December 2018

Heuristic Algorithm to Minimize Total Weighted Tardiness on the Unrelated Parallel Machine with Sequence Dependent Setup and Future Ready Time

Tzu Yi Li
University of Wisconsin-Milwaukee

HEURISTIC ALGORITHM TO MINIMIZE TOTAL WEIGHTED TARDINESS ON THE UNRELATED PARALLEL MACHINE WITH SEQUENCE DEPENDENT SETUP AND FUTURE READY TIME

by

Tzu Yi Li

A Dissertation Submitted in

Partial Fulfillment of the

Requirements for the Degree of

Master of Science

in Engineering

at

The University of Wisconsin-Milwaukee

December 2018

ABSTRACT

HEURISTIC ALGORITHM TO MINIMIZE TOTAL WEIGHTED TARDINESS ON THE UNRELATED PARALLEL MACHINE WITH SEQUENCE DEPENDENT SETUP AND FUTURE READY TIME

by

Tzu Yi Li

The University of Wisconsin-Milwaukee, 2018

Under the Supervision of Professor Jaejin Jang

This study presents a heuristic algorithm to minimize total weighted tardiness on unrelated parallel machines with sequence-dependent setup time and future ready time. We propose a new rule based on Apparent Tardiness Cost (ATC). The performance of the rule is evaluated on unrelated parallel machines. In order to solve a problem, we use a look-ahead method and a job-swap method. When a machine becomes idle, the heuristic compares the jobs on the machine and selects the one with the smallest total tardiness value to carry out a process.

The propose heuristic is divided into three stages: The first stage employs the newly introduced dispatching rule, ATC with continuous setup and ready time for unrelated parallel machines (ATCSR_UP), along with a look-ahead heuristic to select the initial job for each machine. The second stage, consisting of several iterations, schedules the rest of the job on the machine. Each iteration starts by finding the job with the smallest tardiness. The ATCSR_Rm rule proposed by Lin and Hsieh (2013) concerns the unrelated-parallel-machine scheduling which this study examines, so we compare our ATC-based rule with their proposed rule. Although they study a separable setup time in their research, no other paper than Lin and Hsieh (2003) focus on unrelated parallel machine with future ready times. In their WSPT term, they

consider the processing time for each job; our own rule considers processing time, setup time, job ready time, and machine time. We consider the setup time, job ready time, and machine time because — according to the continuous sequence-dependent setup rule — setup time should be included in processing time (Yue and Jang 2013). In addition, job ready time and machine time should also be included in the processing time. Adding setup time $(s_{i,j})$, job ready time (r_j), and machine time (t_m) to the formula thus makes the formula more accurate.

Lin and Hsieh (2013) use $max(r_j, t_i + s_{i,j})$ for the slack term, and they compare the ready time with the sum of the machine available time (t_i) and the setup time $(s_{i,j})$. However, in our formula, we consider ready time, machine time, and current time. Current time (t) is used when a job might come at a future time when the machine in question is idle or has finished the job.

The last term of the propose heuristic is the ready term, which uses both ready time (r_j) and machine time (t_m), because it needs to specify whether ready time (r_j) or machine time (t_m) goes first. If a job is ready to be processed but the machine is not ready, the job has to wait. We use ready time (r_j) and machine time (t_m) because this makes the formula more suitable for practical, real-world use.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGEMENTS

There are several people I would like to acknowledge their contributions toward to this paper.

First of all, I would like to thank is my advisor, Dr. Jaejin Jang, of university of Wisconsin-Milwaukee. The door to Dr. Jang office was always open whenever I ran into a trouble spot or had a question about my research or writing. Without his passionate participation and input, the validation survey could not have been successfully conducted.

Thanks to the committee members Dr. David Yu and Dr. Otieno for their useful advice on my research. Finally, I must express my very profound gratitude to my parents and my lab mates for providing me with unfailing support and continuous encouragement throughout my years of study and through the process of researching and writing this thesis. This accomplishment would not have been possible without them. Thank you.

CHAPTER 1. INTRODUCTION

Parallel-machine scheduling (PMS) is commonly encountered in industries involving machinery, electronics, textiles, transportation, telecommunications, pharmaceuticals, chemicals, and services (Lu, Lin, & Ying, 2011). Since the publication of the first paper about parallel-machine scheduling, researched by McNaughton (1959), various PMSs have attracted great interest among other researchers.

Based on the characteristics of parallel machines, a classical parallel-machine system (PMS) can be categorized as identical, uniform, or unrelated (Cheng & Sin.C, 1990). Allahverdi and Mittenthal (1994) group parallel machines into three groups: *identical parallel machines*, for which the processing time of a job is the same for all machines; *uniform parallel machines*, for which the processing time of a job is determined by the speed factor of the machine used; and *unrelated parallel machines*, for which the processing time of a job can vary in an arbitrary way depending on the machine selected. This paper is concerned with unrelated parallel machines.

According to Chuang and Chang (2014), it is common for newer machines to have faster processing speeds in technology manufacturing. Shahid-Zadeh, Tavakkoli-Moghaddam, Taheri-Moghadam, and Rastgar (2017) and Rasaratnam (2007) consider industrial scheduling to have benefited greatly from the use of unrelated parallel machines, due to these machines' ability to perform the same function but with varying capability or capacity.

Most research on parallel-machine scheduling assumes that the setup time of a job can be either ignored or included in processing time. However, this assumption is appropriate only when the setup time is independent from the job sequence. According to research by Panwalkar, Dudek and Smith (1973), Krajewski, King, Ritzman, and Wong (1987), and Black and Kohser (2017), approximately 70% of schedulers observed that for about 25% of jobs they scheduled, the setup time could not be ignored. In this sense, setup time can be considered an important factor for controlling a whole manufacturing system. Sequence-dependent setup, in which the length of a job's setup time depends on the immediately preceding job, is common and often prevalent in manufacturing.

Examples of sequence-dependent setup are found in metallurgical industries; in back-end semiconductor manufacturing and semiconductor wafer dicing; and at petroleum production plants, printing plants, car-spraying facilities, and textile-dying plants. (Arroyo *et al*., 2009; Kia *et al*., 2017; Luo *et al*., 2006; Kim *et al*., 2002, 2007; Zhang *et al*., 2007).

Literature on the achievement of parallel-machine scheduling focuses mostly on flow time or on tardiness – for example, on minimizing the maximum tardiness, total weighted flow time, and total weighted tardiness (Drieel & Monch, 2011; Kim et al., 2002; Lee & Pinedo, 1997; Xi & Jang, 2012). Flow time is often closely related to job waiting time for factory processing, whereas tardiness is often relevant to manufacturing when the deadlines for expected due dates cannot be adhered to. Such deadlines matter in the real world, since failure to meet them may lead to the loss of future customers. Therefore, this research is interested in how deadlines can be met and job tardiness reduced in the context of unrelated parallel machines.

Although many researchers have studied parallel machines scheduling problems, only a few papers have dealt with scheduling unrelated parallel machines with setup times, ready times and due date-related measures. In this particular study, we focus on unrelated parallel-machine scheduling using ATC-based dispatching rule. It is known that ATC (Apparent Tardiness Cost)- based rules can efficiently reduce tardiness of a schedule. In this research, a new, efficient ATC-based rule is proposed. All proposed heuristic methods mentioned in this research are for minimizing total weighted tardiness.

This study is organized as follows: Chapter Two reviews literature related to parallel-machine scheduling with sequence-dependent setup. In Chapter Three, a new ATC-based rule is introduced to minimize the total weighted tardiness. Chapter Four is an experimental study carried out on unrelated parallel machines, and it evaluates the performance of the new ATC-based rule proposed. Finally, conclusions and future research are discussed in Chapter Five.

CHAPTER 2. LITERATURE REVIEW

2.1 Parallel machines

This section reviews the studies of parallel machines scheduling both with and without a sequence-dependent setup. From the literature, research about parallel-machine scheduling can be grouped by various criteria (Allahverdi, 2015). The first is by machine type, where machines are sorted into identical, uniform, and unrelated machines. A second division is by setup time (Slotnick, 2011), where machines-scheduling studies are divided into those without a setup time, those with a setup time, those with a continuous setup time (setup cannot start before an operation's ready time), and those with separable setup times (setup can start before an operation's ready time). A further grouping is by objective (Pinedo, 2016), where related papers are classified such as minimizing the total completion time (Costa, Cappadonna, & Fichera, 2016), and minimizing the total tardiness (Allahverdi & Aydilek, 2015). Finally, papers can be grouped by approach: for example, metaheuristic, mathematical models, and branch and bound heuristic. In this study, we focus on non-batch, unrelated parallel machines with continuous sequence-dependent setup times and a total weighted tardiness objective.

2.1.1 The scaling parameter k_1, k_2, k_3

The basic format of the ATC-based indexes has four terms. Jobs are scheduled based on this index value assigned to it.

$$\text{Index (i, j, m)} = (\text{WSPT}) \times (\text{Slack}) \times (\text{Setup time}) \times (\text{Ready time}) \quad (1)$$

$$Index\ (t,i,j) = A \times exp\left(-\frac{B}{k_1 C}\right) \exp\left(-\frac{s_{ij}}{k_2 \bar{s}}\right) \exp\left(-\frac{D}{k_3 \bar{p}}\right) \quad (2)$$

The exponent of terms have scale parameters k_1, k_2, k_3 and job information (processing time, setup time, due date, job weight and ready time) A is the modified WSPT term, B represent the numerator of slack term, C represents the denominator of slack and D is the ready-term numerator. When applying ATC- based rules it is important to determine good scaling parameters (the k values). The formula for index does not specify how the scaling parameters are used, therefore, researchers focuses on different methods to determined the suitable scaling parameters.

In literature, two types of approaches have been used to determine the parameters. Xi and Jang (2012) report that there are several methods that can be used to estimate good grid of the parameters: Rachamadugu and Morton (1982) and Vepsalainen and Morton (1987) use the Empirical Value Method; Lee and Pinedo (1997), Pfund *et al*. (2008) for the Regression Method and Kim et al. (1995), Park et al. (2000) for the Artificial Neural Network Method.

Other research, like that of Pfund et al. (2008) and Drieel and Monch (2009, 2011), investigates the use of many grids and selects the optimal pair of parameters for the grid method. Xi and Jang (2012) point out that the grid method is not only used to decide final parameters but also to give information for other methods like the regression method and other heuristics that get an improved result (Christoph et al., 2007, Drieel and Monch, 2009, 2011)

Lee and Pinedo (1997) present a grid parameter setting:k_1= (0.2, 0.4, 0.6..., 6.4), and k_2 = (0.1, 0.2, 0.3..., 1.6), while later Pfund et al. (2008) propose a wider search range for $k_1$and k_2 and further put forward settings for a newly-created parameter k_3:

k_1: 0.2,0.6,0.8,1,1.2,1.4,1.6,1.8,2,2.4,2.8,3.2,3.6,4,4.4,4.8,5.2,5.6,6,6.4,6.8,7.2

k_2: 0.1,0.3,0.5,0.7,0.9,1.1,1.3,1.5,1.7,1.9,2.1

k_3: 0.001,0.0025,0.004,0.005,0.025,0.04,0.05,0.25,0.4,0.6,0.8,1,1.2

Drieel and Monch (2009) propose a narrower search range k_1: [0.2, 6], k_2 : [0.1, 1.9] and k_3 : [0.001, 1.2] and consider fewer grids: 7, 4 and 5 grids. In further research, they use k_1: [0.01, 1.5], k_2 : [0.1, 1.9] and k_3: [0.01,1] as a search set (Drieel & Monch, 2011).

In this study, we employ the parameters set out by Pfund *et al*. (2008) to evaluate the proposed ATC-based rule. Table 1. shows the parameters propose by Pfund *et al*. (2008) and Drieel and Monch (2011).

Table 1. Grid setting comparison Xi and Jang (2012)

	Pfund et al. (2008)	Driessel and Monch (2009)	Driessel and Monch (2009)
k_1	22 values in [0.2, 7.2]	7 values in [0.2, 6]	5 values in[0.01, 1.5]
k_2	11 values in [0.1, 2.1]	4 values in[0.1, 0.9]	4 values in[0.1, 1.9]
k_3	12 values in[0.001, 1.2]	5 values in[0.001, 1.2]	4 values in[0.01, 1.0]

2.1.2 Objectives of parallel machine scheduling with sequence-dependent setup

(i) Minimizing total weighted tardiness

Scheduling problems with the objective of total weighted tardiness is NP-hard even for a single machine environment (Du & Leung, 1990). As the single machine was seen as NP-hard, the parallel machine problem is also viewed as such. Parallel-machine scheduling with a tardiness objective has attracted much attention from researchers in recent years.

To minimizing the total weighted tardiness in unrelated parallel machines scheduling, Luh, Hoitomt, Max, & Pattipati (1990) propose a Lagrangian dual relaxation approach to generate a listing of jobs followed by a greedy approach to form a feasible schedule for the parallel machines. Arkin and Roundy (1991) consider that the weight of each job is proportional to its processing time and developed a pseudo-polynomial time algorithm to minimize total weighted tardiness.

Lee, Bhaskaran and Pinedo (1996) extend the ATC rule to develop a composite-dispatching heuristic called the ATCS (Apparent Tardiness Cost with Setup) rule for solving the sequence-dependent version of the weighted tardiness problem with static job releases and static machine availability. Lee and Pinedo (1997) use the ATCS dispatching rule, proposed by Lee *et al.* (1996), in order to obtain an initial solution for minimizing the total weighted tardiness on identical machines.

Zhu and Heady (2000) propose a mathematical model for setup times that has to satisfy the triangular inequality law, with the setup starting before the job arrives. Ecom, Shin, Kwun, Shim and Kim (2002) put forward a three-phase heuristic for the problem, using a combination of pre-processing by the first due date, grouping and sequencing jobs according to setup types (improved by a Tabu search), and allocating jobs to machines.

Kim *et al*. (2002) propose a simulated annealing-based heuristic for jobs with sequence-dependent setup times on machines. For the same objective, Kim et al. (2002) extend their

investigation on batch scheduling. Liaw *et al.* (2003) propose a two-phase heuristic for solving the job scheduling problem.

Bilge *et al.* (2004) use a Tabu Search (TS) to schedule uniform parallel machines with sequence-dependent setups and ready times in order to minimize total tardiness $\left(Q|r_j, s_{ijk}|\ T_j\right)$. Kim et al. (2007) report on Tabu search use and on simulated annealing (SA) algorithms to minimize total tardiness for uniform parallel machines with sequence-dependent setup times and ready times $\left(Q|r_j, s_{jk}|\ T_j\right)$. Anghinolfi and Paolucci (2007) present a hybrid metaheuristic approach which combines several features from a Tabu Search, Simulated Annealing, and a variable neighborhood search (VNS) for the $R|r_j, s_{ijk}|\sum w_j T_j$ problem. Logendran, McDonell and Smucker (2007) propose four heuristics for initial solutions and six search algorithms based on Tabu Search for minimizing total weighted tardiness (TWT) on unrelated parallel machines with ready times and sequence-dependent setup times ($R|r_j, s_{ijk}|\sum w_j T_j$).

Gharhgozli et al. (2009) apply a fuzzy-mixed-integer goal programming model to simultaneously minimize total weighted flow time and TWT. They apply this to parallel machine scheduling problems with sequence-dependent setup times and release dates.

Drieel and Monch (2011) propose several VNS approaches for scheduling identical parallel machines with sequence-dependent setup times, precedence constraints and ready times to minimize TWT ($P|r_j, s_{ij}|\sum w_j T_j$). They employ a dispatching rule that uses Apparent Tardiness Cost with setup time and ready time (ATCSR) as an initial solution and then applied

VNS that are fast enough to serve as high-quality sub-problem solution procedures in decomposition schemes for large-scale job shops.

Xi and Jang (2012) study the ATC-based dispatching rules with future-ready times and sequence-dependent setup times for minimizing TWT on identical parallel machines ($P|r_j, s_{ij}|\sum w_j T_j$). They show that the both ATCRCS (continuous sequence setup) and ATCRSS (separable sequence setup) outperform ATCSR (ATC-based rule with sequence setup time and future ready job).

Lin and Hsieh (2013) study the ATC-based dispatching rule with separable setup and future ready job for minimzing the TWT on unrelated parallel machines($P_m|r_j, s_{ij}|\sum w_j T_j$). They modify the ATC-based rule (ATCSR_Rm) from ATCSR so it can be use to solve unrelated parallel machine problems. They show that ATCSR_Rm outperforms ATCSR (ATC-based rule with sequence setup time and future ready job).

Diana *et al*. (2018) address the total weighted tardiness minimization problem on unrelated parallel machines with sequence dependent setup times and job ready times. The problem consists of scheduling a set of jobs in such a way as to reduce penalty costs caused by failures to meet the job due dates. In order to resolve this, an ILS-VND hybrid metaheuristic is proposed: A local search heuristic, Variable Neighborhood Descent (VND), is integrated with an Iterated Local Search (ILS) metaheuristic with multiple restarts.

(ii) Minimizing completion time

Weng, Lu, and Ren (2001) research minimization of the total weighted completion time on unrelated parallel machines. Several heuristics are presented. According to their experiment, the best heuristic assigns one job at a time based on the ratio of a job's processing time plus setup time to its weight. Cochran *et al.* (2003) have published a multi-population genetic algorithm that minimizes the makespan and TWT for identical parallel machines with release times and setup times ($P|r_j, s_{jk}|Cmax, \sum w_j T_j$). Using this hybrid approach, the genetic algorithm assigns jobs to machines; dispatching rules are then used to schedule jobs on each individual machine.

(iii) Minimize maximum lateness

Schutten and Leussink (1996) present a branch and bound algorithm (B&B) to solve the problem of parallel-machine scheduling with release dates and family setup times to minimize maximum lateness ($P|r_j, s_i|Lmax$). Their B&B algorithm can find optimal solutions for up to three machines and 25 jobs. Later on, Balakrishnan *et al.* (1999) propose a MIP model for the problem of early/tardy scheduling with sequence-dependent setups on uniform parallel machines ($Q|r_j, s_{ijk}|(e_j E_j + t_j T_j)$).

Ying and Cheng (2010) apply an iterated greedy (IG) heuristic for minimizing maximum lateness on identical parallel machines with ready times and sequence-dependent setup times ($P|r_j, s_i|Lmax$). Lee *et al.* (2010) present a restricted SA for the ($P|r_j, s_{jk}|Lmax$) problem as well. Moreover, Lin *et al.* (2011) propose an IG heuristic with a sinking temperature for the ($P|r_j, s_i|Lmax$) problem.

(iv) Minimizing number of tardy jobs

Chen (2012) presents an Integer Hybrid Metaheuristic (IHM) which integrated the principles of variable-neighborhood descent and TS for unrelated parallel machines problems with ready times and sequence- and machine-dependent setup times to minimize the weighted number of tardy jobs ($R|r_j, s_{ij}|\sum w_j T_j$).

In Table 2, we list the papers that focus on parallel non-batch machines with sequence-dependent setup time. We categorize them with six objective functions: total weighted tardiness, total tardiness, total completion time, total weighted completion time, bi-criteria and others. We then list the machine type for each paper and the approach the researcher uses to solve the problem.

Table 2. Parallel non-batch machine with sequence-dependent setup time

Objective function	References	Machine type	Approach
1. Total weighted tardiness	*Lee and Pinedo (1997)*	Identical parallel machines	Dispatching rule (ATCS)
	Lee and Pinedo (1997)	Identical parallel machines	Three stages method
	Park et al. (2000)	Identical parallel machines	Dispatching rule (ATCS)
	Fowler and Horng (2003)	Identical parallel machines	Hybrid genetic algorithm
	Tamimi and Rajan (1997)	Uniform parallel machines	Genetic Algorithm
	Xi and Jang (2012)	Identical parallel machines	Dispatching rules (ATCS)
	Diana et al. (2018)	Uniform parallel machines	ILS-VND hybrid metaheuristic
2. Total tardiness	*Chen (2009)*	Unrelated parallel machines	Hybrid Approach (ATCS+SA)
	Lee (2017)	Identical parallel machines	Random iteration greedy metaheuristic
3. Total completion time	*Felipe (2005)*	Unrelated parallel machines	Constructive method
	Kurt and Askin (2001)	Identical parallel machines	Integer programming
	Yin et al. (2018)	Unrelated parallel machine	Greedy heuristic
	Framimam and Paz (2017)	Identical parallel machines	Constructive method
4. Total weighted completion time	*Weng et al. (2001)*	Unrelated parallel machine	Evaluate several heuristics
	Fowler and Horng (2003)	Identical parallel machines	Hybrid genetic algorithm
5. Bi-Criteria			
Sum of weighted earliness and weighted tardiness	*Balakrishnan et al. (1999)*	Uniform parallel machine	Mixed integer programming
	Radhakrishnan and Ventura (2000)	Identical parallel machines	Mathematical programming

	Feng and Lau (2005)	Identical parallel machines	Meta-heuristic
Sum of earliness and tardiness	*Heady and Zhu(1998)*	Identical parallel machines	Heuristic
6. Others			
Weighted number of jobs that are completed at theirs due date	*Hirashi et al. (2002)*	Identical parallel machines	
Minimize the maximum lateness	*Kim et al. (2002)*	Identical parallel machines	Restricted Tabu search
Minimize mean completion time	*Michael et al. (2001)*	Identical parallel machines	Hybrid genetic algorithm
Minimize total setup time	*Anglani et al. (2005)*	Identical parallel machines	Fuzzy programming

2.2 Look-ahead method

The purpose of the look-ahead method is to control system to "look-ahead" to what is happening in the future of the schedule generation (Jang *et al.*, 2001). The decision-making base on this type of information allows the scheduler to make a decision indicating the particular system. In this section the look-ahead method we list are mainly divide in to two types. One type can select a part from those waiting for the machine service. Another type of look-ahead looks ahead to the future states of a production shop and uses the information obtained by looking a few steps ahead to see the current decision's effects on the flow time or other performance evaluation criteria of the parts due to arrive next.

Mao and Kincaid (1994) propose a one-step look-ahead method for machine scheduling and combine this with ATCS and Simulated Annealing. This approach outperforms other normal Simulated Annealing heuristics. In their look-ahead method, Mao and Kincaid (1994)

assume the machine is idle at time *t* and at *t* the available queue is Q_1, containing jobs J_1 to J_l ; Q_2 is the look-ahead queue containing job (J_{l+1}). Job (J_{l+1}) is the job that will be available soonest in the future. When there is more than one job due to arrive at r_{l+1}, the job with the shortest processing time will be placed in Q_2. When the machine becomes idle, Q_1 and Q_2 are checked to decide whether it is best to wait until the job in Q_2 arrives, or to schedule the shortest job in Q_1. The scheduler always chooses the option that yields a shorter total completion time based on the assumption that no more jobs will arrive expect the one already in Q_2.

Christos and Milton (1998) reduce interference for one operator of parallel machines. The look-ahead method they propose is to avoid the simultaneous demands for the server. They assume *i* machine with the completion time. When one machine finished current job, a selection must be made for the next job to process from the jobs in the queue in front of each machine. The job should be select such that simultaneous demands for the server with the next machine *j* to finish processing ($i = j$) can be avoid. After a job is finished, Christos and Milton's (1998) algorithm first records the completion time on other machines, then selects a job with a processing time satisfying $T_j + t_x \geq T[i] + L[i]$ and $T_j + t_x + L_x \geq T[i+1]$. If there are more than two jobs, select the one adhering to FIFO rules.

Jang, Suh, Park, and Liu (2001) suggest a heuristic to minimize flow time and tardiness on parallel machines. Each job has a different processing time on different machines, and there is no local buffer. Once a job arrives, a machine is immediately assigned to it. In contrast with other look-ahead heuristics, the look-ahead heuristic in question considers one part which has

just arrived (part 1) and another part due to arrive next (part 2) with three machine *j* (A, B, C). The potential job *i* (*i* = 1, 2) is selected from the machine's perspective, according to the machine's preference for a particular job. The routing rule determines the machines on which the parts will be processed next to reduce the average flow time or average tardiness of the parts.

Chang, Chou, and Lee (2004) propose a one-step look-ahead heuristic in order to minimize the total weighted tardiness with sequence-dependent setup and unequal ready times. They designate a job for the machine using iteration, so that the partial schedule created produces the smallest total weighted tardiness. Once all jobs are scheduled, reducing the total weighted tardiness by using the pairwise exchange, Chang et al.'s (2004) heuristic proves to be an efficient method for dealing with a small problem size. According to the look-ahead method they propose, when a machine finishes one job, it is kept idle and waiting for the next job released to arrive, even if there is a queue for the machine. On-hand work-in-process (WIP) and a newly-arrived job will be put together and resequenced to decide which one gets top priority (having the smallest total weighted tardiness) to be scheduled next.

For more complicated scheduling environments, such as flow shops and job shops, Smith and Stecke (1996) investigate the influence of the look-ahead strategy on machine utilization in a flexible flow shop. The proposed look-ahead strategy ensures that the machine with the earliest upcoming availability time is used for the next operation. When a job is finished, the machine is kept idle and waiting for the next job to arrive. The job in the queue and the next

arriving job are compared, and the job with higher priority is scheduled next (the job with a smaller input buffer).

Ginzburg and Gonik (1997) propose a heuristic that uses the look-ahead concept and pairwise comparison to assign the next incoming job to an idle machine. Available jobs compete with each other. The look-ahead is decided by analyzing the processing time; the upcoming job's finishing time is estimated, then the available job selected is compared with the upcoming job, with the winner getting assigned to the idle machine.

Wang and Cheng (2015) study a double identical-parallel-machine scheduling problem in which one machine is available to process jobs during a limited time interval, while the other machine is always available over the scheduling horizon. Wang and Cheng (2015) develop a heuristic to tackle the problem by incorporating the backward-adjusting and two-step look-ahead strategies into existing heuristics for similar problems without the machine-availability constraint. The two-step look-ahead strategies take into account two machines and two jobs. The first step looks ahead to the pair of jobs available for processing on time, and the second step considers the smallest job index among the job candidates in the first step.

Xi et al. (2015) propose a look-ahead heuristic to minimize the total weighted tardiness on identical parallel machines. When a machine becomes idle, it selects a job from available jobs and near-future jobs to process. Unlike other look-ahead heuristics, the proposed look-ahead heuristic not only looks ahead (considering a limited number of future jobs) but also looks backwards (therefore, the selected job has a chance to be scheduled before the last job

on each machine). The proposed look-ahead heuristic compares favorably with available look-ahead heuristics and non-look-ahead heuristics to minimize total weighted tardiness on the identical parallel machines.

CHAPTER 3 THE NEW FORMULA ATCSR_UP

3.1. Problem description

This paper considers the problem ($P_m|r_j, s_{ij}, con|\sum w_j T_j$). The problem is stated as follows: jobs arrive at *M* unrelated machines at different times. Each job *j* has a ready time (r_j), a processing time for machine *m* ($p_{j,m}$), a due date (d_j), and a job weight (w_j). Unrelated parallel machines have different values for $p_{j,m}$, or processing times, for the same job. For a given job *j* the setup time (s_{ij}) for each pair of jobs,*i* and *j*, is sequence dependent and continuous, which means that the setup time cannot be initiated before the job is ready, *con*. In general, s_{ij} is not equal to s_{ji}. We also consider t_m (completion time of the last job on machine *m* at the current time). This procedure takes into account both the machine time and the current time *t* The objective is to minimize the total weighted tardiness of jobs, $\sum_{j=1}^{n} w_j T_j$, where T_j is the tardiness of job *j*, $\max(0, C_j - d_j)$, and C_j is the completion time of job *j*.

The problem assumes the following:

- The job attributes (r_j,$p_{j,m}$,d_j,w_j,s_{ij}) are known in advance.
- The machines are unrelated, meaning that a given job has different processing times on different machines.
- Each machine can process at most one job at a time
- Job pre-emption is not allowed.
- Production interruptions such as machine breakdown and order cancellation do not happen.

This section gives detailed information about the ATC-based dispatching rules. The new formula and the new sequencing procedure for unrelated parallel machines will also be introduced here.

3.2 New dispatching rule: ATCSR_UP

Proposing a new ATC-based sequencing rule, this section starts by analyzing the WSPT term, the slack term, and the ready-time term of existing ATC-based rules. Unlike Yue and Jang (2013), we discuss unrelated parallel machines: A job has different processing times on different machines. The basic format of the ATC-based index has four terms:

Index (i, j, m) = (WSPT term) *(Slack term) *(Setup term) *(Ready term) (3)

The index is use to calculate job index for all unscheduled job *j* and machine *m.* Relevant literature mentions that the index of the ATC-based term should have at least two terms (Yue & Jang, 2013). To select the next job for a machine, ATC-based rules compute the index value for each unscheduled job and select the job with the biggest value to be processed.

The formula we propose is

$$I_{ATCSR_{UP}}(t, i, j) =$$

$$\frac{w_j}{p_{i,j}+s_{ij}+\max(r_j-t_m,0)} \exp\left(-\frac{\max\left(d_j-p_j-s_{ij}-\max(r_j,t_m,t),0\right)}{k_1(\bar{p}_m+\bar{s}_m)}\right)\exp\left(-\frac{s_{ij}}{k_2\bar{s}}\right)\exp\left(-\frac{\max(r_j-t_m,0)}{k_3\bar{p}_m}\right) \quad (4)$$

where k_1, k_2, k_3 are scaling parameters, job weight (w_j), processing time (p_j), setup time (s_{ij}), ready time (r_j), machine time (t_m), due date (d_j), current time (*t*), average processing time ($\bar{p}_m$),

and average setup time ($\bar{s}_m$). Section 3.2.1 discusses the difference between the new ATCSR_UP, ATCSR_Rm (Lin and Hsieh, 2013) and other ATC-based rules.

3.2.1 New terms of ATC-based rule

(i)The WPST term

Yue and Jang (2013) point out that the denominator of the conventional WSPT term is p_j, which represents the earliest possible completion time of job *j* from the current time. If the setup time is sequence-dependent, and the job is available at time zero, its earliest job completion time is $p_j + s_{ij}$. Yue and Jang (2013) propose $\frac{w_j}{p_{j,m}+s_{ij}+\max(r_j-t,0)}$ as the WSPT term.

For the continuous sequence-dependent setup where a job has a future ready time, different processing times on different machines, and sequence-dependent setup, the denominator should be $p_{j,m} + s_{ij} + \max(r_j - t_m, 0)$. In our formula ATCSR_UP, $\max(r_j - t_m, 0)$ is the potential machine idle time that influences the earliest completion time for the potential jobs, and it is reasonable to treat these possible idle times as part of the processing time. If $r_j - t_m > 0$ this means the job is a future ready job; otherwise it's a current ready job. Based on the respective analysis, we propose the following formula as the new WSPT term:

$$\frac{w_j}{p_{j,m}+s_{ij}+\max(r_j-t_m,0)} \quad (5)$$

(ii)The slack term

One of the important differences among ATC-based rules is the slack term. Slack time is extra time – between the earliest possible start time of the job and the job's due date – allocated to processing a job.

The important part of the slack term is the time at which the respective job starts. Therefore, we must calculate how long the slack time will be. The new form we propose to decide the earliest start time for each job is:

$$\max(r_j, t_m, t) \qquad (6)$$

Using equation (6), we propose a new exponent numerator for the slack term, by comparing the ready time r_j, machine time t_m , and the current time *t* (minimum machine available time).

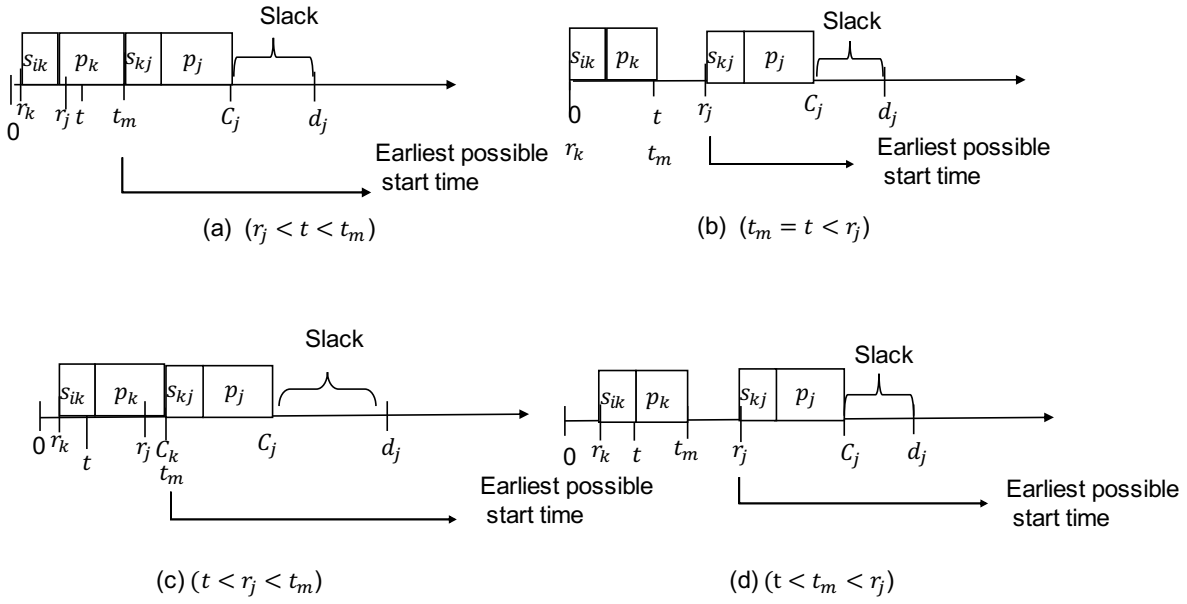


Figure 1. Exponent for slack term.

The slack term exponent we propose is slightly different from the one proposed by Xi and Jang (2012). We use not only current time *t* but also machine available time t_m to calculate

the slack term . Figure 1(a) shows that when the job ready time (r_j) is earlier than the machine available time (t_m), the job can start whenever the machine is ready; therefore, the earliest possible start time will be the machine time (t_m). Figure 1(b) shows that the machine has finished a job or has not yet started any jobs and is therefore currently idle. The potential start time for the machine is whenever a future job arrives at the machine (r_j). According to this concept, the earliest start time is the job ready time (r_j). Figure (c) and (d) we have the current time smaller than both job ready time and machine available time. In this situation, the job will be regard as a future job. However, we still have to consider wether job ready time is select as the earliest start time or the machine available will be select as the earliest start time. For figure (c) the job ready time is smaller than the machine available time, therefore, the job has to wait for the machine to finish the assign job. in this sense the machine available time will be select as the earliest start time. For figure (d) the machine available time is samller than the job ready time. The machine is currently idle, therefore, the job ready time will be the earliest start time.

After the start time is considered in equation (6), we calculate the completion time for the job. Completion time (C_j) is the length between due date and the job's start time, calculated by subtracting from the due date (d_j)the sum of the processing time (p_{jm}), the setup time (s_{ij}) and the start time ($\max(r_j, t_m, t)$). The propose denominator for the term is:

$$\max\left(d_j - p_j - s_{ij} - \max\left(r_j, t_m, t\right), 0\right) \quad (7)$$

Equation (7) shows the completion time: due date (d_j) minus the sum of processing time (p_{jm}), setup time (s_{ij}) and the earliest start time ($\max(r_j, t_m, t)$).

This new formula includes the sequence-dependent setup time in the formula of ATCSR, the machine processing time for different jobs, and the times at which the machines become

available to process the next potential job. The slack time measures the maximum amount of time by which the start of a job may be postponed after the earliest possible start time, max (r_j, t_m, t), while still adhering to the deadline of its due date.

The new slack term we propose is:

$$\exp\left(-\frac{\max(d_j - p_{j,m} - s_{ij} - \max(r_j, t_m, t), 0)}{k_1(\bar{p}_m + \bar{s}_m)}\right) \quad (8)$$

(iii) Exponent denominator of the slack term

For the exponent denominator of the slack term, we take the denominator used by Yue and Jang (2013), which is the combination of average processing time ($\bar{p}_m$) and average setup time ($\bar{s}_m$). $\bar{p}_m$ is the average processing time for each machine, calculated by taking the total processing time on each machine and dividing it by the number of jobs on each machine. $\bar{s}_m$ is the average setup time for each machine, calculated using the average setup time on each machine divided by the number of jobs on each machine:

$$k_1(\bar{p}_m + \bar{s}_m) \quad (9)$$

where k_1 is the scaling parameter for the slack term.

(iv) **The ready term**

In the ready-term numerator, the existing ATC-based rule uses *t* to calculate the numerator of the ready term. In ATCSR_UP, we use machine available time (t_m) to calculate the numerator. Also in the numerator, we use average processing time ($\bar{p}_m$). $\bar{p}_m$ is the average processing time for each machine, calculated using the total processing time on each machine divided by the number of jobs on each machine. We propose the following denominator for the exponent

ready term:

$$\max\left(r_j - t_m, 0\right) \quad (10)$$

Combining all the above terms, we propose the ATC-based rule formula ATCSR_UP (continuous sequence-dependent setup), shown in equation (4) and equation (11) below:

$$I_{ATCSR_{UP}}(t,i,j) = \frac{w_j}{p_{i,j}+s_{ij}+\max\left(r_j-t_m,0\right)} \exp\left(-\frac{\max\left(d_j-p_j-s_{ij}-\max\left(r_j,t_m,t\right),0\right)}{k_1(\bar{p}_m+\bar{s}_m)}\right) \exp\left(-\frac{s_{ij}}{k_2\bar{s}}\right) \exp\left(-\frac{\max\left(r_j-t_m,0\right)}{k_3\bar{p}_m}\right) \quad (11)$$

3.2.2 Index formula comparison

For the formula we propose, we make comparisons with other papers that use an ATC-based rule. Table 3 compares the ATC-based formulas.

Table 3. Comparison of terms ATC-based rule

Rule	WSPT term	Slack term	Setup term	Ready term	Equation
$I_{BATCS}(t,i,j)$ *Mason et al. (2002)*	$\frac{w_j}{p_{i,j}}$	$\exp\left(\frac{-\max(d_j - p_j - t, 0)}{k_1\bar{p}}\right)$	$\exp\left(\frac{-s_{ij}}{k_2\bar{s}}\right)$		(1)
$I_{BATCSmod}(t,i,j)$ *Pfund et al. (2008)*	$\frac{w_j}{p_{i,j}}$	$\exp\left(\frac{-\max(d_j - p_j + \max(r_j - t, 0), 0)}{k_1\bar{p}}\right)$	$\exp\left(\frac{-s_{ij}}{k_2\bar{s}}\right)$		(2)
$I_{ATCSR}(t,i,j)$ *Pfund et al. (2008)*	$\frac{w_j}{p_{i,j}}$	$\exp\left(\frac{-\max(d_j - p_j + \max(r_j, t), 0)}{k_1\bar{p}}\right)$	$\exp\left(\frac{-s_{ij}}{k_2\bar{s}}\right)$	$\exp\left(\frac{-\max(r_j - t, 0)}{k_3\bar{p}}\right)$	(3)
$I_{ATCRCS}(t,i,j)$ *Yue and Jang (2011)*	$\frac{w_j}{p_j + s_j + \max(r_j - t, 0)}$	$\exp\left(\frac{-\max(d_j - p_j - s_{ij} + \max(r_j, t), 0)}{k_1\bar{p}}\right)$	$\exp\left(\frac{-s_{ij}}{k_2\bar{s}}\right)$	$\exp\left(\frac{-\max(r_j - t, 0)}{k_3\bar{p}}\right)$	(4)
$I_{ATCRSS}(t,i,j)$ *Yue and Jang (2011)*	$\frac{w_j}{p_{i,j} + s_{ij} + \max(r_j - t, 0)}$	$\exp\left(\frac{-\max(d_j - p_j - t, 0)}{k_1(\bar{p} + \bar{s})}\right)$	$\exp\left(\frac{-s_{ij}}{k_2\bar{s}}\right)$	$\exp\left(\frac{-\max(r_j - t, 0)}{k_3\bar{p}}\right)$	(5)
$I_{ATCSR\_RM(m,j,l)}$ *Lin and Hsieh (2013)*	$\frac{w_j}{p_{m,j}}$	$\exp\left(\frac{-\max(d_j - p_{j,m} - \max(r_j, t_m + s_{m,j,l}), 0)}{k_1\bar{p}}\right)$	$\exp\left(\frac{-s_{mlj}}{k_2\bar{s}}\right)$	$\exp\left(\frac{-\max(r_j - t_m - s_{mlj}, 0)}{k_3\bar{p}}\right)$	(6)
$I_{ATCRS\_UP(m,i,j)}$ Us	$\frac{w_j}{p_{j,m} + s_{ij} + \max(r_j - t_m, 0)}$	$\exp\left(\frac{-\max(d_j - p_{j,m} - s_{ij} - \max(r_j, t_m, t), 0)}{k_1(\bar{p}_m + \bar{s}_m)}\right)$	$\exp\left(\frac{-s_{ij}}{k_2\bar{s}}\right)$	$\exp\left(\frac{-\max(r_j - t_m, 0)}{k_3\bar{p}_m}\right)$	(7)

In the table, Equations (1) – (4) focus either on a single machine or on identical parallel machines. Therefore, we focus our comparison on equation (5) (Yue & Jang, 2011) and (6) (Lin & Hsieh, 2013).

(i) The WSPT term

Starting with the WSPT term, we compare our own WSPT term with those of Xi and Jang (2011) and Lin and Hsieh (2013). Unlike Lin and Hsieh (2013), who only use $p_{j,m}$ because they have the searable setup case, however, we consider continuous sequence-dependent setup time (s_{ij}), machine ready time (t_m), and processing time ($p_{j,m}$). For continuous sequence-dependent setup, the setup time should be considered with the processing time, and the machine time should also be calculated, because every machine has its own schedule time. Therefore, we include the machine time in the processing time and setup time.

Instead of using current decision time *t* (Xi & Jang, 2011), we use machine time (t_m) for the WSPT term. The $\max(r_j - t_m, 0)$ is the potential machine idle time that influences the earliest completion time for the potential jobs, and it is reasonable to treat these possible idle times as part of the processing time.

(ii) The Slack term

For the slack term, Lin and Hsieh (2013) use job processing time and earliest start time, $\max(r_j, t_m + s_{m,j,l})$, to calculate $\frac{-\max(d_j - p_{j,m} - \max(r_j, t_m + s_{m,j,l}), 0)}{k_1 \bar{p}}$. If the job ready time (r_j) is larger than machine available time + setup time ($t_m + s_{m,j,l}$), the earliest start time for a job will be the job ready time (r_j), and if the machine available time + setup time ($t_m + s_{m,j,l}$) is larger than job ready time (r_j) the machine available time + setup time ($t_m + s_{m,j,l}$) will be

select as the earliest start time. We consider the sum of processing time, setup time and earliest start time $\max(r_j, t_m, t)$ to calculate $\frac{-\max(d_j - p_{j,m} - s_{ij} - \max(r_j, t_m, t), 0)}{k_1(\bar{p}_m + \bar{s}_m)}$. We use the current time, t, the minimum machine available time. We compare the earliest start time of a job by using job ready time, machine available time and current time. The consideration is show in section 3.2.1.

(iii) The Ready term

We lastly compare our own ready term with Lin and Hsieh's (2013). We consider only $-\max(r_j - t_m, 0)$ other than $-\max(r_j - t_m - s_{mlj}, 0)$. The setup time does not feature here, because the setup time in ATCSR_UP is calculated as part of the machine time (t_m) already. The machine time (t_m), when no jobs have been assigned, the machine time (t_m) starts at time 0. However, when the machine has its first job assigned, the machine time (t_m) consists of job j, processing time ($p_{j,m}$), setup time (s_{ij}), and $\max(r_i - t_m, 0)$. Therefore, the setup time (s_{ij}) is already considered in the machine time (t_m).

3.3 Job and machine selection procedure with look-ahead method

Based on the above analysis, this section introduces the job- and machine-selection procedure of ATCSR_UP to minimize the total weighted tardiness with sequence-dependent setup, different processing time and future ready time. In this section, we discuss how a job and a machine are selected based on the index. In the relevant literature, the look-ahead heuristic we use in our procedure is referenced from the look-ahead heuristic proposed by Yue and Jang (2013). In Yue and Jang (2013), the propose look-ahead heuristic not only looks ahead (considers limited numbers of future jobs) but also looks back (schedules each selected job

before the last job is assigned to each machine). In contrast with Yue and Jang (2013), we look ahead to the two jobs with the biggest job indexes and assign the jobs to the respective machines with the smallest tardiness. For traditional ATC-based rule, it selects the job to assign to the machine by choosing the largest job index. However, this procedure can be a look-ahead method because we use the concept of the look-ahead method-look-ahead to the near future states of the production shop and pick one more job to compare the tardiness.

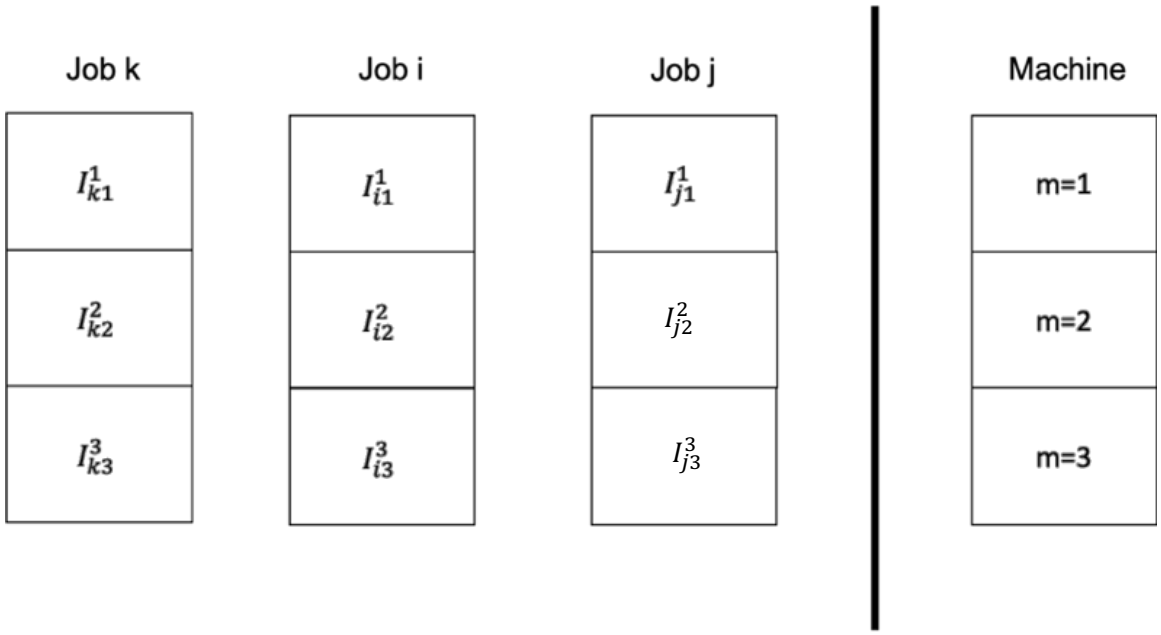


Figure 2. Different index among different machines.

Figure 2 shows that each job has different index values for different machines. In a previous paper by Yue and Jang (2013), the highest index is chosen and the job is scheduled on a parallel machine. This procedure is quite straightforward for identical parallel machines because they have the same processing speed: A higher index gets higher priority for the process to be carried out on the machine. However, in this study, an identical parallel machine has been replaced by an unrelated parallel machine, meaning that each job has its own processing time on each machine. Hence, we consider the different processing times for

different machines in order to schedule the job in question.

The way we choose the job to be processed and the machine to be used involves calculating all job indexes. For all jobs and all machines, we choose the two biggest indexes and compare the tardiness ($max\{0, C_j - d_j\}$ and $C_j = p_{j,m} + s_{ij} + \max(r_j, t_m)$) when the jobs are assigned to the machines; the job with a smallest tardiness value is then assigned to the related machine. We assign all the job with this procedure.

However, there are two situations that needs to be consider. First, if the selected jobs have the same tardiness, we calculate the machine time for the related two machines and assign the job to the relate machine with the smallest machine time (t_m). Second, if the two biggest job indexes are the same job on different machine, we pick the largest job index and the second large job on the other machine and compare the tardiness. The job with the smallest tardiness will be assign to the relate machine.

We show an example of four jobs and two machines; in this example, the index values are calculated based on the new formula, equation (11). Input data for machines and jobs are shown in Tables 4-7.

Table 4. Job data for Machine 1 ($t_1 = 0$)

Machine 1				
Job	1	2	3	4
Ready time	4	0	5	0
Processing time	3	19	3	19
Due date	10	5	13	12
Weight	4	2	8	3
Setup time	3	2	4	5

Table 5. Job data for Machine 2 (t_2 = 0)

Machine 2				
Job	1	2	3	4
Ready time	4	0	5	0
Processing time	1	2	1	4
Due date	10	5	13	12
Weight	4	2	8	3
Setup time	2	4	3	1

Iteration 1

Table 6. Job Index for Machine 1

Job	1	2	3	4
Index	0	0.000121203	0	7.22219E-09

Table 7. Job Index for Machine 2

Job	1	2	3	4
Index	0	3.75117E-08	0	0.004236

Each machine time starts at time t = 0. We calculate all the job indexes first shown in Tables 6 and 7. We choose Machine 1, Job 2 and Machine 2, Job 4 because they are the two biggest job indexes among all job indexes. Now we calculate the tardiness max(C_j-d_j, 0). For Machine 1 Job 2 the tardiness is max (C_j= $p_{j,m}$+ s_{ij} + max (r_j, t_m) = 19+2+0-5,0) = 16 and the tardiness for Machine 2 Job 4 is max (4+1+0-12,0) = 0. Machine 2, Job 4 has no tardiness so we assign Job 4 to Machine 2. The machine time is now (t_m = processing time: 4+ setup time:1+ready time: 0=5). We delete Job 4 from the table and consider the next job to be assign.

The next job is considered in the same way, but there is a precondition in our procedure. We compare the two largest job index and choose the machine to assign the job with a smaller machine time (t_m) when the compared two jobs have the same tardiness.

Iteration 2

Table 8. Job Index for Machine 1 (t_1 = 0)

Job	1	2	3
Index	0	0.000121203	0

Table 9. Job Index for Machine 2 (t_2 = 5)

Job	1	2	3
Index	0.000176	3.75117E-08	1.912E-06

Now we compare the two biggest index, Machine 1 Job 2 and Machine 2 Job 1. Job 1 will be assigned to Machine 2, because Job 1 max (5+1+2-10,0) = 0 has the smaller tardiness than Job 2 max (0,19+2+0) = 21. We update the machine time for Machine 2 (t_2= Job 4 completion time: 5+processing time: 1+setup time :2).

Iteration 3

After the jobs are assigned, we calculate the unscheduled jobs index (Job 2 and Job 3).

Table 10. Job Index for Machine 1 (t_1 = 0)

Job	2	3
Index	0.000121203	0

Table 11. Job Index for Machine 2 (t_2 = 8)

Job	2	3
Index	3.75117E-08	7.71E-06

Now we compare the jobs index. Machine 1 Job 2 and Machine 2 Job 3 are the two biggest job indexes. As we mentioned, we assign the job to the machine with the smallest tardiness, therefore, Job 3 max (0,8+1+3-13) = 0 is assign to Machine 2 due to the smallest tardiness Job 2 max (0,19+2-5) = 16. Now we update the machine time for Machine 2 (Job 4 & Job 1 completion time: 8+processing time: 1+setup time: 3=12)

Table 12. Job Index for Machine 1 (t_1 = 0)

Job	2
Index	0.00012120

Table 13. Job Index for Machine 2 (t_2 = 12)

Job	2
Index	0.00064

Last, only Job 2 in unscheduled. We compare the indexes and found that on Machine 2 the index is larger than the job index on Machine 1 and the tardiness is also smaller on Machine 2 (t_2:12+processing time: 2+setup time: 2-due date: 5 =9 < processing time: 19+setup time: 2-due date: 5 = 16). Therefore, we assign Job 2 to Machine 2 and calculate the total weighted tardiness.

The schedule we have now is Machine 1: no job, Machine 2: Job 4-Job 1- Job 3-Job 2. The weighted tardiness for Machine 1 is 0 and Machine 2 is max (5-12,0) *3+max (8-10,0) *4+max

(12-13,0) *8 + max (0,13) * 2 =26. Therefore, the total weighted tardiness for the whole schedule is 26.

Section 4.3.1. gives our data test results.

3.4 Local Improvement Procedure: Pairwise adjacent job exchange

In order to minimize total weighted tardiness, we apply a job-swap procedure for each machine after the initial schedule is completed. In the job-swap method, we select the job schedule that gives the smallest total weighted tardiness on each machine. For example, we assigned Job 1 and Job 2 to Machine 1, so the initial schedule is Job1–Job2. However, if Job2–Job1 gives a smaller total weighted tardiness, we swap the jobs.

In the following procedure for the job-swap method, we show the procedure first and then provide an example to demonstrate the procedure:

Step 1: Generate initial schedule on each machine by the use of new index formula and job machine selection.

Step 2: List out the possible combinations of the jobs decided from the initial schedule of each machine

Step 3: Calculate the tardiness for each combination schedule.

Step 4: Choose the schedule on each machine that has the smallest total weighted tardiness. Check wether the select schedule can provide better TWT. Go to step 2 if the current schedule can still provide another schedule, otherwise we go to step 5

Step 5: List the schedule on each machine after the job-swap method has been applied.

An example comes after the following procedure.

Example 7 job 3 machine

Here we have 7 jobs along with 3 machines to show an example. All machines start at time t = 0. For the scale parameters for this problem we consider (k_1= 0.2, k_2= 0.1and k_3= 0.001). The input data is from Table 14 to Table 22. Table 14 to 16 is the input data for each machine (machine 1 to 3). Table 17 to 19 is the setup time for each job on each machine. Table 20 to 22 is the job index for three machines. In order to focus on Job swap method, we run the ATCSR_UP without job swap first to get the initial schedule.

Table 14. Machine 1 Job input data

Job	1	2	3	4	5	6	7
Ready time	7	1	8	6	5	7	3
Processing time	-	38	23	28	-	27	38
Due date	72	65	74	71	72	73	74
Job weight	1	4	4	3	2	3	1
Setup time	-	8	14	27	-	15	36

Table 15. Machine 2 Job input data

Job	1	2	3	4	5	6	7
Ready time	7	1	8	6	5	7	3
Processing time	41	-	24	38	25	35	-
Due date	72	65	74	71	72	73	74
Job weight	1	4	4	3	2	3	1
Setup time	36	-	1	31	20	29	-

Table 16. Machine 3 Job input data

Job	1	2	3	4	5	6	7
Ready time	7	1	8	6	5	7	3
Processing time	43	-	-	46	30	-	41
Due date	72	65	74	71	72	73	74
Job weight	1	4	4	3	2	3	1
Setup time	7	-	-	26	22	-	10

Table 17. Machine 1 setup time for each job

M1 Setup time	1	2	3	4	5	6	7
Initial	-	8	14	27	-	15	36
1	-	-	-	-	-	-	-
2	-	-	11	11	-	38	26
3	-	9	-	22	-	34	38
4	-	35	18	–	-	4	20
5	-	-	-	-	-	-	-
6	-	40	29	18	-	-	18
7	-	36	5	38	-	29	-

Table 18. Machine 2 setup time for each job

M2 Setup time	1	2	3	4	5	6	7
Initial	36	-	1	31	20	29	-
1	-	-	39	27	5	21	-
2	–	-	-	-	-	-	-
3	27	-	-	30	15	1	-
4	20	-	23	–	25	3	-
5	34	-	20	17	-	14	-
6	22	-	3	12	34	-	-
7	-	-	-	-	-	-	-

Table 19. Machine 3 setup time for each job

M3 Setup time	1	2	3	4	5	6	7
Initial	7	-	-	26	22	-	10
1	-	-	-	40	26	-	1
2	-	-	-	-	-	-	-
3	-	-	-	-	-	-	-
4	8	-	–	–	18	-	13
5	13	-	-	27	-	-	10
6	-	-	-	-	-	-	-
7	34	-	-	11	32	-	-

Table 20. Job index for Machine 1

Job	1	2	3	4	5	6	7
Index	-	0.00029	1.22E-90	1.03E-64	-	1.05E-76	1.61E-24

Table 21. Job index for Machine 2

Job	1	2	3	4	5	6	7
Index	2.7E-09	-	0.0298	7.67E-08	8.62E-06	1.94E-07	-

Table 22. Job index for Machine 3

Job	1	2	3	4	5	6	7
Index	6.8E-37	-	-	8.79E-31	6.9E-19	-	4.1E-05

This section is an example of how we apply the job swap method. We use the ATCSR_UP without job swap method to get an initial schedule for the 3 machines 7 jobs problem. Machine 1: J2-J4, Machine 2: J3-J6-J5, Machine 3: J7-J1 is the initial schedule generate from ATCSR_UP without job swap method.

Step 1. Receive an initial job on each machine

From ATCSR_UP without job swap method, we get the initial schedule for each machine: Machine 1: J2-J4, Machine 2: J3-J6-J5, Machine 3: J7-J1 and the TWT for the initial schedule is 197.

Step 2: List out the all the possible combinations of the jobs from the initial schedule for each machine

Machine 1: J2-J4 and J4-J2

Machine 2: J3-J6-J5, J3-J5-J6, J6-J3-J5, J5-J6-J3

Machine3: J7-J1 and J1-J7

Step 3. Calculate the TWT for each job schedule.

Now we calculate the tardiness for each machine and swap the jobs.

Table 23. TWT for 3 machines

Machine 1	TWT_{24}**:48**	TWT_{42}:260
Machine 2	TWT_{365}:**116**	TWT_{356}:159
	TWT_{635}:186	TWT_{563}:216
Machine 3	TWT_{71}:56	TWT_{17}:**18**

Step 4. Choose the schedule on each machine that has the smallest total weighted tardiness. If the new schedule is better than the initial schedule, swap the jobs. Go to step 2 if the current schedule can still provide another schedule, otherwise we go to step 5.

Machine 1: J2-J4: 48

Machine 2: J3-J6-J5: 116

Machine 3: J1-J7: 18 (Job swap is applied here; the original schedule is J7-J1)

We found that the schedule on Machine 2 can still provide another schedule. Therefore, we go to Step 2 again.

Step 2: List out the all the possible combination of the jobs decides from the initial schedule for each machine

Machine 1: J2-J4 and J4-J2

Machine 2: J3-J6-J5, J6-J5-J3, J5-J3-J6

Machine3: J7-J1 and J1-J7

Step 3. Calculate the TWT for each job schedule.

Now we calculate the tardiness for each machine and swap the jobs.

Table 24. TWT for 3 machines

Machine 1	TWT_{24}**:48**	TWT_{42}:260
Machine 2	TWT_{365}:**116**	TWT_{653}:474
	TWT_{536}:181	
Machine 3	TWT_{71}:56	TWT_{17}:**18**

Step 4. Choose the schedule on each machine that has the smallest total weighted tardiness. If the new schedule is better than the initial schedule, swap the jobs. Go to step 2 if the current schedule can still provide another schedule, otherwise we go to step 5.

Machine 1: J2-J4: 48

Machine 2: J3-J6-J5: 116

Machine 3: J1-J7: 18 (Job swap is applied here; the original schedule is J7-J1)

Step 5. List the schedule on each machine after the job-swap method has been applied.

Machine 1: J2-J4

Machine 2: J3-J6-J5

Machine 3: J1-J7

The original TWT is 197, and after applying the job swap method the TWT become 182.

3.5 The new ATC-based rule procedure

In this research, we focus on unrelated parallel machines. In contrast with Yue and Jang's (2013) study on identical parallel machines and uniform parallel machines, we cannot simply choose a job with a larger index, because that might lead to all jobs being scheduled on the same machine, due to the differences in processing times for individual jobs. We have to consider the choice of machine and job: In the propose ATC-based rule, we calculate all the jobs' indexes first and then decide how to schedule the jobs on the machines.

For the propose rule, we calculate the job indexes and combine with a look-ahead method. We solve the problem by calculating all the job indexes and picking the two largest indexes among the different machines; we then compare the two jobs, pick the job with smallest tardiness, and then schedule the job to the machine.

Figure 3. shows the procedure according to the propose ATC-based rule works to determine the schedule for jobs and machines.

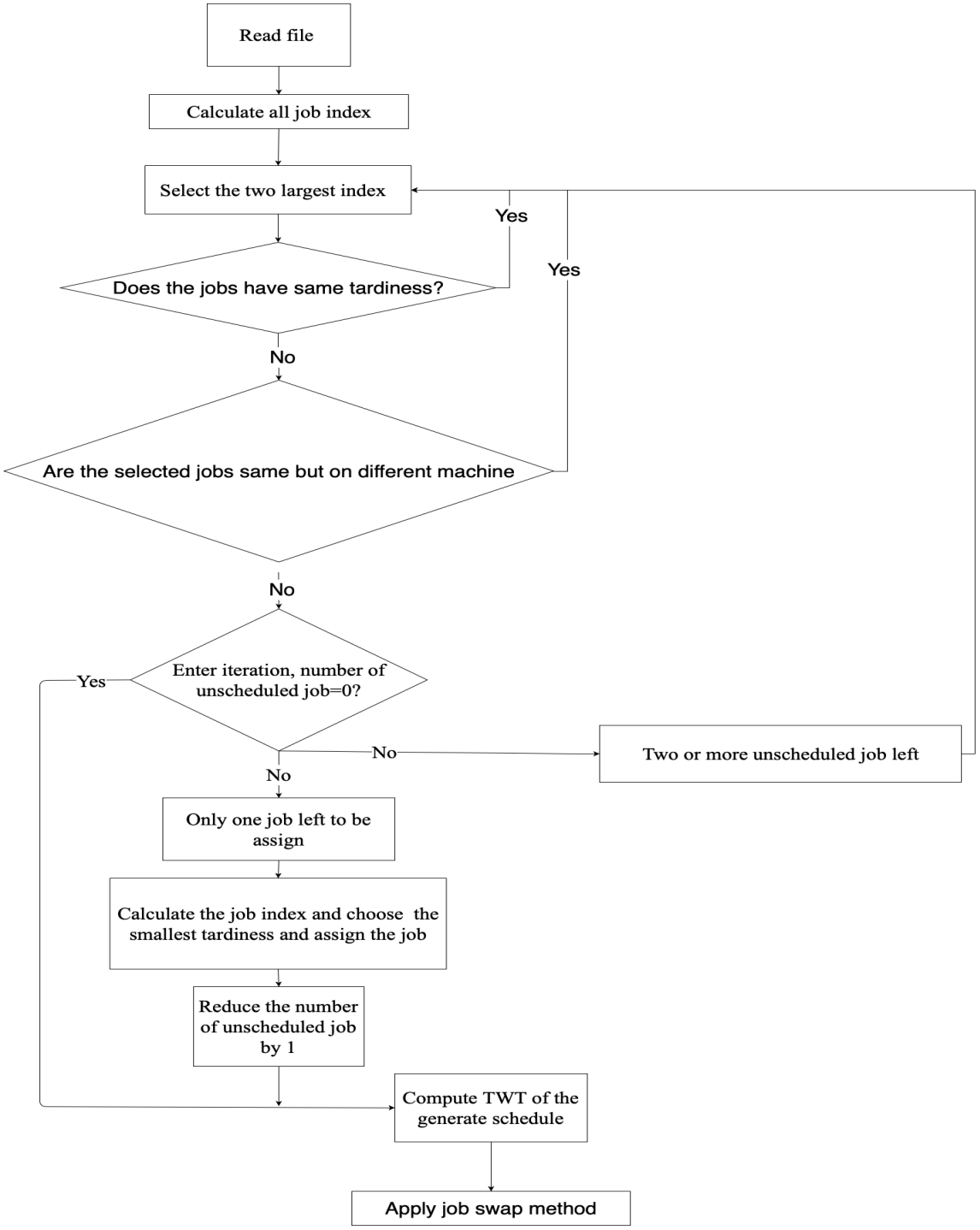


Figure 3. Flow chart for the ATCSR_UP

The steps for the proposed procedure for *n* jobs and *m* machines are:

Step 1: Calculate all the job indexes.

Step2: Determine the first job, j, to assign. For this look-ahead to the two largest respective indexes among all machines. If the two jobs are on the same machine, find the next largest job index on a different machine.

Step 3: Compare the two jobs among the machines, and assign the job to the machine with the smallest tardiness value. If tardiness is the same in both cases, randomly pick one of them and assign, minus the job and machine number by 1. Update the machine time $\max(r_j, t_m) + p_{j,m} + s_j$, setup time ($s_{jk}, k = 1 \dots n$) and

Step 4: Calculate the unscheduled jobs' indexes

Step 5: Select the two largest indexes among the unscheduled jobs.

Step 6: Schedule the job to the relate machine with the smallest tardiness. If tardiness is the same in both cases, pick the job to the relate machine with a smaller completion time and assign, minors the job and machine number by 1. Update machine time to $\max(r_j, t_m) + p_{j,m} + s_j$.

Step 7: Repeat Step 4 to 6 until all the jobs are scheduled.

Step 8: After the initial schedule is finished, apply a job-swap method.

Step 9: List all possible combinations on each machine

Step 10: Calculate the TWT for each job schedule.

Step 11: Choose the schedule on each machine that has the smallest total weighted tardiness. If the new schedule is better than the initial schedule, swap the jobs. Go to step 9 if the current schedule can still provide other schedules, otherwise go to step 12.

Step 12: List the schedule on each machine after the job-swap method has been applied.

CHAPTER 4. PERFORMANCE TEST

This chapter evaluates the performance of the new formula and procedure. In order to compare our results with the most relevant paper, we compare our data test results with Lin and Hsieh (2013) who focus on the ATC-based rule and unrelated parallel machines.

4.1 Comparison of test cases

In this chapter, ATCSR_UP will be tested with the data used by Lin and Hsieh (2013) and the results will be compared. The procedure is coded on a MacBook pro 2016, Intel Core i5, memory 8 GB, Python. Testing problem instances were generated in the manner similar to Lin and Hsieh (2013).

(1) Processing time ($p_{j,m}$) were generated from uniform distribution [50, 150] randomly.

(2) The weighted w_j were chosen from the uniform distribution [1, 10] randomly.

(3) Setup time were generated from uniform distribution [0, 2$\bar{s}$], where the mean setup time $\bar{s}$ was calculated by using the value of setup severity factor. $\bar{s}$ is calculate by setup severity factor * average processing time.

(4) Due dates were generated from uniform distribution $[(1-R)\bar{d}, \bar{d}]$ with probability τ and distribution $[\bar{d}, \bar{d} + (C_{max} - \bar{d})R]$ with the probability (1- τ). *R* is the due date range factor. The makespan was estimated by $C_{max} = (\beta\bar{s} + \bar{p})$ where μ was the job machine factor, which was given by $\mu = n/m$ (*n* is the number of jobs and *m* is the number of machines), and β was the coefficient accounting for the increase in makespan due to setup times, which was given by $\beta = 0.4 + 10/\mu^2 - \eta/7$ (η is the setup severity factor). The mean due date was calculated using $\bar{d} = C_{max}(1-\tau)$.

(5) Ready times r_j were generated from uniform distribution [max(d_j-$r_j \times \bar{p}_j$,0), d_j] where $\bar{p}_j = \bar{p}/m, j = 1 \ldots n$

Table 25. Values of factors for experiment.

Factors	Low level	High level
Setup severity factor η	0.02	2.00
Due date tightness factor τ	0.30	0.90
Due date range factor R	0.25	1.00
Ready time factor r_τ	1.00	10.00

Table 25. shows the factors we use for all the test results. In the figure caption we will show (LLLL) which represents the value factor levels we use (Low Low Low Low). Setup severity factor η: low level, Due date tightness factor τ: low level, Due date range factor R: low level, Ready time factor r_τ: low level and so on.

4.2 The selection of ATC scale parameters

In this section, we try to find better combination of scale parameters for our ATC-based rule. The k values are k_1: [0.2,7.2], k_2: [0.1,2.1], k_1: [0.003,1.2]. According to Pfund *et al.* (2008) the best scale parameters for their propose research, is when $k_1 = 0.6, k_2 = 0.3, k_3 = 0.003$; therefore, we compare the scale parameters with $k_1 = 0.2, k_2 = 0.1, k_3 = 0.001$, $k_1 = 0.6, k_2 = 0.3, k_3 = 0.003$ and $k_1 = 7.2, k_2 = 2.1, k_3 = 1.2$. The reason we chose $k_1 = 0.2, k_2 = 0.1, k_3 = 0.001$ is because these three parameters are the first three parameters among the three k values and $k_1 = 7.2, k_2 = 2.1, k_3 = 1.2$ are the last three among the parameters. We compare the three groups of parameters and try to find the best and

worst combinations of the k values. We will compare the worst and the best combinations in the next section. We name $k_1 = 0.2, k_2 = 0.1, k_3 = 0.001$ as group A, $k_1 = 0.6, k_2 = 0.3, k_3 = 0.003$ as group B, and $k_1 = 7.2, k_2 = 2.1, k_3 = 1.2$ as group C.

From Figure (a) to (p), the y-axis is the total weighted tardiness for the results and the x-axis is the number of cases we test. The figure caption (LLLL) represents the value factors (Low Low Low Low). For each combination we test 20 sets of data. The results are shown as follow:

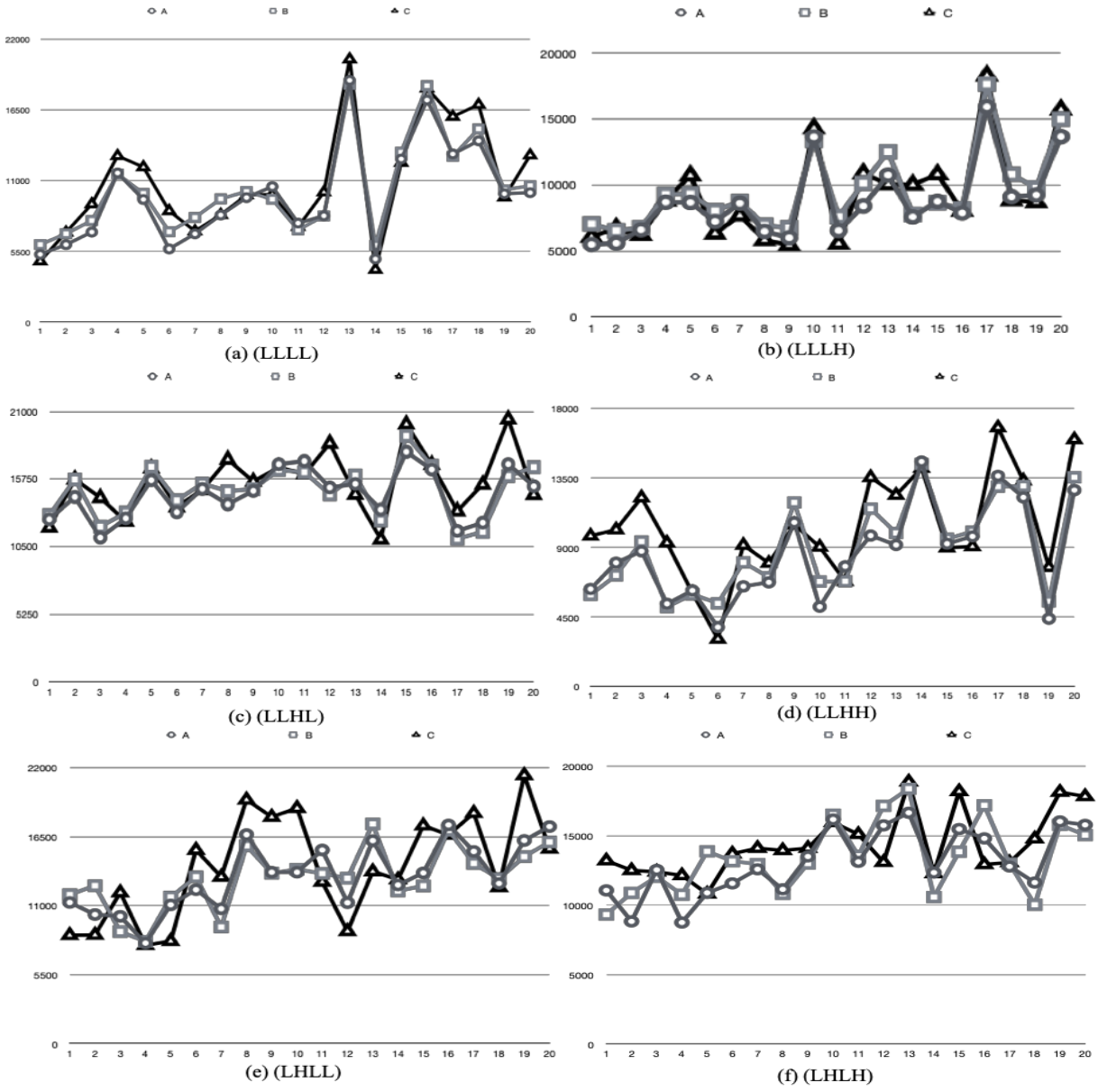
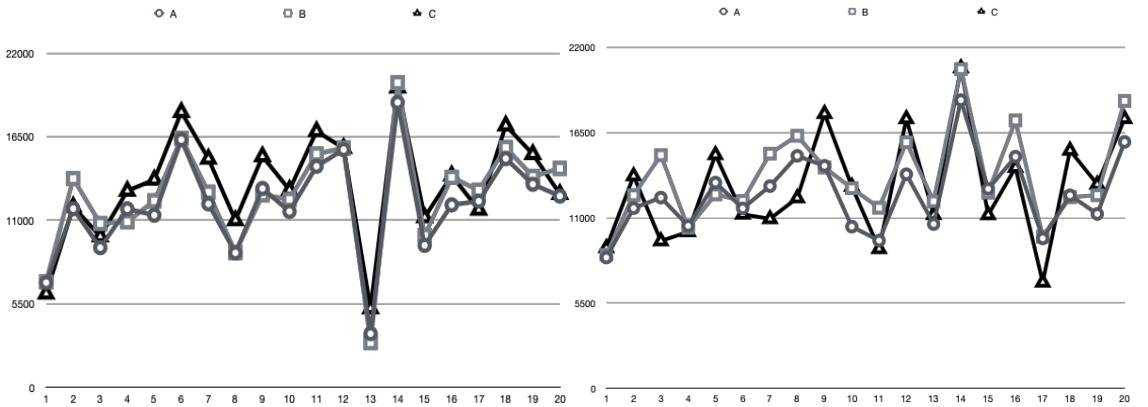
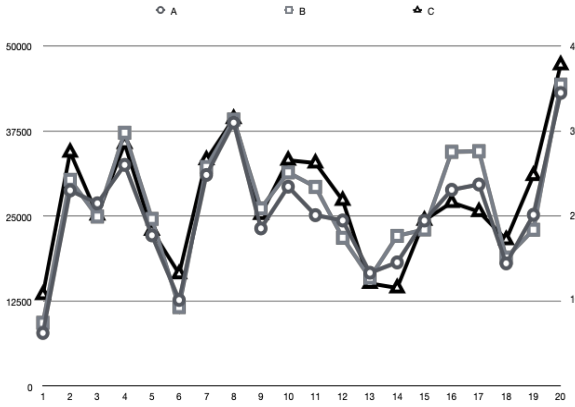


Figure 4. Comparison for group A vs. B vs. C

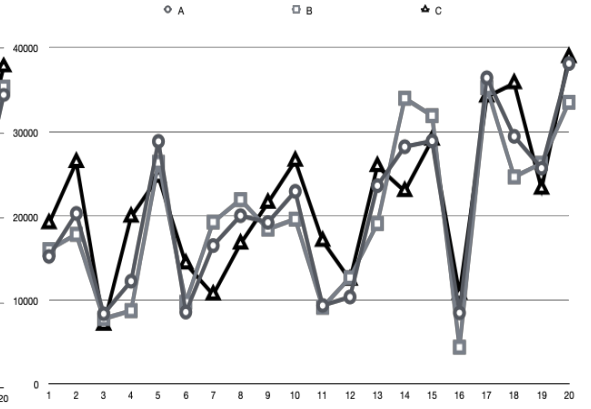


(g) (LHHL)

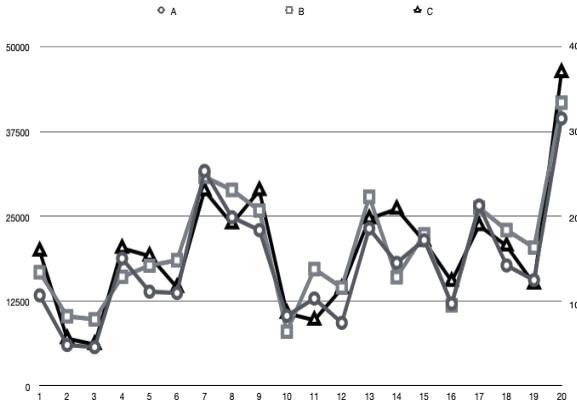
(h) (LHHH)



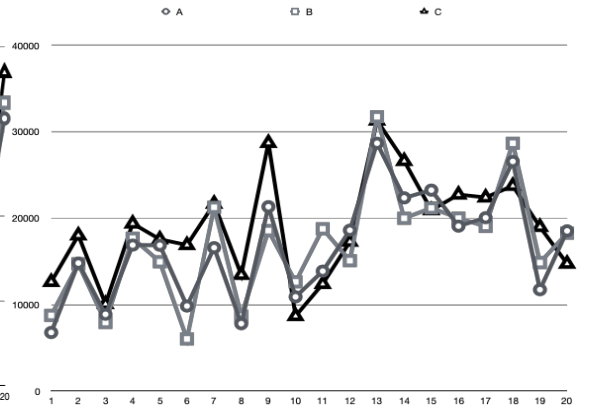
(i) (HLLL)



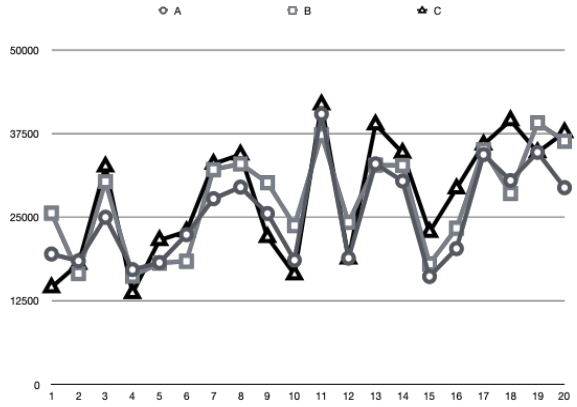
(j) (HLLH)



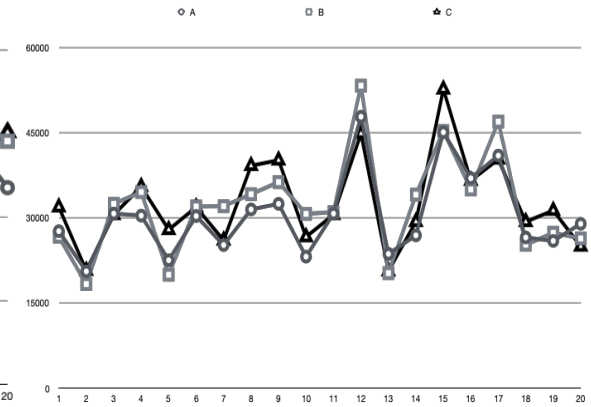
(k) (HLHL)



(l) (HLHH)

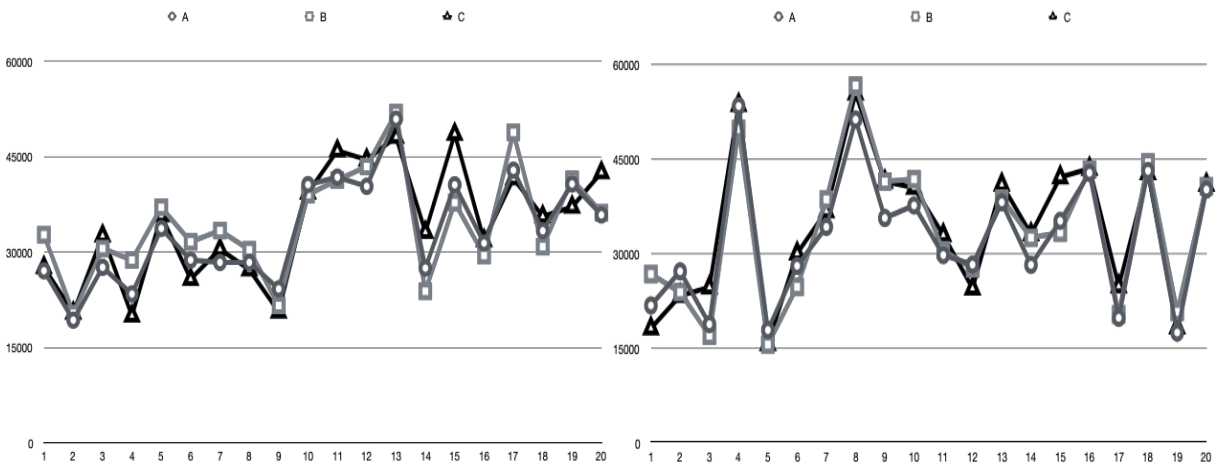


(m) (HHLL)



(n) (HHLH)

Figure 5. Comparison for group A vs. B vs. C



(o) (HHHL) (p) (HHHH)

Figure 6. Comparison for group A vs. B vs. C

From Figure (a) to (p) we compare the average TWT for the tested 16 value factor combinations for the three groups. The y-axis is the total weighted tardiness value and the x-axis is the number of sets. We test 20 cases for each value factor combination and found that in the tested 320 cases $k_1 = 0.2, k_2 = 0.1, k_3 = 0.001$ outperform $k_1 = 0.6, k_2 = 0.3, k_3 = 0.003$ in 205 cases (64%) and outperform $k_1 = 7.2, k_2 = 2.1, k_3 = 1.2$ in 221 cases (70%). From Figure (a) to (p), we found that as the value factors gets higher in level the total weighted tardiness slightly increases and the curve becomes more sensitive. We look into the value factors to see how the value factors effects the TWT. For the setup time $[0,2\bar{s}]$, it is calculated by $\bar{p}_m * \eta$ (average processing time * setup severity facto, therefore, as the factor level goes higher the interval between $[0,2\bar{s}]$ becomes larger. Since the interval for setup time in low level is 2% * average processing time and high level is 200% * processing time. This means that there will be more selection for setup time ($[0,2\bar{s}]$). When due date tightness factors are low in level, the due date values become larger, consequently the slack time for each job becomes longer because the machine will have more time to process the job. This comes with the result

that the TWT with value factors in lower level is smaller than the TWT with value factors in high level. This cause the curve to become more sensitive.

Table 26. shows the average TWT for the three groups with the 16 values factor combination. In section 4.4 we will compare ATCSR_UP with ATCSR_Rm (best and worst k values). From Table 26. we conclude that group A ($k_1 = 0.2, k_2 = 0.1, k_3 = 0.001$) has the smallest average TWT and group C ($k_1 = 7.2, k_2 = 2.1, k_3 = 1.2$) has the largest average TWT among the three-compared group. Therefore, in section 4.4 we will use group A ($k_1 = 0.2, k_2 = 0.1, k_3 = 0.001$) and group C ($k_1 = 7.2, k_2 = 2.1, k_3 = 1.2$) to compare with ATCSR_Rm.

Table 26.Average TWT of the three groups comparison

Group / Value Factors	A	B	C
Low low low low	9875.05	10007.4	10751.2
Low low low high	8758.25	9321.05	10180.8
Low low high low	14591.55	13658.25	15360.65
Low low high high	8543.85	8964.5	9671.05
Low high low low	12007	11264.6	12969.85
Low high low high	12504.95	12265.4	12754.25
Low high high low	12007	12728.25	12396.6
Low high high high	12504.95	13857.1	13617.1
High low low low	25335.7	25972.7	28127.8
High low low high	20538.55	21192.95	20899.05
High low high low	17898.55	19846.8	20051.2
High low high high	16682.75	19045.55	19486.3
High high low low	25522.05	26106.55	27961.1
High high low high	30414.4	33992.4	32079.55
High high high low	33384.3	33787.9	35314.45
High high high high	32392.9	33665	35039.15
Average TWT	18310.1125	19104.775	19791.25625

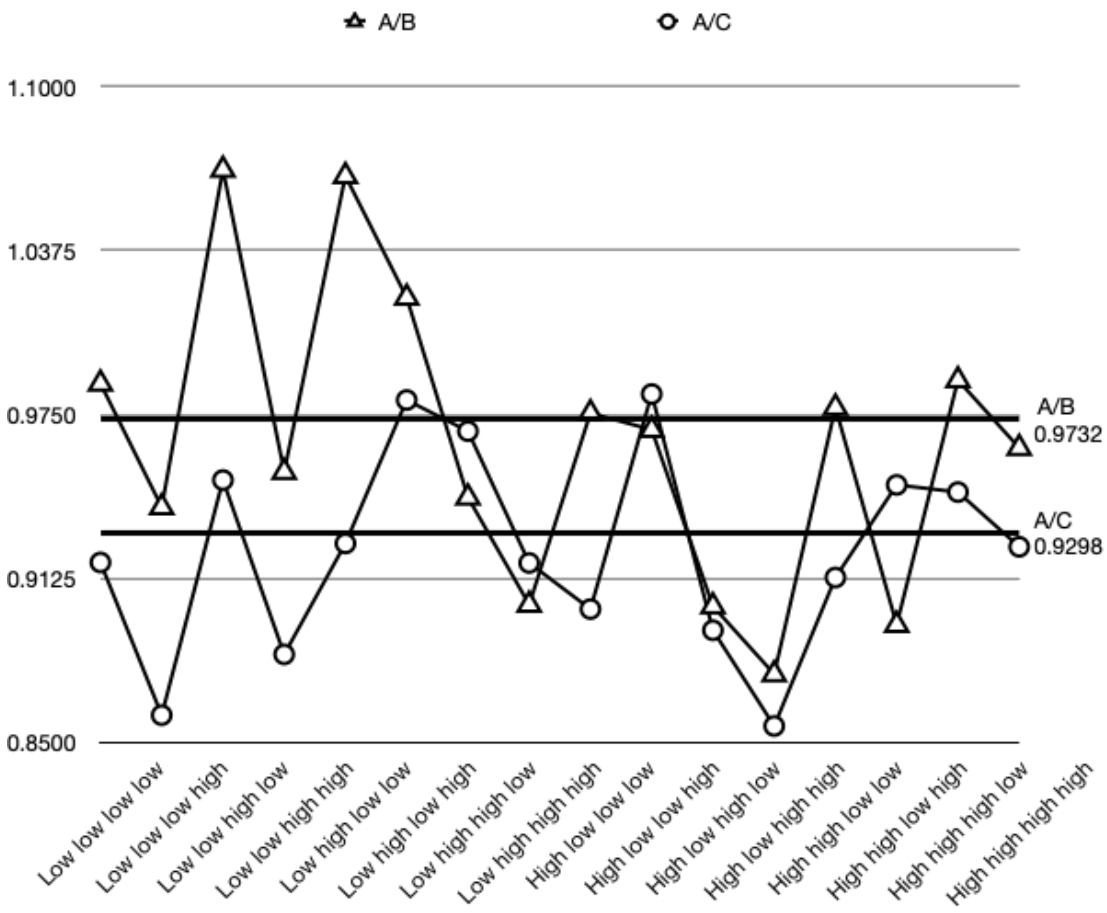


Figure 7. Comparison among the A vs B vs C

In Figure 7, we divide group A by group B $k_1 = 0.6, k_2 = 0.3, k_3 = 0.003$ and C ($k_1 = 7.2, k_2 = 2.1, k_3 = 1.2$). In the figure, the y-axis represents the fraction between group A and B and group A and C, and the x-axis is the 16 values factors of combination. Both results show that group A is smaller than group B and C as the value is less than 1 in average. The worst case happens when the scaling parameter is $k_1 = 7.2, k_2 = 2.1, k_3 = 1.2$. As the value factors are low in levels the curves are more fluctuate and as the value factors gets higher in levels the curve is more steady. Therefore, we can conclude that the value factors in higher level helps the test results remain steady. Hence in our next test, we use the best case $k_1 = 0.2, k_2 = 0.1, k_3 = 0.001$ and worst case $k_1 = 7.2, k_2 = 2.1, k_3 = 1.2$ as our k values to compare the test results with Lin and Hsieh (2013).

4.3 Effect of pairwise exchange of jobs

In this section, we compare our results with and without applying the job swap method. In many research papers, researchers apply job swap method to get an improvement for scheduling problems. In order to compare ATCSR_UP with ATCSR_Rm (Lin & Hsieh, 2013) who also uses job swap method, we apply a comparison to show that the job swap method improves ATCSR_UP.

In each combination value factors, we test 20 sets of data so 320 cases will be tested. From Figure (a) to (p) the y-axis represents the TWT and the x-axis is the number sets of data we tested. The results are shown as follows:

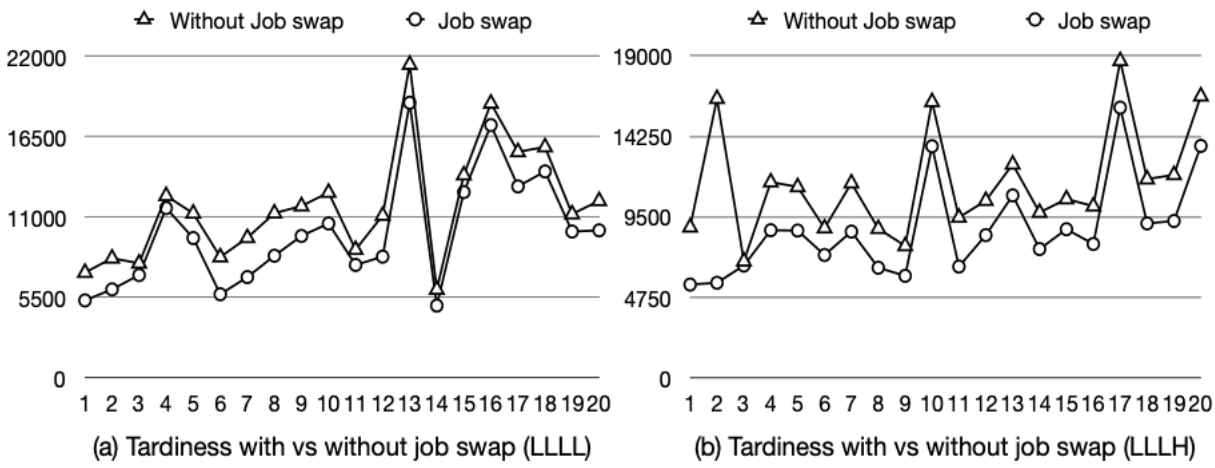


Figure 8. Tardiness comparison with vs without job swap

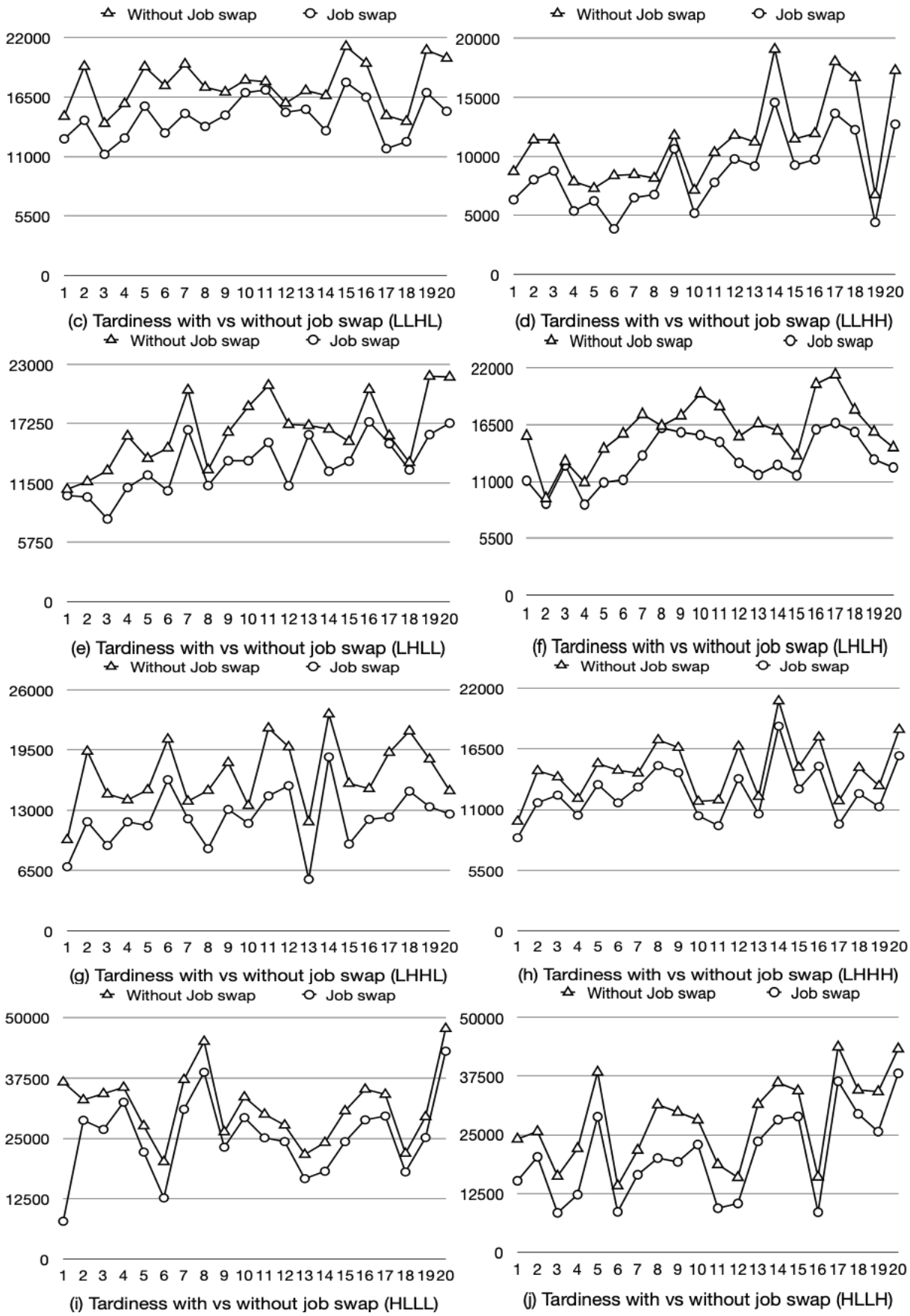


Figure 9. Tardiness comparison with vs without job swap

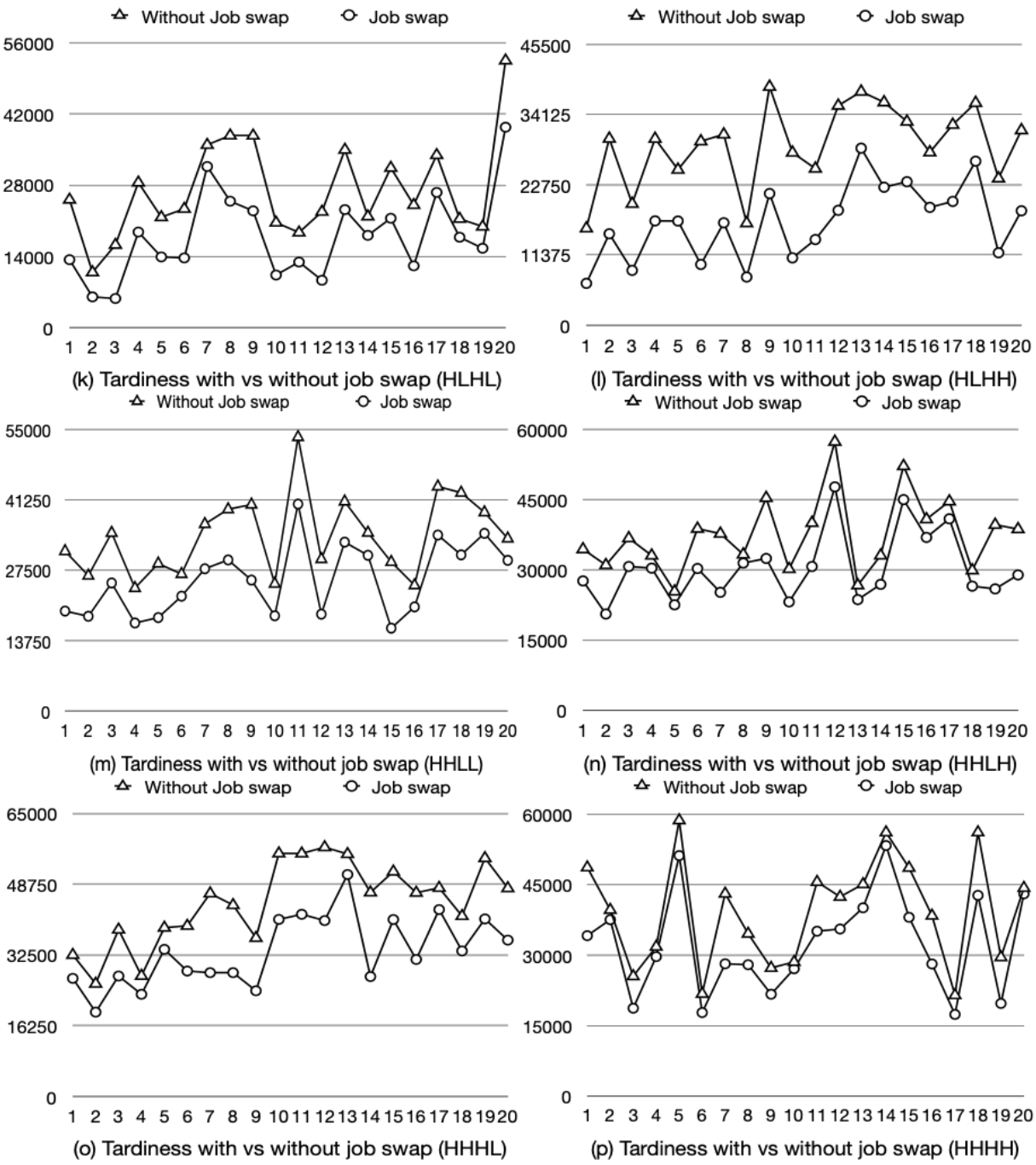


Figure 10. Tardiness comparison with vs without job swap

Figure (a) to (p) shows the overall average TWT for the 16 value factor combinations. We found out that the value factors effects the test results. The higher level the factors are the bigger the TWT is. We are interested in how the factors influence our results and how job swap method help to improve the TWT. In the test 320 cases (each value factor combinations test 20 sets of data) the TWT decreased about 24% in average. When the setup severity factor are in

low level, the average setup time is short (2% of the processing time) the TWT after applying the job swap method decrease about 20% in average and when setup severity factor are high in level (200% of the processing time) the TWT after applying the job swap method decrease about 27% in average. Under this condition, the higher the value factors get the job swap method provides a better improvement.

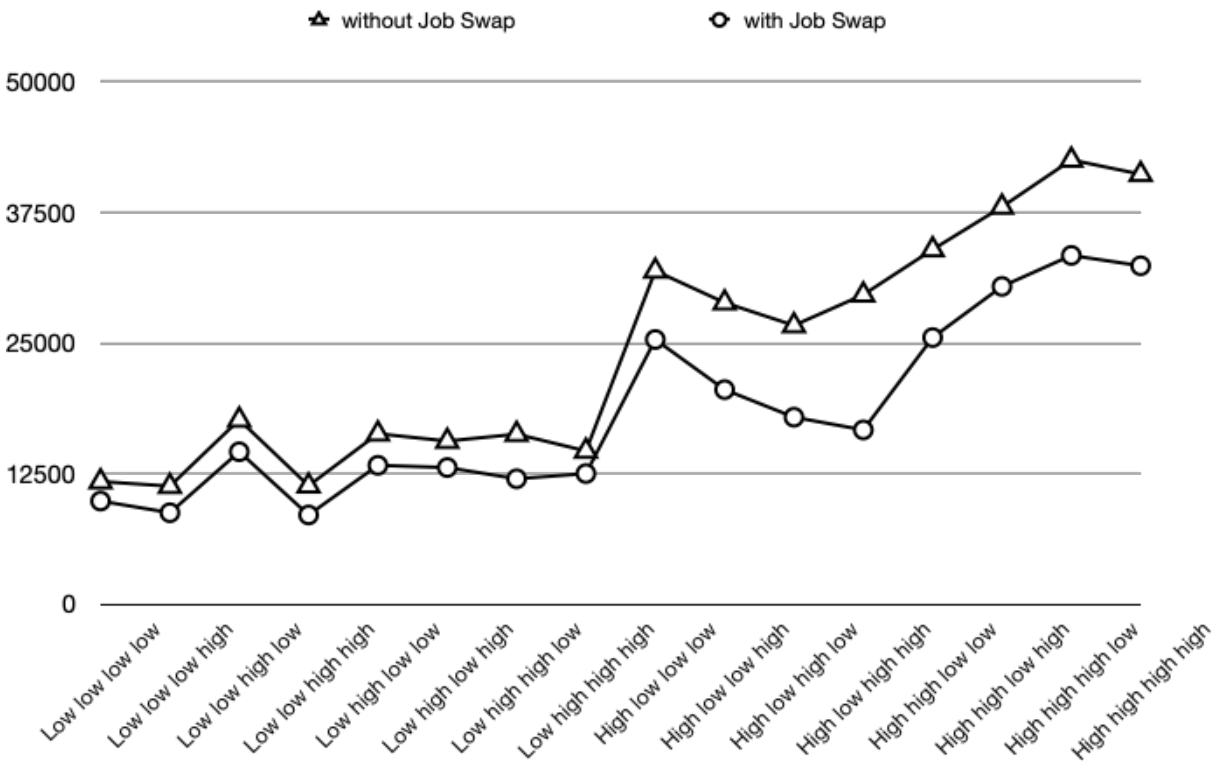


Figure 11. Average TWT result between with and without Job Swap Method

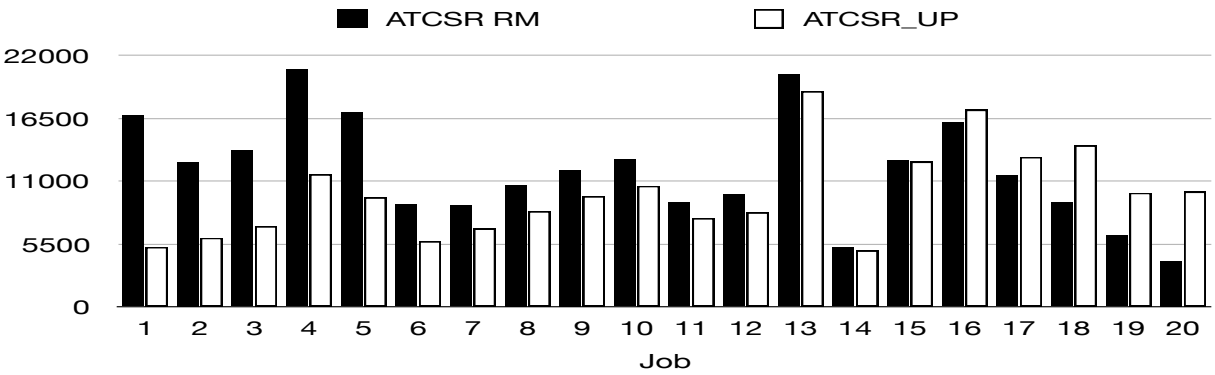
Figure 11. generates from the average TWT and the 16 combinations of value factors and shows the result of ATCSR_UP procedure with and without using the Job swap method. The y-axis shows the average TWT value of the 20 cases we test and the x-axis is the 16 combinations of value factors. We can conclude from the Figure 11. that the average TWT after applying the job swap method is better than the average TWT without applying. Furthermore, as the value factors gets high in level, the job swap method slightly decreases the TWT more. The job swap method decreases the TWT about 24%. We got this conclusion from dividing

(with job swap method − without job swap method)/(without job swap method) ∗ 100%. This proves that the job swap method helps ATCSR_UP getting a better TWT. Therefore, in the next section, we will compare the results between ATCSR_UP (with the Job swap method) and ATCSR_Rm (Lin & Hsieh, 2013) after performing the job swap procedure.

4.4 Performance comparison with ATCSR_Rm

Here we compare the results with 16 combinations of value factors on 20 groups with scale parameters $k_1 = 0.2, k_2 = 0.1, k_3 = 0.001$ and $k_1 = 7.2, k_2 = 2.1, k_3 = 1.2$ with Lin and Hsieh (2013).

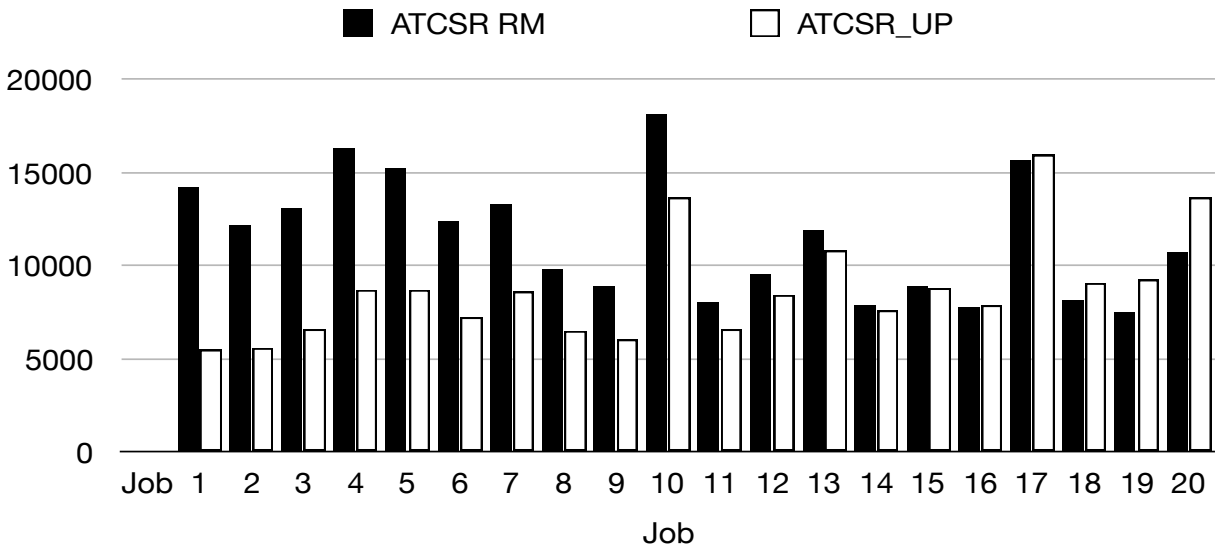
For the factors of the experiment we show in the figure caption (low low low low) represents the value factors. In each combination, we test 20 groups of data. We first show the comparsion using $k_1 = 0.2, k_2 = 0.1, k_3 = 0.001$ with ATCSR_RM (Lin & Hsieh, 2013).



ATCSR_RM vs. ATCSR_UP (Low Low Low Low)

Figure 12. Comparison on 20 instance and parameter (0.02, 0.3, 0.25 & 1)

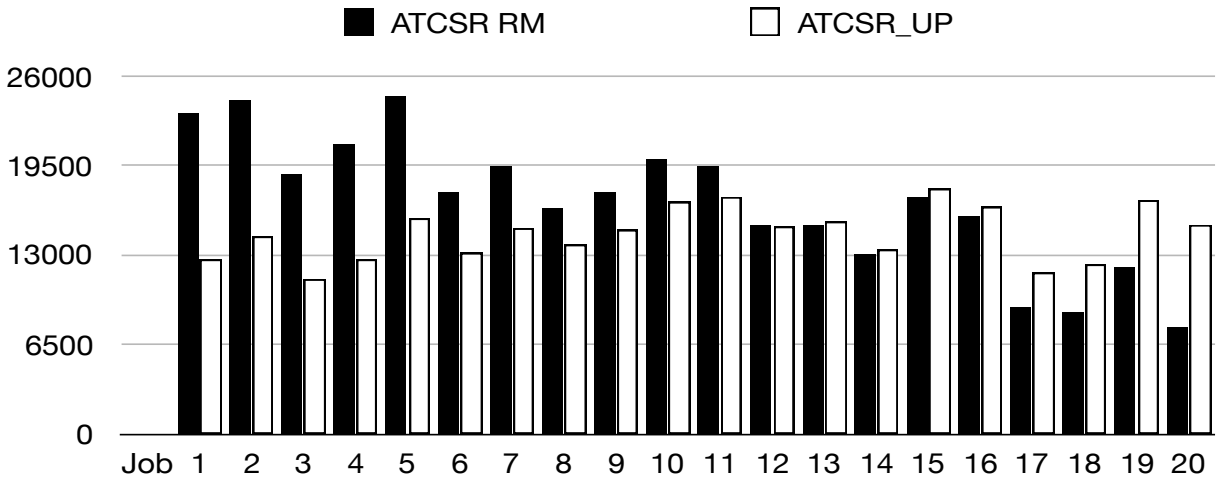
From left to right is the number of the data tested. Among the 20 results, 75% of the jobs yielded a better result and 25% a worse result than the ATCSR_Rm algorithm. The mean total weighted tardiness for our algorithm is 1.322 times better for ATCSR_UP than for ATCSR_Rm (ie. ATCSR_Rm / ATCSR_UP = 1.322).



ATCSR_RM vs. ATCSR_UP (Low Low Low High)

Figure 13. Comparison on 20 instance and parameter (0.02, 0.3, 0.25 & 10)

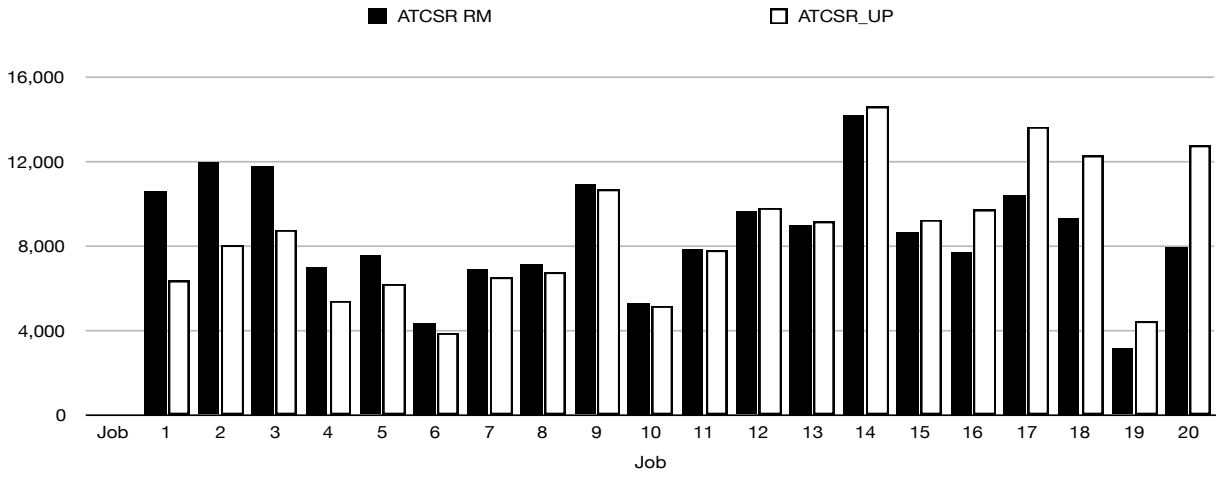
Among the 20 results, 75% of the jobs yielded a better result and 25% a worse result than the ATCSR_Rm algorithm. The mean total weighted tardiness for our algorithm is 1.39 times better for ATCSR_UP than for ATCSR_Rm (ie. ATCSR_Rm / ATCSR_UP = 1.39).



ATCSR_RM vs. ATCSR_UP (Low Low High Low)

Figure 14. Comparison on 20 instance and parameter (0.02, 0.3, 1 & 1)

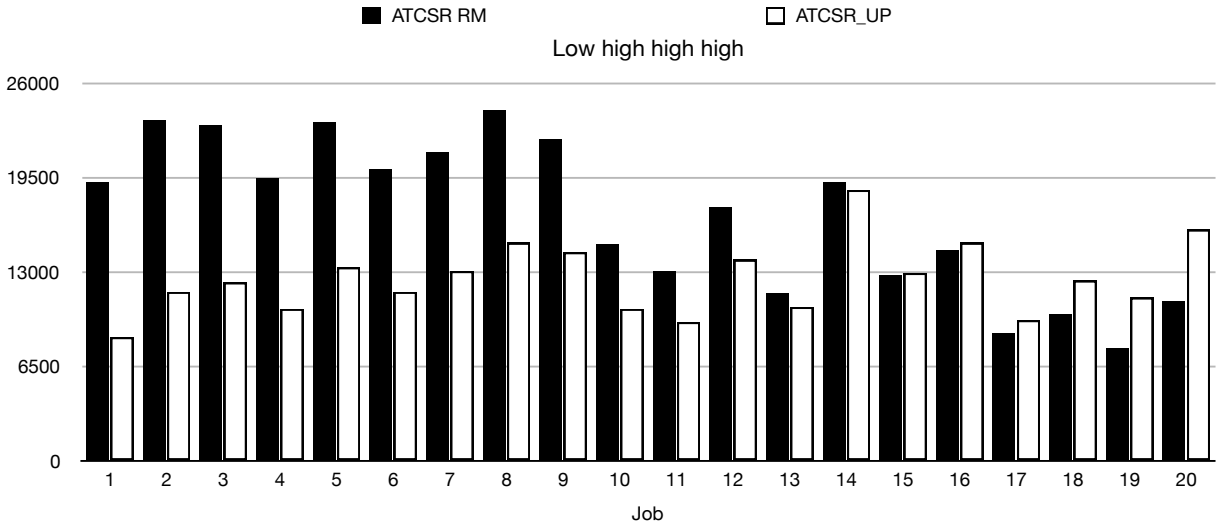
Among the 20 results, 60% of the jobs yielded a better result and 40% a worse result than the ATCSR_Rm algorithm. The mean total weighted tardiness for our algorithm is 1.168 times better for ATCSR_UP than for ATCSR_Rm (ie. ATCSR_Rm / ATCSR_UP = 1.168).



ATCSR_RM vs. ATCSR_UP (Low Low High High)

Figure 15. Comparison on 20 instance and parameter (0.02, 0.3, 1 & 10)

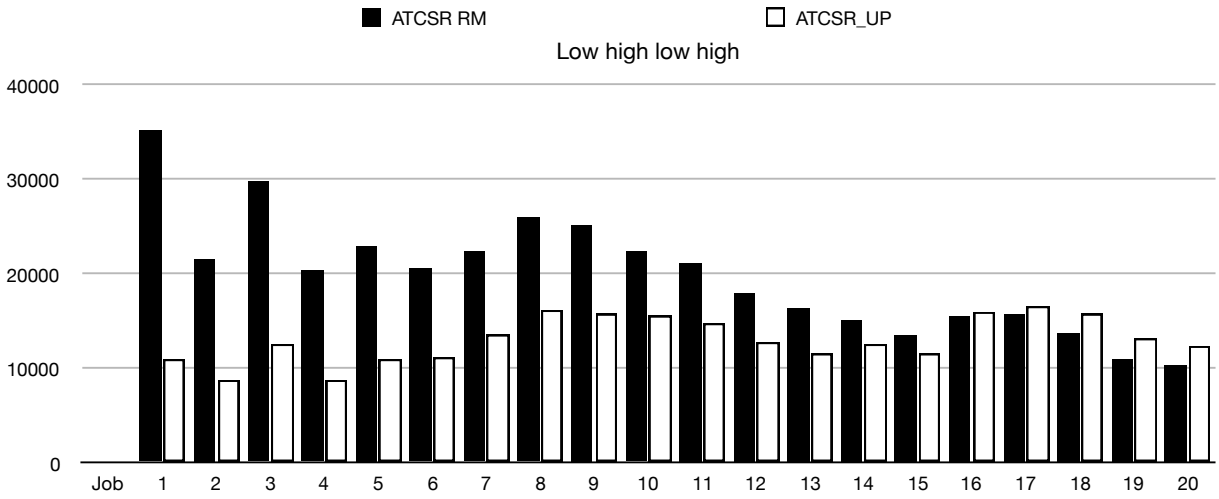
Among the 20 results, 55% of the jobs yielded a better result and 45% a worse result than the ATCSR_Rm algorithm. The mean total weighted tardiness for our algorithm is 1.043 times better for ATCSR_UP than for ATCSR_Rm (ie. ATCSR_Rm / ATCSR_UP = 1.043).



ATCSR_RM vs. ATCSR_UP (Low High High High)

Figure 16. Comparison on 20 instance and parameter (0.02, 0.9, 1 & 10)

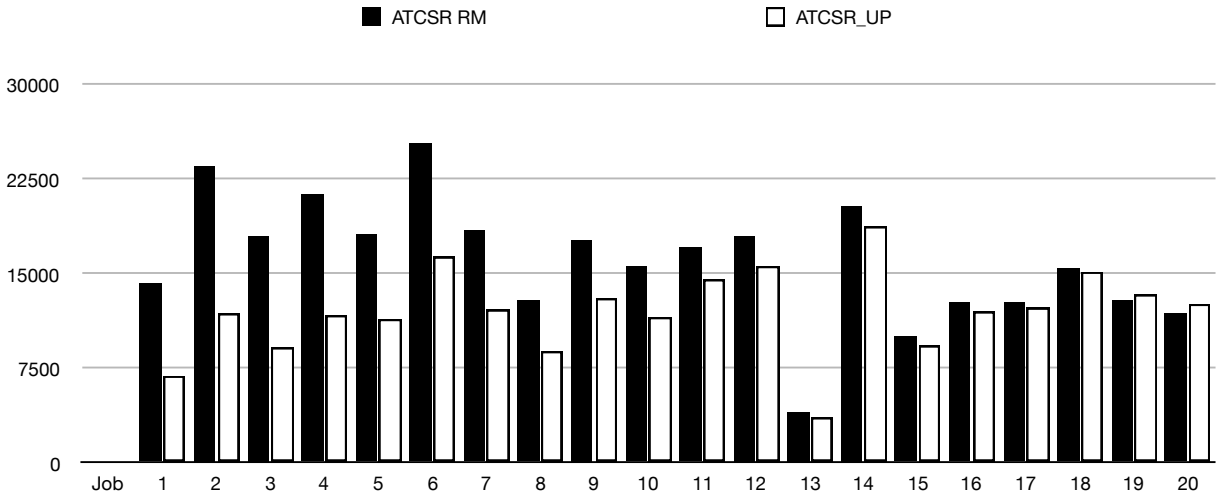
Among the 20 results, 75% of the jobs yielded a better result and 25% a worse result than the ATCSR_Rm algorithm. The mean total weighted tardiness for our algorithm is 1.37 times better for ATCSR_UP than for ATCSR_Rm (ie. ATCSR_Rm / ATCSR_UP = 1.37).



ATCSR_RM vs. ATCSR_UP (Low High Low High)

Figure 17. Comparison on 20 instance and parameter (0.02, 0.9, 0.25 & 10)

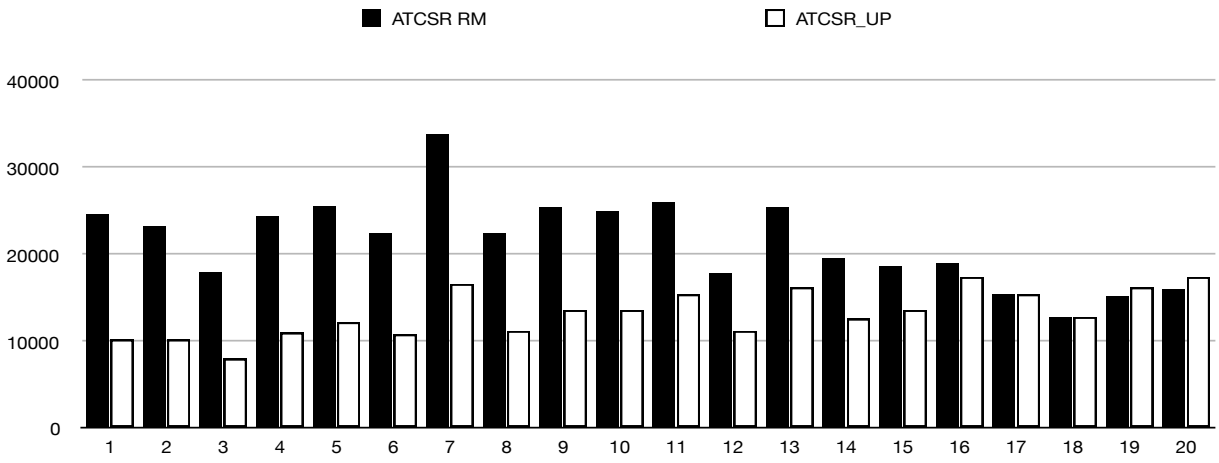
Among the 20 results, 75% of the jobs yielded a better result and 25% a worse result than the ATCSR_Rm algorithm. The mean total weighted tardiness for our algorithm is 1.585 times better for ATCSR_UP than for ATCSR_Rm (ie. ATCSR_Rm / ATCSR_UP = 1.585).



ATCSR_RM vs. ATCSR_UP (Low High High Low)

Figure 18. Comparison on 20 instance and parameter (0.02, 0.9, 1 & 1)

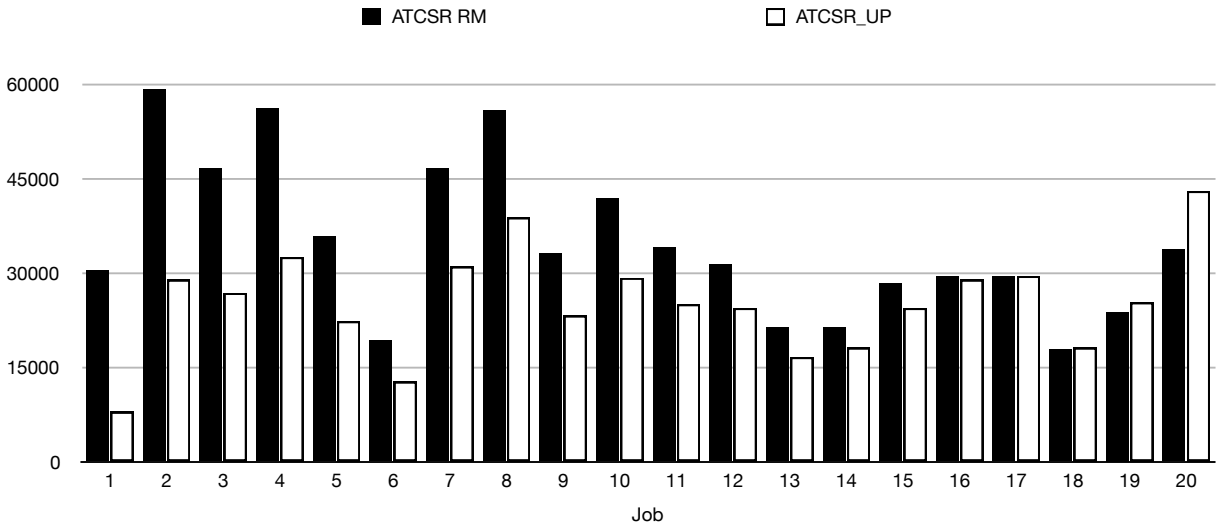
Among the 20 results, 90% of the jobs yielded a better result and 10% a worse result than the ATCSR_Rm algorithm. The mean total weighted tardiness for our algorithm is 1.36 times better for ATCSR_UP than for ATCSR_Rm (ie. ATCSR_Rm / ATCSR_UP = 1.36).



ATCSR_RM vs. ATCSR_UP (Low High Low Low)

Figure 19. Comparison on 20 instance and parameter (0.02, 0.9, 1 & 10)

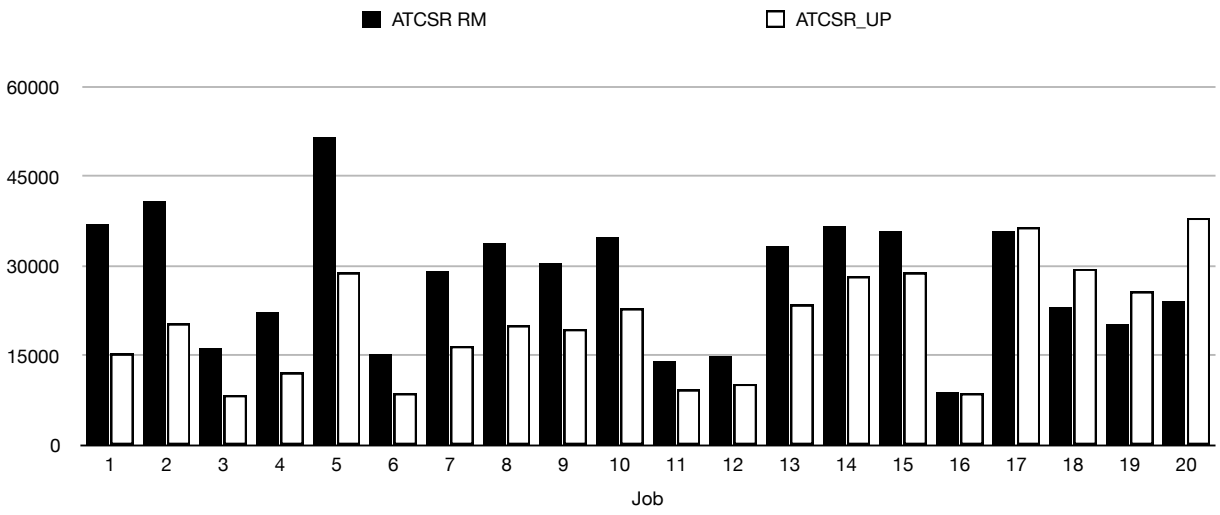
Among the 20 results, 90% of the jobs yielded a better result and 10% a worse result than the ATCSR_Rm algorithm. The mean total weighted tardiness for our algorithm is 1.71 times better for ATCSR_UP than for ATCSR_Rm (ie. ATCSR_Rm / ATCSR_UP = 1.71).



ATCSR_RM vs. ATCSR_UP (High Low Low Low)

Figure 20. Comparison on 20 instance and parameter (2, 0.3, 0.25 & 1)

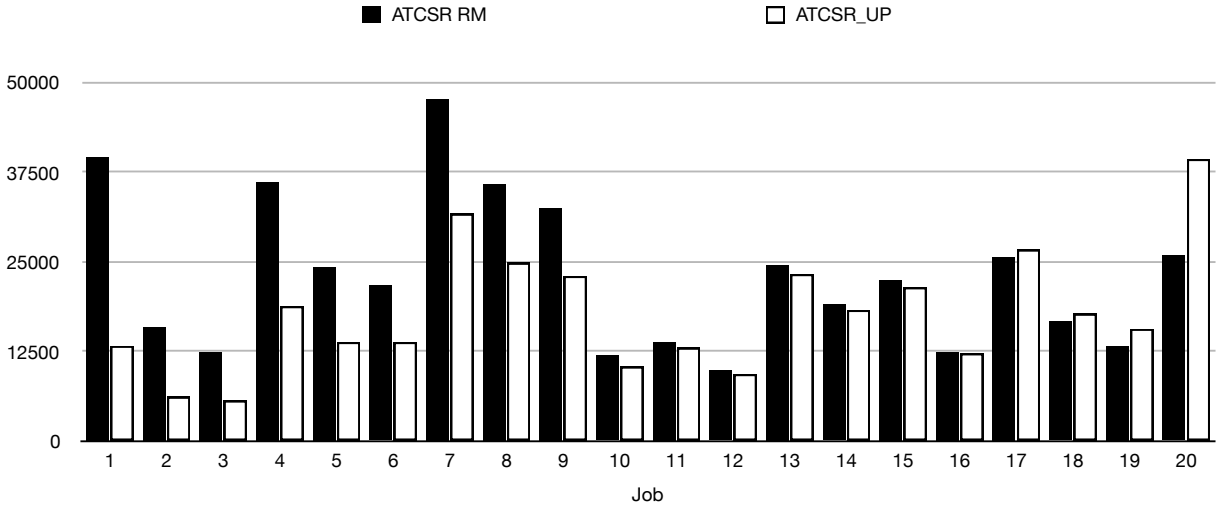
Among the 20 results, 80% of the jobs yielded a better result and 20% a worse result than the ATCSR_Rm algorithm. The mean total weighted tardiness for our algorithm is 1.467 times better for ATCSR_UP than for ATCSR_Rm (ie. ATCSR_Rm / ATCSR_UP = 1.467).



ATCSR_RM vs. ATCSR_UP (High Low Low High)

Figure 21. Comparison on 20 instance and parameter (2, 0.3, 0.25 & 10)

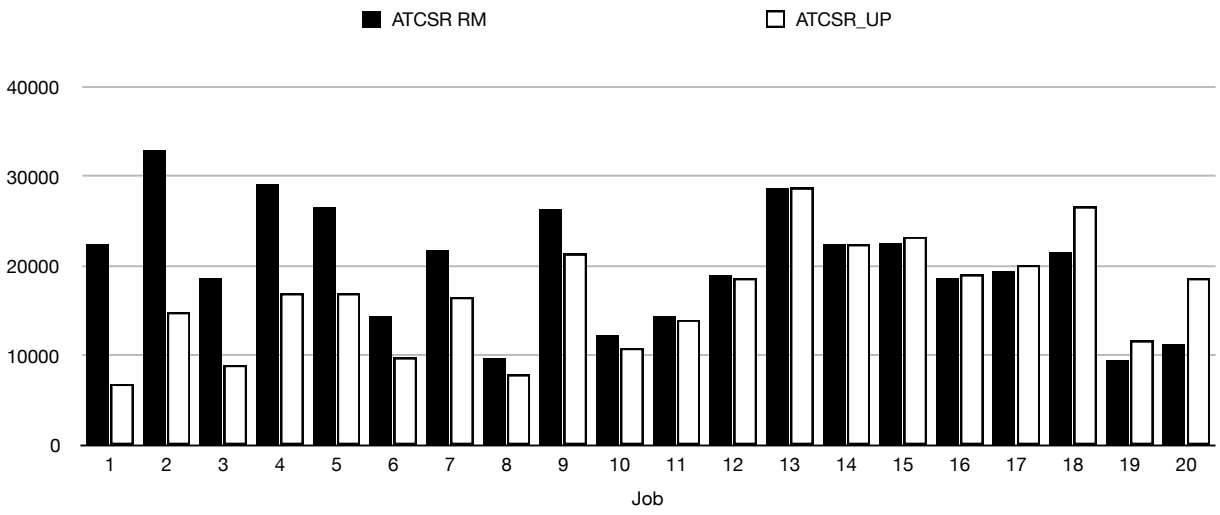
Among the 20 results, 80% of the jobs yielded a better result and 20% a worse result than the ATCSR_Rm algorithm. The mean total weighted tardiness for our algorithm is 1.475 times better for ATCSR_UP than for ATCSR_Rm (ie. ATCSR_Rm / ATCSR_UP = 1.475).



ATCSR_RM vs. ATCSR_UP (High Low High Low)

Figure 22. Comparison on 20 instance and parameter (2, 0.3, 1 & 1)

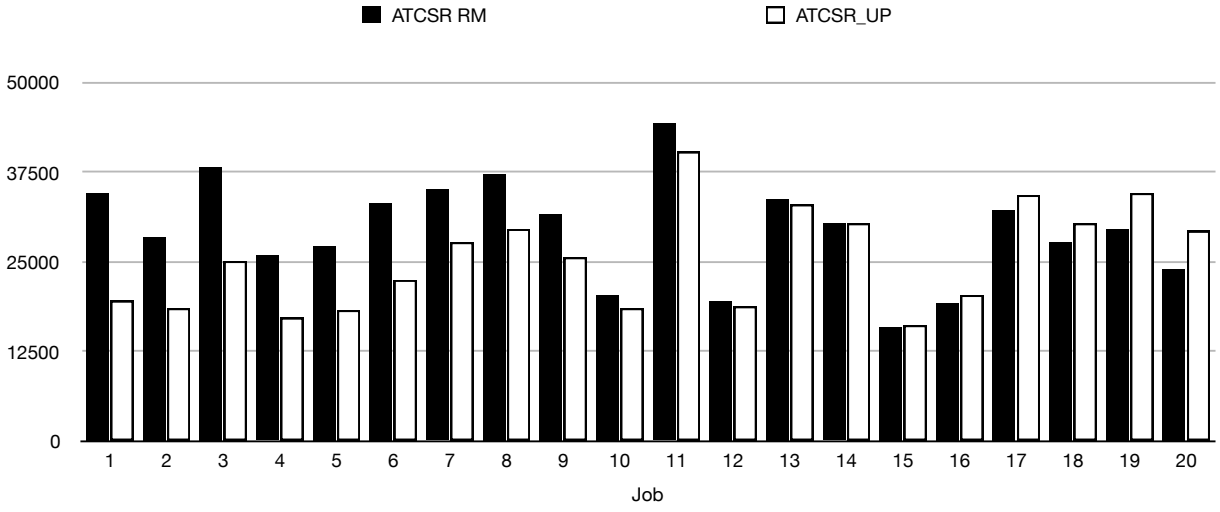
Among the 20 results, 80% of the jobs yielded a better result and 20% a worse result than the ATCSR_Rm algorithm. The mean total weighted tardiness for our algorithm is 1.41 times better for ATCSR_UP than for ATCSR_Rm (ie. ATCSR_Rm / ATCSR_UP = 1.41).



ATCSR_RM vs. ATCSR_UP (High Low High High)

Figure 23. Comparison on 20 instance and parameter (2, 0.3, 1 & 10)

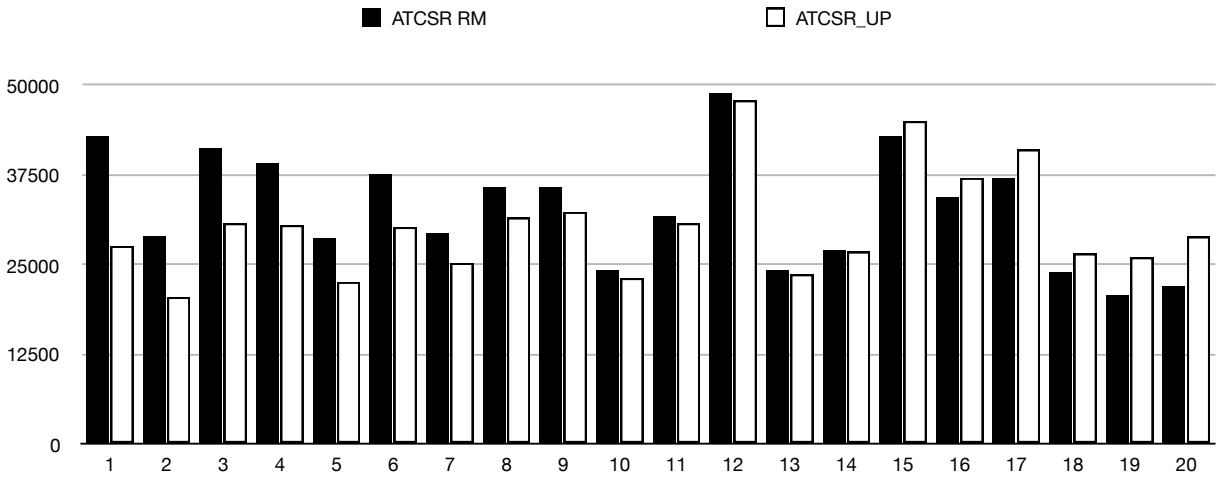
Among the 20 results, 65% of the jobs yielded a better result and 35% a worse result than the ATCSR_Rm algorithm. The mean total weighted tardiness for our algorithm is 1.32 times better for ATCSR_UP than for ATCSR_Rm (ie. ATCSR_Rm / ATCSR_UP = 1.32).



ATCSR_RM vs. ATCSR_UP (High High Low Low)

Figure 24. Comparison on 20 instance and parameter (2, 0.9, 0.25 & 1)

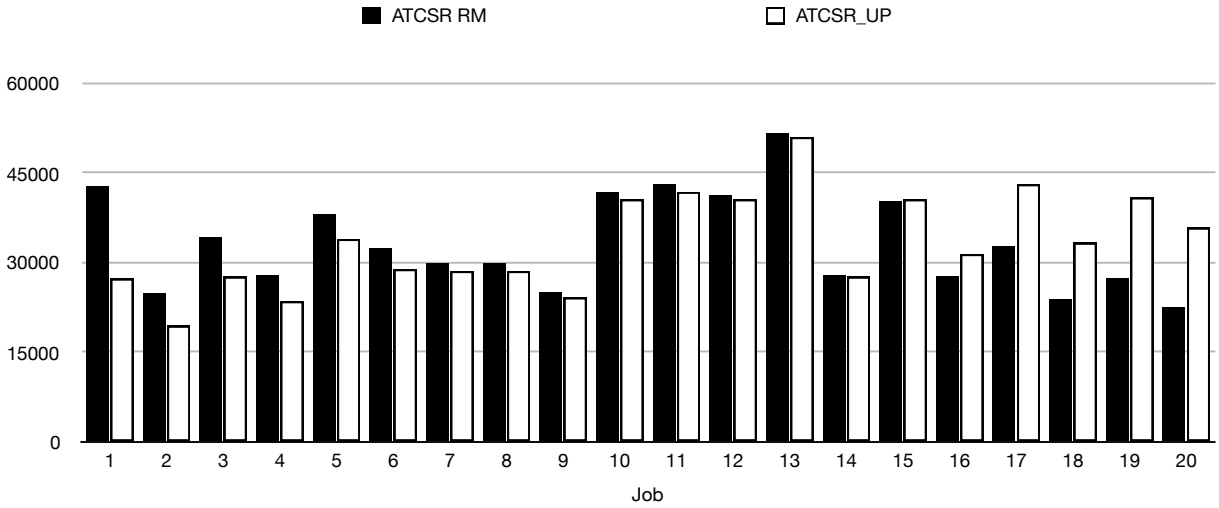
Among the 20 results, 70% of the jobs yielded a better result and 30% a worse result than the ATCSR_Rm algorithm. The mean total weighted tardiness for our algorithm is 1.19 times better for ATCSR_UP than for ATCSR_Rm (ie. ATCSR_Rm / ATCSR_UP = 1.19).



ATCSR_RM vs. ATCSR_UP (High High Low High)

Figure 25. Comparison on 20 instance and parameter (2, 0.9, 0.25 & 10)

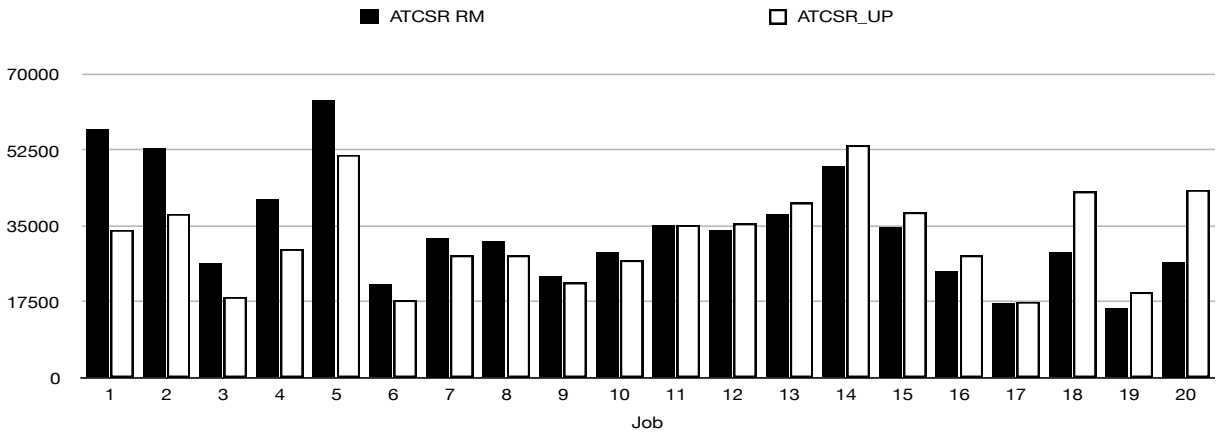
Among the 20 results, 70% of the jobs yielded a better result and 30% a worse result than the ATCSR_Rm algorithm. The mean total weighted tardiness for our algorithm is 1.094 times better for ATCSR_UP than for ATCSR_Rm (ie. ATCSR_Rm / ATCSR_UP = 1.094).m



ATCSR_RM vs. ATCSR_UP (High High High Low)

Figure 26. Comparison on 20 instance and parameter (2, 0.9, 1 & 1)

Among the 20 results, 70% of the jobs yielded a better result and 30% a worse result than the ATCSR_Rm algorithm. The mean total weighted tardiness for our algorithm is 1.02 times better for ATCSR_UP than for ATCSR_Rm (ie. ATCSR_Rm / ATCSR_UP = 1.02)

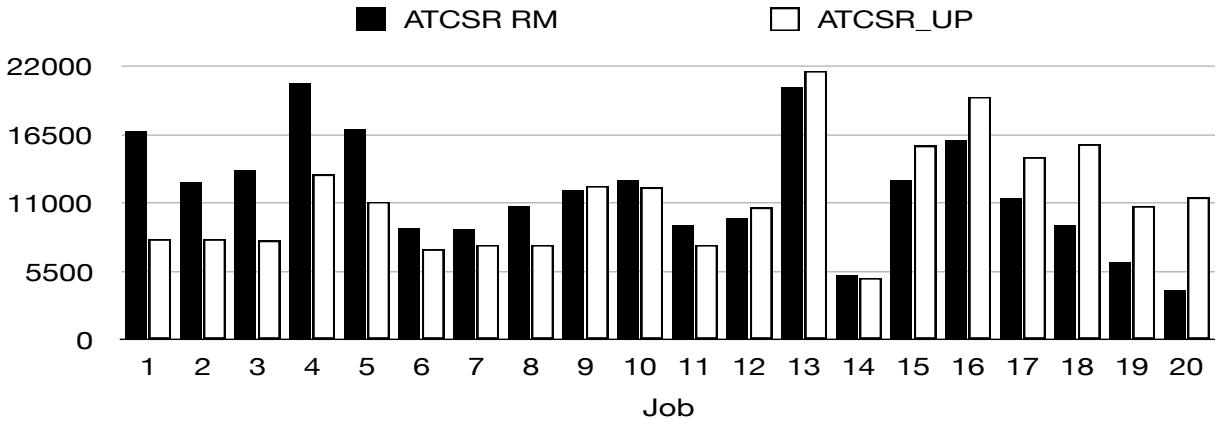


ATCSR_RM vs. ATCSR_UP (High High High High)

Figure 27. Comparison on 20 instance and parameter (2, 0.9, 1 & 10)

Among the 20 results, 55% of the jobs yielded a better result and 45% a worse result than the ATCSR_Rm algorithm. The mean total weighted tardiness for our algorithm is 1.07 times better for ATCSR_UP than for ATCSR_Rm (ie. ATCSR_Rm / ATCSR_UP = 1.07).

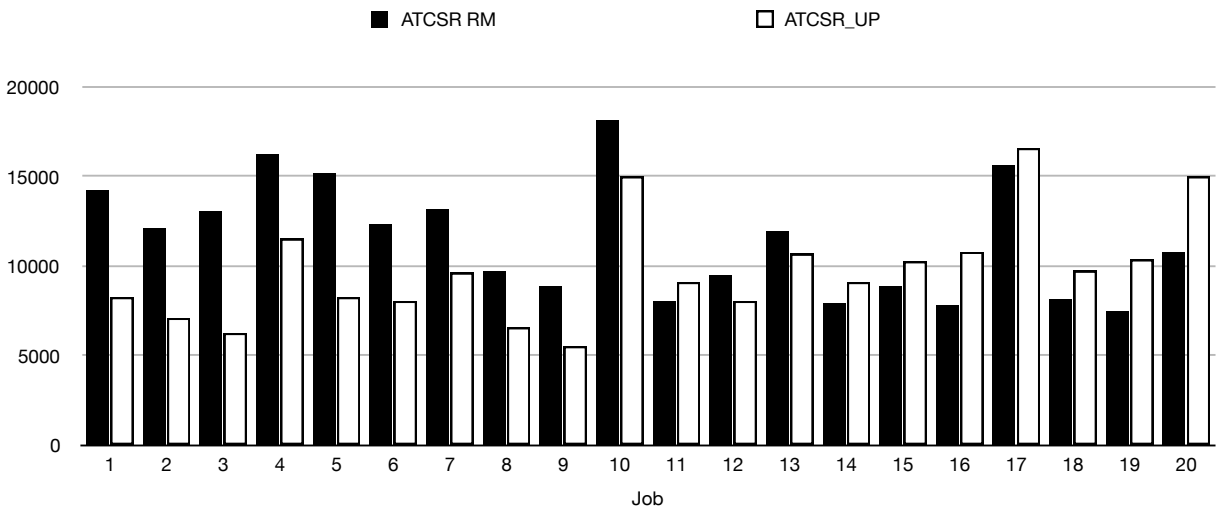
Now we show the comparison using $k_1 = 7.2, k_2 = 2.1, k_3 = 1.2$ with ATCSR_Rm (Lin & Hsieh, 2013).



ATCSR_RM vs. ATCSR_UP (Low Low Low Low)

Figure 28. Comparison on 20 instance and parameter (0.02, 0.3, 0.25 & 1)

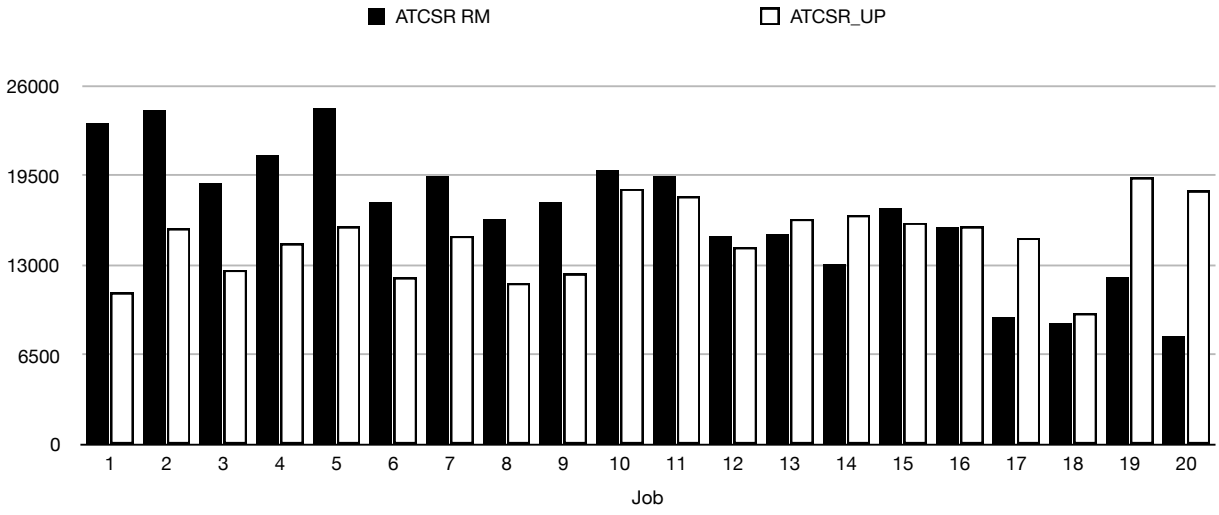
From left to right is the number of the data tested. Among the 20 results, 55% of the jobs yielded a better result and 45% a worse result than the ATCSR_Rm algorithm. The mean total weighted tardiness for our algorithm is 1.11 times better for ATCSR_UP than for ATCSR_Rm (ie. ATCSR_Rm / ATCSR_UP = 1.11).



ATCSR_RM vs. ATCSR_UP (Low Low Low High)

Figure 29. Comparison on 20 instance and parameter (0.02, 0.3, 0.25 & 10)

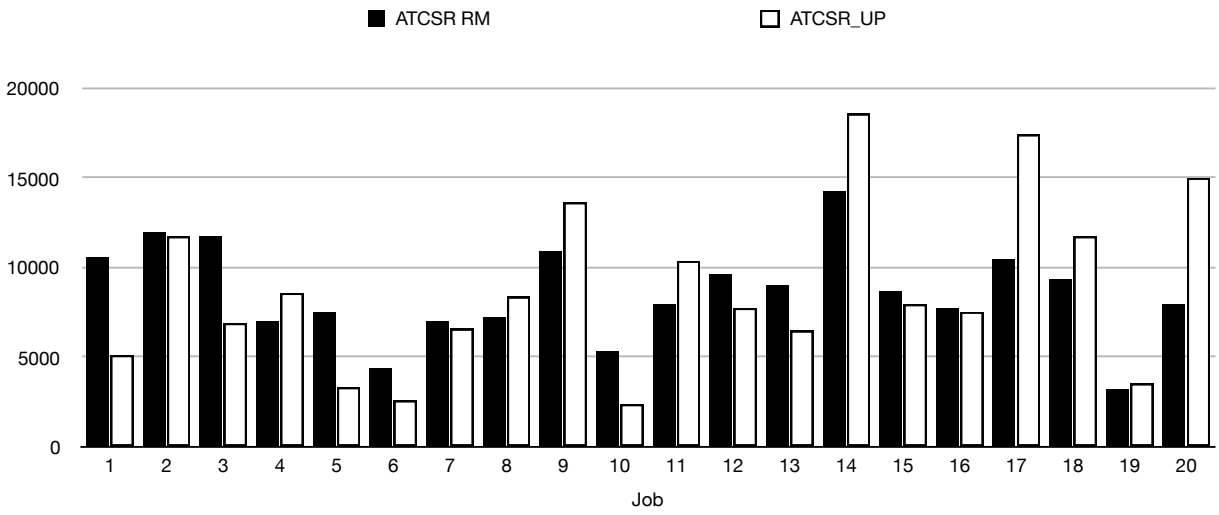
Among the 20 results, 60% of the jobs yielded a better result and 40% a worse result than the ATCSR_Rm algorithm. The mean total weighted tardiness for our algorithm is 1.24 times better for ATCCS_UP than for ATCSR_Rm (ie. ATCSR_Rm / ATCCS_UP = 1.24)



ATCSR_RM vs. ATCSR_UP (Low Low High Low)

Figure 30. Comparison on 20 instance and parameter (0.02, 0.3, 1 & 1)

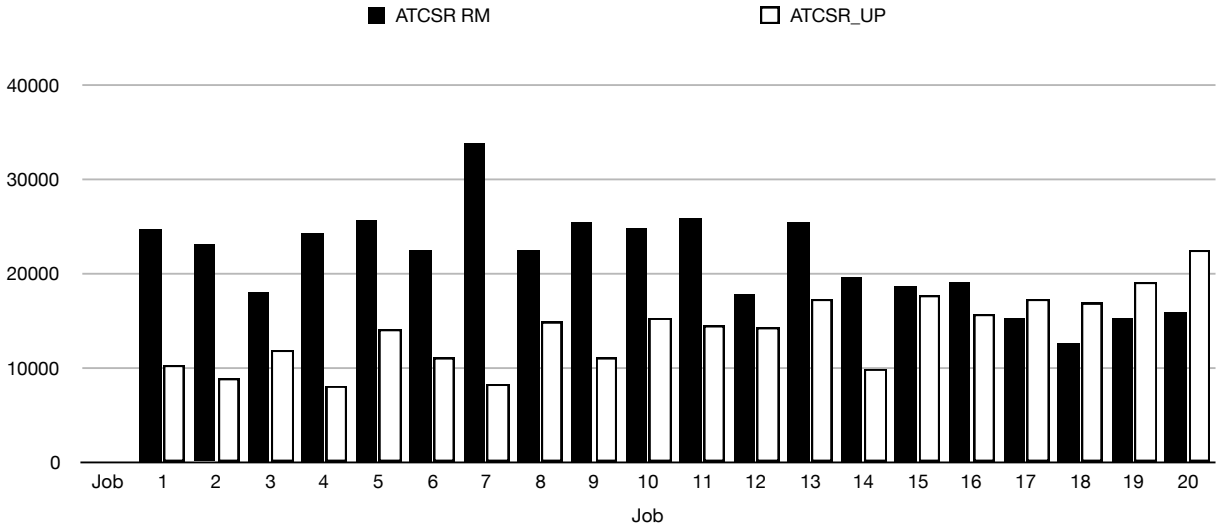
Among the 20 results, 70% of the jobs yielded a better result and 30% a worse result than the ATCSR_Rm algorithm. The mean total weighted tardiness for our algorithm is 1.16 times better for ATCCS_UP than for ATCSR_Rm (ie. ATCSR_Rm / ATCCS_UP = 1.16)



ATCSR_RM vs. ATCSR_UP (Low Low High High)

Figure 31. Comparison on 20 instance and parameter (0.02, 0.3, 1 & 10)

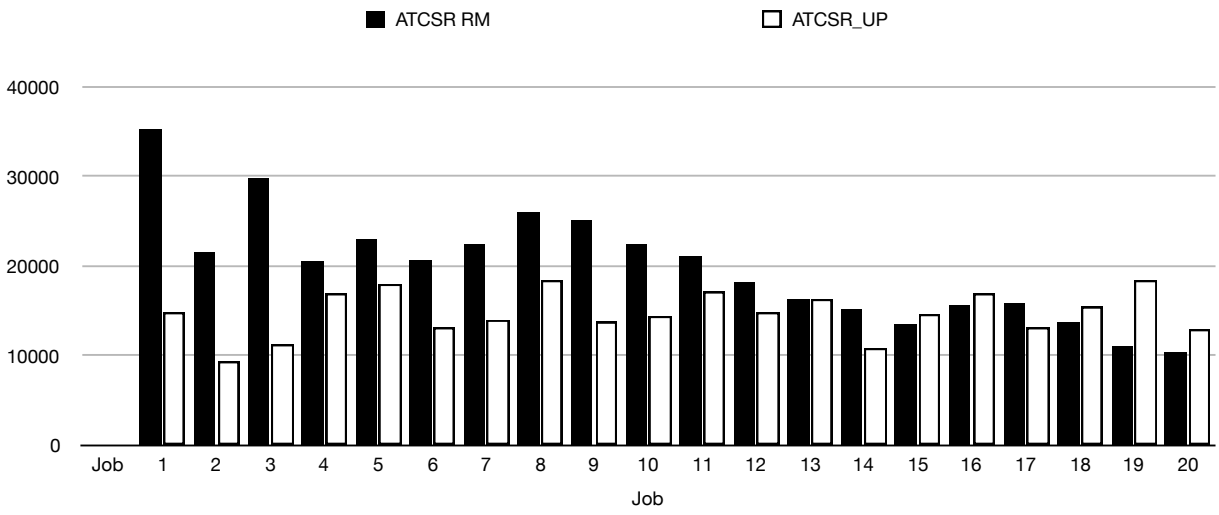
Among the 20 results, 55% of the jobs yielded a better result and 45% a worse result than the ATCSR_Rm algorithm. The mean total weighted tardiness for our algorithm is 1.183 times better for ATCCS_UP than for ATCSR_Rm (ie. ATCSR_Rm / ATCCS_UP = 1.183)



ATCSR_RM vs. ATCSR_UP (Low High Low Low)

Figure 32. Comparison on 20 instance and parameter (0.02, 0.9, 0.25 & 1)

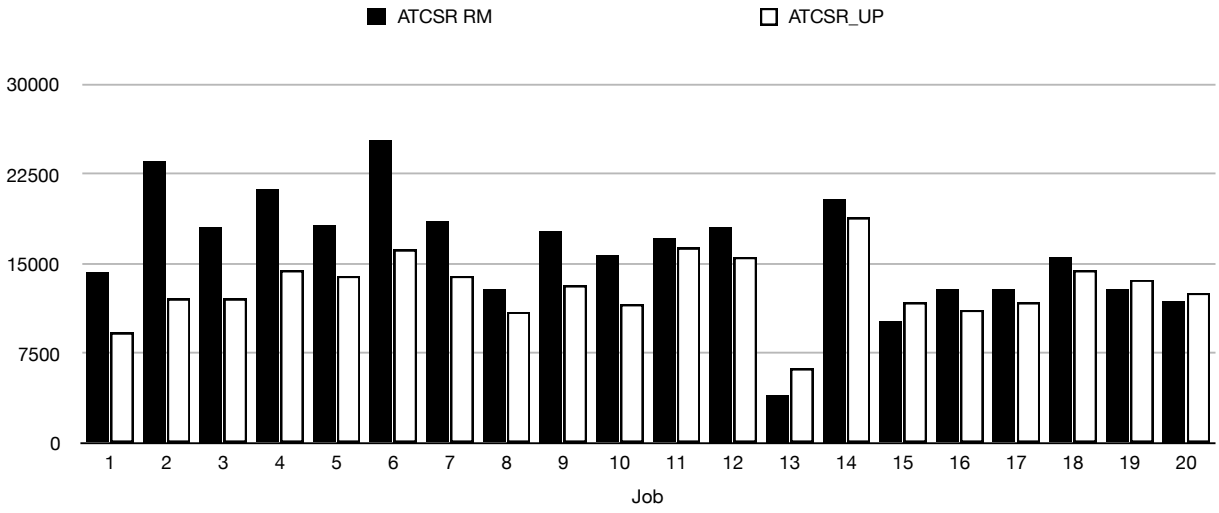
Among the 20 results, 80% of the jobs yielded a better result and 20% a worse result than the ATCSR_Rm algorithm. The mean total weighted tardiness for our algorithm is 1.73 times better for ATCCS_UP than for ATCSR_Rm (ie. ATCSR_Rm / ATCCS_UP = 1.73)



ATCSR_RM vs. ATCSR_UP (Low High Low High)

Figure 33. Comparison on 20 instance and parameter (0.02, 0.9, 0.25 & 10)

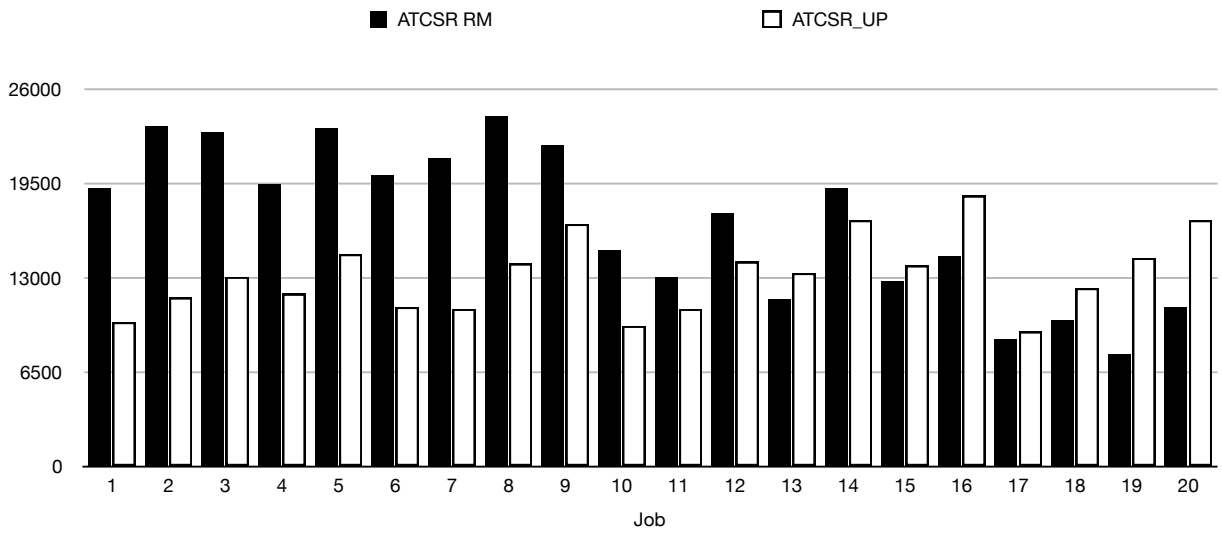
Among the 20 results, 75% of the jobs yielded a better result and 25% a worse result than the ATCSR_Rm algorithm. The mean total weighted tardiness for our algorithm is 1.4 times better for ATCCS_UP than for ATCSR_Rm (ie. ATCSR_Rm / ATCCS_UP = 1.4)



ATCSR_RM vs. ATCSR_UP (Low High High Low)

Figure 34. Comparison on 20 instance and parameter (0.02, 0.9, 1 & 1)

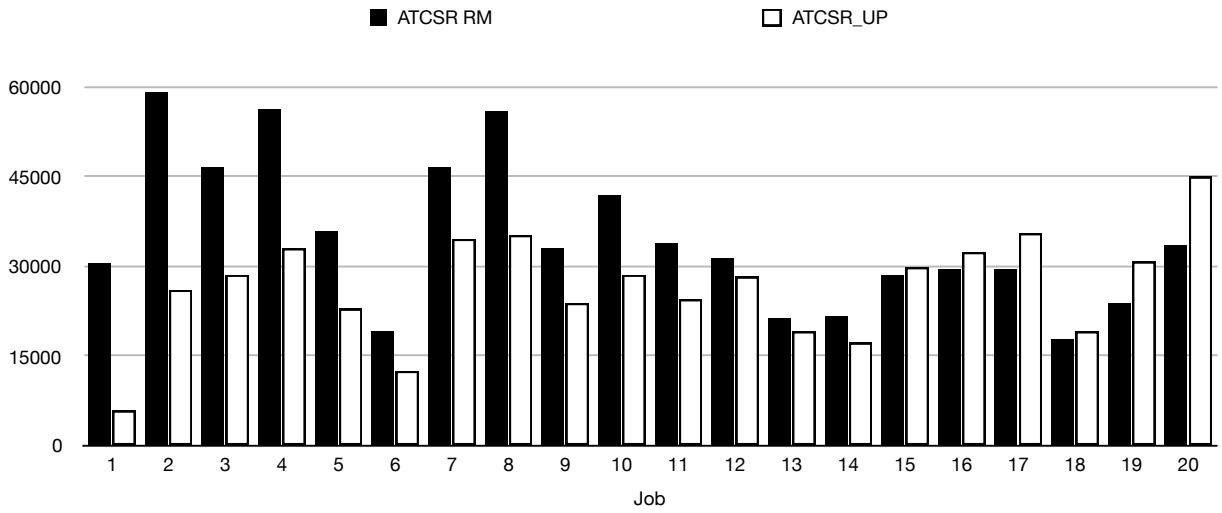
Among the 20 results, 80% of the jobs yielded a better result and 20% a worse result than the ATCSR_Rm algorithm. The mean total weighted tardiness for our algorithm is 1.22 times better for ATCCS_UP than for ATCSR_Rm (ie. ATCSR_Rm / ATCCS_UP = 1.22)



ATCSR_RM vs. ATCSR_UP (Low High High High)

Figure 35. Comparison on 20 instance and parameter (0.02, 0.9, 1 & 10)

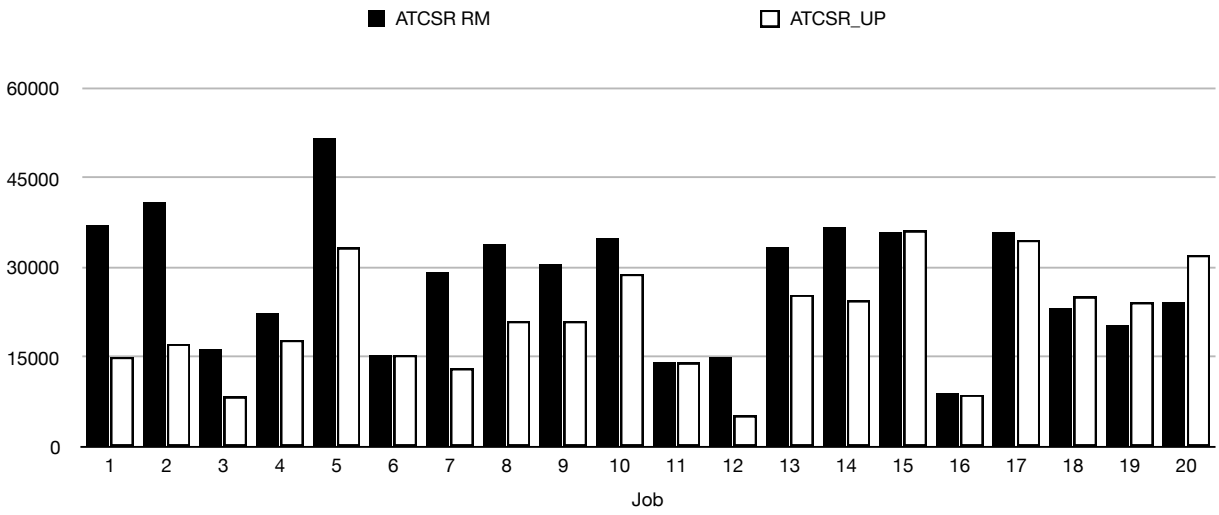
Among the 20 results, 65% of the jobs yielded a better result and 35% a worse result than the ATCSR_Rm algorithm. The mean total weighted tardiness for our algorithm is 1.31 times better for ATCCS_UP than for ATCSR_Rm (ie. ATCSR_Rm / ATCCS_UP = 1.31).



ATCSR_RM vs. ATCSR_UP (High Low Low Low)

Figure 36. Comparison on 20 instance and parameter (2, 0.3, 0.25 & 1)

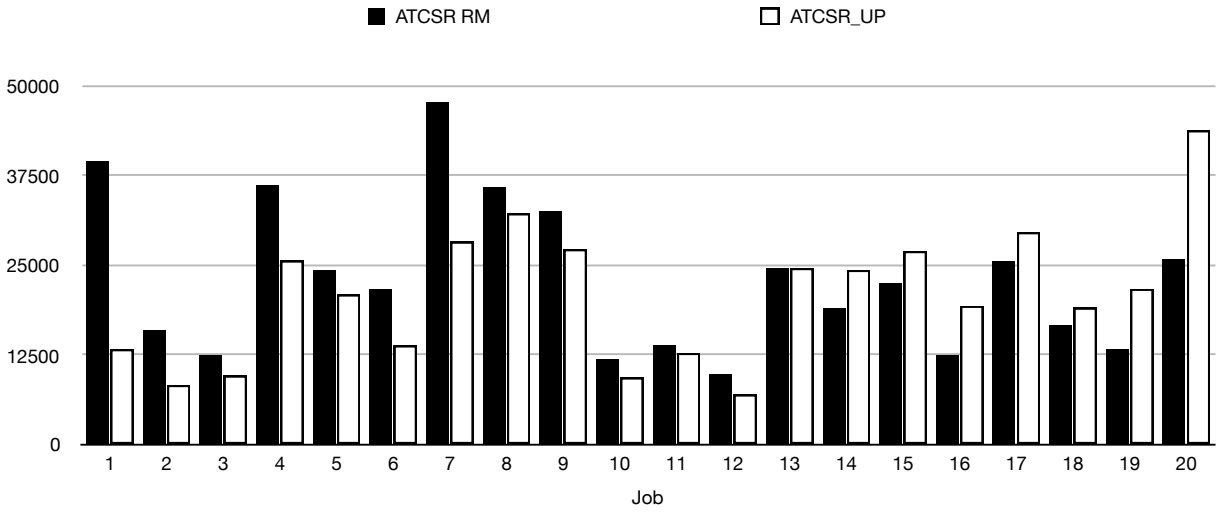
Among the 20 results, 80% of the jobs yielded a better result and 20% a worse result than the ATCSR_Rm algorithm. The mean total weighted tardiness for our algorithm is 1.37 times better for ATCCS_UP than for ATCSR_Rm (ie. ATCSR_Rm / ATCCS_UP = 1.37).



ATCSR_RM vs. ATCSR_UP (High Low Low High)

Figure 37. Comparison on 20 instance and parameter (2, 0.3, 0.25 & 10)

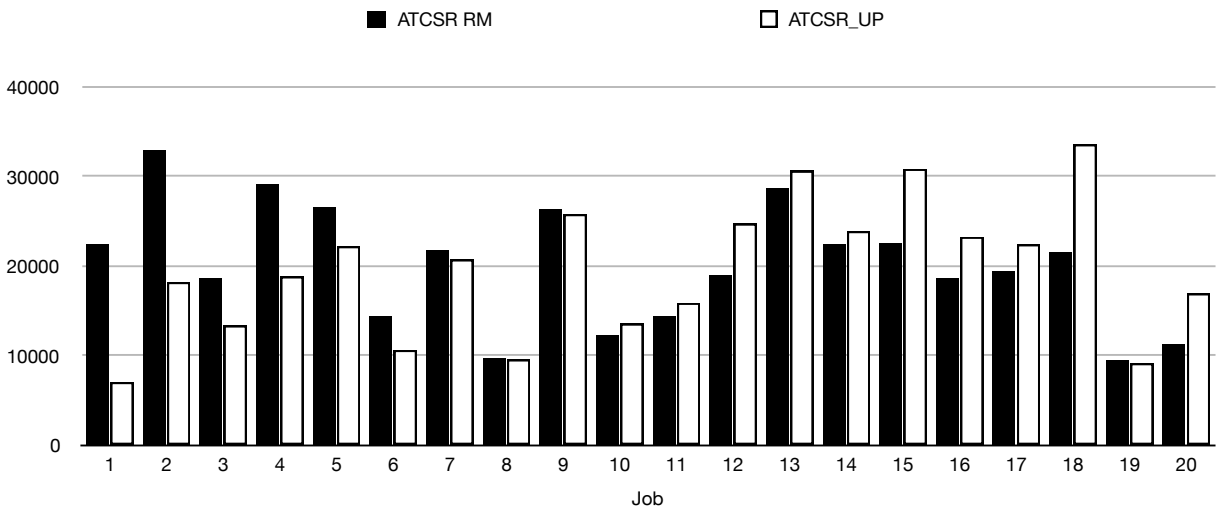
Among the 20 results, 70% of the jobs yielded a better result and 30% a worse result than the ATCSR_Rm algorithm. The mean total weighted tardiness for our algorithm is 1.46 times better for ATCCS_UP than for ATCSR_Rm (ie. ATCSR_Rm / ATCCS_UP = 1.46).



ATCSR_RM vs. ATCSR_UP (High Low High Low)

Figure 38. Comparison on 20 instance and parameter (2, 0.3, 1 & 1)

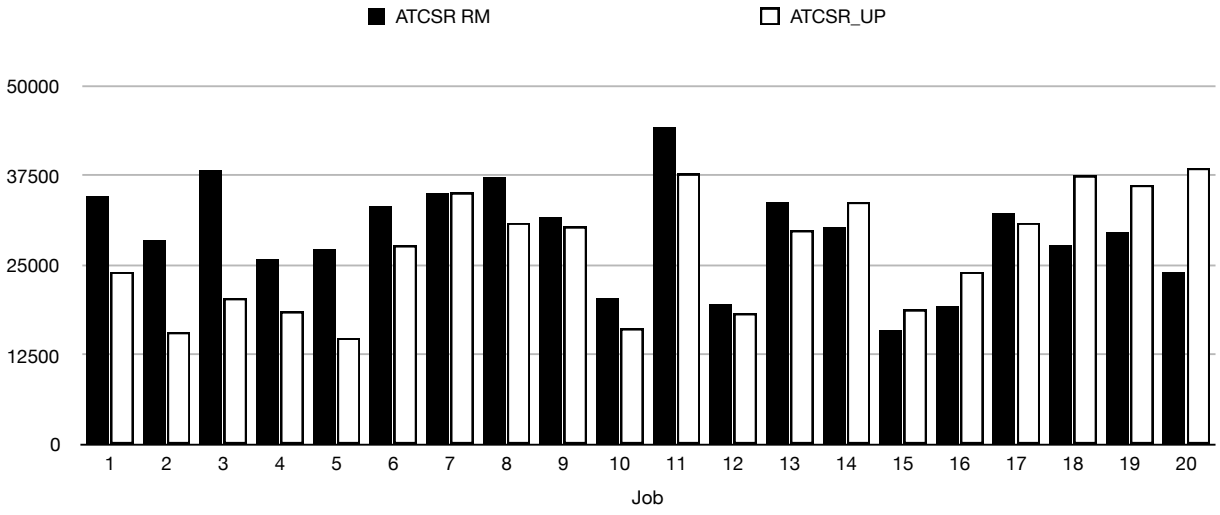
Among the 20 results, 60% of the jobs yielded a better result and 40% a worse result than the ATCSR_Rm algorithm. The mean total weighted tardiness for our algorithm is 1.21 times better for ATCCS_UP than for ATCSR_Rm (ie. ATCSR_Rm / ATCCS_UP = 1.21).



ATCSR_RM vs. ATCSR_UP (High Low High High)

Figure 39. Comparison on 20 instance and parameter (2, 0.3, 1 & 10)

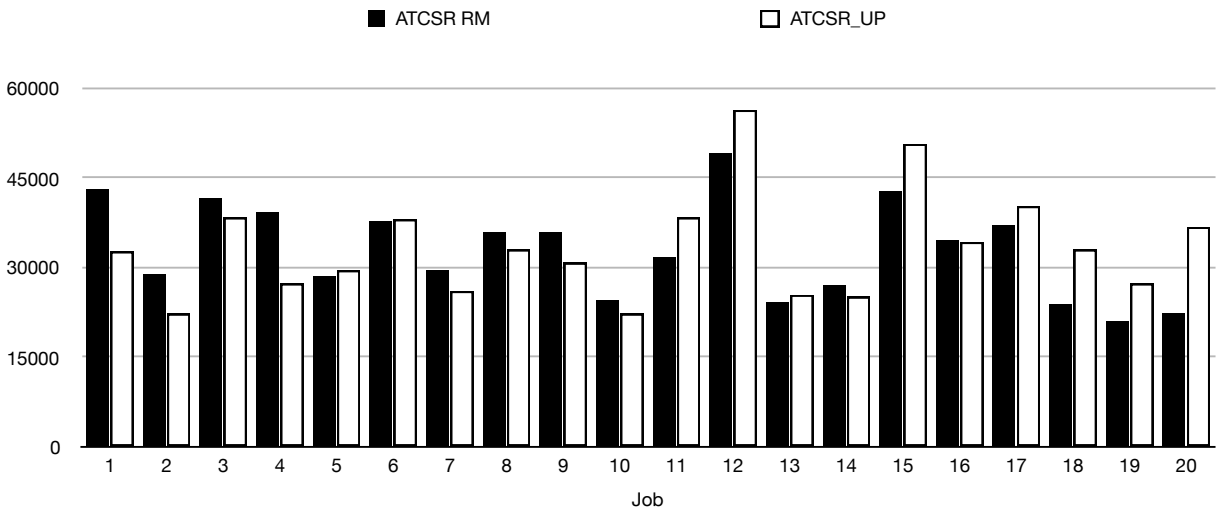
Among the 20 results, 50% of the jobs yielded a better result and 50% a worse result than the ATCSR_Rm algorithm. The mean total weighted tardiness for our algorithm is 1.14 times better for ATCCS_UP than for ATCSR_Rm (ie. ATCSR_Rm / ATCCS_UP = 1.14).



ATCSR_RM vs. ATCSR_UP (High High Low Low)

Figure 40. Comparison on 20 instance and parameter (2, 0.9, 0.25 & 1)

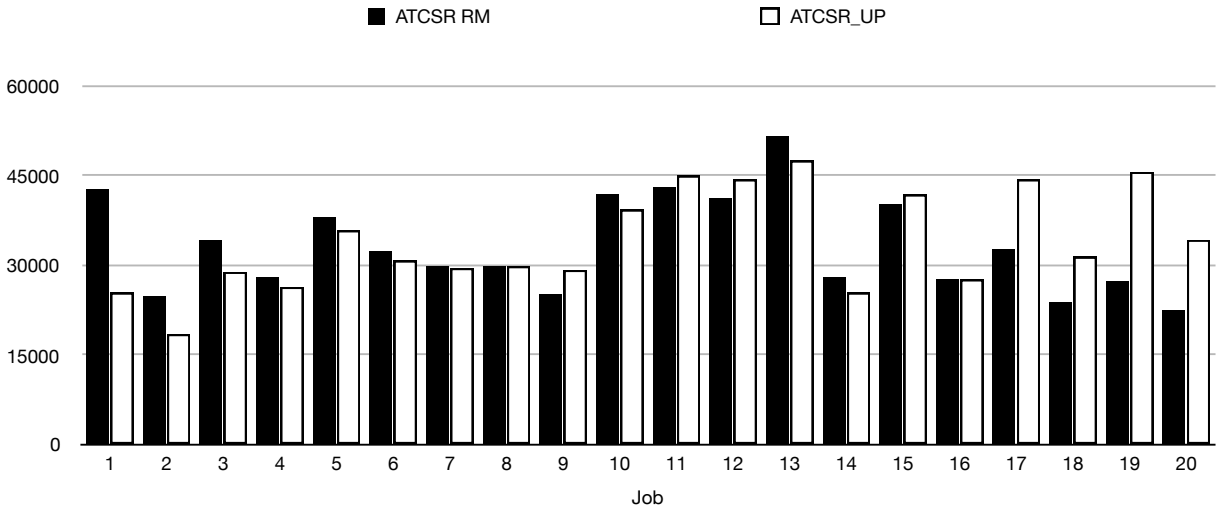
Among the 20 results, 65% of the jobs yielded a better result and 35% a worse result than the ATCSR_Rm algorithm. The mean total weighted tardiness for our algorithm is 1.16 times better for ATCCS_UP than for ATCSR_Rm (ie. ATCSR_Rm / ATCCS_UP = 1.16).



ATCSR_RM vs. ATCSR_UP (High High Low High)

Figure 41. Comparison on 20 instance and parameter (2, 0.9, 0.25 & 10)

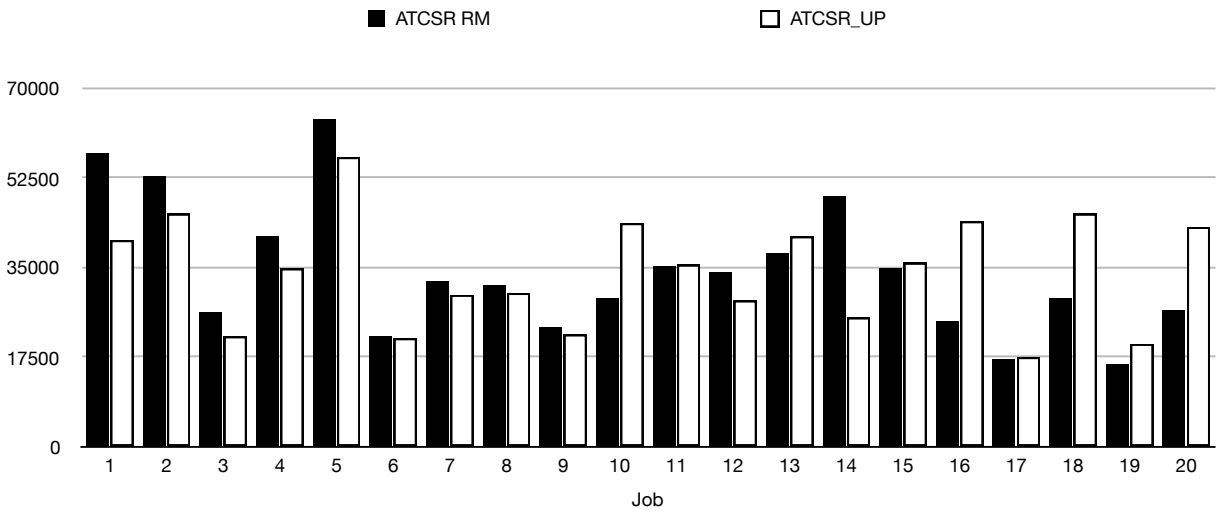
Among the 20 results, 50% of the jobs yielded a better result and 50% a worse result than the ATCSR_Rm algorithm. The mean total weighted tardiness for our algorithm is 1.009 times better for ATCCS_UP than for ATCSR_Rm (ie. ATCSR_Rm / ATCCS_UP = 1.009).



ATCSR_RM vs. ATCSR_UP (High High High Low)

Figure 42. Comparison on 20 instance and parameter (2, 0.9, 1 & 1)

Among the 20 results, 55% of the jobs yielded a better result and 45% a worse result than the ATCSR_Rm algorithm. The mean total weighted tardiness for our algorithm is 1.004 times better for ATCCS_UP than for ATCSR_Rm (ie. ATCSR_Rm / ATCCS_UP = 1.004).



ATCSR_RM vs. ATCSR_UP (High High High High)

Figure 43. Comparison on 20 instance and parameter (2, 0.9, 1 & 10)

Among the 20 results, 55% of the jobs yielded a better result and 45% a worse result than the ATCSR_Rm algorithm. The mean total weighted tardiness for our algorithm is 1.03 times better for ATCCS_UP than for ATCSR_Rm (ie. ATCSR_Rm / ATCCS_UP = 1.03).

From Figure 12. to 43. we test 320 cases for each scaling parameter $k_1 = 0.2, k_2 = 0.1, k_3 = 0.001$ and $k_1 = 7.2, k_2 = 2.1, k_3 = 1.2$. We are interested in how 16 combinations of value factor effects the TWT. We found that as the value factors gets higher in level, the TWT slightly gets higher (we conclude this in section 4.3). In the test of 320 cases, ATCSR_UP outperform ATCSR_RM in 230 cases (71%) (with the scaling parameters of $k_1 = 0.2, k_2 = 0.1, k_3 = 0.001$) and outperforms ATCSR_RM in 183 cases (57%) (with the scaling parameters of $k_1 = 7.2, k_2 = 2.1, k_3 = 1.2$). From Figure 12. to 43. we found that the two procedure (ATCSR_Rm and ATCSR_UP) gives similar results as the value factors gets higher in level. Therefore, we conclude that ATCSR_UP solves the problem better than ATCSR_Rm in low value factor level but perform about the same at high value factor level. There is no

discernible pattern of how ATCSR_UP outperforms ATCSR_RM but with the consideration of the earliest start time $\max\{r_j - t_m, 0\}$ in the WSPT term, machine time (t_m) in the slack term and ready term, the results are better than ATCSR_RM with considering these variables.

4.5 Performance evaluation by test condition

In Figure 44. We compare the average TWT between ATCSR_Rm and ATCSR_UP (with $k_1 = 0.2, k_2 = 0.1, k_3 = 0.001$). The y-axis shows the average TWT value and the x-axis is the number of sets we tested. We sum up the TWT from each value factor combinations for ATCSR_Rm, and for ATCSR_UP (best results) and average them to get Figure 44. From Figure 44. We summaries all the cases and find that the ATCSR_UP rule we propose is better than ATCSR_Rm done by Lin and Hsieh (2013). Average TWT decrease from 22367 to 18426 with the use of ATCSR_UP rule. The two rules ATCSR_Rm and ATCSR_UP performs resemblance when all value factors are in high level. However, as the factors are low in level, ATCSR_UP performs better than ATCSR_Rm. From the first case (low low low low) to (high high high high), ATCSR_UP is on average 1.21 times (71%) better than ATCSR_Rm. ATCSR_UP decreases the TWT about 17% more than ATCSR_Rm.

Figure 45. shows that all 16 combinations of value factors ATCSR_UP performs better than ATCSR_Rm done by Lin and Hsieh (2013). We use ATCSR_Rm divide by ATCSR_UP to demonastrate Figure 45. Y-axis shows the value of ATCSR_Rm divide by ATCSR_UP and x-axis is the 16 combinations of the value factors. The best result is achieved when the value factor is (low high high low) with almost 1.7 times better than ATCSR_Rm, meaning when the

due date tightness factor and the due date range factor are both high, ATCSR_UP performed a better result compare to ATCSR_Rm than the other set ups. The worst result is achieved when the value factors is (high high high high) with only 1.02 times better than ATCSR_Rm, meaning when the value factors are all in high level the two ATC-based rule perform about the same. We also found that once the setup severity factor in is high level, the fraction value between ATCSR_Rm and ATCSR_UP slightly decreases. This brings out the conclusion that ATCSR_UP solves the problem better when the value factors are low in level and performs about the same with ATCSR_Rm when value factors are high in level.

In Figure 46. We compare the average TWT between ATCSR_Rm and ATCSR_UP (with $k_1 = 7.2, k_2 = 2.1, k_3 = 1.2$). The y-axis shows the average TWT value and the x-axis is the number of example we test. We sum up the TWT from each value factors for ATCSR_Rm and ATCSR_UP (worst results) and average them to get Figure 46. From Figure 46. We summaries all the cases and find that the ATCSR_UP rule we propose is better than ATCSR_Rm done by Lin and Hsieh (2013). Average TWT decrease from 22367 to 19757 with the use of ATCSR_UP rule. From the first case (low low low low) to (high high high high), ATCSR_UP is on average 1.13 times (57%) better than ATCSR_Rm. ATCSR_UP decreases the TWT about 11% more than ATCSR_Rm.

Figure 47. shows that all 16 combinations of value factors ATCSR_UP perform better than ATCSR_Rm done by Lin and Hsieh (2013). We use ATCSR_Rm divide by ATCSR_UP to get the figure. Y-axis shows the value of ATCSR_Rm divide by ATCSR_UP and x-axis is the 16 combination of the value factors. The best result is achieved when the value factor is

(low high high low) with almost 1.4 times better than ATCSR_Rm, meaning when the due date tightness factor and the due date range factor are both high, ATCSR_UP performed a better compare result to ATCSR_Rm than the other set ups. The worst result is achieved when the value factors is (high high high high) with only 1.01 times better than ATCSR_Rm, meaning when the value factors are all in high level the two ATC-based rule perform about the same. In this comparison, the two ATC-based rule performs about the same when setup severity factor is in high level and either due date tightness or due date range factor are also in high level. We also found that once the setup severity factor in is high level, the fraction between ATCSR_Rm and ATCSR_UP slightly becomes smaller. Therefore, we conclude that the value factors influence the relations between ATCSR_Rm and ATCSR_UP.

In Figure 48. we compare the average TWT between ATCSR_Rm and ATCSR_UP with $(k_1 = 0.2, k_2 = 0.1, k_3 = 0.001)$ and $(k_1 = 7.2, k_2 = 2.1, k_3 = 1.2)$. We use the average TWT of 16 value factor combinations for ATCSR_Rm, ATCSR_UP (best k value) and ATCSR_UP (worst k value) to demonstrate Figure 48. The y-axis shows the average TWT value and the x-axis is the factor values we test. We find out that with the use of both scaling parameter (best and worst), ATCSR_UP outperforms ATCSR_Rm by 71% and 57%. Furthermore, when we use the best k value to compare the average TWT, the curve tends to be steadier than the worst k value. With the above revises, we conclude that ATCSR_UP is better than ATCSR_Rm in minimizing the total weighted tardiness.

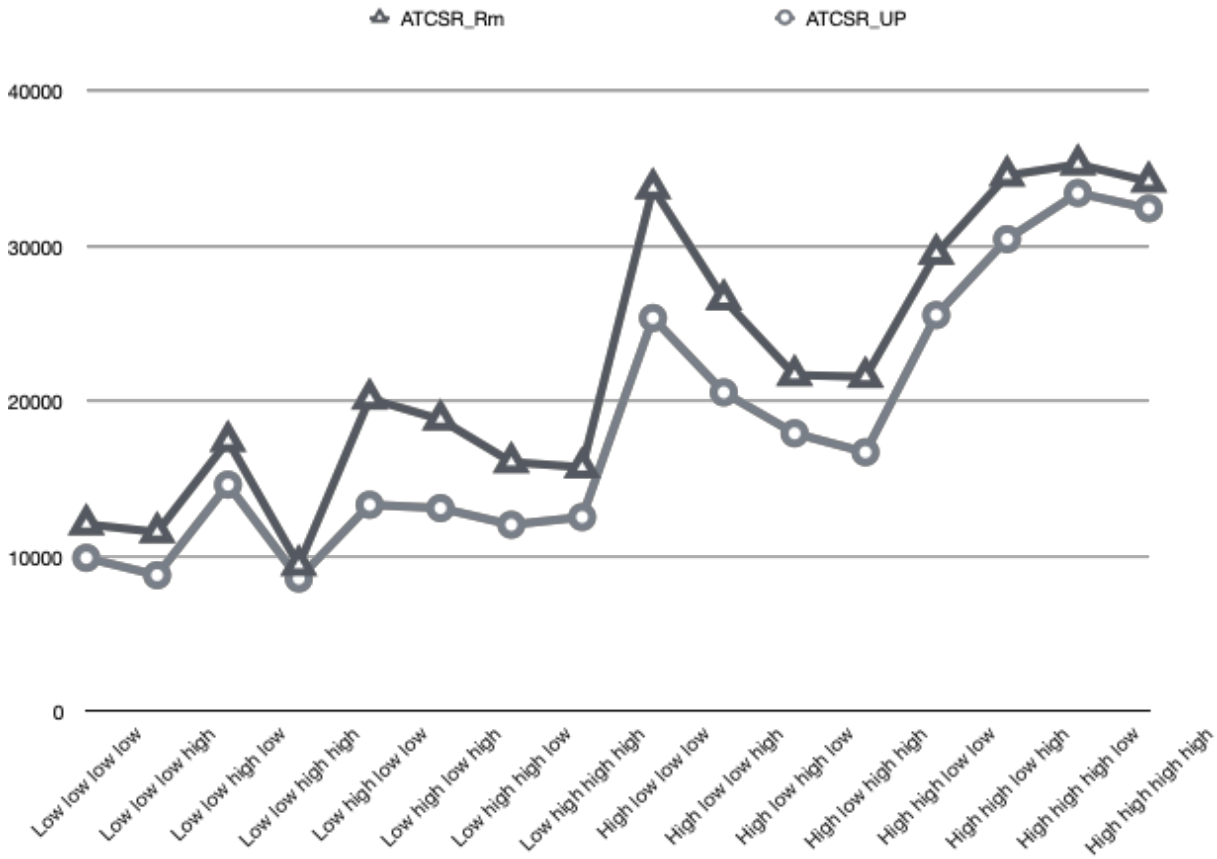


Figure 44. Overall average TWT comparison between ATCSR_Rm and ATCSR_UP

$(k_1 = 0.2, k_2 = 0.1, k_3 = 0.001)$

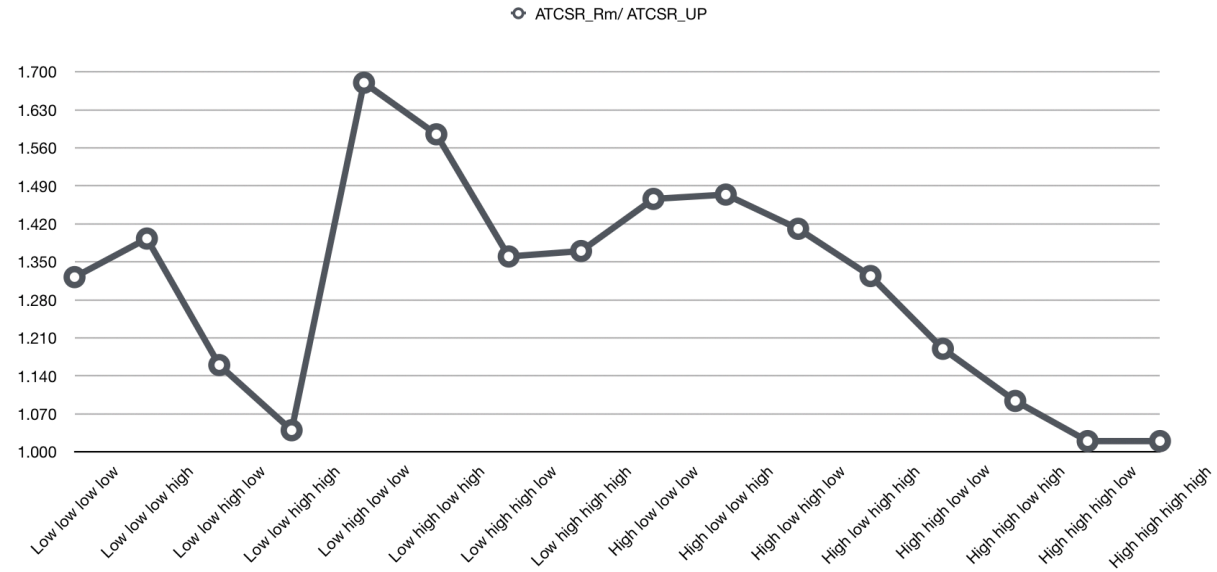


Figure 45. Overall comparison of 16 factor combination

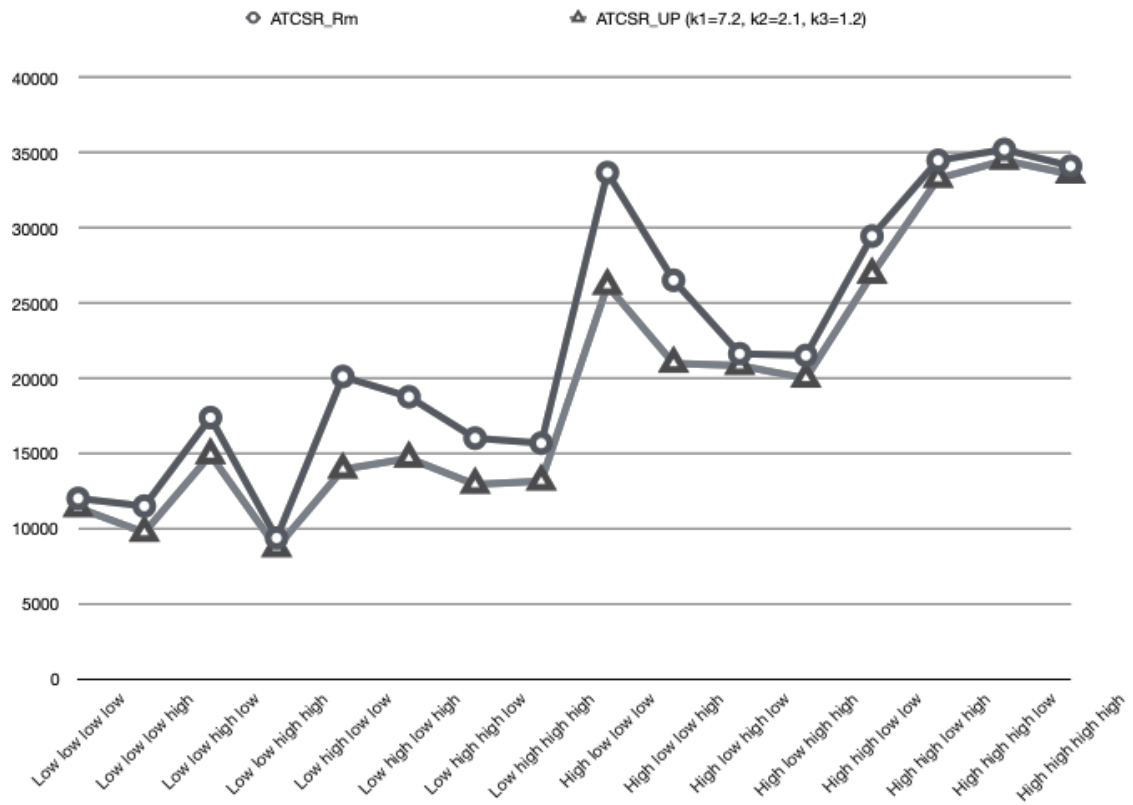


Figure 46. Overall average TWT comparison between ATCSR_Rm and ATCSR_UP

$(k_1 = 7.2, k_2 = 2.1, k_3 = 1.2)$

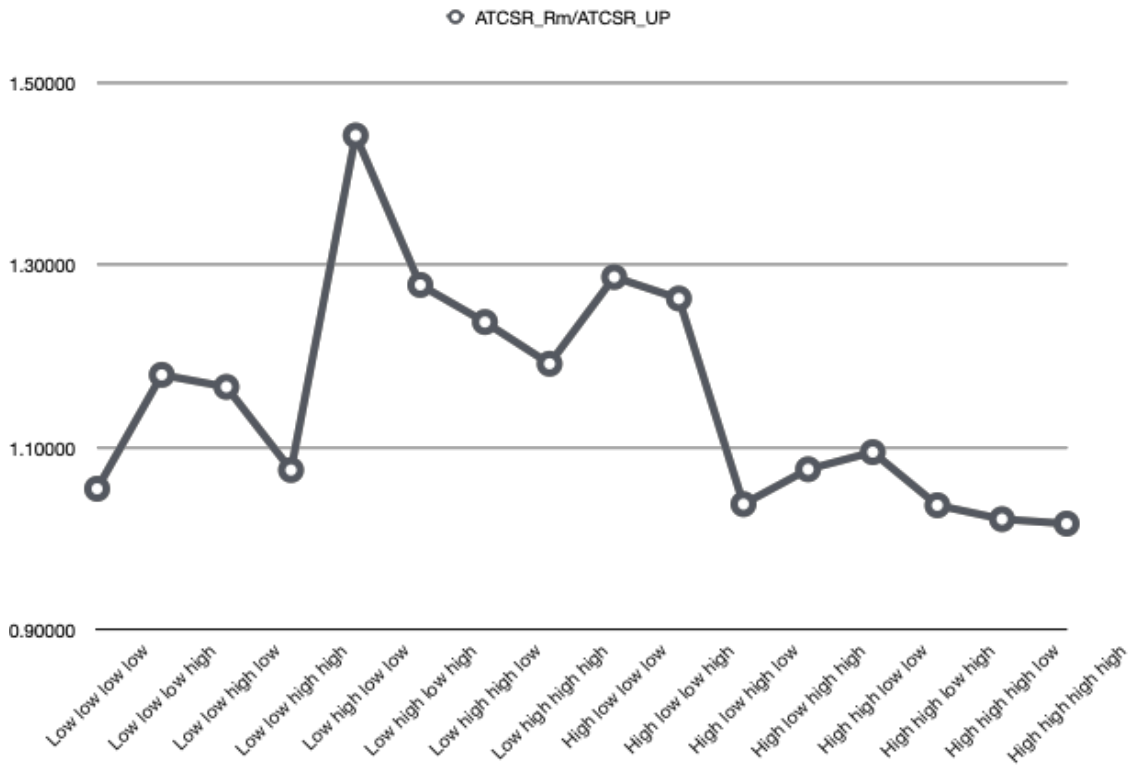


Figure 47. Overall comparison of 16 factor combination

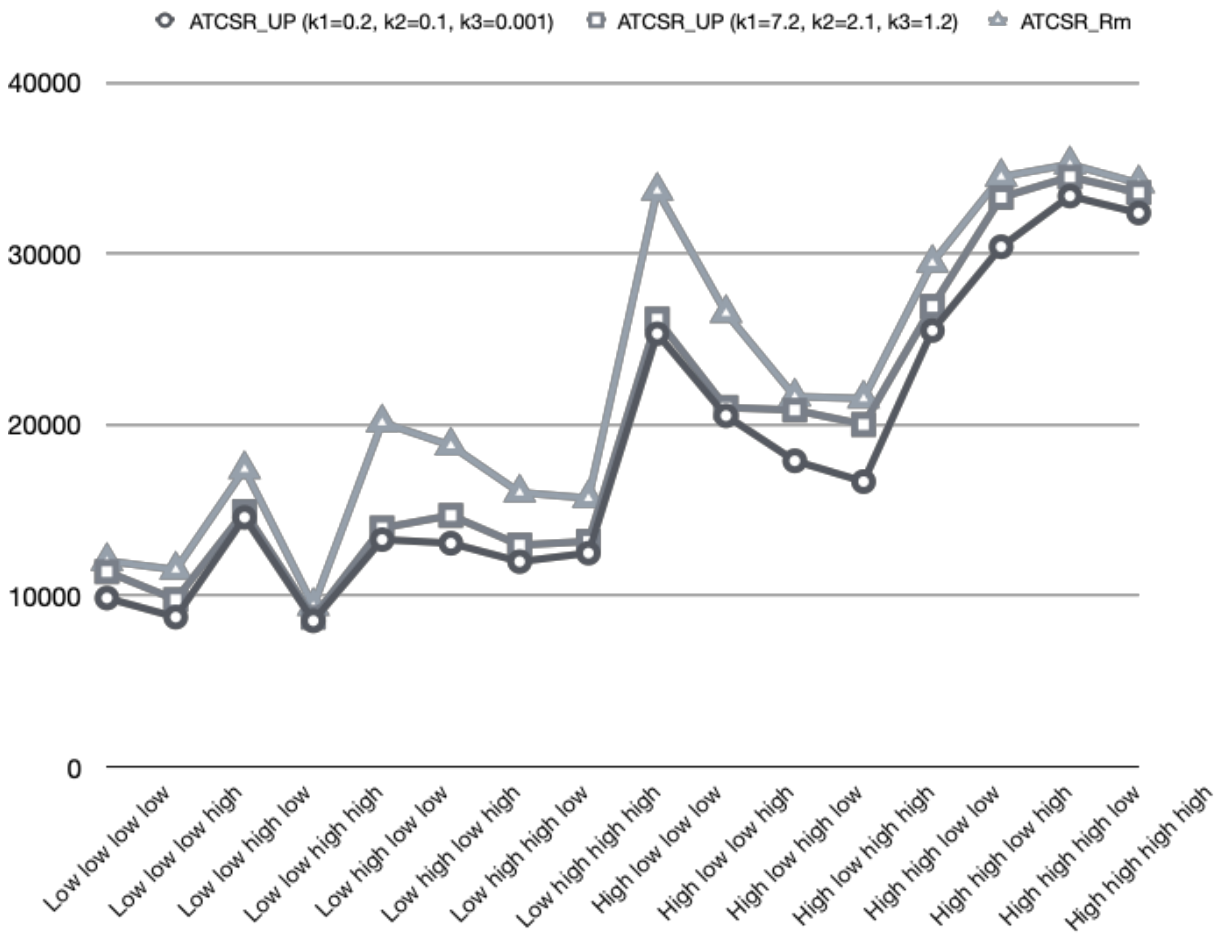


Figure 48. Overall average TWT comparison between ATCSR_Rm and ATCSR_UP

$(k_1 = 0.2, k_2 = 0.1, k_3 = 0.001$ and $k_1 = 7.2, k_2 = 2.1, k_3 = 1.2)$

CHAPTER 5 CONCLUSION AND FUTURE RESEARCH

This research focuses on minimizing the total weighted tardiness on unrelated parallel machines. For the unrelated parallel machine, we analyze ATC-based rules and propose a new ATC-based rule, ATCSR_UP. The performance of the new rule is evaluated on unrelated parallel machines. Experiments show that the new rule outperforms other ATC-based rules in minimizing the total weighted tardiness in unrelated parallel machine scheduling.

We found out that with the use of both scaling parameter (best and worst), ATCSR_UP outperforms ATCSR_Rm in 71% and 57% cases. Furthermore, ATCSR_UP (best k value) decreases the TWT about 17% compare to ATCSR_Rm and ATCSR_UP (worst k value) decreases the TWT about 11% compare to ATCSR_Rm. The results show that ATCSR_UP performs better when the value factors are in low level, and performs similar with ATCSR_Rm when value factors are in high level. With the decrease percentage, we can conclude that ATCSR_UP perform better than ATCSR_Rm. With the consideration of the earliest start time $\max\{r_j - t_m, 0\}$ in the WSPT term, machine time (t_m) in the slack term and ready term, the results are better than ATCSR_RM with considering these variables. No other paper than Lin and Hsieh (2013) focus on unrelated parallel machine with ATC-based rule. They consider separable cases, however, even if they consider continuous setup cases, the result would have been worst. Therefore, we can still conclude that ATCSR_UP outperforms ATCSR_Rm (Lin & Hsieh, 2013)

The paper also uses the proposed look-ahead heuristic for the unrelated parallel machines scheduling with sequence-dependent setup and future ready time. We compare the tardiness among two jobs with the biggest index, then allocate the job to schedule on the machine. After all the jobs were scheduled, the job switching heuristic will generate another schedule to find a better schedule for the jobs and machine. The job swap method we apply for the ATCSR_UP decrease TWT about 24% than without applying the job swap method, therefore, the job swap method improves the ATCSR_UP to find a better schedule. For the actual use of this research, since ATC-based rule works very well on tardiness objectives, the new rule should benefits the real-world manufacturing, however, this should be tested with the real-world manufacturing planning programs and datas.

For the future research, study to find robust scaling parameters for ATCSR_UP procedure is a good topic since in the research we only consider three groups of scaling parameters. Besides of that, the proposed look-ahead heuristic may be modified and used in a more complex production environment such as job shop scheduling. Lastly, consider the actual use of the ATCSR_UP on manufacturing planning programs in the real-world manufacturing system can further prove that ATCSR_UP can be use in the real-world situation.

REFERENCES

Allahverdi, A. (2015, 4 4). Third Comprehensive Survey on Scheduling Problems with Setup Times/Costs. *European Journal of Operational Research, 246(2)*, 345-378.

Allahverdi, A., & Aydilek, H. (2015, 4). The two-stage assembly flowshop scheduling problem to minimize total tardiness. *Journal of Intelligent Manufacturing, 26*, 225-237.

Allahverdi, A., Gupta, J., & Aldowaisan, T. (1999). A review of scheduling research involving setup considerations. *OMEGA, 27*, 219-239.

Allahverdi, A., Ng, C., Cheng, T., & Kovalyov, M. (2008, 6 16). A survey of scheduling problems with setup times or costs. *European Journal of Operational Research, 187(3)*, 985-1032.

Anghinolfi, D., & Paolucci, M. (2007). Parallel Machine Total Tardiness Scheduling with a New Hybrid Metaheuristic Approach. *Computers & Operations Research, 34(11)*, 3471–3490.

Anglani, A., Grieco, A., Guerriero, E., & Musmanno, R. (2005). Robust scheduling of parallel machines with sequence-dependent set-up cost. *European Journal of Operational Research, 161*, 704-720.

Arkin, M., & Roundy, R. (1991). Weighted-tardiness scheduling on parallel machines with proportional weights. *Operation Research, 39*, 64.

Avdeenko, T., & Mesentsev, Y. . (2016). Efficient Approaches to Scheduling for Unrelated Parallel Machines with Release Dates. *IFAC-PapersOnLine, 49(12)*, 1743-1748.

Balakrishnan, N., Kanet, J., & Sridharan, S. (1999). Early/Tardy Scheduling with Sequence Dependent Setups on Uniform Parallel Machines. *Computers & Operations Research, 26(2)*, 127-141.

Black, J., & Kohser, R. (2017). *DeGarmo's materials and process in manufacturing* (12 th ed.): Wiley.

Chang, T., Chou, F., & Lee, C. (2004). A heuristic algorithm to minimize total weighted tardiness on a single machine with release dates and sequence-dependent setup times. *Journal of the Chinese Institute of Industrial Engineers, 21*, 289-300.

Chen, J. (2009). Scheduling on Unrelated Parallel Machines with Sequence- and Machine-dependent Setup times and Due-date Constraints. *International Journal of Advanced Manufacturing Technology, 44 (11–12)*, 1204–1212.

Christoph, H., Lars, M., & Jens, Z. (2007). Planning and scheduling Heuristic for parallel machines. *Proceedings of the 2007 International Engineering Research Conference. Nashivill, Tenn,*, 1545-1550.

Christos, P., & Milton, L. (1998). Look Ahead Scheduling for Minimizing Machine Interference. *INT. J. PROD. RES,, 26(9)*, 1523-1533.

Chung, T., Xue, Z., Wu, T., & Shih, S. (2018, 7 19). Minimising total completion time on single-machine scheduling with new integrated maintenance activities. *International Journal of Production Research*.

Cochran, J., Horng, S., & Fowler, J. (2003). Minimize the total weighted completion time on unrelated parallel machines. *Computers and Operations Research, 30*, 1087–1102.

Costa, A., Cappadonna, F., & Fichera, S. (2016, 1). Minimizing the total completion time on a parallel machine system with tool changes. *Computer and Industrial Engineering, 91*, 290-301.

Diana, R., Souza, S., & Filho, M. (2018, 4). A Variable Neighborhood Descent as ILS local search to the minimization of the total weighted tardiness on unrelated parallel

machines and sequence dependent setup times. *Electronic Notes in Discete Mathematics, 66*, 191-198.

Drieel, R., & Monch, L. (2009). Scheduling jobs on parallel machines with sequence dependent setup times, precedence constraints, and ready times using variable neighborhood search. *IEEE*, 273-278.

Du, J., & Leung, J. (1990). Minimizing total tardiness on one machine is NP-Hard. *Mathematics of Operations Research, 15*, 483-494.

Ecom, D., Shin, H., Kwun, I., Shim, J., & Kim, S. . (2002). Scheduling jobs on parallel machines with sequence-dependent family set-up times. *The International Journal of Advanced Manufacturing Technology, 19*, 926-932.

Felipe, R. (2005). Look-ahead constructive heuristic for the unrelated parallel machine problem with sequenced dependent setup time. *IIE Annual Conference and Exposition 2005*, 1-6.

Feng, G., & Lau, H. (2005, 7 18-21). Efficient algorithm for machine scheduling problems with earliness and tardiness penalties. *Proceedings of the 2nd multidisciplinary international conference on scheduling: theory and application*, 196-211.

Fowler, J., & Horng, S. . (2003). A hybridized genetic algorithm to solve parallel machine scheduling problem with sequence dependent setup time. *International journal of industrial engineering: theory application and practice, 10*, 232-243.

Framimam, J., & Paz, P. (2017, 12). The 2-stage assembly flowshop scheduling problem with total completion time: Efficient constructive heuristic and metaheuristic. *Computer & Operation Research, 88*, 237-246.

Gharhgozli, A., Tavakkoli-Moghaddam, R., & Zaerpour, N. (2009). A Fuzzy-mixed-Integer Goal Programming Model for a Parallel-machine Scheduling Problem with Sequence-dependent Setup times and Release Dates. *Robotics and Computer-Integrated Manufacturing, 25*, 853-859.

Ginzburg, D., & Gonik, A. . (1997). Using "Look Ahead" Techniques in Job-shop Scheduling with Random Operations. *International. J. Production Economics, 50*, 13-22.

Guinet, A. (1991). Textile production systems : a succession of non-identical parallel processor shops. *Journal of the Operation Research Society, 42*, 655-671.

Heady, R., & Zhu, Z. (1998). Minimizing the sum of job earliness and tardiness in a multi-machines system. *Int J Prod Research*, 1619-1632.

Hirashi, K., Levner, E., & Vlach, M. (2002). Scheduling of parallel identical machines to maximize the weighted number of jobs of just-in time jobs. *Computers and operation research, 29*, 841-848.

Holthaus, O., & Ziegler, H. (1997). Improving Job shop Performance by Coordinating Dispatching Rule. *INT. J. PPROD. RES, 35(2)*, 539-549.

Hsieh, J., Chang, P., & Hsu, L. (2003). Scheduling of Drilling Operations in Printed Circuit Board Factory. *Computers and Industrial Engineering, 44*, 461-473.

Jang, J., Suh, J., Park, D., & Liu , R. (2001). A Look-Ahead Routing Procedure for Machine Selection in a Highly Informative Manufacturing System. *The International Journal of Flexible Manufacturing system, 13*, 287-308.

Jihene, K., Chirstophe, V., & Noureddine, Z. (2002). Heuristics for Scheduling Maintenance and Production on a Single Machine. *IEEE*.

Joo, C., & Kim, B. (2015,7). Hybrid genetic algorithms with dispatching rules for unrelated parallel machine scheduling with setup time and production availability. *Computer & Industrial Engineering, 85*, 102-109.

Kia, H., Ghodsypour, S. H., & Davoudpour, H. (2017,9). New scheduling rules for a dynamic flexible flow line problem with sequence-dependent setup times. *Journal of Industrial Engineering International, 13*, 297-306.

Kim, D., Kin, K., Jang, W., & Chen, F. (2002). Unrelated parallel machine scheduling with setup times using simulated annealing. *Journal of Robotics and Computer Integrated Manufacturing, 18*, 223-231.

Kim, D., Na, D., Chen, F., & Dong, G. (2002). Unrelated parallel machine scheduling with setup time and a total weighted tardiness objective. *Journal of Robostics and Computer Integrated Manufacturing, 19*, 173-181.

Kim, S., Choi, H., & Lee, D. (2007). Scheduling Algorithms for Parallel Machines with Sequence-dependent Set-up and Distinct Ready times: Minimizing Total Tardiness. *Proceedings of the Institution of Mechanical Engineers, Part B: Journal of Engineering Manufacture, 221*, 1087-1096.

Kim, S., Lee, Y., & Agnihotri, D. . (1995). A hybrid approach to sequencing jobs using heuristic rules and neural network. *Production Planning and Control, 6(5)*, 445-454.

Krajewski, L., King, B., Ritzman, L., & Wong, D. (1987). Kaban, MRP and shaping the manufacturing environment. . *Management Science, 33*, 39-57.

Kurt, M., & Askin. (2001). Heuristic scheduling of parallel machines with sequence dependent set-up times. *International Journal of Production Research, 39*, 3747-3769.

Lee, C.-H. (2017, 9 13). A dispatching rule and a random iterated greedy metaheuristic for identical parallel machine scheduling to minimize total tardiness. *International Journal of Production Research*, 2292-2308.

Lee, Y., & Pinedo, M. (1997). Scheduling jobs on parallel machines with sequence-dependent setup times. *European Journal of Operational Research, 100*, 464-474.

Lee, Y., Bhaskaran, K., & Pinedo, M. (1996). A heursitic to minimize the total weighted tardiness with sequence-dependent setups. *IIE Transactions, 46/47*, 119-125.

Lee, Z., Lin, S., & Ying, K. (2010). Scheduling Jobs on Dynamic Parallel Machines with Sequence-dependent Setup times. *International Journal of Advanced Manufacturing Technology, 47 (5–8)*, 773–781.

Liaw, C.-F., Lin, Y.-L., & Cheng, M. (2003). Scheduling unrelated parallel machine to minimze total weighted tardiness. *Computers and Operations Research, 30*, 1777-1789.

Lin, S., Lee, Z., Ying, K., & Lu, C. (2011). Minimization of Maximum Lateness on Parallel Machines with Sequence-dependent Setup times and Job Release Dates. . *Computers & Operations Research, 38 (5)*, 809-815.

Lin, Y., & Hsieh, F. (2013, 10 18). Unrelated parallel machine scheduling with setup times and ready times. *International Journal of Production Research, 52*, 1200-1214.

Logendran, R., McDonell, B., & Smucker, B. (2007). Scheduling unrelated parallel machines with sequence-dependent setups. *Computers & OPeration Research, 34*, 3420-3438.

Lu, C., Lin., S., & Ying., K. (2011). Minimization of total tardiness on unrelated parallel machines with sequence-and machine-dependent setup times under due date constraints. . *INt. J. Adv. Manuf. Technol, 53(4)*, 353-361.

Luh, P., Hoitomt, D., Max, E., & Pattipati, K. (1990). Schedule generation and reconfiguration for parallel machines. *IEEE Transactions on Robotics and Automation, 6*, 687-696.

Mao, W., & Kincaid, R. (1994). A Look-Ahead Heuristic for scheduling Jobs with Release Date on a Single Machine. *Computer Ops Res, 21(10)*, 1041-1050.

McNaughton., R. (1959). Scheduling with deadlines and loss functions. *Manage. Sci, 6(3)*, 4-12.

Nailwal, K., Gupta, D., & Sharma, S. (2015, 3 3). Fuzzy bi-criteria scheduling on parallel machines involving weighted flow time and maximum tardiness. *Cogent Mathematics(2)*.

Panwalkar, S. S., Dudek, R. D., & Smith, M. L. (1973). Sequencing Research and the Industrial Scheduling Problem. *Symposium on the Theory of Scheduling and Its Applications*, 29-38.

Park, Y., Kim, & Lee, Y. (2000). Scheduling jobs on parallel machines applying neural network and heuristic rules. *Computer & Industrial Engineering, 38*, 189-202.

Pfund, M., Folwer, J., Gadkari, A., & Chen, Y. (2008). Scheduling jobs on parallel machines with setup times and ready time. *Computers & Industrial Engineering, 54(4)*, 764-782.

Pinedo, M. (2016). *Scheduling: Theory, Algorithms, and Systems*: Springer.

Rachamadugu, R., & Morton, T. (1982). Myopic heuristics for the single machine weighted tardiness problem. *Working Paper, Carnegie Mellon University, Pittsburgh, PA 30*, 82-86.

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

Raman N, R. R. (1989). Real time scheduling of an automated manufacturing center. *Eur J Oper Res, 40*, 222-242.

Rasaratnam, L., Brent, M., & Byran, S. (2007). Scheduling unrelated parallel machines with sequence-dependent setups. *Computers & Operations Research, 34*, 3420-3438.

Schutten, J., & Leussink, R. (1996). Parallel Machine Scheduling with Release Dates, Due Dates and Family Setup times. *International Journal of Production Economics, 46-47*, 119-125.

Shahid-Zadeh, B., Tavakkoli-Moghaddam, R., Taheri-Moghadam, A., & Rastgar, I. (2017, 12). Solving a bi-objective unrelated parallel batch processing machines scheduling problem: A comparison study. *Computers and Operations Research, 88*, 71-90.

Silva, C., & Magalhaes, J. (2006). Heuristic Lot Size Scheduling on Unrelated Parallel Machines with Applications in the Textile Industry. *Computers & Industrial Engineering, 50*, 76-89.

Slotnick, S. (2011). Order Acceptance and Scheduling: A Taxonomy and Review. *European Journal of Operational Research, 212*, 1-11.

Smith, F., & Stecke, K. (1996). On the Robustness of using Balanced Part Mixed Ratios to Determine Cyclic Part input Sequences into Flexible Flow Systems. *INT. J. PROD, 34(10)*, 2925-2941.

Tamimi, S., & Rajan, V. (1997). Reduction of total weighted tardiness on uniform machines with sequence dependent setups. *Industrial engineering research-conference procedding*, 181-185.

Tunali, S. (1997). A Simulation Study Evaluating the Performance of a Look-ahead FMS Machine Scheduling Algorithm. *Microcomputer Application, 16(3)*.

Vepsalainen, A., & Morton, T. (1987). Priority rules for job shops with weighted tardiness costs. *Management Science, 33*, 1035-1047.

Wang, X., & Cheng, T. (2015, 12 6). A heuristic for scheduling jobs on two identical parallel machines with a machine availability constraint. *Int. J. Production Economics, 161*, 74-82.

Weng, M., Lu, J., & Ren, H. (2001). Unrelated parallel machine scheduling with setup consideration and a totl weighted completion time objective. *International Journal of Production Economics, 70*, 215-226.

Xi, Y., & Jang, J. (2011). Minimizing total weighted tardiness on a single machine with sequence-dependent setup and future ready time. *Int J Adv Manuf Technol*.

Xi, Y., Jang, J., Friedman, D., & Hou, W. . (2015, 7). A tardiness-concerned constructive method for the identical parallel machine scheduling. *The International Journal of Advanced Manufacturing Technology, 79*, 851-862.

Xu, L., Wang, Q., & Huang, S. (2015). Dynamic order acceptance and scheduling problem with sequence-dependent setup time. *International Journal of Production Research, 53(19)*, 5797-5808.

Yin, Y., Chen, Y., Qin, K., & Wang, D. (2018, 8 20). Two-agent scheduling on unrelated parallel machines with total completion time and weighted number of tardy jobs criteria. *journal of scheduling*, 1-19.

Ying, K., & Cheng, H. (2010). Dynamic Parallel Machine Scheduling with Sequence-dependent Setup times Using an Iterated Greedy Heuristic. *Expert Systems with Applications, 37 (4)*, 2848–2852.

Yue, X., & Jang, J. (2013, 2 5). Minimizing total weighted tardiness on a single machine with sequence-dependent setup and future ready time. *The International Journal of Advanced Manufacturing Technology*, 281–294.

Zhang, R., & Chiong, R. (2016, 1 20). Solving the energy-efficient job shop scheduling problem: a multi-objective genetic algorithm with enhanced local search for minimizing the total weighted tardiness and total energy consumption. *Journal of Cleaner Production, 112*, 3361-3375.

Zhang, Z., Zheng, L., & Weng, M. (2007). Dynamic Parallel Machine Scheduling with Mean Weighted Tardiness Objective by Q-Learning. *International Journal of Advanced Manufacturing Technology, 34*.

APPENDICES: PROGRAM CODE FOR THE THESIS

The code was coded on MacBook pro, Intel Core i5, memory 8 GB, Python.

```
from tkinter import *
from tkinter import messagebox as tkMessageBox
from tkinter import filedialog as tkFiledialog
from tkinter.filedialog import askopenfilename
import copy
import csv
import math
import os
import fnmatch 。
filenames = ""
text_k1 = ""
text_k2 = ""
text_k3 = ""
#define the swap method
def swap(a,b):
    temp=a
    a=b
    b=temp
    return a,b
#define the index calculation formula
def inintal_acc_index
(weight,ready,due_date,machine_time,process,setup,average_processing_time,average_setup_time,m1,k1,k2,k3
):
    index_value = ((weight/(max(ready-machine_time,0)+process+setup))
    *math.exp((-1*max((due_date-max(ready,machine_time)-process-
setup),0))/((k1)*(average_processing_time+average_setup_time)))
    *math.exp(-1*(m1/((k2)*average_setup_time)))
    *math.exp((-1*(max((ready-machine_time),0)))/((k3)*average_processing_time))
    )
    return index_value
```

```
#define the calculation for updating machine time
def update_machine_time (ready,machine_time,process,setup):

    update_machine_time_value =(max(ready-machine_time,0))+process+setup

    return update_machine_time_value
#define the use of k value for calculating the index formula
def search():
    root.text_k1 = entry_1.get()  # 取值  #get k value from user input
    root.text_k2 = entry_2.get()  # 取值  #get k value from user input
    root.text_k3 = entry_3.get()  # 取值  #get k value from user input
    if  root.text_k1=="" and  root.text_k2=="" and  root.text_k3=="":
        tkMessageBox.showinfo('Error','enter k value！')
        return
    root.filenames = askopenfilename(filetypes=[("CSV Files", ".csv")]) #
    print (root.filenames)
#define job input for the calculation
def acc_job():
    if  root.text_k1 == "" and  root.text_k2 == "" and  root.text_k3 == "":
        tkMessageBox.showinfo('Error', 'enter k value！')
        return
    print("filenames")
    print(root.filenames)
    if not root.filenames:
        tkMessageBox.showinfo('Error','no file chosen！')
        return
    total_job = 0;    #total job starts from 0
    total_machine = 0; #total machine starts from 0
    machine_read_cout = 1; #read total machine number
    machine_read_row_cout = 0;
    k1 = float(root.text_k1); #define k value
    k2 = float(root.text_k2); #define k value
    k3 = float(root.text_k3); #define k value
    ready_cout = 0; #ready time starts from 0
    process_cout = 0; #processing time starts from 0
```

```
    due_date_cout = 0; #due date starts from 0
    weight_cout = 0; #weight starts from 0
    setup_cout = 0; #setup time starts from 0
    machine_time_cout = 0 #machine time starts from 0
    job_queue = 0; #job queue starts from 0
    cout = 0
    ######################################################READ
FILE#################################################################
    with open(root.filenames, newline='') as csvfile: #open csv file
        rows = csv.reader(csvfile) #read data from file row
        for row in rows:
            if row[0] == 'total job': #read "total job" row to get total jobs
                total_job = row[1]
                print(total_job)
            if row[0] == 'total machine': #read "total machine" row to get total machine
                total_machine = row[1]
                print(total_machine)
    machine_job_step_array = [[[[] for x in range(int(total_job))] for y in range(int(total_job) + 1)] for z in
                              range(int(total_machine))] #read input data and set up the total job for total machine

    ready_row = [[] for x in range(int(total_machine))] #read ready row for total machine
    process_row = [[] for x in range(int(total_machine))] #read process row for total machine
    due_date_row = [[] for x in range(int(total_machine))] #read due date row for total machine
    weight_row = [[] for x in range(int(total_machine))] #read weight row for total machine
    setup_row = [[] for x in range(int(total_machine))] #read setup row for total machine
    #inintal_setup_row= [[] for x in range(int(total_machine))]
    machine_time_row = [[0 for y in range(int(total_job))] for z in range(int(total_machine))]
    #inintal_machine_time_row = [[0 for y in range(int(total_job))] for z in range(int(total_machine))]
    average_processing_time = [0 for z in range(int(total_machine))] #define average processing time for
each   machine
    average_setup_time = [0 for z in range(int(total_machine))] #define average processing time for
each   machine

    with open(root.filenames, newline='') as csvfile: #open csv file to start calculate
        rows = csv.reader(csvfile) #get input data from csv file rows
```

```
        for row in rows:
            print(row[0])
            if row[0] == 'ready':#read first ready row, for the next job ready time, use total job + 1, stop when no job can be read in the csv file
                for num in range(1, int(total_job) + 1):
                    ready_row[ready_cout].append(int(row[num]))
                ready_cout = ready_cout + 1
                job_queue = job_queue + 1
            if row[0] == 'process':#read first process row, for the next job process time, use total job + 1, stop when no job can be read in the csv file
                for num in range(1, int(total_job) + 1):
                    process_row[process_cout].append(int(row[num]))
                process_cout = process_cout + 1
            if row[0] == 'due date':#read first due date row, for the next job due date time, use total job + 1, stop when no job can be read in the csv file
                for num in range(1, int(total_job) + 1):
                    due_date_row[due_date_cout].append(int(row[num]))
                due_date_cout = due_date_cout + 1
            if row[0] == 'weight':#read first weight row, for the next job weight time, use total job + 1, stop when no job can be read in the csv file
                for num in range(1, int(total_job) + 1):
                    weight_row[weight_cout].append(int(row[num]))
                weight_cout = weight_cout + 1
            if row[0] == 'setup':#read setup ready row, for the next job setup time, use total job + 1, stop when no job can be read in the csv file
                for num in range(1, int(total_job) + 1):
                    setup_row[setup_cout].append(int(row[num]))
                setup_cout = setup_cout + 1
            if row[0] == 'machine time':#set a row to store machine time
                for num in range(1, int(total_job) + 1):
                    machine_time_row[machine_time_cout].append(int(row[num]))
                machine_time_cout = machine_time_cout + 1
            if row[0] == 'm' + str(machine_read_cout) + ' setup time': #read the machine setup time from file
                # print(row)
```

```
                for num in range(1, int(total_job) + 1):
                    ttt = int(row[num])
                    machine_job_step_array[machine_read_cout - 1][machine_read_row_cout][num - 1] = ttt
#check every job setup time is read
                machine_read_row_cout = machine_read_row_cout + 1
            if row[0] == '' and machine_read_row_cout >= 1 and machine_read_row_cout <= int(total_job):
#check every job setup time is read
                # print(row)
                for num in range(1, int(total_job) + 1):
                    machine_job_step_array[machine_read_cout - 1][machine_read_row_cout][num - 1] =
int(row[num])
                machine_read_row_cout = machine_read_row_cout + 1
                if machine_read_row_cout > int(total_job):
                    print('YES')
                    machine_read_cout = machine_read_cout + 1
                    machine_read_row_cout = 0

    ############################################################READ
FILE#################################################################
    index_row = [[0 for x in range(int(total_job))] for y in range(int(total_machine))] #set up index row for total
jo and toatl machine
    completion_time = [[0 for y in range(int(total_job))] for z in range(int(total_machine))] #set up completion
time row for total jo and toatl machine
    tardiness = [[0 for y in range(int(total_job))] for z in range(int(total_machine))] #set up tardiness row for
total jo and toatl machine
    #tardiness_buffer = [[0 for y in range(int(total_job))] for z in range(int(total_machine))]
    job_schedule = [[0 for x in range(int(int(total_job) / int(total_machine)))] for y in range(int(total_machine))]
#set up job schedule row for total jo and toatl machine
    machine_time_job_schedule = [[0 for x in range(int(int(total_job) / int(total_machine)))] for y in
range(int(total_machine))]
    t_job_schedule = [[0 for x in range(int(int(total_job) / int(total_machine)))] for y in
range(int(total_machine))] #define row for tardiness row for each job
    m_job_schedule = [[0 for x in range(int(int(total_job) / int(total_machine)))] for y in
range(int(total_machine))] #define row for machine time row for each job
```

```
    r_job_schedule = [[0 for x in range(int(int(total_job) / int(total_machine)))] for y in
range(int(total_machine))] #define row for ready time row for each job
    p_job_schedule = [[0 for x in range(int(int(total_job) / int(total_machine)))] for y in
range(int(total_machine))] #define row for processing time row for each job
    s_job_schedule = [[0 for x in range(int(int(total_job) / int(total_machine)))] for y in
range(int(total_machine))] #define row for setup time row for each job
    d_job_schedule = [[0 for x in range(int(int(total_job) / int(total_machine)))] for y in
range(int(total_machine))] #define row for due date row for each job
    step_stage_machine_time = [[0 for x in range(int(int(total_job) / int(total_machine)))] for y in
range(int(total_machine))] #define each machine step
    t_job_schedule_trans = [[0 for x in range(int(int(total_job) / int(total_machine)))] for y in
range(int(total_machine))]
    w_job_schedule = [[0 for x in range(int(int(total_job) / int(total_machine)))] for y in
range(int(total_machine))] #define row for job weight row for each job
    final_job_schedule = [[0 for x in range(int(int(total_job) / int(total_machine)))] for y in
range(int(total_machine))] #define row for final job row for each machine
    final_job_schedule_t_value = [[0 for x in range(int(int(total_job) / int(total_machine)))] for y in
                                  range(int(total_machine))]
    final_job_schedule_weight_value = [[0 for x in range(int(int(total_job) / int(total_machine)))] for y in
                                       range(int(total_machine))]
    job_schedule_cout = [0 for y in range(int(total_machine))] #define job schedule row
    ready_job = [1 for y in range(int(total_job))] #define ready job schedule

    # machine_available = [[1 for y in range(int(total_machine))] for z in range(int(job_queue))]
    machine_available_cout = [int(total_machine)] #read total machine number
    compare1 = 0 #compare selected job
    compare2 = 0 #compare selected job
    big_value_temp = [0 for x in range(int(total_job) * int(total_machine))] #set up row for bigger value for
comparison
    big_value = [0 for x in range(2)]
    big_value_flag_row = [0 for x in range(2)]
    big_value_flag_colum = [0 for x in range(2)]
    machine_available_cout_using = int(total_machine)
    machine_available = [1 for y in range(int(total_machine))]
    print("ready_row")
```

```
    print(ready_row)
    print("machine_job_step_array")
    for i in range(0, int(total_machine)):
        print(machine_job_step_array[i]) #print machine job schedule
    for i in range(0, int(total_machine)):
        for j in range(0, int(total_job)):
            average_processing_time[i] = average_processing_time[i] + process_row[i][j] #calculate average
processing time
            average_setup_time[i] = average_setup_time[i] + setup_row[i][j] #calculate average setup time
        if sum(average_processing_time)>0:
            average_processing_time[i] = average_processing_time[i] / int(total_job) #calculate average
processing time
        if sum(average_setup_time) > 0:
            average_setup_time[i] = average_setup_time[i] / int(total_job) #calculate average setup time
    print(" ")
    print("average_processing_time")
    print(average_processing_time)
    print("average_setup_time")
    print(average_setup_time)
    print("ready_row")
    print(ready_row)
    print("index_row ")
    print("job_schedule ")
    print(job_schedule)
    # for i in range(0, int(total_machine)-1):
    print("")
    print(index_row)
    run = 0
    inintal_setup_row = copy.deepcopy(setup_row) #initial setup row
    machine_job_step_array_flag = 0 #define machine job step array
    while run < int(total_job):

        print("##################################################start")
        #print("machine_job_step_array_flag")
        #print(machine_job_step_array_flag)
```

```
        run = run + 1
        #print("01 machine_available ")
        #print(machine_available)
        # while   machine_available_cout_using >=0:
        ##################################################index value
#############################################################
        for j in range(0, int(total_machine)): #calculate job index
            for k in range(0, int(total_job)):
                if ready_job[k] == 1:
                    index_row[j][k] = inintal_acc_index(weight_row[j][k],
ready_row[j][k],due_date_row[j][k],machine_time_row[j][k], process_row[j][k], setup_row[j][k],
average_processing_time[j],
average_setup_time[j],machine_job_step_array[j][machine_job_step_array_flag+1][k], k1,
k2, k3) #calculate job index from job input
completion_time[j][k] = max(ready_row[j][k],ready_row[j][k]-machine_time_row[j][k])+process_row[j][k] +
setup_row[j][k] #define completion time
                    tardiness[j][k] = due_date_row[j][k] - completion_time[j][k] - machine_time_row[j][k]
#define tardiness
                elif ready_job[k] == 0:
                    index_row[j][k] = -2147483648 #set smallest index as -2147483648
                    tardiness[j][k] = -2147483648 #set smallest tardiness as -2147483648
        #for i in range(0, int(total_machine)):
            #print(index_row[i])

        ##########################################################################################
#####################################
        ################################################ Find the two biggest ones from the index
#################################
        for j in range(0, int(total_machine)):
            for k in range(0, int(total_job)):
                if (machine_available[j] == 1):
                    big_value_temp[(j * int(total_job)) + k] = index_row[j][k] #calculate index
                elif (machine_available[j] == 0):
                    big_value_temp[(j * int(total_job)) + k] = -2147483648
        #print(" big_value_temp")
```

```
        #print(big_value_temp)
        big_value[0] = max(big_value_temp)
        big_value_flag_row[0] = big_value_temp.index(max(big_value_temp)) #set row for first big value
big_1 = big_value_temp.index(max(big_value_temp))
big_value_temp[big_1] =-2147483648              #set big_value_temp[big_1] smallest at -2147483648
 big_value[1] = max(big_value_temp)
  big_value_flag_row[1] = big_value_temp.index(max(big_value_temp)) #set row for second big value
  big_2 = big_value_temp.index(max(big_value_temp))
 big_value_temp[big_2] = -2147483648            #set big_value_temp[big_1] smallest at -2147483648
  #print("big_value_flag")
     #(big_value_flag_row)
    #print("   ")
  ##########################################Find the smallest
tardiness#################################
        compare1 = tardiness[int(big_1 / int(total_job))][big_1 % int(total_job)]    #define compare1, and
calculate the value
        compare2 = tardiness[int(big_2 / int(total_job))][big_2 % int(total_job)]    #define compare1, and
calculate the value
        # print("machine_available ")
        print("compare Machine "+str(int(big_1 / int(total_job)))+" job "+ str(int(big_1 % int(total_job))+1)+"
tardiness") #print compare1 Machine job and tardiness
        print(compare1)
        print("compare Machine " + str(int(big_2 / int(total_job)))+" job "+str(int(big_2 % int(total_job))+1)+ "
tardiness") #print compare1 Machine job and tardiness
        print(compare2)
        if compare1 <= compare2: #compare1 and compare2
            print("Yes compare Machine " + str(int(big_2 / int(total_job))) + " job " + str(
                int(big_2 % int(total_job)) + 1) + " tardiness is big") #print yes compare Machine job tardiness is
big
            machine_job_step_array_flag = int(big_2 % int(total_job))
            machine_available[int(big_2 / int(total_job))] = 0 #define machine available row
            # print(job_schedule_cout[int(big_2/int(total_job))])
            job_schedule[int(big_2 / int(total_job))][job_schedule_cout[int(big_2 / int(total_job))]] = big_2 %
int(total_job) #define job schedule
```

```
            ready_job[job_schedule[int(big_2 / int(total_job))][job_schedule_cout[int(big_2 / int(total_job))]]] =
0            #define ready job schedule
            step_stage_machine_time[int(big_2 / int(total_job))][job_schedule_cout[int(big_2 / int(total_job))]]
= \
            machine_time_row[int(big_2 / int(total_job))][int(big_2 % int(total_job))]#define machine time
schedule
            job_schedule_cout[int(big_2 / int(total_job))] = job_schedule_cout[int(big_2 / int(total_job))] + 1
            ##############################################update_machine_time_value##############
##################
            for k in range(0, int(total_job)):
                if ready_job[k] == 1:
                    machine_time_row[int(big_2 / int(total_job))][k] = update_machine_time(
                            ready_row[int(big_2 / int(total_job))][int(big_2 % int(total_job))],
                            machine_time_row[int(big_2 / int(total_job))][int(big_2 % int(total_job))],
                            process_row[int(big_2 / int(total_job))][int(big_2 % int(total_job))],
                            setup_row[int(big_2 / int(total_job))][int(big_2 % int(total_job))]) #update machine
time for big 2

                    #machine_time_row[int(big_2 / int(total_job))][k] = update_machine_time(
                    #     ready_row[int(big_2 / int(total_job))][k],
                    #     machine_time_row[int(big_2 / int(total_job))][k],
                    #     process_row[int(big_2 / int(total_job))][k],
                    #     setup_row[int(big_2 / int(total_job))][k])
            #machine_time_row[int(big_2 / int(total_job))][big_2 % int(total_job)] = update_machine_time(
            #     ready_row[int(big_2 / int(total_job))][big_2 % int(total_job)],
            #     machine_time_row[int(big_2 / int(total_job))][big_2 % int(total_job)],
            #     process_row[int(big_2 / int(total_job))][big_2 % int(total_job)],
            #     setup_row[int(big_2 / int(total_job))][big_2 % int(total_job)])

            ##############################################update_steup_time_value################
################
            for j in range(0, int(total_machine)):
                for k in range(0, int(total_job)):
                    if ready_job[k] == 1:
```

```
                    setup_row[i][k] = machine_job_step_array[int(big_2 / int(total_job))][(big_2 %
int(total_job)) + 1][k]   #update_steup_time_value
            #setup_row[int(big_2 / int(total_job))] = machine_job_step_array[int(big_2 / int(total_job))][
            #     (big_2 % int(total_job)) + 1]
            #for j in range(0, int(total_machine)):
            #     for k in range(0, int(total_job)):
            #          completion_time[j][k] = process_row[j][k] + setup_row[j][k]
            #          tardiness[j][k] = due_date_row[j][k] - completion_time[j][k] - machine_time_row[j][k]
        elif compare1 > compare2: #compare1 and compare2
            print("Yes compare Machine " + str(int(big_1 / int(total_job))) + " job " + str(
                int(big_1 % int(total_job)) + 1) + " tardiness is big")   #print yes compare Machine job tardiness
is big
            machine_available[int(big_1 / int(total_job))] = 0
            machine_job_step_array_flag = int(big_1 % int(total_job))
            print(big_1)
            print(job_schedule_cout[int(big_1 / int(total_job))])
            job_schedule[int(big_1 / int(total_job))][job_schedule_cout[int(big_1 / int(total_job))]] = big_1 %
int(total_job)   #define job schedule
            ready_job[job_schedule[int(big_1 / int(total_job))][job_schedule_cout[int(big_1 / int(total_job))]]] =
0     #define ready job schedule
            step_stage_machine_time[int(big_1 / int(total_job))][job_schedule_cout[int(big_1 / int(total_job))]]
= machine_time_row[int(big_1 / int(total_job))][int(big_1 % int(total_job))]#define machine time schedule
            job_schedule_cout[int(big_1 / int(total_job))] = job_schedule_cout[int(big_1 / int(total_job))] + 1

            ##############################################update_machine_time_value###############
###################
            for k in range(0, int(total_job)):     #update_machine_time_value
                if ready_job[k] == 1:
                    machine_time_row[int(big_1 / int(total_job))][k] = update_machine_time(
                            ready_row[int(big_1 / int(total_job))][int(big_1 % int(total_job))],
                            machine_time_row[int(big_1 / int(total_job))][int(big_1 % int(total_job))],
                            process_row[int(big_1 / int(total_job))][int(big_1 % int(total_job))],
                            setup_row[int(big_1 / int(total_job))][int(big_1 % int(total_job))]) #update machine
time for big 1
```

```
                #machine_time_row[int(big_1 / int(total_job))][k] = update_machine_time(
                #     ready_row[int(big_1 / int(total_job))][k],
                #     machine_time_row[int(big_1 / int(total_job))][k],
                #     process_row[int(big_1 / int(total_job))][k],
                #     setup_row[int(big_1 / int(total_job))][k])
            #machine_time_row[int(big_1 / int(total_job))][big_1 % int(total_job)] = update_machine_time(
            #     ready_row[int(big_1 / int(total_job))][big_1 % int(total_job)],
            #     machine_time_row[int(big_1 / int(total_job))][big_1 % int(total_job)],
            #     process_row[int(big_1 / int(total_job))][big_1 % int(total_job)],
            #     setup_row[int(big_1 / int(total_job))][big_1 % int(total_job)])
            ##############################################update_steup_time_value################
#################
            for j in range(0, int(total_machine)): #update_steup_time_value for big 1
                for k in range(0, int(total_job)):
                    if ready_job[k] == 1:
                        setup_row[i][k] = machine_job_step_array[int(big_1 / int(total_job))][(big_1 %
int(total_job)) + 1][k]
            #setup_row[int(big_1 / int(total_job))] = machine_job_step_array[int(big_1 / int(total_job))][
            #     (big_1 % int(total_job)) + 1]
            #for j in range(0, int(total_machine)):
            #     for k in range(0, int(total_job)):
            #          completion_time[j][k] = process_row[j][k] + setup_row[j][k]
            #          tardiness[j][k] = due_date_row[j][k]-completion_time[j][k]- machine_time_row[j][k]
            #print("new job"+str())
            #print(tardiness[j][k])
        print("02 machine_available ")
        print(machine_available)
        if (sum(machine_available) == 0):
            machine_available = [1 for y in range(int(total_machine))] #define total available machine
        print("step_stage_machine_time ")
        print(step_stage_machine_time)
        print("job_schedule ")
        print(job_schedule)
        print("ready_job ")
        print(ready_job)
```

```
        print("machine_time_row ")
        print(machine_time_row)
        #print("inintal_machine_time_row ")
        #print(inintal_machine_time_row)
        print("#####################################################end")
        final_job_schedule = copy.deepcopy(job_schedule)
        t_job_schedule_copy= copy.deepcopy(t_job_schedule)
        # for i in range(0, int(total_machine)):
        #     for j in range(0, int(total_job)):
        #         machine_time_row[i][j] = update_machine_time(ready_row[i][j]
        #         ,machine_time_row[i][j],process_row[i][j],setup_row[i][j] )
    ##########################################
    inintal_machine_time_row = copy.deepcopy(machine_time_row)

    for i in range(0, int(total_machine)):
        for j in range(0, int(total_job)):
            completion_time[i][j] = max(ready_row[i][j], ready_row[i][j] - machine_time_row[i][j]) +
process_row[i][j] + setup_row[i][j] #define completion time
            tardiness[i][j] = due_date_row[i][j] - completion_time[i][j] - machine_time_row[i][j]   #define
tardiness

    for i in range(0, int(total_machine)):
        for j in range(0, int(int(total_job) / int(total_machine))):
            t_job_schedule[i][j] = tardiness[i][int(job_schedule[i][j]%int(total_job))] #define tardiness schedule
from job schedule and total job
            w_job_schedule[i][j] = weight_row[i][int(job_schedule[i][j]%int(total_job))] #define job weight
schedule from job schedule and total job
            m_job_schedule[i][j] = machine_time_row[i][int(job_schedule[i][j]%int(total_job))] #define
machine time schedule from job schedule and total job
            d_job_schedule[i][j] = due_date_row[i][int(job_schedule[i][j]%int(total_job))] #define due date
schedule from job schedule and total job
            p_job_schedule[i][j] = process_row[i][int(job_schedule[i][j]%int(total_job))] #define processing
time schedule from job schedule and total job
            r_job_schedule[i][j] = ready_row[i][int(job_schedule[i][j]%int(total_job))] #define ready time
schedule from job schedule and total job
```

```
            s_job_schedule[i][j] = inintal_setup_row[i][int(job_schedule[i][j]%int(total_job))] #define setup time
schedule job schedule and total job
    print("inintal_setup_row ")
    print(inintal_setup_row)
    print("t_job_schedule ")
    print(t_job_schedule)
    print("m_job_schedule ")
    print(m_job_schedule)
    print("s_job_schedule")
    print(s_job_schedule)
    #apply tardiness buffer value for applying swap method
    for i in range(0, int(total_machine)):
        for j in range(0, int(int(total_job) / int(total_machine))):
            for k in range(j, int(int(total_job) / int(total_machine))):
                print("Machine "+ str(i)+" Check job " + str(j) + " and job " + str(k))
                buffer_value_completion = max(r_job_schedule[i][k]
                                              , r_job_schedule[i][k]
                                              - m_job_schedule[i][j]) \
                                          + p_job_schedule[i][k] \
                                          +s_job_schedule[i][k]


                tardiness_buffer = d_job_schedule[i][k] \
                                   - buffer_value_completion- m_job_schedule[i][j]
                print(str(tardiness_buffer) + " = " + str(d_job_schedule[i][k]) + " - max( " + str(
                    r_job_schedule[i][k]) + " , " + str(r_job_schedule[i][k]) + " - " + str(
                    m_job_schedule[i][j]) + ") + " + str(p_job_schedule[i][k]) + " + " + str(
                    s_job_schedule[i][k]) + " + " + str(m_job_schedule[i][j]))
                #apply job swap method
                print(t_job_schedule[i][j])
                print(tardiness_buffer)
                if t_job_schedule[i][j]<tardiness_buffer: #assign a tardiness buffer and compare if tardiness
buffer bigger than tardiness schedule swap
                    final_job_schedule[i][j],final_job_schedule[i][k]=swap(final_job_schedule[i][j],final_job_sc
hedule[i][k])
```

```
                w_job_schedule[i][j],w_job_schedule[i][k]=swap(w_job_schedule[i][j],w_job_schedule[i][k]
)
                t_job_schedule[i][j]=tardiness_buffer
                print("Machine "+ str(i)+" Check job " + str(j) + " and job " + str(k) + " compare sucess")
                m_job_schedule[i][k] = m_job_schedule[i][j]#swap the two seleced job
                r_job_schedule[i][j], r_job_schedule[i][k] = swap(r_job_schedule[i][j],
                                                                 r_job_schedule[i][k])#swap ready time
                d_job_schedule[i][j], d_job_schedule[i][k] = swap(d_job_schedule[i][j],
                                                                 d_job_schedule[i][k])#swap due date
                #m_job_schedule[i][j], m_job_schedule[i][k] = swap(m_job_schedule[i][j],
                #                                                  m_job_schedule[i][k])
                p_job_schedule[i][j], p_job_schedule[i][k] = swap(p_job_schedule[i][j],
                                                                 p_job_schedule[i][k])#swap processing time
                #s_job_schedule[i][j]=s_job_schedule[i][k]
                s_job_schedule[i][j], s_job_schedule[i][k] = swap(s_job_schedule[i][j],
                                                                 s_job_schedule[i][k])#swap setup time

            for t in range(j+1 , int(int(total_job) / int(total_machine))):#update the machine time after job swap
                m_job_schedule[i][t] = update_machine_time(
                    r_job_schedule[i][j],
                    m_job_schedule[i][j],
                    p_job_schedule[i][j],
                    s_job_schedule[i][j])
            for t in range(j+1 , int(int(total_job) / int(total_machine))):
                s_job_schedule[i][t] = machine_job_step_array[i][1+(int(final_job_schedule[i][j] %
int(total_job))) ][int(final_job_schedule[i][t] % int(total_job))]
            for t in range(j + 1, int(int(total_job) / int(total_machine))):
                buffer_value_completion = max(r_job_schedule[i][t]
                                              , r_job_schedule[i][t]
                                              - m_job_schedule[i][t]) \
                                          + p_job_schedule[i][t] \
                                          + s_job_schedule[i][t]
                tardiness_buffer = d_job_schedule[i][t] \
                                   - buffer_value_completion - m_job_schedule[i][t]
                t_job_schedule[i][t] = tardiness_buffer
```

```
                ##############################################update_machine_time_value###########
#######################
                #else:
                #     for t in range(j+1, int(int(total_job) / int(total_machine))):
                #          #if not (k==t):
                #          m_job_schedule[i][t] = update_machine_time(
                #          r_job_schedule[i][j],
                #          m_job_schedule[i][j],
                #          p_job_schedule[i][j],
                #          s_job_schedule[i][j])
                #     for t in range(j+1 , int(int(total_job) / int(total_machine))):
                #          s_job_schedule[i][t] = machine_job_step_array[i][1+(int(final_job_schedule[i][j] %
int(total_job))) ][int(final_job_schedule[i][t] % int(total_job))]

                    #for t in range(j + 1, int(int(total_job) / int(total_machine))):
                    #     s_job_schedule[i][t] = machine_job_step_array[i][int(job_schedule[i][j] %
int(total_job)) + 1][
                    #          int(job_schedule[i][t] % int(total_job))]
                ##############################################update_steup_time_value#############
####################
            print("final_job_schedule")
            print(final_job_schedule)
            print("m_job_schedule")
            print(m_job_schedule)
            print("s_job_schedule")
            print(s_job_schedule)
    print("final_job_schedule")
    print(final_job_schedule)
    print("t_job_schedule")
    print(t_job_schedule)
    print("s_job_schedule")
    print(s_job_schedule)
    string_temp = ""
    string_temp_2 = ""
```

```
string_temp_3 = ""
array_temp = [0 for y in range(int(int(total_job) / int(total_machine)))] #define available machine
array_temp_2 = [0 for y in range(int(int(total_job) / int(total_machine)))] #define available machine
array_temp_3 = [0 for y in range(int(total_machine))]
array_temp_4 = [0 for y in range(int(int(total_job) / int(total_machine)))] #define available machine
for i in range(0, int(total_machine)):
    print("Machine " + str(i + 1) + " Job schedule")
    for j in range(0, int(int(total_job) / int(total_machine))):
        string_temp = string_temp + str(final_job_schedule[i][j] + 1) + " " #define final job schedule
        string_temp_2 = string_temp_2 +str(t_job_schedule[i][j]) + " "        #define tardiness job schedule
        string_temp_3 = string_temp_3 + str(w_job_schedule[i][j]) + " "       #define job weight schedule
    print(string_temp)
    print(string_temp_2)
    print(string_temp_3)
    string_temp = ""
    string_temp_2 = " "
    string_temp_3 = " "
cout_temp=0
with open('output.csv', 'w', newline='') as csvfile:
    writer = csv.writer(csvfile)
    for i in range(0, int(total_machine)):
        string_temp="Machine " + str(i + 1) + " Job schedule"
        writer.writerows([[string_temp]])
        string_temp=""
        string_temp_2 = " "
        string_temp_3 = " "
        for j in range(0, int(int(total_job) / int(total_machine))):
            string_temp = string_temp + str(final_job_schedule[i][j] + 1) + " "
            string_temp_2 = string_temp_2 +   str(t_job_schedule[i][j] ) + " "
            string_temp_3 = string_temp_3 + str(w_job_schedule[i][j]) + " "
            array_temp [j] =final_job_schedule[i][j] + 1
            array_temp_2 [j] =t_job_schedule[i][j]
            array_temp_4[j] = w_job_schedule[i][j]
            if t_job_schedule[i][j]<0:
                cout_temp=cout_temp+((t_job_schedule[i][j])*(w_job_schedule[i][j]))
```

```
            array_temp_3[i]=array_temp_3[i]+cout_temp
            cout_temp=0
            writer.writerows([array_temp])
            writer.writerows([array_temp_2])
            writer.writerows([array_temp_4])
            string_temp = "Machine " + str(i + 1)+" Total Weigth"
            writer.writerows([[string_temp]])
            writer.writerows([[array_temp_3[i]]])
        string_temp = "All Machine Total Weigth"
        writer.writerows([[string_temp]])
        writer.writerows([[sum(array_temp_3)]])



#set up the enter windows for k value
root = Tk()
root.title('Job scheduling Ver0.15')
#root.geometry('300x200+1000+200')  可以使用預設內容的方式，使視窗自動調節
root.geometry()
Label(root, text = 'k1：').grid()
entry_1 = Entry(root)
entry_1.grid(row=0, column =1)
Label(root, text = 'k2：').grid(row = 1, column = 0)
entry_2 = Entry(root)
entry_2.grid(row = 1, column = 1)
Label(root, text = 'k3：').grid(row = 2, column = 0)
entry_3 = Entry(root)
entry_3.grid(row = 2, column = 1)
button = Button(root, text = 'Choose CSV file', command=search)
button.grid(row = 0, column = 4)
button = Button(root, text = 'Start scheduling',command=acc_job)
button.grid(row = 1, column = 4)
#listbox = Listbox(root, width = 80)
#listbox.bind('<Double-Button-1>',click)
#listbox.grid(row = 3, column = 0, columnspan = 5)
```

CURRICULUM VITAE

Name: Li Tzu Yi

Education:

B.A., Chung Yuan University, June 2015
Major: Industril and System Engineering

M.S., Chung Yuan University, June 2017
Major: Industril and System Engineering

M.S., University of Wisconcin-Milwuakee, December 2018
Major: Industril Manufacturing Engineering

Dissertation Title: Heuristic Algorithm to Minimize Total Weighted Tardiness on the Unrelated Parallel Machine with Sequence Dependent Setup and Future Ready Time

Teaching & Research Experience:

Teaching assistant, Chung Yuan University, Sep 2015-June 2017