EFFICIENT ALGORITHMS FOR MACHINE SCHEDULING PROBLEMS WITH EARLINESS AND TARDINESS PENALTIES

Guang FENG
School of Computing
National University of Singapore

Hoong Chuin Lau*
School of Information Systems
Singapore Management University
80 Stamford Road, Singapore (178902) Tel: +65- 6828 0229
hclau@smu.edu.sg

Abstract In this paper, we study the multi-machine scheduling problem with earliness and tardiness penalties and sequence dependent setup times. This problem can be decomposed into two subproblems - sequencing and timetabling. Sequencing focuses on assigning each job to a fixed machine and determine the job sequence on each machine. We call such assignment a semi-schedule. Timetabling focuses on finding an executable schedule from the semi-schedule via idle-time insertion. Sequencing is strongly NP-hard in general. Although timetabling is polynomial-time solvable, it can become a computational bottleneck if the procedure is executed many times within a larger framework. This paper makes two contributions. We first propose a quantum improvement to the computational efficiency of the timetabling algorithm. We then apply it within a squeaky wheel optimization framework to solve the sequencing and overall problem. Finally, we demonstrate the strength of our proposed algorithms by experiments.

Keywords: Earliness-Tardiness, Meta-heuristics, Scheduling, Squeaky Wheel

1. Introduction

Scheduling problems arise commonly in manufacturing and logistics operations. As business move toward greater customer focus, the underlying scheduling problems need to be extended to cater to those needs. Traditionally,

*Corresponding Author

only lateness (or tardiness) is a concern. Research is concerned primarily with regular performance measures, i.e. measures that are non-decreasing functions of job completion times, such as mean weighted flow times, makespan, mean weighted tardiness. Concepts of Just-in-time enriched manufacturing and logistics practices and raised a new class of non-regular performance measures involving earliness. The cost of earliness can be enormous. Cisco, for instance, wrote off $2.25 billion worth of inventory in 2001 because the company had placed large orders in anticipation of future projected sales which did not materialize (cnet.com (2001)). In logistics and supply chain management, a key decision is to determine when to assemble components into final products, and when to ship items across the multi-tier distribution centres across the chain. Again, the cost of holding inventory can be enormous for the organization if the customer does not accept an early shipment. Scheduling problems that consider both earliness and lateness penalties are often called **ET** problems.

In this paper, we use the following notations. There are N jobs to be processed on M identical machines. A job can be processed on any machine with no machine preferences. Each job is processed exactly once non-preemptively. All jobs are ready at time 0. Job i has processing time p_i and due time d_i. The unit penalty (or cost) for earliness is α_i, and that for lateness is β_i. Often, we need to impose setup time s_{ij} between two successive jobs i and j. A schedule (or more precisely, executable schedule) is defined as the set of completion times c_i for all jobs. This is opposed to a semi-schedule which is an intermediate solution specifying the job sequence on each machine (but without the actual completion times). The total penalty (or cost) of the schedule is defined as $\sum(\alpha_i(0, d_i - c_i)^+ + \beta_i(0, c_i - d_i)^+)$. For convenience, we define $E_i = (0, d_i - c_i)^+$ as the earliness of job i, and $T_i = (0, c_i - d_i)^+$ as the *tardiness* of job i . It is called early if $E_i > 0$, or it is called tardy if $T_i > 0$. Notice at least one of E_i and T_i is zero. When $E_i = T_i = 0$, job i is an on-time job. Sometimes the on-time jobs are also treated as early jobs. The performance measure can hence be defined as $\sum(\alpha_i E_i + \beta_i T_i)$. A schedule is optimal if there is no other schedule with a lower cost.

MGET refers to the class of Multiple (as opposed to single) parallel machine Generalized ET problems with non-uniform sequence dependent setup times, job processing times and due times, and earliness/tardiness penalty weights. By classical scheduling classification, this problem is termed $P|r_j, s_{ij}| \sum \alpha_j E_j + \beta_j T_j$ While the single-machine ET problem has been well-studied, **MGET** is relatively new and less regarded. The literature on **MGET** can be broadly divided into two classes. One class formulates **MGET** as mixed-integer programming models. For example, Balakrishnan, Kanet, & Sridharan (1998) solved the general **MGET**, while Zhu & Heady (2000) considered **MGET** with no setup times. Unfortunately, these models are incapable of solving problems with more than 12 jobs and/or 3 machines. Hence, an MIP approach is unlikely

to be practical for large-scale real-world problems. The other class is based on heuristics. Kanet & Sridharan (2000) suggested decomposing the problem into two subproblems: the sequencing problem and timetabling problem. The algorithms will accordingly proceed in two iterative stages: sequencing and timetabling. The sequencing stage generates a semi-schedule specifying which machine a job is assigned to and its execution order on that machine. The timetabling stage completes the schedule by determining the start time for each job on each machine. The results from the timetabling stage will be used by the next sequencing stage to generate a new semi-schedule. The two stages iterate until a satisfying schedule is achieved.

The timetabling subproblem for **MGET** (popularly known as **SEQ**) is polynomial-time solvable. There are numerous extensions to this classical problem, for instance to convex piecewise linear functions. Garey, Tarjan, & Wilfong (1988) developed the first polynomial algorithm for **SEQ** whose time complexity is $O(N \log N)$. We shall name this algorithm as ***SEQ-V1***. Szwarc & Mukhopadhyay (1995) developed another **SEQ** algorithm with time complexity $O(NM_C)$, where M_C is the number of clusters in the schedule and it is no larger than N. We name this algorithm as ***SEQ-V2***. No **SEQ** algorithm is absolutely better than the other. Although both these algorithms run in polynomial time, they can become a computational bottleneck if embedded within an **MGET** algorithmic framework as a base module which is called many times. Hence, it is still desirable to make the timetabling algorithm run as fast as possible.

Sequencing subproblems, on the other hand, are typically strongly NP-hard, since the most basic problem $1||\sum w_j T_j$ is already strongly NP-hard (Lawler (1977)). For large scale problem instances, heuristic approaches have been proposed. Cheng, Gen, & Tosawa (1995) proposed a genetic algorithm to solve **MGET** with no setup times. The semi-schedule is the chromosome. The chromosome has two kinds of symbols: job id and partition symbol. Each chromosome is a permutation of all jobs with $M - 1$ partition symbols inserted. These partition symbols divided the permutation into M subsequences, which were considered as the job sequences on each machine. Radhakrishnan & Ventura (2000) proposed a simulated annealing algorithm for **MGET** with $\alpha_i = \beta_i = 1$, based on job interchange schemes, i.e., selected two jobs and interchange them to generate a neighbourhood solution. This algorithm defined three schemes to perform interchange: "*best preceding jobs*", "*best succeeding jobs*" and adjacent pairs scheme. The best succeeding jobs j of job i are the ones with minimum $|d_i + s_{ij} + p_j - d_j|$. The best preceding jobs j of job i are the ones with minimum $|d_i - s_{ji} - p_i - d_j|$. Kim *et al.*(2002) applied simulated annealing to solve **MGET** where jobs are organized in lots. The setup times only occur between lots, and the jobs are allowed to be assigned to any lot. The neighbourhood schemes include manipulating lots and reassigning job to other lots.

Joslin & Clements (1999) proposed a new meta-heuristic called squeaky wheel optimization (***SWO***) for solving combinatorial optimization problems. They had demonstrated the effectiveness of **SWO** in solving the machine scheduling problem with regular performance measures, which inspired us to extend the work to **MGET**.

In this paper, we propose a provably more efficient **SEQ** algorithm than the ones in the literature. Then we propose a meta-heuristic framework based on **SWO** to solve the sequencing (and thus overall) problem. We will demonstrate the effectiveness of our approach by benchmarking our algorithms against the algorithm proposed by Radhakrishnan & Ventura (2000).

2. A More Efficient Algorithm for SEQ

The **SEQ** problem is concerned with finding an executable schedule on a single machine for a fixed job sequence. Given a sequence of N jobs, $j_1, j_2, \ldots, j_N$, where job j_i must be processed before j_{i+1}, jobs having processing times p_i, due times d_i, unit penalties for earliness α_i and tardiness β_i, find job completion times c_i that minimizes the total penalty. Note that the multi-machine problem can be expressed as independent single-machine problems, each with its own job sequence. Note also that we do not need to consider the sequence dependent setup times since such problems can be converted into problems without setup times, according to Davis & Kanet (1993). Since the job sequence is fixed, **SEQ** algorithms are best described as idle-time-insertion algorithms.

Garey, Tarjan, & Wilfong (1988) and Szwarc & Mukhopadhyay (1995) proposed two different algorithms, **SEQ-V1** and **SEQ-V**2, with time complexity $O(N \log N)$ and $O(NM_C)$ respectively, where M_C is the number of clusters in the original job sequence. Garey, Tarjan, & Wilfong (1988) insert jobs into the schedule one by one, while Szwarc & Mukhopadhyay (1995) first insert all jobs into the schedule and shift them along the time dimension by clusters. By processing more than one jobs at a time as clusters, a more efficient algorithm can be expected. In this paper, we improve the results of Szwarc & Mukhopadhyay (1995) to time complexity $O(N \log M_C)$ by proposing a clever way to manage the clusters.

A *cluster* is a sequence of jobs such that in any optimal schedule, no idle time need to be inserted between them. In other words, we can shift all jobs within the same cluster as an entity. Hence, the entire cluster will be scheduled once the first job has been scheduled. Szwarc & Mukhopadhyay (1995) identifies a sufficient condition for clustering of jobs:

Definition For a sequence of N jobs ($j_1, j_2, \ldots, j_N$), consecutive jobs j_i and j_{i+1} are in the same cluster if $d_{j+1} - d_j < p_{j+1}$.

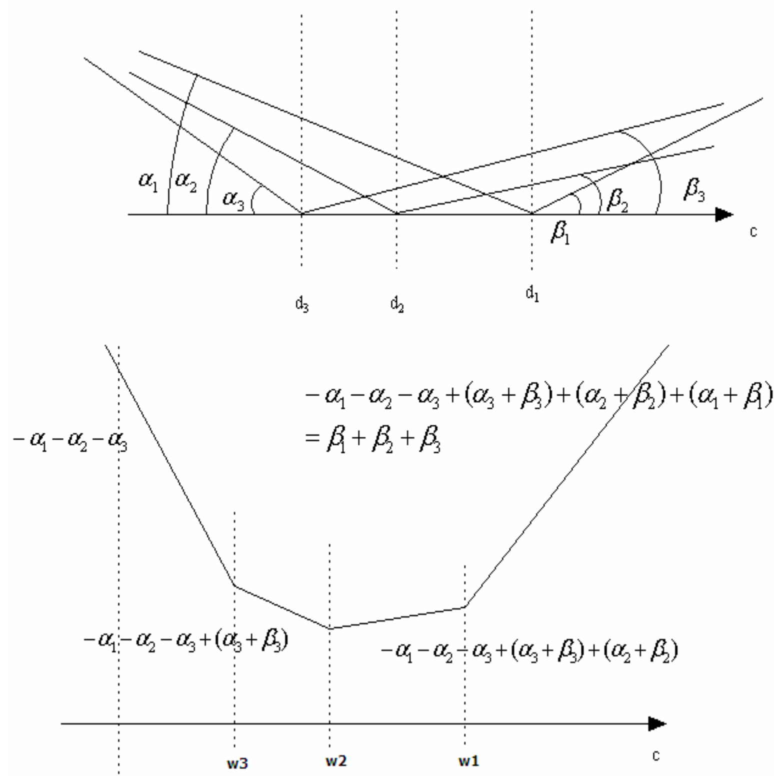

Figure 1. Cluster penalty function

The total penalty of a cluster can be expressed as a function of the completion time of the first job c. Figure 1 shows an example of a 3-job cluster. The upper diagram shows the penalty functions of individual jobs with respect to their completion times. The lower diagram shows the aggregate penalty function of this cluster with respect to c. The function is composed of piecewise-linear line segments, where the slope at any point represents the marginal cost per unit shift of the cluster from that point. Observe that the slope changes only at the points when some job within the cluster is due.

It turns out that, in deciding where to place the first job (and hence the rest of the jobs in the cluster), we only need to consider these limited number of extreme points (Szwarc & Mukhopadhyay (1995)). Let a cluster be denoted by a pair (i, i'), where i and i' are the indexes of the first and last jobs of the cluster respectively. Let $W(i, i')$ denote the set of extreme points (i.e. the set of possible completion times for the first job of cluster (i, i') such that some job within the cluster completes on its due time). More precisely,

$$W(i, i') = \{w_l : w_l = \max(0, d_l - \sum_{q=i+1}^{l} p_q), i \leq l \leq i'\}.$$

There are at most N such extreme points across all clusters. Since the values in $W(i, i')$ are only dependent on the input, this set can be pre-determined, and hence the sorted list W comprising a union of these sets over all M_C clusters can be obtained in O($N \log M_C$) time (via merging M_C sorted sub-lists).

Observe in Figure 1 that the slopes of the lines are non-decreasing as c increases. Let $\Delta^-(c, i, i')$ and and $\Delta^+(c, i, i')$ denote the slope of the line segment to the left and right of $x = c$. Clearly, the optimal position of (the first job of) a cluster is at time point c where $\Delta^-(c, i, i') \geq 0$ and $\Delta^+(c, i, i') \leq 0$. However, from the perspective of global optimality, this cluster may need to start earlier (so that subsequent clusters can incur less penalty). Once a cluster has reached its optimal position, it does not require further shifting since it offers no cost advantage to shift a cluster beyond its optimal position. (Should the subsequent cluster requires to start later, we can always shift that cluster rightwards.)

The strategy of our algorithm is to first schedule the jobs where no left-shifting can reduce cost. Based on the idea of Davis & Kanet (1993), this is equivalent to the situation where each job is at its earliest possible completion time. Then, we perform a plane sweep from left to right, each time right-shifting the first (i.e. leftmost) non-optimal cluster rightwards (positions given by W), and repeat until no such cluster can be found. By Davis & Kanet (1993), we conclude that the solution is optimal. At any point during the algorithm, each cluster is associated with a current completion time point (for its first job). Henceforth, we denote a cluster comprising jobs i to i' by (c, i, i'), where c is the current position of job i.

We define three cluster operations: **shift**, **concatenation** and **consolidation**.

Cluster **shift** shifts the cluster from its current position c to a new position w. It entails the computation of $\Delta^+(w, i, i')$ and $\Delta^-(w, i, i')$, which can be computed in O(1) time using previous Δ values.

Cluster **concatenation** merges two adjacent clusters (c_1, i_1, i'_1) and (c_2, i_2, i'_2) so that they can be treated as one and shifted together thereafter. It entails the

Algorithm 1 SEQ-V3

1 (Initialize) For all $1 \leq k \leq N$, assign $c_k = \sum_{l=1}^{k} p_l$.

2 Cluster the jobs and compute $\Delta^+(c, i, i')$ and $W(i, i')$ for all clusters. For cluster m, define the indexes of the first job and the last job as i_m and i'_m.

3 Perform cluster **concatenation** from the first to the last cluster. Let h denote the first non-optimal cluster.

4 Compute the set W.

5 If $h > M_C$, this algorithm terminates.

6 Find the minimum positive integer $w \in W$. If the corresponding job to w is in some cluster $h' < h$, repeat this step.

7 **Shift** cluster $(i_{h'}, j_{h'})$ from c to w.

8 **Consolidate** cluster $(i_{h'}, j_{h'})$ and update the value of h.

9 Goto step 5.

---

computation of $\Delta^+(c_1, i_1, i'_2) = \Delta^+(c_1, i_1, i'_1) + \Delta^+(c_2, i_2, i'_2)$, which can be computed in O(1) time.

Cluster **consolidation** ensures that each non-optimal cluster (c, i, i') satisfies the property that $\Delta^+(c, i, i') > 0$. This is achieved by three rules:

- If $\Delta^+(c, i, i') > 0$, do nothing.
- If $\Delta^+(c, i, i') \leq 0$ and there is some cluster (c_1, i_1, i'_1) before it, concatenate cluster (c, i, i') and (c_1, i_1, i'_1) and consolidate the new cluster (c_1, i_1, i').
- If $\Delta^+(c, i, i') \leq 0$ and there is no cluster before it, this cluster is optimal. These jobs are marked optimal and exempted from further shifting.

Now we present our algorithm **SEQ-V3** (see Algorithm 1).

Each **concatenation** takes O(1) time (using the set data structure discussed in Tarjan(1983)) and there are at most M_C **concatenations**. Hence, the time complexity of all cluster **consolidations** is $O(M_C)$. One can find the minimum element in O(1) time for each loop. Hence, Steps 5-8 takes $O(M_C + N) = O(N)$ time. Steps $1 - 3$ take $O(N)$ time. Step 4 requires $O(N \log M_C)$ time. Thus, the time complexity of **SEQ-V3** is $O(N \log M_C)$.

Since our algorithm performs cluster concatenation, we define the concatenated clusters as *meta-clusters*. We use the symbol $M(m,n)$ to represent a meta-cluster concatenated from successive clusters

$$(c_{i_m}, i_m, i'_m), (c_{i_{m+1}}, i_{m+1}, i'_{m+1}), \ldots, (c_{i_n}, i_n, i'_n),$$

According to **SEQ-V2** and its proof, **SEQ-V3** is correct if this proposition holds:

Proposition 1 *For any meta-cluster $M(m,n)$, for any k, $m \le k < n$, $\sum_{l=m}^{k} \Delta^+(c, i_l, i'_l) \ge \sum_{l=m}^{n} \Delta^+(c, i_l, i'_l)$.*

We prove the proposition by contradiction. Assume there exists some K, such that

$$\sum_{l=m}^{K} \Delta^+(c, i_l, i'_l) < \sum_{l=m}^{n} \Delta^+(c, i_l, i'_l).$$

Then we must have

$$\sum_{l=K+1}^{n} \Delta^+(c, i_l, i'_l) = \sum_{l=m}^{n} \Delta^+(c, i_l, i'_l) - \sum_{l=m}^{K} \Delta^+(c, i_l, i'_l) > 0.$$

Hence, the **concatenation** of cluster K will never happen, according to the rules of cluster **consolidation**, which contradicts our assumption. Hence, the assumption is not possible and Proposition 1 holds.

3. SWO for Sequencing Problems

Sequencing for **MGET** problems are strongly NP-hard. In this section, we use a new meta-heuristics (Squeaky Wheel Optimization, or **SWO** for short) to solve **MGET** problems.

SWO (Joslin & Clements (1999)) is a relatively new meta-heuristics for solving NP-hard optimization problems. **SWO** algorithms operate in two spaces: ordered list space and semi-schedule space. In Joslin & Clements (1999), the ordered list space is called sequence space. Since we have used the term *sequence in the discussion of timetabling algorithms, we prefer to use the term* ordered list instead. The term originally for semi-schedule space is *solution space. Again, we feel the term* semi-schedule is more precise since we are applying **SWO** to solve only the sequencing subproblem. The reader may observe that the use of alternative search spaces in scheduling is not a new idea (see for example, Storer & Wu & Vaccari (1992) and Herrmann & Lee (1995)). However, the iterative use of 2 search spaces where one provides a feedback mechanism for the other is a relatively novel idea. Particularly, **SWO** has been shown to be effective in scheduling problems with regular performance measures, and in this paper, we apply it to solve **MGET**.

Generally speaking, an **SWO** algorithm is made up of three components: *constructor*, *analyser*, and *prioritizer*:

- *A constructor* that generates an initial solution. In our context, it takes in an ordered list L, and generates a semi-schedule SS based on some greedy algorithm.
- Given an ordered-list L and the corresponding semi-schedule SS, the *analyser* analyzes the "squeaky wheels" that contribute to the poor solution quality. In our context, it computes a value for each job in the schedule. The values are called *blames*.
- The *prioritizer* adjusts the ordered-list L according to the blames of the jobs. The new ordered-list L' will be used as the input to the constructor for next iteration.
- This process repeats until a certain quality of semi-schedule is generated.

In **SWO**, a new semi-schedule is not directly generated from semi-schedules. **SWO** uses ordered lists as an intermediate control mechanism between the generation of two successive semi-schedules. **SWO** focuses on identifying the jobs that mostly affect the quality of semi-schedules and arrange them at proper positions in the ordered lists. Since we construct semi-schedules by picking jobs from the ordered list, much thought goes into how the order list evolves from one iteration to the next:

- The ordered lists can be considered as a prediction of the importance of the jobs. The position of the job reflects its importance.
- The constructor instantiates the prediction by greedily constructing a semi-schedule. The blames are measurements of the importance of the jobs in semi-schedules. We shall consider the blames as feedbacks to the prediction.
- By comparing the predicted importance and real importance of the jobs, we know whether the prediction is successful. And the next prediction will be adjusted according to the feedbacks.

The methodology is shown pictorally in Figure 2.

3.1 Comparison with Other Meta-Heuristics

Tabu search, genetic algorithms, simulated annealing (**SA**) and evolutionary algorithms are widely used meta-heuristics. One problem with these approaches is that they do not have a feedback path from the semi-schedule/solution

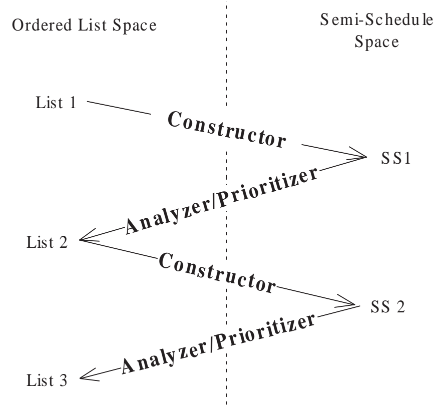

Figure 2. SWO Algorithm Structure

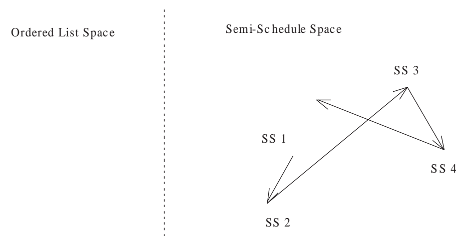

Figure 3. Structure of Direct Generation of New Schedules

space to the ordered list/sequence space. However, depending on the heuristics, the ordering of the lists may not be used in some algorithms.

In those heuristics, there are two ways to generate new semi-schedules: *direct generation* or *indirect generation*. *Direct generation* means that a new semi-schedule is generated by modifying/recombining other semi-schedules. A simple but widely used example is adjacent pair interchange. The whole process of direction generation operates in semi-schedule space, which is shown in Figure 3. Genetic algorithms can be another kind of direct generation scheme, depending on implementations. For genetic algorithms, sometimes the chromosome is represented in the form of semi-schedules. But this may not be necessary for all genetic algorithms.

Indirect generation means that a new semi-schedule is not directly generated from semi-schedules. James & Buchanan (1997) is a good example. The list is composed of whether a job is to be scheduled as an early job or a tardy job. The quality of the list is measured by the total costs of the corresponding schedule. The whole process of generation operates in ordered list space, which is shown

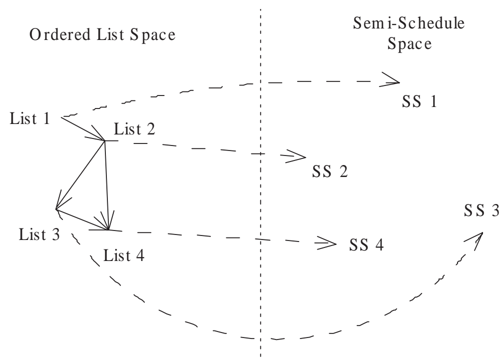

Figure 4. Structure of Indirect Generation of New Schedules

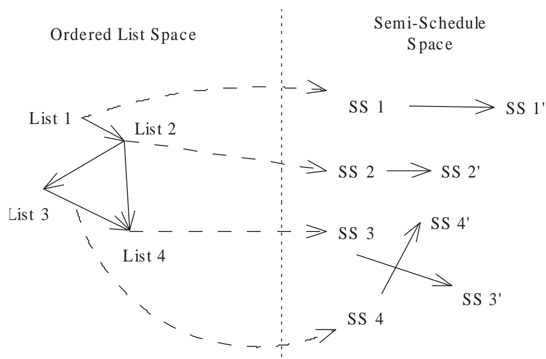

Figure 5. Structure of Combined Generation of New Schedules

in Figure 4. This example is for single machine problems. There is no similar literature for multiple machine problems.

The two generation schemes can be integrated to generate new semi-schedules. For example, the work of James & Buchanan (1997) can be further improved by a neighbourhood search as shown in Figure 5.

In all these three situations, there is no feedback path from the semi-schedule space to the ordered list space. A possible reason may be that such feedback could not be well-defined. The distinctive feature of **SWO** is that it tries to utilize the feedback path from the semi-schedule space to the ordered list space, such that a new list can be generated more effectively. Although the feedbacks are rough, after several iterations, we expect the quality of semi-schedules to be improved.

Algorithm 2 GreedyInsertion(job, schedule)

1 For each machine and each possible insert position, do the following 2 steps

2 Generate the new semi-schedule.

3 Call SEQ-V3 to find the executable schedule and its total cost

4 Pick the schedule with minimum cost as the result, break ties arbitrarily.

Algorithm 3 MGET-SWO Algorithm

1 Generate a random job permutation **L**

2 Construct a schedule **SS** from **L** by calling GreedyInsertion

3 If termination condition is satisfied, STOP

4 Analyser sets the blames for each job according to **SS**

5 Prioritizer reorders the jobs according to the blames of the jobs, and generates a new permutation **L'**

6 Go to step 3.

4. MGET-SWO

We now discuss some design issues of **MGET-SWO**.

Before presenting the SWO algorithm, we first discuss a greedy algorithm (Algorithm 2) that inserts a job to an existing semi-schedule.

Algorithm 3 shows our overall SWO algorithm.

In Step 2, the Contructor greedily inserts jobs into the semi-schedule. In inserting each job, we attempt all possible machines and positions without changing the relative order of jobs already in the semi-schedule. The insertion point is the one that minimizes total cost.

Note that jobs are inserted in the order according to the permutation **L**. Intuitively, jobs inserted early are at a ”disadvantage” in the sense that latter jobs may supercede its position through subsequent insertions. Hence, ideally, we wish to insert jobs early if they are relatively *insensitive* to their positions in the semi-schedule - in the sense that their positions in the schedule will not affect the total cost by a large amount. In other words, if a job is insensitive, it should be put to an earlier position in the list, and vice versa.

With the above intuition, we design the Analyser as follows. The Analyser performs *job reinsertion* to determine the sensitiveness of jobs to positions.

Algorithm 4 Implementation of Analyzer

1 i=1; S is the schedule to be analyzed.

2 While $i \leq N$ do the following:

3 Remove the job i in **L** from S

4 S=GreedyInsertion(job i, S)

5 Set the blame of job i as the cost reduced by the above reinsertion

6 $i = i + 1$

Job reinsertion means that a job is removed from the current semi-schedule, and reinserted into the semi-schedule by selecting a position giving lowest total costs with the relative positions of other jobs unchanged. The reduction of total costs brought by job reinsertion is measured as its sensitiveness, or *blames*. Notice that the blames cannot be negative. If the blame is small, we consider the job insensitive to its order in the list, and vice versa.

Hence, the Analyzer works as follows (see Algorithm 4): the blame of each job is computed as the total cost reduced as a result of the job reinsertion. Notice that when a successful reinsertion is found, we will use the improved schedule as the base for future reinsertions, until an even better schedule is found. Essentially, we perform a local search in the analyser algorithm.

Finally, the Prioritizer sorts all jobs in non-decreasing blame order. In doing so, the sensitive jobs are put to later positions of the next list generated.

5. Experimental Results

In this section, we present three sets of experiments to compare the performances of **SEQ-V1**, **SEQ-V2** and **SEQ-V3**. The test data is generated according to the scheme in Szwarc & Mukhopadhyay (1995). p_j is generated from uniform distribution in a range $[1, 100]$. d_j is generated from uniform distribution in a range $[a \times \sum p_j, b \times \sum p_j]$. $a - b$ can be one of the six pairs of values: $0.1 - 0.9$, $0.2 - 0.8$, $0.3 - 0.6$, $0.1 - 1.3$, $0.1 - 1.7$, $0.1 - 2.1$. There are four choices for N: 100, 200, 400 and 500. We use three sets of sequences: **increasing**, **decreasing** and **random** sequences. For **increasing** sequences, the jobs are sorted in increasing d_j order. For **decreasing** sequences, the jobs are sorted in decreasing d_j order. For **random** sequences, the jobs are arranged randomly. The time measured is averaged for 5000 runs.

In Table 1, the results are summarized. The numbers are generated by this scheme. For the same problem, we measure the three times: t_1 as the average runtime of **SEQ-V1**, t_2 as the average runtime of **SEQ-V**2, t_3 as the average

N	a-b	Increasing	Decreasing	Random
100	0.1-0.9	13.68 %	-3.84 %	13.37 %
	0.2-0.8	58.29 %	8.18 %	46.93 %
	0.3-0.6	2.84 %	16.11 %	15.13 %
	0.1-1.3	26.68 %	19.59 %	48.00 %
	0.1-1.7	0.00 %	26.99 %	39.96 %
	0.1-2.1	-4.86 %	22.16 %	41.39 %
200	0.1-0.9	18.28 %	21.84 %	18.43 %
	0.2-0.8	16.67 %	17.90 %	19.07 %
	0.3-0.6	15.28 %	16.62 %	16.59 %
	0.1-1.3	29.16 %	20.77 %	41.82 %
	0.1-1.7	10.30 %	25.05 %	38.17 %
	0.1-2.1	6.44 %	28.10 %	29.72 %
400	0.1-0.9	64.22 %	25.88 %	8.32 %
	0.2-0.8	61.65 %	27.74 %	21.05 %
	0.3-0.6	24.27 %	25.02 %	21.27 %
	0.1-1.3	27.58 %	32.44 %	44.73 %
	0.1-1.7	22.04 %	34.09 %	37.61 %
	0.1-2.1	10.48 %	36.05 %	38.00 %
500	0.1-0.9	20.86 %	32.00 %	52.39 %
	0.2-0.8	72.63 %	34.85 %	15.82 %
	0.3-0.6	23.23 %	23.90 %	21.33 %
	0.1-1.3	30.82 %	31.20 %	44.95 %
	0.1-1.7	14.80 %	33.54 %	39.57 %
	0.1-2.1	-0.18 %	35.85 %	36.35 %

Table 1. Improvement in runtime of SEQ-V3

runtime of **SEQ-V**3. The improvement is calculated as follows:

$$100\%(1 - \frac{t_3}{\min\{t_1, t_2\}})$$

which is the percentage of time savings compared to the best results from **SEQ-V1** and **SEQ-V2**.

The results indicate that for most of the cases, **SEQ-V3** is the fastest. And for many of the cases, the runtime is reduced by a big amount (up to 72.63%). There are a few cases that **SEQ-V3** is slower than the best performer of **SEQ-V1** and **SEQ-V2**. But in the worst case, **SEQ-V3** takes only 4.86% extra time than the best performer, which is not a big difference. Compared to **SEQ-V1** and **SEQ-V2**, **SEQ-V3** requires quite a mount of data preprocessing. Thus, in certain cases, **SEQ-V3** may not be the fastest. In general **SEQ-V3** is a better choice for **SEQ** problems.

MGET-SA is the simulated annealing algorithm proposed by Radhakrishnan & Ventura (2000). Their algorithm does not perform well. We have made these modifications to their original algorithm, to get a new algorithm **MGET-SA2**:

- Their adjacent pair interchange algorithms have cycling problems, i.e., the algorithm may repeat a few swaps continuously. We solved this problem by allowing the swap of any two jobs once only, i.e., when applying the adjacent pair interchange algorithm, each swap is recorded such that all future swaps of the same pair of jobs will be disallowed.
- The algorithm's performance can be improved by adding a random pair interchange algorithm as local search.

The algorithm **MGET-SWO** is the **SWO** algorithm we have presented.

The set of test problems are generated from the original data generation scheme used by Radhakrishnan & Ventura (2000): p_j and s_{ij} are generated from uniform distribution on $[1, 20]$. d_j is generated from uniform distribution on $[0.225 \times \sum p_j, 0.275 \times \sum p_j]$.

The results can be summarized in Table 2. Note that the values in the table are the average results of 30 runs. We can see that **MGET-SA2** improved **MGET-SA**, and **MGET-SWO** performs better than simulated annealing algorithms.

6. Conclusion and Future Works

In this paper, we made two contributions on the parallel machine scheduling problem with earliness and tardiness penalties as follows:

- We distinguished **SEQ-V1** and **SEQ-V2** as different schemes to manage clusters. We analysed the strengths and weaknesses of **SEQ-V1** and

		MGET-SA		MGET-SA2		SWO	
M	N	Costs	Time	Costs	Time	Costs	Time
2	10	189.933	0.76	183.767	0.988	189.433	0.0287
5	15	114.267	0.264	109.2	0.248	104.967	0.0761
10	50	455.267	46.56	398.1	43.88	397.733	1.276
15	80	680.533	104.2	525.367	95.2061	484.367	3.839
13	297	No Improvement. Initial Cost=195				96.133	406.433

Table 2. The Performance Comparison of algorithms for **MGET**

SEQ-V2, and proposed a more efficient **SEQ** algorithm that combined the strength of **SEQ-V1** and **SEQ-V2** that performs fastest both analytically and experimentally.

- We demonstrated the effectiveness of **SWO** on this problem. While **SWO** is a promising meta-heuristics for NP-hard optimization problems, we like to point out that applying **SWO** on other problems may present these challenges (limitations):
  - SWO structure does not imply efficiency, but the possibility of an efficient algorithm. The main difficulty lies in the design of a greedy algorithm and a proper importance recognition scheme, which is not a trivial pursuit.
  - The variance of the solutions can be fairly large. For our algorithms, the standard deviation of total costs is 4% of the average costs.

Contrary to problems with regular performance measures that have extensive literature, results for **ET** problems are limited. Most of the works are meta-heuristics based, which lack mathematical rigors. It remains to see if the results for regular performance measures can be extended to **ET** performance measures.

The current dominating factor of runtime of **SEQ** algorithms is the sorting of all $W(i_l, j_l)$. The current scheme presorts all values in the set $W = \bigcup W(i_l, j_l)$. We should notice that the presorting of all numbers is not always necessary. On average, the schedule will become optimal before W becomes empty, which means that part of the presorting is not necessary. Since we do not know the exact number of values that will be used, we conjecture that a smarter scheme can perform sorting only when necessary thereby reducing the time complexity further.

References

Balakrishnan, N., Kanet, J. J., and Sridharan, S. V. 1998. Early/tardy scheduling with sequence dependent setups on uniform parallel machines. *Computers and Operations Research* 26:127–141.

Cheng, R., Gen, M., and Tosawa, T. 1995. Minmax earliness/tardiness scheduling in identical parallel machine system using genetic algorithm. *Computers and Industrial Engineering* 29(1-4):513–517.

cnet.com 2001. http://news.cnet.com/news/0-1004-200-5867112.html

Davis, J. S., and Kanet, J. J. 1993. Single-machine scheduling with early and tardy completion costs. *Naval Research Logistics* 40:85–101.

Garey, M. R., Tarjan, R. E., and Wilfong, G. T. 1988. One-processor scheduling with symmetric earliness and tardiness penalties. *Mathematics of Operations Research* 13:330–348.

Herrmann, J.W., and C.-Y. Lee. 1995. Solving a class scheduling problem with a genetic algorithm, *ORSA Journal of Computing*, 7(4):443-452.

James, R., and Buchanan, J. 1997. A neighbourhood scheme with a compressed solution space for the early/tardy scheduling problem. *European Jnl of Oper. Res.* 102(3):513–527.

Joslin, D., and Clements, D. 1999. Squeaky wheel optimization. *Journal of Artificial Intelligence Research* 10:353–373.

Kanet, J. J., and Sridharan, V. 2000. Scheduling with inserted idle time: Problem taxonomy and literature review. *Operations Research* 48(1):99–110.

Kim D. W., Kim, K.-H., Jang, W., and Chen, F. F. 2002. Unrelated parallel machine scheduling with setup times using simulated annealing. *Robotics and Computer Integrated Manufacturing* 18:223–231.

Lawler, E. L. 1977. A ”pseudopolynomial” algorithm for sequencing jobs to minimize total tardiness. *Ann of Discrete Math* 1:331–342.

Radhakrishnan, S., and Ventura, J. A. 2000. Simulated annealing for parallel machine scheduling with earliness-tardiness penalties and sequence-dependent set-up times. *International Journal of Production Research* 38(10):2233–2252.

Storer, R.H., S. Wu, and R. Vaccari. 1992. New search spaces for sequencing problems with application to job shop scheduling, *Operations Research* 39(10):1495-1509.

Szwarc, W., and Mukhopadhyay, S. K. 1995. Optimal timing schedules in earliness-tardiness single machine sequencing. *Naval Research Logistics* 42(7):1109–1114.

Tarjan R. E. *Data Structures and Network Algorithms*. Society for Industrial and Applied MAthematics, Philadelphia, PA, 1983.

Zhu, Z., and Heady, R. B. 2000. Minimizing the sum of earliness/tardiness in multi-machine scheduling: a mixed integer programming approach. *Computers and Industrial Engineering* 38:297–305.