2014 IEEE Congress on Evolutionary Computation (CEC)
July 6-11, 2014, Beijing, China

Evolving Machine-Specific Dispatching Rules for a Two-Machine Job Shop using Genetic Programming

Rachel Hunt Mark Johnston Mengjie Zhang

***Abstract*—Job Shop Scheduling (JSS) involves determining a schedule for processing jobs on machines to optimise some measure of delivery speed or customer satisfaction. We investigate a genetic programming based hyper-heuristic (GPHH) approach to evolving dispatching rules for a two-machine job shop in both static and dynamic environments. In the static case the proposed GPHH method can represent and discover optimal dispatching rules. In the dynamic case we investigate two representations (using a single rule at both machines and evolving a specialised rule for each machine) and the effect of changing the training problem instances throughout evolution. Results show that relative performance of these methods is dependent on the testing instances.**

I. Introduction

Scheduling is an important decision-making process in manufacturing and service industries [1]. Scheduling deals with the allocation of resources over time, in order to complete a number of tasks, subject to given constraints and optimising one or more objectives [2]. In job shop scheduling (JSS) problems we are given a set of machines and a set of jobs. A *job* is made up of a sequence of one or more *operations* in a specified processing order. Each operation requires a specified processing time on a certain *machine* [3]. The development of more effective scheduling methods has the potential to dramatically decrease costs and increase throughput, increasing profitability in the many job shops world-wide [4]. This real-world applicability combined with the computational challenge of creating a schedule in such environments has led to job shop scheduling problems being widely studied in the academic literature over the past 50 years [2], [5].

JSS problems can be either static or dynamic. In *static* JSS, all information is known in advance. In *dynamic* JSS, jobs arrive according to a stochastic process and no information is known until their arrival into the shop. The two-machine job shop, although the simplest job shop, provides a test ground for investigating different approaches with lower computational cost than m-machine job shops. Jackson's algorithm [6] is known to solve the static case with makespan objective function in $O(n \log n)$ time, where n is the number of jobs. In the dynamic case there is no optimisation algorithm, but dispatching rules can be evaluated using a discrete-event simulation model of the shop.

A dispatching rule (DR) is a mathematical function of the attributes of queued jobs, machines and the shop. The rule calculates a priority value for each job awaiting processing at a given machine, scheduling the job with the highest priority value next on the machine [7]. It is generally assumed that the routes of jobs through the machines are randomly chosen but each machine is equally likely to be the next machine on a job's route. The shop is also usually *balanced*, with the average operation processing time equal on all machines. Due to this symmetry, the machines have the *same* expected utilisation and the *same* dispatching rule is applied at each machine. However, if the machines do not have the same expected utilisation, this approach may not be valid.

There has been recent research into using Genetic Programming (GP) to evolve DRs for JSS problems [8], [9], [10], [11]. Geiger et al. [7] considered a GP based approach to the static, balanced two-machine *flow shop* (all jobs follow the same path through the machines). Two approaches were suggested: using a single dispatching rule for both machines, and using a separate dispatching rule for each machine. Johnson's algorithm [12] is known to provide an optimal solution. The best evolved individuals were "quite competitive" with Johnson's algorithm. However it is not clear if any of the evolved rules were optimal or even equivalent to Johnson's algorithm. The two-machine job shop is more complicated than the two-machine flow shop, as there are jobs with different paths through the machines, and jobs with only one operation.

Miyashita [13] compared three approaches to JSS: a homogeneous agent model, where every machine uses the same rule evolved by a single GP process; a distinct agent model, where every machine learns a distinct rule; and a mixed agent model, where a bottleneck agent and a non-bottleneck agent learn rules. A bottleneck is a resource with high demand from many operations. A set of benchmark problems with five machines and ten jobs (of five operations each), and one or two bottleneck machines was used. The mixed agent model is limited in that it requires the bottleneck machines to be chosen a priori. Results show that the mixed agent model learnt good heuristics quicker than the distinct agent model, and outperformed both the homogeneous and distinct agent models as well as well-known DRs.

Pickardt et al. [14] developed a two-stage approach for evolving work-centre-specific rules for semi-conductor manufacturing. The method used GP to generate composite dispatching rules and a standard evolutionary algorithm (EA), where each gene is a specific work centre, to search for a good combination of work centre specific rules. The method was shown to be superior to benchmark rules, although only 20 independent runs were completed which is usually not enough for statistical significance testing.

Two fundamental issues with applying GP to m-machine job shops are whether the GP representation is sufficient, and should we evolve dispatching rules specialised to each machine

Rachel Hunt and Mark Johnston are with the School of Mathematics, Statistics and Operations Research, Victoria University of Wellington, Wellington, New Zealand. (email: {Rachel.Hunt, Mark.Johnston}@msor.vuw.ac.nz) Mengjie Zhang is with the School of Engineering and Computer Science, Victoria University of Wellington, Wellington, New Zealand (email: Mengjie.Zhang@ecs.vuw.ac.nz).

978-1-4799-1488-3/14/$31.00 ©2014 IEEE

(previous studies often assume a balanced shop). Therefore to address these issues we consider the fundamental case of the two-machine job shop. In Section III, for the static case, we investigate *if* a GPHH approach can discover optimal dispatching rules. Since Jackson's algorithm solves the static case optimally, success would be a good stamp of validation, certifying the representation and giving confidence in the ability of GP systems to evolve heuristics for more complicated job shop environments. In Section IV, we use a GP approach to evolve scheduling rules for the dynamic two-machine job shop environment where the machines have different mean processing times and expected utilisations. We are interested in whether *separate* DRs evolved by GP simultaneously (one DR specific to each machine) can outperform a single DR (applied to both machines separately).

II. Background

A. Two-machine Job Shop Scheduling Problem

Each job needs to be processed on either one of the machines (A or B) or both of the machines (in sequence A→B or B→A). Let $N_j \in \{1, 2\}$ be the number of operations of job j; $\sigma_{j,i}$ be the ith operation of job j; O_j be the set of operations of job j; $p(\sigma)$ be the processing time of operation σ; $m(\sigma)$ be the machine required to process operation σ; w_j be the weight of job j; d_j be the due date of job j; and r_j be the release date of job j. Then C_j is the completion time of job j and T_j is the tardiness of job j, i.e., $T_j = \max\{0, C_j - d_j\}$.

a) Static case: For the static case, the makespan (maximum completion time), $C_{max} = \max\{C_1, ..., C_N\}$, is to be minimised where all $r_j = 0$. Johnson's algorithm [12] solves the two-machine *flow shop* problem, $F2\,||\,C_{max}$, i.e., every job follows the same route through two machines and the objective is to minimise the makespan. With machines X and Y, job flow X→Y, and job processing times x_j and y_j at machines X and Y respectively, Johnson's algorithm schedules jobs with $x_j \leq y_j$ first in ascending order of x_j, followed by jobs with $x_j > y_j$ in descending order of y_j. Jackson's algorithm [6] solves the two-machine job shop problem. It puts jobs with only one operation on machine A in an arbitrary order, S_A, likewise with jobs with only one operation on machine B, S_B. Jobs to be processed A→B, and B→A are ordered using Johnson's algorithm, to give $S_{A\to B}$ and $S_{B\to A}$ respectively. The final order of jobs at machine A is ($S_{A\to B}$, S_A, $S_{B\to A}$), and at machine B is ($S_{B\to A}$, S_B, $S_{A\to B}$).

b) Dynamic case: In the dynamic case we minimise the total weighted tardiness, $TWT = \sum w_j T_j$. Traditional approaches to JSS in dynamic environments include the following popular dispatching rules. *First In First Out (FIFO)* [15] processes jobs in the order that they arrive at the machine. *Weighted Shortest Processing Time (WSPT)* [15] processes the job that has the maximum $w_j/p(\sigma)$ next. In the *Minimum Slack (MS)* rule the job with the minimum slack is processed next. The minimum slack of job j is defined by $MS_j = d_j - \sum_{l=k}^{N_j} p(\sigma_{j,l}) - t$ where t is the ready time R_m of the machine, i.e., the time that the scheduling decision must be made and the next job dispatched and k is the number of the current operation [15].

B. Genetic Programming based Hyper-heuristics for JSS

Hyper-heuristics can be thought of as heuristics to select or generate heuristics as they search a search space of heuristics rather than directly searching the solution space [16]. Using hyper-heuristics aims to "automate the design and adaptation of heuristic methods in hard computational search problems" [17] and to increase the generality (robustness) of search methods [18].

GP offers advantages as a hyper-heuristic [18] including variable length encoding (trees evolved are between fixed minimum and maximum depths), and executable data structure as the output of a GP run. Humans can also identify the problem domain, search space, and what can be used in the terminal and function sets.

The GPHH approach is ideal for JSS as the aim is to *discover* new dispatching rules, which are naturally represented as GP trees, and can provide insight into the scheduling problem as well as discovering new rules. There have been several applications of GPHH to JSS. Nguyen et al. [8] proposed a multi-objective GPHH method for the automated design of scheduling policies, simultaneously evolving dispatching rules and due-date assignment rules for the job shop environment. Tay and Ho [11] used GP to evolve composite dispatching rules for the multi-objective flexible job-shop problem with the aim of greater scalability and flexibility. Results showed that the evolved rules outperformed single dispatching rules and composite dispatching rules from the literature. Jakobovic and Budin [9] used a GPHH approach to create priority functions for the static and dynamic single machine shop, and the static job shop. Their GP approach used three trees: a decision tree to determine which heuristic should be used at a given time, and two scheduling heuristics which were used dependent on the value returned by the decision tree. Results showed that for given problems the heuristics evolved perform better than existing scheduling methods. Jakobovic and Marasovic [10] used priority functions evolved through GP within a meta-algorithm for a given environment to form a scheduling heuristic. Hildebrant et al. [19] developed dispatching rules for the dynamic 10 machine job shop, with the minimise mean flow time objective. They compared the approach of changing the problem instances used for the simulations at each generation throughout evolution to using the same problem instances at every generation throughout evolution, as well as different numbers of simulations through evolution. Their results suggested that changing the problem instances every generation and using only one simulation gives the best performance.

Most existing work has focused on evolving dispatching rules for scheduling jobs on all the machines in the job shop, rather than on evolving dispatching rules specific to each machine.

III. Static Two-Machine Job Shop

A. Representation of an Optimal DR as a GP-tree

JSS environments have a very large number of job, machine and system properties that can be used in the terminal set, some of which may not be necessary or useful as components of a scheduling rule. The attributes used affect the learning

TABLE I. CONFIGURATIONS OF PROPORTIONS OF JOB TYPES.

	Job Types					Job Types			
	A only	B only	A→B	B→A		A only	B only	A→B	B→A
P1	0.40	0.40	0.15	0.05	P7	0.45	0.05	0.45	0.05
P2	0.50	0.30	0.10	0.10	P8	0.35	0.05	0.55	0.05
P3	0.30	0.50	0.10	0.10	P9	0.25	0.05	0.65	0.05
P4	0.60	0.10	0.25	0.05	P10	0.00	0.00	1.00	0.00
P5	0.50	0.20	0.25	0.05	P11	0.00	0.00	0.10	0.90
P6	0.10	0.60	0.10	0.10	P12	0.10	0.10	0.45	0.35

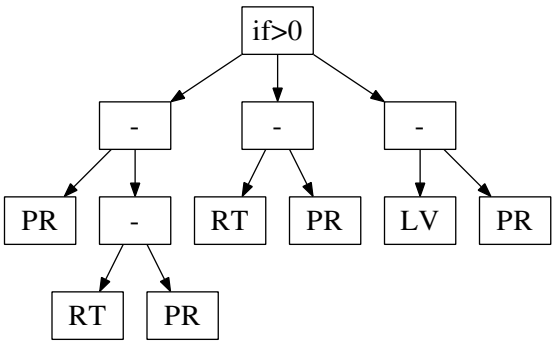

Fig. 1. Optimal dispatching rule for the static two-machine job shop, represented in tree-based GP.

performance and interpretability (how easily understandable the model is by a human).

Whenever a machine finishes processing an operation, it assigns a priority to each job currently waiting to be processed on that machine. In any optimal DR, jobs with only one operation remaining can be scheduled arbitrarily regardless of whether they are, e.g., B only jobs or A→B jobs which have already been completed at machine A, as the order does not alter the total processing time so the makespan is not effected. So any processing order can be simplified to:
Machine A: $(S_{A\to B@A}, S_A \cup S_{B\to A@A})$,
Machine B: $(S_{B\to A@B}, S_B \cup S_{A\to B@B})$.
So jobs with two remaining operations must have a higher priority than any job with only one operation remaining. The following expression, shown in as a tree in Figure 1, is an optimal dispatching rule:

```
(if>0 (- PR (- RT PR))(- RT PR)(- LV PR)),
```

where `PR` is the processing time of the current operation, `RT` is the total processing time remaining for the job and `LV` is some sufficiently large value. The `if>0` function takes three arguments; if the first argument is greater than 0 then it returns the second, else the third is returned.

Let us assume we are scheduling at machine A. In the dispatching rule shown above, if there is only one operation remaining on the job then `(- RT PR)` $= 0$ and `(- PR (- RT PR))` $=$ `PR` > 0. For all such jobs the priority value returned is `(- RT PR)` which is 0. If `(- PR (- RT PR))` > 0 then the processing time of the first operation is longer than the second, the priority given is the processing time of the next operation, `(- RT PR)`. This gives these jobs decreasing priority, ordered by decreasing processing time at the next machine. As `(- RT PR)` > 0 these jobs will have higher priority than the jobs which are on their last operation. Otherwise, if `(- PR (- RT PR))` < 0, the priority given is `(- LV PR)`, i.e. jobs are scheduled by increasing `PR` at this machine. Therefore this dispatching rule creates an ordering which gives the same makespan as Jackson's algorithm, although the ordering of jobs it gives may be different.

B. Can Tree-based GP Evolve Optimal DRs?

Knowing it is possible to create a DR which will always match the performance of Jackson's algorithm, we now want to see how effectively a GPHH approach can evolve such a rule, given all required terminals and functions.

We use a simple scheduling algorithm: while there are operations still to be processed on the machines, when a machine is available we calculate the priorities of all the available operations for that machine (using the GP tree), and schedule the operation with the highest priority, and the job shop system is updated. Only jobs that are currently available can be scheduled on the machines in the shop.

1) GP System: The terminal set is {`PR`, `RT`, `LV`}. The function set is {$+$, $-$, $\times$, $\%$, `if>0`}. The arithmetic operators take two arguments. The first three arithmetic operators, $+$, $-$, $\times$, have their usual meanings. The $\%$ is as usual division except when dividing by zero where the value returned is zero. The initial population is generated using the ramped-half-and-half method [20]. The population size is 1024 and evolution is for 50 generations. Trees have a maximum depth of four, which is deep enough for the DR found above. For the genetic operators crossover, mutation and elitism we use rates of 40%, 55% and 5% respectively. Tournament selection with a tournament size of four is used to select individuals for genetic operators.

2) Training: We randomly create problem instances so that the shop is unbalanced. The proportions of job types follow 12 configurations as in Table I. Each problem instance consists of exactly 10 jobs. The proportions in P1 to P12 are chosen so that there are problem instances where it is important to schedule jobs with two operations ahead of jobs with only one operation, those where it is important to select jobs with longer processing times for their second operation ahead of those where the first operation has longer processing time, etc. The fitness of a DR is then the average makespan from applying the rule to 96 problem instances (8 of each of the 12 configurations). Processing times of operations are generated according to uniform distributions; Uniform(100,110) distribution at machine A, Uniform(200,220) distribution at machine

B, so the shop is unbalanced. Further, the problem instances were changed every 10 generations, i.e., one set of problem instances for generations 1–10, a distinct set for generations 11–20, etc.

3) Testing results: Instead of a traditional testing phase, some simple test cases were used to filter out rules that certainly are not optimal. From 100 GPHH runs, five gave a GP individual which passed all test cases. By manual inspection, all five are optimal. Note that each training problem instance consists of *only* 10 jobs. Although an unusually high mutation rate is used, we want to show that it is *possible* for GP to evolve an optimal DR, not that it is easy for GP to do so.

IV. Dynamic Two-Machine Job Shop

Having validated that GP is sufficiently expressive (in representation) and effective (in search capability) to discover optimal dispatching rules in the static case, we now consider evolving dispatching rules for the dynamic case. Here the key research question is whether, in an unbalanced job shop, dispatching rules should be *specialised* to each machine or whether *one* dispatching rule can be applied to all machines separately.

A. Experimental Design

1) Representations: We investigate two representations for scheduling rules in GP. In representation R1, each GP individual consists of one tree, representing a scheduling rule which is used to schedule the next job at both machines independently.

2) Fitness and Problem Instances: We randomly create problem instances so that the shop is unbalanced. Four different *training* configurations (TC1 to TC4, shown in Table II) of proportions of job types, arrival rate and mean processing times give a variety of mean processing time and utilisation, with machine B having higher utilisation than machine A, and most jobs going to machine A first. Fourteen different configurations (C1 to C14, shown in Table II) are used in *testing*. The fitness of a DR is evaluated by a discrete-event simulation model. In each problem instance, jobs arrive stochastically according to a Poisson process with rate λ. The processing times for machine A and machine B are generated according to an Exponential distribution with mean μ_A and a Uniform$(0.2\mu_B, 1.8\mu_B)$ distribution respectively. We use different processing time distributions to further differentiate between the two machines. The expected utilisation of machine M is calculated as $(\lambda \times p_M)/(1/\mu_M)$, where p_M is the proportion of jobs that need to be processed at machine M.

Due dates are set using Equation (1),

$$d_j = r_j + h \times \sum_{l=1}^{N_j} p(\sigma_{j,l}), \tag{1}$$

where $h = 1.3$ is a due date tightness parameter. Jobs are given weight 1, 2 or 4, with probability $(0.2, 0.6, 0.2)$ [8]. The objective being minimised is TWT. The fitness of an individual is the mean TWT from four simulation runs, one with each of TC1–TC4. A warm up period of 1000 jobs is used, and we collect data from the next 5000 jobs to arrive ($N = 5000$), however new jobs keep arriving in the system until the 6000th job is completed.

We also investigate using the same problem instances at every generation against regenerating the set of training problem instances at every tenth generation, following the work of Hildebrant et al. [19]. In the changing problem instances method, the problem instances used at each generation will still be the same for all GP individuals, but the problem instances for the first 10 generations are distinct from the problem instances for the next ten, and so on.

3) GP System: The terminal set includes `PR` and `RT` as before, plus the remaining number of operations (`RO`), the operation ready time (`RJ`), the job weight (`W`), the job due date (`DD`), the machine ready time (`RM`) and the number of operations in the queue (`NQ`). The function set is $\{+, -, \times, \%$, `if>0`, `max`, `min`, `abs`$\}$. The `abs` function takes one argument and returns the absolute value. The `max` and `min` functions take two arguments and return the maximum and minimum of their arguments respectively. The initial population is generated using the ramped-half-and-half method. The population size is 1024 and evolution is for 50 generations. GP trees have a maximum depth of six. For the genetic operators crossover, mutation and elitism we use rates of 85%, 10% and 5% respectively. Tournament selection with a tournament size of four is used to select individuals for genetic operators.

B. Experimental Results

Here we present the test results of GPHH with each combination of representations R1 and R2, and the same problem instances at every generation (S) versus changing problem instances every ten generations (C), for the dynamic two-machine JSS problem. For each we perform 40 GP evolutionary runs, using 40 common seeds.

The boxplots in Figure 2 show that the relative performance of the four GPHH methods depend on the configuration. Table IV gives the order of performance, from worst to best, of the four GPHH methods for each of the 14 configurations. The orders are established by performing pairwise comparisons between GP runs with the same initial population of GP individuals for each combination of methods used. We examined the ratio of the two rules over these combinations for each of the 40 initial populations to see if rules evolved by one method were consistently better than another. Table IV shows that although there is no consistent order, although there are some patterns that can be observed. Over the four configurations with largest imbalance in expected utilisation (C2, C6, C10, C12) the C methods outperform the S methods. Interestingly this is also the case for C13, which is symmetric and therefore a balanced shop. For the other configurations where the expected utilisations are within 0.02 and the shop is not symmetric, the S methods outperform the C methods. When the performance of two methods is similar, the C methods are inseparable, and the S methods are inseparable. However we cannot say conclusively that any method is better than any other.

Table III gives the mean performance (TWT) of three benchmark dispatching rules (WSPT, FIFO, MS) across the 14 test configurations, and the mean $\pm$ standard deviation of the average performance of each of the 40 GP runs for R1S, R1C, R2S and R2C. The standard deviation of performance is much smaller for the R2 methods both with the same and changing problem instances.

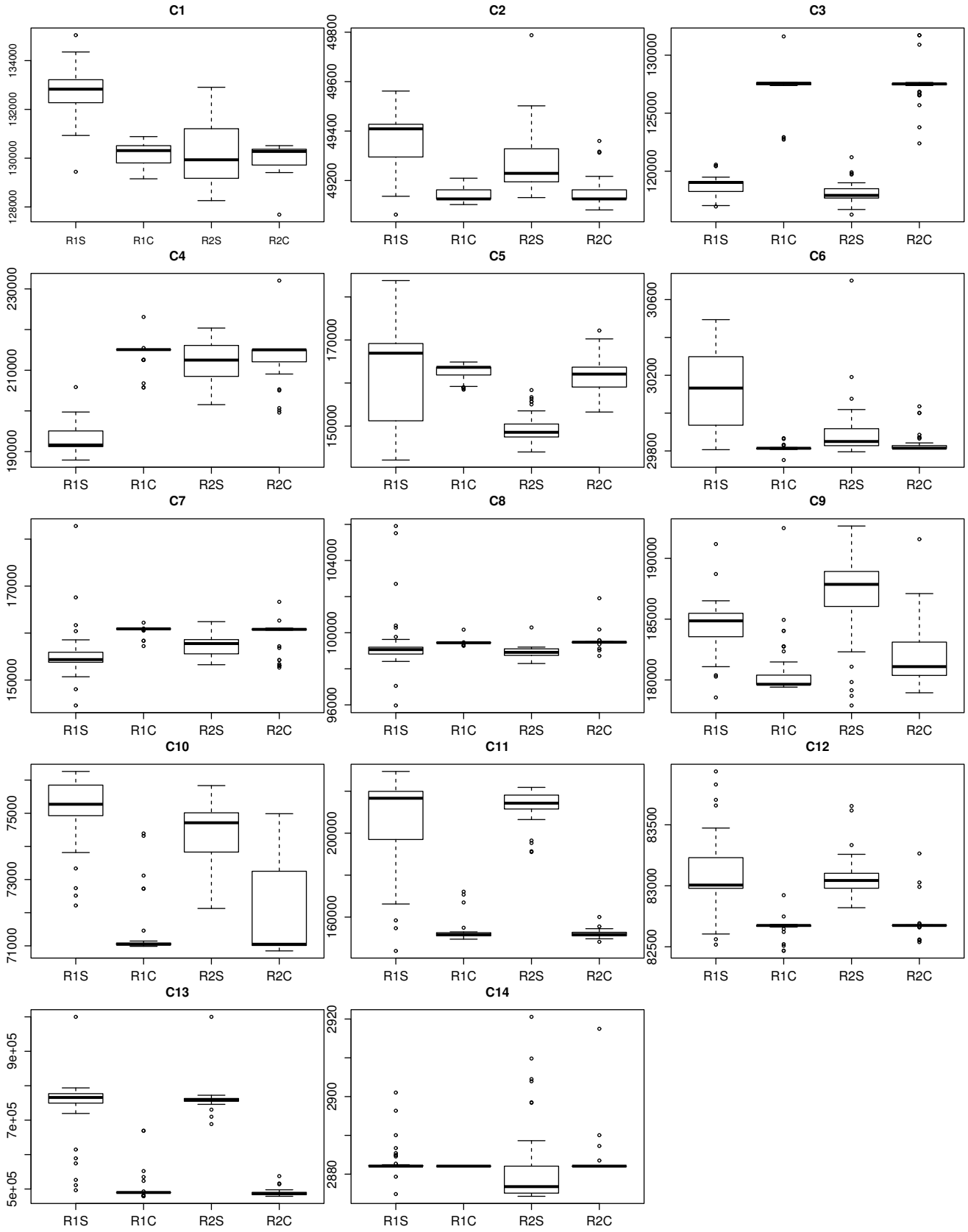

Fig. 2. Boxplots showing results (TWT) of methods R1S, R1C, R2S and R2C across the 14 configurations.

TABLE II. CONFIGURATIONS OF PROPORTIONS OF JOB TYPES FOR TRAINING (TC1–TC4) AND TESTING (C1–C14).

				Job Types				Utilisation	
	λ	$1/\mu_A$	$1/\mu_B$	A only	B only	A→B	B→A	Machine A	Machine B
TC1=C1	0.80	0.80	0.50	0.40	0.05	0.50	0.05	0.95	0.96
TC2=C2	0.80	1.20	0.60	0.30	0.05	0.60	0.05	0.63	0.93
TC3=C3	0.70	0.65	0.40	0.45	0.10	0.30	0.15	0.97	0.96
TC4=C4	0.65	0.60	0.20	0.70	0.10	0.10	0.10	0.98	0.98
C5	0.80	0.50	0.80	0.05	0.40	0.05	0.50	0.96	0.95
C6	0.80	0.60	1.20	0.05	0.30	0.05	0.60	0.93	0.63
C7	0.70	0.40	0.65	0.10	0.45	0.15	0.30	0.96	0.97
C8	0.65	0.20	0.60	0.10	0.70	0.10	0.10	0.98	0.98
C9	0.70	0.60	0.40	0.45	0.15	0.20	0.20	0.99	0.96
C10	0.70	0.60	0.50	0.45	0.15	0.20	0.20	0.99	0.77
C11	0.90	0.80	0.50	0.45	0.15	0.20	0.20	0.96	0.99
C12	0.70	1.00	0.50	0.30	0.10	0.30	0.30	0.63	0.98
C13	0.40	0.30	0.30	0.25	0.25	0.25	0.25	1.00	1.00
C14	0.10	0.70	0.40	0.10	0.20	0.35	0.35	0.11	0.23

TABLE III. COMPARISON OF MEAN PERFORMANCE, MEAN±STANDARD DEVIATION EVOLUTION AND TESTING TIMES OF THE FOUR GPHH METHODS OVER 40 RUNS AND THREE BENCHMARK DISPATCHING RULES.

	R1S	R1C	R2S	R2C	WSPT	FIFO	MS
TWT	159560±8201	140335±3188	161937±3117	139372±1216	382778	554842	534326
Train (min)	117±31	118±27	80±16	90±21	–	–	–
Test (ms)	613±349	469±153	500±245	392±103	250.0	290.0	202.0

TABLE IV. ORDER OF PERFORMANCE OF METHODS ON EACH CONFIGURATION.

Configuration	Worst → Best
TC1=C1	R1S → {R1C,R2S, R2C}
TC2=C2	{R1S,R2S} → {R1C, R2C}
TC3=C3	{R1C, R2C} → {R1S,R2S}
TC4=C4	{R1C,R2S, R2C} → R1S
C5	{R1C, R2C} → R1S →R2S
C6	{R1S,R2S} → R2C → R1C
C7	{R1C, R2C} →R2S → R1S
C8	R2C → R1C →R2S → R1S
C9	R2S → R1S → {R1C, R2C}
C10	{R1S,R2S} → {R1C, R2C}
C11	{R1S,R2S} → {R1C, R2C}
C12	{R1S,R2S} → {R1C, R2C}
C13	{R1S,R2S} → {R1C, R2C}
C14	R2C → R1C → R1S →R2S

Table III also gives the mean and standard deviation of the evolution time in minutes and the total testing time (for all 14 test configurations) in milliseconds. The R2 methods have the lowest mean evolution time and smaller standard deviations than the R1 methods. The testing time is lower with changing problem instances than using the same problem instances and the standard deviation is also smaller. Further R2S has lower mean and smaller standard deviation for testing than R1S, likewise R2C has lower mean and smaller standard deviation than R1S. R2C has the quickest mean test time (approximately 0.4 seconds) and the smallest standard deviation.

Comparison of the GPHH and dispatching rule (WSPT, FIFO, MS) results in the table shows that GPHH is able to evolve rules that are significantly better than these rules.

Due to the way which the GP crossover operator is restricted in R2 so crossover occurs only between trees that schedule jobs on the same machine, only one of the GP individual's trees is affected by crossover generation. This means that the R2 GP runs should be run for twice as many generations as R1 GP runs for the trees to have the same number of crossover operations [20].

C. Analysis of Evolved Programs

To analyse why the dispatching rules work well, we choose some of the best evolved rules to look at more closely.

The two GP individuals from method R1C in Figure 3 performed identically. The right-hand individual simplifies to

$$\frac{\texttt{W} \times \texttt{RJ}}{\texttt{PR} \times \texttt{RM}}$$

As RM will be the same for all jobs in the queue, this can be further simplified to (W/PR) × RJ. This is very similar to the WSPT rule weighted by the ready time of the operation. Jobs with earlier ready times (smaller RJ) have lower priority, all else equal. Interestingly the due date does not appear in this individual, although DD is related to RJ, since both increase throughout simulation time. The left-hand individual of Figure 3 will generally simplify to be the same as the right-hand individual as in general DD>W, and RM>W∗W/PR. It is interesting to note that this tree does not contain the DD

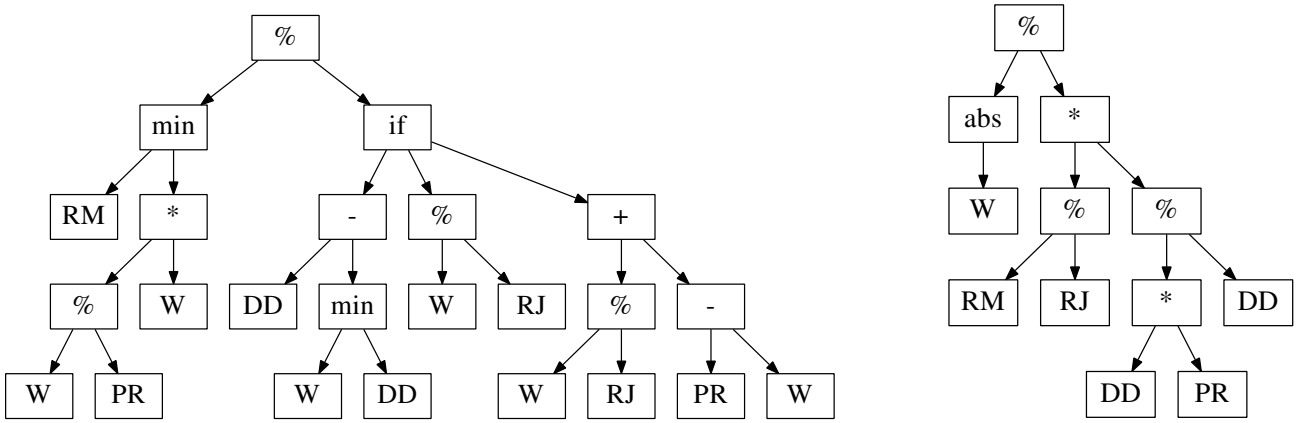

Fig. 3. Best equal evolved R1C individuals.

terminal, which we would expect to appear as it is used in calculating the tardiness of jobs for the objective function.

A similar structure is found in the machine B tree of the R2C individual shown in Figure 5. Once a few jobs have been processed, the ready time of the machine will be greater than the weights of the job (under our settings at least), then the third branch of the `if>0` branch will be scheduling the jobs. As all job weights are positive this branch simplifies to $(\mathtt{W}/\mathtt{PR}) \times \mathtt{DD}$, as the number of jobs in the queue will be the same for all jobs evaluated at a given time. Using this rule to schedule *all* jobs in the shop at both machines gives an average TWT of 132229 across the 14 configurations. The machine A tree of this GP individual (see Figure 4) is harder to analyse. The middle branch will always be used, and simplifies to

$$\mathtt{W} + \mathtt{DD} - \mathtt{RT} - \max\{\mathtt{RO}, (\mathtt{W}/\mathtt{PR}) \times (\mathtt{RM} + \mathtt{RJ})\}.$$

This rule takes the latest time processing of a job would need to start for the job to not be tardy, adds the weight (jobs with greater weight therefore have higher priority) and subtracts a measure of waiting and processing time. Using this rule to schedule *all* jobs in the shop at both machines gives an average TWT of 2423119 across the 14 configurations, which is considerably worse than the performance of the machine B tree.

Further, all terminals and functions appear in at least one of these "best" evolved trees, however `NQ` does not contribute the priority value in the trees it appears in.

V. Conclusions

The goal of this paper was to use a GPHH approach to evolve scheduling rules for both the static and dynamic two-machine job shop environments. This work is the first time that GP has been used to evolve dispatching rules that are optimal for the static two-machine JSS problem with makespan objective function. This was proved by showing that the evolved rule was equivalent to optimal scheduling algorithm, Jackson's algorithm. Our GPHH search found five rules out of 100 that are optimal, i.e., will always schedule jobs to give the minimum possible makespan. This validates both the GPHH approach for generating dispatching rules for the JSS problem, and the representation used.

In the dynamic case our work investigated the combined effects of changing the problem instances throughout evolution, and using one scheduling rule versus machine specific scheduling rules in non-symmetric job shops. Our results in the dynamic case show that R1 methods do not consistently outperform R2 methods in terms of TWT, neither do the C methods outperform the S methods. We have found the mean performance of R2 methods have a smaller standard deviation than their R1 counterparts. As we cannot separate which methods are best, we cannot assume that one rule for all machines is sufficient. However when we consider runtimes, the R2 methods have shorter mean evolution times with smaller standard deviations than their R1 counterparts. C methods have longer mean evolution times but smaller standard deviations than the S methods. Further the R2 methods have shorter mean testing times with smaller standard deviations than their R1 counterparts and C methods have shorter mean evolution times and smaller standard deviations than the S methods.

In future work we plan to extend our job shop environment to three or more machines. We will also incorporate other measures of the current state of job queues and the shop system into the scheduling rules and investigate the use of strongly typed GP to improve interpretability.

References

[1] M. Pinedo, *Scheduling: Theory, Algorithms, and Systems*, 3rd ed. Springer, 2008.

[2] E. Hart, P. Ross, and D. Corne, "Evolutionary scheduling: a review," *Genetic Programming and Evolvable Machines*, vol. 6, pp. 191–220, 2005.

[3] J. Blazewicz, W. Domschke, and E. Pesch, "The job shop scheduling problem: Conventional and new solution techniques," *European Journal of Operational Research*, vol. 93, no. 1, pp. 1–33, 1996.

[4] A. Jones and L. C. Rabelo, "Survey of job shop scheduling techniques," National Institute of Standards and Technology, Gaithersberg, Tech. Rep., 1998.

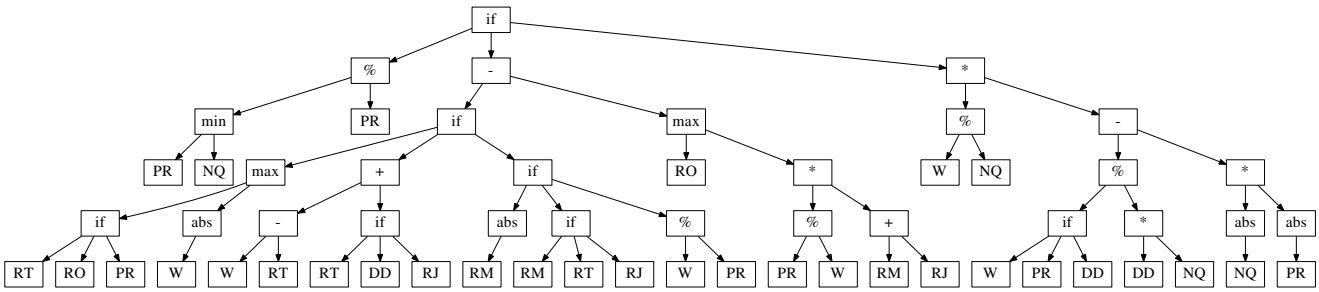

Fig. 4. Best evolved R2C individual - Machine A.

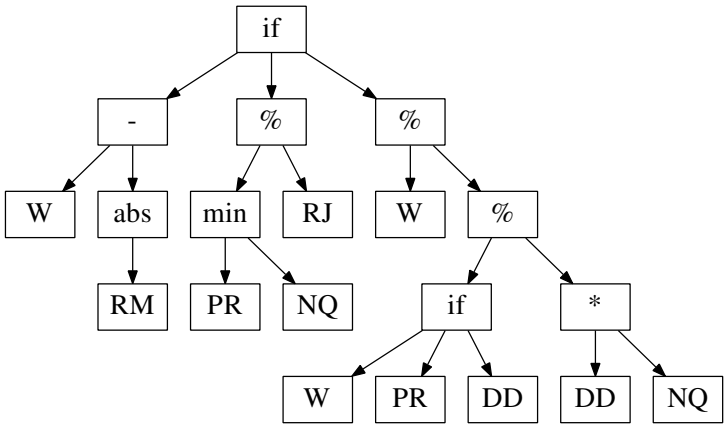

Fig. 5. Best evolved R2C individual - Machine B.

[5] C. Potts and V. Strusevich, "Fifty years of scheduling: a survey of milestones," *Journal of the Operational Research Society*, vol. 60, no. S1, pp. 41–68, 2009.

[6] J. Jackson, "An extension of Johnson's result on job-lot scheduling," *Naval Research Logistics Quarterly*, vol. 3(3), pp. 201–204, 1956.

[7] C. Geiger, R. Uzsoy, and H. Aytug, "Rapid modeling and discovery of priority dispatching rules: An autonomous learning approach," *Journal of Scheduling*, vol. 9, no. 1, pp. 7–34, 2006.

[8] S. Nguyen, M. Zhang, M. Johnston, and K. C. Tan, "A coevolution genetic programming method to evolve scheduling policies for dynamic multi-objective job shop scheduling problems," in *Proceedings of the IEEE Congress on Evolutionary Computation*, 2012, pp. 3332–3339.

[9] D. Jakobović and L. Budin, "Dynamic scheduling with genetic programming," in *Proceedings of the European Conference on Genetic Programming*, 2006, pp. 73–84.

[10] D. Jakobović and K. Marasović, "Evolving priority scheduling heuristics with genetic programming," *Applied Soft Computing*, vol. 12, no. 9, pp. 2781–2789, 2012.

[11] J. C. Tay and N. B. Ho, "Evolving dispatching rules using genetic programming for solving multi-objective flexible job-shop problems," *Computer & Industrial Engineering*, vol. 54, pp. 453–473, 2008.

[12] S. Johnson, "Optimal two- and three-stage production schedules with setup times included," *Naval Research Logistics Quarterly*, vol. 1, pp. 61–68, 1954.

[13] K. Miyashita, "Job-shop scheduling with GP," in *Proceedings of the Genetic and Evolutionary Computation Conference*, 2000, pp. 505–512.

[14] C. Pickardt, T. Hildebrandt, J. Branke, J. Heger, and B. Scholz-Reiter, "Evolutionary generation of dispatching rule sets for complex dynamic scheduling problems," *International Journal of Production Economics*, vol. 145, no. 1, pp. 67–77, 2013.

[15] A. Vepsalainen and T. Morton, "Priority rules for job shops with weighted tardiness costs," *Management Science*, vol. 33, pp. 1035–1047, 1987.

[16] P. Ross, "Hyper-heuristics," in *Search Methodologies: Introductory Tutorials in Optimization and Decision Support Techniques*, E. Burke and G. Kendall, Eds. Springer, 2003, vol. 1, pp. 529–556.

[17] E. Burke, M. Hyde, G. Kendall, G. Ochoa, E. Ozcan, and J. R. Woodward, "A classification of hyper-heuristic approaches," in *Handbook of Meta-Heuristics*, M. Gendreau and J.-Y. Potvin, Eds. Kluwer, 2010, pp. 449–468.

[18] E. K. Burke, M. R. Hyde, G. Kendall, G. Ochoa, E. Ozcan, and J. R. Woodward, "Exploring hyper-heuristic methodologies with genetic programming," in *Computational Intelligence*, ser. Intelligent Systems Reference Library, C. Mumford and L. Jain, Eds. Springer Berlin Heidelberg, 2009, vol. 1, pp. 177–201.

[19] T. Hildebrandt, J. Heger, and B. Scholz-Reiter, "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, pp. 257–264.

[20] J. R. Koza, *Genetic Programming: On the Programming of Computers by Means of Natural Selection*. Cambridge, MA, USA: MIT Press, 1992.