Applied Soft Computing 1 (2001) 35–52

Applied Soft Computing

www.elsevier.com/locate/asoc

Improving robustness and flexibility of tardiness and total flow-time job shops using robustness measures

Mikkel T. Jensen*

Department of Computer Science, University of Aarhus, Aarhus, Denmark

Accepted 23 March 2001

Abstract

The traditional focus of scheduling research is on finding schedules with a low implementation cost. However, in many real world scheduling applications finding a *robust* or *flexible* schedule is just as important. A robust schedule is a quality schedule expected to still be acceptable if something unforeseen happens, while a flexible schedule is a quality schedule expected to be easy to change. In this paper, the robustness and flexibility of schedules produced by minimising different robustness measures are investigated. One kind of robustness measure is the *neighbourhood-based robustness measure*, in which the basic idea is to minimise the implementation costs of a set of schedules located around a centre schedule. For tardiness problems another way of improving robustness is to increase the slack of the schedule by minimising lateness instead of tardiness. The problems used in the experiments are maximum tardiness, summed tardiness and total flow-time job shop problems.

The experiments showed that the neighbourhood-based robustness measures improves robustness for all the problem types. Flexibility is improved for maximum tardiness and loose summed tardiness problems, while it is not improved for tight summed tardiness problems and total flow-time problems. The lateness-based robustness measures are found to also improve robustness and in some cases flexibility for the same problems, but the improvement is not as substantial as with the neighbourhood-based measures.

Based on these observations, it is conjectured that neighbourhood-based robustness can be expected to improve flexibility on problems with few *critical points*. © 2001 Elsevier Science B.V. All rights reserved.

Keywords: Stochastic scheduling; Robustness; Flexibility; Job shop scheduling

1. Introduction

When solving a scheduling problem the focus traditionally is on minimising a measure of the cost of implementing the schedule. However, most real world scheduling systems operate in dynamic environments, in which unforeseen and unplanned events can happen at short notice. Such events include the breakdown of machines, employees getting sick, new jobs appearing, etc. The problem encountered when an unforeseen event and a schedule has to be changed is usually called a *rescheduling problem*. When a rescheduling problem is solved a new schedule incorporating the changes in the environment and the part of the *preschedule* (the schedule followed prior to the breakdown) already implemented is sought. This schedule should ideally have as low an implementation cost as possible. When the unforeseen event is a breakdown (the temporary unavailability of a resource), the simplest way to solve a rescheduling problem is often to keep the processing order of the preschedule, but delay processing when necessary. In the following this kind of rescheduling is called *simple rescheduling* or

* URL: www.daimi.au.dk/~mjensen/.
E-mail address: mjensen@daimi.au.dk (M.T. Jensen).

1568-4946/01/$ – see front matter © 2001 Elsevier Science B.V. All rights reserved.
PII: S1568-4946(01)00005-9

right-shifting. Right-shifting is the simplest and fastest kind of rescheduling, but in order to improve performance more complex methods searching some set of schedules can be used. In the following, this is called *rescheduling using search*.

The difficulty of the rescheduling problem depends on the nature of the breakdown as well as the preschedule. Some preschedules will generally lead to rescheduling problems with lower implementation costs than others. A preschedule which tends to perform better than ordinary schedules after a breakdown and right-shifting is termed *robust*, while a schedule which tends to perform well after a breakdown and rescheduling using search is termed *flexible*.

It is difficult to relate the terms flexibility and robustness to each other. Often a schedule which is robust can also turn out to be flexible to some degree, since robustness means that the schedule is still acceptable if small delays happen during schedule execution. The acceptability of small delays is an advantage if small changes are made to the schedule. On the other hand, the acceptability of small delays does not necessarily say anything about the possibility of making profound changes in the schedule.

The objective of this paper is to investigate two ways of achieving schedule robustness and flexibility for job shop problems. The first way is the neighbourhood-based robustness measure technique used in [10] on makespan problems, which is reformulated for maximum and summed tardiness, and total flowtime problems. The second way is a simpler idea applicable to tardiness problems; by minimising a measure of lateness instead of tardiness, the slack in the schedules can be increased, which may improve the rescheduling performance of the schedules. The slack of an operation in a schedule is the "buffer time" by which the operation can be delayed without worsening the performance of the schedule.

The work presented in this paper is an extension of the work presented in [11], in which the neighbourhood-based robustness idea was compared to ordinary scheduling for the performance measures maximum tardiness, summed tardiness and total flow time on a smaller range of problems.

The outline of the paper is as follows. Section 2 defines the job shop scheduling problem and notation. In Section 3 previous work on robust scheduling is briefly covered. Section 4 introduces the robustness measures for the maximum, summed tardiness and total flow-time job shop problems. In Section 5 the genetic algorithm used to perform the scheduling is described, while Section 6 describes how breakdowns are simulated and how rescheduling is performed in the experiments. Section 7 describes the experiments and reports the results. Section 8 contains the conclusions.

2. Notation

An $N \times M$ job shop scheduling problem consists of N jobs and M machines. A job J_j consists of a sequence of operations $\bar{O}_j = (o_{j1}, o_{j2}, \ldots, o_{jk_j})$. Each operation o_{jl} is to be processed on a specific machine and has a specific processing time τ_{jl}. Each job has at most one operation on each machine. The processing order of the operations in job J_j must be the order specified in the sequence $\bar{O}_j$. These sequences are often called *the technological constraints*. During processing each machine can process at most one operation at a time, and no preemption can take place; once processing of an operation has been started it must run until it has completed. In the following C_j will denote the end of processing time of the last operation of job J_j in a given schedule.

Some problems include a *due date* d_j for each job, a time by which the processing of the job is supposed to be finished, a *release time* r_j for each job, prior to which no processing of the job can be done, or a *initial set-up time* s_{m} for each machine, prior to which on processing can be done on the machine.

A number of different objective functions exist for job shop problems. The most extensively researched is the *makespan* $C_{\max} = \max_{j\in\{1,\ldots,N\}}(C_j)$, the time elapsed from the beginning of processing until the last operation has completed. The makespan objective is not realistic, since it is not well-suited for scheduling on a rolling time horizon-basis (jobs arriving continuously over time), and since it does not include due dates. More realistic objectives include *total flowtime* $F = \sum_{j=1}^{N} C_j - r_j$, *summed lateness* $L_{\sum} = \sum_{j=1}^{N} C_j - d_j$, *summed tardiness* $T_{\sum} = \sum_{j=1}^{N} \max(C_j - d_j, 0)$, *maximum lateness* $L_{\max} = \max_{j\in\{1,\ldots,N\}}(C_j - d_j)$ and *maximum tardiness* $T_{\max} = \max(L_{\max}, 0)$. All of these performance measures reflect schedule implementation cost and are to be minimised, i.e. a low performance measure equals a good schedule.

All of the performance measures $C_{\max}$, $T_{\sum}$, $T_{\max}$, $L_{\sum}$, $L_{\max}$ and F are *regular* measures. This means that starting an operation earlier never worsens the performance of a schedule. This also means that in the set of optimal schedules there will always be an *active* schedule, a schedule in which no operation can be started earlier without delaying the start of another operation. Since an active schedule can be unambiguously described by the operation processing order [8], a natural representation of schedules for this kind of problem is the processing order.

3. Previous work

Key references on robust and flexible scheduling include the following ones.

Uncertain operation processing times have been treated by a number of authors. In [12] the worst case performance under a number of different *scenarios* of one- and two-machine problems is considered. Based on a theoretical analysis of the problems a branch and bound algorithm and some scheduling heuristics are constructed. Using experiments these algorithms are shown to work well. In [17] a branch and bound algorithm is used to decompose a weighted tardiness job shop problem into a series of subproblems, each of which are solved during execution of the schedule. The method is demonstrated to produce more flexible schedules than two other scheduling methods. In [1] job shop scheduling based on pessimistic estimates of the processing times is shown to be superior to scheduling based on processing time averages.

In [9] an *artificial immune system* (AIS) for solving job shop problems is evolved. The schedules produced by the AIS are demonstrated to be more similar to each other than schedules produced by a standard GA approach. Based on this observation the AIS is conjectured to produce robust schedules.

Breakdown of machines has been treated in [14], in which a robustness measure based on *slack* for makespan job shop problems is defined. Experiments verify that schedules found optimising the robustness measure perform better after a series of breakdowns than ordinary schedules.

In [10] a robustness measure is defined for the makespan job shop scheduling problem. Experiments demonstrate that on average the schedules found by minimising the robustness measure perform better than ordinary schedules after a breakdown, when any of four different rescheduling methods are used.

4. Improving robustness and flexibility

When solving a scheduling problem with performance measure, P, the most straight-forward approach is to simply minimise P. Better approaches can be found to improve the dynamic performance of the schedules.

4.1. Minimising lateness instead of tardiness

When a maximum tardiness ($T_{\max}$) problem is solved schedule robustness and flexibility may be improved by instead minimising the maximum lateness ($L_{\max}$). Recall that since $T_{\max} = \max(L_{\max}, 0)$, minimising $L_{\max}$ will also minimise $T_{\max}$. If $T_{\max}$ is minimised, the minimisation process will stop if a schedule with $T_{\max} = 0$ is reached. If $L_{\max}$ is minimised, $L_{\max}$ will be minimised even if $L_{\max} < 0$. This is likely to improve dynamic performance of the schedule, since minimising $L_{\max}$ beyond $L_{\max} < 0$ will add more slack to the schedule. This slack can be thought of as a "buffer time"; if the schedule has, e.g. $L_{\max} < -20$, then every part of the schedule can be delayed by 20 time-units while still achieving the best possible tardiness performance, $T_{\max} = 0$. Note that this approach can only be expected to improve dynamic performance for loose problems; if no solution exists for which $L_{\max} < 0$, minimising $L_{\max}$ is equivalent of minimising $T_{\max}$.

For these reasons, $L_{\max}$ can be seen as a very simple robustness measure for $T_{\max}$ problems. In the experiments on $T_{\max}$ a series of runs is done minimising $T_{\max}$, as well as a series minimising $L_{\max}$.

For summed tardiness problems, minimising $L_{\sum}$ does not necessarily mean minimising $T_{\sum}$. However, since for $T_{\sum}$ problems it also seems as a good idea to reward schedules with jobs finishing before their due date, for the summed tardiness problems experiments were done minimising $L_{\sum}$ as well as $T_{\sum}$.

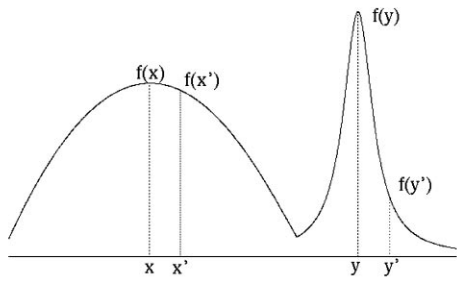

Fig. 1. The idea in neighbourhood-based robustness. If for some reason the solution has to be changed, the broad peak may do much better than the narrow peak, since the solutions lose to x are still reasonable solutions (compare $f(x')$ to $f(y')$). The search space of job shop scheduling problems is very different from this figure, but the basic idea is the same.

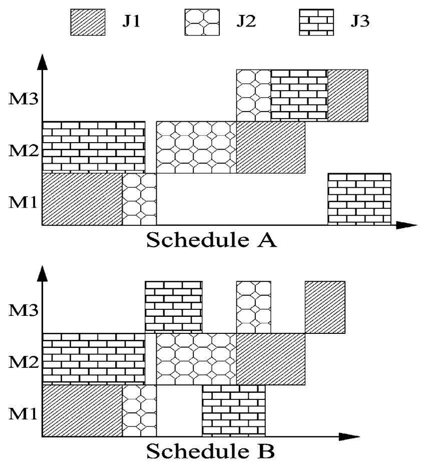

Fig. 2. Gantt-charts of two $\mathcal{N}_1$ neighbours. From A to B exactly one pair of consecutive operations on the same machine 3, jobs 2 and 3, have been exchanged.

4.2. Neighbourhood-based robustness measures

The neighbourhood-based robustness measures are based on an idea introduced for continuous function optimisation problems in [4,16]. Robust solutions are located on broad peaks in the objective function landscape, while brittle solutions are located on narrow peaks (see Fig. 1). In some cases a tradeoff must be made between the width and height of the solution peak. In scheduling the idea behind this is that when a breakdown occurs, maybe a schedule close to the preschedule can work (partly) around the breakdown. If this schedule has a low implementation cost for the original problem, there is a good chance that it will also have a low implementation cost for the rescheduling problem. For this reason, schedules produced using the robustness measures can be expected to be flexible. Since the schedules are created to be good despite small changes, they can also be expected to be robust.

The makespan robustness measure introduced in [10] has the form:

$$R_{C_{\max}}(s) = \sum_{s' \in \mathcal{N}_1(s)} \phi(s, s') C_{\max}(s'),$$

where $\mathcal{N}_1$ is a neighbourhood defined on the schedule searchspace. The neighbourhood works on the processing sequence of the operations. The neighbourhood $\mathcal{N}_1(s)$ of the schedule s is the set of schedules that can be obtained by interchanging two consecutive operations on the same machine (see Fig. 2). The function $\phi(s, s')$ is a weighting function, in [10] $\phi(s, s') = 1/|\mathcal{N}_1(s)|$ is used.

Following this approach we define the robustness measures used in this paper:

$$R_P(s) = \sum_{s' \in \mathcal{N}_1(s)} \phi(s, s') P(s'), \qquad (1)$$

where P is $L_{\sum}$, $L_{\max}$ or F and $\phi(s, s') = 1/|\mathcal{N}_1(s)|$. In order to increase the slack of the schedules, the robustness measures $R_{L_{\max}}$ and $R_{L_{\sum}}$ were used instead of the more obvious measures $R_{T_{\max}}$ and $R_{T_{\sum}}$ (see the previous section).

Because of the need to evaluate the performance of a number of schedules for each fitness evaluation, the algorithm running times when minimising the neighbourhood-based robustness measures were substantially longer than the running times when minimising the other performance measures.

5. The scheduling system

Scheduling was done using a genetic algorithm. The GA was chosen since it is well suited to optimise the robustness measures. Furthermore, GAs have previously been demonstrated to have an acceptable performance on job shop problems. More traditional scheduling techniques (i.e. shifting bottleneck, branch and bound techniques) have sometimes been shown to outperform GA techniques on static scheduling problems, but they are not well suited for the neighbourhood-based robustness measures.

The encoding used in the genetic algorithm is known as permutation with repetition it was introduced in [2]. In this encoding, a schedule is represented by a sequence of job numbers, describing the operation processing order. This representation has the advantage over other job shop schedule representations that it can only represent feasible schedules. Two different decoders were used; a special version of the Giffler–Thompson (GT) algorithm creating active schedules with a bias towards non-delay schedules was used in the total flowtime experiments. This decoder was introduced in [3]. It uses no problem-specific knowledge and is usable on all job shop problems with regular performance measures. In the maximum and summed tardiness problems, a better performing problem-specific decoder was used. It used a semi-active decoding of the gene, followed by a hillclimber. One version of the hillclimber (used when minimising $T_{\max}$, $L_{\max}$, $T_{\sum}$ and $L_{\sum}$) improved the maximum lateness $L_{\max}$ of the schedule. The other version (used when minimising $R_{L_{\max}}$ and $R_{L_{\sum}}$) improved the robustness measure $R_{L_{\max}}$. These decoders are extensions of the decoders for makespan, [15], and the makespan robustness measure [10]. A more detailed description of the decoders can be found in Appendix A.

The use of the $L_{\max}$ and $R_{L_{\max}}$ hillclimbers in the experiments on summed tardiness is non-standard, since the hillclimbers do not work on the problem objectives $T_{\sum}$ and $R_{L_{\sum}}$, but instead improve the related measures $L_{\max}$ and $R_{L_{\max}}$. This choice was made since preliminary experiments indicated that the $L_{\max}$ and $R_{L_{\max}}$ hillclimbing decoders performed better on the summed tardiness problems than the GT-like decoder.

After decoding the constructed schedule was written into the gene, in such a way that a semiactive decoding of the gene would yield the schedule produced by the hillclimber or GT decoder. The procedure is a kind of Lamarckian learning, and is often called *forcing*.

A standard generational genetic algorithm was used, since preliminary experiments showed that it outperformed a GA resembling the diffusion GA used in [15]. The population size was 100, and the algorithm was run for 100 generations. Tournament selection with a tournament size of two was used. All new individuals were created using crossover. Each new individual was mutated with a probability of 0.1. The *generalised order crossover* (GOX) and *position-based mutation* (PBM) operators were used, see [15]. Elitism was not used, but the all time best individual was recorded and returned at the end of the run.

After the completion of the genetic algorithm a hillclimber searching the $\mathcal{N}_1$-neighbourhood was run on the best solution found. This hillclimber was run "on top of" the decoding hillclimbers used in the tardiness experiments. In the case of normal runs using the $T_{\max}$ decoding hillclimber the schedule was made active after the run of the hillclimber using the procedure used for decoding in [2]. This was done since preliminary experiments showed that it improved the algorithm performance.

6. Breakdowns and rescheduling methods

The rescheduling problems used in the experiments were created by simulating a machine breakdown partway through the execution of the preschedule. The breakdown made the machine unavailable for processing for a predefined period of time (the *downtime*), after which it would become operational again. Because of the definition of the job shop problem, the machine needed to become operational again, otherwise the rescheduling problem would be unsolvable. The rescheduling problems were themselves job shop scheduling problems, and could be solved using standard scheduling methods, with the added possibility of using the preschedule (which is known to be a good solution to a closely related problem) in the scheduling process.

The rescheduling job shop problems simulating breakdowns were created from the original problem and the preschedule in the following way:

1. The preschedule s was generated.
2. A random operation o_X of the preschedule was picked uniformly.
3. The starting time of o_X in s was denoted the *breakdown time*.
4. All operations from the original problem with starting times at the breakdown time or later in s where included in the rescheduling problem.

5. In the rescheduling problem, the release time of each job J_j was set the maximum of the breakdown time and the end of processing time of any operation of J_j being processed at breakdown time in s. In the same way, initial set-up time of every machine M_i was set to the maximum of the breakdown time and the end of processing time in s of any operation being processed on M_i at breakdown time.
6. The set-up time of the machine with the breakdown was set to the breakdown time plus the downtime. The downtime was the time during which the broken down machine was unavailable. In all the experiments a downtime of 80 was used (for all the problems the processing time of each operation is between 1 and 100).

The effect of this way of simulating a breakdown was that a random machine would break down precisely at the time it was supposed to start processing an operation. The broken down machine would be unavailable for some predefined (and known) time, an obvious interpretation is that the machine was being repaired. The scheduler was free to reschedule any operations not yet commenced at breakdown time. The fact that the scheduler knew at breakdown time exactly when the broken down machine would become operational again is probably unrealistic for most real-world scheduling problems, a future direction of research could be to remove this assumption.

After the breakdown rescheduling was performed. In this context, rescheduling is the same as solving a job shop problem, perhaps with the aid of the preschedule. In the experiments, this was defined in five different ways:

1. *Right-shifting*. Simply wait for the breakdown to be repaired and use the scheduling order of the preschedule. This is expected to yield low quality results compared to other methods, but at a very low computational cost.
2. $\mathcal{N}_1$*-based rescheduling*. All $\mathcal{N}_1$ neighbours of the preschedule are generated, and the one best solving the rescheduling problem is used. This is the simplest kind of search-based rescheduling used, and is expected to yield better results than right-shifting, still at low computational cost.
3. *Hillclimbing rescheduling*. The preschedule is used as a starting point for a hillclimber, which finds a locally optimal solution to the rescheduling problem. In the $T_{\max}$ experiments, the $L_{\max}$ and $R_{L_{\max}}$ hillclimbers described in Section 5 and Appendix A were used. Since no efficient hillclimber was available for the total flowtime and summed tardiness measures, this kind of rescheduling was not used in the experiments with these measures.
4. *Reduced rescheduling*. Generate a reduced problem as described in [6] by removing all operations not affected by the breakdown from the problem. An operation is affected by a breakdown if it succeeds

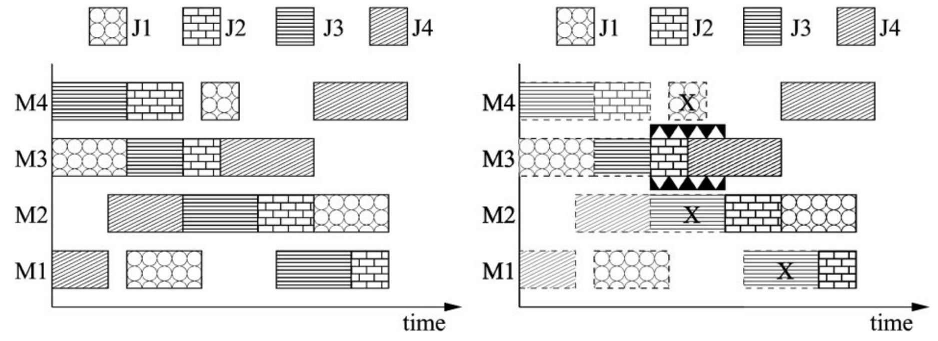

Fig. 3. Gantt-chart example of a the generation of a reduced rescheduling problem. *Left*: preschedule. *Right*: schedule at breakdown. The breakdown happens at machine M3 and has been marked by the black triangles. The lightly coloured operations will not be in the reduced rescheduling problem, while the dark coloured operations will be. The operations marked by '*X*' commence processing after the breakdown but will not be in the rescheduling problem since they are not affected by the breakdown. After rescheduling these operations will be scheduled in the same way as they were in the preschedule. An operation is said to be affected by a breakdown if an increase in the breakdown downtime an lead to a delay of the operation if a right-shifting rescheduling method is used.

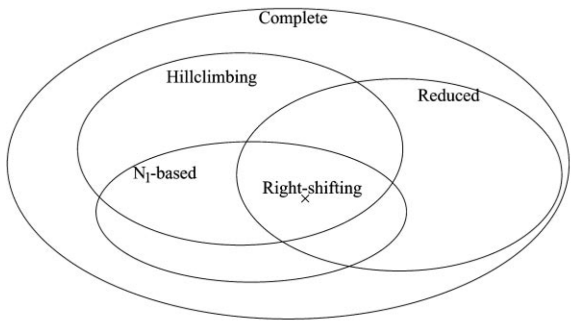

Fig. 4. The relationship between the search-spaces of the five rescheduling methods.

the operation o_X in the graph representation of the preschedule (see Appendix A for the graph representation). This problem is solved from scratch using the GA. Generation of a reduced rescheduling problem is illustrated in Fig. 3.

5. *Complete rescheduling*. Solve the rescheduling problem described above from scratch using the GA. This is expected to give a high schedule quality at a high computational cost.

In the experiments involving robustness measures rescheduling using methods 3–5 minimised the robustness measures, not the “raw” performance measures.

The search-spaces of the five rescheduling methods have been visualised in Fig. 4. Since for this problem a large searchspace can be expected to mean a higher solution quality (and a longer running time), complete rescheduling can be expected to outperform all the other rescheduling methods. In the figure, neither of the search-spaces of hillclimbing and $\mathcal{N}_1$ rescheduling are shown to contain the other. In reality, the part of the $\mathcal{N}_1$ searchspace not contained in the hillclimbing search-space is very small, while the part of the hillclimbing search-space not contained in the $\mathcal{N}_1$ search-space is large. For this reason hillclimbing can be expected to outperform $\mathcal{N}_1$ rescheduling in most cases. Since the “search-space” of right-shifting is just a single point contained in all the other search-spaces, right-shifting can be expected to be the poorest (and fastest) of the rescheduling procedures. Reduced rescheduling can be expected to perform worse than complete rescheduling, while its relationship to the other rescheduling methods cannot be determined from the figure, since the reduced search-space is not contained in and does not contain the other search-spaces. The sizes of the sets are a bit misleading in the figure; for many rescheduling problems the search-spaces of complete and reduced rescheduling will be much larger than the other search-spaces.

7. Experiments

The purpose of the experiments was to investigate if the schedules optimised for the robustness measures perform better after a breakdown and after rescheduling than schedules optimised for the ordinary performance measures, and if the preschedule performance (without breakdowns) was decreased when the robustness measures were optimised. This was done by for each performance criterion having a series of runs in which the performance criterion was optimised, and a series of runs in which the corresponding robustness measures were optimised. Each series was done on 42 different benchmarks, doing 400 runs on each benchmark. During each run a breakdown was simulated, and the rescheduling problem solved using each of the rescheduling methods.

The benchmark problems used were the forty problems from [13] (prefixed by `la` in the following) and the problems `ft10` and `ft20` from [7]. These problems are artificial; they were generated at random. They are all rectangular problems in which each job is processed on each machine exactly once. The problem sizes (number of jobs $\times$ number of machines) are shown in Table 1.

Since the problems were originally generated without due dates, these were added. The due dates were generated by for each problem generating a random active schedule, and setting the due date for each job to the job’s completion time minus 5% (loose problems, labelled $\sigma = 0.95$) or 15% (tight problems, labelled $\sigma = 0.85$). This procedure was followed in order to be able to investigate whether the

Table 1
Sizes of the problems used in the experiments

`la01--la05`: 10×5	`la06--la10`: 15×5
`la11--la15,ft20`: 20×5	`la16--la20,ft10`: 10×10
`la21--la25`: 15×10	`la26--la30`: 20×10
`la31--la35`: 30×10	`la36--la40`: 15×15

Table 2
Summarised results of the maximum tardiness experiments

Average maximum tardiness													
Size	Method	$\sigma = 0.95$						$\sigma = 0.85$					
		P	1	2	3	4	5	P	1	2	3	4	5
10 × 5	$T_{\max}$	6.0	56.5	51.4	48.5	41.4	40.4	50.8	111.2	106.5	104.0	99.0	97.9
	$L_{\max}$	6.0	47.5	42.9	40.3	36.0	35.5	50.8	112.8	108.3	105.6	99.7	98.7
	$R_{L_{\max}}$	6.0	44.5	40.5	39.8	37.2	35.9	53.7	106.7	102.8	102.0	99.6	98.2
15 × 5	$T_{\max}$	18.2	66.8	60.5	55.3	44.4	44.0	84.8	138.9	133.2	128.9	118.7	118.1
	$L_{\max}$	18.2	52.8	48.8	46.0	40.4	40.1	84.8	133.1	127.7	124.2	115.9	115.3
	$R_{L_{\max}}$	18.2	41.5	39.6	39.5	37.8	37.2	85.8	114.7	112.5	112.2	110.6	109.6
20 × 5	$T_{\max}$	21.2	71.4	66.0	61.4	46.6	46.2	108.5	166.7	161.3	156.9	144.4	143.8
	$L_{\max}$	21.2	56.5	53.2	50.1	42.2	42.1	108.4	163.9	158.8	154.9	142.5	142.0
	$R_{L_{\max}}$	21.3	42.5	39.9	38.9	36.7	36.5	109.3	148.9	145.2	143.2	138.0	137.4
10 × 10	$T_{\max}$	0.0	45.4	39.0	35.7	25.9	24.7	57.4	112.6	106.5	103.8	95.8	93.8
	$L_{\max}$	0.0	30.0	25.9	23.7	18.4	17.5	57.3	112.9	107.2	104.4	96.5	94.8
	$R_{L_{\max}}$	0.1	27.1	23.5	22.5	20.4	18.9	64.0	110.5	105.8	103.8	99.8	97.5
15 × 10	$T_{\max}$	0.0	36.8	29.7	25.1	13.0	12.5	26.3	80.7	74.3	70.5	59.2	56.9
	$L_{\max}$	0.0	6.5	5.2	4.5	2.6	2.5	26.2	76.3	70.8	67.1	57.4	55.6
	$R_{L_{\max}}$	0.0	5.2	4.1	3.7	2.5	2.3	27.4	69.2	65.2	63.4	57.2	55.1
20 × 10	$T_{\max}$	0.0	35.6	28.2	22.5	9.4	9.0	28.8	84.5	78.0	73.2	61.2	59.2
	$L_{\max}$	0.0	0.5	0.4	0.3	0.2	0.2	28.7	83.7	77.2	72.3	60.2	57.9
	$R_{L_{\max}}$	0.0	0.3	0.2	0.2	0.2	0.2	28.7	72.6	67.6	65.4	59.4	57.2
30 × 10	$T_{\max}$	0.0	33.4	26.5	20.7	8.2	8.2	104.0	154.1	147.1	140.6	127.1	126.0
	$L_{\max}$	0.0	6.1	4.7	3.9	2.4	2.6	104.1	153.8	146.3	140.0	126.1	126.1
	$R_{L_{\max}}$	0.0	2.2	1.9	1.8	1.4	1.4	97.3	129.1	125.8	123.9	116.3	115.8
15 × 15	$T_{\max}$	0.1	38.1	30.1	25.1	12.3	11.8	74.9	126.8	119.4	114.7	103.7	101.9
	$L_{\max}$	0.1	10.3	8.0	5.0	4.3	4.1	74.8	127.1	119.3	114.6	103.8	102.0
	$R_{L_{\max}}$	0.2	9.3	7.5	6.9	5.3	4.9	80.5	120.3	115.1	113.2	107.8	105.6

tightness of the problems influenced the usefulness of the robustness measures. The benchmark problems are available upon request.

7.1. The maximum tardiness experiments

The results of the maximum tardiness experiments are summarised in Tables 2 and 3. Table 2 reports the average maximum tardiness found for each problem size. In the table, there is a subtable for each size of problem instance (these subtables stretch downwards). In each subtable there is a row dedicated to each scheduling method, labelled with the performance measure minimised. The columns labelled 1–5 hold the results of the rescheduling methods, while the column labelled P holds information about the preschedule performance. In each table the left part (labelled $\sigma = 0.95$) shows results for the loose problems, while the right part (labelled $\sigma = 0.85$) shows results for the tight problems.

Table 3 is organised in the same way as Table 2, and reports the number of problems of each problem size for which each method was found to give the best average performance. Since the tardiness distribution is unknown no statistical testing has been performed; a scheduling method is simply said to be give the best result for a given benchmark if the average tardiness is observed to at least as low as the average tardiness of the other methods. For this reason and because some of the averages were very close, the numbers in Table 3 should not be taken to seriously; another set of experiments could change some of the numbers slightly.

Table 3
Summarised results of the maximum tardiness experiments

Number of best results for each method													
Problem size	Method	$\sigma = 0\ 95$						$\sigma = 0\ 85$					
		P	1	2	3	4	5	P	1	2	3	4	5
10 × 5	$T_{\max}$	5	0	0	0	0	0	4	2	3	3	2	2
	$L_{\max}$	5	3	4	1	2	2	5	0	1	0	1	1
	$R_{L_{\max}}$	5	2	2	4	3	3	3	3	2	2	2	2
15 × 5	$T_{\max}$	5	0	0	0	1	0	4	0	0	0	0	0
	$L_{\max}$	5	1	1	1	1	0	5	0	0	0	1	1
	$R_{L_{\max}}$	5	5	5	5	4	5	3	5	5	5	4	4
20 × 5	$T_{\max}$	5	0	0	0	0	0	5	0	0	0	0	0
	$L_{\max}$	5	1	1	2	2	2	4	0	0	0	1	0
	$R_{L_{\max}}$	2	6	6	6	5	5	3	6	6	6	5	5
10 × 10	$T_{\max}$	5	0	0	0	1	1	3	2	3	2	3	3
	$L_{\max}$	5	1	1	2	6	4	4	2	1	2	2	1
	$R_{L_{\max}}$	4	6	5	4	0	1	0	2	2	2	1	2
15 × 10	$T_{\max}$	5	0	0	0	0	0	2	0	0	0	1	1
	$L_{\max}$	5	1	4	4	4	4	4	0	0	0	2	2
	$R_{L_{\max}}$	5	4	4	4	4	4	1	5	5	5	2	2
20 × 10	$T_{\max}$	5	0	0	0	0	0	1	0	0	0	0	0
	$L_{\max}$	5	4	3	3	4	4	2	0	0	0	2	2
	$R_{L_{\max}}$	5	5	5	5	4	4	2	5	5	5	3	3
30 × 10	$T_{\max}$	5	0	0	0	0	0	5	0	0	0	0	0
	$L_{\max}$	5	2	2	2	2	2	5	2	2	2	2	2
	$R_{L_{\max}}$	5	5	5	5	5	5	5	5	5	5	5	5
15 × 15	$T_{\max}$	3	0	0	0	0	0	2	0	0	1	2	2
	$L_{\max}$	5	1	3	4	5	5	3	0	0	1	3	2
	$R_{L_{\max}}$	3	4	3	2	1	1	0	5	5	3	0	1

7.1.1. The loose problems, $\sigma = 0.95$

Inspecting Table 3, it becomes clear that for all of the loose problems ($\sigma = 0.95$), the standard scheduling method optimising $T_{\max}$ is outperformed by the other two methods with respect to rescheduling performance for every single problem instance and for every rescheduling method. The performance difference is also clear from Table 2; in all cases the performance averages of the $T_{\max}$ method is well above the other methods (recall that a low performance measure indicates a good solution).

Comparing the performance of the neighbourhood-based robustness measure $R_{L_{\max}}$ to the performance of the simple robustness measure $L_{\max}$, it seems that for the simple rescheduling methods (labelled 1–3), the neighbourhood-based robustness measure generally performs best, although there are a few exceptions, especially for the problem sizes 10 × 5 and 15 × 15. The performance difference is largest for the problems with a high job to machine ratio; for problem sizes 15 × 5, 20 × 5, 20 × 10 and 30 × 10 there is a substantial difference between the two methods. Considering the complex rescheduling methods 4 and 5, the neighbourhood-based robustness measure seems to slightly outperform the $L_{\max}$ measure on problems with a high job to machine ratio, while for the other problems, the $L_{\max}$ and $R_{L_{\max}}$ methods seem to do equally well. With respect to preschedule performance, the $T_{\max}$ and $L_{\max}$ methods seem to do equally well, while the $R_{L_{\max}}$ method performs slightly worse than these.

7.1.2. The tight problems, $\sigma = 0.85$

When comparing the $T_{\max}$ and $L_{\max}$ methods on the tight problems, there is still a small performance advantage of the $L_{\max}$ method in a few cases, but by and large the two methods perform equally well. The $R_{L_{\max}}$ method can be seen to outperform the other

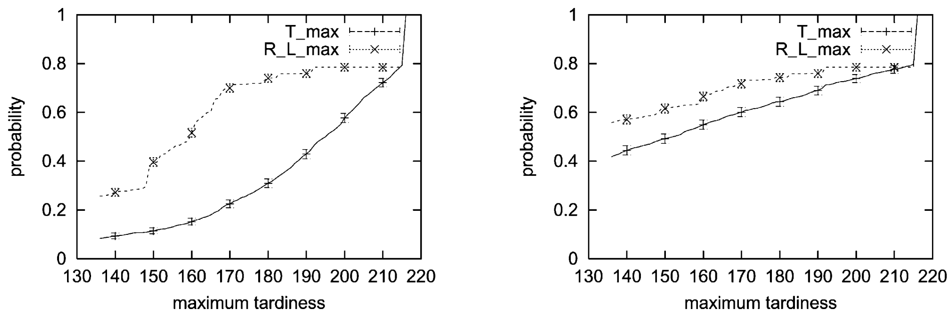

Fig. 5. Left: observed maximum tardiness distribution functions for right-shifting rescheduling for $T_{\max}$ and $R_{L_{\max}}$ scheduling for the `la06` $\sigma = 0.85$ problem. The x-axes indicate maximum tardiness after rescheduling, while the y-axes indicate probability. The error-bars indicate 95% confidence intervals on the percentiles. The plot for $L_{\max}$ scheduling was coincidental with the $T_{\max}$ plot, and was left out for clarity. Right: same plot for complete rescheduling.

methods in most cases for the simple rescheduling methods 1–3.

With regard to rescheduling methods 4 and 5, for the problems with a high job to machine ratio it usually outperforms the other methods, demonstrating an improved flexibility of the schedules. For problems with a low job to machine ratio the $R_{L_{\max}}$ method is outperformed by the other methods in a few cases.

Another way of viewing the performance difference between normal and robust scheduling can be seen in Fig. 5. Distribution functions for maximum tardiness of the `la06` $\sigma = 0.85$ problem after rescheduling have been plotted for right-shifting and complete rescheduling. The distribution functions have been approximated by experiments; the true distribution functions are unknown. Given a maximum tardiness value, the probability of ending up with a performance of that or lower (better) can be read from the graphs. Since the plots for the $R_{L_{\max}}$ runs are clearly above and to the left of the plots for the other runs, the superior performance of the $R_{L_{\max}}$ for this problem is evident. The error-bars on the plots indicate that this result is statistically significant. An example reading of the plots could be that for right-shifting rescheduling of $R_{L_{\max}}$ schedules, there is a 70% probability of ending up with a schedule of maximum tardiness 170 or lower, while for the $T_{\max}$ and $L_{\max}$ schedules the same probability is 23%. The `la06` problem is typical of the tight 15×5 problems.

Comparing the averages produced by the different rescheduling techniques, it becomes clear that the considerations at the end of Section 6 hold; the higher the label number of the rescheduling technique, the better the performance.

The results of the summed tardiness experiments are summarised in Tables 4 and 5. The tables are equivalent to Tables 2 and 3 for the maximum tardiness experiments.

For the loose problems ($\sigma = 0.95$), the results of the summed tardiness problems are resemblant of the maximum tardiness problems. The $R_{L_{\sum}}$ robustness measure seems to improve the performance of rescheduling methods 1 and 2 more than the $L_{\sum}$ measure on all problems, and for problems with a high job/machine ratio, $R_{L_{\sum}}$ outperforms $L_{\sum}$ on the performance of rescheduling methods 4 and 5 as well. In some cases this comes at a cost of a slight increase in preschedule cost. The $L_{\sum}$ measure outperforms the standard $T_{\sum}$ measure as well, but not as much as $R_{L_{\sum}}$.

The similarity in behaviour between loose summed tardiness problems and maximum tardiness problems does not come as a surprise; in a loose summed tardiness problem often only one or a few job will be tardy. In this case, minimising maximum tardiness and summed tardiness is almost the same.

For the tight problems, the picture is a bit different. For rescheduling methods 1 and 2 the schedules produced by using $R_{L_{\sum}}$ still seems superior to those produced using $L_{\sum}$. For the complex rescheduling methods there only seems to be a little difference between the performance of the $L_{\sum}$ schedules and the $T_{\sum}$ schedules, while in several instances the $R_{L_{\sum}}$

Table 4
Summarised results of the summed tardiness experiments. There is no olumn labelled '3' in the table since no hillclimber optimising $T_{\sum}$ was available

Average summed tardiness

		$\sigma = 0\ 95$					$\sigma = 0\ 85$				
Problem size	Method	P	1	2	4	5	P	1	2	4	5
10 × 5	$T_{\sum}$	16.5	200.3	155.6	103.0	98.6	201.7	493.8	431.8	356.5	346.6
	$L_{\sum}$	16.5	161.9	127.2	88.4	85.2	201.3	497.4	434.4	354.5	346.1
	$R_{L_{\sum}}$	17.1	148.1	119.5	91.4	87.9	218.4	472.8	421.0	364.3	356.4
15 × 5	$T_{\sum}$	55.6	250.3	202.6	126.8	124.7	372.8	674.8	615.0	504.1	493.2
	$L_{\sum}$	55.1	204.8	173.5	122.0	120.2	371.9	649.2	596.6	499.9	490.0
	$R_{L_{\sum}}$	59.6	160.5	142.7	117.0	115.4	380.5	605.5	567.1	501.4	493.9
20 × 5	$T_{\sum}$	61.7	315.5	256.9	136.3	134.0	567.1	989.1	915.3	733.0	719.9
	$L_{\sum}$	61.7	237.2	198.9	126.1	124.4	566.9	963.7	895.3	728.2	713.8
	$R_{L_{\sum}}$	62.5	181.4	157.0	117.1	116.7	567.2	912.1	857.8	730.3	720.5
10 × 10	$T_{\sum}$	0.0	166.5	120.5	51.0	46.8	257.8	550.1	498.7	397.0	380.9
	$L_{\sum}$	0.0	87.7	67.4	32.3	29.9	255.5	556.7	500.8	395.6	382.1
	$R_{L_{\sum}}$	0.9	82.5	63.1	38.7	36.8	287.8	542.0	499.1	425.4	412.2
15 × 10	$T_{\sum}$	0.0	152.9	105.6	23.6	21.4	138.3	499.2	423.0	268.5	256.3
	$L_{\sum}$	0.0	26.4	17.5	4.4	4.0	132.5	457.6	391.0	259.4	245.4
	$R_{L_{\sum}}$	0.0	18.9	12.8	5.4	5.2	144.0	410.9	363.3	265.8	254.0
20 × 10	$T_{\sum}$	0.0	167.6	114.5	17.0	16.0	168.6	658.6	555.0	307.5	288.5
	$L_{\sum}$	0.0	6.1	4.1	0.6	0.6	167.3	663.0	563.6	313.8	293.4
	$R_{L_{\sum}}$	0.0	2.0	1.4	0.4	0.3	166.9	560.4	489.6	310.4	293.7
30 × 10	$T_{\sum}$	0.0	188.2	126.6	14.5	14.1	946.5	1648.6	1483.7	1048.7	1015.2
	$L_{\sum}$	0.0	20.3	13.2	4.1	4.3	945.3	1647.7	1489.3	1046.6	1021.1
	$R_{L_{\sum}}$	0.0	8.0	6.6	3.5	3.5	744.3	1209.7	1112.0	862.6	846.7
15 × 15	$T_{\sum}$	0.4	156.5	103.7	22.5	18.9	448.2	876.3	785.9	600.2	576.8
	$L_{\sum}$	0.2	48.9	32.7	8.8	7.8	449.2	869.3	779.8	606.2	581.9
	$R_{L_{\sum}}$	1.0	35.4	25.3	12.0	10.1	489.8	824.8	753.7	634.9	606.6

schedules perform substantially worse than the other schedules, indicating that for tight summed tardiness problems, flexibility seems to be worsened by using this robustness measure.

The use of the $L_{\sum}$ performance measure did not seem to degrade the preschedule cost when compared to the $L_{\sum}$ experiments. For the tight problems, the use of the $R_{L_{\sum}}$ measure increased preschedule cost in many cases, although there were a few intriguing exceptions in which the preschedule costs found by using $R_{L_{\sum}}$ were much lower than the preschedule costs found using the other performance measures.

7.2. *The total flowtime experiments*

The results of the total flow-time experiments are summarised in Table 6. The table is equivalent to the Tables 2 and 3 for maximum tardiness, except there is only one table. Each entry contains two numbers; the first is the number of problems for which the method was observed to produce the lowest average, while the second (in parenthesis) is the average performance. Since there is no lateness-based robustness measure for the total flowtime experiments, only "ordinary scheduling" (minimising F) and "neighbourhood-based robustness scheduling" (minimising R_F) were compared.

Qualitatively the results are similar to the results for the tight summed tardiness problems. For rescheduling using methods 1 and 2 the R_F runs seem slightly superior, while in many cases they are inferior when methods 3 and 4 are used. With respect to preschedule performance, the two methods seem to have comparable performance. For some problems sizes F seems to be better, while on others R_F seems to do best.

Table 5
Summarised results of the summed tardiness experiments

Number of best results for each method											
Problem size	Method	$\sigma = 0\ 95$					$\sigma = 0\ 85$				
		P	1	2	4	5	P	1	2	4	5
10 × 5	T_{Σ}	5	0	0	0	0	3	1	3	1	2
	L_{Σ}	5	1	1	3	3	3	0	0	2	2
	$R_{L_{\Sigma}}$	4	4	4	3	2	0	4	2	2	1
15 × 5	T_{Σ}	2	0	0	1	1	2	0	0	2	3
	L_{Σ}	5	0	1	2	1	4	0	0	2	1
	$R_{L_{\Sigma}}$	2	5	5	4	4	1	5	5	1	1
20 × 5	T_{Σ}	4	0	0	0	0	1	0	0	2	1
	L_{Σ}	4	0	1	1	1	2	0	0	3	4
	$R_{L_{\Sigma}}$	2	6	6	6	6	3	6	6	1	1
10 × 10	T_{Σ}	6	0	0	0	0	3	2	2	3	4
	L_{Σ}	6	3	3	4	5	4	0	2	3	2
	$R_{L_{\Sigma}}$	1	3	3	2	1	0	4	2	0	0
15 × 10	T_{Σ}	5	0	0	0	0	0	0	0	3	1
	L_{Σ}	5	2	2	5	5	5	0	0	2	3
	$R_{L_{\Sigma}}$	5	5	5	3	3	0	5	5	1	1
20 × 10	T_{Σ}	5	0	0	0	0	2	0	0	3	2
	L_{Σ}	5	3	3	3	3	0	0	0	1	2
	$R_{L_{\Sigma}}$	5	5	5	5	5	3	5	5	1	1
30 × 10	T_{Σ}	4	0	0	0	0	0	0	0	0	0
	L_{Σ}	4	1	1	2	2	0	0	0	0	0
	$R_{L_{\Sigma}}$	5	5	5	5	5	5	5	5	5	5
15 × 15	T_{Σ}	2	0	0	0	0	3	1	1	4	3
	L_{Σ}	4	0	0	4	3	2	0	0	1	2
	$R_{L_{\Sigma}}$	0	5	5	1	2	0	4	4	0	0

Table 6
Summarised results of the total flowtime experiments

Number of best results for each method (average)						
Problem size	Method	P	1	2	4	5
10 × 5	F	4 (4572.5)	1 (4947.6)	1 (4896.0)	4 (4783.0)	4 (4766.7)
	R_F	1 (4591.2)	4 (4933.0)	4 (4886.4)	1 (4797.8)	1 (4784.9)
15 × 5	F	5 (9453.3)	0 (10012.3)	1 (9931.7)	5 (9702.8)	4 (9674.7)
	R_F	0 (9481.1)	5 (9991.2)	4 (9916.7)	0 (9728.7)	1 (9705.6)
20 × 5	F	0 (15590.4)	0 (16344.8)	0 (16231.0)	1 (15818.1)	2 (15758.1)
	R_F	6 (15538.5)	6 (16196.7)	6 (16099.1)	5 (15782.1)	4 (15744.6)
10 × 10	F	5 (7451.5)	2 (7842.4)	2 (7774.4)	5 (7638.9)	5 (7616.0)
	R_F	1 (7461.0)	4 (7826.3)	4 (7764.4)	1 (7650.5)	1 (7631.7)
15 × 10	F	4 (12925.0)	0 (13513.1)	0 (13414.3)	3 (13170.9)	3 (13127.9)
	R_F	1 (12945.6)	5 (13455.7)	5 (13370.7)	2 (13180.3)	2 (13145.8)
20 × 10	F	1 (21243.8)	0 (22027.2)	0 (21885.6)	1 (21476.6)	3 (21382.5)
	R_F	4 (21230.9)	5 (21885.9)	5 (21772.9)	4 (21448.5)	2 (21376.5)
30 × 10	F	1 (42390.9)	0 (43501.1)	0 (43300.3)	0 (42493.3)	0 (42366.6)
	R_F	4 (42353.1)	5 (43097.5)	5 (42938.2)	5 (42320.0)	5 (42200.9)
15 × 15	F	4 (17067.9)	1 (17657.9)	2 (17545.3)	5 (17280.8)	5 (17238.7)
	R_F	1 (17096.2)	4 (17617.1)	3 (17522.4)	0 (17308.4)	0 (17265.5)

8. Discussion and conclusion

In this paper, the robustness and flexibility of tardiness and total flow-time job shop schedules facing breakdowns have been investigated. The schedules have been produced with a GA optimising standard performance (cost), neighbourhood-based robustness measures and lateness robustness measures (for the tardiness problems).

The lateness-based robustness measures have been demonstrated to improve schedule robustness and flexibility for loose maximum tardiness and loose summed tardiness problems, while they have been found equivalent to standard scheduling on tight problems.

It has been demonstrated that the neighbourhood-based robustness measures generally seem to improve schedule robustness for all problem performance measures tried, and both for tight and loose tardiness problems. Schedule flexibility in many cases seems to be improved for maximum tardiness problems and loose summed tardiness problems, while it does not seem to be improved for tight summed tardiness problems and total flow-time problems, in which case the flexibility is sometimes seen to be worse than for standard scheduling. For many problems, the improvement in robustness and flexibility when using the neighbourhood-based robustness measures were found to be better than the improvement gained from using the lateness-based robustness measures.

The explanation of the poor flexibility of the schedules produced using neighbourhood-based robustness measures for total flow time and for the deteriorating flexibility of the summed tardiness schedules as the problems become more tight may be that when flexibility is sought, the neighbourhood-based robustness idea works best for problems in which there are few *critical points*. A critical point is a part of the schedule which cannot be changed without worsening the schedule. In makespan problems or maximum tardiness problems this will often be the case. There will usually be one job which is the "worst". The other jobs can be sacrificed in order to alleviate problems (breakdowns) worsening this one job. For loose summed tardiness problems there will often only be one or a few critical jobs as well (if there are only few tardy jobs). As the problems become more tight the situation becomes more diffuse. There is no longer one critical job, but a number of them making the rescheduling problem more complex. The higher complexity of the rescheduling problem may result in a low implementation cost of the preschedule becoming more important for rescheduling performance than anything else, and decreasing the probability finding a good solution to the rescheduling problem close to the preschedule.

These observations are valuable if neighbourhood-based robustness is to be applied to real world scheduling problems or other combinatorial optimisation problems, since it gives the person working with the problem a hint as to whether neighbourhood-based robustness measures will work or not.

Appendix A. The decoders

A.1. The L_{max} decoder

The $L_{\max}$ decoder was used in the $T_{\max}$ and $T_{\sum}$ experiments. It uses a simple semi-active schedule builder, followed by a hillclimber minimising $L_{\max}$ (recall that when $L_{\max}$ is minimised, so is $T_{\max}$). When the hillclimber has terminated, the improved schedule is written back into the gene so that a subsequent semi-active decoding of the gene will give the improved schedule. The details of semi-active schedule building and the $L_{\max}$ hillclimber are in the following.

Semi-active schedule building is the simplest way of decoding the permutation with repetition representation. It is done by reading the gene in a left to right manner, scheduling one operation for every position of the gene. The gene (2, 3, 2, ...) will be decoded "first process the first operation of job 2, then the first operation of job 3, then the second operation of job 2, etc." An illustration of this procedure can be found on Fig. 6.

Any semi-active schedule can be described using the *graph representation*. The schedule is described by a directed graph, in which each operation is represented by a node, and in which the arcs represent "process prior to" relations. An example of this representation can be found in Fig. 7. The number in a node indicates which machine is supposed to process the operation. The nodes have been organised so that each row of nodes belongs to the same job. There are two kinds of arcs: the solid arcs represent the technological constraints specified by the problem, the dashed arcs represent machine processing orders, decided by the

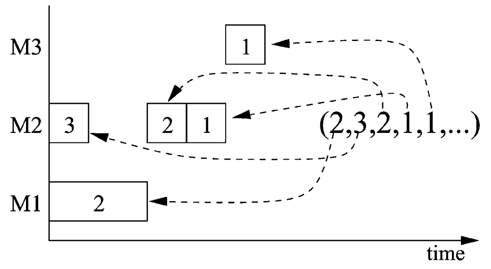

Fig. 6. Semi-active decoding. The genetic string is read in a left-to-right manner, for each position an operation is fixed in the Gantt-chart, without considering “holes” left earlier in the chart (consider the first operation of job 1). This is indicated by the arrows.

scheduler. A graph of this kind represents a semi-active schedule if there are no cycles in the graph, and all operations to be processed on the same machine are connected by a Hamiltonian path. In a graph like this, the operation just prior to operation o in the job sequence is called *the job predecessor* of o, *PJ*(o). The operation just after o in the job sequence is called *the job successor SJ*(o). In the same way, *the machine predecessor PM*(o) and *machine successor SM*(o) are the operations just prior to and just after o in the machine processing sequence.

The starting time of operation o, often called *the head* of o can be calculated:

$$h(o) = \max\,(h(PJ(o)) + \tau_{PJ(o)}, h(PM(o))+\tau_{PM(o)}), \tag{A.1}$$

where $h(PJ(o)$ and $\tau_{PJ(o)}$ are set to zero if $PJ(o)$ is undefined, and the same is done for $PM(o)$. We are assuming here that the problem has release times and initial set-up times of zero; removing this assumption is straight-forward.

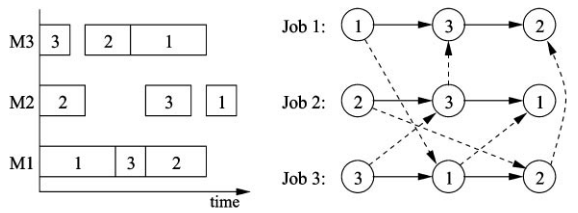

Fig. 7. Example graph representation and corresponding Gantt-chart.

If we denote by E_j the last operation of job J_j, the maximum lateness and maximum tardiness of a schedule can now be calculated:

$$L_{\max} = \max_{j=1,\dots,N}\,(h(E_j) + \tau_{E_j} - d_j) \tag{A.2}$$

$$T_{\max} = \max(L_{\max}, 0)$$

In this way, the $L_{\max}$ (or $T_{\max}$) job shop problem can be formulated as a graph problem in which the task is to choose the Hamiltonian paths (the machine processing sequences) such that (A.2) is minimised.

In order to estimate the effect of hillclimbing moves on schedule performance, the *lateness tail l*(o) of an operation in a semiactive schedule s will be defined. $l(o)$ satisfies

$$L_{\max}(s') \geq l(o) + \sigma$$

where s' is the schedule which has the same processing orders as s, and which is semiactive except that $h(o)$ has been increased by σ (the beginning of processing of o is delayed by σ). For σ large enough, $L_{\max}(s') = l(o) + \sigma$ holds. The lateness tail will later be used to estimate the effect of hillclimbing moves in the schedule. When an operation o is delayed ($h(o)$ is increased), it can affect the schedule performance in three ways:

1. It is the last operation of a job J_j. In this case we define

   $$l_J(o) = h(o) - \tau_o - d_j.$$

2. It is not the last operation of a job. The delay of o may delay the processing of its job successor *SJ*(o). In this case define

   $$l_J(o) = l(SJ(o)) + h(SJ(o)) - h(o) - \tau_o.$$

3. It has a machine successor *SM*(o). The delay of o may cause *SM*(o) to be delayed. Define

   $$l_S(o) = l(SM(o)) + h(SM(o)) - h(o) - \tau_o.$$

   If *SM*(o) is undefined, $l_S(o)$ is set to $-\infty$.

   The lateness tail is then defined to be

   $$l(o) = \max\,(l_J(o), l_S(o)).$$

An operation is termed *critical* in a schedule if it cannot be delayed without worsening schedule performance. Operation o is critical if $l(o) = L_{\max}$. The set

Table 7
The moves in the $\mathcal{N}_{\mathrm{hc}}$ neighbourhood

Block structure	Small block (o_1, o_2)	Block begin (o_1, o_2, SM_{o_2})	Block end (PM_{o_1}, o_1, o_2)
Permutations	$(o_2, o_1)^{\heartsuit}$	$(o_2, o_1, SM_{o_2})^{\heartsuit}$	$(PM_{o_1}, o_2, o_1)^{\heartsuit}$
		$(o_2, SM_{o_2}, o_1)^{\dagger}$	$(o_2, PM_{o_1}, o_1)^{\dagger}$
		$(SM_{o_2}, o_2, o_1)^{\dagger}$	$(o_2, o_1, PM_{o_1})^{\dagger}$

of critical operations in a schedule is called *the critical path*. A number of consecutive critical operations on the same machine are called a *critical block*.

The hillclimber based on the graph-representation optimising $L_{\max}$ will now be outlined. The hillclimber is an extension of the makespan hillclimber used in [15]. The hillclimber is best described using two neighbourhoods. The neighbourhood $\mathcal{N}_{\mathrm{hc}}$ includes moves which could lead to cycles in the schedule graph (leading to infeasible schedules) while the neighbourhood $\mathcal{N}_{\mathrm{hc,feasible}} \subset \mathcal{N}_{\mathrm{hc}}$ includes only feasible moves. It can be shown, [15], that a hillclimbing move (the permutation of a small number of operations in a schedule) can only improve performance if it includes operations at the beginning or the end of a critical block. If the operations o_1 and o_2 are consecutive operations in the beginning of a critical block, all permutations of (PM_{o_1}, o_1, o_2) in which o_1 and o_2 are reversed are in $\mathcal{N}_{\mathrm{hc}}$. If o_1 and o_2 are consecutive operations at the end of a critical block, all permutations of (o_1, o_2, SM_{o_2}) in which o_1 and o_2 have been reversed are in $\mathcal{N}_{\mathrm{hc}}$. The moves in $\mathcal{N}_{\mathrm{hc}}$ are shown in Table 7. If the length of the critical block is two the only move tried is the permutation in the column "small block". If the length of the block is three or more, the permutations labelled "block begin" are tried at the beginning of the block, while the "block end" permutations are tried at the end of the block.

Bounds on schedule performance after the moves can be made using local considerations. In the following, unprimed variables refer to the schedule s before the move, while primed variables refer to the schedule s after the move. The operators *PJ*, *PM*, *SJ* and *SM* refer to the processing sequences before the move.

The "small block" move has been visualised in Fig. 8. The heads of o_1 and o_2 after the "small block"

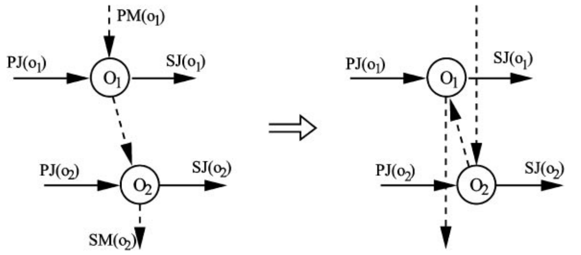

Fig. 8. Visualisation of the "small block" move.

move can be calculated as follows:

$$h'(o_2) = \max\,(h(PM(o_1)) + \tau_{PM(o_1)}, h(PJ(o_2)) + \tau_{PJ(o_2)}),$$

$$h'(o_1) = \max\,(h'(o_2) + \tau_{o_2}, h(PJ(o_1)) + \tau_{PJ(o_1)}).$$

The lateness tail of o_1 and o_2 after the move can be calculated:

$$l'(o_1) = \max(l_J(o_1), l(SM(o_2)) + h(SM(o_2)) - h'(o_1) - \tau_{o_1}),$$

$$l'(o_2) = \max(l_J(o_2), l'(o_1) + h'(o_1) - h'(o_2) - \tau_{o_2}).$$

The lateness of the schedule after the move $L_{\max}(s')$ is bounded by

$$L_{\max}(s') \geq L_{\max,\mathrm{bound}}(s') = \max(l'(o_1), l'(o_2))$$

If o_1 or o_2 is still critical after the move, $L_{\max}(s') = \max l'(o_1), l'(o_2)$ will be the case. Bounds on schedule performance for the other moves can be made in similar ways.

With respect to schedule feasibility, the moves marked $\heartsuit$ are safe, since when performed on critical operations they cannot generate a cycle in the graph [15]. However, the moves marked '†' can generate

```
calculate performance $L_{max}(s)$ of current solution $s$
set $continue$ to true
while ($continue$) do
    set the priority queue $Q$ to empty
    for $s' \in \mathcal{N}_{hc,feasible}(s)$ do
        if $L_{max,est}(s') < L_{max}(s)$ then
            insert $s'$ in $Q$ with priority $L_{max,est}(s')$
    od
    set $continue$ to false
    while $s$ has not been updated and $Q$ is not empty do
        delete $s'$ in $Q$ with lowest priority, calculate $L_{max}(s')$
        if $L_{max}(s') < L_{max}(s)$ then (*)
            update $s$ by setting $s = s'$, set $continue$ to true
    od
```

Fig. 9. Pseudo-code for the $L_{\max}$ hillclimber.

cycles. To avoid this from happening, the order in which the moves are considered is important. It can be shown, [5], that if one of the moves marked '†' generates a cycle in the graph, the performance bound $L_{\max,\text{est}}(s^{\heartsuit})$ on the move marked ♡ at the same place will be at least as low as the performance bound $L_{\max,\text{est}}(s^{\dagger})$ of that move. For this reason, when the beginning or end of a critical block is considered, only the move which has the lowest performance bound is included in $\mathcal{N}_{\text{hc,feasible}}$. If two moves are tied, the move marked ♡ is included in $\mathcal{N}_{\text{hc,feasible}}$.

In each step of the hillclimber, the $L_{\max,\text{est}}$ bound of every move in $\mathcal{N}_{\text{hc,feasible}}$ is calculated, and the move with the lowest $L_{\max,\text{est}}$ bound is implemented. If this move is found not to improve performance, the move with the second lowest $L_{\max,\text{est}}$ is implemented, and so forth. This continues until no improving moves can be found. Pseudo-code for the hillclimber can be seen in Fig. 9.

A.2. The $R_{L_{\max}}$ decoder

The $R_{L_{\max}}$ decoder is very similar to the $L_{\max}$ decoder. It uses a simple semi-active decoding, which is followed by a hillclimber. When the hillclimber has finished, the improved schedule is written to the gene in such a way that subsequent semiactive decoding of it will give the improved schedule.

The $R_{L_{\max}}$ fitness landscape is expected to have approximately the same large scale features as the $L_{\max}$ landscape, since $R_{L_{\max}}$ is simply a weighted average in a small neighbourhood of $L_{\max}$. For this reason, optima for the $R_{L_{\max}}$ problem should be expected to be close to optima of the $L_{\max}$ problem. Since the hillclimber used on $L_{\max}$ problems is known to be able to locate these optima, it makes sense to modify it as little as possible, and let it search the same neighbourhood, but let it optimise $R_{L_{\max}}$ instead of $L_{\max}$. For these reasons, the $R_{L_{\max}}$ hillclimber is identical to the $L_{\max}$ hillclimber (Fig. 9), except that the line marked (∗) has been replaced by

if $RR_{L_{\max}}(s') < RR_{L_{\max}}(s)$ then,

where $RR_{L_{\max}}(s)$ is an upper bound of $R_{L_{\max}}(s)$:

$$RR_{L_{\max}}(s) = \frac{1}{|\mathcal{N}_1(s)|} \times \sum_{s' \in \mathcal{N}_1(s)} \max\left(L_{\max,\text{est}}(s'), L_{\max}(s)\right).$$

Due to the definition of $L_{\max,\text{est}}$, it is clear that $\max(L_{\max,\text{est}}(s'), L_{\max}(s)) \geq L_{\max}(s')$, meaning that $RR_{L_{\max}} \geq R_{L_{\max}}$. Since the $R_{L_{\max}}$ optima are probably coincidental with or close to local $L_{\max}$ optima, for locally optimal solutions we expect $L_{\max,\text{est}}(s) = L_{\max}(s)$ for the majority of schedules in $\mathcal{N}_1(s)$,

```
set $S$ to the empty schedule
set $A = \{o_{i1} | 1 \leq i \leq n\}$, the set of operation which are first on
      a machine
while ($A$ is not empty) do
   find the operation $o \in A$ with the earliest possible
     completion time $h(o) + \tau_o$ (if two or more operations are
     tied, pick one at random)
   set $M^*$ to the machine on which $o$ is to be processed
   find the earliest possible starting time $h(o')$ of an
      operation schedulable on $M^*$
   set $Q$ to the set of operations from $A$ which are to be
     processed on $M^*$ and have potential starting times
      $\min(h(o') + \delta((h(o) + \tau_o) - h(o')), h(o) + \tau_o - 1)$ or earlier
   pick the operation $o^*$ from $Q$ which occurs leftmost in the
     gene $g$.
   add operation $o^*$ to $S$ with starting time $h(o^*)$
   remove $o^*$ from $g$
   if a job successor $SJ(o^*)$ of $o^*$ exists add it to $A$
od
```

Fig. 10. The modified GT algorithm used for decoding in the total flowtime experiments.

meaning that the $RR_{L_{\max}}$ bound becomes a reasonable approximation of $R_{L_{\max}}$.

A.3. The tunable hybrid decoder

The tunable active decoder used in the total flowtime experiments (normal as well as robust) is due to [3]. It is a special version of the GT algorithm. The ordinary GT algorithm produces active schedules, while the algorithm described here has a parameter δ which controls a bias towards non-delay schedules. A non-delay schedule is a schedule in which no machine is ever kept idle if an operation is ready to start on it. It can be shown [8] that for all total flowtime problems, there will be an optimal active schedule. Non-delay schedules do not have this property, but on average non-delay schedules have higher performance than active schedules. In [3] it was demonstrated that the tunable hybrid decoder outperforms a decoder based on the classical GT algorithm on hard problems. The decoder works by building up the schedule operation by operation, using the order in which the operations are stated in the genetic sequence to decide which operations to schedule when.

The decoder is described in detail in [3]. Here, a brief description of it will be given. Pseudo-code for the tunable hybrid decoder is displayed in Fig. 10. An operation is said to be *schedulable* if all of its predecessors in the technological constraints have already been scheduled. In the algorithm the set of schedulable operations is kept in A. In the main loop, the machine M^* with the earliest potential finishing time $h(o) + \tau_o$ of an operation o is located. Scheduling an operation on this machine with a starting time earlier than $h(o) + \tau_o$ will produce an active schedule. The earliest possible starting time $h(o)$ of a schedulable operation on M^* is located. Scheduling an operation with this starting time on M^* will produce a non-delay schedule. The algorithm produces schedules in between active and non-delay schedules by creating the set Q of schedulable operations on M^* which have potential starting times earlier than a value somewhere between $h(o')$ and $h(o) + \tau_o$. The parameter $\delta \in [0; 1]$ controls this value, the extreme value $\delta = 0$ indicates that only non-delay schedules can be produced, the value $\delta = 1$ indicates that all active schedules can be produced. In the experiments the value $\delta = 0.5$ was used, since values in that ball-park were found to give a good performance in [3].

References

[1] I. Al-Harkan, On merging sequencing and scheduling theory with genetic algorithms to solve stochastic job shops, Ph.D. Thesis, University of Oklahoma, OK, 1997.
[2] C. Bierwirth, A Generalized Permutation Approach to Job Shop Scheduling with Genetic Algorithms, OR Spektrum 17 (1995).
[3] C. Bierwirth, D.C. Mattfeld, Production scheduling and rescheduling with genetic algorithms, Evolutionary Computation 7 (1999).
[4] J. Branke, Creating Robust Solutions by Means of Evolutionary Algorithms, Parallel Problem Solving From Nature V, Lecture Notes in Computer Science, Vol. 1498, Springer, New York, 1998, pp. 119–128.
[5] M. Dell'Amico, M. Trubian, Applying tabu search to the job-shop scheduling problem, Annals of Operations Research 41 (1993) 231–252.
[6] H.-L. Fang, Genetic Algorithms in Timetabling and Scheduling, Ph.D. Thesis, Department of Artificial Intelligence, University of Edinburgh, Edinburgh, 1994.
[7] H. Fisher, G.L. Thompson, Probabilistic learning combinations of local job-shop scheduling rules, in: J.F. Muth, G.L. Thompson (Eds.), Industrial Scheduling, Prentice-Hall, Englewood Cliffs, NJ, 1963, pp. 225–251.
[8] S. French, Sequencing and Scheduling, Ellis Horwood, Chichester, UK, 1982.
[9] E. Hart, P. Ross, J. Nelson, Producing robust schedules via an artificial immune system, in: Proceedings of the IEEE World Congress on Computational Intelligence, 1997.
[10] M.T. Jensen, Generating robust and flexible job shop schedules using genetic algorithms, IEEE Transactions on Evolutionary Computation, submitted for publication.
[11] M.T. Jensen, Neighbourhood based robustness applied to tardiness and total flowtime job shops, in: M. Schoenauer, et. al. (Eds.), Proceedings of the LNCS on Parallel Problem Solving from Nature — PPSN VI , Vol. 1917, LNCS, 2000, pp. 283–292.
[12] P. Kouvelis, G. Yu, Robust Discrete Optimization and Its Applications, Kluwer Academic Publishers, Dordrecht, 1997.
[13] S. Lawrence, Resource constrained project scheduling: an experimental investigation of heuristic scheduling techniques (Suppl.), Graduate School of Industrial Administration, Carnegie-Mellon University, 1984.
[14] V.J. Leon, S.D. Wu, R.H. Storer, Robustness measures and robust scheduling for job shops, IIE Transactions 26 (5) (1994) 32–43.
[15] D.C. Mattfeld, Evolutionary Search and the Job Shop. Production and Logistics, Physica-Verlag, 1996.
[16] S. Tsutsui, A. Ghosh, Genetic algorithms with a robust solution searching scheme, IEEE Transactions on Evolutionary Computation 1 (3) (1997) 201–208.
[17] S.D. Wu, E. Byeon, R.H. Storer, A graph-theoretic decomposition of the job shop scheduling problem to achieve scheduling robustness, Operations Research 47 (1) (1999) 113–124.