UNIVERSITY OF OKLAHOMA

GRADUATE COLLEGE

RESOURCE PLANNING WITH EMBEDDED JUST-IN-TIME CHARACTERISTICS FOR MAKE-TO-ORDER DISCRETE PRODUCTION SYSTEMS

A DISSERTATION

SUBMITTED TO THE GRADUATE FACULTY

in partial fulfillment of the requirements for the

Degree of

DOCTOR OF PHILOSOPHY

By

WASSAMA SANGPLUNG
Norman, Oklahoma
2011

RESOURCE PLANNING WITH EMBEDDED JUST-IN-TIME CHARACTERISTICS FOR MAKE-TO-ORDER DISCRETE PRODUCTION SYSTEMS

A DISSERTATION APPROVED FOR THE
SCHOOL OF INDUSTRIAL ENGINEERING

BY

_____

Dr. Scott Moses, Chair

_____

Dr. Hillel Kumin

_____

Dr. Hank Grant

_____

Dr. Suleyman Karabuk

_____

Dr. Le Gruenwald

Acknowledgments

First and foremost, I would like to express my sincere gratitude to my advisor, Dr. Scott Moses, for his excellent guidance, support, patience, and caring during my doctoral studies. His professional expertise and experiences motivated me to accomplish my PhD degree. I would also like to thank Dr. Hillel Kumin, Dr. Hank Grant, Dr. Suleyman Karabuk, and Dr. Le Gruenwald for serving on my dissertation committee and for their valuable advice.

I would like to thank the School of Industrial Engineering, University of Oklahoma for granting me a scholarship and giving me an opportunity to be a part of this prestigious school. I am also thankful to all the staff of the school of industrial engineering, Amy Piper, Jean Shingledecker, and Cheryl Carney for their always kind support.

I would like to thank all the friends I made along the way, for making my stay in Norman so memorable.

Finally, my greatest gratitude goes to my family; my father, my mother, my brother, and my husband, who have always supported, encouraged, and believed in me. Without their unconditional love, I would not have been where I am now.

Contents

List of Tables

List of Figures

Abstract

Adopting a make-to-order (MTO) production mode allows manufacturers to accommodate a wider variety of customer requirements without a prohibitive increase in inventory of finished products. Since MTO production involves a wide variety of process features, a resource plan is necessary to coordinate production of customer orders so that resources are used efficiently and customer order due dates are met.

This dissertation develops optimal and heuristic methods that embed characteristics of the just-in-time (JIT) philosophy to create resource plans for MTO environments. JIT is a well-known productivity concept in which jobs are attempted to be started near to and finished on their due dates in order to reduce work in process (WIP), inventory, lead time, and cost. In the JIT philosophy, an ideal plan for a single order would have zero queue time, zero earliness, and zero lateness. The methods remain cognizant of the ideal plan for an order as they make adjustments to the actual plan for each order that are necessary to accommodate resource constraints.

A new binary integer linear programming (BILP) model is formulated to solve resource planning problems in MTO environments. The objective function contains weighted costs for earliness, tardiness, lead time, and subcontractor capacity. The initial solution is generated using the ideal plan for each order. Extensive computational results show that this initialization method often reduces computational time such that the BILP model can reach the optimal solution within an acceptable amount of time when it otherwise could not.

Due to the extremely limited scalability of optimal methods, which renders them inappropriate for most realistic make-to-order environments, a heuristic method utilizing tabu search is developed to solve resource planning problems. This is a two-phase algorithm. Like the optimal method, the tabu search algorithm in the first phase also generates an initial solution using the ideal plan for each order, and it then creates a finite capacity plan. It furthermore remains cognizant of the ideal plan in the second phase as it searches for solutions that respect resource constraints but that have good performance in terms of order earliness, tardiness, and lead time. A benchmark study of the developed algorithm reveals that the tabu search algorithm provides better solutions in terms of problem scalability and solution quality than other methods including the BILP optimization approach and other heuristic approaches such as FIFO or EDD.

CHAPTER 1

Introduction

1.1 Overview

In recent years, an intensely competitive market has emphasized the concept of customer-driven manufacturing which means that manufacturers are required to deal with more differentiated product features, tight delivery performance and low product cost. The production environment has shifted from make-to-stock (MTS) to make-to-order (MTO). MTO production focuses on creating products when the customer orders are placed. This is why an MTO environment can enhance production performance by reducing inventory, shortening product lead time and increasing the availability of customer preferred features. However, this method also involve more complexity in production management since MTO production processes each job with an individual routing and it also maintains little or no inventory, thus preventing it from absorbing any fluctuations.

A challenge of MTO production is how manufacturers can visualize existing resource capacity and effectively utilize the resources to support customer requirements. The main constraints of MTO production are not only capacity limitations, but also various product customization requirements. The combination of a great number of jobs and various routing types would create unbalanced resource requirements at different times. This situation could induce a severe capacity shortage problem.

Capacity planning is an important tool for production management and improvement. Efficient capacity planning can make production run smoothly and more easily detect certain production problems, such as capacity shortage and product delayed shipment, at an earlier stage. Planning can be categorized into the following three levels based on its function and focused activity: strategic planning, tactical planning, and operational planning. A framework of the position of each planning type (Giebels et al. 2000) can be seen in Figure 1.

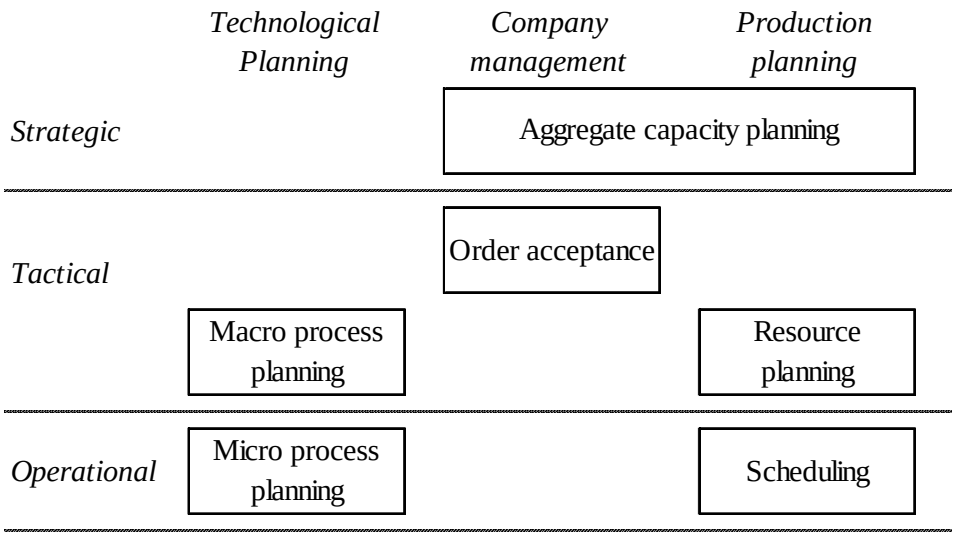


Figure 1: Planning framework

This dissertation emphasizes role of capacity planning at the tactical level, such as resource planning. A tactical plan, unlike a strategic plan or an operational plan, concerns the status of the available capacity of resources in a production system. It typically determines capacity availability and allocates sufficient resources to respond to customer orders as optimally and profitably as possible.

Maximizing profitability is the most desired objective in manufacturing. To achieve the goal, manufacturers must consistently produce high quality products with

low unit costs, and high service levels such as on-time order promising (Revelle 2001). An appropriate resource plan can establish the feasibility and suitability of a set of customer orders which significantly affects manufacturing efficiency and respects to a manufacturer's profitability (Hans 2001).

MTO resource planning is considered as a job shop planning problem. To generate a resource plan, two solving methods, which are an optimization and heuristic approaches, are typically used. The optimization modeling approach is a conventional method that aims to find an optimal solution. The optimization algorithm is successfully applied to solve small problem instances. However, it could not solve large problem instances within a limited time frame. Problem scalability and variability are the key factors that increase problem difficulty and limit solver performance. The optimizer will require more computational time to reach an optimal solution or even a good feasible solution. This is because a job shop planning problem is among the hardest combinatorial optimization problems or NP-hard problems. It means that an optimal solution cannot be executed within the polynomial computational time when a problem is complicated.

Approximation algorithms or heuristic approaches are alternative methods which are widely used in solving planning problems since they are able to provide a near optimal solution with reasonable computational time. Tabu search is introduced to develop an ideal schedule of resource planning since, compared with other heuristic methods, can provide near-optimal solutions that are among the most effective tackling difficult problems (Gendreau 2002).

In manufacturing, many approaches and algorithms from a number of researchers have been proposed to enhance system performance. One of the most effective productivity concepts is just-in-time (JIT) philosophy. The JIT concept is widely used as a basic idea in productivity improvement; its core idea is to produce jobs at the right times and places. Applying JIT to resource planning can create a resource plan as similar as an ideal plan in which jobs attempt to be processed and finished on their due date in order to reduce work in process (WIP), finished goods inventory, job lead times and production cost.

JIT philosophy is adapted to develop these two solving methods, the optimization and heuristic approaches, to increase planning performance as well as an ability to deal with variability in the system. A basic means of applying this ideal concept to each of the solving methods is described below.

1.1.1 An optimization approach

In a generic integer linear programming model, a branch and bound algorithm is used to find an optimal solution. The main procedure of the algorithm is to explore a tree of continuous relaxations of the original mathematical model. Danna et al. (2004) stated that the system is particularly effective when the continuous relaxation of the problem is a good approximation of the convex hull of the feasible solution. This statement also aligns with the suggestion of Tanaka et al. (2003), Rabadi et al. (2004), and Fischetti et al. (2005) that an effective initial solution helps a solver reach a solution faster. Thus, a solution initialization approach is developed for tuning optimizer performance. A new proposed algorithm adapts the ideal concept of JIT planning to

generate initial solutions. It is expected that an ideal initial solution can guide an optimizer to better solution spaces and reach an optimal solution more efficiently.

1.1.2 A tabu search approach

Tabu search is a meta-heuristic method that uses non-randomness to search the directions and creates short-term memory to prevent search cycling (Glover 1986). The neighborhood structure is a main element of tabu search which directly affects the efficiency of new solutions. This is because it is used to design solution spaces and seek efficient solutions. With JIT philosophy, a new local search algorithm is formulated to find feasible solutions that come as close as possible to being ideal. Also, a new parameter which defines the distance between a current solution and an ideal solution is determined in order to narrow the search and guide the search direction toward a desired solution.

1.2 Research objectives

The objectives of this research are summarized as follows.

(1) Develop a new binary integer linear programming (BILP) model to represent resource planning problems. The model is formulated to find an optimal resource plan that satisfies multiple quality criteria such as earliness, tardiness, and lead time. To consider all criteria and solve them simultaneously, a weighted cost approach is implemented into the objective function. An effective initial solution based on the JIT concept is implemented for initializing numerical computation in order to improve optimizer performance. An analysis of the solution's performance is presented in terms of how an effective initial solution enhances planning capability.

(2) Develop a tabu search heuristic approach to solve resource planning problems. A new local search algorithm embedded the JIT concept is developed to determine resource plans that are as close as possible to customer due dates in order to minimize earliness, tardiness, and lead time. An investigation of algorithm performance is emphasized to define the ability to solve combinatorial problems.

(3) Investigate and gain insight into the impact of variability on planning performance of both the optimization and tabu search approaches in terms of solution quality such as computational time and optimality gap.

1.3 Organization of the dissertation

This dissertation is organized as follows. Chapter 2 reviews the related literature on generic planning in MTO environments, the optimization and tabu search approaches to solving generic capacity planning problems and specific planning problems like resource planning.

Chapter 3 presents a new binary integer linear programming (BILP) model to represent resource planning problems. Rather than starting the calculation from scratch or choosing random solutions, an efficient solution initialization concept is introduced to guide an optimizer toward the desired solutions. An investigation of the performance of planning results with different combinatorial problems is examined.

Chapter 4 presents a new tabu search algorithm to solve resource planning problems. The proposed algorithm is created with the JIT concept in which a resource plan aims to improve a solution from earliness, tardiness, and lead time. An experimental study is conducted to investigate the algorithm's performance at different

testing problems. Furthermore, a benchmark study of the proposed algorithm and other heuristic methods is also examined.

Chapter 5 performs a comparative study of the tabu search algorithm and the optimization approach. The experiments investigate the effects of the scalability and planning performance of these two solving methods. An analysis is then performed in order to define the most effective approach.

Chapter 6 presents a new resource planning application which is developed by using the Java language. The details of this application, such as the component list and the function of each component, are presented.

Chapter 7 discusses the conclusions of this dissertation and makes suggestions for future studies.

CHAPTER 2

Literature review

2.1 Introduction

In this chapter, the relevant literatures of tactical planning in discrete time environments are reviewed. The scope of the study consists of generic resource planning in MTO environments, an optimization approach and a tabu search approach for resource planning.

2.2 Resource planning in make-to-order environments

A basic concept of MTO systems is to process jobs when they are required. It differs from make-to-stock (MTS) production in that MTS will early produce orders and store them as inventory. Even though MTS production has an advantage in fast response of customer requirements, it has many disadvantages such as stocking inventory, increasing product lead time, and reflecting a higher total cost. Moreover, it has constraints on supporting a wider variety of product features. Manufacturing hence has been shifted to MTO production.

Capacity planning for MTO can be categorized into three levels which are strategic planning, tactical level planning, and operational planning. The planning position framework can be seen in Figure 1. Strategic planning is a plan which defines the direction of manufacturers made by management level. It is a long range plan which focuses on resources, such as capital and people, and aligns to business strategies. In tactical planning level, a plan aims to create resource assignment to support customer

demands from strategic planning with respect to their due dates. Certain levels of capacity adjustment, such as using alternative resources and working overtime, are allowed in this planning level. Operational planning is a short term planning which presents operating schedule of customer orders by details such as start date and time by operations of each order. The great explanation of distinction among these three planning types can be seen in Hans (2001).

Resource planning is considered as a tactical planning strategy which is located between strategic planning and operational planning. An improvement of tactical planning is necessary. This is because a feasibility and suitability of a tactical plan in resource assignment positively maintains production smoothing and customer order promising. However, there has not been much research on an improvement of tactical planning. Much attention has been paid to research improvement of strategic planning and operational planning since strategic planning directly involves business goal and profitability and operational planning concerns a detailed task plan with respect to production outputs.

The relevant research is addressed by Hans (2001). He proposed two solving methods, which are the deterministic modeling approach and various heuristic methods, for generating resource plan in tactical planning problems. In an optimization approach, the MILP model was developed to find an optimal resource plan which provided reliable due date and minimizes total cost from subcontracting capacity and tardiness penalty. The optimization model provided an order plan which indicated periods of tasks of each order. However since there are exponentially many feasible orders, the model was difficult to obtain an optimal solution within a reasonable time. The heuristic

methods, such as stand-alone heuristic, rounding heuristic, and improvement heuristic, was implemented to solve resource loading problems with the branch and price algorithms.

Regarding resource planning, it appears that several solving methods have been developed to create an efficient resource plan. The general purpose of planning is to optimally manage existing resources and provide sufficient capacity to ensure meeting customer requirements as well as maintain manufacturing efficiency. Various alternatives to increase capacity, such as increasing lead time of some customer orders, expanding operator capacity, and using subcontracting capacity, are considered to smooth production from variability. Wullink et al. (2004a) applied the flexible resource loading under uncertainty approach from Hans (2001) to solve the robust resources loading problems. They used subcontractor capacity as capacity flexibility to deal with uncertainty activities in the production system. Two multi-objective optimization approaches was developed a robust resource plan which trade-off between the costs for using nonregular capacity and the robustness of a plan.

Wullink et al. (2004b) presented the mixed integer linear programming (MILP) model to quote reliable due dates for robust resource planning in manufacturer-to-order production system. Their scenario-based solution approach aimed to manage resource capacity when involved a wide range of variability types such as work content, capacity availability, resource requirement, and activity occurrence. A planning investigation was paid attention to a trade-off between the expected delivery performance, such as tardiness, and the expected costs of using non-regular capacity.

Although the previous studies have developed the computational models to solve resource planning problems and used the alternative methods to increase the robustness of an optimization modeling approach, the previous research has been particularly attempted to minimize either subcontracting cost or tardiness cost. Limitation of concerned objectives in solving methods could drop quality of resource planning since it ignores other performance criteria.

According to planning performance directly reflects manufacturing efficiency which respects to maximizing profitability, several improvement concepts have been proposed to increase a quality of planning. In recent years, significant emphasis has been placed on Just-in-time (JIT) philosophy. JIT has been described as improvement strategy with the objective of producing the right product at the right time. Adapting JIT into planning, customer orders attempt to be processed and finished exactly on their due date to reduce work in process (WIP), inventory, and production costs (Baker and Scudder 1990). Hence, to obtain an efficient resource plan, an integration of multiple objectives should represent an improvement in all facets like the JIT concept. For instance, a resource plan with embedded JIT not only aims to generate a feasible set of operation times of jobs but it also need to improve production performance in terms of job earliness, tardiness, lead time, WIP, inventory, and other operating costs such as alternative capacities.

2.3 An optimization approach for resource planning

Resource planning problems of MTO production is considered as a job shop planning problem. An integer linear programming (ILP) model has been widely used to represent the problems. ILP is the minimization or maximization of a linear function

subject to linear equality and inequality constraints. With n variables and m constraints, the ILP model can be expressed as the form below;

minimize $c^T x$

subject to $Ax <= b$

$l_i <= x_i <= u_i$ for $i = 1 \ldots n$

x_i integer for all i in D which is a subset of $\{1 \ldots n\}$,

where A is a m x n matrix

When all unknown variables are binary (0 and 1), it is called binary integer linear programming (BILP). The binary variables present the accepted or rejected situations. For instance, if an event occurs, the variable is 1. Otherwise, the variable is 0. Meanwhile, when some or all of the variables are integer values, the model is known as mixed integer linear programming (MILP).

NP-hard or nondeterministic polynomial-time hard is a class of problems in which it is not provably NP and at least as hard as the hardest problems in NP. It means that it requires exponential time or even worse to solve a solution. From Fisher et al. (1983), it is clear that resource planning problem is NP-hard since the problem instances of this research involve with 3 or more machines and number of operations per job.

In an ILP model, there are several methods to enhance an ability of optimizer to determine a great solution. This research focuses an improvement in three areas as follows:

(1) Branch and bound improvement

(2) Constraint relaxation

(3) Solution initialization

(1) Branch and bound improvement

The ILP model is a mathematical optimization technique that aims to determine an optimal solution. The problem is considered to be solved when an optimum is found. Complicated problems, such as high variability of input data and a large number of variables, will deter solver performance and result in longer computational time. Many authors have developed and applied various strategies to improve an ability of ILP model for solving a planning problem. In ILP problems, a branch and bound algorithm is typically used to solve an optimal solution. Its procedure consists of three main elements: initialization, branching, and bounding (Clausen 1999). The algorithm uses a dynamically constructed tree structure to represent solution spaces. The search starts at a root node or an initial solution. The branching operation then determines the next set of possible nodes from which the search could progress. The bounding procedure selects the operation which will continue the search and is based on an estimated lower bound (LB) and the currently best achieved or upper bound (UB) solution (Jain and Meeran 1999).

Branch and bound algorithm has been developed so far to enhance solution performance. Fisher (1973) applied Lagrangian Relaxation (LR) to tighten lower bound in branch and bound algorithm. Potts (1980) and Carlier (1982) proposed the techniques to tight bounds of the single machine problem. They found that the strongest bound was obtained from the makespan of the bottleneck resource. Fisher et al. (1983) proposed surrogate duality relaxation by assigning a weighted linear combination to certain constraints to yield strong lower bound. Sourd and Kedad-Sidhoum (2003) focused on producing a good lower bound which was determined based on the decomposition of

the tasks into single execution time operations to obtain good feasible schedules and efficient branching rules. All the past research emphasizes tightening bounds to reduce search space for obtaining an optimal solution.

Currently, a MILP improvement focuses on an integration of branch and bound and local search which provides more promising solutions than the previous research. Fischetti and Lodi (2003) proposed local branching strategy for exploring an explicit neighborhood of a MILP solution. They developed the proposed method based on three main procedures of local search which are defining a neighborhood, searching the neighborhood, and performing diversification. The concept of this solution strategy is exploring neighborhoods by means of a black box general purpose MIP solver, thus exploiting the level of complexity by the MIP solvers. The method efficiently solves a complicated MIP problem. However, the disadvantage of this method mentioned by Danna et al. (2004) is an increasing cost of each node from accumulative dense reverse neighborhood constraints.

Danna et al. (2004) applied some ideas of Fischetti and Lodi (2003) and developed a new technique which was the relaxation induced neighborhood search (RINS) for improving MILP neighborhood search. They proved that the RINS performed better than the local branching method from Fischetti and Lodi (2003). The proposed method can provide the good solutions within limited times and it also improved a given solution which was either good or poor quality. This is because the RINS explores a neighborhood of both the incumbent and the continuous relaxation. Furthermore, this method is convenient to implement in MILP solving since it has been embedded as a tool in the commercial solver package ILOG CPLEX, MILP solver.

Thus to enhance branch and bound performance, the RINS method is applied to solve the resource planning model of this research.

(2) Constraint relaxation

Constraint relaxation is necessary to reduce model complexity. The relaxation will modify some attributes of the relationship defined by a constraint. As the term “relaxation” implies, the modification allows a wider range of relationships. Once a constraint is relaxed, the problem is altered because a relationship that was not allowed in the original problem is now acceptable (Beck 1994). Relaxing problem constraints can reduce some difficulty and enhance an optimizer capability to find an optimal solution. In ILP, relaxing a constraint means increasing the right-hand side for a $\leq$ constraint and decreasing the right-hand side for a $\geq$ constraint (Heyl 2010).

There are many ways to relax constraints in production system. Miyashita (1997) discussed the suggestion about problem adjustment when the solution cannot be solved with a reasonable time in multi-agent distributed scheduling problems. He addressed that some constraints should be relaxed to allow some flexibility in modifying local schedules among agents in order to obtain a feasible job shop plan. The relaxed samples are expanding lead time of jobs, increasing capacity of resources (over time), canceling jobs, subcontracting jobs to other planner agents, and subcontracting some operations to other scheduler agents.

Subcontracting capacity is widely used as flexibility in capacity planning problems. Kamien and Li (1990) proposed the process of distributing capacity load between subcontracting firms and subcontractors and proposed production planning

model to achieve production smoothing. Frederix (2000) improved the make-or-buy decision process between using its own capacity and subcontracted facilities to solve resource constraints. Bertrand and Sridharan (2001) used subcontractor in a make-to-order manufacturing system to minimizing tardy deliveries and maximize delivery reliability.

It obviously sees that subcontracting capacity can make some improvement in various production level instances since it can relax a capacity constraint and absorb variability. However this alternative capacity is expensive. A trade-off between a capacity gain and an increasing production cost needs to be emphasized. Merzifonluoglu et al. (2006) proposed the profit-maximizing production planning model for determining optimal plan and internal production capacity levels of subcontracting and overtime capacities. Bertrand and Sridharan (2001) developed heuristic decision rules for determining when and which orders should be subcontracted in a make-to-order manufacturing system to minimize total cost while minimizing tardy deliveries and maximize delivery reliability.

From the point of view of the above research, subcontracting capacity is applied into the deterministic tactical planning model of this research as capacity constraint relaxation. To generate an efficient resource plan, a trade-off of cost and additional capacity will be investigated.

(3) Solution initialization

An effective initial solution is also an important feature which can improve performance of ILP models. It is well known that performance of branch and bound

algorithm relies on the quality of the implemented upper bound. A good approximation of initial solution, which is served as an upper bound, potentially explores the solution faster and obtains an effective optimal solution since it can speed up the search from guiding solutions to the good paths (Danna et al. 2004)

Various algorithms such as dispatching rules and individual algorithms have been created to initialize a solution. Tanaka et al. (2003) proposed the earliest due date (EDD) sequence rule and the adjacent pairwise interchange (API) heuristic for initializing the solution in the single-machine earliness-tardiness scheduling problem. Zribi et al. (2008) applied the dispatching rules, assignments, earliest due date (EDD), earliest operation due date (EOD), modified due date (MDD), and modified operation due date (MOD), for initializing an algorithm of tabu search to minimize total tardiness in the flexible job shop problem.

Corry and Kozan (2004) investigated several initial solutions settings, which are earliest due date (EDD), first in first out (FIFO), critical ratio rule (CR), and least slack remaining (SL), for the constraint job shop scheduling problems. They studied the algorithm performance when those initial solutions were applied to the meta-heuristic approaches, tabu search and simulated annealing. The results proved that a good initial solution dramatically reduced the computational times to solve the minimizing problem of total tardiness in the system.

Regarding solution initialization method, an individual algorithm on particular theories or objectives is another approach to improve an ILP model. Rabadi et al. (2004) applied the shortest adjusted processing time (SAPT) heuristic for starting their initial

solution in the machine scheduling problem. Branch and bound was used to search an optimal job sequence. The results shown that jobs with shorter adjusted processing times tend to be scheduled closer to the median position of the schedule and those with longer adjusted processing times were far from the median position.

Fischetti et al. (2005) introduced the scheme called Feasibility Pump (FP) to find a feasible solution for a generic MILP problem. Their approach started the problem with an almost feasible solution. The numerical results presented that FP outperformed ILOG CPLEX in most of the cases on the capability to determine the first feasible solution.

In the previous literatures, most authors mentioned that using a good initial solution substantially reduced the amount of computational time and improved the quality of solutions. In addition, it appears that an efficient solution, which provides the closer solution to an optimal solution, will provide better and more robust performance in solving problem instances than just applying a random solution or basic dispatching rules. An initial solution algorithm based on the JIT concept is therefore adapted to improve an optimization model to generate a resource plan which has good earliness, tardiness, and lead time

2.4 A tabu search approach for resource planning

An optimization algorithm successfully provides optimal solutions in small instance problems. Even though there are many reinforcement algorithms to improve an optimizer performance, the main drawback of optimization method is time-consuming when it deals with complicated problems. Also, job shop scheduling problems are

considered NP-hard problems which mean the computational effort grows exponentially with the increment of the problem size (Lawler et al. 1989). To determine an optimal solution, the optimization algorithm may not be an effective method to provide a solution within a reasonable time.

On the other hand, heuristic algorithms, which include priority dispatch rules (He et al. 1993), shifting bottleneck approach (Balas and Vazacopoulos 1998), meta-heuristic methods, and so on, have been developed and widely adapted since they can provide a near optimal solution within a relatively short computational period. In recent years, many meta-heuristic algorithms have been proposed for job shop planning, such as simulated annealing (Li and Mcmahon 2007), Tabu search (Taillard 1989, Nowicki and Smutnicki 1996), genetic algorithm (Zhang et al. 2008), ant colony optimization (Eswaramurthy and Tamilarasi 2009) and particle swarm optimization (Guoa et al. 2009). A greater overview of local search algorithms can be seen from Vaessens et al. (1996) and Jain and Meeran (1999).

Among different heuristic approaches, tabu search is widely recognized as an appropriate and efficient approach. Tabu search has been initially developed by Glover (1986). This method is an enhancement of well-known hill climbing heuristic, which uses a memory function to avoid being trapped at a local minimum. The general procedure of tabu search (Dell' Amico 1993) is presented in Figure 2. Let S denote a set of feasible solutions. The main procedure of tabu search starts from creating an initial solution independently and then using local search algorithm to define a set of neighborhood solutions $N(s)$ which each solution $s \in S$. A solution will be moved from the current solution to the best solution s^* in $N(s)$ when it satisfies the aspiration criteria

such as the minimum cost function *c(s)* and the attribute of the selected best solution is not forbidden. To prevent solution cycling and guide the search to unexplored solution regions, the attributes of solutions between the previous solution and new solution will be stored in the memory, which is called tabu list. If the attribute of the new solution match with information in the tabu list, the best solution of the system *s** would be remained until found new best solution which satisfies the aspiration criteria and does not belong to the tabu list. The neighborhood search will be repeated until the stopping criteria are true.

```
begin
    (find an initial feasible solution s);
    best := c(s);
    s* := s;
    Tabu_list := ∅;
    repeat
        Cand(s) := {s' ∈ N(s): the move from s to s' does not belong to
                    Tabu_list or it satisfies an aspiration criterion};
        (choose s̄ ∈ Cand(s) : s̄ has the minimum estimation of the cost
        function);
        (put a move which leads from s̄ to s in Tabu_list);
        s := s̄;
        if c(s) < best then
        begin
            s* : = s;
            best := c(s)
        end
    until stopping_criteria = TRUE;
    return s*
end
```

Figure 2: A generic tabu search procedure

Tabu search has been successfully applied to a large number of combinatorial optimization problems, especially in production scheduling domains. In job shop planning, Taillard (1989) has initially and successfully used tabu search to solve the job

shop scheduling problem. So far, numerous algorithms have been proposed and developed (Dell'Amico and Trubian 1993, Barnes and Chambers 1995, Nowicki and Smutnicki 1996, Armentano and Scrich 2000, and Zhang et al. 2007).

The implementation of tabu search is problem-oriented in that it needs to be defined in the individual elements such as initial solution, moving scheme, neighborhood searching strategy, tabu list, and aspiration criteria (Nowicki and Smutnicki 1996). A neighborhood searching strategy is a significant procedure since an efficient algorithm is able to guide tabu search to reach a feasible solution faster and more efficiency. Taillard (1989) generated the neighborhoods from moving the pairs of successive operations on the critical path. Dell'Amico and Trubian (1993) developed two neighborhood structures from Van Laarhoven et al. (1992). Nowicki and Smutnicki (1996) developed Taillard's algorithm by considering the move only at the first and the last two blocks of single critical path. Armentano and Scrich (2000) presented the tabu search approach to minimize total tardiness for the job shop scheduling problem. The method used dispatching rules to obtain an initial solution and the neighborhoods are created based on the critical path of the jobs. From that previous research, all research has been focused to generate neighborhoods by selecting possible moves on critical path in order to minimize tardiness and makespan.

According to the JIT concept, it can be seen that improving only tardiness or makespan might not be an optimal way to improve production system. On the other hand, it might create either more WIP or inventory when jobs avoid lateness by finishing early. From this point of view, resource plans need to be developed based on

JIT. This means that jobs will be executed on their due date or closely to the due date as possible in order to minimize both tardiness and earliness.

The most relevant works that involve the neighborhood structure with the JIT concept are discussed as follows. He et al. (1993) proposed the improvement method in job shop planning which attempted to move backward and forward in some specific operations of tardy jobs in order to reduce total tardiness. Their proposed algorithms, the right shift move and the left shift move, successfully improved the tardiness. Even though the algorithm was created based on JIT, an improvement considered only the tardiness problem and ignored other factors such as earliness and lead time. This situation can create longer job lead time or greater number of WIP.

James (1997) presented the tabu search approach to minimize total tardiness for the job shop scheduling problem. His scheme applied the random candidate selection procedure to generate neighborhoods and then used the early and tardy based approach to narrow the candidates down further. This research mainly solved common due date problems in which all jobs are due on the same date.

Imanipour and Zegordi (2006) developed the tabu search with backward scheduling in flexible job shop (FJS) problem. Rather than routing is known and fixed, FJS planning is involved with varied routings, which each operation of jobs can be processed on alternative available machines. Their objective was to define the best routing of each job which minimized total weighted tardiness and earliness. The backward algorithm was used to determine the start time of each operation of jobs in operational planning. In spite of the algorithm focused to process the jobs close to the

due dates, they ignored the limitation of due date tightness. Disregarding of lead time to process each job can cause an infeasible plan.

Zhu et al. (2010) proposed a modified a tabu search method in the job shop planning problems with concerned JIT environment. To improve a solution based on the JIT philosophy, they started the algorithms from developing an initial schedule which was generated by an arbitrarily selected dispatching rule. Their proposed neighborhood structure performed the forward and backward moves with the feasibility and lower bound checks to reduce computational effort and create feasible operational plan. This research restricted to solve the problems with common number of operations per job to minimized three costs of WIP holding cost, earliness cost, and tardiness cost.

Despite some progress gained by applying JIT concept, there are still a few researches which expand this ideal concept to the complex job shop problems such as multiple machines, uncommon due date, and uncommon number of operations. These variabilities reflect more complicated problem. Under these circumstances, this research proposes a tabu search approach which adapts the JIT concept to the main elements of algorithms including initial solution and neighborhood structures. A resource plan at the tactical level aims to create with an efficient performance of earliness, tardiness, and lead time.

CHAPTER 3

An optimization approach for resource planning

3.1 Introduction

This chapter presents an optimization model to study resource planning in make-to-order (MTO) environments. With characteristics of the MTO system, jobs arrive the system with different periods of time and they are also attempted to be processed on different routings. This situation creates unbalanced resource requirement which may cause capacity shortage problem in certain periods. To visualize the production system and efficiently manage existing resources, resource planning is essentially used to obtain an appropriated schedule that accommodates time to utilize the available resources based on customer demands. Customer demands generally need to be satisfied with short lead time and low production cost. In manufacturing, certain production constraints and variabilities, such as demand fluctuation, resource availability, process customization, etc., reflect increasing difficulties of resource planning. To generate an efficient resource plan, a binary integer linear programming (BILP) model is therefore proposed to represent the system. The model is formulated to find an optimal resource plan that satisfies multiple criteria such as minimizing tardiness and earliness, minimizing job lead time, and maximizing resource utilization. To consider all criteria and solve them simultaneously, a weighted cost approach is applied into the objective function in which our primary goal is minimizing total weighed cost. In the optimization model, which involves some variability such as due date tightness, resource availability, and routing variation, an additional capacity is applied as system

flexibility to smooth a production system. This alternative capacity is assumed as a relaxing capacity constraint since regular capacity is restricted.

Even though the optimization approach can provide an optimal solution, a NP-hard problem like a job shop planning problem exponentially increases computational time to reach an optimal solution or a feasible solution, especially when dealing with problem scalability and variability. In a BILP problem, a branch and bound algorithm is used to find an optimal solution. The basic concept of this algorithm is to explore a tree of continuous relaxations of the original BILP model. To increase an optimizer performance, one of the effective techniques is starting the calculation with a good approximate solution. This aligns with the suggestion from the research of Tanaka et al. (2003), Danna et al. (2004), Rabadi et al. (2004), and Fischetti et al. (2005) in which they stated that an effective initial solution can help the solver reach a solution faster.

To improve the quality of planning for dealing with variability, this study focuses on two topics. The first topic is to propose an efficient solution initialization approach for tuning an optimizer performance. To generate a resource plan which is similar to an ideal plan, a just-in-time (JIT) concept is applied for initializing a solution. The idea of JIT is to process and finish jobs on their due dates in order to reduce WIP, inventory, and lead time. This first study aims to investigate the performance of an effective initial solution whether it can provide a better solution quality in terms of objective value, earliness, tardiness, and optimality gap.

The second topic is to study an impact of planning results under variability from job release date, job due date, resource utilization, process time, and penalty cost in

individual area. The BILP model is used to determine the optimal resource plan that satisfies the customer requirements as well as minimizing total cost. Although the additional capacity is used to smooth production system, a higher operating cost from using these capacities affects increasing total cost of the system also. Hence, a trade-off between solution quality and additional cost is an interesting topic to be investigated. An effective plan would be generated by considering all objectives to ensure profitability whether using its own or additional resources.

This chapter is organized as follows. Section 3.2 describes the scope of an interesting problem. Section 3.3 concentrates on the methods of solution initialization. Section 3.4 presents the mathematical model for resource planning in MTO environments. Section 3.5 presents the details of experimental parameters and experimental design. Section 3.6 discusses the results of the resource planning which consist of the optimization improvement based on solution initialization and the effect of variability on production system. Finally, section 3.7 draws the conclusion of this study.

3.2 Problem statement

A resource planning problem is considered as job shop planning under restricted resource availability. The overview of an interesting production system in this research is shown in Figure 3. This job shop system consists of a set of H independent machine groups. In the system, job i needed to be processed on j operations. Each job can be processed on different routings and each operation must be executed on one machine at any instance of time without preemption. An arrival time of job is uniform distribution. Process time for each operation internally is known upon its arrival. Different

processing times in different jobs and operations are allowed in the model. Due to a discrete time concept, a planning horizon is cut into intervals called time buckets. Each time bucket consists of three types of capacities which are a current capacity of the existing resources, an additional capacity from subcontractor, and an additional capacity from extra resources. When the current capacity insufficiently supports customer requirements, the additional capacity from subcontracting and extra resources will be used to cover the excess demands with a high operating cost.

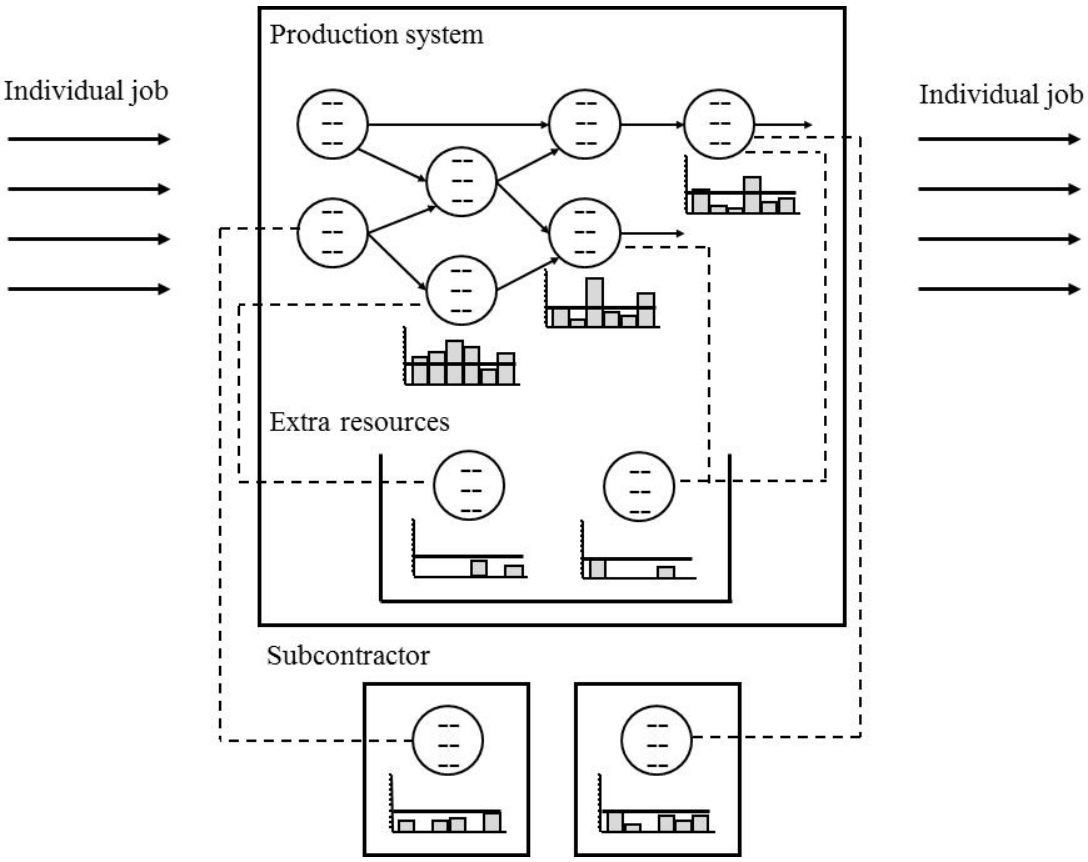


Figure 3: MTO environment

3.3 Solution initialization concept

A complicated problem from data variability, problem size, and model complexity could drop performance of an optimizer to find a solution in terms of

computational time or optimality gap. To enhance its capability, a good approximation of feasible solutions is proposed to start branch and bound approach and to efficiently guide to an optimal solution. In this research, two different settings of initial solution are proposed. The first algorithm generates a first feasible solution which is close to an ideal solution in which jobs are required to finish as close as possible to their due dates. A backward scheduling method is applied to project the solution which is close to optimal planning. Even though this plan creates an ideal resource plan, the limitation of capacity will cause overloaded capacities in certain resources. These excess capacities will be covered with additional resources at highly charged cost. Thus, the solution needs to be improved for obtaining a better resource plan with lower total cost. The second algorithm is used as a benchmark method. This method will assign the first solution that is far from the optimal solution by assuming all jobs are unplanned. Although the initial solution from an unplanned job method is all zero, it might be easier if the optimizer will repair the solution. The details of each method are described as follows.

3.3.1 Backward scheduling

In order to create a solution close to an ideal plan, a backward scheduling, or a latest possible start time concept (LPST), is used. This sequence begins by loading the last operation of a job to finish by its due date. It then continues by loading the preceding operation of the job to finish at the start time of the next operation. This process is continued, working backward in time, until the first operation of the job is loaded. An advantage of backward algorithm is that a schedule can be generated with no tardiness and earliness. Also, this method can reduce inventory, WIP and job lead

time. To initialize a solution, resource constraint relaxation is assigned by assuming ideal resource capacity. This means that resource can support as many jobs as possible without capacity restriction. This situation will create an overloaded capacity of the existing resources. Total loading in some periods might fall in the usage area of the additional capacities from subcontractor and extra resources. It is assumed that the cost of extra resources capacity is much more expensive than the cost of subcontracting capacity. Therefore, for additional capacity utilization, subcontracting capacity will be considered to use firstly then extra resource capacity.

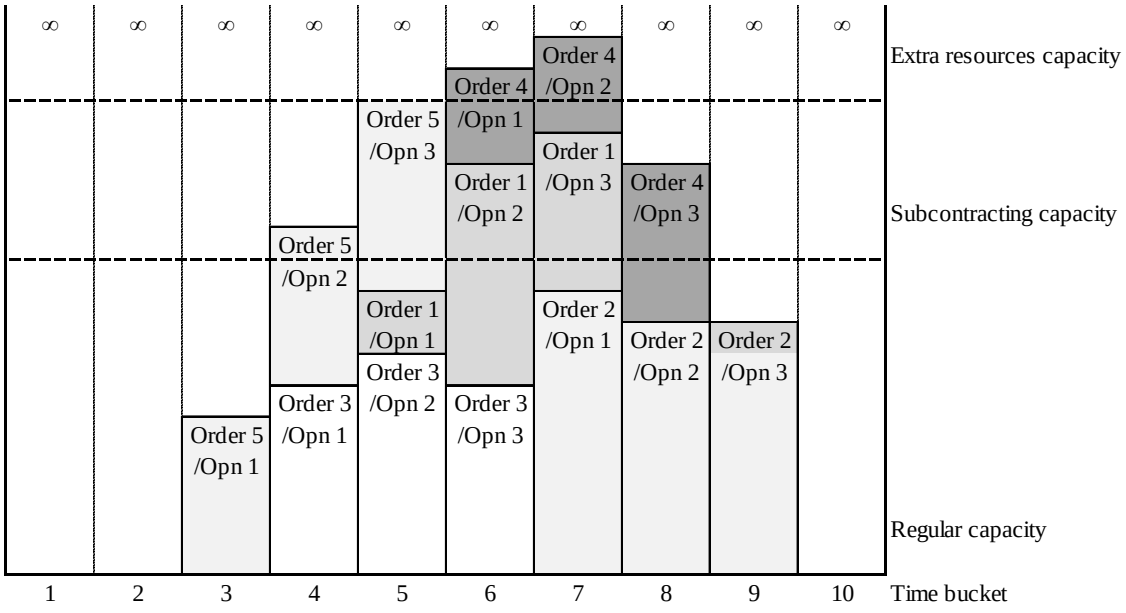


Figure 4: Loading profile of the backward scheduling example

Although the initial solution from the LPST can provide an efficient solution that all jobs can be finished by their due dates, using the additional capacity dramatically increased total cost of the system. To trade off the additional capacity and the operating cost, the optimizer will reassign an operation start time of each operation of each job (X_{ijk}) to minimize the total cost.

For a clearer view, an instance of backward scheduling will be presented. There are 5 jobs needed to be processed in the shop floor. Job list is (3, 2, 1, 5, 4) and due date is (7, 10, 8, 6, 9). Each job requires three identical operations. Jobs will be loaded to the system based on the job list. In this case, job 3 will be the first load. Since job 3 has due date at the beginning of period 7, the last operation (operation 3) should be processed and finished within period 6, and then calculate backward for the rest of operations. Thus, the times of operation 2 and 1 will be period 5 and 4, respectively. Figure 4 illustrates the loading profile of five jobs and the initial solution for this case is ((4,5,6), (7,8,9), (5,6,7), (3,4,5), (6,7,8)).

3.3.2 Scheduling an unplanned job

This method assumes that no jobs are processed in a planning horizon. Therefore, all operation start times of each operation of each job (X_{ijk}) in the initialization stage are assigned to be zero. Another cost function called an unplanned job cost is newly generated to use in the model. This cost will be charged a job that is not planned in the planning horizon. The initial solution for this case would have a very high total cost from penalty of unplanned jobs. According to the proposed initial solution is far from an optimal solution, an optimizer attempts to minimize total cost by reducing the unplanned jobs and assigning a better resource plan.

3.4 Mathematical model

The notation of all parameters in the optimization model is denoted as follows:

P_{ij} = Process time of operation j of job i

R_{ij} = Routing: index of resource that performs operation j of job i

S_i = Earliest start date of job i

D_i = Due date of job *i*

F_i = Finish date of job *i*

B_{hk} = Capacity of bucket *k* for resource *h*

B_{hk}^{s} = Subcontracting capacity of bucket *k* for resource *h*

B_{hk}^{x} = Extra resources capacity of bucket *k* for resource *h*

P_i^e = Earliness penalty for job *i*

P_i^t = Tardiness penalty for job *i*

P_i^l = Lead time penalty for job *i*

P_h^s = Subcontracting penalty for resource *h*

P_h^x = Extra resources capacity penalty for resource *h*

P_i^u = Unplanned penalty for job *i*

C_i^e = Earliness cost for job *i*

C_i^t = Tardiness cost for job *i*

C_i^l = Lead time cost for job *i*

C^s = Total subcontracting cost

C^x = Total extra resources capacity cost

C^u = Total unplanned cost

i = Index for set of jobs; $i = 1 \ldots I$

j = Index for set of operations required by a job; $j = 1 \ldots J$

h = Index for set of resources; $h = 1 \ldots H$

k = Index for set of buckets on each resource; $k = 1 \ldots K$

Decision variable:

x_{ijk} = 1 if operation j of job i is planned in bucket k, 0 otherwise

In order to generate resource loading plans, six cost functions must be achieved: (1) the minimization of costs due to earliness; (2) the minimization of costs due to tardiness; (3) the minimization of costs due to time spent during the process; (4) the minimization of costs due to the additional capacity provided by subcontractor; (5) the minimization of costs due to using extra resources capacity; and (6) the minimization of costs due to unplanned jobs. With the above notation, the planning problem can be stated as the following optimization model.

Minimize

(i) Earliness cost C^e

$$\sum_{i=1}^{I}((D_i - 1) - F_i)(P_i^e) \qquad \text{if } F_i < (D_i - 1) \qquad (1)$$

(ii) Tardiness cost C^t

$$+ \sum_{i=1}^{I}(F_i - (D_i - 1))(P_i^t) \qquad \text{if } F_i \geq (D_i - 1) \qquad (2)$$

(iii) Lead time cost C^l

$$+ \sum_{i=1}^{I}(F_i - kx_{i1k} + 1)P_i^l \qquad (3)$$

(iv) Subcontracting cost C^s

$$+ \sum_{h=1}^{H}\sum_{k=1}^{K}\left(\left(\sum_{i=1}^{I}\sum_{j=1}^{J}(x_{ijk}P_{ij})\right) - B_{hk}\right)P_h^s \qquad \text{where } R_{ij} = h$$

$$\text{if } \sum_{i=1}^{I}\sum_{j=1}^{J} x_{ijk} P_{ij} \;\leq\; B_{hk}\,;\; P_h^s = 0 \tag{4}$$

(v) Extra resources capacity cost C^x

$$+\sum_{h=1}^{H}\sum_{k=1}^{K}\left(\left(\sum_{i=1}^{I}\sum_{j=1}^{J}(x_{ijk}P_{ij})\right)-(B_{hk}+B_{hk}^{s})\right)P_h^x \qquad \text{where } R_{ij} = h$$

$$\text{if } \sum_{i=1}^{I}\sum_{j=1}^{J} x_{ijk} P_{ij} \;\leq\; (B_{hk} + B_{hk}^{s})\,;\; P_h^x = 0 \tag{5}$$

(vi) Unplanned job cost C^u

$$+\sum_{i=1}^{I}\left(1-\sum_{k=1}^{K} x_{i1k}\right)P_i^u \tag{6}$$

Subject to

(vii) Start time constraints

$$\sum_{k=1}^{K} kx_{i1k} \qquad \geq \qquad S_i \qquad\qquad i = 1,\ldots, I \tag{7}$$

(viii) Precedence constraints

$$\sum_{k=1}^{K} kx_{ijk} \qquad < \qquad \sum_{k=1}^{K} kx_{i(j+1)k} \qquad i = 1,\ldots, I, \text{ and } j = 1,\ldots, J \tag{8}$$

(ix) Production constraints

$$\sum_{k=1}^{K} x_{ijk} \qquad \leq \qquad 1 \qquad\qquad i = 1,\ldots, I, \text{ and } j = 1,\ldots, J \tag{9}$$

(x) Capacity constraints

$$\sum_{i=1}^{I}\sum_{j=1}^{J} x_{ijk} P_{ij} \quad \leq \qquad B_{hk} + B_{hk}^{s} + B_{hk}^{x}$$

$$\text{where } R_{ij} = h \qquad h = 1,\ldots, H \text{ and } k = 1,\ldots, K \tag{10}$$

(xi) Binary constraints

$$x_{ijk} \quad \in \quad \{0, 1\} \qquad i = 1,\ldots, I, j = 1,\ldots, J \text{ and } k = 1,\ldots, K \quad (11)$$

Where

$$F_i \quad = \quad \max \{kx_{iJk}\}_{k=1}^{K} \quad (12)$$

The descriptions of six cost functions are presented below:

- *Earliness cost* C^e: The earliness cost represents penalty from an early finished job. If the completion time F_i is less than the due date (D_i-1), the penalty would be charged. As in equation (1), an earliness cost is the amount of earliness time multiplied with an earliness penalty P_i^e
- *Tardiness cost* C^t: The tardiness cost represents penalty from a tardy job. If the completion time F_i is later than the due date (D_i-1), the penalty would be charged. As in equation (2), an tardiness cost is the amount of lateness time multiplied with an tardiness penalty P_i^t
- *Lead time cost* C^l: The lead time cost of equation (3) penalizes each time unit that job is in the system. This cost is charged since the job is processed until it is finished.
- *Subcontracting cost* C^s: The subcontracting cost represents penalty cost from using capacity of subcontractor. This cost occurs when total job loading in each bucket k of resource h is over regular capacity. Then, an additional capacity from subcontractor is needed to support an exceeding loading with a subcontracting penalty P_h^s as shown in equation (4).

- *Extra capacity cost* C^x: The extra capacity cost represents penalty cost (P_h^x) by using capacity of other resources, except from current capacity or subcontractor as Equation (5). This cost occurs when total loading in each bucket *k* of resource *h* is over the summation of regular and subcontracting capacities.
- *Unplanned job cost* C^u: Equation (6) presents total unplanned job penalty. This cost would active when job is not planned on the planning horizon. It normally occurs when the initial solution is set by an unplanned job method.

Manufacturing environment has several particular characteristics that lead to be the constraints of system. In this research, five restrictions are considered in which they can be described as follows:

- *Starting time constraint*: This constraint is used to ensure that an operated bucket of the first operation of job *i* should be processed after the earliest start time S_i or later on. Therefore, equation (7) implies that the operation time for the first operation of each job would be greater or equal the job's release time.
- *Precedence constraint*: The operation precedence constraint states that an operation *j* cannot be started before its previous operation is completed. The precedence relation is given in equation (8).
- *Production constraint*: Due to discrete time planning, each operation *j* of job *i* is restricted to load in one bucket at a time only. Equation (9) implies that the summation of all operated buckets in the planning horizon for an operation *j* of job *i* is equal to one.
- *Capacity constraint*: Total capacity in each bucket consists of capacity from current resource B_{hk}, subcontractor B_{hk}^s, and extra resource B_{hk}^x. Therefore,

total loading of assigned jobs in bucket *k* of resource *h* should not be exceeded total capacity as shown in equation (10).

- *Binary constraint*: Decision variable X_{ijk} of the model represents operation *j* of job *i* is processed on bucket *k.* The solution is represented by a binary variable 0 and 1 as equation (11). If operation *j* of job *i* is loaded on bucket *k*, decision variable is 1. Otherwise, it is 0.

3.5 Computational experiments

A real manufacturing system involves several variabilities both from outside and inside production system. According to law of variability, increasing variability can reduce the performance of a production system (Hopp and Spearman 2004). The computation experiments are numerically conducted to test an effectiveness of the model at different solution initialization settings and variabilities. To determine the ability of the resource planning model and its accuracy, the experiments aim to study the effect of system performance from four variability parameters which are weighted cost, due date tightness, resource utilization, and process time. The details of these parameters and experimental design are described in section 3.5.1 and 3.5.2, respectively.

3.5.1 Experimental parameters

(1) Weighted cost

In the objective function, weighted cost is an important factor that is used to convert multiple objectives function to be a single objective function. Cost in six areas, which are earliness cost (EC), tardiness cost (TC), lead time cost (LC), subcontracting cost (SC), extra capacity cost (XC), and unplanned job cost (UC), is needed to be

minimized. Since the variation of the weighted costs can significantly impact the planning results, it is expected that if any cost area is high, an optimizer will avoid that cost and search for a better result to minimize total cost. For example, if the tardiness cost is the most expensive cost, the planning tends to avoid job lateness. To study the effect of this variation, the experiments are created by varying cost in four areas, which are EC, TC, LC, and SC. For another two costs, UC and XC, which stand for the initial solution purpose, are fixed for all scenarios. UC and XC are assigned to equal to $500 per job and $300 per unit, respectively. Penalty in four areas is charged by dollar per unit and the summation of these four costs per unit is equal to $100. Three cases of weighted cost EC/TC/LC/SC are proposed as follows: 5/15/10/70, 5/70/10/15, and 25/25/25/25.

(2) Due date tightness

Due date tightness is an allowance time for processing job on the shop floor. It is also one of important factors since an ability to quote reliable due date or short due date tightness is necessary for recent competitive market. The experiments will observe the performance of planning when the tightness values are varied with uniform distribution from constant due date to larger range of due date. Since different due date tightness reflects different problem complexity, four levels of due date tightness, such as 3, [3, 6], [3, 9], and [3, 12], are generated to test the model.

(3) Resource utilization

Resource utilization is used to define system congestion. Five levels of utilization are generated as uniform distribution to test the model, which are [65%, 95%], [75%, 95%], [85%, 95%], [75%, 110%], and [55%, 140%]. Each case obtains an

average utilization at 80%, 85%, 90%, 92.5%, and 97.5%, respectively. Furthermore, the resource utilization parameter also uses to generate input data by determining the problem size or the number of jobs based on the congestion level in each testing problem.

(4) Process time

Process time is one of factors that explicitly impact the system performance when it has variability. Varied process time directly affects the number of bucket requirement, utilization of resource, and job cycle time. The experiment proposes three levels of process time variability to study the impacts. First is constant process time at five. Second is medium process time variability with uniform distribution [3, 7]. Last is high process time variability with uniform distribution [1, 9]. The mean process time for all three cases is equal so that the results from those cases are comparable.

3.5.2 Experimental design

The problem represents resource allocation in job shop production. Input data is generated based on resource utilization by using the data generator. Each job has an individual routing, which is randomly selected from three of five independent machines, to process the job. As discrete time planning, the planning horizon is divided into 15 time buckets. Loading capacity per bucket consists of regular capacity, subcontracting capacity, and extra resources capacity. The first two capacities are assumed to have a limited capacity at 100 units for all instances, but the extra resources capacity will have unlimited capacity. In order to study the system behavior when variabilities occur, the performance indicators, such as total cost, earliness, tardiness, and optimality gap are used to evaluate the problem. The experiments will be performed on a Pentium IV 1.73

GHz PC with 2 GB RAM. ILOG CPLEX 12 is implemented as an optimization solver to find a solution. Computational time for each case is limited at 1,800 seconds.

Recall that this research aims to investigate the impact of different solution initialization on the optimizer's improvement. Two proposed initial setting approaches are the latest possible start time (LPST) and unplanned job (UP) methods. In CPLEX, it uses branch and bound method to find an optimal solution. As branch and bound approaches explore a tree of continuous relaxations of the original MIP model, the system would particularly improve when the search area is more robust (Danna et al. 2003). Relaxation Induced Neighborhood Search (RINS) is one of the effective methods to enhance the search quality (Danna et al. 2004). RINS is a heuristic that explores a neighborhood of the current incumbent solution to find a new improved incumbent. The strength of RINS is that it explores a neighborhood both of the incumbent and of the continuous relaxation which plays symmetrical roles. This allows for RINS to improve quickly on poor incumbents and to be robust with respect to lose continuous relaxation. RINS is already included in CPLEX. It easily implement in CPLEX by simply setting parameter IloCplex::MIPEmphasis = 4. In this research, RINS is applied to develop our MIP model in both solution setting experiments. A greater detail of RINS is described in Danna et al. (2003 and 2004).

From Table 1, with three levels of weighted cost, four levels of due date tightness, five levels of resource utilization, three levels of process time, and two levels of initialization method, there are 360 combinations of experiment total.

Table 1: Summary of experimental parameters

Parameter	Level
Weighted cost (EC/TC/LC/SC)	5/15/10/70 5/70/10/15 25/25/25/25
Due date tightness (buckets)	3 [3, 6] [3, 9] [3, 12]
Resource utilization / Average utilization	[65%, 95%] / 80% [75%, 95%] / 85% [85%, 95%] / 90% [75%, 110%] / 92.5% [55%, 140%] / 97.5%
Process time (units)	5 [3, 7] [1, 9]
Solution initialization settings	LPST UP

3.5.3 Data generation

In MTO environments, jobs arrive a factory with different timelines. This situation reflects the variation of resource requirement in each period. To simulate the system, the number of job is generated based on the level of resource utilization. Instead of generating random job numbers to meet a required utilization; the resource utilization is used to define the number of job in each bucket. According to random number, the number of job as well as problem size in each scenario will be different. The procedure for creating an input data is described in the following steps.

Step 1: Generate resource utilization for each bucket *k* by random selecting a desired resource utilization from range [*a*, *b*] where *a* and *b* are the lowest and the highest resource utilizations, respectively.

$$Utlization_k = random[a, b]$$

Step 2: Determine a number of jobs in each bucket *k*.

$$Job_k = \left\lceil \frac{Utilization_k * Regular\ capacity}{Average\ process\ time} \right\rceil$$

Step 3: Find a resource factor. This factor is used to convert the number of jobs to represent total jobs requirement for all resources.

$$Factor = 1 + \left\lceil \frac{Number\ of\ operation}{Number\ of\ resource} + \frac{(Number\ of\ operation\ -\ 1)}{Number\ of\ bucket} \right\rceil$$

Step 4: Calculate total jobs in bucket *k*.

$$Total\ job_k = Round\ down\ (Job_k * Resource\ factor,\ 0)$$

Step 5: Calculate a final number of jobs in bucket *k* by adjusting $Total\ job_k$ based on $utilization_k$.

$$Number\ of\ job_k = (Total\ job_k * bucket_k) - \sum_{m=1}^{k-1} Total\ job_m$$

For instance, to generate an input data, assume that total capacity per bucket is 100 units with total 15 buckets in the planning horizon. Resource utilization is distributed uniformly [65%, 95%]. A job has to be processed on three operations in which each operation has an average process time at five. Table 2 shows the example of how the input data is determined in each bucket. In this case, total job is 96 jobs (the summation of 28, 20, and 48).

Table 2: Example of input data generation

Bucket	Resource utilization	Job loading	Job	Resource factor	Total job	Total job * Bucket	Number of jobs
1	82%	82	16.4	1.73	28	28	28
2	70%	70	14.0	1.73	24	48	20
3	94%	94	18.8	1.73	32	96	48

3.6 Results and analysis

This section presents the results of the optimization approach at various variabilities, such as weighted penalty, due date tightness, resource utilization, and process time. The result explanations are divided into two sections. The first part is the effectiveness of the solution initialization methods on the optimizer's performance improvement. The second part is the analysis of variability impact on planning results.

3.6.1 Impact of method for solution initialization

The numerical results of all experiments are presented in Tables 3-17. These tables are summarized the performance indicators of two initial setting methods at weighted cost 5/15/10/70, 5/70/10/15, and 25/25/25/25, respectively. From the tables, the impact of resource planning based on the performance measurement is analyzed and described in the following section.

Table 3: Results for weighted cost 5/15/10/70 and resource utilization [65%, 95%]

Duedate tightness	Process time	Number of jobs	LPST			UP		
			Optimality gap	Best solution (unit: x$1,000)	Computational time (sec.)	Optimality gap	Best solution (unit: x$1,000)	Computational time (sec.)
3	Constant	364	0%	32.9	74	0%	32.8	132
	Medium	384	0%	33.0	507	0%	33.0	817
	High	319	0%	31.4	42	0%	31.3	66
[3, 6]	Constant	364	2%	30.7	1,800	2%	30.7	1,800
	Medium	429	33%	44.7	1,800	50%	60.1	1,800
	High	390	18%	43.8	1,800	27%	41.0	1,800
[3, 9]	Constant	416	31%	43.6	1,800	49%	58.9	1,800
	Medium	360	11%	33.6	1,800	38%	48.3	1,800
	High	429	28%	41.7	1,800	62%	78.8	1,800
[3, 12]	Constant	372	4%	31.3	1,800	59%	73.1	1,800
	Medium	408	12%	33.9	1,800	74%	115.7	1,800
	High	377	5%	31.7	1,800	64%	83.7	1,800

Table 4: Results for weighted cost 5/15/10/70 and resource utilization [75%, 95%]

Duedate tightness	Process time	Number of jobs	LPST			UP		
			Optimality gap	Best solution (unit: x$1,000)	Computational time (sec.)	Optimality gap	Best solution (unit: x$1,000)	Computational time (sec.)
3	Constant	416	3%	33.5	1,800	2%	33.3	1,800
	Medium	377	3%	32.0	1,800	3%	31.9	1,800
	High	429	3%	33.8	1,800	2%	33.6	1,800
[3, 6]	Constant	377	2%	30.8	1,800	2%	30.8	1,800
	Medium	416	5%	31.5	1,800	11%	33.6	1,800
	High	396	10%	33.2	1,800	19%	37.2	1,800
[3, 9]	Constant	429	22%	38.3	1,800	53%	63.6	1,800
	Medium	377	4%	31.4	1,800	58%	71.7	1,800
	High	403	13%	34.6	1,800	75%	119.1	1,800
[3, 12]	Constant	364	3%	36.4	1,800	55%	66.2	1,800
	Medium	442	9%	33.1	1,800	72%	106.5	1,800
	High	416	4%	31.4	1,800	74%	114.1	1,800

Table 5: Results for weighted cost 5/15/10/70 and resource utilization [85%, 95%]

Duedate tightness	Process time	Number of jobs	LPST			UP		
			Optimality gap	Best solution (unit: x$1,000)	Computational time (sec.)	Optimality gap	Best solution (unit: x$1,000)	Computational time (sec.)
3	Constant	416	0%	33.2	653	0%	33.2	931
	Medium	416	10%	35.6	1,800	5%	33.7	1,800
	High	416	4%	33.4	1,800	4%	33.2	1,800
[3, 6]	Constant	416	8%	32.6	1,800	5%	31.7	1,800
	Medium	429	7%	32.3	1,800	20%	37.6	1,800
	High	416	5%	31.5	1,800	57%	69.6	1,800
[3, 9]	Constant	429	13%	34.3	1,800	61%	76.4	1,800
	Medium	403	5%	31.6	1,800	72%	105.3	1,800
	High	416	8%	32.5	1,800	70%	85.6	1,800
[3, 12]	Constant	429	3%	30.9	1,800	59%	73.7	1,800
	Medium	416	7%	32.4	1,800	39%	49.4	1,800
	High	442	5%	31.6	1,800	72%	106.9	1,800

Table 6: Results for weighted cost 5/15/10/70 and resource utilization [75%, 110%]

Duedate tightness	Process time	Number of jobs	LPST			UP		
			Optimality gap	Best solution (unit: x$1,000)	Computational time (sec.)	Optimality gap	Best solution (unit: x$1,000)	Computational time (sec.)
3	Constant	507	29%	49.8	1,800	23%	46.4	1,800
	Medium	494	31%	50.4	1,800	14%	41.3	1,800
	High	390	3%	34.0	1,800	3%	34.1	1,800
[3, 6]	Constant	456	32%	44.1	1,800	41%	50.9	1,800
	Medium	507	62%	78.4	1,800	75%	118.0	1,800
	High	416	33%	44.8	1,800	28%	41.9	1,800
[3, 9]	Constant	390	17%	36.0	1,800	61%	76.7	1,800
	Medium	455	19%	37.1	1,800	79%	145.6	1,800
	High	442	32%	44.0	1,800	74%	115.1	1,800
[3, 12]	Constant	396	36%	46.6	1,800	63%	80.4	1,800
	Medium	372	22%	38.4	1,800	73%	110.9	1,800
	High	429	29%	42.2	1,800	81%	156.4	1,800

Table 7: Results for weighted cost 5/15/10/70 and resource utilization [55%, 140%]

Duedate tightness	Process time	Number of jobs	LPST			UP		
			Optimality gap	Best solution (unit: x$1,000)	Computational time (sec.)	Optimality gap	Best solution (unit: x$1,000)	Computational time (sec.)
3	Constant	451	44%	64.2	1,800	44%	64.5	1,800
	Medium	598	72%	109.9	1,800	58%	92.8	1,800
	High	416	46%	67.0	1,800	47%	67.6	1,800
[3, 6]	Constant	528	63%	80.6	1,800	58%	70.6	1,800
	Medium	420	71%	103.9	1,800	74%	115.7	1,800
	High	552	96%	121.2	1,800	80%	152.8	1,800
[3, 9]	Constant	455	26%	40.8	1,800	63%	81.0	1,800
	Medium	506	93%	94.9	1,800	79%	142.7	1,800
	High	559	77%	132.8	1,800	83%	172.3	1,800
[3, 12]	Constant	442	63%	81.0	1,800	68%	93.6	1,800
	Medium	450	72%	105.4	1,800	78%	137.2	1,800
	High	585	95%	124.3	1,800	79%	141.2	1,800

Table 8: Results for weighted cost 5/70/10/15 and resource utilization [65%, 95%]

Duedate	Process time	Job	LPST			UP		
			Optimality gap	Best solution (unit: x$1,000)	Computational time (sec.)	Optimality gap	Best solution (unit: x$1,000)	Computational time (sec.)
3	Constant	429	0%	32.8	30	0%	32.8	36
	Medium	416	0%	34.0	70	0%	34.0	72
	High	341	0%	32.0	8	0%	32.0	10
[3, 6]	Constant	416	6%	32.5	1,800	13%	34.4	1,800
	Medium	390	3%	31.7	1,800	15%	35.1	1,800
	High	408	5%	31.5	1,800	31%	43.5	1,800
[3, 9]	Constant	429	7%	32.2	1,800	36%	46.9	1,800
	Medium	348	2%	30.9	1,800	30%	42.6	1,800
	High	377	2%	30.9	1,800	33%	44.5	1,800
[3, 12]	Constant	377	7%	32.3	1,800	33%	45.1	1,800
	Medium	364	2%	30.8	1,800	71%	103.3	1,800
	High	377	5%	31.7	1,800	64%	83.8	1,800

Table 9: Results for weighted cost 5/70/10/15 and resource utilization [75%, 95%]

Duedate tightness	Process time	Number of jobs	LPST			UP		
			Optimality gap	Best solution (unit: x$1,000)	Computational time (sec.)	Optimality gap	Best solution (unit: x$1,000)	Computational time (sec.)
3	Constant	390	0%	32.0	17	0%	32.0	24
	Medium	416	0%	32.4	27	0%	32.4	26
	High	429	0%	33.1	41	0%	33.1	44
[3, 6]	Constant	416	7%	32.2	1,800	17%	35.9	1,800
	Medium	384	3%	31.2	1,800	12%	34.8	1,800
	High	372	0%	31.1	1,705	6%	32.1	1,800
[3, 9]	Constant	396	7%	32.2	1,800	32%	43.2	1,800
	Medium	429	7%	32.1	1,800	67%	90.2	1,800
	High	390	5%	37.9	1,800	43%	52.4	1,800
[3, 12]	Constant	442	6%	32.0	1,800	37%	47.6	1,800
	Medium	403	6%	31.8	1,800	77%	133.2	1,800
	High	377	0%	30.7	649	37%	47.5	1,800

Table 10: Results for weighted cost 5/70/10/15 and resource utilization [85%, 95%]

Duedate tightness	Process time	Number of jobs	LPST			UP		
			Optimality gap	Best solution (unit: x$1,000)	Computational time (sec.)	Optimality gap	Best solution (unit: x$1,000)	Computational time (sec.)
3	Constant	429	0%	32.4	9	0%	32.4	13
	Medium	429	0%	32.3	10	0%	32.3	18
	High	390	0%	31.4	5	0%	31.4	6
[3, 6]	Constant	416	4%	31.4	1,800	6%	31.9	1,800
	Medium	390	3%	31.2	1,800	6%	32.0	1,800
	High	403	3%	31.1	1,800	24%	39.5	1,800
[3, 9]	Constant	416	3%	31.0	1,800	31%	43.2	1,800
	Medium	408	6%	32.0	1,800	44%	53.9	1,800
	High	416	5%	31.4	1,800	74%	114.5	1,800
[3, 12]	Constant	396	5%	31.7	1,800	36%	46.7	1,800
	Medium	396	4%	31.3	1,800	76%	123.0	1,800
	High	416	4%	31.4	1,800	70%	100.0	1,800

Table 11: Results for weighted cost 5/70/10/15 and resource utilization [75%, 110%]

Duedate tightness	Process time	Number of jobs	LPST			UP		
			Optimality gap	Best solution (unit: x$1,000)	Computational time (sec.)	Optimality gap	Best solution (unit: x$1,000)	Computational time (sec.)
3	Constant	456	0%	35.6	478	0%	35.6	451
	Medium	377	0%	32.4	27	0%	32.4	24
	High	432	0%	35.6	498	0%	35.7	499
[3, 6]	Constant	507	14%	34.9	1,800	22%	38.4	1,800
	Medium	429	21%	38.4	1,800	68%	93.8	1,800
	High	420	44%	53.4	1,800	25%	40.2	1,800
[3, 9]	Constant	442	13%	34.3	1,800	46%	55.5	1,800
	Medium	455	15%	35.4	1,800	46%	55.8	1,800
	High	444	44%	53.2	1,800	75%	120.4	1,800
[3, 12]	Constant	442	8%	32.5	1,800	45%	54.3	1,800
	Medium	455	6%	32.0	1,800	72%	106.9	1,800
	High	494	55%	66.9	1,800	79%	142.2	1,800

Table 12: Results for weighted cost 5/70/10/15 and resource utilization [55%, 140%]

Duedate tightness	Process time	Number of jobs	LPST			UP		
			Optimality gap	Best solution (unit: x$1,000)	Computational time (sec.)	Optimality gap	Best solution (unit: x$1,000)	Computational time (sec.)
3	Constant	588	17%	39.5	1,800	16%	39.0	1,800
	Medium	429	3%	37.6	1,800	3%	37.6	1,800
	High	624	60%	115.5	1,800	63%	124.2	1,800
[3, 6]	Constant	442	22%	39.3	1,800	24%	40.3	1,800
	Medium	450	38%	48.9	1,800	73%	109.4	1,800
	High	598	78%	134.6	1,800	73%	112.6	1,800
[3, 9]	Constant	650	44%	53.3	1,800	69%	97.4	1,800
	Medium	456	44%	53.6	1,800	72%	107.0	1,800
	High	495	85%	204.1	1,800	76%	127.6	1,800
[3, 12]	Constant	598	38%	48.1	1,800	72%	107.1	1,800
	Medium	528	45%	54.9	1,800	75%	119.0	1,800
	High	480	87%	238.1	1,800	82%	164.1	1,800

Table 13: Results for weighted cost 25/25/25/25 and resource utilization [65%, 95%]

Duedate tightness	Process time	Number of jobs	LPST			UP		
			Optimality gap	Best solution (unit: x$1,000)	Computational time (sec.)	Optimality gap	Best solution (unit: x$1,000)	Computational time (sec.)
3	Constant	320	0%	79.4	22	0%	79.5	27
	Medium	363	0%	80.4	123	0%	80.4	124
	High	390	0%	78.5	88	0%	78.4	53
[3, 6]	Constant	416	12%	84.9	1,800	16%	89.0	1,800
	Medium	360	4%	78.4	1,800	4%	78.6	1,800
	High	364	2%	77.7	1,800	3%	78.0	1,800
[3, 9]	Constant	336	2%	77.2	1,800	3%	77.6	1,800
	Medium	403	6%	79.5	1,800	31%	109.3	1,800
	High	403	3%	78.1	1,800	28%	104.1	1,800
[3, 12]	Constant	416	8%	81.5	1,800	32%	110.5	1,800
	Medium	325	4%	78.5	1,800	29%	105.4	1,800
	High	364	7%	80.3	1,800	41%	126.6	1,800

Table 14: Results for weighted cost 25/25/25/25 and resource utilization [75%, 95%]

Duedate tightness	Process time	Number of jobs	LPST			UP		
			Optimality gap	Best solution (unit: x$1,000)	Computational time (sec.)	Optimality gap	Best solution (unit: x$1,000)	Computational time (sec.)
3	Constant	416	0%	78.6	45	0%	78.5	55
	Medium	442	0%	81.9	1,185	0%	81.9	1,106
	High	416	0%	80.6	333	0%	80.6	336
[3, 6]	Constant	351	4%	78.5	1,800	7%	80.9	1,800
	Medium	364	2%	77.0	1,800	6%	80.1	1,800
	High	429	5%	78.6	1,800	29%	104.9	1,800
[3, 9]	Constant	442	7%	80.5	1,800	32%	111.0	1,800
	Medium	363	5%	78.7	1,800	28%	104.3	1,800
	High	360	0%	77.0	934	6%	79.7	1,800
[3, 12]	Constant	377	2%	77.1	1,800	6%	79.4	1,800
	Medium	429	7%	80.4	1,800	45%	136.0	1,800
	High	364	0%	76.8	821	2%	77.0	1,800

Table 15: Results for weighted cost 25/25/25/25 and resource utilization [85%, 95%]

Duedate tightness	Process time	Number of jobs	LPST			UP		
			Optimality gap	Best solution (unit: x$1,000)	Computational time (sec.)	Optimality gap	Best solution (unit: x$1,000)	Computational time (sec.)
3	Constant	429	0%	79.8	60	0%	79.8	55
	Medium	390	0%	80.6	9	0%	77.0	11
	High	429	0%	78.9	45	0%	78.8	44
[3, 6]	Constant	416	2%	77.0	1,800	18%	91.9	1,800
	Medium	429	6%	79.7	1,800	10%	83.6	1,800
	High	429	4%	78.3	1,800	16%	88.9	1,800
[3, 9]	Constant	403	4%	78.3	1,800	27%	167.7	1,800
	Medium	429	4%	78.2	1,800	40%	125.3	1,800
	High	416	11%	83.8	1,800	37%	119.2	1,800
[3, 12]	Constant	442	5%	78.7	1,800	34%	113.0	1,800
	Medium	403	4%	80.0	1,800	57%	172.6	1,800
	High	416	4%	83.2	1,800	35%	115.6	1,800

Table 16: Results for weighted cost 25/25/25/25 and resource utilization [75%, 110%]

Duedate tightness	Process time	Number of jobs	LPST			UP		
			Optimality gap	Best solution (unit: x$1,000)	Computational time (sec.)	Optimality gap	Best solution (unit: x$1,000)	Computational time (sec.)
3	Constant	429	5%	84.9	1,800	5%	85.0	1,800
	Medium	507	6%	85.4	1,800	6%	85.4	1,800
	High	468	7%	85.8	1,800	7%	85.8	1,800
[3, 6]	Constant	418	9%	82.6	1,800	23%	97.5	1,800
	Medium	420	18%	91.6	1,800	24%	98.9	1,800
	High	456	15%	87.8	1,800	41%	127.9	1,800
[3, 9]	Constant	390	5%	79.2	1,800	33%	111.5	1,800
	Medium	468	9%	83.3	1,800	63%	201.3	1,800
	High	429	30%	107.9	1,800	34%	113.5	1,800
[3, 12]	Constant	468	11%	84.2	1,800	43%	130.7	1,800
	Medium	396	26%	101.2	1,800	56%	171.8	1,800
	High	372	16%	89.5	1,800	34%	122.2	1,800

Table 17: Results for weighted cost 25/25/25/25 and resource utilization [55%, 140%]

Duedate tightness	Process time	Number of jobs	LPST			UP		
			Optimality gap	Best solution (unit: x$1,000)	Computational time (sec.)	Optimality gap	Best solution (unit: x$1,000)	Computational time (sec.)
3	Constant	507	11%	91.0	1,800	11%	91.0	1,800
	Medium	490	25%	103.6	1,800	22%	100.9	1,800
	High	442	5%	85.4	1,800	6%	86.3	1,800
[3, 6]	Constant	528	29%	106.1	1,800	31%	109.4	1,800
	Medium	550	25%	99.9	1,800	64%	207.7	1,800
	High	504	37%	118.4	1,800	53%	159.3	1,800
[3, 9]	Constant	598	27%	103.2	1,800	41%	126.6	1,800
	Medium	564	93%	221.4	1,800	69%	240.4	1,800
	High	533	38%	120.2	1,800	50%	151.5	1,800
[3, 12]	Constant	564	32%	110.9	1,800	67%	226.4	1,800
	Medium	468	48%	145.5	1,800	66%	202.8	1,800
	High	650	50%	150.8	1,800	68%	237.8	1,800

(1) Total cost (objective function value)

In order to visualize the effect of weighted cost, the graphs between total cost and resource utilization is shown in Figures 5-7. The graphs present total cost impacts of three weighted costs at different ranges of due date tightness. Two initialization methods, LPST and UP, are compared in each figure. The results show that the LPST method obtained a lower total cost than the UP method. This can be explained that LPST assign the initial solution that is closer to a local minimum or an optimal solution than UP. A narrow search influences a better result and a faster computational time in individual node until it meets the optimal solution. At different resource utilizations, LPST provides more consistent outputs than UP. Resource utilization is a major factor which reflects the increasing of total cost, especially in the UP method. According to the zero initial solution from UP, the optimizer would need more computational time to reach the optimal solution. Moreover, if the problems involve high congestion in the system, it would be more difficult to obtain a solution within limited time.

For a clearer view, the scenario with weighted cost 5/70/15/10 is selected to observe more result details between two initial solution settings. In Figure 6, the graphs show that different ranges of due date do not provide different total costs in LPST when resource utilization is less than 100%. Meanwhile, total cost in UP tends to increase when due dates are spread out. This can be explained with the search space and variability concepts. A large search space with high variability of data increases problem complexity while lower variability of parameters can reduce time to reach an optimal solution and gain more optimizer power.

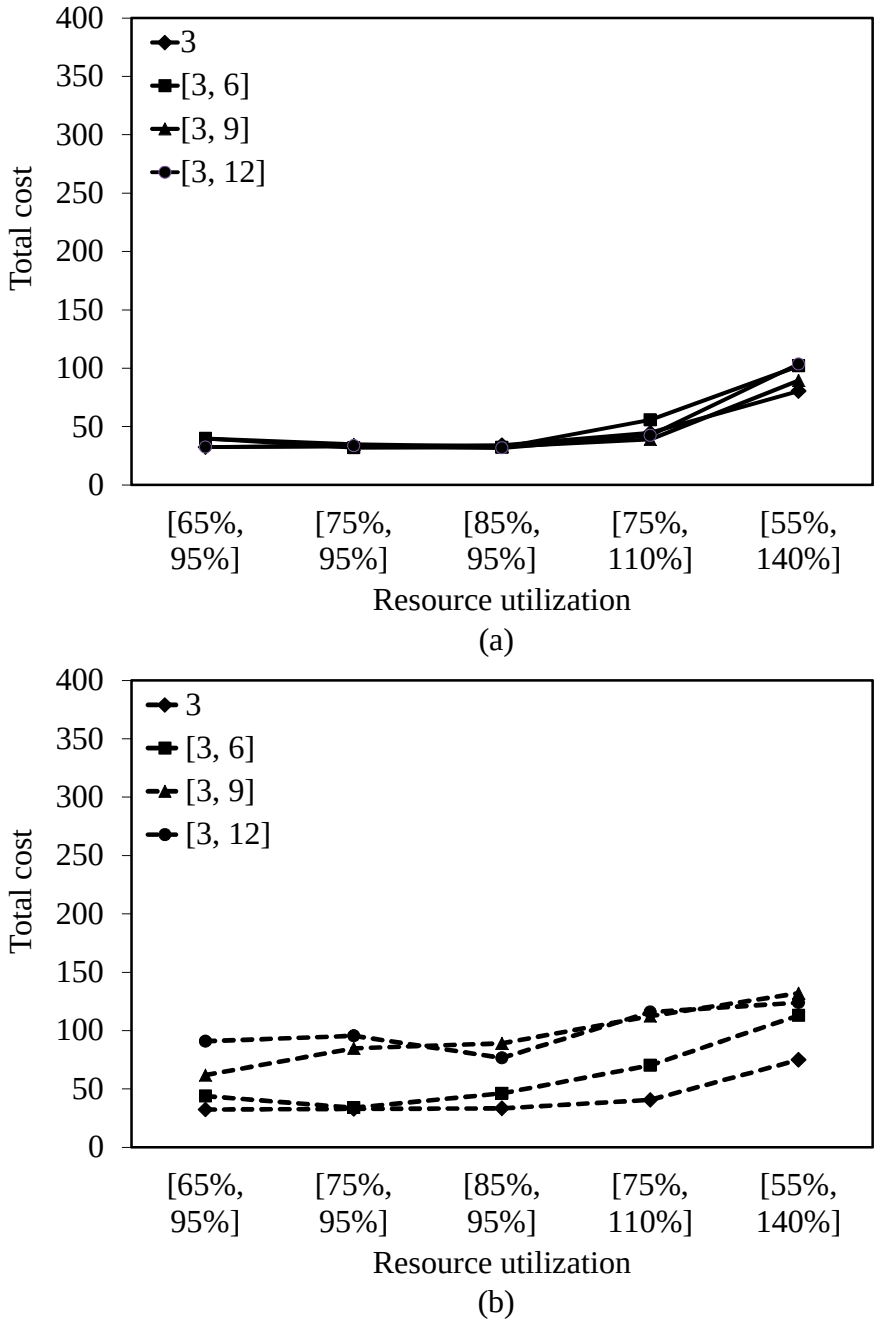


Figure 5: Total cost impact at 5/15/10/70 for; (a) LPST and (b) UP

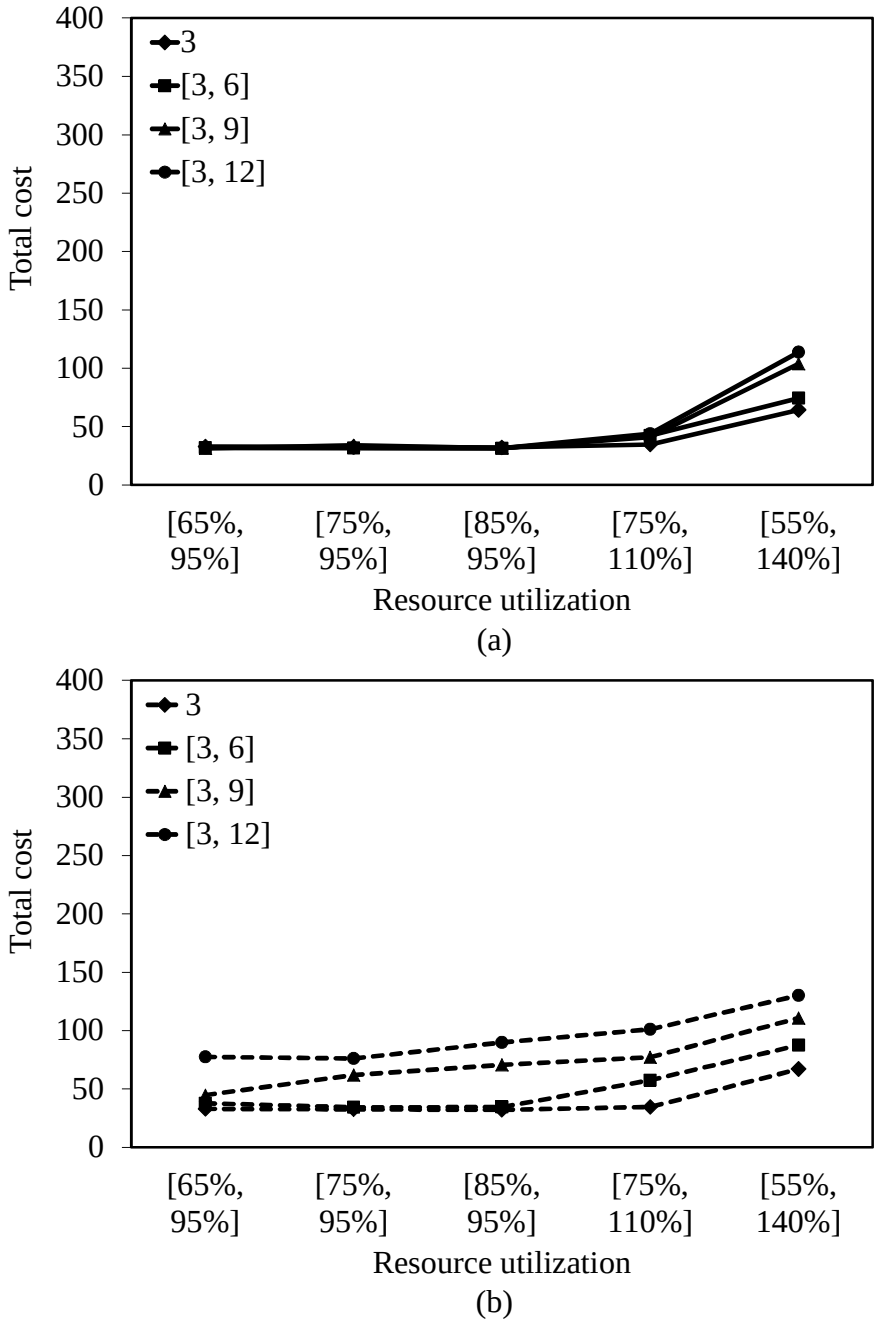


Figure 6: Total cost impact at 5/70/10/15 for; (a) LPST and (b) UP

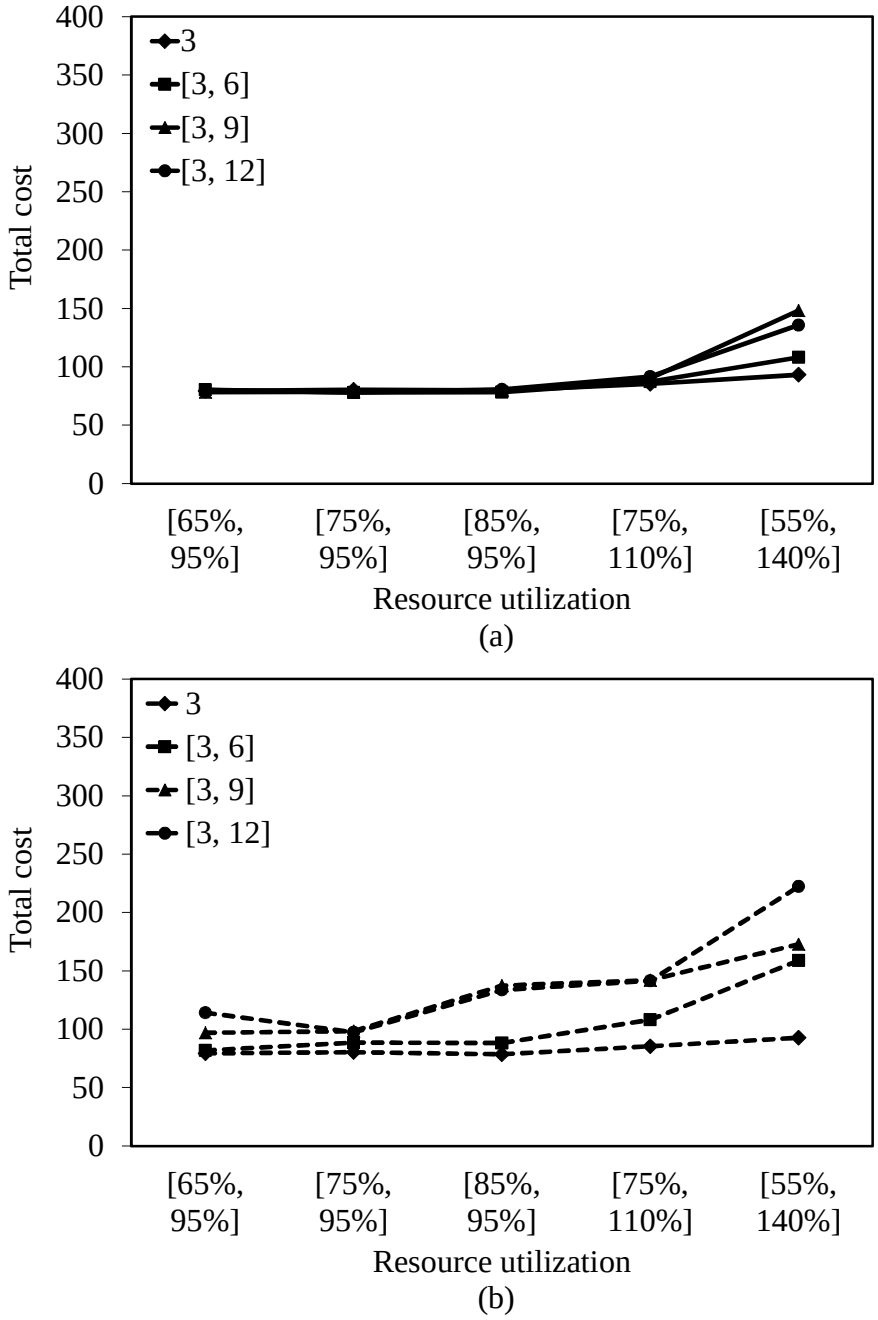


Figure 7: Total cost impact at 25/25/25/25 for; (a) LPST and (b) UP

In addition, it can be seen that an interaction of variability parameters would turn down the quality of the solution. As a result, total cost of both setting methods

would be increased when due date tightness range is wider and resource utilization is higher, for instance at due date tightness [3, 9] and [3, 12], and average resource utilization above 90%, [75%, 110%] and [55%, 140%] . This is because high variability of due date tightness and resource utilization generate less available spots in the production plan. To meet customer requirements, using an additional capacity, producing jobs earlier or delaying jobs, are considered as options. Then, these influence an increased total cost.

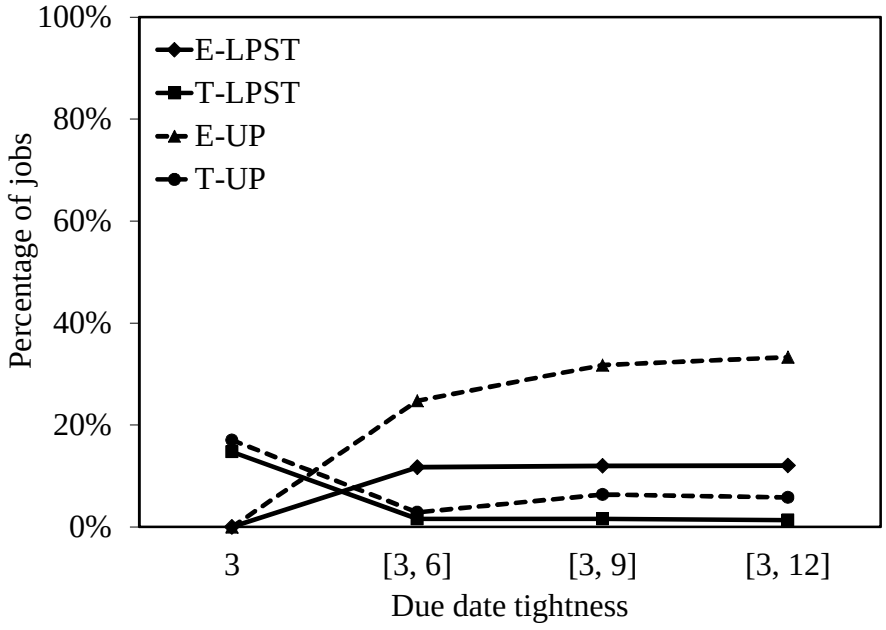
(2) Earliness and tardiness

Figure 8 presents the percentage of early jobs (E) and tardy jobs (T) at different weighted costs. The graphs display the comparison of the early and tardy job rates between two initialization methods. It can be seen that LPST provides a lower number of early and tardy jobs than UP in all weighted cost scenarios. The same explanation as in the total cost impact is also used to describe this result that an effective initial solution influences a better solution and a faster execution time.

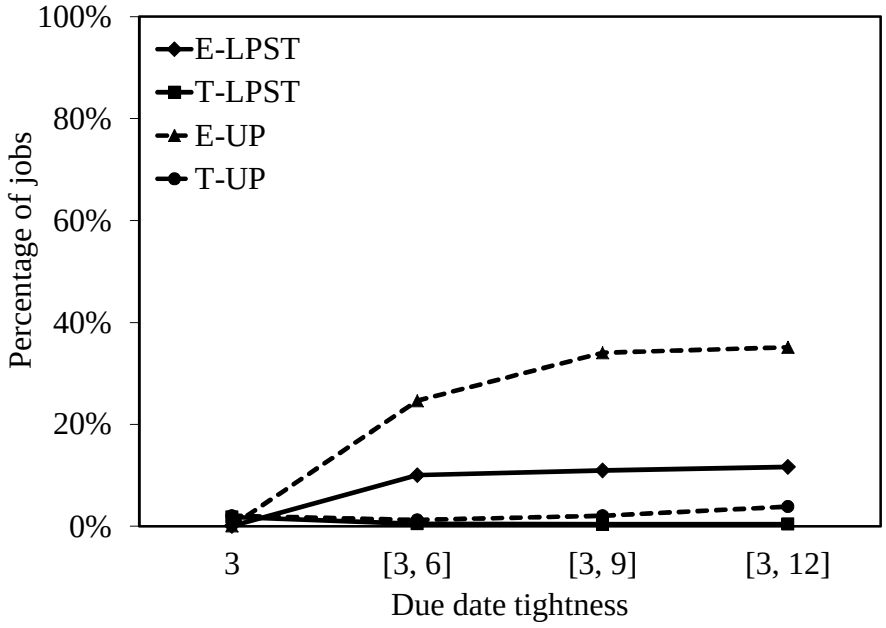
Figure 8(a) shows the different trend lines of the percentage of early and tardy jobs at weighted cost 5/15/10/70. It can be seen that the number of early jobs is increased when due date tightness has a larger range. Meanwhile, the number of tardy jobs is decreased. This can be explained that a majority cost reflects a reduced cost in its area. At weighted cost 5/15/10/70, a subcontracting cost is the highest penalty cost so that the optimizer attempts to generate the resource plan that meets all customer requirements by using some alternative methods, either pulling the demands to produce ahead or postponing the shipment, rather than applying additional capacity from subcontractor or extra resources. If the earliness cost is cheaper than the tardiness cost,

this will impact the increasing percentage of early jobs instead. For the spiked tardy job rate at due date tightness 3, it is because each job requires at least 3 periods to process and finish the job. To avoid a highly charged cost from subcontractor, lateness is only a possible option to complete a job when the capacity is a constraint. The tardiness rate then drops when due date tightness is varied.

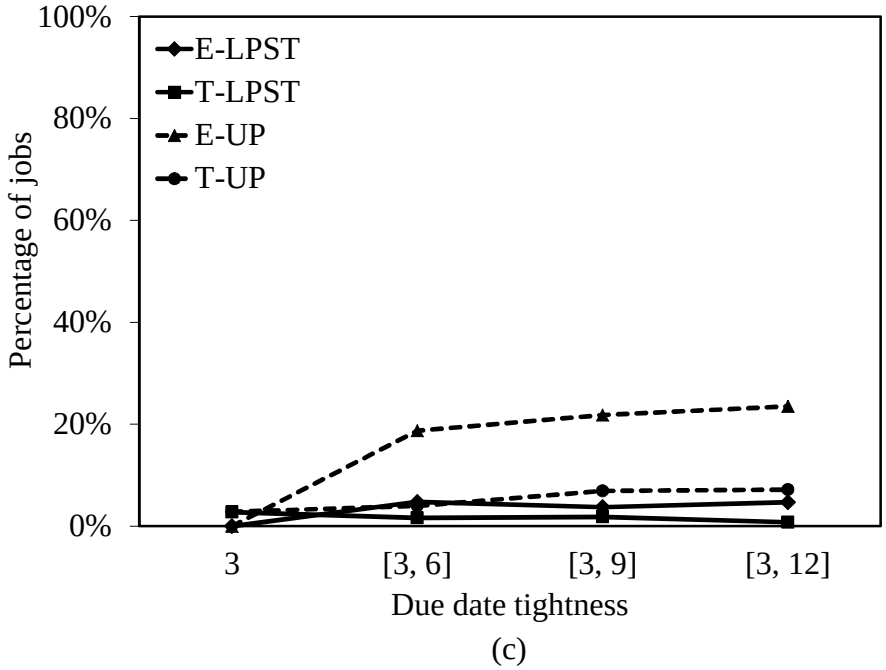
The other cost scenarios, Figure 8(b) and 8(c), provide the same trend for both the earliness and tardiness rates. They illustrate that the percentage of early jobs is greater than the percentage of tardy jobs. The explanation can be drawn by the majority cost concept as well. For example, at weighted cost 5/70/10/15, with the highest tardiness cost, finishing early or using subcontracting capacity are preferred options to minimize total cost in the system.



(a)



(b)

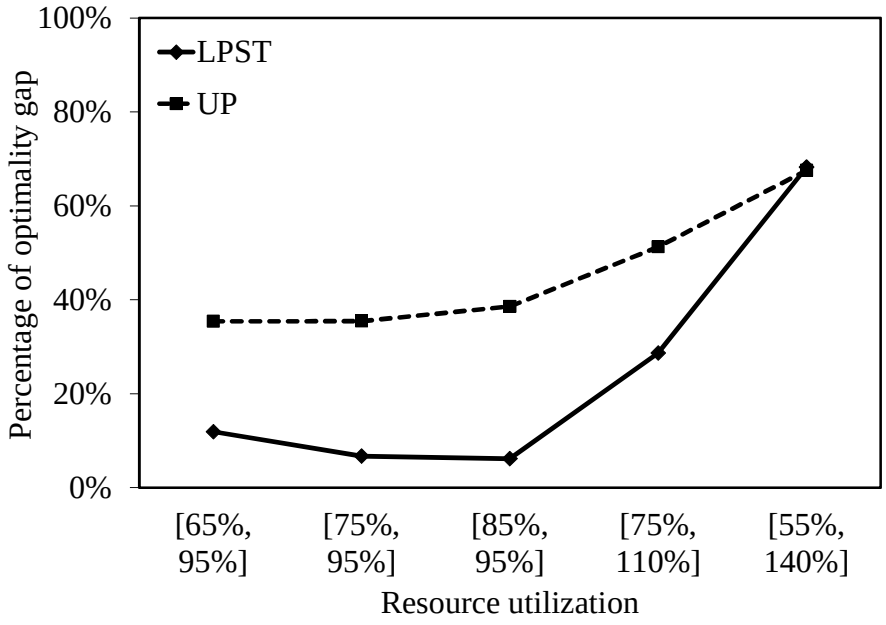


(c)

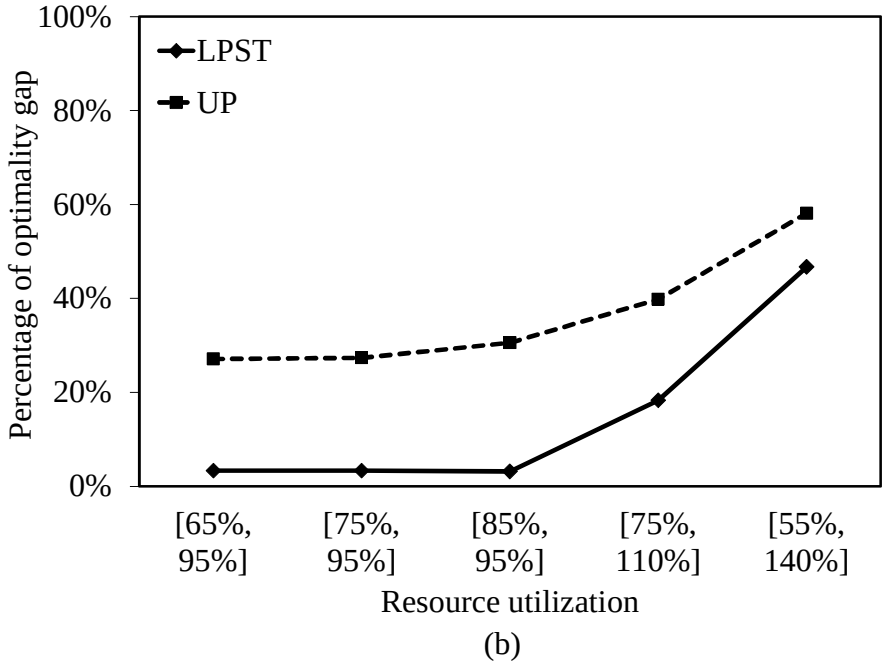
Figure 8: Percentage of early and tardy jobs for; (a) 5/15/10/70, (b) 5/70/10/15, and (c) 25/25/25/25

(3) Optimality gap

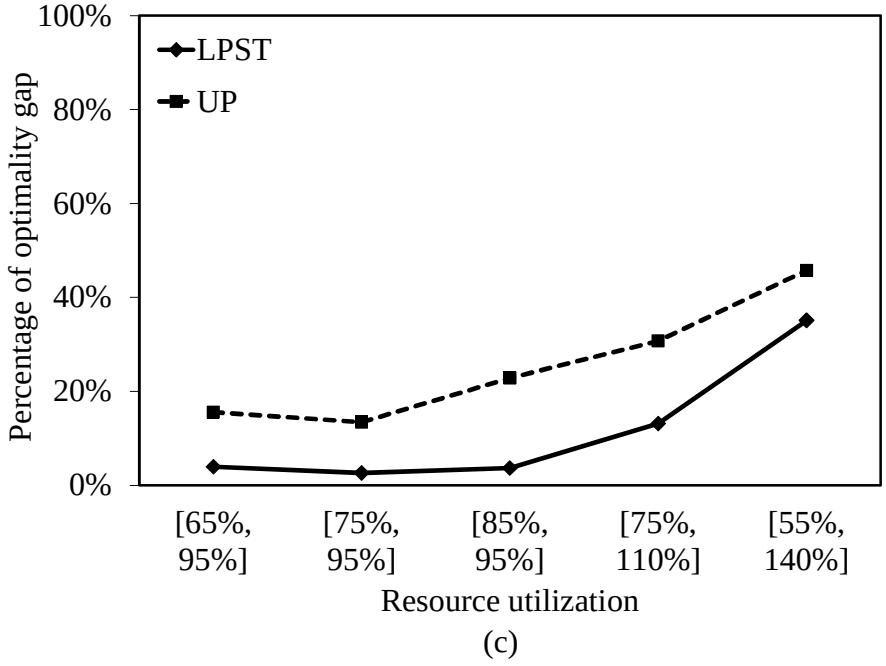
Figure 9 presents the optimality gap after computational time 1,800 seconds at different resource utilization and weighted costs. All graphs show in the same trend lines that the optimality gaps are increased when the problem utilization is higher and range of due date is wider. It makes more sense that when the problem is more complicated, the optimizer may hardly find the optimal solution within the limited time. The average utilization range 80%-90%, [65%, 95%], [75%, 95%], [85%, 95%], provides a lower optimality gap, while the average utilization above 90%, [75%, 110%] and [55%, 140%], is turned to be a harder problem that causes the optimality gap to increase.



(a)



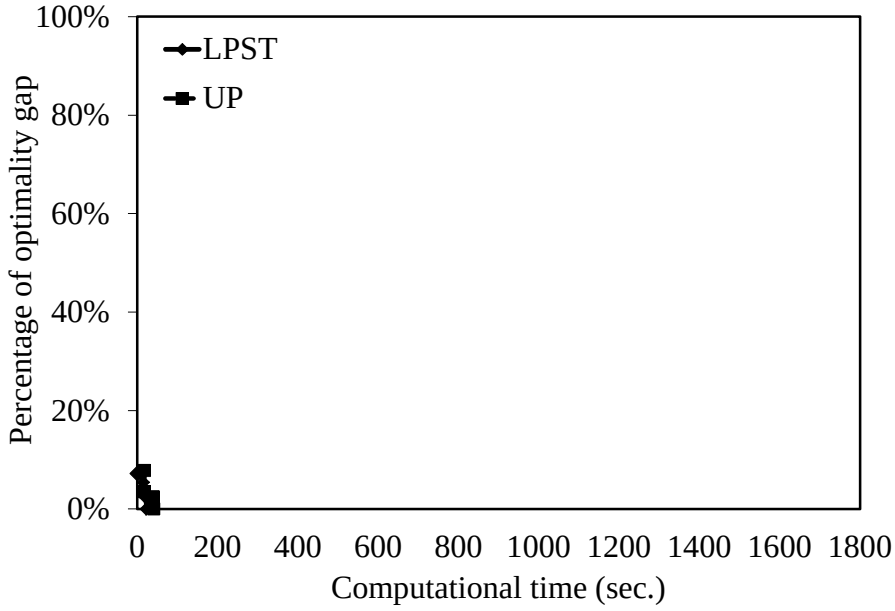
(b)



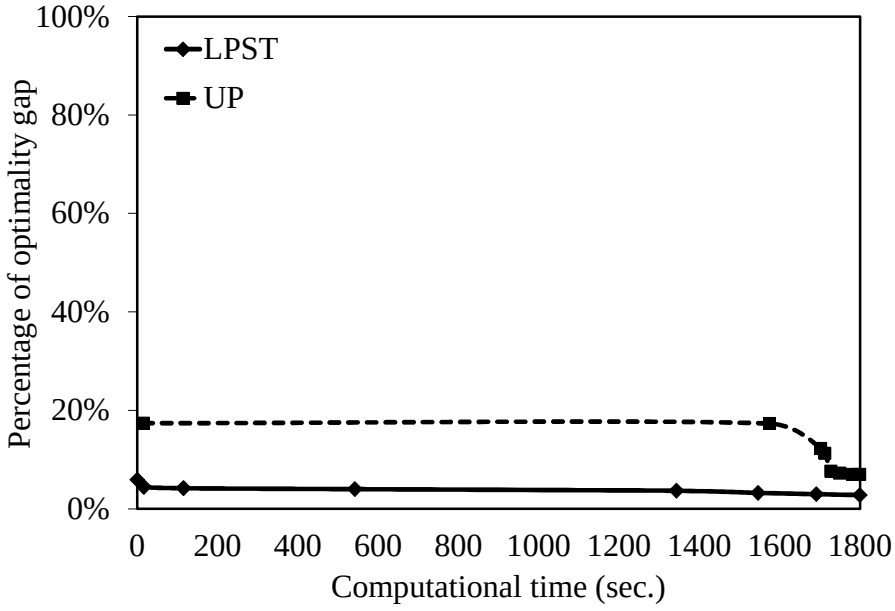
(c)

Figure 9: Optimality gap at different objective costs for; (a) 5/15/10/70, (b) 5/70/10/15, and (c) 25/25/25/25

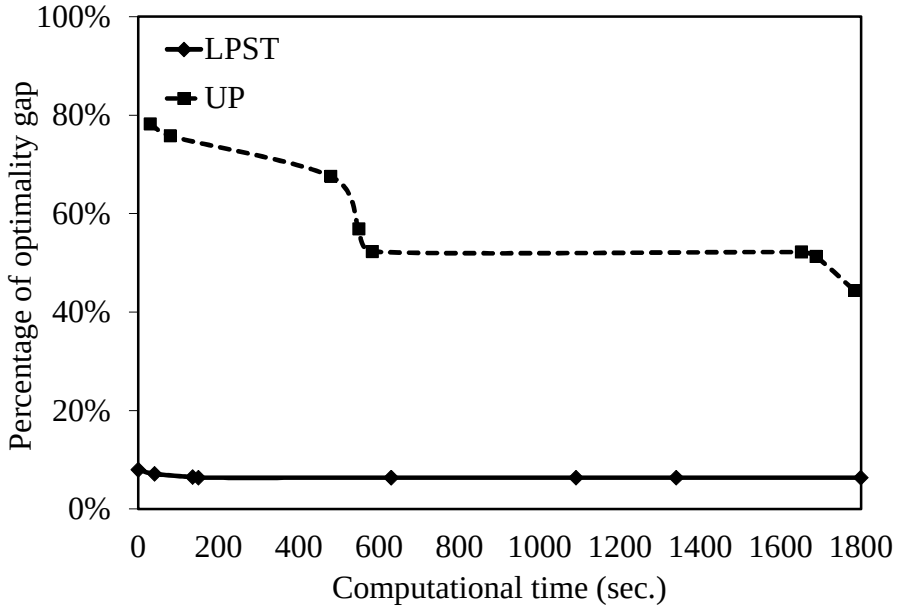
For further investigation on the planning performance, the optimality gap reduction is collected from the start time until either the time of finding the optimal solution or reaching computational time at 1,800 seconds. The cases at medium process time, resource utilization [85%, 95%], and weighted cost 5/70/10/15 is chosen to discuss this study. Figure 10 shows the relationships of computational time and solution optimality gap at different due date tightness. The effect obviously shows that LPST is able to reach the optimal or good solution faster than UP. Since LPST provides a closer feasible solution at the beginning of the period, the optimizer requires shorter computational time to solve an optimal solution. In addition, it can be observed that when the problem has low variability such as constant due date, both initial setting methods are able to find the optimal solution with short computational time as shown in Figure 10(a). Meanwhile, in Figure 10(b)-(d), wider ranges of due date make the problem more difficult and then they influence the increasing optimality gap. The difference of optimality gap between the LPST and UP methods is more increased when the range of due date tightness is larger. This is because when the job's due dates are scattered along the planning horizon, they create some fractional capacities in time buckets. Moreover, because the planning is considered operation dependent sequencing, it is more difficult to load jobs in the planning to fit with small available slot of capacities and obtain an optimal solution. Hence, it reflects a higher optimality gap.


100%
80%
60%
40%
20%
0%
LPST
UP
Percentage of optimality gap
0
200
400
600
800
1000
1200
1400
1600
1800
Computational time (sec.)

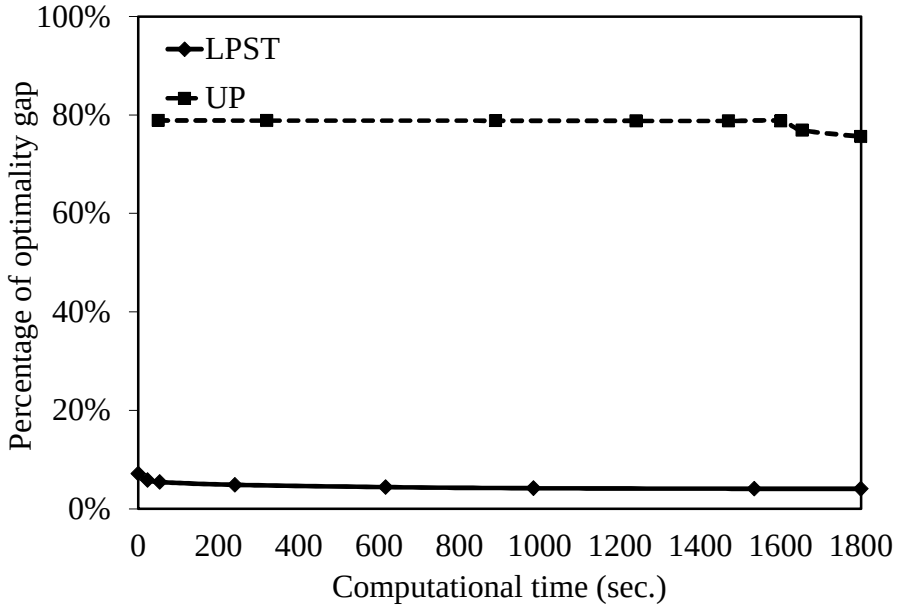
(a)


100%
80%
60%
40%
20%
0%
LPST
UP
Percentage of optimality gap
0
200
400
600
800
1000
1200
1400
1600
1800
Computational time (sec.)

(b)



(c)



(d)

Figure 10: Optimality gap of resource utilization [85%, 95%], weighted cost 5/70/10/15, and medium process time at due date tightness; (a) 3, (b) (3, 6), (c) (3, 9), and (d) (3, 12)

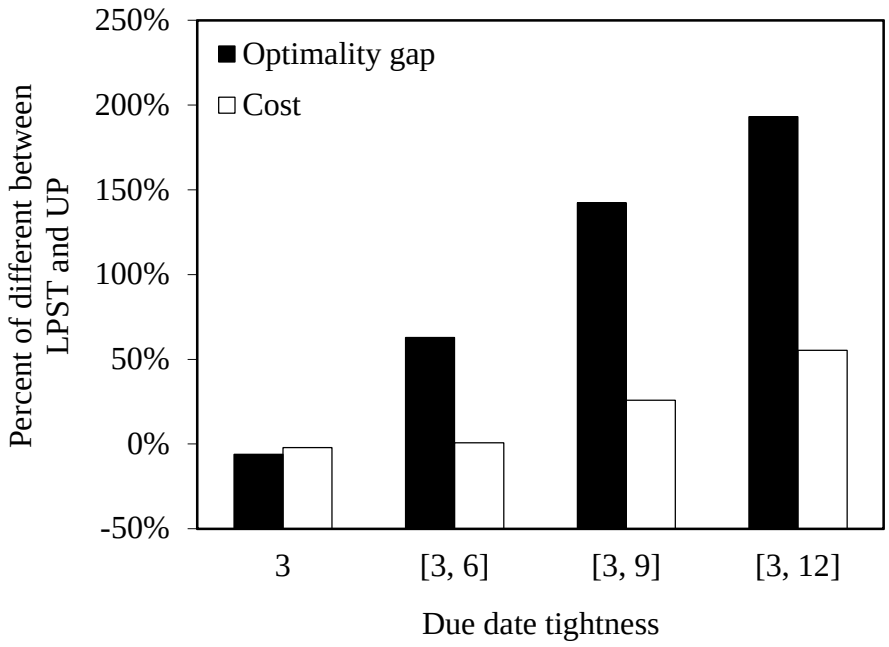


Figure 11: Percentage of result difference between LPST and UP

From all experimental results, the conclusions can be drawn as follows. First, LPST is the most effective method. It can provide an efficient solution faster and better than UP. Figure 11 shows the summarized comparison results between LPST and UP of two indicators, optimality gap and total cost. This plot shows an incremental percentage of result difference of LPST over UP. It means that LPST provides increasingly better solutions in both indicators when the system has more variability. This supports the conclusion that an efficient initial solution significantly improves the optimizer's performance. Second, the interactions of variability influence the solution quality dropping. Even though an initial solution is provided, because of problem difficulty, the optimizer cannot find an optimal solution within limited time. This case occurs when the system's parameters have a wider range of due date and higher resource utilization. When a problem is difficult to solve and some specified initial solutions do not lead directly to an optimal solution, CPLEX will apply a quick heuristic to repair the

solution. It therefore requires more computational time to search and obtain the solution. Last, UP also performs effectively in certain conditions. It is able to provide a good solution when the problem is not complicated such as at constant due date 3. Although it is not obvious that the UP method provided the better performance, the proper parameter conditions also allow UP to reach the optimal solution with short computational time.

3.6.2 Impact of variability

This section describes the behavior of system performance when dealing with variability. There are four factors affecting the planning results, which are weighted cost, due date tightness, resource utilization, and process time. The results from weighted cost 5/70/10/15 and LPST initialized method is selected to describe the solution impact, since these conditions are similar to a real manufacturing environment when tardy jobs need to be reduced as many as possible due to customer satisfaction.

(1) Weighted cost

A weighted cost is a main factor that substantially impacts the planning results. The highest penalty will determine the direction of the loading plan. For instance, at 5/70/10/15 (EC/TC/LC/SC), a high tardiness cost leads planning to complete jobs on time or earlier to avoid a penalty cost from lateness. An additional capacity might be used to relieve congestion in the system as well as pulling the jobs to produce early. In this case, the majority cost comes from either earliness cost or subcontracting cost. The main idea of the weighted cost variability impact is to avoid a major charged cost and to create the resource plan with minimum total cost.

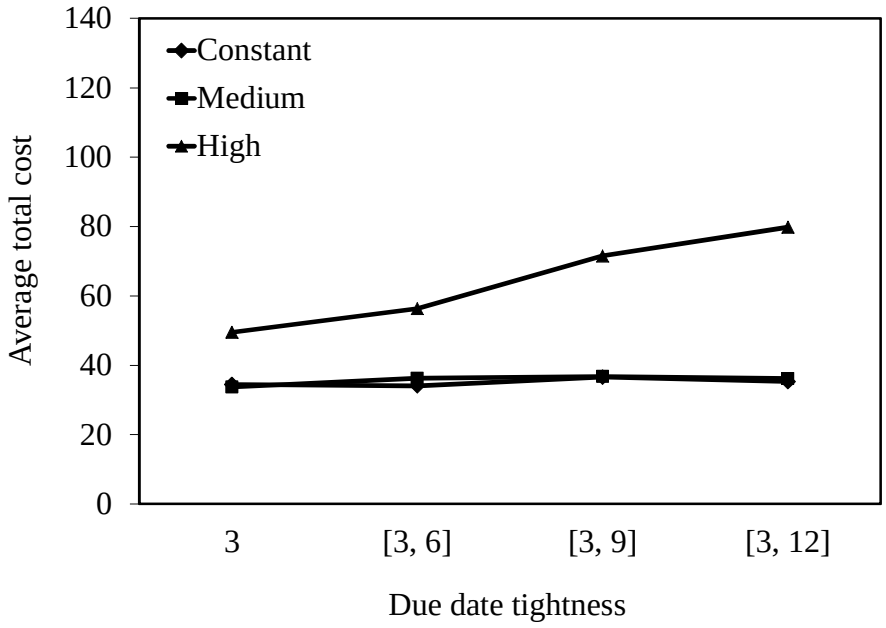


Figure 12: Effects of due date tightness at cost ratio 5/70/10/15

(2) Due date tightness

Figure 12 presents the relation of due date tightness and average total cost at different process times. In constant and medium process time variabilities, total costs among due date tightness level are not significantly different. But it clearly shows a huge difference when process time and due date tightness have greater data variance. The interaction of due date tightness and process time increases the problem difficulty. The optimizer cannot provide an efficient solution with short computational time. A fractional capacity might be a major factor. When process time is constant, jobs are able to be loaded in time buckets without or less waiting time. As variability is increased, some longer process times would create holes or fractional capacities in time buckets. To avoid a tardiness penalty, producing job in the early period or using subcontracting capacity are considered as alternatives. Although the due date tightness does not show much difference of impact among their levels, it can be seen that some variability factor

allows relaxation in the system and provide a better solution. For instance, at due date tightness [3, 6], it is the proper range of due date variability in which jobs can be loaded on the existing resources without congestion.

(3) Resource utilization

Figure 13 shows the relation of resource utilization and average total cost at different weighted costs. From the graph, total cost is not apparently different when resource utilization is at [65%, 95%] to [85%, 95%] in all three weighted cost cases. This is because subcontracting capacity can cover the excess requirements and smooth production system. While at resource utilization [75%, 110%] and [55%, 140%], some overload capacities occur in certain periods. Subcontracting capacity is required more to meet the customer demands. However, some tardiness jobs cannot be avoided, because resources are fully utilized. This reflects tardiness cost to be increased. In addition, when the resource planning problem is more complicated, the optimizer hardly reaches an optimal solution or even a good feasible solution within limited time. This causes the ineffective planning's results as well.

(4) Process time

In Figure 12, the effect of process time variability does not show a significant different result when process time is 5 and [3, 7] but the variability obviously impact the objective value, or total cost, when process time is [1, 9]. This is because the more difference of process times, the more unbalance of resource requirements. An option of using an alternative capacity, producing job early, or delaying a shipment, is considered to smooth the production system so that the total cost is increased.

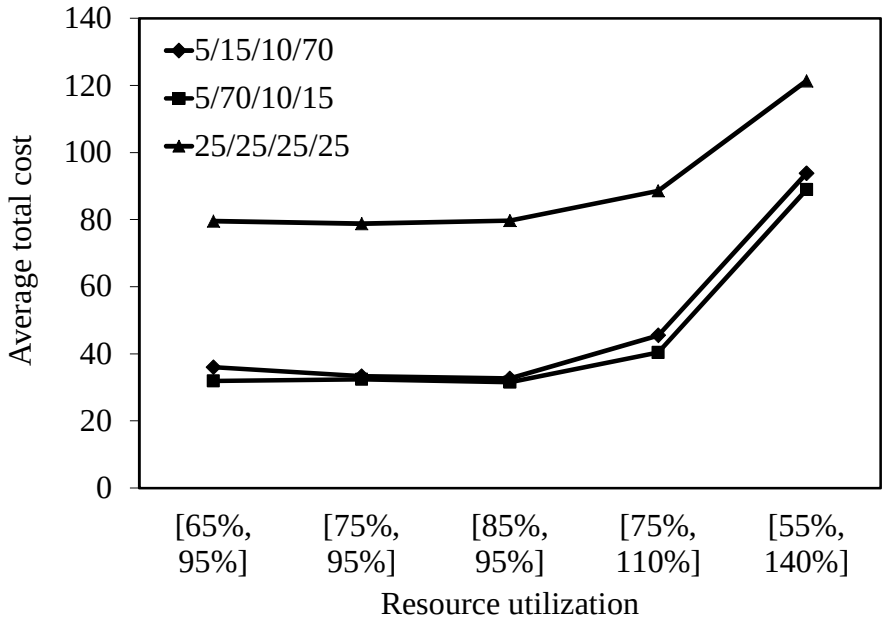


Figure 13: Effects of resource utilization at different weighted costs

3.7 Summary

This chapter proposes a new model for resource planning problems in MTO environments. The planning model is formulated as a binary integer linear programming with multiple objective functions in which the goal of the model is to minimize weighted costs from earliness, tardiness, lead time, subcontracting capacity, extra resources capacity, and unplanned jobs. The output of the planning model is an operation start time to process each job. In a production system, variability always gets involved and it is the main factor to drop the system performance and efficiency of resource planning. The research is divided in the study into two parts which are investigating solution initialization to improve resource planning and observing the impact of resource planning under variability.

In the first part, two solution initialization approaches, including the latest possible start time method (LPST) and the unplanned job method (UP), are generated to improve the optimizer efficiency. From the results, the conclusion is that LPST provides the optimal solutions faster and better than UP. LPST is an effective method that requires shorter computational time to search and obtain a feasible solution. This is because LPST can create an initial solution that is close to a local minimum or an ideal planning. A narrows search leads the optimizer to find the feasible solution directly. However, the initial solution method also has a limitation. It cannot help the optimizer to determine the solution easier if the problem is more complicated with high variability of various parameters. Even though LPST performs very well in most cases, UP also provides a good solution in some problem conditions, for instance at due date tightness 3.

In the second part, the results of resource planning under different variabilities, such as weighted cost, due date tightness, resource utilization, and process time, are examined. From this experiment, some behaviors of the system can be concluded as follows. First, variability creates either congestion or fractional resource availability, which influences ineffective planning and drops performance of optimizer. The interaction of variability affect the efficiency of production planning more when variability have larger scale, for instance the interaction of wider range of due date tightness and higher resource utilization. Second, some variability allows relaxation in the system. For example, the problem with due date tightness [3, 6] can reduce the congestion of the system and provide more sufficient resources for other jobs. Last, subcontracting is an essential flexibility to absorb variability. It can relieve the

congestion issues in a production system and also improve resource planning to be more efficient.

A resource planning model with initial solution successfully improves the quality of planning solutions. The effective initial solution based and ideal planning (LPST) is able to enhance the planning performance, optimality gap and total cost, up to 59% averagely from another method (LP). In practice, this model is also useful for decision support such as forecasting capability of a system or predicting the impact when short-term capacity is inadequate and congestion cannot be reduced with subcontracting. However, an optimization approach has its drawbacks to solve problems, when the problems become larger or more complicated. It results in a long computational time. Applying a heuristic approach is an interesting method to enhance the solving performance. The next chapter will therefore present the study of a heuristic method for solving resource planning problems.

CHAPTER 4

A tabu search approach for resource planning

4.1 Introduction

In the previous chapter, the optimization method is successfully applied for solving the small planning instances. However, it could not solve the large scale instances in reasonable time. There are many factors that limit the planning execution and its performance, such as problem size and data variability. Approximation algorithms or heuristic approaches are quite a good alternative method for solving planning problems since they are able to provide a near optimal solution with reasonable computational time.

Planning performance directly reflects manufacturing efficiency, which mainly respects to maximizing profitability. Several improvement concepts have been proposed to increase a quality of planning. The just-in-time (JIT) philosophy is recognized as an efficient productivity strategy. It has been described as an approach with the objective of producing the right product at the right time. Adapting JIT into planning, customer demands attempt to be processed and finished exactly on their due date to reduce WIP, inventory, and production costs (Baker and Scudder 1990). The planning with the JIT concept seems to be an ideal planning in which the system would not have both earliness and tardiness. From this perspective, this study is motivated to create a resource plan based on the JIT concept. A particular resource planning problem under MTO environments, which is the same as the previous chapter, is presented. The system

represents a job shop production with distinct job features. Each job has different machine routings, process lengths and due dates. The resource planning aims to determine a feasible resource assignment in which the objective is to improve total weighted cost of earliness, tardiness, and lead time.

To deal with a complicated planning problem, tabu search is applied to solve the problem since a feasible solution can be obtained with a limited time (Glover 1986). In order to create an efficient resource plan, the JIT concept is implemented to develop a proposed algorithm through all main procedures of tabu search from the solution initialization to the solution performance measurement. A resource plan is initially generated by simulating an ideal plan from a backward scheduling approach. According to finite capacity, the initial plan might be infeasible due to overloading capacities in some periods. An improvement algorithm is generated to improve the initial resource plan to be feasible and toward an ideal plan. The improvement algorithm is decomposed into two sub-algorithms based on the objective function: the overloading improvement algorithm (OIA) and the makespan improvement algorithm (MIA). These two algorithms basically are used to search a new solution and to generate a feasible plan that comes as close as possible to an ideal solution. OIA aims to improve the initial solution from overloading capacities. Meanwhile, MIA is used to continually improve the solution obtained from OIA. The objective of MIA is to improve earliness, tardiness, and lead time. An improvement of all these parameters can reduce makespan in the system so that this algorithm is called the makespan improvement. Neighborhood structure is an important procedure in tabu search that affects the efficiency of a new solution. A latest possible start time (LPST) concept, which presents the latest start time

to finish a job on time, is proposed to design a moving space in the neighborhood structure. Then, an integration of pull and push approaches is developed as a scheduler to create a new resource plan based on an ideal plan.

In the remainder, this chapter is organized as follows. Section 4.2 describes the problem statement. Then, Section 4.3 illustrates the mathematical model for tabu search. Section 4.4 describes the details of procedure and parameter required in the proposed tabu search algorithm. Section 4.5 presents the computational experiments and Section 4.6 discusses the planning results of the tabu search algorithm. Finally, Section 4.7 summarizes the conclusion of this research.

4.2 Problem statement

A set of *I* jobs need to be planned on a set of *H* resources in order to minimize a weighted cost function with costs for the earliness, tardiness and lead time of each job. A variable number of operations for each job are allowed, and each job follows a different routing (sequence that the job visits resources). A discrete-time model of capacity is used. On each resource, a planning horizon is uniformly divided into intervals (time buckets), which capacity can vary if desired. Since tactical-level planning is emphasized, consecutive operations for a job are not allowed to be processed in the same bucket, and each operation is performed in a single bucket.

4.3 Mathematical model

In this section, a new binary integer linear programming formulation of job shop planning problems is described. The model provides optimal solutions for small instances that can be compared to solutions obtained with the more scalable tabu search

method described in Section 4.4. This model is similar to the model in the previous chapter, but a difference is that this model emphasizes job planning on only existing resources. The additional capacities from subcontractor and extra resources are not allowed in the model. The notation of all parameters is denoted as follows.

$T(i, j)$	=	Index of bucket when operation j of job i is planned
L_{ij}	=	Latest possible start time of operation j of job i
P_{ij}	=	Process time of operation j of job i
R_{ij}	=	Routing: index of resource that performs operation j of job i
J_i	=	Number of operations of job i
S_i	=	Earliest start date of job i
D_i	=	Due date of job i
F_i	=	Finish date of job i
B_{hk}	=	Capacity of bucket k for resource h
P_i^e	=	Earliness penalty for job i
P_i^t	=	Tardiness penalty for job i
P_i^l	=	Lead time penalty for job i
C_i^e	=	Earliness cost of job i
C_i^t	=	Tardiness cost of job i
C_i^l	=	Lead time cost of job i
i	=	Index for set of jobs; $i = 1 \ldots I$
j	=	Index for set of operations required by a job; $j = 1 \ldots J_i$
h	=	Index for set of resources; $h = 1 \ldots H$
k	=	Index for set of buckets on each resource; $k = 1 \ldots K$

Decision variable:

x_{ijk} = 1 if operation j of job i is planned in bucket k, 0 otherwise

Model:

Minimize $\sum_{i=1}^{I} (C_i^e + C_i^t + C_i^l)$

subject to

$$\sum_{k=1}^{K} kx_{i1k} \geq S_i \qquad i = 1\ldots I \qquad (13)$$

$$\sum_{k=1}^{K} kx_{ijk} < \sum_{k=1}^{K} kx_{i(j+1)k} \qquad i = 1\ldots I, j = 1\ldots J_i - 1 \qquad (14)$$

$$\sum_{k=1}^{K} x_{ijk} \leq 1 \qquad i = 1\ldots I, j = 1\ldots J_i \qquad (15)$$

$$\sum_{i=1}^{I} \sum_{j=1}^{J_i} x_{ijk} P_{ij} \leq B_{hk}$$

where $R_{ij} = h$ $\qquad h = 1\ldots H, k = 1\ldots$K (16)

$x_{ijk} \in \{0,1\}$ (17)

where

$T(i, j) = \sum_{k=1}^{K} kx_{ijk}$ (18)

$F_i = T(i,J_i)$ (19)

$C_i^e = ((D_i - 1) - F_i)(P_i^e)$ if $F_i < (D_i - 1)$ (20)

$C_i^t = (F_i - (D_i - 1))(P_i^t)$ if $F_i \geq (D_i - 1)$ (21)

$C_i^l = (F_i - T(i, 1) + 1)(P_i^l)$ (22)

The explanations of each constraint and equation are described as follows. Constraint (13) ensures that jobs are not planned on resources before they are available to start. Constraint (14) enforces operation precedence constraints. Constraints (15) and (17) ensure operations are planned in only one bucket. Constraint (16) enforces resource capacity constraints. Equation (18) defines the operation start time of each operation of a job. Equation (19) defines the finish date of a job, which is used to calculate the various costs for each job. Equation (20) defines the earliness cost for a job, equation (21) defines the tardiness cost for a job, and equation (22) defines the lead time cost for a job.

4.4 Tabu search for resource planning

Tabu search is a global iterative optimization approach which means the search moves from one solution to another better solution in neighborhood spaces (Taillard 1989). The approach tries to find local optimality with a strategy of forbidding certain moves in order to prevent search cycling. A forbidden move is called tabu. The tabu move will be held in a memory with a relatively short time and then it will be released from the tabu status and changed to be accessible (Glover 1986).

A tabu search procedure begins with generating an initial solution. Then the next step is to define a solution space and search for neighborhoods. The generated neighborhoods are all possible moves from the current solution space. The next solution is selected by evaluating all neighborhoods and moving to the neighborhood that provides the best performance. The move will be updated in the short term memory called tabu list. This tabu restriction is used to prevent cycling of moving. Then, the aspiration criteria checking procedure is performed to accept or reject the neighborhood

to be the best solution of the system. The last procedure is a stopping criteria procedure. The stopping criteria are used to terminate the search procedure when a solution is found or any stopping condition is satisfied.

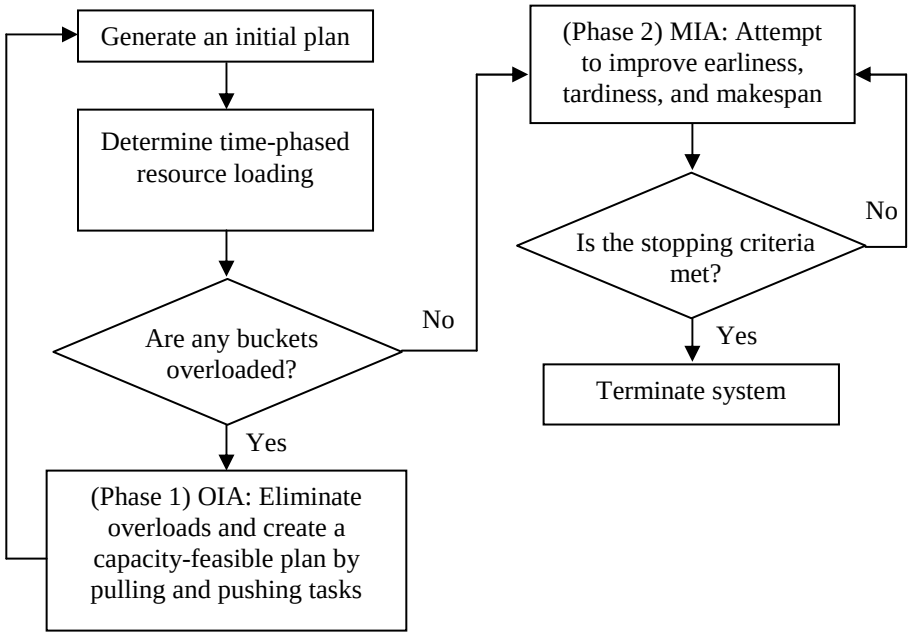


Figure 14: Tabu search algorithm procedure

This study proposes the new tabu search method to generate resource plans. Recall that an ideal resource plan is a plan that jobs need to be completed on their due date in order to avoid the costs from WIP, finished goods inventory, and backorder. The procedure of the proposed algorithm is illustrated in Figure 14. The details of each procedure will be described based on the following main elements of tabu search: solution initialization, search space and neighborhood structure, tabu list, aspiration condition, and stopping criteria.

4.4.1 Solution initialization

Starting a problem with a good solution will help the solver reach the solution faster (Danna et al. 2004). Therefore, to create the first resource plan that is similar to an ideal schedule, the backward scheduling approach with no capacity constraints is used to initialize a solution. The backward scheduling approach begins by loading the last operation of a job to finish at its due date. It then continues by loading the job's preceding operation to finish at the start time of the last operation. This process is continued, working backward in time, until the first operation of the job is loaded. The advantage of a backward algorithm is that a schedule can be generated with no late job. Also, this method helps to minimize WIP and shorten job lead time. To initialize a solution, resource constraint relaxation is assigned by assuming an ideal resource capacity. It means that a resource can support as many jobs as possible without capacity restriction. Even though this method provides the optimal planning with zero earliness and lateness, it still affects overloaded capacity where the total loading might be greater than available capacity in some periods. A greater detail of the backward scheduling approach can be seen in section 3.3.1.

4.4.2 Neighborhood structure

In local search, neighborhood structure is one of the most important procedures which are used to search and develop new solutions. It directly dominates the efficiency of new solutions in terms of solution quality. Unnecessary and infeasible moves need to be eliminated as much as possible to reach a desired solution more efficient (Zhang et al. 2007). The JIT concept has been applied into neighborhood structure in order to narrow the search and reduce inappropriate moves as well as generate a feasible plan,

which is close to an ideal solution. With the JIT concept, the proposed algorithm based on tabu search is developed and divided into two phases, which are the overloading improvement algorithm (OIA) and the makespan improvement algorithm (MIA). The approach uses the ideal plan based on JIT as the initial resource plan. Phase 1, the Overloading Improvement Algorithm (OIA), will make the ideal plan feasible rather than attempting to construct a good feasible plan from scratch. Minimal adjustments to task times are performed to create a finite-capacity plan. Phase 2, the Makespan Improvement Algorithm (MIA), searches for alternate finite-capacity plans which have decreased earliness, tardiness and lead time. The details of both algorithms are described as follows.

(1) Overloading Improvement Algorithm (OIA)

According to the limitation of capacity, the initial solution from the backward planning approach will create some overloaded capacities. The proposed algorithm attempts to improve the resource plan from these capacity shortages and make the plan become feasible. The OIA procedure begins with choosing the maximum overloading bucket in the horizon. The jobs in the bucket are identified and put in the list for assigning the move. Each job represents individual neighborhood. Since we allow to move one job at a time, the neighborhood is a new solution from moving the job out of the overloaded bucket to reduce the capacity shortage. The selected job will be offloaded to other sufficient buckets by using the pull and push methods. These both methods are similar to the solution generating concept implemented in He et al. (1993). The pull method is used to move a job to process in the earlier bucket positions.

Meanwhile, the push method is used for pushing a job to process in the later bucket positions.

In the backward capacity approach, each job needs to be processed at the latest possible start time. On the other hand, this approach generates the upper bound of the solution. Hence, when some buckets are overloaded, the pull method will be the first approach applied to reduce the excess loading. The pull method is used until either finding a new sufficient bucket loading or reaching the earliest start date of the job. If the job is shifted back until reaching the start date without finding a new available bucket, the push method will be implemented. The moving also includes the rest of the operations in the job. However, the moving in the other operations will be selected based on the moving type of the first move. If the pull method is applied, the precedence operations will be moved to the position ahead of the current bucket. If the push method is implemented, the successive operations will be focused to move. After moving all concerned operations, a new neighborhood is generated. This procedure will be repeated until all jobs in the overloaded bucket are moved. To enumerate the total neighborhood for individual iterations, it can be determined that the total neighborhood is equal to the number of jobs in the maximum overloaded bucket. For example, if the max overloaded bucket contains two jobs, two different neighborhoods can be created.

To clarify the procedure of pull and push methods, Figures 15 and 16 present load graphs of these methods at different machines and buckets. The dotted line presents loading capacity for each resource. It assumes that a job has to be processed on three operations. The job that plans to be offloaded from the overloaded bucket is called the target job. From Figure 15(a), the target job has the operation start time sequence at

bucket (3, 4, 5). It can be seen that the overloaded capacity is at bucket 4 of machine 2 or at operation 2 of the target job. To eliminate the overloaded capacity, the pull method will be used by pulling the overloaded operation in bucket 4 to the nearest available bucket in bucket 3. Once operation 2 is moved, the precedence operation like operation 1 has to be moved also. Operation 1 will be processed in bucket 2. Figure 15(b) shows the consequence which the new operation start time sequence is (2, 3, 5).

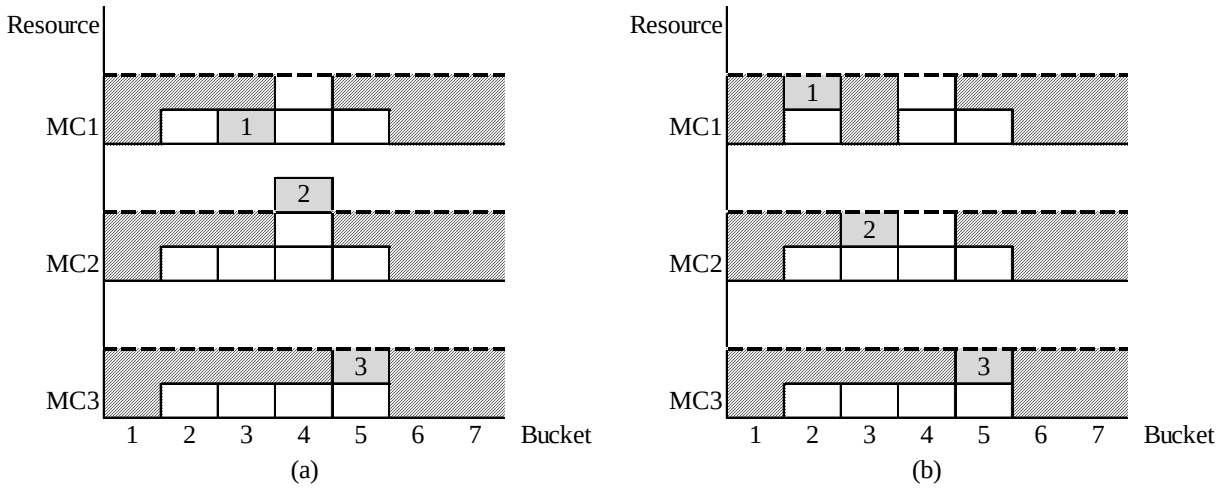


Figure 15: Pull method results (a) Before: Operation start time = (3, 4, 5); and (b) After: Operation start time = (2, 3, 5)

In Figure 16, the push system will be applied in case the overloaded operation cannot be shifted to the left side due to reaching the earliest start date. The overloaded operation at bucket 2 will be pushed out to bucket 3. Also, the successive operation has to be pushed out as well. In this case, the new operation start time sequence is (1, 3, 5).

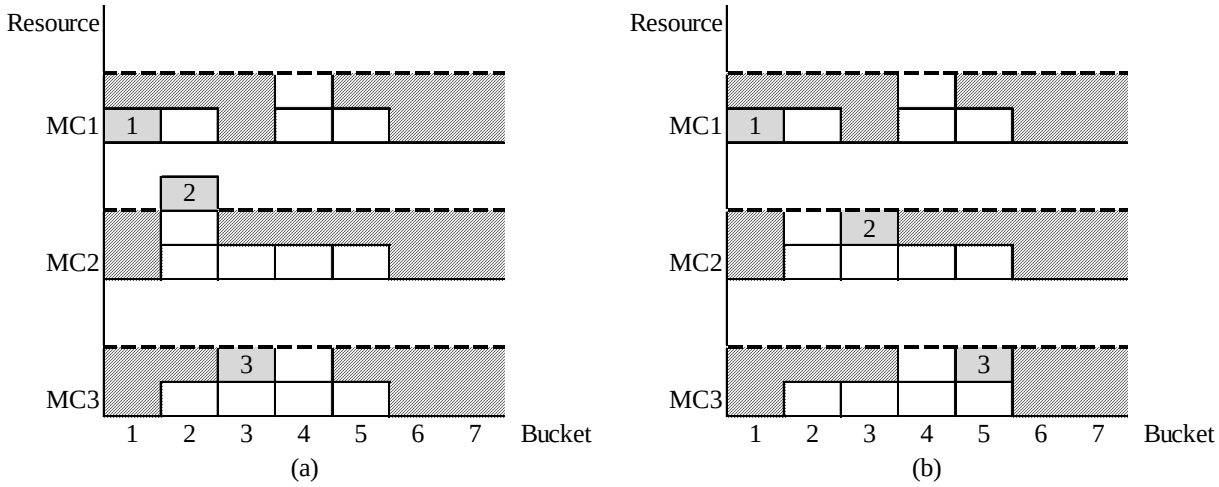


Figure 16: Push method results (a) Before: Operation start time = (1, 2, 3); and (b) After: Operation start time = (1, 3, 5)

To define the best neighborhood, three-tier hierarchy decision making, which consists of smallest maximum overload, minimum total tardiness, and minimum number of overloaded buckets, is implemented. The procedure of the best neighborhood determination considers one tier of measurement at a time by starting from the first tier. If the first tier provides more than one best solution, the second and third tier will be used to determine the final best neighborhood respectively. Otherwise, the best neighborhood is obtained.

After the best neighborhood is found, the next step is to decide whether the best neighborhood can be the best solution of the problem. Aspiration criteria, which are the conditions to determine the acceptance of a new best neighborhood as a new best solution for the next iteration, are used to evaluate the solution. The criteria consist of three-tier hierarchy of decision making such as minimum of maximum overload, total tardiness, and total lead time. The procedure to find the best solution is as same as the procedure of the best neighborhood determination in which if the first tier such as

minimum of maximum overload cannot define the best solution, the second and third tier will be successively considered. Whether or not the best neighborhood becomes the best solution, the best neighborhood will be used to start the next iteration.

(2) Makespan Improvement Algorithm (MIA)

The algorithm continues to improve a resource plan in terms of earliness, tardiness, and lead time. The JIT concept is applied into search space and neighborhood structure, which are important procedures of local search, to lead the search to a desire solution. The algorithm attempts to narrow the search space by identifying the distance between a current solution and an ideal solution and defining appropriate moves. To explore new solutions toward an ideal resource plan, neighborhood structure must efficiently perform. Several schemes have been proposed to generate neighborhoods (Tsubakitani and Evans 1992, Dell'Amico and Trubian 1993, James 1997). Three main schemes include insert, swap, and a combined method of insert and swap. James (1997) compared these three schemes and concluded that the best scheme was the hybrid of insert and swap neighborhood since it effectively provide a variety of new solutions. Therefore, in this algorithm both insert and swap schemes are applied to neighborhood structure in which the purpose is to move the late or early jobs back in order to close to an ideal solution.

The MIA procedure begins with obtaining an initial solution from OIA which it may contain either early or tardy jobs. To improve the solution, a target job is defined as a job which has maximum absolute lateness. A target job (i, j) represents each operation j of target job i. A set of possible neighborhood (NBH) points, where the notation (y, z) represents a NBH point (NBH job y, NBH operation z), is then generated. The

insert/swap method will interchange the target job with the selected NBH jobs in the horizon. In this algorithm, two new parameters, earliest possible start time (EPST) and latest possible start time (LPST) are introduced. These two parameters are used to determine the range of possible operation start time of each job and also define properly selected NBH jobs. The EPST is the earliest time that each operation of a job can be processed immediately since the job arrives the system. The EPST (E_{ij}) of each job i and operation j can be determined as equation (23). Whereas the LPST is the latest time that each operation of a job is processed and the job can be finished on time. This parameter is used to identify jobs that can be the NBH jobs. The LPST (L_{ij}) of each job i and operation j can be determined as equation (24). The notation of the parameters is described in section 4.3.

$$E_{ij} = S_{i+} j - 1 \tag{23}$$

$$L_{ij} = (D_i - 1) - (J_i - j) \tag{24}$$

The search of NBH points is different depending on the type of the target job. If the target job is tardy, the search will focus on the resource of the last operation of the target job. This is to guarantee that the new solution of the target job will not be processed later than the current plan. Meanwhile, if the target job is an early job, the search will emphasize the resource of the first operation of the target job.

In order to define the NBH points, the results from two methods, the insert and swap methods, are considered. In the insert method, the sufficient buckets, which can process the target job without moving any other jobs, are considered. In this case, the NBH job will be updated as null (∅) in a set of NBH points. For the swap method, if the

target job is tardy, the job which has equal or greater LPST will be accepted to be a NBH point. Otherwise, the job which has equal or lesser LPST will be selected. The search space for both two schemes is considered dependent on the type of target job as follows:

Tardy job: Start = E_{iJi}

End = $T(i, J_i) - 1$

Early job: Start = $T(i, 1) + 1$

End = L_{i1}

When a set of NBH points is defined, the next step is creating the neighborhood of each NBH point. The neighborhood of MIA is a new solution when interchange the target job with the NBH point and other related jobs. To generate new neighborhoods, several jobs in the horizon will be considered to move by using the pull and push methods. These jobs can be categorized into two types which are a NBH job and move job. The NBH job (y, z) represents each NBH point or the job that swaps with the target job (i, j) when $j = J_i$ in the tardy target job case or $j = 1$ in the early target job case. Meanwhile, the move job (y, z) represents the job that swaps with the target job (i, j) when $j \neq J_i$ in the tardy target job case or $j \neq 1$ in the early target job case. The details of neighborhood generation are summarized as follows.

Procedure:

Step 1: Insert or swap the target job (i, j) with a NBH job (y, z) and determine the new operation start times of the target job.

Tardy job: $T(i, J_i) = T(y, z)$

Early job: $T(i, 1) = T(y, z)$

Step 2: Find the new operation start time ($T(y, z)$) for the NBH job (y, z) or move job (y, z). This also includes the successive operations in the NBH (or move) job. The search method will be performed based on the condition of operation z as shown below.

(1) If operation z is equal to one, the pull method is used to search the new operation start time. The search would not be processed earlier than the earliest start date of the NBH (or move) job y. If the pull method cannot provide the new solution, the push method is then used to search an available bucket.

(2) If operation z is not equal to one, the push method is used to search the new operation start time by starting the search after the operation start time of the previous operation of the NBH (or move) job (y, z).

Step 3: Find the new operation start times for the rest operations (m) of the target job (i, j) and their possible move jobs as the following steps.

(1) For each operation, define the search range which depends on whether a target job is tardy or early. For tardy jobs, the search considers earlier operation start times that could reduce tardiness. For early jobs, the search considers later operation start times that could reduce earliness, but it would not cause the job to become tardy.

Tardy jobs: Start = E_{im} where $m = J_i$ -1,…, 1

End = $T(i, j) - (J_i - m)$

Early jobs: Start = $T(i, j) + (m\text{-}1)$ where $m = 2,\ldots, J_i$

End = L_{im}

(2) Determine the new operation start time for the operation *m* of the target job by using the insert and swap methods. The insert method is implemented first. If it cannot provide any solution, the swap method will be used. If the solution cannot be found, skip to the next NBH point or terminate the system if the NBH point list is empty.

(i) Insert method: When a bucket with availability is located in the obtained search range, the operation *m* will be inserted into the bucket and then the search of the operation *m* will be terminated. Also, if an operation of jobs in the obtained search range is 1 and it can be pulled to process in the previous available buckets but not earlier than its earliest start date, the bucket can be free up and process the operation *m*. The move job in this case will be considered to be null.

(ii) Swap method: Comparing the L_{im} of the target job with the LPST of the jobs in the obtained search range. The move job is the job which its LPST is equal or greater than L_{im} for the tardy target job or equal or lesser than L_{im} in the early target job. The search will be break once one move job is found.

(3) Update the new operation start time (*T*) for the operation *m* of the target job.

(4) If the move job is null, proceed to task (5) of step 3. Otherwise, proceed to step 2 to find the new operation start time (*T*) of move job (*y, z*).

(5) Check whether there are any operation *m* of target job (*i*, *j*) that has not been updated. If yes, repeat task (1) of step 3. Otherwise, proceed to step 4.

Step 4: Update the new solution and evaluate the solution performance, such as earliness, tardiness, and lead time.

The above procedure is repeated for all NBH points. To evaluate the best neighborhood, three-tier hierarchy decision making is used to determine the solution in which the three tiers consist of total tardiness, total earliness and total lead time. These criteria, total tardiness, total earliness, and total lead time, is also applied for accepting the best solution of the system in the aspiration criteria procedure.

To clarify the MIA procedure, the simple instance is presented as follows. There are four jobs needed to be planned. All assumptions, such as job routing, earliest start date and due date, are shown in Table 18. An initial solution is generated by using the backward scheduling approach. The load graph of the initial solution is presented in Figure 17(a) in which the dotted line represents resource capacity and the diagonal strip presents an available bucket. Each box presents a job which is processed in a specific bucket. In each box, the first number represents a job number and the second number represents an operation number. From Figure 17(a), it can be seen that the original resource capacity cannot support all requirements in period 2 at MC1, period 3 at MC2, and period 4 at MC3. OIA is applied to improve these overloading capacities. It is assumed that the OIA output is illustrated in Figure 17(b).

Table 18: Data for the testing instance

Job	Routing	Earilest start date	Due date
1	M1, M2, M3	1	5
2	M2, M3, M1	1	5
3	M3, M1, M2	1	5
4	M1, M2, M3	1	5

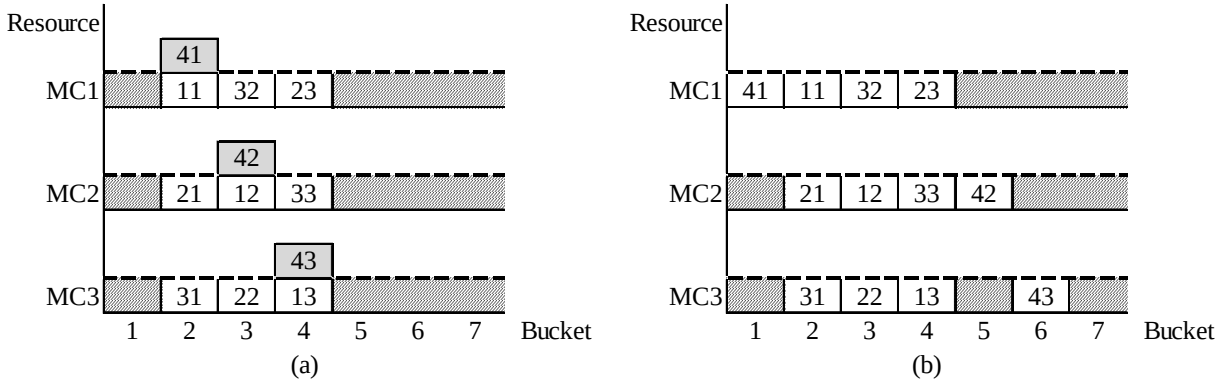


Figure 17: OIA load graph (a) Ideal initial solution; and (b) The output of OIA

The MIA procedure starts from defining a maximum deviation time. From Figure 17(b), the target job with a maximum tardiness is selected, which is job 4 (tardiness = 2 buckets). The next step is determining the NBH points. The search focuses at the resource of the last operation of job 4 which is MC3. The search range starts from the earliest start time of job (4, 3), which is bucket 3, to the previous current bucket of job (4, 3), which is bucket 5. The NBH points are determined by using LPST parameter and accept the job which has an equal or larger LPST than the LPST's target job (4, 3) (LPST = 4). The NBH points can be created as Table 19 below and the example of NBH generation is shown in the following steps.

Table 19: Example of NBH point generation

Bucket number	Job in bucket	LPST	NBH point {NBH job(y , z), bucket}
3	(2, 2)	(2, 2) = 3	{-}
4	(1, 3)	(1, 3) = 4	{{(1, 3), 4}}
5	∅	∅	{{(1, 3), 4}, {∅, 5}}

Step 1: At the NBH point = {(1, 3), 4}, the target job (4, 3) is swapped with the NBH job (1, 3) and then update the new operation start time (T).

$T(4, 3) = 4$

Step 2: Find the new operation start time of the NBH job 1. The push method is used and it can be found that the nearest available bucket for the NBH job 1 is bucket 5. Update the new operation start time of the NBH job (1, 3) to be 5.

$T(1, 3) = 5$

New operation start times of NBH job 1 = {2, 3, 5}

Step 3: Find the new operation start time for the rest operations of the target job 4.

Target job (4, 2):

(1) Create the search range: Start = bucket 2 and End = bucket 3.

(2) Consider job (2, 1) in bucket 2. The insert method can be used to find the nearest available bucket by pulling job (2, 1) to process in bucket 1. Then insert job (4, 2) into bucket 2.

(3) Update the new operation start times (T).

$T(4, 2) = 2$

$T(2, 1) = 1$

New operation start time of move job 2 = {1, 3, 4}.

Target job (4, 1):

(1) Create the search range: Start = bucket 1 and End = bucket 1.

(2) Do not need to search for the new operation start time since the start time and the end time are in the same bucket.

New operation start time of target job 4 = {1, 2, 4}

Step 4: Update the new solution of this NBH point.

Operation start time = {{2, 3, 5}, {1, 3, 4}, {2, 3, 4}, {1, 2, 4}}

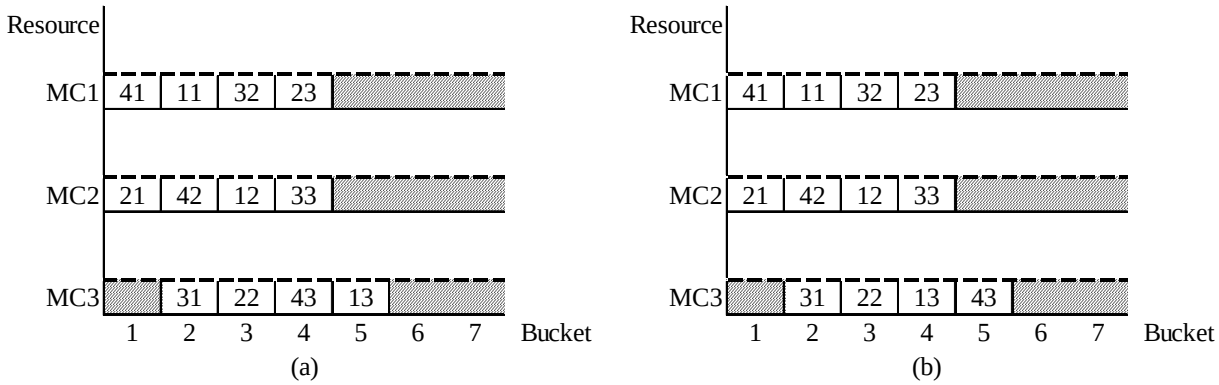


Figure 18: Solution for move point (a) {(1, 3)}; and (b) {ϕ}

This procedure will be repeated for another NBH point. Figure 18 presents the load graph results for both NBH points. In Figure 18(a) and (b), these two results provide the same performance results which they have one bucket of total tardiness, zero buckets of total earliness, and fifteen buckets of total lead time. Either solution can be chosen to start for the next iteration. MIA will keep improving the solution until it meets the stopping criteria. The details of stopping criteria are discussed in section 4.4.5. The final result for this instance is presented in Figure 19. It can be observed that

this plan can improve total tardiness, total earliness, and total lead time to zero buckets, one bucket, and fourteen buckets, respectively.

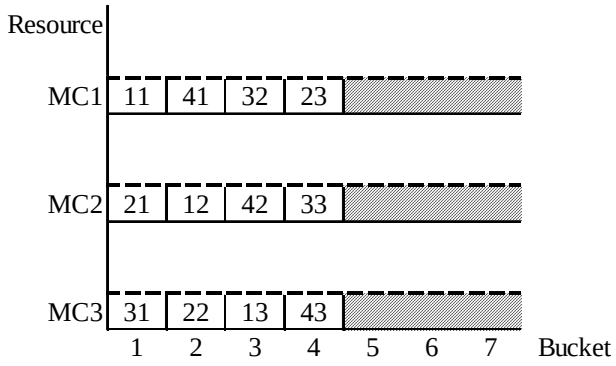


Figure 19: Load graph of the MIA result

4.4.3 Tabu list

Tabu list is the short term memory that is used to prevent cycle searching. This list contains a set of tabu points or an array of pairs of moving jobs and moving positions. When the best neighborhood is chosen, this data will be updated in the tabu list. Tabu point will be prohibited from reversing until either passing a predefined number of iterations or finding a better solution from this reversing. In this research, the size of tabu list is varied based on number of jobs in problem instances. Glover (1986) reported that the best tabu size is approximately seven. However, Tsubakitani and Evans (1992) mentioned that the size of tabu list depends on the problem size and local search heuristics. It should be set as small as possible but long enough to prevent the cycling.

4.4.4 Aspiration criteria

Aspiration criteria are the conditions which determine the acceptance of a new best neighborhood as a new best solution for the next iteration. This research divided

the aspiration criteria into two parts based on the proposed algorithms: OIA and MIA. Both algorithms use a hierarchical decision making approach to accept or reject a new neighborhood. In OIA, the three-tier decision making hierarchy which consists of minimum of maximum overload, total tardiness, and total lead time is applied. Meanwhile, in MIA, the three emphasized tiers are total tardiness, total earliness, and total lead time. In hierarchical decision making, the best solution will be determined by measuring one parameter at a time. The measurement will consider from the highest priority parameter which is the first tier parameter. If the first tier measurement cannot decide the best solution, the second and third tier will be successively considered.

4.4.5 Termination criteria

The terminate condition in the algorithm procedure consists of three criterion. In OIA, the process will be repeated until one of three stopping criteria is met. The first criterion is zero overloaded capacity. The second criterion is reaching the number of sequential iterations without improvement of an objective value. The third criterion is reaching the predefined number of total iteration. If the first condition is satisfied, the system will be moved forward to MIA. Otherwise, the system will be terminated.

In MIA, there also have three stopping criteria. The first is a number of sequential iterations without improvement of an objective value. The second is a predefined computational time. The third is a predefined number of total iteration. If any condition is satisfied, the system will be terminated.

4.5 Computational experiments

The experiments are conducted to test the performance of the tabu search procedure. The study is separated into two sections. The first section is to investigate the performance of the tabu search algorithm in combinatorial problems. The second section is to test the efficiency of the tabu search algorithm on large problem instances and compare the solutions to the dispatching rules, such as first in first out (FIFO) and earliest due date (EDD). Both experiments are to investigate the impacts of planning improvement at different variabilities and then identify the most impacted parameter on the solution performance. The algorithms are coded up on Eclipse 3.5.0 with the Java language and run on a personal computer with a 1.73 GHz processor.

4.5.1 Experimental parameters

In order to observe the solution impact at different problem scalability and difficulty, the instances are generated at different variability levels of three interesting parameters: number of operations per job, due date tightness, and resource utilization. The number of operations presents process requirement in each job. Since the problem presents job shop planning with multiple machines, each job requires an individual routing based on an assigned number of operations. These different process lengths reflect different problem complexities. Three levels of number of operations are proposed. The factor level is varied from small range to larger range of uniform distribution of number of operations as shown in Table 20. To interpret the data, if an instance has the number of operations at [3, 10], it means that each job will randomly generate the number of operations from 3 to 10 operations.

The second parameter, due date tightness, is time allowed to process a job. Since different ranges of due time reflect machine requirement at varied periods of time, four levels of due date tightness, as in Table 20, are considered. These due date ranges are varied based on the number of operations per job.

The last parameter, resource utilization, represents the congestion in the system as well as number of jobs. To be able to measure the levels of planning difficulty and determine the performance impact from that variance, resource utilization is used to define a number of jobs in each planning instance. Seven levels of percentage of loading per bucket at the bottleneck resource are varied with the uniform distribution [25%, 95%], [45%, 95%], [65%, 95%], [7%, 95%], [85%, 95%], [75%, 110%], and [55%, 140%]. An average resource utilization of each level is 60%, 70%, 80%, 85%, 90%, 92.5%, and 97.5%, respectively.

For other fixed parameters, processing time of each job on each machine is randomly generated with uniform distribution [300, 700] seconds. In the planning horizon, loading capacity per bucket is 86,400 seconds and the length of the horizon is varied based on the number of operations parameter. Cost is used as performance measurement. All costs are assumed as follows: earliness cost = $30/bucket, tardiness cost = $50/bucket, and lead time cost = $20/bucket. A size of tabu list is assumed to be 7 when the number of jobs (N) is less than 225 jobs and a round down number of N/32 when the number of jobs (N) is greater than 225 jobs (Tsubakitani and Evans 1992). An unimproved solution number for terminating the system is set to 500.

Table 20: Summary of experimental parameters

Number of operations	Due date tightness	Planning horizon (buckets)
[3, 5]	5, [5, 10], [5, 15], [5, 20]	25
[3, 10]	10, [10, 15], [10, 20], [10, 25]	35
[3, 15]	15, [15, 20], [15, 25], [15, 30]	45

4.5.2 Experimental design

In the experiment, the problems are coordinately tested with the complicated conditions on variability and scalability from distinct job configurations (routings and due dates) and resource availability. Since there are three levels of number of operations, four levels of due date tightness, and seven levels of resource utilization, 84 total experiments are studied. The computational time is assumed to be unlimited for OIA, but it is limited at 1,800 seconds for MIA. The experiment is divided into two parts which are the study of performance of the tabu search algorithm at different variabilities and the comparative study of the tabu search algorithm with the other heuristics, FIFO and EDD.

4.6 Results and analysis

This section presents the numerical results of performance indicators, such as earliness time, tardiness time, total lead time, and computational time at different combinatorial problems. Tables 21-32 illustrate the summary of the solution improvement in each stage of the tabu search algorithm, solution initialization, OIA, and MIA.

Table 21: Results for number of operations [3, 5] and due date tightness [5]

Resource utilization	Number of jos	Stage	Iteration	Tardiness time (Bucket)	Earliness time (Bucket)	Total lead time (Bucket)	Number of overloaded buckets	Computational time (seconds)
[25%, 95%]	230	Initial solution	0	0	0	933	4	0
		OIA	4	2	2	935	0	8
		MIA	509	0	5	934	0	1,800
[45%, 95%]	304	Initial solution	0	0	0	1,222	5	0
		OIA	7	1	8	1,222	0	11
		MIA	472	0	11	1,223	0	1,800
[65%, 95%]	350	Initial solution	0	0	0	1,434	23	0
		OIA	41	114	14	1,489	0	76
		MIA	79	51	26	1,459	0	1,800
[75%, 95%]	379	Initial solution	0	0	0	1,530	25	0
		OIA	60	293	10	1,630	0	141
		MIA	99	214	16	1,565	0	1,800
[85%, 95%]	402	Initial solution	0	0	0	1,593	29	0
		OIA	73	424	15	1,693	0	128
		MIA	119	359	15	1,624	0	1,800
[75%, 110%]	389	Initial solution	0	0	0	1,541	32	0
		OIA	59	252	11	1,647	0	131
		MIA	101	208	17	1,583	0	1,800
[55%, 140%]	410	Initial solution	0	0	0	1,617	35	0
		OIA	88	439	11	1,748	0	181
		MIA	118	395	14	1,679	0	1,800

Table 22: Results for number of operations [3, 5] and due date tightness [5, 10]

Resource utilization	Number of jos	Stage	Iteration	Tardiness time (Bucket)	Earliness time (Bucket)	Total lead time (Bucket)	Number of overloaded buckets	Computational time (seconds)
[25%, 95%]	287	Initial solution	0	0	0	1,002	6	0
		OIA	11	2	10	1,012	0	16
		MIA	393	0	29	1,010	0	1,800
[45%, 95%]	306	Initial solution	0	0	0	1,207	12	0
		OIA	31	4	47	1,238	0	84
		MIA	77	0	60	1,278	0	1,800
[65%, 95%]	354	Initial solution	0	0	0	1,426	25	0
		OIA	53	148	49	1,499	0	84
		MIA	92	101	68	1,537	0	1,800
[75%, 95%]	384	Initial solution	0	0	0	1,499	26	0
		OIA	77	236	31	1,691	0	138
		MIA	106	181	54	1,622	0	1,800
[85%, 95%]	405	Initial solution	0	0	0	1,614	36	0
		OIA	106	359	15	1,839	0	231
		MIA	84	276	49	1,723	0	1,800
[75%, 110%]	411	Initial solution	0	0	0	1,624	42	0
		OIA	107	369	44	1,813	0	249
		MIA	120	286	75	1,760	0	1,800
[55%, 140%]	405	Initial solution	0	0	0	1,596	44	0
		OIA	122	521	54	1,873	0	275
		MIA	75	407	85	1,786	0	1,800

Table 23: Results for number of operations [3, 5] and due date tightness [5, 15]

Resource utilization	Number of jos	Stage	Iteration	Tardiness time (Bucket)	Earliness time (Bucket)	Total lead time (Bucket)	Number of overloaded buckets	Computational time (seconds)
[25%, 95%]	225	Initial solution	0	0	0	889	7	0
		OIA	12	5	13	901	0	12
		MIA	488	0	45	891	0	1,800
[45%, 95%]	318	Initial solution	0	0	0	1,287	20	0
		OIA	65	94	59	1,473	0	131
		MIA	146	37	168	1,510	0	1,800
[65%, 95%]	351	Initial solution	0	0	0	1,414	30	0
		OIA	88	197	84	1,661	0	124
		MIA	119	98	171	1,730	0	1,800
[75%, 95%]	386	Initial solution	0	0	0	1,544	36	0
		OIA	137	313	67	1,921	0	234
		MIA	108	265	86	1,881	0	1,800
[85%, 95%]	404	Initial solution	0	0	0	1,587	40	0
		OIA	137	483	121	1,934	0	231
		MIA	93	359	140	1,965	0	1,800
[75%, 110%]	409	Initial solution	0	0	0	1,603	43	0
		OIA	165	463	185	2,085	0	288
		MIA	103	290	284	2,126	0	1,800
[55%, 140%]	415	Initial solution	0	0	0	1,657	37	0
		OIA	168	595	127	2,178	0	330
		MIA	10	412	234	2,135	0	1,800

Table 24: Results for number of operations [3, 5] and due date tightness [5, 20]

Resource utilization	Number of jos	Stage	Iteration	Tardiness time (Bucket)	Earliness time (Bucket)	Total lead time (Bucket)	Number of overloaded buckets	Computational time (seconds)
[25%, 95%]	269	Initial solution	0	0	0	1,097	21	0
		OIA	48	36	35	1,224	0	59
		MIA	221	0	215	1,243	0	1,800
[45%, 95%]	324	Initial solution	0	0	0	1,298	28	0
		OIA	95	199	139	1,562	0	182
		MIA	198	77	306	1,595	0	1,800
[65%, 95%]	358	Initial solution	0	0	0	0	0	0
		OIA	0	221	186	1,883	0	277
		MIA	156	111	301	1,920	0	1,800
[75%, 95%]	388	Initial solution	0	0	0	1,564	39	0
		OIA	153	436	153	2,003	0	298
		MIA	113	279	273	2,106	0	1,800
[85%, 95%]	394	Initial solution	0	0	0	1,586	39	0
		OIA	188	434	197	2,175	0	314
		MIA	96	298	270	2,295	0	1,800
[75%, 110%]	406	Initial solution	0	0	0	1,612	43	0
		OIA	181	450	293	2,145	0	373
		MIA	95	287	324	2,309	0	1,800
[55%, 140%]	433	Initial solution	0	0	0	1,699	40	0
		OIA	231	691	260	2,481	0	568
		MIA	78	489	382	2,530	0	1,800

Table 25: Results for number of operations [3, 10] and due date tightness [10]

Resource utilization	Number of jos	Stage	Iteration	Tardiness time (Bucket)	Earliness time (Bucket)	Total lead time (Bucket)	Number of overloaded buckets	Computational time (seconds)
[25%, 95%]	390	Initial solution	0	0	0	0	10	0
		OIA	18	2	10	2,584	0	101
		MIA	56	0	11	2,582	0	1,800
[45%, 95%]	437	Initial solution	0	0	0	2,836	12	0
		OIA	19	0	24	2,847	0	123
		MIA	31	0	21	2,850	0	1,800
[65%, 95%]	486	Initial solution	0	0	0	3,133	23	0
		OIA	66	338	28	3,262	0	437
		MIA	50	276	47	3,210	0	1,800
[75%, 95%]	528	Initial solution	0	0	0	3,450	35	0
		OIA	95	645	35	3,843	0	750
		MIA	30	556	63	3,675	0	1,800
[85%, 95%]	554	Initial solution	0	0	0	3,561	42	0
		OIA	102	860	33	4,062	0	788
		MIA	26	778	57	3,893	0	1,800
[75%, 110%]	580	Initial solution	0	0	0	3,769	56	0
		OIA	142	1,228	37	4,572	0	1,228
		MIA	16	1,159	62	4,485	0	1,800
[55%, 140%]	560	Initial solution	0	0	0	3,620	55	0
		OIA	119	1,039	17	4,193	0	984
		MIA	30	1,034	17	3,994	0	1,800

Table 26: Results for number of operations [3, 10] and due date tightness [10, 15]

Resource utilization	Number of jos	Stage	Iteration	Tardiness time (Bucket)	Earliness time (Bucket)	Total lead time (Bucket)	Number of overloaded buckets	Computational time (seconds)
[25%, 95%]	365	Initial solution	0	0	0	2,430	12	0
		OIA	20	1	20	2,477	0	124
		MIA	6	0	18	2,490	0	1,456
[45%, 95%]	442	Initial solution	0	0	0	2,878	18	0
		OIA	46	66	1,446	2,947	0	286
		MIA	35	37	114	3,024	0	1,800
[65%, 95%]	490	Initial solution	0	0	0	3,154	26	0
		OIA	79	307	94	3,446	0	530
		MIA	41	231	120	3,378	0	1,800
[75%, 95%]	535	Initial solution	0	0	0	3,383	37	0
		OIA	96	449	78	3,708	0	898
		MIA	35	360	121	3,680	0	1,800
[85%, 95%]	547	Initial solution	0	0	0	44	152,100	0
		OIA	130	725	36	4,027	0	1,008
		MIA	20	577	130	3,995	0	1,800
[75%, 110%]	574	Initial solution	0	0	0	3,734	56	0
		OIA	148	877	98	4,343	0	1,279
		MIA	32	728	174	4,272	0	1,800
[55%, 140%]	578	Initial solution	0	0	0	3,725	54	0
		OIA	162	1,136	102	4,431	0	1,427
		MIA	30	939	226	4,321	0	1,800

Table 27: Results for number of operations [3, 10] and due date tightness [10, 20]

Resource utilization	Number of jos	Stage	Iteration	Tardiness time (Bucket)	Earliness time (Bucket)	Total lead time (Bucket)	Number of overloaded buckets	Computational time (seconds)
[25%, 95%]	391	Initial solution	0	0	0	2,538	14	0
		OIA	25	15	36	2,613	0	134
		MIA	40	2	59	2,622	0	1,800
[45%, 95%]	459	Initial solution	0	0	0	3,071	39	0
		OIA	101	309	94	3,603	0	733
		MIA	47	185	151	3,561	0	1,800
[65%, 95%]	512	Initial solution	0	0	0	3,361	44	0
		OIA	443	442	186	4,095	0	1,191
		MIA	40	330	246	4,099	0	1,800
[75%, 95%]	522	Initial solution	0	0	0	3,342	55	0
		OIA	137	616	104	4,001	0	1,369
		MIA	40	514	162	3,891	0	1,800
[85%, 95%]	554	Initial solution	0	0	0	3,575	69	0
		OIA	191	927	176	4,525	0	1,567
		MIA	38	753	223	4,508	0	1,800
[75%, 110%]	586	Initial solution	0	0	0	3,786	77	0
		OIA	223	1,061	101	5,087	0	1,917
		MIA	26	959	175	4,870	0	1,800
[55%, 140%]	617	Initial solution	0	0	0	4,059	90	0
		OIA	263	1,337	173	5,326	0	2,630
		MIA	26	1,171	271	5,239	0	1,800

Table 28: Results for number of operations [3, 10] and due date tightness [10, 25]

Resource utilization	Number of jos	Stage	Iteration	Tardiness time (Bucket)	Earliness time (Bucket)	Total lead time (Bucket)	Number of overloaded buckets	Computational time (seconds)
[25%, 95%]	386	Initial solution	0	0	0	2,516	32	0
		OIA	69	106	106	2,886	0	371
		MIA	59	51	205	2,959	0	1,800
[45%, 95%]	442	Initial solution	0	0	0	2,844	42	0
		OIA	120	289	146	3,447	0	817
		MIA	57	173	278	3,466	0	1,800
[65%, 95%]	493	Initial solution	0	0	0	3,149	62	0
		OIA	176	547	160	4,261	0	1,255
		MIA	37	404	263	4,298	0	1,800
[75%, 95%]	531	Initial solution	0	0	0	3,437	65	0
		OIA	194	706	209	4,602	0	1,477
		MIA	41	527	219	4,653	0	1,800
[85%, 95%]	560	Initial solution	0	0	0	3,578	83	0
		OIA	238	937	204	5,164	0	1,932
		MIA	38	742	249	5,150	0	1,800
[75%, 110%]	574	Initial solution	0	0	0	3,707	90	0
		OIA	234	1,117	216	5,131	0	1,934
		MIA	36	967	284	5,131	0	1,800
[55%, 140%]	599	Initial solution	0	0	0	3,884	98	0
		OIA	303	1,299	238	5,731	0	2,955
		MIA	26	1,119	361	5,678	0	1,800

Table 29: Results for number of operations [3, 15] and due date tightness [15]

Resource utilization	Number of jos	Stage	Iteration	Tardiness time (Bucket)	Earliness time (Bucket)	Total lead time (Bucket)	Number of overloaded buckets	Computational time (seconds)
[25%, 95%]	477	Initial solution	0	0	0	4,355	8	0
		OIA	24	0	14	4,382	0	485
		MIA	9	0	13	4,383	0	1,800
[45%, 95%]	544	Initial solution	0	0	0	4,932	17	0
		OIA	42	29	55	5,037	0	825
		MIA	17	14	51	5,053	0	1,800
[65%, 95%]	641	Initial solution	0	0	0	5,726	30	0
		OIA	104	604	82	6,136	0	2,861
		MIA	11	511	121	6,074	0	1,800
[75%, 95%]	679	Initial solution	0	0	0	6,240	62	0
		OIA	144	1,134	96	7,149	0	3,904
		MIA	4	1,125	96	7,101	0	1,800
[85%, 95%]	716	Initial solution	0	0	0	6,599	76	0
		OIA	177	1,397	83	7,734	0	6,358
		MIA	9	1,297	107	7,655	0	1,800
[75%, 110%]	735	Initial solution	0	0	0	6,563	73	0
		OIA	197	1,808	99	8,025	0	5,713
		MIA	10	1,700	139	7,896	0	1,800
[55%, 140%]	756	Initial solution	0	0	0	6,662	84	0
		OIA	225	2,323	129	8,366	0	6,590
		MIA	13	2,163	190	8,197	0	1,800

Table 30: Results for number of operations [3, 15] and due date tightness [15, 20]

Resource utilization	Number of jos	Stage	Iteration	Tardiness time (Bucket)	Earliness time (Bucket)	Total lead time (Bucket)	Number of overloaded buckets	Computational time (seconds)
[25%, 95%]	487	Initial solution	0	0	0	4,385	16	0
		OIA	26	0	73	4,410	0	885
		MIA	14	0	58	4,427	0	1,800
[45%, 95%]	562	Initial solution	0	0	0	4,961	25	0
		OIA	49	115	136	5,098	0	1,019
		MIA	24	66	130	5,166	0	1,800
[65%, 95%]	624	Initial solution	0	0	0	5,648	39	0
		OIA	105	433	172	6,126	0	2,934
		MIA	15	385	194	6,049	0	1,800
[75%, 95%]	671	Initial solution	0	0	0	5,859	43	0
		OIA	140	837	173	6,704	0	3,543
		MIA	9	737	254	6,562	0	1,800
[85%, 95%]	712	Initial solution	0	0	0	6,415	77	0
		OIA	185	1,364	111	7,909	0	5,537
		MIA	11	1,214	202	7,772	0	1,800
[75%, 110%]	718	Initial solution	0	0	0	6,359	70	0
		OIA	195	1,374	121	7,616	0	5,529
		MIA	14	1,266	166	7,527	0	1,800
[55%, 140%]	796	Initial solution	0	0	0	7,115	122	0
		OIA	279	2,562	135	9,533	0	8,539
		MIA	10	2,398	221	9,401	0	1,800

Table 31: Results for number of operations [3, 15] and due date tightness [15, 25]

Resource utilization	Number of jos	Stage	Iteration	Tardiness time (Bucket)	Earliness time (Bucket)	Total lead time (Bucket)	Number of overloaded buckets	Computational time (seconds)
[25%, 95%]	496	Initial solution	0	0	0	4,317	15	0
		OIA	39	12	118	4,542	0	715
		MIA	9	7	66	4,604	0	1,800
[45%, 95%]	564	Initial solution	0	0	0	4,932	27	0
		OIA	87	197	135	5,495	0	1,863
		MIA	15	142	173	5,502	0	1,800
[65%, 95%]	673	Initial solution	0	0	0	6,011	83	0
		OIA	195	1,046	289	7,321	0	5,168
		MIA	13	968	296	7,320	0	1,800
[75%, 95%]	672	Initial solution	0	0	0	6,052	57	0
		OIA	160	836	179	7,027	0	6,697
		MIA	14	764	210	7,005	0	1,800
[85%, 95%]	715	Initial solution	0	0	0	6,519	104	0
		OIA	240	1,292	173	8,353	0	7,691
		MIA	11	1,145	268	8,255	0	1,800
[75%, 110%]	726	Initial solution	0	0	0	6,463	93	0
		OIA	238	1,413	125	8,442	0	6,947
		MIA	11	1,326	189	8,276	0	1,800
[55%, 140%]	794	Initial solution	0	0	0	7,152	132	0
		OIA	334	2,393	135	10,007	0	10,232
		MIA	9	2,270	199	9,864	0	1,800

Table 32: Results for number of operations [3, 15] and due date tightness [15, 30]

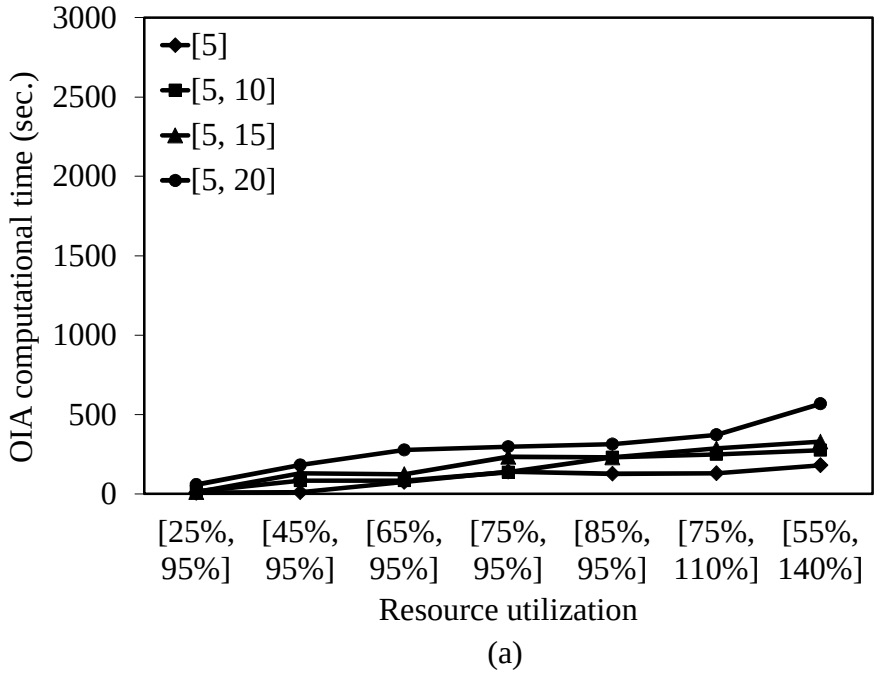
Resource utilization	Number of jos	Stage	Iteration	Tardiness time (Bucket)	Earliness time (Bucket)	Total lead time (Bucket)	Number of overloaded buckets	Computational time (seconds)
[25%, 95%]	458	Initial solution	0	0	0	4,059	21	0
		OIA	38	7	209	4,230	0	767
		MIA	18	4	88	4,348	0	1,800
[45%, 95%]	566	Initial solution	0	0	0	5,108	50	0
		OIA	138	240	254	6,117	0	3,081
		MIA	20	169	244	6,226	0	1,800
[65%, 95%]	656	Initial solution	0	0	0	5,842	90	0
		OIA	216	833	323	7,716	0	5,525
		MIA	16	755	255	7,839	0	1,800
[75%, 95%]	683	Initial solution	0	0	0	6,091	103	0
		OIA	252	1,020	303	7,879	0	6,677
		MIA	13	924	328	7,879	0	1,800
[85%, 95%]	713	Initial solution	0	0	0	6,466	122	0
		OIA	303	1,442	301	8,789	0	9,702
		MIA	14	1,347	314	8,829	0	1,800
[75%, 110%]	733	Initial solution	0	0	0	6,651	131	0
		OIA	339	1,501	348	9,158	0	9,873
		MIA	8	1,402	425	9,086	0	1,800
[55%, 140%]	786	Initial solution	0	0	0	7,092	156	0
		OIA	359	2,285	330	10,036	0	11,263
		MIA	9	2,152	428	9,897	0	1,800

4.6.1 Effects of variability on algorithm performance

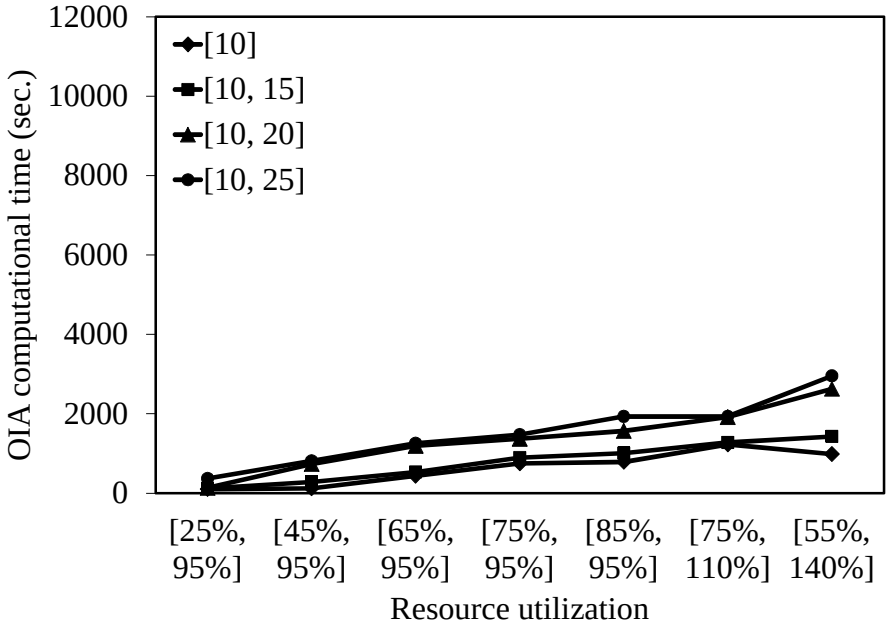
From the results above, the execution performance of both algorithms, OIA and MIA, when dealing with the variability in the system is investigated and discussed.

(1) OIA performance

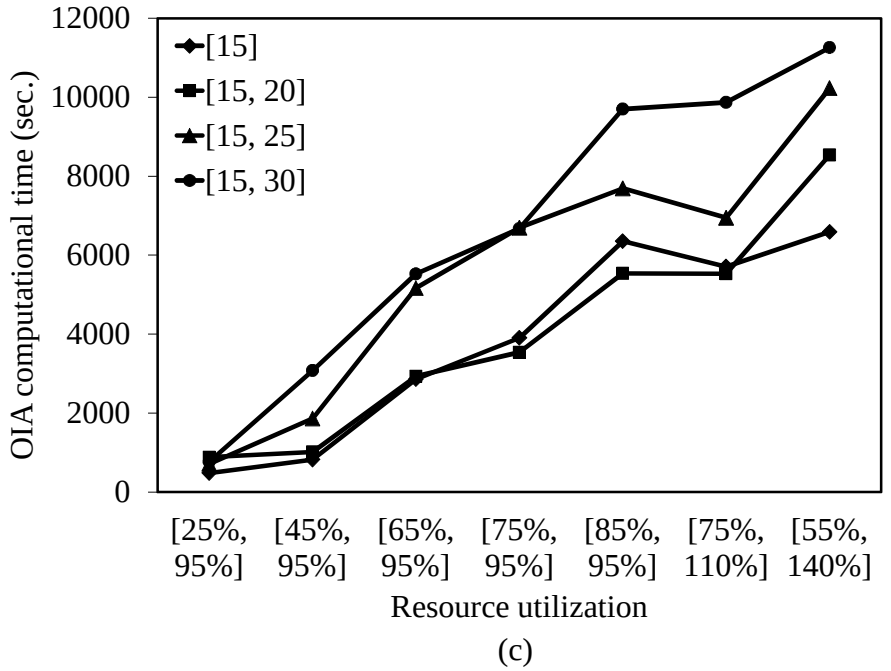
The computational times required to improve infeasible solutions to be feasible solutions are observed. Figure 20 displays the graphs of computational time at different number of operations. Each graph presents the times at varied due date tightness and resource utilization. The results show that an increasing of computational time depends on an incremental range of three combined parameters, number of operations, due date tightness and resource utilization. The largest effect of the solution comes from the number of operations parameter. When the range of number of operations becomes large, it reflects a longer process time and then affects a greater computational time to solve a feasible resource plan.



(a)



(b)



(c)

Figure 20: OIA computational times at number of operations; (a) [3, 5], (b) [3, 10], and (c) [3, 15]

In Figure 21, the graph plots the average computational times at different groups of number of operations. Obviously, the computational times are dramatically increased when process flow is longer. Resource utilization is also a significant factor which drops the performance of solution when it is increased. The congestion in the system leads the solver to require more execution time to search and obtain the feasible solution. The least solution impact is the results of the due date tightness parameter. It can be seen that the different ranges of due date tightness insignificantly reflect the computational time in most instances. However, when the instances involve all combined variabilities such as the problem with number of operations [3, 15] and resource utilization above [45%, 95%], the solver apparently needs more time to solve the solution.

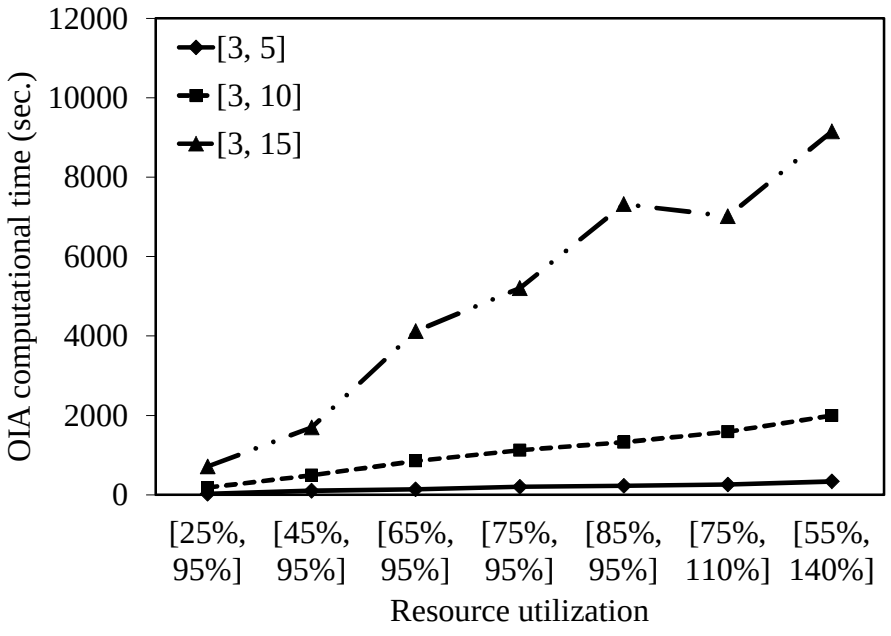


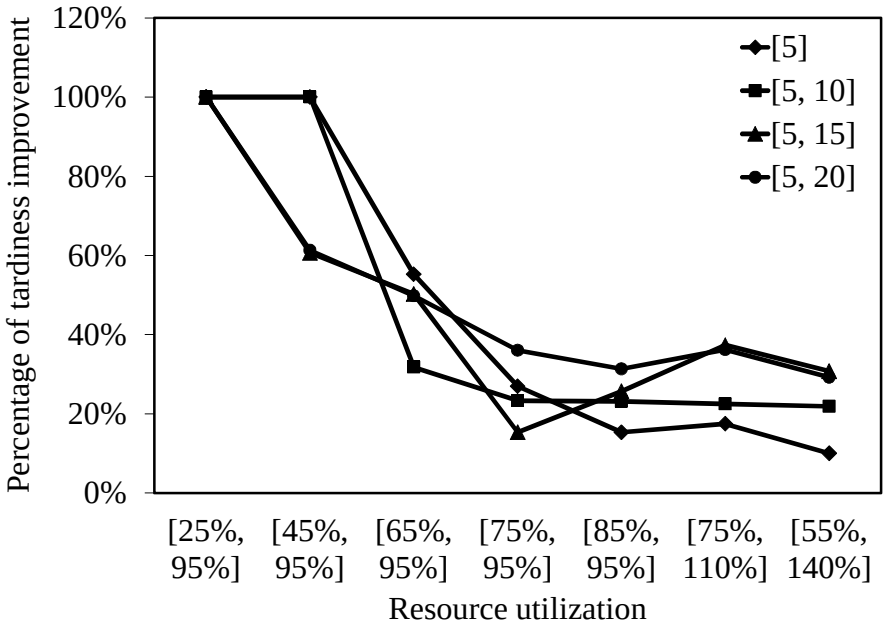
Figure 21: OIA Computational time comparison

Table 33: Percentage of tardiness improvement in MIA

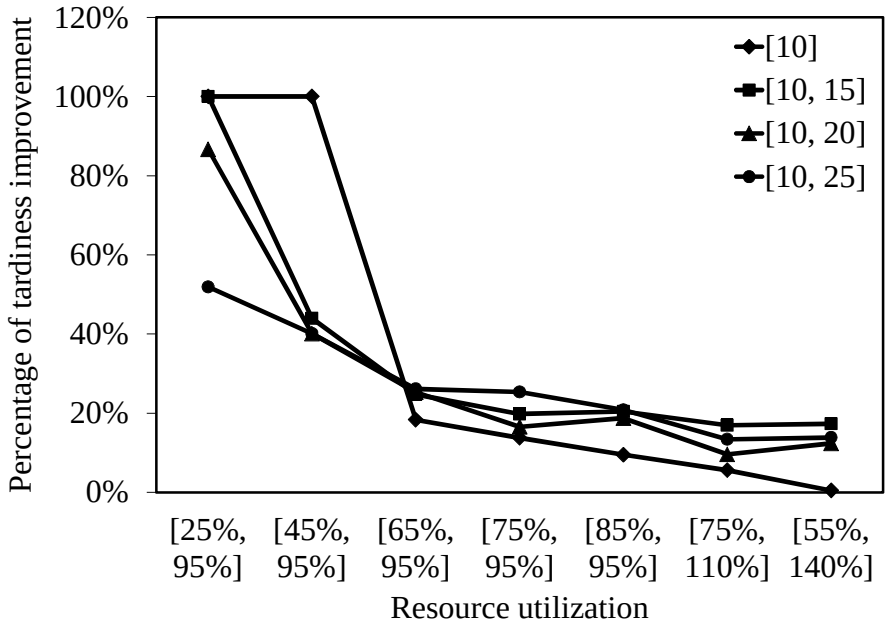
Number of operations	Due date tightness	Resource utilization						
		[25%, 95%]	[45%, 95%]	[65%, 95%]	[75%, 95%]	[85%, 95%]	[75%, 110%]	[55%, 140%]
[3, 5]	[5]	100%	100%	55%	27%	15%	17%	10%
	[5, 10]	100%	100%	32%	23%	23%	22%	22%
	[5, 15]	100%	61%	50%	15%	26%	37%	31%
	[5, 20]	100%	61%	50%	36%	31%	36%	29%
[3, 10]	[10]	100%	100%	18%	14%	10%	6%	0%
	[10, 15]	100%	44%	25%	20%	20%	17%	17%
	[10, 20]	87%	40%	25%	17%	19%	10%	12%
	[10, 25]	52%	40%	26%	25%	21%	13%	14%
[3, 15]	[15]	100%	52%	15%	1%	7%	6%	7%
	[15, 20]	100%	43%	11%	12%	11%	8%	6%
	[15, 25]	42%	28%	7%	9%	11%	6%	5%
	[15, 30]	43%	30%	9%	9%	7%	7%	6%

(2) MIA performance

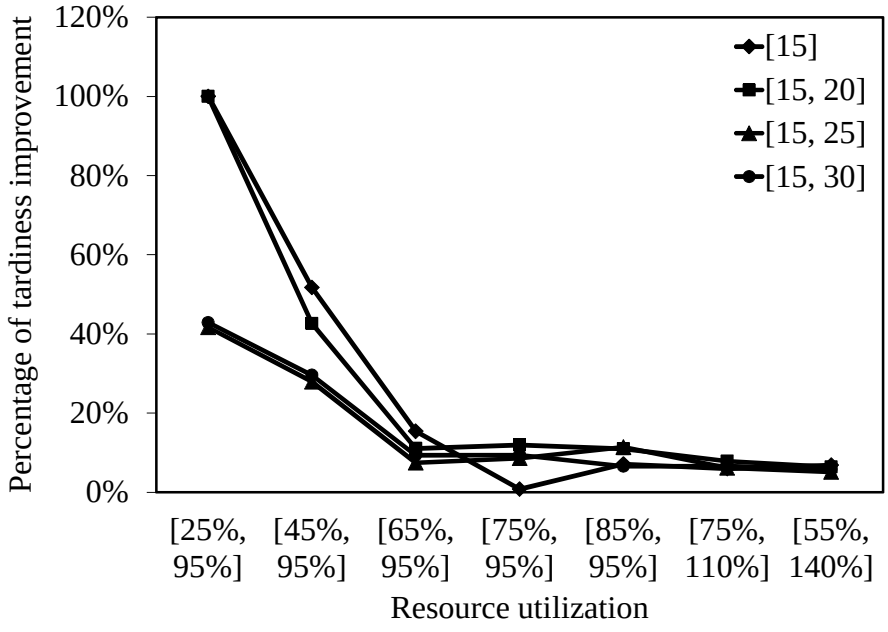
In the section, the efficiency of MIA, such as the percentage of tardiness improvement, at different sizes of number of operation is observed. Table 33 shows the results summary in each scenario. Meanwhile, Figure 22 presents the graphs of these results. The percentage of tardiness improvement is determined from the tardiness deviation between the OIA and MIA outputs. As a result, the smallest size of the problem like number of operations [3, 5] seems easier to be improved the tardiness than the other cases of number of operations, number of operation [3, 10] and [3, 15]. The longer process and dependent machine requirements increase the difficulty of problem in finding an efficient planning, especially when the production system is busy. Thus, the interaction of number of operation and resource utilization is a major impact parameter that reflects the solver performance to improve the planning solution.



(a)



(b)



(c)

Figure 22: Percentage of tardiness improvement at number of operations; (a) [3, 5], (b) [3, 10], and (c) [3, 15]

Figure 23 summarizes the percentage of tardiness improvement at different number of operations. Tardiness improvement is mostly influenced by complicated problems. A graph shows that MIA performs well at low resource utilization from [25%, 95%] to [65%, 95%]. This is because low congestion allows jobs to move around and search for a better solution. Meanwhile, for higher resource utilization, a tight capacity limits the search space. When there is not much room for improvement, the output turns out with a low percent improvement. The percentage of the improvement also highly depends on the number of operations and resource utilization factors. The percentage of tardiness improvement is dropped because of longer number of operations and higher resource utilization.

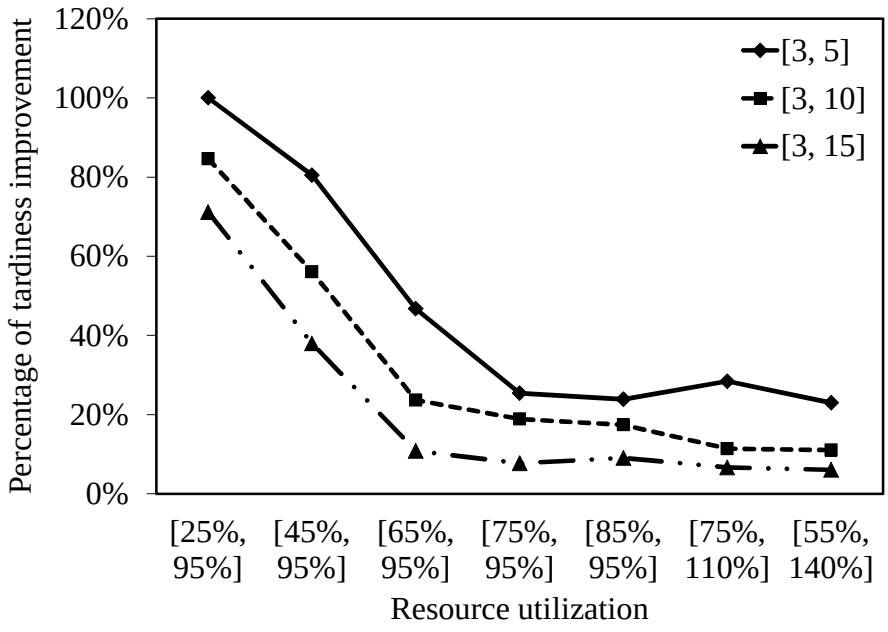


Figure 23: Percentage of tardiness improvement at different number of operations

4.6.2 A comparative study of heuristic approaches

This experiment is an algorithm comparative study with the dispatching rules, FIFO and EDD. In FIFO, jobs will be forwardly processed as early as the resources are available. While in EDD, jobs will be initially sorted from the earliest due date to the latest due date and then loaded in the plan with backward planning method. Both methods consider the planning procedure as a finite capacity. Recall that total cost is weighted with the ratio 30:50:20 for earliness: tardiness: lead time. Tables 34-36 present the total weighted cost results among three solving methods at different number of operations. Figure 24 graphically presents a total weighted cost comparison of the solving approaches at distinguished resource utilization in each number of operations case. The results illustrate that the tabu search algorithm outperforms all instances of the other approaches. Tabu search effectively provides the lowest total costs. With all averaged outputs, tabu search has 49% lower expected total weighted cost than FIFO and 59% lower than EDD. In the variability impact, the total weighted cost has the same result trend as the previous experiment in which a greater level of variability, particularly from high variability of due date tightness and resource utilization, will drop the quality of solution.

Table 34: Total weighted cost comparison results at number of operations [3, 5]

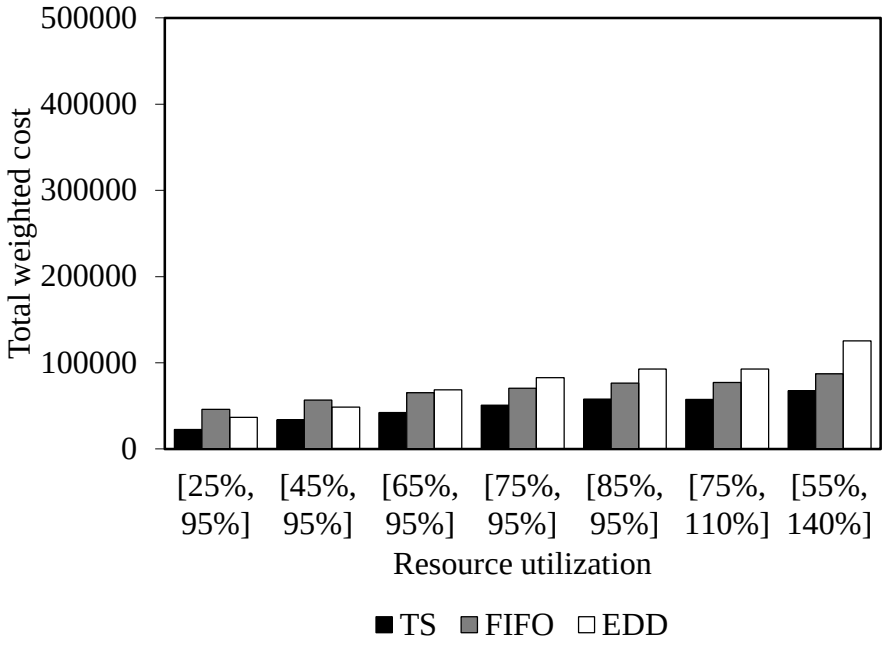
Due date tightness	Resource utilization	Number of jobs	Total weighted cost		
			TS	FIFO	EDD
[5]	[25%, 95%]	230	18,830	25,140	18,880
	[45%, 95%]	304	24,790	33,430	24,850
	[65%, 95%]	350	32,510	40,030	38,280
	[75%, 95%]	379	42,480	46,530	52,290
	[85%, 95%]	402	50,880	53,600	63,690
	[75%, 110%]	389	42,570	49,620	47,950
	[55%, 140%]	410	53,750	60,010	66,830
[5, 10]	[25%, 95%]	287	21,070	42,780	33,460
	[45%, 95%]	306	27,360	53,090	43,550
	[65%, 95%]	354	37,830	59,180	59,400
	[75%, 95%]	384	43,110	64,920	73,210
	[85%, 95%]	405	49,730	68,660	78,690
	[75%, 110%]	411	51,750	70,520	86,970
	[55%, 140%]	405	58,620	75,830	94,520
[5, 15]	[25%, 95%]	225	19,170	50,980	39,390
	[45%, 95%]	318	37,090	64,390	63,690
	[65%, 95%]	351	44,630	74,940	78,880
	[75%, 95%]	386	53,450	80,050	92,060
	[85%, 95%]	404	61,450	89,230	107,090
	[75%, 110%]	409	65,540	90,170	115,740
	[55%, 140%]	415	70,320	97,370	190,650
[5, 20]	[25%, 95%]	269	31,310	64,510	54,830
	[45%, 95%]	324	44,930	75,880	62,430
	[65%, 95%]	358	52,980	86,620	98,390
	[75%, 95%]	388	64,260	91,020	112,620
	[85%, 95%]	394	68,900	94,640	121,150
	[75%, 110%]	406	70,250	98,440	120,120
	[55%, 140%]	433	86,510	114,520	149,890

Table 35: Total weighted cost comparison results at number of operations [3, 10]

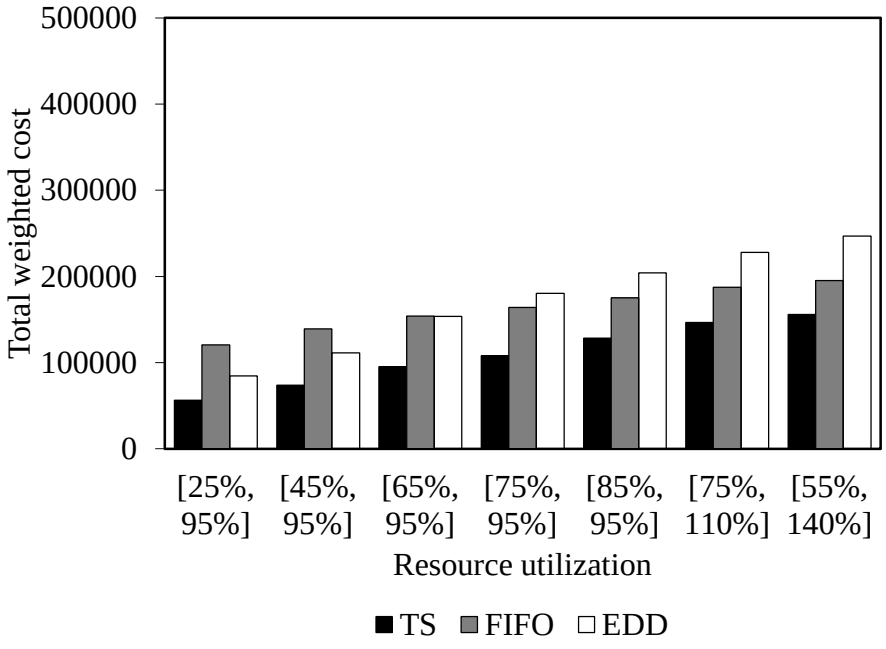
Due date tightness	Resource utilization	Number of jobs	Total weighted cost		
			TS	FIFO	EDD
[10]	[25%, 95%]	390	51,970	90,600	52,440
	[45%, 95%]	437	57,630	101,650	58,140
	[65%, 95%]	486	79,410	112,820	96,430
	[75%, 95%]	528	103,190	125,680	143,880
	[85%, 95%]	554	118,470	136,890	163,230
	[75%, 110%]	580	149,510	153,970	193,450
	[55%, 140%]	560	132,090	146,030	183,500
[10, 15]	[25%, 95%]	365	50,340	108,480	71,750
	[45%, 95%]	442	65,750	130,990	92,760
	[65%, 95%]	490	82,710	143,000	141,500
	[75%, 95%]	535	95,230	155,110	160,830
	[85%, 95%]	547	112,650	158,070	167,800
	[75%, 110%]	574	127,060	172,300	203,860
	[55%, 140%]	578	140,150	177,370	217,130
[10, 20]	[25%, 95%]	391	54,310	134,700	91,750
	[45%, 95%]	459	85,000	159,460	142,480
	[65%, 95%]	512	105,860	173,880	175,690
	[75%, 95%]	522	108,380	176,140	187,960
	[85%, 95%]	554	134,500	189,100	226,940
	[75%, 110%]	586	150,600	203,700	240,600
	[55%, 140%]	617	171,460	221,660	277,160
[10, 25]	[25%, 95%]	386	67,880	149,340	122,190
	[45%, 95%]	442	86,310	164,460	152,740
	[65%, 95%]	493	114,050	186,680	201,450
	[75%, 95%]	531	125,980	199,180	229,860
	[85%, 95%]	560	147,570	216,400	259,070
	[75%, 110%]	574	159,490	220,130	273,970
	[55%, 140%]	599	180,340	235,880	309,340

Table 36: Total weighted cost comparison results at number of operations [3, 15]

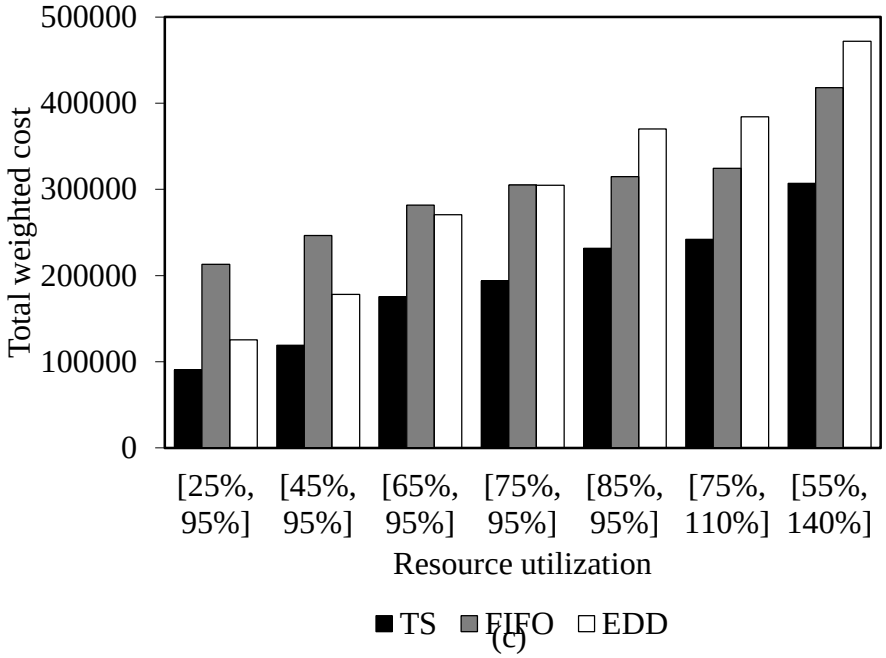
Due date tightness	Resource utilization	Number of jobs	Total weighted cost		
			TS	FIFO	EDD
[15]	[25%, 95%]	477	88,050	169,830	88,560
	[45%, 95%]	544	103,290	192,120	114,470
	[65%, 95%]	641	150,660	222,760	186,610
	[75%, 95%]	679	201,150	293,120	274,600
	[85%, 95%]	716	221,160	263,470	318,710
	[75%, 110%]	735	247,090	282,170	354,310
	[55%, 140%]	756	277,790	305,220	397,910
[15, 20]	[25%, 95%]	487	90,280	206,400	116,680
	[45%, 95%]	562	110,520	233,930	154,040
	[65%, 95%]	624	146,050	254,870	212,390
	[75%, 95%]	671	175,710	275,280	274,580
	[85%, 95%]	712	222,200	299,720	357,460
	[75%, 110%]	718	218,820	297,380	350,900
	[55%, 140%]	796	314,550	562,130	483,510
[15, 25]	[25%, 95%]	496	94,410	240,050	146,130
	[45%, 95%]	564	122,330	267,740	194,920
	[65%, 95%]	673	203,680	311,360	329,310
	[75%, 95%]	672	184,600	303,600	299,390
	[85%, 95%]	715	230,390	337,300	377,100
	[75%, 110%]	726	237,490	341,190	387,680
	[55%, 140%]	794	316,750	387,050	487,030
[15, 30]	[25%, 95%]	458	89,800	235,250	150,320
	[45%, 95%]	566	140,290	292,740	249,320
	[65%, 95%]	656	202,180	338,570	354,540
	[75%, 95%]	683	213,620	349,110	370,580
	[85%, 95%]	713	253,350	359,150	427,150
	[75%, 110%]	733	264,570	378,050	444,790
	[55%, 140%]	786	318,380	417,630	519,940


500000
400000
300000
200000
100000
0
Total weighted cost
[25%, 95%]
[45%, 95%]
[65%, 95%]
[75%, 95%]
[85%, 95%]
[75%, 110%]
[55%, 140%]
Resource utilization
TS
FIFO
EDD

(a)


500000
400000
300000
200000
100000
0
Total weighted cost
[25%, 95%]
[45%, 95%]
[65%, 95%]
[75%, 95%]
[85%, 95%]
[75%, 110%]
[55%, 140%]
Resource utilization
TS
FIFO
EDD

(b)



(c)

Figure 24: Heuristic comparisons at number of operations (a) [3, 5]; (b) [3, 10]; and (c) [3, 15]

4.7 Summary

This study proposes a new tabu search algorithm approach to solve resource planning problems. To provide efficient planning, the planning algorithm based on the JIT philosophy is developed in which the algorithm's objective is to improve earliness, tardiness, and lead time in the system. The proposed algorithm is divided into two sub-algorithms: the overloading improvement algorithm (OIA) and the makespan improvement algorithm (MIA). The algorithm procedure begins with initializing a solution as an ideal solution by using the backward scheduling method. Since the initial plan would not be feasible due to overload capacity, OIA is first used to improve these overloaded capacities. After the resource plan becomes feasible, MIA is implemented to improve earliness, tardiness, and lead time. The neighborhood searching procedure of these two algorithms is created by using the pull and push methods coordinated with the

latest possible start time (LPST) factor, in order to develop a resource plan to come closest to an ideal plan.

The algorithm is tested on varied scales of planning problems. The experiments show that the algorithms effectively perform in the combinatorial problems of the variabilities of due date tightness, number of operations, and resource utilization. The obtained results illustrate the efficiency of the tabu search algorithm in terms of solution quality, both total weighted cost and percentage of solution improvement. Tabu search works well in the infeasible solution improvement and the tardiness and lead time improvement when the instances have small and medium sizes with less and medium variability of input data. However, when the problem conditions have more variability, such as larger range of due date tightness or greater resource utilization, difficulty to manage and allocate jobs to process on the available resources is increased. It also hardly obtains a good planning result within a limited time. In addition, the performance of tabu search over the other heuristic methods, such as FIFO and EDD, is examined. The results show that the tabu search algorithm outperforms these two heuristic approaches. In the following chapter, the study continues to investigate a comparative study of the optimization and tabu search approaches.

CHAPTER 5

A comparative study of the optimization and tabu search approaches

5.1 Introduction

In the previous chapter, the tabu search approach has been studied and compared with the other heuristic methods. The output presented that tabu search can provide the better results of the objective values in all cases. In this chapter, a benchmark study is presented to further examine the performance of the tabu search approach. The comparison between the tabu search and optimization approaches aims to investigate a capability of the optimizer to generate resource plans. According to the limitation of the optimization approach, testing problems will be created with easier and simpler assumptions by excluding some variation of the studied parameters in order to reduce the complexity of the problem instances. Next, the numerical experiments and computational results of this study are presented.

5.2 Computational experiments

This experiment will use the same parameters as in the previous tabu search study that the problem will involve with data variability from due date tightness and resource variability. However, to maintain a lesser difficulty level of testing problems, a variability of the number of operations will be fixed. All parameters of this experiment are summarized as shown in Table 37.

Table 37: Summary of experimental parameters

Parameters	Level of factors	
	Number of operations = 3	**Number of operations = 5**
Due date tightness	3	5
	[3, 6]	[5, 10]
	[3, 9]	[5, 15]
	[3, 12]	[5, 20]
Resource utilization	[25%, 95%]	
	[45%, 95%]	
	[65%, 95%]	
	[75%, 95%]	
	[85%, 95%]	
	[75%, 110%]	
	[55%, 140%]	
Available machines	10	
Planning bucket	25	
Weighted cost ($/unit)		
Tardiness cost	50	
Earliness cost	30	
Lead time cost	20	
Overloading cost	100	

The experiment attempts to observe the impact of solution performance when the number of operations for all jobs is fixed at three operations and five operations while the available machines are 10 machines. This means that each job can be randomly selected to process on 10 different types of machine. In the solving procedure, the tabu search problem instances are solved by using the tabu search algorithm which is implemented by the Java language and executed on a personal computer with a 1.73 GHz processor. For the optimization approach, ILOG CPLEX 12.0 is implemented to

solve an optimal solution of the BILP model. The computational times of both methods are limited at 1,800 seconds. A total cost and an optimality gap are used as a performance indicator to evaluate the solution quality.

5.3 Results and analysis

In this section, the results comparison between the tabu search (TS) and BILP optimization approaches are presented at different due date tightness and resource utilization. Table 38 reports total weighted cost and computational time where the number of operations is three. The results show that TS yields reasonable results that are close to the BILP output. The BILP method seemingly provides better solutions than the TS method. However, it appears only when the problem has less variabilities of due date tightness and resource utilization. It can be seen that TS can perform better than BILP when the instances involve a high level of the variabilities from due date tightness and resource utilization. An interaction of these variabilities reduces an ability of BILP to reach the optimal solution within the limited computational time.

To clarify more about the efficiency of the TS method, an extension instance is generated by increasing the number of operations to five operations per job. Table 39 illustrates the performance results for fixed number of operations at five. In this case, the quality of the TS solutions is often higher than the BILP results after 1,800 sec. It is obvious that TS can solve the instances that BILP does not solve.

Table 38: Comparative results for constant number of operation at 3

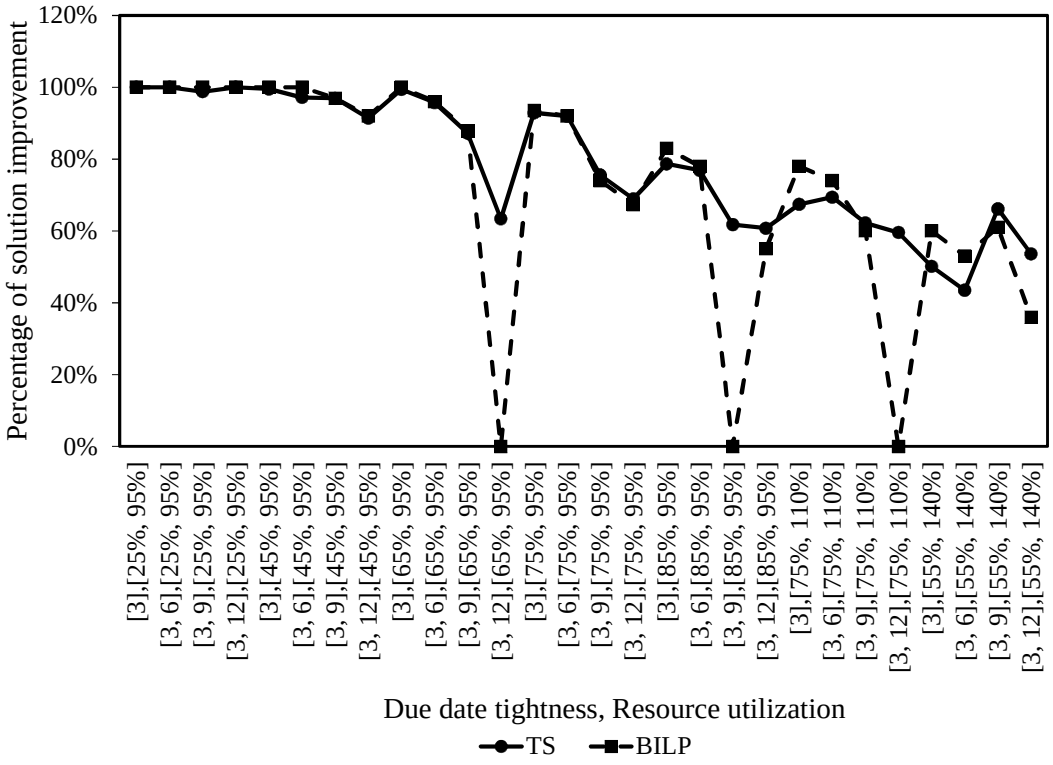
Duedate tightness	Resource utilization	Total weighted cost		Computaitonal time (sec)	
		TS	BILP	TS	BILP
[3]	[25%, 95%]	15,560 *	15,600	1,800	75
	[45%, 95%]	20,250	20,150	1,800	1,092
	[65%, 95%]	23,120	22,980	1,800	1,730
	[75%, 95%]	28,070	25,539	1,800	1,800
	[85%, 95%]	37,210	29,660	1,800	1,800
	[75%, 110%]	52,270	35,280	1,800	1,800
	[55%, 140%]	64,710	51,910	1,800	1,800
[3, 6]	[25%, 95%]	14,900	15,050	1,800	35
	[45%, 95%]	20,390	19,810	1,800	1,459
	[65%, 95%]	25,260	23,740	1,800	1,800
	[75%, 95%]	25,320	25,240	1,800	1,800
	[85%, 95%]	35,370	33,643	1,800	1,800
	[75%, 110%]	42,110	35,800	1,800	1,800
	[55%, 140%]	68,650	57,090	1,800	1,800
[3, 9]	[25%, 95%]	15,400	15,210	1,800	84
	[45%, 95%]	18,870	18,410	1,800	1,800
	[65%, 95%]	26,430	25,220	1,800	1,800
	[75%, 95%]	31,600 *	33,600	1,800	1,800
	[85%, 95%]	35,750 *	46,990	1,800	1,800
	[75%, 110%]	40,950 *	43,390	1,800	1,800
	[55%, 140%]	37,500 *	43,200	1,800	1,800
[3, 12]	[25%, 95%]	16,880	16,440	1,800	156
	[45%, 95%]	22,120	20,630	1,800	1,800
	[65%, 95%]	34,700 *	41,230	1,800	1,800
	[75%, 95%]	35,320 *	37,129	1,800	1,800
	[85%, 95%]	41,430 *	47,540	1,800	1,800
	[75%, 110%]	43,960 *	514,940	1,800	1,800
	[55%, 140%]	65,740 *	90,760	1,800	1,800

* TS provided a better result than BILP.

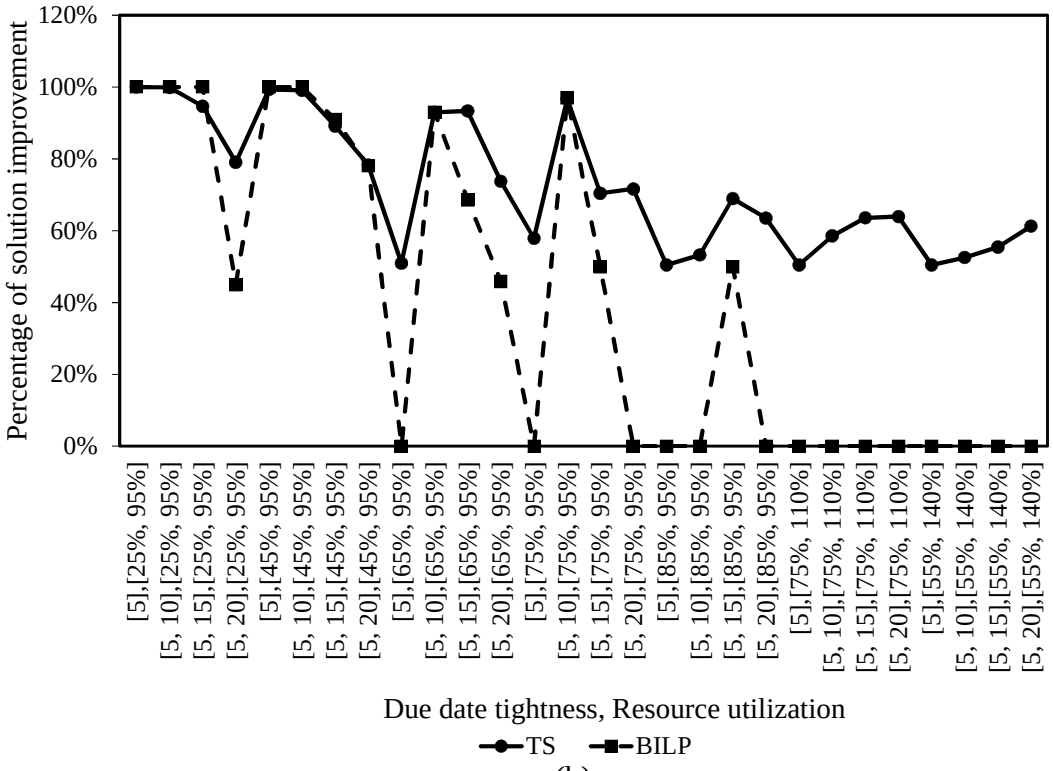
Table 39: Comparative results for constant number of operation at 5

Duedate tightness	Resource utilization	Total weighted cost		Computaitonal time (sec)	
		TS	BILP	TS	BILP
[5]	[25%, 95%]	25,300 *	25,660	1,800	103
	[45%, 95%]	33,750	33,510	1,800	714
	[65%, 95%]	48,850 *	18,280,200	1,800	1,800
	[75%, 95%]	45,700 *	1,363,030	1,800	1,800
	[85%, 95%]	63,790 *	3,080,300	1,800	1,800
	[75%, 110%]	69,260 *	26,748,100	1,800	1,800
	[55%, 140%]	86,190 *	44,502,000	1,800	1,800
[5, 10]	[25%, 95%]	26,850	26,810	1,800	355
	[45%, 95%]	30,930	30,650	1,800	1,111
	[65%, 95%]	37,680	37,420	1,800	1,800
	[75%, 95%]	48,110	44,200	1,800	1,800
	[85%, 95%]	59,060 *	585,390	1,800	1,800
	[75%, 110%]	55,460 *	37,779,200	1,800	1,800
	[55%, 140%]	53,350 *	558,410	1,800	1,800
[5, 15]	[25%, 95%]	36,660	34,690	1,800	1,590
	[45%, 95%]	36,470	35,720	1,800	1,800
	[65%, 95%]	50,130 *	68,220	1,800	1,800
	[75%, 95%]	58,500 *	82,390	1,800	1,800
	[85%, 95%]	62,420 *	86,070	1,800	1,800
	[75%, 110%]	70,920 *	71,186,300	1,800	1,800
	[55%, 140%]	101,100 *	2,962,360	1,800	1,800
[5, 20]	[25%, 95%]	45,100 *	75,240	1,800	100
	[45%, 95%]	40,400 *	40,560	1,800	1,800
	[65%, 95%]	56,840 *	88,250	1,800	1,800
	[75%, 95%]	63,350 *	1,199,810	1,800	1,800
	[85%, 95%]	74,410 *	96,409,500	1,800	1,800
	[75%, 110%]	83,050 *	112,144,600	1,800	1,800
	[55%, 140%]	82,260 *	110,819,200	1,800	1,800

* TS provided a better result than BILP.



(a)



(b)

Figure 25: Percentage of improvement at number of operations; (a) 3 and (b) 5

To visualize the method performance in terms of the ability for determining solutions, Figure 25 illustrates the percentage of solution improvement at two scenarios of the number of operations. The percentage of solution improvement can be derived from the reverse of optimality gap. Each graph plots the optimality gap of the TS and BILP methods at different levels of due date tightness and resource utilization. From these graphs, some observations can be summarized as follows.

i) At the number of operations 3 in Figure 25(a), the solution improvement between two solving methods is not significantly different in most cases. Even though BILP performs better than TS especially at the smaller range of due date tightness and lower resource utilization problem conditions, a greater variability of due date tightness and resource utilization, for instance when due date tightness has a wider range up to [3, 12] and resource utilization is greater than [65%, 95%] or an average of 80%, drops the capability of the BILP's optimizer in solving a good solution. According to the complexity of problem, the optimization approach like BILP cannot provide a good solution as expected. The heuristic approach tends to be more promising method to find an efficient resource plan.

ii) At the enlarged problem, Figure 25(b) presents the percentage of solution when the number of operations is five. The graph shows that the percentage of solution improvement of these two solving methods is significantly different when the problem has an interaction with high variability parameters. TS can provide a greater percentage of solution improvement than BILP when the resource utilization is increased above [75%, 95%] or an average of 85% for all due date tightness cases. Based on the experiments, they can be concluded that the tabu search approach with relatively large

problem instances effectively obtains reasonable result plans when dealing with the combinatorial variabilities.

5.4 Summary

In this chapter, the benchmark study of the quality of resource plans between the tabu search and optimization approaches is presented. The results show the effect of solution improvement rates in which the interaction of variability between number of operations, due date tightness and resource utilization mainly reflect the solver performance. As a result, the tabu search method provides a lower percentage of improvement than the optimization approach when the planning instances have small problem sizes and less variabilities. However, the results of these two methods are not significantly different. On the other hand, at the larger problem size and greater range of due date tightness and resource utilization, the tabu search algorithm outperforms the optimization approach to obtain an efficient resource plan.

CHAPTER 6

Resource planning application

6.1 Introduction

In this chapter, the new resource planning application is presented. This planning application is developed to generate a resource plan based on the JIT philosophy by using the tabu search algorithm. It is implemented as one of the solving algorithms in a CONPLAN which is a concurrent both discrete and continuous events simulation system developed by Dr. Scott Moses, chair of the dissertation committee. The proposed method is considered in a part of a discrete event system which events occur instantly in specific periods of time. The application is created by using the Java language in which it is divided into several components based on the functions of each element of the tabu search procedure. The details of each component of the planning application are described in the next section.

6.2 The components of the planning application

The planning application consists of three main components which include solution initialization stage, Overloading Improvement Algorithm (OIA) stage, and Makespan Improvement Algorithm (MIA) stage. Each main component has several subcomponents which represents a function in the tabu search algorithm. The summary of all java source files, both the main component and subcomponent parts, is presented in Table 40. The function explanations of each component are described as follows. The java code of all source files can be seen in Appendix A.

Table 40: Java source files for the resource planning application

Main function	Sub function
1. initTabu.java	
2. tabuSearch.java	2.1 localSearch.java 2.2 tabuBucketedHeap.java 2.3 evaluate.java 2.4 loadingRequirement.java 2.5 updateTabu.java
3. tabuSearchMIA.java	3.1 targetJob.java 3.2 localSearchMIA.java 3.3 evaluateMIA.java 3.4 loadingRequirementMIA.java 3.5 updateTabuMIA.java

6.2.1 Solution initialization stage

As mentioned before, this application is embedded in the CONPLAN system. Some input data and basic functions of the CONPLAN will be used in the application. For instance, the input data of job number, job amount, release date, due date, routing, and process time. The required data will be read from several input data text files (.txt) in the CONPLAN and then they will be converted to the desired data matrices for implementing in the algorithm's calculation. Figure 26 presents the example of the customer demand data file. Some data from this file, such as job name, amount, routing type, job arrival time, and job due time, will be used. Figure 27 presents the example of the routing data which consists of machine requirement and process time. The summary of required data and the samples of the data are shown in Table 41.

```
order_12340,1,1,2,0,A,Item_E1,Item_E1,Item_E1,2009,09,02,08,00,00,2009
,10,08,08,00,00,2009,10,08,08,00,00,2009,10,08,08,00,00,2009,10,08,08,
00,00,2009,10,08,08,00,00,2009,10,08,08,00,00,100,100,100,0,,Facility1
,truck,truck,truck,0.0
```

```
order_12341,1,1,2,0,A,Item_E15,Item_E15,Item_E15,2009,09,03,08,00,00,2
009,10,12,08,00,00,2009,10,12,08,00,00,2009,10,12,08,00,00,2009,10,12,
08,00,00,2009,10,12,08,00,00,2009,10,12,08,00,00,100,100,100,0,,Facili
ty1,truck,truck,truck,0.0
order_12342,1,1,2,0,A,Item_E15,Item_E15,Item_E15,2009,09,08,08,00,00,2
009,10,13,08,00,00,2009,10,13,08,00,00,2009,10,13,08,00,00,2009,10,13,
08,00,00,2009,10,13,08,00,00,2009,10,13,08,00,00,100,100,100,0,,Facili
ty1,truck,truck,truck,0.0
order_12343,1,1,2,0,A,Item_E16,Item_E16,Item_E16,2009,09,11,08,00,00,2
009,10,13,08,00,00,2009,10,13,08,00,00,2009,10,13,08,00,00,2009,10,13,
08,00,00,2009,10,13,08,00,00,2009,10,13,08,00,00,100,100,100,0,,Facili
ty1,truck,truck,truck,0.0
order_12344,1,1,2,0,A,Item_E10,Item_E10,Item_E10,2009,09,12,08,00,00,2
009,10,09,08,00,00,2009,10,09,08,00,00,2009,10,09,08,00,00,2009,10,09,
08,00,00,2009,10,09,08,00,00,2009,10,09,08,00,00,100,100,100,0,,Facili
ty1,truck,truck,truck,0.0
```

Figure 26: Example of customer demand data (salesOrder.txt)

```
Routing_E1,Facility1,production,,,0,O-
01,M1,false,1,0.0,0,0,0,,611,0,0.0,0,0.0,null,0.0,0,0.0,0.0,0.0,0.0
Routing_E1,Facility1,production,,,0,O-
02,M8,false,1,0.0,0,0,0,,601,0,0.0,0,0.0,null,0.0,0,0.0,0.0,0.0,0.0
Routing_E1,Facility1,production,,,0,O-
03,M3,false,1,0.0,0,0,0,,616,0,0.0,0,0.0,null,0.0,0,0.0,0.0,0.0,0.0
Routing_E1,Facility1,production,,,0,O-
04,M16,false,1,0.0,0,0,0,,603,0,0.0,0,0.0,null,0.0,0,0.0,0.0,0.0,0.0
Routing_E2,Facility1,production,,,0,O-
05,M15,false,1,0.0,0,0,0,,601,0,0.0,0,0.0,null,0.0,0,0.0,0.0,0.0,0.0
Routing_E2,Facility1,production,,,0,O-
06,M8,false,1,0.0,0,0,0,,589,0,0.0,0,0.0,null,0.0,0,0.0,0.0,0.0,0.0
Routing_E2,Facility1,production,,,0,O-
07,M14,false,1,0.0,0,0,0,,605,0,0.0,0,0.0,null,0.0,0,0.0,0.0,0.0,0.0
Routing_E2,Facility1,production,,,0,O-
08,M11,false,1,0.0,0,0,0,,597,0,0.0,0,0.0,null,0.0,0,0.0,0.0,0.0,0.0
```

Figure 27: Example of routing data (routing.txt)

Table 41: Sample input data for the resource planning application

Data list	Sample data
salesOrder	order_12340
releaseDate	2010,09,02,08,00,00
plannedDueDate	2010,10,08,08,00,00
amount	100
routitng	[M1, M8, M3, M16]
processTime	[611, 601, 616, 603]

6.2.2 Overloading improvement algorithm (OIA) stage

The objective of the OIA function is to improve the initial solution from overloaded capacities. The explanation of the OIA concept to find the feasible solution is described in section 4.4.2. The java source file of OIA is separated to six source files based on the functions of tabu search, such as searching neighborhoods, evaluating new solutions, and updating the best solution and other parameters. These files include tabusearch.java, localSearch.java, tabuBucketedHeap.java, evaluate.java, loadingRequirement.java, and updateTabu.java. The function of each source file is described in Table 42.

Table 42: Summary of functions in OIA

Source file	Function
tabuSearch.java	It is a main function that is used to call the other sub functions. The procedure starts from initializing a solution to obtaining a new feasible solution without overloaded capacity.
localSearch.java	Seeking neighborhood solutions to find a new best solution.
tabuBucketedHeap.java	It is a sub function in localSearch.java in which the function is used to determine a new operated time of each operation of each job.
evaluate.java	Evaluating the performance of solutions in terms of tardiness and lead time.
loadingRequirement.java	Determining loading requirements per resource and bucket. This also includes defining overloaded capacity buckets that need to be improved.
updateTabu.java	Updating tabu data, such as tabu job and movement postions, after receiving a new best solution.

6.2.3 Makespan improvement algorithm (MIA) stage

After OIA improves the solutions to be feasible, the next improvement is to use MIA to improve the solution from earliness, tardiness, and lead time. The details of the MIA concept can be seen in section 4.4.2. In this stage, there consist of six source files, which are tabuSearchMIA.java, targetJob.java, localSearchMIA.java, evaluateMIA.java, loadingRequirementMIA.java, and updateTabuMIA.java. The function of each source file is summarized in Table 43.

Table 43: Summary of functions in MIA

Source file	Function
tabuSearchMIA.java	It is a main function that is used to call the other sub functions. The procedure starts from calling the initial solution from OIA and improving the solution until meeting the stopping criteria.
targetJob.java	Identifying a target job which reperesents the job which needs to be improved in the first priority due to maximum tardiness or earliness.
localSearchMIA.java	Seeking neighborhood solutions to find a new best solution.
evaluateMIA.java	Evaluating the performance of solutions in terms of earliness, tardiness and lead time.
loadingRequirementMIA.java	Determining loading requirements per resource and bucket.
updateTabuMIA.java	Updating tabu data, such as tabu job and movement postions, after receiving a new best solution.

A small instance for the application running can be seen in Appendix B. It will present an instance of how to connect the proposed planning application with CONPLAN and some examples of input data and planning results.

CHAPTER 7

Conclusions and suggestions

7.1 Conclusions

In this dissertation, the solving approaches for resource planning problems in MTO environments are studied. The purpose of study is to create an efficient resource plan with embedded the JIT philosophy. The basic concept of the JIT concept is that jobs will be processed when they are required. This leads to reduce WIP, finished goods inventory, job lead time and also to increase flexibility of the production system to accept new orders. In a real manufacturing system, production involves variability, both from outside and inside the system, in which they significantly reduce an ability to manage resources in effective ways. The dissertation proposes two new solving methods, which are the optimization approach or the tabu search heuristic approach, to solve a resource planning problem. The JIT concept is applied as a fundamental concept to develop the algorithms and enhance performance of resource planning.

In Chapter three, a new binary integer linear programing model for resource planning is presented. The solution initialization approach with JIT is proposed to start the calculation since a good starting solution is expected to guide the optimizer to desired solutions. The goal of the model is to minimize total weighted costs from earliness, tardiness, lead time, subcontracting capacity, extra resources capacity, and unplanned jobs. The effect of factors of interest, such as weighted cost ratio, due date tightness, resource utilization, and process time, is examined. The experiments

demonstrate that the optimization algorithm struggles to obtain optimal solutions for instances as variability of data increases and the size of instances increases, where size is the number of jobs and operations per job being planned. However, using the JIT-based initial solution improves performance of the optimizer. It allows optimal solutions to be obtained for moderately larger instances.

In Chapter four, a new planning algorithm coordinated with the heuristic method, called tabu search, is introduced. The algorithm is formulated by adapting the JIT concept into core procedures of tabu search, search space and neighborhood structure, in order to improve the quality of planning solutions in terms of earliness, tardiness, and lead time. The experiments investigate the impact of factors of interest, such as number of operations, due date tightness, and resource utilization. Furthermore, the benchmarking results of the tabu algorithm and the other heuristic methods, FIFO and EDD, are examined. The results illustrate that tabu algorithm outperforms the other two methods. It can provide good solutions though dealing with combinatorial problems of variability and scalability.

In Chapter five, the comparative study between the tabu search approach and the optimization approach is presented. The numerical studies examine the performance of these two solving methods by measuring the performance indicators, including computational time, optimality gap, and objective solution. The analysis illustrates that the tabu search solutions are not quite different from the optimization solutions when the instances have small size and less variability. But the results of tabu search are significantly better than another method when the size of problems and the range of variability are greater.

In Chapter six, the details of resource planning application are presented. The application is coded with the Java language. The architecture of the application is created and categorized based on the basic functions of tabu search: solution initialization, local search, evaluation, and tabu update. The function explanations of each source file are also presented in this chapter.

Up until now, the contribution of this dissertation can be concluded as follows. (1) The resource planning model for job shop planning problems is developed. An effective initial solution can improve the performance of the optimizer since it obtains good optimal solutions within reasonable time in specific problem conditions, such as small problem sizes and less variabilities. The model is useful for planning problem instances which need to analyze and decide what is an efficient method to support customer requirements either managing existing resources or using additional capacities. (2) The new two-phase resource planning algorithm that embeds JIT concepts into a tabu search procedure is proposed, which are overloading improvement algorithm (OIA) and makespan improvement algorithm (MIA). The proposed algorithm obviously provides a better solution than the optimization problem when problems involve scalability and variability. In addition, tabu search solutions also insignificantly differ from the optimal solutions by the optimization approach in the small problem instances. (3) With variability concerned, the tabu search method can provide more promising solutions than the optimization approach and the heuristic approaches, FIFO and EDD, since the tabu search algorithm, based on the JIT concept, can explore and attempt to move solutions to ideal solution spaces. It therefore turns out with a lesser

computational time and optimality gap in the optimization benchmark analysis and a better objective value in the other heuristic methods analysis.

7.2 Future study

Tactical-level planning does not allow consecutive operations for a job to be processed in the same time bucket. Each operation can be loaded in a single bucket only. From this condition, the future work can expand the study by applying tabu search with the JIT concept to the operational capacity planning like weekly or daily planning. This means that consecutive operations for a job can be processed next to each other. With the concept of producing the right job at the right time, the operational plan might be improved more on smoothing production flow and reducing WIP. So that is an interesting topic for investigation.

Furthermore, this dissertation studies the results of resource planning from the heuristic methods, tabu search, FIFO, and EDD. In order to differentiate the planning results, other heuristic methods such as genetic algorithms, ant colony optimization, and particle swarm optimization might be a good alternative method to investigate numerical results and performance of resource planning.

References

Armentano, V.A. and Scrich, C.R. (2000), Tabu search for minimizing total tardiness in a job shop, *International Journal Production Economics*, vol. 63, pp. 131-140.

Baker, K.R. and Scudder, G.D. (1990), Sequencing with earliness and tardiness penalties: a review, *Operations Research*, vol. 38, pp. 22-36.

Balas, E. and Vazacopoulos, A. (1998), Guided local search with shifting bottleneck for job shop scheduling. *Management Science*, vol. 44, no. 2, pp.262-275.

Barnes, J.W. and Chambers, J.B. (1995), Solving the job shop scheduling problem using tabu search, *IIE Transactions*, vol. 27, pp. 257-263.

Beck, J.C. (1994), *A Schema for Constraint Relaxation with Instantiations for Partial Constraint Satisfaction and Schedule Optimization*, Master thesis, Department of Computer Science, University of Toronto.

Bertrand, J.W.M. and Sridharan, V. (2001), A study of simple rules for subcontracting in make-to-order manufacturing, *European Journal of Operational Research*, vol. 128, pp. 509-531.

Carlier, J. (1982), The one-machine sequencing problem, *European Journal of Operational Research*, vol. 11, pp. 42-47.

Clausen, J. (1999), *Branch and bound algorithm-Principles and Examples*, Department of Computer Science, University of Copenhagen, Denmark.

Corry, P. and Kozan, E. (2004), Job scheduling with technical constraints, *Journal of the operational research society*, vol. 55, pp. 160-169.

Danna, E., Rothberg, E., and Le, Pape C. (2003), Integrating mixed integer programming and local search: A case study on job shop scheduling problems, *Proceedings CPAIOR*.

Danna, E., Rothberg, E., and Le, Pape C. (2004), Exploring relaxation induced neighborhoods to improve MIP solutions, *Springer-Verlag*, 10.1007/s10107-004-0518-7.

Dell'Amico, M. and Trubian, M. (1993), Applying tabu search to the job shop scheduling problem, *Annals of Operations Research*, vol. 41, pp. 231-252.

Eswaramurthy, V.P. and Tamilarasi, A. (2009), Hybridizing tabu search with ant colony optimization for solving job shop scheduling problems, *International Journal of Advanced Manufacturing Technology*, vol. 40, pp. 1004-1015.

Fisher, M. L. (1973), Optimal solution of scheduling problems using Lagrange multipliers: Part I, *Operations Research*, vol. 21, pp. 1114-1127.

Fisher, M.L., Lageweg, B.J., and Lenstra, J.K. (1983), Surrogate duality relaxation for job shop scheduling, *Discrete Applied Mathematics*, vol. 5, pp. 65-75.

Fischetti, M. and Lodi, A. (2003), Local branching, *Mathematical Programming*, vol. 98, pp. 23–47.

Fischetti, M., Glover, F., and Lodi, A. (2005), The feasibility pump, *Mathematical Programming*, vol. 104, pp. 91-104.

Frederix, F. (2000), An extended enterprise planning methodology for the discrete manufacturing industry, *European Journal of Operational Research*, vol. 129, pp. 317-325.

Gendreau, M. (2002), *An introduction to tabu search*, [Online] Available at: http://opim.wharton.upenn.edu/~sok/papers/g/Gendreau_ANINTRODUCTIONTOTABUSEARCH.pdf

Giebels, M.M.T., Hans, E.W., Gerritsen, M.P.H., and Kals, H.J.J. (2000), Capacity planning for make- or engineer-to-order manufacturing; the importance of order classification, *33rd CIRP Manufacturing Systems Conference*, Stockholm.

Glover, E. (1986), Future paths for integer programming and links to artificial intelligence, *Computers and Operations Research*, vol. 13, pp. 533-549.

Guoa, Y.W., Lib, W.D., Milehama, A.R., and Owena, G.W. (2009), Optimization of integrated process planning and scheduling using a particle swarm optimization approach, *International Journal of Production Research*, vol. 47, no. 14, pp. 3775-3796.

Hans, E.W. (2001), *Resource loading by branch-and-price techniques*, Ph.D. thesis, University of Twente, Netherlands.

He, Z., Yang, T., and Deal, D.E. (1993), A multiple-pass heuristic rule for job shop scheduling with due dates, *International Journal of Production Research*, vol. 31, pp. 2677-2692.

Heyl, J. (2010). *Linear programming powerpoint slides for Operations Management, by Krajewski/Ritzman/Malhotra*, Pearson Education.

Hopp, W. and Spearman, M. (2004), *Factory Physics*, 2nd edn, McGraw-Hill/Irwin.

Imanipour, N. and Zegordi, S.H. (2006), A heuristic approach based on tabu search for early/tardy flexible job shop problems, *Scientia Iranica*, vol. 13, no. 1, pp. 1-13.

Jain, A.S. and Meeran, S. (1999), Deterministic job shop scheduling: past, present and future, *European Journal of Operational Research*, vol. 113, pp. 390-434.

James, R.J.W. (1997), Using Tabu search to solve the common due date early/tardy machine scheduling problem, *Computers and Operations Research*, vol. 24, no. 3, pp. 199-208.

Kamien, M.I. and Li, L. (1990), Subcontracting coordination flexibility and production smoothing in aggregate planning, *Management Science*, vol. 36, pp. 1352-1363.

Lawler, E.L., Lenstra, J.K., Rinnooy Kan, A.H.G., and Shmoys, D.B. (1989), *Sequencing and scheduling: Algorithm and Complexity*, Report BS-R89xx, Centrum voor Wiskunde en Informatica, Amsterdam, Netherlands.

Li, W.D. and Mcmahon, C.A. (2007), A simulated annealing-based optimization approach for integrated process planning and scheduling, *International Journal of Computer Integrated Manufacturing*, vol. 20, pp. 80-95.

Merzifonluoglu, Y., Geunes, J., and Romeijn, H.E. (2006), Integrated capacity, demand, and production planning with subcontracting and overtime options, *Naval Research Logistics*, vol. 54, pp. 433-447.

Miyashita, K. (1997), *Iterative constraint-based repair for multiagent scheduling*, AAAI Technical report WS-97-05.

Nowicki, E. and Smutnicki, C. (1996), A fast taboo search algorithm for the job shop problem, *Management Science*, vol. 42, no. 6, pp. 797-813.

Potts, C.N. (1980), Analysis of a heuristic for one machine sequencing with release dates and delivery times, *Operations Research*, vol. 28, pp. 1436-1441.

Rabadi, G., Mollghasemi, M., and Anagnostopoulos, C. G. (2004), A branch and bound algorithm for the early/tardy machine scheduling problem with a common due-date and sequence dependent setup time, *Computers and Operations Research*, vol. 31, pp. 1727-1751.

Revelle, J.B. (2001), *Manufacturing handbook of best practices: An innovation, productivity and quality focus*, CRC Press, Florida.

Sourd, F. and Kedad-Sidhoum, S. (2003), The one-machine problem with earliness and tardiness penalties, *Journal of Scheduling*, vol. 6, no. 6, pp. 533-549.

Taillard, E. (1989), Parallel taboo search technique for the job shop scheduling problem, *Working Paper ORWP*, Departement de Mathematiques, Ecole Polytechnique Federale De Lausanne, Lausanne, Switzerland.

Tanaka, S., Sasaki, T., and Araki, M. (2003), A Branch and bound algorithm for the single machine weighted earliness-tardiness scheduling problem with job independent weights, *IEEE*, 0-7803-7952-7/03.

Tsubakitani, S. and Evans, J. R. (1992), *Applying tabu search to the mean tardiness sequencing problem*, University of Cincinnati.

Van Laarhoven, P.J.M., Aarts, E.H.L., and Lenstra, J.K. (1992), Job shop scheduling by simulated annealing, *Operation Research*, vol. 40, no. 1, pp. 113-125.

Vaessens, R.J.M., Aarts, E.H.L., and Lenstra, J.K. (1996), Job shop scheduling by local search, *INFORMS Journal on Computing*, vol. 8, pp. 302-317.

Wullink, G., Hans, E.W., and Harten, A. V. (2004a), *Robust resource loading for engineer-to-order manufacturing*, Beta Research School for Operations, Management and Logistics, University of Twente, Netherlands.

Wullink, G., Gademann, A.J.R.M., Hans, E.W., and Harten, A. V. (2004b), Scenario-based approach for flexible resource loading under uncertainty, *International Journal of Production Research*, vol. 42, no. 24, pp. 5079-5098.

Zhang, C.Y., Li, P., Guan, Z., and Rao, Y. (2007), A tabu search algorithm with a new neighborhood structure for the job shop scheduling problem, *Computers and Operations Research*, vol. 34, pp. 3229-3242.

Zhang, C.Y., Rao, Y., and Li, P. (2008), An effective hybrid genetic algorithm for the job shop scheduling problem, *International Journal of Advanced Manufacturing Technology*, vol. 39, pp. 965-974.

Zribi, N., Kacem, I., El-kamel, A., and Borne, P. (2008), *Minimizing the total tardiness in a flexible job-shop*, Ecole Centrale de Lille, University of technology of Troyes, France.

Zhu, ZC., Ng, KM., and Ong, HL. (2010), A modified tabu search algorithm for cost-based job shop problem, *Journal of the Operational Research Society*, vol. 61, pp. 611 –619.

Appendix A

Java code for the resource planning application

1. initTabu.java

```
public class initTabu extends tabuSearch
{
public void initData(ArrayList conplanInitList, ArrayList conplanOrderList) {
   try{
     Database database = Database.getDatabase( Database.PLANNING );
     ConfigReaderWriter config = new ConfigReaderWriter( "conf/index.conf" );
     bucketSize = Integer.parseInt( config.get( "BucketedHeap.bucketSize" ));
     depth = Integer.parseInt( config.get( "BucketedHeap.depth" ));
     bucketedHeap = new BucketedHeap(depth, bucketSize);
     baseIndex = bucketedHeap.buckets();
     List<SalesOrder> sales = (List<SalesOrder>) database.getTable(
     "SalesOrder" ).getAll(); //unchecked cast
     for( Iterator<SalesOrder> i = sales.iterator(); i.hasNext(); ) {
        SalesOrder so = (SalesOrder) i.next();
        plannedDueDate.add(so.getShipDateRequested());
        release.add(so.getArrivalTime());
        amount.add(so.getQuantityRequested());
        List<ItemBOMRouting> ItemRoutings = (List<ItemBOMRouting>)
        databse.getTable( "ItemBOMRouting" ).getAll( new Where( "item",
        so.getItemRequested() ) ); //unchecked cast
        for( Iterator<ItemBOMRouting> j = ItemRoutings.iterator();
        j.hasNext(); ) {
            ItemBOMRouting ibr = (ItemBOMRouting) j.next();
            List<Routing> routings = (List<Routing>) database.getTable(
            "Routing" ).getAll( new Where( "name", ibr.getRoutingName() ) );
            //unchecked cast
            ArrayList mc = new ArrayList();
            ArrayList pt = new ArrayList();
            ArrayList initNode = new ArrayList();
            String name = so.getNumber();
            int t=0;
            for(int d = 0; d<conplanOrderList.size(); d++) {
               String init = (String) conplanOrderList.get(d);
               if(name.equals(init)&&t==0){
                  ArrayList conplanList = (ArrayList) conplanInitList.get(d);
                  ++t;
                  int g = 1;
                  for(Iterator<Routing> k = routings.iterator(); k.hasNext();)
                  {
                     int opnIndex = Integer.parseInt(conplanList.get(g-
                     1).toString())+baseIndex;
                     Routing r = (Routing) k.next();
                     initNode.add(opnIndex);
                     mc.add(r.getResource());
                     pt.add(r.getUnitRuntime());
                     ++g;
                  }
               }
```

```
            }
            machine.add(mc);
            processTime.add(pt);
            PST.add(initNode);
         }
      }
      List<Resource> rs = (List<Resource>) database.getTable( "Resource"
      ).getAll(); //unchecked cast
      for( Iterator<Resource> r = rs.iterator(); r.hasNext(); ) {
         Resource m = (Resource) r.next();
         machineName.add(m.getName());
      }
      //set tabu list size
      if(PST.size()>=200){
         tabuListSize = (int)(PST.size()/32);
      }
      else{
         tabuListSize = 7;
      }
      }
      catch( java.io.FileNotFoundException notfound ) {
      System.err.println( "Failed to open conf/index.conf file" );
      }
   }
}
```

2. tabuSearch.java

```
public class tabuSearch {
    public int timeOffset = 1253923200;
    public static int iteration = 1000;
    public static int runTime=1800;
    public static int terminatedIteration = 30;
    public static int terminatedMIA = 10;

    static ArrayList amount = new ArrayList();
    static ArrayList bestList = new ArrayList();
    static ArrayList candOperatedList = new ArrayList();
    static ArrayList candLoadingList = new ArrayList();
    static ArrayList diffTimeList = new ArrayList();
    static ArrayList idealplanList = new ArrayList();
    static ArrayList jobList = new ArrayList();
    static ArrayList machine = new ArrayList();
    static ArrayList machineName = new ArrayList();
    static ArrayList mc = new ArrayList();
    static ArrayList mcList = new ArrayList();
    static ArrayList newPST = new ArrayList();
    static ArrayList nPST = new ArrayList();
    static ArrayList newPSTList = new ArrayList();
    static ArrayList overloadedOrder = new ArrayList();
    static ArrayList overloadedOperation = new ArrayList();
    static ArrayList PST = new ArrayList();
    static ArrayList processTime = new ArrayList();
    static ArrayList plannedDueDate = new ArrayList();
    static ArrayList release = new ArrayList();
    static ArrayList swapPairList = new ArrayList();
    static ArrayList startList = new ArrayList();
    static ArrayList tdvList = new ArrayList();
    public static ArrayList timeList = new ArrayList();
    public static ArrayList indexList = new ArrayList();

    static ArrayList [] resourceLoad;
    static ArrayList [] candResourceLoad;
    static ArrayList [] startResourceLoad;

    public static String maxMachine;
    public static String[] candMaxMachine;

    public static int baseIndex;
    public static int bucketSize;
    public static int bestMakespan;
    public static int bestTardiness;
    public static int bestMaxOverload;
    public static int bestNumOverload;
    public static int bestEarliness;
    public static int countSolution;
    public static int depth;
    public static int deviatedPosCount;
    public static int maxBucket;
    public static int minPoint;
    public static int msCountList;
    public static int MIACount;
    public static int nJob;
    public static int nMachine;
    public static int numElements;
    public static int maxRootBucket;
    public static int planningBucket;
```

```
public static int startTime;
public static int totalBucket;
public static int tabuListSize;

public static int[] candMakespan;
public static int[] candMaxOverload;
public static int[] candNumOverload;
public static int[] candTardiness;
public static int[] candEarliness;
public static int[] candBucketFrom;
public static int[] candBucketTo;
public static int[] candHeap;
public static int[] candMaxBucket;
public static int[] deviatedTime;
public static int[] dueDate;
public static int[] earliness;
public static int[] earlinessList;
public static int[] heap;
public static int[] makespanList;
public static int[] msTardiness;
public static int[] msEarliness;
public static int[] tabuCount;
public static int[] tardiness;
public static int[] tardinessList;
public static int[] jobFrom;
public static int[] jobTo;
public static int[] maxJList;
public static int[] tabuBucketFrom;
public static int[] tabuBucketTo;
public static int[] tabuJ;
public static int[] tabuO;
public static int[][] bestLoading;
public static int[][] candLoading;
public static int[][] loading;
public static int[][] nbhLoading;
public static int[][] startLoading;

public static boolean[] jobCondition;

public static boolean checkBucket;
public static boolean exitCondition;
public static boolean tCheck;

public static BucketedHeap bucketedHeap;

public static void main(ArrayList conplanInitList, ArrayList
conplanOrderList) {
   initTabu it = new initTabu();
   it.initData(conplanInitList, conplanOrderList);
   totalBucket = baseIndex;
   planningBucket = baseIndex;
   nMachine = machineName.size();
   jobFrom = new int [tabuListSize];
   jobTo = new int [tabuListSize];
   maxJList = new int[tabuListSize];
   tabuBucketFrom = new int[tabuListSize];
   tabuBucketTo = new int[tabuListSize];
   tabuJ = new int[tabuListSize];
   tabuO = new int[tabuListSize];
   loading = new int [nMachine][totalBucket];
   nbhLoading = new int [nMachine][totalBucket];
   bestLoading = new int [nMachine][totalBucket];
   candLoading = new int [nMachine][totalBucket];
```

```
startLoading = new int [nMachine][totalBucket];
resourceLoad = new ArrayList [machineName.size()];
candResourceLoad = new ArrayList [machineName.size()];
startResourceLoad = new ArrayList [machineName.size()];
startList = new ArrayList();
jobList = new ArrayList();
bestList = new ArrayList();
idealplanList = new ArrayList();
startTime = (int)(System.currentTimeMillis()/1000);
countSolution = 0;

for (int itn=0; itn<iteration; itn++) {
   //Solution initialization
   if (itn==0) {
   //(1) Get initial solution from CONPLAN
   for (int p1=0; p1<PST.size(); p1++){
      ArrayList jl = (ArrayList) PST.get(p1);
      jobList.add(jl);
      startList.add(jl);
      bestList.add(jl);
      idealplanList.add(jl);
   }

   //(2) Initialize tabu list
   updateTabu tabu = new updateTabu();
   tabu.initList();

   //(3) Evaluate Tardiness and Makespan of initial solution
   evaluate value = new evaluate();
   value.objValue(jobList);
   bestTardiness =  value.totalTardiness;
   bestEarliness = value.totalEarliness;
   bestMakespan = value.makespan;

   //(4) Find overloading in each bucket
   loadingRequirement load = new loadingRequirement();
   load.loadingReq(jobList);
   bestMaxOverload = load.maxOverload;
   bestNumOverload = load.nOverloadedBucket;

   }

   else {
   //(5) Start next iteration with the best solution from previous run
   jobList = new ArrayList();
   for (int p=0; p<startList.size(); p++){
      ArrayList start = (ArrayList) startList.get(p);
      jobList.add(start);
   }
   loadingRequirement load = new loadingRequirement();
   load.loadingReq(jobList);
   }

   //Set a resource heap load
   for(int i=0; i<machineName.size(); i++){
      startResourceLoad[i] = resourceLoad[i];
   }

   //**(6) Skip to MIA
   if(bestMaxOverload == 0){
      tabuSearchMIA ms = new tabuSearchMIA();
      ms.msTabuSearch();
      int OIATime = (int)(System.currentTimeMillis()/1000);
```

```
   break;
}

else {
   //(7) Generate neighborhood candidate
   newPSTList = new ArrayList();
   localSearch candidate = new localSearch();
   candidate.searching();

   //(8) Select the best candidate
   bestCandidate(candidate.nMove);
   int newBestMaxOverload = candMaxOverload[minPoint];
   int newBestNumOverload = candNumOverload[minPoint];
   int newBestTardiness = candTardiness[minPoint];
   int newBestEarliness = candEarliness[minPoint];
   int newBestMakespan = candMakespan[minPoint];


   //(9) Check Tabu list
   boolean tc = tabuCheck(minPoint);

   //(10) update status to start the next iteration
   int caseNumber;

   //Change sign in maxOverload condition from > to <
   //Case1: Get the best answer #1
   if (((tc==true) || (tc==false)) &&
   (bestMaxOverload<newBestMaxOverload)) {
      caseNumber = 1;
      bestMaxOverload = candMaxOverload[minPoint];
      bestNumOverload = candNumOverload[minPoint];
      bestTardiness = candTardiness[minPoint];
      bestEarliness = candEarliness[minPoint];
      bestMakespan = candMakespan[minPoint];
      maxMachine = candMaxMachine[minPoint];
      maxBucket = candMaxBucket[minPoint];

      bestList = (ArrayList) newPSTList.get(minPoint);
      startList = bestList;
   }

   //Case2: Get the best answer #2
   else if (((tc==true) || (tc==false)) &&
   (bestMaxOverload==newBestMaxOverload) &&
   (bestTardiness>newBestTardiness))
   {
      caseNumber = 2;
      bestMaxOverload = candMaxOverload[minPoint];
      bestNumOverload = candNumOverload[minPoint];
      bestTardiness = candTardiness[minPoint];
      bestEarliness = candEarliness[minPoint];
      bestMakespan = candMakespan[minPoint];
      maxMachine = candMaxMachine[minPoint];
      maxBucket = candMaxBucket[minPoint];
      bestList = (ArrayList) newPSTList.get(minPoint);
      startList = bestList;
   }

   //Case3: Get the best answer #2
   else if (((tc==true) || (tc==false)) &&
   (bestMaxOverload==newBestMaxOverload) &&
   (bestTardiness==newBestTardiness)&&(bestMakespan>newBestMakespan)
   )
```

```
{
   caseNumber = 3;
   bestMaxOverload = candMaxOverload[minPoint];
   bestNumOverload = candNumOverload[minPoint];
   bestTardiness = candTardiness[minPoint];
   bestEarliness = candEarliness[minPoint];
   bestMakespan = candMakespan[minPoint];
   maxMachine = candMaxMachine[minPoint];
   maxBucket = candMaxBucket[minPoint];
   bestList = (ArrayList) newPSTList.get(minPoint);
   startList = bestList;
}

//Case4: Some improvement (not Tabu) #1
else if ((tc==false) && (bestMaxOverload>=newBestMaxOverload) &&
(bestTardiness<=newBestTardiness))
{
   caseNumber = 4;
   startList = (ArrayList) newPSTList.get(minPoint);
   maxMachine = candMaxMachine[minPoint];
   maxBucket = candMaxBucket[minPoint];
}

//Case5: Some improvement (not Tabu) #2
else if ((tc==false) && (bestMaxOverload>newBestMaxOverload) &&
(bestTardiness>newBestTardiness))
{
   caseNumber = 5;
   startList = (ArrayList) newPSTList.get(minPoint);
   maxMachine = candMaxMachine[minPoint];
   maxBucket = candMaxBucket[minPoint];
}

//Case6: Some improvement (Tabu)
else {
   caseNumber = 6;
   int maxOverload= bucketSize*totalBucket;
   int minTardiness = bucketSize*totalBucket;
   int minMakespan = bucketSize*totalBucket;
   int maxOverload1;
   int minPoint1;

   for (int m=0; m< candidate.nMove; m++) {
      boolean tc1= tabuCheck(m);
      if ((candMaxOverload[m] < maxOverload) && (tc1==false)) {
         maxOverload = candMaxOverload[m];
         minPoint = m;
      }
      else if ((candMaxOverload[m] == maxOverload) &&
      (tc1==false)) {
         if (candTardiness[m] < minTardiness){
            maxOverload1 = candMaxOverload[m];
            minPoint1 = m;
         }
         else if
         (candTardiness[m]==minTardiness&&candMakespan[m]<minMak
         espan){
            maxOverload1 = candMaxOverload[m];
            minPoint1 = m;
         }
         else {
            maxOverload1 = maxOverload;
            minPoint1 = minPoint;
```

```
                    }
                    maxOverload = maxOverload1;
                    minPoint = minPoint1;
                }
            }

            startList = (ArrayList) newPSTList.get(minPoint);
            maxMachine = candMaxMachine[minPoint];
            maxBucket = candMaxBucket[minPoint];
        }

        //(12) Update tabu list
        updateTabu tabuList = new updateTabu();
        tabuList.updateList(minPoint);

        //(13) Solution summary
        System.out.println("Best solution summary:");
        System.out.println("  bestList= "+bestList);
        System.out.println("  FinalBestMaxOverload ="+bestMaxOverload);
        System.out.println("  FinalBestNumOverload ="+bestNumOverload);
        System.out.println("  FinalBestTardiness ="+bestTardiness);
        System.out.println("  FinalBestEarliness ="+bestEarliness);
        System.out.println("  FinalBestMakespan ="+bestMakespan);
        System.out.println("  Iteration ="+itn);


        //(14) Count the unimproved answers to terminate
        if (caseNumber==4||caseNumber==5||caseNumber==6) {
        countSolution = countSolution + 1;
        }
        else {
        countSolution = 0;
        }
    }
  }

  int endTime = (int)(System.currentTimeMillis()/1000);
  int computationalTime = endTime-startTime;
}//Main class

public static void bestCandidate(int nMove){
//Find a minimum overloadedBucket, tardiness, and earliness
  int minO = bucketSize*totalBucket;
  int minT = bucketSize*totalBucket;
  int minM = bucketSize*totalBucket;
  int minN = bucketSize*totalBucket;

  for (int m=0; m < nMove; m++) {
     if (candMaxOverload[m] < minO) {
        minO = candMaxOverload[m];
        minT = candTardiness[m];
        minM = candMakespan[m];
        minN = candNumOverload[m];
        minPoint = m;
     }
     else if (candMaxOverload[m] == minO) {
        if (candTardiness[m] < minT){
           minO = candMaxOverload[m];
           minT = candTardiness[m];
           minM = candMakespan[m];
           minN = candNumOverload[m];
           minPoint = m;
        }
```

```
            else if (candTardiness[m] == minT) {
            if (candNumOverload[m] < minN){
               minO = candMaxOverload[m];
               minT = candTardiness[m];
               minM = candMakespan[m];
               minN = candNumOverload[m];
               minPoint = m;
            }
            else if (candNumOverload[m] == minN){
            if (candMakespan[m] < minM){
                minO = candMaxOverload[m];
                minT = candTardiness[m];
                minM = candMakespan[m];
                minN = candNumOverload[m];
                minPoint = m;
            }
            }
            }
         }
      }
   }

   public static boolean tabuCheck(int minPoint) {
      //Check Tabu list
      tCheck=false;
      int countTabu = 0;
      for (int t=0; t<tabuListSize; t++) {
         int tabuJob =
         Integer.parseInt(overloadedOrder.get(minPoint).toString());
         int tabuOpn =
         Integer.parseInt(overloadedOperation.get(minPoint).toString());
         if ( (tabuJob==tabuJ[t])&&((tabuOpn==tabuO[t]))){
         ++countTabu;
         }
         if(countTabu>0)
             tCheck = true;
         else
            tCheck = false;
      }
      return tCheck;
   }
}
```

2.1 localSearch.java

```
public class localSearch extends tabuSearch {
    boolean checkTime = true;
    boolean condition;
    boolean pushCondition;
    int nMove;
    int overloadedOpn;
    int newBucket;
    public tabuBucketedHeap bh;

    public void searching() {
        overloadedOrder = new ArrayList();
        overloadedOperation = new ArrayList();

        //Create candidate list from overloadedBucket
        Iterator m = machine.iterator(); m.hasNext();
        for(int c=0; c<machine.size(); c++){
            mc = (ArrayList)m.next();
            for(int d=0; d<mc.size(); d++){
                String mt = ((ArrayList)machine.get(c)).get(d).toString();
                int opnTime =
                Integer.parseInt(((ArrayList)jobList.get(c)).get(d).toString());

                if ((mt.equals(maxMachine))&&(opnTime == maxBucket)) {
                    overloadedOrder.add(c);
                    overloadedOperation.add(d);
                }
            }
        }

        nMove = overloadedOrder.size();
        candMaxOverload = new int [nMove];
        candNumOverload = new int [nMove];
        candTardiness = new int [nMove];
        candEarliness = new int [nMove];
        candMakespan = new int [nMove];
        tabuCount = new int [nMove];
        candBucketFrom = new int [nMove];
        candBucketTo = new int [nMove];
        candMaxMachine = new String [nMove];
        candMaxBucket = new int [nMove];
        bh = new tabuBucketedHeap();

        for(int l=0; l<nMove; l++){
            //Generate Neighborhood
            int ovOrder = Integer.parseInt((overloadedOrder.get(l).toString()));
            int ovOp = Integer.parseInt(overloadedOperation.get(l).toString());
            newSequence(ovOrder, ovOp);

            //Evaluate the solution
            evaluate candValue = new evaluate();
            candValue.objValue(newPST);
            candValue.bucketLocation(l, startList, newPST);
            candBucketFrom[l] = candValue.bucketFrom;
            candBucketTo[l] = candValue.bucketTo;
            candTardiness[l] = candValue.totalTardiness;
            candEarliness[l] = candValue.totalEarliness;
            candMakespan[l] = candValue.makespan;
            loadingRequirement loading = new loadingRequirement();
            loading.updateLoading();
```

```
        loading.overloadedBucket();
        candMaxOverload[l] = loading.maxOverload;
        candNumOverload[l] = loading.nOverloadedBucket;
        candMaxMachine[l] = maxMachine;
        candMaxBucket[l] = maxBucket;
    }
}

public void newSequence(int ovOrder, int ovOpn) {
    //Create new lists of order in each operation
    newPST = new ArrayList();
    Iterator m = machine.iterator(); m.hasNext();
    for(int a=0; a<machine.size(); a++){
        nPST = new ArrayList();
        mc = (ArrayList)m.next();

        if(a==ovOrder){
            overloadedOpn = ovOpn;
            checkPositiveTime(a);
            if(checkTime == true){
                pullBucket(a);
            }
            else if (checkTime == false){
                pushBucket(a);
            }
        }
        else {
            for(int b=0; b<mc.size(); b++){
                newBucket =
                Integer.parseInt(((ArrayList)jobList.get(a)).get(b).toString(
                ));
                nPST.add(newBucket);
        }}
        newPST.add(nPST);
    }
    newPSTList.add(newPST);
}

public void pullBucket(int a) {
    long pet;
    condition = true;
    nPST = new ArrayList();
    pushCondition = false;
    mcList = new ArrayList();

    //(1) search the newBuckets for overloaded operation and predecessor
    operation
    for (int b=0; b<overloadedOpn+1; b++){
        int c = overloadedOpn-b;
        int mcName = 0;
        int length =
        Integer.parseInt(((ArrayList)processTime.get(a)).get(c).toString())*
        Integer.parseInt(amount.get(a).toString());
        int oprBucket =
        Integer.parseInt(((ArrayList)jobList.get(a)).get(c).toString());

        if(c==overloadedOpn){
            pet = timeOffset+bh.endTimeOf(oprBucket);
        }
        else{
            oprBucket = Integer.parseInt((nPST.get(0)).toString())-1;
            pet = timeOffset+bh.endTimeOf(oprBucket);
        }
```

```
        String m = ((ArrayList)machine.get(a)).get(c).toString();
        for(int i=0; i<machineName.size(); i++){
           String mcn = machineName.get(i).toString();
           if(m.equals(mcn)){
               mcName = i;
           }
        }
        startHeap(mcName);
        long pst = bh.findPSTBackward(pet, length);
        int newBucket = (int)Math.ceil(baseIndex + (pst-
        timeOffset)/bucketSize);

        checkCondition(a, c, newBucket);
        if(pushCondition == true){
           pushBucket(a);
           break;
        }

        //update loading
        heap[oprBucket] += length;
        heap[newBucket] -= length;

        //update parent
        bh.updateParents(oprBucket);
        bh.updateParents(newBucket);

        //update heap in adjusted resoruce
        mcList.add(mcName);
        candResourceLoad[mcName] = new ArrayList();

        for( int i = 0; i < heap.length; i++ ){
           candResourceLoad[mcName].add(heap[i]);
        }
        nPST.add(0,newBucket);
     }

     //(2) search the newBuckets for successor Operation
     for (int b=0; b<mc.size()-(overloadedOpn+1); b++){
        int c = b+overloadedOpn+1;
        if((condition=true) && (c>overloadedOpn)){
           newBucket =
           Integer.parseInt(((ArrayList)jobList.get(a)).get(c).toString());
           nPST.add(newBucket);
         }
     }
  }

  public void checkCondition(int job, int opn, int newBucket){
     //Check the bucket whether start earlier than release date. If yes, go
     to pushBucket.
     long r = Integer.parseInt(release.get(job).toString());
     int rd = (int)baseIndex + (int)(r-timeOffset)/bucketSize;
     if(newBucket< baseIndex || newBucket<rd||(newBucket==baseIndex&opn!=0))
     {
        condition = false;
        pushCondition = true;
     }
  }

  public void pushBucket(int a) {
     int mcName = 0;
     nPST = new ArrayList();
```

```
    mcList = new ArrayList();

    for (int b=0; b<mc.size(); b++){
        String m = ((ArrayList)machine.get(a)).get(b).toString();
        //Get heap for machine in opn b
        for(int i=0; i<machineName.size(); i++){
            String mcn = machineName.get(i).toString();
            if(m.equals(mcn)){
                mcName = i;
            }
        }
        startHeap(mcName);

        if(b < overloadedOpn){
            newBucket                                                    =
            Integer.parseInt(((ArrayList)jobList.get(a)).get(b).toString());
        }
        else{
            int length =
            Integer.parseInt(((ArrayList)processTime.get(a)).get(b).toString(
            ))*Integer.parseInt(amount.get(a).toString());
            int oprBucket =
            Integer.parseInt(((ArrayList)jobList.get(a)).get(b).toString());
            long est = timeOffset+bh.endTimeOf(Integer.parseInt((nPST.get(b-
            1)).toString()))+bucketSize;
            long pst = (int) bh.findPSTForward(est, length);
            newBucket = (int)Math.ceil(baseIndex + (pst-
            timeOffset)/bucketSize);

            //update loading
            heap[oprBucket] += length;
            heap[newBucket] -= length;

            //update parent
            bh.updateParents(oprBucket);
            bh.updateParents(newBucket);

            //update heap in adjusted resoruce
            mcList.add(mcName);
            candResourceLoad[mcName] = new ArrayList();

            for( int i = 0; i < heap.length; i++ ){
                candResourceLoad[mcName].add(heap[i]);
            }
        }
        nPST.add(newBucket);
    }
}

public boolean checkPositiveTime(int a) {
    int count = 0;
    for(int b=0; b<mc.size(); b++){
        int processOpn =
        Integer.parseInt(((ArrayList)jobList.get(a)).get(b).toString());
        int rDate = (int)
        (baseIndex+((Integer.parseInt((release.get(a)).toString())-
        timeOffset)/bucketSize));
        if ((processOpn==0) || (processOpn==rDate)) {
            ++count;
        }
        if (count > 1) {
            checkTime = false;
        }
```

```
        }
        return checkTime;
    }

    public void startHeap(int mcName){
        for(int i=0; i<heap.length; i++){
            heap[i] =
            Integer.parseInt(startResourceLoad[mcName].get(i).toString());
        }
    }
}
```

2.2 tabuBucketedHeap.java

```
public class tabuBucketedHeap extends tabuSearch {
    private int maxOverlapNodes;
    private double minNodeAvailability;
    private int bucketPosition;

    public int buckets() {
       return baseIndex(); // half of the nodes are leaf nodes, which is what
       // baseIndex actually calculates
    }

    public long findPSTBackward( long PET, int length ) {
       int time = (int) (PET - timeOffset);
       long PST = findPSTDescendMax( 0, maxRootBucket, time, length );
       if( PST == -1 )
          return -1; // never found capacity
       else
          return PST + timeOffset;
    }

    public long findPSTForward( long EST, int length ) {
       int time = (int) (EST - timeOffset);
       long PST = 0;

       // if space is available
       if( heap[0] > length ) {
          // calculate the PST
          PST = timeOffset + findPSTDescend( 0, maxRootBucket, time, length );
          boolean acceptPST;
          do {
             acceptPST = true;
             List<Integer> overlapNodes = overlap( (int)(PST - timeOffset),
             length, null );

             if( maxOverlapNodes > 0 && overlapNodes.size() > maxOverlapNodes)
             {
                acceptPST = false;
             }

             if( overlapNodes.size() > 2 ) {
                overlapNodes.remove( 0 );
                overlapNodes.remove( overlapNodes.size() - 1 );

             // check to see if each node has adequate availability
             for( Iterator<Integer> i = overlapNodes.iterator(); i.hasNext();
             ) {
                int index = ((Integer) i.next()).intValue();
                if( ((double) heap[index] / (double) bucketSize) <=
                minNodeAvailability )
                                   acceptPST = false;
             }
          }

          // if we do not like the PST (acceptPST is false), move search
          forward
          if( !acceptPST ) {
             time += bucketSize;
             PST = Math.max( timeOffset + findPSTDescend( 0, maxRootBucket,
             time, length ), EST );
          }
```

```
    } while( !acceptPST );

    // now that we have a PST, adjust it's position within the node
    switch( this.bucketPosition ) {
       case 1: PST = Math.max( PST, EST );
       break;
       case 2: PST = PST + bucketSize - length; // end of bucket - length
       break;
       case 3: // already at start of node
       break;
       case 4: PST = PST + bucketSize / 2; // midpoint of node
       break;
       default:
       System.err.println( "Invalid bucketPosition - assuming 3" );
    }
    // our final PST
    return PST;
    } else {
       return timeOffset + maxRootBucket + 1; // return, basically, an
       invalid time
    }
}

public int amountBefore( int index, int PET ) {
    int start = startTimeOf( index );
    int end = endTimeOf( index );
    if( end < PET ){
        return heap[index];
    }
    if( PET < start ){
        return 0;
    }
    else{
        return Math.min( heap[index], PET - start );
    }
}

private int findPSTDescend( int index, int maxBucketSize, int time, int
length ) {
    if( heap[index] <= 0 )
       return maxRootBucket + 1; // not available for consumption
    if( index < baseIndex() ) {
       if( time < startTimeOf( rightChild( index ) ) ) {
           int possiblePST = findPSTDescend( leftChild( index ),
           maxBucketSize / 2, time, length );
           if( possiblePST > maxRootBucket ) {
              return findPSTDescend( rightChild( index ),
                                     maxBucketSize / 2,
                                     time,
                                     length );
           } else {
               return possiblePST;
           }
       } else {
       // only right child eligible
           return findPSTDescend( rightChild( index ), maxBucketSize / 2,
                                  time, length );
       }
    } else {
        // this node has available capacity, is located on the
        // base level, and is far enough to the right
        return (index - baseIndex()) * maxBucketSize;
    }
```

```
}

public int findPSTDescendMax( int index, int maxBucketSize, int PET, int
length ) {
   if( amountBefore( index, PET ) > 0 ) {
      if( index < baseIndex() ) {
         return   Math.max(   findPSTDescendMax(   leftChild(   index   ),
         maxBucketSize / 2, PET, length ), findPSTDescendMax( rightChild(
         index ), maxBucketSize / 2, PET, length ) );
      } else {
         // accumulate to the end point to determine PST
         int amountAfter = 0;
         for( int i = index + 1; i < baseIndex() + (PET/bucketSize); i++ )
         {
            amountAfter += amountBefore( i, PET );
         }
         if( amountAfter + amountBefore( index, PET ) >= length ){
            return Math.min( PET - length, endTimeOf( index ) - ( length
            - amountAfter ) );
         } else
         return -1;
      }
   } else {
       return -1;
   }
}

private void indexOutOfBounds( PlannedOperation op ) {
   RTPDate pst = new RTPDate( op.PST );
   System.err.println( "Attempting to insert operation outside of the
   planning horizon at time " + pst.getDateAs(RTPDate.ISO) + " for order "
   + op.salesOrderID + " operation " + op.id + " with task size " +
   op.operationTime );
}


private List<Integer> overlap( int time, int amount, PlannedOperation op )
{
   List<Integer> updatedNodes = new ArrayList<Integer>();
   int nodeIndex = baseIndex() + time / bucketSize;
   int leftToRemove = amount;

   // go along base and remove capacity
   try {
       while( leftToRemove > 0 ) {
          if( heap[ nodeIndex ] > leftToRemove ) {
              leftToRemove = 0;
          } else {
              leftToRemove -= Math.max( heap[ nodeIndex ], 0 );
          }
          updatedNodes.add( new Integer( nodeIndex ) );

          if( leftToRemove > 0 )
              nodeIndex++;
       }
   } catch( ArrayIndexOutOfBoundsException array ) {
       indexOutOfBounds( op );
   }
   return updatedNodes;
}

private int startTimeOf( int index ) {
   int pos = index;
```

```
        int depthMultiplier = 1;
        int depth = 0;
        if( index == 1 ) {
            depth = 1;
        } else {
           while( pos >= depthMultiplier ) {
              pos -= depthMultiplier;
              depthMultiplier *= 2;
              depth++;
           }
        }
        int bucketSizeForDepth = maxRootBucket / (int) Math.pow( 2, depth );
        int offsetWithinDepth = index - (int) Math.pow( 2, depth ) + 1;
        return bucketSizeForDepth * offsetWithinDepth;
    }

    public int endTimeOf( int index ) {
        int pos = index;
        int depthMultiplier = 1;
        int depth = 0;
        if( index == 1 ) {
              depth = 1;
        } else {
            while( pos >= depthMultiplier ) {
              pos -= depthMultiplier;
              depthMultiplier *= 2;
              depth++;
            }
        }
        int bucketSizeForDepth = maxRootBucket / (int) Math.pow( 2, depth );
        int offsetWithinDepth = index - (int) Math.pow( 2, depth ) + 1;
        return bucketSizeForDepth * (offsetWithinDepth + 1) - 1;
    }

    private int baseIndex() {
        return heap.length / 2;
    }

    public void updateParents( int index ) {
        if( index != 0 ) {
           int parentIndex = parent( index );
           heap[parentIndex ] = heap[ leftChild( parentIndex ) ] + heap[
           rightChild parentIndex ) ];
           updateParents( parentIndex );
        }
    }
    private int rightChild( int index ) {
        return index * 2 + 2;
    }

    private int parent( int index ) {
        if( index == 0 )
            return 0;
        return (index - 1) / 2; // integer division is always 'floored' in Java
    }

    private int leftChild( int index ) {
        return index * 2 + 1;
    }
}
```

2.3 evaluate.java

```
public class evaluate extends tabuSearch {
    int minTime;
    int maxTime;
    int makespan;
    int totalTardiness;
    int totalEarliness;
    int bucketFrom;
    int bucketTo;
    int totalMs = 0;
    int[] startTime = new int[PST.size()];
    int[] finishTime = new int[PST.size()];
    int[] ms = new int[PST.size()];

    public void objValue(ArrayList jobList){
        int[] dueDateBucket = new int [PST.size()];
        int[] tardiness = new int[PST.size()];
        int[] earliness = new int[PST.size()];

        //Find duedate
        int j = 0;
        Iterator d = plannedDueDate.iterator();
        while(d.hasNext()){
            dueDateBucket[j] = baseIndex+(Integer.parseInt(d.next().toString())-
            timeOffset)/bucketSize;
            ++j;
        }

        //Calculate finish time and makespan
        int k = 0;
        int minStartTime = bucketSize;
        int maxFinishTime = 0;
        Iterator pst = jobList.iterator(); pst.hasNext();
        for (int p=0; p<jobList.size(); p++){
            ArrayList pst1 = (ArrayList)pst.next();
            for (int p1=0; p1<pst1.size(); p1++){
                if(p1==(pst1.size()-1)){
                    startTime[k] =
                    Integer.parseInt(((ArrayList)jobList.get(p)).get(0).toString(
                    ));
                    finishTime[k] =
                    Integer.parseInt(((ArrayList)jobList.get(p)).get((pst1.size()
                    -1)).toString());

                    //Calculate earliest start time
                    if(startTime[k]<minStartTime){
                        minStartTime = startTime[k];
                    }

                    //Calculate latest finish time
                    if(finishTime[k]>maxFinishTime){
                        maxFinishTime = finishTime[k];
                    }

                    //Calculate makespan by job
                    ms[p] = (finishTime[p]-startTime[p])+1;
                    totalMs = totalMs + ms[p];
                    ++k;
                }
        }}
```

```
            minTime = minStartTime;
            maxTime = maxFinishTime;
            makespan = totalMs;

            //Calculate total earliness and tardiness
            for (int i=0; i<PST.size(); i++){
                if ((finishTime[i]+1)>=(dueDateBucket[i])){
                   earliness[i] = 0;
                   tardiness[i] = ((finishTime[i]+1)-dueDateBucket[i]);

                }
                else {
                    earliness[i] = (dueDateBucket[i]-(finishTime[i]+1));
                    tardiness[i] = 0;
                }
                totalEarliness = totalEarliness+earliness[i];
                totalTardiness = totalTardiness+tardiness[i];
            }
        }

        public  void  bucketLocation(int  list,  ArrayList  startList,  ArrayList
        jobList) {
            Iterator p = startList.iterator(); p.hasNext();
            for (int i=0; i<startList.size(); i++){
                ArrayList p1 = (ArrayList)p.next();
                for (int j=0; j<p1.size(); j++){
                    int tJob =
                    Integer.parseInt((overloadedOrder.get(list).toString()));
                    int tOpn =
                    Integer.parseInt((overloadedOperation.get(list).toString()));

                    if(i==tJob && j==tOpn) {
                        bucketFrom =
                        Integer.parseInt(((ArrayList)startList.get(i)).get(j).toStrin
                        g());
                        bucketTo =
                        Integer.parseInt(((ArrayList)jobList.get(i)).get(j).toString(
                        ));
                    }
                }
            }
        }
    }
```

2.4 loadingRequirement.java

```
public class loadingRequirement extends tabuSearch {
   int maxOverload;
   int nOverloadedBucket;
   ArrayList heapLoad = new ArrayList();
   ArrayList heapList = new ArrayList();

   public void loadingReq(ArrayList jobList) {
       bucketLoading(jobList);
       overloadedBucket();
   }

   public void bucketLoading(ArrayList jobList) {
       initHeap (depth, bucketSize);
       updateLoad();
   }

   public void initHeap( int depth, int bucketSize ) {
       numElements = (int) Math.pow( 2, depth ) - 1;
       if( numElements <= 0 ) {
          System.err.println( "loadingRequirement: Invalid number of elements,
          probably caused by an invalid depth (negative or overly large)" );
          System.exit( 1 );
       }

       // create the heap
       heap = new int[ numElements ];

       // initialize the elements
       int index = numElements - 1;
       int bucketForLevel = bucketSize;
       int n = 1;
       int elemForLevel = (int) Math.pow( 2, depth - n );

       while( index >= 0 ) {
          for( int i = 0; i < elemForLevel; i++ )
             heap[ index - i ] = bucketForLevel;
             index -= elemForLevel;
             bucketForLevel *= 2;
             n++;
             elemForLevel /= 2;
       }

       maxRootBucket = heap[0];

       for( int j = 0; j < numElements; j++ ){
          heapList.add(heap[j]);
       }
       for( int k = 0; k < machineName.size(); k++ ){
          resourceLoad[k] = heapList;
       }
   }

   private void updateLoad() {
       Iterator j= jobList.iterator(); j.hasNext();
          for(int a=0; a<jobList.size(); a++){
             ArrayList k = (ArrayList)j.next();
             for(int b=0; b<k.size(); b++){
```

```
                int index =
                Integer.parseInt(((ArrayList)jobList.get(a)).get(b).toString(
                ));
                int time = Integer.parseInt(
                (ArrayList)processTime.get(a)).get(b).toString());
                int quantity = Integer.parseInt(amount.get(a).toString());
                String mcID = ((ArrayList)machine.get(a)).get(b).toString();
                int ID = 0;
                for(int c=0; c<machineName.size(); c++){
                   String m = (machineName.get(c)).toString();
                   if(m.equals(mcID)){
                      ID = c;
                   }
                }
                removeCapacity(ID, index, time, quantity);
                updateHeapLoad(ID);
             }
          }
}

private void removeCapacity(int ID, int nodeIndex, int time, int amount) {
    for( int i = 0; i < heap.length; i++ ){
       heap[i] = Integer.parseInt(resourceLoad[ID].get(i).toString());
    }

    int leftToRemove = amount*time;
    // go along base and remove capacity
    while( leftToRemove > 0 ) {
       //Infinite planning
       heap[ nodeIndex ] -= leftToRemove;
       leftToRemove = 0;
       tabuBucketedHeap parent = new tabuBucketedHeap();
       parent.updateParents(nodeIndex);
    }
}

private void updateHeapLoad(int ID){
    heapLoad = new ArrayList();
    for( int i = 0; i < heap.length; i++ ){
       heapLoad.add(heap[i]);
    }
    resourceLoad[ID] = heapLoad;
}

public void overloadedBucket() {
    int overload = 0;
    int count = 0;
    maxOverload = 0;

    for(int i=0; i<resourceLoad.length; i++){
    for(int j=0; j<heap.length; j++){
       heap[j] = Integer.parseInt(resourceLoad[i].get(j).toString());
       if (heap[j] < 0&&j>=(int)heap.length/2){
          overload = heap[j];
          ++count;
       }

       if (overload < maxOverload) {
          maxOverload = overload;
          maxMachine = machineName.get(i).toString();
          maxBucket = j;
       }
    }}
```

```
        nOverloadedBucket = count;
        if (count == 0){
            maxOverload = 0;
         }
    }

    public void updateLoading(){
        for(int i=0; i<mcList.size(); i++){
        for(int j=0; j<machineName.size(); j++){
            int k = Integer.parseInt(mcList.get(i).toString());
            if(j==k)
                resourceLoad[j]=candResourceLoad[j];
            else
                resourceLoad[j]=resourceLoad[j];
        }}
     }

}
```

2.5 updateTabu.java

```
public class updateTabu extends tabuSearch {
   int tJob;
   int tOpn;
   int bucketFrom;
   int bucketTo;

   public void initList() {
      for (int t=0; t<tabuListSize; t++) {
         tabuJ[t] = 0;
         tabuO[t] = 0;
         tabuBucketFrom[t] = 0;
         tabuBucketTo[t] = 0;
      }
   }

   public void updateList(int mPoint) {
      for (int t=0; t<tabuListSize; t++) {
         if (t < tabuListSize-1) {
            tabuJ[t] = tabuJ[t+1];
            tabuO[t] = tabuO[t+1];
            tabuBucketFrom[t] = tabuBucketFrom[t+1];
            tabuBucketTo[t] = tabuBucketTo[t+1];
         }
         else if (t == (tabuListSize-1)){
            for(int l=0; l<overloadedOrder.size(); l++){
               if(l == mPoint){
                  tJob =
                  Integer.parseInt((overloadedOrder.get(l).toString()));
                  tOpn =
                  Integer.parseInt((overloadedOperation.get(l).toString()));
               }
            }
            tabuJ[t] = tJob;
            tabuO[t] = tOpn;
            tabuBucket(tJob, tOpn);
            tabuBucketFrom[t] = bucketFrom;
            tabuBucketTo[t] = bucketTo;
         }
      }
   }

   @SuppressWarnings("unchecked")
   public void tabuBucket(int tJob, int tOpn) {
      Iterator p = startList.iterator(); p.hasNext();
         for (int i=0; i<startList.size(); i++){
            ArrayList p1 = (ArrayList)p.next();
            for (int j=0; j<p1.size(); j++){
               if(i==tJob && j==tOpn) {

                  bucketFrom =
                  Integer.parseInt(((ArrayList)jobList.get(i)).get(j).toStri
                  ng());
                  ArrayList bestCandidateList = (ArrayList)
                  newPSTList.get(minPoint);
                  bucketTo =
                  Integer.parseInt(((ArrayList)bestCandidateList.get(i)).get
                  (j).toString());
               }
         }}
```

```
    }
}
```

3. tabuSearchMIA.java

```
public class tabuSearchMIA extends tabuSearch {
   int minTardiness;
   int minEarliness;
   int minMakespan;
   int bestT;
   int bestE;
   int bestM;
   int newBestT;
   int newBestE;
   int newBestM;
   boolean checkTabu;
   boolean[] tabuCondition;

   public void msTabuSearch() {
       int msStartTime = (int)(System.currentTimeMillis()/1000);
       //(1) Solution initialization
       updateTabuMIA tb = new updateTabuMIA();
       tb.initList();
       bestT = bestTardiness;
       bestE = bestEarliness;
       bestM = bestMakespan;
       exitCondition = false;
       countSolution = 0;

       //initialize tabu
       for(int i=0; i<tabuListSize; i++){
          maxJList[i] = -1000;
       }

       //(2) Start searching
       for(int msItn=0; msItn<iteration; msItn++){
          jobCondition = new boolean [machine.size()];
          for(int i=0; i<machine.size(); i++){
             jobCondition[i] = true;
          }
          jobList = startList;
          loadingRequirementMIA lr = new loadingRequirementMIA();
          lr.bucketLoading(jobList);
          //Start loading
          for(int i=0; i<nMachine; i++){
             for(int j=0; j<totalBucket; j++){
                startLoading[i][j] = loading[i][j];
             }
          }

          //(3) Find a target job
          targetJob tj = new targetJob();
          tj.deviationPlan();

          //(3-1)If no tardiness or earliness, terminate
          if(deviatedPosCount==0){
             System.out.println("Best solution summary:");
             System.out.println("  Iteration ="+msItn);
             System.out.println("  bestList= "+bestList);
             System.out.println("  FinalBestTardiness ="+bestT);
             System.out.println("  FinalBestEarliness ="+bestE);
             System.out.println("  FinalBestMakespan ="+bestM);
             System.out.println("  FinalMinPoint ="+minPoint);
             break;
```

```
}

//(4) Search for neighbors
MIACount = 0;
localSearchMIA ls = new localSearchMIA();
ls.jobPerBucket();
if(MIACount>=terminatedMIA){
   break;
}

//(5) Update maxJob in tabu
ls.tabuMaxJob(ls.maxJob);

//(6)Evaluate the solutions
earlinessList = new int [msCountList];
tardinessList = new int [msCountList];
makespanList = new int [msCountList];
evaluateMIA em = new evaluateMIA();
for(int e=0; e<msCountList; e++){
   ArrayList co = (ArrayList) candOperatedList.get(e);
   em.msEvaluate(co, e);
}

if(exitCondition == true){
   break;
}

//(5) Check Tabu list
tabuCondition = new boolean [msCountList];
for (int i=0; i<msCountList; i++){
   tabuCondition[i] = true;
}

//(6) Choose the best candidate
selectBestCase();
newBestT = minTardiness;
newBestE = minEarliness;
newBestM = minMakespan;

//(7) Update Tabu list
boolean check = tabuListCheck(minPoint);

//(8) update status to start the next round
int caseNumber;
//Case1: Get the best answer #1
if (((check==true) || (check==false)) && (bestT>newBestT))
{
   caseNumber = 1;
   bestT = newBestT;
   bestE = newBestE;
   bestM = newBestM;
   bestList = (ArrayList) candOperatedList.get(minPoint);
   startList = bestList;
}

//Case2: Get the best answer #2
else if (((check==true) || (check==false)) && (bestT==newBestT) &&
(bestE>newBestE))
{
   caseNumber = 2;
   bestT = newBestT;
   bestE = newBestE;
   bestM = newBestM;
```

```
    bestList = (ArrayList) candOperatedList.get(minPoint);
    startList = bestList;
}

//Case3: Get the best answer #2
else if (((check==true) || (check==false)) && (bestT==newBestT) &&
(bestE==newBestE) && (bestM>newBestM))
{
    caseNumber = 3;
    bestT = newBestT;
    bestE = newBestE;
    bestM = newBestM;
    bestList = (ArrayList) candOperatedList.get(minPoint);
    startList = bestList;
}

//Case4: Some improvement (not Tabu) #1
else if ((check==false) && (bestT<=newBestT) &&
(bestE<=newBestE) && (bestM<=newBestM))
{
    caseNumber = 4;
    startList = (ArrayList) candOperatedList.get(minPoint);

}

//Case5: Some improvement (not Tabu) #2
else if ((check==false) && (bestT<newBestT) && (bestE>newBestE) ||
(bestM>newBestM))
{
    caseNumber = 5;
    startList = (ArrayList) candOperatedList.get(minPoint);
}

//Case6: Some improvement (not Tabu) #2
else if ((check==false) && (bestT==newBestT) &&
(bestE<newBestE) && (bestM>newBestM))
{
    caseNumber = 6;
    startList = (ArrayList) candOperatedList.get(minPoint);
}

//Case7: Some improvement (Tabu)
else {
    caseNumber = 7;
    for(int i=0; i<msCountList; i++){
        tabuCondition[i] = true;
    }
    for(int i=0; i<msCountList; i++){
        for(int j=0; j<tabuListSize; j++){
            int swap1 =
            Integer.parseInt(((ArrayList)swapPairList.get(i)).get(0).t
            oString());
            int swap2 =
            Integer.parseInt(((ArrayList)swapPairList.get(i)).get(1).t
            oString());
            if(swap1==jobFrom[j]&&swap2==jobTo[j] ||
            swap2==jobFrom[j]&&swap1==jobTo[j] ) {
                tabuCondition[i] = false;
            }
    }}
    selectBestCase();
    startList = (ArrayList) candOperatedList.get(minPoint);
}
```

```
        //(9) Update tabu list
        updateTabuMIA tabuUpdate = new updateTabuMIA();
        tabuUpdate.updateList(minPoint);

        //(10) Update Loading
        lr.bucketLoading(startList);

        //(11) Solution summary
        System.out.println("Best solution summary:");
        System.out.println("  Iteration ="+msItn);
        System.out.println("  bestList= "+bestList);
        System.out.println("  FinalBestTardiness ="+bestT);
        System.out.println("  FinalBestEarliness ="+bestE);
        System.out.println("  FinalBestMakespan ="+bestM);
        System.out.println("  FinalMinPoint ="+minPoint);

        //(12) Count the unimproved answers to terminate
        if (caseNumber==4||caseNumber==5||caseNumber==6||caseNumber==7) {
           countSolution = countSolution + 1;
        }
        else {
           countSolution = 0;
        }

        //(13-1) terminated criteria: iteration
        if (countSolution >= terminatedIteration) {
           iteration = msItn;
        }
        //(13-2) terminated criteria: time
        int currentTime = (int)(System.currentTimeMillis()/1000);

        if(currentTime-msStartTime>=runTime){
           iteration = msItn;
        }

        //(13-3) No tardiness or Earliness
        if(bestT==0&&bestE==0){
           iteration = msItn;
        }

     }// Itn loop
     //Print out the best loading
     loadingRequirementMIA lr = new loadingRequirementMIA();
     lr.bucketLoading(bestList);
     for(int i=0; i<nMachine; i++){
        String mcName = machineName.get(i).toString();
     }

     //Convert to time
     convertToTime(bestList);
     int msEndTime = (int)(System.currentTimeMillis()/1000);
     int msTotalTime = msEndTime-msStartTime;
  }

  public void selectBestCase(){
     //Find  a  minimum  tardiness,  earliness,  and  makespan  among  the
     candidates
     int minE = bucketSize*totalBucket;
     int minT = bucketSize*totalBucket;
     int minM = bucketSize;
     int minP = 0;
```

```
    for (int m=0; m < msCountList; m++) {
        if (tardinessList[m] < minT && tabuCondition[m]==true) {
            minT = tardinessList[m];
            minE = earlinessList[m];
            minM = makespanList[m];
            minP = m;
        }
        else if (tardinessList[m] == minT && tabuCondition[m]==true) {
            if (earlinessList[m] < minE && tabuCondition[m]==true){
                minT = tardinessList[m];
                minE = earlinessList[m];
                minM = makespanList[m];
                minP = m;
            }
            else if (earlinessList[m] == minE && tabuCondition[m]==true) {
            if (makespanList[m] < minM && tabuCondition[m]==true){
                minT = tardinessList[m];
                minE = earlinessList[m];
                minM = makespanList[m];
                minP = m;
            }
        }
    }
}
minTardiness = minT;
minEarliness = minE;
minMakespan = minM;
minPoint = minP;
}

public boolean tabuListCheck(int minPoint) {
    checkTabu=false;
    int tabuCount = 0;
    for (int t=0; t<tabuListSize; t++) {
        int swapFrom =
        Integer.parseInt(((ArrayList)swapPairList.get(minPoint)).get(0).toSt
        ring());
        int swapTo =
        Integer.parseInt(((ArrayList)swapPairList.get(minPoint)).get(1).toSt
        ring());
        if (((swapFrom==jobFrom[t]) && (swapTo==jobTo[t])) ||
        ((swapFrom==jobTo[t]) && (swapTo==jobFrom[t]))) {
            ++tabuCount;
         }
    }
    if(tabuCount>0){
        checkTabu = true;
    }
    else {
        checkTabu = false;
    }
    return checkTabu;
}

public void convertToTime(ArrayList bList){
    ArrayList convertedTime;
    ArrayList node;
    timeList = new ArrayList();
    indexList = new ArrayList();
    Iterator b = bList.iterator(); b.hasNext();
    for(int x=0; x<bList.size(); x++){
        ArrayList bl = (ArrayList)b.next();
        convertedTime = new ArrayList();
```

```
            node = new ArrayList();
            for(int y=0; y<bl.size(); y++){
               int                        currentBucket                        =
               Integer.parseInt(((ArrayList)bList.get(x)).get(y).toString());
               int nodeIndex = currentBucket-baseIndex;
               long  time  =  ((currentBucket-baseIndex)*bucketSize)+timeOffset;
               convertedTime.add(time);
               node.add(nodeIndex);
            }
            timeList.add(convertedTime);
            indexList.add(node);
         }
      }

   }
```

3.1 targetJob.java

```
public class targetJob extends tabuSearch {
   ArrayList lpstList = new ArrayList();
   public void deviationPlan() {
       tdvList = new ArrayList();
       deviatedPosCount = 0;
       Iterator m = machine.iterator(); m.hasNext();
       for(int i=0; i<machine.size(); i++){
          ArrayList n = (ArrayList)m.next();
          int totalDeviation = 0;
          for(int j=0; j<n.size(); j++){
             int ojList =
             Integer.parseInt(((ArrayList)jobList.get(i)).get(j).toString());
             int idList =
             Integer.parseInt(((ArrayList)idealplanList.get(i)).get(j).toStrin
             g());

             int dv = ojList-idList;
             totalDeviation = totalDeviation+dv;

             if(j==0){
                lpstList.add(idList);
             }

             if(dv!=0){
                ++deviatedPosCount;
             }
          }
          tdvList.add(totalDeviation);
       }
       lateness();
   }

   public void lateness(){
       diffTimeList = new ArrayList();
       for(int i=0; i<jobList.size(); i++){
          ArrayList j = (ArrayList)jobList.get(i);
          int dd = (int)(baseIndex +
          (Integer.parseInt(plannedDueDate.get(i).toString())-
          timeOffset)/bucketSize);
          int finish =
          Integer.parseInt(((ArrayList)jobList.get(i)).get(j.size()-
          1).toString());
          int diffTime = (finish+1)-dd;
          diffTimeList.add(diffTime);
       }
   }
}
```

3.2 localSearchMIA.java

```
public class localSearchMIA extends tabuSearch {
   ArrayList addJob = new ArrayList();
   ArrayList jobInBucket = new ArrayList();
   ArrayList jobInMachine = new ArrayList();
   ArrayList jobListInBucket = new ArrayList();
   ArrayList newBkList = new ArrayList();
   ArrayList newMPSTList = new ArrayList();
   ArrayList mbkList = new ArrayList();
   ArrayList mbkIndex = new ArrayList();
   ArrayList nbList = new ArrayList();
   ArrayList tnbList = new ArrayList();
   ArrayList mnbList = new ArrayList();
   ArrayList tmnbList = new ArrayList();
   ArrayList swPairList = new ArrayList();
   ArrayList candPSTList = new ArrayList();
   ArrayList startJobInBucket = new ArrayList();
   int maxdiffTime;
   int maxJob;
   int newBk;
   int currentBk;
   int problemSize;
   boolean insertCondition;
   boolean nbhCondition;
   boolean maxJobCondition;
   boolean updateCondition;
   boolean pullCondition;
   ArrayList clList = new ArrayList();
   ArrayList oldIndex = new ArrayList();
   ArrayList newIndex = new ArrayList();
   ArrayList updateJob = new ArrayList();
   ArrayList newJob = new ArrayList();
   int type;
   int start = 0;
   int end = 0;
   int focusBk;
   int mdType = 0;

   public void jobPerBucket(){
      //Identify job per machine per bucket
      for(int i=0; i<nMachine; i++){
         jobInMachine = new ArrayList();
         for(int j=0; j<(int)(heap.length/2)+1; j++){
            int b = j+(int)heap.length/2;
            jobInBucket = new ArrayList();
            Iterator m = machine.iterator(); m.hasNext();
            for(int c=0; c<machine.size(); c++){
               ArrayList mc = (ArrayList)m.next();
               for(int d=0; d<mc.size(); d++){
                  String mt = ((ArrayList)machine.get(c)).get(d).toString();
                  int mcNumber = mcNumber(mt);
                  int oprBucket =
                  Integer.parseInt(((ArrayList)startList.get(c)).get(d).toSt
                  ring());
                  if (((mcNumber)==i)&&(oprBucket==b)) {
                     addJob = new ArrayList();
                     addJob.add(c);
                     addJob.add(d);
                     jobInBucket.add(addJob);
                  }
```

```
            }}
            jobInMachine.add(jobInBucket);
        }
    jobListInBucket.add(jobInMachine);
    }
    startJobInBucket = jobListInBucket;
    generateNbh();
}


public void generateNbh(){
    for(int x=0; x<nMachine; x++){
        for(int y=0; y<totalBucket; y++){
            candLoading[x][y]= startLoading[x][y];
        }
    }
    candLoadingList = new ArrayList();
    candOperatedList = new ArrayList();
    swapPairList = new ArrayList();
    maxJob = 0;

    //(1) Choose a target job
    maxJobCondition = true;
    chooseTargetJob();

    //(2) Insert method
    insertMethod();

    //(3) Swap method: If cannot pull, swap method will be used.
    if(insertCondition == false){
        candOperatedList = new ArrayList();
        candLoadingList = new ArrayList();
        newPSTList = new ArrayList();
        swapMethod();
    }

    //(4) If cannot find the solution for this maxJob, choose other maxJob
    if(maxJobCondition==false){
        MIACount++;
        //if it can't find the new solution up to 10 itn, then quit.
        if(MIACount<terminatedMIA){
            jobCondition[maxJob]=false;
            jobPerBucket();
        }
    }
}

public void insertMethod(){
    clList = new ArrayList();
    newBkList = new ArrayList();
    jobList = startList;
    insertCondition = true;
    ArrayList mc = (ArrayList)machine.get(maxJob);
    for(int d=0; d<mc.size(); d++){
        //Start from the first operation
        currentBk =
        Integer.parseInt(((ArrayList)jobList.get(maxJob)).get(d).toString())
        ;

        //check new available bucket
        if(type==1){//tardiness
            if(d==0){
```

```
                start =
                Math.max((int)(baseIndex+(Integer.parseInt(release.get(maxJob
                ).toString())-timeOffset)/bucketSize), baseIndex);
            }
            else{
                start = Integer.parseInt((newBkList.get(d-1)).toString())+1;
            }
            end = currentBk;
        }
        else {//earliness
            if(d==0){
                start = currentBk;
            }
            else{
                start = Integer.parseInt((newBkList.get(d-1)).toString())+1;
            }
            end = (int)(baseIndex +
            (Integer.parseInt(plannedDueDate.get(maxJob).toString())-
            timeOffset)/bucketSize)-(mc.size()-d-1);
        }

        //Start insertion
        if(end-start==0){
           newBk = currentBk;
        }
        else if (end-start!=0) {
           newbucketForInsert(start, end, d);
        }

        //if found new available bucket, break and then move to next opn.
        if(insertCondition == false){
           break;
        }
        newBkList.add(newBk);
        updateCandLoading(maxJob, d, currentBk, newBk);
    }
    //update new operated bucket
    if(insertCondition == true){

    //Get the new solution
    candPSTList = new ArrayList();
    Iterator jl = jobList.iterator(); jl.hasNext();
    for(int a=0; a<jobList.size(); a++){
        ArrayList ojl = (ArrayList)jobList.get(a);
        if(a!=maxJob){
            candPSTList.add(ojl);
        }
        else{
            candPSTList.add(newBkList);
        }
    }
    msCountList = 1;
    insertCondition = true;
    candOperatedList.add(candPSTList);

    //Update tabu
    swPairList = new ArrayList();
    swPairList.add(-1);
    swPairList.add(-1);
    swapPairList.add(swPairList);
    }
}
```

```
public void newbucketForInsert(int start, int end, int currentOpn) {
   for(int c=0; c<end-start; c++){
      checkBucket = false;
      int nb = c+start;
      checkAvailability(maxJob, currentOpn, nb);
      if (checkBucket == true){
          newBk = nb;
          break;
      }
   }
   if(checkBucket==false){
      insertCondition = false;
   }
}

public void swapMethod() {
   swapPairList = new ArrayList();
   int moveJob;
   int moveOpn;
   int mscount = 0;
   int focusJob = 0;
   ArrayList jobSize = (ArrayList)jobList.get(maxJob);
   problemSize = jobSize.size();

   //(1)Search for nbh points
   //tardiness
   if(type==1){
      start =
      (int)(baseIndex+(Integer.parseInt(release.get(maxJob).toString())-
      timeOffset)/bucketSize)+(problemSize-1);
      end =
      Integer.parseInt(((ArrayList)jobList.get(maxJob)).get(problemSize-
      1).toString());
      searchNBHForSwap(start, end, problemSize-1);
   }
   //earliness
   else{
      start =
      Integer.parseInt(((ArrayList)jobList.get(maxJob)).get(0).toString())
      ;
      end =
      (int)(baseIndex+(Integer.parseInt(plannedDueDate.get(maxJob).toStrin
      g())-timeOffset)/bucketSize)-(problemSize-1);
      searchNBHForSwap(start, end, 0);
   }

   for(int a=0; a<tnbList.size(); a++){
      nbhCondition = true;
      mbkList = new ArrayList();
      mbkIndex = new ArrayList();
      newMPSTList = new ArrayList();
      clList = new ArrayList();
      newBkList = new ArrayList();
      updateJob = new ArrayList();
      oldIndex = new ArrayList();
      newIndex = new ArrayList();
      int focusP;

      //(2)setting
      //(2-1)set candLoading
      for(int x=0; x<nMachine; x++){
          for(int y=0; y<totalBucket; y++){
```

```
    candLoading[x][y]= startLoading[x][y];
     }
}

//(2-2)set jobList
jobList = startList;

//(2-3)set jobInBucket
jobListInBucket = new ArrayList();
jobListInBucket = startJobInBucket;

//(3)Find new bucket from the last opn
for(int b=0; b<problemSize; b++){
    int e;
    if(type==1){
        e = problemSize-(b+1);
        focusP = problemSize-1;
    }
    else{
        e = b;
        focusP = 0;
    }
    currentBk =
    Integer.parseInt(((ArrayList)jobList.get(maxJob)).get(e).toString
    ());

    //First opn or last opn
    if(e==focusP){
        moveJob =
        Integer.parseInt(((ArrayList)tnbList.get(a)).get(0).toString(
        ));
        moveOpn =
        Integer.parseInt(((ArrayList)tnbList.get(a)).get(1).toString(
        ));
        newBk =
        Integer.parseInt(((ArrayList)tnbList.get(a)).get(2).toString(
        ));

        if(moveJob<0&&moveOpn<0){
            focusJob = -1;
        }
        else{
            focusJob = moveJob;
        }

        //Assign the new bucket to max job (from the last operation)
        if(type==1){
            newBkList.add(0,newBk);
        }
        else{
            newBkList.add(newBk);
        }
        updateCandLoading(maxJob, e, currentBk, newBk);
        updateJobListInBucket(maxJob, e, currentBk, newBk);
        if((moveJob!=maxJob)&&(focusJob>=0)){
            newbucketForMovedJob(moveJob, moveOpn, newBk, e);
            if(nbhCondition==false){
                break;
            }
            //Update list for move job
            mbkList.add(newMPSTList);
            mbkIndex.add(moveJob);
        }
```

```
    }
    //other operations
    else{
       if(type==1){
          focusBk = Integer.parseInt(newBkList.get(0).toString());
       }
       else{
          focusBk = Integer.parseInt(newBkList.get(e-1).toString());
       }
       //Searching for max job, create tmnbList
       searchForOtherOperation(focusBk, e);

       //If no any available bucket, move to next NBH
       if(tmnbList.size()==0){
          nbhCondition = false;
          break;
       }

       moveJob =
       Integer.parseInt(((ArrayList)tmnbList.get(0)).get(0).toString
       ());
       moveOpn =
       Integer.parseInt(((ArrayList)tmnbList.get(0)).get(1).toString
       ());
       newBk =
       Integer.parseInt(((ArrayList)tmnbList.get(0)).get(2).toString
       ());

       if(moveJob<0&&moveOpn<0){
          focusJob = -1;
       }
       else{
          focusJob = moveJob;
       }

       //Assign the new bucket to max job (from the last operation)
       if(type==1){
          newBkList.add(0,newBk);
       }
       else{
          newBkList.add(newBk);
       }
       updateCandLoading(maxJob, e, currentBk, newBk);
       updateJobListInBucket(maxJob, e, currentBk, newBk);

       if((moveJob!=maxJob)&&(focusJob>=0)){
          //Assign the new bucket to move job
          newbucketForMovedJob(moveJob, moveOpn, newBk, e);
          if(nbhCondition==false){
             break;
          }
          //Update list for move job
          mbkList.add(newMPSTList);
          mbkIndex.add(moveJob);
       }
    }//end else
}// end all operations

if(nbhCondition==true){
    //Get the new solution
    candPSTList = new ArrayList();
    ArrayList nPSTList = jobList;
    Object[] np = nPSTList.toArray();
```

```
            Object[] ml = mbkList.toArray();
            for(int d=0; d<mbkIndex.size(); d++){
            for (int n=0; n<np.length; n++){
               int mj = Integer.parseInt(mbkIndex.get(d).toString());
               if(n==mj){
                  np[n] = ml[d];
               }
               else if(n==maxJob){
                  np[n] = newBkList;
               }
               else{
                  np[n] = np[n];
               }
               if(d==mbkIndex.size()-1){
                  candPSTList.add(np[n]);
               }
            }
            }
            if (candPSTList.size()>0){
               candOperatedList.add(candPSTList);
               candLoadingList.add(clList);
               swPairList = new ArrayList();
               swPairList.add(maxJob);
               swPairList.add(focusJob);
               swapPairList.add(swPairList);
               ++mscount;
            }
         }
         }// end all NBH
      //If it didn't give any solution, find next maxJob
      if(candOperatedList.size()==0){
         maxJobCondition=false;
      }
      msCountList = mscount;
      insertCondition = true;
   }

   public void searchForOtherOperation(int focusBk, int maxOpn){
      tmnbList = new ArrayList();
      int range;
      int nb;
      String mt = ((ArrayList)machine.get(maxJob)).get(maxOpn).toString();
      int mn = mcNumber(mt);
      int start = (int)(baseIndex +
      (Integer.parseInt(release.get(maxJob).toString())-
      timeOffset)/bucketSize);
      int maxJobT =
      Integer.parseInt(((ArrayList)processTime.get(maxJob)).get(maxOpn).toStr
      ing());
      int maxJobQ = Integer.parseInt(amount.get(maxJob).toString());
      ArrayList x = (ArrayList)jobListInBucket.get(mn);
      int duedate = (int)(baseIndex +
      (Integer.parseInt(plannedDueDate.get(maxJob).toString())-
      timeOffset)/bucketSize);

      //Find time range to search bk
      if(type==1){//tardiness
         range = focusBk-(start+maxOpn);
      }
      else{//earliness
         range = (duedate-maxOpn)-focusBk;
      }
```

```
//Check LPST backward from prev to currentBk
for(int b=0; b<range; b++){
   boolean findBk = false;
   if(type==1){
      nb = focusBk-(b+1);
   }
   else{
      nb = focusBk+(b+1);
   }
   ArrayList y = (ArrayList) x.get(nb-baseIndex);

   //(1) If have enough capacity, add in list
   if((bucketSize-candLoading[mn][nb-baseIndex])>=(maxJobT*maxJobQ)){
      mnbList = new ArrayList();
      mnbList.add(-1);
      mnbList.add(-1);
      mnbList.add(nb);
      tmnbList.add(mnbList);
      break;
   }

   //(2) If have some jobs, check LPST
   else{
      //(2-1) check in current jobInBucket
      for(int c=0; c<y.size(); c++){
         int gap = 0;
         ArrayList jobPair = (ArrayList) y.get(c);
         int job = Integer.parseInt(jobPair.get(0).toString());
         int opn = Integer.parseInt(jobPair.get(1).toString());
         checkUpdate(job, opn, nb);
         if(updateCondition==true){
            int mvSt =
            Integer.parseInt(((ArrayList)PST.get(job)).get(opn).toStri
            ng();
            int maxSt =
            Integer.parseInt(((ArrayList)PST.get(maxJob)).get(maxOpn).
            toString());

            //Check gap between operation of move job
            if(opn==0){
               gap = mvSt-(int)(baseIndex +
               (Integer.parseInt(release.get(job).toString())-
               timeOffset)/bucketSize);
            }
            else{
               gap = mvSt-
               Integer.parseInt(((ArrayList)PST.get(job)).get(opn-
               1).toString());
            }

            //Check if off load this job, is it enough for the current
            job?
            int moveT =
            Integer.parseInt(((ArrayList)processTime.get(job)).get(opn
            ).toString());
            int moveQ = Integer.parseInt(amount.get(job).toString());
            int currentT =
            Integer.parseInt(((ArrayList)processTime.get(maxJob)).get(
            maxOpn).toString());
            int currentQ =
            Integer.parseInt(amount.get(maxJob).toString());
            int freeCap = (bucketSize-candLoading[mn][nb-
            baseIndex])+(moveT*moveQ);
```

```
        //Add in the list
        if(type==1){//tardiness
            if (mvSt>=maxSt && (freeCap>=(currentT*currentQ))){
                mnbList = new ArrayList();
                mnbList.add(job);
                mnbList.add(opn);
                mnbList.add(nb);
                tmnbList.add(mnbList);
                findBk = true;
                break;
            }

            //If st<maxSt but it's possible to pull in this job,
            add in the list.
            else if(mvSt<maxSt && freeCap>=(maxJobT*maxJobQ) &&
            gap>0){
                mnbList = new ArrayList();
                mnbList.add(job);
                mnbList.add(opn);
                mnbList.add(nb);
                tmnbList.add(mnbList);
                findBk = true;
                break;
            }
        }
        else if (type==2){//earliness
            if (mvSt<=maxSt && (freeCap>=(currentT*currentQ))){
                mnbList = new ArrayList();
                mnbList.add(job);
                mnbList.add(opn);
                mnbList.add(nb);
                tmnbList.add(mnbList);
                findBk = true;
                break;
            }
        }
    }
}//end (2-1)

if(findBk==true){
    break;
}

//(2-2) Check any new job of the bucket from the update list
searchForJob(nb, mn);
for(int d=0; d<newJob.size(); d++){
    int gap = 0;
    ArrayList jobPair = (ArrayList) newJob.get(d);
    int job = Integer.parseInt(jobPair.get(0).toString());
    int opn = Integer.parseInt(jobPair.get(1).toString());
    int mvSt =
    Integer.parseInt(((ArrayList)PST.get(job)).get(opn).toString(
    ));
    int maxSt =
    Integer.parseInt(((ArrayList)PST.get(maxJob)).get(maxOpn).toS
    tring());

    //Check gap between operation of move job
    if(opn==0){
        gap = mvSt-(int)(baseIndex +
        (Integer.parseInt(release.get(job).toString())-
        timeOffset)/bucketSize);
```

```
                }
                else{
                    gap = mvSt-
                    Integer.parseInt(((ArrayList)PST.get(job)).get(opn-
                    1).toString());
                }

                //If offload this job, is it enough for the current job?
                int moveT =
                Integer.parseInt(((ArrayList)processTime.get(job)).get(opn).t
                oString());
                int moveQ = Integer.parseInt(amount.get(job).toString());
                int currentT =
                Integer.parseInt(((ArrayList)processTime.get(maxJob)).get(max
                Opn).toString());
                int currentQ =
                Integer.parseInt(amount.get(maxJob).toString());
                int freeCap = (bucketSize-candLoading[mn][nb-
                baseIndex])+(moveT*moveQ);

                if(type==1){
                    if (mvSt>=maxSt && (freeCap>=(currentT*currentQ))){
                        mnbList = new ArrayList();
                        mnbList.add(job);
                        mnbList.add(opn);
                        mnbList.add(nb);
                        tmnbList.add(mnbList);
                        findBk = true;
                        break;
                    }
                    //If st<maxSt but it's possible to pull in this job, add
                    in the list.
                    else if(mvSt<maxSt && freeCap>=(maxJobT*maxJobQ) &&
                    gap>0){
                        mnbList = new ArrayList();
                        mnbList.add(job);
                        mnbList.add(opn);
                        mnbList.add(nb);
                        tmnbList.add(mnbList);
                        findBk = true;
                        break;
                    }
                }

                else if (type==2){//earliness
                    if (mvSt<=maxSt && (freeCap>=(currentT*currentQ))){
                        mnbList = new ArrayList();
                        mnbList.add(job);
                        mnbList.add(opn);
                        mnbList.add(nb);
                        tmnbList.add(mnbList);
                        findBk = true;
                        break;
                    }
                }
            }//end (2)
        }//end else
    }// end search
}

public void newbucketForMovedJob(int mJob, int mOpn, int mBucket, int
maxOpn){
    boolean updateCondition = true;
```

```
    //(1)Find new bucket for move job
    newMPSTList = new ArrayList();
    ArrayList j = (ArrayList) jobList.get(mJob);
    for(int c=0; c<j.size(); c++){
        switch (type){
        case 1://tardiness-push
            if(c<mOpn){
                newBk =
                Integer.parseInt(((ArrayList)jobList.get(mJob)).get(c).toStri
                ng());
            }
            else {
                int oldBk =
                Integer.parseInt(((ArrayList)jobList.get(mJob)).get(c).toStri
                ng());
                int prevOpnBk = newBk;

                if(c==0&&mOpn==0){
                    pull(mJob, c, oldBk);
                    if(pullCondition==false){
                        push(mJob, c, oldBk);
                    }
                }
                else{
                    push(mJob, c, prevOpnBk);
                }
                updateCandLoading(mJob, c, oldBk, newBk);
                updateJobListInBucket(mJob, c, oldBk, newBk);
            }
            newMPSTList.add(newBk);
            break;

        case 2://earliness-pull
            int d = (j.size()-1)-c;
            if(d>mOpn){
                newBk =
                Integer.parseInt(((ArrayList)jobList.get(mJob)).get(d).toStri
                ng());
            }
            else {
                int oldBk =
                Integer.parseInt(((ArrayList)jobList.get(mJob)).get(d).toStri
                ng());
                int sucOpnBk = newBk;
                pull(mJob, d, sucOpnBk);
                if(pullCondition==false){
                    nbhCondition = false;
                    updateCondition = false;
                    break;
                }
                updateCandLoading(mJob, d, oldBk, newBk);
                updateJobListInBucket(mJob, d, oldBk, newBk);
            }
            newMPSTList.add(0, newBk);
            break;
        }
    }
    if(updateCondition==true){
        updateJobList(mJob,newMPSTList);
    }
}

public void push(int mJob, int mOpn, int prevBk){
```

```
   checkBucket = true;
   newBk = prevBk+1;
   checkAvailability(mJob, mOpn, newBk);
   if(checkBucket == false){
      push(mJob, mOpn, newBk);
   }
}

public void pull(int mJob, int mOpn, int prevBk){
   pullCondition = true;
   checkBucket = true;
   newBk = prevBk-1;
   int est = (int)(baseIndex +
   (Integer.parseInt(release.get(maxJob).toString())-
   timeOffset)/bucketSize)+mOpn;

   if(newBk<baseIndex||newBk<est){
      pullCondition = false;
   }

   if(pullCondition==true){
      checkAvailability(mJob, mOpn, newBk);
      if(checkBucket == false){
         pull(mJob, mOpn, newBk);
      }
   }
}

public void searchNBHForSwap(int start, int end, int focusOpn){
   int gap = 0;
   tnbList = new ArrayList();
   //Find the nbh from the last operation
   String mt = ((ArrayList)machine.get(maxJob)).get(focusOpn).toString();
   int mcNumber = mcNumber(mt);
   int maxJobT =
   Integer.parseInt(((ArrayList)processTime.get(maxJob)).get(focusOpn).toS
   tring());
   int maxJobQ = Integer.parseInt(amount.get(maxJob).toString());

   //Check the jobs that are in the same machine and the range from Est to
   current bucket
   ArrayList x = (ArrayList)jobListInBucket.get(mcNumber);

   for(int b=0; b<end-start; b++){
      int nb;
      if(type==1){//tardiness
         nb = b+start;
      }
      else{//earliness
         nb = (b+1)+start;
      }
      ArrayList y = (ArrayList) x.get(nb-baseIndex);

      //(1) Check the capacity. If true, add in list
      if((bucketSize-startLoading[mcNumber][nb-
      baseIndex])>=(maxJobT*maxJobQ)){
         nbList = new ArrayList();
         nbList.add(-1);
         nbList.add(-1);
         nbList.add(nb);
         tnbList.add(nbList);
      }
```

```
//(2) If have some jobs, check LPST
else{
   for(int c=0; c<y.size(); c++){
      ArrayList jobPair = (ArrayList) y.get(c);
      //(1)Check LPST of the move job with the max job
      int job = Integer.parseInt(jobPair.get(0).toString());
      int opn = Integer.parseInt(jobPair.get(1).toString());
      int st =
      Integer.parseInt(((ArrayList)PST.get(job)).get(opn).toString(
      ));
      int maxSt =
      Integer.parseInt(((ArrayList)PST.get(maxJob)).get(focusOpn).t
      oString());

      //(2)Check gap between operation of move job
      if(opn==0){
         gap = st-(int)(baseIndex +
         (Integer.parseInt(release.get(job).toString())-
         timeOffset)/bucketSize);
      }
      else{
         gap = st-
         Integer.parseInt(((ArrayList)PST.get(job)).get(opn-
         1).toString());
      }

      //(3)Check if off load this move job, is it enough for the
      current job?
      int moveT =
      Integer.parseInt(((ArrayList)processTime.get(job)).get(opn).t
      oString());
      int moveQ = Integer.parseInt(amount.get(job).toString());
      int freeCap = (bucketSize-startLoading[mcNumber][nb-
      baseIndex])+(moveT*moveQ);

      //(4)Add the possible move
      if(type==1){//tardiness
         if (st>=maxSt && (freeCap>=(maxJobT*maxJobQ))){
            nbList = new ArrayList();
            nbList.add(job);
            nbList.add(opn);
            nbList.add(nb);
            tnbList.add(nbList);
         }

         //If st<maxSt but it's possible to pull in this job, add
         in the list.
         else if(st<maxSt && freeCap>=(maxJobT*maxJobQ) && gap>0){
            nbList = new ArrayList();
            nbList.add(job);
            nbList.add(opn);
            nbList.add(nb);
            tnbList.add(nbList);
         }
      }

      else if(type==2){//earliness
         if (st<=maxSt && (freeCap>=(maxJobT*maxJobQ))){
            nbList = new ArrayList();
            nbList.add(job);
            nbList.add(opn);
            nbList.add(nb);
            tnbList.add(nbList);
```

```
                }
            }
        }
    }}
}

public void updateCandLoading(int job, int opn, int oldBk, int nBk){
    String m = ((ArrayList)machine.get(job)).get(opn).toString();
    int mn = mcNumber(m);
    oldBk -=baseIndex;
    nBk -=baseIndex;
    int machineTime =
    Integer.parseInt(((ArrayList)processTime.get(job)).get(opn).toString())
    ;
    int quantity = Integer.parseInt(amount.get(job).toString());

    //update new loadings
    candLoading[mn][oldBk] = candLoading[mn][oldBk]-(machineTime*quantity);
    candLoading[mn][nBk] = candLoading[mn][nBk]+(machineTime*quantity);

    //Put the change in the list
    ArrayList cList = new ArrayList();
    cList.add((mn));
    cList.add(oldBk);
    cList.add(candLoading[mn][oldBk]);
    clList.add(cList);
    cList = new ArrayList();
    cList.add((mn));
    cList.add(nBk);
    cList.add(candLoading[mn][nBk]);
    clList.add(cList);
}

public boolean checkAvailability(int i, int j, int nb) {
    boolean condition = false;
    nb = nb-baseIndex;
    int p =
    Integer.parseInt(((ArrayList)processTime.get(i)).get(j).toString());
    int q = Integer.parseInt(amount.get(i).toString());
    String m = ((ArrayList)machine.get(i)).get(j).toString();
     int mn = mcNumber(m);

    //Check with update bucket from updateJob
    for(int a=0; a<updateJob.size(); a++){
        int x =
        Integer.parseInt(((ArrayList)updateJob.get(a)).get(0).toString());
        int y =
        Integer.parseInt(((ArrayList)updateJob.get(a)).get(1).toString());
        int z = Integer.parseInt(newIndex.get(a).toString());
        if(x==i&&y==j&&z==nb){
            checkBucket = true;
            condition = true;
            break;
        }
    }

    if(condition==false){
        if((p*q) <= (bucketSize-candLoading[mn][nb])) {
            checkBucket = true;
        }
        else{
            checkBucket = false;
        }
```

```
    }
    return checkBucket;
}

public void chooseTargetJob(){
    maxdiffTime = 0;
    int count = 0;
    for(int i=0; i<diffTimeList.size(); i++){
        int md = Math.abs(Integer.parseInt(diffTimeList.get(i).toString()));
        if (md>maxdiffTime&&jobCondition[i]==true&&md!=0) {
            maxdiffTime = md;
            maxJob = i;
        }
        else{
            ++count;
        }
    }

    // if can't find max job, clear tabu search
    if(count==diffTimeList.size()){
        for (int t=0; t<machine.size(); t++) {
            jobCondition[t] = true;
        }
        for (int r=0; r<tabuListSize; r++) {
            maxJList[r] = -1000;
        }
        chooseTargetJob();
    }
    else{
        mdType = Integer.parseInt(diffTimeList.get(maxJob).toString());
        if(mdType>0){
            type = 1;
        }
        else{
            type = 2;
        }
        for(int j=0; j<tabuListSize; j++){
            if(maxJob==maxJList[j]){
                jobCondition[maxJob] = false;
                  chooseTargetJob();
                  break;
            }
        }
    }
}

public void updateJobList(int changeJob, ArrayList newList){
   ArrayList jList = new ArrayList();
   for(int a=0; a<jobList.size(); a++){
        if(a==changeJob){
            jList.add(newList);
        }
        else{
            jList.add(startList.get(a));
        }
   }
   jobList = jList;
}

public void updateJobListInBucket(int job, int opn, int oldBk, int newBk){
    String m = ((ArrayList)machine.get(job)).get(opn).toString();
    int n = mcNumber(m);
    ArrayList addJ = new ArrayList();
```

```
    addJ.add(job);
    addJ.add(opn);
    addJ.add(n);
    updateJob.add(addJ);
    newIndex.add(newBk);
    oldIndex.add(oldBk);
}

public void checkUpdate(int job, int opn, int nb){
    updateCondition = true;
    for (int a=0; a<updateJob.size(); a++){
        int x =
        Integer.parseInt(((ArrayList)updateJob.get(a)).get(0).toString());
        int y =
        Integer.parseInt(((ArrayList)updateJob.get(a)).get(1).toString());
        int z = Integer.parseInt((newIndex.get(a)).toString());
        if(x==job&&y==opn&nb!=z){
            updateCondition = false;
        }
    }
}

public void searchForJob(int nb, int mc){
    newJob = new ArrayList();
    //find new job in this bucket
    for (int a=0; a<newIndex.size(); a++){
        int bk = Integer.parseInt((newIndex.get(a)).toString());
        ArrayList j = (ArrayList) updateJob.get(a);
        if(bk==nb){
           newJob.add(j);
        }
    }

    //check with old bk, then delete if found the same job
    for (int b=0; b<oldIndex.size(); b++){
        int obk = Integer.parseInt((oldIndex.get(b)).toString());
        if(obk==nb){
            int j =
            Integer.parseInt(((ArrayList)updateJob.get(b)).get(0).toString())
            ;
            int o =
            Integer.parseInt(((ArrayList)updateJob.get(b)).get(1).toString())
            ;

            for (int c=0; c<newJob.size(); c++){
                int nj =
                Integer.parseInt(((ArrayList)newJob.get(c)).get(0).toString()
                );
                int no =
                Integer.parseInt(((ArrayList)newJob.get(c)).get(1).toString()
                );

                if(nj==j&&no==o){
                    newJob.set(c, "");
                }
            }
        }
    }
}

public void tabuMaxJob(int maxJob){
    for (int t=0; t<tabuListSize; t++) {
    if (t < tabuListSize-1) {
```

```
            maxJList[t] = maxJList[t+1];
        }
        else {
            maxJList[t] = maxJob;
        }
        }
    }

    public int mcNumber(String mc){
        int mcName = 0;
        for(int a=0; a<machineName.size(); a++){
            String mcn = machineName.get(a).toString();
            if(mc.equals(mcn)){
                mcName = a;
            }
        }
        return mcName;
    }
}
```

3.3 evaluateMIA.java

```
public class evaluateMIA extends tabuSearch {
   int msMakespan;

   @SuppressWarnings("unchecked")
   public void msEvaluate(ArrayList oprb, int count){
      msTardiness = new int [machine.size()];
      msEarliness = new int [machine.size()];
      deviatedTime = new int [machine.size()];
      int []eachMS = new int [machine.size()];
      int minStartTime = bucketSize;
      int maxFinishTime = 0;
      int totalE = 0;
      int totalT = 0;
      int totalMS = 0;
      Iterator o = oprb.iterator(); o.hasNext();
      for (int i=0; i<machine.size(); i++){
         int start =
         Integer.parseInt(((ArrayList)oprb.get(i)).get(0).toString());
         ArrayList ob = (ArrayList)o.next();
         int finish =
         Integer.parseInt(((ArrayList)oprb.get(i)).get(ob.size()-
         1).toString());
         int dueBucket = (int)(baseIndex +
         (Integer.parseInt(plannedDueDate.get(i).toString())-
         timeOffset)/bucketSize);

         if((finish+1)>=dueBucket){
            msEarliness[i] = 0;
            msTardiness[i] = (finish+1)-dueBucket;
         }
         else {
            msEarliness[i] = dueBucket-(finish+1);
            msTardiness[i]= 0;
         }

         for (int d=0; d<ob.size(); d++){
            int jobPt =
            Integer.parseInt(((ArrayList)oprb.get(i)).get(d).toString());
            int jobDd = dueBucket-(ob.size()-d);
            int dvt = jobPt-jobDd;
            deviatedTime[i] = deviatedTime[i]+dvt;
         }

         eachMS[i] = (finish-start)+1;
         //Calculate earliest start time
         if(start<minStartTime){
            minStartTime = start;
         }
         //Calculate latest finish time
         if(finish>maxFinishTime){
            maxFinishTime = finish;
         }
         totalE = totalE + msEarliness[i];
         totalT = totalT + msTardiness[i];
         totalMS = totalMS +eachMS[i];
      }
      earlinessList[count] = totalE;
      tardinessList[count] = totalT;
      msMakespan = (maxFinishTime-minStartTime)+1;
```

```
            makespanList[count] = totalMS;
        }
    }
```

3.4 loadingRequirementMIA.java

```
public class loadingRequirementMIA extends tabuSearch {
   public void bucketLoading(ArrayList jobList) {
       for(int i=0; i<nMachine; i++){
          for(int j=0; j<totalBucket; j++){
             loading[i][j] = 0;
       }}

       //Loading per machine per bucket
       for(int i=0; i<nMachine; i++){
          String mcName = machineName.get(i).toString();
          for(int j=0; j<baseIndex+1; j++){
             int k = j+baseIndex;
             Iterator m = machine.iterator(); m.hasNext();
             for(int c=0; c<machine.size(); c++){
                ArrayList mc = (ArrayList)m.next();
                for(int d=0; d<mc.size(); d++){
                   String mt = ((ArrayList)machine.get(c)).get(d).toString();
                   int opnTime =
                   Integer.parseInt(((ArrayList)jobList.get(c)).get(d).toStri
                   ng());
                   int machineTime =
                   Integer.parseInt(((ArrayList)processTime.get(c)).get(d).to
                   String());
                   int quantity = Integer.parseInt(amount.get(c).toString());

                   if ((mt.equals(mcName))&&(opnTime==k)) {
                      loading[i][j] +=(machineTime*quantity);
                   }
             }}
          }
       }
   }

   public void updateLoading(){
       for(int x=0; x<nMachine; x++){
          for(int y=0; y<totalBucket; y++){
             if(candLoading[x][y]>0) {
                loading[x][y]=candLoading[x][y];
             }
             else{
                loading[x][y]=startLoading[x][y];
             }
          }
       }
   }
}
```

3.5 updateTabuMIA.java

```
public class updateTabuMIA extends tabuSearch {
   int moveFrom;
   int moveTo;

   public void initList() {
       for (int t=0; t<tabuListSize; t++) {
          jobFrom[t] = 0;
          jobTo[t] = 0;
       }
   }

   public void updateList(int mPoint) {
       for (int t=0; t<tabuListSize; t++) {
          if (t < tabuListSize-1) {
              jobFrom[t] = jobFrom[t+1];
              jobTo[t] = jobTo[t+1];
          }
          else if (t == (tabuListSize-1)){
              for(int l=0; l<msCountList; l++){
                 if(l == mPoint){
                    moveFrom =
                    Integer.parseInt(((ArrayList)swapPairList.get(l)).get(0).t
                    oString());
                    moveTo =
                    Integer.parseInt(((ArrayList)swapPairList.get(l)).get(1).t
                    oString());
                 }
              }
              jobFrom[t] = moveFrom;
              jobTo[t] = moveTo;
          }
       }
    }
}
```

Appendix B

An instance of the application running

1. Service architecture file

In order to implement the proposed planning application into the CONPLAN, a XML description file is used to define service architecture. The XML file will list all events occurring in the process flow and then map the events with specific services, for instance in order to generate a resource plan (an event), the tabu search algorithm (a service) is chosen to be a solving method. The planning algorithm is thus embedded in the CONPLAN. In this planning application, the XML file, called tabu.xml, is created for illustrating services implementation. The code is shown as follows.

```
<architecture>
   <events>
   </events>
   <external-event-generators>
       <generator name="init" class="src.conplan.generators.InitGenerator" />
   </external-event-generators>
   <services>
      <service name="planner1" archetype="Planner"
      state="src.conplan.state.IndexPlanner">
         <listens for="src.conplan.events.Init"
         class="src.conplan.listeners.Ignore" />
         <listens for="src.conplan.events.UnplannedOrder"
         class="src.conplan.listeners.PlanOrder" />
         <listens for="src.conplan.events.SystemState"
         class="src.conplan.listeners.Ignore" />
         <listens for="src.conplan.events.CapacityShortage"
         class="src.conplan.listeners.Ignore" />
         <listens for="src.conplan.events.GlobalPlanning"
         class="src.conplan.listeners.PerformGlobalPlanning" />
      </service>

      <service name="order3" archetype="OrderCreation"
      state="src.conplan.state.OrderCreationFromFile">
         <listens for="src.conplan.events.Init"
         class="src.conplan.listeners.CreateUnplannedOrder" />
      </service>
      <service name="utility" archetype="Other"
      state="src.conplan.state.HaltOnComplete">
```

```
            <listens for="src.conplan.events.IncrementWait"
            class="src.conplan.listeners.CreateHalt" />
            <listens for="src.conplan.events.DecrementWait"
            class="src.conplan.listeners.CreateHalt" />
            <listens for="src.conplan.events.Init"
            class="src.conplan.listeners.CreatePeriodic" />
            <listens for="src.conplan.events.PeriodicTrigger"
            class="src.conplan.listeners.CreatePeriodic" />
        </service>
    </services>
</architecture>
```

2. Input data arrays

Input data from the text files will be converted to a desired array pattern. For a data format of date and time, the data will be inputted with the format pattern YYYY,MM,DD,HH,MM,SS. It then needs to be converted to long number format for using in the tabu search algorithm. For instance a release date of job 1 is 2009,09,02,08,00,00. To start the calculation, this data will be changed to 1251878400. The example of all input data arrays needed for the planning algorithm is shown below.

```
Hardcoded TimeOffset:  2009,09,26,00,00,00,
due date= [1254988800, 1255334400, 1255420800, 1255420800, 1255075200,
1255593600, 1255593600, 1255680000, 1255593600, 1255248000, 1255075200,
1254643200, 1254902400, 1255161600, 1255507200]
release date= [1251878400, 1251964800, 1252396800, 1252656000, 1252742400,
1252915200, 1253001600, 1253174400, 1253433600, 1253520000, 1253606400,
1253692800, 1253692800, 1253779200, 1253865600]
routing=  [[M19, M11, M3, M2], [M14, M2, M7, M16, M3, M10], [M14, M2, M7, M16,
M3, M10], [M1, M5, M12, M11, M2, M8, M19, M13, M15], [M19, M11, M3, M2], [M1,
M5, M12, M11, M2, M8, M19, M13, M15], [M14, M2, M7, M16, M3, M10], [M14, M2,
M7, M16, M3, M10], [M1, M5, M12, M11, M2, M8, M19, M13, M15], [M19, M11, M3,
M2], [M14, M2, M7, M16, M3, M10], [M19, M11, M3, M2], [M19, M11, M3, M2],
[M14, M2, M7, M16, M3, M10], [M14, M2, M7, M16, M3, M10]]
process time=  [[594, 594, 605, 203], [599, 594, 585, 628, 600, 658], [599,
594, 585, 628, 600, 658], [580, 585, 582, 596, 597, 593, 615, 628, 605], [594,
594, 605, 203], [580, 585, 582, 596, 597, 593, 615, 628, 605], [599, 594, 585,
628, 600, 658], [599, 594, 585, 628, 600, 658], [580, 585, 582, 596, 597, 593,
615, 628, 605], [594, 594, 605, 203], [599, 594, 585, 628, 600, 658], [594,
594, 605, 203], [594, 594, 605, 203], [599, 594, 585, 628, 600, 658], [599,
594, 585, 628, 600, 658]]
amount=  [100, 100, 100, 100, 100, 100, 100, 100, 100, 100, 100, 100, 100,
100, 100]
machine=  [M1, M10, M11, M12, M13, M14, M15, M16, M17, M18, M19, M2, M20, M3,
M4, M5, M6, M7, M8, M9]
```

3. An example of the planning results

In the planning application, there are two types of number formats: long number and time bucket number, which are used to present the output, an operation start time of each operation of each job. The long number represents time at the beginning of a bucket in a planning horizon. Meanwhile, the bucket number represents a number of discrete interval times in a planning horizon. The example results of calculation obtained from the tabu search algorithm are shown as follows.

```
Hardcoded TimeOffset:  2009,09,26,00,00,00,
due date= [1254988800, 1255334400, 1255420800, 1255420800, 1255075200,
1255593600, 1255593600, 1255680000, 1255593600, 1255248000, 1255075200,
1254643200, 1254902400, 1255161600, 1255507200]
release date= [1251878400, 1251964800, 1252396800, 1252656000, 1252742400,
1252915200, 1253001600, 1253174400, 1253433600, 1253520000, 1253606400,
1253692800, 1253692800, 1253779200, 1253865600]
routing=  [[M19, M11, M3, M2], [M14, M2, M7, M16, M3, M10], [M14, M2, M7, M16,
M3, M10], [M1, M5, M12, M11, M2, M8, M19, M13, M15], [M19, M11, M3, M2], [M1,
M5, M12, M11, M2, M8, M19, M13, M15], [M14, M2, M7, M16, M3, M10], [M14, M2,
M7, M16, M3, M10], [M1, M5, M12, M11, M2, M8, M19, M13, M15], [M19, M11, M3,
M2], [M14, M2, M7, M16, M3, M10], [M19, M11, M3, M2], [M19, M11, M3, M2],
[M14, M2, M7, M16, M3, M10], [M14, M2, M7, M16, M3, M10]]
process time=  [[594, 594, 605, 203], [599, 594, 585, 628, 600, 658], [599,
594, 585, 628, 600, 658], [580, 585, 582, 596, 597, 593, 615, 628, 605], [594,
594, 605, 203], [580, 585, 582, 596, 597, 593, 615, 628, 605], [599, 594, 585,
628, 600, 658], [599, 594, 585, 628, 600, 658], [580, 585, 582, 596, 597, 593,
615, 628, 605], [594, 594, 605, 203], [599, 594, 585, 628, 600, 658], [594,
594, 605, 203], [594, 594, 605, 203], [599, 594, 585, 628, 600, 658], [599,
594, 585, 628, 600, 658]]
amount=  [100, 100, 100, 100, 100, 100, 100, 100, 100, 100, 100, 100, 100,
100, 100]
machine=  [M1, M10, M11, M12, M13, M14, M15, M16, M17, M18, M19, M2, M20, M3,
M4, M5, M6, M7, M8, M9]
initPST= [[39, 40, 41, 42], [41, 42, 43, 44, 45, 46], [42, 43, 44, 45, 46,
47], [39, 40, 41, 42, 43, 44, 45, 46, 47], [40, 41, 42, 43], [41, 42, 43, 44,
45, 46, 47, 48, 49], [44, 45, 46, 47, 48, 49], [45, 46, 47, 48, 49, 50], [41,
42, 43, 44, 45, 46, 47, 48, 49], [42, 43, 44, 45], [38, 39, 40, 41, 42, 43],
[35, 36, 37, 38], [38, 39, 40, 41], [39, 40, 41, 42, 43, 44], [43, 44, 45, 46,
47, 48]]
************************************
Iteration(OIA) =0
************************************
jobList= [[39, 40, 41, 42], [41, 42, 43, 44, 45, 46], [42, 43, 44, 45, 46,
47], [39, 40, 41, 42, 43, 44, 45, 46, 47], [40, 41, 42, 43], [41, 42, 43, 44,
45, 46, 47, 48, 49], [44, 45, 46, 47, 48, 49], [45, 46, 47, 48, 49, 50], [41,
42, 43, 44, 45, 46, 47, 48, 49], [42, 43, 44, 45], [38, 39, 40, 41, 42, 43],
[35, 36, 37, 38], [38, 39, 40, 41], [39, 40, 41, 42, 43, 44], [43, 44, 45, 46,
47, 48]]
MaxOverload= -112700 at machine= M2 and bucket= 45
newPSTList= [[[39, 40, 41, 42], [41, 42, 43, 44, 45, 46], [42, 43, 44, 45, 46,
47], [39, 40, 41, 42, 43, 44, 45, 46, 47], [40, 41, 42, 43], [35, 36, 37, 38,
```

```
40, 46, 47, 48, 49], [44, 45, 46, 47, 48, 49], [45, 46, 47, 48, 49, 50], [41,
42, 43, 44, 45, 46, 47, 48, 49], [42, 43, 44, 45], [38, 39, 40, 41, 42, 43],
[35, 36, 37, 38], [38, 39, 40, 41], [39, 40, 41, 42, 43, 44], [43, 44, 45, 46,
47, 48]]]
MaxOverload= -53000 at machine= M2 and bucket= 43
newPSTList= [[[39, 40, 41, 42], [41, 42, 43, 44, 45, 46], [42, 43, 44, 45, 46,
47], [39, 40, 41, 42, 43, 44, 45, 46, 47], [40, 41, 42, 43], [35, 36, 37, 38,
40, 46, 47, 48, 49], [44, 45, 46, 47, 48, 49], [45, 46, 47, 48, 49, 50], [41,
42, 43, 44, 45, 46, 47, 48, 49], [42, 43, 44, 45], [38, 39, 40, 41, 42, 43],
[35, 36, 37, 38], [38, 39, 40, 41], [39, 40, 41, 42, 43, 44], [43, 44, 45, 46,
47, 48]], [[39, 40, 41, 42], [41, 42, 43, 44, 45, 46], [42, 43, 44, 45, 46,
47], [39, 40, 41, 42, 43, 44, 45, 46, 47], [40, 41, 42, 43], [41, 42, 43, 44,
45, 46, 47, 48, 49], [37, 40, 46, 47, 48, 49], [45, 46, 47, 48, 49, 50], [41,
42, 43, 44, 45, 46, 47, 48, 49], [42, 43, 44, 45], [38, 39, 40, 41, 42, 43],
[35, 36, 37, 38], [38, 39, 40, 41], [39, 40, 41, 42, 43, 44], [43, 44, 45, 46,
47, 48]]]
MaxOverload= -53300 at machine= M2 and bucket= 45
newPSTList= [[[39, 40, 41, 42], [41, 42, 43, 44, 45, 46], [42, 43, 44, 45, 46,
47], [39, 40, 41, 42, 43, 44, 45, 46, 47], [40, 41, 42, 43], [35, 36, 37, 38,
40, 46, 47, 48, 49], [44, 45, 46, 47, 48, 49], [45, 46, 47, 48, 49, 50], [41,
42, 43, 44, 45, 46, 47, 48, 49], [42, 43, 44, 45], [38, 39, 40, 41, 42, 43],
[35, 36, 37, 38], [38, 39, 40, 41], [39, 40, 41, 42, 43, 44], [43, 44, 45, 46,
47, 48]], [[39, 40, 41, 42], [41, 42, 43, 44, 45, 46], [42, 43, 44, 45, 46,
47], [39, 40, 41, 42, 43, 44, 45, 46, 47], [40, 41, 42, 43], [41, 42, 43, 44,
45, 46, 47, 48, 49], [37, 40, 46, 47, 48, 49], [45, 46, 47, 48, 49, 50], [41,
42, 43, 44, 45, 46, 47, 48, 49], [42, 43, 44, 45], [38, 39, 40, 41, 42, 43],
[35, 36, 37, 38], [38, 39, 40, 41], [39, 40, 41, 42, 43, 44], [43, 44, 45, 46,
47, 48]], [[39, 40, 41, 42], [41, 42, 43, 44, 45, 46], [42, 43, 44, 45, 46,
47], [39, 40, 41, 42, 43, 44, 45, 46, 47], [40, 41, 42, 43], [41, 42, 43, 44,
45, 46, 47, 48, 49], [44, 45, 46, 47, 48, 49], [45, 46, 47, 48, 49, 50], [35,
36, 37, 38, 40, 46, 47, 48, 49], [42, 43, 44, 45], [38, 39, 40, 41, 42, 43],
[35, 36, 37, 38], [38, 39, 40, 41], [39, 40, 41, 42, 43, 44], [43, 44, 45, 46,
47, 48]]]
MaxOverload= -53000 at machine= M2 and bucket= 43
newPSTList= [[[39, 40, 41, 42], [41, 42, 43, 44, 45, 46], [42, 43, 44, 45, 46,
47], [39, 40, 41, 42, 43, 44, 45, 46, 47], [40, 41, 42, 43], [35, 36, 37, 38,
40, 46, 47, 48, 49], [44, 45, 46, 47, 48, 49], [45, 46, 47, 48, 49, 50], [41,
42, 43, 44, 45, 46, 47, 48, 49], [42, 43, 44, 45], [38, 39, 40, 41, 42, 43],
[35, 36, 37, 38], [38, 39, 40, 41], [39, 40, 41, 42, 43, 44], [43, 44, 45, 46,
47, 48]], [[39, 40, 41, 42], [41, 42, 43, 44, 45, 46], [42, 43, 44, 45, 46,
47], [39, 40, 41, 42, 43, 44, 45, 46, 47], [40, 41, 42, 43], [41, 42, 43, 44,
45, 46, 47, 48, 49], [37, 40, 46, 47, 48, 49], [45, 46, 47, 48, 49, 50], [41,
42, 43, 44, 45, 46, 47, 48, 49], [42, 43, 44, 45], [38, 39, 40, 41, 42, 43],
[35, 36, 37, 38], [38, 39, 40, 41], [39, 40, 41, 42, 43, 44], [43, 44, 45, 46,
47, 48]], [[39, 40, 41, 42], [41, 42, 43, 44, 45, 46], [42, 43, 44, 45, 46,
47], [39, 40, 41, 42, 43, 44, 45, 46, 47], [40, 41, 42, 43], [41, 42, 43, 44,
45, 46, 47, 48, 49], [44, 45, 46, 47, 48, 49], [45, 46, 47, 48, 49, 50], [35,
36, 37, 38, 40, 46, 47, 48, 49], [42, 43, 44, 45], [38, 39, 40, 41, 42, 43],
[35, 36, 37, 38], [38, 39, 40, 41], [39, 40, 41, 42, 43, 44], [43, 44, 45, 46,
47, 48]], [[39, 40, 41, 42], [41, 42, 43, 44, 45, 46], [42, 43, 44, 45, 46,
47], [39, 40, 41, 42, 43, 44, 45, 46, 47], [40, 41, 42, 43], [41, 42, 43, 44,
45, 46, 47, 48, 49], [44, 45, 46, 47, 48, 49], [45, 46, 47, 48, 49, 50], [41,
42, 43, 44, 45, 46, 47, 48, 49], [37, 38, 39, 41], [38, 39, 40, 41, 42, 43],
[35, 36, 37, 38], [38, 39, 40, 41], [39, 40, 41, 42, 43, 44], [43, 44, 45, 46,
47, 48]]]
MaxOverload= -92400 at machine= M2 and bucket= 45
Best candiddate summary:
  MaxOverload= -92400
  BestNumOverload= 11
  BestTardiness= 0
  BestEarliness 4
  BestMakespan= 90
```

```
  bestList= [[39, 40, 41, 42], [41, 42, 43, 44, 45, 46], [42, 43, 44, 45, 46,
47], [39, 40, 41, 42, 43, 44, 45, 46, 47], [40, 41, 42, 43], [41, 42, 43, 44,
45, 46, 47, 48, 49], [44, 45, 46, 47, 48, 49], [45, 46, 47, 48, 49, 50], [41,
42, 43, 44, 45, 46, 47, 48, 49], [37, 38, 39, 41], [38, 39, 40, 41, 42, 43],
[35, 36, 37, 38], [38, 39, 40, 41], [39, 40, 41, 42, 43, 44], [43, 44, 45, 46,
47, 48]]
  startList= [[39, 40, 41, 42], [41, 42, 43, 44, 45, 46], [42, 43, 44, 45, 46,
47], [39, 40, 41, 42, 43, 44, 45, 46, 47], [40, 41, 42, 43], [41, 42, 43, 44,
45, 46, 47, 48, 49], [44, 45, 46, 47, 48, 49], [45, 46, 47, 48, 49, 50], [41,
42, 43, 44, 45, 46, 47, 48, 49], [37, 38, 39, 41], [38, 39, 40, 41, 42, 43],
[35, 36, 37, 38], [38, 39, 40, 41], [39, 40, 41, 42, 43, 44], [43, 44, 45, 46,
47, 48]]
--------------------------
Best solution summary:
  bestList= [[39, 40, 41, 42], [41, 42, 43, 44, 45, 46], [42, 43, 44, 45, 46,
47], [39, 40, 41, 42, 43, 44, 45, 46, 47], [40, 41, 42, 43], [41, 42, 43, 44,
45, 46, 47, 48, 49], [44, 45, 46, 47, 48, 49], [45, 46, 47, 48, 49, 50], [41,
42, 43, 44, 45, 46, 47, 48, 49], [37, 38, 39, 41], [38, 39, 40, 41, 42, 43],
[35, 36, 37, 38], [38, 39, 40, 41], [39, 40, 41, 42, 43, 44], [43, 44, 45, 46,
47, 48]]
  FinalBestMaxOverload =-92400
  FinalBestNumOverload =11
  FinalBestTardiness =0
  FinalBestEarliness =4
  FinalBestMakespan =90
  Iteration =0
************************************
Iteration(OIA) =1
************************************
jobList= [[39, 40, 41, 42], [41, 42, 43, 44, 45, 46], [42, 43, 44, 45, 46,
47], [39, 40, 41, 42, 43, 44, 45, 46, 47], [40, 41, 42, 43], [41, 42, 43, 44,
45, 46, 47, 48, 49], [44, 45, 46, 47, 48, 49], [45, 46, 47, 48, 49, 50], [41,
42, 43, 44, 45, 46, 47, 48, 49], [37, 38, 39, 41], [38, 39, 40, 41, 42, 43],
[35, 36, 37, 38], [38, 39, 40, 41], [39, 40, 41, 42, 43, 44], [43, 44, 45, 46,
47, 48]]
startList= [[39, 40, 41, 42], [41, 42, 43, 44, 45, 46], [42, 43, 44, 45, 46,
47], [39, 40, 41, 42, 43, 44, 45, 46, 47], [40, 41, 42, 43], [41, 42, 43, 44,
45, 46, 47, 48, 49], [44, 45, 46, 47, 48, 49], [45, 46, 47, 48, 49, 50], [41,
42, 43, 44, 45, 46, 47, 48, 49], [37, 38, 39, 41], [38, 39, 40, 41, 42, 43],
[35, 36, 37, 38], [38, 39, 40, 41], [39, 40, 41, 42, 43, 44], [43, 44, 45, 46,
47, 48]]
MaxOverload= -92400 at machine= M2 and bucket= 45
newPSTList= [[[39, 40, 41, 42], [41, 42, 43, 44, 45, 46], [42, 43, 44, 45, 46,
47], [39, 40, 41, 42, 43, 44, 45, 46, 47], [40, 41, 42, 43], [34, 35, 36, 37,
39, 46, 47, 48, 49], [44, 45, 46, 47, 48, 49], [45, 46, 47, 48, 49, 50], [41,
42, 43, 44, 45, 46, 47, 48, 49], [37, 38, 39, 41], [38, 39, 40, 41, 42, 43],
[35, 36, 37, 38], [38, 39, 40, 41], [39, 40, 41, 42, 43, 44], [43, 44, 45, 46,
47, 48]]]
MaxOverload= -53000 at machine= M2 and bucket= 43
newPSTList= [[[39, 40, 41, 42], [41, 42, 43, 44, 45, 46], [42, 43, 44, 45, 46,
47], [39, 40, 41, 42, 43, 44, 45, 46, 47], [40, 41, 42, 43], [34, 35, 36, 37,
39, 46, 47, 48, 49], [44, 45, 46, 47, 48, 49], [45, 46, 47, 48, 49, 50], [41,
42, 43, 44, 45, 46, 47, 48, 49], [37, 38, 39, 41], [38, 39, 40, 41, 42, 43],
[35, 36, 37, 38], [38, 39, 40, 41], [39, 40, 41, 42, 43, 44], [43, 44, 45, 46,
47, 48]], [[39, 40, 41, 42], [41, 42, 43, 44, 45, 46], [42, 43, 44, 45, 46,
47], [39, 40, 41, 42, 43, 44, 45, 46, 47], [40, 41, 42, 43], [41, 42, 43, 44,
45, 46, 47, 48, 49], [37, 39, 46, 47, 48, 49], [45, 46, 47, 48, 49, 50], [41,
42, 43, 44, 45, 46, 47, 48, 49], [37, 38, 39, 41], [38, 39, 40, 41, 42, 43],
[35, 36, 37, 38], [38, 39, 40, 41], [39, 40, 41, 42, 43, 44], [43, 44, 45, 46,
47, 48]]]
MaxOverload= -53000 at machine= M2 and bucket= 43
newPSTList= [[[39, 40, 41, 42], [41, 42, 43, 44, 45, 46], [42, 43, 44, 45, 46,
47], [39, 40, 41, 42, 43, 44, 45, 46, 47], [40, 41, 42, 43], [34, 35, 36, 37,
```

```
39, 46, 47, 48, 49], [44, 45, 46, 47, 48, 49], [45, 46, 47, 48, 49, 50], [41,
42, 43, 44, 45, 46, 47, 48, 49], [37, 38, 39, 41], [38, 39, 40, 41, 42, 43],
[35, 36, 37, 38], [38, 39, 40, 41], [39, 40, 41, 42, 43, 44], [43, 44, 45, 46,
47, 48]], [[39, 40, 41, 42], [41, 42, 43, 44, 45, 46], [42, 43, 44, 45, 46,
47], [39, 40, 41, 42, 43, 44, 45, 46, 47], [40, 41, 42, 43], [41, 42, 43, 44,
45, 46, 47, 48, 49], [37, 39, 46, 47, 48, 49], [45, 46, 47, 48, 49, 50], [41,
42, 43, 44, 45, 46, 47, 48, 49], [37, 38, 39, 41], [38, 39, 40, 41, 42, 43],
[35, 36, 37, 38], [38, 39, 40, 41], [39, 40, 41, 42, 43, 44], [43, 44, 45, 46,
47, 48]], [[39, 40, 41, 42], [41, 42, 43, 44, 45, 46], [42, 43, 44, 45, 46,
47], [39, 40, 41, 42, 43, 44, 45, 46, 47], [40, 41, 42, 43], [41, 42, 43, 44,
45, 46, 47, 48, 49], [44, 45, 46, 47, 48, 49], [45, 46, 47, 48, 49, 50], [34,
35, 36, 37, 39, 46, 47, 48, 49], [37, 38, 39, 41], [38, 39, 40, 41, 42, 43],
[35, 36, 37, 38], [38, 39, 40, 41], [39, 40, 41, 42, 43, 44], [43, 44, 45, 46,
47, 48]]]
MaxOverload= -53000 at machine= M2 and bucket= 43
Best candiddate summary:
  MaxOverload= -53000
  BestNumOverload= 12
  BestTardiness= 0
  BestEarliness 4
  BestMakespan= 97
  bestList= [[39, 40, 41, 42], [41, 42, 43, 44, 45, 46], [42, 43, 44, 45, 46,
47], [39, 40, 41, 42, 43, 44, 45, 46, 47], [40, 41, 42, 43], [34, 35, 36, 37,
39, 46, 47, 48, 49], [44, 45, 46, 47, 48, 49], [45, 46, 47, 48, 49, 50], [41,
42, 43, 44, 45, 46, 47, 48, 49], [37, 38, 39, 41], [38, 39, 40, 41, 42, 43],
[35, 36, 37, 38], [38, 39, 40, 41], [39, 40, 41, 42, 43, 44], [43, 44, 45, 46,
47, 48]]
  startList= [[39, 40, 41, 42], [41, 42, 43, 44, 45, 46], [42, 43, 44, 45, 46,
47], [39, 40, 41, 42, 43, 44, 45, 46, 47], [40, 41, 42, 43], [34, 35, 36, 37,
39, 46, 47, 48, 49], [44, 45, 46, 47, 48, 49], [45, 46, 47, 48, 49, 50], [41,
42, 43, 44, 45, 46, 47, 48, 49], [37, 38, 39, 41], [38, 39, 40, 41, 42, 43],
[35, 36, 37, 38], [38, 39, 40, 41], [39, 40, 41, 42, 43, 44], [43, 44, 45, 46,
47, 48]]
--------------------------
Best solution summary:
  bestList= [[39, 40, 41, 42], [41, 42, 43, 44, 45, 46], [42, 43, 44, 45, 46,
47], [39, 40, 41, 42, 43, 44, 45, 46, 47], [40, 41, 42, 43], [34, 35, 36, 37,
39, 46, 47, 48, 49], [44, 45, 46, 47, 48, 49], [45, 46, 47, 48, 49, 50], [41,
42, 43, 44, 45, 46, 47, 48, 49], [37, 38, 39, 41], [38, 39, 40, 41, 42, 43],
[35, 36, 37, 38], [38, 39, 40, 41], [39, 40, 41, 42, 43, 44], [43, 44, 45, 46,
47, 48]]
  FinalBestMaxOverload =-53000
  FinalBestNumOverload =12
  FinalBestTardiness =0
  FinalBestEarliness =4
  FinalBestMakespan =97
  Iteration =1
************************************
Iteration(OIA) =2
************************************
jobList= [[39, 40, 41, 42], [41, 42, 43, 44, 45, 46], [42, 43, 44, 45, 46,
47], [39, 40, 41, 42, 43, 44, 45, 46, 47], [40, 41, 42, 43], [34, 35, 36, 37,
39, 46, 47, 48, 49], [44, 45, 46, 47, 48, 49], [45, 46, 47, 48, 49, 50], [41,
42, 43, 44, 45, 46, 47, 48, 49], [37, 38, 39, 41], [38, 39, 40, 41, 42, 43],
[35, 36, 37, 38], [38, 39, 40, 41], [39, 40, 41, 42, 43, 44], [43, 44, 45, 46,
47, 48]]
startList= [[39, 40, 41, 42], [41, 42, 43, 44, 45, 46], [42, 43, 44, 45, 46,
47], [39, 40, 41, 42, 43, 44, 45, 46, 47], [40, 41, 42, 43], [34, 35, 36, 37,
39, 46, 47, 48, 49], [44, 45, 46, 47, 48, 49], [45, 46, 47, 48, 49, 50], [41,
42, 43, 44, 45, 46, 47, 48, 49], [37, 38, 39, 41], [38, 39, 40, 41, 42, 43],
[35, 36, 37, 38], [38, 39, 40, 41], [39, 40, 41, 42, 43, 44], [43, 44, 45, 46,
47, 48]]
MaxOverload= -53000 at machine= M2 and bucket= 43
```

```
newPSTList= [[[39, 40, 41, 42], [41, 42, 43, 44, 45, 46], [37, 38, 44, 45, 46,
47], [39, 40, 41, 42, 43, 44, 45, 46, 47], [40, 41, 42, 43], [34, 35, 36, 37,
39, 46, 47, 48, 49], [44, 45, 46, 47, 48, 49], [45, 46, 47, 48, 49, 50], [41,
42, 43, 44, 45, 46, 47, 48, 49], [37, 38, 39, 41], [38, 39, 40, 41, 42, 43],
[35, 36, 37, 38], [38, 39, 40, 41], [39, 40, 41, 42, 43, 44], [43, 44, 45, 46,
47, 48]]]
MaxOverload= -39200 at machine= M13 and bucket= 48
newPSTList= [[[39, 40, 41, 42], [41, 42, 43, 44, 45, 46], [37, 38, 44, 45, 46,
47], [39, 40, 41, 42, 43, 44, 45, 46, 47], [40, 41, 42, 43], [34, 35, 36, 37,
39, 46, 47, 48, 49], [44, 45, 46, 47, 48, 49], [45, 46, 47, 48, 49, 50], [41,
42, 43, 44, 45, 46, 47, 48, 49], [37, 38, 39, 41], [38, 39, 40, 41, 42, 43],
[35, 36, 37, 38], [38, 39, 40, 41], [39, 40, 41, 42, 43, 44], [43, 44, 45, 46,
47, 48]], [[39, 40, 41, 42], [41, 42, 43, 44, 45, 46], [42, 43, 44, 45, 46,
47], [32, 33, 34, 35, 38, 44, 45, 46, 47], [40, 41, 42, 43], [34, 35, 36, 37,
39, 46, 47, 48, 49], [44, 45, 46, 47, 48, 49], [45, 46, 47, 48, 49, 50], [41,
42, 43, 44, 45, 46, 47, 48, 49], [37, 38, 39, 41], [38, 39, 40, 41, 42, 43],
[35, 36, 37, 38], [38, 39, 40, 41], [39, 40, 41, 42, 43, 44], [43, 44, 45, 46,
47, 48]]]
MaxOverload= -39200 at machine= M13 and bucket= 48
newPSTList= [[[39, 40, 41, 42], [41, 42, 43, 44, 45, 46], [37, 38, 44, 45, 46,
47], [39, 40, 41, 42, 43, 44, 45, 46, 47], [40, 41, 42, 43], [34, 35, 36, 37,
39, 46, 47, 48, 49], [44, 45, 46, 47, 48, 49], [45, 46, 47, 48, 49, 50], [41,
42, 43, 44, 45, 46, 47, 48, 49], [37, 38, 39, 41], [38, 39, 40, 41, 42, 43],
[35, 36, 37, 38], [38, 39, 40, 41], [39, 40, 41, 42, 43, 44], [43, 44, 45, 46,
47, 48]], [[39, 40, 41, 42], [41, 42, 43, 44, 45, 46], [42, 43, 44, 45, 46,
47], [32, 33, 34, 35, 38, 44, 45, 46, 47], [40, 41, 42, 43], [34, 35, 36, 37,
39, 46, 47, 48, 49], [44, 45, 46, 47, 48, 49], [45, 46, 47, 48, 49, 50], [41,
42, 43, 44, 45, 46, 47, 48, 49], [37, 38, 39, 41], [38, 39, 40, 41, 42, 43],
[35, 36, 37, 38], [38, 39, 40, 41], [39, 40, 41, 42, 43, 44], [43, 44, 45, 46,
47, 48]], [[39, 40, 41, 42], [41, 42, 43, 44, 45, 46], [42, 43, 44, 45, 46,
47], [39, 40, 41, 42, 43, 44, 45, 46, 47], [34, 35, 38, 41], [34, 35, 36, 37,
39, 46, 47, 48, 49], [44, 45, 46, 47, 48, 49], [45, 46, 47, 48, 49, 50], [41,
42, 43, 44, 45, 46, 47, 48, 49], [37, 38, 39, 41], [38, 39, 40, 41, 42, 43],
[35, 36, 37, 38], [38, 39, 40, 41], [39, 40, 41, 42, 43, 44], [43, 44, 45, 46,
47, 48]]]
MaxOverload= -39200 at machine= M13 and bucket= 48
Best candiddate summary:
  MaxOverload= -39200
  BestNumOverload= 7
  BestTardiness= 0
  BestEarliness 4
  BestMakespan= 102
  bestList= [[39, 40, 41, 42], [41, 42, 43, 44, 45, 46], [37, 38, 44, 45, 46,
47], [39, 40, 41, 42, 43, 44, 45, 46, 47], [40, 41, 42, 43], [34, 35, 36, 37,
39, 46, 47, 48, 49], [44, 45, 46, 47, 48, 49], [45, 46, 47, 48, 49, 50], [41,
42, 43, 44, 45, 46, 47, 48, 49], [37, 38, 39, 41], [38, 39, 40, 41, 42, 43],
[35, 36, 37, 38], [38, 39, 40, 41], [39, 40, 41, 42, 43, 44], [43, 44, 45, 46,
47, 48]]
  startList= [[39, 40, 41, 42], [41, 42, 43, 44, 45, 46], [37, 38, 44, 45, 46,
47], [39, 40, 41, 42, 43, 44, 45, 46, 47], [40, 41, 42, 43], [34, 35, 36, 37,
39, 46, 47, 48, 49], [44, 45, 46, 47, 48, 49], [45, 46, 47, 48, 49, 50], [41,
42, 43, 44, 45, 46, 47, 48, 49], [37, 38, 39, 41], [38, 39, 40, 41, 42, 43],
[35, 36, 37, 38], [38, 39, 40, 41], [39, 40, 41, 42, 43, 44], [43, 44, 45, 46,
47, 48]]
---------------------------
Best solution summary:
  bestList= [[39, 40, 41, 42], [41, 42, 43, 44, 45, 46], [37, 38, 44, 45, 46,
47], [39, 40, 41, 42, 43, 44, 45, 46, 47], [40, 41, 42, 43], [34, 35, 36, 37,
39, 46, 47, 48, 49], [44, 45, 46, 47, 48, 49], [45, 46, 47, 48, 49, 50], [41,
42, 43, 44, 45, 46, 47, 48, 49], [37, 38, 39, 41], [38, 39, 40, 41, 42, 43],
[35, 36, 37, 38], [38, 39, 40, 41], [39, 40, 41, 42, 43, 44], [43, 44, 45, 46,
47, 48]]
  FinalBestMaxOverload =-39200
```

```
  FinalBestNumOverload =7
  FinalBestTardiness =0
  FinalBestEarliness =4
  FinalBestMakespan =102
  Iteration =2
************************************
Iteration(OIA) =3
************************************
jobList= [[39, 40, 41, 42], [41, 42, 43, 44, 45, 46], [37, 38, 44, 45, 46,
47], [39, 40, 41, 42, 43, 44, 45, 46, 47], [40, 41, 42, 43], [34, 35, 36, 37,
39, 46, 47, 48, 49], [44, 45, 46, 47, 48, 49], [45, 46, 47, 48, 49, 50], [41,
42, 43, 44, 45, 46, 47, 48, 49], [37, 38, 39, 41], [38, 39, 40, 41, 42, 43],
[35, 36, 37, 38], [38, 39, 40, 41], [39, 40, 41, 42, 43, 44], [43, 44, 45, 46,
47, 48]]
startList= [[39, 40, 41, 42], [41, 42, 43, 44, 45, 46], [37, 38, 44, 45, 46,
47], [39, 40, 41, 42, 43, 44, 45, 46, 47], [40, 41, 42, 43], [34, 35, 36, 37,
39, 46, 47, 48, 49], [44, 45, 46, 47, 48, 49], [45, 46, 47, 48, 49, 50], [41,
42, 43, 44, 45, 46, 47, 48, 49], [37, 38, 39, 41], [38, 39, 40, 41, 42, 43],
[35, 36, 37, 38], [38, 39, 40, 41], [39, 40, 41, 42, 43, 44], [43, 44, 45, 46,
47, 48]]
MaxOverload= -39200 at machine= M13 and bucket= 48
newPSTList= [[[39, 40, 41, 42], [41, 42, 43, 44, 45, 46], [37, 38, 44, 45, 46,
47], [39, 40, 41, 42, 43, 44, 45, 46, 47], [40, 41, 42, 43], [32, 33, 34, 35,
38, 45, 46, 47, 49], [44, 45, 46, 47, 48, 49], [45, 46, 47, 48, 49, 50], [41,
42, 43, 44, 45, 46, 47, 48, 49], [37, 38, 39, 41], [38, 39, 40, 41, 42, 43],
[35, 36, 37, 38], [38, 39, 40, 41], [39, 40, 41, 42, 43, 44], [43, 44, 45, 46,
47, 48]]]
MaxOverload= -53000 at machine= M2 and bucket= 38
newPSTList= [[[39, 40, 41, 42], [41, 42, 43, 44, 45, 46], [37, 38, 44, 45, 46,
47], [39, 40, 41, 42, 43, 44, 45, 46, 47], [40, 41, 42, 43], [32, 33, 34, 35,
38, 45, 46, 47, 49], [44, 45, 46, 47, 48, 49], [45, 46, 47, 48, 49, 50], [41,
42, 43, 44, 45, 46, 47, 48, 49], [37, 38, 39, 41], [38, 39, 40, 41, 42, 43],
[35, 36, 37, 38], [38, 39, 40, 41], [39, 40, 41, 42, 43, 44], [43, 44, 45, 46,
47, 48]], [[39, 40, 41, 42], [41, 42, 43, 44, 45, 46], [37, 38, 44, 45, 46,
47], [39, 40, 41, 42, 43, 44, 45, 46, 47], [40, 41, 42, 43], [34, 35, 36, 37,
39, 46, 47, 48, 49], [44, 45, 46, 47, 48, 49], [45, 46, 47, 48, 49, 50], [32,
33, 34, 35, 38, 45, 46, 47, 49], [37, 38, 39, 41], [38, 39, 40, 41, 42, 43],
[35, 36, 37, 38], [38, 39, 40, 41], [39, 40, 41, 42, 43, 44], [43, 44, 45, 46,
47, 48]]]
MaxOverload= -53000 at machine= M2 and bucket= 38
Best candiddate summary:
  MaxOverload= -53000
  BestNumOverload= 7
  BestTardiness= 0
  BestEarliness 4
  BestMakespan= 104
  bestList= [[39, 40, 41, 42], [41, 42, 43, 44, 45, 46], [37, 38, 44, 45, 46,
47], [39, 40, 41, 42, 43, 44, 45, 46, 47], [40, 41, 42, 43], [34, 35, 36, 37,
39, 46, 47, 48, 49], [44, 45, 46, 47, 48, 49], [45, 46, 47, 48, 49, 50], [41,
42, 43, 44, 45, 46, 47, 48, 49], [37, 38, 39, 41], [38, 39, 40, 41, 42, 43],
[35, 36, 37, 38], [38, 39, 40, 41], [39, 40, 41, 42, 43, 44], [43, 44, 45, 46,
47, 48]]
  startList= [[39, 40, 41, 42], [41, 42, 43, 44, 45, 46], [37, 38, 44, 45, 46,
47], [39, 40, 41, 42, 43, 44, 45, 46, 47], [40, 41, 42, 43], [32, 33, 34, 35,
38, 45, 46, 47, 49], [44, 45, 46, 47, 48, 49], [45, 46, 47, 48, 49, 50], [41,
42, 43, 44, 45, 46, 47, 48, 49], [37, 38, 39, 41], [38, 39, 40, 41, 42, 43],
[35, 36, 37, 38], [38, 39, 40, 41], [39, 40, 41, 42, 43, 44], [43, 44, 45, 46,
47, 48]]
---------------------------
Best solution summary:
  bestList= [[39, 40, 41, 42], [41, 42, 43, 44, 45, 46], [37, 38, 44, 45, 46,
47], [39, 40, 41, 42, 43, 44, 45, 46, 47], [40, 41, 42, 43], [34, 35, 36, 37,
39, 46, 47, 48, 49], [44, 45, 46, 47, 48, 49], [45, 46, 47, 48, 49, 50], [41,
```

```
42, 43, 44, 45, 46, 47, 48, 49], [37, 38, 39, 41], [38, 39, 40, 41, 42, 43],
[35, 36, 37, 38], [38, 39, 40, 41], [39, 40, 41, 42, 43, 44], [43, 44, 45, 46,
47, 48]]
  FinalBestMaxOverload =-39200
  FinalBestNumOverload =7
  FinalBestTardiness =0
  FinalBestEarliness =4
  FinalBestMakespan =102
  Iteration =3
************************************
Iteration(OIA) =4
************************************
jobList= [[39, 40, 41, 42], [41, 42, 43, 44, 45, 46], [37, 38, 44, 45, 46,
47], [39, 40, 41, 42, 43, 44, 45, 46, 47], [40, 41, 42, 43], [32, 33, 34, 35,
38, 45, 46, 47, 49], [44, 45, 46, 47, 48, 49], [45, 46, 47, 48, 49, 50], [41,
42, 43, 44, 45, 46, 47, 48, 49], [37, 38, 39, 41], [38, 39, 40, 41, 42, 43],
[35, 36, 37, 38], [38, 39, 40, 41], [39, 40, 41, 42, 43, 44], [43, 44, 45, 46,
47, 48]]
startList= [[39, 40, 41, 42], [41, 42, 43, 44, 45, 46], [37, 38, 44, 45, 46,
47], [39, 40, 41, 42, 43, 44, 45, 46, 47], [40, 41, 42, 43], [32, 33, 34, 35,
38, 45, 46, 47, 49], [44, 45, 46, 47, 48, 49], [45, 46, 47, 48, 49, 50], [41,
42, 43, 44, 45, 46, 47, 48, 49], [37, 38, 39, 41], [38, 39, 40, 41, 42, 43],
[35, 36, 37, 38], [38, 39, 40, 41], [39, 40, 41, 42, 43, 44], [43, 44, 45, 46,
47, 48]]
MaxOverload= -53000 at machine= M2 and bucket= 38
newPSTList= [[[39, 40, 41, 42], [41, 42, 43, 44, 45, 46], [36, 37, 44, 45, 46,
47], [39, 40, 41, 42, 43, 44, 45, 46, 47], [40, 41, 42, 43], [32, 33, 34, 35,
38, 45, 46, 47, 49], [44, 45, 46, 47, 48, 49], [45, 46, 47, 48, 49, 50], [41,
42, 43, 44, 45, 46, 47, 48, 49], [37, 38, 39, 41], [38, 39, 40, 41, 42, 43],
[35, 36, 37, 38], [38, 39, 40, 41], [39, 40, 41, 42, 43, 44], [43, 44, 45, 46,
47, 48]]]
MaxOverload= -34600 at machine= M15 and bucket= 49
newPSTList= [[[39, 40, 41, 42], [41, 42, 43, 44, 45, 46], [36, 37, 44, 45, 46,
47], [39, 40, 41, 42, 43, 44, 45, 46, 47], [40, 41, 42, 43], [32, 33, 34, 35,
38, 45, 46, 47, 49], [44, 45, 46, 47, 48, 49], [45, 46, 47, 48, 49, 50], [41,
42, 43, 44, 45, 46, 47, 48, 49], [37, 38, 39, 41], [38, 39, 40, 41, 42, 43],
[35, 36, 37, 38], [38, 39, 40, 41], [39, 40, 41, 42, 43, 44], [43, 44, 45, 46,
47, 48]], [[39, 40, 41, 42], [41, 42, 43, 44, 45, 46], [37, 38, 44, 45, 46,
47], [39, 40, 41, 42, 43, 44, 45, 46, 47], [40, 41, 42, 43], [31, 32, 33, 34,
37, 45, 46, 47, 49], [44, 45, 46, 47, 48, 49], [45, 46, 47, 48, 49, 50], [41,
42, 43, 44, 45, 46, 47, 48, 49], [37, 38, 39, 41], [38, 39, 40, 41, 42, 43],
[35, 36, 37, 38], [38, 39, 40, 41], [39, 40, 41, 42, 43, 44], [43, 44, 45, 46,
47, 48]]]
MaxOverload= -34600 at machine= M15 and bucket= 49
newPSTList= [[[39, 40, 41, 42], [41, 42, 43, 44, 45, 46], [36, 37, 44, 45, 46,
47], [39, 40, 41, 42, 43, 44, 45, 46, 47], [40, 41, 42, 43], [32, 33, 34, 35,
38, 45, 46, 47, 49], [44, 45, 46, 47, 48, 49], [45, 46, 47, 48, 49, 50], [41,
42, 43, 44, 45, 46, 47, 48, 49], [37, 38, 39, 41], [38, 39, 40, 41, 42, 43],
[35, 36, 37, 38], [38, 39, 40, 41], [39, 40, 41, 42, 43, 44], [43, 44, 45, 46,
47, 48]], [[39, 40, 41, 42], [41, 42, 43, 44, 45, 46], [37, 38, 44, 45, 46,
47], [39, 40, 41, 42, 43, 44, 45, 46, 47], [40, 41, 42, 43], [31, 32, 33, 34,
37, 45, 46, 47, 49], [44, 45, 46, 47, 48, 49], [45, 46, 47, 48, 49, 50], [41,
42, 43, 44, 45, 46, 47, 48, 49], [37, 38, 39, 41], [38, 39, 40, 41, 42, 43],
[35, 36, 37, 38], [38, 39, 40, 41], [39, 40, 41, 42, 43, 44], [43, 44, 45, 46,
47, 48]], [[39, 40, 41, 42], [41, 42, 43, 44, 45, 46], [37, 38, 44, 45, 46,
47], [39, 40, 41, 42, 43, 44, 45, 46, 47], [40, 41, 42, 43], [32, 33, 34, 35,
38, 45, 46, 47, 49], [44, 45, 46, 47, 48, 49], [45, 46, 47, 48, 49, 50], [41,
42, 43, 44, 45, 46, 47, 48, 49], [37, 38, 39, 41], [38, 39, 40, 41, 42, 43],
[33, 34, 36, 37], [38, 39, 40, 41], [39, 40, 41, 42, 43, 44], [43, 44, 45, 46,
47, 48]]]
MaxOverload= -34600 at machine= M15 and bucket= 49
Best candiddate summary:
  MaxOverload= -34600
```

```
  BestNumOverload= 3
  BestTardiness= 0
  BestEarliness 4
  BestMakespan= 105
  bestList= [[39, 40, 41, 42], [41, 42, 43, 44, 45, 46], [36, 37, 44, 45, 46,
47], [39, 40, 41, 42, 43, 44, 45, 46, 47], [40, 41, 42, 43], [32, 33, 34, 35,
38, 45, 46, 47, 49], [44, 45, 46, 47, 48, 49], [45, 46, 47, 48, 49, 50], [41,
42, 43, 44, 45, 46, 47, 48, 49], [37, 38, 39, 41], [38, 39, 40, 41, 42, 43],
[35, 36, 37, 38], [38, 39, 40, 41], [39, 40, 41, 42, 43, 44], [43, 44, 45, 46,
47, 48]]
  startList= [[39, 40, 41, 42], [41, 42, 43, 44, 45, 46], [36, 37, 44, 45, 46,
47], [39, 40, 41, 42, 43, 44, 45, 46, 47], [40, 41, 42, 43], [32, 33, 34, 35,
38, 45, 46, 47, 49], [44, 45, 46, 47, 48, 49], [45, 46, 47, 48, 49, 50], [41,
42, 43, 44, 45, 46, 47, 48, 49], [37, 38, 39, 41], [38, 39, 40, 41, 42, 43],
[35, 36, 37, 38], [38, 39, 40, 41], [39, 40, 41, 42, 43, 44], [43, 44, 45, 46,
47, 48]]
---------------------------
Best solution summary:
  bestList= [[39, 40, 41, 42], [41, 42, 43, 44, 45, 46], [36, 37, 44, 45, 46,
47], [39, 40, 41, 42, 43, 44, 45, 46, 47], [40, 41, 42, 43], [32, 33, 34, 35,
38, 45, 46, 47, 49], [44, 45, 46, 47, 48, 49], [45, 46, 47, 48, 49, 50], [41,
42, 43, 44, 45, 46, 47, 48, 49], [37, 38, 39, 41], [38, 39, 40, 41, 42, 43],
[35, 36, 37, 38], [38, 39, 40, 41], [39, 40, 41, 42, 43, 44], [43, 44, 45, 46,
47, 48]]
  FinalBestMaxOverload =-34600
  FinalBestNumOverload =3
  FinalBestTardiness =0
  FinalBestEarliness =4
  FinalBestMakespan =105
  Iteration =4

|
|
|
|
|
|

************************************
Iteration(OIA) =11
************************************
jobList= [[39, 40, 41, 42], [41, 42, 43, 44, 45, 46], [36, 37, 44, 45, 46,
47], [39, 40, 41, 42, 43, 44, 45, 46, 47], [33, 34, 38, 43], [34, 35, 36, 37,
40, 43, 44, 45, 48], [44, 45, 46, 47, 48, 49], [45, 46, 47, 48, 49, 50], [32,
33, 34, 35, 41, 46, 47, 48, 49], [37, 38, 39, 41], [38, 39, 40, 41, 42, 43],
[35, 36, 37, 38], [38, 39, 40, 41], [37, 38, 41, 42, 43, 44], [43, 44, 45, 46,
47, 48]]
startList= [[39, 40, 41, 42], [41, 42, 43, 44, 45, 46], [36, 37, 44, 45, 46,
47], [39, 40, 41, 42, 43, 44, 45, 46, 47], [33, 34, 38, 43], [34, 35, 36, 37,
40, 43, 44, 45, 48], [44, 45, 46, 47, 48, 49], [45, 46, 47, 48, 49, 50], [32,
33, 34, 35, 41, 46, 47, 48, 49], [37, 38, 39, 41], [38, 39, 40, 41, 42, 43],
[35, 36, 37, 38], [38, 39, 40, 41], [37, 38, 41, 42, 43, 44], [43, 44, 45, 46,
47, 48]]
MaxOverload= -13900 at machine= M2 and bucket= 41
newPSTList= [[[39, 40, 41, 42], [41, 42, 43, 44, 45, 46], [36, 37, 44, 45, 46,
47], [39, 40, 41, 42, 43, 44, 45, 46, 47], [33, 34, 38, 43], [34, 35, 36, 37,
40, 43, 44, 45, 48], [44, 45, 46, 47, 48, 49], [45, 46, 47, 48, 49, 50], [31,
32, 33, 34, 38, 46, 47, 48, 49], [37, 38, 39, 41], [38, 39, 40, 41, 42, 43],
[35, 36, 37, 38], [38, 39, 40, 41], [37, 38, 41, 42, 43, 44], [43, 44, 45, 46,
47, 48]]]
MaxOverload= -53000 at machine= M2 and bucket= 38
newPSTList= [[[39, 40, 41, 42], [41, 42, 43, 44, 45, 46], [36, 37, 44, 45, 46,
47], [39, 40, 41, 42, 43, 44, 45, 46, 47], [33, 34, 38, 43], [34, 35, 36, 37,
```

```
40, 43, 44, 45, 48], [44, 45, 46, 47, 48, 49], [45, 46, 47, 48, 49, 50], [31,
32, 33, 34, 38, 46, 47, 48, 49], [37, 38, 39, 41], [38, 39, 40, 41, 42, 43],
[35, 36, 37, 38], [38, 39, 40, 41], [37, 38, 41, 42, 43, 44], [43, 44, 45, 46,
47, 48]], [[39, 40, 41, 42], [41, 42, 43, 44, 45, 46], [36, 37, 44, 45, 46,
47], [39, 40, 41, 42, 43, 44, 45, 46, 47], [33, 34, 38, 43], [34, 35, 36, 37,
40, 43, 44, 45, 48], [44, 45, 46, 47, 48, 49], [45, 46, 47, 48, 49, 50], [32,
33, 34, 35, 41, 46, 47, 48, 49], [32, 33, 36, 40], [38, 39, 40, 41, 42, 43],
[35, 36, 37, 38], [38, 39, 40, 41], [37, 38, 41, 42, 43, 44], [43, 44, 45, 46,
47, 48]]]
MaxOverload= 0 at machine= M2 and bucket= 38
MaxOverload= 0
newPSTList= [[[39, 40, 41, 42], [41, 42, 43, 44, 45, 46], [36, 37, 44, 45, 46,
47], [39, 40, 41, 42, 43, 44, 45, 46, 47], [33, 34, 38, 43], [34, 35, 36, 37,
40, 43, 44, 45, 48], [44, 45, 46, 47, 48, 49], [45, 46, 47, 48, 49, 50], [31,
32, 33, 34, 38, 46, 47, 48, 49], [37, 38, 39, 41], [38, 39, 40, 41, 42, 43],
[35, 36, 37, 38], [38, 39, 40, 41], [37, 38, 41, 42, 43, 44], [43, 44, 45, 46,
47, 48]], [[39, 40, 41, 42], [41, 42, 43, 44, 45, 46], [36, 37, 44, 45, 46,
47], [39, 40, 41, 42, 43, 44, 45, 46, 47], [33, 34, 38, 43], [34, 35, 36, 37,
40, 43, 44, 45, 48], [44, 45, 46, 47, 48, 49], [45, 46, 47, 48, 49, 50], [32,
33, 34, 35, 41, 46, 47, 48, 49], [32, 33, 36, 40], [38, 39, 40, 41, 42, 43],
[35, 36, 37, 38], [38, 39, 40, 41], [37, 38, 41, 42, 43, 44], [43, 44, 45, 46,
47, 48]], [[39, 40, 41, 42], [41, 42, 43, 44, 45, 46], [36, 37, 44, 45, 46,
47], [39, 40, 41, 42, 43, 44, 45, 46, 47], [33, 34, 38, 43], [34, 35, 36, 37,
40, 43, 44, 45, 48], [44, 45, 46, 47, 48, 49], [45, 46, 47, 48, 49, 50], [32,
33, 34, 35, 41, 46, 47, 48, 49], [37, 38, 39, 41], [38, 39, 40, 41, 42, 43],
[35, 36, 37, 38], [32, 33, 36, 40], [37, 38, 41, 42, 43, 44], [43, 44, 45, 46,
47, 48]]]
MaxOverload= 0 at machine= M2 and bucket= 38
MaxOverload= 0
Best candiddate summary:
  MaxOverload= -53000
  BestNumOverload= 2
  BestTardiness= 0
  BestEarliness 5
  BestMakespan= 121
  bestList= [[39, 40, 41, 42], [41, 42, 43, 44, 45, 46], [36, 37, 44, 45, 46,
47], [39, 40, 41, 42, 43, 44, 45, 46, 47], [33, 34, 38, 43], [34, 35, 36, 37,
40, 43, 44, 45, 48], [44, 45, 46, 47, 48, 49], [45, 46, 47, 48, 49, 50], [32,
33, 34, 35, 41, 46, 47, 48, 49], [37, 38, 39, 41], [38, 39, 40, 41, 42, 43],
[35, 36, 37, 38], [38, 39, 40, 41], [37, 38, 41, 42, 43, 44], [43, 44, 45, 46,
47, 48]]
  startList= [[39, 40, 41, 42], [41, 42, 43, 44, 45, 46], [36, 37, 44, 45, 46,
47], [39, 40, 41, 42, 43, 44, 45, 46, 47], [33, 34, 38, 43], [34, 35, 36, 37,
40, 43, 44, 45, 48], [44, 45, 46, 47, 48, 49], [45, 46, 47, 48, 49, 50], [32,
33, 34, 35, 41, 46, 47, 48, 49], [37, 38, 39, 41], [38, 39, 40, 41, 42, 43],
[35, 36, 37, 38], [32, 33, 36, 40], [37, 38, 41, 42, 43, 44], [43, 44, 45, 46,
47, 48]]
---------------------------
Best solution summary:
  bestList= [[39, 40, 41, 42], [41, 42, 43, 44, 45, 46], [36, 37, 44, 45, 46,
47], [39, 40, 41, 42, 43, 44, 45, 46, 47], [33, 34, 38, 43], [34, 35, 36, 37,
40, 43, 44, 45, 48], [44, 45, 46, 47, 48, 49], [45, 46, 47, 48, 49, 50], [32,
33, 34, 35, 41, 46, 47, 48, 49], [37, 38, 39, 41], [38, 39, 40, 41, 42, 43],
[35, 36, 37, 38], [38, 39, 40, 41], [37, 38, 41, 42, 43, 44], [43, 44, 45, 46,
47, 48]]
  FinalBestMaxOverload =-13900
  FinalBestNumOverload =1
  FinalBestTardiness =0
  FinalBestEarliness =5
  FinalBestMakespan =120
  Iteration =11
************************************
Iteration(OIA) =12
```

```
************************************
jobList= [[39, 40, 41, 42], [41, 42, 43, 44, 45, 46], [36, 37, 44, 45, 46,
47], [39, 40, 41, 42, 43, 44, 45, 46, 47], [33, 34, 38, 43], [34, 35, 36, 37,
40, 43, 44, 45, 48], [44, 45, 46, 47, 48, 49], [45, 46, 47, 48, 49, 50], [32,
33, 34, 35, 41, 46, 47, 48, 49], [37, 38, 39, 41], [38, 39, 40, 41, 42, 43],
[35, 36, 37, 38], [32, 33, 36, 40], [37, 38, 41, 42, 43, 44], [43, 44, 45, 46,
47, 48]]
startList= [[39, 40, 41, 42], [41, 42, 43, 44, 45, 46], [36, 37, 44, 45, 46,
47], [39, 40, 41, 42, 43, 44, 45, 46, 47], [33, 34, 38, 43], [34, 35, 36, 37,
40, 43, 44, 45, 48], [44, 45, 46, 47, 48, 49], [45, 46, 47, 48, 49, 50], [32,
33, 34, 35, 41, 46, 47, 48, 49], [37, 38, 39, 41], [38, 39, 40, 41, 42, 43],
[35, 36, 37, 38], [32, 33, 36, 40], [37, 38, 41, 42, 43, 44], [43, 44, 45, 46,
47, 48]]
MaxOverload= 0 at machine= M2 and bucket= 38
MaxOverload= 0
newPSTList= [[[39, 40, 41, 42], [41, 42, 43, 44, 45, 46], [36, 37, 44, 45, 46,
47], [39, 40, 41, 42, 43, 44, 45, 46, 47], [33, 34, 38, 43], [34, 35, 36, 37,
40, 43, 44, 45, 48], [44, 45, 46, 47, 48, 49], [45, 46, 47, 48, 49, 50], [32,
33, 34, 35, 41, 46, 47, 48, 49], [37, 38, 39, 41], [38, 39, 40, 41, 42, 43],
[31, 32, 35, 37], [32, 33, 36, 40], [37, 38, 41, 42, 43, 44], [43, 44, 45, 46,
47, 48]]]
MaxOverload= 0 at machine= M2 and bucket= 38
MaxOverload= 0
newPSTList= [[[39, 40, 41, 42], [41, 42, 43, 44, 45, 46], [36, 37, 44, 45, 46,
47], [39, 40, 41, 42, 43, 44, 45, 46, 47], [33, 34, 38, 43], [34, 35, 36, 37,
40, 43, 44, 45, 48], [44, 45, 46, 47, 48, 49], [45, 46, 47, 48, 49, 50], [32,
33, 34, 35, 41, 46, 47, 48, 49], [37, 38, 39, 41], [38, 39, 40, 41, 42, 43],
[31, 32, 35, 37], [32, 33, 36, 40], [37, 38, 41, 42, 43, 44], [43, 44, 45, 46,
47, 48]], [[39, 40, 41, 42], [41, 42, 43, 44, 45, 46], [36, 37, 44, 45, 46,
47], [39, 40, 41, 42, 43, 44, 45, 46, 47], [33, 34, 38, 43], [34, 35, 36, 37,
40, 43, 44, 45, 48], [44, 45, 46, 47, 48, 49], [45, 46, 47, 48, 49, 50], [32,
33, 34, 35, 41, 46, 47, 48, 49], [37, 38, 39, 41], [38, 39, 40, 41, 42, 43],
[35, 36, 37, 38], [32, 33, 36, 40], [35, 36, 41, 42, 43, 44], [43, 44, 45, 46,
47, 48]]]
MaxOverload= 0 at machine= M2 and bucket= 38
MaxOverload= 0
Best candiddate summary:
  MaxOverload= 0
  BestNumOverload= 0
  BestTardiness= 0
  BestEarliness 6
  BestMakespan= 127
  bestList= [[39, 40, 41, 42], [41, 42, 43, 44, 45, 46], [36, 37, 44, 45, 46,
47], [39, 40, 41, 42, 43, 44, 45, 46, 47], [33, 34, 38, 43], [34, 35, 36, 37,
40, 43, 44, 45, 48], [44, 45, 46, 47, 48, 49], [45, 46, 47, 48, 49, 50], [32,
33, 34, 35, 41, 46, 47, 48, 49], [37, 38, 39, 41], [38, 39, 40, 41, 42, 43],
[35, 36, 37, 38], [32, 33, 36, 40], [35, 36, 41, 42, 43, 44], [43, 44, 45, 46,
47, 48]]
  startList= [[39, 40, 41, 42], [41, 42, 43, 44, 45, 46], [36, 37, 44, 45, 46,
47], [39, 40, 41, 42, 43, 44, 45, 46, 47], [33, 34, 38, 43], [34, 35, 36, 37,
40, 43, 44, 45, 48], [44, 45, 46, 47, 48, 49], [45, 46, 47, 48, 49, 50], [32,
33, 34, 35, 41, 46, 47, 48, 49], [37, 38, 39, 41], [38, 39, 40, 41, 42, 43],
[35, 36, 37, 38], [32, 33, 36, 40], [35, 36, 41, 42, 43, 44], [43, 44, 45, 46,
47, 48]]
---------------------------
Best solution summary:
  bestList= [[39, 40, 41, 42], [41, 42, 43, 44, 45, 46], [36, 37, 44, 45, 46,
47], [39, 40, 41, 42, 43, 44, 45, 46, 47], [33, 34, 38, 43], [34, 35, 36, 37,
40, 43, 44, 45, 48], [44, 45, 46, 47, 48, 49], [45, 46, 47, 48, 49, 50], [32,
33, 34, 35, 41, 46, 47, 48, 49], [37, 38, 39, 41], [38, 39, 40, 41, 42, 43],
[35, 36, 37, 38], [32, 33, 36, 40], [35, 36, 41, 42, 43, 44], [43, 44, 45, 46,
47, 48]]
  FinalBestMaxOverload =0
```

```
  FinalBestNumOverload =0
  FinalBestTardiness =0
  FinalBestEarliness =6
  FinalBestMakespan =127
  Iteration =12
************************************
Iteration(OIA) =13
************************************
jobList= [[39, 40, 41, 42], [41, 42, 43, 44, 45, 46], [36, 37, 44, 45, 46,
47], [39, 40, 41, 42, 43, 44, 45, 46, 47], [33, 34, 38, 43], [34, 35, 36, 37,
40, 43, 44, 45, 48], [44, 45, 46, 47, 48, 49], [45, 46, 47, 48, 49, 50], [32,
33, 34, 35, 41, 46, 47, 48, 49], [37, 38, 39, 41], [38, 39, 40, 41, 42, 43],
[35, 36, 37, 38], [32, 33, 36, 40], [35, 36, 41, 42, 43, 44], [43, 44, 45, 46,
47, 48]]
startList= [[39, 40, 41, 42], [41, 42, 43, 44, 45, 46], [36, 37, 44, 45, 46,
47], [39, 40, 41, 42, 43, 44, 45, 46, 47], [33, 34, 38, 43], [34, 35, 36, 37,
40, 43, 44, 45, 48], [44, 45, 46, 47, 48, 49], [45, 46, 47, 48, 49, 50], [32,
33, 34, 35, 41, 46, 47, 48, 49], [37, 38, 39, 41], [38, 39, 40, 41, 42, 43],
[35, 36, 37, 38], [32, 33, 36, 40], [35, 36, 41, 42, 43, 44], [43, 44, 45, 46,
47, 48]]
MaxOverload= 0 at machine= M2 and bucket= 38
MaxOverload= 0
********************************
Iteration(MIA)= 0
********************************
startlist= [[39, 40, 41, 42], [41, 42, 43, 44, 45, 46], [36, 37, 44, 45, 46,
47], [39, 40, 41, 42, 43, 44, 45, 46, 47], [33, 34, 38, 43], [34, 35, 36, 37,
40, 43, 44, 45, 48], [44, 45, 46, 47, 48, 49], [45, 46, 47, 48, 49, 50], [32,
33, 34, 35, 41, 46, 47, 48, 49], [37, 38, 39, 41], [38, 39, 40, 41, 42, 43],
[35, 36, 37, 38], [32, 33, 36, 40], [35, 36, 41, 42, 43, 44], [43, 44, 45, 46,
47, 48]]
NBH list= [[39, 40, 41, 42], [41, 42, 43, 44, 45, 46], [36, 37, 44, 45, 46,
47], [39, 40, 41, 42, 43, 44, 45, 46, 47], [33, 34, 38, 43], [34, 35, 36, 37,
40, 43, 44, 45, 48], [44, 45, 46, 47, 48, 49], [45, 46, 47, 48, 49, 50], [32,
33, 34, 35, 41, 46, 47, 48, 49], [38, 39, 40, 44], [38, 39, 40, 41, 42, 43],
[35, 36, 37, 38], [32, 33, 36, 40], [35, 36, 41, 42, 43, 44], [43, 44, 45, 46,
47, 48]]
totalEarliness 0= 3 totalTardiness 0= 0 totalMakespan (by job)= 129
      msMakespan= 19
Best candidate summary:
  minTardiness= 0
  minEarliness= 3
  minMakespan= 129
  minPoint= 0
  bestList= [[39, 40, 41, 42], [41, 42, 43, 44, 45, 46], [36, 37, 44, 45, 46,
47], [39, 40, 41, 42, 43, 44, 45, 46, 47], [33, 34, 38, 43], [34, 35, 36, 37,
40, 43, 44, 45, 48], [44, 45, 46, 47, 48, 49], [45, 46, 47, 48, 49, 50], [32,
33, 34, 35, 41, 46, 47, 48, 49], [38, 39, 40, 44], [38, 39, 40, 41, 42, 43],
[35, 36, 37, 38], [32, 33, 36, 40], [35, 36, 41, 42, 43, 44], [43, 44, 45, 46,
47, 48]]
  startList= [[39, 40, 41, 42], [41, 42, 43, 44, 45, 46], [36, 37, 44, 45, 46,
47], [39, 40, 41, 42, 43, 44, 45, 46, 47], [33, 34, 38, 43], [34, 35, 36, 37,
40, 43, 44, 45, 48], [44, 45, 46, 47, 48, 49], [45, 46, 47, 48, 49, 50], [32,
33, 34, 35, 41, 46, 47, 48, 49], [38, 39, 40, 44], [38, 39, 40, 41, 42, 43],
[35, 36, 37, 38], [32, 33, 36, 40], [35, 36, 41, 42, 43, 44], [43, 44, 45, 46,
47, 48]]
Best solution summary:
  Iteration =0
  bestList= [[39, 40, 41, 42], [41, 42, 43, 44, 45, 46], [36, 37, 44, 45, 46,
47], [39, 40, 41, 42, 43, 44, 45, 46, 47], [33, 34, 38, 43], [34, 35, 36, 37,
40, 43, 44, 45, 48], [44, 45, 46, 47, 48, 49], [45, 46, 47, 48, 49, 50], [32,
33, 34, 35, 41, 46, 47, 48, 49], [38, 39, 40, 44], [38, 39, 40, 41, 42, 43],
```

```
[35, 36, 37, 38], [32, 33, 36, 40], [35, 36, 41, 42, 43, 44], [43, 44, 45, 46,
47, 48]]
  FinalBestTardiness =0
  FinalBestEarliness =3
  FinalBestMakespan =129
  FinalMinPoint =0
*********************************
Iteration(MIA)= 1
*********************************
startlist= [[39, 40, 41, 42], [41, 42, 43, 44, 45, 46], [36, 37, 44, 45, 46,
47], [39, 40, 41, 42, 43, 44, 45, 46, 47], [33, 34, 38, 43], [34, 35, 36, 37,
40, 43, 44, 45, 48], [44, 45, 46, 47, 48, 49], [45, 46, 47, 48, 49, 50], [32,
33, 34, 35, 41, 46, 47, 48, 49], [38, 39, 40, 44], [38, 39, 40, 41, 42, 43],
[35, 36, 37, 38], [32, 33, 36, 40], [35, 36, 41, 42, 43, 44], [43, 44, 45, 46,
47, 48]]
NBH list= [[39, 40, 41, 42], [41, 42, 43, 44, 45, 46], [36, 37, 44, 45, 46,
47], [39, 40, 41, 42, 43, 44, 45, 46, 47], [33, 34, 38, 43], [34, 35, 36, 37,
40, 43, 44, 45, 48], [44, 45, 46, 47, 48, 49], [45, 46, 47, 48, 49, 50], [32,
33, 34, 35, 41, 46, 47, 48, 49], [38, 39, 40, 44], [38, 39, 40, 41, 42, 43],
[35, 36, 37, 38], [34, 38, 39, 41], [35, 36, 41, 42, 43, 44], [43, 44, 45, 46,
47, 48]]
totalEarliness 0= 2 totalTardiness 0= 0 totalMakespan (by job)= 128
      msMakespan= 19
Best candidate summary:
  minTardiness= 0
  minEarliness= 2
  minMakespan= 128
  minPoint= 0
  bestList= [[39, 40, 41, 42], [41, 42, 43, 44, 45, 46], [36, 37, 44, 45, 46,
47], [39, 40, 41, 42, 43, 44, 45, 46, 47], [33, 34, 38, 43], [34, 35, 36, 37,
40, 43, 44, 45, 48], [44, 45, 46, 47, 48, 49], [45, 46, 47, 48, 49, 50], [32,
33, 34, 35, 41, 46, 47, 48, 49], [38, 39, 40, 44], [38, 39, 40, 41, 42, 43],
[35, 36, 37, 38], [34, 38, 39, 41], [35, 36, 41, 42, 43, 44], [43, 44, 45, 46,
47, 48]]
  startList= [[39, 40, 41, 42], [41, 42, 43, 44, 45, 46], [36, 37, 44, 45, 46,
47], [39, 40, 41, 42, 43, 44, 45, 46, 47], [33, 34, 38, 43], [34, 35, 36, 37,
40, 43, 44, 45, 48], [44, 45, 46, 47, 48, 49], [45, 46, 47, 48, 49, 50], [32,
33, 34, 35, 41, 46, 47, 48, 49], [38, 39, 40, 44], [38, 39, 40, 41, 42, 43],
[35, 36, 37, 38], [34, 38, 39, 41], [35, 36, 41, 42, 43, 44], [43, 44, 45, 46,
47, 48]]
Best solution summary:
  Iteration =1
  bestList= [[39, 40, 41, 42], [41, 42, 43, 44, 45, 46], [36, 37, 44, 45, 46,
47], [39, 40, 41, 42, 43, 44, 45, 46, 47], [33, 34, 38, 43], [34, 35, 36, 37,
40, 43, 44, 45, 48], [44, 45, 46, 47, 48, 49], [45, 46, 47, 48, 49, 50], [32,
33, 34, 35, 41, 46, 47, 48, 49], [38, 39, 40, 44], [38, 39, 40, 41, 42, 43],
[35, 36, 37, 38], [34, 38, 39, 41], [35, 36, 41, 42, 43, 44], [43, 44, 45, 46,
47, 48]]
  FinalBestTardiness =0
  FinalBestEarliness =2
  FinalBestMakespan =128
  FinalMinPoint =0
*********************************
Iteration(MIA)= 2
*********************************
startlist= [[39, 40, 41, 42], [41, 42, 43, 44, 45, 46], [36, 37, 44, 45, 46,
47], [39, 40, 41, 42, 43, 44, 45, 46, 47], [33, 34, 38, 43], [34, 35, 36, 37,
40, 43, 44, 45, 48], [44, 45, 46, 47, 48, 49], [45, 46, 47, 48, 49, 50], [32,
33, 34, 35, 41, 46, 47, 48, 49], [38, 39, 40, 44], [38, 39, 40, 41, 42, 43],
[35, 36, 37, 38], [34, 38, 39, 41], [35, 36, 41, 42, 43, 44], [43, 44, 45, 46,
47, 48]]
NBH list= [[39, 40, 41, 42], [41, 42, 43, 44, 45, 46], [36, 37, 44, 45, 46,
47], [39, 40, 41, 42, 43, 44, 45, 46, 47], [33, 34, 38, 43], [34, 35, 36, 37,
```

```
40, 43, 44, 45, 48], [44, 45, 46, 47, 48, 49], [45, 46, 47, 48, 49, 50], [32,
33, 34, 35, 41, 46, 47, 48, 49], [40, 41, 44, 45], [38, 39, 40, 41, 42, 43],
[35, 36, 37, 38], [34, 38, 39, 41], [35, 36, 41, 42, 43, 44], [43, 44, 45, 46,
47, 48]]
totalEarliness 0= 1 totalTardiness 0= 0 totalMakespan (by job)= 127
      msMakespan= 19
Best candidate summary:
  minTardiness= 0
  minEarliness= 1
  minMakespan= 127
  minPoint= 0
  bestList= [[39, 40, 41, 42], [41, 42, 43, 44, 45, 46], [36, 37, 44, 45, 46,
47], [39, 40, 41, 42, 43, 44, 45, 46, 47], [33, 34, 38, 43], [34, 35, 36, 37,
40, 43, 44, 45, 48], [44, 45, 46, 47, 48, 49], [45, 46, 47, 48, 49, 50], [32,
33, 34, 35, 41, 46, 47, 48, 49], [40, 41, 44, 45], [38, 39, 40, 41, 42, 43],
[35, 36, 37, 38], [34, 38, 39, 41], [35, 36, 41, 42, 43, 44], [43, 44, 45, 46,
47, 48]]
  startList= [[39, 40, 41, 42], [41, 42, 43, 44, 45, 46], [36, 37, 44, 45, 46,
47], [39, 40, 41, 42, 43, 44, 45, 46, 47], [33, 34, 38, 43], [34, 35, 36, 37,
40, 43, 44, 45, 48], [44, 45, 46, 47, 48, 49], [45, 46, 47, 48, 49, 50], [32,
33, 34, 35, 41, 46, 47, 48, 49], [40, 41, 44, 45], [38, 39, 40, 41, 42, 43],
[35, 36, 37, 38], [34, 38, 39, 41], [35, 36, 41, 42, 43, 44], [43, 44, 45, 46,
47, 48]]
Best solution summary:
  Iteration =2
  bestList= [[39, 40, 41, 42], [41, 42, 43, 44, 45, 46], [36, 37, 44, 45, 46,
47], [39, 40, 41, 42, 43, 44, 45, 46, 47], [33, 34, 38, 43], [34, 35, 36, 37,
40, 43, 44, 45, 48], [44, 45, 46, 47, 48, 49], [45, 46, 47, 48, 49, 50], [32,
33, 34, 35, 41, 46, 47, 48, 49], [40, 41, 44, 45], [38, 39, 40, 41, 42, 43],
[35, 36, 37, 38], [34, 38, 39, 41], [35, 36, 41, 42, 43, 44], [43, 44, 45, 46,
47, 48]]
  FinalBestTardiness =0
  FinalBestEarliness =1
  FinalBestMakespan =127
  FinalMinPoint =0
*********************************
Iteration(MIA)= 3
*********************************
startlist= [[39, 40, 41, 42], [41, 42, 43, 44, 45, 46], [36, 37, 44, 45, 46,
47], [39, 40, 41, 42, 43, 44, 45, 46, 47], [33, 34, 38, 43], [34, 35, 36, 37,
40, 43, 44, 45, 48], [44, 45, 46, 47, 48, 49], [45, 46, 47, 48, 49, 50], [32,
33, 34, 35, 41, 46, 47, 48, 49], [40, 41, 44, 45], [38, 39, 40, 41, 42, 43],
[35, 36, 37, 38], [34, 38, 39, 41], [35, 36, 41, 42, 43, 44], [43, 44, 45, 46,
47, 48]]
Final solution summary:
  bestList= [[39, 40, 41, 42], [41, 42, 43, 44, 45, 46], [36, 37, 44, 45, 46,
47], [39, 40, 41, 42, 43, 44, 45, 46, 47], [33, 34, 38, 43], [34, 35, 36, 37,
40, 43, 44, 45, 48], [44, 45, 46, 47, 48, 49], [45, 46, 47, 48, 49, 50], [32,
33, 34, 35, 41, 46, 47, 48, 49], [40, 41, 44, 45], [38, 39, 40, 41, 42, 43],
[35, 36, 37, 38], [34, 38, 39, 41], [35, 36, 41, 42, 43, 44], [43, 44, 45, 46,
47, 48]]
  FinalBestTardiness =0
  FinalBestEarliness =1
  FinalBestMakespan =127
  FinalMinPoint =0
Resource loading:
M1     0      58000  0      58000  0      0      0      0      58000  0      0
       0      0      0      0      0      0      0      0      0      0      0
       0      0      0      0      0      0      0      0      0
M10    0      0      0      0      0      0      0      0      0      0      0
       0      65800  65800  0      65800  65800  65800  65800  65800  0      0
       0      0      0      0      0      0      0      0      0
```

```
M11    0      0      0      59400  59600  59400  59600  59400  0      59400  59400
       59600  0      0      0      0      0      0      0      0      0      0
       0      0      0      0      0      0      0      0      0
M12    0      0      0      58200  0      58200  0      0      0      0      58200
       0      0      0      0      0      0      0      0      0      0      0
       0      0      0      0      0      0      0      0      0
M13    0      0      0      0      0      0      0      0      0      0      0
       0      0      0      62800  62800  0      62800  0      0      0      0
       0      0      0      0      0      0      0      0      0
M14    0      0      0      0      59900  59900  0      59900  0      0      59900
       0      59900  59900  59900  0      0      0      0      0      0      0
       0      0      0      0      0      0      0      0      0
M15    0      0      0      0      0      0      0      0      0      0      0
       0      0      0      0      0      60500  60500  60500  0      0      0
       0      0      0      0      0      0      0      0      0
M16    0      0      0      0      0      0      0      0      0      0      62800
       62800  0      62800  62800  62800  62800  62800  0      0      0      0
       0      0      0      0      0      0      0      0      0
M17    0      0      0      0      0      0      0      0      0      0      0
       0      0      0      0      0      0      0      0      0      0      0
       0      0      0      0      0      0      0      0      0
M18    0      0      0      0      0      0      0      0      0      0      0
       0      0      0      0      0      0      0      0      0      0      0
       0      0      0      0      0      0      0      0      0
M19    0      0      59400  59400  59400  0      0      0      59400  59400  0
       0      0      61500  61500  0      61500  0      0      0      0      0
       0      0      0      0      0      0      0      0      0
M2     0      0      0      0      0      59400  59400  20300  59400  59700  80000
       79700  80000  59400  79700  59400  0      0      0      0      0      0
       0      0      0      0      0      0      0      0      0
M20    0      0      0      0      0      0      0      0      0      0      0
       0      0      0      0      0      0      0      0      0      0      0
       0      0      0      0      0      0      0      0      0
M3     0      0      0      0      0      0      60500  60500  60500  0      60500
       60000  60000  60500  60000  60000  60000  60000  60000  0      0      0
       0      0      0      0      0      0      0      0      0
M4     0      0      0      0      0      0      0      0      0      0      0
       0      0      0      0      0      0      0      0      0      0      0
       0      0      0      0      0      0      0      0      0
M5     0      0      58500  0      58500  0      0      0      0      58500  0
       0      0      0      0      0      0      0      0      0      0      0
       0      0      0      0      0      0      0      0      0
M6     0      0      0      0      0      0      0      0      0      0      0
       0      0      0      0      0      0      0      0      0      0      0
       0      0      0      0      0      0      0      0      0
M7     0      0      0      0      0      0      0      0      0      58500  58500
       0      58500  58500  58500  58500  58500  0      0      0      0      0
       0      0      0      0      0      0      0      0      0
M8     0      0      0      0      0      0      0      0      0      0      0
       0      59300  59300  0      59300  0      0      0      0      0      0
       0      0      0      0      0      0      0      0      0
M9     0      0      0      0      0      0      0      0      0      0      0
       0      0      0      0      0      0      0      0      0      0      0
       0      0      0      0      0      0      0      0      0
bestList= [[1254614400, 1254700800, 1254787200, 1254873600], [1254787200,
1254873600, 1254960000, 1255046400, 1255132800, 1255219200], [1254355200,
1254441600, 1255046400, 1255132800, 1255219200, 1255305600], [1254614400,
1254700800, 1254787200, 1254873600, 1254960000, 1255046400, 1255132800,
1255219200, 1255305600], [1254096000, 1254182400, 1254528000, 1254960000],
[1254182400, 1254268800, 1254355200, 1254441600, 1254700800, 1254960000,
1255046400, 1255132800, 1255392000], [1255046400, 1255132800, 1255219200,
1255305600, 1255392000, 1255478400], [1255132800, 1255219200, 1255305600,
1255392000, 1255478400, 1255564800], [1254009600, 1254096000, 1254182400,
```

```
1254268800, 1254787200, 1255219200, 1255305600, 1255392000, 1255478400],
[1254700800, 1254787200, 1255046400, 1255132800], [1254528000, 1254614400,
1254700800, 1254787200, 1254873600, 1254960000], [1254268800, 1254355200,
1254441600, 1254528000], [1254182400, 1254528000, 1254614400, 1254787200],
[1254268800, 1254355200, 1254787200, 1254873600, 1254960000, 1255046400],
[1254960000, 1255046400, 1255132800, 1255219200, 1255305600, 1255392000]]
indexList= [[8, 9, 10, 11], [10, 11, 12, 13, 14, 15], [5, 6, 13, 14, 15, 16],
[8, 9, 10, 11, 12, 13, 14, 15, 16], [2, 3, 7, 12], [3, 4, 5, 6, 9, 12, 13, 14,
17], [13, 14, 15, 16, 17, 18], [14, 15, 16, 17, 18, 19], [1, 2, 3, 4, 10, 15,
16, 17, 18], [9, 10, 13, 14], [7, 8, 9, 10, 11, 12], [4, 5, 6, 7], [3, 7, 8,
10], [4, 5, 10, 11, 12, 13], [12, 13, 14, 15, 16, 17]]
msStartTime= 1295854924
msEndTime= 1295854925
msComputationalTime(sec)= 1
OIA calculation time= 1295854925
startTime= 1295854924
endTime= 1295854925
computationalTime(sec)= 1
```