A Problem Space Algorithm for Single Machine Weighted Tardiness Problems

Selçuk Avci
Robert H. Storer
Lehigh University

M. Selim Akturk
Bilkent University
Ankara, Turkey

Report No. 00T-003

A PROBLEM SPACE ALGORITHM FOR SINGLE MACHINE WEIGHTED TARDINESS PROBLEMS

Selcuk Avci [1], **M. Selim Akturk** [2], **Robert H. Storer** [1,3]

[1] Dept. of Industrial and Manufacturing Systems Engr., Lehigh University, Bethlehem, PA 18015

[2] Dept. of Industrial Engineering, Bilkent University, Ankara, Turkey

Abstract

We propose a new problem space genetic algorithm to solve single machine total weighted tardiness scheduling problems. The proposed algorithm utilizes a set of global and time-dependent local dominance rules, which are very useful for reducing the search space. They are also a powerful exploitation (intensifying) tool since we know that the global optimum is one of the local optimum solutions. Furthermore, problem space search method significantly enhances exploration (diversification) capability of the genetic algorithm. In sum, we can improve the solution quality in terms of the average deviation and total match as well as the robustness in terms of the maximum deviation from the optimum solution as compared to the other local search algorithms reported in the literature.

[3]Corresponding author. Robert H. Storer

Address: Dept. of Industrial and Manufacturing Systems Engineering, Lehigh University, Bethlehem, PA 18015

Telephone: (610) 758-4036, Fax: (610) 758-4886, E-mail: rhs2@lehigh.edu

1 Introduction

The single machine total weighted tardiness problem, $1|\,|\sum w_jT_j$, may be stated as follows. A set of jobs (numbered $1,\ldots,n$) is to be processed without interruption on a single machine that can handle only one job at a time. All jobs become available for processing at time zero. Job j has an integer processing time p_j, a due date d_j and has a positive weight w_j. Furthermore, a weighted tardiness penalty is incurred for each time unit of tardiness T_j if job j is completed after its due date d_j. The problem can be formally stated as: find a schedule S that minimizes $f(S) = \sum_{j=1}^{n} w_jT_j$.

Lawler [9] shows that the $1|\,|\sum w_jT_j$ problem is strongly NP-hard. Various enumerative solution methods have been proposed for both the weighted and unweighted cases. Emmons [7] derives several dominance rules that restrict the search for an optimal solution to the $1|\,|\sum T_j$ problem. Rinnooy Kan et al. [16] extended these results to the weighted tardiness problem. Rachamadugu [15] identifies a condition characterizing adjacent jobs in an optimal sequence for $1|\,|\sum w_jT_j$. Volgenant and Teerhuis [21] use this condition as an improvement step for different dispatching rules. The branch and bound (BB) algorithm of Potts and Van Wassenhove [13] can solve problem instances with up to 50 jobs. Exact approaches used in solving the weighted tardiness problem are tested by Abdul-Razaq et al. [1] who use Emmons' dominance rules to form a precedence graph for finding upper and lower bounds. Szwarc [20] proves the existence of a special ordering for the single machine earliness-tardiness problem with job independent penalties where the arrangement of two adjacent jobs in an optimal schedule depends on their start time. Although the presented results cannot be extended to the $1|\,|\sum w_jT_j$ problem with job dependent penalties as stated by the author. Recently, Akturk and Yildirim [2] proposed a new dominance rule and a lower bounding scheme for the $1|\,|\sum w_jT_j$ problem that can be used to reduce the number of alternatives in any exact approach.

Implicit enumerative algorithms for the total weighted tardiness problem, such as the BB algorithm proposed by Potts and Wassenhove [13], guarantee optimality, but require considerable computer resources both in terms of computation time and memory requirements. It is important to note that, with respect to neighborhoods based on pairwise interchange, the number of local minima is very high because of the nature of the scheduling problems. Therefore, several heuristics and dispatching rules have been proposed to generate good, but not necessarily optimal, solutions as discussed in Potts and Van Wassenhowe [14] and Akturk and Yildirim [3]. The obvious disadvantage of these methods is that the solutions generated by simple heuristic methods may be far from the optimum. This problem can also be tackled by local search methods. An overview of local search methods for machine scheduling problems can be found in Anderson et al. [4]. Crauwels et al. [6] present several local search heuristics for the $1|\,|\sum w_jT_j$ problem. They introduce a new binary encoding scheme to represent solutions, together with a heuristic to decode the binary representations into actual sequences. This binary encoding scheme is also compared to the usual permutation representation for descent, simulated annealing, threshold accepting, tabu search and genetic algorithms on a large set of problems.

In this study, we propose a new local search algorithm by using a problem space genetic algorithm (PSGA), (Storer et al. [18], [19], and Naphade et al. [12]), for the $1| \, |\sum w_j T_j$ problem. Several features that demonstrate the effectiveness of local search heuristics are their ability to adapt to a particular realization, avoid entrapment at local optima and exploit the basic structure of the problem. Therefore, we will incorporate an efficient base heuristic, namely the ATC rule with global and time-dependent local dominance relationships into the proposed PSGA algorithm to improve the solution quality and robustness. Furthermore, the running time behavior of the proposed algorithm as a function of the number of jobs seems much better than the local search algorithms discussed in Crauwels et al. [6]. Each of these properties are discussed below starting with the global dominance relationships.

2 Dominance Rules

The following lemma by Emmons [7] plays a major rule in the enumerative algorithms in the literature.

Lemma 1: If $d_i \leq d_j$, $p_i \leq p_j$ and $w_i \geq w_j$ then job i globally precedes job j, i.e. $i \Rightarrow j$.

Assume that this rule has already been applied to get a sequence for each job h, and let B_h and A_h be the set of jobs which precede and succeed job h respectively in at least one optimal sequence. The following theorem presents the three conditions of Rinnooy Kan et al. [16] generalizations based on the Emmons' rule.

Global Dominance Theorem: There exists an optimal sequence in which job i is sequenced before job j, i.e. $i \Rightarrow j$, if one of the following three conditions is satisfied.

(a) $p_i \leq p_j$, $w_i \geq w_j$ and $d_i \leq \max\{d_j, \sum_{h \in B_j} p_h + p_j\}$;

(b) $w_i \geq w_j$, $d_i \leq d_j$ and $d_j \geq \sum_{h \in S - A_i} p_h - p_j$;

(c) $d_j \geq \sum_{h \in S - A_i} p_h$.

Whenever jobs i and j satisfy the above theorem, an arc (i, j) is added to the precedence graph with any other arcs that are implied by transitivity. Let us demonstrate how these global dominance rules can be implemented on the following 20-job problem. In the 0-1 global dominance matrix below, an entry of 1 indicates that the job in row i globally dominates the job in column j. Furthermore, RowSum(i) and ColSum(i) give the number of jobs guaranteed to be succeeding or preceding job i, respectively, in an optimum sequence.

j	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
p_j	62	69	1	91	34	61	56	6	51	71	15	77	79	37	43	90	39	20	72	48
d_j	124	236	328	329	373	385	405	412	488	494	547	571	657	713	758	836	901	912	986	1074
w_j	4	3	7	2	2	8	10	3	10	4	5	2	7	3	1	8	6	3	7	7

j	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	RowSum
1	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	19
2	0	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	17
3	0	1	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	18
4	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	1	1	4
5	0	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1	1	1	10
6	0	0	0	1	1	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	16
7	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	13
8	0	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1	1	1	10
9	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1	1	1	1	11
10	0	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1	1	1	10
11	0	0	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1	1	9
12	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	1	1	4
13	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	1	1	4
14	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	6
15	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	1	1	4
16	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	1	1	4
17	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	1	3
18	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	2
19	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1
20	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
colsum	0	2	1	4	4	3	4	5	5	6	9	10	10	10	11	11	16	17	18	19	

Let N be the number of unscheduled jobs. If RowSum(i) $= N - 1$ then job i will be scheduled to the first available position. Similarly, if ColSum(j) $= N - 1$ then job j will be scheduled to the last available position. If we proceed iteratively in the same manner, we obtain the following sequence {1, 3, 2, 6, –, –, –, –, –, –, –, –, –, –, –, –, 17, 18, 19, 20 }. Hence, we fix the first and last four positions, and our problem is reduced to a 12-job problem.

Throughout the paper, we use the test problems generated by Crauwels et al. [6]. In their paper, 125 test instances are available for each problem size $n = 40$, $n = 50$ and $n = 100$. The instances were randomly generated as follows: For each job j ($j = 1, \ldots, n$), an integer processing time p_j was generated from the uniform distribution [1,100] and an integer processing weight w_j was generated from the uniform distribution [1,10]. Instance classes of varying difficulty were generated by using different uniform distributions for generating the due dates. For a given relative range of due dates RDD (RDD = 0.2, 0.4, 0.6, 0.8, 1.0) and a given average tardiness factor TF (TF = 0.2, 0.4, 0.6, 0.8, 1.0), an integer due date d_j for each job j was randomly generated from the uniform distribution [P(1-TF-RDD/2), P(1-TF+RDD/2)], where $P = \sum_{j=1,\ldots,n} p_j$. Five instances were generated for each of the 25 pairs of values of RDD and TF, yielding 125 instances for each value of n.

These problem sets can be found at J. E. Beasley's Weighted tardiness OR-Library at the web site http://mscmga.ms.ic.ac.uk/jeb/orlib/wtinfo.html. In this web site, there are three files wt40, wt50, and wt100 containing the instances of size 40, 50, and 100 respectively. Optimal values of solutions are available for 124 and 115 of the 40 and 50 job problem instances, respectively. The values for the unsolved problems given in the files wtopt40 and wtopt50 are the best known to Crauwels et al. [6]. The values of the solutions not known to optimality have not been improved upon since and may well be optimal. The best solution values known to Crauwels et al. for the 100 job problems are given in file wtbest100a. These solution values were used as the best known by both Crauwels et al. and Congram et al. [5]. Therefore using the best solution values known to Crauwels et al. allows results from future heuristics to be compared directly with the tables given in these papers. A dynamic programming based local search heuristic dynasearch by Congram et al. has in some cases found better solutions to the 100 job problems than those known by Crauwels et al. The best known solutions to date are given in the file wtbest100b in OR-Library.

We first used these data sets to evaluate the impact of the global dominance rule on each problem set as summarized in Table 1. As a result, we can reduce the problem size for certain problem instances.

$* * * * * * * * * * * *$ Insert Table 1 around here $* * * * * * * * * * *$

These results are also consistent with Hall and Posner's [8] suggestion that the number of jobs fixed by the global dominance rule indicates which problems are hard and which ones are easy. This supports the idea that varying RDD and TF, as originally proposed by Potts and Van Wassenhove [13], thus creating a range of problem difficulties.

2.1 Local Dominance Rule

We denote the global dominance theorem as a static dominance rule, and also employ a time-dependent local dominance rule proposed by Akturk and Yildirim [2]. They show that the arrangement of adjacent jobs in an optimal schedule depends on their start times. For each pair of jobs i and j that are adjacent in an optimal schedule, there can be a critical value t_{ij} such that i precedes j if processing of this pair starts earlier than t_{ij} and j precedes i if processing of this pair starts after t_{ij}. In their rule, there are two possibilities for each pair of jobs. Either there is at least one breakpoint or an unconditional ordering. A *breakpoint* is a critical start time for each pair of adjacent jobs after which the ordering changes direction such that if $t \leq breakpoint$, i precedes j, denoted by $i \prec j$, (or j precedes i) and then j precedes i, denoted by $j \prec i$, (or i precedes j). If i *unconditionally* precedes j, denoted by $i \rightarrow j$, then the ordering does not change, i.e. i always precedes j when they are adjacent, but it does not imply that an optimal sequence exists in which i precedes j. They show that there are at most three possible breakpoints for the $1| |\sum w_i T_i$ problem as shown below.

$$t_{ij}^1 = [(w_i d_i - w_j d_j)/(w_i - w_j)] - (p_i + p_j) \tag{1}$$

$$t_{ij}^2 = d_j - p_i - p_j(1 - w_i/w_j) \tag{2}$$

$$t_{ij}^3 = d_i - p_j - p_i(1 - w_j/w_i) \tag{3}$$

As a result, they state the following general rule that provides a sufficient condition for schedules that cannot be improved by adjacent job interchanges. They show that if any sequence violates the proposed dominance rule, then switching these jobs either lowers the total weighted tardiness or leaves it unchanged.

General Rule:

```
IF $d_i = d_j$
THEN IF $p_i w_j \geq p_j w_i$
    THEN $j \rightarrow i$
    ELSE IF $w_i \geq w_j$
        THEN $i \rightarrow j$
        ELSE $j \prec i$ for $t \leq t_{ij}^3$
             $i \prec j$ for $t \geq t_{ij}^3$
ELSE IF $p_j(w_j - w_i) > (d_j - d_i)w_j$
    THEN $i \prec j$ for $t \leq t_{ij}^1$
        IF $p_i w_j < p_j w_i \wedge p_i(w_j - w_i) > (d_j - d_i)w_i$
        THEN $j \prec i$ for $t_{ij}^1 \leq t \leq t_{ij}^3$
             $i \prec j$ for $t \geq t_{ij}^3$
        ELSE $j \prec i$ for $t \geq t_{ij}^1$
    ELSE IF $p_i w_j \leq p_j w_i$
        THEN $i \rightarrow j$
        ELSE $i \prec j$ for $t \leq t_{ij}^2$
             $j \prec i$ for $t \geq t_{ij}^2$
```

It is well-known that the shortest weighted processing time (SWPT) rule gives an optimal sequence for the $1| |\sum w_j T_j$ problem when either all due dates are zero or all jobs are tardy, i.e.

$t > max_{i \in N}\{d_i - p_i\}$. Under this situation the problem reduces to total weighted completion time problem which is known to be solved optimally by the SWPT rule as shown by Smith [17], in which jobs are sequenced in non-increasing order of w_j/p_j. Furthermore, let V be the set of pairs (i, j) for which there is at least one breakpoint t_{ij}, $i, j \in V$. The largest of these breakpoints is equal to $t_l = \max_{(i,j) \in V}\{t^1_{ij}, t^2_{ij}, t^3_{ij}\}$. The following lemma by Akturk and Yildirim [2] can be used to find an optimal sequence for the remaining jobs on hand after a time point t_l. We know that $t_l \leq max_{i \in N}\{d_i - p_i\}$, so we enlarge the region for which the $1| \mid \sum w_j T_j$ problem can be solved optimally by the SWPT rule.

Lemma 2: If $t > t_l$ then the SWPT rule gives an optimum sequence for the remaining unscheduled jobs.

Potts and Van Wassenhove [14] applied an adjacent pairwise interchange (API) method starting with the heuristic sequence obtained by applying the apparent tardiness cost (ATC) rule. The ATC rule is a composite dispatching rule that combines the SWPT rule and the minimum slack rule. Under the ATC rule jobs are scheduled one at a time; that is, every time the machine becomes free, a priority index is computed for each remaining job i. The job with the highest priority index is then selected to be processed next. The priority index is a function of the time t at which the machine became free as well as the p_i, the w_i, and the d_i of the remaining jobs. The index is defined as:

$$a_i(t) = \frac{w_i}{p_i} \cdot \exp\left(-\ max\ (0\,,\ d_i - t - p_i)\ /\ (k \cdot \bar{p})\right)$$

where we set the look-ahead parameter k at 2 as suggested in Morton and Pentico [11], and $\bar{p}$ is the average processing time of the remaining unscheduled jobs. Akturk and Yildirim [3] show that the ATC rule performs quite well relative to the other dispatching rules. In the following example, we will demonstrate how the proposed local dominance rule can be used to improve the weighted tardiness criterion even for a reasonably efficient rule like ATC. Consider the 2-job problem where (Job, d, p, w)= (i, 30, 6, 4), (j, 40, 4, 10). Since $\bar{p} = 5$, the following ranking indexes can be calculated for each job:

$$a_i(t) = \frac{4}{6}\exp\left(-(\max(0, 30 - t - 6))\ /\ (2*5)\right)\ <\ a_j(t) = \frac{10}{4}\exp\left(-(\max(0, 40 - t - 4))\ /\ (2*5)\right)$$

Therefore, we can easily show that $j \Rightarrow i$ for all t under the ATC rule. But the general rule indicates that there is a breakpoint t^2_{ij} for this pair, and i should precede j if their processing starts in the time interval [20, 31.6]. In this work the constructive heuristic we use is ATC augmented with global dominance criteria denoted as ATC_GD. At each decision point we must decide which job to schedule next. We first determine the set of eligible jobs using global dominance criteria. If only one job is eligible, it is scheduled. If more than one job is eligible, we use ATC priorities to determine the next job.

There are different ways to implement an API method. The most obvious one is a strict descent method (STRICT), in which the only adjacent pairwise interchanges leading to a decrease in the objective function value are accepted. But, there can be many neutral moves for the $1| \; |\sum w_j T_j$ problem, especially at the beginning of the sequence. Hence, another API method would be to implement the earliest due date (EDD) rule as a tie breaking criterion when there is a neutral move. In Tables 2 and 3, we compare three different API methods, namely STRICT, EDD as a tie breaking criterion, and the proposed local dominance rule (LDR) to improve the initial sequence given by the ATC_GD rule on each test problem. As we have discussed above there are 125 instances for each value of n. In Table 2, we give statistics on the pairwise comparison of STRICT, EDD and LDR based API methods. For example, when we compare the STRICT method with EDD, $(<)$ represents the number of instances (out of 125) in which the STRICT gives a smaller weighted tardiness value than the EDD, where as $(=)$ represents the number of instances in which both methods give the same value, and $(>)$ represents the number of instances for which the EDD gives better results.

* * * * * * * * * * * * Insert Tables 2 and 3 around here * * * * * * * * * * **

As it can be seen from these tables, all of the API methods provide a significant improvement over the ATC rule in terms of the average deviation and the maximum deviation from the optimum (or best known) solution, and the number of times (out of 125) that an optimum (or best known) solution is found, denoted as total match, for each problem type. These runs are taken on a PC Pentium II 400 MHz, and the average CPU times are in seconds. Furthermore, both EDD and LDR based API methods are better than the strict API method as expected. Since the computation times are so small for the smaller problems, we can only make meaningful comparisons for larger problems. In sum, the proposed local dominance rule not only provides a better solution quality, but also results in a smaller CPU time compared to other API methods. The LDR method is faster because there is no need to consider the change in the objective function value for each pair of jobs and make a change accordingly. We swap jobs if they violate the general rule. Therefore, we stop when all pairs satisfy the general rule, as opposed to defining an arbitrary fixed number of iterations in the descent methods with neutral moves. We also improve the solution quality because the descent methods with neutral moves swap a job pair arbitrarily (or according to the EDD rule as in our case), whereas in the LDR method there is actually no neutral move and at each time point for each job pair one job dominates the other one.

3 Problem Space Genetic Algorithms

In this section we describe Problem Space Genetic Algorithms (PSGA) for the single machine weighted tardiness problem which rely heavily on the methods developed in previous sections. We then present computational results on the test problems of Crauwels et al. [6] introduced earlier.

Problem Space Genetic Algorithms have been used successfully in the past on various scheduling problems [12], [19]. At the heart of a PSGA is a constructive heuristic $H()$ which maps a problem instance to a sequence. Given any sequence, the objective function $V()$ (total weighted tardiness) can be calculated. The heuristic $H()$ operates on data from the problem instance. For example if the base heuristic is shortest processing time (SPT) first dispatch, then the relevant data used by $H()$ are the job processing times. Now suppose the processing times are perturbed randomly, and $H()$ is applied to the perturbed processing times. The likely result is a new sequence. This new sequence can be evaluated with the objective function $V()$. Of course the original (unperturbed) processing times must be used when evaluating the objective function of any sequence. Let P be an N-vector of processing times and b be a vector of perturbations. We form the following optimization problem:

$$\min_{b} V[H(P+b)]$$

The result is an optimization problem defined on $b \in \mathcal{R}^N$, the space of perturbations. Properties of this optimization problem have been discussed previously in [18], but may be summarized as follows. The optimal solution exists in the space under quite general conditions. The gradient of $V()$ is zero at every point in the space. Intuitively, good solutions will tend to lie near the point $b = 0$. This makes sense since we expect the heuristic to provide reasonable solutions to the original problem when perturbations are small. Conversely, with large perturbations, the perturbed data bears little resemblance to the original data, and the heuristic can be expected to produce poor sequences. We need a derivative free means to search this real valued perturbation space as the gradient is zero everywhere. We propose two methods, random search and genetic algorithms. In random search we generate each element of the perturbation vector from a Uniform $(-\theta, \theta)$ distribution where θ is a tuning parameter. Thus in our example we would perturb each processing time, apply SPT to obtain a sequence, then apply $V()$ to measure the objective function value of the sequence. We repeat this procedure many times (i.e. generate many sequences) and report the best solution found.

A PSGA uses the perturbation vector as the encoding of a solution (or chromosome). Note that a chromosome (perturbation vector) b may be decoded into a sequence by applying $H(P+b)$, and its value by applying $V[H(P+b)]$. Unlike many applications of genetic algorithms to sequencing problems, standard crossover operators may be applied under this encoding.

In this work, the perturbation scheme involves assigning a perturbation b_j to each job j. At each decision point, ATC priorities a_j are calculated for each eligible job j. Next the ATC priorities a_j are normalized into the interval [0,1] yielding n_j as follows:

$$\text{Let } a_{min} = \min_j a_j \text{ and } a_{max} = \max_j a_j \quad \text{Then } n_j = (a_j - a_{min})/(a_{max} - a_{min})$$

Perturbations are then added to the normalized priorities, and the job with the highest perturbed normalized priority $n_j + b_j$ is scheduled next. Once the entire sequence has been constructed in this fashion, we have the additional option of performing adjacent pair-wise interchange (API) to further improve the sequence. We experiment with two different base heuristics for the PSGA, one with API and one without. For the tuning experiments, we used the base heuristic with API included.

As is standard with genetic algorithms, several design parameters must be specified. We first define these parameters and provide further detail on the PSGA. Then we report the results of experiments designed to determine appropriate *a priori* values of the tuning parameters in an unbiased manner.

3.1 PSGA Description

In each iteration (or generation) of the PSGA a population of chromosomes (perturbation vectors) is created. The number of such perturbation vectors, 'POPSIZE' is one parameter of the PSGA. The initial population of perturbation vectors is created using random search as described above. We simply create POPSIZE N-vectors where each element of each vector is Uniform $(-\theta, \theta)$. Thus the perturbation magnitude θ is the second tuning parameter of the PSGA. The number of generations (iterations) for which the PSGA is set to 200 in all experiments. Clearly more iterations will yield better results, but with diminishing returns. 200 generations seems to balance performance and computation time in a reasonable way. In the testing phase we will contrast the merits of one long run against several shorter runs.

A new generation of solutions is created from the previous one by 'selectively breeding solutions'. Selectivity is accomplished by selecting members of the old generation for breeding in such a way that preference is given to better solutions. In our algorithm we compute a 'fitness' $f(i)$ for each member i of the current generation. Specifically, $f(i)$ is the probability that solution i will be selected for breeding. Fitness is calculated from $V(i)$ the total weighted tardiness of solution i as follows:

$$\text{Let } V_{max} = \max_i V(i) \quad \text{Then } f(i) = (V_{max} - V(i))^{\pi} / \sum_i (V_{max} - V(i))^{\pi}$$

The tuning parameter π determines the selectivity of the algorithm. Note that as π increases, better solutions having increasingly greater chances of being selected. If π is too large, the population will quickly loose diversity and converge so that all members of a generation are identical. If π is too small, the algorithm will converge very slowly thus using excessive computation time.

In our case we use two reproduction methods, sexual and asexual. In asexual reproduction, an individual from the current population is selected randomly according to fitness $f(i)$, and passed directly to the next generation. In sexual reproduction, two 'parent solutions' are selected according to $f(i)$ and combined through crossover to produce an 'offspring' solution which is passed to the next generation. The percent of solutions '%SEXUAL' generated by sexual rather than asexual reproduction is another tuning parameter of our PSGA. We also implemented 'elitist reproduction'. Here the best member of generation k is automatically passed as the first member of generation $k+1$. This guarantees that the best solution is not lost to random selection.

When using sexual reproduction, parent solutions must be combined to form an offspring using a crossover operator. We experiment with two well known and simple crossover operators, single point and uniform. In single point, an integer uniform$[1, N-1]$ random number C is generated. Then elements 1 to C from parent 1 and elements $C+1$ to N of parent 2 are copied to create the new offspring vector. In uniform crossover, we select elements of the offspring one at a time. For each element j in the offspring vector, we select element j from either parent with probability 0.5.

Once a new generation of perturbation vectors has been created, mutation is applied. Each element of each vector in the new generation may be mutated. The probability of mutating an element is given by the mutation probability tuning parameter 'MUTPROB'. If selected for mutation, the element is replaced by a newly generated uniform $U(-\theta, \theta)$ random perturbation.

3.2 Tuning Experiments

With so many tuning parameters, it is necessary to conduct experiments to determine appropriate values for each of the parameters. Further, it is necessary to assure that the experiments do not bias the final results of the tuned algorithm. To maintain unbiasedness, we generated a set of problems independent of the problems ultimately used in testing. Based on some initial trials, an experiment was designed to study the tuning parameters at levels given in Table 4. The last two factors RDD and TF determined the type of problem generated. By varying the RDD and TF factors, the experiments covered a broad range of problem types. This will help find tuning parameter values that work well across the range of possible problem types. For each of the 25 combinations of RDD and TF, 1 problem with 100 jobs was generated. A full factorial experiment was conducted over all factors. For each combination of tuning parameters and for each test problem, two algorithm runs were made with different random number seeds, yielding two replicates.

* * * * * * * * * * * * Insert Table 4 around here * * * * * * * * * **

In our first pass analysis of the results of the experiment we observed that the variance was not constant across the factors RDD and TF. As non-constant variance violates the basic assumptions of the analysis of variance, remedial measures were necessary. The reason for non-constant variance was readily apparent. Some combinations of RDD and TF produced easy problems with 'loose' due dates. Regardless of the tuning parameter values, the PSGA easily found a solution with zero total weighted tardiness. Other combinations of RDD and TF produced problems with a spectrum of difficulty levels. In general we observed higher variance on harder problems. Within each of the 25 cells formed by combinations of RDD and TF, we had 192 responses. To stabilize the variance we transformed each response to be percentage from best solution among the 192 in each cell. We then performed an analysis of variance on the transformed data.

From the results of an analysis of variance, we first note that crossover type and selectivity parameter π have no significant effect on the results. For subsequent testing runs of the algorithms, we chose to use single point crossover and selectivity $\pi = 4$ arbitrarily. We also note the percent sexual reproduction has little effect. We discuss this further below.

Perturbation magnitude θ, population size, and mutation probability all showed significant effects. It is expected that population size would be significant. Since the number of generations was fixed at 100, twice the number of solutions are generated with POPSIZE 100 as opposed to 50. As one would expect, both solution quality and computation time increase with population size. In the testing phase we will use both population sizes to indicate the marginal returns of generating more solutions.

The perturbation magnitude θ was also significant as expected. The experiment results show that $\theta = 0.5$ performed poorly, and that $\theta = 1$ was marginally better than $\theta = 2$. Our experience with Problem Space Search methods indicates that performance is poor when θ is too small, but that performance is reasonably robust when θ is larger than its optimal value. The experimental results match precisely what we have learned from experience. Since $\theta = 1$ provided the best results, we chose this level in subsequent testing.

The final significant tuning parameter was mutation probability. A closer examination revealed an interaction between population size and mutation probability. When both mutation probability and population size were at their low levels (0.01 and 50 respectively), algorithm performance was poor. At all other combinations of levels, performance was roughly the same. Mutation is necessary to maintain diversity in the gene pool of a genetic algorithm. This is especially true in problem space genetic algorithms where we have found that aggressive selectivity works well. Our conclusion is that when both POPSIZE and MUTPROB are at their low levels, diversity is lost too quickly leading to poor performance. Since we will test with both small and large population sizes, and since mutation probability interacts with POPSIZE, we decided not to fix MUTPROB *a priori*, but rather to examine its effects in testing as well.

We noted previously that no difference in performance was observed when the percent sexual reproduction varied from 80% to 100%. Mason [10] suggests that in many applications of genetic algorithms, crossover is unnecessary, and that 100% asexual reproduction works as well. Mason's suggestion is somewhat controversial as it calls into question one of the basic premises of genetic algorithms. To test Mason's conjecture in the context of our problem, we set percent sexual reproduction to 0 as well as to 0.8 in the testing phase. In addition, we also include in the testing phase the random search algorithm described previously. This may be viewed as a 'naive' search strategy. By comparing results from random search to the various versions of genetic algorithms, we can determine the value of an evolutionary strategy on this problem. While we will discuss results of testing in greater detail soon, the quick summary is that random search performed poorly relative to all versions of genetic algorithms. Further, the algorithm without crossover performed worse than the versions with crossover, but the difference was quite small.

To summarize, we proceed to the testing phase having fixed the following tuning parameters as perturbation magnitude $\theta = 1.0$, selectivity $\pi = 4$, and the crossover type is single point.

4 Computational Results

To test the efficiency of the proposed PSGA, the required programs were coded in C language, compiled with Borland compiler, and run on a Gateway 2000 model GP6-400 PC Pentium II 400 MHz with a memory of 96 MB Ram. The proposed algorithm was tested on a series of randomly generated problems developed by Crauwels et al. [6] as discussed in Section 2. In addition, we also generated new 125 test problems for $n = 200$ using the same uniform distributions for p_j, w_j, RDD and TF. These new test problems and the computer codes for PSGA and LDR methods can be obtained from the authors.

In addition to the parameters discussed in the previous section, we also tested a few versions of the PSGA. We created two versions with different base heuristics; ATC with global dominance, and ATC with global dominance and the local dominance rule based API method. We also experimented with several short GA runs against one longer run. In the first case we made 1 run of 1000 generations, denoted as single-start. In the second case we ran the GA five times for 200 generations each, denoted as multi-start. To summarize, we will test various PSGA algorithms as summarized in Table 5. In each problem instance, all combinations of the levels of these parameters are investigated. It is a 2^5 full-factorial design, which corresponds to 32 alternative parameters settings.

∗ ∗ ∗ ∗ ∗ ∗ ∗ ∗ ∗ ∗ ∗ ∗ Insert Table 5 around here ∗ ∗ ∗ ∗ ∗ ∗ ∗ ∗ ∗ ∗∗

These alternative algorithms are compared in terms of the average deviation and the maximum deviation from the optimum (or best known) solution, the number of times (out of 125) that an optimum (or best known) solution is found, denoted as total match, and the average CPU times in seconds as summarized in Tables 6 to 9 for each value of n. We also report the average number of generations that gives the iteration number in which the best solution is found. If this number is small that means the same solution can be found in a shorter run with a less CPU time. The best known solution values known to Crauwels et al. for the 100 job problems are given in file wtbest100a. In our tables, these results are given as $n = 100$ (A), whereas the best known solutions to date are given as $n = 100$ (B). Therefore, in some cases we found better solutions to the 100 job problems than those known by Crauwels et al. indicated as the number of improvements. As a result, total match values for $n = 100$ (A) indicate the number of times we found the same solution given in file wtbest100a, hence the actual match to the best known solution is the summation of total match with the number of improvements. In order to find a best solution to each one of the 125 instances for 200 job problems, we took a very long run with an API method with the following parameters: POPSIZE=100, the maximum number of generations = 5000 (as opposed to 200 in all experiments), RSEXUAL = 0.8, MUTPROB = 0.01, $\theta = 1.0$, $\pi = 4$, the crossover type is single point, and the number of restarts is 10 (instead of 5 in the multi-start).

∗ ∗ ∗ ∗ ∗ ∗ ∗ ∗ ∗ ∗ ∗ ∗ Insert Tables 6, 7, 8 and 9 around here ∗ ∗ ∗ ∗ ∗ ∗ ∗ ∗ ∗ ∗∗

In this section, we also test the validity of several questions. The first question is raised by Mason [10] that in many applications of genetic algorithms the crossover operation is unnecessary. In Tables 6 and 7, we compare these algorithms for the single-start PSGA, whereas in Tables 8 and 9 for the multi-start PSGA. The algorithm without crossover (i.e. RSEXUAL=0 or asexual) performed worse than the versions with crossover (i.e. RSEXUAL=0.8) for both the single and multi-start PSGA, especially for the maximum deviation criterion. This difference becomes even more evident for the multi-start PSGA. The second question is related to the number of restarts. Crauwels et al. reported that in order to obtain higher quality solutions, multi-start versions are preferable to single-start longer run versions. Specifically, their multi-start GA version, denoted as GA(B,5) in their notation, performed better than the single-start GA, denoted as GA(B,1), although GA(B,5) takes approximately five times longer CPU time than the GA(B,1). In our case, the average CPU times are more or less equal. When we compare Tables 6 and 7 with Tables 8 and 9, we can also conclude that the multi-start PSGA is slightly better than the single-start PSGA for $n = 40$, 50 and 100. But, when $n = 200$ the single-start PSGA is better, implying that 200 iterations in each run may not be long enough for the multi-start PSGA.

Next, we discuss the importance of capturing local minima information to guide the local search heuristic. As we have mentioned above, there are 32 alternative PSGA algorithms. For each algorithm, we compare a straightforward PSGA implementation, denoted as GA, with another using a local dominance rule based API search, denoted as GA+LDR. The difference between GA and GA+LDR is quite striking. By defining and searching through a set of local minima, we were able to improve the solution quality in all measures significantly with a relatively small increase in the CPU time. In terms of the overall solution quality, the best solution is given by the parameter setting of RSEXUAL=0.8, MUTPROB=0.01 and POPSIZE=100 for both single and multi-start versions of PSGA+LDR. They are also better than the corresponding best single and multi-start local search heuristics reported in the literature. It is important to note that a genetic algorithm with the permutation representation performed so poorly compared to other local search heuristics, Crauwels et al. did not even report their results. In their study, the best results were obtained with tabu search. Genetic algorithm with their binary encoding scheme became comparable and used less computation time than the other local search heuristics, but still tabu search gave the best results in terms of the average and maximum deviation. The overall results of Crauwels et al. were very good, so that there was little margin to improve upon them. In PSGA an intelligent mixture of exploiting the basic structure of the problem through global and time-dependent local dominance rules, and an efficient problem space search heuristic with the genetic algorithm improved the results quite significantly. Based on our computational results, the proposed PSGA provided the largest total match, and the smallest average and maximum deviation (which also indicates its robustness) on a large set of test problems compared to the different versions of other local search algorithms, namely simulated annealing, threshold accepting, tabu search and genetic algorithm, reported in Crauwels et al.

We will now elaborate on computational time requirements of the proposed algorithm. Another important contribution of the PSGA is its running time behavior as a function of the number of jobs. In Crauwels et al., the local search heuristics were run on a HP 9000 - G50 computer. In their study, the genetic algorithm was the fastest among the other local search heuristics. Since our runs were taken on a PC Pentium II, we could not compare the computational difficulty of the PSGA with the other local search heuristics in terms of the actual CPU requirements. Instead, we normalized both of the algorithms, i.e. single and multi-start PSGA+LDR, and GA(B,1) and GA(B,5) by Crauwels et al., in terms of the average CPU requirements for $n = 40$, and plotted them as a function of the number of jobs as shown in Figure 1. It is important to note that Crauwels et al. did not take runs for $n = 200$. A gradual increase in running time is another advantage of problem space search heuristic over the other local search heuristics in the literature.

$* * * * * * * * * * * *$ Insert Figure 1 around here $* * * * * * * * * * **$

Finally, these various genetic algorithms are compared to random search with $\theta = 1.0$ as summarized in Table 1. It can easily be seen from these results that random search performed poorly relative to all versions of genetic algorithms. This result indicates the value of an evolutionary strategy of the genetic algorithms on this problem. When we analyze the impact of the local dominance rule based API search, denoted as RS+LDR, over the random search, RS, the results are consistent with the previous ones. It makes a significant improvement over the RS algorithm, but the amount of improvement still was not good enough to be comparable with the other genetic algorithms.

$* * * * * * * * * * * *$ Insert Table 10 around here $* * * * * * * * * * **$

5 Concluding Remarks

We propose a new problem space genetic algorithm to solve single machine total weighted tardiness scheduling problems. In sum, the global dominance rules are very useful for reducing the search space. The time-dependent local dominance rule based API local search method is a powerful exploitation (intensifying) tool since we know that the global optimum is one of the local optimum solutions. If we search through a set of local optimum solutions, it is most likely that our search space will contain the good solutions. Finally, problem space search method significantly enhances exploration (diversification) capability of the genetic algorithm. When we combine all of these attributes, we can improve the solution quality in terms of the average deviation and total match as well as the robustness in terms of the maximum deviation from the optimum solution. Another important positive characteristic of the proposed algorithm is its running time behavior as a function of the number of jobs compared to the other local search algorithms reported in Crauwels et al. [6] as discussed in the previous section.

Acknowledgment

The authors would like to thank Prof. C.N. Potts and Prof. H.A.J. Crauwels for sharing their test results with us. This work was supported in part by the NATO Collaborative Research Grant CRG–971489 and NSF Grant DMI 9809479.

References

[1] Abdul-Razaq, T.S., Potts, C.N. and Van Wassenhove, L.N. (1990) A survey of algorithms for the single-machine total weighted tardiness scheduling problem. *Discrete Applied Mathematics*, **26**, 235-253.

[2] Akturk, M.S. and Yildirim, M.B. (1998) A new lower bounding scheme for the total weighted tardiness problem. *Computers & Operations Research*, **25** (4), 265-278.

[3] Akturk, M.S. and Yildirim, M.B. (1999) A new dominance rule for the total weighted tardiness problem. *Production Planning & Control*, **10** (2), 138-149.

[4] Anderson, E.J., Glass, C.A. and Potts, C.N. (1997) Machine Scheduling. In: *Local Search in Combinatorial Optimization*, edited by E. Aarts and J.K. Lenstra, 361-414.

[5] Congram, R.K., Potts, C.N. and van de Velde, S.L. (1998) An iterated dynasearch algorithm for the single-machine total weighted tardiness scheduling problem. In preparation.

[6] Crauwels, H.A.J., Potts, C.N. and Van Wassenhove, L.N. (1998) Local search heuristics for the single machine total weighted tardiness scheduling. *INFORMS Journal on Computing*, **10** (3), 341-350.

[7] Emmons, H. (1969) One machine sequencing to minimize certain functions of job tardiness. *Operations Research*, **17**, 701-715.

[8] Hall, N.G. and Posner, M.E. (1999) Generating experimental data for machine scheduling problems. To appear in *Operations Research*.

[9] Lawler, E.L. (1977) A 'Pseudopolynomial' algorithm for sequencing jobs to minimize total tardiness. *Annals of Discrete Mathematics*, **1**, 331-342.

[10] Mason, A.J. (1996) A non-linearity measure of a problem's crossover suitability, *IEEE International Conference on Evolutionary Programming*, Perth, Australia.

[11] Morton T.E. and Pentico, D.W. (1993) *Heuristic Scheduling Systems with Applications to Production Systems and Project Management* (New York: John Wiley).

[12] Naphade, K., Wu, S.D. and Storer, R.H. (1997) Problem space search algorithms for resource constrained project scheduling. *The Annals of Operations Research*, **70**, 307-326.

[13] Potts, C.N. and Van Wassenhove, L.N. (1985) A branch and bound algorithm for total weighted tardiness problem. *Operations Research*, **33**, 363-377.

[14] Potts, C. N. and Van Wassenhove, L. N. (1991) Single machine tardiness sequencing heuristics. *IIE Transactions*, **23**, 346-354.

[15] Rachamadugu, R.M.V. (1987) A note on weighted tardiness problem. *Operations Research*, **35**, 450-452.

[16] Rinnooy Kan, A.H.G., Lageweg, B.J. and Lenstra, J.K. (1975) Minimizing total costs in one-machine scheduling. *Operations Research*, **23**, 908-927.

[17] Smith, W.E. (1956) Various optimizers for single-stage production. *Naval Research Logistics Quarterly*, **3**, 59-66.

[18] Storer, R.H., Wu, S.D. and Vaccari, R. (1992) New search spaces for sequencing problems with application to job shop scheduling. *Management Science*, **38** (10), 1495-1509.

[19] Storer, R.H., Wu, S.D. and Vaccari, R. (1995) Local search in problem and heuristic space for job shop scheduling. *ORSA Journal on Computing*, **7** (4), 453-467.

[20] Szwarc, W. (1993) Adjacent orderings in single machine scheduling with earliness and tardiness penalties. *Naval Research Logistics*, **40**, 229-243.

[21] Volgenant, A. and Teerhuis, E. (1998) Improved heuristics for the n−job single machine weighted tardiness problem. *Computers & Operations Research*, **26** (1), 35-44.

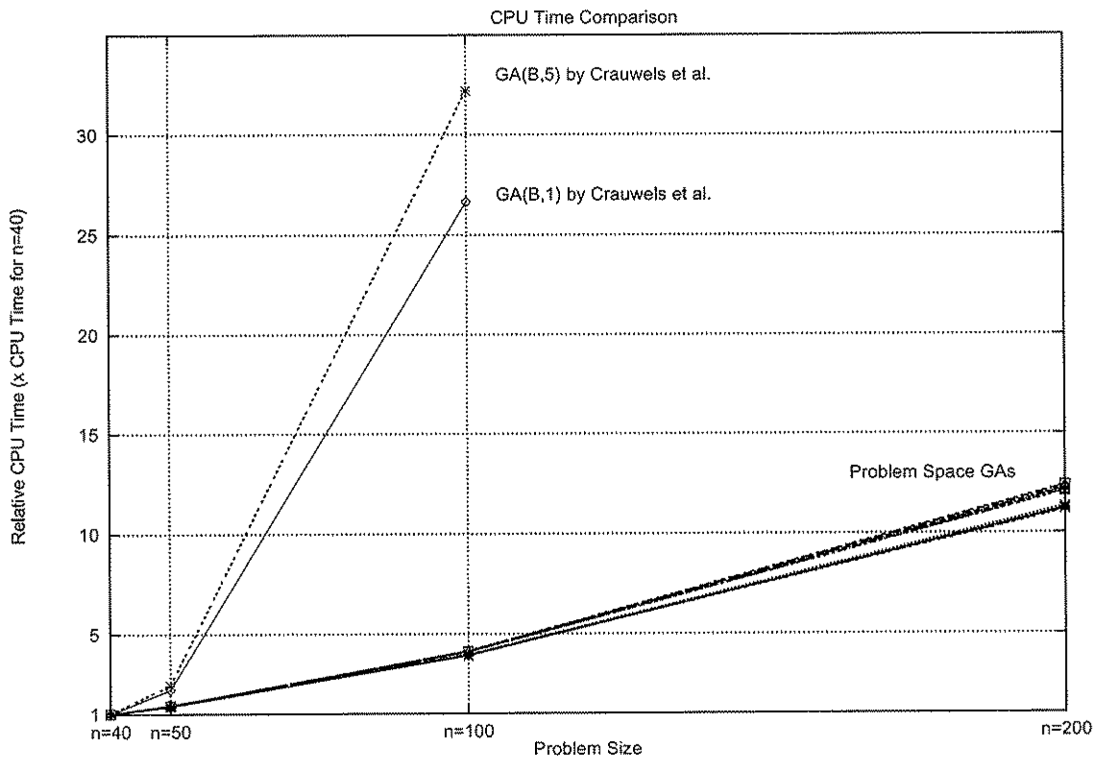

Figure 1: CPU Time Comparison

	Ave. # of Jobs in the First block	Ave. # of Jobs in the Last Block	Ave. # of Remaining Jobs	Global Dom. Matrix Density	# of Problems Solved	Percent Reduction	Ave. CPU time
n=40	10.0720	0.2720	29.6560	59.2226%	18	25.8600%	0.003496
n=50	12.4400	0.1920	37.3680	59.8740%	17	25.2640%	0.002288
n=100	25.5920	1.0640	73.3440	60.8698%	18	26.6560%	0.017752
n=200	55.9840	0.0960	143.9200	61.5298%	20	28.0400%	0.109696

Table 1: Problem Reduction Analysis

	STRICT < EDD	STRICT = EDD	STRICT > EDD	STRICT < LDR	STRICT = LDR	STRICT > LDR	EDD < LDR	EDD = LDR	EDD > LDR
n=40	5	59	61	5	59	61	0	125	0
n=50	6	45	74	6	45	74	0	125	0
n=100	2	36	87	2	36	87	0	121	4
n=200	3	30	92	3	30	92	0	124	1

Table 2: Comparison of Adjacent Pairwise Interchange Algorithms

		ATC	ATC_GD	ATC_GD + API_STRICT	ATC_GD + API_EDD	ATC_GD + API_LDR
n=40	Total Match	19	19	27	45	45
	Av. Dev.	18.3198%	17.0541%	10.6982%	5.3883%	5.3883%
	Max. Dev.	274.4681%	355.3191%	274.4681%	106.4815%	106.4815%
	CPU Time	0.000880	0.000032	0.000040	0.000000	0.000000
n=50	Total Match	18	18	23	30	30
	Av. Dev.	50.2877%	12.1905%	6.0318%	5.5521%	5.5521%
	Max. Dev.	4200.0000%	181.8182%	128.2828%	128.2828%	128.2828%
	CPU Time	0.001928	0.000152	0.000168	0.000000	0.000048
n=100 (A)	Total Match	18	18	22	24	25
	Av. Dev.	41.4829%	39.5478%	12.3147%	10.6911%	10.6910%
	Max. Dev.	2225.0000%	2225.0000%	161.7647%	161.7647%	161.7647%
	CPU Time	0.006624	0.001264	0.001368	0.000712	0.000792
n=100 (B)	Total Match	18	18	22	24	25
	Av. Dev.	41.4833%	39.5482%	12.3151%	10.6915%	10.6914%
	Max. Dev.	2225.0000%	2225.0000%	161.7647%	161.7647%	161.7647%
	CPU Time	0.006624	0.001264	0.001368	0.000712	0.000792
n=200	Total Match	21	21	24	24	24
	Av. Dev.	16.1682%	15.4521%	12.1565%	11.9809%	11.9809%
	Max. Dev.	154.0462%	157.0588%	141.7910%	141.7910%	141.7910%
	CPU Time	0.026304	0.004000	0.003680	0.003080	0.002536

Table 3: Comparison of Single Pass Algorithms

Factors	# Levels	Values				
POPSIZE	2	50	100			
Perturbation magnitude θ	3	0.5	1.0	2.0		
Selectivity π	2	2	4			
%SEXUAL	2	80%	100%			
Crossover type	2	Single point	Uniform			
MUTPROB	2	0.01	0.05			
RDD	5	0.2	0.4	0.6	0.8	1.0
TF	5	0.2	0.4	0.6	0.8	1.0

Table 4: Tuning Experimental Settings

Parameter	Level 1	Level 2
Base Heuristic	ATC with global dominance	ATC with global dominance and LDR
POPSIZE	50	100
%SEXUAL	0%	80%
MUTPROB	0.01	0.05
Single/Multi-start	1 run, 1000 generations	5 runs, 200 generations each

Table 5: Problem Space Genetic Algorithm Parameters

		RSEXUAL=0.8							
		MUTPROB=0.01				MUTPROB=0.05			
		POPSIZE=50		POPSIZE=100		POPSIZE=50		POPSIZE=100	
		GA	GA+LDR	GA	GA+LDR	GA	GA+LDR	GA	GA+LDR
n=40	Total Match	81	122	86	125	73	125	79	125
	Ave. Dev.	0.2562%	0.0024%	0.1829%	0.0000%	0.0610%	0.0000%	0.0481%	0.0000%
	Max. Dev.	8.1181%	0.2027%	6.7039%	0.0000%	0.8360%	0.0000%	0.5564%	0.0000%
	Ave. Gen.	225.3	25.9	224.38	9.74	244.31	5.97	277.18	4.78
	Av. CPU Time	13.2922	14.6906	26.2695	29.3271	12.8452	14.4796	25.842	29.1102
n=50	Total Match	57	119	66	121	56	121	57	124
	Ave. Dev.	0.4275%	0.0118%	0.2109%	0.0025%	0.2527%	0.0135%	0.2103%	0.0001%
	Max. Dev.	9.6561%	0.6918%	8.0882%	0.1455%	8.0882%	1.2690%	8.0882%	0.0160%
	Ave. Gen.	274.31	54.53	238.95	33.75	405.3	34.15	365.57	42.11
	Av. CPU Time	18.3628	20.7581	36.2945	41.5422	17.8326	20.5417	35.8942	41.0212
n=100 (A)	Total Match	40	95	39	101	39	86	42	83
	Ave. Dev.	1.0003%	0.0208%	0.1324%	0.0032%	0.3505%	0.0110%	0.2703%	0.0162%
	Max. Dev.	82.0000%	0.9733%	2.6787%	0.0990%	11.4206%	0.2158%	2.1103%	0.2996%
	Ave. Gen.	557.8	211.09	565.89	148.24	618.26	208.38	604.62	233.98
	# of Improv.	0	3	0	5	0	0	0	0
	Av. CPU Time	52.29	60.8376	103.8529	120.75	50.4594	59.166	101.5403	118.9731
n=100 (B)	Total Match	40	94	39	103	39	86	42	83
	Ave. Dev.	1.0007%	0.0212%	0.1328%	0.0035%	0.3509%	0.0113%	0.2707%	0.0166%
	Max. Dev.	82.0000%	0.9733%	2.6787%	0.0990%	11.4206%	0.2158%	2.1103%	0.2996%
	Ave. Gen.	557.8	211.09	565.89	148.24	618.26	208.38	604.62	233.98
	# of Improv.	0	0	0	0	0	0	0	0
	Av. CPU Time	52.29	60.8376	103.8529	120.75	50.4594	59.166	101.5403	118.9731
n=200	Total Match	30	55	33	62	32	47	33	48
	Ave. Dev.	0.8536%	0.1377%	0.4040%	0.0409%	0.7844%	0.1344%	0.6591%	0.0707%
	Max. Dev.	16.8116%	4.6004%	7.7960%	1.2159%	9.0829%	1.9015%	6.5804%	0.5871%
	Ave. Gen.	709.29	429.25	685.41	418.91	611.95	429.07	628.56	433.7
	Av. CPU Time	149.251	181.9651	297.3957	362.0465	143.7175	176.8252	287.5416	351.7058

Table 6: Results for Single-Start PSGA with RSEXUAL=0.8

		RSEXUAL=0 (ASEXUAL GA)							
		MUTPROB=0.01				MUTPROB=0.05			
		POPSIZE=50		POPSIZE=100		POPSIZE=50		POPSIZE=100	
		GA	GA+LDR	GA	GA+LDR	GA	GA+LDR	GA	GA+LDR
n=40	Total Match	72	123	88	124	72	125	77	125
	Ave. Dev.	0.3203%	0.0025%	2.3112%	0.0005%	0.0801%	0.0000%	0.0532%	0.0000%
	Max. Dev.	9.3750%	0.2517%	274.4681%	0.0590%	0.8360%	0.0000%	1.1415%	0.0000%
	Ave. Gen.	271.04	42.36	239.42	23.25	270.93	12.16	278	6.61
	Av. CPU Time	12.938	14.6666	26.1992	29.4514	12.7987	14.5248	25.5321	28.9674
n=50	Total Match	56	116	59	121	55	124	52	123
	Ave. Dev.	0.4174%	0.0235%	0.2813%	0.0018%	0.2716%	0.0009%	0.2208%	0.0007%
	Max. Dev.	9.6561%	1.4074%	8.0882%	0.0840%	8.0882%	0.1065%	8.0882%	0.0840%
	Ave. Gen.	328.94	84.66	346.13	67.87	402.54	46.53	368.82	40.58
	Av. CPU Time	17.9765	20.7405	36.5096	41.5517	17.8655	20.5303	35.6569	40.9649
n=100 (A)	Total Match	33	93	40	99	40	85	38	85
	Ave. Dev.	1.2025%	0.1432%	1.0512%	0.0604%	0.2223%	0.0309%	0.2688%	0.0150%
	Max. Dev.	82.0000%	5.3400%	82.0000%	5.2851%	1.4748%	2.2962%	3.1843%	0.4862%
	Ave. Gen.	607.5	255.3	584.32	187.98	621.67	241.76	595.98	187.92
	# of Improv.	0	1	0	2	0	1	0	2
	Av. CPU Time	51.9356	60.7868	103.9084	121.3718	50.5797	59.7257	101.4332	118.6591
n=100 (B)	Total Match	33	93	40	101	40	86	38	86
	Ave. Dev.	1.2029%	0.1436%	1.0516%	0.0608%	0.2226%	0.0313%	0.2692%	0.0154%
	Max. Dev.	82.0000%	5.3400%	82.0000%	5.2851%	1.4748%	2.2962%	3.1843%	0.4862%
	Ave. Gen.	607.5	255.3	584.32	187.98	621.67	241.76	595.98	187.92
	# of Improv.	0	0	0	0	0	0	0	0
	Av. CPU Time	51.9356	60.7868	103.9084	121.3718	50.5797	59.7257	101.4332	118.6591
n=200	Total Match	31	50	34	46	30	47	33	48
	Ave. Dev.	1.0872%	0.4533%	0.7553%	0.2038%	0.9147%	0.2054%	0.8067%	0.1085%
	Max. Dev.	14.9957%	23.8433%	8.9499%	7.5132%	11.0941%	6.0470%	8.8588%	1.1656%
	Ave. Gen.	698.38	450.03	716.74	461.05	668.12	494.25	681.4	447.09
	Av. CPU Time	148.2765	180.0397	296.1482	360.6524	144.1365	178.562	287.4377	352.0535

Table 7: Results for Single-Start PSGA with RSEXUAL=0 (ASEXUAL)

		RSEXUAL=0.8							
		MUTPROB=0.01				MUTPROB=0.05			
		POPSIZE=50		POPSIZE=100		POPSIZE=50		POPSIZE=100	
		GA	GA+LDR	GA	GA+LDR	GA	GA+LDR	GA	GA+LDR
n=40	Total Match	77	125	94	125	70	125	71	125
	Average Dev.	0.1281%	0.0000%	0.1588%	0.0000%	0.0473%	0.0000%	0.0554%	0.0000%
	Max. Dev.	5.4124%	0.0000%	9.3750%	0.0000%	0.5095%	0.0000%	0.6311%	0.0000%
	Ave. Gen.	78.41	10.83	78.46	7.89	90.22	8.64	90.57	5.72
	Av. CPU Time	13.4297	14.7395	26.5028	29.6225	12.8930	14.5849	25.9505	29.2781
n=50	Total Match	63	123	68	123	53	124	54	123
	Average Dev.	0.1892%	0.0070%	0.1488%	0.0102%	0.2131%	0.0000%	0.1875%	0.0013%
	Max. Dev.	8.0882%	0.8075%	8.0882%	1.2690%	8.0882%	0.0038%	8.0882%	0.1455%
	Ave. Gen.	99.70	19.45	97.84	15.02	109.02	15.73	108.16	15.84
	Av. CPU Time	18.6542	20.9085	36.9680	41.7975	18.0119	20.6300	36.4179	41.4591
n=100 (A)	Total Match	38	94	40	103	41	85	39	87
	Average Dev.	0.9569%	0.0750%	0.1875%	0.0030%	0.4149%	0.0137%	0.4016%	0.0135%
	Max. Dev.	82.0000%	5.2851%	1.5324%	0.1532%	2.8310%	0.2505%	3.2094%	0.2996%
	Ave. Gen.	134.26	73.09	133.75	69.48	124.62	70.48	122.84	65.25
	# of Improv.	0	4	0	3	0	1	0	0
	Av. CPU Time	52.2604	60.7866	104.7119	121.5111	50.8340	59.5725	102.1507	119.4516
n=100 (B)	Total Match	38	98	40	105	41	85	39	87
	Average Dev.	0.9573%	0.0754%	0.1879%	0.0034%	0.4153%	0.0141%	0.4020%	0.0138%
	Max. Dev.	82.0000%	5.2851%	1.5324%	0.1532%	2.8310%	0.2505%	3.2094%	0.2996%
	Ave. Gen.	134.26	73.09	133.75	69.48	124.62	70.48	122.84	65.25
	# of Improv.	0	0	0	0	0	0	0	0
	Av. CPU Time	52.2604	60.7866	104.7119	121.5111	50.8340	59.5725	102.1507	119.4516
n=200	Total Match	30	47	29	55	31	48	31	48
	Average Dev.	0.9134%	0.1280%	0.5751%	0.0541%	0.9648%	0.1933%	0.8092%	0.1129%
	Max. Dev.	12.7390%	4.6639%	7.0449%	1.4723%	9.2044%	7.1767%	4.3796%	1.0511%
	Ave. Gen.	141.85	113.71	138.16	110.88	104.04	99.91	102.49	95.75
	Av. CPU Time	150.5809	180.2802	297.5878	361.1792	144.0840	176.5761	289.8469	353.2201

Table 8: Results for Multiple-Start PSGA with RSEXUAL=0.8

		RSEXUAL=0 (ASEXUAL GA)							
		MUTPROB=0.01				MUTPROB=0.05			
		POPSIZE=50		POPSIZE=100		POPSIZE=50		POPSIZE=100	
		GA	GA+LDR	GA	GA+LDR	GA	GA+LDR	GA	GA+LDR
n=40	Total Match	68	124	79	125	72	125	75	125
	Average Dev.	0.2672%	0.0002%	0.0913%	0.0000%	0.0637%	0.0000%	0.0516%	0.0000%
	Max. Dev.	9.3750%	0.0217%	6.7039%	0.0000%	0.6094%	0.0000%	0.8133%	0.0000%
	Ave. Gen.	89.94	17.88	91.69	12.44	90.25	9.54	89.19	7.97
	Av. CPU Time	13.0656	14.793	26.2642	29.541	12.8786	14.5952	25.8398	29.2362
n=50	Total Match	49	121	59	121	47	124	49	123
	Average Dev.	0.4080%	0.0147%	0.1960%	0.0059%	0.2690%	0.0000%	0.2366%	0.0048%
	Max. Dev.	9.6561%	1.2690%	8.0882%	0.5999%	8.0882%	0.0038%	8.0882%	0.5999%
	Ave. Gen.	110.52	28.27	114.26	25.71	109.39	20.24	107.34	16.39
	Av. CPU Time	18.3262	20.87	36.7861	41.8314	18.016	20.6431	36.2967	41.5255
n=100 (A)	Total Match	33	89	36	91	38	86	40	89
	Average Dev.	1.1953%	0.1503%	0.3965%	0.0197%	0.3942%	0.0155%	0.3804%	0.0106%
	Max. Dev.	82.0000%	8.5119%	5.3555%	0.4841%	3.4492%	0.4056%	3.1372%	0.2996%
	Ave. Gen.	138.38	78.13	138.48	78.33	131.92	75.38	130.79	69.51
	# of Improv.	0	1	0	2	0	1	0	1
	Av. CPU Time	51.921	60.4638	104.1166	121.1499	50.99	59.7868	102.0101	119.5655
n=100 (B)	Total Match	33	90	36	92	38	86	40	90
	Average Dev.	1.1957%	0.1507%	0.3968%	0.0201%	0.3946%	0.0159%	0.3808%	0.0110%
	Max. Dev.	82.0000%	8.5119%	5.3555%	0.4841%	3.4492%	0.4056%	3.1372%	0.2996%
	Ave. Gen.	138.38	78.13	138.48	78.33	131.92	75.38	130.79	69.51
	# of Improv.	0	0	0	0	0	0	0	0
	Av. CPU Time	51.921	60.4638	104.1166	121.1499	50.99	59.7868	102.0101	119.5655
n=200	Total Match	28	45	30	47	30	43	32	46
	Average Dev.	1.3365%	0.3806%	1.1073%	0.2375%	1.1123%	0.2192%	1.0092%	0.1889%
	Max. Dev.	13.8390%	12.1552%	19.1154%	7.3135%	10.5611%	7.7178%	8.6886%	7.8672%
	Ave. Gen.	145.94	113.73	146.72	112.44	125.28	105.22	122.51	107.71
	Av. CPU Time	149.5396	179.8973	298.0862	359.9304	145.5843	177.9311	290.1471	355.2068

Table 9: Results for Multiple-Start PSGA with RSEXUAL=0 (ASEXUAL)

		25000 Iterations		50000 Iterations	
		RS	RS+LDR	RS	RS+LDR
n=40	Total Match	31	103	33	110
	Average Dev.	2.4793%	0.0314%	2.2619%	0.0119%
	Max. Dev.	26.7423%	0.9102%	26.7423%	0.5139%
	Ave. Gen.	9148.06	3742.90	18414.38	6440.94
	Av. CPU Time	7.0350	8.2294	12.9610	15.2357
n=50	Total Match	26	80	27	86
	Average Dev.	3.1341%	0.0902%	2.8740%	0.0553%
	Max. Dev.	13.6284%	1.2690%	12.1585%	1.2690%
	Ave. Gen.	8803.82	4990.54	19836.09	9146.75
	Av. CPU Time	9.7129	11.5011	18.1188	21.1330
n=100 (A)	Total Match	26	42	27	43
	Average Dev.	5.3077%	1.0659%	4.9205%	0.9978%
	Max. Dev.	35.6467%	11.2013%	35.6467%	11.2013%
	Ave. Gen.	9777.14	7790.74	17960.20	16411.95
	# of Improv.	0	0	0	0
	Av. CPU Time	25.3008	29.9983	49.2616	57.8269
n=100 (B)	Total Match	26	42	27	43
	Average Dev.	5.3081%	1.0663%	4.9209%	0.9982%
	Max. Dev.	35.6467%	1.9547%	35.6467%	11.2013%
	Ave. Gen.	9777.14	7790.74	17960.20	16411.95
	# of Improv.	0	0	0	0
	Av. CPU Time	25.3008	29.9983	49.2616	57.8269
n=200	Total Match	26	32	27	32
	Average Dev.	7.4089%	3.2840%	7.0378%	3.0545%
	Max. Dev.	39.7894%	33.1734%	37.1451%	32.6274%
	Ave. Gen.	9766.06	9098.85	18789.19	17955.82
	Av. CPU Time	71.7826	85.4151	142.1866	169.7462

Table 10: Results for Random Search Algorithms

Biographies

Selcuk Avci is a Ph.D. candidate in the Department of Industrial and Manufacturing Systems at Lehigh University. He received his B.S. in Mechanical Engineering from the Middle East Technical University, Turkey and his M.S. in Industrial Engineering from the Bilkent University, Turkey. His current research interests include production planning and scheduling, inventory theory and modern optimization heuristics. Mr. Avci is a student member of INFORMS.

M. Selim Akturk is an Assistant Professor of Industrial Engineering at Bilkent University, Turkey. He holds a Ph.D. in Industrial Engineering from Lehigh University, U.S.A., and B.S.I.E. and M.S.I.E. from Middle East Technical University, Turkey. His current research interests include hierarchical planning of large scale systems, production scheduling, cellular manufacturing systems, and advanced manufacturing technologies. Dr. Akturk is a senior member of IIE and member of INFORMS.

Dr. Robert H. Storer is Professor of Industrial and Manufacturing Systems Engineering and Co-Director of the Manufacturing Logistics Institute at Lehigh University. He received his B.S. in Industrial and Operations Engineering from the University of Michigan in 1979, and M.S. and Ph.D. degrees in Industrial and Systems Engineering from the Georgia Institute of Technology in 1982 and 1986 respectively. His interests lie in operations research and applied statistics with particular interest in heuristic optimization, scheduling and logistics.