A Population-Based ACO for solving Single Machine Total Weighted Tardiness Problem

A. Jafari[1], S. Maleki[2], H. Farvaresh[3], and S. E. Tabari[4]

[1] Assistant Professor of Industrial Engineering Department of
Science and Culture University, Tehran, Iran, *E-mail*: jafari@usc.ac.ir
[2] MSc of Industrial Engineering Department of
Science and Culture University, Tehran, *E-mail*: maleki@usc.ac.ir
[3] PhD Candidate of Industrial Engineering of Tarbiat Modares University, Tehran, *E-mail*: farvaresh@modares.ac.ir
[4] MSc of Computer Sciences, University of Tehran, *E-mail*: Ehsan@tabari.info

Abstract
Machine scheduling is a central task in operations and production planning. The main focus of machine scheduling is on the efficient allocation of resources to activities over time. The weighted tardiness as the objective with the sequence-dependent setups as constraint is one of the most common machine scheduling problem characteristics. Modern heuristic as solving techniques for machine scheduling problems, are a family of procedures which benefit some sort of intelligence in their search for finding the solution of a problem. Studies shows that ACO, a heuristic technique, can solve single machine total weighted tardiness problem satisfactorily and the solution quality can compete with that of other leading heuristics. In this paper we propose a new population based ACO for solving single machine total weighed tardiness problem with sequence-dependent setup times. This new ACO has a different manner in updating pheromone matrix. The proposed ACO is experimented on the benchmark problem instances, compared with the ACO presented in Liao and Juan and shows its advantage in most of the complex instances.
Keywords: Scheduling; Ant colony optimization; Weighted tardiness; Sequence-dependent setups

1-Introduction

Machine scheduling is a central task in operations and production planning. The main focus is on the efficient allocation of resources to activities over time. The first systematic approach to scheduling problems was undertaken in the mid-1950s (Sena et al., 2003). Since then, thousands of papers on different scheduling problems have appeared in the literature. One of the most researched problems is the single-machine total weighted tardiness (SMTWT) problems. In 1990, an excellent survey of research on the SMTWT problem was provided by Abdul-Razaq et al. (Abdul-Razaq et al., 1990). Other noteworthy papers include Elmaghraby (Elmaghraby, 1968), Picard and Queyranne (Picard and Queyranne, 1978), and Schrage and Baker (Schrage and Baker, 1978).

It is well known that the SMTWT problem is NP-hard (Lenstra and Kan, 1980). Like most research on scheduling problems, it is assumed that setup times are independent of the sequence of tasks on a machine. It is assumed that setup times are negligible or are added to the processing times of the tasks. However, significant setup times are incurred in some situations whenever a machine switches service from one task to another. In these cases, the machine processes many different jobs, and the setup time for a job depends on the job that has just finished processing before it. Some noteworthy examples are found in petroleum producing plants, car spraying facilities, textile dying plants and pharmaceutical industries, and are described in Allahverdi et al. (Allahverdi et al., 1999).

In the single-machine environment, the solving techniques can be broadly classified into two main categories: Heuristic methods and exact optimizing techniques. In the field of exact techniques, most of the optimization techniques used branch-and-bound or dynamic programming as the basic searching strategy. One of the papers that primarily dealt with the SMTWT problem is by Potts and Van Wassenhove who presented a branch-and-bound algorithm for the problem (Potts and Wassenhove., 1991). A survey on theoretical results on ant colony optimization is provided by Dorigo and Blum (Dorigo and Blum, 2005). They review some convergence results and discuss relations between ant colony optimization algorithms and other approximate methods for optimization. There are varieties of heuristics methods reported in the literature. One of these is the shortest processing time (SPT) rule. Another heuristic is the well known earliest due date (EDD) rule which schedules all the jobs in the non-decreasing order of d_j. This rule is optimal, if the EDD sequence produces no more than one tardy job (K.R.Baker and Martin, 1974). More recent heuristics for the SMTWT problem were obtained by Crauwels et al. (Crauwels et al., 1998) who compared such local search techniques as tabu search, simulated annealing, steepest descent, threshold search and genetic algorithms by utilizing both permutation and binary representations of actual sequences for a large set of test problems. Recently, algorithms tends to falling into the framework of the Ant Colony Optimization (ACO). Besten et al. (Besten et al., 2000) apply ACO to solve SMTWT problem. Liang (Liang, 2001) applies an ant colony system (ACS) to a set of test problems of size up to 100 jobs from the literature with excellent results. The study presented shows that ACO can solve SMTTPs satisfactorily and the solution quality can compete with that of other leading heuristics.

This paper proposes a new Ant Colony Optimization (ACO) approach to face the single machine total weighted tardiness problem. The proposed ACO algorithm is a novel population-based ACO alternative, but it has a different manner in updating pheromone matrix. Similar population-based ACO algorithms (P-ACO) were designed by Guntsch and Middendorf (Guntsch and Middendorf, 2002b, Guntsch and Middendorf, 2002a) for dynamic combinatorial optimization problems. There are two differences between our ACO and P-ACO. The first difference is that our ACO does not allow identical individuals in its population, and the second difference is that our ACO updates τ every K iteration, while P-ACO updates K every iteration.

The rest of the paper is organized as follows: First, we introduce problem definition in Section 2. Next, the proposed ACO algorithm is presented and discussed in Section 3. This is followed by the presentation of how ACO solves SMTWT problem. We then discuss the extended experimental campaign performed on the benchmark set generated by Cicirello (Cicirello, 2003b), whose best known results have been very recently updated in Liao and Juan (Liao and Juan, 2007) in Section 4. The comparisons between the proposed ACO with the one presented in Liao and Juan and Cicirello (Cicirello, 2003b) also will presented in Section 4. Finally, Section 5 concludes the paper.

2- Problem Statement

In the area of scheduling, two of the most researched problems are the single-machine total tardiness (SMTT) and total weighted tardiness (SMTWT) problems. For defining SMTWT problems, we will borrow the assumptions and terminology used by Conway et al. (Conway et al., 1967). In the single-machine environment, we have n jobs, $1,2,\ldots,\mathrm{n}$, all ready at time 0, to be processed on a single machine which is never idle. No preemption of jobs is allowed. Associated to each job j are a processing time $\mathrm{p_j}$, a due date $\mathrm{d_j}$ and a weight $\mathrm{w_j}$. The jobs are all available for processing right from the start. The tardiness of job j is $\mathrm{T_j} = \max\{0, \mathrm{c_j} - \mathrm{d_j}\}$ where $\mathrm{c_j}$ is the completion time of job j in the current job sequence; The SMTWT problem requires minimizing the total tardiness T such that $\mathrm{T} = \sum_{\mathrm{j}\in\mathrm{J}} \mathrm{w_j}\,\mathrm{T_j} = \sum_{\mathrm{j}\notin\mathrm{J}} \mathrm{w_j}\ \max(\mathrm{c_j} - \mathrm{d_j}, 0)$.

The goal of the SMTWTP is to find a job sequence which minimizes the sum of the weighted tardiness. The problem is complicated by the fact that it takes variable amounts of time to reconfigure (or setup) the machine when switching between any two jobs. The completion time $\mathrm{c_j}$ of a job can be defined formally as $\mathrm{c_j} = \sum_{\mathrm{i}\in\mathrm{Predecessors(j)}\cup\mathrm{j}} \mathrm{p_j} + \mathrm{s_{Presious(i),i}}$ where $\mathrm{p_i}$, $\mathrm{s_{k,j}}$, are the process time of job i and the setup time of job i if it immediately follows job k, respectively. $\mathrm{Predecessors(j)}$ is the set of all jobs that come before job j in the sequence and $\mathrm{Presious(i)}$ is the single job that immediately precedes job i.

This problem, which is denoted as $1//\sum w_j T_j$, has been proved to be strongly NP-hard by Lawler (Lawler, 1979); the complexity of the considered problem, confirmed by the fact that the special case without setups and with unitary weights is still NP-hard (Du and Leung, 1990), justifies the research of heuristic approaches for its solution in practical cases. Nevertheless, exact algorithms based on branch and bound (B&B) or dynamic programming approaches have been proposed, but they are able to tackle instances of small scale.

3- Ant Colony Optimization algorithm

Modern heuristic techniques, also called metaheuristics are a family of procedures which benefit some sort of intelligence in their search for finding the solution of a problem. Ant Colony Optimization (ACO) is one of these metaheuristics that dates back to the early nineties and was first introduced by Dorigo et al (Dorigo et al., 1991) as a novel nature-inspired metaheuristic for the solution of hard combinatorial optimization problems. That is, for problems in which the best known algorithms that guarantee to identify an optimal solution, have exponential time worst case complexity. Many of these problems can be classified into one of the following categories: *routing problems, assignment problems*, *scheduling problems* & *subset problems*. In addition, ACO has been successfully applied to other problems emerging in fields such as machine learning and bioinformatics. Other popular applications are to dynamic shortest path problems arising in telecommunication networks problems. This wide range of successful applications has motivated Researchers to adopt ACO for the solution of industrial problems (Gravel et al., 2002).

Ant Colony Optimization models a nature-based, multi-agent process. The foraging behavior of real ants is the inspiring source of ACO. Real ants searching for food are capable to find the shortest path between their nest and a food source without strength of vision but by exchanging information via pheromones. Varying quantities of pheromone which are laid down by each ant on the path taken indicate the distance and quality of the food source and will guide other ants to the food source.

Therefore Indirect communication between the ants via pheromone trails enables them to find shortest paths between their nest and food sources.

The ACO algorithm is an adaptation of Ant Colony System (ACS) for the SMTWT problem (Dorigo and Gambardella, 1997). A feasible solution for the SMTWT problem, also called a *sequence*, consists of a permutation of the jobs. When applied to the SMTWT problem, each ant starts with an empty sequence and then iteratively appends an unscheduled job to the partial sequence constructed so far. The decisions an ant makes are probabilistic in nature and influenced by two factors: the pheromone information, which is gained from the choices made by previous good ants, and heuristic information, which indicates the immediate benefit of making the corresponding choice. Depending on the type of problem being processed, the pheromone and heuristic information have different interpretations. Local search is applied to all solutions constructed by ants in each iteration.

In the following, we will first describe the representation graph then the process of solution generation, our strategies to update pheromone trails and finally a generalized flow chart of the ant algorithm. To apply Ant Colony Optimization a graphical representation of the problem is needed. Figure 1 consists of some nodes representing a set of jobs, and some arcs representing adding a specific job to the sequence.

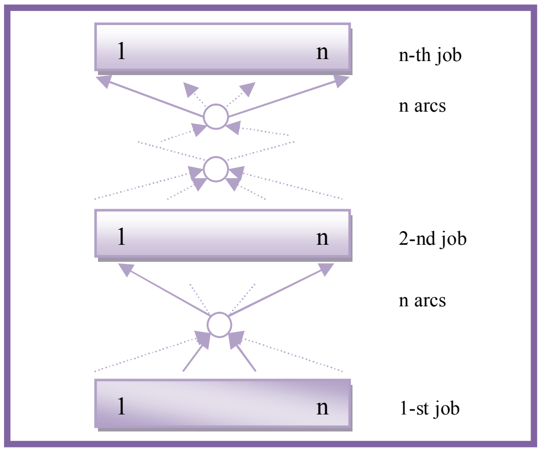


Figure 1: Graphical representation of the single machine problem

Ants move from one node to another. A move represents the decision to schedule job j as $i-th$ job in the sequence, no matter which jobs have been scheduled previously. Any path from the source node to the sink node represents a feasible solution; the optimal sequence of jobs is the path that minimizes the total tardiness of all jobs. For a problem with n jobs the number of arcs is $\sum_{i=0}^{n-1}\left[\binom{n}{i}(n-1)\right] = 2^n.\frac{n}{2}$ and the number of nodes is $\sum_{i=0}^{n-1}\binom{n}{i} = 2^n$ (Bauer et al., 1999).

Each artificial ant iteratively and independently decides which job to append to the sub-sequence generated so far until all jobs are scheduled. Each ant generates a complete solution by selecting a job j to be on the $i-th$ position of the sequence. This selection process is influenced through problem specific heuristic information, called visibility and denoted by η_{ij}, as well as pheromone trails, denoted by τ_{ij} . The former is an indicator of how good the choice of that job seems to be, and the latter indicates how good the choice of that job was in former runs. Both matrices are only two dimensional .The transition probability p_{ij}that job j be selected to be processed on position i in the sequence is formally given by :

$$p_{ij} = \begin{cases} \frac{[\tau_{ij}]^{\alpha}\,[\eta_{ij}]^{\beta}}{\sum_{h\in\Omega}[\tau_{ih}]^{\alpha}\,[\eta_{ih}]^{\beta}} & if \quad j \in \Omega \\ 0 & otherwise \end{cases}$$

Where Ω is the set of non-scheduled jobs, α determines the relative influence of the pheromone trails and β regulates the relative influence of the visibility. The most obvious heuristic information to be used is based on the Earliest Due Date heuristic, which sorts and schedules jobs according to ascending due dates. So the visibility would read as follows:

$$\eta_{ij} = \frac{1}{d_j}$$

After each ant has performed the selection process according to the procedure described above and each ant has generated a feasible sequence, the pheromone trails are globally updated. Good solutions are characterized by relatively low total tardiness and emphasize those sub-sequences which are part of a good solution; whereas those sub-sequences that are part of poor solutions will be only slightly marked. At the same time, evaporation reduces the pheromone trails evenly. The global trail update may formally be written as follows:

$$\tau_{ij}(t+1) = (1-\rho).\tau_{ij}(t) + \rho.\Delta\tau_{ij}(t)$$

Where the parameter $\rho \in (0,1]$ controls the pheromone decay, and $\Delta\tau_{ij}(t)$ is the amount of pheromone trail added to τ_{ij} by the ants. In this paper only the best solution contributes to the pheromone trail update. Thus, we have $\Delta\tau_{ij}(t) = \frac{1}{T^*}$ for all edges $(i\,,j)$ belonging to the best solution found so far, where T^* is the total tardiness of that best solution. Besides the global update described above, we also make a local trail update. After an artificial ant has selected a job to be appended to the existing sub-sequence, the corresponding pheromone trail is updated as follows:

$$\tau_{ij}(t+1) = (1-\rho).\tau_{ij}(t) + \rho.\ \tau_0$$

with the initial trail intensity:

$$\tau_0 = \frac{1}{n.T_{EDD}}$$

Where T_{EDD} is the total tardiness for a processing sequence generated according to the Earliest Due Date rule. ACO uses a pheromone matrix $\tau = \{\tau_{ij}\}$ for the construction of potential good solutions. At every generation of the algorithm, each ant of a colony constructs a complete tour, starting at a randomly chosen city. Pheromone evaporation is applied for all (i, j) according to:

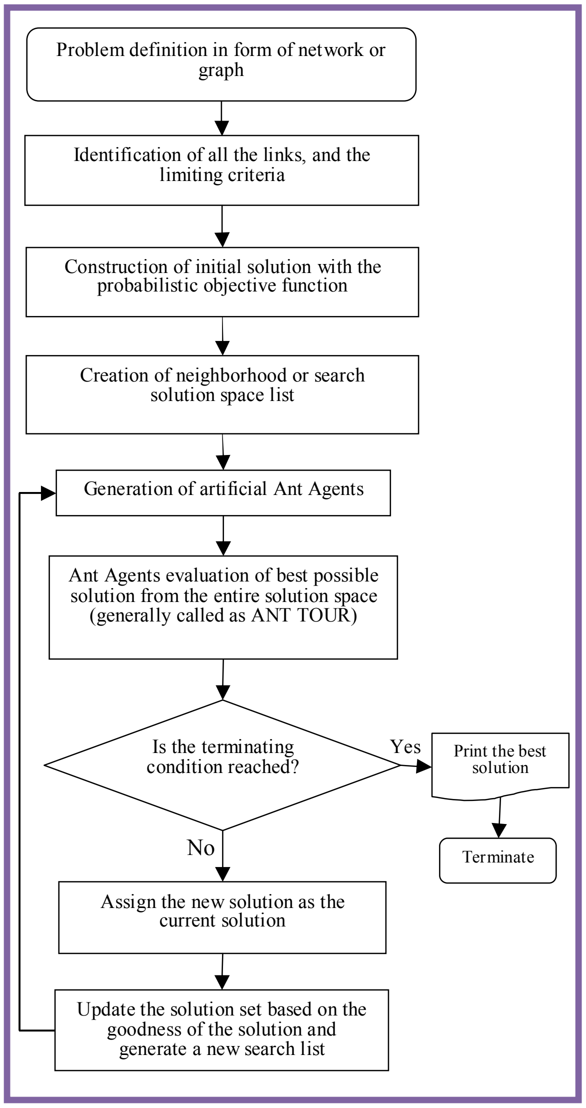
$\tau_{ij} = (1-\rho).\tau_{ij}$

Figure 2 represents a generalized flow chart of the ant algorithm.

In our ACO, a constant pheromone matrix τ with $\tau_{ij} = 1, \forall i, j$ is defined. Our ACO maintains a population $P = \{P_x\}$ of m individuals or solutions, the best unique ones found so far. The best individual of P at any moment is called P^*, while the worst individual P_{worst}. In our ACO the first population is chosen using $\tau_{ij} = 1$. At every iteration a new P_{new} individual is generated, replacing $P_{worst} \in P$ if P_{new} is better than P_{worst} and different from any other $P = \{P_x\}$. After K iterations, τ is recalculated. First, $\tau_{ij} = 1$; then, $\frac{O}{m}$ is added to each element τ_{ij} for each time an $arc(i, j)$ appears in any of the m individuals present in P .The above process is repeated every K iterations until an end condition is reached. Note that $1 \leq \tau_{ij} \leq (1+O)$, where $\tau_{ij} = 1$ if $arc(i, j)$ is not present in any P_x, while $\tau_{ij} = (1+O)$ if $arc(i, j)$ is in every $P_x \in P$.

4- Experimental analysis of the proposed ACO approach

In order to analyze proposed ACO performances, algorithm was coded in C# and a systematic grid search was conducted on a Pentium IV with a 3 GHz Core Due and 1 GB of RAM. The adopted benchmark was the set of 80 of problem instances provided by Cicirello (Cicirello, 2003a). It should be mentioned that the same benchmark was used to test the ACO_{LJ} in Liao and Juan (Liao and Juan, 2007). The benchmark was produced by generating 10 instances for each combination of three different factors. Each problem instance is characterized by three parameters: the due-date tightness factor δ ; the due-date range factor R; and the setup time severity factor ξ. These parameters are defined as follows: $\tau = 1 - \frac{\bar{d}}{\tilde{C}_{max}}$, $R = \frac{d_{max} - d_{min}}{\tilde{C}_{max}}$, and $\xi = \frac{\bar{s}}{\bar{p}}$, where $\bar{d}$, $\bar{p}$ and $\bar{s}$ are the average duedate, average process time, and average setup time, d_{max}, d_{min} are the maximum and minimum duedates, and $\tilde{C}_{max}$ is an estimation of the makespan C_{max} (or completion time of the last job). As computing the actual makespan for the instance is NP-Hard, thus the estimator suggested by Lee et al (Lee et al., 1997) is used: $\tilde{C}_{max} = n(\bar{p} + \beta\bar{s})$ where n is the number of jobs in the problem instance. This estimator for the makespan amounts to a sum of the process times of the jobs which is the same regardless of sequence, and an estimate of the sum of the setup times (Cicirello, 2007). Cicirello provides experimental data for setting the value of β for 4 different size problems (20, 40, 60, and 80 job instances). In the set of benchmark instances, moderate duedates, and tight duedates refer to values of the duedate tightness factor δ of 0.6, and 0.9, respectively. Narrow duedate range and wide duedate range refer to values of the duedate range factor of 0.25 and 0.75, respectively. Mild setups and severe setups refer to values of the setup time severity factor ξ of 0.25 and 0.75, respectively.



Problem definition in form of network or graph
Identification of all the links, and the limiting criteria
Construction of initial solution with the probabilistic objective function
Creation of neighborhood or search solution space list
Generation of artificial Ant Agents
Ant Agents evaluation of best possible solution from the entire solution space (generally called as ANT TOUR)
Is the terminating condition reached?
Yes
Print the best solution
Terminate
No
Assign the new solution as the current solution
Update the solution set based on the goodness of the solution and generate a new search list

Figure 2: Generalized flow chart for ACO algorithm

For each test problem, two possible reference results were considered: the best known solutions available from Cicirello (Cicirello, 2003b) and the best solutions provided by the ACO_{LJ} algorithm in Liao and Juan (Liao and Juan, 2007). Comparatives with these two are presented in following.

Table 1: The results of Proposed ACO in comparison with Cicirello (2003)and Liao and Juan (ACO_{LJ})

	Prob.	Cicirello (2003)	ACO_{LJ} (2007)	Proposed ACO	Improvement Rate
Moderate Duedates, Narrow Duedate Range, Mild Setups	1	73176	73578	4853	-0.93404
	2	61859	60914	36000	-0.41803
	3	149990	149670	38538	-0.74306
	4	38726	37390	28045	-0.27581
	5	62760	62535	31810	-0.49315
	6	37992	38779	56370	0.453622
	7	77189	76011	48832	-0.36737
	8	68920	68852	55107	-0.20042
	9	84143	81530	47772	-0.43225
	10	36235	35507	40754	0.124714
Moderate Duedates, Narrow Duedate Range, Severe Setups	11	58574	55794	46264	-0.21016
	12	105367	105203	60916	-0.42187
	13	95452	96218	67734	-0.29604
	14	123558	124132	56290	-0.54653
	15	76368	74469	22436	-0.70621
	16	88420	87474	108729	0.229688
	17	70414	67447	22336	-0.68279
	18	55522	52752	0	-1
	19	59060	56902	33837	-0.42707
	20	73328	72600	10773	-0.85308
Moderate Duedates, Wide Duedate Range, Mild Setups	21	79884	80343	51160	-0.36323
	22	47860	46466	33576	-0.29845
	23	78822	78081	35415	-0.5507
	24	96378	95113	38906	-0.59632
	25	134881	132078	81400	-0.39651
	26	64054	63278	43830	-0.31573
	27	34899	32315	22557	-0.35365
	28	26404	26366	47264	0.790032
	29	75414	64632	51632	-0.31535
	30	81200	81356	31890	-0.60802
Moderate Duedates, Wide Duedate Range, Severe Setups	31	161233	156272	57134	-0.64564
	32	56934	54849	84014	0.475638
	33	36465	34082	60595	0.66173
	34	38292	33725	37965	-0.00854
	35	30980	27248	11491	-0.62908
	36	67553	66847	44512	-0.34108
	37	40558	37257	52182	0.286602
	38	25105	24795	71924	1.864927
	39	125824	122051	49444	-0.60704
	40	31844	26470	8327	-0.73851
Tight Duedates, Narrow Duedate Range, Mild Setups	41	387148	387866	46764	-0.87943
	42	413488	413181	79460	-0.80783
	43	466070	464443	0	-1
	44	331659	330714	0	-1
	45	558556	562083	40045	-0.92876
	46	365783	365199	67446	-0.81561
	47	403016	401535	81960	-0.79663
	48	436855	436925	24051	-0.94495
	49	416916	412359	50946	-0.8778
	50	406939	404105	68240	-0.83231
Tight Duedates, Narrow Duedate Range, Severe Setups	51	347175	345421	40065	-0.8846
	52	365779	365217	120492	-0.67059
	53	410462	412986	26218	-0.93652
	54	336299	335550	93527	-0.72189
	55	527909	526916	123890	-0.76532
	56	464403	464184	53768	-0.88422
	57	420287	419370	104784	-0.75068
	58	532519	533106	92218	-0.82702
	59	374781	370080	77238	-0.79391
	60	441888	441794	133090	-0.69882
Tight Duedates, Wide Duedate Range, Mild Setups	61	355822	355372	8554	-0.97596
	62	496131	495980	7824	-0.98423
	63	380170	379913	45090	-0.8814
	64	362008	360756	33400	-0.90774
	65	456364	454890	0	-1
	66	459925	459615	65032	-0.8586
	67	356645	354097	62288	-0.82535
	68	468111	466063	81360	-0.8262
	69	415817	414896	4383	-0.98946
	70	421282	421060	7972	-0.98108
Tight Duedates, Wide Duedate Range, Severe Setups	71	350723	347233	106875	-0.69527
	72	377418	373238	120816	-0.67989
	73	263200	262367	51084	-0.80591
	74	473197	470327	26850	-0.94326
	75	460225	459194	27292	-0.9407
	76	540231	527459	107672	-0.80069
	77	518579	512286	10946	-0.97889
	78	357575	352118	9265	-0.97409
	79	583947	584052	27544	-0.95284
	80	399700	398590	82902	-0.79259

Regarding to parameter setting, to determine the best values of parameters, a systematic grid search was conducted. The best values for our problem are as follows: $max\,interation = 1000, m = 30, \alpha = 0.1, \beta = 0.5$, $\rho = 0.1,\ q_0 = 0.9, O = 600,\ m = 15$.

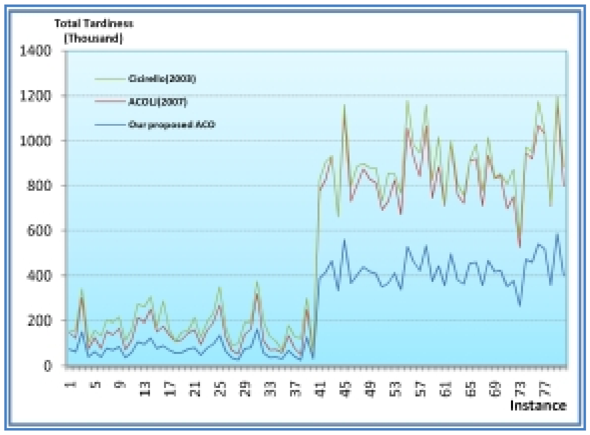


Figure 3: The relative total tardiness of our ACO versus the best previous studies

5- Conclusions

In this paper, we have proposed an ACO algorithm for minimizing the weighted tardiness on a single machine where setup times depend on sequence. Our ACO has a different manner in updating pheromone matrix in every k iteration. The algorithm updates 90% of the benchmark instances for the weighted tardiness problem. Instances of the benchmark library sets cover a spectrum from moderate to tightly constrained problem instances. The results of Proposed ACO in comparison with Cicirello (2003) and Liao and Juan (ACO_{LJ}) were showed in Table 1. The relative total tardiness of our ACO versus the best previous studies was depicted in Figure 3.

The SMTWTP with sequence-dependent setup times is important in nowadays production systems because the tardiness is recognized as the most important criterion. The importance of the problem in practice justifies the researchers paying more attention to the problem and its extensions, such as in the more complicated job shop environment. With the help of recently developed metaheuristics, the researchers should be able to tackle more difficult problems in the real world.

References

Abdul-Razaq, T. S., Potts, C. N. & Wassenhove, L. N. V. (1990) A survey of algorithms for the single machine total weighted tardiness problem. Discrete Applied Mathematics 26, 235–253.

Allahverdi, A., Gupta, J. N. & Aldowaisan, T. (1999) A review of scheduling research involving setup considerations. Omega, 27 219–239.

Bauer, A., Bullnheimer, B., Hartl, R. F. & Strauss, C. (1999) An ant colony optimization approach for the single machine total tardiness problem. In Proceedings of the 1999 Congress on Evolutionary Computation (CEC'99), 1445–1450.

Besten, M. D., Stützle, T. & Dorigo, M. (2000) Ant colony optimization for the total weighted tardiness problem. Proceeding PPSN VI, Sixth International Conference Parallel Problem Solving from Nature, Lecture Notes in Computer Science, 1917, 611–620.

Cicirello, V. A. (2003a) http://www.ozone.ri.cmu.edu/.

Cicirello, V. A. (2003b) Weighted Tardiness Scheduling with Sequence-Dependent Setups: A Benchmark Library. Intelligent Coordination and Logistics Laboratory The Robotics Institute Carnegie Mellon University Pittsburgh, Pennsylvania 15213.

Cicirello, V. A. (2007) The Challenge of Sequence-Dependent Setups: Proposal for a Scheduling Competition Track on One Machine Sequencing Problems.

Conway, R., Maxwell, W. & L. Miller (1967) Theory of Scheduling. IN Addison-Wesley, R., Ma (Ed.

Crauwels, H. A., Potts, C. N. & Wassenhove, L. N. V. (1998) Local search heuristics for the single machine total weighted tardiness scheduling problem. INFORMS Journal on Computing, 10, 341-350.

Dorigo, M. & Blum, C. (2005) Ant colony optimization theory: A survey. Theoretical Computer Science, 344, 243 – 278.

Dorigo, M. & Gambardella, L. M. (1997) Ant Colony System: A cooperative learning approach to the traveling salesman problem. IEEE Transactions on Evolutionary Computation, 1, 53–66.

Dorigo, M., Maniezzo, V. & Colorni, A. (1991) Positive Feedback as a Search Strategy. IN Dip. Elettronica, P. D. M. (Ed. Technical Report 91-016. Italy.

Du, J. & Leung, J. Y. T. (1990) Minimizing total tardiness on one machine np-hard problems. Mathematics of Operations Research, 15, 483–495.

Elmaghraby, S. E. (1968) The one machine scheduling problem with delay costs. Journal of Industrial Engineering, 19, 105–108.

Gravel, M., Price, W. L. & Gagn´E, C. (2002) Scheduling continuous casting of aluminum using a multiple objective ant colony optimization metaheuristic. European Journal of Operational Research, 143, 218–229.

Guntsch, M. & Middendorf, M. (2002a) Applying Population Based ACO to Dynamic Optimization Problems. In Ant Algorithms, Proceedings of Third International Workshop ANTS 2002, 2463, 111–122.

Guntsch, M. & Middendorf, M. (2002b) A Population Based Approach for ACO. Applications of Evolutionary Computing, Proceedings of EvoWorkshops2002: EvoCOP, EvoIASP, EvoSTim, 2279, 71–80.

K.R.Baker & Martin, J. B. (1974) An experimental comparison of solution algorithms for the single machine tardiness problem. Naval Logistics Quarterly, 20, 187–199.

Lawler, E. L. (1979) Efficient implementation of dynamic programming algorithms for sequencing problems. Report BW 106. Amsterdam, Mathematisch Centrum.

Lee, Y. H., Bhaskaran, K. & Pinedo, M. (1997) A heuristic to minimize the total weighted tardiness with sequence dependent setups. IIE Transactions, 29, 45-52.

Lenstra, J. K. & Kan, A. H. G. R. (1980) Complexity results for scheduling chains on a single machine. European Journal of Operational Research, 4, 270–275.

Liang, Y. C. (2001) Ant colony optimization approach to combinatorial problems. Department of Industrial and Systems Engineering. Auburn University.

Liao, C.-J. & Juan, H.-C. (2007) An ant colony optimization for single-machine tardiness scheduling with sequence-dependent setups. Computers & Operations Research 34, 1899–1909.

Picard, J. & Queyranne, M. (1978) The time-dependent traveling salesman problem and its application to the tardiness problem in one machine scheduling. Operations Research, 26, 86–110.

Potts, C. N. & Wassenhove., L. N. V. (1991) Single machine tardiness sequencing heuristics. IIE Transactions, 23, 346–354.

Schrage, L. & Baker, K. R. (1978) Dynamic programming solution of sequencing problems with precedence constraints. Operations Research, 26, 444–449.

Sena, T., Suleka, J. M. & Dileepan, P. (2003) Static scheduling research to minimize weighted and unweighted tardiness:A state-of-the-art survey. Int. J. Production Economics 83, 1–12.

Appendix

Computer CODE:

```
using System;
using System.Collections.Generic;
using System.Text;
using System.IO;
namespace WtsAco {

   class Solution : Wts, IAcoSolution, IComparable, IComparable<Solution> {

      private double totalTardiness;
      private double completionTime;
      public double TotalTardiness {
         get { return totalTardiness; }
      }

      public double CompletionTime {
         get { return completionTime; }
      }

      public List<WtsJob> JobsDone {
         get { return this; }
      }

      public static double Distance(WtsJob job1, WtsJob job2) {
         double SetupTime = 0;
```

```
        if (job1 != null)
            job2.SetupTime.TryGetValue(job1, out SetupTime);
        else SetupTime = job2.SelfSetupTime;

        return job2.ProcessTime + SetupTime;
    }

    public new void Add(WtsJob item) {
        if (Count == 0)
            completionTime += Distance(null, item);
        else
            completionTime += Distance(this[Count - 1], item);
        totalTardiness   =   item.Weight   *   Math.Max(0,   completionTime   -
                item.DueTime);
        base.Add(item);
    }

    public override bool Equals(object obj) {
        Solution other = (Solution)obj;
        return other.ToString() == ToString();
    }

    public override int GetHashCode() {
        return ToString().GetHashCode();
    }

    public override string ToString() {
        StringBuilder s = new StringBuilder();
        foreach (WtsJob job in this) {
            s.Append(job.Id);
            s.Append(" ");
        }
        return s.ToString();
    }
    public int CompareTo(object obj) {
        Solution other = (Solution)obj;
        return CompareTo(other);
    }

    public int CompareTo(Solution other) {
        int r = other.completionTime.CompareTo(completionTime);
        if (r != 0)
            return r;
        return other.TotalTardiness.CompareTo(totalTardiness);
    }

    public static int BestToWorseComparer(Solution first, Solution second) {
        return first.completionTime.CompareTo(second.completionTime);
    }

}

class OmAco : IAco {

    Wts wts;
    Random random;
    double initialPheromone = 1;
    double alpha = .1;
    double betha = 1;
    int solutionCount = 15; // m
    double O = 600;
    double[,] pheromone;
    List<Solution> solution;
    public OmAco(Wts wts) {
        this.wts = wts;
        pheromone = new double[wts.Count, wts.Count];
        solution = new List<Solution>(solutionCount);
        random = new Random();
    }
    public void SetParameters() {
        initialPheromone                                                        =
                double.Parse(System.Configuration.ConfigurationManager.AppSetti
                ngs["InitPhermone"]);
        alpha                                                                   =
                double.Parse(System.Configuration.ConfigurationManager.AppSetti
                ngs["Alpha"]);
        betha                                                                   =
                double.Parse(System.Configuration.ConfigurationManager.AppSetti
                ngs["Beta"]);
        solutionCount                                                           =
                int.Parse(System.Configuration.ConfigurationManager.AppSettings[
                "m"]);
        O                                                                       =
                double.Parse(System.Configuration.ConfigurationManager.AppSetti
                ngs["O"]);
    }
    private void InitializePheromoneMatrix() {
        for (int i = 0; i < wts.Count; i++)
            for (int j = 0; j < wts.Count; j++)
                pheromone[i, j] = initialPheromone;
    }
    private void InitializePopulation() {
        int x = 0;
        while (x < solutionCount) {
            Solution c = ConstructASolution();
            if (!solution.Contains(c)) {
                solution.Add(c);
                x++;
            }
        }
        solution.Sort();
    }
    private Solution ConstructASolution() {
        Solution s = new Solution();
        Wts reamining = (Wts)wts.Clone();
        int r = random.Next(wts.Count);

        s.Add(reamining[r]);
        reamining.RemoveAt(r);
        while (reamining.Count > 0) {
            WtsJob job = SelectJob(s[s.Count - 1], reamining);
            s.Add(job);
            reamining.Remove(job);
        }
        return s;
    }
    private WtsJob SelectJob(WtsJob LastJob, Wts RemainingJobs) {
        Dictionary<WtsJob,  double>  JobProbability  =  new  Dictionary<WtsJob,
                double>();
        double probSum = 0;

        foreach (WtsJob job in RemainingJobs) {
            JobProbability[job] = Math.Pow(pheromone[LastJob.Id, job.Id], alpha)
                * Math.Pow(Solution.Distance(LastJob, job), betha);
            probSum += JobProbability[job];
        }

        WtsJob selectedJob = null;
        foreach (KeyValuePair<WtsJob, double> JobProb in JobProbability) {
            if (selectedJob == null)
                selectedJob = JobProb.Key;
            else if (JobProb.Value > JobProbability[selectedJob])
                selectedJob = JobProb.Key;
        }
        return selectedJob;
    }
    private void UpdatePopulation(Solution c) {
        solution[0] = c;
        solution.Sort();
    }
    private void UpdatePheromonesMatrix() {
        InitializePheromoneMatrix();
        foreach (Solution s in solution)
            for (int i = 1; i < s.Count; i++)
                pheromone[s[i - 1].Id, s[i].Id] += O / solutionCount;
    }
    public void Run(int times, int K) {
        InitializePheromoneMatrix();
        InitializePopulation();

        for (int i = 0; i < times; i++) {
            for (int k = 0; k < K; k++) {
                Solution c = ConstructASolution();
                if     (c.CompletionTime     <     solution[0].CompletionTime     &&
                !solution.Contains(c))
                    UpdatePopulation(c);
            }
            UpdatePheromonesMatrix();
        }
    }
    public IAcoSolution GetBestResult() {
        return solution[solution.Count - 1];
    }

  }
}
```