A Genetic Algorithm to Minimize the Total Tardiness for M-Machine Permutation Flowshop Problems

Chia-Shin Chung[*], *James Flynn*[**], *Walter Rom*[***], *Piotr Staliński*[****]

Abstract

The m-machine, n-job, permutation flowshop problem with the total tardiness objective is a common scheduling problem, known to be NP-hard. Branch and bound, the usual approach to finding an optimal solution, experiences difficulty when n exceeds 20. Here, we develop a genetic algorithm, GA, which can handle problems with larger n. We also undertake a numerical study comparing GA with an optimal branch and bound algorithm, and various heuristic algorithms including the well known NEH algorithm and a local search heuristic LH. Extensive computational experiments indicate that LH is an effective heuristic and GA can produce noticeable improvements over LH.

Keywords: genetic algorithm, scheduling, permutation flowshop, tardiness.

Introduction

In the *permutation flowshop problem*, each of n jobs has to be processed on machines 1,…,*m* in that order. The processing times of each job on each machine are known. At any time, each machine can process at most one job and each job can be processed on at most one machine. Once the processing of a job on a machine has started, it must be completed without interruption. Also, each job must be processed in the same order at every machine. The usual objectives are the minimization of the make-span, total flow time, weighted total flow time, total tardiness, weighted total tardiness, and the

* *Dr. Chia-Shin Chung, Department of Operations and Supply Chain Management, Cleveland State Univesity, Cleveland, Ohio, 44115, c.chung@csuohio.edu.*

** *Dr. James Flynn, Department of Operations and Supply Chain Management, Cleveland State Univesity, Cleveland, Ohio, 44115, j.flynn@csuohio.edu.*

*** *Dr. Walter Rom, Department of Operations and Supply Chain Management, Cleveland State Univesity, Cleveland, Ohio, 44115, w.rom@csuohio.edu.*

**** *Dr. Piotr Staliński, Department of Quantitative Methods in Management, Wyższa Szkoła Biznesu-National Louis University, 33-300 Nowy Sącz, pstalinski@wsb-nlu.edu.pl.*

number of jobs late (see Pinedo 2002 for a review of the general flowshop problem). This article deals specifically with the objective of minimizing the total tardiness; however, its results can be adapted to other objectives. Tardiness equals the amount by which a job's completion time exceeds its due date. Practical effects of tardiness might include contractual penalty costs and loss of customer goodwill. Koulamas 1994 provides a general review of scheduling problems with tardiness criteria.

Schedules where each job must be processed in the same order at every machine are called *permutation schedules*. For $m \leq 2$, the restriction to such schedules is harmless; however, for larger m, there might exist a general schedule that performs better than any permutation schedule (Pinedo 2002). Finding such a schedule is often computationally impractical; moreover, as discussed in Kim 1995 there are many real situations where only permutation schedules are feasible. Most approaches restrict attention to permutation schedules.

Most optimal algorithms for single machine tardiness problems combine dynamic programming or branch and bound with decomposition properties developed by Lawler 1997, Potts and Wassenhove 1982, and Szwarc 1993. Szwarc et al. 1998 employ an improved decomposition rule, which allows them to solve problems with $n = 300$. For multi-machine tardiness problems, Vallada et al. 2008 report that the literature contains only a handful of papers dealing with optimal algorithms. Kim 1995 applies branch and bound using a “backward branching” scheme. His results include a problem size reduction procedure, which sometimes yields a problem with a smaller n. More recently, Chung et al. 2006 obtain a more effective branch and bound algorithm by combining a different branching scheme with better bounds. Extensive computational experiments involving over 40,000 test problems suggest that Chung et al. 2006 can handle problems with $n \leq 20$, but often experiences difficulty for problems with larger n. This is not surprising, since the m machine permutation flow shop problem with the total tardiness objective is NP-hard for $m \geq 1$ (Pinedo 2002).

A practical way of dealing with multi-machine tardiness problems is to develop effective heuristic solutions. Kim 1993 evaluates several heuristics and recommends an adaptation of the NEH algorithm of Nawaz et al. 1993, while Armentano and Ronconi 1999 propose a tabu search heuristic, which they compare with the NEH heuristic and the optimal branch and bound algorithm of Kim 1995. See Framinan et al. 2005, Kim 1993, Ruiz and Maroto 2005, and Vallada et al. 2008 for reviews of the literature on heuristic algorithms.

This article develops a genetic algorithm heuristic for multi-machine permutation flowshop problems with the total tardiness objective. It fills a gap by providing a solution procedure for problems that are not solvable by the branch and bound algorithm of Chung et al. 2006. The genetic algorithm concept, due to Holland 1975, has been successfully applied to many combinatorial optimization problems (see Reeves 1997). For flowshop problems, Arroyo and Armentano 2005, Etiler et al. 2004, and Tang and Liu 2002, respectively, present genetic algorithms for multi-objective criteria, the makespan objective, and the total flow time objective.

How useful a heuristic solution is depends on how well it performs against an optimal solution. Although the ideal way to evaluate performance is through comparison studies, such studies are rarely reported in articles dealing with genetic algorithms for multi-machine scheduling problems, possibly because effective optimal algorithms are often unavailable. One advantage of this article is that the authors had developed the most up to date optimal algorithm for multi-machine tardiness problems and could use it to evaluate the performance of their genetic algorithm heuristic. Our numerical study finds that the genetic algorithm performs remarkably well making it a practical substitute for an optimal algorithm when $n \geq 15$.

§2 introduces notation and describes steps that precede our various algorithms. One of these steps is a local search heuristic, LH, which provides a starting solution for our genetic algorithm, GA. §3 describes GA in detail. Some of its notable features include clone removal, maintenance of two populations with immigration, and probabilistic local search, *i.e.*, each time one finds a new solution, one performs a local search with a given probability. §4 reports on a numerical study that compares the performance of GA, the branch and bound algorithm of Chung et al. 2006, and various heuristic algorithms. Finally, §5 states our conclusions.

Notation and Preliminaries

This section introduces notation and describes two steps that precede our branch and bound and genetic algorithms. The first step is a problem size reduction procedure of Kim 1995, which sometimes yields problems with reduced n that are easier to solve. The second step is a local search heuristic, LH, which extends the well-known NEH algorithm (see Kim 1993 and Nawaz et al. 1983) and outperforms the *m*-machine heuristic of Chung et al. 2002 and Chung et al. 2006. LH provides a starting solution for our genetic and branch and bound algorithms.

Notation. For $i = 1,\dots,n$, $k = 1,\dots,m$, and any algorithm, A, for generating a schedule, denote

p_{ik} processing time of job *i* on machine *k*,
d_i due date of job *i*,
g^A total tardiness under the schedule generated by A.

Next, turn to Kim's problem size reduction procedure. By Proposition 7 of Kim 1995, the completion time of all jobs is bounded above by

$$K = \min\left[\left\{\sum_{i=1}^{n} p_{i(\max)} + (m-1)\max_{1\le i\le n} p_{i(\max)}\right\}, \left\{\sum_{j=1}^{n} p_{(\max)j} + (n-1)\max_{1\le j\le m} p_{(\max)j}\right\}\right]$$

where $p_{i(max)} = \max_{1\le j\le m} p_{ij}$ and $p_{(max)j} = \max_{1\le j\le n} p_{ij}$. Jobs whose due dates exceed *K* have zero tardiness and can be scheduled last. Kim calls these jobs *dominated jobs*, since schedules where they precede other jobs are dominated by other schedules. This leads to the following procedure. Find all dominated jobs. Schedule them in the last available positions in ascending order of their due dates. Then delete them from the problem. Finally, reduce *n* and re-label jobs accordingly. Repeat until either a problem is found with no dominated jobs or n has been reduced to 0. Let n_r denote the value of the *reduced n* obtained by this procedure. From the above analysis, there exists an optimal schedule where the deleted jobs are placed in positions n_r+1 through n in ascending order of their due dates. Given such a schedule, the total tardiness of the deleted jobs equals 0. Hence, for problems with n_r equal to 0, the EDD schedule is optimal and the optimal total tardiness equals 0. Our local search heuristic, LH, below applies when $n_r > 0$.

Our heuristic, LH, combines Kim's reduction procedure above with the procedure below, which depends on a single parameter, MAXREP. First, compute the EDD schedule. Stop if this step (or any of the steps below) yields a zero tardiness schedule. Second, compute the NEH schedule (see below). Third, starting with EDD as the initial incumbent, apply the ENS (extensive neighborhood search) algorithm of Kim 1993. ENS visits all neighbors of the current incumbent schedule, where a neighbor is obtained by interchanging a pair of jobs. If the best (in terms of the objective function) of the $n(n$-1)/2 neighbors is better than the incumbent, this best schedule becomes the new incumbent. Repeat until the best neighbor is no better than the current incumbent or the number of repetitions exceeds MAXREP. Fourth, apply ENS with NEH as the initial incumbent. Among the schedules obtained above, select the one with the smallest tardiness. Evaluating the tardiness of a schedule requires $O(mn)$ calculations. Hence, visiting all neighbors of a given schedule requires $O(mn^3)$ calculations. For the nontrivial test problems of §4, the mean computation time for LH ranged from 0.001 seconds for n = 15 to 1.3 seconds for n = 120. Note that for these test problems, replacing EDD above with the more complicated m-machine heuristic of Chung et al. 2002 and Chung et al. 2006 has a negligible effect on performance.

This section closes with a description of our implementation of the NEH algorithm. Let $\sigma = (\sigma(1),\ldots, \sigma(s))$ denote a *partial schedule* of *length s*, where $0 \le s \le n$, indicating that job $\sigma(j)$ occupies the jth position on each machine, for $1 \le j \le s$. A partial schedule of length 0 is the null schedule, and a partial schedule of length n is a complete schedule. For each job i not included in σ, define $C_i(\sigma)$ as the *makespan* of the partial schedule $(\sigma(1),\ldots, \sigma(s),i)$. The NEH algorithm constructs a sequence of partial schedules of successively increasing lengths until a complete schedule of length n is obtained. First, NEH uses a *dispatching rule* to select a job i for a partial schedule (i) of length 1. Next, for s = 1 to n - 1, NEH gets a partial schedule of length s+1 from a partial schedule of length s using three steps. The first step selects a job i not included in the partial schedule σ using the dispatching rule. The second step lists the s+2 partial schedules of length s+1 that can be obtained by inserting job i at some position in the partial schedule σ. (Job i could be inserted before job $\sigma(1)$, after job $\sigma(s)$, or between jobs $\sigma(j)$ and $\sigma(j+1)$, where $1 \le j < s$.) The third step selects one of these s +2 partial schedules according to some *objective*. Our version of the NEH algorithm is different from the one in Nawaz et al. 1983, which used maximization of the total processing time $\sum_{k=1}^{m} p_{ik}$ for dispatching and minimum makespan as the objective. Following Armentano and Ronconi 1999, we use the MDD (modified due date rule) for dispatching, which selects the job i that minimizes max$(d_i, C_i(\sigma))$. For the objective, we use minimum total tardiness with minimum makespan as a tiebreaker.

The Genetic Algorithm

Maintaining a population of solutions, genetic algorithms imitate genetic evolution. Periodically, the "fittest" members of the current population or generation breed to produce the next generation. Traits are passed on from parents to offspring in ways that resemble genetic mechanisms such as selection, crossover and mutation. The genetic algorithm stops after a finite number of generations with the best solution found as the proposed solution. For our genetic algorithm, GA, each solution or schedule s = $(s_1,s_2,...,s_n)$ is an element of X, the set of all permutation of the integers 1 through *n*. For *t* = 1, 2,...,t_{max}, the *t*th *generation* $S^{(t)} = \{\mathbf{s}_1^{(t)},\mathbf{s}_2^{(t)},\ldots,\mathbf{s}_N^{(t)}\} \subseteq X$¸ where the *population size N* is a multiple of 4 and the *maximum generation number* t_{max} is a positive integer. For $s \in X$, let *g(s)* denote the total tardiness of the schedule s. Also, let s^* denote the incumbent best schedule, i.e., the current best solution found, and let $g^*=g(s^*)$. Note that each evaluation of *g(s)* requires *0(mn)* calculations.

Under GA, initial values of s^* and g^* are provided by the local search heuristic, LH, of §2. The first generation $S^{(1)}$ consists of the schedules produced by EDD, NEH, ENS starting from EDD, ENS starting from NEH (see §2 for definitions), and *N-4* schedules chosen at random. Subsequently, for *t* = 1, 2, ..., t_{max} - 1, generation *t* +1 is obtained from generation *t*, using the selection, crossover, mutation, and clone removal, immigration, and local search procedures described next.

Selection. One half of generation $S^{(t)}$ is selected for breeding. Generation $S^{(t+1)}$ will then consist of these breeders and their offspring. Selection depends on two positive valued parameters, an *elitism factor*, p_{elit}, and a *spread factor*, p_{sprad}. Specifically, each $s \in S^{(t)}$ is assigned a *fitness value*, *f(s)*, as follows. If $g(s) \le g^* + p_{elit}$, then *f(s)* = *g(s)*; else *f(s)* = *g(s)(*1+*ε*), where each *ε* is a uniform random variable on $[0,p_{sprad}/t]$ and is statistically independent of all other *ε*. In words, fitness equals tardiness if tardiness is within p_{elit} of the best tardiness value found; else fitness equals tardiness plus a perturbation times tardiness. (Since the objective function is to be minimized, schedules with smaller fitness values tend to be more desirable.) Solutions are ranked in ascending order of fitness with ties broken arbitrarily and the *N*/2 solutions with the smallest fitness values are selected for breeding. Specifically, for j = 1 to N/4, solutions $\mathbf{s}_{2j-1}^{(t)}$ and $\mathbf{s}_{2j}^{(t)}$ produce two offspring using the crossover operation below. Note that since p_{sprad}/t decreases with *t*, selection depends more on *g(s)* as the *generation number t* increases.

Crossover. GA employs a two point crossover to generate two offspring from two parents. This procedure requires that *n* ≥ 4. First, we randomly generate two distinct integers, n_1 and n_2, strictly between 1 and *n*. Second, we obtain trial offspring as follows. If $n_1 < n_2$, offspring are generated by exchanging jobs in positions n_2 through n_1. If $n_1 > n_2$, offspring are generated by exchanging jobs outside positions n_2 through n_1. These trial offspring need not correspond to feasible schedules, since some jobs may be duplicated and others may be missing. We must correct them by replacing duplicated jobs with missing jobs. The example below illustrates how we do this.

A crossover example. Let *n* = 8, n_1 = 3, and n_2 = 5. Let parent 1 = (5,2,3,8,4,6,7,1) and parent 2 = (3,5,6,4,2,7,1,8). Then trial offspring 1 = (5,2,**6,4,2**,6,7,1), which duplicates jobs 2 and 6, but misses jobs 3 and 8. Similarly, trial offspring 2 = (3,5,**3,8,4**,7,1,8), which duplicates jobs 3 and 8, but misses jobs 2 and 6. Note that the exchanged jobs are boldfaced. For j = 1 and 2, we correct trial offspring j by replacing its non-boldfaced duplicate jobs by its missing jobs, using the same order as parent j. Here, we replace the non-boldfaced 2 and 6 in trial offspring 1 by 3 and 8, respectively, because 3 precedes 8 in parent 1. This procedure yields offspring 1 = (5,3,**6,4,2**,8,7,1) and offspring 2 = (6,5,**3,8,4**,7,1,2).

Mutation. GA uses two methods to perform a mutation *on a solution: exchange* and *inversion*. In an exchange, one randomly generates an integer n_1, where $1 \leq n_1 < n$, and then exchanges the jobs in position n_1 and $n_1 + 1$. In an inversion, one randomly generates two integers n_1 and n_2, where $1 \leq n_1 < n_2 \leq n$, and then reverses the order of the jobs in positions n_1 through n_2. When applied to a *population*, our mutation procedure depends on a *mutation parameter* p_{mutn}, where $0 \leq p_{mutn} \leq 1$, as follows. For each solution s in the population, one applies a mutation to s with probability p_{mutn}. Note that mutations are never performed when $p_{mutn} = 0$.

Clone removal. One can increase diversity by eliminating *clones* or duplicate solutions from the population. Clone removal is important because it eliminates the real possibility that all solutions in the population are the same. Checking the population for identical schedules is demanding computationally, however. To reduce computation time, our clone removal procedure compares the objective function values of schedules rather than the schedules themselves. Whenever two or more schedules have the same objective function values, we perform a mutation on all but one of them.

Probabilistic local search. One can often improve on a given solution s by examining the objective function values of all of its *neighbors*. If the smallest objective function value of the neighbors of s improves on $g(s)$, then a neighbor with the smallest objective function value replaces s. Our implementation of local search depends on a *local search parameter* p_{locs}, where $0 \leq p_{locs} \leq 1$, as follows. Each time a new solution is obtained in Steps 1 though 4 below, one performs a local search with probability p_{locs}. One can, of course, prevent local searches by setting $p_{locs} = 0$. Note that probabilistic local search has been used before, e.g., Ombuki and Ventresca 2004 incorporate it in a genetic algorithm for job shop scheduling.

Neighborhood definition. Under LH, the local search heuristic of §2, a neighbor of solution is defined through the exchange of an arbitrary pair of jobs, resulting in each solution having $O(n^2)$ neighbors. A pilot study, however, found that the computations for GA were too burdensome under this definition. To ensure that each solution has only $O(n)$ neighbors, GA defines neighbor *via generalized adjacent pairwise* interchanges, i.e., the interchange of jobs in any positions j and $j + l$, where $1 \leq j, j + l \leq n$, and $1 \leq l \leq k$, for some prescribed positive integer k. (This type of interchange reduces to an *adjacent pairwise* interchange when k = 1.) Since evaluating any $g(s)$ requires $O(mn)$ calculations, a local search requires $O(mn^2)$ calculations. Our pilot study found that k = 5 was effective in trading off solution quality versus computation time, so the value k = 5 is used in all of our reported results.

Other local search methods are possible. First, our method performs exactly one pass through the neighbors of s. One alternative is to perform two passes when the first pass results in an improvement. Another is to make multiple passes, stopping when a pass makes no improvement. (This is used for LH and leads to a true local optimal solution.) Second, the neighbors of s could consist of *circular rotations* of the form, $(s_k,\ldots,s_n,s_1,\ldots,s_{k-1})$, where $1 < k \le n$. Our pilot study indicated that the solution quality and computation time trade-offs are better under our local search method and our neighborhood definition.

Termination criterion. GA stops before t exceeds t_{max} or when the number of successive generations with no improvement in the incumbent best schedule equals a stop parameter, p_{stop}, whichever occurs first. Of course, GA keeps track of the generation number t_{limp} where the last improvement occurred.

The five-step scheme below is controlled with the flag NOCLONE. This flag determines whether the clone removal operator is executed in Step 4 below. Notice that at the end of Steps 2, 3, and 4, our algorithm calls a procedure *update*. If the best solution in the current population is better than the incumbent solution, the update procedure replaces the incumbent solution with the best solution and sets $t_{limp} = t$.

Step 1. Obtain the initial population and initialize the incumbent solution. Set $t = 1$ and $t_{limp} = 0$.

Step 2. Use the selection and crossover procedures to generate a new population and call update.

Step 3. Apply the mutation procedure to the population and call update.

Step 4. If NOCLONE = TRUE, then apply the clone removal procedure and call update.

Step 5. If $t = t_{max}$ or $t - t_{limp} = p_{stop}$ then stop; else set $t = t + 1$ and go to Step 2.

Multiple populations and immigration. One way of maintaining diversity is to have multiple populations and to apply the five-step scheme above to each population. The best of the solutions obtained for the individual populations would then be the final solution. Note that given multiple populations, one also has the option of allowing or not allowing immigration between populations. GA employs two populations and incorporates the flag, IMMIGRATION, which determines whether there is immigration. If IMMIGRATION = TRUE, then the overall computations are controlled by an *periodicity parameter*, p_{perd}, as follows: Every p_{perd} periods 20% of the solutions in each population are selected at random and transferred to the other population.

In summary, GA incorporates eight parameters, N, t_{max}, p_{stop}, p_{elit}, p_{sprd}, p_{mutn}, p_{locs}, p_{perd}, together with two flags, NOCLONE and IMMIGRATION, combined with two types of mutation. Selecting options for GA entails trade-offs between the computation time and the solution quality. For example, increasing N, t_{max}, p_{locs} and p_{stop} tends to make the computation time worse and the solution quality better.

To simplify the numerical analysis of §4, we undertook a pilot study involving an extensive number of test problems with a variety of options and found several apparent trends: First, the type of mutation does not affect the average solution quality and computation time. Second, setting NOCLONE and IMMIGRATION equal to TRUE does a good job of trading-off average solution quality and average computation time. Given these results, the test problems reported in §4 have NOCLONE and IMMIGRATION set equal to TRUE. These problems also employ only the exchange method of mutation. Our pilot study tested N = 120, 160, and 200 and p_{perd} = 20, 40, and 60 and found negligible differences in performance. Values of t_{max} up to 10000 combined with various values of, p_{stop}, p_{elit}, p_{sprd}, p_{mutn}, and p_{locs} were also tested. There seemed to be little advantage to increasing t_{max} beyond 5000 or p_{stop} beyond 400. Also $p_{elit} = 0.10$, $p_{sprd} = 0.05$, and $p_{mutn} = 0.15$ performed well. Therefore, the test problems of §4 employ the following values:

N = 120, $t_{max} = 5000$, $p_{stop} = 400$, $p_{elit} = 0.10$, $p_{sprd} = 0.05$, $p_{mutn} = 0.15$, $p_{perd} = 40$.

One of our striking findings is that small values of p_{locs} are effective in trading-off computation time and solution quality. Our original plan was to compare the options *never use local search* ($p_{locs} = 0$) and *always use local search* ($p_{locs} = 1$). Neither option did well in our pilot study. The first had problems with solution quality, while the second had problems with computation time. Our choice of probabilistic local search is designed to circumvent such problems. §4 performs a sensitivity analysis on p_{locs} that compares the values 0, 0.01, 0.10, and 1. Its results suggest that $p_{locs} = 0.01$ or 0.10 are good choices.

Numerical Study

This section reports on a numerical study that assesses the effectiveness of GA, our genetic algorithm, BB, the branch and bound algorithm of Chung et al. 2006, and LH, the local search heuristic of §2. Measuring the performance of these algorithms when n is large is not straightforward because optimal solutions are not always readily obtainable. We deal with this issue as follows. *First*, we evaluate LH by comparing it with the well known NEH algorithm, which is sometimes used as a benchmark in numerical studies (e.g., see Armentano and Ronconi 1999 and Etiler et al. 2004). Our numerical results find that LH significantly outperforms NEH, which suggests that LH is a more appropriate benchmark than NEH. *Second*, we evaluate GA and BB by comparing them with LH. *Third*, we further evaluate GA by comparing its objective function with an optimal objective function for test problems where BB provides an optimal solution. Note that most test problems with $n \leq 20$ are solved to optimality by BB.

Our test bank consists of 2160 randomly generated problems encompassing a wide variety of situations, with n assuming the values, 10, 15, 20, 30, 60, and 120. The numerical results suggest that LH is an effective heuristic and that both BB and GA can yield noticeable improvements over LH-at the cost of extra programming effort and computation time. In terms of trading-off computation time and solution quality, BB is superior to GA for $n = 10$ while GA is superior to BB for $n \geq 20$. When $n = 15$, both algorithms perform acceptably, but GA has the advantage with computation time. For large n, the solution quality under BB is much worse that under GA and not much better than under LH. Finally, a sensitivity analysis indicates that GA does well for small values of p_{locs}, the local search probability.

Turn to our problem generation procedure. All processing times are generated by the scheme of Chung et al. 2002, and Chung et al. 2006. Specifically, for i = 1,…,n and k = 1,…,m, p_{ik} has a discrete uniform distribution on [a_{ik}, b_{ik}], where a_{ik} and b_{ik} depend on a trend and a correlation factor. A positive trend in the processing time for job i indicates that p_{ik} is increasing in k, while a negative trend indicates that p_{ik} is decreasing in k. Similarly, a correlation between the processing times of job i exists if p_{i1},…,p_{in} are consistently relatively large or relatively small. For problems with correlation, additional integers, r_1, $i = 1,…,n$, are randomly drawn from {0,1,2,3,4}. Depending on the existence of a trend and/or a correlation, we obtain the following six *p-types*.

(I) Neither correlation nor trend: a_{ik} = 1 and b_{ik} = 100.

(II) Correlation only: a_{ik} = 20 r_i and b_{ik} = 20r_i + 20.

(III) Positive trend only: a_{ik} = 12 ½ (k-1) + 1, and b_{ik} = 12 ½ (k-1) + 100.

(IV) Correlation and positive trend: a_{ik} =2½ (k-1) + 20r_i +1, and b_{ik} =2½ (k-1) +20r_i +20.

(V) Negative trend only: a_{ik} = 12½ (m-k) + 1, and b_{ik} = 12½ (m-k) + 100.

(VI) Correlation and negative trend: a_{ik} =2½ (m-k) +20r_i + 1, and b_{ik} =2½ (m-k) +20r_i +20.

Due dates are generated from the scheme of Kim 1995, which employs two parameters: a *tardiness factor* τ and a *relative due-date range* ρ. Specifically, for i = 1,…,n, d_i has a discrete uniform distribution on $[P(1-\tau-\rho/2), P(1-\tau+\rho/2)]$, where P is the following lower bound on the *makespan* (i.e., the time needed to complete all jobs):

$$\text{P} = \max_{1\le j\le m}\left\{\sum\nolimits_{i=1}^{n} p_{ij} + \min_{1\le i\le n}\sum\nolimits_{l=1}^{j-1} p_{il} + \min_{1\le i\le n}\sum\nolimits_{l=j+1}^{m} p_{il}\right\}$$

The range and mean of d_i are approximately equal to ρP and $P(1\text{-}\tau)$, respectively. Varying τ and ρ as in Armentano and Ronconi 1999, we obtain the following four *d-types:*

(I) low tardiness factor (τ = 0.2) and wide relative due-date range (ρ = 1.2).

(II) low tardiness factor (τ = 0.2) and narrow relative due-date range (ρ = 0.6).

(III) high tardiness factor (τ = 0.4) and wide relative due-date range (ρ = 1.2).

(IV) high tardiness factor (τ = 0.4) and narrow relative due-date range (ρ = 0.6).

We use eighteen pairs of (m, n) values, i.e., all combinations of m = 2, 4, 8 and n = 10, 15, 20, 30, 60, and 120. For each (m, n), *p-type*, and *d-type*, we generate 5 problems. There are thus 2160 test problems (i.e., 120 for each m and n value).

Kim's problem size reduction procedure, described in §2, was applied to all test problems and LH was applied to all problems where n_r, the reduced n, was positive. Problems with n_r < 8 are easily solved by BB, but are too small for GA handle. Also LH is automatically optimal if g^{LH} = 0. Therefore, classify a problem as active if n_r ≥ 8 and > 0. Among our 2160 test problems, 1513 were active, 269 had < 8, and 378 had 8 and g^{LH} = 0. BB and GA were applied to the active problems, with LH yielding the initial incumbent.

GA was coded in Fortran 90 and run under Compaq Visual Fortran version 6.1 on a 2.4 Ghz Pentium 4 under Windows XP; BB, LH, and NEH were coded in C and run under Microsoft Visual C++ version 6.0 under Windows XP. All random numbers were generated by the *ran1* procedure from Press et al. 1992. The parameter MAXREP of LH was set to 120. BB *stopped prematurely* whenever the node count reached a specified *stop number* M-with a schedule that might or might not be optimal. We tried several values of *M*, since it strongly affects the computation time and the solution quality of BB. Initially, we applied BB to all active problems after setting $M = 10^7$ for $n = 10$, 15, 20, and 30 and setting $M = 4{\cdot}10^6$ for $n = 60$ and 120. This led to optimal solutions whenever $n = 10$. Next, we resolved the problems with $n = 15$ and $n = 20$ after setting $M = 2{\cdot}10^7$. This resulted in 94.3% of problems with n = 15 and in 43.6% of problems with n = 20 being solved to optimality. Then, we performed additional computations in which all of the problems with $n = 15$ where BB stopped prematurely were solved to optimality by setting $M = 2{\cdot}10^9$. Among these 44 problems, 8 had their objective functions improved. (The other 36 were already optimal.)

The hardest problem took 1.44 hours and had a node count of 1.2 billion. All these computations together with all the computations for LH and NEH were performed on a 3.0 Ghz Pentium D. The remaining computations described next were arduous. They were run on several Pentium computers several with clock speeds ranging from 1.7 Ghz to 3.0 Ghz, which must be taken into account when comparing computation times. First, to better depict optimal solutions, we resolved the problems with $n = 20$ after setting $M = 4{\cdot}10^8$. This resulted in 63.7% of problems being solved to optimality. Second, we resolved problems with $n = 30$ after setting $M = 2{\cdot}10^7$and problems with $n = 60$ and 120 after setting $M = 10^7$. Incidentally, these changes in M greatly increased the mean computation time for BB, but had only a small effect on the average solution quality.

Active test problems can be classified according to whether *BB succeeds* in finding schedule which is known to be optimal or *BB stops prematurely* with a schedule that might be suboptimal. Table 1 reports on the number of test problems in each classification as a function of n and d-type (using the largest *M* values employed, i.e., $M = 10^7$ for $n = 10$, $M = 2{\cdot}10^9$ for $n = 15$, $M = 4{\cdot}10^8$ for $n = 20$, $M = 2{\cdot}10^7$ for $n = 30$, and $M = 10^7$for $n \geq 60$). Notice that d-type I and II problems are less likely to be active than d-type III and IV problems, especially as *n* increases. In particular, when $n = 120$, almost none of the d-type I and II problems and almost all of d-type III and IV problems are active.

All tables, but Table 1, deal with the 1513 active test problems. Table 2 reports on the number of active problems and the percent where BB stopped prematurely versus *n* and *M*. Notice that for the largest *M* values employed, BB stops prematurely for 0%, 0%, 36.3%, 83.0%, 89.5%, and 88.8% of the active test problems when $n = 10$, 15, 20, 30, 60, and 120, respectively. Note that the test problems of this article are harder than those of Chung et al. 2006, as shown by the percentage of problems where BB stops early.

Tables 3, 4, and 5 list the mean and standard deviations of the computation times of LH versus n, BB versus *n* and *M*, and GA versus *n* and ρ_{locs}, respectively, for the active test problems. Notice that the standard deviation values in these and other tables tend to be high relative to the mean values, reflecting the high degree of variability. The mean computation times for LH are much smaller than the mean computation times for GA and BB, but tend to increase rapidly with *n*. For large *n*, one might want to reduce the times for LH by making the parameter MAXREP smaller and redefining neighborhood along the lines of §3 in order as to get *O(n)* instead of *O(n^2)* neighbors for each schedule and this bring the number of calculations for LH down from *O(mn^3)* to *O(mn^2)*.

Table 1. Number of problems in each classification vs. n and d-type (using highest M values for BB)

Classification	**d-types I and II**						**d-types III and IV**						**Row Total**
	n						***n***						
	10	**15**	**20**	**30**	**60**	**12**	**10**	**15**	**20**	**30**	**60**	**120**	
$n_r < 8$	13	16	25	33	74	100	1	2	1	2	0	2	269
$n_r \geq 8$ and	31	40	58	73	85	78	1	2	3	5	1	1	378
BB stops normally	136	124	44	15	1	0	178	176	130	27	20	20	871
BB stops prematurely	0	0	53	59	20	2	0	0	46	146	159	157	642
Column Total	180	180	180	180	180	180	180	180	180	180	180	180	2160

Table 2. Number of active problems and % where BB stopped prematurely vs. *n* and *M*

n	**Active problems**	**% stopped M = 4·10^6**	**M = 10^7**	**M = 2·10^7**	**M = 4·10^8**	**M = 2·10^9**
10	314	-	0	-	-	-
15	300	-	9.0	5.7	-	0
20	273	-	60.8	56.4	36.3	-
30	247	-	85.4	83.0	-	-
60	200	90.0	89.5	-	-	-
120	179	91.1	88.8	-	-	-

Table 3. Computation times for LH in seconds

n = 10		***n = 15***		***n = 20***		***N = 30***		***n = 60***		***n = 120***	
mean	s.d.	mean	s.d.	mean	s.d.	Mean	s.d.	mean	s.d.	mean	s.d.
0.0002	0.002	0.001	0.004	0.002	0.005	0.010	0.011	0.118	0.122	1.287	1.549

Table 4. Computation times for BB in seconds

n	***M* = 4·10^6**		***M* = 10^7**		***M* = 2·10^7**		***M* = 4·10^8**		***M* = 2·10^9**	
	mean	**s.d.**	**mean**	**s.d.**	**mean**	**s.d.**	**mean**	**s.d.**	**mean**	**s.d.**
10	-	-	0.01	0.03	-	-	-	-	-	-
15	-	-	17.5	34.7	23.8	52.0	-	-	57.7	322.9
20	-	-	166.2	160.9	311.3	314.4	3683.7	4762.1	-	-
30	-	-	448.6	313.1	1808.8	1530.2	-	-	-	-
60	717.0	617.6	1887.2	1765.1	-	-	-	-	-	-
120	2688.6	2526.0	6725.2	6575.7	-	-	-	-	-	-

Table 5. Computation time for GA in seconds

n	ρ_{locs} = 0		ρ_{locs} = 0.01		ρ_{locs} = 0.10		ρ_{locs} = 1	
	mean	s.d.	mean	s.d.	mean	s.d.	mean	s.d.
10	3.3	0.8	3.4	0.8	4.6	1.2	15.4	5.4
15	4.5	1.2	4.8	1.3	8.2	2.7	37.6	17.0
20	6.1	2.2	7.1	2.8	14.5	6.8	77.5	40.1
30	9.9	5.7	13.0	6.7	37.0	23.5	239.4	140.9
60	24.7	23.3	52.6	46.9	277.6	268.3	2010.6	1801.4
120	49.1	58.4	317.9	424.9	2509.3	3561.1	-	-

Table 6. *π (NEH, LH)*

n = 10		*n = 15*		*n = 20*		*n = 30*		*n = 60*		*n = 120*	
mean	s.d.	Mean	s.d.	mean	s.d.	mean	s.d.	mean	s.d.	mean	s.d.
-11.1	40.5	-25.1	103.5	-34.8	218.8	-50.4	235.2	-37.2	135.1	-20.1	61.9

source: authors' elaboration

Table 7. % of cases where $g^{LH} < g^{NEH}$

n = 10	*n = 15*	*n = 20*	*n = 30*	*n = 60*	*n = 120*
59.9	68.0	80.2	89.9	89.0	89.9

Table 8. *π (BB, LH)*

n	*M = 4·10⁶*		*M = 10⁷*		*M = 2·10⁷*		*M = 4·10⁸*		*M = 2·10⁹*	
	mean	s.d.	mean	s.d.	mean	s.d.	mean	s.d.	mean	s.d.
10	-	-	1.8	9.2	-	-	-	-	-	-
15	-	-	3.7	12.2	3.7	12.2	-	-	3.8	12.2
20	-	-	3.7	11.0	3.9	11.4	4.7	12.3	-	-
30	-	-	2.0	10.1	2.1	10.1	-	-	-	-
60	0.5	3.9	0.5	3.9	-	-	-	-	-	-
120	0.01	0.1	0.03	0.2	-	-	-	-	-	-

Table 9. *π (GA, LH)*

n	*ρ_{locs} = 0*		*ρ_{locs} = 0.01*		*ρ_{locs} = 0.10*		*ρ_{locs} = 1*	
	Mean	s.d.	Mean	s.d.	Mean	s.d.	Mean	s.d.
10	1.7	9.2	1.6	8.2	1.8	9.2	1.8	9.2
15	3.0	11.2	3.5	11.7	3.7	12.2	3.7	12.2
20	3.2	10.1	4.6	11.3	5.3	12.9	5.3	13.0
30	3.8	12.9	6.8	16.7	7.8	17.8	8.4	18.7
60	2.9	11.4	7.3	16.6	8.2	17.3	8.6	17.5
120	1.6	7.9	3.9	10.6	4.5	11.6	-	-

Table 10. % of cases where selected algorithms outperform one another

n	*M*	$g^{BB} < g^{LH}$	$\rho_{locs} = 0$			$\rho_{locs} = 0.10$		
			$g^{GA}<g^{LH}$	$g^{GA}< g^{BB}$	$g^{BB}< g^{GA}$	$g^{GA}<g^{LH}$	$g^{GA}< g^{BB}$	$g^{BB}< g^{GA}$
10	10^7	23.9	22.9	0	1.9	23.9	0	0
15	10^7	38.7	32.3	0	14.0	39.3	0.7	1.3
15	$2{\cdot}10^9$	39.3	32.3	0	14.7	39.3	0	13.3
20	$2{\cdot}10^7$	46.5	45.4	11.0	22.3	56.4	19.0	3.7
20	$4{\cdot}10^8$	52.7	45.4	4.0	26.4	56.4	7.7	4.4
30	$2{\cdot}10^7$	23.9	55.1	42.1	12.1	73.7	60.3	1.2
60	$4{\cdot}10^6$	4.0	55.3	54.2	0.6	78.0	74.5	0
120	$4{\cdot}10^6$	1.7	45.3	43.6	0.6	81.0	79.3	0

Table 11. % of problems where BB stops normally and $\pi^*(GA)$ when BB stops normally

n	*M*	% stopped normally	$\pi^*(GA)$							
			$\rho_{locs} = 0$		$\rho_{locs} = 0.01$		$\rho_{locs} = 0.10$		$\rho_{locs} = 1$	
			Mean	s.d.	Mean	s.d.	Mean	s.d.	Mean	s.d.
10	10^7	100	0.1	0.7	0.3	4.2	0	0	0	0
15	$2{\cdot}10^9$	100	1.1	7.1	0.3	2.9	0.05	0.6	0.01	0.1
20	$4{\cdot}10^8$	63.7	2.5	9.2	1.2	7.3	0.1	0.9	0.01	0.1
30	$2{\cdot}10^7$	17.0	0.4	1.3	0.1	0.6	0.05	0.2	0.02	0.1
60	10^7	10.5	2.6	10.3	0	0	0	0	0	0
120	10^7	11.2	0.01	0.04	0	0	0	0	-	-

Turn to BB and GA. For $n = 10$, the mean computation time under BB is much smaller than under GA, while for $n \geq 20$, the mean computation times for GA tend to be much smaller than for BB-especially when ρ_{locs} is small. Notice that the mean computation times for GA in Table 5 appear to grow linearly with n when $\rho_{locs} = 0$ and quadratically in n when $\rho_{locs} > 0$. This is not surprising since each evaluation of the tardiness function, $g(s)$, requires $O(mn)$ calculations, while each local search for GA search requires $O(mn^2)$ calculations. As discussed later in this section, our numerical results indicate that small values of ρ_{locs} are effective in trading-off computation time and solution quality.

The remaining tables deal with the solution quality of our algorithms on active test problems. In general, one can evaluate the solution quality of a schedule by comparing it with a benchmark schedule or with an optimal schedule-when an optimal schedule is obtainable. Our benchmark schedule is the one produced by LH and our criterion is the percentage average measure defined below. This measure is appropriate when the goal is to minimize a nonnegative objective function.

Define the *percentage advantage of algorithm* A over LH by

$$\pi (A,LH) = 100(g^{LH} - g^{A}) / g^{LH} \tag{4.1}$$

For active test problems, g^{LH} is positive so (4.1) is defined. The quantity $\pi (A,LH)$ represents the difference between g^{LH} and g^{A} expressed as a percentage of g^{LH}. Tables

6, 8, and 9 evaluate the solution quality of NEH, BB, and GA on active test problems using *π(NEH, LH)*, *π(BB, LH)*, and *π(GA, LH)*, respectively. Note that NEH provides a starting solution for LH, and LH provides a starting solution for BB and GA. Hence, $g^{LH} \le g^{NEH}$, $g^{BB} \le g^{LH}$, and $g^{GA} \le g^{LH}$, implying that $\pi(NEH, LH) \le 0$, $\pi(BB, LH) \ge 0$, and $\pi(GA, LH) \ge 0$ for active test problems.

Similarly, define the percentage advantage of an optimal solution over A by

$$\pi^*(A) = \begin{cases} 0, & \text{if } g^A = g^*, \\ 100\left(g^A - g^*\right)/g^A, & \text{if } g^A > g^*, \end{cases} \tag{4.2}$$

where g^* denotes the optimal total tardiness. Note that (4.2) is always defined since $g^* \ge 0$, implying that $g^A > 0$ when $g^A > g^*$. The quantity $\pi^*(A)$ represents the difference between g^A and g^* expressed as a percentage of g^A. When BB stops normally, $g^* = g^{BB}$, so one can obtain $\pi^*(A)$. Table 11 uses $\pi^*(GA)$ to evaluate how close GA is to optimality on the set of test problems where BB stops normally.

Comparing LH with NEH, Table 6 reports on the mean and standard deviation of, *π(NEH, LH)*, the percentage advantage of NEH over LH versus n, while Table 7 reports on the percentage of cases where $g^{LH} < g^{NEH}$ versus *n*. These tables indicate that the solution quality under NEH tends to be much worse that under LH. For instance, when *n* = 60, the mean value of *π(NEH)* equals -37.2, i.e., on the average g^{NEH} is 37.2% larger than g^{LH}; furthermore, $g^{LH} < g^{NEH}$ in 89.0 % of the cases. To verify that the differences between LH and NEH in Tables 6 and 7 also hold for large *n*, we generated additional test problems for *n* = 180, 240, and 300, using the same scheme as before, and got results similar to those for *n* = 120. These results suggest LH is a more appropriate benchmark than NEH when evaluating algorithms.

Table 8 compares BB with the LH by reporting in the mean and standard deviation of *π(BB, LH)* versus *n* and *M*. Similarly, Table 9 compares GA with LH by reporting in the mean and standard deviation of *π(GA, LH)* versus *n* and ρ_{locs}. Notice that GA does much better than LH for all *n* tested, while BB does substantially better for small *n* only. Indeed, BB does only slightly better than LH when n is large, the mean percentage advantage equaling 0.5% for *n* = 60 and 0.03% for *n* = 120 for the largest *M* employed. These small values for large n might be due to BB often stopping prematurely.

Table 10 reports on the percent of cases where GA and BB outperform LH, GA outperforms BB, and BB outperforms GA for selected values *M* and for $\rho_{locs} = 0$ and 0.10. Notice that $g^{GA} = g^{BB}$ for all test problems when *n* = 10, $M = 10^7$ and $\rho_{locs} = 0.10$. Since BB is always optimal those problems, GA must also be optimal. The results in Table 10 are more favorable to BB over LH when when *n* = 60 or 120 than the results in Table 8; however, when BB does better than LH, the difference tends to be small. For example, consider the cases reported in Tables 8 and 10 where *n* = 120 and $M = 4{\cdot}10^6$. There, BB outperforms LH in 1.7% of the problems, but the average difference in the percentage advantage over NEH is only 0.01%.

Turn to a comparison of GA and BB. As indicated in the analysis of the various cases below, our numerical results indicate that BB is better for small *n* and GA is better for moderate and large *n*.

The case n = 10: As mentioned above, BB and LH obtain optimal schedules for all problems when M = 10^7 and ρ_{locs} = 0.10; however, BB is the clear winner, since its mean computation time of 0.01 seconds is far superior to GA's mean computation time of 4.6 seconds. (Both times are acceptable, however.)

The case n = 15: In terms of average solution quality, BB with *M* = 10^7 and LH with ρ_{locs} = 0.10 do equally well, with mean relative advantages over LH of 3.7% in Tables 8 and 9. Nevertheless, GA with a mean time of 8.2 seconds seems to be the winner over BB with a mean time of 17.5 seconds. On the other hand, BB can obtain optimal solutions for all test problems (with, however, an increase in the mean time to 57.7 seconds and only a 0.1% improvement in the mean relative advantage over LH).

The case n = 20. In terms of average solution quality and computation time, GA with ρ_{locs} = 0.10 does better than BB for any of the *M* values tested, i.e., the mean values of *π(GA, LH)* in Table 9 are greater than the mean values of *π(BB, LH)* in Table 8, and the mean times for BB in Table 4 are greater than the mean times for GA in Table 5.

The case n ≥ 30. In terms of average solution quality and computation time, GA with ρ_{locs} = 0.01 does much better than BB for any of the *M* values tested. Furthermore, for *n ≥ 60*, the mean values of *π(BB, LH)* are close to 0 and BB never outperforms GA when ρ_{locs} = 0.10 (see Tables 8 and 10).

Table 11 reports on the percent of test problems where BB stops normally versus *n*, and the mean and standard deviation of $\pi^*(GA)$, the percentage advantage of an optimal solution over GA, versus *n* and ρ_{locs}. In order to maximize the number of problems where BB stops normally and thus g^* is available, Table 11 utilizes the largest value of *M* employed by BB. For *n* = 10 and 20, all problems are solved to optimality by BB, and GA is almost as good as BB when ρ_{locs} ≥ 0.01. For *n* = 20, 63.7% of the problems are solved to optimality, and the average solution quality of GA is good for ρ_{locs} = 0.01 and outstanding for ρ_{locs} ≥ 0.10. Finally, for *n* ≥ 30, between 10% and 17% of the problems are solved to optimality by BB and GA is almost as good BB for such problems when ρ_{locs} ≥ 0.01. For problems where *n* ≥ 30 and BB stops prematurely, the superior performance of GA indicates that on the average BB is not close to optimality; however, the average closeness of GA to optimality is unknown.

The parameter ρ_{locs} strongly affects the computation time and the solution quality of GA. One of our more interesting findings is that small values of ρ_{locs} are effective in trading-off computation time and solution quality. Tables 5 and 9 perform a sensitivity analysis on ρ_{locs} that examines computation time and the solution quality of GA for ρ_{locs} = 0, 0.01, 0.10, and 1. As expected, both computation time and solution quality tend to increase with ρ_{locs}. The data suggest that ρ_{locs} = 0.01 and ρ_{locs} = 0.10 are effective choices.

Incidentally, the entries for *n* = 120 and ρ_{locs} = 1 are missing from Tables 5 and 9. A sample of over 100 test problems found that increasing ρ_{locs} from 0.10 to 1 when *n* = 120 increased the mean of *ρ(GA, LH)* by under 0.5%, while greatly increasing computation times. (Many test problems took over 24 hours to solve.) Also, observe that in the row for $n = 10$ of Table 9, the mean of $\rho(GA, LH)$ is smaller for $\rho_{locs} = 0.01$ than for $\rho_{locs} = 0$, illustrating that $\rho(GA, LH)$ is not always a nondecreasing function of ρ_{locs}.

Conclusions

The *m*-machine, *n*-job, permutation flowshop problem with the total tardiness objective is a common scheduling problem, known to be NP-hard. Branch and bound, the usual approach to finding an optimal solution, experiences difficulty when *n* exceeds 20. This article fills a gap by providing a solution procedure for problems that are not solvable by the branch and bound algorithm of Chung et al. 2006, developing a genetic algorithm, GA, which can handle problems with larger *n*. GA incorporates clone removal, two populations with immigration, and probabilistic local search. We also undertake a numerical study comparing GA with an optimal branch and bound algorithm BB, and various heuristic algorithms, including the well known NEH algorithm and a new heuristic LH.

One critical advantage of this article is that the authors had developed the state of the art optimal algorithm for multi-machine tardiness problems and could use it to evaluate the performance of their algorithms. Extensive computational experiments indicate that LH is an effective heuristic and GA can produce noticeable improvements over LH. Furthermore, GA seems to do a much better job than BB of trading-off computation time and solution quality for $n \geq 15$, and the solution quality under BB is not noticeably better than under LH for large *n*. One striking result is that GA appears to do well for small values of the local search probability.

For future research, we believe that the following topics are potentially useful: (i) the application of other solution techniques to the problem, e.g., Lagrangean relaxation and slack variable decomposition; (ii) extending GA to other objectives, e.g., total weighted tardiness, total flowtime, total weighted flowtime, and makespan.

References

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

Arroyo, J., and Armentano, V. (2005). Genetic local search for multi-objective flowshop scheduling problems. *European Journal of Operational Research* 167, 717-738.

Chung, C.S., Flynn, J., and Kirca, O. (2002). A branch and bound algorithm to minimize the total flowtime for m-machine permutation flowshop problems. *International Journal of Production Economics* 79, 185-196.

Chung, C.S., Flynn, J., and Kirca, O. (2006). A branch and bound algorithm to minimize the total tardiness for m-machine permutation flowshop problems. *European Journal of Operational Research*, 174, 1-10.

Etiler, O., Toklu, B., Atak, M., and Wilson, J. (2004). A genetic algorithm for flow shop scheduling problems. *Journal of the Operational Research Society* 55, 830-835.

Framinan, J., Leisten, R., and Ruiz-Usano (2005). Comparison of heuristics for flowtime minimization in permutation flowshops. *Computers & Operations Research* 32, 1237-1254.

Holland, H. (1975). Adaptation in Natural and Artificial Systems, University of Michigan Press, Ann Arbor.

Kim, Y.D. (1993). Heuristics for Flowshop Scheduling Problems Minimizing Mean Tardiness. J. *Operational Research Society* 44, 19-28.

Kim, Y.D. (1995). Minimizing tardiness in permutation flowshops. *European Journal of Operational Research* 85, 541-555.

Koulamas, C. (1994). The total tardiness problem: review and extensions. *Operations Research* 42, 1025-1040.

Lawler, E. (1997). A pseudopolynomial algorithm for sequencing jobs to minimize total tardiness. *Annals of Discrete Mathematics* 1, 331-342.

Nawaz, M. ,Enscore, E., and Ham, I. (1983). A heuristic algorithm for the *m*-machine, *n*-job flowshop scheduling problem, OMEGA 11, 91-95.

Ombuki, B., and Ventresca, M. (2004). Local search genetic algorithms for the job shop scheduling problem. *Applied Intelligence* 21, 99–109.

Pinedo, M. (2002). *Scheduling*, 2nd Edition, Prentice-Hall, New Jersey.

Potts, C., and Wassenhove, L. (1982). A decomposition algorithm for the single machine total tardiness problem. *Operations Research Letters* 1, 177-181.

Press, W., Teukolsky, S., Vetterling, W., and Flannery, B. (1992). *Numerical Recipes in C : The Art of Scientific Computing*, 2nd ed., Cambridge University Press, Cambridge.

Reeves, C. (1997). Genetic Algorithms for the Operations Researcher. INFORMS *Journal on Computing* 9, 231-250.

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

Szwarc, W. (1993). Single machine total tardiness problem revisited, in Y. Ijiri (ed.). *Creative and Innovative Approaches to the Science of Management*, Quorum Books, 407-419.

Szwarc, W., Croce, F., and Grosso, A. (1999). Solution of the single machine total tardiness problem. *Journal of Scheduling* 2, 55-71.

Tang, L., and Liu, J. (2002). A modified genetic algorithm for the flow shop sequencing problem to minimize mean flow time. *Journal of Intelligent Manufacturing* 13, 61-67.

Vallada, E., Ruiz, R., and Minella, G. (2008). Minimizing total tardiness in the m-machine flowshop problem: a review and evaluation of heuristics and metaheuristics. *Computers & Operations Research* 35, 1350-1373.

Abstract (in Polish)

Permutacyjny problem przepływowy (ang. permutation flowshop problem) z m maszynami i n zadaniami oraz minimalizacją sumy opóźnień jest znanym zagadnieniem z zakresu szeregowania zadań. Zagadnienie to należy do kategorii NP-trudnych problemów optymalizacji kombinatorycznej. Metoda podziału i ograniczeń (ang. branch and bound), popularne podejście do rozwiązania problemu, doświadcza trudności (biorąc pod uwagę czas potrzebny dla znalezienia rozwiązania optymalnego) gdy n przekracza 20. W niniejszej pracy, proponujemy algorytm genetyczny GA dla rozwiązywania zagadnienia dla dużych wartości n. Przytaczamy wyniki obszernego studium obliczeniowego, które porównuje fukcjonowanie algorytmu GA z metodą podziału i ograniczeń oraz metodami heurystycznymi - znanym algorytmem NEH i heurystyką lokalnego przeszukiwania LH. Rezultaty obliczeniowe wskazują, że metoda LH jest wydajnym algorytmem heurystycznym i że metoda GA oferuje możliwość poprawy wyników w porównaniu z algorytmem LH.

Słowa kluczowe: algorytm genetyczny, planowanie, permutacja, opóźnienia.