Int. J. Mach. Learn. & Cyber. (2012) 3:327–333
DOI 10.1007/s13042-011-0067-3

ORIGINAL ARTICLE

Hybrid heuristics for the single machine scheduling problem with quadratic earliness and tardiness costs

Alok Singh · Jorge M. S. Valente · Maria R. A. Moreira

Received: 25 June 2011 / Accepted: 26 December 2011 / Published online: 3 January 2012

Abstract In this paper we present three hybrid heuristics for the single machine scheduling problem with quadratic earliness and tardiness costs, and no machine idle time. Our heuristic is a combination of a steady-state genetic algorithm and three improvement procedures. The two computationally less expensive of these three improvement procedures are used inside the genetic algorithm to improve the schedule obtained after the application of genetic operators, whereas the more expensive one is used to improve the best solution returned by the genetic algorithm. We have compared our hybrid approaches against existing recovering beam search and genetic algorithms. The computational results show the effectiveness of our hybrid approaches. Indeed, our hybrid approaches outperformed the existing heuristics in terms of solution quality as well as running time.

Keywords Single machine scheduling · Quadratic earliness and tardiness costs · Heuristic · Genetic algorithm

A. Singh (✉)
Department of Computer and Information Sciences,
University of Hyderabad, Hyderabad 500046, India
e-mail: alokcs@uohyd.ernet.in

J. M. S. Valente
LIAAD–INESC Porto L.A., Faculdade de Economia,
Fundação Universidade do Porto,
Rua Dr. Roberto Frias, s/n, 4200-464 Porto, Portugal

M. R. A. Moreira
EDGE, Faculdade de Economia,
Fundação Universidade do Porto, Rua Dr. Roberto Frias, s/n,
4200-464 Porto, Portugal

1 Introduction

Consider n independent jobs $J_1, J_2, \ldots, J_n$, which have to be scheduled on a single machine under the following assumptions: the machine can process at most one job at a time, the machine and jobs are available throughout, the machine can not be idle throughout the processing of jobs, and the preemption of jobs is forbidden. On this machine, each job J_i ($i = 1$ to n) needs a processing time p_i and should ideally be completed on its due date d_i. Completing a job earlier than its due date incurs earliness cost and completing it after its due date incurs tardiness cost. The earliness cost E_i of a job J_i in a particular schedule is defined as $\max(0, d_i - C_i)$ and the tardiness cost T_i as $\max(0, C_i - d_i)$, where C_i is the completion time of job J_i in the schedule in question. The single machine scheduling problem with quadratic earliness and tardiness costs (QETSP) that this paper considers seeks a schedule that minimizes

$$\sum_{i=1}^{n} (h_i E_i^2 + w_i T_i^2)$$

where h_i is the penalty for earliness of job J_i and w_i the same for tardiness.

It may seem at a first glance that scheduling models with a single processor rarely conform to real life applications. However, single machine scheduling environments do indeed occur in many applications (for a specific example, see Wagner et al. [17]). In addition, the performance of many production systems is quite often depends on the quality of the schedules for a single bottleneck machine. Besides, insights gained by studying the single machine scheduling problems can be used to more complex scheduling environments, such as flow shops or job shops.

In this paper, we have assumed that the machine can not be idle throughout the processing of jobs. This assumption is valid in those cases, where either the operating and startup costs of the machine surpass the cost of completing some jobs early or there is a necessity for operating the machine continuously to satisfy the demands. Some real life examples of such production environments can be found in Korman [9] and Landis [10].

As both earliness and tardiness in schedules are punished by early/tardy scheduling models, these models compel the jobs to complete as close as possible to their due dates, thereby backing the just-in-time production philosophy, i.e., producing something exactly when it is needed. The earliness cost may correspond to the cost of deterioration in the quality of perishable commodities and/or the cost of storing the commodities produced till they are needed. Similarly, the tardiness cost may correspond to lost sales, rush shipping costs or loss of goodwill.

We have considered the quadratic earliness and tardiness costs in the objective function in lieu of the commonly used linear costs. Our objective function punishes more heavily the schedules in which some jobs have completed either quite early or quite late, i.e., such an objective function prefers schedules in which costs are more evenly distributed over schedules in which only a few jobs are contributing to nearly all of the cost. Therefore, such an objective function is suitable for those production environments where the cost for non-compliance with due dates increases in a non-linear manner.

Early-tardy scheduling models with quadratic cost functions have been investigated since 1980s. Gupta and Sen [8] investigated a scheduling model which was similar to our model except for the fact that all earliness and tardiness penalties are assumed to be one and proposed a branch-and-bound algorithm and a heuristic rule for this model. Bagchi et al. [2] considered a model where all jobs have a common due date, the same earliness penalty and the same tardiness penalty. The earliness penalty can be different from tardiness penalty. They proposed an enumerative procedure using some dominance rules. Cheng [6] considered the quadratic cost function for the problem of determining the optimal due-dates and scheduling of jobs on a single machine. Baker and Scudder [3] provides an excellent survey of earlier works on early/tardy scheduling models including those with quadratic and non-linear cost functions.

A branch-and-bound algorithm based on a new lower bounding procedure has been proposed by Valente [13] for the QETSP. Valente and Alves [14] developed several dispatching rules and improvement procedures. We are particularly concerned with two improvement procedures proposed in this paper viz. adjacent pairwise interchange (API) and 3-swaps (3SW). Both of these improvement procedures follow an iterative approach. At each iteration, the API procedure starts with the first job in the schedule and considers in succession each pair of adjacent jobs. It swaps two adjacent jobs if it leads to an improvement in the objective function value. This procedure is repeated till it fails to improve the objective function value. The 3SW procedure is analogous but it considers three consecutive jobs at a time. The best among the six possible permutations of these three consecutive jobs is selected to be the new ordering of these consecutive jobs in the schedule. This procedure is also repeated till no further improvement is possible. The complexity of each iteration of API as well as 3SW is $O(n)$. Another improvement procedure that is of concern to us is the non-adjacent pairwise interchange (NAPI) procedure. The NAPI procedure is also similar to API. The only difference is that it tries to interchange a job with all other jobs in the schedule and the interchange that leads to maximum reduction in the value of the objective function is performed. The complexity of each iteration of NAPI is $O(n^2)$. Therefore, NAPI is computationally more expensive than API and 3SW.

Among the metaheuristic techniques, Valente [15] presented several beam search heuristic procedures. The schedules obtained through each of the beam search procedure are improved further by the 3SW procedure. Among several beam search heuristic procedures proposed in this paper the recovering beam search (RBS) procedure produced the best results. Valente et al. [16] proposed six random-key encoding [4] based elitist generational genetic algorithms. These genetic algorithms differ only slightly from one another. All of them use parameterized uniform crossover. For crossover one parent is selected uniformly at random from the elite members of the population, whereas another parent is selected uniformly at random from the entire population. Mutation is not used at all, and therefore, to avoid premature convergence and to maintain diversity in the population, these genetic algorithms use a migration mechanism. During each generation depending on the genetic algorithm version, migration mechanism generates some members either randomly or using RCL_VB procedure of Valente [15]. Except for two versions of the genetic algorithm, where entire initial population is generated randomly, some members of initial population are generated using four dispatch procedures proposed in Valente and Alves [14] and RCL_VB. Three of these six genetic algorithms use API procedure as local search. Among these six genetic algorithms, MA_IN and MA_GR performed better than the others.

In this paper, we have proposed three genetic algorithm based hybrid heuristics for the QETSP. Our heuristics are combinations of a steady-state genetic algorithm and API, 3SW and NAPI improvement procedures. As described in Sect. 2, the genetic algorithm used here is altogether different from those used in Valente et al. [16], i.e., solution encoding, population replacement strategy, genetic operators viz. selection, mutation and crossover are all different. Moreover, no migration mechanism is used. We have compared our hybrid heuristics with RBS, MA_IN and MA_GR on the same benchmark instances as used in Valente [15] and Valente et al. [16]. These hybrid heuristics not only obtained better quality solutions, but also are faster.

The remainder of this paper is organized as follows: Sect. 2 describes our hybrid heuristics. Section 3 reports the computational results of our heuristics and compares them with RBS, MA_IN and MA_GR. Finally, some concluding remarks are presented in Sect. 4.

2 Hybrid heuristics for the QETSP

We have developed three hybrid heuristics for the QETSP combining a genetic algorithm with API, 3SW and NAPI improvement procedures. We first describe our genetic algorithm, then the manner in which this genetic algorithm is combined with the improvement procedures, and finally, the three heuristics.

2.1 The genetic algorithm

Our genetic algorithm uses the steady-state population replacement model [7] instead of the commonly used generational model. Unlike generational replacement, where the entire parent population is replaced with an equal number of newly created children every generation, in the steady-state population replacement method a single child is produced in every generation and it replaces a less fit member of the population. In comparison to the generational method, the steady-state population replacement method generally finds better solutions faster. This is because of permanently keeping the best solutions in the population and the immediate availability of a child for selection and reproduction. Another advantage of the steady-state population replacement method is the ease with which duplicate copies of the same individuals are prevented in the population. In the generational approach, duplicate copies of the highly fit individuals may exist in the population. Within few generations, these highly fit individuals can dominate the whole population. When this happens, crossover becomes totally ineffective and the mutation becomes the only possible way to improve solution quality. Under this situation improvement, if any, in solution quality is very slow. Such a situation is called premature convergence. In the steady-state approach we can easily avoid this situation by simply checking each newly generated child against current population members and discarding it if it is identical to any member.

Our genetic algorithm uses permutation encoding to represent a chromosome, i.e., a linear permutation of jobs represents a chromosome. Using permutation encoding for QETSP has several advantages. First, only feasible solutions are represented by this encoding, and, hence, no repair mechanism is required to repair the infeasible solution generated after the application of genetic operators. Second, there is a one-to-one correspondence between a schedule and the corresponding permutation representation, and, hence, this encoding does not suffer from the problem of redundant solutions. The presence of redundancy in an encoding can make genotype space (the space of all possible chromosomes) to be much larger than the phenotype space (the space of all possible solutions). As a result, a genetic algorithm, which works in genotype space, is forced to unnecessarily search a bigger space. This can adversely affect the performance of the genetic algorithm. Third, unlike some other encodings such as random-keys [4], no decoder is required to convert a chromosome into its equivalent schedule. We have taken the fitness of a chromosome to be equal to the cost of the schedule represented by it. The less the cost of a schedule represented by a chromosome, the more fit it is.

We have used uniform order based (UOB) crossover [7]. UOB crossover copies, with probability p_u, job at each position from one of the two parents to the corresponding position in the child. The remaining positions in the child are filled from the unused jobs in the order in which these jobs appear in the second parent. Following the philosophy of fitness based crossover proposed by Beasley and Chu [5], we have set p_u to be equal to $f(p_2)/(f(p_1) + f(p_2))$, where $f(p_1)$ and $f(p_2)$ are fitness of first and second parents respectively. Figure 1 explains the UOB crossover with the help of an example. The first parent has fitness 10 and the second parent has fitness 15. Therefore, the job at each position of the first parent will be copied to the corresponding position in the child with probability 0.6. If for each position a random number is generated as per the figure then the jobs at first, fourth, fifth and seventh position will be copied to the corresponding positions in child. Therefore, the partially formed child will be 4 _ _ 1 5 _ 8 _, where '_' indicates an empty position. These empty positions will be filled by so far unused jobs in the order in which they occur in the second parent. Therefore, the complete child will be 4 6 2 1 5 7 8 3.

We have tried two different mutations—swap mutation and insert mutation. Multiple swap or insert mutations are used to mutate a chromosome. However, a job is swapped only with nearby jobs. Likewise a job is inserted only in the vicinity of its original position.

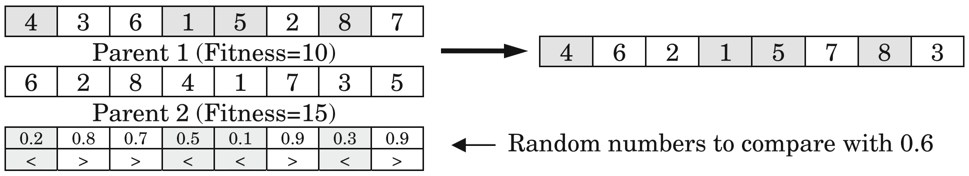

Fig. 1 Uniform order based crossover

Crossover and mutation are used in a mutually exclusive way, i.e., a child is generated either by crossover or by mutation but never by both. The probability with which crossover is used needs to be determined empirically (mutation probability will be simply 1—crossover probability). Parents are selected for crossover using probabilistic binary tournament selection, whereas for mutation a single parent is randomly selected.

We have generated the initial population randomly. However, as our genetic algorithm is steady-state, we have therefore guaranteed the uniqueness of each member of the initial population. Moreover, during each generation the newly produced child is compared with current population members, and it is discarded if it is identical to any member. If the newly created child is unique with respect to the current population, then it is included in the current population replacing the worst member.

2.2 API/3SW/NAPI improvement procedures

As API and 3SW are computationally less expensive than NAPI, therefore, these improvement procedures are used inside the genetic algorithm to improve the solution obtained after the application of genetic operators during each iteration, whereas, NAPI is used to improve further the best solution returned by the genetic algorithm. As API and 3SW are used during each iteration of the genetic algorithm, therefore, instead of iterating them till no improvement is possible in solution quality, we have iterated them for a fixed number of times. NAPI is repeatedly applied until it is not possible to improve the schedule further.

2.3 Three hybrid heuristics

Different combinations of genetic algorithm and improvement procedures give the three hybrid heuristics. NAPI is invariably used in all the three hybrid heuristics to improve the best solution returned by the genetic algorithm. Our first heuristic uses genetic algorithm with swap mutation along with the API procedure in the aforementioned manner. We shall call this heuristic GASM-API. In our second heuristic, the genetic algorithm is the same, but 3SW is used in place of API. We shall call this heuristic GASM-3SW. In our third heuristic (GAIM-3SW), we have used genetic algorithm with insert mutation along with 3SW.

```
generate initial population;
current_best ← best solution of the initial population;
while (termination criterion remains unsatisfied){
    if (u01 < fixed value) {
        select two parents $p_1$ and $p_2$ using binary tournament selection;
        C ← crossover($p_1$,$p_2$);
    }
    else {
        select a parent $p_1$ randomly;
        C ← mutate($p_1$);
    }
    C←IP(C);
    if (unique(C)) {
        include C in the population replacing the worst member;
        if f(C) < f(current_best)
            current_best ← C;
    }
}
current_best ← NAPI(current_best);
return current_best;
```

Fig. 2 Pseudo-code of the hybrid approach

The pseudo-code of our hybrid approach is given in Fig. 2, where u01 is a uniform variate in [0, 1], unique is a function that returns 1 if the newly generated solution C is unique with respect to existing population members, otherwise it returns 0. IP is the improvement procedure consisting of multiple passes of either 3SW or API as described in Sect. 2.2.

3 Computational results

The three hybrid heuristics viz. GASM-API, GASM-3SW and GAIM-3SW have been coded in C and executed on a Linux based 3.0 GHz Pentium 4 system with 512 MB RAM. We have used a population of 400 chromosomes and have selected two parents for crossover with different probability of the better being the winner in binary tournament selection. The first parent is selected from the better of the two randomly chosen chromosomes with probability 0.9, whereas for other parent this probability is 0.8. Crossover is used to generate a new child with probability 0.85, otherwise mutation is used. Insert mutation or swap mutation is used $\lfloor n/10 \rfloor$ times to mutate a schedule. In case of swap mutation, a job is swapped only with the second or third or fourth neighbor in either direction, whereas in case of insert mutation, a job is inserted only after third or fourth or fifth neighbor in either direction. API or 3SW is applied $\lceil n/4 \rceil$ times at the most (may be less if one complete iteration of them fails to improve the schedule) during each iteration of the genetic algorithm.

All these parameter values are determined empirically after large number of trials. These parameter values provide good results, though they may not be optimal on all instances. We have executed our heuristics once on each instance. During each execution the genetic algorithm terminates when either the best solution does not improve over $100n$ iterations or it has run for a total of $1{,}000n$ iterations.

We have compared GASM-API, GASM-3SW and GAIM-3SW with RBS [15] and MA_IN and MA_GR [16] on the same QETSP test instances as used in Valente [15] and Valente et al. [16]. These instances consist of 10, 15, 20, 25, 50, 75, 100, 250 and 500 jobs. Due to high computational times, all the genetic algorithms including MA_IN and MA_GR proposed in Valente et al. [16] could not be executed on instances of size 250 and 500. All these instances were generated randomly by slightly modifying the instance generation procedure used in Abdul-Razaq and Potts [1], Li [11] and Liaw [12]. For each job J_i, an integer processing time p_i, an integer earliness penalty h_i and an integer tardiness penalty w_i are generated uniformly at random from one of the two intervals [45, 55] and [1, 100], to create low (L) and high variability (H), respectively. The integer due date for each job J_i is selected uniformly at random from the interval $[P(1 - T - RDD/2), P(1 - T + RDD/2)]$, where P is the sum of processing times of all the jobs, T is the tardiness factor and RDD is the range of due dates. The T was set to 0.0, 0.2, 0.4, 0.6, 0.8, 1.0 and RDD was set to 0.2, 0.4, 0.6, 0.8. This leads to 24 different combinations of T and RDD values. For each problem size, for each of the two variabilities (L or H) and for each of these 24 combinations, 50 instances were generated uniformly at random. Therefore, there are 1,200 instances for each combination of problem size and variability.

3.1 Comparison of solution quality on small instances

Table 1 compares GASM-API, GASM-3SW and GAIM-3SW with RBS, MA_IN and MA_GR on small instances of size 10, 15 and 20 in terms of average relative deviation from optimum expressed in percentage (%dev) and the number of times an optimal solution was found expressed in percentage (%opt). The optimum solutions for these instances were determined through the branch-and-bound method proposed in Valente [13]. For any instance the relative deviation from optimum in percentage is calculated as $100 \times (HSol - Opt)/Opt$, where $Hsol$ is the solution obtained through heuristic and Opt is the optimum solution. Data for RBS are taken from Valente [15] and data for MA_IN and MA_GE are taken from Valente et al. [16].

Table 1 clearly shows the superiority of our three heuristics over RBS, MA_IN and MA_GR. GASM-API, GASM-3SW and GAIM-3SW were able to find an optimum solution in every run.

3.2 Comparison of solution quality with RBS on remaining instances

Table 2 compares GASM-API, GASM-3SW and GAIM-3SW with RBS on instances of size 25, 50, 75, 100, 250 and 500 in terms of average relative improvement over RBS expressed in percentage (%imp) and number of times the solutions obtained by our heuristics are better (B) or worse (W) than or equal (E) to RBS. Data for RBS are taken from Valente [15]. Table 2 clearly shows the superiority of our heuristics over RBS. GASM-3SW always finds a solution as good as or better than RBS, whereas GASM-API does the same on all but five instances and GAIM-3SW on all but three instances. As far as

Table 1 Comparison of different heuristics on small instances for which optimum solutions are known

Var	Heuristic	$n = 10$		$n = 15$		$n = 20$	
		%dev	%opt	%dev	%opt	%dev	%opt
L	RBS	0.00	99.92	0.00	100.00	0.00	99.67
	MA_IN	0.00	100.00	0.00	100.00	0.00	100.00
	MA_GR	0.00	100.00	0.00	100.00	0.00	100.00
	GASM-API	0.00	100.00	0.00	100.00	0.00	100.00
	GASM-3SW	0.00	100.00	0.00	100.00	0.00	100.00
	GAIM-3SW	0.00	100.00	0.00	100.00	0.00	100.00
H	RBS	0.22	95.67	0.91	88.00	1.40	82.50
	MA_IN	0.00	99.99	0.00	98.90	0.00	96.63
	MA_GR	0.00	99.99	0.00	98.91	0.00	96.82
	GASM-API	0.00	100.00	0.00	100.00	0.00	100.00
	GASM-3SW	0.00	100.00	0.00	100.00	0.00	100.00
	GAIM-3SW	0.00	100.00	0.00	100.00	0.00	100.00

Table 2 Comparison of GASM-API, GASM-3SW and GAIM-3SW with RBS

Var	n	GASM-API				GASM-3SW				GAIM-3SW			
		%imp	B	E	W	%imp	B	E	W	%imp	B	E	W
L	25	0.000028	8	1,192	0	0.000028	8	1,192	0	0.000028	8	1,192	0
	50	0.000156	35	1,165	0	0.000156	35	1,165	0	0.000156	35	1,165	0
	75	0.000062	85	1,115	0	0.000062	85	1,115	0	0.000062	85	1,115	0
	100	0.000053	126	1,074	0	0.000053	126	1,074	0	0.000053	126	1,074	0
	250	0.000034	335	865	0	0.000034	335	865	0	0.000034	335	865	0
	500	0.000011	539	659	2	0.000011	540	660	0	0.000011	538	659	3
H	25	1.367578	284	916	0	1.367578	284	916	0	1.367578	284	916	0
	50	2.234419	538	661	1	2.234581	538	662	0	2.234577	538	662	0
	75	2.433543	700	499	1	2.433656	700	500	0	2.433678	699	501	0
	100	2.782419	793	407	0	2.782871	795	405	0	2.782919	795	405	0
	250	2.891773	1,029	170	1	2.892286	1,031	169	0	2.892316	1,032	168	0
	500	3.002776	1,140	60	0	3.003949	1,141	59	0	3.004072	1,141	59	0

Table 3 Relative Improvement of MA_IN and MA_GR over RBS

Var	n	MA_IN		MA_GR	
		%imp (best case)	%imp (average)	%imp (best case)	%imp (average)
L	25	0.000028	0.000027	0.000028	0.000025
	50	0.000156	0.000156	0.000156	0.000156
	75	0.000062	0.000062	0.000062	0.000062
	100	0.000053	0.000052	0.000053	0.000052
H	25	1.367464	1.298696	1.365337	1.283716
	50	2.233715	2.201221	2.234009	2.180008
	75	2.433641	2.411501	2.432683	2.396436
	100	2.782789	2.768149	2.782185	2.755444

comparison among our three heuristics is concerned, they perform more or less the same. GASM-3SW performed equally well on low as well as high variability instances. GAIM-3SW performed better than others on high variability instances in terms of relative improvement. Overall, GASM-API performed slightly worse than the other two.

3.3 Comparison of solution quality with MA_IN and MA_GR on remaining instances

MA_IN and MA_GR were executed 10 times on each instance. Therefore, here average relative improvement was defined in two ways—average relative improvement over RBS with respect to best solution found during 10 trials and average relative improvement over RBS with respect to average solution quality of 10 trials. Table 3 reports these values for MA_IN and MA_GR. Data for MA_IN and MA_GR are taken from Valente et al. [16]. Comparing Tables 2 and 3, we can see that GASM-3SW and GAIM-3SW performed as well as or better than the MA_IN and MA_GR on instances of all sizes. Even the relative improvements of GASM-3SW and GAIM-3SW are better than the best case relative improvement of MA_IN and MA_GR. This is significant as GASM-3SW and GAIM-3SW are executed only once on each instance. Had these heuristics been executed ten times and the average relative improvement in the best case of these heuristics would have been compared with MA_IN and MA_GR, the results of these heuristics would have been much better.

3.4 Comparison of running time of different heuristics

Table 4 shows the average execution times of different heuristics. Data for RBS are taken from Valente [15], whereas data for MA_IN and MA_GR are taken from Valente et al. [16]. RBS, MA_IN and MA_GR were executed on a 2.8 GHz Pentium 4 system, whereas our heuristics are executed on a 3.0 GHz Pentium 4 system. Therefore, it is not possible to exactly compare the execution times of different heuristics. Except for $n = 25$, our three heuristics require less time than MA_IN and MA_GR. RBS is the fastest among all up to $n = 100$, whereas on instances of size 500 and 250, GASM-API is

Table 4 Average execution time in seconds for different heuristics

Var	n	RBS	MA_IN	MA_GR	GASM-API	GASM-3SW	GAIM-3SW
L	25	0.01	0.03	0.02	0.03	0.04	0.04
	50	0.04	0.21	0.15	0.09	0.14	0.14
	75	0.10	0.68	0.48	0.18	0.29	0.30
	100	0.22	1.64	1.14	0.29	0.50	0.52
	250	3.24	–	–	1.65	3.05	3.14
	500	25.87	–	–	6.86	12.62	12.91
H	25	0.01	0.03	0.03	0.04	0.05	0.05
	50	0.04	0.24	0.18	0.12	0.16	0.17
	75	0.11	0.86	0.63	0.25	0.35	0.36
	100	0.24	2.26	1.52	0.41	0.60	0.62
	250	3.38	–	–	2.35	3.81	3.88
	500	27.55	–	–	9.77	16.06	16.36

the fastest. On instances of size 500, RBS require more time than any of our heuristics. Among our three heuristics, GASM-API is the fastest followed by GASM-3SW.

4 Conclusions

In this paper we have presented three genetic algorithm based hybrid heuristics for the single machine scheduling problem with quadratic earliness and tardiness costs. We have compared our heuristics against two genetic algorithms and a recovering beam search procedure. Computational results show the effectiveness of our heuristics. Our heuristics not only obtained better quality solutions, but are also faster.

As a future work, we intend to investigate the performance of other population based metaheuristic approaches on this problem.

References

1. Abdul-Razaq TS, Potts CN (1988) Dynamic programming state-space relaxation for single-machine scheduling. J Oper Res Soc 39:141–142
2. Bagchi U, Chang Y, Sullivan R (1987) Minimizing absolute and squared deviations of completion times with different earliness and tardiness penalties and a common due date. Nav Res Logist 34:739–751
3. Baker KR, Scudder GD (1990) Sequencing with earliness and tardiness penalties: a review. Oper Res 38:22–36
4. Bean JC (1994) Genetic algorithms and random keys for sequencing and optimization. ORSA J Comput 6:154–160
5. Beasley JE, Chu PC (1996) A genetic algorithm for the set covering problem. Eur J Oper Res 94:394–404
6. Cheng TCE (1984) Optimal due date determination and sequencing of n jobs on a single machine. J Oper Res Soc 35:433–437
7. Davis L (1991) Handbook of genetic algorithms. Van Nostrand Reinhold, New York
8. Gupta SK, Sen T (1983) Minimizing a quadratic function of job lateness on a single machine. Eng Costs Prod Econ 7:181–194
9. Korman K (1994) A pressing matter. Video, pp 46–50
10. Landis K (1993) Group technology and cellular manufacturing in the Westvaco Los Angeles VH Department. Project Report in IOM 581, School of Business, University of Southern California
11. Li G (1997) Single machine earliness and tardiness scheduling. Eur J Oper Res 96:546–558
12. Liaw CF (1999) A branch-and-bound algorithm for the single machine earliness and tardiness scheduling problem. Comput Oper Res 26:679–693
13. Valente JMS (2007) An exact approach for single machine scheduling problem with quadratic earliness and tardiness penalties. Working Paper 238, Faculdade de Economia, Universidade do Porto, Portugal
14. Valente JMS, Alves RAFS (2008) Heuristics for the single machine scheduling problem with quadratic earliness and tardiness penalties. Comput Oper Res 35:3696–3713
15. Valente JMS (2010) Beam search heuristics for quadratic earliness and tardiness scheduling. J Oper Res Soc 61:620–631
16. Valente JMS, Moreira MRA, Singh A, Alves RAFS (2011) Genetic algorithms for single machine scheduling with quadratic earliness and tardiness costs. Int J Adv Manuf Technol 54:251–265
17. Wagner BJ, Davis DJ, Kher H (2002) The production of several items in a single facility with linearly changing demand rates. Decis Sci 33:317–346