Total tardiness minimisation in permutation flow shops: a simple approach based on a variable greedy algorithm

Jose Manuel Framinan, Rainer Leisten

Total tardiness minimisation in permutation flow shops: a simple approach based on a variable greedy algorithm

Journal:	*International Journal of Production Research*
Manuscript ID:	TPRS-2006-IJPR-0476.R1
Manuscript Type:	Original Manuscript
Date Submitted by the Author:	05-Feb-2007
Complete List of Authors:	Framinan, Jose; University of Seville, Industrial Management, School of Engineering Leisten, Rainer; University Duisburg-Essen, Production Management
Keywords:	SCHEDULING, FLOW SHOP SCHEDULING, META-HEURISTICS
Keywords (user):	

Total tardiness minimisation in permutation flow shops: a simple approach based on a variable greedy algorithm

J.M. FRAMINAN[†] * and R. LEISTEN[‡]

Word count (incl. references): 5,681

Keywords: Scheduling, flow shop, due dates, greedy algorithm.

[†] Industrial Management, School of Engineering, University of Seville, Ave. Descubrimientos s/n, E41092 Seville, Spain; jose@esi.us.es. Tel: +34 954487214, Fax: + 34 954487329.

[‡] Production and Operations Management, Department of Technology and Operations Management, Mercator School of Management, University Duisburg-Essen, Lotharstr. 65, D-47057 Duisburg, Germany; rainer.leisten@uni-due.de.

* To whom correspondence should be addressed.

Total tardiness minimisation in permutation flow shops: a simple approach based on a variable greedy algorithm

J.M. FRAMINAN[†] * and R. LEISTEN[‡]

In this paper, we address the problem of scheduling jobs in a permutation flowshop with the objective of minimizing the total tardiness of jobs. To tackle this problem, we suggest employing a procedure based on a greedy algorithm that successively iterates over an increasing number of candidate solutions. The computational experiments carried out show this algorithm to outperform the best existing one for the problem under consideration. In addition, we carry out some tests to analyse the efficiency of the adopted design.

1. Introduction

A flowshop is a common manufacturing layout where the operation sequence for all jobs is the same. Additionally, it is usually assumed that jobs cannot overtake one another from machine to machine and therefore, once a sequence is fixed for all jobs on the first machine, this sequence is maintained for all machines (permutation flowshop).

[†] Industrial Management, School of Engineering, University of Seville, Ave. Descubrimientos s/n, E41092 Seville, Spain; jose@esi.us.es.

[‡] Production and Operations Management, Department of Technology and Operations Management, Mercator School of Management,, University Duisburg-Essen, Lotharstr. 65, D-47057 Duisburg, Germany; rainer.leisten@uni-due.de.

* To whom correspondence should be addressed.

Several objectives may be considered when scheduling in a flowshop. Without any doubt, the most widely studied is the makespan objective (see e.g. Reza Hejazi and Saghafian 2005; or Ruiz and Maroto 2005 for recent reviews on this problem). In contrast, significantly less attention has been paid until recent years to other objectives, particularly those involving due dates. Among due-date related objectives, minimisation of total tardiness aims to find schedules that satisfy the due dates committed to customers and it is therefore of significance and importance to real life considerations (Hasija and Rajendran 2004).

In this paper, we propose an algorithm with the objective of minimizing the total tardiness of jobs. The algorithm is based on a variable greedy algorithm concept. A salient feature of the algorithm is that it employs a tailored design for the problem, which is based on the slack-concept, i.e. jobs with minimum slacks are removed from a given solution in order to find for them a better position in the schedule. Besides, the algorithm is parameter-free (with the exception of the running time), so it does not require any parameter setting and parameter tuning. This simplicity and the lack of control parameters is a desired characteristic of any heuristic approach as makes it more user-friendly (Hansen and Mladenović 2001).

The remainder of the paper is organised as follows: in the next section we review in more detail the problem under consideration and the contributions up-to now. Next, we present the proposed algorithm for the problem. We then discuss and set up an extensive computational experiment in order to validate the suggested approach and to compare with the best-so-far ones. Finally, we present the main conclusions and some future research lines.

2. Problem statement and literature review

The problem under consideration refers to the scheduling of n jobs in a permutation flow shop consisting of m machines. Each job i has a processing time on each machine j, which is denoted by p_{ij}. Additionally, a due date d_i for each job i is given. In a permutation flowshop, the completion time of the i-th job in a given schedule σ on a machine j, $C_{[i]j}(\sigma)$, is given by the following recursive equations:

$$C_{[i]j}(\sigma) = \max\{ C_{[i-1]j}(\sigma) ; C_{[i]j-1}(\sigma) \} + p_{ij}$$

$$C_{[0]j}(\sigma) := 0; \; C_{[i]0}(\sigma) := 0;$$

With these equations, $C_i(\sigma)$, the completion time of a job i (i.e. the earliest time when this job can be can released to the customer), can be computed. T_i, the tardiness of a job i is then defined as the maximum of the difference between its completion time and its due date, and zero, i.e.: $T_i = \max\{ C_i(\sigma) - d_i ; 0\}$. The total tardiness $TARD$ is the sum of the tardiness across all jobs: $TARD = \Sigma T_i$.

Clearly, tardiness minimisation is related to maintaining high service levels, as it aims to respect the committed due dates. Research on the topic has been carried out by Gelders and Sambandam (1978), Kim (1993), Parthasarathy and Rajendran (1998), Armentano and Ronconi (1999), and Hasija and Rajendran (2004).

Gelders and Sambandam develop constructive heuristics for the problem, while Kim suggests an algorithm based on the NEH heuristic (Nawaz *et al.*, 1983) for makespan minimisation. His proposal is not compared to these from Gelders and Sambandam. Parthasarathy and Rajendran present two versions of Simulated Annealing (SA) for the problem and compare them with those of Gelders and Sambandam, and that of Kim. They found their proposal to outperform all these heuristics. Armentano

and Ronconi design a tabu search procedure that is compared with an adaptation of the NEH heuristic for the problem under consideration, but not with other heuristics available for the problem. Hasija and Rajendran propose a SA algorithm for the problem (HR in the following) by introducing new perturbation and improvement schemes. They compare their algorithm with the most efficient version of the heuristic by Parthasarathy and Rajendran (PR in the following) and the one by Armentano and Ronconi using an adaptation of the benchmark problems by Taillard (1993). The results of this computational experience show that (at least for the test-bed employed), the heuristic by Hasija and Rajendran clearly outperforms the existing ones and henceforth it should be considered the best heuristic for the problem up to date. In their computational experience, the heuristic by Parthasarathy and Rajendran also outperforms the one by Armentano and Ronconi.

As a summary, it can be concluded that the most efficient heuristics for the problem under consideration are PR and HR. Both heuristics use the same heuristic for generating the initial sequence, i.e. the so-called Earliest Apportioned Due Date Method (EADD). In EADD, jobs are ordered for each machine in ascending order of a value proportional to the processing times of the job and its due date. In this manner, one order of jobs is obtained for each machine. Among these, the ordering with the least value of total tardiness is chosen. PR employs a Random Insertion Perturbation Scheme (RIPS) for generating new solutions from a given sequence in the following manner: for each job in the seed sequence, two random positions are generated (one on its right and other on its left) to insert it, thus obtaining two new sequences. Obviously, the first (last) job cannot be inserted on any position on its left (right), hence only one new sequence can be obtained from these extreme positions. In total, $2 \cdot (n - 1)$ new sequences are obtained. Among them, the one with the least total tardiness is selected.

HR employs sophisticated mechanisms for the generation of new solutions, namely Job Index Based Insertion Scheme (JIBIS), Job Shift-Based (JSB) scheme, and Probabilistic Step Swap (PSS) perturbation scheme. Using JIBIS, each job (starting from job 1 to job n) in a seed sequence is selected for insertion in all possible positions of the sequence. If the total tardiness of the so-obtained sequence is lower than the one for the seed, it replaces the seed. Note that JIBIS obtains a solution which is not worse than the initial seed. In the JSB scheme, a job in the seed sequence is chosen based on a probabilistic function and then inserted either on the right or on the left of its original position (the same probability is assigned to the insertion on the left and on the right). The job in the seed sequence is chosen according to the relative displacement of the job's position in the current sequence from the job's position in the best-so-far sequence. Finally, PSS generates sequences from a seed sequence by probabilistically swapping jobs close to each other, and far from each other. Among them, the one with the least total tardiness is chosen. In Figures 1 and 2 we present the detailed pseudocode of both heuristics, keeping the same steps and the programme structure as in the original descriptions.

[INSERT FIGURE 1 ABOUT HERE]

[INSERT FIGURE 2 ABOUT HERE]

3. Proposed heuristic

In recent years, a number of general-purpose heuristic approaches (also known as meta-heuristics) have been introduced to tackle combinatorial optimisation problems. Two of the newest approaches are the Iterated Greedy (IG) algorithm and the Variable

Neighbourhood Search (VNS). As our proposal combines features of both meta-heuristics, we first briefly introduce these.

IG (see e.g. Marchiori and Steenbeek 2000, and Ruiz and Stützle 2006) generates a sequence of solutions by iterating over a constructive heuristic using two phases, namely destruction and construction. During the destruction phase, k components (destruction level) are removed from a given solution π, thus obtaining a set of removed components $S_k(\pi)$ and a partial solution $\pi_{R(k)}$ formed by the non-removed components of π. During the construction phase, a greedy constructive heuristic is applied to the problem in order to obtain a (possibly new) solution π' by adding to $\pi_{R(k)}$ the components in $S_k(\pi)$. Then, it is decided whether or not π' substitutes π as the incumbent solution for the next iteration according to some pre-defined acceptance criterion (e.g. if π' is better than π). In addition, a local search procedure can be implemented before this step in order to improve π'.

IG is closely related to another technique, Iterated Local Search (ILS), in which the algorithm iterates over a local search neighbourhood (see e.g. Lourenço *et al*. 2002). To the best of our knowledge, IG has been only applied to few problems, including crew scheduling (Marchiori and Steenbeek 2000) and flowshop scheduling with makespan objective (Ruiz and Stützle 2006). Salient features of IG are its simplicity and the possibility to use a tailored heuristic to obtain good solutions for the problem under consideration.

Clearly, an obvious drawback of the method is the stagnation due to the 'greedy' nature of the procedure. In order to overcome this problem, Ruiz and Stützle (2006) adopt a simulated annealing-based acceptance criterion, i.e.: accepting worse solutions with some probability depending on the number of iterations of the algorithm. A similar approach is also adopted in Marchiori and Steenbeek (2000).

In this paper, we propose a different procedure for overcoming local optima, which is based on the VNS concept (Mladenović and Hansen 1997). VNS uses a finite set of pre-selected neighbourhood structures that we denote by N_k (k= 1, …, k_{max}). Let us denote by N_k (x) the set of solutions in the neighbourhood of x. The pseudo-code of the basic structure of a VNS algorithm can be described as follows:

(1) Generate an initial solution x;

(2) While termination conditions are not met, repeat:

(2.1) Set $k \leftarrow 1$;

(2.2) Until $k = k_{max}$, repeat:

(2.2.1) Find the best neighbour x' of x ($x' \in N_k(x)$);

(2.2.2) If x' is better than x, then set $x \leftarrow x'$ and set $k \leftarrow 1$;

otherwise set $k \leftarrow k + 1$;

Sophisticated features can be added to this most basic procedure. For these, we refer the reader to Hansen and Mladenović (2001). VNS has shown to be efficient for a number of NP-hard problems, including some scheduling problems (see e.g. Gagné *et al*. 2003, Schuster and Framinan 2003, and Tasgetiren *et al*. 2004). In our proposal, we use the VNS concept of varying the neighbourhood, but we apply it to the destruction and construction phases of the IG algorithm. To do so, a greedy mechanism specifically tailored for the problem is described in the next section.

3.1. Slack-based greedy algorithm

For a schedule $\sigma := (\sigma_1, ...)$, let us define $s_i(\sigma)$ the slack of job σ_i in the sequence σ in the following manner: $s_i(\sigma) := d_{\sigma_i} - C_{\sigma_i,m}$. It is clear that $s_i(\sigma)$ is sequence-

dependent and that it is possible to express the total tardiness in terms of the slacks, i.e.: $TARD(\sigma) = \sum_i \max[-s_i(\sigma);0]$. Consequently, if a job σ_i has a negative value for $s_i(\sigma)$, then this job is contributing to increase the total tardiness.

Based on this idea, it is possible to define a greedy algorithm to obtain a sequence σ' from another sequence σ in the following manner:

(1) Remove from σ the k jobs with lowest $s_i(\sigma)$. Store them in R_k. Let μ be the remaining subsequence from σ after removing the k jobs.

(2) For each job $\sigma_r \in R_k$, repeat:

(2.1) Insert σ_r in all possible slots of μ (starting from the first position). Let ν be the best subsequence obtained, and b the position of σ_r in ν.

(2.2) Perform an adjacent pairwise exchange among all jobs in positions b+1 to (n – k + 1). Let π be the best sequence obtained by this approach.

(2.3) If π improves ν, then set $\mu \leftarrow \pi$; otherwise set $\mu \leftarrow \nu$.

(3) Set $\sigma' \leftarrow \mu$.

According to this approach, it is possible to define k different types of movements, depending on the number of jobs to be removed. Aside, this construction mechanism is related to the heuristic by Kim, which in turn is based on the NEH heuristic. Nevertheless, we added step (2.2) as we try to allow some small perturbation of the solution for those jobs after the one for which the minimum tardiness has been found.

In addition, it is possible to introduce a speed up in order to reduce the computational effort. When executing step (2.1), instead of calculating the completion times from scratch for every slot j (j = 1, …, n – k + 1) in which the job σ_r is inserted, the completion time in all machines for jobs preceding job (j – 1) do not need to be

recomputed. By this procedure, the computation times are reduced roughly to the half of the time.

3.2. Insertion-based local search

In order to improve the solution provided by the slack-based greedy algorithm, a local search in the neighbourhood of the solution is accomplished. Usual neighbourhood definitions for flowshop scheduling problems include adjacent pairwise interchange (exchanging the positions of two adjacent jobs), general pairwise interchange (exchanging the positions of each two jobs), and insertion (removing one job from its position and inserting it into a different one). Among these possibilities, the experiences conducted produced the best result for the last one. Consequently, an insertion-based neighbourhood is adopted.

The local search algorithm then obtains n positions at random (without repetition), and for each position, the job occupying this position is removed and inserted in all possible slots. If the best solution out of the n insertions is better than the current solution, then it replaces the latter as the best incumbent solution. The pseudo-code of this procedure is as follows:

(1) Let σ be the current solution.

(2) For i = 1 to n do:

(2.1) Obtain k = random (1, n) (without repetition).

(2.2) Remove σ_k, the job in position k, in σ and insert it in all possible slots. Let σ' be the best solution found by this way.

(2.3) If σ' improves σ, then set $\sigma \leftarrow \sigma'$.

(3) Return σ as solution.

3.3. *Pseudo-code of the algorithm*

The algorithm operates as follows:

(1) While computation time does not exceed a given time limit, repeat:

(1.1) Generate a solution x at random.

(1.2) Set $k \leftarrow 1$;

(1.3) Until $k = (n - 1)$, repeat:

(1.3.1) Obtain x' from x according to the procedure explained in Section 3.1. i.e. using the slack-based greedy algorithm.

(1.3.2) Obtain x'' from x' according to the procedure explained in Section 3.2. , i.e. using the insertion-based local search.

(1.3.3) If x'' is better than x, then set $x \leftarrow x''$ and set $k \leftarrow 1$;

otherwise set $k \leftarrow k + 1$;

The algorithm starts with a random solution. Then, this solution is improved by a stepwise approach that systematically enhances the capabilities of the greedy algorithm until it shuffles all jobs. If a better solution is found, then the process starts again. Otherwise, if the greedy algorithm has been expanded without reaching a better solution (i.e., $k = n - 1$), then a new random solution is tried. This allows a diversification of the exploration once all greedy structures have been tried. Note that this algorithm is parameter-free, with the exception of the CPU time.

With respect to the stopping criterion, it has been chosen as the decision interval for scheduling problems is usually very short and hence the total running time of a heuristic is a critical issue. However, replacing the original criterion by a different one

based e.g. on the number of solutions evaluated, the number of iterations without improvement, etc. is straightforward.

4. Computational experiments

In this section we present the results of the computational experiments carried out in order to validate our proposed procedure. We first discuss the experimental design and particularly the setting of due dates in the problem instances. We then compare our proposal with the best-so-far algorithms for the problem under consideration. Finally, we analyse how the different design decisions in the algorithm (namely the design of the destruction phase, and the concept of 'variable destruction') affect its performance.

4.1. Experimental design

It is clear that the design of the experimental test-bed may have a huge influence on the results. Particularly, being a due-date problem, several types of problems can be distinguished, each one with a different degree of difficulty. Henceforth, we first review the different settings for due dates that have been adopted in the literature.

Rather different procedures for establishing due dates in permutation flowshops can be found in the papers by Gelders and Sambandam (1978), Armentano and Ronconi (1999), Demirkol *et al*. (1998), and Hasija and Rajendan (2004). In all these, the processing times of the problems are obtained from a uniform distribution. In fact, the [1, 99] distribution is employed for all test-beds except for the one by Demirkol *et al*., where processing times are drawn from a [1, 200] uniform distribution. Therefore, in the following we focus our discussion on how due dates are established in these papers.

Armentano and Ronconi suggest using the following distribution to generate due dates:

$$Uniform[P(1-T-\frac{R}{2}), P(1-T+\frac{R}{2})]$$

where T and R are the tardiness factor of jobs and the dispersion range of due dates, respectively, and P is a lower bound of the makespan, defined as follows:

$$P = \max\left\{\max_{1\le j\le m}\left\{\sum_{i=1}^{n} p_{ij} + \min_{i}\sum_{l=1}^{j-1} p_{il} + \min_{i}\sum_{l=j+1}^{m} p_{il}\right\}, \max_{i}\sum_{j=1}^{m} p_{ij}\right\}$$

With the above setting, they define four scenarios depending on the values of T and R. This way of generating due dates is nearly identical to the one employed by Demirkol *et al*. who build a test-bed for several due date problems, including 160 problem instances of flowshops with due dates. They employ different T and R factors that produce much tighter due dates as compared with the paper by Armentano and Ronconi. Additionally, from a bigger subset of problems, the 160 hardest instances were collected.

Gelders and Sambandam propose a different method for generating due dates, consisting of employing the following uniform distribution:

$[\sum_{j=1}^{m} p_{ij}, \sum_{j=1}^{m} p_{ij} + 0.5 \times$ mean processing time of a job on machine $m]$.

This due-date setting would result in approximately 50% of the jobs being late, and it has been used also by Chakravarthy and Rajendran (1999), and Framinan and Leisten (2006) due to its ability to produce tight due dates.

Finally, we describe the due date setting in the test-bed developed by Hasija and Rajendran, which is based on Taillard's test-bed (Taillard 1993). Taillard's test-bed consists of a collection of 120 flowshop problem instances chosen so as 'good' makespan values are difficult to obtain. In this test-bed, $(n,m) \in \{(20,5), (20,10),$

(20,20), (50,5), (50,10), (50,20), (100,5), (100,10), (100,20), (200,10), (200,20), (500,20)}. For each problem size, 10 instances are available.

In order to provide due dates d_i for the problem under consideration, Hasija and Rajendran propose the following method to generate them:

$$d_i = [1 + 3u] \sum_{j=1}^{m} p_{ij}$$

where u is a real random number uniformly distributed in the interval [0,1].

In view of the different contributions, we select three different test-beds for the comparison of the heuristics. For these, we use the following acronyms:

DMU: The test-bed by Demirkol *et al*. As mentioned before, this test-bed consists of 160 instances, where $(n,m) \in \{(20,15), (20,20), (30,15), (30,20), (40,15), (40,20), (50,15), (50,20)\}$. For each problem size, there are 20 instances.

T-GS: The test-bed obtained using the processing times of the test-bed of Taillard, and due dates generated according to the method proposed by Gelders and Sambandam. In view of the prohibitive computation times required to solve the 10 biggest instances of Taillard's test-bed (instances ta110-ta120, n = 500) by using the existing algorithms for the problem, we exclude them from T-GS. Therefore, the test-bed under consideration consists of 110 instances.

T-HR: The test-bed obtained using the processing times of the test-bed of Taillard, and due dates generated according to the method proposed by Hasija and Rajendran. Analogously to the previous test-bed, the 10 biggest instances by Taillard are removed.

4.2. Comparison with existing heuristics

As mentioned in the introduction, the procedure proposed by Hasija and Rajendran (HR in the following) is the best algorithm so far for the problem under consideration,

followed by the procedure of Parthasarathy and Rajendran (PR in the following). In order to compare them with our algorithm, we employ the three test-beds described in the previous section. Note that neither HR nor PR have a simple termination criterion (such as the computation time, or the number of iterations without improvement), but use a rather sophisticated combination of parameters depending on the temperature (number of iterations, number of accepted moves, and number of sequences generated). Moreover, it is not possible to establish a fair comparison based on the number of solutions evaluated by each algorithm, as our proposal constructs (complete) solutions by exhaustively selecting and exploring partial sequences, therefore the number of final solutions evaluated is much lower than in PR and HR (less than 1% in the DMU test-bed, as we will show later). Consequently, the fairest comparison with HR is running this algorithm exactly with the same parameter setting described by the authors, and then compare its results to those obtained by the new heuristic proposal, using as termination criterion the same CPU time taken by the HR algorithm for this problem.

Each instance is solved by 24 runs of each algorithm. The average and the best value of the total tardiness for each algorithm and instance are obtained. Then, the quality of the solutions obtained for each instance and each heuristic i is measured in terms of RPD (Relative Percentage Deviation) with respect to the average and the best solution, in the following manner:

$$\overline{PRD}_i = \frac{(\overline{T}_i - T^*)}{T^*} \times 100$$

$$PRD^*_i = \frac{(T^*_i - T^*)}{T^*} \times 100$$

where $\overline{T}_i$ is the average tardiness obtained over the 24 runs of algorithm i, T^*_i is the best tardiness among the 24 runs, and T^* is the best known solution for this instance. For all instances of a given problem size, the average and the standard deviation of both

$\overline{PRD_i}$ and PRD^*_i are obtained, and shown in Tables 1, 2, and 3 for the DMU, T-GS, and T-HR test-beds, respectively.

The tables show the quality of the solutions obtained in terms of the $ARPD_i$ (Average Relative Percentage Deviation) of heuristic i for each problem size. $ARPD_i$ values are obtained by averaging across a number of instances the Relative Percentage Deviation (RPD_i), measured as follows:

$$RPD = \frac{(RPD_i - RPD_{\min})}{RPD_{\min}} \times 100$$

where RPD_i is the tardiness obtained by the application of the heuristic i to the problem instance, and $RPD_{\min}$ is the minimum tardiness found by any heuristic for this instance.

[INSERT TABLE 1 ABOUT HERE]

[INSERT TABLE 2 ABOUT HERE]

[INSERT TABLE 3 ABOUT HERE]

As can be seen from Tables 1, 2, and 3, the algorithm proposed exhibits a very good performance for the three test-beds. Both the best solution (out of 24 runs) and the average obtain better results than those of HR and PR.

In order to test the performance of our proposal for different CPU times, we change the original design of PR and HR in order to introduce CPU time as stopping criterion. Then, we run all three algorithms with fractions of the CPU time invested for the original design. More specifically, let *TIME* be the CPU time required (average over 24 runs) for a given instance in the experiments presented in Tables 1, 2, and 3. Then we solve the DMU test-bed with all three algorithms for a CPU time of *TIME*/2, and

TIME/4. The quality of the solutions obtained is shown in Table 4 in terms of the best solution and the average over 24 runs of each algorithm.

[INSERT TABLE 4 ABOUT HERE]

As can be seen from Table 4, the algorithm outperforms HR and PR for the different CPU times. It is interesting to note that, for all three algorithms, the results for different CPU times are similar with respect to the best solution. Regarding the average solution, there is a steady improvement for our proposal, but not for HR and PR, which may indicate that both algorithms could stagnate for longer CPU times.

Finally, in Table 5 we give the average number of solutions explored by HR and our proposal in the DMU test bed for each problem size (the CPU allowed is that of the original HR description, see Table 1). It has to be noted that this value is not an indicator of the computational effort of each algorithm, as in the case of our proposal, most of the effort is devoted to insert the jobs removed in the destruction phase in the different slots of the remaining schedules. Therefore, the number of (complete) solutions explored by our proposal is very low as compared to HR, which use most of the CPU time to explore and evaluate new (complete) solutions.

[INSERT TABLE 5 ABOUT HERE]

4.3. Effectiveness of the variable greedy design

The proposed algorithm is based on a variable neighbourhood depending on k, which determines the number of elements in the sequence to be destroyed and later reconstructed in a range from 1 to (n – 1). Therefore, it is of interest to assess the

contribution of this design to the performance of the algorithm. More specifically, we are interested in answering the following questions:

a) Does the variable design contribute to improve the solution with respect e.g. fixed value of k?

b) Are there alternative upper values of k (other than n – 1) for which the algorithm yields a higher performance?

In order to answer the first question, we analyse the contribution of each value of k to improve the current solution of the algorithm. More specifically, for a given problem instance, we record the value of k for which a new best-so-far solution is obtained. We then calculate the frequency of each value of k and plot the results. In Figure 3, we show the plot for different problems in the T-HR test-bed. Each line shows the frequency of the improvements of k for 20 runs of 10 instances of a given (n,m).

[INSERT FIGURE 3 ABOUT HERE]

In Figure 3, it is shown that all values of k in the range contribute to the improvement of the solution. The frequency decreases with k, which is not surprising, since in the algorithm, every time a new solution is found, the value of k is set to 1 and starts increasing once the solutions found with this destruction level are worse than the best-so-far sequence. The only exception is k = 1, indicating the relatively low ability of this destruction level to find better solutions as compared to k = 2. Nevertheless, the second highest frequencies are obtained for k = 1. Finally, it has to be noted that the pattern of the frequencies is the same for different number of machines.

In a different experiment also aimed to the first question, we compare the results obtained by this algorithm in the DMU test-bed with those obtained using different (fixed) numbers of components in the destruction phase. For all runs, the stopping criterion is the same as in the previous sub-section, i.e. the CPU time required for each problem by the HR heuristic. Consequently, the CPU time is again the same for all algorithms. The results are shown in Table 6 in terms of the APRD with respect to the best solution found by the different settings.

[INSERT TABLE 6 ABOUT HERE]

In view of the results in Table 6, the best overall results are obtained by the variable destruction design. While some results with a fixed k are very good, these are strongly dependent on the particular problem size.

Once the convenience of using a variable k as compared to employing fixed values has been established, we check whether there exist other upper values of k yielding better results than the original proposal (n – 1). To do so, we run different versions of our heuristic with several upper levels of k for the DMU test-bed. More specifically, we test n/10, n/5 and n/2. All versions are allowed to use the same CPU time (that employed for the comparison of the different heuristics in Section 4.2., see Table 1). The results are shown in Table 7 in terms of the RPD.

[INSERT TABLE 7 ABOUT HERE]

As can be seen from Table 7, employing other values as upper limit of the level of destruction is less efficient than the original proposal. Particularly, incorporating large

destruction levels seems suitable to obtain better results for a given CPU time, confirming the results in Figure 3 showing that also large k contribute to improve current best solutions.

4.4. Effectiveness of the slack-based greedy algorithm

In this subsection we investigate whether the mechanism adopted for selecting the k components to be destroyed (i.e. those k with least slack) is more efficient than, for instance, picking these k at random, or picking them in reverse order (descending order of their slacks). The results are shown in Table 8 in terms of the ARPD with respect to the best results obtained for the three approaches. In this table, 'NEW' denotes the approach proposed, while 'random' and 'inverse' indicates picking the components in the destruction phase at random, or with descending order of the slacks, respectively.

[INSERT TABLE 8 ABOUT HERE]

As can be seen from Table 8, the slack-based definition of the greedy algorithm is, on the overall results, more efficient than picking the solutions at random or in descending order of their slack. It is to note that, for the problem size $n = 50$ and $m = 15$, the inverse ordering obtains slightly better results. However, this small difference is not supported for the rest of the problem sizes.

5. Conclusions

We have presented a new algorithm for the problem of minimising total tardiness in permutation flowshops. A salient feature of the proposed method is that it does not require any parameter, with the exception of the computation time. This means that

there is no need to adjust or tune it for different problems. The computational experience carried out shows the proposal to outperform the best-so-far available algorithms. The algorithm is shown to be efficient for different computational efforts, and its design is checked to be more suitable as compared to different options regarding a variable/fixed destruction level, different upper limits for k, and other mechanisms for selecting the components to be destroyed. Further sophisticated features may be added to the suggested algorithm, such employing a more complex criterion for accepting new solutions. However, these enhancements must be contrasted with the simplicity and the speed of the current design.

Acknowledgements

The authors wish to thank the referees for the insightful comments on earlier versions of the paper. This research was accomplished while the first author was with the University of Duisburg-Essen (Germany) as recipient of an Alexander Von Humboldt Fellowship.

References

Armentano, V.A., Ronconi, D., 1999, Tabu search for total tardiness minimization in flowshop scheduling problems, Computers & Operations Research, 26, pp. 219-235.

Chakravarthy, K. C. Rajendran, 1999, A heuristic for scheduling in a flowshop with the bicriteria of makespan and maximum tardiness minimization, Production Planning and Control, 10, pp. 707-714.

Demirkol, E., Mehta, S., Uzsoy, R., 1998, Benchmarks for shop scheduling problems, European Journal of Operational Research, 109, pp. 137-141.

Framinan, J.M., Leisten, R., 2006, A heuristic for scheduling a permutation flow shop with makespan objective subject to maximum tardiness, International Journal of Production Economics, 99, pp. 28-40.

Gagné C, Gravel M, Price WL., 2003, A new hybrid Tabu-VNS metaheuristic for solving multiple objective scheduling problems. Proceedings of the Fifth Metaheuristics International Conference. Kyoto, Japan.

Gelders, L.F., Sambandam, N., 1978, Four simple heuristics for scheduling a flowshop, International Journal of Production Research, 16, pp. 221-231.

Hansen, P., Mladenović, N., 2001, Variable neighbourhood search: Principles and applications, European Journal of Operational Research, 2001, pp. 449-467.

Hasija, S., Rajendran, C., 2004, Scheduling in flowshops to minimize total tardiness of jobs, International Journal of Production Research, 42, pp. 2289-2301.

Kim, Y.D., 1993, Heuristics for flow shop scheduling problems minimizing mean tardiness, Journal of the Operational Research Society, 44, pp. 19-28.

Lourenço, H.R., Martin, O., Stützle, T., 2002, Iterated Local Search. In: Glover, F., and Kochenberger, G., editors, Handbook of Metaheuristics, pp. 321-353. Kluwer Academic Publishers.

Marchiori, E., Steenbeek, A., 2000, An evolutionary algorithm for large set covering problems with applications to airline crew scheduling. In: Cagnoni, S., et al., editor, Real Applications of Evolutionary Computing, EvoWorkshops 2000, Lecture Notes in Computer Science, 1803, pp. 367-381. Springer Verlag, Berlin.

Mladenović, N., Hansen, P., 1997, Variable Neighborhood Search, Computers Operations Research, 24, pp. 1097–1100.

Nawaz, M., Enscore, E. E., Ham, I., 1983, A heuristic algorithm for the m-machine, n-job flow-shop sequencing problem, OMEGA, 11, pp. 91-95.

Parthasarthy, S., Rajendran, C. 1998, Scheduling to minimize mean tardiness and weighted mean tardiness in flowshop and flowline-based manufacturing cell, Computers & Industrial Engineering, 34, pp. 531-546.

Schuster, C., Framinan, J.M., 2003, Approximative procedures for no-wait job shop scheduling, Operations Research Letters, 31, pp. 308-318.

Taillard, E., 1993, Benchmark for basic scheduling problems, European Journal of Operational Research, 64, pp. 278-285.

Tasgetiren MF, Sevkli M, Liang YC, Gencyilmaz G, 2004, Particle swarm optimization algorithm for single machine total weighted tardiness problem. In: Proceedings of the IEEE congress on evolutionary computation, Oregon, Portland, pp. 1412–1419.

Reza Hejazi, S., and Saghafian, S., 2005, Flowshop-scheduling problems with makespan criterion: a review, International Journal of Production Research, 43, pp. 2895-2929.

Ruiz, R., Maroto, C., 2005, A comprehensive review and evaluation of flowshop heuristics, European Journal of Operational Research, 165, pp. 479-494.

Ruiz, R., Stützle, T., 2006, A simple and effective iterated greedy algorithm for the permutation flowshop scheduling problem, European Journal of Operational Research (in press, available online).

Uzsoy, R., Benchmark scheduling problems [Last consulted 11.04.2006], http://palette.ecn.purdue.edu/~uzsoy2/benchmark/problems.html

		PR				HR				NEW				Avg. CPU (secs)
		Average		Standard dev.		Average		Standard dev.		Average		Standard dev.		
n	m	$\overline{PRD}_i$	PRD^*_i	$\overline{PRD}_i$	PRD^*_i	$\overline{PRD}_i$	PRD^*_i	$\overline{PRD}_i$	PRD^*_i	$\overline{PRD}_i$	PRD^*_i	$\overline{PRD}_i$	PRD^*_i	
20	15	5.918	3.119	2.478	1.990	2.275	0.573	1.349	0.579	0.675	0.020	0.470	0.070	1.379
20	20	5.664	2.618	2.787	2.523	1.397	0.536	0.891	0.593	0.429	0.020	0.228	0.051	1.644
30	15	8.303	4.644	2.083	2.317	3.483	1.196	1.854	1.443	1.647	0.254	0.848	0.461	4.875
30	20	6.822	3.549	2.366	1.416	2.493	0.993	1.388	1.285	1.119	0.114	0.375	0.265	5.801
40	15	9.815	4.895	3.646	1.771	3.760	1.271	1.698	1.488	2.031	0.279	0.911	0.508	12.143
40	20	7.482	3.778	2.217	1.946	3.047	1.241	1.315	1.411	1.633	0.189	0.794	0.463	14.329
50	15	8.631	4.411	2.975	2.087	3.793	0.815	1.933	0.865	2.310	0.330	0.978	0.501	24.928
50	20	8.352	4.704	2.664	1.830	3.217	1.001	1.477	1.200	2.216	0.519	1.245	1.113	29.121
		7.623	**3.965**	**2.652**	**1.985**	**2.933**	**0.953**	**1.488**	**1.108**	**1.508**	**0.215**	**0.731**	**0.429**	

Table 1. ARPD obtained in DMU test-bed (all heuristics use the same CPU time)

		PR				HR				NEW				Avg. CPU (secs)
		Average		Standard dev.		Average		Standard dev.		Average		Standard dev.		
n	m	$\overline{PRD}_i$	PRD^*_i	$\overline{PRD}_i$	PRD^*_i	$\overline{PRD}_i$	PRD^*_i	$\overline{PRD}_i$	PRD^*_i	$\overline{PRD}_i$	PRD^*_i	$\overline{PRD}_i$	PRD^*_i	
20	5	6.829	2.526	3.077	1.986	3.405	1.427	2.012	1.545	1.197	0.083	0.399	0.156	0.880
20	10	7.230	2.474	2.315	1.597	4.190	1.831	2.179	1.627	0.953	0.000	0.222	0.000	1.112
20	20	6.086	2.439	1.964	1.362	3.201	1.613	1.418	0.976	0.943	0.000	0.407	0.000	1.641
50	5	6.216	2.600	1.264	1.215	3.810	0.838	1.577	0.698	2.198	0.008	0.521	0.024	16.739
50	10	7.939	4.026	2.037	1.454	4.162	1.198	2.034	1.664	1.957	0.047	0.590	0.085	20.544
50	20	8.858	4.785	2.522	2.036	4.529	1.948	1.852	1.735	2.086	0.135	0.589	0.426	29.108
100	5	4.045	1.627	1.595	1.216	2.376	0.455	1.136	0.672	2.148	0.404	0.614	0.493	182.619
100	10	5.732	2.948	1.748	1.570	3.339	1.109	0.923	0.907	1.724	0.085	0.516	0.197	212.163
100	20	6.135	3.288	1.339	1.058	3.040	1.190	1.199	1.082	1.331	0.000	0.351	0.000	281.492
200	10	4.131	2.109	1.089	0.638	4.491	3.111	1.167	1.089	1.073	0.000	0.180	0.000	2486.117
200	20	4.688	2.467	0.804	0.963	4.337	2.941	0.804	0.666	1.022	0.000	0.297	0.000	3012.467
		6.172	2.844	1.796	1.372	3.716	1.605	1.482	1.151	1.512	0.069	0.426	0.126	

Table 2. ARPD obtained in the T-GS test-bed (all heuristics use the same CPU time)

		PR				HR				NEW				Avg. CPU (secs)
		Average		Standard dev.		Average		Standard dev.		Average		Standard dev.		
n	m	$\overline{PRD}_i$	PRD^*_i	$\overline{PRD}_i$	PRD^*_i	$\overline{PRD}_i$	PRD^*_i	$\overline{PRD}_i$	PRD^*_i	$\overline{PRD}_i$	PRD^*_i	$\overline{PRD}_i$	PRD^*_i	
20	5	14.764	7.267	8.064	6.969	3.621	1.639	4.019	3.455	2.352	0.208	1.600	0.657	0.880
20	10	7.747	1.249	7.531	2.348	1.294	0.277	2.868	0.807	1.159	0.000	3.067	0.000	1.107
20	20	0.752	0.632	2.378	1.999	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	1.644
50	5	2.909	0.863	0.637	0.573	1.828	0.397	0.642	0.501	1.343	0.297	0.415	0.330	16.686
50	10	7.208	2.884	2.072	1.672	5.827	2.366	1.238	1.460	2.527	0.106	0.778	0.252	20.444
50	20	29.604	14.699	12.312	10.201	14.088	2.725	5.728	6.268	18.565	8.436	11.678	7.564	29.083
100	5	1.999	0.854	0.579	0.542	1.214	0.497	0.877	0.694	1.247	0.331	0.421	0.452	182.924
100	10	4.073	1.961	0.880	0.866	2.565	0.812	1.314	0.835	1.166	0.118	0.228	0.213	211.100
100	20	4.820	1.391	0.832	1.176	4.135	1.517	1.398	1.132	1.608	0.133	0.449	0.294	283.643
200	10	3.222	2.150	0.617	0.556	2.332	1.588	0.886	0.865	0.789	0.000	0.175	0.000	2480.877
200	20	4.325	2.843	0.616	0.724	3.716	2.688	0.572	0.518	0.836	0.000	0.123	0.000	3002.736
		7.402	3.345	3.320	2.512	3.693	1.319	1.777	1.503	2.872	0.875	1.721	0.887	

Table 3. ARPD obtained in the T-HR test-bed (all heuristics use the same CPU time)

		PR				HR				NEW			
		$T/2$		$T/4$		$T/2$		$T/4$		$T/2$		$T/4$	
n	m	$\overline{PRD}_i$	PRD^*_i	$\overline{PRD}_i$	PRD^*_i	$\overline{PRD}_i$	PRD^*_i	$\overline{PRD}_i$	PRD^*_i	$\overline{PRD}_i$	PRD^*_i	$\overline{PRD}_i$	PRD^*_i
20	15	6.117	3.628	6.096	3.543	2.548	0.517	2.858	1.267	0.806	0.123	1.039	0.156
20	20	5.509	2.185	5.972	2.642	1.510	0.672	1.772	0.851	0.567	0.068	0.749	0.197
30	15	8.326	4.786	8.216	4.549	3.862	1.447	4.006	1.570	2.047	0.699	2.285	0.789
30	20	7.098	4.169	7.071	3.897	2.798	1.181	2.946	1.392	1.394	0.374	1.608	0.431
40	15	10.007	4.798	10.052	4.805	4.283	1.540	4.400	1.625	2.610	0.877	2.757	0.759
40	20	7.651	3.863	7.691	3.831	3.247	1.152	3.520	1.163	1.923	0.608	2.227	0.843
50	15	8.666	4.426	8.742	4.362	4.288	1.333	4.561	1.890	2.934	1.097	3.228	0.954
50	20	8.355	4.418	8.586	4.992	3.375	1.154	3.574	1.237	2.623	1.088	3.018	1.258
		7.716	4.034	7.803	4.078	3.239	1.124	3.455	1.374	1.863	0.617	2.114	0.673

Table 4. ARPD obtained in for different CPU times (average on 24 runs)

n	m	HR	NEW
20	15	109152,5	1947,6
20	20	108766,9	1796,2
30	15	243373,2	2223,3
30	20	243452,5	2014,5
40	15	431356,5	2449,7
40	20	432307,4	2235,5
50	15	673924,0	2771,9
50	20	673929,0	2474,2

Table 5. Number of solutions explored by HR and NEW in the DMU-testbed

n	m	k=1	k=2	k=10	k=19	k=20	k=29	k=30	k=39	k=40	k=49	NEW
20	15	2.14	1.79	**0.39**	0.42							0.40
20	20	1.68	1.42	0.45	**0.17**							0.37
30	15	3.68	3.88	1.88	1.10	1.00	**0.82**					0.84
30	20	3.16	2.81	0.84	0.61	0.62	0.64					**0.58**
40	15	4.56	4.79	1.94	1.64	1.69	1.92	**1.14**	1.68			1.51
40	20	3.83	3.88	1.66	1.23	1.12	**1.11**	1.33	1.14			1.36
50	15	6.07	6.78	3.19	1.87	2.21	1.96	1.12	2.00	**1.06**	1.66	2.28
50	20	4.29	4.30	1.67	1.69	1.73	1.17	1.49	1.51	1.44	1.58	**1.15**
Average:		3.68	3.71	1.50	1.09	1.40	1.27	1.27	1.58	1.25	1.62	**1.06**

Table 5. ARPD for different greedy algorithms for the instances in the DMU test-bed.

		Upper limit of k			
n	m	n/10	n/5	n/2	n-1 (NEW)
20	15	8.165	3.833	0.291	0.225
20	20	6.956	3.473	0.500	0.203
30	15	11.503	4.137	0.465	0.544
30	20	7.938	3.488	0.594	0.235
40	15	13.850	5.017	0.917	0.515
40	20	9.831	3.478	0.519	0.511
50	15	14.022	5.428	0.602	0.535
50	20	9.773	4.439	0.769	0.601
		10.255	**4.161**	**0.582**	**0.421**

Table 6. ARPD for several upper limits of k for the instances in the DMU test-bed (n-1 corresponds to the original proposal).

n	m	NEW	inverse	random
20	15	**0.390**	0.469	1.605
20	20	**0.147**	0.230	0.481
30	15	**0.341**	0.622	1.807
30	20	**0.258**	0.597	1.803
40	15	**0.977**	1.023	3.225
40	20	**0.407**	0.603	1.490
50	15	1.110	**0.949**	3.573
50	20	**0.530**	0.704	2.497
Average:		**0.520**	0.650	2.060

Table 7. ARPD for different mechanism for picking the components to be removed (DMU test-bed).

Step 1: Obtain one initial sequence using heuristic EADD and assign it to S and B.

Step 2: Initialize SA parameters, i.e. T:=475; $ACCEPT$:= 0; $TOTAL$:=0; FR_CNT:=0.

Step 3: If (FR_CNT = 5) or ($T \leq 20$) then go to Step 14.

Step 4: Invoke perturbation routine RIPS to obtain S'.

$$\Delta = \frac{TARDINESS(S') - TARDINESS(S)}{TARDINESS(S)} \cdot 100$$

If $\Delta > 0$ then go to Step 8.

Step 5: S:= S'; $ACCEPT$:= $ACCEPT$ + 1

Step 6: If $\frac{TARDINESS(S') - TARDINESS(B)}{TARDINESS(B)} \cdot 100 > 0$ then go to Step 10.

Step 7: B := S'; FR_CNT := 0. Go to Step 10.

Step 8: If $\exp(-\frac{\Delta}{T}) <$ Random[0;1], then go to Step 10.

Step 9: S:= S', and $ACCEPT$:= $ACCEPT$ + 1.

Step 10: $TOTAL$:= $TOTAL$ + 1.

Step 11: If ($TOTAL > 2\,n$) or ($ACCEPT > n/2$) go to Step 12; else go to Step 4.

Step 12: If ($ACCEPT \cdot 100/TOTAL \leq 15$) then FR_CNT := FR_CNT + 1

Step 13: Set T := $T \cdot 0.9$; $ACCEPT$:= 0; $TOTAL$:= 0. Go to Step 3.

Step 14: The algorithm is frozen. B contains the final solution.

Figure 1. Pseudocode of PR

Step 1: Obtain one initial sequence using EADD. Assign it to S and B.

Step 2: Apply JIBIS to S and B.

Step 3: $Z = TARDINESS(S)$, $Z_B := TARDINESS(B)$

Step 4: Initialize SA parameters, i.e. T :=540, $COUNT$:= 0; $REPLACE$:=0; $FREEZE$:=0.

Step 5: Generate and evaluate nine random sequences and store them (together with B) in an archive of the best ten sequences.

Step 6: If ($FREEZE$ = 5) or ($T \leq 20$) then go to Step 21.

Step 7: Set Z'' a large number.

Step 8: Do the following steps twice:

Do the following steps n times:

Invoke JSB to obtain S'. $Z' := TARDINESS(S')$.

If Z' is less than the maximum value of the objective function of a sequence amongst all sequences in the archive, then replace that sequence and its objective function by S' and Z'. If $Z' < Z''$ then $S'' := S'$ and $Z'' := Z'$.

Step 9: Do the following steps twice:

Invoke perturbation scheme PSS to obtain S'; $Z' := TARDINESS(S')$

If Z' is less than the maximum value of the objective function of a sequence amongst all sequences in the archive, then replace that sequence and its objective function by S' and Z'.

If $Z' < Z''$ then $S'' := S'$ and $Z'' := Z'$.

Step 10: $\Delta = \dfrac{TARDINESS(S') - TARDINESS(S)}{TARDINESS(S)} \cdot 100$

Step 11: If $\Delta > 0$ then go to Step 15.

Step 12: $S := S''$; $Z := Z''$; $REPLACE = REPLACE + 1$.

Step 13: If $\dfrac{TARDINESS(S'') - TARDINESS(B)}{TARDINESS(B)} \cdot 100 > 0$, then go to Step 17.

Step 14: B:=S''; $Z_B := Z''$; $FREEZE = 0$. Go to Step 17.

Step 15: If $\exp(-\dfrac{\Delta}{T}) <$ Random[0;1], then go to Step 17.

Step 16: $S := S''$; $Z := Z''$; $REPLACE := REPLACE + 1$.

Step 17: $COUNT := COUNT + 1$.

Step 18: If ($COUNT \leq 2n$) go to Step 7.

Step 19: If $REPLACE \cdot 100/COUNT \leq 15$, then $FREEZE := FREEZE + 1$.

Step 20: $T = T \cdot 0.9$; $COUNT :=$ 0; $REPLACE := 0$. Go to Step 6.

Step 21: Implement JIBIS on all sequences stored in the archive. The best amongst them is returned as solution.

Figure 2. Pseudocode of HR

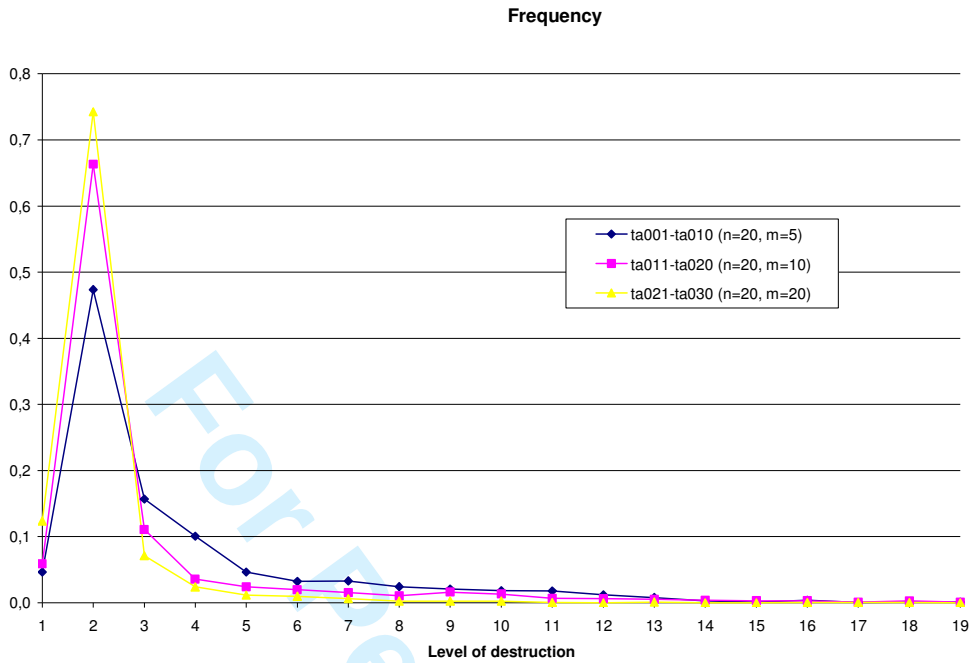

Figure 3. Frequencies corresponding to the values of *k* for which a better solution is found