

**DOKUZ EYLÜL UNIVERSITY
GRADUATE SCHOOL OF NATURAL AND APPLIED
SCIENCES**

**VARIABLE NEIGHBORHOOD SEARCH BASED
ALGORITHMS FOR THE PARALLEL MACHINE
CAPACITATED LOTSIZING AND SCHEDULING
PROBLEM**

**by
Sel ÖZCAN**

**February, 2014
İZMİR**

**VARIABLE NEIGHBORHOOD SEARCH BASED
ALGORITHMS FOR THE PARALLEL MACHINE
CAPACITATED LOTSIZING AND SCHEDULING
PROBLEM**

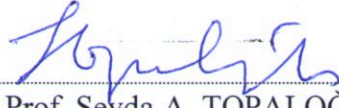
**A Thesis Submitted to the
Graduate School of Natural and Applied Sciences of
Dokuz Eylül University
In Partial Fulfillment of the Requirements for the Degree of Master of Science
in Industrial Engineering, Industrial Engineering Program**

**by
Sel ÖZCAN**

**February, 2014
İZMİR**

M.Sc THESIS EXAMINATION RESULT FORM

We have read the thesis entitled “**VARIABLE NEIGHBORHOOD SEARCH BASED ALGORITHMS FOR THE PARALLEL MACHINE CAPACITATED LOTSIZING AND SCHEDULING PROBLEM**” completed by **SEL ÖZCAN** under supervision of **ASSOC. PROF. ŞEYDA A. TOPALOĞLU** and we certify that in our opinion it is fully adequate, in scope and in quality, as a thesis for the degree of Master of Science.



Assoc. Prof. Şeyda A. TOPALOĞLU

Supervisor

Assoc. Prof. M. Fatih TAŞGETİREN

Assist. Prof. Özcan KILINÇCI

(Jury Member)



(Jury Member)



Prof.Dr. Ayşe OKUR
Director

Graduate School of Natural and Applied Sciences

ACKNOWLEDGEMENTS

First of all, I would like to express the deepest appreciation to my supervisor Assoc. Prof. Şeyda A. TOPALOĞLU for her kind help and continuous support. I also thank Assoc. Prof. M. Fatih TAŞGETİREN for his guidance and encouragement on this study. Finally, I am very grateful to my family and friends, for their continuous support and understanding during this challenging process. I also place on record, my sense of gratitude to one and all who, directly or indirectly, have lent their helping hand in this study.

Sel ÖZCAN

VARIABLE NEIGHBORHOOD SEARCH BASED ALGORITHMS FOR THE PARALLEL MACHINE CAPACITATED LOTSIZING AND SCHEDULING PROBLEM

ABSTRACT

In this paper, the capacitated lot-sizing and scheduling problem on parallel machines with eligibility constraints and sequence-dependent setup times and costs is addressed. The aim of this study is to find a production plan that minimizes production, setup and inventory holding costs while meeting the demands of products for each period without delay for a given planning horizon. Since this problem is NP-hard, various types of variable neighborhood search (VNS), variable neighborhood descent (VND) and reduced variable neighborhood search (RVNS) algorithms are used in order to analyse their performances on this problem. At first, a problem specific initial solution method is presented, which satisfies the demand of each period. In order to generate neighborhood solutions, three types of moves are defined which are respectively, insert move, swap move, and fractional insert move.. To evaluate the effectiveness and efficiency of each solution approach, a computational study is made using the benchmark problem instances which are taken from the literature. The results indicate that VNS algorithm performs well on small sized instances. The performance of VND approach is somehow similar when it is compared with the existing solution techniques in literature, and the solution times are relatively shorter. Additionally, although relatively high computation times, all instances are improved with RVNS algorithm.

Keywords: Capacitated lot sizing and scheduling problem, parallel machines, heuristics, variable neighborhood search, variable neighborhood descent, reduced variable neighborhood search.

PARALEL MAKİNELERDE KAPASİTELİ PARTİ BÜYÜKLÜĞÜ BELİRLEME VE ÇİZELGELEME PROBLEMİ İÇİN DEĞİŞKEN KOMŞULUK ARAMA TABANLI YÖNTEMLER

ÖZ

Bu çalışmada, farklı yetkinliklere sahip paralel makinelerde sıra bağımlı ayar zamanlı ve maliyetli, kapasiteli parti büyüklüğü belirleme ve çizelgeleme problemi ele alınmıştır. Çalışmanın amacı, belirlenen zaman süresi için talebi eksiksiz karşılayacak ve toplam üretim, ayar ve stok maliyetini en aza indirecek bir üretim planı oluşturmaktır. Problem NP-zor olduğu için, Değişken Komşuluk Arama (DKA) yöntemi ve onun türevleri olan Değişken Komşu İniş (DKİ) ve İndirgenmiş Değişken Komşuluk Arama (İDKA) yöntemleri kullanılarak her bir yöntemin performansları analiz edilmiştir. İlk başta, her periyodun talebini karşılamayı garanti eden, probleme özgü bir başlangıç çözüm yöntemi geliştirilmiştir. Komşuluk çözümlerini oluşturmak için, 3 farklı hareket tanımlanmıştır, bunlar sırasıyla, yerleştirme, ikili yer değiştirme ve kısmi yerleştirmedir. Her bir çözüm yaklaşımının etkinliğini ve verimliliğini değerlendirmek amacıyla, literatürdeki mevcut problemler kullanılarak analizler yapılmıştır. Analizlerin sonucunda, Değişken Komşuluk Arama (DKA) yönteminin küçük ölçekli problemlerde iyi sonuçlar verdiği görülmüştür. Değişken Komşu İniş (DKİ) metodunun literatürdeki mevcut çözüm teknikleriyle benzer sonuçlar verdiği saptanırken, İndirgenmiş Değişken Komşuluk Arama (İDKA) yönteminin ise yüksek çözüm sürelerine rağmen tüm test edilen problemlerde literatürdeki sonuçların hepsinden iyi sonuçlar bulduğu gözlemlenmiştir.

Anahtar Kelimeler: Kapasiteli parti büyüklüğü belirleme ve çizelgeleme problemi, paralel makineler, sezgiseller, değişken komşuluk arama, değişken komşu iniş, indirgenmiş değişken komşuluk arama.

CONTENTS

	Page
M.Sc THESIS EXAMINATION RESULT FORM.....	ii
ACKNOWLEDGEMENTS	iii
ABSTRACT	iv
ÖZ	v
LIST OF FIGURES	ix
LIST OF TABLES	xi
CHAPTER ONE - INTRODUCTION	1
CHAPTER TWO - LITERATURE REVIEW	3
2.1 The Capacitated Lot Sizing Problem	5
2.2 Capacitated Lot Sizing Problem (CLSP) and Its Extensions.....	6
2.3 Solution Approaches in the Literature	10
2.4 Evaluation of a Solution in Lot Sizing Problems.....	11
2.5 Definition of the Neighborhood in Lot Sizing Problems.....	12
CHAPTER THREE - PROBLEM DEFINITION.....	13
3.1 Mathematical Formulation of the Parallel Machine Capacitated Lotsizing and Scheduling Problem with Sequence-dependent Setups (CLSD-PM)	14
CHAPTER FOUR - VARIABLE NEIGHBORHOOD SEARCH.....	17
4.1 Basic Schemes of Variable Neighborhood Search (VNS).....	17
4.2 Variable Neighborhood Descent (VND).....	18
4.3 Reduced Variable Neighborhood Search (RVNS).....	19
4.4 Basic Variable Neighborhood Search (BVNS).....	20
4.5 General Variable Neighborhood Search (GVNS).....	22
4.6 Skewed Variable Neighborhood Search (SVNS)	22
4.7 Variable Neighborhood Decomposition Search (VNDS)	23
4.8 Hybrid Approaches with Variable Neighborhood Search	23

4.9 Applications of Variable Neighborhood Search in Industry.....	24
CHAPTER FIVE - CONSTRAINT HANDLING TECHNIQUES	25
5.1 Static Penalties	28
5.2 Dynamic Penalties.....	28
5.3 Annealing Penalties.....	29
5.4 Adaptive Penalties.....	30
5.5 Co-evolutionary Penalties	31
5.6 Death Penalties.....	32
CHAPTER SIX - PROPOSED ALGORITHMS.....	33
6.1 Initial Solution Generation.....	33
6.2 Objective Function Calculation.....	37
6.3 Moves.....	37
6.3.1 Insert Move	38
6.3.2 Swap Move	42
6.3.3 Fractional Insert Move	44
CHAPTER SEVEN - COMPUTATIONAL STUDY	48
7.1 Problem Instances Tested.....	48
7.2 Algorithms Tested.....	49
7.2.1 Variable Neighborhood Search (VNS)	50
7.2.2 Variable Neighborhood Descent (VND).....	55
7.2.3 Reduced Variable Neighborhood Search (RVNS).....	56
7.3 Computational Study.....	58
7.3.1 Experimental Results for Variable Neighborhood Search (VNS)	58
7.3.2 Experimental Results for Variable Neighborhood Descent (VND).....	60
7.3.3 Experimental Results for Reduced Variable Neighborhood Search (RVNS)	63
CHAPTER EIGHT - CONCLUSION.....	73
REFERENCES.....	75

APPENDICES 87

LIST OF FIGURES

	Page
Figure 2.1 MIP model of CLSP	6
Figure 3.1 MIP model for CLSD-PM	14
Figure 4.1 Best Improvement Local Search.....	18
Figure 4.2 First Improvement Local Search	19
Figure 4.3 Steps of Variable Neighborhood Descent (VND)	19
Figure 4.4 Steps of Reduced Variable Neighborhood Search (RVNS)	20
Figure 4.5 Steps of Basic Variable Neighborhood Search (BVNS)	21
Figure 4.6 Steps of General Variable Neighborhood Search (GVNS)	22
Figure 6.1 Schedules for both machines (after product 5 is assigned).....	34
Figure 6.2 Schedules for both machines (after all products are assigned for period 2)	35
Figure 6.3 Final schedules for both machines.....	35
Figure 6.4 Pseudocode for the initial solution generation.....	36
Figure 6.5 Initial schedule.....	39
Figure 6.6 Schedule obtained after an insert move	39
Figure 6.7 Pseudocode for setup updates used in this study	40
Figure 6.8 Pseudocode for the update of all variables	40
Figure 6.9 Pseudocode for the insert move used in this study	41
Figure 6.10 Initial schedule.....	42
Figure 6.11 Schedule obtained after a swap move.....	42
Figure 6.12 Pseudocode for the swap move used in this study.....	43
Figure 6.13 Initial schedule.....	44
Figure 6.14 Schedule obtained after a fractional insert move.....	44
Figure 6.15 Pseudocode for the fractional insert move used in this study.....	46
Figure 7.1 Pseudocode for Best Improvement for Insert Move.....	50
Figure 7.2 Pseudocode for Best Improvement for Swap Move	51
Figure 7.3 Pseudocode for comparison function.....	52
Figure 7.4 Pseudocode for VNS Algorithm used in this study (kmax=2, N1=insert move ,N2=swap move)	53

Figure 7.5 Pseudocode for VND Algorithm used in this study (kmax=2, N1=swap move, N2=insert move).....	55
Figure 7.6 Pseudocode for RVNS Algorithm used in this study (kmax=2, N1=fractional insert move, N2=swap move).....	57

LIST OF TABLES

	Page
Table 6.1 Demand matrix.....	33
Table 6.2 Capability matrix.....	34
Table 6.3 Setup matrix	34
Table 6.4 Machine capacities by period.....	34
Table 6.5 Demand matrix.....	38
Table 6.6 Capability matrix.....	38
Table 6.7 Setup matrix	39
Table 6.8 Machine capacities by period.....	39
Table 7.1 Average % deviations from lower bound of the proposed VNS algorithm (Compared with the benchmark results)	59
Table 7.2 Differencens of average % deviations from lower bound of the proposed VNS algorithms.....	60
Table 7.3 Average % deviations from lower bound of the proposed VND algorithm (Compared with the benchmark results)	61
Table 7.4 Differencens of average % deviations from lower bound of the proposed VND algorithms	62
Table 7.5 Average Computation Times of the proposed VND algorithm (Compared with the benchmark results)	63
Table 7.6 Randomly Selected Instances and their iteration numbers where convergence starts tested algorithm: RVNS (N1: insert, N2:swap).....	64
Table 7.7 Randomly Selected Instances and their iteration numbers where convergence starts Tested Algorithm: RVNS (N1: swap, N2:insert)	64
Table 7.8 Randomly Selected Instances and their iteration numbers where convergence starts Tested Algorithm: RVNS (N1: fractional insert, N2:insert)	65
Table 7.9 Randomly Selected Instances and their iteration numbers where convergence starts Tested Algorithm: RVNS (N1: fractional insert, N2:swap).....	65
Table 7.10 Randomly Selected Instances and their iteration numbers where convergence starts Tested Algorithm: RVNS (N1: insert, N2:fractional insert)	66
Table 7.11 Randomly Selected Instances and their iteration numbers where convergence starts Tested Algorithm: RVNS (N1: swap, N2:fractional insert).....	66

Table 7.12 Average % deviations of RVNS (N1: insert, N2:swap).....	67
Table 7.13 Average % deviations of RVNS (N1: swap, N2:insert).....	68
Table 7.14 Average % deviations of RVNS (N1: insert, N2:fractional insert).....	68
Table 7.15 Average % deviations of RVNS (N1: swap, N2:fractional insert)	69
Table 7.16 Average % deviations of RVNS (N1: fractional insert, N2:swap)	69
Table 7.17 Average % deviations of RVNS (N1: fractional insert, N2:insert).....	70
Table 7.18 Summary of average % deviations of all tested RVNS Algorithms	71
Table 7.19 Summary of differences of average % deviations of all tested RVNS Algorithms from best so far lower bounds.....	72

CHAPTER ONE

INTRODUCTION

Production planning and control is a popular topic in industrial engineering that has been studied over many years. Production planning and control covers lots of subjects; like forecasting, scheduling theory and inventory management issues. Accordingly, lot sizing and scheduling is also examined under production planning and control.

Firstly, lot sizing problems emerged as the Economic Order Quantity (EOQ) (Harris, 1913). Generally, lot sizing and scheduling problem is the determination of the production quantities and schedules on the production lines for providing all demands without any shortages. The main aim of this problem is minimizing the overall production, inventory, and setup costs.

In the literature, lot sizing and scheduling problems mainly focus on discrete production. On the other hand, increase in the customer needs forces the manufacturers to change their production system to make-to-order. Besides, with the increasing importance of lean production, due to the fluctuations on customer demand and high setup costs, production is done in lots. Thus, lot sizing becomes an important issue on manufacturing systems and a tactical level of decision making problem in the literature.

There are various lot sizing and scheduling problems studied in the literature. Detailed explanation related with these problems will be given in the literature review part of this study. Moreover, there are many practical cases of the Parallel Machine Capacitated Lot Sizing and Scheduling Problem in real world. For instance, pharmaceutical, chemical, electronics, food, tile manufacturing, tire industry, injection molding, the alloy foundry industry and multi-layer ceramics can be given as industrial examples (Jans, 2009). In this thesis, parallel machine capacitated lot sizing and scheduling problem with sequence-dependent setups (CLSD-PM) is examined using three variants of Variable Neighborhood Search (VNS) algorithm.

The aim of this thesis is to develop VNS based algorithms to solve the CLSD-PM problem. The reason why VNS and its variants, Variable Neighborhood Descent (VND) and Reduced Variable Neighborhood Search (RVNS) are chosen is that VNS and its variants indicated high performance on different kinds of scheduling problems (Hansen & Mladenović, 2010). Besides, there is no study that uses VNS for CLSD-PM problem. In this thesis, furthermore, the effect of using Constraint Handling Techniques, which is explained in Chapter 5, can be seen.

CHAPTER TWO

LITERATURE REVIEW

In this chapter, a comprehensive literature review is given about the Capacitated Lot Sizing and Scheduling (CLSP) Problem. Firstly, the history of how CLSP has emerged is explained, then extensions of CLSP problems are clarified. Afterwards, solution approaches are explained which are presented in the literature so far. Finally, evaluation of a solution in lot sizing problems and neighborhood structures used for the CLSP problem are examined.

First idea on lot sizing came up with the classical economic order quantity (EOQ) model (Erlenkotter, 1990). As in the research of Drexl and Kimms stated (1997), the assumptions for the EOQ model are a single-level production process with no capacity constraints, which makes the problem become single-item problem. Moreover, demand has a constant rate, which means that it is stationary. For the EOQ model, optimal solution can easily be derived (Drexl & Kimms, 1997).

Since EOQ cannot cover all, other models were developed. For example, the first one is the Economic Lot Scheduling Problem (ELSP), which includes capacity restrictions (Elmagharby, 1978). Since resources are usually shared in common by several items, the ELSP is a single-level, multi-item problem, where the demand is assumed to be still occurring continuously with a constant rate. Solving the ELSP is NP-hard, thus, heuristic methods were utilized (Drexl & Kimms, 1997).

Another different point of view from the EOQ model is that where demand is dynamic. The so-called Wagner-Whitin (WW) problem assumes a finite planning horizon which is subdivided into several discrete periods. Demand is given per period and may vary over time. However, capacity limits are not considered which means that the single-level WW problem is a single-item problem (Drexl & Kimms, 1997).

Afterwards, as lot sizing problems become more complex, models include both dynamic and capacitated approaches (Drexl & Kimms, 1997). Moreover, scheduling was integrated with lot size decisions. Mainly, there are six variations of single level lot sizing and scheduling problem that have been studied, and are known to be NP-hard. These six variations can be listed as below:

- *Economic Lot Scheduling (ELSP)* in which the planning horizon is infinite,
- *Capacitated Lot Sizing (CLSP)*, also called the large-bucket problem, where lots of several part types processed in each period, then, jobs are scheduled in each period separately.
- *Discrete Lot Sizing and Scheduling (DLSP)*, also called the small-bucket problem, where macro periods of CLSP are subdivided into micro periods in which only one part type may be processed at full capacity.
- *Continuous Setup Lot Sizing Problem (CSLP)*, adapts DLSP, allowing at most one part type each period but using less than full capacity.
- *Proportional Lot Sizing and Scheduling (PLSP)*, allows unused capacity to process a second part type in a period.
- *General Lot Sizing and Scheduling (GLSP)*, incorporates a user-defined parameter to restrict the number of lots per period.

Also, a general classification of lot sizing and scheduling models in the literature can be seen as follows:

- *Single Level Lot Sizing and Scheduling*
 - a. Economic lot scheduling problem (ELSP)
 - b. The capacitated lot sizing problem (CLSP)
 - c. The discrete lot sizing and scheduling problem (DLSP)
 - d. The continuous setup lot sizing problem (CSLP)
 - e. The proportional lot sizing and scheduling problem (PLSP)
 - f. The general lot sizing and scheduling problem (GLSP)
- *Continuous Time Lot Sizing and Scheduling*
- *Multi-level Lot Sizing and Scheduling*

2.1 The Capacitated Lot Sizing Problem

Capacitated lot sizing problem is an extension of WW problem to capacity constraints. CLSP is a multi-item problem. The objective is to minimize the sum of total setup and inventory holding costs. On the other hand, setup variables are defined as binary values. Another condition is that, production of an item can only take place if the machine is set up for that particular item (Drexl & Kimms, 1997).

CLSP is a large bucket problem since several items can be produced per period. In reality, this time period can generally be one week. CLSP problem is NP-complete and therefore, only a few studies resulted in optimality (Chen & Thizy, 1990; Eppen & Martin, 1987; Gelders et al., 1986).

However, scheduling decisions are considered in CLSP. As a result, the usual approach is to solve the CLSP first, and to solve a scheduling problem for each period separately afterwards (Drexl & Kimms, 1997). The full mathematical model of CLSP can be seen below.

Parameters

C_t = Available capacity of the machine in period t

d_{it} = Demand for product i in period t

h_i = Cost of carrying one unit of product i for one period

I_{i0} = initial inventory for product i

p_i = consumption of capacity for producing one unit of product i

s_i = setup cost for product i

I = number of products

T = number of periods

Decision Variables

I_{it} = inventory for product i at the end of period t

x_{it} = production quantity for product i in period t

y_{it} = binary variable indicating whether a setup for product i occurs in period t

$$\min \sum_{i=1}^I \sum_{t=1}^T (s_i y_{it} + h_i I_{it}) \quad (2.1)$$

$$I_{it} = I_{i(t-1)} + x_{it} - d_{it} \quad i = 1, \dots, I \quad t = 1, \dots, T \quad (2.2)$$

$$p_i x_{it} \leq C_t y_{it}, \quad i = 1, \dots, I \quad t = 1, \dots, T \quad (2.3)$$

$$\sum_{i=1}^I p_i x_{it} \leq C_t, \quad t = 1, \dots, T \quad (2.4)$$

$$y_{it} \in \{0,1\}, \quad i = 1, \dots, I \quad t = 1, \dots, T \quad (2.5)$$

$$I_{it}, x_{it} \geq 0 \quad i = 1, \dots, I \quad t = 1, \dots, T \quad (2.6)$$

Figure 2.1 MIP model of CLSP

The objective function on (2.1) minimizes the overall setup and inventory holding costs. Constraint (2.2) is for the inventory balance. Constraint (2.3) forces that product i can be produced if the machine is set up for that product. Constraint (2.4) is for the capacity limitations. Setup variables are defined as binary in (2.5) and (2.6) is for non-negativity of variables.

2.2 Capacitated Lot Sizing Problem (CLSP) and Its Extensions

As mentioned above, the capacitated lot sizing problem provides a mathematical visualization for large bucket lot sizing problems where a pre-determined number of periods and constant demands are incurred (Quadt & Kuhn, 2008). In fact, Quadt & Kuhn (2008), presented various extensions of CLSP such as, back-orders, setup carry-over, sequencing, and parallel machines.

Standard CLSP is defined in Quadt & Kuhn (2008), as follows: multiple products have to be produced while a deterministic, discrete demand quantity for every product is given. In addition, number and duration of the periods is known, too. Moreover, producing a product consumes machine capacity and when there is a change from one product to another, duration and cost for setup occurs. In addition, when a product unit is produced in the previous period(s), unique inventory holding costs are incurred. Finally, the objective is to find an optimal production plan minimizing setup and inventory costs as well as finding optimal lot-sizes for each period and for each product in order to satisfy each period's demand in terms of each product.

One of the extensions is *parallel machines*. Parallel machine CLSP can be seen in various areas of industry in real world such as, chemical, electronics food and textile (Wittrock, 1988; Riane, 1998; Moursli & Pochet, 2000; Quadt & Kuhn, 2008). As Quadt & Kuhn (2008), explained, “using parallel machines provides that a product may be produced on any of the parallel machines. This increases the complexity of the problem since a decision has to be made on which machine to produce a product unit and how many machines to use in parallel for each product in each period”.

Setup carry-over is another extension of the CLSP. Carrying over a setup between periods is given in many industries, for instance, in semiconductor industry, where production runs 24 hours a day and 7 days a week (Quadt & Kuhn, 2005). Setup carry-over means that a machine’s setup state can be preserved between two consecutive periods and no extra setup is required. As it is stated in Quadt & Kuhn (2008), in the standard CLSP, a setup is made for each product produced per period (and machine), whereas, with the setup carry over, the last product per period may be produced without any additional setup in the following period”. In Haase’s study (1998), it can be denoted that solutions become significantly different when setup carry over is considered. Quadt & Kuhn (2008), claim that “if setup carry-over is accounted together with parallel machines, a lot-for-lot policy could substantially reduce the number of setup operations”. Carrying over a setup state makes the problem more complicated because “scheduling” decisions has to be carried out, which means, for each machine, a decision has to be made which product shall be the first and the last in a period. Gopalakrishnan et al. (1995), developed a model for CLSP problem with constant setup times and setup carryovers; however, they claimed that, due to the complexity of the proposed model, alternative ways should be developed. On the other hand, Suerie & Stadtler (2003), present a different mathematical model for CLSP with sequence independent setup costs and times. They use the idea of the standard facility location formulation, and propose new sets of variables and constraints to model the setup carryover.

While including scheduling decisions for all products, a total *sequence* of all products on each machine has to be concluded. Sequence-dependent setup costs and times can be found in various industries, for example, in chemical industry where quick changes among products with fewer setup decrease the energy costs (Quadt & Kuhn, 2008).

Back-order means allowing a product to be produced after its dedicated demand period. On the other hand, back-order costs are considered for every period and very important because otherwise, no feasible plan would exist (Quadt & Kuhn, 2008). Even though back-order can be seen in real life problems, there are only a few researches considering shortages (Smith-Daniels, 1986; Pochet & Wolsey, 1988; Millar & Yang, 1994; Cheng et al., 2001; Hung & Chien, 2000; Quadt & Kuhn, 2009).

Bitran & Yanasse (1982) show that the CLSP is NP-hard even without setup times; no approach is provided for gathering the optimality. When setup times are also taken into account, CLSP problem becomes NP-complete Maes et al. (1991), implying that it is hard to find even a feasible solution.

It is claimed that CLSP with sequence-dependent setup times show many similarities with the traveling salesman problem (TSP) and the vehicle routing problem (VRP) (Laporte 1992A, 1992B). More specifically, Laporte (1992A, 1992B) noticed that “setup cost matrix in CLSP is similar to the distance matrix in the TSP or VRP. However, solving the multi-period CLSP is equivalent to solving multiple dependent TSPs”.

Haase & Kimms (2000) and Gupta & Magnusson (2005) studied CLSP with sequence-dependent setups. Haase & Kimms (2000) introduced a model for CLSP where the efficient product sequences are pre-determined. Therefore, instead of sequencing all produced products, which pre-determined sequences will be used in each period are determined. As a solution approach, they used tailor-made branch-and-bound method.

In the study of Gupta & Magnusson (2005), they proposed a heuristic to the single machine CLSP where they introduce sequence-dependent setup costs, non-zero setup times and setup carryover. They took constant setup times s_0 instead of sequence-dependent setup times. Gupta & Magnusson (2005), explained the reason of this simplification as follows: “The reason of this simplification is that it is impossible to check if or not the problem is feasible without solving a mathematical program which is of similar complexity as the original lot sizing problem”. They noticed that there are three main things that makes the CLSP problem much harder, which are, tight capacity, large and sequence-dependent setup costs, and non-zero setup times.

Almada-Lobo et al. (2007), presented two new linear MIP models for single machine CLSP with sequence dependent setup times and costs as well as considering setup carryover. Likewise, no backlogging is allowed, i.e., each product’s demand should fully satisfied for each period. In order to keep track of schedules, they introduced a constraint influenced by the traveling salesman problem. On the other hand, they claimed that their formulations are simpler than others in the literature. Furthermore, they proposed a five-step heuristic for finding feasible solutions.

Afterwards, Almada-Lobo (2011) extended their work to the parallel machine case while introducing machines’ eligibility restrictions among products. They proposed MIP-based heuristics; in a more detailed way, they introduced an iterative stochastic MIP-based neighborhood structure in order to obtain better feasible solutions. Moreover, they divided the problem into many subMIPs and solved them randomly. The stochasticity of the solution algorithm arises from the idea of selection of the subMIPs.

Recently, in the study of Xiao et al. (2013), they proposed CLSP with sequence-dependent setup times, costs, and setup carryover on parallel machines. In addition, similar to the study of James and Almada-Lobo (2011), machine eligibilities were also introduced. Differently, they introduced machine preference constraints which are soft constraints and in case of any violation of these constraints, it is penalized in the objective function. Moreover, they allowed backlogging. In order to find good

quality solutions to this problem, they proposed two MIP-based fix-and-optimize algorithms, where the binary decision variables related with the assignment of machines are first fixed using the randomized least flexible machine rule and the rest of the decision variables are settled by an MIP solver.

2.3 Solution Approaches in the Literature

There are a lot of work on lot sizing and scheduling problem on a single machine in the literature. Mainly, the case involving sequence-dependent setup has been studied in various literatures (Thizy & Van Wassenhove, 1985; Dobson et al., 1987; Trigeiro et al., 1989; Fleischmann, 1990; Cattrysse et al., 1993; Blocher et al., 1999). In terms of solution methods, Blocher et al. (1999), used Branch and Bound; whereas Cattrysse et al. (1993), used a column generation based heuristic for a set partitioning formulation.

Earlier work of Haase (1998), included a heuristic priority rule for the case with sequence-dependent setup costs and used a local search to derive appropriate values of parameters for use by the priority rule. Moreover, Miller et al. (1999), proposed a MIP formulation where backlogging is allowed and as a solution methodology, they used Genetic Algorithm (GA) combined with Hill Climbing (HC) technique.

On the other hand, some metaheuristic methods are also used for lot sizing and scheduling problems. GA is used for the single level CLSP by Kohlmorgen et al., (1999). Moreover, a tabu search based approach is utilized by Hindi (1996). In addition to these, while interfering setup time criterion, Özdamar and Bozyel (2000), suggested a hybrid method where GA and simulated annealing (SA) are considered together. A hybrid TS-SA was developed by Özdamar et al., (2002). Also Gopalarakrishnan et al. (2001), used a hybrid GA-TS so as to solve this problem. Pedroso & Kubo (2005), suggested a hybrid tabu search with relax-and-fix heuristic for CLSP with multiple resources. Beraldi et al. (2008), introduced a rolling-horizon and relax-and-fix heuristic for the identical parallel machine CLSP case, where they guarantee that their solution mechanism will provide feasibility when data set is

feasible. Besides, Ferreira et al. (2009), presented relax-and-fix heuristics for solving a case study related with CLSP in a soft drink company in Brazil. Özdamar & Birbil (1998), used hybrid heuristics involving simulated annealing, genetic algorithm and tabu search on CLSP with parallel machines. In terms of combined lot sizing and scheduling on parallel machines, Kang et al. (1999), used a hybrid branch and bound algorithm and column generation approach. Meyr (2002), generalized the problem, allowing nonzero sequence-dependent setup times. He extended his previous study on single machine to parallel machines, using combined dual reoptimization with simulated annealing and threshold accepting methods. Recently, Dastidar & Nagi (2005), presented a MIP model where there are unrelated parallel machines with sequence-dependent setup criterion; the methodology used here is a two-phase decomposition methodology.

2.4 Evaluation of a Solution in Lot Sizing Problems

Various options are used in order to evaluate the solution. One common alternative is to evaluate the objective function. However, in a genetic algorithm, it is possible to obtain an infeasible solution after applying the genetic operators, and for tabu search and simulated annealing, a move can also lead to an infeasible neighbor. A main issue is how to treat those infeasibilities (Jans & Degraeve, 2008). One option is to ignore all infeasible solutions or attach an infinite cost to them (Kimms, 1999). Another example can be attaching a backlog cost for demand which cannot be met in time (Barbarosoğlu & Özdamar, 2000; Özdamar & Barbarosoğlu, 2000) or a penalty cost in case of capacity violation (Özdamar & Birbil, 1998; Gopalakrishnan et al., 2001); or a high cost for the initial inventory (Meyr, 2000). Another treatment option might be using some repair operators for infeasible solutions (Özdamar & Birbil, 1998). Moreover, in order to penalize infeasible solutions, many constraint handling techniques can be utilized which are explained in Chapter 5.

2.5 Definition of the Neighborhood in Lot Sizing Problems

The definition of the *neighborhood* can vary according to the solution technique or the solution representation. (Jans & Degraeve, 2008). If there are both integer and continuous variables used in the representation, moves can be defined among each variable, i.e., a move can be carried out within setups and production quantities as well (Gopalarakrishnan et al., 2001). Neighborhood schemes can be defined either moving fractionally or completely the production amount of a product (Özdamar & Birbil, 1998; Özdamar & Barbarosoğlu, 2000; Özdamar et al., 2002). Changing the setup state of a product in a period is the most widely used neighborhood definition (Kuik & Salomon, 1990; Hindi, 1996; Kuik et al., 1993; Salomon et al., 1993). In addition to the usual swap and insert moves; Almada-Lobo & James (2010), introduced a different type of move, called *fractional insert move*. *Fractional insert move* means splitting a lot randomly into two lots, where the total quantity produced is the same as the original quantity. One part will be inserted to another location whereas the remaining part will be left in the same position.

CHAPTER THREE

PROBLEM DEFINITION

In this chapter, the full mathematical model used in this study is presented. Parallel machine capacitated lotsizing and scheduling problem with sequence-dependent setups (CLSD-PM) has been formulated previously in James & Almada-Lobo, (2011). The problem considers a planning interval with $t=1, \dots, T$ periods and $i, j=1, \dots, N$ products processed on $m=1, \dots, M$ machines. The parameters and decision variables of the problem can be listed as follows:

Parameters

d_{it} = demand of product i in period t

s_{mij} = setup time incurred when a setup occurs from product i to j on machine m

c_{mij} = setup cost incurred when a setup occurs from product i to j on machine m

h_i = unit inventory holding cost for product i from one period to the next

p_{mi} = the processing time of one unit of product i on machine m

C_{mt} = the capacity of machine m available in period t

G_{mit} = upper bound on the production quantity of product i in period t to machine m

A_{mi} = product i 's capability of machine m

Decision Variables

X_{mit} = quantity of product i produced in period t on machine m

I_{it} = inventory level of product i at the end of period t

V_{mit} = an auxiliary variable that assigns product i on machine m in period t

T_{mijt} = 1 if a setup occurs from product i to j on machine m in period t

Y_{mit} = 1 if the machine m is set up for product i at the beginning of period t

3.1 Mathematical Formulation of the Parallel Machine Capacitated Lotsizing and Scheduling Problem with Sequence-dependent Setups (CLSD-PM)

$$\min \sum_m \sum_i \sum_j \sum_t C_{mij} \cdot T_{mijt} + \sum_i \sum_t h_i \cdot I_{it} \quad (3.1)$$

$$I_{i(t-1)} + \sum_m X_{mit} - d_{it} = I_{it}, \quad i \in [N], t \in [T] \quad (3.2)$$

$$I_{i0} = 0, \quad i \in [N] \quad (3.3)$$

$$\sum_i p_{mi} \cdot X_{mit} + \sum_i \sum_j s_{mij} \cdot T_{mijt} \leq C_{mt}, \quad m \in [M], t \in [T] \quad (3.4)$$

$$X_{mit} \leq G_{mit} \cdot (\sum_j T_{mjit} + Y_{mit}), \quad m \in [M], i \in [N], t \in [T] \quad (3.5)$$

$$Y_{mi(t+1)} + \sum_j T_{mjit} = Y_{mit} + \sum_j T_{mjit}, \quad m \in [M], i \in [N], t \in [T] \quad (3.6)$$

$$\sum_i Y_{mit} = 1, \quad m \in [M], t \in [T] \quad (3.7)$$

$$V_{mit} + N \cdot T_{mijt} - (N - 1) - N \cdot Y_{mjt} \leq V_{mjt}, \\ m \in [M], i \in [N], j \in [N] \setminus \{i\}, t \in [T] \quad (3.8)$$

$$\sum_t X_{mit} \leq G_{mit} A_{mi}, \quad m \in [M], i \in [N] \quad (3.9)$$

$$(X_{mit}, I_{it}) \geq 0, (T_{mijt}, Y_{mit}) \in \{0,1\}, X_{mit} \in Z, V_{mit} \in R \quad (3.10)$$

Figure 3.1 MIP model for CLSD-PM

Objective function (3.1) minimizes overall inventory and setup costs. Constraint (3.2) is for production and inventory balance and constraint (3.3) indicates that initial inventory level is zero. On the other hand, constraint (3.4) controls that production and setup times do not exceed the available capacity. Constraint (3.5) forces that whenever a product is produced, a setup should be made. In other words, if $\sum_j T_{mjit} + Y_{mit} = 0$, meaning that either there is no setup occurred from product j to product i or the machine is not set up for product i , then product i cannot be produced on that machine, forcing the quantity produced $X_{mit} = 0$. In addition, constraint (3.6) is for setup carryover for two consecutive periods. In fact, constraint (3.6) provides that flow in equals to flow out, i.e., if there is an input setup and no output setup for product i on period t on machine m , it means that this setup was the last one to be performed on the machine m in period t and accordingly, the machine is configured for product i at the beginning of the next period $(t+1)$, forcing $Y_{mi(t+1)}$ to be 1. On the other hand, if there is an output setup and no input one, it means that the machine

is configured for product i at the beginning of period t , $Y_{mi(t+1)} = 1$. Constraint (3.7) ensures that each machine should be set up for one product at the beginning of each time period. Constraint (3.8) eliminates disconnected subtours. In other words, this constraint works whenever a subtour occurs in a period, forcing the respective machine to be set up at the beginning of that period to one of the products that are part of the subtour. In order to achieve a feasible solution, only a single connected component is linked to Y 's. Constraint (3.8) provides this by using auxiliary variable V_{mit} that value the machine state through any sequence. V_{mit} indicates sequence number in which product i is produced on machine m in period t . Constraint (3.8) provides sequencing which is also a similar modeling in the traveling salesman problem proposed in the study of Nemhauser & Wolsey (1988).

Constraints (3.2)-(3.7) and (3.10) allow not disjoint cycles. However, due to the triangle inequality $c_{mij} + c_{mjk} \geq c_{mik}$, the inflow of every product is at most 1 and such a scenario will not occur in an optimal solution. Constraints (3.6), (3.7), and (3.8) determine the sequence of the products on machine m in period t and keep track of the machine configuration state by recording the product that a machine is ready to process. Constraint (3.9) indicates each machine's ability to produce that product and constraint (3.10) is for nonnegativity and integrality of the decision variables.

The variability of machine capacities within periods may be due to the nature of the real system where slight changes in the capacities are reflected to the mathematical model of the problem. Moreover, all unit production costs are same for each product, which is 1, the reason might be that the cost of running the machine does not depend on the product type produced. On the other hand, the reason why there is an upper bound on the production quantity of each product i in period t to machine m might be the shortage of the raw materials used for product i . Furthermore, this model does not consider backlogging, i.e., all demands should be satisfied for each period t . It can also be deduced that shortage cost of a product i is very high when it is compared to setup and inventory holding costs of that product. Moreover, another constraint that makes this problem harder is machine eligibility restriction proposed in Constraint (3.9). More specifically, each product can be

produced on at least one machine, however there may be more than one eligible machine that can produce that product, as well.

CHAPTER FOUR

VARIABLE NEIGHBORHOOD SEARCH

In Chapter 4, a detailed explanation of Variable Neighborhood Search (VNS) algorithm and its variants are given. Moreover, a comprehensive literature study about VNS, in addition to the various variants of VNS, covering the hybrid approaches and applications of this algorithm in real life are presented in this chapter.

VNS, proposed by Mladenović & Hansen (1997), is a metaheuristic used for solving combinatorial and global optimization problems. Moreover, in VNS, neighborhoods change in a systematic manner in order to escape from local minima. Many variants of VNS, for example, *variable neighborhood descent* (VND), *reduced VNS* (RVNS), *basic VNS* (BVNS), *general VNS* (GVNS), and *skewed VNS* (SVNS) have been developed so far.

4.1 Basic Schemes of Variable Neighborhood Search (VNS)

A finite set of pre-selected neighborhood structures are defined as N_k ($k = 1, \dots, k_{max}$) and $N_k(x)$ is the set of solutions in the k^{th} neighborhood of x . An *optimal solution* x_{opt} (or global minimum) is a feasible solution where a minimum is reached. $x' \in X$ is a *local minimum* with respect to N_k , if there is no solution $x \in N_k(x') \subseteq X$ such that $f(x) < f(x')$. Metaheuristics (based on local search procedures) try to continue the search by other means after finding the first local minimum (Mladenović & Hansen, 1997). VNS is based on the following three simple facts:

“Fact 1: A local minimum w.r.t one neighborhood structure is not necessary so with another;”

“Fact 2: A global minimum is a local minimum w.r.t all possible neighborhood structures”.

“Fact 3: For many problems local minima w.r.t one or several N_k are relatively close to each other”.

Hansen et al. (2010), indicated in their study that “the last observation is empirical, implying that a local optimum usually give some idea about the global optimum. For example, several variables having the same value in both”. However, which variables have the same value is not clear. As a result, neighborhood of the local optimum is in an ordered way, till a better local minimum is kept (Hansen et al., 2010).

VNS can be classified into three different ways: (i) deterministic; (ii) stochastic; (iii) both deterministic and stochastic.

4.2 Variable Neighborhood Descent (VND)

Variable Neighborhood Descent (VND) method is obtained if change of neighborhoods is performed in a deterministic way. Firstly, an initial solution x is taken, the *best* (minimum of $f(x)$) within the neighborhood N_k is found, and if there is no further improvement on that neighborhood, the heuristic continues with the next neighborhood until $k = k_{\max}$. Furthermore, VND is a *steepest descent heuristic* where *best improvement* local search is used (Hansen & Mladenović, 2005). In other words, at each step, the neighborhood is explored entirely. In fact, since searching the whole neighborhood can be time consuming in some cases, maybe *first improvement (first descent heuristic)* can also be used in the local search phase. Steps of *best improvement* and *first improvement* can be identified in the figures below.

Initialization.

Choose f, X , neighborhood structure $N(x)$, initial solution x ;

Repeat (1) and (2):

(1) Find $x' = \operatorname{argmin}_{x \in N(x)} f(x)$;

(2) If $f(x') < f(x)$ set $x \leftarrow x'$ and iterate; otherwise, stop.

Figure 4.1 Best Improvement Local Search

Initialization. Choose f, X , neighborhood structure $N(x)$, initial solution x ;
Repeat (1), (2) and (3):
 (1)*Find first solution $x' \in N(x)$;*
 (2)*If $f(x') > f(x)$, find next solution $x'' \in N(x)$;*
set $x' \leftarrow x''$ and iterate (2);
otherwise, set $x \leftarrow x'$ and iterate (1);
 (3)*If all solutions of $N(x)$ have been considered, stop.*

Figure 4.2 First Improvement Local Search

The steps of VND is as follows,

Repeat the following sequence until no improvement is obtained:
 (1)*Set $k \leftarrow 1$;*
 (2)*Repeat the following steps until $k = k_{max}$:*
 (a)Exploration of neighborhood.
 Find the best neighbor x' of x ($x' \in N_k(x)$);
 (b)Move or not. If the solution thus obtained x' is better than x ,
 set $x \leftarrow x'$ and $k \leftarrow 1$; otherwise, set $k \leftarrow k + 1$

Figure 4.3 Steps of Variable Neighborhood Descent (VND)

Mainly in most cases as a local search a single or sometimes two neighborhoods are used, i.e., $k_{max} \leq 2$. It is indicated in the research of Hansen et al. (2010), that the last solution should be a local minimum w.r.t all k_{max} neighborhoods, and the possibility of reaching the global minimum is higher than using a single neighborhood structure.

4.3 Reduced Variable Neighborhood Search (RVNS)

In *reduced VNS* (RVNS) method, random points are selected from the $N_k(x)$ neighborhood without being followed by descent. In a more detailed manner, starting with an initial solution x , a random solution is chosen from the first neighborhood, if this random solution made any improvement, the search continues with that solution and continues to iterate until no improvement is found on that neighborhood,

otherwise a jump is made to the next neighborhood. The algorithm continues until $k = k_{max}$.

Basic steps of RVNS can be examined below.

Initialization: Select the set of neighborhood structures N_k for $k = 1, \dots, k_{max}$, that will be used in the search; find an initial solution x ; choose a stopping condition;

Repeat the following sequence until the stopping condition is met:

(1) *Set $k \leftarrow 1$;*

(2) *Repeat the following steps until $k = k_{max}$:*

(a) *Shaking: Generate a point x' at random from the k^{th} neighborhood of x ($x' \in N_{k'}(x)$);*

(b) *Move or not: If this point is better than the incumbent, move there ($x \leftarrow x'$), and continue the search with $N_1(k \leftarrow 1)$; otherwise set $k \leftarrow k + 1$*

Figure 4.4 Steps of Reduced Variable Neighborhood Search (RVNS)

As it is asserted in Hansen et al. (2010), “RVNS is useful for very large instances for which local search is costly. It is observed that the best value for the parameter k_{max} is often 2. In addition, the maximum number of iterations between two improvements is usually used as stopping condition”.

4.4 Basic Variable Neighborhood Search (BVNS)

Basic VNS method includes both deterministic and stochastic changes of neighborhood. Firstly, neighborhood structures used in the BVNS algorithm are defined as N_k for $k = 1, \dots, k_{max}$. Afterwards, starting with an initial solution, randomly a solution x' is taken from the first neighborhood $N_1(x)$ of x and a descent x' is done with the local search. This leads to a new local minimum x'' .

At that moment, three alternatives can be encountered:

(i) $x'' = x$, i.e. when no improvement is achieved after local search, then a move to the next neighborhood is carried out, which means that $N_k(x)$, $k \geq 2$;

(ii) $x'' \neq x$ but $f(x'') \geq f(x)$, i.e., another local optimum has been found, which is not better than the previous incumbent solution; similar to the previous case, the procedure is iterated using the next neighborhood $N_k(x)$, $k \geq 2$.

(iii) $x'' \neq x$ but $f(x'') < f(x)$: i.e., another local optimum, better than the incumbent has been found; in this case the search restart with the new incumbent solution x'' at the first neighborhood $N_1(x)$ until a stopping criteria, e.g. a maximum time or maximum number of iterations or maximum number of iterations from the last improvement is met. Steps of Basic VNS can be seen as follows.

Initialization: Select the set of neighborhood structures N_k for $k = 1, \dots, k_{max}$, that will be used in the search; find an initial solution x ;
choose a stopping condition;
Repeat the following sequence until the stopping condition is met:
 (1) Set $k \leftarrow 1$;
 (2) Repeat the following steps until $k = k_{max}$:
 (a) *Shaking: Generate a point x' at random from the k^{th} neighborhood of x ($x' \in N_{k'}(x)$);*
 (b) *Local Search: Apply some local search method with x' as initial solution; denote with x'' the so called local minimum;*
 (c) *Move or not: If this local optimum is better than the incumbent, move there ($x \leftarrow x''$),*
and continue the search with $N_1(k \leftarrow 1)$; otherwise set $k \leftarrow k + 1$

Figure 4.5 Steps of Basic Variable Neighborhood Search (BVNS)

As Hansen et al. (2010), claimed in their work, “often successive neighborhoods N_k will be nested. Observe that point x' is generated at random in *Shaking* in order to avoid cycling, which might occur if any deterministic rule was used. However, some difficulties can be encountered while solving large instances using basic VNS”.

Basic VNS has several extensions. In the *move or not*, even if the solution is worse than the incumbent, that solution can be accepted with some probability. Also in the *Local Search* phase, *First Improvement* or *Best Improvement* can be used.

4.5 General Variable Neighborhood Search (GVNS)

General VNS is a variate of the Basic VNS where in the local search phase, VND is used (Mladenović et al., 2008).

*Initialization: Select the set of neighborhood structures N_k for $k = 1, \dots, k_{max}$, that will be used in the search; find an initial solution x ;
choose a stopping condition;
Repeat the following sequence until the stopping condition is met:*

(1) Set $k \leftarrow 1$;

(2) Repeat the following steps until $k = k_{max}$:

- (a) *Shaking: Generate a point x' at random from the k^{th} neighborhood of x ($x' \in N_{k'}(x)$);*
- (b) *Local Search: Apply VND a local search method with x' as initial solution; denote with x'' the so called local minimum;*
- (c) *Move or not: If this local optimum is better than the incumbent, move there ($x \leftarrow x''$),*

and continue the search with $N_1(k \leftarrow 1)$; otherwise set $k \leftarrow k + 1$

Figure 4.6 Steps of General Variable Neighborhood Search (GVNS)

4.6 Skewed Variable Neighborhood Search (SVNS)

As it is emphasized on Fact 3 of VNS in Section 4.1, local minima with respect to one or more neighborhoods are close to each other (Hansen et al., 2010). Therefore, schemes VNS should be adjusted so as to explore valleys more entirely which are far away from the incumbent solution. This approach is denoted as *Skewed VNS* (SVNS).

Different than BVNS, in SVNS there is a function $\rho(x, x'')$ that measures the distance between the incumbent solution x and the local optimum found x'' used together with the parameter α . The parameter α stands for accepting solutions that are far from x when $f(x'')$ is larger than $f(x)$ but not too much (otherwise one will always leave x). A remarkable value of α should be determined experimentally. Moreover, in order not to move frequently from a solution x to a solution that is not

so far away, a large value for α can be taken when $\rho(x, x'')$ is small (Hansen et al., 2010)

4.7 Variable Neighborhood Decomposition Search (VNDS)

When solving very large instances VNS can sometimes be very weak. Therefore, another variant is emerged, called Variable Neighborhood Decomposition Search (VNDS) method proposed by Hansen & Mladenović (2001), which divides the basic VNS into a two-level VNS scheme based upon decomposition of the problem (Hansen et al., 2010).

4.8 Hybrid Approaches with Variable Neighborhood Search

On the other hand, hybrid approaches, i.e., a combination of metaheuristics and VNS is used as a powerful mechanism in the literature. For example, *Tabu Search* (TS) and VNS can be hybridized. One alternative is to use TS within VNS another alternative is to use VNS within TS (Hansen et al., 2010). Moreover, GRASP and VNS combination is also used for many problems, where VNS mainly used in the second phase of GRASP (Hansen et al., 2010). On the other hand, Particle Swarm Optimization is combined with VNS in a case study for parallel machine scheduling problem (Chen et al., 2013). Behnamian et al. (2009), introduced the use of VNS with Ant Colony Optimization again on parallel machine scheduling problem. Besides, Liu & Zhou (2013), proposed a hybrid algorithm for the restrictive single-machine earliness/tardiness problem where harmony search is combined with VNS. Another combination of VNS with Fix&Optimize Heuristic is stated in the work of Seeanner et al. (2013). A combination of VND with iterated local search (ILS) is proposed by Martins et al. (2012), for the routing and wavelength assignment problem. Furthermore, Shifting Bottleneck procedure is also used with the combination of VNS (Reiter et al., 2013). A hybrid VNS and Simulated Annealing (SA) algorithm is presented by Abbasi et al. (2011), in a study where the three parameters of Weibull distribution is estimated.

4.9 Applications of Variable Neighborhood Search in Industry

In the literature it can be found that there are many applications of VNS and its hybrids, on various different areas. These areas can be listed as follows:

- “industrial applications”,
- “design problems in communication”,
- “location problems”,
- “data mining”,
- “graph problems”,
- “knapsack and packing problems”,
- “mixed integer problems”,
- “time tabling”,
- “scheduling”,
- “vehicle routing problems”,
- “problems in biosciences and chemistry”,
- “continuous optimization and other optimization problems” (Hansen et al., 2010).

CHAPTER FIVE

CONSTRAINT HANDLING TECHNIQUES

In this chapter, a comprehensive study about constraint handling techniques including its definition, aim of use and its classification are presented.

As Deb (2000), formerly defined in his study, the origin of the constraint handling techniques is as follows, “Many real-life optimization problems have inequality and/or equality constraints and therefore denoted as constrained optimization problems. While trying to solve these type of optimization problems using evolutionary algorithms or classical optimization tools, penalty function techniques become popular since they are simple and easy to implement. Unfortunately, the success level of penalty functions are not satisfactory at all time since finding appropriate parameters is difficult”.

Furthermore, the reason why constraint handling techniques emerged is that in most cases, the optimal solution can be found on the boundaries of the feasible region. Therefore, limiting the search within the feasible solutions’ region only or imposing very severe penalties for infeasible solutions makes it difficult to understand the optimum solution path as shown in the research of (Smith & Tate, 1993; Anderson & Ferris, 1994; Coit et al., 1995; Michalewicz, 1995). Contrarily, if the penalty is not severe enough, then search region will be too large and much of the search time will be used to explore regions far from the feasible region (Smith & Tate, 1993). In other words, if the penalty is too high and the optimum solution is close to the boundary of the feasible region, the search region will be restricted within the feasible region in the very beginning of the search process, accordingly, returning back towards the boundary with the infeasible region will be discarded and the possibility of exploration of the infeasible region will be less (Coello, 2002).

For example, Deb (2000), generated a constraint handling method emphasizing the superiority of feasible solutions. The algorithm lies on the idea of the following three conditions:

1. “Any feasible solution is preferred to any infeasible solution.”
2. “Among two feasible solutions, the one having better objective function value is preferred.”
3. “Among two infeasible solutions, the one having smaller constraint violation is preferred.”

The most important difference of Deb’s (2000), research is that, there is no penalty parameter. According to the three aspects mentioned above, careful pair-wise comparisons are made and a direction to the feasible region is achieved.

On the other hand, Takahama & Sakai (2005, 2006, 2010), introduced another constraint handling approach called the *ε -constrained method*. In the *ε -constrained method*, the relaxation of the constraints is controlled by using the ε parameter. Also, this method has a lexicographical ordering mechanism in which the minimization of the sum of constraint violation comes prior to the minimization of the objective function of a given problem. In other words, solution having a total violation smaller than ε are labeled as feasible when a pair-wise comparison is made.

As it is mentioned in Coello’s study (2002), the most common approach for handling constraints is to use penalties. It is explained that the relationship between an infeasible solution and the feasible region of the search space plays a significant role in penalizing such a solution. There are three main alternatives to identify this relationship.

1. A solution might be penalized just for being infeasible regardless of its amount of constraint violation
2. The amount of its infeasibility can be measured and used to determine its corresponding penalty, or
3. The effort of ‘repairing’ the solution (i.e., the cost of making it feasible) might be taken into account.

In order to explain a general penalty function, at first, assume that an optimization problem is given as,

$$\begin{aligned} \min f(x) \\ \text{s. t. } x \in A \text{ and } x \in B \end{aligned} \tag{5.1}$$

where x is a vector of decision variables, the constraints " $x \in A$ " are relatively easy to satisfy, and the constraints " $x \in B$ " are more difficult to satisfy when they are compared to the constraints " $x \in A$ ".

The problem can be reformulated as follows:

$$\begin{aligned} \min f(x) + p(d(x, B)) \\ \text{s. t. } x \in A \end{aligned} \tag{5.2}$$

"Where $d(x, B)$ is a metric function describing the distance of the solution vector x from the region B , and $p(\cdot)$ is a monotonically non-decreasing penalty function such that $p(0) = 0$. If the exterior penalty function, $p(\cdot)$, grows quickly enough outside of B , the optimal solution of (5.1) will also be optimal for (5.2). Furthermore, any optimal solution of (5.2) will provide an upper bound on the optimum for (5.1), and this bound will in general be tighter than that obtained by simply optimizing $f(x)$ over A " (Smith & Tate, 1993).

There are six types of penalty functions defined in Coello's research (2002), which can be listed as follows,

- Static Penalty
- Dynamic Penalty
- Annealing Penalty
- Adaptive Penalty
- Co-evolutionary Penalty
- Death Penalty

5.1 Static Penalties

Static penalties imply that the penalty remain constant during the whole search process. In other words, disregarding of violation amount, a constant penalty is applied. On the other hand, penalty function can also be designed as metric based depending on the number of constraints violated.

This penalty function is based only on the number of constraints violated, and is generally less valuable than another approach where some distance metric from the feasible region is considered as generating the penalty function (Goldberg, 1989; Richardson et al., 1989). Moreover, another approach incurred by Homaifar, Lai & Qi (1994), has a user defined violation level mechanism and penalty coefficients for each level is selected accordingly, in a more detailed way, when violation level l increases, penalty coefficient rises, as well. The weakness of this method is in the number of parameters: for m constraints the method requires $m(2l+1)$ parameters in total.

Besides, there is another approach, where a solution's fitness value is not computed if it is infeasible. Moreover, all infeasible solutions deserve the same penalty value even if its violation amounts vary from each other, it only deals with the number of violated constraints (Michalewicz, 1995).

Hoffmeister and Sprave (1996) introduced also a penalty function where infeasible solutions are always worse than the feasible ones, however, the reverse can be true in some cases.

5.2 Dynamic Penalties

Different than static penalty functions, in dynamic penalties, current iteration number is considered as another criteria for the penalty functions. In fact, annealing penalties and adaptive penalties are dynamic penalties, too (Coello, 2002).

Furthermore, with dynamic penalty functions, in the beginning of the search, solutions with high infeasibility are not ignored, however, the penalty will increase when there is a move to a solution that lies on the feasible region during the search phase. In other words, the basic idea of using a dynamic approach is to incorporate a dynamic aspect which (generally) raises the level of the penalty while the search continues.

For example, Joines & Houck (1994) generated an approach where there are too many user defined constants like C , α , and β . Thus, the solution is very sensitive depending on the values of C , α , and β . Michalewicz (1995), indicated that these parameters produce premature convergence most of the time in other examples.

On the other hand, Kazarlis & Petridis (1998) proposed another approach, *Varying Fitness Function Technique (abbreviated as VFF)*, where again various parameters are used and also, these parameters did not have a standard, i.e., they can vary depending on the problem type.

5.3 Annealing Penalties

In most of the annealing penalties, the idea of simulated annealing is used as a base. For example, Michalewicz & Attia (1994) considered a method where the penalty coefficients are changed once in many generations (after the algorithm has been trapped in a local optima). Only active constraints are considered at each iteration, and the penalty is increased over time (i.e., the temperature decreases over time) so that infeasible individuals have more severe penalties at the end of the search phase.

Similar to the other dynamic approaches, Michalewicz & Attia's approach (1994) has a disadvantage in that solution quality can have huge changes when parameters change, i.e. it is too much sensitive to the values of its parameters, also, choosing an appropriate cooling schedule is difficult (Coello, 2002).

Joines & Houck's approach (1994) is also based on simulated annealing, however, has an exponential term and this exponential part would sometimes become invalid due to the numerical range of the computer. On the other hand, Joines and Houck's approach is easier to implement since it does not distinguish between linear and non linear constraints and its authors leave to the evolutionary algorithm (EA) itself the task of generating feasible solutions from an initial set of random values (Coello, 2002).

On the other hand, Carlson (1995), introduced a different method where two parameters have an influence on the penalty, the first one measures a constraint's violation amount, whereas the second one is a function of the running time of the algorithm. However, in this method, these parameters were obtained empirically and work well for some engineering problems.

5.4 Adaptive Penalties

In adaptive penalties, search duration and distance from feasibility is considered together (Smith & Tate, 1993; Tate & Smith, 1995). Since neighborhood generation operators used in this study may result in infeasible solutions, NFT approach is used to handle the violation of constraints.

In the works of Smith & Tate (1993), and Tate & Smith (1995), which were enhanced by the work of Coit et al. (1995), length of search and severity levels of constraints are used in their penalty function. In this penalty function, a near-feasible threshold (*NFT*) is computed for each constraint. The *NFT* is defined as a threshold distance from feasible region. With the penalty function, the algorithm is encouraged to explore within the feasible region and the *NFT*-neighborhood of the feasible region. In a more detailed way, the search that is surpassed the threshold is discouraged more severely (Smith & Tate, 1993; Tate & Smith, 1995). The formulation is below:

$$f_p(x, t) = f(x) + (F_{feas}(t) - F_{all}(t)) \sum_{i=1}^m \left(\frac{d_i}{\underbrace{NFT_i}_{\text{dynamic part of NFT}}} \right)^K \quad (5.3)$$

As it is stated in equation (5.3), $F_{all}(t)$ denotes the unpenalized value of the best solution yet found, and $F_{feas}(t)$ denotes the value of the best feasible solution yet found. The $F_{all}(t)$ and $F_{feas}(t)$ terms serve several purposes. On the other hand, d_i denotes the violation amount of constraint i , where there are m constraints in total. First, they provide adaptive scaling of the penalty based on the results of the search. Second, they combine it with the NFT_i term to provide a search specific and constraint specific penalty (Smith & Tate, 1993; Tate & Smith, 1995).

The general form of NFT is:

$$NFT = \frac{NFT_0}{1+\Lambda} \quad (5.4)$$

According to the proposed equation (5.4), NFT_0 is an upper bound of NFT . Λ is a dynamic search parameter which is used to update NFT considering the whole search period. Λ can also be defined as a function of the search, for example, a function of the generation number (t), i.e., $\Lambda = f(x) = \lambda t$ (Baeck et al., 1995). When λ becomes positive, then NFT would be monotonically decreasing function and accordingly a larger penalty occurs. As a result, when λ increases, NFT decreases faster as the search continues.

Gen & Cheng (2000), enhanced NFT by adding a more severe penalty for infeasible solutions. Moreover, as noted in Gen & Cheng (2000), the adaptive term may lead to zero or over-penalty. For instance, if $F_{feas}(t)$ and $F_{all}(t)$ are identical, the penalty would be zero, resulting in unpenalized infeasible solutions. For this reason, only the dynamic part of the penalty function, denoted in equation 5.3, with NFT threshold is used.

5.5 Co-evolutionary Penalties

Coello (2000), introduced a penalty function where objective function for the given set of variable values are encoded in a chromosome. Main drawback of this method is that four parameters needed to be defined. Also, the values of these

parameters should be carefully determined because a little change of these values have a huge effect on the penalty function (Coello, 2002).

5.6 Death Penalties

Death penalty means rejecting infeasible solutions. Rejection of infeasible solutions is computationally easy since level of violation is not considered. Furthermore, when a constraint violation occurred for a solution, a zero fitness value is assigned to that solution. However, when there is no initial feasible solution, no further improvement can be made and accordingly the evolutionary process will stuck because all solutions have the same fitness value, zero (Coello, 2002).

As a result, in this study two techniques of constraint handling were utilized. Firstly, if a solution is infeasible, then it is penalized by using *NFT*. Secondly, the *superiority of feasible solutions* is used when making a selection between two solutions during the search phase.

CHAPTER SIX

PROPOSED ALGORITHMS

In this chapter, algorithms that are used in this study are explained. More specifically, initial solution generation, objective function calculation, and moves used are clarified by using various examples.

6.1 Initial Solution Generation

Firstly, products are selected in a random order. Starting from the last period, a randomly chosen product's whole demand on that period is placed to one of the capable machines. In addition, while assigning the selected product's whole demand on that period, the machine with the minimum usage is chosen among the capable ones. On the other hand, when a product's whole demand cannot be produced on the selected machine on that period, i.e., when there is not enough capacity on the chosen machine for that period, then a part of the demand is assigned to that machine until its capacity is full and the remaining demand of that product is assigned to the previous period on the same machine since setup carryover is allowed among periods. Furthermore, when a part of the demand assigned to the previous period, it will have an additional effect on the objective function in terms of inventory holding cost. The initial solution algorithm stops when all products' demands are assigned to the machines. In order to visualize the initial solution generation method, a small example is provided with the following data presented in Table 6.1-6.4.

Table 6.1 Demand matrix

Product	Demand in period	
	Period 1	Period 2
1	2	2
2	2	1
3	1	3
4	3	4
5	2	2

Table 6.2 Capability matrix

Product	Capability of Machines	
	Machine 1	Machine 2
1	1	1
2	1	1
3	0	1
4	1	1
5	0	1

Table 6.3 Setup matrix

Product	Setup time to product				
	1	2	3	4	5
1	0	1	2	1	1
2	2	0	1	2	2
3	2	1	0	2	2
4	1	2	1	0	1
5	2	1	2	1	0

Table 6.4 Machine capacities by period

Machine	Capacity in Period	
	Period 1	Period 2
1	10	10
2	10	10

Assume that each product's production time is 1 unit and assume that firstly product 5 is chosen randomly. Since it can only be produced on machine 2, it is placed as it can be seen in Figure 6.1.

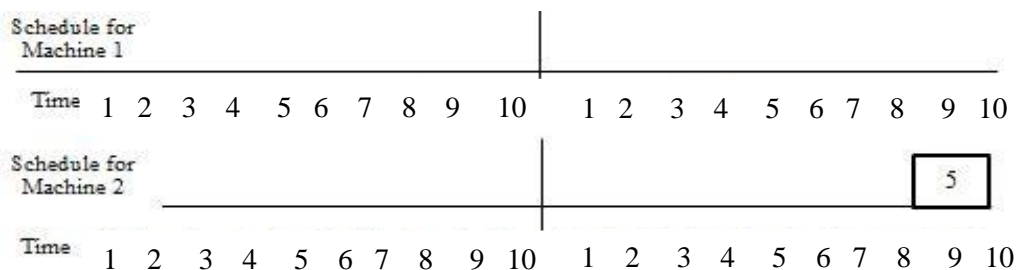


Figure 6.1 Schedules for both machines (after product 5 is assigned)

Then, product 1 is chosen randomly. By looking at the capability matrix, product 1 can be produced on both machines. Considering the machine usages, machine with the minimum current usage is chosen.

Afterwards, product 3 can only be produced on the second machine. Then, product 2 has two alternatives. Since machine 1 has less usage, it is assigned to machine 1. Finally, product 4 is assigned to machine 1 since it has the minimum usage. Product 4 cannot be fully produced on period 2 therefore, the remaining part is assigned to period 1.

Current schedules of both machines are as follows:

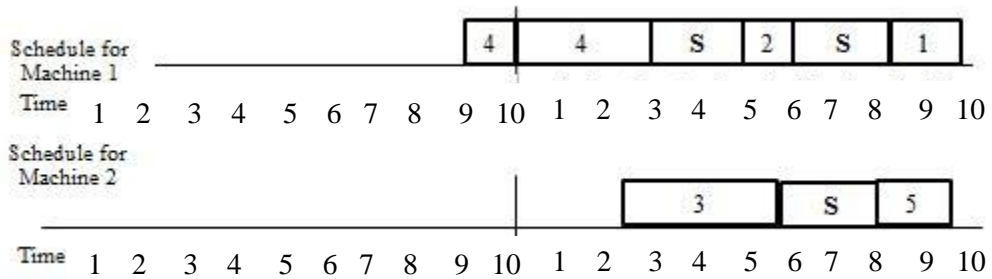


Figure 6.2 Schedules for both machines (after all products are assigned for period 2)

Assume that product 1 is chosen. Since it can be produced on both machines, it is assigned to machine 2. Then, product 4 has also two alternatives. Since the usage of machine 1 is less, it is assigned to machine 1. Product 5 can be produced only on machine 2. Afterwards, product 3 is assigned to machine 1 due to the machine capability. Finally, product 2 is assigned to machine 1 because it has the minimum usage.

The final schedule for both machines is as follows:

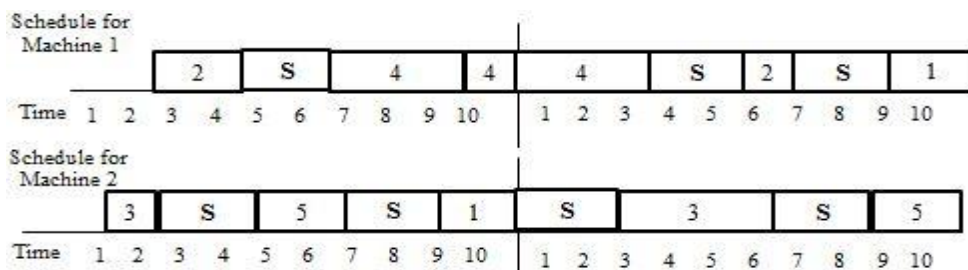


Figure 6.3 Final schedules for both machines

Since all products' demands are assigned, the algorithm stops. Feasibility is not a must criteria to be met because constraint handling techniques are utilized. The pseudocode for the initial solution generation can be examined in Figure 6.4 below.

```

INITIALIZE:  $t = t_{max}$ 
WHILE  $t > 0$ {
    Choose randomly a product  $i$  from the set for products  $I$  that has a positive demand
     $d_{it}$  on period  $t$ 
    // Find a capable machine
    IF there is only a single capable machine for product  $i$ , i.e.,
         $\sum_m A_{mi} = 1$  for  $i$  Assigned_machine:  $m$ , where  $A_{mi} = 1$ 
    ELSEIF (if there is more than one capable machine for product  $i$ ,  $\sum_m A_{mi} > 1$  for  $i$ )
        Assigned_machine: find  $m$  where  $m$  has (min (usage( $m, t$ ))) for all
         $A_{mi} > 0$  for  $i$ 
    ENDIF
    // Capacity Check
    IF  $usage(m, t) + d_{it} \leq capacity_{mt}$ 
        Assign whole demand  $d_{it}$  to that period  $t$ 
    ELSEIF
        // Calculate the maximum quantity as assignable quantity that can be assigned to
        that period
         $assignable\ quantity = capacity_{mt} - usage(m, t)$ 
        // If there is not enough capacity on the previous period  $t-1$  on the same machine,
        calculate the capacity violation amount
        IF  $assignable\ quantity = 0$ 
            Calculate violation amount for that (machine-period) pair
        ELSEIF
            Assign calculated assignable quantity to that period
        // Calculate the remaining quantity
         $remaining\ quantity = d_{it} - assignable\ quantity$ 
        // Assign this remaining quantity to the previous period  $t-1$  on the same
        machine

        ENDIF
    ENDIF
} // Repeat until all products' demands are assigned for period  $t$  ( $t = t - 1$ )
ENDWHILE

```

Figure 6.4 Pseudocode for the initial solution generation

6.2 Objective Function Calculation

Objective function calculation is somehow different than the regular calculation where in the regular calculation only setup costs and inventory holding costs are incurred. However since *NFT* is used, which is defined in Section 5.4, infeasible solutions are also accepted with some violation amount. In other words, a near-feasible threshold (*NFT*) is computed for each violating constraint (Smith & Tate, 1993; Tate & Smith 1995). During the construction of the initial solution, when there is no capacity limitation, then the whole demand is produced. In other words, since in the initial solution generation phase, the assignment of the lots is started from the last period, there will not be any tardiness, i.e., whole demand would be satisfied. Thus, only capacity violation amount is calculated as the *total violation* for the initial solution.

However, the situation differs for the solutions generated after infeasible moves. The whole demand might not be satisfied since there is no limitation for the moves. For example, when two products are swapped from different periods, there is no guarantee that the demand for one product or even both products would definitely be satisfied. Therefore, in addition to the capacity violation amount, unsatisfied demand amount and excess of the upper bound production levels of each product on each machine should also be included in the calculation of *total violation* amount, after a move is made.

6.3 Moves

Three different types of moves have been defined previously by Almada-Lobo & James (2010), for the single machine case. However, in this study, those three moves are applied for the parallel multi machine case. These moves can be explained as follows.

6.3.1 Insert Move

In the insert move, randomly a whole product lot is taken and inserted randomly before another product. Different from the single machine case explained in Almada-Lobo & James (2010), this move can be done within that machine and among multiple machines while considering machine's ability of producing that product. In a more detailed way, selected product can only be inserted to another machine if it can be produced on there. On the other hand, since constraint handling is applied, an insert move can be done, although capacity of the machines and upper bound production quantity are exceeded.

An example the sample data presented in Table 6.5-6.8 is given. Assume that each product's production time is 1 unit.

Table 6.5 Demand matrix

Product	Demand in period	
	Period 1	Period 2
1	2	2
2	2	1
3	1	3
4	3	3
5	2	2

Table 6.6 Capability matrix

Product	Capability of Machines	
	Machine 1	Machine 2
1	1	1
2	1	1
3	0	1
4	1	1
5	0	1

Table 6.7 Setup matrix

Product	Setup time to product				
	1	2	3	4	5
1	0	1	2	1	1
2	2	0	1	2	2
3	2	1	0	2	2
4	1	2	1	0	1
5	2	1	2	1	0

Table 6.8 Machine capacities by period

Machine	Capacity in Period	
	Period 1	Period 2
1	10	10
2	10	10

Assume that, initial sequence is given in Figure 6.5 below.

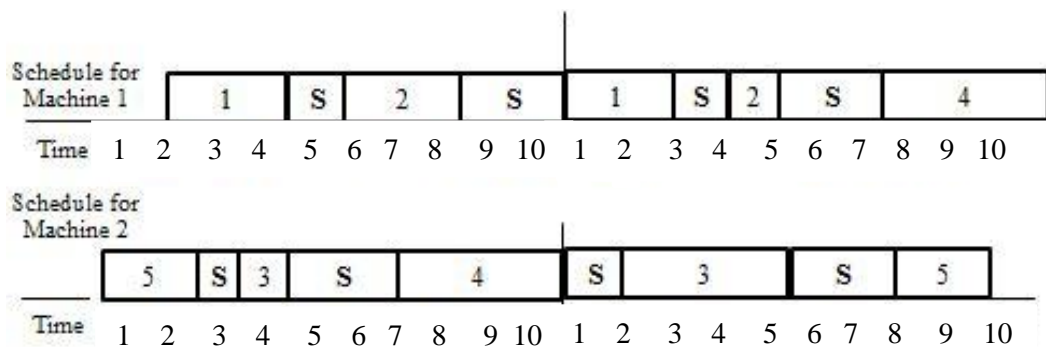


Figure 6.5 Initial schedule

An insert move of product 2 on machine 1 on period 2 can be done as follows.

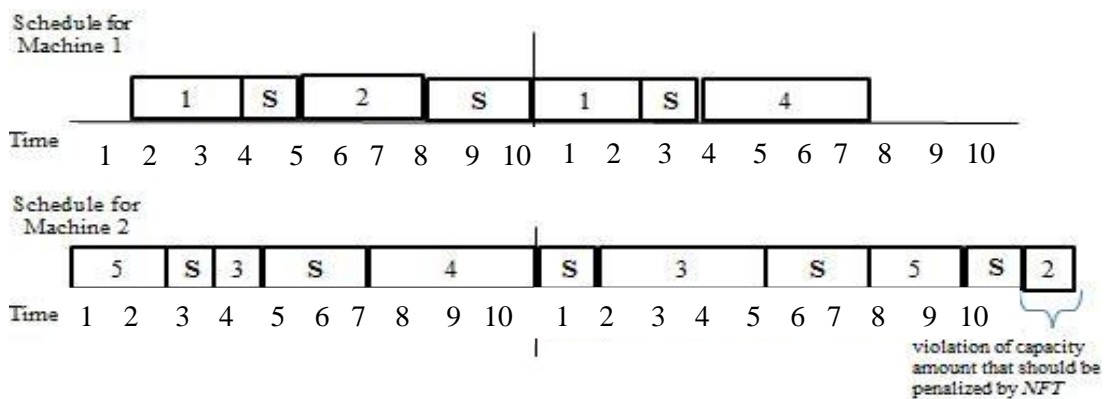


Figure 6.6 Schedule obtained after an insert move

As it can be seen in Figure 6.6 above, even if it results in an infeasible solution in terms of the excess capacity of 1 unit on period 2 on machine 2, whole lot of product 2 on machine 1 is inserted on machine 1 after product 5. The following pseudocode, proposed in Figure 6.9, indicates the flow of a single insert move.

$$\begin{aligned}
 & \text{SetupUpdate}(\text{setup}_{old,origin}, \text{setup}_{old,dest}) \\
 & \text{setup}_{old,origin} = S_{m_{origin},i_{origin-1},i_{origin}} + S_{m_{origin},i_{origin},i_{origin+1}} \\
 & \text{setup}_{new,origin} = S_{m_{origin},i_{origin-1},i_{origin+1}} \\
 & \text{setup}_{old,dest} = S_{m_{dest},i_{dest-1},i_{dest}} + S_{m_{dest},i_{dest},i_{dest+1}} \\
 & \text{setup}_{new,dest} = S_{m_{dest},i_{dest-1},i_{origin}} + S_{m_{dest},i_{origin},i_{dest+1}}
 \end{aligned}$$

Figure 6.7 Pseudocode for setup updates used in this study

$$\begin{aligned}
 & \text{UpdateAll}(\text{usage}(m_{origin}, t_{origin}), \text{usage}(m_{dest}, t_{dest}), \pi_{m_{origin},t_{origin}}, \pi_{m_{dest},t_{dest}}) \\
 & \text{usage}(m_{origin}, t_{origin}) = \\
 & \text{usage}(m_{origin}, t_{origin}) - \text{setup}_{old,origin} + \text{setup}_{new,origin} - X_{m_{origin},i_{origin},t_{origin}} \\
 & \text{usage}(m_{dest}, t_{dest}) = \\
 & \text{usage}(m_{dest}, t_{dest}) - \text{setup}_{old,dest} + \text{setup}_{new,dest} + X_{m_{origin},i_{origin},t_{origin}}
 \end{aligned}$$

Figure 6.8 Pseudocode for the update of all variables

In Figure 6.7, necessary calculations related with the setups are made. On the other hand, using the updated setups, other updates such as usages and sequences are done in Figure 6.8. Additionally, *SetupUpdate* and *UpdateAll* functions are used in other moves as well.

```

insert( $\pi$ ) {
SELECT randomly a period  $t_{origin}$ , where  $t_{origin} \in [T]$  for the origin
SELECT randomly another period  $t_{dest}$ , where  $t_{dest} \in [T]$  for the destination
SELECT randomly an origin machine  $m_{origin}$ , where  $m_{origin} \in [M]$ 
SELECT randomly a destination machine  $m_{dest}$ , where  $m_{dest} \in [M]$ 
SELECT randomly a product  $i_{origin}$  from the sequence  $\pi_{m_{origin}, t_{origin}}$ , where  $i_{origin} \in [N]$ .
IF destination machine  $m_{dest}$  is capable of producing product  $i_{origin}$ , i.e.,  $A_{m_{dest}, i_{origin}} = 1$ 
    SELECT randomly a destination_index from the sequence  $\pi_{m_{dest}, t_{dest}}$ .
    destination_index is a random place where a fraction of the demand
     $X_{m_{origin}, i_{origin}, t_{origin}}$  of product  $i_{origin}$  that is currently produced on  $m_{origin}$  at period
     $t_{origin}$ , will be placed before. Currently, the product on the destination_index is
    denoted as  $i_{dest}$ .
ELSEIF // If destination machine  $m_{dest}$  is not capable of producing product  $i_{origin}$ ; repeat
the following step until there is a product  $i_{origin}$  that can be produced on the destination
machine  $m_{dest}$  is found.
    WHILE  $A_{m_{dest}, i_{origin}} = 0$ , i.e., destination machine is not capable
    of producing product  $i_{origin}$ :
    SELECT randomly another product  $i_{origin}$  from the sequence  $\pi_{m_{origin}, t_{origin}}$  from the
    origin machine  $m_{origin}$  where  $i_{origin} \in [N]$ , which destination machine is capable of,
    i.e.,  $A_{m_{dest}, i_{origin}} = 1$ .
ENDIF
//update setups, current capacity usages and sequences of machines after insert
SetupUpdate( $setup_{old, origin}$ ,  $setup_{old, dest}$ )
UpdateAll( $usage(m_{origin}, t_{origin})$ ,  $usage(m_{dest}, t_{dest})$ ,  $\pi_{m_{origin}, t_{origin}}$ ,  $\pi_{m_{dest}, t_{dest}}$ )
}

```

Figure 6.9 Pseudocode for the insert move used in this study

In the insert move, insertion is done only capability condition is satisfied. Moreover, insertion within the same machine is also allowed. An insert move can also lead to infeasible solutions in terms of capacity constraints. However it is not an important criteria because violation amount will be penalized in the objective function.

6.3.2 Swap Move

In the swap move, randomly two product lots are selected and swapped without considering any violation. The only criteria that must be checked before swapping is whether that machine(s) is capable of producing the selected product or not.

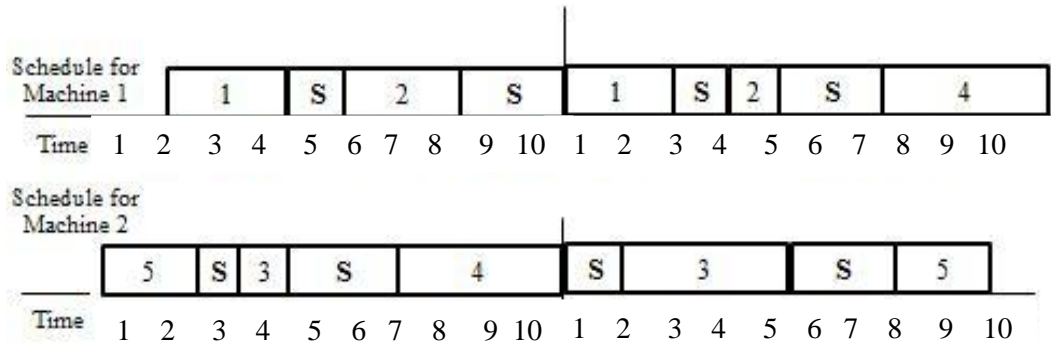


Figure 6.10 Initial schedule

A swap move can be done as follows.

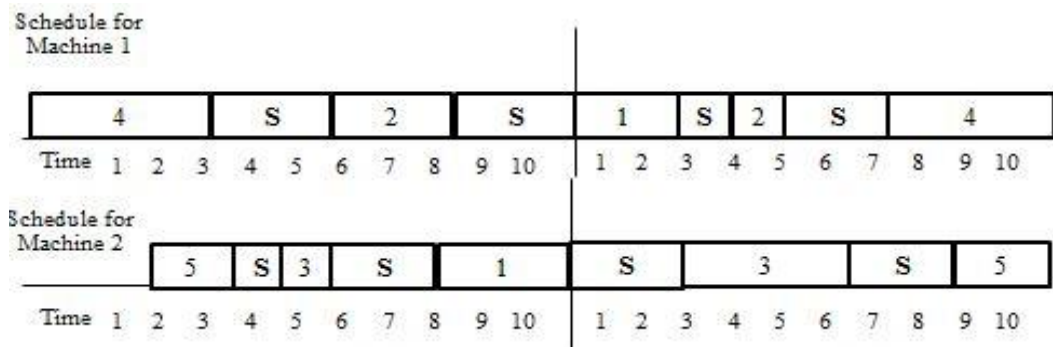


Figure 6.11 Schedule obtained after a swap move

After a random single swap move of product 1 on machine 1 on the first period and product 4 on machine 2 on the first period, final sequence can be seen as in Figure 6.11 above. The pseudocode of the one single swap move can be visualized in Figure 6.12.

```

swap( $\pi$ ) {
Starting with an initial production sequence on machine  $m$  for period  $t$ , which is denoted as
 $\pi_{m,t}$ : for all  $t$  and for all  $m$ , where  $t \in [T]$  and  $m \in [M]$ .
SELECT randomly a period  $t_{origin}$ , where  $t_{origin} \in [T]$  for the origin
SELECT randomly another period  $t_{dest}$ , where  $t_{dest} \in [T]$  for the destination
SELECT randomly an origin machine  $m_{origin}$ , where  $m_{origin} \in [M]$ 
SELECT randomly a destination machine  $m_{dest}$ , where  $m_{dest} \in [M]$ 
SELECT randomly an origin index  $origin\_index$  where product  $i_{origin}$  from the sequence
 $\pi_{m_{origin}, t_{origin}}$  is currently produced and has a current production lot  $X_{m_{origin}, i_{origin}, t_{origin}}$ ,
and SELECT randomly a  $destination\_index$  from the sequence  $\pi_{m_{dest}, t_{dest}}$ , where the lot
 $X_{m_{dest}, i_{dest}, t_{dest}}$ , will be swapped with the lot  $X_{m_{origin}, i_{origin}, t_{origin}}$ . Currently, the product on
the  $destination\_index$  is denoted as  $i_{dest}$ .
IF destination machine  $m_{dest}$  is capable of producing product  $i_{origin}$ , i.e.,  $A_{m_{dest}, i_{origin}} = 1$ 
and origin machine  $m_{origin}$  is capable of producing product  $i_{dest}$ , i.e.,  $A_{m_{origin}, i_{dest}} = 1$ :
ELSEIF // If destination machine  $m_{dest}$  is not capable of producing product  $i_{origin}$ ; or if
origin machine  $m_{origin}$  is not capable of producing product  $i_{dest}$ 
.
    WHILE  $A_{m_{dest}, i_{origin}} = 0$  or  $A_{m_{origin}, i_{dest}} = 0$ 
        SELECT randomly another product  $i_{origin}$  from the sequence  $\pi_{m_{origin}, t_{origin}}$ 
        from the origin machine  $m_{origin}$  and another product  $i_{dest}$  from the
        sequence  $\pi_{m_{dest}, t_{dest}}$ .
    ENDWHILE
ENDIF
//update current capacity usages and sequences of machines after the lot is inserted
SetupUpdate( $setup_{old, origin}, setup_{old, dest}$ )
UpdateAll( $usage(m_{origin}, t_{origin}), usage(m_{dest}, t_{dest}), \pi_{m_{origin}, t_{origin}}, \pi_{m_{dest}, t_{dest}}$ )
}

```

Figure 6.12 Pseudocode for the swap move used in this study

Similar to insert move, swap move considers only capability constraint. A swap move can be made after capability is satisfied for both products. On the other hand, similar to insert move, two product lots can be swapped within the same machine. Due to the use of constraint violation techniques, even if a swap move results in

infeasible solutions in terms of capacity, demand satisfaction, or upper bound production quantity, it would not be prohibited.

6.3.3 Fractional Insert Move

This move is similar to the insert move, however it allows the option of splitting a lot into two lots, where the total quantity produced is the same as the original lot. One of these new lots is left in the same position as the original lot, while second part is inserted randomly into a new location. On the other hand, capacity violation is not permitted in fractional insert move. If there is enough capacity, then all of the lot will be moved; if not then only the amount that can fit in the period will be moved. All locations within the period are tested as the capacity available in the period will vary depending on the position the new lot is inserted into and the lots surrounding it because of the sequence dependency of the setup times.

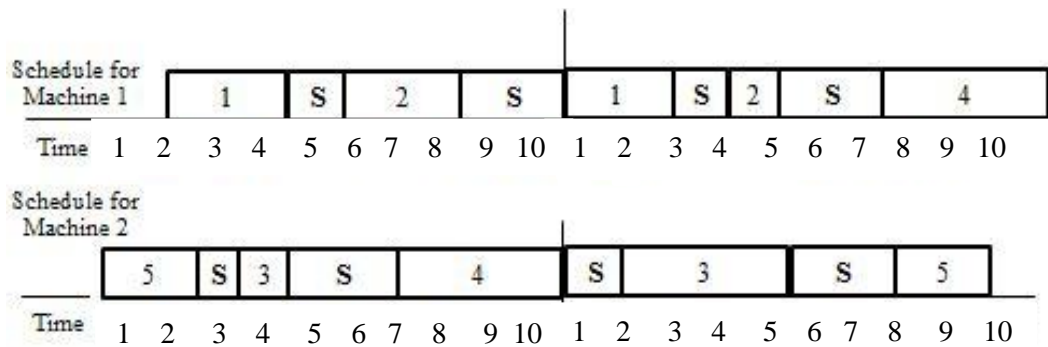


Figure 6.13 Initial schedule

An example of fractional insert move is as follows.

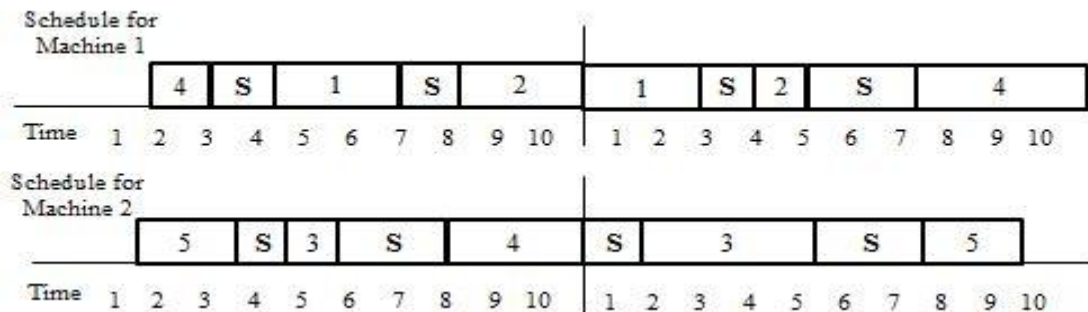


Figure 6.14 Schedule obtained after a fractional insert move

As it can be observed in Figure 6.14, product 4 on machine 2 on the first period is splitted into two parts. One part is inserted before product 1 on machine 1 on period 1.

```

fractional insert( $\pi$ ) {
Starting with an initial production sequence on machine  $m$  for period  $t$ , which is denoted as  $\pi_{m,t}$ : for all  $t$  and for all  $m$ , where  $t \in [T]$  and  $m \in [M]$ .
SELECT randomly a period  $t_{origin}$ , where  $t_{origin} \in [T]$  for the origin
SELECT randomly another period  $t_{dest}$ , where  $t_{dest} \in [T]$  for the destination
SELECT randomly an origin machine  $m_{origin}$ , where  $m_{origin} \in [M]$ 
SELECT randomly a destination machine  $m_{dest}$ , where  $m_{dest} \in [M]$ 
//If there is no free space on the destination machine  $m_{dest}$ , search
WHILE  $capacity(m_{dest}, t_{dest}) \leq usage(m_{dest}, t_{dest})$ 
    SELECT randomly another destination machine  $m_{dest}$ .
ENDWHILE
SELECT randomly a product  $i_{origin}$  from the sequence  $\pi_{m_{origin}, t_{origin}}$ , where  $i_{origin} \in [N]$ .
IF destination machine  $m_{dest}$  is capable of producing product  $i_{origin}$ , i.e.,  $A_{m_{dest}, i_{origin}} = 1$ :
    SELECT randomly a destination_index from the sequence  $\pi_{m_{dest}, t_{dest}}$ .
    destination_index is a random place where a fraction of the demand
     $X_{m_{origin}, i_{origin}, t_{origin}}$  of product  $i_{origin}$  that is currently produced on  $m_{origin}$  at period
     $t_{origin}$ , will be placed before. Currently, the product on the destination_index is
    denoted as  $i_{dest}$ .
ELSEIF // If destination machine  $m_{dest}$  is not capable of producing product  $i_{origin}$ ; repeat
the following step until there is a product  $i_{origin}$  that can be produced on the destination
machine  $m_{dest}$  is found.
    WHILE  $A_{m_{dest}, i_{origin}} = 0$ 
        SELECT randomly another product  $i_{origin}$  from the sequence  $\pi_{m_{origin}, t_{origin}}$  from
        the origin machine  $m_{origin}$  where  $i_{origin} \in [N]$ , which destination machine is capable
        of, i.e.,  $A_{m_{dest}, i_{origin}} = 1$ .
    ENDWHILE
ENDIF

```

```

// Calculate the current space,  $free\_usage(m_{dest}, t_{dest})$ , on the destination machine  $m_{dest}$  at
period  $t_{dest}$  after required setups are carried out.
 $free\_usage(m_{dest}, t_{dest}) = usage(m_{dest}, t_{dest}) - setup_{old,dest} + setup_{new,dest}$ 
// Check if there is enough capacity that fraction of lot  $X_{m_{origin}, i_{origin}, t_{origin}}$  can be inserted
to machine  $m_{dest}$  on period  $t_{dest}$ .
IF  $capacity(m_{dest}, t_{dest}) - free\_usage(m_{dest}, t_{dest}) > 0$ 
    // Check if the whole lot  $X_{m_{origin}, i_{origin}, t_{origin}}$  can be inserted to machine  $m_{dest}$  on
    period  $t_{dest}$ .
    IF  $capacity(m_{dest}, t_{dest}) - free\_usage(m_{dest}, t_{dest}) \leq X_{m_{origin}, i_{origin}, t_{origin}}$ 
        //update current capacity usages and sequences of machines
    ELSEIF // Calculate the fraction of lot that can be inserted to machine  $m_{dest}$  on
    period  $t_{dest}$ .
         $fraction_{m_{origin}, i_{origin}, t_{origin}} = capacity(m_{dest}, t_{dest}) - free\_usage(m_{dest}, t_{dest})$ 
        //update current capacity usages and sequences of machines
         $SetupUpdate(setup_{old,origin}, setup_{old,dest})$ 
         $UpdateAll(usage(m_{origin}, t_{origin}),$ 
         $usage(m_{dest}, t_{dest}), \pi_{m_{origin}, t_{origin}}, \pi_{m_{dest}, t_{dest}})$ 
    ENDIF
ENDIF
}

```

Figure 6.15 Pseudocode for the fractional insert move used in this study

Different than insert and swap moves, in fractional insert move, the lot is transferred after the capacity check is carried out. The reason is that if there is no capacity limitation, then the whole lot will be transferred and fractional insert move will behave the same as the insert move. On the other hand, in the *Local Search Phase* of VNS algorithms, only either insert or swap move can search the whole solution space entirely within their neighborhoods, i.e., they can be used in *Best Improvement Algorithm* or *Steepest Descent Heuristic*, which is defined in Section 4.2. However, fractional insert move does not work as a move for *Steepest Descent Heuristic*, i.e., not all possible fractional insert moves are considered in the tested algorithms since the transferred quantity of lot would change in every iteration, and accordingly there will be infinitely many possible moves. Thus, only a single random

move is carried out at each iteration when fractional insert move is used and fractional insert move is used only in the *Shaking Phase*. To sum up $insert(\pi)$, $swap(\pi)$ and $fractional\ insert(\pi)$ moves, proposed in Figure 6.9, 6.12 and 6.15 are only used in the *Shaking Phase*, i.e., when there is only a random single move. When whole neighborhood is explored entirely in insert and swap neighborhood schemes, then *Insert Best Improvement* (π) or *Swap Best Improvement* (π) are used which will be explained in Section 7.2.

CHAPTER SEVEN

COMPUTATIONAL STUDY

In this chapter, the algorithms and their results are presented and compared with the benchmark results found in the literature so far.

7.1 Problem Instances Tested

Benchmark problem instances used in this study were generated by James & Almada-Lobo (2011). Firstly, they started to generate mid-sized problem instance with 15 products, 10 periods and 80% capacity utilization. Afterwards, they used various perturbations based on the change in number of machines, number of products, number of periods, capacity utilization per period, cost of setup per unit of time, balance of products across the machines, and possible product-machine allocations. Problem type can be represented as follows:

M-N-T-Cut-CutVar- θ -MProb-MBal.

Where M denotes the number of machines; N denotes the number of products; T denotes the number of periods; Cut denotes the capacity utilization per period. $CutVar$ controls the maximum total allowed variation from Cut , so the actual capacity utilization can vary. However, in this thesis, benchmark instances have no capacity variation. θ indicates the cost of setup per unit of time, $MProb$ represents the total number of possible product-machine allocations, i.e., when $MProb$ increases, the problem becomes harder to solve. Lastly, $MBal$ indicates the balance of products across the machines (James & Almada-Lobo, 2011).

For each combination, 10 instances were generated. The variation of parameters which may give an idea of the difficulty level of the benchmark instances can be observed as follows:

M : number of machines from 2 \rightarrow 3

N : number of products from 15 → 20

T : number of periods from 5 → 10

Cut : capacity utilization per period 0.6 → 0.8

θ : cost of setup per unit of time 50 → 100

$MProb$: total number of possible product-machine allocations 60 → 80

$MBal$: balance of products across the machines 10 → 20

7.2 Algorithms Tested

The types of moves which were described previously in Section 6.3, are used in the VNS, VND and reduced VNS algorithms proposed.

In VNS, which is defined formerly in Section 4.1, two neighborhood structures are used, i.e., $(N_k \text{ where } k_{max} = 2)$. *Fractional insert move* is used only in the *Shaking Phase*, however, in the *Local Search Phase*, *insert move* and *swap move* are used respectively. Moreover, in the *Local Search Phase*, whole neighborhood is explored, i.e., *Best Improvement Local Search* is used, which is previously defined in Section 4.2. For the stopping condition, maximum number of 3000 iterations is chosen.

Similar to VNS, in VND algorithm, the number of neighborhood structure is again two. Furthermore, binary combinations of *swap move* and *insert move* are used. For example, in one case, first neighborhood is *swap move*, second neighborhood is *insert move*, and in the other case the first neighborhood is *insert move* and second neighborhood is *swap move*. Likewise, the whole neighborhood is searched entirely and the best neighbor of x' of x ($x' \in N_k(x)$) is found.

In reduced VNS (RVNS), pair-wise combinations of three moves are used. Two techniques of constraint handling are utilized for all of the algorithms tested. Firstly, if a solution is infeasible, then it is penalized by using *NFT* (Smith & Tate, 1993; Tate & Smith, 1995). Secondly, the *superiority of feasible solutions* is used when making a selection between two solutions during the search phase.

7.2.1 Variable Neighborhood Search (VNS)

Insert Best Improvement and *Swap Best Improvement* heuristics are explained, which are used both in VNS and VND algorithms proposed in this study. *Insert Best Improvement* and *Swap Best Improvement* heuristics are the variants of *Best Improvement Local Search* which is explained previously in Section 4.1. With the *Insert Best Improvement* heuristic, all possible insertions are carried out. The only constraint is the capability of machines. Similarly, *Swap Best Improvement* heuristic makes all possible swap moves as long as the capability constraint is satisfied. In VNS and VND algorithms, the search space within insert and swap neighborhoods is explored entirely. The general framework for the entire search of insert and swap moves, which are *Insert Best Improvement* and *Swap Best Improvement* heuristics, are explained in the following figures.

```

Insert Best Improvement ( $\pi$ )
FOR  $t = 1$  to  $t_{max}$ 
  FOR  $m_{origin} = 1$  to  $m_{max}$ 
    FOR  $i_{origin} = 1$  to  $\text{length}(\pi_{m_{origin},t})$ 
      FOR  $m_{dest} = 1$  to  $m_{max}$ 
        FOR  $i_{dest} = 1$  to  $(\text{length}(\pi_{m_{dest},t}) + 1)$ 
          IF  $A_{m_{dest},i_{origin}} = 1$  // Insert the whole lot  $X_{m_{origin},i_{origin},t_{origin}}$ 
            Insert( $\pi$ )
            SetupUpdate( $setup_{old,origin}, setup_{old,dest}$ )
            UpdateAll(  $usage(m_{origin}, t_{origin})$  ,
             $usage(m_{dest}, t_{dest}), \pi_{m_{origin},t_{origin}}, \pi_{m_{dest},t_{dest}}$  )
            //Calculate the objective function and find the best
          ENDFOR
        ENDFOR
      ENDFOR
    ENDFOR
  ENDFOR
ENDFOR
return( $\pi_{move}, obj_{move}, total\ violation\ move$ )

```

Figure 7.1 Pseudocode for Best Improvement for Insert Move

```

Swap Best Improvement ( $\pi$ )
FOR  $t = 1$  to  $t_{max}$ 
  FOR  $m_{origin} = 1$  to  $m_{max}$ 
    FOR  $i_{origin} = 1$  to  $\text{length}(\pi_{m_{origin},t})$ 
      FOR  $m_{dest} = 1$  to  $m_{max}$ 
        FOR  $i_{dest} = 1$  to  $\text{length}(\pi_{m_{dest},t})$ 
          IF  $A_{m_{dest},i_{origin}} = 1 \ \&\& \ A_{m_{origin},i_{dest}} = 1$ 
            // Swap lots  $X_{m_{origin},i_{origin},t_{origin}}$  and  $X_{m_{dest},i_{dest},t_{dest}}$ 
            Swap( $\pi$ )
            SetupUpdate( $\text{setup}_{old,origin}, \text{setup}_{old,dest}$ )
            UpdateAll( $\text{usage}(m_{origin}, t_{origin})$  ,
               $\text{usage}(m_{dest}, t_{dest}), \pi_{m_{origin},t_{origin}}, \pi_{m_{dest},t_{dest}}$ )
          ENDIF
        ENDFOR
      ENDFOR
    ENDFOR
  ENDFOR
ENDFOR
return( $\pi_{move}, \text{obj}_{move}, \text{total violation move}$ )

```

Figure 7.2 Pseudocode for Best Improvement for Swap Move

In the *comparison* function defined in Figure 7.3, two solutions are compared using *Superiority of Feasible Solutions* and *NFT*. According to the *Superiority of Feasible Solutions*, when both solutions are feasible, then the solution with the minimum objective value is updated as the new incumbent solution and algorithm continues to explore within the same neighborhood (Deb, 2000). On the other hand, when the incumbent solution is feasible and the new solution is infeasible, then, a move to the next neighborhood is carried out. If the incumbent solution is infeasible and the new solution is feasible, then, the new solution will be as the incumbent solution and since an improvement is made, algorithm continues with the same neighborhood. Lastly, if both solutions are infeasible, then the solution with the least violation amount is chosen as the new incumbent solution.


```

comparison( $\pi_{best}$ , total violation bsf, best so far,  $\pi_{move}$ , total violation move, objmove)
improvement = 1
  IF total violation bsf == 0 && total violation move == 0
    IF best so far < objmove
      improvement = 0
    ELSE
      // update best so far and  $\pi_{best}$ 
      best so far  $\leftarrow$  objmove
       $\pi_{best}$   $\leftarrow$   $\pi_{move}$ 
    END
  ELSEIF total violation bsf == 0 && total violation move > 0
    improvement = 0
  ELSEIF total violation bsf > 0 && total violation move == 0
    // update best so far,  $\pi_{best}$ , and total violation bsf
    best so far  $\leftarrow$  objmove,  $\pi_{best}$   $\leftarrow$   $\pi_{move}$ 
    total violation bsf  $\leftarrow$  total violation move
  ELSEIF total violation bsf > 0 && total violation move > 0
    //Calculate NFT for both solutions
    
$$NFT = \sum_{i=1}^m \left( \frac{d_i}{NFT_i} \right)^K$$

    IF NFT bsf > NFT move
      // update best so far,  $\pi_{best}$ , and total violation bsf
      best so far  $\leftarrow$  objmove,  $\pi_{best}$   $\leftarrow$   $\pi_{move}$ 
      total violation bsf  $\leftarrow$  total violation move
    ELSEIF
      improvement = 0
    ENDIF
  ENDIF
  return(improvement)

```

Figure 7.3 Pseudocode for comparison function

In the *comparison* function defined in Figure 7.3, two solutions are compared using *Superiority of Feasible Solutions* and *NFT*. According to the *Superiority of Feasible Solutions*, when both solutions are feasible, then the solution with the minimum objective value is updated as the new incumbent solution and algorithm continues to explore within the same neighborhood (Deb, 2000). On the other hand, when the incumbent solution is feasible and the new solution is infeasible, then, a

move to the next neighborhood is carried out. If the incumbent solution is infeasible and the new solution is feasible, then, the new solution will be as the incumbent solution and since an improvement is made, algorithm continues with the same neighborhood. Lastly, if both solutions are infeasible, then the solution with the least violation amount is chosen as the new incumbent solution.

```

VNS( $\pi_{best}$ ,  $k_{max} = 2$ ,  $CPU_{max} = 3600$ )
CPU = 0,
 $\pi_{best} \leftarrow \pi_{initial}$ ,  $\pi \leftarrow \pi_{initial}$ ,
best so far  $\leftarrow obj_{initial}$ ,
total violation bsf  $\leftarrow violation_{initial}$ .
WHILE CPU  $\leq$  CPUmax
    k = 1
    // Shaking Phase,
     $\pi_{fractional\ insert}$ ,  $obj_{fractional\ insert}$ , total violation fractional insert  $\leftarrow$ 
fractional insert move( $\pi_{best}$ )
     $\pi_{best} \leftarrow \pi_{fractional\ insert}$ 
    WHILE k  $\leq$  kmax
        // Local Search Phase,
        IF (k = 1)
             $\pi_{move}$ ,  $obj_{move}$ , total violation move  $\leftarrow$  argmin(Insert Best Improvement( $\pi$  ))
        ELSEIF
             $\pi_{move}$ ,  $obj_{move}$ , total violation move  $\leftarrow$  argmin(Swap Best Improvement( $\pi$  ))
        ENDIF
        //compare  $\pi_{best}$  and  $\pi_{move}$ , use Superiority of Feasible Solutions and NFT
        improvement  $\leftarrow$  comparison( $\pi_{best}$ , total violation bsf, best so far,  $\pi_{move}$ , total violation move,  $obj_{move}$ )
        IF improvement = 0
            k = k + 1
        ELSEIF
            k = 1
        ENDIF
    ENDWHILE
ENDWHILE

```

Figure 7.4 Pseudocode for VNS Algorithm used in this study (kmax=2, N1=insert move ,N2=swap move)

On the other hand, in Figure 7.4, the pseudocode of the VNS algorithm used in this study is presented where. $N_1 = \text{insert move}$ and $N_2 = \text{swap move}$.

In VNS algorithm, there are both *Shaking Phase* and *Local Search Phases*. VNS algorithm used in this study works as follows: Firstly, initial parameters $\pi_{best} = \pi_{initial}$, $best\ so\ far = obj_{initial}$, and $total\ violation\ bsf = violation_{initial}$ are found from the initial solution. Afterwards, a this fractional insert move is made for diversification. After a single fractional insert move, local search phase starts where an entire search is made within the 1st neighborhood, $N_1 = \text{insert move}$, using *Insert Best Improvement* method, which is described in Section 4.2. Local minimum, denoted as obj_{insert} , among all solutions of the *Insert Best Improvement* is found and compared with the *best so far* value using *comparison* function described in Figure 7.3.. If there is improvement, then, *best so far*, π_{best} and *total violation bsf* are updated and the neighborhood turns to be the first neighborhood, i.e., $k = 1$. When there is no improvement, then a move to the second neighborhood is made. When there is no improvement on the second neighborhood, the algorithm stops, takes π_{best} , and continues with the next iteration. More specifically, the proposed VNS algorithm continues taking π_{best} as the new incumbent solution until a predetermined maximum CPU time is reached.

Other VNS algorithms used in this study has the same *Shaking Phase*. However, in the *Local Search Phase*, swap and insert moves are used, which are presented on Figure 7.1 and 7.2, respectively, i.e., either $N_1 = \text{insert move}$ $N_2 = \text{swap move}$. or the reverse. The reason why fractional insert move is not used in the *Local Search Phase* is that, the trasferred quantity of lot would change in each step, and accordingly there will be infinite possible moves when the entire search space would be explored. Thus, fractional insert move is used only in the *Shaking Phase* of the proposed VNS algorithms.

7.2.2 Variable Neighborhood Descent (VND)

Pseudocode of VND algorithm used in this study can be observed in Figure 7.5 in detail.

```

VND ( $\pi_{best}$ ,  $k_{max} = 2$ )
 $\pi_{best} \leftarrow \pi_{initial}$ ,  $best\ so\ far \leftarrow obj_{initial}$ ,  $total\ violation\ bsf \leftarrow violation_{initial}$ .  $k = 1$ 
WHILE  $k \leq k_{max}$ 
    IF ( $k = 1$ )
         $\pi_{move}$ ,  $obj_{move}$ ,  $total\ violation\ move \leftarrow argmin(Swap\ Best\ Improvement(\pi_{best}))$ 
    ELSEIF
         $\pi_{move}$ ,  $obj_{move}$ ,  $total\ violation\ move \leftarrow argmin(Insert\ Best\ Improvement(\pi_{best}))$ 
    ENDIF
 $improvement \leftarrow comparison(\pi_{best}, total\ violation\ bsf, best\ so\ far, \pi_{move}, total\ violation\ move, obj_{move})$ 
    IF  $improvement = 0$ 
         $k = k + 1$ 
    ELSEIF
         $k = 1$ 
    ENDIF
ENDWHILE

```

Figure 7.5 Pseudocode for VND Algorithm used in this study ($k_{max}=2$, $N_1=swap$ move, $N_2=insert$ move)

The main difference of VND and VNS is that, VND does not have a *Shaking Phase*.

VND works as follows: Firstly, initial best parameters; $\pi_{best} = \pi_{initial}$, $best\ so\ far = obj_{initial}$, and $total\ violation\ bsf = violation_{initial}$. are found from the initial solution.

Then, an entire search is made within the first neighborhood, $N_1 = swap\ move$, using *Swap Best Improvement* method, proposed in Figure 7.2. The minimum objective function value, denoted as obj_{swap} , among all solutions of the *Swap Best Improvement* is found and compared with the *best so far* value using *comparison*

function described in Figure 7.3. If there is improvement, then, *best so far*, π_{best} and *total violation bsf* are updated. Afterwards, *Swap Best Improvement* continues with the new incumbent solution π_{best} . If no improvement occurs, then, algorithm continues with the next neighborhood $N_2 = \textit{insert move}$. *Insert Best Improvement* takes the incumbent solution π_{best} as the input and explores the entire search region. The minimum in this neighborhood is found and compared with the *best so far* value again with the *comparison* function. Likewise, if, there is an improvement, *best so far*, π_{best} and *total violation bsf* are updated and algorithm returns to the first neighborhood $N_1 = \textit{swap move}$ again. If no improvement is achieved on the second neighborhood also, then, the algorithm stops.

In this study, two different VND algorithms are tested. In the first one $N_1 = \textit{swap move}$, $N_2 = \textit{insert move}$, and in the second one is $N_1 = \textit{insert move}$, $N_2 = \textit{swap move}$. The only difference between the two VND variants is the sequence of the neighborhoods.

7.2.3 Reduced Variable Neighborhood Search (RVNS)

The pseudocode of RVNS heuristic used in this study can be seen in Figure 7.6. Different from VNS, RVNS consists of *Shaking Phase* only.

As it can be seen from Figure 7.6, starting with the initial solution, let $\pi_{best} = \pi_{initial}$, *best so far* = $obj_{initial}$, and *total violation bsf* = $violation_{initial}$. be the best so far values.

Starting from the first neighborhood, a fractional insert move is made, and the results are compared with the incumbent solution using *comparison* function described in Figure 7.3. When there is an improvement, then the search continues within the first neighborhood scheme, otherwise, a move to the next neighborhood is done. Furthermore, the algorithm starts from the first neighborhood and repeats all these steps, taking the best so far solution, π_{best} obtained from the last iteration as the input, until the maximum number of iterations is achieved.

```

RVNS ( $\pi_{best}$ ,  $k_{max} = 2$ ,  $iteration_{max} = 3000$ )
 $\pi_{best} = \pi_{initial}$ , best so far =  $obj_{initial}$ ,  $iteration = 1$ ,
total violation bsf =  $violation_{initial}$ .
WHILE  $iteration \leq iteration_{max}$ 
 $k = 1$ 
// Shaking
    WHILE  $k \leq k_{max}$ 
        IF ( $k = 1$ )
             $\pi_{move}$ ,  $obj_{move}$ , total violation move  $\leftarrow$  fractional insert move( $\pi_{best}$ )
        ELSE
             $\pi_{move}$ ,  $obj_{move}$ , total violation move  $\leftarrow$  swap move( $\pi_{best}$ )
        ENDIF
        //compare  $\pi_{best}$  and  $\pi_{move}$ , use Superiority of Feasible Solutions and NFT
        improvement  $\leftarrow$  comparison( $\pi_{best}$ , total violation bsf, best so far,  $\pi_{move}$ , total violation move,  $obj_{move}$ )
        IF improvement = 0
             $k = k + 1$ 
        ELSE
             $k = 1$ 
        ENDIF
    ENDWHILE
iteration  $\leftarrow$  iteration + 1
ENDWHILE

```

Figure 7.6 Pseudocode for RVNS Algorithm used in this study ($k_{max}=2$, N_1 =fractional insert move, N_2 =swap move)

In this study, six different RVNS algorithms are used. The reason is that, there are three types of moves defined in Section 6.3, pair-wise combinations of three moves are used, i.e., $k_{max} = 2$. Figure 7.6 represents only one RVNS with neighborhoods $N_1 =$ *fractional insert move*, and $N_2 =$ *swap move*.

7.3 Computational Study

All benchmark instances are tested on the proposed algorithms. The average % deviations from the lower bounds and average computation times for each 10 different instance type are calculated and compared with the results of James & Almada-Lobo (2011). The NFT parameters used in this study are, NFT_0 : 0.001 and λ : 0.4. All computational experiments were performed on a Intel(R) Core(TM) i-5 2430M CPU: 2.40 GHz with 4GB RAM and algorithms were coded in MATLAB R2010A.

For each 10 different instance combination, the average % deviation from lower bound is calculated with the following formula,

$$\left[\sum_{i=1}^{10} \frac{(best\ so\ far - Lower\ Bound) \times 100}{Lower\ Bound} \right] / 10 \quad (7.2)$$

7.3.1 Experimental Results for Variable Neighborhood Search (VNS)

In the proposed VNS algorithms mentioned before, fractional insert move is used only in the Shaking Phase. Insert and swap neighborhood schemes are used in the Local Search Phase. Also, both VNS algorithms are set to run to 3600 seconds time limit so as to use the same time limit with the compared benchmark instances.

As it can be seen in Table 7.1, both VNS algorithms performs well on various problems. Also, when the differences of average deviations between each proposed VNS algorithm and current best known solutions are compared, it can be deduced that the performance of both VNS algorithms are good. A negative value on Table 7.2 and 7.3 indicate an improved dataset. Another deduction is that when total number of possible product-machine allocations, $MProb$, is at low level, then the neighborhood combination of N_1 : insert move N_2 : swap has a better solution quality compared to the other neighborhood pair.

Table 7.1 Average % deviations from lower bound of the proposed VNS algorithm (Compared with the benchmark results)

Problem Type	Average % deviations from lower bound				
	VNS N_1 : insert move, N_2 : swap move	VNS N_1 : swap move, N_2 : insert move	XPHRF	INSRF	FOHRF9
Data2-15-5-0.8-0.0-50-80-20	1.36	1.52	1.49	1.47	2.03
Data2-15-10-0.8-0.0-50-80-20	2.55	2.59	2.60	2.92	3.06
Data2-15-10-0.8-0.0-100-60-20	8.11	7.08	7.10	8.24	7.84
Data2-15-10-0.8-0.0-100-80-10	6.27	4.94	5.90	5.06	5.19
Data2-15-10-0.8-0.0-100-80-20	6.17	6.59	6.02	6.01	7.13
Data2-20-10-0.8-0.0-100-80-20	7.60	6.86	6.98	8.03	7.68
Data3-15-5-0.8-0.0-50-80-20	9.33	6.74	10.78	6.93	6.63
Data3-15-10-0.6-0.0-100-60-20	4.51	4.28	4.37	5.13	4.64
Data3-15-10-0.8-0.0-50-80-20	13.02	12.97	12.86	14.29	14.31
Data3-15-10-0.8-0.0-100-60-20	8.02	9.40	8.00	9.31	9.78

Table 7.2 Differencens of average % deviations from lower bound of the proposed VNS algorithms

Problem Type	Difference of Average % deviations from lower bound	
	VNS <i>N</i> ₁ : insert move. <i>N</i> ₂ : swap move and Best So Far	VNS <i>N</i> ₁ : swap move. <i>N</i> ₂ : insert move and Best So Far
Data2-15-5-0.8- 0.0-50-80-20	-0.11	0.05
Data2-15-10-0.8- 0.0-50-80-20	-0.05	-0.01
Data2-15-10-0.8- 0.0-100-60-20	1.01	-0.02
Data2-15-10-0.8- 0.0-100-80-10	1.21	-0.12
Data2-15-10-0.8- 0.0-100-80-20	0.16	0.58
Data2-20-10-0.8- 0.0-100-80-20	0.62	-0.12
Data3-15-5-0.8- 0.0-50-80-20	2.70	0.11
Data3-15-10-0.6- 0.0-100-60-20	0.14	-0.09
Data3-15-10-0.8- 0.0-50-80-20	0.16	0.11
Data3-15-10-0.8- 0.0-100-60-20	0.02	1.40

7.3.2 Experimental Results for Variable Neighborhood Descent (VND)

Following tables indicate the average % deviations from lower bound and average CPU time in seconds for the two VND algorithms tested on the benchmark instances.

According to the Table 7.3, it can be seen that various instance types were improved. The differences among percentage deviations from lower bound can be seen in Table 7.4. On the other hand, Table 7.5 indicates that the computation time of

both VND algorithms are approximately 18 minutes, which implies that VND algorithm works somehow efficient for small to mid-size problems.

Table 7.3 Average % deviations from lower bound of the proposed VND algorithm (Compared with the benchmark results)

Problem Type	Average % deviations from lower bound				
	VND <i>N₁: insert move, N₂: swap move</i>	VND <i>N₁: swap move, N₂: insert move</i>	XPHRF	INSRF	FOHRF9
Data2-15-5-0.8-0.0-50-80-20	2.01	1.71	1.49	1.47	2.03
Data2-15-10-0.8-0.0-50-80-20	2.71	2.52	2.60	2.92	3.06
Data2-15-10-0.8-0.0-100-60-20	6.96	7.07	7.10	8.24	7.84
Data2-15-10-0.8-0.0-100-80-10	5.12	5.09	5.90	5.06	5.19
Data2-15-10-0.8-0.0-100-80-20	6.11	6.38	6.02	6.01	7.13
Data2-20-10-0.8-0.0-100-80-20	6.74	6.88	6.98	8.03	7.68
Data3-15-5-0.8-0.0-50-80-20	7.58	7.49	10.78	6.93	6.63
Data3-15-10-0.6-0.0-100-60-20	5.21	5.17	4.37	5.13	4.64
Data3-15-10-0.8-0.0-50-80-20	14.38	14.42	12.86	14.29	14.31
Data3-15-10-0.8-0.0-100-60-20	9.14	9.26	8.00	9.31	9.78

Table 7.4 Differencens of average % deviations from lower bound of the proposed VND algorithms

Problem Type	Difference of Average % deviations from lower bound	
	VND N_1 : <i>insert move</i> . N_2 : <i>swap move</i> and Best So Far	VND N_1 : <i>swap move</i> . N_2 : <i>insert move</i> and Best So Far
Data2-15-5-0.8- 0.0-50-80-20	0.54	0.24
Data2-15-10- 0.8-0.0-50-80-20	0.11	-0.08
Data2-15-10- 0.8-0.0-100-60- 20	-0.14	-0.03
Data2-15-10- 0.8-0.0-100-80- 10	0.06	0.03
Data2-15-10- 0.8-0.0-100-80- 20	0.10	0.37
Data2-20-10- 0.8-0.0-100-80- 20	-0.24	-0.10
Data3-15-5-0.8- 0.0-50-80-20	0.95	0.86
Data3-15-10- 0.6-0.0-100-60- 20	0.84	0.80
Data3-15-10- 0.8-0.0-50-80-20	1.52	1.56
Data3-15-10- 0.8-0.0-100-60- 20	1.14	1.26

Table 7.5 Average Computation Times of the proposed VND algorithm (Compared with the benchmark results)

Problem Type	Average time in seconds				
	VND N_1 : insert move, N_2 : swap move	VND N_1 : swap move, N_2 : insert move	XPHRF	INSRF	FOHRF9
Data2-15-5-0.8-0.0-50-80-20	634.8	594.4	323.7	969.2	74.1
Data2-15-10-0.8-0.0-50-80-20	727.4	779.2	527.1	1173.7	244.4
Data2-15-10-0.8-0.0-100-60-20	708.2	726.1	1371.2	851.1	559.8
Data2-15-10-0.8-0.0-100-80-10	1294.4	1343.8	3756.1	1909.3	2524.1
Data2-15-10-0.8-0.0-100-80-20	945.2	1053.0	322.2	1170.0	198.1
Data2-20-10-0.8-0.0-100-80-20	1174.2	1328.9	1254.5	930.4	626.7
Data3-15-5-0.8-0.0-50-80-20	1584.3	1272.1	7623.0	1427.5	941.3
Data3-15-10-0.6-0.0-100-60-20	1149.2	1076.5	3023.8	1266.9	417.0
Data3-15-10-0.8-0.0-50-80-20	1545.7	1293.5	2910.9	1220.9	950.9
Data3-15-10-0.8-0.0-100-60-20	1617.9	1337.8	960.3	1035.0	239.1
Average	1138.13	1080.53	2207.3	1195.4	677.5

7.3.3 Experimental Results for Reduced Variable Neighborhood Search (RVNS)

Since there are ten different combinations of the benchmark instances, randomly a single benchmark instance is tested on each RVNS algorithm in order to see when *best so far* value converges. The figures in the appendix are the convergence plots of selected four instances having different characteristics. Six different RVNS algorithms were tested. The convergence points of randomly selected ten instances from different combinations are summarized in tables 7.6-11, where each table

indicates the iteration numbers when convergence started over the tested RVNS algorithm.

Table 7.6 Randomly Selected Instances and their iteration numbers where convergence starts tested algorithm: RVNS (N1: insert, N2:swap)

Randomly Selected Instance RVNS (N ₁ : insert, N ₂ :swap)	Iteration Number where best so far value started to converge (after 3000 iterations)
Data2-15-5-0.8-0.0-50-80-20-1	2567
Data2-15-10-0.8-0.0-50-80-20-3	2572
Data2-15-10-0.8-0.0-100-60-20-4	2603
Data2-15-10-0.8-0.0-100-80-10-8	2311
Data2-15-10-0.8-0.0-100-80-20-6	2652
Data2-20-10-0.8-0.0-100-80-20-8	2189
Data3-15-5-0.8-0.0-50-80-20-7	2783
Data3-15-10-0.6-0.0-100-60-20-1	2451
Data3-15-10-0.8-0.0-50-80-20-5	2664
Data3-15-10-0.8-0.0-100-60-20-3	2660

Table 7.7 Randomly Selected Instances and their iteration numbers where convergence starts Tested Algorithm: RVNS (N1: swap, N2:insert)

Randomly Selected Instance RVNS (N ₁ : swap, N ₂ :insert)	Iteration Number where best so far value started to converge (after 3000 iterations)
Data2-15-5-0.8-0.0-50-80-20-1	2711
Data2-15-10-0.8-0.0-50-80-20-3	2579
Data2-15-10-0.8-0.0-100-60-20-4	2703
Data2-15-10-0.8-0.0-100-80-10-8	2643
Data2-15-10-0.8-0.0-100-80-20-6	2589
Data2-20-10-0.8-0.0-100-80-20-8	2244
Data3-15-5-0.8-0.0-50-80-20-7	2688
Data3-15-10-0.6-0.0-100-60-20-1	2430
Data3-15-10-0.8-0.0-50-80-20-5	2219
Data3-15-10-0.8-0.0-100-60-20-3	2827

Table 7.8 Randomly Selected Instances and their iteration numbers where convergence starts Tested Algorithm: RVNS (N1: fractional insert, N2:insert)

Randomly Selected Instance RVNS (N ₁ :fractional insert, N ₂ :insert)	Iteration Number where best so far value started to converge (after 3000 iterations)
Data2-15-5-0.8-0.0-50-80-20-1	2696
Data2-15-10-0.8-0.0-50-80-20-3	2727
Data2-15-10-0.8-0.0-100-60-20-4	2203
Data2-15-10-0.8-0.0-100-80-10-8	2651
Data2-15-10-0.8-0.0-100-80-20-6	2538
Data2-20-10-0.8-0.0-100-80-20-8	2478
Data3-15-5-0.8-0.0-50-80-20-7	2430
Data3-15-10-0.6-0.0-100-60-20-1	2666
Data3-15-10-0.8-0.0-50-80-20-5	2649
Data3-15-10-0.8-0.0-100-60-20-3	2488

Table 7.9 Randomly Selected Instances and their iteration numbers where convergence starts Tested Algorithm: RVNS (N1: fractional insert, N2:swap)

Randomly Selected Instance RVNS (N ₁ :fractional insert, N ₂ :swap)	Iteration Number where best so far value started to converge (after 3000 iterations)
Data2-15-5-0.8-0.0-50-80-20-1	2508
Data2-15-10-0.8-0.0-50-80-20-3	2733
Data2-15-10-0.8-0.0-100-60-20-4	2234
Data2-15-10-0.8-0.0-100-80-10-8	2251
Data2-15-10-0.8-0.0-100-80-20-6	2586
Data2-20-10-0.8-0.0-100-80-20-8	2392
Data3-15-5-0.8-0.0-50-80-20-7	2794
Data3-15-10-0.6-0.0-100-60-20-1	2613
Data3-15-10-0.8-0.0-50-80-20-5	2648
Data3-15-10-0.8-0.0-100-60-20-3	2861

Table 7.10 Randomly Selected Instances and their iteration numbers where convergence starts Tested Algorithm: RVNS (N1: insert, N2: fractional insert)

Randomly Selected Instance RVNS (N ₁ : insert, N ₂ : fractional insert)	Iteration Number where best so far value started to converge (after 3000 iterations)
Data2-15-5-0.8-0.0-50-80-20-1	2869
Data2-15-10-0.8-0.0-50-80-20-3	2723
Data2-15-10-0.8-0.0-100-60-20-4	2645
Data2-15-10-0.8-0.0-100-80-10-8	2633
Data2-15-10-0.8-0.0-100-80-20-6	2714
Data2-20-10-0.8-0.0-100-80-20-8	2451
Data3-15-5-0.8-0.0-50-80-20-7	2807
Data3-15-10-0.6-0.0-100-60-20-1	2711
Data3-15-10-0.8-0.0-50-80-20-5	2648
Data3-15-10-0.8-0.0-100-60-20-3	2607

Table 7.11 Randomly Selected Instances and their iteration numbers where convergence starts Tested Algorithm: RVNS (N1: swap, N2: fractional insert)

Randomly Selected Instance RVNS (N ₁ : swap, N ₂ : fractional insert)	Iteration Number where best so far value started to converge (after 3000 iterations)
Data2-15-5-0.8-0.0-50-80-20-1	2631
Data2-15-10-0.8-0.0-50-80-20-3	2538
Data2-15-10-0.8-0.0-100-60-20-4	2340
Data2-15-10-0.8-0.0-100-80-10-8	2407
Data2-15-10-0.8-0.0-100-80-20-6	2594
Data2-20-10-0.8-0.0-100-80-20-8	2373
Data3-15-5-0.8-0.0-50-80-20-7	2299
Data3-15-10-0.6-0.0-100-60-20-1	2643
Data3-15-10-0.8-0.0-50-80-20-5	2254
Data3-15-10-0.8-0.0-100-60-20-3	2386

As it can be deduced from the convergence plots, which can be seen in Appendices, and convergence tables (Table 7.6-11), the solution converges approximately after 2500 iterations. In fact, in some cases, the convergence point is closer to 2900 iterations, however, for those instances, only slight changes in the best

so far value is detected after 2500 iterations. It can also be understood from the plots, fractional insert move results in more fluctuations on the solution.

By looking at the convergence plots, it can also be claimed that, starting with a good initial solution will have no major effect on the convergence point. In other words, even a good initial solution is used, the proposed RVNS algorithms converge nearly on the same iteration number and best so far values are close to each other. It can be concluded that the effect of a good initial solution might be not so clearly seen here, due to the excessively high computational time, i.e., approximately 8 hours for each instance. Only effect of starting with a good initial solution that can be seen from the plots is that, in some of the above convergence plots, there is a jump within the first 20-25 iterations. The reason might possibly be that the initial solution is infeasible. However, when feasible solutions were achieved, the best so far value would decline due to the *Superiority of Feasible Solutions* (Deb, 2000). The following tables show the average % deviations from the lower bound compared to the best so far solutions in the literature, when six RVNS algorithms were tested.

Table 7.12 Average % deviations of RVNS (N1: insert, N2:swap)

Problem Type	Average % deviations from lower bound			
	RVNS N_1 : insert move, N_2 : swap move	XPHRF	INSRF	FOHRF9
Data2-15-5-0.8-0.0-50-80-20	1.62	1.49	1.47	2.03
Data2-15-10-0.8-0.0-50-80-20	2.38	2.60	2.92	3.06
Data2-15-10-0.8-0.0-100-60-20	7.26	7.10	8.24	7.84
Data2-15-10-0.8-0.0-100-80-10	5.14	5.90	5.06	5.19
Data2-15-10-0.8-0.0-100-80-20	7.18	6.02	6.01	7.13
Data2-20-10-0.8-0.0-100-80-20	6.89	6.98	8.03	7.68
Data3-15-5-0.8-0.0-50-80-20	6.59	10.78	6.93	6.63
Data3-15-10-0.6-0.0-100-60-20	4.33	4.37	5.13	4.64
Data3-15-10-0.8-0.0-50-80-20	12.81	12.86	14.29	14.31
Data3-15-10-0.8-0.0-100-60-20	10.14	8.00	9.31	9.78

Table 7.13 Average % deviations of RVNS (N1: swap, N2:insert)

Problem Type	Average % deviations from lower bound			
	RVNS <i>N₁: swap move, N₂: insert move</i>	XPHRF	INSRF	FOHRF9
Data2-15-5-0.8-0.0-50-80-20	1.66	1.49	1.47	2.03
Data2-15-10-0.8-0.0-50-80-20	2.63	2.60	2.92	3.06
Data2-15-10-0.8-0.0-100-60-20	6.93	7.10	8.24	7.84
Data2-15-10-0.8-0.0-100-80-10	6.15	5.90	5.06	5.19
Data2-15-10-0.8-0.0-100-80-20	6.33	6.02	6.01	7.13
Data2-20-10-0.8-0.0-100-80-20	6.50	6.98	8.03	7.68
Data3-15-5-0.8-0.0-50-80-20	8.94	10.78	6.93	6.63
Data3-15-10-0.6-0.0-100-60-20	5.44	4.37	5.13	4.64
Data3-15-10-0.8-0.0-50-80-20	12.91	12.86	14.29	14.31
Data3-15-10-0.8-0.0-100-60-20	11.75	8.00	9.31	9.78

Table 7.14 Average % deviations of RVNS (N1: insert, N2:fractional insert)

Problem Type	Average % deviations from lower bound			
	RVNS <i>N₁: insert move, N₂: fractional insert move</i>	XPHRF	INSRF	FOHRF9
Data2-15-5-0.8-0.0-50-80-20	1.57	1.49	1.47	2.03
Data2-15-10-0.8-0.0-50-80-20	2.64	2.60	2.92	3.06
Data2-15-10-0.8-0.0-100-60-20	6.92	7.10	8.24	7.84
Data2-15-10-0.8-0.0-100-80-10	5.10	5.90	5.06	5.19
Data2-15-10-0.8-0.0-100-80-20	5.88	6.02	6.01	7.13
Data2-20-10-0.8-0.0-100-80-20	6.90	6.98	8.03	7.68
Data3-15-5-0.8-0.0-50-80-20	8.42	10.78	6.93	6.63
Data3-15-10-0.6-0.0-100-60-20	9.69	4.37	5.13	4.64
Data3-15-10-0.8-0.0-50-80-20	17.24	12.86	14.29	14.31
Data3-15-10-0.8-0.0-100-60-20	11.49	8.00	9.31	9.78

Table 7.15 Average % deviations of RVNS (N1: swap, N2:fractional insert)

Problem Type	Average % deviations from lower bound			
	RVNS <i>N₁: swap move, N₂: fractional insert move</i>	XPHRF	INSRF	FOHRF9
Data2-15-5-0.8-0.0-50-80-20	1.51	1.49	1.47	2.03
Data2-15-10-0.8-0.0-50-80-20	2.45	2.60	2.92	3.06
Data2-15-10-0.8-0.0-100-60-20	7.08	7.10	8.24	7.84
Data2-15-10-0.8-0.0-100-80-10	5.03	5.90	5.06	5.19
Data2-15-10-0.8-0.0-100-80-20	6.34	6.02	6.01	7.13
Data2-20-10-0.8-0.0-100-80-20	6.91	6.98	8.03	7.68
Data3-15-5-0.8-0.0-50-80-20	7.23	10.78	6.93	6.63
Data3-15-10-0.6-0.0-100-60-20	4.36	4.37	5.13	4.64
Data3-15-10-0.8-0.0-50-80-20	13.14	12.86	14.29	14.31
Data3-15-10-0.8-0.0-100-60-20	9.20	8.00	9.31	9.78

Table 7.16 Average % deviations of RVNS (N1: fractional insert, N2:swap)

Problem Type	Average % deviations from lower bound			
	RVNS <i>N₁: fractional insert move, N₂: swap move</i>	XPHRF	INSRF	FOHRF9
Data2-15-5-0.8-0.0-50-80-20	1.43	1.49	1.47	2.03
Data2-15-10-0.8-0.0-50-80-20	2.51	2.60	2.92	3.06
Data2-15-10-0.8-0.0-100-60-20	7.12	7.10	8.24	7.84
Data2-15-10-0.8-0.0-100-80-10	5.08	5.90	5.06	5.19
Data2-15-10-0.8-0.0-100-80-20	5.86	6.02	6.01	7.13
Data2-20-10-0.8-0.0-100-80-20	6.84	6.98	8.03	7.68
Data3-15-5-0.8-0.0-50-80-20	8.26	10.78	6.93	6.63
Data3-15-10-0.6-0.0-100-60-20	4.81	4.37	5.13	4.64
Data3-15-10-0.8-0.0-50-80-20	13.40	12.86	14.29	14.31
Data3-15-10-0.8-0.0-100-60-20	7.96	8.00	9.31	9.78

Table 7.17 Average % deviations of RVNS (N1: fractional insert, N2:insert)

Problem Type	Average % deviations from lower bound			
	RVNS <i>N₁: fractional insert move, N₂: insert move</i>	XPHRF	INSRF	FOHRF9
Data2-15-5-0.8-0.0-50-80-20	1.45	1.49	1.47	2.03
Data2-15-10-0.8-0.0-50-80-20	2.56	2.60	2.92	3.06
Data2-15-10-0.8-0.0-100-60-20	7.04	7.10	8.24	7.84
Data2-15-10-0.8-0.0-100-80-10	5.12	5.90	5.06	5.19
Data2-15-10-0.8-0.0-100-80-20	6.01	6.02	6.01	7.13
Data2-20-10-0.8-0.0-100-80-20	6.85	6.98	8.03	7.68
Data3-15-5-0.8-0.0-50-80-20	7.14	10.78	6.93	6.63
Data3-15-10-0.6-0.0-100-60-20	5.07	4.37	5.13	4.64
Data3-15-10-0.8-0.0-50-80-20	12.91	12.86	14.29	14.31
Data3-15-10-0.8-0.0-100-60-20	8.14	8.00	9.31	9.78

From Table 7.12-17, all instances were improved in one of the tested RVNS algorithms. All results of the tested RVNS algorithms can be seen as a summary in Table 7.18.

It can be claimed that, neighborhoods of N_1 : insert move, N_2 : swap move results well on 3 machine case. Also, neighborhoods of N_1 : fractional insert move, N_2 : insert move show little improvement. However, the gap of the average % deviations between the lower bounds and best so far values of RVNS is still large. One reason might be due to the complexity of the problem. In more detail, in single machine benchmark results of James & Almada-Lobo (2011), the average % deviations are not greater than 3%, however, when number of machines increases, solving the problem is getting much more difficult. Another reason might be the tightness of the lower bound. James & Almada-Lobo (2011), stated that, their proposed lower bound procedure might not appropriate for the multi machine problem instances.

Table 7.18 Summary of average % deviations of all tested RVNS Algorithms

Problem Type	Average % deviations from lower bound					
	RVNS <i>N₁: insert move N₂: swap move</i>	RVNS <i>N₁: swap move, N₂: insert move</i>	RVNS <i>N₁: insert move N₂: fractional insert move</i>	RVNS <i>N₁: swap move, N₂: fractional insert move</i>	RVNS <i>N₁: fractional insert move, N₂: swap move</i>	RVNS <i>N₁: fractional insert move N₂: insert move</i>
Dataset1	1.62	1.66	1.57	1.51	1.43	1.45
Dataset2	2.38	2.63	2.64	2.45	2.51	2.56
Dataset3	7.26	6.93	6.92	7.08	7.12	7.04
Dataset4	5.14	6.15	5.10	5.03	5.08	5.12
Dataset5	7.18	6.33	5.88	6.34	5.86	6.01
Dataset6	6.89	6.50	6.90	6.91	6.84	6.85
Dataset7	6.59	8.94	8.42	7.23	8.26	7.14
Dataset8	4.33	5.44	9.69	4.36	4.81	5.07
Dataset9	12.81	12.91	17.24	13.14	13.40	12.91
Dataset 10	10.14	11.75	11.49	9.20	7.96	8.14

Even though all benchmark instances were improved with one of the RVNS methods presented, the computational time is almost 8 hours for 3000 iterations. However, the differences of average deviations from best so far values, which can be seen in Table 7.19, are satisfactory.

Table 7.19 Summary of differences of average % deviations of all tested RVNS Algorithms from best so far lower bounds

Differences of average % deviations from lower bound (Best So Far values)						
Problem Type	RVNS <i>N₁: insert move N₂: swap move</i>	RVNS <i>N₁: swap move N₂: insert move</i>	RVNS <i>N₁: insert move, N₂: fractional insert move</i>	RVNS <i>N₁: swap move N₂: fractional insert move</i>	RVNS <i>N₁: fractional insert move N₂: swap move</i>	RVNS <i>N₁: fractional insert move N₂: insert move</i>
Dataset1	0.15	0.19	0.10	0.04	-0.04	-0.02
Dataset2	-0.22	0.03	0.04	-0.15	-0.09	-0.04
Dataset3	0.16	-0.17	-0.18	-0.02	0.02	-0.06
Dataset4	0.08	1.09	0.04	-0.03	0.02	0.06
Dataset5	1.17	0.32	-0.13	0.33	-0.15	0.00
Dataset6	-0.09	-0.48	-0.08	-0.07	-0.14	-0.13
Dataset7	-0.04	2.31	1.79	0.60	1.63	0.51
Dataset8	-0.04	1.07	5.32	-0.01	0.44	0.70
Dataset9	-0.05	0.05	4.38	0.28	0.54	0.05
Dataset10	2.14	3.75	3.49	1.20	-0.04	0.14

CHAPTER EIGHT

CONCLUSION

In this study, a different type of the Capacitated Lot Sizing and Scheduling Problem (CLSP) is tested over different VNS variants. Different than the classical single machine case, there are parallel machines with different production capabilities. On the other hand, sequence-dependent setup times, costs and setup carryover are considered. Moreover, each machine has different capacities varying from period to period. These above criteria makes the problem more complex.

In the literature, 100 benchmark instances have been generated which are classified into 10 different combinations and have different characteristics and the overall best known solutions are presented in the paper of James & Almada-Lobo, (2011).

In order to improve the overall best known solutions in the literature, variants of VNS are utilized. As neighborhood strategies, three types of moves are used. In fact, these moves were previously used for the single machine case by Almada-Lobo & James, (2010). However, in this study, these moves have been adapted to the parallel machine case.

Another difference of this study is the use of Constraint Handling Techniques. In order to give a chance to the infeasible solutions, violation of some constraints like capacity violation, demand satisfaction, and violation on the upper bound production quantity are penalized by *Near Feasible Threshold* (Smith & Tate, 1993; Tate & Smith, 1995). Also, *Superiority of Feasible Solutions* is used (Deb, 2000).

For the VNS, fractional insert move is used only in the *Shaking Phase* and the remaining two moves are used respectively in the *Local Search Phase*. Maximum computational time is used as the terminating condition, which is 3600 seconds. As a result, both VNS algorithms work well almost for every problem instance. On the other hand, when total number of possible product-machine allocations, *MProb*, is at

low level, then the neighborhood combination of N_1 : insert move N_2 : swap move has a better solution quality compared to the other neighborhood pair.

Another variant of VNS, VND is also used as a tested heuristic. Two different VND algorithms have been tested over the 100 benchmark problems. Since the entire search space within a neighborhood will be made in VND, the neighborhood structure is the pairwise combinations of insert and swap moves. Moreover, it is clear that both VND algorithms works relatively well since the differences from average deviations are quite satisfactory. One advantage of the VND algorithm is its efficiency in terms of the computation time. The computation time of both VND algorithms are approximately 18 minutes, which is quite reasonable.

Lastly, six pairwise combinations of RVNS algorithm have also been tested. In this case, maximum number of 3000 iterations is used as the terminating criteria, which took almost 8 hours for each instance. Also, average convergence points are determined for each combination of the test problems. RVNS variants resulted best and nearly all overall best known solutions were improved. However, the weakness of the RVNS is the high computation time. As a result, the efficiency of the heuristic is low. Similar to all algorithms tested, an increase in the number of products results in low % deviation from the lower bound.

For the future work, algorithms presented in this study can be applied to the various lot sizing and scheduling problems in the literature.

REFERENCES

- Abbasi, B., Niaki, S. T. A., Khalife, M. A. & Faize, Y. (2011). A hybrid variable neighborhood search and simulated annealing algorithm to estimate the three parameters of the Weibull distribution. *Expert Systems with Applications*, 38 (1), 700-708.
- Almada-Lobo, B., Klbjan, D., Carravilla, M. A. & Oliveira, J. F. (2007). Single machine multi-product capacitated lot sizing with sequence-dependent setups. *International Journal of Production Research*, 45 (20), 4873-4894.
- Almada-Lobo, B. & James, R. (2010). Neighborhood search metaheuristics for capacitated lot sizing and sequence-dependent setups. *International Journal of Production Research*, 48, 861-878.
- Anderson, E. J. & Ferris, M. C. (1994). Genetic algorithms for combinatorial optimization: the assembly line balancing problem, *Journal on Computing*, 6, 161-173.
- Baeck, T., Fogel, D. & Michalewicz, Z. (Eds.) (1997). *Handbook of evolutionary computation*. Oxford: Oxford.
- Barbarosoğlu, G. & Özdamar, L. (2000). A simulated annealing approach for the multi-level multi-item capacitated lot sizing problem. *Computers and Operations Research*, 27, 895-904.
- Behnamian, J., Zandieh, M. & Fatemi Ghomi, S. M. T. (2009). Parallel-machine scheduling problems with sequence-dependent setup times using an ACO, SA and VNS hybrid algorithm. *Expert Systems with Applications*, 36 (6), 9637-44.

- Beraldi, P., Ghiani, G., Grieco, A. & Guerriero, E. (2008). Rolling-horizon and fix-and-relax heuristics for the parallel machine lot-sizing and scheduling problem with sequence-dependent set-up costs. *Computers and Operations Research*, 35 (11), 3644-56.
- Bitran, G. R. & Yanasse, H. H. (1982). Computational complexity of the capacitated lot size problem. *Manage Science*, 28 (10), 1174–1186.
- Blocher, J. D., Chand, S. & Sengupta, K. (1999). The changeover scheduling problem with time and cost considerations: analytical results and a forward algorithm. *Operations Research*, 47(4), 559–569.
- Brüggemann, W. & Jahnke, H. (2000). The discrete lot-sizing and scheduling problem: complexity and batch modification for batch availability. *European Journal of Operational Research*, 124 (3), 511-528.
- Carlson, S. E. (1995). A general method for handling constraints in genetic algorithms. *Proceedings of the Second Annual Joint Conference on Information Science*, 663-667.
- Cattrysse, D., Salomon, M., Kuik, R. & Van Wassenhove, L. N. (1993). Adual ascent and column generation heuristic for the discrete lotsizing and scheduling problem with setup times. *Management Science*, 39(4), 477–486.
- Chen, Y. Y., Chen-Yang, Y., Wang, L. & Chen, T. (2013). A hybrid approach based on the variable neighborhood search and particle swarm optimization for parallel machine scheduling problems—a case study for solar cell industry. *International Journal of Production Economics*, 141(1), 66-78.
- Chen, W. H. & Thizy, J. M. (1990). Analysis of relaxations for the multi-item capacitated lot-sizing problem. *Annals of Operations Research*, 26, 29-72.

- Cheng, C. H., Madan, M. S., Gupta, Y. & So, S. (2001). Solving the capacitated lot-sizing problem with backorder consideration. *Journal of Operations Research Society*, 52, 952–959.
- Coello, C. C. (2000). Use of a self-adaptive penalty approach for engineering optimization problems. *Computers in Industry*, 41 (2), 113-127.
- Coello, C. C. (2002). Theoretical and numerical constraint-handling techniques used with evolutionary algorithms: a survey of the state of the art. *Computer Methods in Applied Mechanics and Engineering*, 191, 1245-1287.
- Coit, D. & Smith, A. E. (1995). Using a neural network as a function evaluator during GA search for reliability optimization. *Intelligent Engineering Systems Through Artificial Neural Networks*, 5, 369-374.
- Dastidar, G. S. & Nagi, R. (2005). Scheduling injection molding operations with multiple resource constraints and sequence dependent setup times and costs. *Computers and Operations Research*, 32, 2987-3005.
- Deb, K. (2000) An efficient constraint handling method for genetic algorithms. *Computer Methods in Applied Mechanical Engineering*, 186, 311-338.
- Drexl, A. & Kimms, A. (1997). Lot sizing and scheduling – survey and extensions. *European Journal of Operational Research*, 99, 221-235.
- Dobson, G., Karmarkar, U. S. & Rummel, J. L. (1987). Batching to minimize flow times on one machine. *Management Science*, 33, 784–799.
- Dobson, G. (1992). The cyclic lot scheduling problem with sequence dependent setups. *Operations Research*, 40 (4), 736–749.

- Elmaghraby, S. E. (1978). The economic lot scheduling problem (ELSP): review and extensions. *Management Science*, 24, 587-598.
- Eppen, G. D. & Martin, R. K. (1987). Solving multi-item capacitated lot-sizing problems using variable redefinition. *Operations Research*, 35, 832-848.
- Erlenkotter, D. (1990). Ford Whitman Harris and the economic order quantity model. *Operations Research*, 38 (6), 937-946.
- Ferreira, D., Morabito, R. & Rangel, S. (2009). Solution approaches for the soft drink integrated production lot sizing and scheduling problem. *European Journal of Operational Research*, 196, 697-706.
- Fleischmann, B. (1990). The discrete lot-sizing and scheduling problem. *European Journal of Operations Research*, 44, 337-348.
- Fleischmann, B. (1994). The discrete lot-sizing and scheduling problem with sequence-dependent setup costs. *European Journal of Operational Research*, 75(2), 395-404.
- Fleischmann, B. & Meyr, H. (1997). The general lotsizing and scheduling problem. *OR Spektrum*, 19(1), 11-21.
- Gelders, L. F., Maes, J. & van Wassenhove, L. N. (1986). A Branch and Bound Algorithm for the multi-item single level capacitated dynamic lotsizing problem. *Lecture notes in economics and mathematical systems*, 266, 92-108.
- Gen, M. & Cheng, R. (2000). *Genetic Algorithms and engineering optimization*, Canada: John Wiley & Sons.
- Goldberg, D. (1989). *Genetic Algorithms in search, optimization and machine learning*. Reading, Massachusetts: Addison-Wesley Publishing Co.

- Gopalakrishnan, M., Miller, D. M. & Schmidt, C. P. (1995). A framework for modeling setup carryover in the capacitated lot-sizing problem. *International Journal of Production Research*, 33 (7), 1973-1988.
- Gopalakrishnan, M., Ding, K., Bourjolly, J. M. & Mohan, S. (2001). A tabu-search heuristic for the capacitated lot-sizing problem with set-up carryover. *Management Science*, 47 (6), 851-863.
- Gupta, D. & Magnusson, T. (2005). The capacitated lot-sizing and scheduling problem with sequence-dependent setup costs and setup times. *Computers & Operations Research*, 32(4), 727–747.
- Harris, F. W. (1913). How many parts to make at once. *Factory: The Magazine of Management*, 1(10), 135-136.
- Haase, K. (1998). *Capacitated lot-sizing with linked production quantities of adjacent periods*, Berlin: Springer.
- Haase, K. & Kimms, A. (2000). Lot sizing and scheduling with sequence dependent setup costs and times and efficient rescheduling opportunities. *International Journal of Production Economics*, 66 (2), 159–169.
- Hansen, P., Mladenović, N. & Perez-Britos, D. (2001). Variable neighbourhood decomposition search. *Journal of Heuristics*, 7 (4), 335- 350.
- Hansen, P. & Mladenović, N. (2005). *Variable neighborhood search* (211-238). Berlin: Springer.
- Hansen, P., Mladenović, N. & Moreno Pérez, J. A. (2010). Variable neighbourhood search: Methods and applications, *Annals of Operations Research*, 175 (1) , 367-407.

- Hindi, K. S. (1996). Solving the CLSP by a tabu search heuristic. *Journal of the Operational Research Society*, 47, 151-161.
- Hoffmeister, F. & Sprave, J. (1996). Problem-independent handling of constraints by use of metric penalty functions. *Proceedings of the Fifth Annual Conference on Evolutionary Computing*. 289-294. San Diego, California.
- Homaifar, A., Lai, S. H. Y. & Qi, X. (1994). Constrained optimization via genetic algorithms. *Simulation*, 62 (4), 242-254.
- Hung Y. F. & Chien K. L. (2000). A multi-class multi-level capacitated lot sizing model. *Journal of Operations Research Society*, 51 (11), 1309–1318.
- James, J. W. R. & Almada-Lobo, B. (2011). Single and parallel machine capacitated lotsizing and scheduling: new iterative MIP-based neighborhood search heuristics. *Computers & Operations Research*, 12 (38), 1816-1825.
- Jans, R. (2009). Solving lot-sizing problems on parallel identical machines using symmetry-breaking constraints. *Inform Journal on Computing*, 21 (1), 123-36.
- Jans, R. & Degraeve, Z. (2008). Modeling industrial lot sizing problems: a review. *International Journal of Production Research*, 46 (6), 1619-43.
- Joines, J. & Houck, C. (1994). On the use of non-stationary penalty functions to solve nonlinear constrained optimization problems with GAs. *Proceedings of the first IEEE Conference on Evolutionary Computation*. 579-584. Orlando, Florida.
- Kang S, Malik K. & Thomas L. J. (1999). Lotsizing and scheduling on parallel machines with sequence-dependent setup costs. *Management Science*, 45 (2), 273–289.

- Kazarlis, S. & Petridis, V. (1998). Varying fitness functions in genetic algorithms: studying the rate of increase of the dynamic penalty terms. *Parallel Problem Solving from Nature – V PPSN V*, Amsterdam, Netherlands: Springer Verlag.
- Kimms, A. (1999). A genetic algorithm for multi-level, multi-machine lot sizing and scheduling. *Computers & Operations Research*, 26, 829-848.
- Kohlmorgen, U., Schmeck, H. & Haase, K. (1999). Experiences with fine-grained parallel genetic algorithms. *Annals of Operations Research*, 90, 203–219.
- Kuik, R. & Salomon, M. (1990). Multi-level lot sizing problem: evaluation of a simulated-annealing heuristic. *European Journal of Operational Research*, 45, 25-37.
- Kuik, R., Salomon, M., van Wassenhove, L. N. & Maes, J., (1993). Linear programming, simulated annealing and tabu search heuristics for lotsizing in bottleneck assembly systems. *IIE Transactions*, 25 (1), 62-72.
- Laporte, G. (1992). The traveling salesman problem: An overview of exact and approximate algorithms. *European Journal of Operational Research*, 59, 231-47.
- Laporte, G. (1992). The vehicle routing problem: An overview of exact and approximate algorithms. *European Journal of Operational Research*, 59, 345-58.
- Liu, L. & Zhou, H. (2013). Hybridization of harmony search with variable neighborhood search for restrictive single-machine earliness/tardiness problem. *Information Sciences*, 226, 68-92.
- Maes, J. & Van Wassenhove, L. N. (1991). Multilevel capacitated lotsizing complexity and LP-based heuristics. *European Journal of Operational Research*, 53, 131–148.

- Martins, A. X., Duhamel, C., Mahey, P., Saldanha, R. R. & de Souza, C. M. (2012) Variable neighborhood descent with iterated local search for routing and wavelength assignment. *Computers and Operations Research*, 39 (9), 2133-41.
- Mladenović, N. & Hansen, P. (1997). Variable neighborhood search. *Computers and Operations Research*, 24, 1097-1100.
- Mladenović, N., Dražić, M., Kovačević-Vujčić, V. & Čangalović, M. (2008) General variable neighborhood search for the continuous optimization, *European Journal of Operational Research*, 191 (3), 753- 770.
- Meyr, H. (2000). Simultaneous lotsizing and scheduling by combining local search with dual reoptimization. *European Journal of Operational Research*, 120 (2), 311–326.
- Meyr, H. (2002). Simultaneous lotsizing and scheduling on parallel machines. *European Journal of Operational Research*, 139 (2), 277–292.
- Michalewicz, Z. & Attia, N. F. (1994). Evolutionary optimization of constrained problems. *Proceedings of the third annual conference on Evolutionary Programming*, 98-108. World Scientific.
- Michalewicz, Z. (1995). A survey of Constraint Handling Techniques in Evolutionary Computation Methods. *Proceedings of the fourth Annual Conference on Evolutionary Programming*, 135-155.
- Millar, H. H. & Yang, M. (1994). Lagrangian heuristics for the capacitated multi-item lot-sizing problem with backordering. *International Journal of Production Economics*, 34 (1), 1–15.
- Miller, M. D., Chen, H. C., Matson, J. & Liu, Q. (1999). A hybrid genetic algorithm for the single machine scheduling problem. *Journal of Heuristics*, 5, 437–454.

- Moursli, O. & Pochet, Y. (2000). A Branch-and-Bound algorithm for the hybrid flowshop. *International Journal of Production Economics*, 64, 113-125.
- Nemhauser, G. L. & Wolsey, L. A. (1988). *Integer and combinatorial optimization*. New York: Wiley.
- Özdamar, L. & Birbil, S. I. (1998). Hybrid heuristics for the capacitated lot sizing and loading problem with setup times and overtime decisions. *European Journal of Operations Research*, 110, 525-547.
- Özdamar, L. & Barbarosoğlu, G. (2000). A lagrangean relaxation - simulated annealing approach to the multi-level multi-item capacitated lot sizing problem. *International Journal on Production Economics*, 68, 319-331.
- Özdamar, L. & Bozyel, M. A. (2000). The capacitated lot sizing problem with overtime decisions and setup times. *IIE Transactions*, 32, 1043-1057.
- Özdamar, L., Birbil, S. I. & Portmann, M. C. (2002). New results for the capacitated lot sizing problem with overtime decisions and setup times. *Production Planning and Control*, 13, 2-10.
- Pedroso, J. P. & Kubo, M. (2005). Hybrid tabu search for lot sizing problems. *Hybrid Metaheuristics, Lecture Notes in Computer Science*. 3636, Berlin: Springer.
- Pochet, Y. & Wolsey, L. (1988). Lot size models with back-logging: strong reformulations and cutting planes. *Mathematical Programming*, 40, 317-335.
- Quadt, D. & Kuhn, H. (2005). A conceptual framework for lot-sizing and scheduling of flexible flow lines. *International Journal of Production Research*, 43 (11), 2291-2308.

- Quadt, D. & Kuhn, H. (2008). Capacitated lot-sizing with extensions: a review. *4OR*, 6, 61-83.
- Quadt, D. & Kuhn, H. (2009). Capacitated lot-sizing and scheduling with parallel machines, back-orders, and setup carry-over. *Naval Research Logistics*, 56 (4), 366-384.
- Riane, F., (1998). *Scheduling hybrid flowshops: algorithms and applications*, Ph.D. Thesis. Facultes Universitaires Catholiques de Mons.
- Richardson, J. T., Palmer, M. R., Liepins, G. & Hilliard, M. (1989). Some Guidelines for Genetic Algorithms with Penalty Functions. *Proceedings of the Third International Conference on Genetic Algorithms*. 191-197, George Mason University.
- Reiter, B. S., Hildebrandt, T. & Tan, Y. (2013). Effective and efficient scheduling on dynamic job shops-combining the shifting bottleneck procedure with variable neighborhood search. *CIRP Annals-Manufacturing Technology*, 62 (1), 423-426.
- Salomon, M., Kuik, R. & van Wassenhove, L. N. (1993). Statistical search methods for lotsizing problems. *Annals of Operations Research*, 41, 453-468.
- Salomon, M, Solomon, M. M., Van Wassenhove, L. N., Dumas, Y. & Dauzere-Peres, S. (1997). Solving the discrete lotsizing and scheduling problem with sequence dependent set-up costs and set-up times using the travelling salesman problem with time windows. *European Journal of Operational Research*, 100, 494-513.
- Seeanner, F., Almada-Lobo, B. & Meyr, H. (2013). Combining the principles of variable neighborhood decomposition search and the fix&optimize heuristic to solve multi-level lot-sizing and scheduling problems. *Computers and Operations Research*, 40 (1), 303-317.

- Smith, A. E. & Tate, D. M. (1993). Genetic optimization using penalty function. *Proceedings of the Fifth International Conference on Genetic Algorithms*. 499-503, San Mateo, California.
- Smith-Daniels V. L. & Smith-Daniels D. E. (1986). A mixed integer programming model for lot sizing and sequencing packaging lines in the process industries. *IIE Transactions*, 18, 278–285.
- Suerie, C. & Stadtler, H. (2003). The capacitated lot-sizing problem with linked lot sizes. *Management Science*, 49 (8), 1039-1054.
- Takahama, T. & Sakai, S. (2005). Constrained optimization by applying the α constrained method to the nonlinear simplex method with mutations. *IEEE Transactions on Evolutionary Computation*, 9 (5), 437–451.
- Takahama, T., Sakai, S. & Iwane, N. (2006). Solving nonlinear constrained optimization problems by the ϵ constrained differential evolution. *Proceedings of the 2006 IEEE Conference on Systems, Management, and Cybernetics*. 2322–2327.
- Takahama, T. & Sakai, S. (2010). Constrained optimization by the ϵ -constrained differential evolution with an archive and gradient-based mutation. *IEEE Congress on Evolutionary Computation*, 1-9. Barcelona.
- Tate, D. M. & Smith, A. E. (1995). Unequal area facility layout using genetic search. *IIE Transactions*, 27, 465-472.
- Thizy J. M. & Van Wassenhove L. N. (1985). Lagrangean relaxation for the multi-item capacitated lot-sizing problem: a heuristic implementation. *IIE Transactions*, 17 (4), 308–313.

- Trigeiro, W. W., Thomas, L. J. & McClain, J. O. (1989). Capacitated lot sizing with setup times. *Management Science*, 35(3), 353–366.
- Wagner, H. M. & Whitin, T. M. (1958). Dynamic version of the economic lot size model. *Management Science*, 5, 89-96.
- Wagner, B. J. & Davis, D. J. (2002). A search heuristic for the sequence dependent economic lot scheduling problem. *European Journal of Operational Research*, 141(1), 133–146.
- Wittrock, R. J. (1988). An adaptable scheduling algorithm for flexible flow lines. *Operations Research*, 36 (4), 445-453.
- Xiao, J., Zhang, C., Zheng, L. & Gupta, J. N. D. (2013). MIP-based fix-and-optimize algorithms for the parallel machine capacitated lot-sizing and scheduling problem. *International Journal of Production Research*, DOI:10.1080/00207543.2013.790570.

APPENDICES

