Genetic Programming based Hyper-heuristics for Dynamic Job Shop Scheduling: Cooperative Coevolutionary Approaches

John Park[1], Yi Mei[1], Su Nguyen[1,2], Aaron Chen[1], Mark Johnston[1], Mengjie Zhang[1]

[1]Evolutionary Computation Research Group, Victoria University of Wellington, PO Box 600, Wellington, 6140, New Zealand
[2]International University - VNU HCMC, Vietnam
{John.Park,Yi.Mei,Su.Nguyen,Aaron.Chen,Mengjie.Zhang}@ecs.vuw.ac.nz, Mark.Johnston@msor.vuw.ac.nz

Abstract. Job shop scheduling (JSS) problems are optimisation problems that have been studied extensively due to their computational complexity and application in manufacturing systems. This paper focuses on a dynamic JSS problem to minimise the total weighted tardiness. In dynamic JSS, jobs' attributes are only revealed after they arrive at the shop floor. Dispatching rule heuristics are prominent approaches to dynamic JSS problems, and Genetic Programming based Hyper-heuristic (GP-HH) approaches have been proposed to automatically generate effective dispatching rules for dynamic JSS problems. Research on static JSS problems shows that high quality ensembles of dispatching rules can be evolved by a GP-HH that uses cooperative coevolution. Therefore, we compare two coevolutionary GP approaches to evolve ensembles of dispatching rules for dynamic JSS problem. First, we adapt the Multilevel Genetic Programming (MLGP) approach, which has never been applied to JSS problems. Second, we extend an existing approach for static JSS problem, called Ensemble Genetic Programming for Job Shop Scheduling (EGP-JSS), by adding "less-myopic" terminals that take job and machine attributes outside of the scope of the attributes commonly used in the literature. The results show that MLGP for JSS evolves ensembles that are significantly better than single "less-myopic" rules evolved using GP with only little difference in computation time. In addition, the rules evolved using EGP-JSS perform better than the MLGP-JSS rules, but MLGP-JSS evolves rules significantly faster than EGP-JSS.

1 Introduction

Job shop scheduling (JSS) problems are combinatorial optimisation problems that have been studied over the past 50 years [1]. In a JSS problem instance, there are typically a fixed number of machines on the shop floor. Jobs arrive on the shop floor over time, and need to be processed on a specific sequence of machines. However, a machine can only process one job at a time. The sequence of jobs and the times when the jobs are selected to be processed by a machine is called a *schedule*. A JSS problem instance has a measure of "quality" for the final schedule called an *objective function* [2]. Therefore, the goal for solving a JSS problem instance is to find the schedule that will give the best possible outcome according to the objective function.

Various approaches to different JSS problems have been proposed in the literature. Exact mathematical optimisation techniques have been proposed [1, 2] to find the *optimal* (best feasible) solution to a static JSS problem instance. In a static JSS problem instance the properties of the jobs and the machines on the shop floor are known *a priori*. On the other hand, in a *dynamic* JSS problem instance [3] unforeseen events occur and affect the shop floor, e.g., the breakdown of a machine. Unforeseen events mean that it is not possible to determine whether a schedule is optimal until all arriving jobs have been processed, meaning that mathematical optimisation techniques are not suitable for dynamic JSS problems. Instead, researchers have proposed various heuristic techniques for dynamic JSS problems. Heuristic techniques range from simple dispatching rules [2] to complex meta-heuristic techniques [1]. A dispatching rule is a heuristic that iteratively selects a job to be processed by a machine whenever the machine becomes available. Dispatching rule approaches to dynamic JSS problem are prominent in the literature because in practice they can react quickly and effectively to unforeseen changes on the shop floor [4]. In addition, recent approaches have proposed a *Genetic Programming based Hyper-heuristic* (GP-HH) approaches for dynamic JSS problems [5, 6]. A GP-HH automatically evolves dispatching rules for a dynamic JSS problem, bypassing the need for human experts and extensive trial-and-error testing required to construct a dispatching rule manually. In addition, GP evolved rules generally perform better than manually designed rules [6]. However, there are many challenges that can arise when evolving dispatching rules using GP-HH. To generate a schedule for a JSS problem instance using a dispatching rule, there can be potential scenarios where complex decisions need to be made. Decisions made early in the schedule can greatly affect the overall quality of the schedule. However, many existing GP-HH approaches evolve "myopic" dispatching rules. These rules only take into account the current state of the job and the machine that the job is waiting at when selecting a job to process [7]. This can result in situations where the evolved rules make good local decisions, but make poor global decisions. In addition, most GP-HH approaches evolve single dispatching rules to handle dynamic JSS problems. In classification, research has shown that an *ensemble* of decision makers (i.e. a diverse set of "experts" that cover for each other's errors) generally performs better than a single decision maker for handling a complex and difficult classification problem [8]. Therefore, it may be more effective to evolve ensembles of dispatching rules instead of evolving single dispatching rules for JSS problems. However, research into evolving ensembles of dispatching rules for JSS problems has been limited.

The goal of this paper is to investigate two GP-HH approaches and determine which of the two approaches will evolve higher quality ensembles of dispatching rules for a dynamic JSS problem. The two GP-HH approaches combine coevolutionary techniques with GP to evolve ensembles, which have not been applied to dynamic JSS problems previously. The first GP-HH approach is an adaptation of the approach called Multilevel Genetic Programming (MLGP) [9]. MLGP has not previously been applied to JSS problems. MLGP is promising due to its ability to automatically find a group of individuals that work together effectively. The adaptation of MLGP for the dynamic JSS problem will be denoted as Multilevel Genetic Programming for Job Shop Scheduling (MLGP-JSS). In addition, a set of "less-myopic" terminals proposed by Hunt et al.

[7] will be used in MLGP-JSS. These terminals consist of job and machine attributes outside of the attributes associated with the current state of a job and the machine that the job is waiting at. By combining job and machine attributes which can reduce the myopic nature of dispatching rules [7] with MLGP-JSS, it may be possible to evolve a set of rules that can cover for each other's errors by handling different "locality" of decisions.

The second GP-HH approach is an extension of an existing GP-HH approach called Ensemble Genetic Programming for Job Shop Scheduling (EGP-JSS). EGP-JSS is an approach proposed by Park et al. [10], and uses Potter and De Jong's *cooperative co-evolution* [11] to evolve ensembles of dispatching rules for a static JSS problem. This will be updated with the "less-myopic" terminals and will be applied to the dynamic JSS problem for the first time. Therefore, the research objectives in this paper are to:

(a) Adapting the MLGP framework to the dynamic JSS problem.
(b) Incorporate the "less-myopic" terminals to the both MLGP-JSS and EGP-JSS for the dynamic JSS problem.
(c) Comparing the evolved rules from MLGP-JSS, EGP-JSS and a benchmark GP-HH that evolves single dispatching rules.

2 Background

This section firstly describes the notation and definitions for JSS, and approaches to dynamic JSS problems. Afterwards, GP-HH approaches to JSS are covered, along with some Cooperative Coevolutionary Algorithms (CCEAs) which have been proposed in the literature.

2.1 Dynamic Job Shop Scheduling

The notation used for JSS problem instances is as follows. In a JSS problem instance, there are M machines on the shop floor. A job j arrives at the shop floor requiring a sequence of N_j operations $\sigma_{1j}, \ldots, \sigma_{N_j j}$ which need to be processed in order for the job to be completed. The operations must be processed sequentially, i.e., operation σ_{2j} for a job j cannot be processed before σ_{1j}, etc. Each operation σ_{ij} of job j needs to be processed by a specific machine $m(\sigma_{ij})$. Job j needs to be processed at the machine m for time $p(\sigma_{ij})$ (shortened to p_{ij}), during which machine m cannot process any other jobs. The time when the last operation of job j is completed is called the completion time, denoted as C_j. The time when job j arrives at a machine to process operation σ_{ij} is called the arrival time, denoted as $r(\sigma_{ij})$. This means that job j arrives at the shop floor at time $r(\sigma_{1j})$, which is abbreviated to r_j. In a JSS problem with total weighted tardiness (TWT) minimisation, there are two additional attributes associated with a job j: a due date d_j and a penalty factor w_j. Job j is considered tardy if its completion time C_j is greater than its due date. Job j's tardiness T_j is given by $T_j = \max\{C_j - d_j, 0\}$. The goal of the JSS problem with TWT minimisation is to process all N jobs while minimising $\sum_{j=1}^{N} w_j T_j$.

A dispatching rule is applied to a JSS problem instance as follows. When a machine m becomes available, the dispatching rule first determines the set of jobs that can potentially be selected. The decision to select the job to be processed by machine m is called a *dispatching decision*. If the dispatching rule only considers the jobs already waiting at the machine, then it is said to generate a *non-delay* schedule [2]. On the other hand, a dispatching rule that considers future jobs which arrive earlier than a job which can be completed on the machine before the future job's arrival is said to generate an *active* schedule [2]. From the considered jobs, the dispatching rule selects a job to begin processing based on the job and the machine attributes. This process is repeated until all jobs that have arrived on the shop floor are completed and a schedule is generated. For example, shortest processing time (SPT) is a dispatching rule in which the SPT rule selects the job with the shortest processing time to be processed at a dispatching decision.

Constructing dispatching rules which use a good combination of job and machine attributes showed better performance than dispatching rules which use single attributes for specific dynamic JSS problems [12]. Therefore, many priority-based dispatching rule approaches have been proposed for dynamic JSS problems [12]. A priority-based dispatching rule consists of a priority function, which takes job, machine and shop floor attributes as inputs. The priority function assigns priority value to jobs waiting at a machine. The job with the highest priority is then selected to be processed by the machine. Examples of well-known priority-based dispatching rules are the apparent tardiness cost (ATC) and cost over time (COVERT) rules [13] for dynamic JSS problems with the TWT minimisation objective.

2.2 Genetic Programming based Hyper-heuristics for Dynamic JSS

Many GP-HH approaches have been proposed to generate reusable scheduling heuristics. In most cases, the output heuristic is a dispatching rule [6]. Hildebrandt et al. [14] evolved priority-based dispatching rules for a dynamic JSS problem that models a semiconductor manufacturing environment. They showed that the rules evolved using GP perform significantly better than existing man-made dispatching rules. Nguyen et al. [4] explored three different GP representations for static JSS problems to evolve dispatching rules, and applied the evolved rules to a dynamic JSS problem. Although the evolved dispatching rules outperformed benchmark man-made dispatching rules for the static JSS problems, they did not perform as well as man-made dispatching rules for the dynamic JSS problem. Pickardt et al. [15] proposed a two-step procedure, where priority-based dispatching rules are evolved using GP, and combined with existing rules using genetic algorithm (GA) to assign the rules to specific machines on the shop floor. The two-step approach performed significantly better than evolving rules using GP or allocating rules to specific machines using GA separately, and outperformed the state-of-the-art approaches to a dynamic semiconductor manufacturing problem. Hunt et al. [16] evolved priority-based dispatching rules for a dynamic two-machine JSS problem with TWT minimisation objective. The evolved rules performed significantly better than some benchmark dispatching rules. Hunt et al. [7] then proposed a set of "less-myopic" terminals to the GP process to reduce the myopic nature of dispatching rules. They showed that the rules evolved with the added terminals performed significantly better than the benchmark GP-HH approach which uses the standard job and machine properties.

2.3 Cooperative Coevolution in Genetic Programming

CCEAs are techniques where behaviours from multi-agent system are incorporated with evolutionary computation [17]. First, the problem is decomposed into smaller subproblems. Afterwards, the agents that make up the multi-agent system are applied to the subproblems. In CCEA approaches, the aim is to allow different individuals of the population to fill different "ecological niches" [11]. The evolved individuals that can best handle the different niches are then combined together as a cohesive solver. A survey of CCEA techniques is provided by Panait and Luke [17].

Nguyen et al. [18] used Potter and De Jong's cooperative coevolution [11] to evolve multiple scheduling policies for a dynamic JSS problem with three objectives. In their approach, they used two subpopulations. The individuals in the first subpopulation were used to assign due dates to the jobs during a dispatching decision. The individuals in the second subpopulation were used as priority-based dispatching rules. Park et al. [10] used cooperative coevolution to evolve ensembles of dispatching rules for a static JSS problem with the makespan minimisation objective. In this approach, individuals in all subpopulations are "voters", and vote on the job to be selected for processing. Other CCEA techniques include Orthogonal Evolution of Teams (OET) [19] and MLGP [9]. OET groups individuals into separate "teams" that compete against other teams during the selection procedure. MLGP is a CCEA which evolves multiple levels of evolution simultaneously. Neither OET and MLGP have previously been applied to dynamic JSS problems. In addition, Potter and De Jong's cooperative coevolution has not been applied to dynamic JSS problems to evolve ensembles of dispatching rules.

3 CCGP Approaches to JSS: MLGP-JSS and EGP-JSS

The two CCGP approaches for evolving ensembles of dispatching rules to the dynamic JSS problem with TWT minimisation objective are given below. The first approach, MLGP-JSS, adapts the MLGP approach of Wu and Banzhaf [9]. The description of MLGP-JSS is broken down into two steps: a high-level overview of the approach, and the evaluation procedure, which includes the the fitness functions used. The second approach extends EGP-JSS proposed by Park et al. [10] that uses Potter and De Jong's cooperative coevolution [11]. Finally, the representation of individuals used for MLGP-JSS and EGP-JSS is described.

3.1 MLGP-JSS process overview

A key component of MLGP-JSS is *groups*. A group is a set of individuals in the population that cooperate with each other. For our approach, we use groups as ensembles to solve JSS problem instances. This is discussed in further detail in Section 3.3. The MLGP-JSS process is broken down into three major steps. The first step is to carry out evolution on the *group level*, where groups are bred, evaluated and added to the GP population. The second step is to carry out evolution on the *individual level*, where GP individuals are bred, evaluated and added to the population. The final step is the

Algorithm 1 Algorithm for MLGP-JSS.

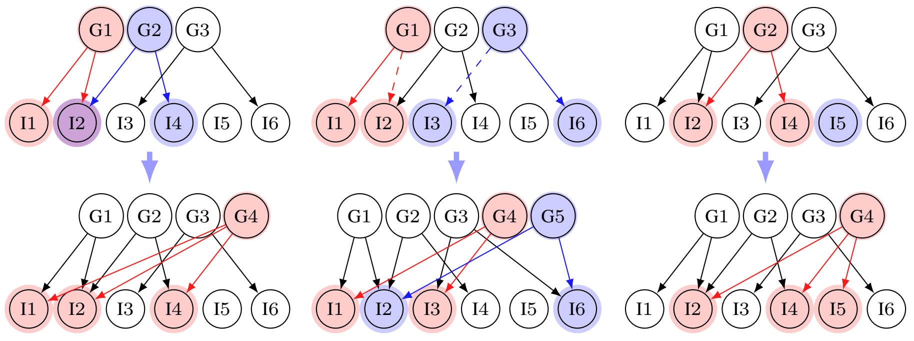
```
Initialise Population $P$ with $I_R$ Individuals;
Evaluate Fitnesses of Individuals $idv_1, \ldots, idv_{I_R} \in P$;
while Maximum generation is not reached do
    for $g \leftarrow 1$ to $G_B$ do
        Breed a Group $grp$ from Population $P$;
        Evaluate the Fitness of Group $grp$;
        Add Group $grp$ to Population $P$;
    end
    for $i \leftarrow 1$ to $I_B$ do
        Breed an Individual $idv$ from Population $P$;
        Evaluate the Fitness of Individual $idv$;
        Add Individual $idv$ to Population $P$;
    end
    Retain the $G_R$ Best Groups and the $I_R$ Best Individuals from Population $P$;
end
```

selection procedure, where only the elite groups and individuals are retained in the population for the next generation. After the termination criterion, i.e., the maximum number of generations, is reached, the final output is the best group of individuals found so far, an ensemble of dispatching rules that can be applied to dynamic JSS problem instances. The overall MLGP-JSS process is shown in Algorithm 1, where G_B is the number of groups bred at each generation, and I_B is the number of individuals bred at each generation.

Evolution on the Group Level: There are three evolutionary operators which breed new groups from existing individuals and groups in the population. The first operator is *cooperation*. Cooperation combines two *entities* together to form a new group containing all individuals from both entities without duplicates. An entity can be either an individual or a group. This means that two individuals, an individual and a group, or two groups can be merged to form another group. The entities for cooperation are selected using roulette wheel selection over all entities. An example is shown in Fig. 1a, where group G1 is combined with individual G2 to form the new group G4, which contains individuals I1, I2 and I4.

The second evolutionary operator on the group level is the group crossover operator. In group crossover, roulette wheel selection selects two groups from the GP population as parents. The two parents randomly exchange one individual to produce the child groups. Individuals in a parent group have equal probabilities of being exchanged. In Fig. 1b, crossover occurs between parents G1 and G3, exchanging individuals I2 and I3 respectively. This generates groups G4 and G5.

The final evolutionary operator on the group level is the group mutation operator. In the group mutation, a group is selected through roulette wheel selection, and either an individual is added to or removed from the group. If an individual is being added, then an individual is selected through roulette wheel selection over the individuals in the GP population. If an individual is being removed, then an individual in the group is randomly selected with uniform probability. An example of mutation operator adding an



(a) Cooperation Operator (b) Crossover Operator (c) Mutation Operator

Fig. 1: Examples of group operators used to breed new groups.

individual is shown in Fig. 1c, where individual I5 is added to G2 to produce offspring G4.

Evolution on the Individual Level: Crossover and mutation operators used to breed the individuals are the standard operators for tree-based GP [20]. To select the parent individuals to breed new offspring, a group grp is selected based on a probability proportional to the group's fitness. It is expected that individuals in a group contribute to achieve cooperation despite their fitnesses [9]. Therefore, an appropriate number of parent individuals are selected from group grp with a uniform probability. After crossover or mutation is carried out, the newly bred children do not automatically become part of the group from which the parent individuals were selected from, but are inserted back into the pool of individuals after evaluating the children's fitness.

3.2 Selection

After G_B groups and I_B individuals are breed and added to the population, the selection procedure retains the G_R groups with the highest fitnesses out of all the groups in the population, while discarding the groups with fitnesses lower than the top G_R groups. A similar procedure is then applied to the individuals, where the top I_R fittest individuals in the population are retained and the rest discarded. However, if an individual idv does not belong to the top I_R fittest individuals, but belongs to a group grp, then individual idv is still retained by group grp. This is due to the fact that individual idv may have poor individual fitness, but work well together with other individuals in group grp.

3.3 Evaluation Procedure

To evaluate a group grp in the population, group grp is applied as a non-delay [2] dispatching rule to the set Δ_{train} of JSS training instances. When a machine becomes available in a discrete-event simulation of the job shop on JSS problem instance I, the individuals that are part of group grp act as an ensemble, "voting" on the next job to be processed by the machine. An individual in group grp first assigns priorities to the non-delay jobs waiting at the machine, and then votes on the job with the highest

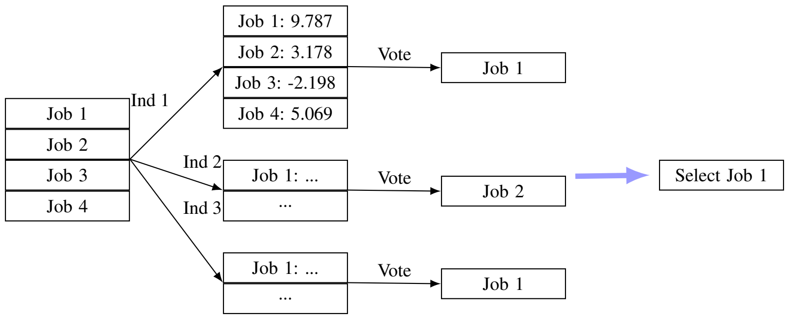


Fig. 2: An example of a voting procedure for individuals in a group.

assigned priority. The job with the most number of votes is then selected by the group to be processed. In the case of a tie in the number of votes between two jobs, the ATC dispatching rule [13] is used as a tie-breaker. The job with the higher priority assigned by the ATC rule is ranked higher than the other job. An example of the voting procedure is shown in Fig. 2 for a group consisting of 3 individuals. The individuals vote out of 4 jobs waiting at the machine to be processed. Rule 1 votes for job 1 which has the priority of 9.787, rule 2 votes for job 2, and rule 3 votes for job 1, resulting in job 1 being selected to be processed.

After group grp is applied to the training instances, we get the fitness Fit_{grp} of group grp as shown in Equation (1). The group fitness function used as the dynamic JSS problem is modified from the group fitness function proposed by Wu and Banzhaf [9]. In the equation, $Obj(grp, I)$ is the TWT objective value of the solution generated for a JSS problem instance I by group grp. GS_{grp} denotes the size of group grp, and is used for a penalty factor to prevent groups from increasing in size with minimal improvement in performance [9].

$$Fit_{grp} = \frac{1}{|\Delta_{\text{train}}|} \sum_{I \in \Delta_{\text{train}}} Obj(grp, I) + \sqrt{\frac{2 + \text{GS}_{grp}}{2 \times \text{GS}_{grp}}} \tag{1}$$

On the other hand, to evaluate an individual idv, individual idv is applied to the JSS training instances as a non-delay priority-based dispatching rule. The fitness of individual idv, denoted as Fit_{idv}, is the average TWT of the solutions over the training instances as shown in Equation (2).

$$Fit_{idv} = \frac{1}{|\Delta_{\text{train}}|} \sum_{I \in \Delta_{\text{train}}} Obj(idv, I) \tag{2}$$

3.4 EGP-JSS process overview

The EGP-JSS approach incorporates Potter and De Jong's cooperative coevolution [11] to evolve ensembles of dispatching rules. EGP-JSS is an extension of a previous approach by Park et al. [10] for a static JSS problem, where they provide a full description of the EGP-JSS approach. In cooperative coevolution [11], a population of individuals is partitioned into subpopulations, and individuals from each subpopulation only interact with representatives of other subpopulation. A representative in a subpopulation is the best individual of the subpopulation found so far. To evaluate an individual of a subpopulation, the individual is grouped up with the representatives of the other subpopulations to form an ensemble of dispatching rules. The ensemble is then applied to the training instances. An ensemble's voting procedure for selecting jobs is given by Fig. 2, and the ATC rule is used as the tie-breaker between jobs with equal number of votes. The fitness of the ensemble over the training instances is used as the fitness of the individual.

3.5 GP Representation, Terminals and Function Sets

For MLGP-JSS and EGP-JSS, we use a tree-based GP, where the individuals represent arithmetic function trees. An individual's function tree is used as a priority function in a non-delay dispatching rule. When a machine m^* becomes available at time R_{m^*}, the non-delay jobs waiting in the queue in front of machine m^* are considered for selection. The set of non-delay jobs in the queue front of machine m^* will be denoted as A_{m^*}. The priority of the operation for a job $j \in A_{m^*}$ waiting at machine m^* is calculated by inputting the attributes of j and the machines on the shop floor into the function tree.

The terminals in the GP terminal set either correspond to base level attributes of jobs and machines or are constructed from multiple base level attributes. Examples of a base level attribute are the due date (d_j) or the processing time ($p(\sigma_{kj})$) of the kth operation of job j. As an example, suppose that job j is waiting at machine m^* during a dispatching decision, requiring the ith operation σ_{ij} to be processed by the machine. When job j is inputted into an individual's function tree, the DD terminal corresponds to the due date d_j of job j. The RT terminal corresponds to the sum of the remaining total processing times of input job j from the ith operation to the final operation, i.e., $\sum_{k=i}^{N_j} p(\sigma_{kj})$.

The terminal set consists of "basic" terminals and the "less-myopic" terminals. The basic terminals are terminals used by Nguyen et al. [4] to evolve rules using arithmetic representations for individuals in the GP population. The less-myopic terminals are terminals proposed by Hunt et al. which are used to evolve "wider-looking" rules [7]. The less-myopic terminals are used because they allow individuals that are part of the ensemble to focus on different "locality" of the decisions, with some individuals focusing on very local decisions and other individuals focusing on decisions further down the line. An example of Hunt et al's [7] less-myopic terminal is the average wait time of last 5 jobs processed at the next machine (NQW). Let $q_{1m}, \ldots, q_{N_m m}$ be the waiting times of the 1st, $\ldots$, N_mth jobs processed by machine m, and let m_j^{**} be the machine for which the job j's next operation is to be processed on. Then the NQW terminal corresponds to $\frac{1}{5}\sum_{k=1}^{5} q_{(N_{m_j^{**}}+1-k),m_j^{**}}$ for j. On the other hand, the function set consists of arithmetic operators $+$, $-$, $\times$, protected $/$, `if`, $\max$ and $\min$. Protected division returns a value of 1 if the denominator is zero. `if` is a ternary operator which returns the value of the second argument if the value of the first argument is ≥ 0, and returns the value of the third argument otherwise. $\max$ and $\min$ are binary operators which return the maximum and minimum of the arguments respectively. The full list of terminals and functions are given in Table 1.

Table 1: The terminal set used for the GP representation, where job j is one of the job waiting at the machine m^* to process operation σ_{ij}.

Terminal		Description	Value
Basic	RJ	Operation ready time	$r(\sigma_{ij})$
	RO	Remaining number of operations of job j	$N_i - m + 1$
	RT	Remaining total processing times of job j	$\sum_{k=i}^{N_j} p(\sigma_{jk})$
	PR	Operation processing time of job j	$p(\sigma_{ij})$
	RM	Machine m^* ready time	R_{m^*}
	NJ	Non-delay jobs waiting at machine m^*	$\lvert A_{m^*}\rvert$
	DD	Due date of job j	d_j
	W	Tardiness penalty of job j	w_j
	#	Constant	Uniform[0,1]
Less-myopic	NPR	Next operation processing time	$p(\sigma_{(i+1)j})$
	NNQ	Number of idle jobs waiting at the next machine	$\lvert A_{m(\sigma_{(i+1)j})}\rvert$
	NQW	Average waiting time of last 5 jobs at the next machine	$\frac{1}{5}\sum_{k=1}^{5} q_{(N_{m_j^{**}}+1-k)m_j^{**}}$
	AQW	Average waiting time of last 5 jobs at all machines	$\frac{1}{5M}\sum_{m=1}^{M}\sum_{k=1}^{5} q_{(N_m+1-k)m}$
Function		$+, -, \times, /,$ `if`, max, min	

4 Experimental Design

This section covers the dataset that is used for training and testing. Description of the benchmark GP-HH approach used to compare with MLGP-JSS and EGP-JSS will also be presented. In addition, this section gives the GP parameter settings used to evolve the rules.

4.1 Dataset

The effectiveness of evolved rules can be significantly affected by the number and the frequency of job arrivals [4, 14]. In addition, long-running discrete-event simulations are commonly used to evaluate rules for dynamic JSS problems [6]. On the other hand, applying a dispatching rule to a simulation is expensive, and a large training set may not necessarily lead to significantly higher quality heuristics [6]. Therefore, the dataset that is used for evaluating the GP-HH approaches are discrete-event simulations that were used by Hunt et al. [7], where a small training and test set with large number of job arrival is used. Each problem instance in the dataset has 10 machines, and jobs are randomly generated. A job's operations have random processing times that follow a discrete uniform distribution with mean μ. Jobs arrive according to a Poisson process with mean λ, where λ is configured so that it meets a desired utilisation rate ρ. Therefore, $\lambda = \frac{\rho \times p_M}{(1/\mu)}$, where p_M is the expected number of operations per job divided by the number of machines. If all jobs arriving on the shop floor require 4 operations in a 10 machine environment shop, then $p_M = 0.4$. Due date d_j of job j is given by

Table 2: Configurations used for the discrete-event simulation representing dynamic JSS problem instances.

Training, 4op	Mean processing time, μ	25
	Expected utilisation rate, ρ	$0.85, 0.95$
	Due date tightness, h	$\{3, 5, 7\}$
	# of operations per job	4
	# of configurations	2
Training, 8op	Mean processing time, μ	25
	Expected utilisation rate, ρ	$0.85, 0.95$
	Due date tightness, h	$\{3, 5, 7\}$
	# of operations per job	8
	# of configurations	2
Testing	Mean processing time, μ	$25, 50$
	Expected utilisation rate, ρ	$0.90, 0.97$
	Due date tightness, h	$\{2, 4, 6\}$
	# of operations per job	$4, 6, 8, 10, X \sim \text{Unif}\ (2, 10)$
	# of configurations	20

$d_j = r_j + h \sum_{i=1}^{N_j} p(\sigma_{ji})$, where h is the due date tightness parameter. For each problem instance, a list of possible due date tightness values is provided. A job's due date tightness h is randomly selected from the list with an equal probability. Penalty factor w_j for tardy job j is selected from 1, 2, or 4 with probabilities of 0.2, 0.6 and 0.2 respectively [2]. There is a warm up period of 500 jobs. Thereafter, the weighted tardiness values of subsequent jobs are added to the objective function value. The jobs stop arriving after $N = 2500$ jobs have been processed, but all jobs that have arrived at the shop floor must be completed.

To evolve the rules, there are two training sets which have a different number of operations for each job. A problem instance in the first training set, 4op, has 4 operations per job. A problem instance in the second training set, 8op, has 8 operations per job. Both training sets have 2 problem instances, where the first problem instance in both sets have $\rho = 0.85$, and the second problem instance in both sets have $\rho = 0.95$. On the other hand, the test set has 20 problem instances, divided into four subsets of five problem instances. In a subset, the first four problem instances have $4, 6, 8, 10$ operations per job. The last problem instance in the subset has anywhere between 2 and 10 operations per job, where the number of operations per job is random with an equal probability. Therefore, $p_M = 0.4, 0.6, 0.8, 1.0, 0.6$ for each problem instance in the subset respectively. The five instances in each of the four subsets have $\mu = 25, 50, 25, 50$ and $\rho = 0.90, 0.90, 0.97, 0.97$. For all problem instances the machine which the operation is carried out is random, and there is no re-entry for the jobs, i.e., no two operations for a single job occurs on the same machine. The overall parameter settings for the dataset is shown in Table 2.

4.2 GP-HH Benchmark Methods for Comparison

As MLGP-JSS has not been applied to JSS problems, it will first be compared against a benchmark GP-HH approach. As the most prominent method for evolving dispatching

Table 3: GP parameters used by the GP-HH approaches for evolving ensembles

Approaches		GP-JSS	EGP-JSS	MLGP-JSS
Population related	Number of subpopulations	1	4	1
	Subpopulation size	1024	256	1024
Common Parameters	Generations	51		
	Crossover rate	80%		
	Mutation rate	10%		
	Reproduction rate	10%		
	Max-depth	8		
	Selection method	Tournament Selection		
	Selection size	7		
MLGP Specific	Number of groups breed G_B	-	-	200
	Number of groups retained G_R	-	-	100
	Group cooperation rate	-	-	31.25%
	Group crossover rate	-	-	50%
	Group mutation rate	-	-	18.75%
Tie-breaker Specific	ATC k value	-	3.0	3.0
Function set		$if, +, -, \times, /, max, min$		
Terminal set		as shown in Table 1		

rules [6], the benchmark GP-HH approach in this paper evolves single priority-based dispatching rules. The benchmark GP-HH will be abbreviated to GP-JSS, and uses the same terminals as both MLGP-JSS and EGP-JSS (Table 1), which includes the less-myopic terminals.

4.3 Parameter Settings

The following parameters are used for GP-JSS, EGP-JSS and MLGP-JSS. The total population size for all three GP-HH is set to 1024. Similar to parameter configurations used by Park et al. [10], the population for EGP-JSS is divided to 4 subpopulations of size 256. The rest of the GP parameter configuration is the same as the configuration suggested by Koza [20]. However, tournament selection is used in the MLGP process when there are no groups in the population. The additional parameters for MLGP-JSS are the number of individuals breed (I_B), the number of groups bred (G_B), the number of groups retained (G_R), and the group cooperation/crossover/mutation rate. The number of individuals bred is equal to the number of individuals in the population, i.e., 1024. MLGP-JSS starts off with no groups in the population, and the number of groups breed and retained are set to 200 and 100 respectively. The probability of using a specific group operator is based off the probability used by Wu and Banzhaf [9] for their MLGP approach. Finally, the k value for the tie-breaker ATC heuristic is set to k [13]. All these parameter settings are listed in Table 3.

Table 4: The evolution times and TWTs for the MLGP-JSS rules and the GP-JSS rules over the training and the test JSS problem instances.

Approach		4op		8op	
		MLGP-JSS	GP-JSS	MLGP-JSS	GP-JSS
Evolution Time (s)		7079 ± 1532	6701 ± 3337	6202 ± 1456	5418 ± 1932
TWT $(\times 10^5)$	Training Set	4.74 ± 1.53	4.69 ± 1.54	4.74 ± 1.55	4.82 ± 1.85
	$\langle 25, 0.90 \rangle$	4.50 ± 0.33	5.17 ± 1.17	4.52 ± 0.33	5.37 ± 1.35
		4.61 ± 0.51	5.34 ± 1.88	4.70 ± 0.66	5.26 ± 1.81
		11.88 ± 0.88	14.27 ± 4.83	11.93 ± 0.92	14.70 ± 4.87
		8.68 ± 0.64	9.83 ± 2.20	8.67 ± 0.64	10.31 ± 2.79
		17.46 ± 1.66	20.08 ± 3.31	17.44 ± 1.56	20.59 ± 4.19
	$\langle 50, 0.90 \rangle$	12.66 ± 0.98	14.33 ± 3.06	12.54 ± 0.98	14.95 ± 3.90
		31.89 ± 3.03	38.30 ± 6.34	31.76 ± 3.15	39.43 ± 6.99
		35.32 ± 5.77	43.18 ± 7.45	35.09 ± 5.88	44.37 ± 8.71
		3.56 ± 0.33	4.61 ± 1.79	3.62 ± 0.38	4.84 ± 2.03
		25.92 ± 3.01	34.27 ± 9.20	25.69 ± 2.33	35.64 ± 11.20
	$\langle 25, 0.97 \rangle$	40.13 ± 4.18	45.96 ± 6.14	40.03 ± 4.31	46.20 ± 6.09
		64.07 ± 14.23	92.04 ± 58.84	63.98 ± 13.77	88.68 ± 38.84
		11.23 ± 1.01	13.54 ± 3.73	11.43 ± 1.00	13.99 ± 4.45
		15.53 ± 1.78	21.71 ± 9.24	15.46 ± 1.71	20.74 ± 7.30
		73.13 ± 3.90	84.67 ± 11.92	75.45 ± 8.97	84.82 ± 10.85
	$\langle 50, 0.97 \rangle$	79.52 ± 5.96	93.83 ± 22.25	79.96 ± 6.85	96.82 ± 18.52
		6.22 ± 0.58	10.29 ± 17.80	6.09 ± 0.46	7.13 ± 2.10
		11.61 ± 1.01	14.00 ± 3.56	11.44 ± 0.86	14.45 ± 3.87
		21.01 ± 1.90	26.61 ± 5.95	20.91 ± 1.83	26.59 ± 4.96
		56.86 ± 4.46	72.84 ± 11.19	57.54 ± 5.05	72.57 ± 13.03

5 Results

This section covers the results of the evaluation between the different GP-HH approaches, which is broken down into two steps. First, MLGP-JSS will be evaluated against GP-JSS. Second, MLGP-JSS will be evaluated against EGP-JSS in order to determine which CCEA approach out of the two performs better on dynamic JSS problems. For each GP-HH approach, rules are evolved from the training sets 4op and 8op over 30 independent runs. The computation times taken to evolve a rule for the GP-HH are also measured and analysed.

5.1 MLGP-JSS vs GP-JSS

The evolved rules from MLGP-JSS and GP-JSS are applied to the 20 JSS problem instances in the test set. From this, we get sets of TWT solution values for each JSS problem instance that are used to compare the performance of the rules. For a fair comparison, the MLGP-JSS rules evolved over a specific training set are compared against the GP-JSS rules evolved over the same training set, e.g., rules evolved over training set 4op are compared against each other. One set of rules is significantly better than another set of rules for a particular JSS problem instance if the obtained p-value under the one-sided Z-test is less than 0.05. The results are shown in Table 4. In the table, $\langle x, y \rangle$ denotes that $\mu = x$ and $\rho = y$ for the JSS problem instance. The problem instance for which the MLGP-JSS rules are significantly better than the GP-JSS rules are highlighted, and vice versa.

From the results, the MLGP-JSS rules perform significantly better than the GP-JSS rules for all problem instances. In addition, although the time required to evolve the MLGP-JSS rules is slightly greater than the time required to evolve GP-JSS rules, the

Table 5: The evolution times and TWTs for the MLGP-JSS rules and the EGP-JSS rules over the training and the test JSS problem instances.

Approach		4op		8op	
		MLGP-JSS	EGP-JSS	MLGP-JSS	EGP-JSS
Evolution Time (s)		7079 ± 1532	16555 ± 5468	6202 ± 1456	21128 ± 7243
TWT $(\times 10^5)$	Training Set	4.74 ± 1.53	3.83 ± 1.00	4.74 ± 1.55	3.81 ± 0.94
	$\langle 25, 0.90\rangle$	4.50 ± 0.33	3.82 ± 0.62	4.52 ± 0.33	3.82 ± 0.66
		4.61 ± 0.51	4.50 ± 0.92	4.70 ± 0.66	4.57 ± 1.09
		11.88 ± 0.88	12.88 ± 3.45	11.93 ± 0.92	12.51 ± 2.35
		17.46 ± 1.66	15.95 ± 2.54	17.44 ± 1.56	16.46 ± 2.94
		8.68 ± 0.64	7.45 ± 1.30	8.67 ± 0.64	7.47 ± 1.30
	$\langle 50, 0.90\rangle$	12.66 ± 0.98	10.92 ± 1.70	12.54 ± 0.98	10.91 ± 1.76
		31.89 ± 3.03	29.09 ± 6.01	31.76 ± 3.15	28.70 ± 5.55
		35.32 ± 5.77	33.06 ± 6.16	35.09 ± 5.88	33.27 ± 6.61
		3.56 ± 0.33	3.29 ± 0.73	3.62 ± 0.38	3.27 ± 0.68
		25.92 ± 3.01	21.99 ± 5.08	25.69 ± 2.33	21.65 ± 5.01
	$\langle 25, 0.97\rangle$	40.13 ± 4.18	44.84 ± 7.78	40.03 ± 4.31	45.07 ± 8.11
		64.07 ± 14.23	80.52 ± 36.05	63.98 ± 13.77	78.94 ± 27.27
		11.23 ± 1.01	10.20 ± 1.99	11.43 ± 1.00	9.98 ± 1.77
		15.53 ± 1.78	13.59 ± 2.95	15.46 ± 1.71	13.52 ± 2.71
		73.13 ± 3.90	71.60 ± 15.63	75.45 ± 8.97	70.27 ± 14.08
	$\langle 50, 0.97\rangle$	79.52 ± 5.96	70.90 ± 13.03	79.96 ± 6.85	68.32 ± 13.05
		6.22 ± 0.58	5.41 ± 0.99	6.09 ± 0.46	5.42 ± 0.90
		11.61 ± 1.01	10.19 ± 1.79	11.44 ± 0.86	10.15 ± 1.79
		21.01 ± 1.90	19.61 ± 3.21	20.91 ± 1.83	19.33 ± 2.96
		56.86 ± 4.46	56.65 ± 9.00	57.54 ± 5.05	55.00 ± 10.03

difference is not statistically significant. Therefore, the MLGP-JSS is more effective than the standard GP-HH approach for the dynamic JSS problems with TWT minimisation objective. This supports the hypothesis that an ensemble of dispatching rules is more capable of handling complex decisions in a dynamic JSS problem instance than a single rule. Hence, it would be promising to carry out further research into evolving new and effective ensembles of rules using coevolutionary GP for dynamic JSS problems.

5.2 MLGP-JSS vs EGP-JSS

The MLGP-JSS and EGP-JSS rules evolved from the same training set Δ_{train} are compared against each other over the test set in terms of the qualities of the solutions generated, i.e., the TWT of the solutions for the JSS problem instances. The results are shown in Table 5, where a set of rules that is significantly better than the other set of rules for a particular JSS problem instance is highlighted.

From the results, the EGP-JSS rules have significantly lower TWT values than the MLGP-JSS rules for most of the JSS problem instances in the test set. A possible reason for this observation may be due to the MLGP-JSS's breeding and selection procedure. As described in Section 3.1, the MLGP uses roulette wheel selection for choosing different groups to breed new groups and individuals. During the training procedure, the groups generally have similar fitnesses to each other. This means that roulette wheel selection may not be biased enough towards a potentially good group, resulting in a lack of exploitation. Both exploitation, the idea of greedily pursuing the local optima, and exploration, the idea of moving towards areas of the search space to potentially find better points, need to be finely balanced for a GP-HH to find good heuristics in the heuristic space [20].

Another possible reason why the EGP-JSS rules perform better than the MLGP-JSS rules may be linked to the difference in time taken to evolve the rules using EGP-JSS and MLGP-JSS and the performance of the evolved rules on the training sets. MLGP-JSS requires much lower computation time to evolve the rules during the training procedure than EGP-JSS. On average, MLGP-JSS took less than half of the amount of time to evolve the rules over 4op compared to EGP-JSS, and slightly more than a third the amount of time to evolve the rules over 8op. During an evaluation procedure, the number of times the individuals in the GP population is applied to the training instances is significantly lower for MLGP-JSS than EGP-JSS. Therefore, it might be the case that EGP-JSS takes a longer time to explore for good rules in a single generation than MLGP-JSS. This is supported by the differences in the performance over the training sets of the MLGP-JSS rules and the EGP-JSS rules, the EGP-JSS rules perform slightly better than the MLGP-JSS rules. However, the differences in the performances over the training sets are not statistically significant.

In summary, two major findings were made:

1. MLGP-JSS evolves significantly better rules than GP-JSS for the dynamic JSS problem, supporting the hypothesis that ensembles of dispatching rules are better than single dispatching rules. The differences in the evolution times is insigificant.
2. EGP-JSS evolves significantly better rules than MLGP-JSS for the dynamic JSS problem, but EGP-JSS takes longer to evolve the rules than MLGP-JSS.

6 Conclusions

In this paper, we proposed the MLGP-JSS approach for the dynamic JSS problem with the TWT minimisation objective. MLGP-JSS is an adaptation of the MLGP approach proposed by Wu and Banzhaf [9] that has not been applied to dynamic JSS problems. In addition, we extended the EGP-JSS approach by Park et al. for a static JSS problem [10] by incorporating new “less-myopic” terminals [7] and applying it to a dynamic JSS problem for the first time. The experimental results show that MLGP-JSS evolves rules with insignificant increase in time compared to a benchmark GP-HH approach but with significant improvements in the performance. On the other hand, EGP-JSS produces higher quality evolved rules than MLGP-JSS, but MLGP-JSS evolves rules significantly faster than EGP-JSS.

For future work, it may be promising to investigate alternative selection methods during the breeding and selection procedure of MLGP, such as replacing the roulette wheel selection with tournament selection. In addition, it is likely that MLGP-JSS and EGP-JSS can evolve better rules by adding niching or diversity measures, which have been shown in classification to improve the quality of the ensembles [8]. In addition, the evolved rules from both EGP-JSS and MLGP-JSS will also be investigated for further analysis.

References

1. Potts, C.N., Strusevich, V.A.: Fifty years of scheduling: a survey of milestones. Journal of the Operational Research Society **60** (2009) S41–S68

2. Pinedo, M.L.: Scheduling: Theory, Algorithms, and Systems. 4 edn. Springer (2012)
3. Ouelhadj, D., Petrovic, S.: A survey of dynamic scheduling in manufacturing systems. Journal of Scheduling **12**(4) (2009) 417–431
4. Nguyen, S., Zhang, M., Johnston, M., 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
5. Burke, E.K., Gendreau, M., Hyde, M., Kendall, G., Ochoa, G., Ozcan, E., Qu, R.: Hyper-heuristics: a survey of the state of the art. Journal of the Operational Research Society **64**(12) (2013) 1695–1724
6. Branke, J., Nguyen, S., Pickardt, C., Zhang, M.: Automated design of production scheduling heuristics: A review. 10.1109/TEVC.2015.2429314 (2015)
7. Hunt, R., Johnston, M., Zhang, M.: Evolving "less-myopic" scheduling rules for dynamic job shop scheduling with genetic programming. In: Proceedings of the 2014 Conference on Genetic and Evolutionary Computation. (2014) 927–934
8. Polikar, R.: Ensemble based systems in decision making. IEEE Circuits and Systems Magazine **6**(3) (2006) 21–45
9. Wu, S.X., Banzhaf, W.: Rethinking multilevel selection in genetic programming. In: Proceedings of the 13th Annual Conference on Genetic and Evolutionary Computation. (2011) 1403–1410
10. Park, J., Nguyen, S., Zhang, M., Johnston, M.: Evolving ensembles of dispatching rules using genetic programming for job shop scheduling. Proceedings of the 2015 European Conference on Genetic Programming **9025** (2015) 92–104
11. Potter, M.A., De Jong, K.A.: Cooperative coevolution: An architecture for evolving coadapted subcomponents. Evolutionary Computation **8**(1) (2000) 1–29
12. Jayamohan, M.S., Rajendran, C.: Development and analysis of cost-based dispatching rules for job shop scheduling. European Journal of Operational Research **157**(2) (2004) 307–321
13. Vepsalainen, A.P.J., Morton, T.E.: Priority rules for job shops with weighted tardiness costs. Management Science **33**(8) (1987) 1035–1047
14. Hildebrandt, T., Heger, J., Scholz-Reiter, B.: Towards improved dispatching rules for complex shop floor scenarios: A genetic programming approach. In: Proceedings of the 12th Annual Conference on Genetic and Evolutionary Computation. (2010) 257–264
15. Pickardt, C.W., Hildebrandt, T., Branke, J., Heger, J., Scholz-Reiter, B.: Evolutionary generation of dispatching rule sets for complex dynamic scheduling problems. International Journal of Production Economics **145**(1) (2013) 67–77
16. Hunt, R., Johnston, M., Zhang, M.: Evolving machine-specific dispatching rules for a two-machine job shop using genetic programming. In: Proceedings of the IEEE Congress on Evolutionary Computation. (2014) 618–625
17. Panait, L., Luke, S.: Cooperative multi-agent learning: The state of the art. Autonomous Agents and Multi-Agent Systems **11**(3) (2005) 387–434
18. Nguyen, S., Zhang, M., Johnston, M., Tan, K.C.: A coevolution genetic programming method to evolve scheduling policies for dynamic multi-objective job shop scheduling problems. In: CEC '12: Proceedings of the IEEE Congress on Evolutionary Computation. (2012) 1–8
19. Soule, T., Komireddy, P.: Orthogonal evolution of teams: A class of algorithms for evolving teams with inversely correlated errors. In: Genetic Programming Theory and Practice IV. Volume 5 of Genetic and Evolutionary Computation. Springer (2007) 79–95
20. Koza, J.R.: Genetic Programming: On the Programming of Computers by Means of Natural Selection. MIT Press (1992)