A Hyper-Heuristic Ensemble Method for Static Job-shop Scheduling

Emma Hart e.hart@napier.ac.uk
Institute for Informatics and Digital Innovation, Edinburgh Napier University, Edinburgh, EH10, UK

Kevin Sim k.sim@napier.ac.uk
Institute for Informatics and Digital Innovation, Edinburgh Napier University, Edinburgh, EH10, UK

Abstract

We describe a new hyper-heuristic method *NELLI-GP* for solving job-shop scheduling problems (JSSP) that evolves an ensemble of heuristics. The ensemble adopts a divide-and-conquer approach in which each heuristic solves a unique subset of the instance set considered. *NELLI-GP* extends an existing ensemble method called NELLI by introducing a novel heuristic generator that evolves heuristics composed of linear sequences of dispatching rules: each rule is represented using a tree structure and is itself evolved. Following a training period, the ensemble is shown to outperform both existing dispatching rules and a standard genetic programming algorithm on a large set of new test instances. In addition, it obtains superior results on a set of 210 benchmark problems from the literature when compared to two state-of-the-art hyper-heuristic approaches. Further analysis of the relationship between heuristics in the evolved ensemble and the instances each solves provides new insights into features that might describe similar instances.

Keywords

Job-shop-scheduling, dispatching rule, heuristic ensemble, hyper-heuristic, genetic programming

1 Introduction

The Job Shop Scheduling problem (JSSP) is one of the most researched combinatorial problems studied by practitioners over recent decades, due to its relevance in many industrial applications. The simplest form is known as the *static* JSSP: a number of operations that need to be scheduled across multiple machines, with all operations available at the start. Dispatching rules that derive a priority index for each operation to be scheduled provide a quick and simple method for creating a schedule (Baker, 1984), and hence for practical reasons are commonly used in real-world applications. Many dispatching rules are described in the literature, designed by hand by human-experts, although according to Geiger et al. (2006), often through considerable effort. In an effort to automate this, hyper-heuristic methods have been used to create novel, reusable dispatching rules. A comprehensive recent survey is provided by Branke et al.

(2015) who summarize the state-of-the-art, noting that the most common approach has been to use variable-length grammar-based methods (e.g. genetic programming) to generate new dispatching rules. While this has achieved much success, they highlight a number of open questions and challenges, the first of which is the need to deal with complex scenarios by evolving *ensembles* of heuristics, rather than a single rule.

Currently, a few examples of small ensembles that evolve *pairs* of heuristics exist, e.g. (Miyashita, 2000; Nguyen et al., 2014a; Park et al., 2013). Most recently, Park et al. (2015) evolve a larger ensemble of rules in which each heuristic in the ensemble votes to determine the priority of an operation at each iteration, using a similar methodology to ensemble approaches to classification typically seen in machine learning (Valentini and Masulli, 2002). From an optimisation perspective, Smith-Miles et al. (2014) point out that any given algorithm will have individual weaknesses on a given set of instances. Ensemble methods in which a mixture of algorithms is used to solve a set of instances would therefore appear a fruitful line of research. We propose an ensemble approach in which a set of heuristics is evolved using GP, each of which generates a complete solution to a subset of instances in a distinct region of the instance space (i.e. the set of problem instances of interest). This can be be loosely described as 'mixture-of-experts' model to borrow terminology from the machine learning literature. It is also similar to the *algorithm portfolio* methods that are common in Operations Research, particularly for solving SAT problems, e.g. (Leyton-Brown et al., 2003). However, in contrast to many portfolio methods that focus mainly on selection from a portfolio, our approach addresses portfolio *composition*, ensuring that the portfolio contains a behaviourally diverse range of algorithms.

Our approach extends a previously described ensemble method called NELLI, that has been extensively tested in the bin-packing domain (Sim et al., 2015) and on a small set of 62 JSSP instances (Sim and Hart, 2014). NELLI-GP extends NELLI by evolving a pool of novel dispatching rules, each represented as an expression tree. The novel rules are combined into variable length sequences called *heuristics*, which are themselves subject to evolution. Using a large set of static JSSP instances, we address two main questions using NELLI-GP:

- Do evolved *heuristics*, composed of variable length *sequences* of evolved dispatching rules, outperform the individual evolved rules?
- Do *ensembles* of evolved heuristics outperform individual heuristics?

Our approach extends existing work on tree-based rule evolution of single dispatching-rules by considering an extended node set and multiple methods for selection of eligible jobs. It extends current approaches to *ensemble* methods in that the role of each member of the ensemble is to solve a subset of *instances*, rather than voting to prioritise individual *operations*. Evaluation on over 700 problem instances shows significant improvement with respect to existing dispatching rules, and to state-of-the approaches (Park et al., 2015; Nguyen et al., 2013b), improving on published results on benchmark sets by up to 16%. Finally, analysis of the ensemble in terms of mapping heuristics to instances provides new insights into the relationship between simple instance parameters and algorithmic performance.

The paper proceeds with a brief introduction to JSSP and to previous approaches for automating the production of dispatching rules. In section 4 we provide both a

conceptual and algorithmic description of the approach, clarifying the modifications that were made to NELLI and describing the new problem generator. After providing detailed experimental results, the paper concludes with an analysis section that provides some insight into the function of the individual elements of the ensembles, the diversity of heuristics, and the difficulty of the problems in the instance space.

2 Background

Using the $\alpha|\beta|\gamma$ notation (Graham et al., 1979) the two JSSP problems investigated are denoted as $Jm||C_{max}$ and $Jm||\sum \omega_i T_i$ where the term J_m translates as a Job Shop environment and the last term corresponds to the objective (Makespan (C_{max}) and Summed Weighted Tardiness (SWT) respectively). An n xm JSSP problem has a set of n jobs, $\{J_i\}_{1\leq j\leq n}$ and m machines, $\{M_k\}_{1\leq k\leq m}$. Each job J_i contains m operations, each of which has processing time p_{jk} and weight given by w_i. Every job is processed on every machine with no recursion allowed. Each job has a release time r_i which imposes a hard constraint on the earliest start time of a job and a due time of d_i which imposes a soft constraint on the time that the job should be completed by. Machines process a single operation at a time and all machines are available for the duration of the schedule.

2.1 Dispatching Rules

Many simple dispatching rules have been developed to quickly prioritise operations in order to select an appropriate operation for scheduling in an iterative process (see Panwalkar and Iskander (1977); Haupt (1989); Blackstone et al. (1982)). Typically, rules are hand-designed by experts in a trial and error-process (Geiger et al., 2006), and each one optimises one or a limited number of scheduling objectives based on some particular conditions. *Composite* DRs combines single dispatching rules into a new rule and have been shown to outperform single DRs (Nguyen et al., 2013a). Both types of DR typically fall into one of two categories: DRs that can be evaluated before a schedule is commenced (e.g. Job Arrival Date and Operation Processing Time) or DRs that return different priorities for operations as the schedule is built (e.g. Number of Remaining Operations).

Many simple dispatching rules have been known for decades — a comprehensive survey in Panwalkar and Iskander (1977) describes over one hundred. Some of the earliest hyper-heuristic approaches to scheduling (though the term was not in use at the time) recognised improved performance could be achieved by combining existing dispatching rules into sequences that could be applied iteratively to build a solution. For instance, Ulrich and Erwin (1995) and Hart and Ross (1998) both use evolutionary algorithms to search for promising sequences of dispatching rules, showing promising results compared to the single rules.

In addition to specifying priority, a dispatching rule needs to also specify the eligible job set, i.e., the subset of jobs that should be prioritised by the rule. *Non-delay* selection only considers jobs that are currently waiting to be scheduled, and hence minimises idle-time. In the majority of hyper-heuristics that generate dispatching rules, eligibility is limited to waiting jobs, i.e. follows the non-delay method. Non-delay schedules are not guaranteed to be optimal; the Giffler-Thompson algorithm (Giffler and Thompson, 1960) includes in the list of eligible jobs those arriving in the near future, before the shortest operation of waiting jobs can be completed. Hyper-heuristic meth-

ods that adopt this approach (or variants of it) include (Geiger et al., 2006; Miyashita, 2000; Pickardt et al., 2013). Rather than pre-defining eligibility, the hyper-heuristic itself can optimise the method by which eligible jobs are selected. Hart and Ross (1998) evolved an extra parameter for each rule that defined the method that should be used to generate the conflict set of operations; Nguyen et al. (2013a) propose a representation for evolving scheduling rules that incorporates a hybrid scheduling strategy between non-delay and active scheduling. InNguyen et al. (2013b), a separate function for the non-delay factor is evolved which can then adapt to changing shop conditions.

2.2 Evolving Novel Dispatching Rules

Despite the large number of existing dispatching rules and proposed methods for combining them, interactions between scheduling rules can be complex; in order to address this, more recent research has attempted to automate the process of designing rules, through the use of Genetic Programming to evolve arithmetic expressions of rules. This is a *hyper-heuristic* approach in that the space of heuristics (dispatching rules) is searched, rather than the space of potential solutions (schedules). A very comprehensive survey of hyper-heuristic approaches to automated design of dispatching rules is provided in (Branke et al., 2015).

Most hyper-heuristic methods evolve a single dispatching rule. For example, Tay and Ho (2008) successfully evolved rules that determined the priority of operations to be scheduled for static problems. This work was later shown to be less useful in dynamic cases. Hildebrandt et al. (2010) advanced this work by using four varied training scenarios and found effective but complex rules. Noting that most rule-based approaches lack a global-perspective in that they only consider the current state of a machine and its queue.

Nguyen et al. (2013b) evolve iterated dispatching rules that iteratively improve the schedules by utilising the information from completed schedules; in (Nguyen et al., 2013a), they extend the terminal set available to GP to include both recorded information regarding a previous schedule,'look-ahead' information and composite DRs, finding good results on static JSSP problems. Hunt et al. (2014) extend this approach of using 'less-myopic' information, evolving new rules with an extended terminal set that were tested on a set of randomly created test scenarios. Their results show evolution of better DRs in terms of total weighted tardiness across the test simulation scenarios with high utilisation. In relation to dynamic JSSP problems, Nguyen et al. (2014b) compare surrogate-assisted selection methods in conjunction with GP, and automatic programming via iterated local search in (Nguyen et al., 2015). Note that in addition to trees, other representations of dispatching-rules are possible, for example using neural networks to represent a function or simple linear combinations of attributes. However a recent paper by Branke et al. (2014) evaluated potential representations and concluded that in terms of solution quality then the expression tree representation was preferable.

The GP systems described above evolve a *single* rule that should generalise across a range of benchmarks. Another strand of research utilises GP to evolve multiple rules that are used collaboratively to solve problems. Miyashita (2000) develop a system called GP-3 that evolves two dispatching rules and a choice function that selects between the two rules depending on whether a machine creates a bottleneck; this shows improvement over the use of a single evolved rule. Nguyen et al. (2014a) consider two approaches to evolving rules for dynamic JSSPs. In the first approach, a single individ-

ual is evolved that contains two separate subtrees that together specify a scheduling policy. In their second approach, a cooperative co-evolution method inspired by Potter and De Jong (2000) is used to evolve trees in two separate sub-populations: trees from each sub-population are combined to specify a complete policy, and three objectives are used to select individuals based on dominance criteria.

2.3 Collectives of Rules

GP has previously been used to evolve ensembles of rules in the classification domain, e.g. (Downey et al., 2012; Bhowan et al., 2013, 2014). The evolved ensembles promote behavioural diversity amongst the evolved classifiers, and have proved promising when applied to classifying unbalanced data-sets. Each evolved classifier votes on the class of a given data-record, with the majority vote recorded. In the hyper-heuristic JSSP domain, at the time of writing we are aware of only one ensemble approach. Park et al. (2015) extend the cooperative co-evolution method first proposed by Nguyen et al. (2014a) to evolve s sub-populations of trees; a set consisting of one tree from each sub-population is formed and each member of the subset votes as to which operation should be selected for scheduling. The operation with the most votes is then scheduled. The approach shows promising results but was limited to testing on 80 problems.

In contrast to the above ensemble method in which members of the ensemble *each* votes as to which operation should be scheduled, we have previously described a system called NELLI that evolves an ensemble in which each member of the ensemble 'wins' a unique subset of the instances under consideration (Sim et al., 2015, 2013; Sim and Hart, 2014). NELLI uses a method inspired by Artificial Immune Systems (AIS) to evolve a set of heuristics, which are behaviourally diverse in the sense that each solves different subsets of a large instance set. This approach bears considerable resemblance to *bagging* (bootstrap aggregating) approaches from the machine learning literature (Breiman (1996)) which have been shown to improve the stability and accuracy of machine learning algorithms, by reducing variance and avoiding overfitting. The approach can also be considered as *mixture of experts* methods (Valentini and Masulli (2002)): in these methods a supervising learner divides the input space and assigns an appropriate element of the ensemble to each division. While *NELLI-GP* retains the core AIS element of NELLI which ensures diverse heuristics are retained, it replaces the component that evolves the heuristics, using GP to evolve new dispatching rules which are sequenced into new heuristics.

3 Method

In NELLI-GP, an ensemble $\mathcal{E}$ is defined by a tuple $\langle H, R, D \rangle$. H specifies the maximum number of *heuristics* in the ensemble, each of which is composed of a sequence of *rules*. The maximum length of the rule sequence is denoted by R. Each rule is defined by a tree that can have maximum depth D. Thus, an ensemble signified by the expression $H5 - R3 - D10$ for example describes an ensemble composed of a maximum of 5 heuristics, each of which is composed of a sequence of rules of maximum length 3, and where each rule has maximum depth 10. The creation of trees, rules, and heuristics that forms a key contribution of this paper are described in turn below.

3.1 Tree-based Dispatching Rules

In contrast to previous work in which NELLI used dispatching rules taken directly from the literature (Sim and Hart, 2014), in NELLI-GP we formulate novel dispatching rules as trees of depth d, where $0 \leq d \leq d_{max}$. The tree returns a real value for each operation that can be considered for scheduling — these values are subsequently used to prioritise each operation. Trees are formulated using a set of terminal and function nodes described below.

3.1.1 Terminals & Function Nodes

We use a large set of function nodes as defined in table 2. The terminal node set contains 28 dispatching rules, compared to 14 in Hunt et al. (2014) and 7 in Tay and Ho (2008). The majority of these have been obtained from the literature and includes examples of both simple and composite dispatching rules and examples from each of the three classes defined in section 2.1 — *static, dynamic, less-myopic*. Additional terminals have been added to increase diversity.

Note that many of the acronyms used here differ from those commonly used in the literature due to the fact that the set of priorities returned by a rule can be ordered either *ascending* or *descending*. For example the Shortest Processing Time rule (SPT) named OpPT here can sort operations from smallest to largest or from largest to smallest, depending on whether the terminal is set as positive or negative. Terminals prefixed with Op, J, or M denote that they operate on the operation, its associated job or its allocated machine. For example for each operation to be prioritised, MTPT returns the total processing time of the operations to be scheduled on the associated machine and JTPT returns the total processing time of the job.

The list includes simple DRs such as Weighted Shortest Processing Time (WSPT), Earliest Due Date (EDD) and Minimum Slack (MS) (named JWPT, JDD and JS respectively). These DRs have been shown to provide optimal solutions for the $1||\sum \omega_j T_j$ single machine problem when release dates and due dates are zero (WSPT), or when they are sufficiently spread out (EDD or MS). The terminal set also includes composite DRs from the literature such as Apparent Tardiness Cost (JATC) which encapsulates features of both WSPT and MS. JATC has been shown to outperform its component parts in Vepsalainen and Morton (1987) for the SWT objective. Some less myopic rules such as Next Operation Processing Time (OpNPT) are also implemented.

When a terminal node is added to a tree, it is randomly assigned as a *positive* or *negative* node, i.e. a rule can order its list of operations from largest to smallest or vice-versa. Thus in effect, the size of the terminal set is doubled.

3.1.2 Rule Wrappers: Eligible Operations

Each newly generated tree is encapsulated in a wrapper named a rule. A rule is randomly assigned a label A, B, C specifying the algorithm that should be used to generate the set $S*$ of operations that are eligible for scheduling, thus expanding the search-space of possible rules. Algorithm A is a modified version of the Giffler and Thompson algorithm (Giffler and Thompson, 1960) described in Algorithm 1; Algorithm B is the Non-Delay algorithm (Baker, 1984) described in Algorithm 2; Algorithm C is the set of all available waiting operations.

Table 1: Terminals: Dispatching Rules

Terminal Nodes	
Acronym	**Description**
JAPT	Average Job Processing Time
JATC	Job Apparent Tardiness Cost **var** = 0.35
JDD	Job Due Date
JEγD	From Arkin 1991 $\gamma = 1$
JPTSF	Job Processing Time So Far
JRPTI	Job Remaining Processing Time (including this op)
JRPT	Job Remaining Processing Time (NOT including this op)
JS	Job Slack Time
JTPT	Job Total Processing Time
JWPT	Job Weighted Processing Time
JW	Job Weight
JAD	Job Arrival Date
MIT	machine idle time (so far)
MLFT	Machine Last Op Finish Time
MPTRI	Machine Processing Time Remaining (inclusive of the operation)

Terminal Nodes	
Mnemonic	**Description**
MPTR	Machine Processing Time Remaining (not including the operation)
MPTSF	Machine Processing Time So Far (not including idle)
MTPT	Machine Total Processing Time (sum of all operations to be scheduled)
OpEST	Operation Earliest Schedule Time
OpFCFS	Operation First Come First served
OpNPT	Next operation in the job's processing time (or 0 if last op in job)
OpNRO	Number of Remaining Operations
OpPPT	Previous operation from the job's processing time (or 0 if first op in job)
OpPT	Operation Processing Time
OpWPT	Weighted Processing Time
OpRT	Random Operation
INT	Random Integer Value
REAL	Random double between 0 and 1

Table 2: Function Nodes

Function Set		
Mnemonic	**#Ops**	**Returns**
ABS	1	Absolute Value of Op
+	2	Sum of Op 1 and Op 2
-	2	Difference between Op 1 and Op 2
X	2	Product of Op 1 and Op 2
/	2	Protected Division of Op 1 by Op 2 *
NEG	1	Negation of Op 1
IGTZ	3	If Op 1 $\geq = 0$; Op 2 else Op 3
Exp	1	The Exponential of Op 1
MAX	2	The maximum value of Op 1 and Op 2
Comp	2	If Op 1 $<$ Op 2 returns -1 If Equal returns 0: Else returns 1

* If $Op\ 2 = 0$, ∞ is returned

Algorithm 1 Giffler & Thomson

1: Calculate the set C of all operations that can be scheduled next
2: Calculate the completion time of all operations in C, and let m* equal the machine(s) on which the minimum completion time t is achieved.
3: Let $S*$ denote the conflict set of operations on machine(s) m* - this is the set of operations in C which take place on m*, and whose start time is less than t.

Algorithm 2 Non Delay

1: Calculate the set C of all operations that can be scheduled next
2: Calculate the starting time of each operation in C and let $S*$ equal the subset of operations that can start earliest

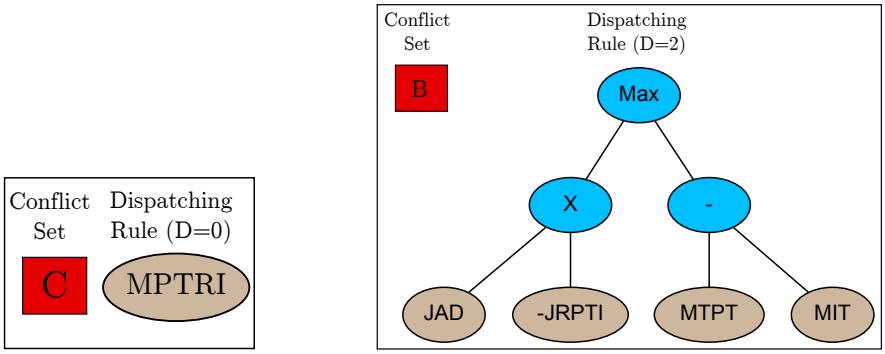


Figure 1: Example rules comprising a conflict set and a tree dispatching rule of depth 0 (left) and 2 (right). Terminal nodes prefixed with **-** return a negated value

The first two algorithms are known to generate schedules from a reduced solution space that includes the optimal solution. The Giffler and Thompson algorithm is modified as shown in algorithm 1 such that if more than one machine shares the same minimum completion time, then all machines with that time are included in $S*$. The third method is included given that hyper-heuristics are often used in situations in which optimality is not required, merely an acceptable solution found quickly — note however, that the evolutionary process might subsequently discard this rule. An operation is chosen for scheduling from $S*$ according to the value returned by the priority rule.

Note that there are $28 * 2 * 3 = 168$ possible dispatching rules described by trees of depth = 0 only: a rule can contain one of 28 terminals, each of which can be set as positive or negative and use one of the 3 possible conflict sets. The number of potential complex rules composed of trees of depth > 0 is obviously considerably larger.

3.2 Heuristics

A heuristic, h_i, is defined as a variable length sequence of rules, of minimum length 1 and maximum length R. To produce a solution for a problem instance, each rule in the heuristic is applied in turn and schedules a single operation. After the last rule in the sequence is used the procedure repeats starting from the first rule until all operations have been scheduled. When a rule is selected, the subset $S*$ of operations to be considered for scheduling is generated according to the scheduling rule A,B,C defined by the rule. The tree defined by the rule is then applied to sort the available operations. The first operation from the sorted list is then selected. If more than one operation may be placed at the head of the sorted list (i.e. set of equal values is returned) then one is selected randomly. The selected operation is scheduled at the earliest available time on its predetermined machine as determined by the maximum of the completion time

of the last operation to be processed on the machine, the completion time of the last operation from the selected operations associated job and the job's arrival time.

3.3 Evolving Ensembles of Heuristics

Evolution of the ensemble is managed using the NELLI method previously mentioned in section 2, modified in order to cope with the use of tree-based rules. In brief, the algorithm provides a cooperative method of evolving a set of heuristics that interact to cover a problem space, favouring *heuristics* that are able to find a niche in solving at least one problem better than any other heuristic in the system. Heuristics are sustained in the system by 'winning' instances, e.g. achieving a result on an instance that is better than anything else in the ensemble.

The heuristics in the ensemble continuously adapt due to evolutionary processes that continually trial new heuristics, and the size of the ensemble is an emergent property of the system. On the one hand, this enables the network to both continually improve its response *and* adapt to changing problem instances. On the other, it provides an immediate solution to new problem instances that appear as each heuristics generalises over some part of the problem space. A full description of the algorithm can be found in Sim et al. (2013, 2015); Sim and Hart (2014); Hart and Sim (2014) which describes applications in which heuristics were formed from pre-defined rules only.

4 Implementation

A conceptual view of the system is given in figure 2 and is described by Algorithm 3. The system contains three key components: a continual stream of problem instances (the *problem-space*), a continual stream of novel heuristics and a network of heuristics inspired by the idiotypic model of the immune system. The dynamics of the network determine the final constitution of the system in retaining only problems that are representative of characteristic regions of the problem space, and heuristics that solve problems that cannot be solved by other heuristics. The system was originally proposed as a *lifelong learning* approach that could autonomously adapt to changing problem characteristics (Sim et al., 2015) — here we use a modified form to evolve a fixed ensemble of heuristics that collectively map the problem space based on a training set of problems. As in any machine learning method, the quality of the system relies on the training set being representative of the problem space, hence a large, diverse set of problem instances is desirable.

The key improvements to previous work using NELLI Sim and Hart (2014) are provided by the new heuristic generator. Heuristics are now represented as sequences of novel tree-based dispatching rules. This also differs from previous work in Sim et al. (2015) that uses Single-Node genetic programming to represent bin-packing heuristics. The new approach provides increased diversity and results in a trained ensemble that better generalises across the problem space. A brief descripton of the core network is provided before describing the novel modifications in detail.

4.1 Immune Network

The immune network lying at the heart of the system is unchanged from the system used previously, e.g. (Hart and Sim, 2014), and is covered here briefly for clarity. The

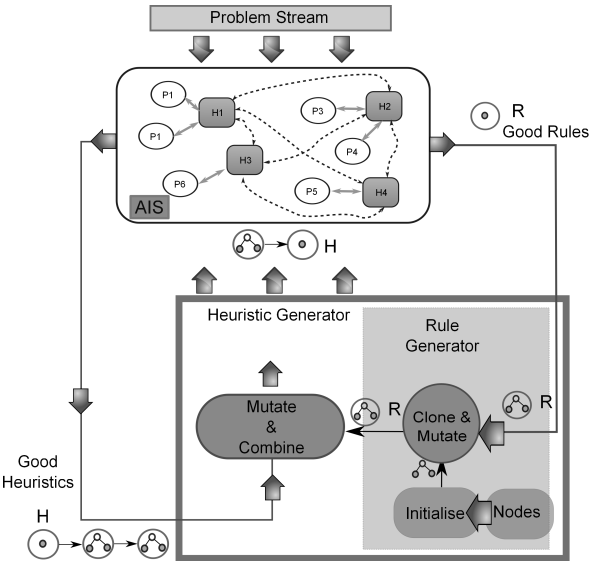


Figure 2: NELLI-GP comprises three components: a heuristic generator to supply a continual stream of novel heuristics, a set of problem instances and a network inspired by the idiotypic network theory of the immune system. The heuristic generator is expanded (the shaded area) here by the addition of a Rule Generator

$$h_{stim} = \sum_{p \in \mathcal{P}} \delta_p \begin{cases} \delta_p = \min\left(f(\mathcal{H}'_p)\right) - f(h_p) & : \text{ if } \min\left(f(\mathcal{H}'_p)\right) - f(h_p) > 0 \\ \delta_p = 0 & : \text{ otherwise} \end{cases} \quad (1)$$

$$p_{stim} = \sum_{h \in \mathcal{H}} \delta_h \begin{cases} \delta_h = \min\left(f(\mathcal{H}'_p)\right) - f(h_p) & : \text{ if } \min\left(f(\mathcal{H}'_p)\right) - f(h_p) > 0 \\ \delta_h = 0 & : \text{ otherwise} \end{cases} \quad (2)$$

Where $f(h_p)$ is the result (MS or SWT) achieved by heuristic h on problem p and $\min\left(f(\mathcal{H}'_p)\right)$ is the best result obtained on problem p by the rest of the ensemble $\mathcal{H}' = \mathcal{H} - h$

immune network models heuristics as antibodies and problems as pathogen. The objective is to find a minimal repertoire of heuristics that effectively solve the problems being supplied to the system. A heuristic 'wins' a problem instance if it obtains a solution to the instance that is better than *any* other heuristic in the system. The immune network promotes *behavioural* diversity within the ensemble in that for a heuristic to persist it must win at least one problem. Similarly, a problem only survives in the system if it is won by only *one* heuristic, i.e. it is representative of a hard part of the instance space. Both heuristics and problems have their behaviours governed by two variables — concentration and stimulation. The key steps of the immune network simulation are described from line 10 onwards in Algorithm 3. Each iteration, after new problems and heuristics have been injected into the system, the *stimulation* levels of both heuristics and problems are calculated using equations 1 and 2. Based on the stimulation received, concentration levels are adjusted up or down by a fixed amount (Δ_c) depending on whether the stimulation value of a cell lies within upper and lower thresholds. Heuristics and problems are injected into the system with an initial level

Algorithm 3 NELLI-GP Pseudo Code

Require: $\mathcal{H} = \emptyset$:The set of heuristics
Require: $\mathcal{P} = \emptyset$:The set of problems currently in the system
Require: $\mathcal{E}$:The set of all available problems
repeat
  $p = U(0, 1)$
  if $(p \leq prob_{hm}) \wedge (\mathcal{H} \neq \emptyset)$ **then**
    Generate a new *heuristic* h via heuristic mutation[1]
  else
    Initialise a new *heuristic* h via heuristic initialisation[2]
  end if
  Add h to $\mathcal{H}$ with concentration c_{init}
  Add $max(n_p, |\mathcal{E} - \mathcal{P}|)$ randomly selected problem instances from $\mathcal{E} - \mathcal{P}$ to $\mathcal{P}$ with concentration c_{init}
  Calculate $h_{stim} \forall h \in \mathcal{H}$ using Equation 1
  Calculate $p_{stim} \forall p \in \mathcal{P}$ using Equation 2
  Increment all concentrations (both $\mathcal{H}$ and $\mathcal{P}$) that have concentration $< c_{max}$ and stimulation > 0 by Δ_c
  Decrement all concentrations (both $\mathcal{H}$ and $\mathcal{P}$) with stimulation ≤ 0 by Δ_c
  Remove heuristics (from $\mathcal{H}$) and problems (from $\mathcal{P}$) with concentration ≤ 0
  while $|\mathcal{H}| > \mathcal{H}_{max}$ **do**
    Remove the heuristic that contributes the least[3] to the ensemble
  end while
until $stopping\ criteria\ met$

For all experiments conducted here $prob_{hm} = 0.9,\ n_p = 10,\ p_m = 0.9$
$\Delta_c = 50, c_{max} = 1000, c_{init} = 200$
$\mathcal{H}_{max}$ is varied between 1 and infinity.
The $stopping\ criteria$ is set to 10000 iterations

[1] [2] are described by Algorithm 5 and Algorithm 4 respectively.
[3] is determined as the heuristic that contributes the least to the global fitness achieved by greedily applying all $h \in \mathcal{H}$ to all $p \in \mathcal{P}$

of concentration (c_{init}) that allows them to persist in the network for a number of iterations, even without receiving any stimulation. If any concentration level falls to zero or below it is removed from the system. Heuristics have an implicit affect on each others behaviour leading to complex network behaviour; for example a newly introduced heuristic that outperforms an existing heuristic may limit the stimulation received by the existing heuristic to such an extent that its concentration is reduced and it is eventually removed from the network.

Note that there is no *global* fitness function: a new heuristic is only included in the ensemble if it wins at least one instance from another heuristic, hence every addition of a new heuristic necessarily improves global fitness. The stimulation level of a heuristic provides a measure of its *individual fitness*: this drives selection for mutation, providing evolutionary pressure for individuals to improve. Specialisation of heuristics to specific niches with the instance space occurs due to the explicit need to win instances to survive.

4.2 Heuristic Generator

As previously described, a heuristic is composed of a sequence of *rules*. Each rule comprises two parts; a method of generating a conflict set of waiting operations and a tree dispatching rule that is used to prioritise the operations from the conflict set. New heuristics are created through cloning, initialisation and mutation processes, the latter two of which are described by Algorithms 4 and 5 and remain identical to our previous work described in Sim and Hart (2015a).

Algorithm 4 Heuristic Initialisation

$n = U(1 \ldots r_{max\ initial})$: random integer value between 1 and $r_{max\ initial}$
$\mathcal{R} = \emptyset$
while $|\mathcal{R}| < n$ **do**
  $\mathcal{R} = \mathcal{R}$ + getRule {using Algorithm 6}
end while
Return a new heuristic created using the set of rules $\mathcal{R}$

Algorithm 5 Heuristic Mutation

Require: $\mathcal{H}$: The set of heuristics currently in the system
if $\mathcal{H} = \emptyset$ **then**
  generate new heuristic through initialisation. Algorithm 4
else
  $h = roulette(\mathcal{H})$: Roulette selection based on heuristic stimulation
  $\mathcal{R} = \{r_1 \ldots r_n\}$: The set of n rules that make up heuristic h
  With equal probability generate a new heuristic by:
    1 Cloning h
    2 Adding a rule to $\mathcal{R}$ at a random position.
      the rule is selected using GetRule[1]
    3 Removing a random rule from $\mathcal{R}$
      (if $|\mathcal{R}| > 1$ else do nothing)
    4 Swap the position of 2 randomly selected rules from $\mathcal{R}$
      (if $|\mathcal{R}| > 1$ else do nothing)
    5 Replace a random rule from $\mathcal{R}$ with a new rule selected using GetRule[1]
    6 Select a second heuristic $h' = roulette(\mathcal{H})$
      generate a new heuristic by concatenating the rules in R' to R
end if
Return the new heuristic

Both the heuristic mutation and heuristic initialisation procedures may result in the requirement for new rules to be generated. In previous work (Sim and Hart, 2015a) when a rule was required it was selected randomly from a set of 13 rules taken directly from the scheduling literature. In the work presented here, a rule generator is implemented that creates novel rules formulated as tree structures as described in section 3.1. The process described in Algorithm 6 outlines the method for generating a new rule. New rules can be formed either through initialisation, using the ramped half and half method of Koza (1992), as described in Algorithm 7, or by mutating an existing rule using sub-tree swap mutation.

Algorithm 6 Get Rule

Require: $\mathcal{H}$: The set of heuristics currently in the system
if $\mathcal{H} = \emptyset$ **then**
  generate a new rule r using a random conflict set and a priority rule formulated as a tree generated through tree initialisation. Algorithm 7
else
  $h = roulette(\mathcal{H})$: Roulette selection based on heuristic stimulation
  $\mathcal{R} = \{r_1 \ldots r_n\}$: The set of n rules that make up heuristic h
  $r = U(\mathcal{R})$: Select a random rule from $\mathcal{R}$
end if
$r' = mutate(r)$[1]
if $depth(r') < depth_{max}$ **then**
  $r = r'$
end if
return r
[1] $mutate(r)$ replaces a random node from rule r with a new sub-tree generated using Algorithm 7

The current set of heuristics sustained by the network in NELLI-GP represents a

Algorithm 7 Initialise Tree

Require: $depth_{initial}$: The maximum initial depth for a tree
1: $d = U(0 \ldots depth_{initial})$: Random integer between 0 and $depth_{initial}$
2: $p = U(0, 1)$: Random double between 0 and 1
3: **if** $p < 0.5$ **then**
4: generate a tree of maximum depth d using the *grow* tree method[1]
5: **else**
6: generate a tree of depth d using the *full* tree method[1]
7: **end if**
8: return tree

[1] If the procedure detailed above is used recursively to generate a population of *trees* then it is identical to the classic *ramped half and half* method of initialising a GP populations taken from Koza (1992) in which can also be found details of the *grow tree* and *full tree* methods used at lines 4 and 6.

population of available rules that is used as the starting point to generate new rules. This is motivated by the fact that any heuristic maintained in the system has already proved it provides a contribution to the ensemble, and thus can be considered as 'high fitness'. Unlike conventional GP, which uses crossover to converge to a optimal solution, only cloning and mutation procedures are used here to generate new heuristics and rules. The rationale for this implementation decision is that the purpose of the ensemble is to find *diverse heuristics*, therefore utilising crossover was deemed unhelpful as crossover inevitably leads to convergence. Initial empirical investigation including crossover validated this assertion, leading to convergence of heuristics and no improvement in the collective performance of the ensemble: this is directly antagonistic to the goal of NELLI-GP to sustain a diverse population of heuristics.

4.3 Problem Generator

In addition to a stream of heuristics, NELLI-GP requires a continuous stream of problem instances that ideally are widely distributed over the potential problem space. Many JSSP datasets are available in the literature (Applegate and Cook, 1991; Lawrence, 1984; Taillard, 1993). Often, these problems have known lower bounds which provide useful benchmarks, but tend to consist of small sets of problems generated from a specified parameter set. A common approach from researchers studying JSSP has been to generate new problems using a specified problem generator, e.g. Hunt et al. (2014). This method has the advantage of being able to create very large numbers of problems; furthermore, by varying the parameters of the generator, a wide area of the potential problem space can be covered. For both of these reasons, we use the latter approach.The generator is made available on-line at at Sim and Hart (2015b).

We consider problems specified by n jobs and m machines, where $n \in [10, 25, 50, 100]$ and $m \in [5, 10, 15, 20, 25]$. Instances are generated from each possible combination (n, m). The processing time for an operation is selected randomly from a uniform distribution following $p_{ir} = U[2, ..., 10]$where p_{ik} refers to the processing time of the operation of job j on machine k. Release dates are drawn randomly from one of two distributions as in Tay and Ho (2008) using Equation 3 depending on the number of jobs in the problem instance.

$$r_i = \begin{cases} U[0, 20] & \text{if } n < 50 \\ U[0, 40] & \text{if } n \geq 50 \end{cases} \tag{3}$$

Due dates are defined as in Baker (1984) using Equation 4. The term c is fixed at 1.3, the average of the values used in Singer and Pinedo (1998) which investigated the difficulty of relatively "tight" problems generated with $c = 1.4$ and $c = 1.2$.

$$d_i = r_i + c \times \sum_{j=1}^{n} p_{ij} \tag{4}$$

Job weights ($w_i \ldots w_n$) are selected using the 4:2:1 rule taken from Zhou et al. (2009) which is informed by research suggesting that 20% of a company's customers are the most important, 60 % are of of average importance and 20% are less important. Consequently the first 20% of jobs in an instance are assigned a weight of 4, the next 60% receive a weight of 2 and the remaining 20% of an instances jobs are given a weight of 1. The order in which a job's operations are assigned to machines may differ for each job and is allocated randomly during the problem generation process. All jobs visit each machine exactly once. In total, 20 different instance classes are defined by the combinations of (n, m, p, r, d). Within a class, each instance is unique due to the stochastic nature of p (processing time) and r (release date).

4.4 Training and Test Sets

35 instances are generated from each of the 20 classes. Within each class, 25 instances are selected at random and added to a *training set*, with the remaining 10 allocated to a *test* set. Thus, 700 problem instances are generated in total with 500 instances in the training set, and 200 in the test set.

Each of the data sets is considered using two different objective fitness functions. The *makespan* is a measure of the time to completion of the last job and is given by $C_{max} = max\{C_i\} : 1 \leq i \leq n$, where C_i is the completion time of job i. The Summed Weighted Tardiness relates to the lateness of a job and is given by $\sum_{i=1}^{n} \omega_i T_i$, where the tardiness of a job is denoted by T_i and $T_i = max\{C_i - d_i, 0\}$. In the following section we describe a number of experiments that are conducted separately using each of these objectives. All problem instances are available at Sim and Hart (2015b).

5 Experimental Method

The experiments described below have the following objectives:

- to investigate whether there is a performance benefit from evolving tree-based rules in comparison to a large set of pre-defined scheduling rules
- to investigate if a performance benefit is obtained by combining rules into heuristics composed of linear sequences of rules, compared to using single rules
- to investigate the benefit of using an ensemble of heuristics compared to single heuristics
- to compare the results obtained to the state-of-the-art from the literature

To address the first three objectives, ensembles described by $\langle H_x, D_y, R_z \rangle$ are evolved using the training and test sets described in section 4.4. Note that experiments

in which $x > 1$ describe *ensembles* consisting of more than one heuristic. Experiments in which $y = 0$ have rules of depth 0, i.e only use the scheduling rules defined in the literature. Experiments in which $z = 1$ use a single scheduling rule (either evolved or from the literature), rather than heuristic composed of a sequence of rules.

Parameters used in all experiments are given in table 3. These are unchanged from those used in previous works with additional parameters added relating to the initial and maximum depth for trees which are both set to either 0 or to 6 (d_{init}) and 17 (d_{max}) respectively. The probability p_{ais} and p_m are set to high values to encourage exploitation, i.e. mutation of existing rules and/or heuristics. This follows work reported in Sim and Hart (2014) that noted that the best results were obtained when new heuristics were formulated predominantly by mutating existing heuristics. Only two values of R_{max} and D_{max} were investigated: R_{max} was either set to 1 or U = *unlimited*, i.e. the length of the sequence of rules defining a heuristics was either exactly one or set to unlimited, so that the sequences could grow to any length depending on the evolutionary operators applied. The maximum depth of the trees was either set to 0 (use only a single terminal node) or 17 as in Koza (1992). The size of the ensemble was varied such that $H \in 1, 2, 4, 8, 16, 32, 64, 128, 256, U$ where U indicates *unlimited*, i.e. no enforced restriction on the size of the ensemble.

As the rules are all stochastic (more than one operation may receive the highest priority score which results in a random selection from the set of highest priority operations) all experiments were conducted 30 times, each for 10,000 iterations. This results in 30 sets of results for each training scenario: the heuristic(s) that gave the best results for each scenario during training were evaluated 30 times on set of 200 test problems using the corresponding objective function.

Table 3: Parameters

New heuristics per iteration	n_h	1
Number of iterations	$iter_{max}$	10000
Problem instances per iteration	n_p	10
Maximum Iterations	$iter_{max}$	10000
Minimum stimulation level	$stim_{min}$	0
Maximum stimulation level	$stim_{max}$	∞
Initial concentration level	c_{init}	200
Maximum concentration level	c_{max}	1000
Concentration change	δ_c	50
Maximum number of heuristics	$\mathcal{H}_{max}$	$\in \{1, 2, 4, 8, 16, 32, 64, 128, 256, U\}$
Maximum rule length	$\mathcal{R}_{max}$	$\in \{1, U\}$
Maximum Initial depth	D_{init}	$\in \{0, 6\}$
Maximum bloat depth	D_{max}	$\in \{0, 17\}$
Probability of heuristic mutation	p_m	0.9
Probability of selecting existing rule	p_{ais}	0.9

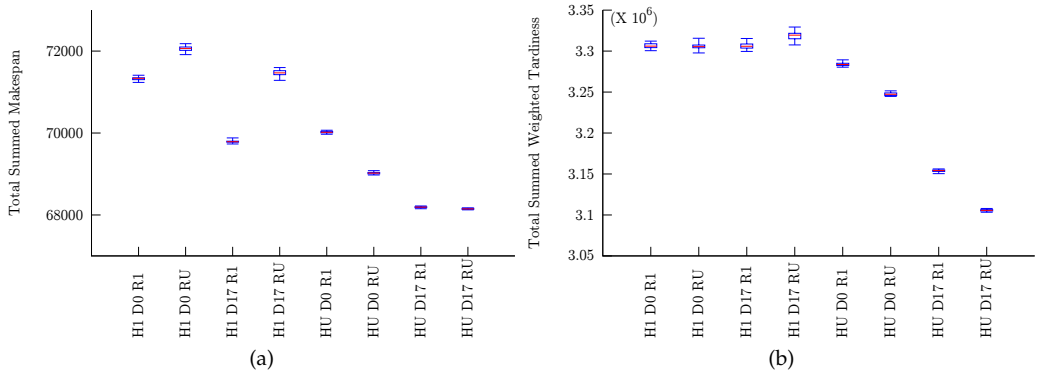


Figure 3: Results on the test problems for Makespan (left) and SWT (right)

5.1 Results

This section gives results from experiments in which ensembles were evolved using the training set of 500 instances and test set of 200 instances described in 4.4. Figure 3 shows boxplots that directly contrast performance when switching between trees, heuristics and ensembles on the test sets for both objectives. In each case, the four left-hand results (H=1) indicate 'one-size-fits-all' experiments in which a single heuristic is evolved. The right-hand four results represent ensembles, in which the components can consist of single rules (R=1) or sequences of rules (R=U). Configurations with $D_{max} = 0$ represent rules composed only of terminal nodes. Those with $D_{max} = 17$ contain tree-based rules. For the C_{max} objective, $H1\ D0\ R1$ corresponds exactly to the rule $-JRPT$ (annotated with conflict set B) shown in table 1 which is the best performing single terminal node and to $-JACT$ (again with conflict set B) for the SWT objective. The box plots present the summed fitness over the 200 test instances obtained from each configuration over 30 runs. The following general observations are made:

- Using an evolved rule ($D_{max} = 17$) is always preferable to creating heuristics from the terminal nodes only, regardless of whether ensembles or rule sequences are used
- For the non-ensemble methods (H=1), then a heuristic that contains a single rule rather than a sequence of rules is preferable: this is not observed in the ensemble methods in which the sequences (R=U) provide better results. This may result from overfitting on the training set when a single sequence is evolved, particularly for $D = 0$.
- Ensembles provide better results than a single 'one-size-fits-all' approach ($H = 1$), regardless of R and D.
- The ensemble method used in conjunction with $DU-RU$ is the clear winner, showing the benefit of an ensemble, of evolving trees, and sequencing them into heuristics

Two-tailed t-tests between all pairs of results show that all observations are statistically significant at the 5% confidence level, with the exception of the comparison

Table 4: C_{max} For each configuration <H*D*R*> , the table shows the median ratio to theoretical optimal and summed fitness over all instances, and average rank compared to the other configurations. Results taken from the single best run in all cases.

		Train			Test		
		Median Ratio (num optimal)	Summed fitness	Average Rank (num rank 1)	Median Ratio (num optimal)	Summed Fitness	Average Rank (num rank 1)
One heuristic (rules=1)	$H1\ D0\ R1$	1.12 (4)	178329	7.452(4)	1.30 (1)	71236	6.355(2)
	$H1\ D17\ R1$	1.09 (62)	173299	4.486 (81)	1.10 (20)	69729	3.935(36)
One heuristic (rules > 1)	$H1\ D0\ RU$	1.10 (78)	174113	4.232 (113)	1.15(0)	71914	6.945 (0)
	$H1\ D17\ RU$	1.09 (67)	172892	4.036 (90)	1.15 (0)	71286	6.415 (1)
Ensemble (rules=1)	$HU\ D0\ R1$	1.10 (49)	175151	5.474 (58)	1.10(19)	69970	4.225 (28)
	$HU\ D17\ R1$	**1.05 (121)**	169696	**1.302 (363)**	1.07 (40)	68146	**1.405** (129)
Ensemble (rules > 1)	$HU\ D0\ RU$	1.07 (94)	171841	2.694 (191)	1.09 (25)	68976	2.555 (60)
	$HU\ D17\ RU$	**1.05 (121**	**169777**	1.400 (337)	**1.07 (45)**	**68125**	1.430 **(133)**

$H1D0R1$ to $H1D17R1$ using the SWT objective, where we were unable to find a statistically better single rule that generalised better than the JATC rule, known for its performance on the SWT objective. However our results using ensembles ($H > 1$) improved on the single best rule on all occasions.

Tables 4 and 5 provide results from each configuration from the single best run obtained for both objectives. As the summed fitness metric used in figure 3 can be distorted by the larger instances, and masks performance on individual instances, we also provide an comparison to the theoretical optimal. For makespan this can be easily calculated (Taillard, 1993); the table gives the median value for the ratio *actual/optimal* makespan. For the TWT objective, we estimate the *theoretical* optimal as the sum of the weighted *due-dates*, and calculate the ratio of the summed weighted *actual* arrival date to the summed weighted due-dates. For both objectives, we also give the actual number of optimal solutions found. Additionally, we rank each of the 8 configurations on each instance (where rank 1 is best) and calculate the average rank over all instances, and the number of instances assigned rank 1 for each configuration.

In terms of the summed fitness, the ensemble composed of heuristics containing rule-sequences (HUD17RU) provides the best result for both objectives. Comparing average ranks and instances of rank 1, the same ensemble proves best for the TWT objectives. It is marginally beaten by HUD17R1 for makespan in terms of average rank, but has more instances assigned rank 1. In terms of number of optimal solutions and median ratio, although the ensemble methods are clearly preferable, there is little difference between the ensemble of rule sequences (HUD17RU) and the ensemble of single rules (HUD17R1). The RU configuration appears to faciliate the emergence of heuristics that improve fitness (reducing the sum) on some of the larger instances, while having little impact on the overall ratio to optimal, most likely due to the increased size of the search space.

5.2 Comparison to a Disposable Hyper-heuristic Approach

Hyper-heuristic methods aim to find quick and acceptable solutions to problems, often trading some loss in quality against speed of producing a solution. Given a new problem instance p and a trained ensemble containing h heuristics, then exactly h heuristics need to be executed once to find the best solution to the instance. In contrast, a typical meta-heuristic approach applied to each instance to obtain a heuristic would run

Table 5: TWT For each configuration <H*D*R*> , the table shows the median ratio to theoretical optimal and summed fitness over all instances, and average rank compared to the other configurations. Results taken from the single best run in all cases.

		Train			Test		
		Median Ratio (num optimal)	Summed fitness	Average Rank (num rank 1)	Median Ratio (num optimal)	Summed Fitness	Average Rank (num rank 1)
One heuristic (rules=1)	$H1\ D0\ R1$	1.64 (7)	8289211	6.784 (7)	1.64 (0)	3300505	5.625 (0)
	$H1\ D17\ R1$	1.62 (3)	7974321	4.956 (14)	1.65 (0)	3300097	5.790 (0)
One heuristic (rules > 1)	$H1\ D0\ RU$	1.64 (7)	8112045	5.318 (8)	1.64 (0)	3301616	5.725 (0)
	$H1\ D17\ RU$	1.59 (8)	7915908	3.852 (37)	1.64 (0)	3307548	6.00 (0)
Ensemble (rules=1)	$HU\ D0\ R1$	1.64(26)	8237717	5.93 (29)	1.57 (9)	3280346	4.505 (14)
	$HU\ D17\ R1$	1.56 **(39)**	7793957	1.760 (217)	1.57 **(12)**	3151370	1.855 (49)
Ensemble (rules > 1)	$HU\ D0\ RU$	1.60(35)	8073411	4.086 (64)	1.60 (10)	3244691	3.160 (22)
	$HU\ D17\ RU$	**1.55** (39)	**7752270**	**1.600 (279)**	**1.56** (11)	**3103310**	**1.285 (157)**

for i iterations. Although not entirely fair, it is instructive to compare the quality of solutions obtained by greedy selection from an evolved ensemble to the quality of the solutions obtained by directly evolving a new heuristic to solve each individual instance. A standard GP algorithm with parameters set as in Koza (1992) (population size = 500, initialised using ramped half and half up to a depth of 6 using crossover 90% and cloning 10% and maximum tree depth of 17) was executed for 20 generations (10,000 evaluations as with NELLI-GP) is used. After initialisation, each tree is randomly assigned one of three possible rules to determine operation eligibility defined in section 3.1.2. This rule does not undergo mutation. The following experiments are performed 30 times:

- GP (1P) : GP is run on each of the 200 instances in the test set in isolation. i.e. a single rule is evolved for each test instance.
- GP (200P) : GP is run on the full set of the 200 test instances. i.e. 1 rule is evolved for the complete test set.

Table 6 shows the best, mean and standard deviation obtained over 30 runs. T-test results in each column show a comparison with the previous columns results. As well as results from the two experiments described, 3 sets of results from the experiments conducted in the previous section are included for comparison. GP (1P) and $HU\ D17\ R1$ both evolves exactly one rule per problem, and hence can be directly compared. Similarly $H1\ D17\ R1$ and GP (200P) both evolve a single rule that is evaluated on all 200 problem instances and hence are directly comparable. The results of applying the best ensemble obtained during training ($H1\ D17\ RU$) to the test set are also included.

As would be expected GP(1P), which evolves a different rule for each problem instance, obtains better results than both GP(200P) and $H1\ D17\ R1$ which evolve only a single generalist rule. Interestingly, when the evolved $H1\ D17\ R1$ rule is applied to the test set it outperforms the single rule that was evolved by GP directly on those instances, demonstrating good generalisation from the training set. All of the above results are eclipsed by the results obtained by applying the reusable *ensembles* of heuristics generated by NELLI-GP. Furthermore the best trained ensemble ($HU\ D17\ RU$), comprising of 264 heuristics, requires less than 3 % of the evaluations required for either of the GP experiments.

Table 6: Comparison of ensembles (C_{max}) to disposable hyper-heuristic approaches

	GP (200P)	H1 D17 R1	GP (1P)	HU D17 R1	HU D17 RU
Mean	70761.43	69788.20	69127.93	68183.57	68148.07
Best	69795	69729	69068	68146	68125
SD	296.59	29.46	34.80	19.75	13.94
T-test	—	2.06E-17	1.67E-34	3.56E-40	5.32E-08

5.3 Comparison to Existing Approaches

We compare our method to two recent approaches from the literature. Nguyen et al. (2013b) describe a new GP based approach to learning new iterative dispatching rules, using four well-known benchmarks datatsets (LA, ORB, TA, DMU). A training set is formed consisting of all odd numbered instances from each of the four sets, with the remainder making up a test-set. Each set has 105 instances. Park et al. (2015) use GP to evolve an ensemble of rules that vote to determine which operation is selected for scheduling at each iteration. Their approach is tested on the 80 instance Taillard (TA) set (which is included in Nguyen et al. (2013b)), using three different training sets each containing 5 instances, with the remaining 65 instances making up the test set in each case.

A direct comparison of ensemble methods is given by comparing NELLI-GP to the ensemble method of Park et al. (2015), i.e. the 'Mixture-of-experts' ensembles of the former and the majority voting approach taken by the latter. We use the same training and test sets described in (Park et al., 2015) and evolve an ensemble of heuristics using NELLI-GP in the form $H4\ D17\ RU$ in order to allow a fair comparison to the 4 islands used by Park et al. (2015). In order to provide an exact comparison, NELLI-GP is first limited to exactly the same set of function and terminal nodes used by Park et al. (2015). The experiment is then repeated using the full set of nodes given in table 7. Both NELLI-GP ensembles outperform the results from Park et al. (2015). T-tests at the 5% significance level confirm this result. T-tests additionally show no significant difference between the NELLI-GP results with different terminal sets. To compare to Nguyen et al. (2013b), we evolve an ensemble defined by HUD17RU using exactly the same training and test sets as their approach. Results averaged over 30 runs are given in the final line of table 8. NELLI-GP significantly outperforms the published approach on both training and test sets.

The ensembles evolved by NELLI-GP should be *reusable*. We take ensembles evolved on the new 500 training instances (as described in section 4.4) and reuse on the test sets used by Nguyen et al. (2013b) and Park et al. (2015), comparing to their published results. Results are given in table 8 for three ensembles ($H4, H64, HU$). All three significantly outperform EGP-JSS from Park et al. (2015) on the 65 Taillard instances. $H8$ and HU significantly outperform the results of the new GP method Δ' proposed by Nguyen et al. (2013b) and the results from their simple GP algorithm Δ. Clearly the NELLI-GP ensembles are *reusable* — ensembles evolved on one data-set can be directly applied to a *different* dataset, and demonstrate improved performance over methods that were tailored to the tested dataset. In addition, the ensemble provides a *robust* optimisation method. The instances in the datasets given in table 8 have similar parameters in terms of *(jobs, machines)* but contain operations whose processing times

Table 7: Comparison of NELLI-GP to EGP-JSS from Park et al. (2015)

Training Set	EGP-JSS		NELLI-GP: Node Set from Park et al. (2015)		NELLI-GP: Node set from Table 1	
	Train	Test	Train	Test	Train	Test
T1	0.45±0.03	0.26±0.04	0.38±0.07	**0.2±0.09**	0.38±0.03	0.22 ± 0.13
T2	0.33±0.04	0.26±0.03	0.36±0.08	**0.18±0.03**	0.38±0.05	**0.18±0.05**
T3	0.06±0.01	0.26±0.01	0.02±0.01	**0.18±0.04**	0.03±0.02	**0.18±0.05**

Table 8: Reusability and robustness of ensembles: NELLI-GP$_T$ refers to ensembles trained on the 500 new instances and applied directly to new test sets. Results from EGP-SS and Δ, Δ' taken directly from Park et al. (2015), Nguyen et al. (2013b) respectively. For comparison, NELLI-GP$_E$ gives the result from evolving an ensemble on the same training set given in the relevant publications

		Park et al. (2015)	Nguyen et al. (2013b)	
		65 test instances	105 train	105 test
(Park, 2015)	EGP-JSS	0.26 ±0.001		
(Nguyen, 2013)	Δ		0.179 ± 0.003	0.187 ± 0.005
	Δ'		0.151 ± 0.004	0.160 ± 0.005
NELLI-GP$_T$	$H4\ D17\ RU$	**0.18** ±0.005	0.178 ± 0.001	0.171 ±0.002
	$H64\ D17\ RU$	**0.14** ±0.002	**0.140** ±0.001	**0.136** ± 0.001
	$HU\ D17\ RU$	**0.15**±0.002	**0.140** ±0.001	**0.136** ± 0.001
NELLI-GP$_E$	$HU\ D17\ RU$	**0.18 ± 0.05**	**0.119**±0.006	**0.127 ± 0.001**

are drawn from different distributions to those used by the instances used to evolve the ensembles. Hence, they can be considered perturbed versions of the original instance set.

6 Analysis

This section provides an analysis of the ensemble in terms of the role and structure of the constituent heuristics, and relationship between heuristic performance and problem structure.

6.1 Effect of ensemble size

The experiments in section 5 allow ensembles of unlimited size ($H = U$) to evolve, although in practice evolution limits the actual size that emerges. In the following experiments, we examine the effect of restricting the size of the ensemble to $H \in (1, 2, 4, 8, 16, 32, 64, 128, 256)$ rather than allowing evolution to proceed unrestricted. Figure 4 shows results on the test sets for both objective functions as H is varied.

A clear correlation between the size of the ensemble and the collective performance of the ensemble is apparent in all cases. However the benefit clearly tails off as the size of the ensemble increases beyond a saturation level. An ensemble size of $|H| \approx |P|/10$ appears appropriate as a rule of thumb given the investigations conducted. However, this is clearly problem dependent, the larger and more diverse the training set the better the evolved heuristics.

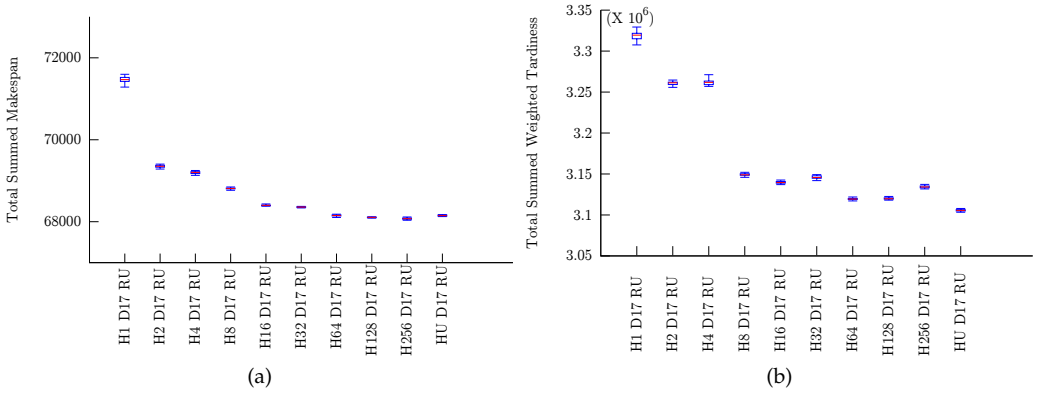


Figure 4: Varying ensemble size: Makespan (left) and SWT (right) results

6.2 Structure of evolved heuristics

The heuristics contained in evolved ensembles of size $1 - 264$ are analysed in terms of the number of rules contained in each heuristic, and the number of nodes in each rule, with results given in table 9. As previously noted the number of heuristics in the ensemble, the number of rules in a heuristic and the tree depth of the rules is an emergent property of NELLI-GP. Some general trends are observed. Heuristics tend to increase in complexity in terms of the number of rules-per-heuristic as the ensemble size increases, peaking at $|H| = 64$ when the trend reverses. This suggests that large ensembles favour less complex rules, as each heuristic needs to operate in a smaller region of the instance space. In terms of nodes-per-rule, the pattern is less obvious; the number of nodes tends to increase as the size of the ensemble increases, but does not grow linearly. The average depth per rule is given in the final column — note that in all cases, this is significantly less that the maximum allowed depth of 17 and often produces small, easy to analyse heuristics with depth around 5. In contrast, in typical GP, bloat is common. We suggest that in NELLI-GP, bloat is largely suppressed by the elitist requirement that a heuristic must solve at least one instance better than the existing heuristics to replace another in the ensemble. In standard GP with a large population, weak trees can be sustained due to the replace-worst step, encouraging bloat. A typical rule taken from 1 of the heuristics from $H64\ \ R17\ \ DU$ is shown in Figure 5

6.3 The role of the ensemble

To understand how each heuristic within the ensemble contributes to the collective performance we analyse an ensemble of h heuristics applied to the set of $i = 200$ test instances. For each instance we calculate f_i^*, i.e. the best result achieved on the instance by the ensemble, and calculate c_i as the number of heuristics that achieve f_i^* for a given instance i.

For instances with $c_i > 1$, multiple heuristics achieve the same result f^*. From an algorithmic perspective, we suggest that these instances do not represent 'interesting' regions in the instance space, given many heuristics perform equally well. In contrast,

Table 9: Analysis of the structure of the evolved heuristics

	Actual H per H	Avg R per R	Avg Tnodes per R	Avg Nodes	Avg (max) Depth Per R
$H1\ D17\ RU$	1	2	1	1	0 (0)
$H2\ D17\ RU$	2	1.5	2.33	3.67	1.33 (2)
$H4\ D17\ RU$	4	1	4.75	10	4.25 (5)
$H8\ D17\ RU$	8	2.5	10.4	22.95	6.00 (7)
$H16\ D17\ RU$	16	3.875	7.60	18.16	7.39 (12)
$H32\ D17\ RU$	32	2.25	7.76	15.07	5.24 (9)
$H64\ D17\ RU$	64	8.125	8.53	17.92	6.26 (10)
$H128\ D17\ RU$	128	6.31	7.24	20.24	9.94 (12)
$H256\ D17\ RU$	253	4.75	6.20	13.36	4.58 (9)
$HU\ D17\ RU$	264	3.19	8.24	17.22	4.89 (8)

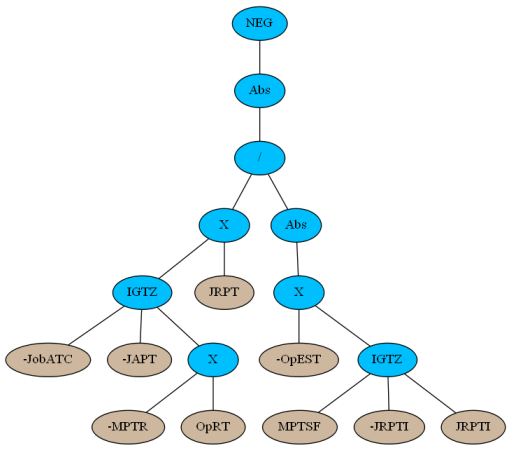
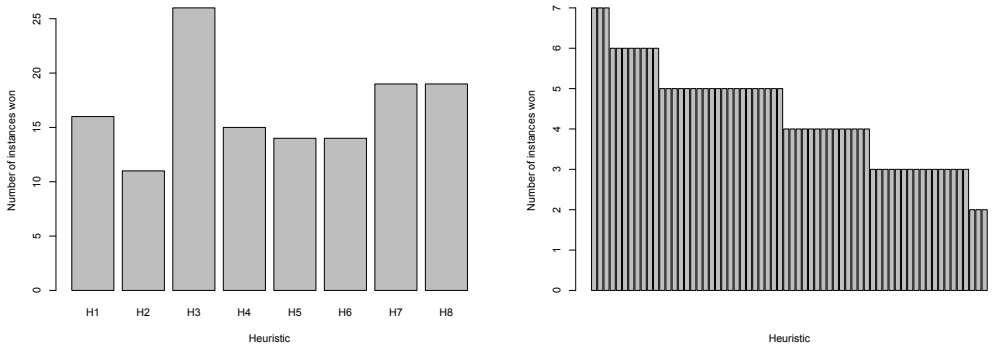


Figure 5: An average rule from 1 of the 64 heuristics evolved in experiment $H64\ R17\ DU$. The 64 heuristics contained on average 8 rules with rules having an average depth of 6. In general the depth of rules that emerged was much less than the maximum value of 17



(a) (b)

Figure 6: Number of instances uniquely won by each heuristic a) Ensemble of 8 heuristic (b) Ensemble of 64 heuristics (sorted by wins)

if $c_i = 1$, then the instance has a unique winner, and these instances represent regions in which algorithm-selection becomes a key issue.

Let $w(h_j)$ be the number of instances won by heuristic, h_j. A *specialist* heuristic, one with small $w(h_j)$, operates in a niche region of the instance space. A *generalist* heuristic on the other hand, with large $w(h_j)$, operates across large regions of the space. The balance between the specialist/generalist nature of the heuristics evolved is an emergent property of NELLI-GP that is closely linked to the structure of the instances in the dataset. We consider the 20 classes of makespan instances described in section 4.4 and the heuristics generated by the experiment $H8D17RU$ and $H64D17RU$. Figure 6 shows the number of instances $w(h_i)$ won by each of the heuristics for both experiments. For an ensemble of 8 heuristics, we observe that for 32% of the 200 instances, there is no uniquely best heuristic. For the remaining 134 instances 'interesting' instances, no particular heuristic is dominant; the most general heuristic (H3) wins 26 instances while the most specialist (H2) wins 11. In the ensemble of 64 heuristics, only 10% of instances do not have a unique winner. The 64 heuristics tend to specialise equally: the number of instances won ranges from 2-7, with no heuristic generalising across many instances.

6.3.1 Relationship between instance class and heuristic performance

NELLI-GP promotes the emergence of heuristics which are behaviourally diverse in that each heuristic has to uniquely win at least one instance in order to survive. Assuming that the instances within the subset won by each heuristic share similar features, we investigate the *intra*-class and *inter*-class membership of each subset, given the 20 classes defined in section 4.4. Figure 7 uses a network representation to capture the relationships between heuristics and classes for an 8 heuristic ensemble. An edge exists between a heuristic and a class if the heuristic wins at least one instance in that class. The weight of the edge reflects how many instances within the class were won, and the size of a heuristic denotes how many different classes it wins instances in. The degree of a *heuristic* indicates the number of different classes the heuristic wins instances from. The degree of a *class* indicates how many different heuristics win in-

	H1	H2	H3	H4	H5	H6	H7	H8
Total processing time	+			+			++	
Mean processing time			++	+			++	
Processing time range				++			++	
Ratio jobs:machines	++		++					

Table 10: Examining the distribution of features in set I_{H_i} compared to I_r, i.e. all remaining instances. Table shows p-values obtained from Wilcoxon rank sum test for each feature

	H1	H2	H3	H4	H5	H6	H7	H8
Total processing time	++			+			+	
Mean processing time								
Processing time range	+			+			+	+
Ratio jobs:machines	++		++					

Table 11: Examining the distribution of features in set I_{H_i} compared to I_u, i.e. the subset of remaining instances that are uniquely solved. Table shows p-values obtained from Wilcoxon rank sum test for each feature

stances from that class, thus is representative of intra-class niches. Heuristics have a median degree of 8.5, indicating there are clearly many *inter*-class relationships between instances. Classes have a median degree of 4, i.e. a typical class of 10 instances generated from the same parameter set has at least 4 *intra*-class clusters. H3 is strongly associated with class 18 (defined by 25 jobs and 25 machines) winning 7/10 instances. Only 3 of the 20 classes are uniquely associated with a heuristic, implying uniformity across instances within these classes. Note that two of these classes (1,2) contain the easiest (in terms of finding optimal solutions) instances, due to the small number of jobs and machines.

Clusters of similar instances — those won by the same heuristic — naturally emerge from NELLI-GP. Each cluster can be examined with respect to a feature set to potentially identify correlations across instances. Ingimundardottir and Runarsson (2012) consider a set of 16 features and attempt to relate them to instance difficulty. 13 of these features are recalculated on a step-by-step basis after every operation has been scheduled (e.g. current makespan); the remaining 3 are static and apply to the instance as whole (e.g total job processing time). Smith-Miles et al. (2009) cluster instances (using a tardiness objective) according to 6 features, then examine the distribution of fitness values to infer knowledge about the relationships between instance structure and heuristic performance. Taking the $< H8 >$ ensemble as an example, for each heuristic in turn we examine whether the features of the instances within the cluster defined by the instances it uniquely won differ significantly from the remainder of the instances. As NELLI-GP assigns a single heuristic to an instance, only static features are relevant. We consider four features: *mean processing time, processing time range, total processing time* are used in both (Ingimundardottir and Runarsson, 2012; Smith-Miles et al., 2009); we add *ratio machines:jobs* as this is known to influence problem hardness (Streeter and Smith, 2006).

If I_t is the 200 instances in the test set then let I_{H_j} be the subset of instances uniquely won by heuristic H_j; I_r the subset of *all* remaining instances from I_t; I_u the

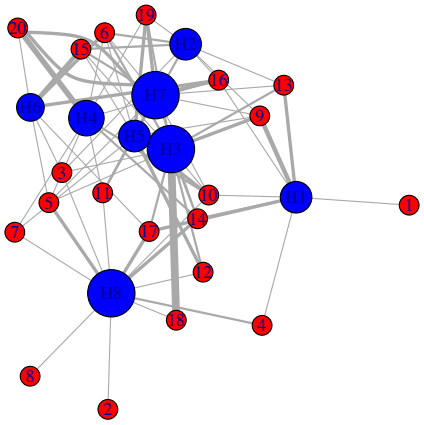


Figure 7: A network capturing the relationship between a heuristic the instance classes in which it wins instances. Edges are weighted according to the number of instances in the class won

instances in I_t *uniquely won*[1] by the remaining heuristics. For each of the four features, we apply a Wilcoxon rank sum test to compare the feature values in the instances in I_{H_i} to those in subset I_r, and then separately to those in I_u.

Results are shown in table 10 which differentiates between results significant at the 5% level (+) and at the 1% level (++). For three heuristics (H2, H5, H6) none of the features are discriminatory. However, for the remaining heuristics, at least one feature discriminates the instances won by the heuristic from the remainder of all instances or the remainder of heuristics uniquely won. All features are discriminatory in at least two tests. The results generally concur with the findings of Ingimundardottir and Runarsson (2012) who attempt to correlate features with hardness, noting that '*the features distinguishing hard problems were scarce. Most likely due to their more complex data structure their key features are of a more composite nature*'. NELLI-GP removes the need to define features in order to discriminate between instances; the clustering of instances *emerges* as a result of applying the algorithm. The emergent clusters capture structure within the problems that cannot be easily defined by humanly-intuitive features or by the features used to *generate* instances.

7 Conclusion

The automated design of reusable production scheduling heuristics is currently of great interest to the optimisation community as evidenced in detail by Branke et al. (2015). The need to move towards *ensemble* methods has been highlighted as an open challenge, given the requirement to deal with complex decisions and varied problem instances. We have described a novel ensemble method for evolving JSSP heuristics called NELLI-GP in which each heuristic in the ensemble generates solutions to problems in a niche region of the instance space.

[1] as a Shapiro-Wilk test showed that the null hypothesis that the distribution is normal can be rejected

Experiments on a large set of newly generated problem instances as well as existing benchmarks have shown that novel rules can be evolved that outperform existing scheduling rules and recent hyper-heuristic methods. The power of the method is attributed to (a) evolving new dispatching rules that both define job-eligibility and use a large set of terminal; (b) Combining rules into variable length sequences to form new heuristics; (c) evolving ensembles of heuristics, in which each operates in a distinct part of the instance space. The reusability and robustness of the ensemble is demonstrated by applying an ensemble evolved on one set of instances to new instance sets, outperforming existing benchmarks. Analysing the evolved ensemble in terms of performance on different classes has enabled new insights to be gained relating to instance-structure. In agreement with Ingimundardottir and Runarsson (2012) we find that intuitive problem features are insufficient to characterise instances. Using the instance clusters that emerge from running NELLI-GP however, future work will focus on attempting to characterise the clusters in terms of combinations of known features or deriving new complex features. In addition, we intend to apply NELLI-GP to *dynamic* scheduling problems in which it is possible to derive more fine-grained features.

Acknowledgment

This work is funded by EPSRC Grant Number EP/J021628/1

References

Applegate, D. and Cook, W. (1991). A computational study of the job-shop scheduling problem. *ORSA Journal on Computing*, 3(2):149–156.

Baker, K. R. (1984). Sequencing rules and due-date assignments in a job shop. *Management Science*, 30(9):1093–1104.

Bhowan, U., Johnston, M., Zhang, M., and Yao, X. (2013). Evolving diverse ensembles using genetic programming for classification with unbalanced data. *Evolutionary Computation, IEEE Transactions on*, 17(3):368–386.

Bhowan, U., Johnston, M., Zhang, M., and Yao, X. (2014). Reusing genetic programming for ensemble selection in classification of unbalanced data. *Evolutionary Computation, IEEE Transactions on*, 18(6):893–908.

Blackstone, J. H., Phillips, D. T., and Hogg, G. L. (1982). A state-of-the-art survey of dispatching rules for manufacturing job shop operations. *International Journal of Production Research*, 20(1):27–45.

Branke, J., Hildebrandt, T., and Scholz-Reiter, B. (2014). Hyper-heuristic evolution of dispatching rules: A comparison of rule representations. *Evolutionary Computation*, 23(2).

Branke, J., Nguyen, S., Pickardt, C., and Zhang, M. (2015). Automated design of production scheduling heuristics: a review. *IEEE Transactions on Evolutionary Computation*.

Breiman, L. (1996). Bagging predictors. *Machine learning*, 24(2):123–140.

Downey, C., Zhang, M., and Liu, J. (2012). Parallel linear genetic programming for multi-class classification. *Genetic Programming and Evolvable Machines*, 13(3):275–304.

Geiger, C. D., Uzsoy, R., and Aytug, H. (2006). Rapid modeling and discovery of priority dispatching rules: An autonomous learning approach. *J. of Scheduling*, 9(1):7–34.

Giffler, B. and Thompson, G. L. (1960). Algorithms for solving production-scheduling problems. *Operations Research*, 8(4):487–503.

Graham, R. L., Lawler, E. L., Lenstra, J. K., and Rinnooy Kan, A. H. G. (1979). Optimization and approximation in deterministic sequencing and scheduling: a survey. *Annals of discrete mathematics*, 5(2):287–326.

Hart, E. and Ross, P. (1998). A heuristic combination method for solving job-shop scheduling problems. In Eiben, A., Bäck, T., Schoenauer, M., and Schwefel, H.-P., editors, *Parallel Problem Solving from Nature PPSN V*, volume 1498 of *Lecture Notes in Computer Science*, pages 845–854. Springer Berlin / Heidelberg.

Hart, E. and Sim, K. (2014). On the life-long learning capabilities of a nelli*: A hyper-heuristic optimisation system. In Bartz-Beielstein, T., Branke, J., Filipič, B., and Smith, J., editors, *Parallel Problem Solving from Nature - PPSN XIII*, volume 8672 of *Lecture Notes in Computer Science*, pages 282–291. Springer International Publishing.

Haupt, R. (1989). A survey of priority rule-based scheduling. *Operations-Research-Spektrum*, 11(1):3–16.

Hildebrandt, T., Heger, J., and Scholz-Reiter, B. (2010). 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*, GECCO '10, pages 257–264, New York, NY, USA. ACM.

Hunt, R., Johnston, M., and Zhang, M. (2014). Evolving "less-myopic" scheduling rules for dynamic job shop scheduling with genetic programming. In *Proceedings of the 2014 Conference on Genetic and Evolutionary Computation*, GECCO '14, pages 927–934, New York, NY, USA. ACM.

Ingimundardottir, H. and Runarsson, T. P. (2012). Determining the characteristic of difficult job shop scheduling instances for a heuristic solution method. In *Learning and Intelligent Optimization*, pages 408–412. Springer.

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

Lawrence, S. (1984). Resource constrained project scheduling: An experimental investigation of heuristic scheduling techniques. Graduate School of Industrial Administration, Carnegie Mellon University, Pittsburgh.

Leyton-Brown, K., Nudelman, E., Andrew, G., McFadden, J., and Shoham, Y. (2003). A portfolio approach to algorithm select. In *Proceedings of the 18th International Joint Conference on Artificial Intelligence*, IJCAI'03, pages 1542–1543, San Francisco, CA, USA. Morgan Kaufmann Publishers Inc.

Miyashita, K. (2000). Job-shop scheduling with gp. In *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO-2000)*, pages 505–512. Morgan Kaufmann.

Nguyen, S., Zhang, M., Johnston, M., and Tan, K. C. (2013a). A computational study of representations in genetic programming to evolve dispatching rules for the job shop scheduling problem. *Evolutionary Computation, IEEE Transactions on*, 17(5):621–639.

Nguyen, S., Zhang, M., Johnston, M., and Tan, K. C. (2013b). Learning iterative dispatching rules for job shop scheduling with genetic programming. *The International Journal of Advanced Manufacturing Technology*, 67(1-4):85–100.

Nguyen, S., Zhang, M., Johnston, M., and Tan, K. C. (2014a). Automatic design of scheduling policies for dynamic multi-objective job shop scheduling via cooperative coevolution genetic programming. *Evolutionary Computation, IEEE Transactions on*, 18(2):193–208.

Nguyen, S., Zhang, M., Johnston, M., and Tan, K. C. (2014b). Selection schemes in surrogate-assisted genetic programming for job shop scheduling. In *Simulated Evolution and Learning*, pages 656–667. Springer.

Nguyen, S., Zhang, M., Johnston, M., and Tan, K. C. (2015). Automatic programming via iterated local search for dynamic job shop scheduling. *Cybernetics, IEEE Transactions on*, 45(1):1–14.

Panwalkar, S. S. and Iskander, W. (1977). A survey of scheduling rules. *Operations Research*, 25(1):45–61.

Park, J., Nguyen, S., Zhang, M., and Johnston, M. (2013). Genetic programming for order acceptance and scheduling. In *Evolutionary Computation (CEC), 2013 IEEE Congress on*, pages 1005–1012. IEEE.

Park, J., Nguyen, S., Zhang, M., and Johnston, M. (2015). Evolving ensembles of dispatching rules using genetic programming for job shop scheduling. pages 92–104. Springer International Publishing.

Pickardt, C., Hildebrandt, T., Branke, J., Heger, J., and Scholz-Reiter, B. (2013). Evolutionary generation of dispatching rule sets for complex dynamic scheduling problems. *Int. J. Prod. Econ.*, 145(1):6777.

Potter, M. A. and De Jong, K. A. (2000). Cooperative coevolution: An architecture for evolving coadapted subcomponents. *Evol. Comput.*, 8:1–29.

Sim, K. and Hart, E. (2014). An improved immune inspired hyper-heuristic for combinatorial optimisation problems. In *GECCO '14: Proceeding of the sixteenth annual conference on Genetic and evolutionary computation conference*.

Sim, K. and Hart, E. (2015a). A novel heuristic generator for jssp using a tree-based representation of dispatching rules. In *GECCO '15: Proceeding of the seventeenth annual conference on Genetic and evolutionary computation conference*.

Sim, K. and Hart, E. (2015b). Roll project job shop scheduling benchmark problems. `http://dx.doi.org/10.17869/ENU.2015.9365`.

Sim, K., Hart, E., and Paechter, B. (2013). Learning to solve bin packing problems with an immune inspired hyper-heuristic. In *Advances in Artificial Life, ECAL 2013: Proceedings of the Twelfth European Conference on the Synthesis and Simulation of Living Systems*, pages 856–863. MIT Press.

Sim, K., Hart, E., and Paechter, B. (2015). A lifelong learning hyper-heuristic method for bin packing. *Evolutionary Computation Journal*, 23(1):37–67.

Singer, M. and Pinedo, M. (1998). A computational study of branch and bound techniques for minimizing the total weighted tardiness in job shops. *IIE Transactions*, 30(2):109–118.

Smith-Miles, K., Baatar, D., Wreford, B., and Lewis, R. (2014). Towards objective measures of algorithm performance across instance space. *Computers & Operations Research*, 45:12–24.

Smith-Miles, K. A., James, R. J. W., Giffin, J. W., and Tu, Y. (2009). A knowledge discovery approach to understanding relationships between scheduling problem structure and heuristic performance. In *Learning and Intelligent Optimization: Third International Conference, LION 3, Trento, Italy, January 14-18, 2009. Selected Papers*, pages 89–103. Springer Berlin Heidelberg, Berlin, Heidelberg.

Streeter, M. J. and Smith, S. F. (2006). How the landscape of random job shop scheduling instances depends on the ratio of jobs to machines. *J. Artif. Intell. Res.(JAIR)*, 26:247–287.

Taillard, E. (1993). Benchmarks for basic scheduling problems. *European Journal of Operational Research*, 64(2):278–285.

Tay, J. C. and Ho, N. B. (2008). Evolving dispatching rules using genetic programming for solving multi-objective flexible job-shop problems. *Computers & Industrial Engineering*, 54(3):453–473.

Ulrich, D. and Erwin, P. (1995). Evolution based learning in a job shop scheduling environment. *Comput. Oper. Res.*, 22(1):25–40. 197887.

Valentini, G. and Masulli, F. (2002). Ensembles of learning machines. In Marinaro, M. and Tagliaferri, R., editors, *Neural Nets*, volume 2486 of *Lecture Notes in Computer Science*, pages 3–20. Springer Berlin Heidelberg.

Vepsalainen, A. P. J. and Morton, T. E. (1987). Priority rules for job shops with weighted tardiness costs. *Management Science*, 33(8):1035–1047.

Zhou, H., Cheung, W., and Leung, L. C. (2009). Minimizing weighted tardiness of job-shop scheduling using a hybrid genetic algorithm. *European Journal of Operational Research*, 194(3):637 – 649.