On the Automatic Discovery of Variants of the NEH Procedure for Flow Shop Scheduling Using Genetic Programming

José Antonio Vázquez-Rodríguez and Gabriela Ochoa

ASAP Research Group, School of Computer Science

University of Nottingham, Nottingham, U.K.

jav@cs.nott.ac.uk, gxo@cs.nott.ac.uk

February 19, 2010

Abstract

We use genetic programming to find variants of the well known Nawaz, Enscore and Ham (NEH) heuristic for the permutation flow shop problem. Each variant uses a different *ranking function* to prioritise operations during schedule construction. We have tested our ideas on problems where jobs have release times, due dates, and weights and have considered five objective functions: makespan, sum of tardiness, sum of weighted tardiness, sum of completion times and sum of weighted completion times. The implemented genetic programming system has been carefully tuned and used to generate one variant of NEH for each objective function. The new NEHs, obtained with genetic programming, have been compared with the original NEH and randomised NEH versions on a large set of benchmark problems. Our results indicate that the NEH variants discovered by genetic programming are superior to the original NEH and its stochastic version on most of the problems investigated.

keywords: Heuristics, Production

1 Introduction

This paper proposes an approach for the improvement of deterministic algorithms and the discovery of interesting algorithm variants. Typically, researchers in the combinatorial and heuristic optimisation community incorporate elements of randomness into one or more components of an algorithm in order to improve its performance. The

idea is that the repeated use of the randomised algorithm may lead to a better solution than the one found deterministically. Although this is often the case, and indeed very powerful meta-heuristics have been devised in this way, the price to pay in computational time is often excessive. It is common, for example, that such algorithms are incorporated into population based iterative procedures where the algorithm is invoked hundreds or thousands of times. We propose to use a search mechanism, genetic programming (Koza, 1992, Poli et al., 2008), to find a variant of the original algorithm that is more effective for a problem or is specialised on a reduced set of problem instances.

In the proposed idea, one of the components of the deterministic algorithm is replaced by an alternative compatible component discovered automatically by genetic programming. In contrast to randomised versions of an algorithm, the new algorithm found by genetic programming is still deterministic and, if it has been obtained after a large enough search, it may be more effective than the original one. Once an improved algorithm has been found, this is readily available to solve any instance of the problem in a time equal or very close to the one required by the original algorithm. Alternatively, genetic programming can be used to find algorithms that are tailored to a reduced set of the possible problem instances. This is of a particular relevance in practice given that companies usually require to solve the same type of instance repeatedly, each time with minor differences in input data.

It is important to mention that in this paper genetic programming solves a combinatorial problem which requires to find the best among a large set of possible heuristic components. Such methodologies, which search the space of other lower-level heuristics or heuristic components are known as hyper-heuristics (Cowling et al., 2000, Burke et al., 2003a). This paper, then, proposes a genetic programming-based hyper-heuristic for the discovery of improved algorithm variants.

The proposed hyper-heuristic was tested in the context of the permutation flow shop problem (Pinedo, 2002), with five different objective functions, and using the Nawaz, Enscore and Ham (NEH) procedure as the "base" algorithm (Nawaz et al., 1983). The NEH procedure consists of two main steps. In the first step, jobs are ranked and sorted according to a certain criterion that depends on the job parameters. We shall refer to this criterion as *ranking function*. The ranking of jobs is used in step two to build a schedule. We will discuss this procedure in detail in Section 3. The NEH procedure is a well established heuristic for the permutation flow shop problem in its most basic form, when all jobs are available at the same time and the objective to minimise is the makespan. The task of genetic programming is to find new ranking functions that lead to NEH variants that are superior to the original one on problems that consider extra constraints and objective functions that are inline with real world requirements.

Experimentation was carried out in two stages. In the first stage genetic programming was used to find five NEH variants, one for each objective function investigated.

In the second stage, these new algorithms were compared with the original NEH procedure, and a randomised version of NEH, on a large set of instances. The analysis of our results indicate that the ranking functions discovered by genetic programming lead to algorithms that are superior to the original NEH and its stochastic version on most of the problems investigated.

The rest of the paper is organised as follows. Section 2 describes the permutation flow shop problems that are investigated here. This section introduces the relevant notation to be used through out the rest of the paper. The NEH procedure is described in detail in Section 3. Section 4 is a literature review on hyper-heuristics including recent applications of genetic programming to the discovery or improvement of algorithms. Section 5 provides an introduction to genetic programming. It explains the overall functioning of the algorithm and the implementation details of our particular application. Section 6 describes the experimentation that was carried out in order to discover new NEH variants. These variants are evaluated and compared with the original NEH in Section 7. Section 8 is the discussion and Section 9 concludes the paper.

2 The Permutation Flow Shop Problem

The permutation flow shop problem requires to find the order in which n jobs are to be processed in m consecutive machines. The jobs are processed in the order machine 1, machine $2, \ldots,$ machine m. Machines can only process one job at a time and jobs can be processed by only one machine at a time. No job can jump over any other job, meaning that the order in which jobs are processed in machine 1 is maintained throughout the system. Moreover, no machine is allowed to remain idle when a job is ready for processing. Each job i requires a processing time on machine j denoted by p_{ij}. Each job i has also a release time r_i and a due date d_i which are the earliest time when job i can start in machine 1 and the time when the job is required to be completed, respectively. Job i has also a weight w_i which indicates its priority.

Given a permutation $\pi = \pi(1), \ldots, \pi(n)$, where $\pi(q)$ is the index of the job assigned in the q-th place, a unique schedule is obtained by calculating the starting and completion time of each job on each machine. The starting time $start_{\pi(q),j}$ of the q-th job on machine j is calculated as:

$$start_{\pi(q),j} = \max\{start_{\pi(q),j-1}, start_{\pi(q-1),j}\},$$

with

$$start_{\pi(0),j} = 0 \quad \text{and} \quad start_{\pi(q),0} = r_{\pi(q)},$$

and its completion time is calculated as:

$$C_{\pi(q),j} = start_{\pi(q)} + p_{\pi(q),j}.$$

Given a schedule, let C_i be the time when job i finishes its processing on machine m (last machine on the shop) and $T_i = \max\{0, C_i - d_i\}$ be the tardiness of job i. The permutation flow shop problem investigated here requires to find the processing order of n jobs in such a way that the resultant schedule minimises one of the five cost functions given in Table 1. Each objective function leads to a different variant of the problem, denoted, using the standard notation proposed in (Graham et al., 1979), as shown in the second column of Table 1.

[Table 1 about here.]

3 The NEH Procedure

Many heuristics have been proposed for the permutation flow shop problem, excellent reviews and comparisons are provided in (Framinan et al., 2004, Ruiz and Maroto, 2005). Despite the abundance of heuristic approaches, the NEH procedure (Nawaz et al., 1983), originally proposed for the special case where all jobs are available at time 0 and for the makespan objective (denoted $F_m|prmu|C_{\max}$), is the absolute favorite. The NEH heuristic is frequently used as the basis of more elaborated (meta-)heuristics to deal with different objective functions and flow shop scenarios (García and Maroto, 2006, Ruiz and Stützle, 2007, Framinan et al., 2003). Despite its simplicity, so far there is no evidence that confirms that the NEH procedure has been outperformed by any other heuristic for the $F_m|prmu|C_{\max}$ problem. The functioning of the NEH procedure is described in Figure 1, where $sumP_i = \sum_{j=1}^{m} p_{ij}$ is the sum of processing times of job i, i.e. the total processing time required by job i. In step 1, operations are sorted in decreasing order of their sum of processing times. In step 2, a schedule is built from scratch by assigning jobs in the order given by π. The first job, $\pi(1)$, is assigned to an empty schedule. Job $\pi(2)$ is then considered in places 1 and 2 in the new permutation and fixed to the place where it leads to the partial schedule with the smallest objective function value. Job $\pi(3)$ is considered in places 1, 2 and 3 and fixed to the place where it leads to the smallest cost function value, and so on. The complexity of the NEH heuristic, as proposed in (Nawaz et al., 1983), is $O(n^3m)$. For certain problems, this can be improved to $O(n^2m)$ by exploiting the graph representation of the problem, as explained in (Taillard, 1990).

[Figure 1 about here.]

We can see, by looking at Figure 1, that one can generate new NEH variants by modifying the way in which jobs are sorted in step 1 of the algorithm. One could, for example, talk of *NEH-random schedules* if the initial ordering is given by a random permutation. The alternative that we pursue here is to use genetic programming to find a function of the job parameters to be used to sort the operations. This is a natural approach given that genetic programming is particularly well suited for function approximation problems.

It is important to mention that NEH-random schedules are of a remarkably good quality. As an illustration, Figure 2 shows the distribution of the $C_{\max}$ value of ten thousand NEH-random schedules and ten thousand random schedules for a 20×10 (20 jobs and 10 machines) Taillard instance (Taillard, 1993). The mean of the random schedules is 2795.26 with a standard deviation of 172.06, whereas the mean of the NEH-random schedules is 1970.96 with a standard deviation of 45.59. The difference between both means is approximately 4.8 standard deviations. This means that an NEH-random schedule is, in terms of quality, roughly one in a million among the whole set of random schedules. The same exercise was repeated on the 120 Taillard instances and observed that, on average, the mean completion time of the NEH-random schedules is 6.25 standard deviations smaller than the mean completion time of the whole set of random schedules.

[Figure 2 about here.]

The sorting of jobs by decreasing order of sum of processing times (step 1) of NEH, seems simplistic but it is, by no means, naïve. Indeed, some authors have tested many other job ranking functions with no success and have conjectured that this ranking function is "optimum" for the $F_m|prmu|C_{\max}$ problem (Framinan et al., 2003, Kalczynski and Kamburowski, 2007). We have compared the schedules produced by the NEH procedure against a set of NEH-random schedules on the set of Taillard instances. On average, the NEH procedure generated schedules with a makespan that is 1.04 standard deviations smaller than the makespan of the average NEH-random schedule. Moreover, the NEH procedure generated a better than average NEH-random schedule in 99 out of 120 instances. These observations, however, only hold for the $F_m|prmu|C_{\max}$ problem and not for the more realistic problems investigated here. Of course, as mentioned in the introduction, one could trivially improve the NEH performance by calling the NEH procedure many times using different random permutations. We suggest a more intelligent approach: to substitute the ranking function used by NEH with alternative functions discovered by genetic programming.

4 Hyper-heuristics

As discussed above, the proposed approach can be considered as a genetic programming based hyper-heuristic to improve the performance of the NEH heuristic for the permutation flow shop problem. The term hyper-heuristic was first used in 1997 (Denzinger et al., 1997) to describe a protocol that combines several Artificial Intelligence methods in the context of automated theorem proving. It was independently used in 2000 (Cowling et al., 2000) to describe 'heuristics to choose heuristics' in the context of combinatorial optimisation. In this context, the first journal paper to use the term was (Burke et al., 2003b). The idea of automating the design of heuristics, however, is not new. It can be traced back to the early 1960s (Fisher and Thompson, 1961).

One of the main motivations behind hyper-heuristic research is to develop methodologies that can generalise and thus solve a class of problems, instead of a single problem or instance. The main distinguishing feature of these methods is that they operate on a search space of heuristics, rather than directly through a search space of problem solutions (Burke et al., 2003a, Ross, 2005). In (Burke et al., 2009) a unified classification and definition of hyper-heuristics is presented which captures the work that is being undertaken in this field. Two main hyper-heuristic categories are identified: heuristic *selection* and heuristic *generation*. In heuristic selection, the idea is to come up with automated methods for choosing or combining existing human designed (low-level) heuristics, taken from the literature, and with good performance in practice. On the other hand, in heuristic generation, the idea is to automatically produce new heuristics from a set of potential heuristic *components*.

Heuristic selection methodologies have been successfully applied to solve a wide range of real world problems including university timetabling (Burke et al., 2007b, 2006c,b), nurse rostering (Aickelin and Li, 2007), shelf space allocation (Bai et al., 2008, Bai and Kendall, 2005), cutting stock (Terashima-Marín et al., 2006), production scheduling (Petrovic et al., 2008, Rodríguez et al., 2007) and others (Cowling et al., 2002, Kendall and Mohamad, 2004, Ross et al., 2002).

In automated heuristic generation , genetic programming (introduced in the next section) is the most widely used methodology. It has been successfully applied to the automated generation of heuristics that solve hard combinatorial problems, such as boolean satisfiability, (Bader-El-Den and Poli, 2008, Kibria and Li, 2006, Fukunaga, 2008), bin packing (Burke et al., 2006a, 2007c), traveling salesman problem (Keller and Poli, 2007a,b) and production scheduling (Dimopoulos and Zalzala, 2001, Geiger et al., 2006, Tay and Ho, 2008). One approach to use genetic programming as a hyper-heuristic has been to evolve local search heuristics or even evolutionary algorithms (Fukunaga, 2002, 2004, 2008, Oltean, 2005, Poli et al., 2005, Oltean, 2003, Bader-El-Den and Poli, 2008). An alternative idea has been to use genetic programming to

evolve a function that is part of the processing of a given problem specific construction heuristic. The hyper-heuristic approach for bin-packing proposed in (Burke et al., 2006a, 2007c), for instance, evolves a function that receives as parameters the size of the items to be packed and the bin capacities. Whenever a new item has to be placed, the evolved function assigns a score to each of the bins and the item is located in the bin with the lowest score. If no bin receives a score greater or equal to a certain value, then a new bin is open. A related approach in production scheduling (Dimopoulos and Zalzala, 1999, 2001, Geiger et al., 2006, Tay and Ho, 2008) uses genetic programming for learning a function of the job parameters, such as processing times and release dates, as well as some floor shop parameters such as the release time of machines, etc. In this way genetic programming is used to evolve dispatching rules that are used to schedule the floor shop. The proposed approach, discussed in detail below, belongs to this last category of hyper-heuristic methods.

5 Genetic programming, overview and implementation details

Genetic programming is a paradigm from evolutionary computation concerned with the automatic generation of computer programs (Koza, 1992). It is a problem-independent methodology that genetically breeds a population of computer programs to solve a problem. Genetic programming typically starts from a randomly generated set (*population*) of computer programs (*individuals*) composed of the available algorithmic components (given by the human designer). The methodology iteratively transform the population of computer programs into a new generation by applying analogs of naturally occurring genetic operations (such as mutation and recombination). A fitness function assigns a value to each individual depending on its performance on the problem. The individuals are probabilistically selected to participate in the genetic operations based on their fitness. This process is repeated until a termination condition is satisfied. Genetic programming is, therefore, a method of generating syntactically valid programs, according to some predefined grammar, and a fitness function is used to decide which programs are better suited to the task at hand.

A distinction can often be drawn between an optimisation and a learning (or modeling) task. In the former, we seek the highest quality solution with respect to some evaluation function. An example is the minimisation of a non-linear function, where we seek a value of x such that $f(x)$ is a minimum. In the latter, which may be seen as a special case of the former, we seek a representation or model which better adjusts to the validation data. For example, in symbolic function regression we seek a representation of a function $f(x)$ that best matches a set of fitness cases. In this paper, the task is

to find a function $f(\vec{i}, \vec{k}) \mapsto \mathbb{R}$ of the job i and problem instance k parameters in such a way that when the jobs are ordered according to $f(\vec{i}, \vec{k})$, the resulting permutation leads NEH to produce high quality schedules. This is an optimisation task.

Genetic programming starts from a high-level statement of the problem requirements and attempts to generate a computer program that solves it. The human designer communicates this statement by specifying the following major components:

1. the individual representation
2. the fitness measure
3. the set of terminals (i.e. the independent problem variables, constants and zero argument functions)
4. the set of operators
5. the evolutionary control parameters (such as population size and genetic operator's rates)
6. the termination criterion (generally, a predefined maximum number of generations).

The rest of this section describes each of these components in general terms and in greater detail for our specific application. There are numerous tutorials, introductory articles and text books on genetic programming. The series of books by Koza (Koza, 1992, 1994, Koza et al., 1999, 2003) and the book by Banzhaf et al. (Banzhaf et al., 1998), treat the subject thoroughly. Also, (Koza and Poli, 2005) and (Poli et al., 2008) are more recent introductory texts. Introductory articles can also be found in most current textbooks on machine learning and evolutionary computation.

5.1 Individual representation and fitness evaluation

In genetic programming, the programs that comprise the population are traditionally represented as *syntax trees* rather than lines of code. Other program structures can be evolved, such as linear sequences of instructions, and grammars. In this paper we use the tree-based representation. Consider the simple arithmetic expression discussed in (Koza and Poli, 2005):

$$max(x \times x,\ x + 3 \times y).$$

This is represented as a tree in Figure 3. The tree contains *nodes*, which indicate instructions to execute, and *links*, which indicate the arguments for each instruction. In genetic programming terminology, the internal nodes in a tree are called *operators*, while the leaves are called *terminals*. Interpreting a program tree means executing the

nodes in the tree in an order that guarantees that nodes are not executed before the value of their arguments (if any) is known. This is usually done by traversing the tree in a recursive way starting from the root node, and postponing the evaluation of each node until the value of its children (arguments) is known.

[Figure 3 about here.]

Irrespective of the execution strategy adopted, the fitness of a program may be measured in many different ways, including, for instance, in terms of the amount of error between its output and the desired output, the amount of time (or other resources) required to bring a system to a desired target state, the accuracy of the program in pattern recognition or classification, etc. For many problems, each program in the population is executed over a representative sample of different *fitness cases*. These cases may represent different values of the program's input(s), different conditions of a system or different environments.

In this paper, a tree or individual Ind is interpreted as a ranking function $Ind(\vec{i}, \vec{k}) \mapsto \mathbb{R}$, where $\vec{i}$ is a vector that refers to the data specific to job i and $\vec{k}$ is a vector that refers to the data specific to a problem instance k. Given an instance k an individual can be used as a ranking function for the NEH procedure by following the steps 1 to 4 described in Figure 4, where $Rank_i = Ind(\vec{i}, \vec{k})$ is a real value assigned by the individual being evaluated to job i on instance k. The fitness function of an individual is evaluated not only on one but on a set of instances. Let f_k refer to the function value obtained by an individual on instance k and let f_k^* be the best of one million NEH-random schedules that were generated a priori for the same instance. In order to be able to aggregate on a single fitness measure the results of an individual on different instances, the percentage deviation (PD) of the f_k value obtained by an individual was calculated as

$$PD_k = \frac{f_k - f_k^*}{f_k^*} \times 100. \quad (1)$$

The fitness of an individual is the sum of the PD values on the N instances in the fitness case set: $\sum_{k=1}^{N} PD_k$. Notice that PD_k may be a negative quantity. This has no harmful effect given that the sum of PD_k values is to be minimised. The process to evaluate an individual on a set of fitness cases is given in algorithmic form in Figure 4.

[Figure 4 about here.]

5.2 Terminals and operators

The function and terminal sets conform the ingredients that are available to create the computer programs. Identifying these sets for a particular problem (or class of problems) is usually a straightforward process. For some problems, the function set may

simply consist of the arithmetic functions of addition, substraction, multiplication, and division as well as a conditional branching operator. The terminal set may consist of the program's external inputs (independent variables) and numerical constants. For other problems, the ingredients include specialised functions and terminals. For example if the goal is to automatically control a robot to mop the floor of an obstacle-laden room (Koza and Poli, 2005), the function set needs to specify what the robot is capable of doing. For example, the robot may be capable of executing functions such as moving, turning and swishing the mop. The terminal set may provide sensory information, such as how far an obstacle is from the robot.

In this work the terminal set consists of job specific data, including: the release time, r_i, the due date, d_i and the weight, w_i, of job i, as well as the following four indicators:

- the sum of processing times of job i:

$$sumP_i = \sum_{j=1}^{m} p_{ij}$$

- the weighted sum of processing times of job i:

$$wSumP_i = \sum_{j=1}^{m} (m - j + 1) p_{ij}$$

- the sum of the absolute differences of a job i's processing times with respect to the other jobs:

$$sumAbsDif_i = \sum_{j=1}^{m} \sum_{\substack{\hat{i}=1 \\ \hat{i} \neq i}}^{n} |p_{ij} - p_{\hat{i}j}|$$

- the weighted sum of the absolute differences of a job i's processing times with respect to other jobs:

$$wSumAbsDif_i = \sum_{j=1}^{m} \sum_{\substack{\hat{i}=1 \\ \hat{i} \neq i}}^{n} (m - i + 1) |p_{ij} - p_{\hat{i}j}|.$$

The set of constants consists of the number of jobs, n, the number of machines, m, and the integer numbers $1, 2, \ldots, 10$.

The operators set consists of the common arithmetic operators: $+, -, \times, \div$ and two specialised monadic operators, the sum of the processing times of job i from machine

a to machine m (last machine)

$$FSumP_i(a) = \sum_{j=a}^{m} p_{ij}$$

where $a \in \{1, 2, \dots, m\}$, and the sum of the processing times of job i from machine 1 to machine a

$$BSumP_i(a) = \sum_{j=a}^{1} p_{ij}.$$

Following genetic programming convention, the $\div$ operator returns a very large penalty number when attempting to divide by zero. The $FSumP_i(a)$ and $BSumP_i(a)$ operators do also require protection against a values that are not in $\{1, 2, \dots, m\}$. In both cases, rather than penalising for illegal values, the input value is repaired by first transforming a into an integer using the floor operator, and second, by setting a to 1 or m if the resultant integer is less than 1 or larger than m, respectively.

5.3 Genetic operators and reproduction

The implemented genetic programming algorithm is the standard well established design described in (Koza, 1992) and illustrated in Figure 5. In this, the population is initialised using the *ramped half-and-half* method. In this method, a maximum initial depth for trees (D_{max}) is selected, and each member of the initial population is randomly generated from the set of functions and terminals using, with equal probability, one of the two following methods:

- *Full method*: each branch of the tree has depth D_{max}. The content of nodes at depth d are chosen from the function set if $d < D_{max}$ or from the terminal set if $d = D_{max}$.
- *Growth method*: the branches of the tree may have different depths (up to the limit D_{max}). The tree is generated starting from the root, with the content of each node selected probabilistically from the union of function and terminal sets (if $d < D_{max}$).

[Figure 5 about here.]

At every iteration (*generation*) a whole new population of individuals is created and evaluated. This is done by selecting parents from the current population and modifying them using genetic operators in order to produce new solutions. Individuals are selected to participate as parents in the new population using a *K tournament selection* mechanism, which gives higher chances of being selected to those individuals that are the fittest. In tournament selection, K parents are randomly selected and only

the fittest participates as a parent in the new population. Parameter K, which has to be tuned, controls to some extent the speed of convergence of the algorithm.

Creating a new individual is a two step process. First an individual is generated with *crossover* with a probability X_p or with direct *reproduction* with probability $1 - X_p$. The crossover operator receives two parents, selects randomly one subtree from each of them, and swaps the sub-trees between the two parents creating two new individuals. In direct reproduction, the parent is copied directly from the current population. In the second step the new individual is mutated with a certain probability μ_p, in which case a randomly chosen subtree of the individual is replaced with a new random subtree.

In many genetic programming applications there is a naturally occurring ideal state that determines the *stopping condition* of the algorithm. In these cases, it is often easy to measure how close is a given solution to the desired one. Once a good enough solution has been found, the algorithm stops. This is the case, for example, of symbolic regression applications, where the algorithm stops once a solution that delivers a close to zero sum of errors is found. Of course, the algorithm may never find such high quality solutions and is customary to stop the algorithm after it reaches a certain number of solution evaluations or iterations (*generations*). In our application, it is not easy to determine such an ideal state since this would require knowing a priori the solution of the problems that we are ultimately trying to solve. Therefore we only use the second type of stopping condition and stop the algorithm after a certain number, *iter*, of iterations or solution evaluations has been reached.

A major concern when implementing genetic programming algorithms is the control of the tree sizes. Without a controlling mechanism, the standard genetic and selection operators lead to ever increasing trees. This phenomenon is known as *bloat*. In our implementation, the crossover operator was modified in order to only return trees that are equal or smaller to a certain size, where size is the number of nodes in the tree. Notice that after applying the crossover operator, if one of the new individuals is greater than the permitted size then the other will always be equal or smaller. A second implementation detail is whether or not the algorithm keeps the best solution found so far in the population at all times. If this is true, as in our implementation, the algorithm is said to be *elitist*.

6 Discovering new NEH variants with genetic programming

6.1 Training set

For each problem investigated (each objective function), genetic programming was run on a randomly generated set of instances of sizes given by each $n \in \{10, 20, 30, 40, 50\}$

and $m \in \{4, 5, 10, 20\}$ combination, giving a total of $5 \times 4 = 20$ instances. The reason for keeping the instances relatively small is to allow genetic programming to run for a large enough number of generations within a reasonable amount of time. As it is common practice, the processing times, job weights and release times of jobs are random integers within the ranges $[1, \ldots, 99]$, $[1, \ldots, 10]$ and $[0, 1.5 \cdot \sum_{i=1}^{n} P_{i1}]$, respectively. The due dates were generated as proposed in (Ruiz-Torres and Centeno, 2008): as random integers in the range $[0.3W, 0.7W]$, where

$$W = \frac{1}{n}\left(\sum_i r_i + \sum_{i=1}^{n}\sum_{j=1}^{m} P_{ij} + (n-1)\max_j \sum_{i=1}^{n} P_{ij}\right).$$

Factor W controls the difficulty to meet a particular due date; due dates between $0.3W$ and $0.5W$ are considered tight, i.e. difficult to meet, whereas due dates between $0.5W$ and $0.7W$ are loose. In all cases, data was drawn from a discrete uniform distribution, which is known to lead to difficult instances due to its large variance.

6.2 Parameter Tuning

Most of the genetic programming parameter values were decided after a two stages *full factorial design* experimentation (Montgomery, 2005) in which the crossover probability, X_p, the mutation probability μ_p, the population size, *pop*, and the tournament size, K, were allowed to vary at 3 equally spaced levels each, giving a total of $3^4 = 81$ experiments. The best combination of parameters is the one that lead to the individual with the best average function value on the training set, after 10 runs. The parameter values of such best combination were used as the central values for a second full factorial experimentation. The two extreme values were set at half the distance used on the first factorial experiment from the central value. A similar third full factorial experimentation was carried out but without further improvement. The stopping condition was set to 100 000 solution evaluations, which means that the number of generations depends on the population size. The design of the implemented genetic programming system and the final values of the pertinent parameters are summarised in Table 2. More on design of experiments and its application to the tuning and comparison of heuristics can be found in (Bartz-Beielstein, 2006).

[Table 2 about here.]

6.3 Training results

The genetic programming system described in Table 2 was run on the training set for each of the problems described in Section 2. The genetic programming implementation was done in Java and all experiments, including those for parameter tuning, were

carried out on a Cluster of identical dual Opteron 248 (2.2GHz) processors, running Unix, part of the High Performance Computing of the University of Nottingham, UK.

The best ranking function for each problem, found after 5 training runs, are given in Table 3. We can see that genetic programming found a generalisation of the ranking function used by the original NEH to deal with the release times in the problems studied here. In fact, the expression found for the $F_m|r_i, prmu|C_{\max}$ problem can be reduced to $r_j \times sumP_{ij}$. This is because the argument for the $FSumP_i$ operator is, on all problems investigated, including those used later to evaluate the ranking functions, a negative number. Recall that negative numbers are repaired to 1 by the operator and we know that $FSumP_i(1) = sumP_{ij}$. The ranking function found by genetic programming for the second problem gives high priority to jobs with large processing times, particularly in the last stages of the shop floor. For the third problem, genetic programming found a ranking function that gives preference to jobs with long processing times and large weights. The ranking functions found for the last two problems assign a certain priority to the sum of the processing times of jobs, the release and due dates and the weights. However, the relation between such variables and the rank of jobs is not intuitive. We note, nonetheless, that the capability of genetic programming for finding such non intuitive solutions is partly responsible for its success in many domains.

[Table 3 about here.]

7 Evaluation of Ranking Functions

The best solution found on the training set was incorporated into the NEH algorithm and used to solve variants of the 120 Taillard instances (Taillard, 1993) which were modified to match the problems investigated here. The original Taillard instances are given in 12 sizes (10 instances of each size): the nine combinations of $n \in \{20, 50, 100\}$ and $m \in \{5, 10, 20\}$ and the sizes 200×10, 200×20 and 500×20. The processing times are uniformly distributed integers in $\{1, \ldots, 99\}$. Extra data was added to each job in order to generate new instances: a weight, a release time and a due date; all generated in the same way as the training set described in Section 6.1.

The competing algorithms are three variants of the NEH procedure; each distinguishing from the others on the method used to prioritise operations in step 1. The first competitor, NEH_{rand}, uses a random permutation to prioritise operations. Results reported for the NEH_{rand} algorithm are the minimum on 100 runs. The second competitor, NEH_{GP}, uses the ranking function proposed by genetic programming and the third competitor, NEH, is the original algorithm, which ranks jobs in decreasing order of $sumP_i$ values. Notice that, except NEH_{rand}, the algorithms are deterministic and were run only once on each problem instance.

A visual summary of the results is given in Figure 6, which presents the box plots of the results obtained by the NEH variants on each problem size and for each objective function. In all plots, the $x-$axis represents the NEH variant, and the $y-$axis represents the corresponding objective function value normalised to the $[0, 1]$ range. In 55 out of 60 cases NEH$_{GP}$ is either clearly better or potentially better than the other competitors. These are the cases where the median of NEH$_{GP}$ is smaller to that of the other algorithms or is equal or very close, but the lower end of the box is lower to that of the competitors. In 41 of these cases, the box of NEH$_{GP}$ does not overlap with the box of any of the other algorithms; it is safe in these cases to conclude that the difference in performance of the algorithms is statistically significant. For the remaining 19 cases, where the boxes overlap, we carried out the adequate $t-$test pair comparisons between NEH$_{GP}$ and the other two algorithms. The results are given in Table 4. In there, the first three columns indicate the group of instances being analysed, the third and fourth columns present the comparison tests between NEH$_{GP}$ and NEH$_{rand}$ and the fifth and sixth columns are the comparisons between NEH$_{GP}$ and NEH. We can see that NEH$_{GP}$ was superior to NEH in 15 cases, non-distinguishable in 3 cases, and inferior in only one case. When compared to NEH$_{rand}$, NEH$_{GP}$ was superior in 8 cases, non-distinguishable in 10 cases, and inferior in 1 case. Summarising, NEH$_{GP}$ had a statistically significant superiority to the other two algorithms in 49 out of 60 cases. It was superior to one algorithm but non distinguishable from the other in 7 cases. It was non-distinguishable from both algorithms in 2 cases. Inferior to one algorithm but non-distinguishable from the second one in 2 cases. In no instance, NEH$_{GP}$ was significantly inferior to both of the competitors.

[Figure 6 about here.]

[Table 4 about here.]

Results were also summarised by taking the Percentage Deviation (PD) from the best solution found. For each instance k, let f_k^* be the best of the 102 solutions found by all NEH variants (100 by NEH$_{rand}$, one by NEH$_{GP}$ and one by NEH). The PD value was calculated using Equation 1 for each instance and each algorithm. Note that in this occasion, subindex $k = 1, \ldots, 120$, refers to one of the modified Taillard instances used in this section. The Average PD values on the instances grouped according to their size and for each algorithm are given in Table 5, where the best results are given in bold. Again, NEH$_{GP}$ obtained the best results on all, except one, groups of instances. Remarkably, NEH$_{GP}$ very frequently obtained an Average PD value of 0.0, meaning that for that particular group, NEH$_{GP}$ found the best solution to all instances. The original NEH obtained the worst results overall, very close to those found by NEH$_{rand}$.

[Table 5 about here.]

The Friedman non-parametric test was used to validate our results statistically on large groups of instances. The Friedman test acts on the ranks of the results obtained by the algorithms, giving smaller ranks to small objective function values. By taking the ranks, the test allows to block the effect due to instance characteristics and to obtain conclusions that are valid not only on one instance or type of instance but on a wide range of instances. The ranks achieved by each algorithm are added up and the test evaluates whether the independent controllable variable, in this case the algorithm type, has a significant effect on the sum of the ranks. The null hypothesis is that there is no significant effect; if this is rejected, we conclude that one or more algorithms are superior to one or more of the other algorithms. In order to obtain precise conclusions, the test was performed on two algorithms at a time. If the test rejects the null hypothesis, i.e. if a $P-$value ≤ 0.05 is obtained, then we conclude in favor of the algorithm with the smallest sum of ranks. The results of the test on the whole set of instances, and on the instances grouped by objective function are given in Table 6. In there, the first column indicates the group of instances being analysed, the next columns indicate the sum of the ranks obtained by the algorithms being compared and the $P-$value of the corresponding tests. In all tests, NEH$_{GP}$ is demonstrated to be superior to the alternative algorithm. In most tests involving NEH$_{rand}$ and NEH, the former is superior to the latter. The exception is in the C$_{max}$ objective where both algorithms are considered equivalent. Similar tests were carried out grouping instances by size. We do not provide detailed information given that the results were the same; NEH$_{GP}$ is superior to the other two in all cases, and NEH$_{rand}$ is equivalent to NEH in two cases and superior in the rest.

[Table 6 about here.]

8 Discussion

One could easily argue that the fact that NEH$_{GP}$ outperformed the original NEH on the investigated instances does not necessarily confirm that the ranking functions discovered by genetic programming are any "good". It probably indicates that the original NEH ranking function is "bad" on the investigated problems. We claim that the ranking functions found by genetic programming are as successful or more successful for the investigated problems as it is the original ranking function for the $F_m|prmu|C_{\max}$ problem. In order to sustain this claim we use as a reference point the performance of NEH$_{rand}$, i.e. we evaluate the performance of NEH$_{GP}$ in terms of the number of standard deviations that its results are shifted from the mean of those obtained by NEH$_{rand}$. This was done with the purpose of isolating the effect due to the ranking function from the rest of the algorithm. The use of deviations from a best known

solution, as it is commonly done, is only an indicator of the performance of an algorithm as a whole and is not appropriate for our purpose.

We measured, for each instance, the number of standard deviations of the solution found by NEH$_{GP}$ from the mean of the 100 solutions found by NEH$_{rand}$. The average of the deviations obtained by NEH$_{GP}$ and NEH on each group of instances are given in Table 7. A negative value means that the algorithm in the corresponding column found better solutions on average than NEH$_{rand}$. For example, for the ten problems of size 500×20 and the $C_{\max}$ objective, NEH$_{GP}$ found solutions that are 23.87 standard deviations smaller than the average solution found by NEH$_{rand}$. This result can be interpreted as that one would require to generate a very large number (billions or probably more) of solutions with NEH$_{rand}$ in order to find one that is as good as the average solution found by NEH$_{GP}$. Except for two groups of instances, NEH$_{GP}$ obtained better results (a negative deviation) than the average NEH$_{rand}$. NEH, on the other hand, consistently obtained poor results.

Recall that for the $F_m|prmu|C_{\max}$ problem, NEH obtained solutions that were, on average -1.04 standard deviations from the solutions found by NEH$_{rand}$. We can claim, therefore, that NEH$_{GP}$ found three very successful ranking functions with deviations of -7.1, -3.73 and -3.77 for the C_{max}, $sumWC$ and $sumWT$ objective, respectively, and two which are also good, with deviations of -0.61 and -0.60 for the $sumC$ and $sumT$ objectives, respectively, but that are not as successful as the original NEH ranking function is for the $F_m|prmu|C_{\max}$ problem.

[Table 7 about here.]

8.1 Scalability

A property of the newly found ranking functions is that their performance remains as good or it even improves as the problem escalates in size. The results of NEH$_{GP}$ were remarkably good, compared to the other algorithms, for the medium and large instances. This is surprising if one considers that the training set contains only relatively small instances. This behaviour has been previously observed and thoroughly analysed in (Burke et al., 2007a) for evolved heuristics for online bin packing. The authors concluded that the genetic programming-evolved heuristics, which were obtained after training on a set of small instances, appeared to show a certain *look-ahead* behaviour that was particularly useful on large instances. The question that remained unanswered, however, is why heuristics that were evolved on small instances acquired such features that allowed them to escalate well, specially as it was conceded that the heuristics do not have explicit access to state information. In this paper, we believe that what originated this situation is the fact that heuristics were evolved on a training set with instances of varying sizes, ranging from very-small to small-medium. In order

to be successful, an individual had to perform well on a variety of instance sizes. Such ability extrapolated to the largest cases. In the work of (Burke et al., 2007a), however, heuristics were obtained after training on instances of one size only. The heuristic that escalated the best was obtained after training on instances of size 500, and tested later on instances of size 100 000. This is probably because an instance of size 500 is already large enough to benefit from the look-ahead behaviour and hence the evolution process lead to its discovery.

8.2 Algorithm Tailoring

Our genetic programming algorithm was trained on a set of instances of varying sizes hoping that it would be able to find a ranking function that performs well on a variety of instances. Genetic programming succeeded overall, but for two of the objective functions, namely $sumT$ and $sumC$, the superiority of NEH_{GP} over its competitors is more modest than for the rest of the objective functions. On a real world scenario, one may be able to specialise on a particular type of problem: fixed size, data distribution, a specific objective function and may be able to train genetic programming for a longer period of time. Instances that consider objective functions $sumT$ and $sumC$ are the best candidates for specialisation at the problem size level. One would expect, however, that such over-specialisation would compromise the performance of NEH on other types of instances. Our point is that genetic programming can be easily used as a tool for algorithm tailoring.

As an illustration of how algorithm tailoring could work, we used genetic programming to find a ranking function specialised on problems of size 20×20 with the $sumC$ objective. One can see in Table 7 that it is on these instances where NEH_{GP} performed the poorest. We trained genetic programming on a set of 20 randomly generated instances of size 20×20 with the rest of the data generated as in Section 6.1. We allowed genetic programming to run for 100 000 evaluations using the genetic programming specifications given in Table 2. The best ranking function, after 5 genetic programming runs, was incorporated to NEH and used to solve the 10 test problems of interest. The ranking function lead to a deviation on the $sumC$ 20×20 instances of -2.19, which is considerable better than the original 0.021. The same ranking function obtained, overall, a deviation of -0.51 standard deviations, which is slightly worse than the -0.61 previously found; we payed the price for the specialisation on instances of size 20×20, with a slightly worse performance on the rest of the instances. This observation agrees with the conclusions of (Burke et al., 2007c), where it was found that heuristics for bin packing could be evolved to be specialists in one sub-problem, at the expense of their performance in other sub-problems. There is a trade-off between performance and generalisation.

8.3 Results on the $F_m|prmu|C_{\max}$ problem

Researches have tried unsuccessfully to find better ranking functions than that of the original NEH procedure for the $F_m|prmu|C_{\max}$ problem. In (Framinan et al., 2003), 176 different ranking functions, which ranged from the very simple to the very sophisticated, were tested and none lead to an improvement. In (Kalczynski and Kamburowski, 2007), the authors repeated some of these experiments and obtained the same results. Here, a much difficult test was put on the ranking function of NEH; literally hundreds of thousands of new ranking functions were compared. Genetic programming, as one would expect, very frequently rediscovered the original NEH ranking function. This was a problem since the algorithm got very easily stuck on such a solution. In our attempt at improving on such a ranking function, we penalised ranking functions that were equivalent to $sumP_{ij}$. After 10 runs, using the specifications in Table 2, we found a ranking function that lead to the results given in Table 8, where NEH$_{GP}$ is compared with NEH. We can see that NEH$_{GP}$ obtained better results on the small instances, both algorithms had a similar performance on the medium instances and NEH was best on the large instances. Overall, NEH obtained better average results than NEH$_{GP}$ and obtained a better solution than NEH$_{GP}$ on 59 out of 120 instances, whereas NEH$_{GP}$ obtained the best solution on 56 instances. The results, in all cases, are fairly close, and indeed their difference is statistically non significant. Our results agree with those in (Framinan et al., 2003, Kalczynski and Kamburowski, 2007), where the authors observed that the ranking function of NEH is indeed very good for the $F_m|prmu|C_{\max}$ problem.

[Table 8 about here.]

9 Summary and Conclusion

This paper proposes a genetic programming based hyper-heuristic to improve the performance of the NEH heuristic for the permutation flow shop problem with release times, due dates, job weights and five objective functions (one at a time). The NEH heuristic is known to be highly influenced by a ranking function that it uses to prioritise operations previous to the construction of a schedule. The proposed hyper-heuristic aims at discovering ranking functions which lead the NEH procedure towards good solutions. Experimentation was carried out in two stages (1) a training stage in which five ranking functions (one for each objective function investigated) where obtained by genetic programming and (2) an evaluation stage in which NEH with the newly discovered ranking functions (NEH$_{GP}$) was used to solve modified versions of the 120 Taillard instances (Taillard, 1993).

The improved algorithms obtained very good results on all the problems investigated. There were problems, where, for instance, the average function value obtained by NEH_{GP} was 3 standard deviations smaller than the mean of the results obtained by a randomised NEH. For the makespan objective, the difference was of more than 7 standard deviations, and for the instances of size 500×20 the difference was of 23 standard deviations. As a comparison point, when NEH is used to solve the problem for which it was originally designed, it obtains results that are 1.04 standard deviations smaller than those obtained by the randomised NEH. Even though 1.04 does not seem impressive, neither (Framinan et al., 2003), or ourselves, after testing millions of ranking functions, could improve on it.

We have discussed the reasons of why the NEH_{GP} heuristics escalate well with problem size. We attribute this behaviour to the fact that the training set contains instances of varying sizes. Which means that an individual, in order to be successful, has to be able to escalate well on a range of instance sizes. We have also suggested that genetic programming can be used as an algorithm tailoring tool, and have provided an example of how genetic programming can be used to specialise an algorithm to a particular type of instance at the price of a decrease in performance on the rest of the instances.

Our observations on the scalability of the proposed heuristics motivates interesting paths for future research. In general terms, it is interesting to know how the selection of the training set affects the properties of the evolved heuristics. Moreover, and closely related, we would like to know how to assign a fitness value to an individual based on its performance on a set of problem instances. In this paper, the aggregated deviations from previously known good solutions worked well. However, this approach may be difficult to execute in other circumstances, e.g. when good solutions are difficult to know a priori. Moreover, one may desire heuristics with different properties, e.g. that are stable, obtain good solutions on average, or that escalate well, among others. Of course, it is also natural, as future research, to test the proposed idea on other problem domains.

References

U. Aickelin and J. Li. An estimation of distribution algorithm for nurse scheduling. *Annals of Operations Research*, 155(1):289–309, 2007.

M. Bader-El-Den and R. Poli. Generating sat local-search heuristics using a gp hyperheuristic framework. In *Proceedings of Evolution Artificielle*, volume 4926 of *Lecture Notes in Computer Science*, pages 37–49. Springer-Verlag, 2008.

R. Bai and G. Kendall. An investigation of automated planograms using a simulated annealing based hyper-heuristic. In T. Ibaraki, K. Nonobe, and M. Yagiura, editors, *Metaheuristics: Progress as Real Problem Solvers - (Operations Research/Computer Science Interfaces Series, Vol. 32)*, pages 87–108, Berlin, Heidelberg, New York, 2005. Springer.

R. Bai, E. K. Burke, and G. Kendall. Heuristic,meta-heuristic and hyper-heuristic approaches for

fresh produce inventory control and shelf space allocation. *Journal of the Operational Research Society*, 59:1387–1397, 2008.

W. Banzhaf, P. Nordin, R. Keller, and F. Francone, editors. *Genetic Programming - An Introduction*. Morgan Kaufmann, San Francisco, CA, 1998.

T. Bartz-Beielstein. *Experimental Research in Evolutionary Computation*. Natural Computing Series. Springer, 2006.

E. Burke, E. Hart, G. Kendall, J. Newall, P. Ross, and S. Schulenburg. Hyper-heuristics: An emerging direction in modern search technology. In Fred Glover and Gary Kochenberger, editors, *Handbook of Metaheuristics*, pages 457–474. Springer, 2003a.

E. K. Burke, G. Kendall, and E. Soubeiga. A tabu-search hyper-heuristic for timetabling and rostering. *Journal of Heuristics*, 9:451–470, 2003b.

E. K. Burke, M. Hyde, and G. Kendall. Evolving bin packing heuristics with genetic programming. In *Proceedings of the 9th International Conference on Parallel Problem Solving from Nature (PPSN 2006),*, volume 4193 of *Lecture Notes in Computer Science*, pages 860–869, 2006a.

E. K. Burke, B. L. MacCarthy, S. Petrovic, and R. Qu. Multiple-retrieval case based reasoning for course timetabling problems. *Journal of the Operational Research Society*, 57(2):148–162, 2006b.

E. K. Burke, S. Petrovic, and R. Qu. Case-based heuristic selection for timetabling problems. *Journal of Scheduling*, 9(2):115–132, 2006c.

E. K. Burke, M. R. Hyde, G. Kendall, and J. R. Woodward. The scalability of evolved on line bin packing heuristics. In Dipti Srinivasan and Lipo Wang, editors, *2007 IEEE Congress on Evolutionary Computation*, pages 2530–2537, Singapore, 25-28 September 2007a. IEEE Computational Intelligence Society, IEEE Press. ISBN 1-4244-1340-0.

E. K. Burke, B. McCollum, A. Meisels, S. Petrovic, and R. Qu. A graph-based hyper-heuristic for educational timetabling problems. *European Journal of Operational Research*, 2007b.

E. K. Burke, M. Hyde, G. Kendall, G. Ochoa, E. Ozcan, and J. Woodward. *Handbook of Metaheuristics*, chapter A Classification of Hyper-heuristic Approaches. International Series in Operations Research & Management Science. Springer, 2009.

K. Burke, M. R. Hyde, G. Kendall, and J. Woodward. Automatic heuristic generation with genetic programming: evolving a jack-of-all-trades or a master of one. In *Proceedings of the 9th annual conference on Genetic and evolutionary computation*, volume 2, pages 1559–1565, 2007c.

P. Cowling, G. Kendall, and E. Soubeiga. A hyperheuristic approach to scheduling a sales summit. In E. K. Burke and W. Erben, editors, *LNCS 2079, Practice and Theory of Automated Timetabling III : Third International Conference, PATAT 2000*, pages 176–190. Springer-Verlag, 2000.

P. Cowling, G. Kendall, and L. Han. An investigation of a hyperheuristic genetic algorithm applied to a trainer scheduling problem. In *Proceedings of Congress on Evolutionary Computation (CEC2002)*, pages 1185–1190. IEEE, 2002.

J. Denzinger, M. Fuchs, and M. Fuchs. High performance ATP systems by combining several ai methods. In *Proceedings of the Fifteenth International Joint Conference on Artificial Intelligence (IJCAI '97)*, pages 102–107, 1997.

C. Dimopoulos and A. M. S. Zalzala. Investigating the use of genetic programming for a classic one-machine scheduling problem. *Advances in Engineering Software*, 32(6):489–498, 2001.

C. Dimopoulos and A. MS Zalzala. A genetic programming heuristic for the one-machine total tardiness problem. In *Proceedings of the 1999 Congress on Evolutionary Computation (CEC '99)*, pages 2207–2214, 1999.

H. Fisher and G. L. Thompson. Probabilistic learning combinations of local job-shop scheduling rules. In *In Factory Scheduling Conference*, pages 225–251, Carnegie Institute of Technology, 1961.

J. M. Framinan, R. Leisten, and C. Rajendran. Different initial sequences for the heuristic of nawaz, enscore and ham to minimize makespan, idletime or flowtime in the static permutation flowshop sequencing problem. *International Journal of Production Research*, 41:121–148, 2003.

J. M. Framinan, J. N. D. Gupta, and R. Leisten. A review and classification of heuristics for permutation flow-shop scheduling with makespan objective. *Journal of the Operational Research Society*, 55:1243–1255, 2004.

A. Fukunaga. Automated discovery of composite SAT variable selection heuristics. In *Proceedings of the National Conference on Artificial Intelligence (AAAI)*, pages 641–648, 2002.

A. S. Fukunaga. Evolving local search heuristics for SAT using genetic programming. In *Genetic and Evolutionary Computation – GECCO-2004, Part II*, Lecture Notes in Computer Science, pages 483–494. Springer-Verlag, 2004.

A. S. Fukunaga. Automated discovery of local search heuristics for satisfiability testing. *Evolutionary*

Computation, 16(1):31–61, 2008. ISSN 1063-6560.

R. R. García and C. Maroto. A genetic algorithm for hybrid flow shops with sequence dependent setup times and machine elegibility. *European Journal of Operational Research*, 169:781–800, 2006.

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

R. L. Graham, E. L. Lawler, J. K. Lenstra, and A. H. G. Rinnooy Kan. Optimization and approximation in deterministic sequence and scheduling: a survey. *Annals of Discrete Mathematics*, 5: 287–326, 1979.

P. J. Kalczynski and J. Kamburowski. On thenehheuristic for minimizing the makespan in permutation flowshops. *OMEGA-International Journal of Management Science*, 35:53–60, 2007.

R. E. Keller and R. Poli. Linear genetic programming of parsimonious metaheuristics. In *Proceedings of IEEE Congress on Evolutionary Computation (CEC 2007)*, pages 4508–4515, 2007a.

R. E. Keller and R. Poli. Cost-benefit investigation of a genetic-programming hyperheuristic. In *Proceedings of Evolution Artificielle*, volume 4926 of *Lecture Notes in Computer Science*, pages 13–24, 2007b.

G. Kendall and M. Mohamad. Channel assignment optimisation using a hyper-heuristic. In *Proceedings of the 2004 IEEE Conference on Cybernetic and Intelligent Systems (CIS2004)*, pages 790–795. IEEE, 2004.

R. H. Kibria and Y. Li. Optimizing the initialization of dynamic decision heuristics in DPLL SAT solvers using genetic programming. In *Proceedings of the 9th European Conference on Genetic Programming*, volume 3905 of *Lecture Notes in Computer Science*, pages 331–340, 2006.

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

J. R. Koza. *Genetic Programming II: Automatic Discovery of Reusable Programs*. MIT Press, 1994.

J. R. Koza and R. Poli. Genetic programming. In E. K. Burke and G. Kendall, editors, *Search Methodologies: Introductory Tutorials in Optimization and Decision Support Techniques*, chapter 5, pages 127–164. Springer, 2005.

J. R. Koza, F. H. Bennett, D. Andre, and K. A. Keane. *Genetic Programming III: Darwinian Invention and Problem solving*. Morgan Kaufmann, 1999.

J. R. Koza, K. A. Keane, M. J. Streeter, W. Mydlowec, J. Yu, and H. Lanza. *Genetic Programming IV : Routine Human-Competitive Machine Intelligence*. Springer, 2003.

D. C. Montgomery. *Design and analysis of experiments*. J. Wiley & Sons, Inc., 6 edition, 2005.

M. Nawaz, E. Enscore-Jr., and I. Ham. A heuristic algorithm for the m-machine, n-job flow-shop sequencing problem. *OMEGA-International Journal of Management Science*, 11(1):91–95, 1983.

Mihai Oltean. Evolving evolutionary algorithms for function optimization. In *Proceedings of the 5th International Workshop on Frontiers in Evolutionary Algorithms*, pages 295–298, 2003.

Mihai Oltean. Evolving evolutionary algorithms using linear genetic programming. *Evolutionary Computation*, 13(3):387–410, 2005.

S. Petrovic, C. Fayad, D. Petrovic, E. Burke, and G. Kendall. Fuzzy job shop scheduling with lot-sizing. *Annals of Operations Research*, 159(1):275–292, 2008.

M. Pinedo. *Scheduling Theory, Algorithms and Systems*. Prentice Hall, 2002.

R. Poli, C. Di Chio, and W. B. Langdon. Exploring extended particle swarms: a genetic programming approach. In *GECCO 2005: Proceedings of the 2005 conference on Genetic and evolutionary computation*, pages 169–176. ACM Press, 2005.

R. Poli, W. B. Langdon, and N. F. McPhee. *A field guide to genetic programming*. Published via `http://lulu.com` and freely available at `http://www.gp-field-guide.org.uk`, 2008. With contributions by J. R. Koza.

J. A. V. Rodríguez, S. Petrovic, and A. Salhi. A combined meta-heuristic with hyper-heuristic approach to the scheduling of the hybrid flow shop with sequence dependent setup times and uniform machines. In P. Baptiste, G. Kendall, A. Munier-Kordon, and F. Sourd, editors, *Proceedings of the 3rd Multidisciplinary International Conference on Scheduling: Theory and Applications*, pages 506–513, Paris, France, August 2007.

P. Ross. Hyper-heuristics. In E. K. Burke and G. Kendall, editors, *Search Methodologies: Introductory Tutorials in Optimization and Decision Support Techniques*, chapter 17, pages 529–556. Springer, 2005.

P. Ross, S. Schulenburg, J. G. Marín-Blázquez, and E. Hart. Hyper-heuristics: Learning to combine simple heuristics in bin-packing problems. In *Genetic and Evolutionary Computation Conference (GECCO 2002)*, 2002.

R. Ruiz and C. Maroto. A comprehensive review and evaluation of permutation flowshop heuristics. *European Journal of Operational Research*, 165(2):479–494, 2005.

R. Ruiz and T. G. Stützle. A simple and effective iterated greedy algorithm for the permutation flowshop scheduling problem. *European Journal of Operational Research*, 177:2033–2049, 2007.

A. J. Ruiz-Torres and G. Centeno. Minimizing the number of late jobs for the permutation flowshop problem with secondary resources. *Computers & Operations Research*, 35(4):1227–1249, 2008.

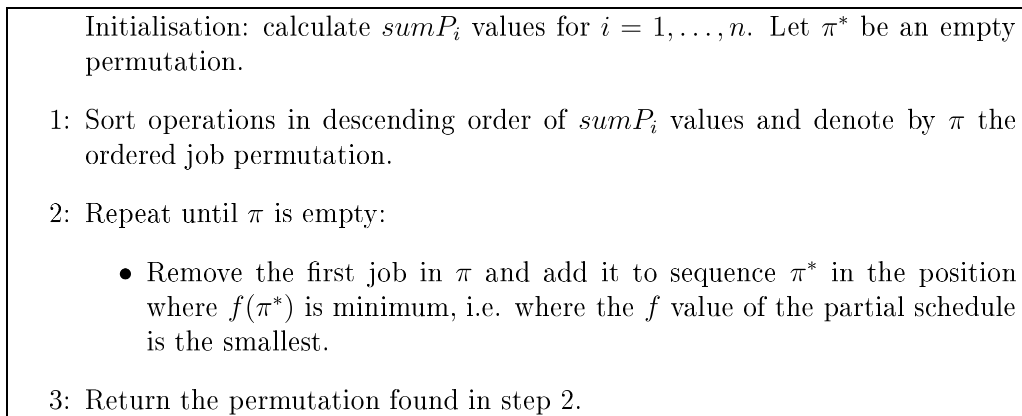
E. Taillard. Some efficient heuristic methods for the flow shop sequencing problem. *European Journal of Operational Research*, 47:65–74, 1990.

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

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

H. Terashima-Marín, C. J. F. Zárate, P. Ross, and M. Valenzuela-Rendón. A ga-based method to produce generalized hyper-heuristics for the 2d-regular cutting stock problem. In *Proceedings of the 8th annual conference on Genetic and evolutionary computation (GECCO 2006)*, 2006.

Figures



Initialisation: calculate $sumP_i$ values for $i = 1, \ldots, n$. Let π^* be an empty permutation.

1: Sort operations in descending order of $sumP_i$ values and denote by π the ordered job permutation.

2: Repeat until π is empty:

- Remove the first job in π and add it to sequence π^* in the position where $f(\pi^*)$ is minimum, i.e. where the f value of the partial schedule is the smallest.

3: Return the permutation found in step 2.

Figure 1: NEH procedure (Nawaz et al., 1983)

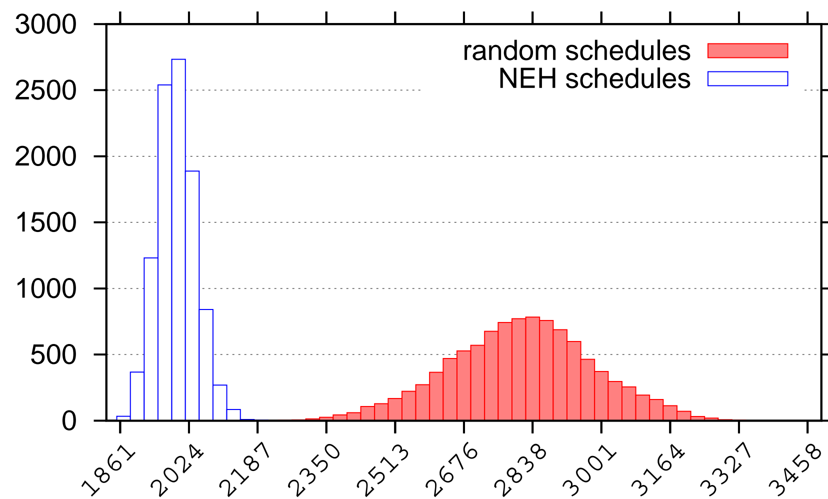


Figure 2: Distribution of the $C_{\max}$ value of 10 000 NEH-random schedules and 10 000 random schedules

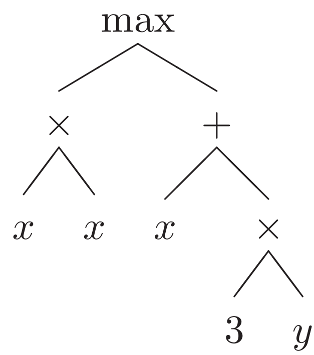
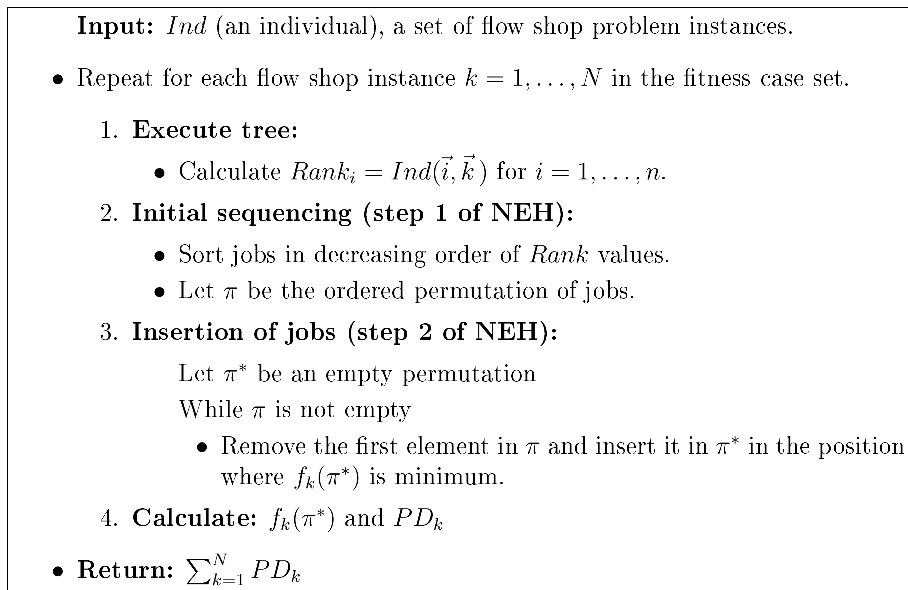


Figure 3: Basic tree-like program representation used in genetic programming. The tree corresponds to the arithmetic expression: $max(x \times x, x + 3 \times y)$.



Input: Ind (an individual), a set of flow shop problem instances.

- Repeat for each flow shop instance $k = 1, \dots, N$ in the fitness case set.
  1. **Execute tree:**
     - Calculate $Rank_i = Ind(\vec{i}, \vec{k})$ for $i = 1, \dots, n$.
  2. **Initial sequencing (step 1 of NEH):**
     - Sort jobs in decreasing order of $Rank$ values.
     - Let π be the ordered permutation of jobs.
  3. **Insertion of jobs (step 2 of NEH):**

     Let π^* be an empty permutation

     While π is not empty
     - Remove the first element in π and insert it in π^* in the position where $f_k(\pi^*)$ is minimum.
  4. **Calculate:** $f_k(\pi^*)$ and PD_k
- **Return:** $\sum_{k=1}^{N} PD_k$

Figure 4: Procedure to calculate the fitness of a genetic programming tree

1. **Initialisation**: Use ramped-half-and-half method to generate initial population

2. **Evolution:** Repeat *iter* times

   2.1 **Fitness evaluation**: execute each program in the population and calculate its fitness.

   2.2 Create new individual program(s) either by

   - **Reproduction**: Copy the selected individual program to the new population.
   - **Crossover**: Create new offspring program(s) for the new population by recombining randomly chosen parts from two selected programs.

   2.3 **Mutation**: For each individual generated in 2.2 proceed as follows. Toss a biased coin, if heads, copy the solution into the new population. If tails, mutate the individual and include in new population.

3. **Return** the best-so-far individual program.

Figure 5: Genetic programming algorithm

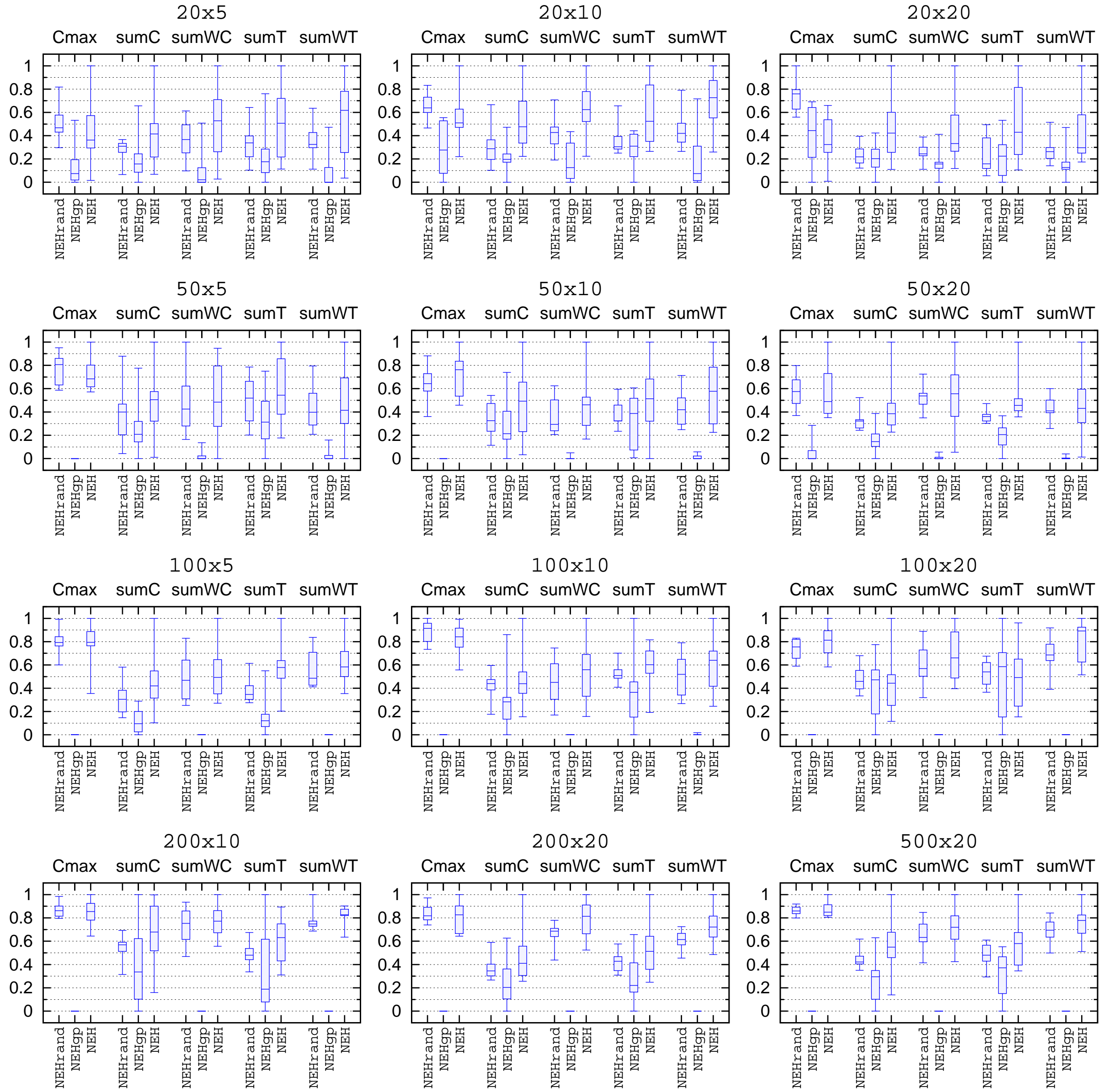


Figure 6: Boxplots of the normalised results obtained by NEH$_{rand}$, NEH$_{GP}$ and NEH on the modified Taillard instances. All objective functions are to be minimised.

Tables

Table 1: Objective functions and problems investigated

objective function	problem
$C_{\max} = \max_i C_i$	$F_m\|r_i, prmu\|C_{\max}$
$sumC = \sum_i C_i$	$F_m\|r_i, prmu\|sumC$
$sumWC = \sum_i w_i C_i$	$F_m\|r_i, w_i, prmu\|sumWC$
$sumT = \sum_i T_i$	$F_m\|r_i, prmu\|sumT$
$sumWT = \sum_i w_i T_i$	$F_m\|r_i, w_i, prmu\|sumWT$

Table 2: Parameter values used during the genetic programming training experimentation

Objective	Find a program to assign ranking values to jobs. The ranking should lead NEH to obtain good solutions for a specific variant of the flow shop problem.
Terminals	r_i, d_i, w_i, n, m, $sumP_i$, $wSumP_i$, $absDif_i$, $wAbsDif_i$, constants $1, \ldots, 10$
Operators	$+, -, \div, \times, FSumP_i, BSumP_i$ Note that $\div, FSumP_i$ and $BSumP_i$ are protected operators.
Initial Pop	Created using "ramped half-and-half" method Koza (1992) with a maximum initial depth of $D_{max} = 7$
Fitness	$\sum_{k=1}^{20} PD_k$
Selection	$K = 5$ tournament selection, elitist
Parameters	Population size $pop = 200$, Crossover probability $X_p = 96\%$, mutation probability $\mu_p = 4\%$, maximum tree size of 500 nodes
Termination	$iter$=500 generations (100 000 solution evaluations)
Fitness cases	20 randomly generated flow shop problems

Table 3: Best ranking functions found by genetic programming after five training runs for each of the investigated problems

problem	ranking function
$F_m\|r_i, prmu\|C_{\max}$	$FSumP_i(4 - 2sumPij) \cdot r_i$
$F_m\|r_i, prmu\|sumC$	$FSumP_i(\frac{7}{sumAbsDif})+$ $FSumP_i(7) + \frac{9}{25} \cdot sumAbsDif + 7$
$F_m\|r_i, w_i, prmu\|sumWC$	$BSumP_i((FSumP_i(10) - 100 \cdot w_i))+$ $BSumP_i(w_i) + 10 \cdot (w_i)^3 + 10$
$F_m\|r_i, prmu\|sumT$ and	$(sumPij - m) \cdot (d_i - \frac{BSumP_i(wSumPij)}{sumPij})$
$F_m\|r_i, w_i, prmu\|sumWT$	$(r_i - d_i - BSumP_i(m)) \cdot \frac{w_i}{n} + d_i \cdot m+$ $BSumP_i(m) + m - w_i + FSumP_i(w_i)$

Table 4: Paired $t-$test ($\alpha = 0.05$) of the average results obtained by the algorithms on the instances for which the boxplots of Figure 6 overlap. A P value ≤ 0.05 indicates a significant difference on means. In case of a difference, the best algorithm is indicated in column "winner".

n	m	obj. function	NEH_{GP} vs NEH_{rand}		NEH_{GP} vs NEH	
			winner	P value	winner	P value
20	5	*sumC*	NEH_{GP}	0.0420	NEH_{GP}	0.0003
20	5	*sumT*	NEH_{GP}	0.0106	NEH_{GP}	0.0001
20	10	$C_{\max}$	NEH_{GP}	0.0002	NEH_{GP}	0.0000
20	10	*sumC*		0.1441	NEH_{GP}	0.0000
20	10	*sumT*		0.0762	NEH_{GP}	0.0001
20	20	$C_{\max}$	NEH_{rand}	0.0001		0.1438
20	20	*sumC*		0.4266	NEH_{GP}	0.0040
20	20	*sumT*		0.4055	NEH	0.0043
50	5	*sumC*		0.1286	NEH_{GP}	0.0431
50	5	*sumT*		0.0621	NEH_{GP}	0.0165
50	10	*sumC*		0.1310	NEH_{GP}	0.0359
50	10	*sumT*		0.1408	NEH_{GP}	0.0307
100	20	*sumC*		0.2615		0.4369
100	20	*sumT*		0.3318		0.4866
200	10	*sumC*	NEH_{GP}	0.0363	NEH_{GP}	0.0018
200	10	*sumT*	NEH_{GP}	0.0216	NEH_{GP}	0.0014
200	20	*sumC*	NEH_{GP}	0.0089	NEH_{GP}	0.0001
200	20	*sumT*	NEH_{GP}	0.0084	NEH_{GP}	0.0002
500	20	*sumT*	NEH_{GP}	0.0000	NEH_{GP}	0.0001

Table 5: Average PD values (to be minimised) obtained by NEH$_{rand}$, NEH$_{GP}$ and NEH on the modified Taillard instances

		$C_{\max}$			*sumC*			*sumWC*			*sumT*			*sumWT*		
n	m	NEH$_{rand}$	NEH$_{GP}$	NEH	NEH$_{rand}$	NEH$_{GP}$	NEH	NEH$_{rand}$	NEH$_{GP}$	NEH	NEH$_{rand}$	NEH$_{GP}$	NEH	NEH$_{rand}$	NEH$_{GP}$	NEH
20	5	5.771	**1.645**	4.997	2.088	**1.468**	3.281	3.608	**0.956**	5.085	10.917	**7.684**	16.559	22.255	**5.904**	33.211
	10	4.659	**2.126**	4.054	1.936	**1.551**	3.001	2.760	**1.229**	4.135	9.807	**8.020**	15.716	16.693	**8.100**	24.659
	20	3.501	1.940	**1.727**	1.651	**1.584**	2.466	2.478	**1.589**	4.077	8.162	**8.000**	12.272	14.601	**9.272**	23.306
50	5	11.731	**0.000**	11.154	1.558	**1.160**	1.820	2.896	**0.150**	3.172	9.738	**7.325**	11.471	25.778	**1.604**	27.877
	10	7.208	**0.000**	7.990	2.200	**1.908**	2.769	3.859	**0.053**	5.165	9.221	**8.098**	11.523	24.912	**0.757**	32.494
	20	5.047	**0.477**	4.929	1.700	**0.911**	2.285	3.142	**0.069**	3.268	7.346	**4.172**	9.848	15.906	**0.204**	16.765
100	5	15.737	**0.000**	15.469	1.520	**0.569**	2.276	4.518	**0.000**	5.018	9.721	**4.021**	14.307	42.929	**0.000**	47.257
	10	12.814	**0.000**	11.725	1.416	**0.970**	1.626	3.707	**0.000**	4.429	7.716	**5.488**	8.556	27.456	**0.099**	32.390
	20	9.329	**0.000**	10.142	1.333	**1.107**	1.263	3.562	**0.000**	4.034	6.469	**5.652**	5.873	21.437	**0.000**	24.602
200	10	17.243	**0.000**	16.722	1.071	**0.774**	1.336	4.199	**0.000**	4.441	6.962	**5.172**	8.512	40.923	**0.000**	43.557
	20	13.631	**0.000**	13.019	1.068	**0.728**	1.363	4.263	**0.000**	5.158	5.780	**3.957**	7.244	30.190	**0.000**	36.493
500	20	19.706	**0.000**	19.920	0.617	**0.403**	0.757	3.987	**0.000**	4.358	4.388	**2.855**	5.320	41.970	**0.000**	45.447
total		10.531	**0.516**	10.154	1.513	**1.094**	2.020	3.582	**0.337**	4.362	8.019	**5.870**	10.600	27.087	**2.162**	32.338

Table 6: Friedman multicomparison tests ($\alpha = 0.05$) of the algorithms on the whole set of instances and grouped by objective function. A P value ≤ 0.05 indicates a significant difference on the sum of the ranks. Small ranks are preferred.

	NEH_{GP} vs NEH_{rand}			NEH_{GP} vs NEH			NEH_{rand} vs NEH		
group	NEH_{GP}	NEH_{rand}	P value	NEH_{GP}	NEH	P value	NEH_{rand}	NEH	P value
overall	**673**	1127	0.0000	**664**	1136	0.0000	**795**	1005	0.0000
$C_{\max}$	**123**	237	0.0000	**126**	234	0.0000	189	171	0.100
sumC	**154**	206	0.0000	**145**	215	0.0000	**150**	210	0.0000
sumT	**150**	210	0.0000	**145**	215	0.0000	**149**	211	0.0000
sumWC	**123**	237	0.0000	**124**	236	0.0000	**151**	209	0.0000
sumWT	**123**	237	0.0000	**124**	236	0.0000	**156**	204	0.0000

Table 7: Average deviation of the results obtained by NEH$_{GP}$ and NEH from those obtained by NEH$_{rand}$. The deviation is measured in standard deviations of the results obtained by NEH$_{rand}$. Large negative values are preferred.

		$C_{\max}$		*sumC*		*sumWC*		*sumT*		*sumWT*	
n	m	NEH$_{GP}$	NEH	NEH$_{GP}$	NEH	NEH$_{GP}$	NEH	NEH$_{GP}$	NEH	NEH$_{GP}$	NEH
20	5	**-1.563**	-0.300	**-0.536**	1.140	**-1.300**	0.721	**-0.619**	1.091	**-1.359**	0.825
	10	**-1.086**	-0.228	**-0.250**	1.194	**-1.108**	0.930	**-0.264**	1.122	**-1.067**	0.931
	20	-0.852	**-0.999**	**0.021**	0.811	**-0.257**	1.031	**0.084**	0.865	**-0.361**	0.952
50	5	**-4.836**	-0.208	**-0.335**	0.343	**-2.446**	0.252	**-0.299**	0.443	**-2.485**	0.144
	10	**-3.397**	0.321	**-0.263**	0.566	**-2.807**	0.949	**-0.229**	0.611	**-2.739**	0.833
	20	**-2.866**	-0.073	**-1.086**	0.740	**-2.803**	0.129	**-1.068**	0.826	**-2.895**	0.106
100	5	**-7.990**	-0.127	**-1.438**	1.248	**-4.604**	0.546	**-1.426**	1.295	**-4.403**	0.468
	10	**-7.642**	-0.651	**-0.701**	0.412	**-3.944**	0.706	**-0.680**	0.327	**-4.033**	0.691
	20	**-6.562**	0.578	**-0.383**	-0.115	**-4.080**	0.544	**-0.383**	-0.162	**-4.051**	0.625
200	10	**-13.309**	-0.425	**-0.660**	0.565	**-6.095**	0.370	**-0.717**	0.548	**-5.994**	0.388
	20	**-11.411**	-0.542	**-0.820**	0.731	**-5.903**	1.257	**-0.791**	0.716	**-6.249**	1.252
500	20	**-23.870**	0.267	**-0.887**	0.569	**-9.444**	0.815	**-0.868**	0.553	**-9.636**	0.755
	total	**-7.115**	-0.199	**-0.611**	0.684	**-3.733**	0.688	**-0.605**	0.686	**-3.773**	0.664

Table 8: Average makespan value obtained by NEH and NEH$_{GP}$ on the Taillard instances for the $F_m|r_i, prmu|C_{\max}$ problem

n	m	NEH	NEH$_{GP}$
20	5	1261.1	**1253**
	10	1583.1	**1581.8**
	20	2318.7	**2316.4**
50	5	**2756.1**	2757.5
	10	**3134.8**	3145.3
	20	3956.9	**3946.8**
100	5	5272.3	**5267.5**
	10	**5752**	5774
	20	6634.4	**6623.3**
200	10	**10804.2**	10815
	20	**11752.6**	11769.4
500	20	**26907.4**	26944.7
total		**6844.46**	6849.55