, namely, a greedy dealing shortest processing time (GDSPT) heuristic, a recycled differential evolution (RDE) algorithm, and the RDE with only one iteration, which is denoted by IDE. GDDR-CCGP was coded in Java on the basis of the ECJ library (https://cs.gmu.edu/~eclab/projects/ecj/ (accessed on 1 October 2021)) and the benchmark algorithms were coded in C++. All algorithms were run on an Inter Xeon CPU 2.4 GHz computer with 16.0 GB of RAM.

The GDSPT heuristic forms batches by using a modified full batch policy (FB), then sorts batches in non-decreasing order of the batch processing time BPT and total processing time $BPT + BWL$, which results in two batch sequencing solutions. After that, jobs are processed on the discrete processor according to the SPT rule. Finally, the solution is returned with the better objective value of the two solutions. Thus, the GDSPT heuristic can be written as FB-GDSPT-SPT.

The RDE algorithm extends the differential evolution with a recycle technique, which allows the solution of a differential evolution iteration to be used in the population of the next iteration for improving the solution quality. The RDE (IDE) was proposed to make batch sequencing decisions and is cooperated with the FB and the SPT for batch formation and job sequencing, respectively. Thus, the RDE (IDE) algorithm is essentially a schedule policy of FB-RDE (IDE)-SPT. In our experiments, the RDE algorithm was terminated after sustaining the same best solution after 40 iterations.

Since the GDDR-CCGP is stochastic, we preliminarily ran the GDDR-CCGP 50 times independently with different seeds, then the best schedule policy, denoted by GDDR, was selected for comparison in the following experiments.

For the sake of simplicity, some new notations are introduced as follows:

Obj: the objective value reached by applying an algorithm on an instance;
H_h: any algorithm h;

LB: the composite lower bound developed by [3];
Dif: the difference in percentage between an algorithm and the lower bound;
$\prec$: $H_1 \prec H_2$ indicates H_2 outperforms H_1.

In this section, the total completion time difference relative to the objective value reached by the LB is reported, which is calculated by

$$Dif(H) = \frac{Obj(H) - Obj(LB)}{Obj(LB)} \times 100.$$

It is obvious that, $Dif(H)$ is greater than zero and $Dif(H_2) < Dif(H_1)$ indicates that $H_1 \prec H_2$.

Tables 3 and 4 report the differences between the corresponding algorithm and LB when tested on instance sets T_s and T_m. The $Time(s)$, $Avg(\%)$, and $Max(\%)$ represent the average run time, average differences, and maximum differences among 50 instances of each problem configuration. All average performance values were tested for statistically significant differences using the paired t-test. The values marked with letters indicate the GDDR significantly outperforms different benchmark algorithms at 5% significant level with regard to corresponding problem configuration, where "a", "b", and "c" represent the GDSPT, IDE, and RDE algorithms, respectively. The run times of GDSPT are not reported because the time costs for all instances are less than 0.01 s.

As observed from Table 3, in terms of Avg values, GDDR dominates GDSPT, IDE, and RDE on 23 (95.83%), 22 (91.67%), and 22 (83.33%) configurations out of all 24 configurations. As for the Max values, GDDR dominates all the other algorithms with 21 (87.50%) configurations. Statistically, GDDR significantly outperforms GDSPT, IDE, and RDE on 21 (87.50%), 18 (75.00%), and 18 (75.00%) configurations, respectively. In terms of average run time, GDDR obtains results within 0.01 s for all configurations.

Table 3. Numerical results of the small-scale instance set T_s, relative to the lower bound. The values marked with letters indicate the GDDR significantly outperforms different benchmark algorithms at 5% significant level, where "a", "b", and "c" represent the GDSPT, IDE, and RDE algorithms, respectively. Run time of GDSPT is omitted because it finishes all instances within a second.

	GDSPT			IDE			RDE			GDDR	
n-m-b-h	Avg (%)	Max (%)	Time (s)	Avg (%)	Max (%)	Time (s)	Avg (%)	Max (%)	Time (s)	Avg (%)	Max (%)
40-2-5-2	6.62	22.94	0.07	6.38	22.94	1.13	6.26	22.76	0.01	4.50 ^{a,b,c}	15.90
40-2-5-3	6.72	30.60	0.07	6.53	30.17	1.06	6.39	30.17	0.00	5.16 ^{a,b,c}	25.31
40-2-10-2	2.08	6.07	0.04	2.08	6.07	0.04	2.08	6.07	0.00	1.59 ^{a,b,c}	5.57
40-2-10-3	1.96	7.45	0.04	1.96	7.45	0.03	1.96	7.45	0.00	1.60 ^{a,b,c}	6.65
40-4-5-2	5.45	13.50	0.06	5.03	13.50	0.95	4.76	13.50	0.00	4.93 ^{a}	16.31
40-4-5-3	4.13	17.59	0.06	3.79	15.75	0.96	3.59	15.44	0.00	4.06 ^{b,c}	13.88
40-4-10-2	1.34	5.00	0.01	1.34	5.00	0.01	1.34	5.00	0.00	1.24	5.09
40-4-10-3	1.36	6.42	0.01	1.36	6.42	0.01	1.36	6.42	0.00	1.41	8.49
60-2-5-2	7.93	31.27	0.11	7.60	29.18	2.50	7.49	28.62	0.00	5.12 ^{a,b,c}	21.22
60-2-5-3	5.17	29.18	0.12	4.97	28.29	2.45	4.88	27.66	0.00	3.94 ^{a,b,c}	22.32
60-2-10-2	2.11	8.94	0.09	2.05	8.19	1.11	2.02	7.85	0.00	1.67 ^{a,b,c}	4.90
60-2-10-3	2.32	7.68	0.10	2.14	7.32	1.30	2.11	7.32	0.00	1.91 ^{a}	5.29
60-4-5-2	5.70	18.92	0.11	5.32	18.21	2.93	5.10	16.81	0.00	4.12 ^{a,b,c}	11.57
60-4-5-3	6.51	23.06	0.10	6.30	22.77	2.43	6.10	22.64	0.00	5.22 ^{a,b,c}	18.83
60-4-10-2	2.40	9.21	0.07	2.11	7.32	1.09	1.99	6.64	0.00	1.96 ^{a}	5.22
60-4-10-3	2.11	6.53	0.08	1.79	6.27	1.10	1.67	6.19	0.00	1.69 ^{a}	4.70
80-2-5-2	7.19	28.80	0.19	6.93	28.51	5.26	6.77	27.30	0.00	4.35 ^{a,b,c}	19.18
80-2-5-3	6.33	21.19	0.22	6.23	21.19	4.81	6.13	20.88	0.00	4.46 ^{a,b,c}	18.04
80-2-10-2	2.52	9.49	0.11	2.42	8.83	2.19	2.39	8.58	0.00	1.69 ^{a,b,c}	4.52
80-2-10-3	2.07	9.06	0.12	1.94	8.89	2.18	1.91	8.89	0.00	1.50 ^{a,b,c}	4.26
80-4-5-2	6.97	22.85	0.18	6.46	21.58	5.45	6.19	20.70	0.00	4.43 ^{a,b,c}	12.62
80-4-5-3	6.60	17.39	0.18	6.40	17.34	5.15	6.11	17.20	0.00	4.31 ^{a,b,c}	11.02
80-4-10-2	2.12	6.99	0.13	1.84	6.58	2.28	1.70	6.43	0.00	1.66 ^{a}	5.84
80-4-10-3	2.48	8.53	0.13	2.09	7.45	2.21	1.92	7.45	0.00	1.72 ^{a,b,c}	5.34

Table 4. Numerical results of the medium-scale instance set T_m, relative to the lower bound. The values marked with letters indicate the GDDR significantly outperforms different benchmark algorithms at 5% significant level, where "a", "b", and "c" represent the GDSPT, IDE, and RDE algorithms, respectively. Run time of GDSPT is omitted because it finishes all instances within a second.

	GDSPT			IDE			RDE			GDDR	
n-m-b-h	**Avg (%)**	**Max (%)**	**Time (s)**	**Avg (%)**	**Max (%)**	**Time (s)**	**Avg (%)**	**Max (%)**	**Time (s)**	**Avg (%)**	**Max (%)**
100-2-10-2	2.28	9.65	0.39	2.21	9.45	4.14	2.16	9.27	0.01	1.63 [a,b,c]	4.58
100-2-10-3	2.42	8.37	0.16	2.32	8.25	3.91	2.28	8.19	0.01	1.64 [a,b,c]	5.04
100-2-20-2	1.07	4.13	0.12	0.97	3.37	1.93	0.95	3.25	0.00	0.75 [a,b,c]	2.81
100-2-20-3	1.02	3.19	0.12	0.93	3.02	1.95	0.91	3.02	0.00	0.71 [a,b,c]	2.31
100-4-10-2	2.28	8.37	0.16	2.03	7.88	4.37	1.89	7.70	0.00	1.68 [a,b,c]	5.67
100-4-10-3	2.91	8.32	0.16	2.63	7.17	4.29	2.46	6.91	0.00	1.97 [a,b,c]	4.58
100-4-20-2	0.93	2.81	0.09	0.73	2.37	1.81	0.69	2.27	0.00	0.74 [a]	2.65
100-4-20-3	0.82	3.75	0.08	0.65	2.09	1.79	0.58	1.83	0.00	0.65 [a]	2.87
100-10-10-2	3.29	8.24	0.11	2.53	5.59	4.32	1.99	4.70	0.00	2.32 [a,b]	5.50
100-10-10-3	2.89	8.64	0.11	2.18	6.11	4.37	1.72	5.30	0.00	2.04 [a,b]	5.34
200-2-10-2	3.27	12.24	0.71	3.22	12.18	27.22	3.16	11.77	0.01	1.67 [a,b,c]	6.87
200-2-10-3	3.05	8.62	0.69	3.03	8.59	26.08	2.98	8.53	0.01	1.67 [a,b,c]	4.80
200-2-20-2	1.40	3.79	0.40	1.28	3.69	12.20	1.25	3.64	0.01	0.84 [a,b,c]	2.14
200-2-20-3	1.54	3.93	0.41	1.40	3.93	12.94	1.37	3.93	0.01	0.94 [a,b,c]	2.30
200-4-10-2	2.71	12.44	0.69	2.61	12.25	27.42	2.47	11.91	0.00	1.65 [a,b,c]	6.70
200-4-10-3	3.44	9.54	0.66	3.33	9.52	28.24	3.18	9.21	0.01	1.75 [a,b,c]	4.36
200-4-20-2	1.41	4.06	0.41	1.20	3.38	12.19	1.07	2.98	0.01	1.00 [a,b]	2.63
200-4-20-3	1.46	4.06	0.41	1.19	3.08	12.26	1.08	2.94	0.00	1.02 [a,b]	2.44
200-10-10-2	2.90	6.51	0.67	2.65	5.84	28.64	2.13	5.74	0.00	1.82 [a,b,c]	4.34
200-10-10-3	2.93	6.98	0.66	2.69	6.90	28.88	2.14	6.55	0.01	1.85 [a,b,c]	5.74
200-10-20-2	1.65	4.55	0.28	1.12	2.42	13.50	0.82	2.03	0.00	1.02 [a,b]	2.06
200-10-20-3	1.51	3.49	0.29	1.15	3.11	12.69	0.85	2.19	0.00	1.04 [a,b]	2.43
300-2-10-2	3.19	13.99	1.82	3.13	13.99	83.25	3.07	13.78	0.03	1.76 [a,b,c]	7.65
300-2-10-3	3.37	10.31	1.81	3.36	10.31	83.09	3.31	10.21	0.02	1.92 [a,b,c]	7.25
300-2-20-2	1.29	4.92	0.99	1.24	4.84	37.36	1.22	4.65	0.02	0.71 [a,b,c]	2.04
300-2-20-3	1.24	4.14	1.03	1.19	4.14	36.02	1.16	4.12	0.01	0.71 [a,b,c]	2.06
300-4-10-2	4.42	12.21	1.83	4.25	11.96	103.03	4.02	11.30	0.01	1.97 [a,b,c]	5.43
300-4-10-3	3.29	7.54	1.81	3.26	7.54	99.55	3.09	7.47	0.02	1.65 [a,b,c]	3.85
300-4-20-2	1.47	4.77	1.03	1.33	4.22	41.67	1.24	4.14	0.01	0.82 [a,b,c]	2.03
300-4-20-3	1.49	4.37	0.99	1.30	3.80	40.71	1.19	3.62	0.01	0.81 [a,b,c]	1.97
300-10-10-2	3.10	8.15	1.80	2.97	6.83	105.32	2.43	6.40	0.01	1.61 [a,b,c]	4.00
300-10-10-3	3.20	9.79	1.83	3.12	8.73	97.75	2.71	7.58	0.01	1.73 [a,b,c]	4.77
300-10-20-2	1.62	5.35	0.88	1.21	3.31	39.39	0.93	2.96	0.01	0.99 [a,b]	2.60
300-10-20-3	1.64	4.70	0.89	1.27	3.64	40.20	0.97	3.26	0.01	0.95 [a,b]	2.59
400-2-10-2	4.13	15.82	3.83	3.92	14.82	217.06	3.77	14.45	0.03	2.26 [a,b,c]	9.80
400-2-10-3	4.28	11.40	4.07	4.26	11.40	161.52	4.24	11.40	0.03	2.54 [a,b,c]	8.06
400-2-20-2	1.58	4.51	2.29	1.56	4.51	83.83	1.53	4.49	0.02	0.76 [a,b,c]	1.77
400-2-20-3	1.49	4.76	2.03	1.45	4.76	84.41	1.41	4.68	0.02	0.70 [a,b,c]	1.92
400-4-10-2	3.49	11.62	3.85	3.37	11.50	198.54	3.17	10.16	0.02	1.46 [a,b,c]	4.95
400-4-10-3	3.20	9.44	3.82	3.19	9.44	189.06	3.11	9.28	0.02	1.58 [a,b,c]	5.25
400-4-20-2	1.46	3.61	2.02	1.38	3.54	92.61	1.28	3.52	0.01	0.81 [a,b,c]	1.71
400-4-20-3	1.55	4.19	1.99	1.42	3.57	99.64	1.31	3.53	0.02	0.83 [a,b,c]	1.82
400-10-10-2	3.16	10.53	3.80	3.03	9.51	214.82	2.74	8.27	0.01	1.49 [a,b,c]	4.02
400-10-10-3	2.97	7.74	3.78	2.92	7.60	216.61	2.57	7.60	0.01	1.42 [a,b,c]	4.46
400-10-20-2	1.40	4.30	2.04	1.15	2.67	98.30	0.77	2.16	0.01	0.82 [a,b]	2.48
400-10-20-3	1.66	4.26	2.02	1.39	3.38	106.48	0.98	2.83	0.01	1.02 [a,b]	2.70

Table 4 shows the comparison results on the medium-scale instance set T_m. Similarly, GDDR dominates GDSPT on all configurations in terms of both *Avg* values and *Max* values. When compared with IDE and RDE with regard to the *Avg* values, GDDR wins on 44 (95.65%) and 37 (80.43%) configurations out of all 46 configurations. As for the *Max* values, GDDR wins on 44 (95.65%) and 39 (84.78%) configurations. Statistically, GDDR

significantly outperforms GDSPT, IDE, and RDE on 46 (100.00%), 44 (95.65%) and 34 (73.91%) configurations, respectively. Furthermore, the maximum values of *Avg* and *Max* for GDDR are 2.87% and 6.60%, respectively. Indicating the near optimal performance of the evolved schedule policy. In terms of overall performance, it can be calculated from Table 4 that, GDDR provides 40.55%, 33.24% and 24.55% average improvement to GDSPT, IDE, and RDE, respectively. It is noticeable that, GDDR still costs almost no time, whereas the average time cost of the RDE increases sharply as the complexity of the instance increases. Specifically, the maximum average time cost of GDDR is about 0.03 s, in sharp contrast to 217.06 s for RDE.

Combing the results of Tables 3 and 4, it can be concluded that GDDR can provide promising solutions that significantly outperform GDSPT, IDE, and RDE in most tested configurations. Figure 4 graphically shows the *Avg* values of all algorithms based on the numerical results in Tables 3 and 4. In Figure 4, the average difference of the GDDR is always the smallest and gaps between the GDDR and other algorithms increase with the increasing of number of jobs. This result indicates that GDDR outperforms benchmark algorithms especially in complex problem configurations.

Figure 5 presents the box plots of all *Avg* values and *Max* values reached by the GDDR and three benchmark algorithms in Tables 3 and 4. It can be observed that the GDDR results in a smaller interquartile range than the benchmark rules in terms of both *Avg* value and *Max* value, which indicates that the performance of the GDDR is stabler. Furthermore, the slot of GDDR has a lower maximum, minimum and three quartiles values. Figure 5 shows the same result as Figure 4 and indicates the performance of the GDDR is superior to the three benchmark algorithms.

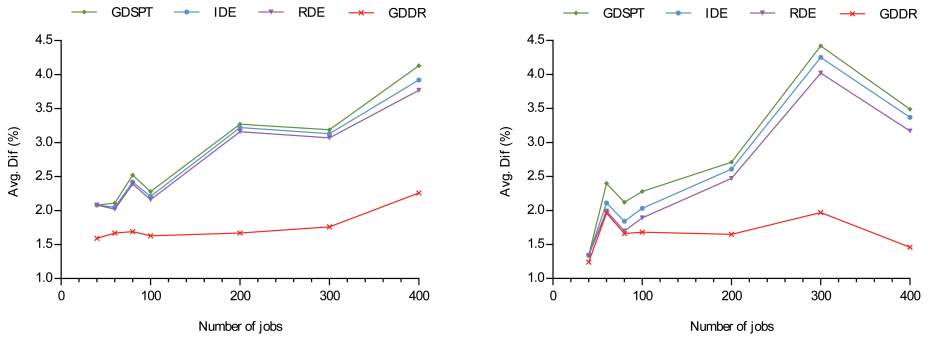

Figure 4. Average difference analysis for different numbers of job family of numerical results from T_s and T_m. (**a**) $m = 2$, $b = 10$, $h = 2$. (**b**) $m = 4$, $b = 10$, $h = 2$.

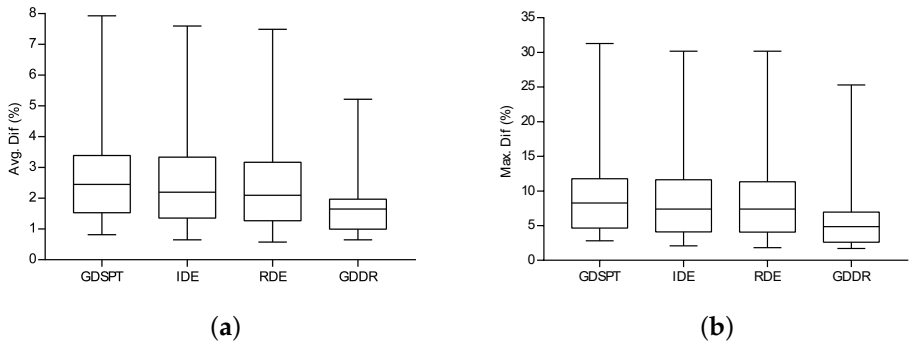

Figure 5. Box-plots of average and maximum differences for each method based on numerical results from T_s and T_m. (**a**) Box-plot of average differences. (**b**) Box-plot of maximum differences.

To verify the observed result is statistically significant, we conducted a paired *t*-test with a significance level of 5% between GDDR and each of the compared algorithms, based

on the results in Tables 3 and 4. The hypothesis and the calculated p-values are listed as follows:

- T1 ($H_0 : Avg_{(GDDR-GDSPT)} = 0$ and $H_1 : Avg_{(GDDR-GDSPT)} < 0$), $p = 2.72 \times 10^{-19}$;
- T2 ($H_0 : Avg_{(GDDR-IDE)} = 0$ and $H_1 : Avg_{(GDDR-IDE)} < 0$), $p = 6.61 \times 10^{-15}$;
- T3 ($H_0 : Avg_{(GDDR-RDE)} = 0$ and $H_1 : Avg_{(GDDR-RDE)} < 0$), $p = 2.08 \times 10^{-11}$.

Obviously, the null hypothesis H_0 is rejected for all hypothesis, which indicates that GDDR significantly outperforms GDSPT, IDE, and RDE. Consequently, based on the dominant ratios and p-values, we can further conclude that GDSPT $\prec$ IDE $\prec$ RDE $\prec$ GDDR.

In order to evaluate the robustness ability of GDDR, we further test GDDR on large-scale instance set T_l, which contains instances that the GDDR never saw during the training phase. Table 5 presents the computational results on large-scale instance set T_l. The run time of RDE is limited to around 600 s, whereas other parameter settings remain unchanged.

Table 5. Numerical results on the large-scale instance set T_l, relative to the lower bound. The values marked with letters indicate the GDDR significantly outperforms different benchmark algorithms at 5% significant level, where "a", "b", and "c" represent the GDSPT, IDE, and RDE algorithms, respectively. Run time of GDSPT is omitted because it finishes all instances within a second.

	GDSPT			IDE			RDE			GDDR	
n-m-b-h	**Avg (%)**	**Max (%)**	**Time (s)**	**Avg (%)**	**Max (%)**	**Time (s)**	**Avg (%)**	**Max**	**Time (s)**	**Avg (%)**	**Max (%)**
600-10-10-2	4.55	7.92	4.14	4.22	7.04	478.53	3.72	5.87	0.02	1.42 ^{a,b,c}	2.40
600-10-20-2	1.44	2.19	1.93	1.37	2.19	321.06	0.82	1.88	0.01	0.69 ^{a,b,c}	1.23
600-20-10-2	3.58	7.15	3.79	3.56	6.95	406.82	3.25	5.30	0.01	1.72 ^{a,b,c}	3.34
600-20-20-2	1.34	2.90	1.88	1.22	2.57	341.80	0.61	1.72	0.01	0.61 ^{a,b,c}	1.33
900-10-10-2	3.95	10.37	11.84	3.81	9.62	613.62	3.62	9.12	0.04	1.27 ^{a,b,c}	3.13
900-10-20-2	1.76	2.85	5.93	1.70	2.85	584.91	1.51	2.85	0.02	0.71 ^{a,b,c}	1.17
900-20-10-2	4.58	8.97	11.67	4.29	7.41	613.89	4.03	6.83	0.03	1.48 ^{a,b,c}	2.54
900-20-20-2	1.65	3.49	5.88	1.55	2.96	584.99	1.24	2.19	0.02	0.86 ^{a,b,c}	1.70
1200-10-10-2	6.14	14.10	26.79	5.43	13.16	638.08	5.08	12.33	0.06	1.90 ^{a,b,c}	6.12
1200-10-20-2	1.79	3.40	13.51	1.76	3.40	621.51	1.67	3.40	0.03	0.69 ^{a,b,c}	1.18
1200-20-10-2	5.12	10.45	26.81	4.48	8.08	643.89	4.16	7.78	0.04	1.21 ^{a,b,c}	2.49
1200-20-20-2	1.58	2.79	12.93	1.51	2.08	616.20	1.44	1.95	0.02	0.80 ^{a,b,c}	1.15

As we can observe from Table 5, GDDR significantly outperforms all benchmark algorithms in all configurations with regard to *Avg* value and *Max* value. As for time cost, the run time of RDE starts exceeding 600 s when n increases to 1200 and GDDR keeps finishing calculations within a second. These results demonstrate the reusable ability of GDDR, it sustains effective and efficient performances when being applied to unseen complex problem instances.

7. Discussion

To investigate more insights of the studied problem and discuss the effectiveness of the proposed GDDR-CCGP algorithm, we conducted additional experiments to compare the GDDR-CCGP with GP's counterpart. The dispatching rules of some decision points are unchanged, and they only evolve the dispatching rules of the remaining decision points. Based on the GDSPT (FB-GDSPT-SPT) algorithm, we substitute the dispatching rule of each decision point with GP and do not change the other dispatching rules. As a result, the counterpart GPs are denoted as BFGP (GP-GDSPT-SPT), BSGP (FB-GP-SPT) and BatchCCGP (GP-GP-SPT, a cooperative co-evolutionary genetic programming with two co-evolving sub-populations). For fair comparisons, the terminal sets and parameter settings are identical to corresponding sub-parts of the GDDR-CCGP as listed in Tables 1 and 2. The population sizes of BFGP, BSGP and BatchCCGP are set to 3000, 3000 and 750, respectively. Therefore, all GPs get the same number of fitness evaluations per generation as GDDR-CCGP.

In this experiment, we ran all GPs 50 times and common random seeds were utilized across the GPs to provide the same set of evaluation environments, and result in 50 schedule policies for each GP. Then, each schedule policy was tested on the instance set T_m to obtain the total objective value of all test instances as the performance criterion of the schedule policy. After that, the average performance of the 50 schedule policies of a GP were compared with others. Table 6 reports the absolute mean performance differences of the GPs based on 50 runs of each GP, where a positive value indicates the GP in the column header is better than the GP in the first column of the current row. Furthermore, in order to provide indicators of statistical significance, we again employed the paired *t*-test with a significance level of 5% and "+" indicates significant, "o" otherwise.

Table 6. Absolute mean performance differences of the GPs based on 50 runs.

	BSGP	**BatchCCGP**	**GDDR-CCGP**
BFGP	−686,434.96 (+)	453,248.68 (+)	414,854.98 (+)
BSGP		1,139,683.64 (+)	1,101,289.94 (+)
BatchCCGP			−38,393.70 (o)

As observed from Table 6, BFGP significantly outperforms BSGP. In addition, in terms of absolute mean performance difference values, BatchCCGP improves BSGP by the value of 1,139,683.64, which is much larger than the improvement in BFGP. These results imply that the GP for the batch formation phase can better improve the optimization of the problem than the GP for the batch sequencing phase. Meanwhile, the GPs with cooperative co-evolution (BatchCCGP and GDDR-CCGP) are significantly better than the GPs which only evolve dispatching rules on a single decision point (BFGP and BSGP). This result demonstrates the necessity of a cooperative co-evolution strategy for the studied problem. As for the comparison between BatchCCGP and GDDR-CCGP, although the absolute mean performance difference shows the superiority of the former, the statistical result reports a non-significant indicator. In summary, we can at least conclude that BSGP $\prec$ BFGP $\prec$ BatchCCGP $\approx$ GDDR-CCGP.

In order to compare the optimization ability of BatchCCGP and GDDR-CCGP, a comparison was conducted based on results of the 50 evolved schedule policies of each GP. As shown in Table 7, each GP was compared with the three benchmark algorithms and the overall improvement ratio (a negative ratio indicates the improvement provided by GPs) and dominant ratio (number of dominant configurations divided by total number of configurations) was obstained against each benchmark algorithm. The average and best values are reported.

Table 7. Overall improvement ratio and dominant ratio of GPs compared with benchmark algorithms.

	Improvement Ratio (%)						**Dominant Ratio (%)**					
	GDSPT		**IDE**		**RDE**		**GDSPT**		**IDE**		**RDE**	
	Avg	**Best**	**Avg**	**Best**	**Avg**	**Best**	**Avg**	**Best**	**Avg**	**Best**	**Avg**	**Best**
BSGP	41.45	−31.17	60.99	−23.19	79.14	−13.04	50.13	93.48	39.48	82.61	30.26	65.22
BFGP	−16.43	−22.48	−4.81	−11.39	10.34	3.10	87.22	100.00	56.26	65.22	50.35	58.70
BatchCCGP	−29.16	−37.90	−19.43	−29.66	−8.39	−19.68	92.17	100.00	80.30	93.48	63.26	76.09
GDDR-CCGP	−27.68	−40.25	−17.89	−32.20	−6.81	−22.74	90.65	100.00	80.09	93.48	61.91	78.26

As observed from Table 7, BatchCCGP outperforms the GDDR-CCGP in all average values, this result is consistent with the statistical results of Table 6. However, in terms of best values, GDDR-CCGP shows a better performance than BatchCCGP. Overall, GDDR-CCGP provided additional 2–4% improvements in comparison to BatchCCGP and dominated RDE on a larger number of configurations. As we described earlier, the SPT rule

provides near optimal solutions for the job sequencing phase, the results in Tables 6 and 7 show that the addition of GP for job sequencing phase does not promise a better result, but does provide opportunity for a GP to evolve better dispatching rules. Thus, we conclude that the GDDR-CCGP is capable of generating better dispatching rules than BatchCCGP without guaranteeing superior performance on every run.

Moreover, when comparing the results between BFGP and BSGP, the former significantly outperforms the latter in all average values. Although the best values of BSGP are better, in our experiments, only two runs of BSGP yield better results than BFGP among 50 runs, whereas the other 48 runs are significantly worse than BFGP. This indicates that, the GP for the batch formation phase can contribute much more than that for the batch sequencing phase.

Finally, as for the comparison between BSGP and RDE, since they are both optimized on the batch sequencing phase, this is a comparison between the GP-generated heuristic and well-designed meta-heuristic. As well-known, it is difficult for a heuristic to beat a meta-heuristic. However, it is noticeable that, the best schedule policy evolved by BSGP outperforms RDE with a 13.04% improvement ratio and a 65.22% dominant ratio. This remarkable result indicates that some dispatching rules can compete with a meta-heuristic, and again demonstrates the effectiveness of the proposed GDDR-CCGP algorithm.

8. Conclusions

In this study, we address the two-stage hybrid flow shop scheduling problem with a batch processor followed by a discrete processor. An incompatible job family and size-limited buffer are also considered. In order to provide an online real-time decision-making solution, we propose the GDDR heuristic with three dispatching rules for the three corresponding decision points of batch formation, batch sequencing and job sequencing. In addition, a genetic programming with cooperative co-evolution (GDDR-CCGP) is proposed as the hyper-heuristic for evolving schedule policies.

The experimental results show that the evolved best schedule policy GDDR outperforms the benchmark heuristic and meta-heuristics in terms of solution quality and run time, and that the GDDR can provide high quality solutions especially for large-scale instances within few seconds. These results demonstrate that the proposed GDDR-CCGP is capable of generating high-quality schedule policies for making real-time scheduling decisions for the studied problem.

We further analyze the optimization contribution of each decision point. The numerical results show that, optimization of the batch formation phase contributes the most, followed by the batch sequencing phase and then the job sequencing phase. Since the SPT rule is near optimal for the job sequencing phase, evolving dispatching rules for the job sequencing phase provides opportunity for better schedule polices. Thus, the hyper-heuristic for the job sequencing phase is optional but strongly recommended.

The proposed GDDR-CCGP can be easily extended to other flow shop set-ups where cooperation between the batch processor and discrete processor exists. More importantly, the evolved schedule policy offers a powerful schedule tool in a real dynamic manufacturing environment, which provides competitive solution quality in a short time and allows for quick reactions to the dynamic changes. Even when the production environment is changed, a new run of the GDDR-CCGP can be implemented offline by simply updating training data and obtaining a more suitable scheduling policy without interrupting the production.

Future works will focus on extending the proposed GDDR-CCGP algorithm to multi-stage hybrid flow shop scheduling problems and dealing with multi-objective functions. Additionally, research on the scheduling problem of a more general complex dynamic job shop where batch processors and discrete processors exist, is also a meaningful direction.

Author Contributions: Conceptualization, L.S.; methodology, L.L. and L.S.; computation experiments, L.L.; writing, L.L. All authors have read and agreed to the published version of the manuscript.

Funding: This research was supported in part by National Science Foundation of China [grant number 71690232].

Institutional Review Board Statement: Not applicable.

Informed Consent Statement: Not applicable.

Acknowledgments: The authors would like to thank Cheng Zhang for his support and contributions during the development of this work.

Conflicts of Interest: The authors declare no conflict of interest.

Notations

n	total number of jobs;
m	total number of incompatible job families;
b	batch capacity;
h	the maximum number of batches the buffer can hold;
F	incompatible job families set, where $F = \{1, 2, \cdots, m\}$;
n_k	total number of jobs in family k, where $k = 1, 2, \cdots, m$;
F_k	set of jobs belonging to family k, $F_k = \{1, 2, \cdots, n_k\}$, $k \in F$;
a_k	number of batches in family k, i.e., $a_k = \lceil \frac{n_k}{b} \rceil$, where $k = 1, 2, \cdots, m$;
q_k	processing time on the batch processor for family k, where $k = 1, 2, \cdots, m$;
p_i	processing time on the discrete processor for job i, where $i = 1, 2, \cdots, n$;
N	job set, where $N = \{1, 2, \cdots, n\}$;
B	the batch set, where $B = \{1, 2, \cdots, a\}$ and $a = n/b$;
ε	a sufficient small positive number;
M_i	a sufficient large positive number, where $i = 1, 2, 3$.

Variables

x_{il}	$x_{il} = 1$ if job i is assigned to batch l, and 0 otherwise;
y_{kl}	$y_{kl} = 1$ if batch l consists of the jobs in job family k, and otherwise, $y_{kl} = 0$;
z_{li}	$z_{li} = 1$ if the position of job i in the process sequence of the discrete processor is less than or equal to $(l-h)b$, and otherwise, $z_{li} = 0$, for all $l > h$, let $\beta = (l-h)b$;
u_{ij}	$u_{ij} = 1$ if job i is processed before job j on the discrete processor, and otherwise, $u_{ij} = 0$;
t_l	the processing time of batch l;
r_l	the release time of batch l on the batch processor;
C_i	the completion time of job i.

References

1. Mathirajan, M.; Sivakumar, A.I. A literature review, classification and simple meta-analysis on scheduling of batch processors in semiconductor. *Int. J. Adv. Manuf. Technol.* **2006**, *29*, 990–1001. [CrossRef]
2. Graham, R.L.; Lawler, E.L.; Lenstra, J.K.; Kan, A.R. Optimization and approximation in deterministic sequencing and scheduling: A survey. In *Annals of Discrete Mathematics*; Elsevier: Amsterdam, The Netherlands, 1979; Volume 5, pp. 287–326.
3. Zhang, C.; Shi, Z.; Huang, Z.; Wu, Y.; Shi, L. Flow shop scheduling with a batch processor and limited buffer. *Int. J. Prod. Res.* **2017**, *55*, 3217–3233. [CrossRef]
4. Ahmadi, J.H.; Ahmadi, R.H.; Dasu, S.; Tang, C.S. Batching and scheduling jobs on batch and discrete processors. *Oper. Res.* **1992**, *40*, 750–763. [CrossRef]
5. Branke, J.; Nguyen, S.; Pickardt, C.W.; Zhang, M. Automated design of production scheduling heuristics: A review. *IEEE Trans. Evol. Comput.* **2015**, *20*, 110–124. [CrossRef]
6. Su, L.H.; Yang, D.L.; Chou, H.K. A two-stage flowshop scheduling with limited buffer storage. *Asia-Pac. J. Oper. Res.* **2009**, *26*, 503–522. [CrossRef]
7. Fu, Q.; Sivakumar, A.I.; Li, K. Optimisation of flow-shop scheduling with batch processor and limited buffer. *Int. J. Prod. Res.* **2012**, *50*, 2267–2285. [CrossRef]
8. Shi, Z.; Huang, Z.; Shi, L. Two-stage flow shop with a batch processor and limited buffer. In Proceedings of the 2016 IEEE International Conference on Automation Science and Engineering (CASE), Fort Worth, TX, USA, 21–25 August 2016; pp. 395–400.

9. Mauluddin, Y. Three-stage flow-shop scheduling model with batch processing machine and discrete processing machine. In *MATEC Web of Conferences*; EDP Sciences: Les Ulis, France, 2018; Volume 197, p. 14002.
10. Ross, H.L.F.P.; Corne, D. A promising hybrid GA/heuristic approach for open-shop scheduling problems. In Proceedings of the 11th European Conference on Artificial Intelligence, Amsterdam, The Netherlands, 8–12 August 1994; pp. 590–594.
11. Rodríguez, J.V.; Petrovic, S.; Salhi, A. A combined meta-heuristic with hyper-heuristic approach to the scheduling of the hybrid flow shop with sequence dependent setup times and uniform machines. In Proceedings of the 3rd Multidisciplinary International Conference on Scheduling: Theory and Applications, MISTA, Paris, France, 28–31 August 2007; pp. 506–513.
12. Burke, E.K.; McCollum, B.; Meisels, A.; Petrovic, S.; Qu, R. A graph-based hyper-heuristic for educational timetabling problems. *Eur. J. Oper. Res.* **2007**, *176*, 177–192. [CrossRef]
13. Sim, K.; Hart, E.; Paechter, B. A hyper-heuristic classifier for one dimensional bin packing problems: Improving classification accuracy by attribute evolution. In Proceedings of the International Conference on Parallel Problem Solving from Nature, Taormina, Italy, 1–5 September 2012; Springer: Berlin/Heidelberg, Germany, 2012; pp. 348–357.
14. Runka, A. Evolving an edge selection formula for ant colony optimization. In Proceedings of the 11th Annual Conference on Genetic and Evolutionary Computation, Montreal, QC, USA, 8–12 July 2009; pp. 1075–1082.
15. Bader-El-Den, M.; Poli, R.; Fatima, S. Evolving timetabling heuristics using a grammar-based genetic programming hyper-heuristic framework. *Memetic Comput.* **2009**, *1*, 205–219. [CrossRef]
16. Koza, J.R. *Genetic Programming II: Automatic Discovery of Reusable Subprograms*; MIT Press: Cambridge, MA, USA, 1994; Volume 13, p. 32.
17. Miyashita, K. Job-shop scheduling with genetic programming. In Proceedings of the 2nd Annual Conference on Genetic and Evolutionary Computation, Las Vegas, NV, USA, 10–12 July 2000; pp. 505–512.
18. Dimopoulos, C.; Zalzala, A.M. Investigating the use of genetic programming for a classic one-machine scheduling problem. *Adv. Eng. Softw.* **2001**, *32*, 489–498. [CrossRef]
19. Geiger, C.D.; Uzsoy, R.; Aytuğ, H. Rapid modeling and discovery of priority dispatching rules: An autonomous learning approach. *J. Sched.* **2006**, *9*, 7–34. [CrossRef]
20. Geiger, C.D.; Uzsoy, R. Learning effective dispatching rules for batch processor scheduling. *Int. J. Prod. Res.* **2008**, *46*, 1431–1454. [CrossRef]
21. Hildebrandt, T.; Goswami, D.; Freitag, M. Large-scale simulation-based optimization of semiconductor dispatching rules. In Proceedings of the Winter Simulation Conference 2014, Savannah, GA, USA, 7–10 December 2014; pp. 2580–2590.
22. Shi, Z.; Gao, S.; Du, J.; Ma, H.; Shi, L. Automatic design of dispatching rules for real-time optimization of complex production systems. In Proceedings of the 2019 IEEE/SICE International Symposium on System Integration (SII), Paris, France, 14–16 January 2019; pp. 55–60.
23. Nguyen, S.; Zhang, M.; Johnston, M.; Tan, K.C. Genetic programming for job shop scheduling. In *Evolutionary and Swarm Intelligence Algorithms*; Springer: Berlin/Heidelberg, Germany, 2019; pp. 143–167.
24. Park, J.; Nguyen, S.; Zhang, M.; Johnston, M. Genetic programming for order acceptance and scheduling. In Proceedings of the 2013 IEEE Congress on Evolutionary Computation, Cancún, México, 20–23 June 2013; pp. 1005–1012.
25. Nguyen, S.; Zhang, M.; Johnston, M.; Tan, K.C. Automatic design of scheduling policies for dynamic multi-objective job shop scheduling via cooperative coevolution genetic programming. *IEEE Trans. Evol. Comput.* **2013**, *18*, 193–208. [CrossRef]
26. Yska, D.; Mei, Y.; Zhang, M. Genetic programming hyper-heuristic with cooperative coevolution for dynamic flexible job shop scheduling. In Proceedings of the European Conference on Genetic Programming, Parma, Italy, 4–6 April 2018; Springer: Berlin/Heidelberg, Germany, 2018; pp. 306–321.
27. Uzsoy, R. Scheduling batch processing machines with incompatible job families. *Int. J. Prod. Res.* **1995**, *33*, 2685–2708. [CrossRef]
28. Burke, E.K.; Hyde, M.R.; Kendall, G.; Ochoa, G.; Ozcan, E.; Woodward, J.R. Exploring hyper-heuristic methodologies with genetic programming. In *Computational Intelligence*; Springer: Berlin/Heidelberg, Germany, 2009; pp. 177–201.
29. Luke, S. Essentials of Metaheuristics. Lulu. 2011. Available online: http://cs.gmu.edu/sean/book/metaheuristics/ (accessed on 1 October 2021).
30. Wieg, R.P.; Liles, W.C.; De Jong, K.A. An empirical analysis of collaboration methods in cooperative coevolutionary algorithms. In Proceedings of the Genetic and Evolutionary Computation Conference (GECCO), San Francisco, CA, USA, 7–11 July 2001; Volume 2611, pp. 1235–1245.
31. Hunt, R.; Johnston, M.; Zhang, M. Evolving "less-myopic" scheduling rules for dynamic job shop scheduling with genetic programming. In Proceedings of the 2014 Annual Conference on Genetic and Evolutionary Computation, Vancouver, BC, Canada, 12–16 July 2014; pp. 927–934.
32. Chang, Y.L.; Sueyoshi, T.; Sullivan, R.S. Ranking dispatching rules by data envelopment analysis in a job shop environment. *IIE Trans.* **1996**, *28*, 631–642. [CrossRef]