Dynamic and Stochastic Job Shop Scheduling Problems Using Ant Colony Optimization Algorithm

Rong Zhou[1*], Mark Goh[1,4,5], Gang Chen[2], Ming Luo[3], Robert De Souza[1]
1 The Institute of Logistics – Asia Pacific, Singapore
2 School of Electrical and Electronics Engineering, Nanyang Technological University, Singapore
3 Singapore Institute of Manufacturing Technology (SIMTech), Singapore
4 Business School, National University of Singapore
5 University of South Australia
EMAIL: tlizr@nus.edu.sg

Abstract: Reactive scheduling is often been criticized for its inability to provide timely optimized and stable schedules. So far, the extant literature has focused on generating schedules that optimize shop floor efficiency. Only a few have considered optimizing both shop floor efficiency and schedule stability. This paper applies a unique self-adaptation mechanism of the ant colony optimization (ACO) algorithm to enable the reactive scheduling approach to generate better and timely stable and quality schedules for dynamic and stochastic job shop scheduling problems.

Keywords: Self-Adaptation Mechanism, Ant Colony Optimization, Schedule Stability, Dynamic and Stochastic Job Shop Scheduling.

I. Introduction

In a job shop scheduling problem (JSSP) with *n* jobs and *m* machines, each job has *m* operations to be processed on *m* different machines. The sequence of operations for each job to be processed is unique. A schedule is thus sought to complete all jobs considering shop efficiency, which can be measured through makespan, tardiness, flow time, etc. The JSSP is a well-known NP-hard problem.
A JSSP is dynamic (stochastic) when dynamic (stochastic) events are considered. A typical dynamic event is an unexpected arrival of a new job while stochastic events include machine breakdowns or variations in processing times. Such events, which are common in practice, can immediately render a schedule obsolete by changing the underlying JSSP. Thus, a schedule has to be updated within a short time period for smooth production. Hence, any newly updated schedule must have a high performance in terms of shop floor efficiency and be as consistent as possible to the original one since many related supply chain activities such as purchasing, materials handling, inventory, and distribution were planned based on the original schedule. The attribute that an updated schedule is consistent to the original one is referred to as schedule stability. The main challenge for scheduling dynamic or stochastic JSSPs is to generate both quality and consistent schedules under tight computing time constraints.
In this regard, the solution approach has been twofold: before and after the actual occurrence of the disruption. Given this, there are two categories of scheduling approaches: proactive and reactive. Proactive scheduling, also known as robust scheduling, seeks to pro-actively build robust schedules to sustain interruptions through strategies like inserting idle times or introducing redundancy in either resource or schedule. The main disadvantage of proactive scheduling is that shop efficiency performance will be sacrificed if the expected interruptions do not occur at a later stage.
Reactive scheduling, or rescheduling, focuses on generating new schedules or modifying an existing schedule after the actual occurrence of an interruption [1]. Its advantages are: 1) there is no wasted computing effort a priori nor the sacrifice of shop efficiency from inserted idle times; 2) an optimized new schedule with a global view can be targeted if a complete schedule is generated. Despite this, besides the prohibitive computing costs for most scheduling problems, is that the newly generated schedule may cause “nervousness” on a shop floor due to the inconsistency between the updated and the original schedules. Scheduling for schedule stability is also referred to as minimally disruptive, minimal perturbation, and minimum deviation scheduling.
The current study shows that a complete regeneration of the schedule in the paradigm of reactive scheduling, against the common belief in the literature, can also provide timely stable and quality schedules for dynamic and stochastic JSSPs using the unique self-adaptation mechanism of the ACO algorithm.
Two attributes of the ACO, which is inspired from the behavior of foraging ants, make it amenable for solving moderate dynamic or stochastic JSSPs. First, ACO can generate optimized solutions and the pioneering work in this direction has been reported in [2][3][4][5][6] for many types of scheduling problems. Next, ACO has a unique self-adaptation mechanism [7][8][9] to update an obsolete schedule without losing quality and stability.
Beyond those achievements, this paper aims to answer some additional research questions other than what the above mentioned: 1) what is the stability performance of an updated schedule generated through this mechanism in addition to its shop floor efficiency? 2) How does the response time affect those two performances? We investigate both questions for moderate dynamic and stochastic JSSPs.
The paper is structured as follows. Section II presents the ACO algorithm and its self-adaptation mechanism. The experiments and solution analyses on dynamic and stochastic JSSPs are found in Sections III and IV respectively. Section V concludes.

II. ACO and its self-adaptation behavior

The notations used in this paper are listed as follows.
h is the index of iteration number; $p_{ij}(h)$ is the probability that an ant travels from node *i* to node *j* at h^{th} iteration; $\tau_{ij}(h)$ is the quantity of pheromone on the edge connecting nodes *i* and *j* at h^{th} iteration; d_{ij} is the heuristic distance between nodes *i* and *j*; ρ is the evaporation coefficient, a real number between 0 and 1; $\Delta\tau_{ij}(h)$ is the amount of increased pheromone on the edge connecting nodes *i* and *j* at the h^{th} iteration; *Q* is the constant representing the total quality of pheromone on a route; $f_{evaluation}(best\_so\_far)$ is the best value obtained so far for a given objective.

a. ACO
The ACO algorithm in basic form is given in Figure 1.

```
Step 1: initiate iteration counter $h$, the best global route, and the
        pheromone matrix
Step 2: initiate ants
        Step 2.1: each ant finds a route according to formula (1)
        Step 2.2: find the best route in the current iteration; update the
        best global solution
        Step 2.3: update the pheromone matrix according to (2) and (3)
Step 3: increase $h$ by one. Go to step 2 unless termination criteria are
        met.
```

Figure 1.The ACO algorithm

Ant *i* chooses its next destination according to the State Transition Rule in eqn (1) [3].

$$p_{ij}(h) = \frac{\left[\tau_{ij}(h)\right]^{\alpha} \cdot \left[\frac{1}{d_{ij}}\right]^{\beta}}{\sum_{j \in allowed-nodes} \left[\tau_{ij}(h)\right]^{\alpha} \cdot \left[\frac{1}{d_{ij}}\right]^{\beta}} \tag{1}$$

Each ant leaves some amount of pheromone on the edges it has passed. The values of the pheromone on all edges are recorded in a pheromone matrix. Its updating involves evaporation and enhancement, which are represented in eqn (2) and (3) [3].

$$\tau_{ij}(h+1) = (1-\rho)\cdot\tau_{ij}(h) + \rho\cdot\Delta\tau_{ij}(h+1) \tag{2}$$

$$\Delta\tau_{ij}(h+1) = \begin{cases} \dfrac{Q}{f_{evaluation}(best\_so\_far)} \\ 0, \quad otherwise \end{cases} \tag{3}$$

$$\begin{cases} State\,Transition\,Rule, & u > v \\ p_{ij}(h) = \dfrac{\left[\frac{1}{d_{ij}}\right]}{\sum_{j \in allowed-nodes}\left[\frac{1}{d_{ij}}\right]} & u < v \end{cases} \tag{4}$$

u: random number between [0..1]
v: percentage of variation rate [0..1]

Next, a variation rate parameter *v* can be introduced to further diversify the search space through eqn (4), where *u* is a random real number in [0, 1] and *v* is a preset value. If *u* is greater than *v*, the probability of an ant choosing a next destination follows the State Transition Rule, which has the impact from both the pheromone and heuristic distance d_{ij}; otherwise, only the information heuristic distance is used to decide the next destination.

b. Apply ACO to a classic JSSP
A classic *n*-job and *m*-machine JSSP for optimizing makespan can be represented in the form of $n/m/G/C_{max}$, where *G* is the job shop and C_{max} is the makespan. The problem graph of a $2/3/G/C_{max}$ JSSP can be illustrated in Figure 2. Nodes 1 to 6 represent operations O_{11}, O_{12}, to O_{13}, and O_{21}, O_{22}, to O_{23}. They are connected by horizontal directional edges indicating the precedence constraints given in the technical matrix *T*. The bi-directional edges indicate no ordering constraints among those operations. Dummy nodes 0 and 7 representing the source and the sink of the graph are the start and the end points of the route.

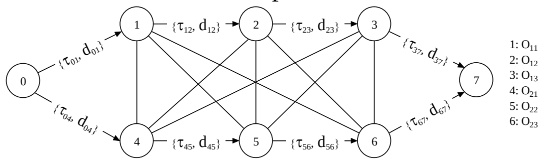
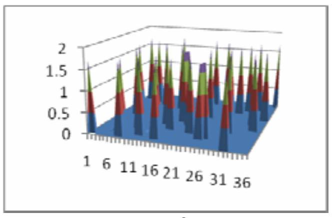


Figure 2. Graph representing a 2 x 3 JSSP

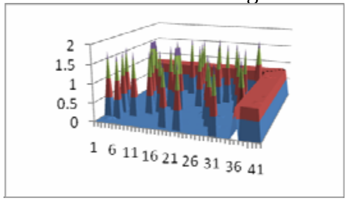
Each edge is associated with a pair of values $\{\tau_{ij}, d_{ij}\}$, representing the amount of pheromone on it and the heuristic distance between the two nodes it connects. The value for τ_{ij} is found in the pheromone matrix, which records the pheromone values of all the edges connecting every two nodes and is updated using eqn (2) and (3) by the ants that find the best solutions. d_{ij} represents the processing time of node *i*.

c. ACO self-adaptation mechanism
The ACO self-adaptation mechanism is inspired from the following natural phenomenon. When the current route from the nest of a group of ants to a food source is blocked, the ants do not return to their nest to begin a new search for a totally different short route. Instead, they quickly form a new short route that is similar to the previous one using exactly the same strategy based on the remaining pheromone information in the environment. Thus, the remaining pheromone information on the ground can direct ants to find a new good route, which is similar to the original one, in a changed environment.
This mechanism is realized by keeping the old pheromone information on all remained edges while setting a preset amount of pheromone value on all newly generated edges. Then, ACO is run until a new good solution is found in the allowed time span. The new schedule is regarded as a self-adapted result from the obsolete one.

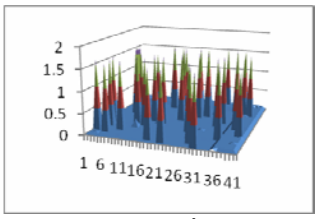
Figure 3 illustrates the changes in the pheromone matrix after an unexpected job arrives. Figure 3a represents the pheromone matrix of the original schedule found for the *6/6/G/C_{max}* problem after 1000 iterations. Figure 3b is the updated pheromone matrix removing the nodes for completed operations and adding in new nodes representing new operations. The edges incurred by the new operations are initiated with a pheromone value of 0.8 in Figure 3b. Figure 3c shows the new pheromone matrix of the new schedule found by ACO after 250 iterations.



(a) Pheromone matrix for original schedule



(b) Initiate the pheromone matrix keeping previous pheromone information



(c) Pheromone matrix after 250 iterations

Figure 3. Rescheduling with the ACO self-adaptation mechanism

It is clear that the solution information of the original schedule recorded in the lower left part of the pheromone matrix (Figure 3b) can most likely be kept in its updated schedule (Figure 3c). However, if all the old pheromone information is erased by initiating a new problem graph with exactly the same pheromone value on each edge, there will be no connection between the two problem graphs and the solution adaptation will not happen.

III. Experiment I: aco for dynamic jssps

The main goal of experiment I is to study the ability of the ACO self-adaptation mechanism in promptly generating high quality and stable intermediate schedules for dynamic JSSPs given different response times.

a. Experimental design

A schedule is issued to a job shop floor sought using ACO as the initial executing schedule. The *MT-6/6/G/C_{max}* problem is used here for its known optimum 55 [10] and its simplicity. An unexpected job (the first job in the *MT-6/6/G/C_{max}* problem) arrives after the schedule is executed for one time unit. The reason for setting the arrival time at an early stage is to make the rescheduling problem nontrivial.

A new complete schedule is then found for the changed set of operations through two ways: with and without ant self-adaptation mechanism. For each approach, five levels of response times are allowed. The five levels of response times are represented as 10, 50, 100, 150, and 200 iterations. Thus, there are totally ten problem instances and each has ten replications. Two performance measures in this study are makespan and schedule stability. Makespan is the timespan of the schedule from the start time of the first job to the finish time of the last job. Schedule stability is described in terms of the deviation of the start times of all existing operations.

b. ACO Parameters

The ACO algorithm has nine parameters. α and β in eqn (1) decide the possibility of the next node that an ant is going to choose in the problem graph; ρ in eqn (2) and Q in eqn (3) are used to update the pheromone matrix; v and u in eqn (4) are explained in section IIIa; Other parameters not directly shown in the equations are l, τ_0, and n. l is the number of ants per iteration; τ_0 is the initial pheromone value on the edges; n is the minimal iteration number for the algorithm to find a solution. Here, n is 600 for the original *MT-6/6/G/C_{max}* problem. The impact of many of the parameters based on the *MT-6/6/G/C_{max}* problem is analyzed in [6] and the following values can generate good solutions: $\alpha = 10$, $\beta = 10$, $\rho = 0.01$. Further, this paper suggests that the values of τ_0 and Q do not have significant impact on the final solution and the speed to convergence. $\tau_0 = 1.5$ and $Q = 100$ are used in this study. Finally, $v = 0.15$ and $l = 36$. For the rescheduling problem, $\tau_0' = 0.8$ is the initial pheromone value for all the newly added edges.

c. Results

The average performance values of the ten replications regenerating complete schedules with and without the ACO self-adaptation mechanism are given in Tables 1 and 2, respectively.

Table 1: Results with self-adaptation for dynamic JSSPs

Responding iterations	10	50	100	150	200
Makespan (original)	61.57	60.86	61.47	61.68	60.47
Makespan (new)	71.58	66.97	65.67	65.26	64.06
Makespan deviations	10.01	6.11	4.6	3.59	3.59
STD	289.59	148.13	113.29	102.98	103.12

Table 2: Results without self-adaptation for dynamic JSSPs

Responding iterations	10	50	100	150	200
Makespan (original)	61.17	60.57	61.77	62.98	62.77
Makespan (new)	91.0	85.91	81.49	77.99	76.19
Makespan	29.83	25.34	19.71	15.01	13.42

deviations					
STD	495.53	507.27	456.81	440.14	437.91

The second row records the mean makespan values of the schedules for the original *MT-6/6/G/C$_{max}$* problem and the third row records the mean makespan values of the schedules for the new JSSPs after new jobs arrive. The difference in the makespan values between the original and new schedules is the makespan deviation, which is recorded in the fourth row. Finally, the last row records the mean starting time deviations (STD), which are calculated through eqn (5):

$$\text{Starting time deviation} = \sum_{i=0}^{n}\sum_{j=0}^{m}(t_{ij} - t_{ij}^{'}) \qquad (5)$$

where t_{ij} and $t_{ij}^{'}$ are the new and the original start times of operation O_{ij}.

The other statistics of the results such as maximum, minimum, and mean values of the makespan and the starting time deviation, as well as their respective standard deviations, are recorded in Table 3. The values before the slash "/" are the results using the ACO self-adaptation mechanism while the values after the slash are results without using the mechanism.

Table 3: Results with/without the self-adaptation for dynamic JSSPs

Responding iterations	10	50	100	150	200
Makespan (max)	76.08/ 94.12	71.09/ 88	68.06/ 85.09	70.06/ 83.06	69.04/ 85.09
Makespan (Min)	66.07/ 89.1	62.06/ 80	64.05/ 79.08	61.04/ 68.12	61.04/ 69.1
Makespan (Mean)	71.58/ 91	66.97/ 85.8	65.67/ 81.49	65.26/ 77.99	64.06/ 76.19
Makespan Std dev	2.81/ 1.8	3.22/ 2.62	1.64/ 2.27	2.79/ 4.30	2.26/ 4.70
STD (max)	503.28/ 642.62	393.46/ 732.76	248.06/ 550.18	213.85/ 634.48	194.96/ 594.51
STD (min)	96.91/ 368.27	2.00/ 373.59	19.08/ 334.11	13.16/ 242.14	6.99/ 246.54
STD (mean)	289.59/ 495.53	148.13/ 507.27	113.29/ 456.81	102.98/ 440.14	103.12/ 437.91
STD Std dev	150.28/ 84.33	124.25/ 98.55	60.99/ 77.67	60.02/ 124.18	60.29/ 114.97

The second row in Table 3 records the maximum makespan values among the ten replications for each problem; the third and fourth rows record the minimum and the mean values. Then follow the makespan standard deviations. Similar statistics are recorded in the rest of the rows for the performance of the starting time deviation.

d. Solution analysis

Overall, the results show that the approach with the ACO self-adaptation mechanism produces new schedules with much better makespan and stability performance than those without using the ACO self-adaptation mechanism for all five levels of the response times. For the makespan deviation measure, the performance comparison between two approaches is illustrated in Figure 4, where labels 1, 2, 3, 4, and 5 refer to the five response times defined by 10, 50, 100, 150, and 200 iterations.

Figure 4 shows that the makespan values of the JSSP after the arrival of an unexpected job increase for both approaches. If the ACO adaptation mechanism is used, the average makespan values increase only 10, 6.11, 4.6, 3.59 and 3.59 time units (Table 1) for the five levels of the response times, respectively. However, the corresponding values jump to 29.83, 25.34, 19.71, 15.01, and 13.42 time units (Table 2), without the adaptation mechanism. Clearly, rescheduling using the ACO adaptation mechanism improves the makespan performance of the new schedules.

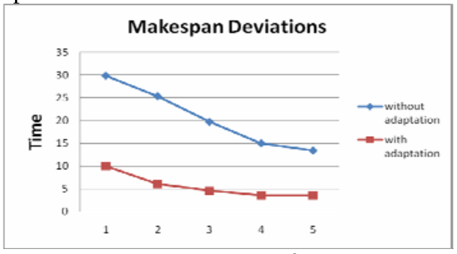


Figure 4. Makespan deviations for dynamic JSSPs

The adaptation mechanism is very efficient as high quality schedules can be obtained with the response time as short as 10 or 50 iterations. Tables 1 and 2 show that an average makespan value of about 61 is obtained for the original *MT-6/6/G/C$_{max}$* problem by ACO using 600 iterations. Thus, with only 10 iterations, it is hard, if not impossible, for ACO without using the adaptation mechanism to find a near optimal solution from scratch. However, if the ACO adaptation mechanism is used, one of the best experiments obtains a new schedule with only an increased makespan of 3.97 and the starting time deviation of 130.06 after 10 iterations. Figure 4 also shows that the quality of the new schedules improves as response times increase for both approaches.

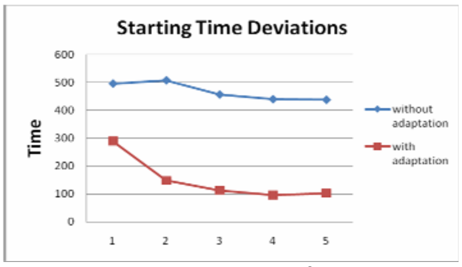


Figure 5. Starting time deviations for dynamic JSSPs

Similarly, for the measure of the starting time deviation, the performance comparison between two approaches is shown in Figure 5. The start times of some operations are changed due to an arriving job. If the ACO adaptation mechanism is used, the average starting time deviation are 289.59, 148.13, 113.29, 95.99, and 103.12 for the five levels of response times, respectively. However, for the approach without using the mechanism, the corresponding values are 495.53, 507.27, 456.81, 440.14, and 437.91. As a smaller value implies better stability, rescheduling using the ACO adaptation mechanism improves schedule stability.

Finally, Table 3 shows that the approach using the ACO adaptation mechanism generates better schedules in all problem instances in terms of maximum, minimum, and mean values for both performance measures of makespan and starting time deviation. As for the standard deviation, the approach with the self-adaptation mechanism outperforms the one without the self-adaptation when the reaction times increase to 100, 150, and 200 iterations. The analysis of the starting time deviations follows a similar pattern.

IV. Experiment II: aco for stochastic jssps

Experiment II seeks to show that the self-adaptation of ACO can generate solutions with both high performance and stability for stochastic JSSPs upon machine breakdowns.

a. Experimental design

The *MT-6/6/G/C*$_{max}$ problem is again used here with a minor modification. Workcenter 5 has two identical machines. An optimal or near optimal schedule is sought in advance using ACO with 600 iterations and released to guide production. During the execution of the schedule, one of the parallel machines in workcenter 5 unexpectedly breaks down and the current schedule becomes obsolete. A new complete schedule is then quickly sought with only one workable machine left in workcenter 5. To make the updating problem nontrivial, the machine breakdowns occur right at the starting time of the original schedule. Thus, the JSSP problem with parallel machines changes to the classic *MT-6/6/G/C*$_{max}$ problem, which has an optimal makespan value of 55. The rest setting and parameter values of experiments are similar to the previous section.

b. Results

The average performance values for the ten replications regenerating complete schedules with and without the ACO self-adaptation mechanism are given in Tables 4 and 5, respectively.

Table 4: Results of ACO with self-adaptation for stochastic JSSPs

Responding iterations	10	50	100	150	200
Makespan (original)	57.46	57.46	57.36	58.86	58.46
Makespan (new)	63.95	63.26	62.95	63.26	62.35
Makespan dev	6.49	5.79	5.59	4.39	3.89
Starting time dev	84.44	80.56	86.53	122.73	109.53

Table 5: Results of ACO without self-adaptation for stochastic JSSPs

Responding iterations	10	50	100	150	200
Makespan (original)	57.36	57.06	58.36	57.57	57.80
Makespan (new)	82.69	79.08	74.28	69.59	68.68
Makespan dev	25.32	22.02	15.92	12.03	10.88
Starting time dev	384.02	415.24	312.44	358.74	318.24

Further, like Table 3, Table 6 also records the statistics of maximum, minimum, and mean values, as well as the standard deviations of the two performance measures.

Table 6: Results with/without the self-adaptation for stochastic JSSPs

Responding iterations	10	50	100	150	200
Makespan (max)	68.04/ 87.11	69.09/ 83.09	67.05/ 80.07	68.08/ 77.09	67.05/ 76.08
Makespan (Min)	58.06/ 80.09	59.04/ 71.06	59.05/ 69.08	59.05/ 63.06	60.05/ 64.08
Makespan (Mean)	63.95/ 82.69	63.26/ 79.08	62.95/ 74.28	63.26/ 69.59	62.35/ 68.68
Makespan Std dev	2.73/ 1.96	3.33/ 3.90	1.64/ 3.55	2.41/ 4.43	2.11/ 3.44
STD (max)	225.21/ 486.22	114.93/ 541.26	169.05/ 441.47	237.48/ 484.54	166.99/ 488.18
STD (min)	21.08/ 292.02	37.94/ 265.2	41.03/ 157.05	10.93/ 238.32	66.00/ 174.24
STD (mean)	84.44/ 384.02	80.56/ 415.24	86.53/ 312.44	122.73/ 358.74	109.53/ 318.239
STD Std dev	57.56/ 57.09	9.28/ 93.95	15.15/ 94.4	67.66/ 86.51	32.89/ 109.88

c. Results analysis

Tables 3 and 4 again show that the approach with ACO self-adaptation produces new schedules with significantly better makespan and stability performances than those without using the ACO self-adaptation for all five levels of response times.

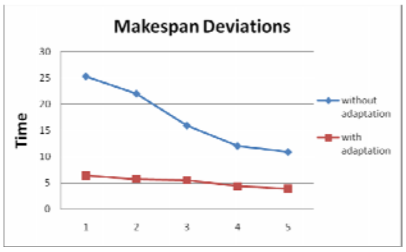


Figure 6. Makespan deviations for JSSPs with machine breakdown

Figure 6 shows the performance comparison between two approaches for the makespan deviation measure, where the labels 1, 2, 3, 4, and 5 refer to the five response times defined by 10, 50, 100, 150, and 200 scheduling iterations. Generally, the makespan values of the JSSP after a machine breakdown increase for both approaches. When the ACO adaptation mechanism is used, the average makespan values increase only 6.49, 5.79, 5.59, 4.39 and 3.89 time units for the five levels of response times respectively. However, the corresponding values are 25.32, 22.02, 15.92, 12.03, and 10.88 time units without the adaptation mechanism.

The ACO adaptation mechanism again shows its ability in generating relatively high quality schedule (63.95 and 63.26) within very short response times even at 10 and 50 iterations. As a comparison, ACO without using adaptation mechanism can only yield makespan values of 82.69 and 79.08 for the same computational times. Indeed, one of the best results in the experiments using 10 responding iterations shows that the updated schedule has an increased makespan value of only 2.01 with the starting times deviation at 21.08. For the starting time deviation measure, the experiments show that updating schedules with the ACO adaptation can significantly improve schedule stability. If the mechanism is used, the average starting time deviations are 84.44, 80.56, 86.53, 122.73, and 109.53 (Table 4) respectively for the five levels of response times. When the adaptation mechanism is not applied, the corresponding values increase to 384.02, 415.24, 312.44, 358.74, and 318.24 (Table 5), respectively.

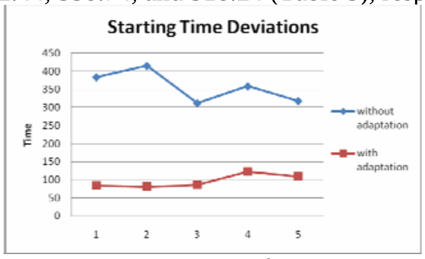


Figure 7. Starting time deviations for JSSPs with machine breakdown

Figure 7 presents the performance comparison between the two approaches for the starting time deviation measure. Similar to those in dynamic JSSP, the ACO with the adaptation mechanism can produce stable schedules within very short response times, e.g. 10 or 50 iterations. However, as the response times continue to increase, the stability reduces for both approaches.

Table 6 shows that the ACO using the adaptation mechanism again generates better schedules in all problem instances in terms of maximum, minimum, and mean values for both performance measures of makespan and starting time deviation. As for the standard deviation, the approach with the self-adaptation mechanism outperforms the one without the self-adaptation in all problem instances except for the makespan measure where 10 iterations are used to find new schedules.

Overall, the above experiments show that the ACO self-adaptation mechanism can help to generate quality and stable schedules for stochastic JSSPs. This behavior can be explained as follows. When the ACO adaptation is applied, a small number of responding iterations implies a small number of possible variations from the original schedule. Thus, the updated schedule is likely to be close to the original schedule in terms of operation sequence and schedule stability. When the response time continues to increase, ACO explores larger solution spaces for better schedules to replace the original one. This can cause more operations to be reallocated and subsequently deteriorate the schedule stability. If the ACO adaptation is not used, the schedule stability is hardly guaranteed as the updated and the original schedules have no connections at all.

V. Conclusion

This study explores the ability of the ACO self-adaptation mechanism to provide quality and stable schedules for dynamic and stochastic JSSPs within limited response times. Our results suggest that the quality and the stability of the schedules generated with the proposed mechanism are significantly superior to those without using the mechanism for all five levels of response times, especially, when the response times are extremely tight which is often the case in practice. This is critical for practical applications, where the schedule quality and stability provided with promptness are important for operation, profit, and competition.

Another merit of this approach is that no extra procedure or computational cost is required except for the normal overhead of the ACO algorithm. Further, the application of the ACO adaptation mechanism is simple and even the original optimal schedule is not necessarily generated by ACO, rather, it can be provided by another suitable approach. Finally, it is against the common belief in the reactive scheduling literature that regenerating complete schedules leads to schedule instability, which may be true for scheduling techniques that do not use an adaptation mechanism.

Acknowledgements

This work was partly supported by the Agency for Science, Technology, and Research (A*STAR) of Singapore under SERC TSRP on IMSS Project 0521160078.

References

[1] Aytug, H., Lawley, M.A., McKay, K., Mohan, S., and Uzsoy, R., 2005. Executing production schedules in the face of uncertainties: A review and some future directions, European Journal of Operational Research, 161, 86-110.

[2] Dorigo, M., Maniezzo, M., and Colorni, A., 1991. Distributed optimization by ant colonies, Proceedings of ECAL91 – European Conference on Artificial Life, Elsevier Publishing, pp. 134-142.

[3] Dorigo, M., Maniezzo V., and Colorni, A., 1996. Ant System: Optimization by a colony of cooperating agents. IEEE Transactions on Systems, Man, and Cybernetics, Part B, 26(1).

[4] Dorigo, M., Di Caro, M., and Gambardella, M., 1999. Ant algorithms for discrete optimization, Artificial Life, Vol.5, No.3, 137-172.

[5] Colorni, A., Dorigo, A., Maniezzo, A., and Trubian, A., 1994. Ant system for job-shop scheduling, Belgian J. Oper. Res., Statist. Comput. Sci. (JORBEL), vol. 34, no. 1, 39–53.

[6] Zwaan S.v.d. and Marques, C., 1999. Ant colony optimisation for job shop scheduling. In Proceedings of the ’99 Workshop on Genetic Algorithms and Artificial Life GAAL’99.

[7] Zhou, R., Chen, G., Yang, Z.H., Luo, M., and Zhang, J.B., 2008. Self-organized manufacturing resource management: an ant-colony inspired approach, the Proceedings of the 10th International Conference on Control, Automation, Robotics and Vision ICARCV, December 17-20, 2008, Hanoi, Vietnam, 904-909.

[8] Zhou, R., Ren, W., Chen, G., Yang, Z.H., Zhang, J.B., and Luo, M., 2008. Ant Colony Inspired Self-Healing for Resource Allocation in Service-Oriented Environment Considering Resource Breakdown. The 2008 IEEE/WIC/ACM International Conference on Web Intelligence.

[9] Zhou, R., Ren W., Chen, G., Yang, Z.H., Shen, H.F., Zhang, J.B., and Luo, M., 2008. Ant Colony Inspired Self-Healing for Resource Allocation in Service-Oriented Environment Considering Resource Breakdown, in the proceedings of the 2008 IEEE/WIC/ACM International Conference on Web Intelligence and Intelligent Agent Technology，Sydney, Australia, WI-IAT, 66-69.

[10] OR-Library, http://people.brunel.ac.uk/~mastjjb/jeb/info.html (accessed Jan. 2010).