Production Planning in a Multi-Product Manufacturing Environment Using Constant Work-In-Process

Wen Zhang

A Thesis

in

The Department

of

Mechanical and Industrial Engineering

Presented in Partial Fulfillment of the Requirements

for the Degree of Doctor of Philosophy at

Concordia University

Montréal, Québec, Canada

April 2007

Your file Votre référence
ISBN: 978-0-494-30146-3
Our file Notre référence
ISBN: 978-0-494-30146-3

ABSTRACT

Production Planning in a Multi-Product Manufacturing Environment Using Constant Work-In-Process

Wen Zhang, Concordia University, 2007

Push-based MRP and pull-based Kanban systems are effective production control policies for a wide range of manufacturing environment. Both of them, however, have certain limitations when they are implemented in different production environments. In recent years, CONWIP (CONstant Work-In-process), a hybrid push/pull control policy, was proposed and studied to take advantages of MRP and Kanban systems for optimal work-in-process (WIP) inventory control. CONWIP is a closed production system in which a constant number of containers traverse a closed loop that includes the entire production system.

In order to effectively implement CONWIP control in a manufacturing environment, several issues, such as the number of containers, lot sizes and job sequence, need to be addressed. This research aims at the development of mathematical models to address these issues and to help implement CONWIP systems in different manufacturing environments. Two mathematical programming models are developed to address issues on a single serial CONWIP line system. The first model can be used in a make-to-stock environment and the objective function of the model is to minimize the setup costs and the costs associated with an unbalanced workload at the bottleneck machine. The solution of the first model simultaneously determines the optimal job sequence on the part list and the lot size associated with each entry on the part list. The second model can be used in a make-to-order environment or in a make-to-stock environment with a known part list. The objective function of the second model is to minimize the system makespan. Two essential CONWIP system parameters, number of containers and job sequence can be determined by solving the second model. A third model is developed for an assembly-type CONWIP system with multiple fabrication lines feeding an assembly station. The objective of the third model is to synchronize system production by minimizing makespan differences among all the fabrication lines. The solution of the third model determines the number of the containers and job sequence for each of the fabrication lines.

The solution of such models is NP-hard. The general branch and bound approach used by most off-the-shelf optimization software cannot be used to solve real sized problems of this nature. In order to solve real sized problems efficiently, we develop a heuristic search method based on simulated annealing. Several example problems are used to test the developed models and algorithms. Computational results validate the modelling and computational efficiency of the solution methods.

ACKNOWLEDGEMENTS

I would like to express my sincere gratitude to **Dr. Mingyuan Chen** of Concordia University for his time, support and guidance throughout the course of this research. He spent many hours of his time to teach and guide me. His supervision and encouragement were really of a paramount importance to me in completing this work. The journal papers and refereed conference papers published from my thesis and those submitted for review were made possible by his most responsive follow-ups and incisive critiques on the original drafts. It was really my great pleasure to have worked with him. I would like to acknowledge the examiners **Dr. Vince J. Thomson**, **Dr. Shahin Hashtrudi Zad**, **Dr. Akif Bulgak**, and **Dr. Kudret Demirli** for their time and helpful suggestions.

I would like to thank my family for their support and understanding for the time I spent in the Ph.D. Program.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

Chapter 1

Introduction

1.1. Background

As international competition increases, manufacturing companies are now facing much higher pressure to improve their efficiency, quality, flexibility and profitability than they ever have. To stay in the competition, they need to provide products with highest performance and reliability together with improvement in cost, cycle times and overall operation efficiency. Nowadays, not only manufacturing industry's big names, but also small sized companies are looking for better solutions. Many of them have found that lean manufacturing changed their mind about how a factory should be organized. As the chief economist at Canadian Manufacturers and Exporters, Jay Myers said, "If you're a manufacturing company out there today, you have to be focused on lean manufacturing because lean manufacturing is just a systematic way of doing two things: delivering what customers value and eliminating waste (Macdonald, 2006)." Lean manufacturing refers to a set of techniques aimed at reducing waste by cutting out activities that add cost but not value (Macdonald, 2006). Not just lean manufacturing, but many other managerial and operational approaches have been developed to cope with today's markets. Just-In-Time (JIT), agile manufacturing, constant work-in-process (CONWIP), the new generation of Enterprise Resource Planning (ERP), etc. are widely adopted by manufacturers to help them survive and be successful in the fierce competition. CONWIP is the main focus of this study.

Traditionally, manufacturing companies may not be particularly responsive to changing customer demands, a manufacturer may rely on forecasting future demand and scheduling the release of work into the system to meet expected demand. It is a so called push-based material requirement planning (MRP) method. Such production systems often have excess inventory, higher WIP levels, and longer quoted lead times from order to delivery. The high WIP is a result of long lead times from MRP systems to handle shop floor uncertainties. In contrast, just-in-time production, a pull-based production planning and control, relies on actual demand triggering the release of work into the system, and pulling work through the system to fill the demand order. Just-in-time production is better able to respond to changing customer demands as it advocates producing the right products at the right time and in the right amounts. Pull-based production systems have many advantages over push-based production systems but pull systems are more applicable to repetitive manufacturing. Limitations of pure pull and pure push production systems led researchers to investigate the potential of combining the two. It is becoming more apparent that, in many instances, a hybrid production system is more effective than either a pure push or a pure pull production systems alone. Since the early 1990's, the CONstant Work-In-Process (CONWIP) production system, a pull and push hybrid system, has received significant attention because of its simplicity in implementation and effectiveness in inventory control.

1.1.1. Push and Pull Production Control Systems

In a push controlled production system, production planning is based on forecasted demand, bill of materials, component lead times and inventory status. The information flow in a push system is from the beginning of the production line to the end of the line. Based on the production plan, production starts at the initial workstation in the production line when the required raw material arrives. Once the job is finished at the workstation, it is "pushed" to the subsequent workstation for further processing. The problem with the push system is its high dependency on forecasted lead time. Since it is very difficult to accurately calculate the lead times required by components or end products shipped to customers, to make it "safe", production lead times in a push controlled system are often inflated and thus lead to excessive Work-In-Process (WIP) inventories and plant congestion. In addition, MRP controlled production systems are not flexible in responding to market changes (Ignizio, 2003).

In contrast to a push type system, a pull type production control seeks to reduce the inventory to a minimal level and shorten the lead time. A pull system (such as a Kanban system) does not use forecasted demands directly for production control. Production is triggered by actual component or end product orders. It reacts to make products to satisfy demands from its customers or downstream stations. When demand arrives at the final station of the production line, components used for demanded products are checked to see if they are available. If so, the production at the station starts; otherwise, a request is sent to the previous workstation to request parts. A similar procedure is followed backwards through each workstation until reaching the first workstation of the production line. Since the number of Kanbans or cards allowed to be used in the system is creating an upper limit on the total amount of WIP, control of WIP becomes much easier and hence it can be significantly reduced in a pull system. In general, push systems schedule releases, control release rate and hence throughput and observe work-in-progress (WIP). While pull systems authorize releases, control WIP and observe throughput.

Push systems are generally considered to be applicable to more manufacturing firms than pull systems, but the latter seem to produce superior results when they can be used. This is due to the fact that controlling WIP is easier than controlling throughput in complex production systems (Spearman et al., 1990). In order to take advantage of the wide applicability of push systems and superior performance of pull systems, a number of hybrid production control strategies were proposed and studied by different researchers such as Synchro-MRP (Hall, 1981), CONWIP model (Spearman et al., 1990), generic Kanban model (Chang and Yih, 1994) and drum-buffer-rope (DBR) model (Goldratt, 1984). Among all these variations, CONWIP has drawn more attention because of its simplicity of implementation.

1.1.2. CONWIP Production Control Systems

In a CONWIP system, production is also triggered by actual demand of the products. When demand arrives at the final workstation, components used for demanded products are checked to see if they are available. If so, the production at the station starts; otherwise, a request is sent to the first workstation of the production line to request parts. Once jobs are finished at the first workstation of the production line, they are then pushed to the next workstation until they reach the final workstation. Its pull mechanism is similar to a typical Kanban system in that the production of the first workstation is also triggered by demand. It differs from a Kanban systems in that Kanban systems are pulled everywhere between two consecutive workstations while CONWIP systems are only pulled between the final workstation and the first workstation. CONWIP control systems are also similar to the MRP systems inside the production line. Components are pushed from up-stream workstations to down-stream workstations. It is different from MRP in that it controls WIP by observing the throughput while MRP controls the throughput (Hopp et al., 1998). CONWIP control systems have many advantages over pure push or pure pull systems. They include (Hopp et al., 1998; Marek et al., 2001; Ignizio, 2003):

1. Observability: CONWIP controls WIP and therefore WIP is directly observable while capacity is not.

2. Efficiency: Since CONWIP has the properties of both push and pull systems, it can achieve the same throughput rate as a push system but with smaller average WIP level.

3. Variability: As CONWIP systems regulate the WIP level, flow times are less variable than in those push systems.

4. Robustness: CONWIP systems are less sensitive to errors in WIP level than push systems to errors in release rate.

5. Adaptability: CONWIP systems can be easily used in a non-repetitive manufacturing environment where Kanban systems are difficult to use. Monden (1983) concluded that Kanban systems are difficult to use in several situations:

    - Job orders with short production runs
    - Significant setups
    - Large, unpredictable fluctuations in demand

6. Simplicity: CONWIP systems are easier to implement than Kanban systems.

1.2. Problem Statement

Since its introduction, CONWIP systems have attracted much attention from practitioners and academics. As a pull system, it shares the advantages of pull systems with respect to WIP control, while it is considered more robust, flexible and easier to implement than a typical pull system. These are important characteristics for manufacturing companies that try to control inventory levels in uncertain and dynamic environments where Kanban systems do not perform well (Framinan et al., 2003). From today's predominant ERP (Enterprise Resource Planning) viewpoint, CONWIP is a tool for manufacturing control based on synchronous production (Shtub, 1999). In recent years, the range of applications of CONWIP, like other pull systems, has been enhanced by the possibility of using these control systems not only within the manufacturing stages, but also for the different echelons of the supply chain (Knolmayer, 2002).

Common questions when establishing a CONWIP system are, how to forecast the backlog list, number of cards operating in the system, and how to sequence the jobs in the system. Most of the available research articles focus on the card setting and job sequencing for single production line while card setting and job sequencing are usually investigated separately. There are almost no papers on mathematical models for multiple CONWIP production lines. Also a question that has not been well studied is the impact of lot-sizing, i.e., the number of jobs to attach to a card or a container, on system performances (Framinan et al., 2003).

It is clearly beneficial to study both card setting and job sequencing at the same time and to investigate how these two factors would impact on the overall CONWIP system performance.

1.3. Contributions of the Thesis

In this research, we develop mathematical programming models to address the major issues required to implement CONWIP system in an existing manufacturing system.

A non-linear integer programming model is developed to simultaneously determine optimal job sequence and transfer lot sizes in a serial CONWIP production line. The model considers sequence dependent setup cost and workload balancing. These factors have not been well studied in published papers on CONWIP. The model also considers transfer lot size and its effect on system performance. Research on this issue is also limited in papers published so far.

We develop a second mathematical model for a single serial CONWIP line to determine not only the job sequence, but also the minimal number of containers or WIP level in the system. The major contribution of this model is that it can decide the number of containers directly.

Another major contribution of this research is that we develop a mathematical model for multiple line CONWIP systems. No such mathematical model is seen in papers published in the field of CONWIP system research. The model determines the best number of containers or WIP settings for each fabrication line to obtain the best overall system performance.

To solve these complicated mathematical programming models for large and close to real world problems, we propose an efficient heuristic based on simulated annealing algorithm. Six perturbation schemes and other specific simulated annealing parameters are investigated to see how they affect the solution quality and the computational efforts. The developed simulated annealing algorithm is implemented using C# in a software CONLine. We design CONLine with a user friendly GUI. CONLine allows user to solve the model, enter the CONWIP system data and control the SA parameters easily via the GUI.

1.4. Thesis Organization

Following this chapter of introduction, Chapter 2 presents an extensive review of the literature for the CONWIP systems, as well as several solution methods developed to solve CONWIP and similar problems in manufacturing systems.

Chapter 3 presents two mathematical models developed for a single CONWIP production line. Lot size, backlog list and job sequence can be obtained from the first model simultaneously, but the WIP level is handled outside the model. Also the bottleneck machine is required to be identified before using the model. The second model presented in this chapter overcomes the shortcomings of the first model in that identifying the bottleneck machine beforehand is not necessary. The second model can compute the WIP level along with the best job sequence.

Chapter 4 gives a mathematical model developed for assembly-type multiple line CONWIP systems. Job sequence and WIP level for each of the fabrication lines can be obtained at the same time from the solution of the model.

Chapter 5 presents an algorithm based on simulated annealing developed to resolve the second model of the single CONWIP line and the model for assembly-type multiple CONWIP lines. Such a heuristic method is needed due to the extensive computational requirement in solving these models optimally.

Chapter 6 presents several numerical examples to illustrate the developed models and solution method. First model is solved using software package LINGO and a small size problem is illustrated to evaluate the model. The second model and the third model are solved using the developed heuristic algorithm. Simulated annealing based heuristic method parameters and results of several numerical examples for single CONWIP line and assembly-type multiple CONWIP lines are presented. The system performances of different examples is also illustrated in this chapter.

Chapter 7 offers conclusions of this research and some suggestions for further research in this area.

C# source code for the second and third models using the proposed SA based algorithm is included in Appendix C.

Chapter 2

Literature Review

Since Spearman et al. (1990) introduced CONWIP systems, many research articles studying such systems have been published. Hopp and Spearman (1991) studied CONWIP production lines in which processing times were deterministic but machines were subject to exponential failures and repairs. Duenyas and Hopp (1992) developed structural results and an approximation for the throughput of an assembly system. The system is fed by multi-station lines where releases were governed by the CONWIP protocol and all machines had deterministic processing times but were subject to random outrages. Spearman and Zazanis (1992) compared CONWIP systems with pure Kanban systems and offered theoretical analysis for the apparent superior performance of pull systems. Using a simulation study, Roderick et al. (1994) compared CONWIP and typical MRP with respect to due dates and cycle times to check the validity of the CONWIP model in an actual plant environment. Chang and Yih (1994) proposed a generic Kanban system to control a dynamic production system and compared it with the original Kanban system and the CONWIP system using simulation. Gstettner and Kuhn (1996) classified different pull production systems and analyzed Kanban and CONWIP with respect to production rate and average WIP. Hopp and Roof (1998) developed an adaptive production control method for setting WIP levels to meet target production rates in a CONWIP system. Huang et al. (1998) introduced a simulation study that compared the CONWIP system and the original control system for a cold rolling plant. Bonvik et al. (2000) presented a decomposition method for approximation performance analysis of tandem production systems that are controlled by the CONWIP finite buffer control policy. Leu (2000) used group scheduling heuristics and single-stage heuristics to generate the backlog list of CONWIP based flow line and then compared the performance of two heuristics by simulation. Beamon and Bermudo (2000) presented a hybrid control logic and a structured algorithm for a multi-line, multi-stage assembly-type production system. A simulation model was then used to test the algorithm on system performance based on output, lead time and WIP. Ryan and Choobineh (2003) proposed a planning procedure to set the constant level of WIP for each product type in a job shop controlled by CONWIP. Weitzman and Rabinowitz (2003) defined a modified CONWIP algorithm to compare push and pull strategies for production planning and control under different updating rates for inventory information. Framinan and Gonzalez (2003) reviewed different contributions on CONWIP production control system according to operation, application and comparison of CONWIP. They pointed out that most studies focused on card setting and job sequencing, but no research has been conducted to show the relative importance of the different implementation decisions on the system performance. Koh and Bulfin (2004) analyzed and compared drum-buffer-rope (DBR), another alternative of push-pull hybrid control policy, and CONWIP in a three-stage unbalanced tandem production line.

From the above general review of the CONWIP related literatures published recently in a chronological order, some researchers used simulation to compare push and pull, push and different push/pull hybrid alternatives, while some others used stochastic models to analyze CONWIP systems to set WIP levels. All this research helps us to understand the properties of CONWIP systems, but they do not address all the issues raised in CONWIP systems. Our particular interests in this research are in the area of research on CONWIP mathematical programming models and methods leading to solve them. Fewer literatures are reported in this area. In the next sections, we will review some of such research papers in this area.

2.1. Mathematical Programming Model for CONWIP Systems

The essential system parameters in CONWIP are the size of the WIP (number of containers) and the sequencing rules for the backlog list (Golany et al., 1999). Very little has been published in this area and the research that has been reported has focused mainly on the issue of WIP level. This was done mainly by analyzing single item systems characterized with either stochastic processing times or with deterministic processing times and stochastic breakdowns.

Herer and Masin (1997) developed a deterministic mathematical programming model for a single serial CONWIP line to generate an optimal sequence of jobs. The objective function of their model is to minimize inventory related costs including finished goods holding cost, shortage cost, WIP holding cost and overtime cost. Optimal job order and schedule are obtained based on mean throughput and flow time using mean value analysis (MVA). Lot sizes, number and the effects of bottleneck machines on job orders were not considered in their model. No algorithm leading to solving the model was developed in their published paper.

Golany et al. (1999) developed a mathematical model for a multi-cell, multi-family CONWIP production environment. The model was used to simultaneously find the best level of WIP and the sequence of the backlog list. Two variant CONWIP control mechanisms were compared. In the first one, containers are restricted to stay within given cells all the time. In the other, containers are allowed to move through the entire system. The model was solved by a simulated annealing heuristic. The objective function of the model was to minimize the overall completion time. Inventory costs and setup costs were not considered in the model. Lot size (the capacity of the container) was not considered in the model.

Luh et al. (2000) developed a mathematical programming model for a single CONWIP based serial production line in a job shop for Sikorsky Aircraft. The objective function of the model is to minimize weighted penalties on tardiness and on early releasing of raw materials. The model was approximately solved to schedule a set of jobs over a specific time period to meet fixed due dates at a given WIP level. A synergistic combination of Lagrangian relaxation, dynamic programming and heuristic methods were used to solve the model. The number of containers and lot size were not addressed in their work.

Cao and Chen (2005) developed a nonlinear mixed integer programming model for a CONWIP-based production system where an assembly station is fed by two parallel fabrication lines. The model was linearized and solved by enumerating a series of solutions of TSP sub-problems. The optimal job sequence and lot sizes can be simultaneously determined by solving the model. Other issues involved in the CONWIP system, such as number of containers or WIP level were not discussed in the paper and not computed in the model.

As mentioned earlier, aside from the number of containers and job order in the backlog list, lot size or the capacity of container is one of the crucial aspects in implementing CONWIP systems. A lot size larger than necessary will needlessly increase inventory costs, thus offsetting the objective of employing the CONWIP system, while a lot size too small will incur excessive setup costs and may create backorders that will ultimately cause a CONWIP system to collapse completely.

The models developed in the above three CONWIP publications did not address this essential issue on lot size for the CONWIP systems.

2.2. Meta-Heuristic Algorithms

Mathematical programming models developed to obtain job sequence, WIP level and lot sizes are often NP-complete. This follows from the fact that flow shop sequencing problems are typically NP-hard (Garey et al., 1979). In the case of CONWIP, when the number of containers is greater than the number of items, the problem is already NP-hard (Golany et al., 1999). Thus, solution to solve the CONWIP system mathematical model should be found by some heuristic algorithms for practical applications.

Many meta-heuristics have been implemented to solve combinatorial production planning optimization problems by researchers and practitioners. These heuristic algorithms include Simulated Annealing (SA), Genetic Algorithm (GA), Tabu Search (TS), Hill climbing procedures, and Ant Colony Optimization etc. In addition, some hybrid heuristic algorithms, combining these meta-heuristic algorithms, are also introduced in many research papers.

2.2.1. Simulated Annealing Algorithm

Simulated Annealing (SA) was first introduced by Kirkpatrick et al. in 1983. It is a random search technique that exploits an analogy between annealing process of metals and the search for an optimum in a more general system. The main procedure of SA can be described as follows. It starts with an initial solution of the problem and then searches in the neighborhood of the current solution to generate a new testing solution. If the new testing solution is better than the current solution based on the value of the objective function, it is accepted and used as the new current solution. Otherwise, it may be accepted or rejected depending on an acceptance probability, which is determined by the difference between the objective function values of the two solutions and by a control parameter called temperature. This process then continues from the new current solution. Initially, the temperature was set at a high level, as in the annealing process, so that almost all moves will be accepted. It is then decreased slowly during the procedure until almost no move will be accepted (Hejazi et al., 2005). To implement a SA algorithm, generation mechanisms and cooling schedules must be determined. The generation mechanisms are the ways to generate neighborhood solutions. They are often called perturbation schemes. The cooling schedule includes determining parameters, such as initial temperature, final temperature, the number of iterations at each temperature, and the temperature changing schemes. Various perturbation schemes have been presented in literature. Osman and Potts (1989) used interchange neighborhood and shift neighborhood schemes to solve a flow shop scheduling problem. Ogbu and Smith (1990) employed the insertion and pair wise exchange as the perturbation schemes. The performances of these two perturbation schemes were compared. Sridhar and Rajendran (1993) chose adjacent interchange, insertion scheme and random interchange schemes as perturbation schemes. Tian et al. (1999) introduced six perturbation schemes in their SA algorithm: interchanging two adjacent jobs; interchanging two random jobs; moving a single job; moving a subsequence of jobs; reversing a subsequence of jobs and reversing and/or moving a subsequence of jobs. Performances of these six schemes were then evaluated in solving Traveling Salesman Problems (TSP), Flow-shop Scheduling Problems (FSP) and Quadratic Assignment Problems (QAP). The cooling schedule was also discussed by many researchers in their papers. Collins et al. (1988) and Hajek (1988) suggested a great variety of cooling schedules. Haddock and Mittenthal (1992) applied SA with a heuristic cooling function to a simulation optimization problem in which the total expected profit of a hypothetical automated manufacturing system was maximized. Their results showed that a lower final temperature, a slower rate of temperature decrease and a large number of iterations performed at each temperature level yielded better solutions. Alrefaei et al. (1999) presented a SA algorithm that used a constant temperature instead of decreasing temperature. They used two approaches to get the optimum solution and showed that both converged to the global optimal solutions.

There is substantial literature on applications of SA to solving sequencing problems. Osman et al. (1989) proposed a SA algorithm to minimize the maximum completion time of a permutation flow shop scheduling problem. A relatively large experimental problem with 20 machines and 100 jobs was solved. It showed that simulated annealing performs better than some known constructive heuristics. Ishibuchi et al. (1995) applied a modified simulated annealing algorithm with the best move strategy to solve a m-machine n-job flow shop sequencing problem with the objective of minimizing the makespan. They claimed that their modified SA was less sensitive to the choice of a cooling schedule than that of the standard simulated annealing algorithm. Golany et al. (1999) presented a SA algorithm specifically developed for solving CONWIP system scheduling problems. After comparing SA solutions with optimal solutions and random solutions, they concluded that the SA algorithm is an effective method to solve the multi-cell CONWIP problem. McMullen and Frazier (2000) presented a SA heuristic that simultaneously considers both setups and the stability of parts usages rates when sequencing jobs for production in a Just-In-Time environment. Several test problems were solved by the SA heuristic and the solutions were compared to the solutions obtained by a Tabu Search approach. Comparison showed that the simulated annealing approach provided better results compared to the Tabu Search approach. Ozdamar and Bozyel (2000) used several heuristic algorithms, including generic algorithm and simulated annealing, to solve a capacitated lot sizing problem. The problem was to decide lot sizes of multiple items over a planning horizon with the objective of minimizing setup and inventory holding costs. Computational results demonstrated that the SA approach produced high quality schedules and was most computationally efficient. Hejazi et al. (2005) gave a complete survey of flow shop scheduling problems with minimizing makespan as a criterion. In the paper, they reviewed a variety of methods for solving flow shop problems. The methods included some exact methods, constructive heuristics and meta-heuristics. The meta-heuristic methods covered simulated annealing, genetic algorithm, tabu search, etc. Vallada et al. (2006) reviewed and evaluated a variety of heuristics and meta-heuristics for m-machine flowshop scheduling problems with objective of minimizing total tardiness. They analyzed a total of 40 different heuristics and meta-heuristics and evaluated their performances under the same benchmark of instances for comparison. Jans and Degraeve (2007) provided a review of various meta-heuristics specifically developed to solve lot sizing problems. Simulated annealing was among the meta-heuristics they have reviewed.

2.2.2. Genetic Algorithm

Genetic Algorithm (GA) was first introduced by Holland in 1975. It is a stochastic heuristic that encompasses semi-random search methods whose mechanisms are based on the simplifications of evolutionary processes observed in nature (Gaafar et al., 2005). According to Gaafar et al. (2005), GA algorithm can be described as follows. In GA, a solution is represented as a chromosome. A set of such chromosomes is generated randomly to form an initial population. The generated chromosomes are evaluated for the corresponding objective function values and the population is updated. This process continues until an improved solution cannot be obtained or a prescribed number of iterations is reached. Genetic operators generate new chromosomes (children) from existing ones (parents) by manipulating the order of some chromosomes mainly in two ways: crossover and mutation. Every crossover operator is applied to two chromosomes (parents) and results in two new ones (children). Every mutation operator is applied to one chromosome and results in a different chromosome.

Many papers have been published in the areas of implementations of GA in production planning. Reeve (1995) compared the performance of SA and GA for flow shop problems ranging from small sized problems to large sized problems. They found that SA outperformed GA in most small sized problems while GA provided better solutions for large problems. Wang and Zheng (2003) investigated the effect of different initialization, crossover and mutation operators on the performances of a GA. They then proposed a hybrid heuristic for permutation flow shop problems. Simulation results showed that the hybrid heuristic method is effective. Gaafar and Masoud (2005) applied both GA and SA algorithms to scheduling in agile manufacturing systems to minimize the makespan. They compared the GA and SA algorithms with other heuristics and concluded that both GA and SA algorithms performed well. When comparing GA and SA, they stated that SA outperformed GA with a more robust performance.

2.2.3. Tabu Search Algorithm

Tabu search (TS) was first proposed by Glover in 1986 (Glover, 1986). Since its introduction, it has been used to solve various optimization problems in different areas by many researchers. According to Hejazi and Saghafian (2005), the TS algorithm can be described as follows. It starts from an initial solution and then applies a move mechanism to search the neighborhood of the current solution and to choose the most appropriate one. A neighborhood solution is accepted if it is not "Tabu" or if a criterion is fulfilled. To use the information about the search history, selected moves are stored in a data structure called "Tabu list". This list contains elements at a time and once a move is entered the oldest one is deleted. The selected move is put into the Tabu list and remains there for the following iterations. The length of the Tabu list is controlled by the Tabu list size parameter. To implement a TS algorithm, defining neighborhood, searching among neighbors and setting tabu list size are three important issues.

TS algorithms have been adopted by many researches to solve various optimization problems in production and manufacturing systems. Lutz et al. (1998) addressed a problem of buffer location and storage size in manufacturing lines and used Tabu search to find the optimal solutions. Martin et al. (1998) implemented four different variations of Tabu search to determine the number of Kanbans and lot sizes in a generic Kanban system. They reported that Tabu search performed much better than local search, but provided the same results as a simulated annealing search with longer computing times. They concluded that the Tabu search algorithm could rapidly identify optimal or near-optimal schedules for a broad range of industrial settings. Solimanpur et al. (2004) developed a Neuro-Tabu search method, and Grabowski and Wodecki (2004) proposed a fast Tabu search approach. Both of the methods were developed to solve permutation flow shop problems to minimize production makespan.

In addition to the implementation of SA, GA and TS as well as their variations for solving manufacturing system and other problems, many researchers compared the performance of two or three meta-heuristics and observed that one method might perform better than the others in certain applications. From the literature reviewed in this research, it appears that SA outperformed GA or TS in solving many manufacturing problems. SA is also simpler to implement in terms of coding efforts. We therefore chose the SA method and customized it to solve our CONWIP production line problems.

Chapter 3

Modeling a Single CONWIP Production Line

Planning and scheduling module is an important component in manufacturing planning and control framework. While the missing link between CONWIP-oriented mathematical models and operative planning systems is one reason why CONWIP control mechanism has not been widely used in the past. To this end, in this chapter, two mathematical models for a single serial CONWIP production line are developed. The mathematical models presented in this chapter aim to make the following decisions when establishing a CONWIP line:

1. Creating a part list, deciding the length of the part list and the lot size of each entry of the part list
2. Deciding the number of cards or standard containers operating in the system
3. Sequencing jobs on the part list

Depending on the production environment and the specific nature of the production line, the mathematical model may be implemented differently. In general, production systems fall into two categories: make-to-stock environment and make-to-order environment. In a make-to-order environment, it may not be required to create a part list as the part list may directly come from customer orders. While in a make-to-stock environment, the part list should be generated to meet the forecasted demands. Also in a make-to-order environment, customers usually expect their orders to be filled by certain given dates. When modelling this type of system, the objective function is usually to minimize flow time, makespan, maximum tardiness, or weighted average tardiness, etc. In addition, a production line may be unbalanced or balanced. In an unbalanced line, there will be certain types of bottleneck machines or stations. The bottlenecks could be shared bottlenecks where the bottlenecks of the system are sequence-independent; or non-shared bottlenecks where the bottlenecks of the system are sequence-dependent (Framinan et al., 2003). In a non-shared bottleneck production line, the setup times and lot sizes are important factors for job sequencing.

We first present a mathematical model for a single serial production line in a make-to-stock environment. A hybrid control policy of drum-buffer-rope (DBR) and CONWIP is employed in the model. In this model, the length of the part list, the lot sizes (the amount put into each container) of each part list entry and the sequence of the jobs are decided simultaneously. The number of containers to be used in the system then can be decided approximately by Little's law. A numerical example of this model is provided in Chapter 6 to illustrate basic characteristics of a single serial line CONWIP system. A second mathematical model will be presented for a single CONWIP serial line in a make-to-order environment. In the second model, the length of the part list is not a concern as it is decided mainly by customer orders. The lot size is also not a concern and we assume they are known. Since the lot size remains constant, the transfer lot sizes can, without loss of generality, be considered of unit size (Herer et al., 1997). In the second model, the number of containers or the WIP levels is decided in a different way from the first model. A simulated annealing based heuristic method is employed to solve the second model. The SA heuristic method will be presented in Chapter 5. The numerical examples and results of the second model will be given in Chapter 6.

3.1. Problem Description

The CONWIP system considered in this section is a single production line with a number of workstations in sequence. The workstations process a number of different types of parts specified in a part list. The part list is simply a list of parts that need to be processed. Each entry of the part list represents a standard order that has an associated part number and corresponding lot size. The order placed in the part list waits for its turn for a free container to enter the system.

The number of containers is limited or capped to limit the WIP level. Demands for finished products at the final station will cause the standard containers detached from the products and sent to the first workstation of the line. This will authorize the release of new materials to the system for processing. When a container is available, the parts associated with the order are placed in the free container. The quantity of parts placed in the container is the lot size of the parts. During a container's cycle all items in the container are identical. While proceeding through the system, the parts in the containers are processed according to a first-come-first-served discipline (Herer et al., 1997). When a container reaches the end of the line the finished parts are removed; the container is then sent back to the beginning of the line where it is again loaded with raw materials (if there are orders waiting in the part list) or it waits in a queue for another order (if the part list is empty) and the cycle repeats.

The above described process is a typical CONWIP based system depicted in Figure 3.1. Material flow is "pulled" by demands at the finished goods inventory while at each workstation inside the line it is "pushed" by upstream production.

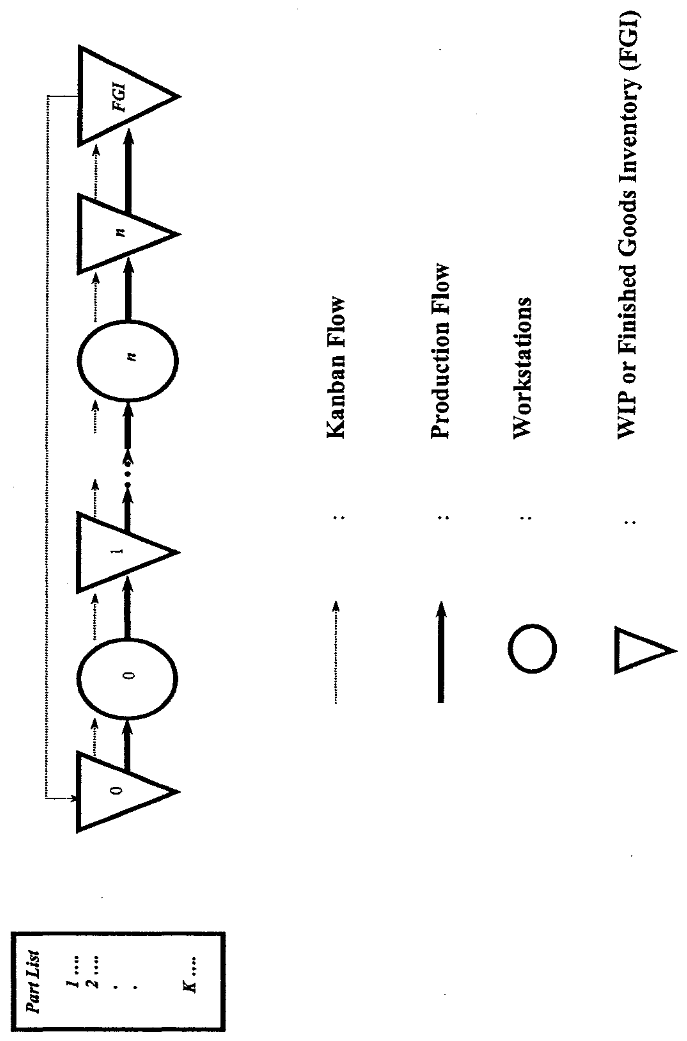


Figure 3.1: Single Serial Line CONWIP Based Production System

3.2. Mathematical Model 1 - Make-To-Stock Single Serial CONWIP Production Line

3.2.1. Assumptions

We made some assumptions when modelling the CONWIP based production line. We assumed that the production line was an unbalanced line in a make-to-stock environment. As we mentioned earlier in this chapter that for an unbalanced line there are bottlenecks. We assumed that there was only one dominant bottleneck for this production line. To model this type of production line, we followed the combined DBR (drum-buffer-rope) and CONWIP methodology.

DBR was developed by Goldratt in 1984 (Goldratt, 1984) and has been implemented by a number of manufacturing companies. It is a key component of the theory of constraints (TOC) used in ERP software for production planning. Schragenheim and Ronen (1990) stated that this approach can reduce work-in-process (WIP) and improve productivity of job shop operations.

DBR consists of three major components, drum, buffer and rope. The drum is the bottleneck machine, i.e. the constraint of the system; the constraint controls the overall pace of the system. The buffer is a protection. Buffers are used to protect the bottleneck from disruptions in the processing steps preceding the constraint. The rope is a mechanism to force all the parts of the system to work up to the pace dictated by the drum and not to exceed it. The DBR is best described as a combination push/pull logistics procedure since materials are pulled into the shop through the rope based on the rate of use of these materials at the bottleneck. Once released, materials are then pushed to subsequent work stations.

From the above discussion, we can see that although DBR and CONWIP are developed based on different reasoning, they both are push/pull hybrid systems and there are some similarities between these 2 methodologies. In our model, we will follow the drum and rope approach. In other words, a bottleneck is identified first and then a job sequence is generated to keep the bottleneck busy without interruption. At the same time, the work load on the bottleneck is balanced by using proper transfer lot size to keep the rate at the bottleneck as uniform as possible. Since the bottleneck determines the rate of the entire production line, if the rate at the bottleneck is uniform, then the rate of the entire production line is close to uniform. Last, we then can determine the WIP level based on Little's law. The WIP level of the production line is limited to this WIP calculated from Little's law. This capped WIP level instead of a buffer in the DBR is used to protect the bottleneck from disruptions in the processing steps. Since the WIP is capped in the system, this production line is still a CONWIP based production line but with DBR approach employed. This system can be viewed as a hybrid system of CONWIP and DBR.

Processing times are assumed deterministic and known for all parts. The sequence dependent setup times are also known. The parts are assumed to be processed in batches. The DBR tries to move the orders as fast as possible through the production line by splitting work orders, i.e., by transferring small batches between machines to reduce waiting time. Since the processing lot size is determined by the setup time and the transfer lot size does not have a lower limit, the transfer lot size can be reduced to increase throughput and to reduce inventory (Shtub, 1999). The lot size in our model refers to transfer lot size.

In summary, the assumptions we use in developing Model 1 are:

1. The production line is an unbalanced line and there is only one dominant bottleneck machine in the system.

2. Parts follow the same process routing in the line and are processed on each workstation sequentially.
3. Parts are produced in batches.
4. Setup times and processing times at all workstations are considered known and deterministic.
5. Setup times are sequence dependent. There is no setup incurred for the same consecutive parts. Setup time is only incurred when two consecutive entries on the part list represent different types of parts.
6. Each entry in the part list can only represent one type of parts.
7. The standard container is large enough to hold the whole lot of each part list entry (all parts in each of the part list entries).
8. There are no machine breakdowns.
9. A perfect quality conformance is assumed for the entire system. Hence, there is no defective on all workstations.
10. Demands are continuous so the parts are removed from the containers as soon as they finish processing.
11. Raw material supplies are continuous without interruption.

3.2.2. Objective Function and Constraints

Since the model developed is for a single line in a make-to-stock environment, the flow time or the make span is not our major concern. The major concern here is to minimize the overall production cost including setup cost, WIP cost, etc.

The objective function of the mathematical programming model developed in this section is to minimize setup cost at the bottleneck machine and the cost associated with the variances between consecutive batches of production in the system. The first part of the objective function, setup cost, is not uncommon in production line modeling. It is not a distinct property of the CONWIP production system since it is also widely used in MRP models. The second part of the objective function, the cost associated with the variances between consecutive batches, is a WIP related cost. In this CONWIP production line, each standard container can be seen as a standard product even though it can carry different batches of different parts. In the approach of DBR and theory of TOC, the bottleneck resource, i.e., the constraint of the system, dictates the overall pace of the system. If the bottleneck machine has a nearly constant production rate, the overall production will be close to uniform. According to Little's law, the average inventory level is equal to the average throughput rate multiplied by the average flow time. If the system throughput rate is almost constant, the WIP level, or the number of containers, should have minimum variations. This is the essential requirement for implementing a CONWIP system. So the second part of the objective function is specific to CONWIP systems. Also it is a key part to determine the lot size for each entry of the part list. To determine the transfer lot sizes (the number of parts put into each container) for each part, Spearman et al. (1990) suggests using an equal amount of work for the bottleneck machine. The amount of work is measured by both processing time and setup time.

The weighted sum of these two cost functions comprises the objective function of the model. The minimization of the objective function is subject to a number of constraint functions. We first give the notations used in these functions in the model.

Indices:

Part type index: $i = 1, 2, ..., I$

Part-list entry index: $k = 1, 2, ..., K$

Parameters:

α	-	Weighted cost associated with sequence dependent unit setup time for changing production from one product to another
β	-	Weighted cost for unbalanced workload between production batches
CAP_b	-	Production capacity of the bottleneck machine
D_i	-	Demand for part i
P_i	-	Processing time of part i on the bottleneck machine
T_{ij}	-	Sequence dependent setup time for changing production from part i to part j on the bottleneck machine, $i, j = 1, 2, ..., I$ and $i \neq j$; if $i = j, T_{ij} = 0$.

Variables:

ϵ_k	Workload difference between the k-th container and the $(k+1)$-th container, $k = 1, 2, ..., K-1$
n_{ik}	Number of standard units of parts of the k-th entry in the part list

$$x_{ik} = \begin{cases} 1, & \text{if part } i \text{ is at the } k\text{-th entry in the part-list,} \\ & i = 1, 2, ..., I, k = 1, ..., K \\ 0, & \text{otherwise} \end{cases}$$

$$y_{ijk} = \begin{cases} 1, & \text{if part } i \text{ is processed at the } k\text{-th entry in the part-list} \\ & \text{and followed by product } j, \\ & i = 1, 2, ..., I, j = 1, 2, ..., I, i \neq j, k = 1, ..., K \\ 0, & \text{otherwise} \end{cases}$$

The objective function of the mathematical programming model is to minimize the setup cost (associated with setup times) and the cost associated with the workload difference between adjacent batches at the bottleneck machine. Using the notations presented above, the objective function in our model can be expressed by:

$$\min(\sum_{k=1}^{K}(\sum_{i=1}^{I}\sum_{j=1}^{I}\alpha T_{ij}y_{ijk}+\beta\epsilon_k)) \tag{3.1}$$

The constraint functions to be satisfied in minimizing the above function are given below. The first constraint considered in this model is that the bottleneck machine should have sufficient capacity to complete all the jobs. That is, the available time must be greater than the required production time and setup times. This constraint is expressed by:

$$\sum_{k=1}^{K}\sum_{i=1}^{I}(n_{ik}x_{ik}P_i+\sum_{j=1}^{I}y_{ijk}T_{ij})\leq CAP_b \tag{3.2}$$

The next constraint is production smoothness. It is required that each time the bottleneck machine is setup, it should process the amount of work similar to that in the previous setup (Spearman et al., 1992). This is one of the key CONWIP requirements and can be expressed by:

$$|\sum_{i=1}^{I}(x_{ik}n_{ik}P_i+\sum_{j=1}^{I}y_{ijk}T_{ij})-\sum_{j=1}^{I}(x_{j,k-1}n_{j,k-1}P_j+\sum_{i=1}^{I}y_{ijk-1}T_{ij})|\leq\epsilon_k \tag{3.3}$$

Eq.(3.3) enforces that the difference of production workload between the $(k-1)$-th and the k-th entries be less than or equal to ϵ_k. If $\epsilon_k=0$, exactly the same amount of work will be processed in these two adjacent entries. Eq.(3.3) can be replaced by Eqs.(3.3.1) and (3.3.2) after the absolute value sign is removed:

$$\sum_{i=1}^{I}(x_{ik}n_{ik}P_i + \sum_{j=1}^{I} y_{ijk}T_{ij}) - \sum_{j=1}^{I}(x_{j,k-1}n_{j,k-1}P_j + \sum_{i=1}^{I} y_{ijk-1}T_{ij}) \leq \epsilon_k \tag{3.3.1}$$

$$\sum_{i=1}^{I}(x_{ik}n_{ik}P_i + \sum_{j=1}^{I} y_{ijk}T_{ij}) - \sum_{j=1}^{I}(x_{j,k-1}n_{j,k-1}P_j + \sum_{i=1}^{I} y_{ijk-1}T_{ij}) \geq -\epsilon_k \tag{3.3.2}$$

In solving the CONWIP production control problem, we require that all product demands be satisfied. This is expressed by:

$$\sum_{k=1}^{K} x_{ik}n_{ik} = D_i \tag{3.4}$$

Eqs.(3.2), (3.3.1), (3.3.2) and (3.4) are non-linear functions and have the same non-linear terms. In fact, these non-linear terms can easily be linearized since x_{ik} is a binary integer variable. Define a new integer variable $w_{ik} \geq 0$ and let:

$$w_{ik} = x_{ik}n_{ik}$$

From the definition of x_{ik} and n_{ik}, we require that:

$$w_{ik} = \begin{cases} n_{ik}, & \text{if } x_{ik} = 1 \\ 0, & \text{if } x_{ik} = 0 \end{cases}$$

Such requirement can be realized by enforcing the following three linear inequalities simultaneously:

$$w_{ik} \geq Mx_{ik} + n_{ik} - M \tag{3.4.1}$$

$$w_{ik} \leq n_{ik} \tag{3.4.2}$$

$$w_{ik} \leq Mx_{ik} \tag{3.4.3}$$

where M is a large positive number. When $x_{ik} = 1$, Eqs.(3.4.1) and (3.4.2) enforce $w_{ik} = n_{ik}$ while Eq.(3.4.3) has no effect on the value of w_{ik}; when $x_{ik} = 0$, Eqs.(3.4.1) and (3.4.2) have no effect on the value of w_{ik} while Eq.(3.4.2) enforces $w_{ik} = 0$. Non-linear terms in Eqs. (3.2), (3.3.1), (3.3.2) and (3.4), after linearized, will be replaced by w_{ik} with Eqs.(3.4.1), (3.4.2) and (3.4.3) enforced in the model. In solving our problem, we also require that each entry in the part list is assigned one and only one type of product to process. This is expressed by:

$$\sum_{i=1}^{I} x_{ik} = 1 \tag{3.5}$$

The next constraint enforces that if the $(k-1)$-th entry in the part list is product i while the kth entry is product j, then a setup is required for changing production from i to j. This relation is given by:

$$y_{ijk} = x_{i,k-1}x_{jk}, i \neq j \tag{3.6}$$

This nonlinear function can also be linearized. The standard method to linearize such nonlinear term is to substitute Eq.(3.6) by the following two inequalities (Herer et al. 1997):

$$y_{ijk} - x_{i,k-1} - x_{jk} + 1.5 \geq 0 \tag{3.6.1}$$

$$1.5y_{ijk} - x_{i,k-1} - x_{jk} \leq 0 \tag{3.6.2}$$

Eqs.(3.6.1) and (3.6.2) are linear and serve the same function as Eq.(3.6). After the non-linear constraint functions are linearized, the CONWIP production control problem can be expressed by the following integer programming model:

CONL1:

$$\min(\sum_{k=1}^{K}(\sum_{i=1}^{I}\sum_{j=1}^{I}\alpha T_{ij}y_{ijk}+\sum_{k=1}^{K}\beta\epsilon_k))$$

Subject to

$$\sum_{k=1}^{K}\sum_{i=1}^{I}(w_{ik}P_i+\sum_{j=1}^{I}y_{ijk}T_{ij})\le CAP_b$$

$$\sum_{i=1}^{I}(w_{ik}P_i+\sum_{j=1}^{I}y_{ijk}T_{ij})-\sum_{j=1}^{I}(w_{j,k-1}P_j+\sum_{i=1}^{I}y_{ijk-1}T_{ij})\le\epsilon_k, i=1,...I;j=1,...,I;k=2,...,K$$

$$\sum_{i=1}^{I}(w_{ik}P_i+\sum_{j=1}^{I}y_{ijk}T_{ij})-\sum_{j=1}^{I}(w_{j,k-1}P_j+\sum_{i=1}^{I}y_{ijk-1}T_{ij})\ge-\epsilon_k, i=1,...I;j=1,...,I;k=2,...,K$$

$$\sum_{k=1}^{K}w_{ik}=D_i, i=1,...,I$$

$$w_{ik}\ge Mx_{ik}+n_{ik}-M, i=1,...,I,k=1,...,K$$

$$w_{ik}\le n_{ik}, i=1,...,I,k=1,...,K$$

$$w_{ik}\le Mx_{ik}, i=1,...,I,k=1,...,K$$

$$\sum_{i=1}^{I}x_{ik}=1, k=1,...,K$$

$$y_{ijk}-x_{i,k-1}-x_{jk}+1.5\ge0, i=1,...,I;j=1,...,I,i\ne j;k=2,...,K$$

$$1.5y_{ijk}-x_{i,k-1}-x_{jk}\le0, i=1,...,I;j=1,...,I,i\ne j;k=2,...,K$$

$$x_{i,k}, y_{ijk} = 0, 1, n_{ik}, w_{ik} \geq 0 \text{ are integers.}$$

The integer programming model CONL1 can be solved directly using available computer software such as CPLEX or LINDO. In this research, we solved a small size example problem for testing purposes using LINGO, a version of LINDO (LINDO Systems Inc., 1995) software. The testing problem and its solutions in different scenarios are presented and explained in Chapter 6.

3.2.3. Features of the Model

In a CONWIP single serial line, the bottleneck machine largely determines both mean and variance of the cycle time and hence the output characteristics of the line. Consequently, in a CONWIP line with multiple products, we measure the WIP contribution of each job with respect to a standard part defined in terms of time at the bottleneck (Spearman et al., 1989). Our model is unique in a way that it simplified the serial CONWIP production line problem by focusing on the bottleneck machine. Since the capacity of the bottleneck machine determines the throughput of the line and also the WIP level, by minimizing the setup costs and the costs associated with unbalanced workload at the bottleneck machine, we maximized the utilization of the bottleneck and therefore increased the throughput. With an increased throughput for a given WIP level, the flow time is minimized. While with a given flow time, when throughput is maximized, a minimal WIP level is reached. The solution of the model simultaneously determines the optimal job sequence on the part list and the lot size associated with each entry on the part list. Then the number of containers circulating in the system can be determined by:

$$N = T_m/T_b$$

$$T_m = (\sum_{m=1}^{M}\sum_{k=1}^{K}\sum_{i=1}^{I}(n_{ik}x_{ik}P_{im} + \sum_{j=1}^{I} y_{ijk}T_{ijm}))/K$$

$$T_b = (\sum_{k=1}^{K}\sum_{i=1}^{I}(n_{ik}x_{ik}P_{ib} + \sum_{j=1}^{I} y_{ijk}T_{ijb}))/K$$

where N is the number of the containers; M is the total number of machines in the production line; T_m is the average flow time for the parts in a standard container; and T_b is the average time taken on the bottleneck machine by parts in a standard container. The good feature of the model is that it can be used to determine all the following CONWIP parameters and no similar model was found in the existing literature: 1. the length of the part list; 2. the transfer lot size for the container; 3. the sequence of the parts; and 4. the number of the containers in the system.

3.3. Mathematical Model 2 - Make-To-Order Single CONWIP Production Line

In section 3.2, we presented a model to simultaneously determine major parameters of a CONWIP single serial production line in a make-to-stock environment where the cost is a major concern. For a make-to-stock environment, the part list forecasting is essential to a successful CONWIP implementation.

Model 1 has its distinct features as mentioned in the previous section, but it also has the following limitations:

1. The bottleneck machine has to be identified in advance before using Model 1. However, in actual production, the bottleneck machine may change from one machine to another. Even though the capped WIP level in our model will moderate the effects of the shifting bottleneck, it still has some negative effects on the system performance.

2. The WIP level is not computed directly by Model 1. It is handled separately from the model.

To address the above 2 issues related to the Model 1, in this section, we will develop a second model that does not need to identify a bottleneck machine first and also the WIP level can be handled directly by the model. The new model can be used in a make-to-order environment or in a make-to-stock environment with a known part list. In this model, our main concern is the minimum makespan of the production. Since the part list is already known or we can directly use customer orders to form the part list, no part list forecasting is considered in this model. The task of this model is to find the best job sequence and the WIP level at the same time to meet the customers' demand.

3.3.1. Assumptions

The assumptions we use in developing the new model, CONL2, are given as follows:

1. Parts follow the same process routing in the line and are processed on each workstation sequentially.

2. The lot size is assumed as 1 for all parts.

3. Setup times and processing times at all workstations are considered known and deterministic.

4. Setup times are not sequence dependent.

5. Setup times are included in the processing times.

6. No machine breakdowns.

7. A perfect quality conformance is assumed for the entire system. Hence, there is no defective on all workstations.

8. Demands are continuous so the containers are removed from parts as soon as they finish being processed.

9. Raw material supplies are continuous so production does not stop to wait for supplies.

3.3.2. Objective Function and Constraints

The objective function of the mathematical programming model developed in this section is to minimize the makespan of production while at the same time the WIP level and job sequence are determined.

Indices:

Part type index: $i = 1, 2, ..., I$

Part-list entry index: $k = 1, 2, ..., K$

Machine index: $m = 1, 2, ..., M$

Parameters:

$t_{k_i m}$	-	Processing time of part i in part-list entry k_i on machine m, $i = 1, 2, ..., I; k_i = 1, 2, ..., K; m = 1, 2, ..., M$
M_1	-	The first machine in the production line

M_L	-	The last machine in the production line
G	-	A large positive number
W	-	The desired WIP level

Variables:

$$\delta_{k_ik_j} = \begin{cases} 1, & \text{if entry } k_i \text{ enters the line while entry } k_j \text{ is still in process} \\ & k_i = 1,2,...,K; k_j = 1,2,...,K; k_i \neq k_j \\ 0, & \text{otherwise} \end{cases}$$

w_{k_i}	Number of loaded containers in the system when k_i enters the system
w	Total number of containers in the system
b_{k_im}	Starting time of k_i representing part i on machine m
f_{k_im}	Finishing time of k_i representing part i on machine m
f_{max}	Makespan (overall completion time of all parts)

The goal is to minimize the makespan with low WIP. Using the notations presented above, the objective function in our model can be expressed by:

$$\min f_{max} \tag{3.7}$$

The constraint functions to be satisfied in minimizing the above function are given below.

1. Processing time constraints

$$f_{k_im} = b_{k_im} + t_{k_im} \tag{3.8}$$

The finishing time of entry k_i representing part i on machine m is equal to its starting time on machine m plus its required processing time on machine.

2. Job sequence constraints

$$b_{1M_1} = 0 \tag{3.9}$$

$$b_{k_iM_1} \geq \begin{cases} f_{(k_i-1)M_1} + 1, & \text{if container is available} \\ \max(f_{(k_i-w)M_L}, f_{(k_i-1)M_1}) + 1, & \text{if container is unavailable} \\ & k_i = 2, ..., K \end{cases} \tag{3.10}$$

The first entry 1 in the part list starts its operation on the first machine M_1 at time 0.

Starting time of entry k_i other than entry 1 on the first machine of the line depends on the availability of the containers. As we mentioned earlier, the part is allowed to enter the production line only when the container is available. If a container is available, the part will enter the line immediately after the previous entry finishes its operation on the first machine. If there is no container available, and even the first machine is idle, the entry cannot enter the line. It will have to wait until a container is freed from the entry that entered the line prior to it.

$$b_{k_im} \geq \max(f_{k_i(m-1)}, f_{(k_i-1)m}) + 1, m = 2, ..., M; k_i = 2, ..., K \tag{3.11}$$

Entry $k_i(k_i \neq 1)$ representing part i can only start on machine $m(m \neq 1)$ until it finishes its proceeding operations on the previous machine, $m-1$, and the entry prior to it finishes the operations on machine m.

$$\delta_{k_ik_j} = \begin{cases} 1, & b_{k_jM_1} < b_{k_iM_1} < f_{k_jM_L} \\ & k_i \neq k_j; k_i = 1, 2, ..., K; k_j = 1, 2, ..., K \\ 0, & \text{otherwise} \end{cases} \tag{3.12}$$

This constraint is a 0-1 indictor. When entry k_i representing part i enters the system, if entry k_j representing part j is still in process, the indictor is 1; otherwise it equals to 0.

3. WIP constraints

The next two constraints ensure that there is a constant number of containers or a constant WIP level in the system and the number of the loaded containers should be always less than the total number of containers in the system.

$$w_{k_i} = \sum_{k_j=1}^{K} \delta_{k_i k_j}, k_i \neq k_j; k_i, k_j = 1, 2, ..., K \tag{3.13}$$

$$w \geq w_{k_i}, k_i = 1, 2, ..., K \tag{3.14}$$

4. Makespan constraint

The maximum makespan should be greater than the completion time of each entry.

$$f_{k_i M_L} \leq f_{max}, k_i = 1, 2, ..., K \tag{3.15}$$

Put the objective function and all above constraints together, the second CONWIP model, CONL2, can be expressed as follows:

CONL2:

$$min f_{max}$$

Subject to:

$$f_{k_i m} = b_{k_i m} + t_{k_i m}, k_i = 1, 2, ..., K; m = 1, 2, ..., M$$

$$b_{1M_1} = 0$$

$$b_{k_iM_1} \geq \begin{cases} f_{(k_i-1)M_1} + 1, & \text{if container is available} \\ \max(f_{(k_i-w)M_L}, f_{(k_i-1)M_1}) + 1, & \text{if container is unavailable} \\ & k_i = 2, ..., K \end{cases}$$

$$b_{k_im} \geq \max(f_{k_i(m-1)}, f_{(k_i-1)m}) + 1, m = 2, ..., M; k_i = 2, ..., K$$

$$\delta_{k_ik_j} = \begin{cases} 1, & b_{k_jM_1} < b_{k_iM_1} < f_{k_jM_L} \\ & k_i = 1, 2, ..., K; k_j = 1, 2, ..., K; k_i \neq k_j \\ 0, & \text{otherwise} \end{cases}$$

$$w_{k_i} = \sum_{k_j=1}^{K} \delta_{k_ik_j}, k_i \neq k_j; k_i, k_j = 1, 2, ..., K$$

$$w \geq w_{k_i}, k_i = 1, 2, ..., K$$

$$f_{k_iM_L} \leq f_{max}, k_i = 1, 2, ..., K$$

The above model is a mixed integer programming problem. Even though the model assumes deterministic processing and setup times, due to a large number of integer variables, the model cannot be solved within an acceptable amount of time for a large problem using general optimization algorithm. Thus, solutions must be found via some heuristic mechanism if large size problems are to be solved. In order to solve the problem efficiently, after comparing different heuristics, a simulated annealing based heuristic algorithm was chosen in this research to solve the proposed model. Details of the simulated annealing heuristic algorithm are given in Chapter 5. The algorithm is coded in C# which is presented in Appendix C. The C# code is then used to solve the model CONL2 in a PC (Centrino Due 1.83G). The results of several numerical example problems using model CONL2 will be presented in Chapter 6.

Chapter 4

Modelling Multiple CONWIP Production Lines

Assembly-type production systems are prevalent in many manufacturing systems. They are characterized as several components are produced in separate fabrication lines and then assembled together. Despite their prevalence in many manufacturing environments, limited research has been done on these systems for different production control policies, especially CONWIP. To this end, this research aims at addressing issues raised in implementing CONWIP in assembly-type production systems.

4.1. Problem Description

A typical assembly-type production system considered in this research is shown in Figure 4.1. Figure 4.1 shows a production system where several CONWIP controlled fabrication lines feed an assembly station. In the above production system, a finished product is assembled at the assembly station with a number of different types of parts fabricated by the fabrication lines. The assembly process can start only after all the required parts from all fabrication lines have been processed and are ready for assembling. A CONWIP control mechanism to control such type of system is presented below.

Arriving demand or new order generates a request for an assembled product and the request goes to the assembled product buffer. If the inventory level of the assembled product buffer is not empty, then an assembled product is released to satisfy the demand; otherwise, a request is sent to the processed product buffers that are located at the end of each fabrication line. If the processed buffers are not empty and all required parts for assembling on each of the fabrication lines are ready, the required parts from each of the processed buffers are matched and then released to satisfy the assembly request. Otherwise, a request is sent to the front of all fabrication lines where it goes into a CONWIP control module. The CONWIP control module determines the number of containers and the job sequence for each fabrication line. The numbers of containers obtained from the CONWIP control module are then assigned to each of the fabrication lines. The job sequence determined by the CONWIP control module is used to form a part list for each fabrication line. The part list is a list of jobs that need to be processed. The order of the jobs on the part list is the same as the job sequence determined by the CONWIP control module. Each fabrication line has its own part list. The part list of each fabrication line consists of several entries and each entry corresponds to one type of part on the line. When a container is available and the part list is not empty, raw material of the job on the part list is put into an available container to enter the line for processing. The job on the part list can enter the line only when there is free container. Once the job is released to the first machine in each of the fabrication lines, it is pushed through the system. Each time a job finishes all its processes and reaches the end of the fabrication line, its container is removed and sent to the front of the line where it is again loaded with raw material if there are jobs waiting in the part list or it waits in a queue for another order if the part list is empty. The number of containers in each fabrication line is limited, therefore limiting the WIP inventory level for each line. Since the WIP inventory level of each fabrication line is controlled and kept as constant, the entire assembly-type production system WIP level is constant. The above described control mechanism combines both pull/push controls and has a constant WIP level, therefore it is a typical CONWIP control mechanism.

As described above, all required parts from each fabrication line are matched before sending to the assembly station for processing. If the required parts cannot be matched because some required parts from some fabrication lines are not available, the assembly process is delayed. Thus, to reduce system makespan and reduce overall system WIP inventory cost, synchronization of the fabrication lines to assembly is crucial. In the CONWIP controlled system, the synchronization of fabrication lines is maintained by properly sequencing the jobs on the part list and controlling the WIP inventory level for each of the fabrication lines.

As we mentioned before, the WIP inventory level of each fabrication line is determined by the number of containers on each line. To determine the number of the containers for each line, the release mechanism has to be decided first. There are two possible release mechanisms for a CONWIP assembly-type production system. One is that each line has the same number of containers which is determined by the critical line. The other one is that each line can have their own number of containers, or, their own WIP inventory levels. If the first mechanism is followed, there is only one set of cards to decide and maintain. Once the number of containers is decided based on the critical line, it then applies to all other lines. This is easier to implement to control a CONWIP assembly-type production system. The second mechanism is more difficult to implement as there are several sets of containers whose numbers are to be decided. Synchronization among all fabrication lines is important to reduce the WIP inventory levels of the entire system. Because parts finished in the faster lines always have to be put into inventory in front of the assembly station waiting for the parts from the slower lines to start assembling, then using the first mechanism, the WIP inventory levels in the faster lines would be higher than those if the second mechanism is used. In our research, we will focus on the second release mechanism, i.e., we allow each line to have its own WIP inventory levels. In summary, the problem considered in this chapter is an assembly station fed by multiple fabrication lines. Each of the fabrication lines processes multiple products by multiple machines. Different sets of containers are used in different lines to control the WIP inventory levels and thus each line maintains its own job sequence on its own part list.

For the above given system, the WIP inventory level and the job sequence of each fabrication line are two essential parameters. Since these two parameters are closely related, we include them in one module which is called the CONWIP control module. The CONWIP control module should determine the WIP inventory level and the job sequence for each line. This research aims at developing a mathematical programming model that can be used in the CONWIP control module to simultaneously determine the WIP inventory level and the job sequence for each fabrication line. The mathematical model developed for the system is given in the next section.

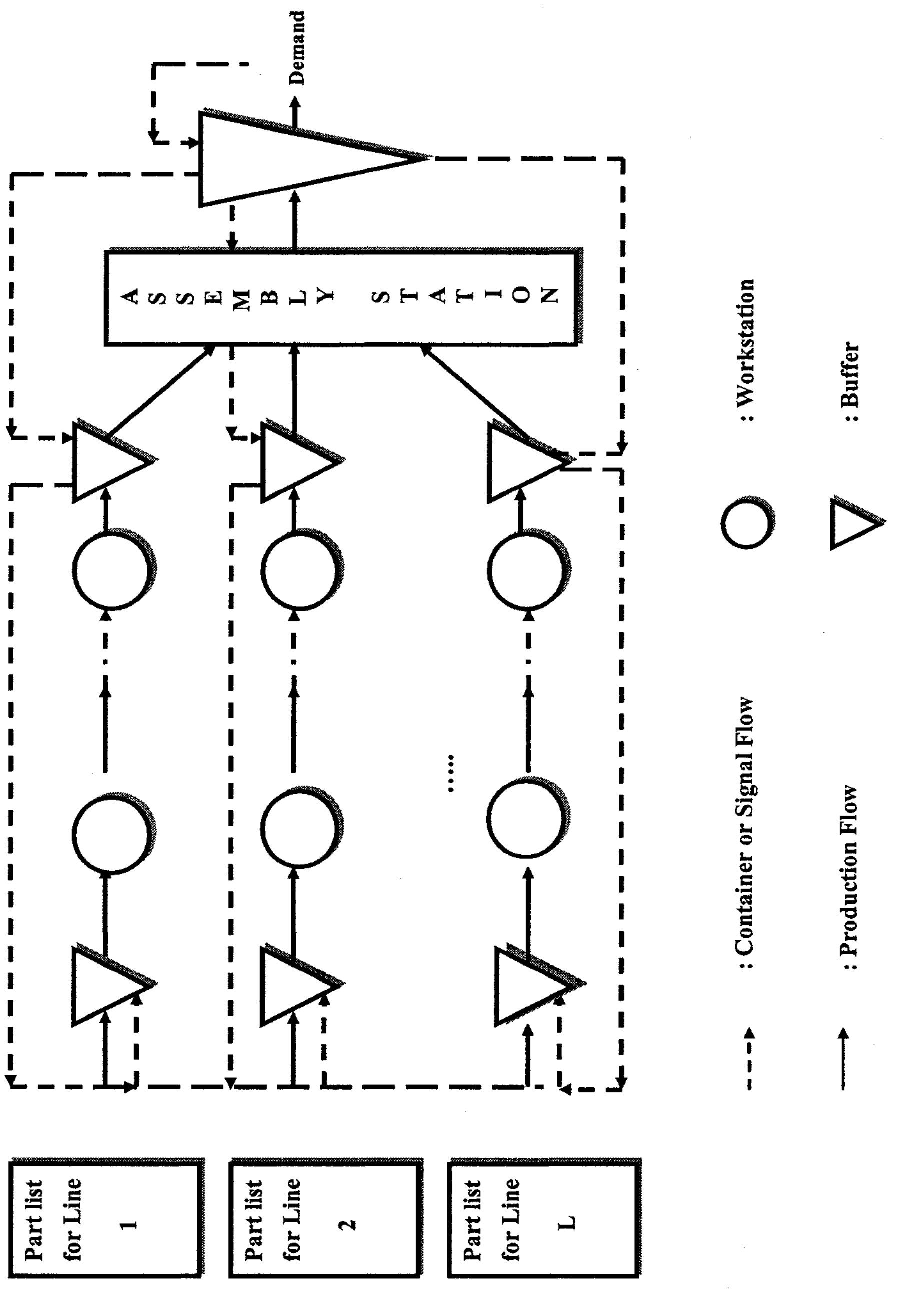


Figure 4.1: Multi-Line CONWIP Based Fabrication-Assembly System

4.2. Assumptions

The following assumptions are made in developing the mathematical programming model for the given problem:

1. An assembly station is fed by multiple fabrication lines and each line has multiple workstations to process multiple products.
2. Parts follow the same process routing in each line and are processed on each workstation sequentially.
3. A perfect quality conformance is assumed for the entire system. Hence, there is no defective at any workstation.
4. The lot size is assumed to be 1 for all parts.
5. Setup times and processing times at all workstations are considered known and deterministic.
6. Setup times are not sequence dependent.
7. Setup times are included in the processing times.
8. There are no machine breakdowns.
9. Demands are continuous so the parts are removed from containers as soon as they finish processing.
10. Raw material supplies are continuous so production will not stop to wait for supplies.

4.3. Model Development for a CONWIP system with Multiple Lines

In this section, we formulate a mixed-integer programming model for solving the problem of simultaneously finding optimal job sequence, optimal WIP inventory level for each fabrication line in a multiple line, multiple stage assembly-type CONWIP system. As discussed earlier, the second CONWIP release mechanism is used in the modeling. The following notations are used in this model:

Indices:

Part type index: $li = 1, 2, ..., I_l$

Part-list entry index: $kl = 1, 2, ..., K_l$

Machine index: $m_l = 1, 2, ..., M_l$

Fabrication line index: $l = 1, 2, ..., L$

Parameters:

$t_{k_{li}m_l l}$	-	Processing time of part i in part list entry k_{li} on machine m_l of line l, $k_{li} = 1, 2, ..., K_l; m_l = 1, 2, ..., M_l; l = 1, 2, ..., L$
M_{l1}	-	The first machine in the production line l
M_{lL}	-	The last machine in the production line l
G	-	A large positive number
W_l	-	Desired WIP inventory level for line l

Variables:

$$\delta_{k_{li}k_{lj}l} = \begin{cases} 1, & \text{if entry } k_{li} \text{ enters line } l \text{ while entry } k_{lj} \text{ is still in process} \\ & k_{li} = 1, ..., K_l; k_{lj} = 1, ..., K_l; k_{li} \neq k_{lj}; l = 1, ..., L \\ 0, & \text{otherwise} \end{cases}$$

$w_{k_{li}l}$	Number of loaded containers in line l when k_{li} enters line l
w_l	Total number of containers in line l
$b_{k_{li}m_l l}$	Starting time of k_{li} representing part li on machine m_l of line l
$f_{k_{li}m_l l}$	Finishing time of k_{li} representing part li on machine m_l of line l
f_{max}	Maximum overall completion time of all parts in all lines
f_{min}	Minimum overall completion time of all parts in all lines

By putting different number of containers in different fabrication lines to coordinate the finishing time for each of the fabrication lines, the WIP inventory level of the whole system is reduced. Thus, the objective function of the model is to minimize the differences of the finishing times of all the fabrication lines.

$$\min(f_{max} - f_{min}) \tag{4.1}$$

The constraints to be satisfied in minimizing the above objective function are given below.

1. Processing time constraints

$$f_{k_{li}m_l l} = b_{k_{li}m_l l} + t_{k_{li}m_l l} \tag{4.2}$$

$$k_{li} = 1, ..., K_l; m_l = 1, ..., M_l; l = 1, ..., L$$

The finishing time of entry k_{li} representing part li of line l on machine m_l is equal to its starting time on machine m_l plus its required processing time on machine m_l of line l.

2. Job sequence constraints

$$b_{l1M_{l1}l} = 0 \tag{4.3}$$

$$l = 1, ..., L$$

$$b_{k_{li}M_{l1}l} \geq \begin{cases} f_{(k_{li}-1)M_{l1}l} + 1, & \text{if container is available} \\ \max(f_{(k_{li}-w_l)M_{lL}l}, f_{(k_{li}-1)M_{l1}l}) + 1, & \text{if container is unavailable} \\ k_{li} = 2, ..., K_l; l = 1, ..., L \end{cases} \tag{4.4}$$

The first entry 1 in the part list of line l starts its operations on machine 1 of line l at time 0.

The starting times of the other entries k_{li} on the first machine of the line depend on the availability of the containers. As we mentioned earlier, the part is allowed to enter the fabrication line only when a container is available. If there is a container available, the entry k_{li} representing part li will be entering the line immediately after the previous entry finishes its operations on the first machine. If there is no container available, even the first machine is idle, the entry still cannot enter the line. It will wait until the container is freed from other entries that entered the line prior to it.

$$b_{k_{li}m_l l} \geq \max(f_{k_{li}(m_l-1)l}, f_{(k_{li}-1)m_l l}) + 1 \tag{4.5}$$

$$k_{li} = 2, ..., K_l; m_l = 2, ..., M_l; l = 1, ..., L$$

Entry $k_{li}(k_{li} \neq 1)$ can start on machine $m_l(m_l \neq 1)$ only after it finishes its proceeding operations on the previous machine $m_l - 1$ and the entry prior to it finishes the operations on machine m_l.

$$\delta_{k_{li}k_{lj}l} = \begin{cases} 1, & b_{k_{lj}M_{l1}l} < b_{k_{li}M_{l1}l} < f_{k_{lj}M_{lL}l} \\ & k_{li} \neq k_{lj}; k_{li}, k_{lj} = 1, ..., K_l, l = 1, ..., L \\ 0, & \text{otherwise} \end{cases} \tag{4.6}$$

This constraint is a 0-1 indictor. When the entry k_{li} representing part li on line l enters the system, if the entry representing part j on line l is still in process, the indictor is 1; otherwise it equals to 0.

3. WIP constraints

The next two constraints ensure that there is a constant number of containers in the CONWIP production system. Alternatively, the WIP inventory level of each fabrication line and the number of loaded containers should be always smaller than the total number of containers in the line.

$$w_{k_{li}l} = \sum_{k_{lj}=1}^{K_l} \delta_{k_{li}k_{lj}l}, k_{li} \neq k_{lj}, k_{li}, k_{lj} = 1, ..., K_l; l = 1, ..., L \tag{4.7}$$

$$w_l \geq w_{k_{li}l}, k_{li} = 1, ..., K_l; l = 1, ..., L \tag{4.8}$$

4. Completion time constraint

This constraint measures the overall parts completion time differences among the fabrication lines.

$$f_{min} \leq f_{k_{li}M_{lL}l} \leq f_{max}, k_{li} = 1, ..., K_l; l = 1, ..., L \tag{4.9}$$

Put the objective function and all above constraints together, the model can be expressed as follows:

MCONL:

$$min(f_{max} - f_{min})$$

Subject to:

$$f_{k_{li}m_l l} = b_{k_{li}m_l l} + t_{k_{li}m_l l}$$

$$k_{li} = 1, ..., K_l; m_l = 1, ..., M_l; l = 1, ..., L$$

$$b_{1M_{l1}l} = 0, l = 1, ..., L$$

$$b_{k_{li}M_{l1}l} \geq \begin{cases} f_{(k_{li}-1)M_{l1}l} + 1, & \text{if container is available} \\ \max(f_{(k_{li}-w_l)M_{lL}l}, f_{(k_{li}-1)M_{l1}l}) + 1, & \text{if container is unavailable} \\ k_{li} = 2, ..., K_l; l = 1, ..., L & \end{cases}$$

$$b_{k_{li}m_l l} \geq \max(f_{k_{li}(m_l-1)l}, f_{(k_{li}-1)m_l l}) + 1$$

$$k_{li} = 2, ..., K_l; m_l = 2, ..., M_l; l = 1, ..., L$$

$$\delta_{k_{li}k_{lj}l} = \begin{cases} 1, & b_{k_{lj}M_{l1}l} < b_{k_{li}M_{l1}l} < f_{k_{lj}M_{lL}l} \\ & k_{li} \neq k_{lj}; k_{li}, k_{lj} = 1, ..., K_l, l = 1, ..., L \\ 0, & \text{otherwise} \end{cases}$$

$$w_{k_{li}l} = \sum_{k_{lj}=1}^{K_l} \delta_{k_{li}k_{lj}l}, k_{li} \neq k_{lj}, k_{li}, k_{lj} = 1, ..., K_l; l = 1, ..., L$$

$$w_l \geq w_{k_{li}l}, k_{li} = 1, ..., K_l; l = 1, ..., L$$

$$f_{min} \leq f_{k_{li}M_{lL}l} \leq f_{max}, k_{li} = 1, ..., K_l; l = 1, ..., L$$

The model given above is a generic model for an assembly-type CONWIP production system with multiple fabrication lines. From the review of recently published literatures, we did not find any research done using a mathematical programming model to solve the assembly-type multiple CONWIP line problems of this type.

The model is coded in C# and solved by simulated annealing (SA) based heuristic algorithm on a PC (Intel Centrino duo CPU 1.83GHz). Detailed discussion on the SA heuristic is presented in Chapter 5. Several numerical example problems to illustrate the model MCONL above are presented in Chapter 6.

Chapter 5

Simulated Annealing Based Heuristic Algorithm

Simulated annealing heuristics have been used to find optimal or near-optimal solutions to combinatorial optimization problems. A SA algorithm attempts to overcome the inherent difficulties encountered by various descent algorithms that stop the search once a local optimum is obtained. A distinct characteristic of SA is that it sometimes replaces a current solution by an inferior solution to avoid getting stuck at local optima. SA has been successfully implemented to solve a variety of difficult problems, especially sequencing jobs in flow-shop production systems (Golany et al., 1999).

Simulated annealing is a general optimization method that stochastically simulates the slow cooling process of a physical system. The basic idea behind simulated annealing is that there is a cost function F that associates a cost with a state of the system, a "temperature" T, and various ways to change the state of the system. The simulated annealing algorithm works by iteratively proposing changes or configurations and either accepting or rejecting each change or configuration. For each proposed change or configuration, we may evaluate the change ΔF in cost function F. The proposed configuration may be accepted or rejected by the so called Metropolis criterion. The Metropolis criterion can be described as follows. If the cost function decreases, the change or configuration is accepted unconditionally; otherwise, it is accepted only with probability $exp(-\Delta F/K_bT)$. A random number chosen from A uniform distribution (0,1) is used to make a comparison with the probability $exp(-\Delta F/K_bT)$. If it is less than $exp(-\Delta F/K_bT)$, the change or the configuration is accepted; otherwise, it is rejected. A population of configurations of a given optimization problem are generated by a generation mechanism at certain temperatures of cooling. These temperatures are control parameters. The Simulated Annealing process consists of first "melting" the system being optimized at a high effective temperature, then lowering the temperature by slow paces until the system "freezes" and no further changes occur. At each temperature level, the simulation must proceed long enough for a system to reach a steady state. The sequence of temperatures and the number of rearrangements of the parameters attempted to reach equilibrium at each temperature can be considered an annealing schedule (Kirkpatrick et al., 1983). In general, the SA algorithm can be described as follows:

1. Initializing all the SA parameters.

2. Selecting a generation mechanism: generate a new solution from the current solution by a small perturbation.

3. Evaluating the new solution: computing the difference between the current solution and the new solution.

4. If the new solution is better than the current solution, make it the current solution; otherwise, use the Metropolis criterion to decide if the new solution

is accepted or rejected.

5. Stop the procedure based on the given stop criterion.

To implement the SA algorithm, a perturbation scheme and cooling schedule are essential.

- **Perturbation Scheme**

  The SA algorithm is an iterative search procedure based on a neighborhood structure. The quality of the annealing solution is sensitive to the way the candidate solutions are selected (Kirkpatrick et al., 1983). So the generation mechanism, which generates a new solution from the current one, often called perturbation scheme, is required to be determined prior to implementing a SA based algorithm. We use six perturbation schemes in our proposed SA based heuristic algorithm. The six perturbation schemes are:

  1. Interchanging two adjacent jobs (adjacent interchange):

     P1 P2 $\underline{\text{P3}}$ $\underline{\text{P4}}$ P5 P6

     P1 P2 $\underline{\text{P4}}$ $\underline{\text{P3}}$ P5 P6

     Algorithm:

     Step1: Randomly select an integer $i \leq n$

     Step2: $P_{new}(i+1) = P(i); P_{new}(i) = P(i+1)$

  2. Interchanging two jobs that are randomly selected (random interchange or pair wise exchange):

     P1 P2 $\underline{\text{P3}}$ P4 P5 $\underline{\text{P6}}$

     P1 P2 $\underline{\text{P6}}$ P4 P5 $\underline{\text{P3}}$

     Algorithm:

Step1: Randomly select an integer $i \leq n$

Step2: Random select an integer $j \leq n$

Step3: If $P(i) \neq P(j)$, go to Step4; otherwise, return to step2.

Step4: $P_{new}(j) = P(i); P_{new}(i) = P(j)$

3. Inserting a randomly selected job to a randomly selected position (inserting scheme):

   P1 P2 P3 P4 <u>P5</u> P6

   P1 <u>P5</u> P2 P3 P4 P6

   Algorithm:

   Step1: Randomly select an integer $i \leq n$

   Step2: Random select an integer $j \leq n$

   Step3: If $P(i) \neq P(j)$, go to Step4; otherwise, return to step2.

   Step4: Insert $P(i)$ at position $j+1$

4. Inserting a randomly selected subsequence of jobs to a randomly selected position

   P1 P2 <u>P3</u> <u>P4</u> <u>P5</u> P6

   P1 <u>P3</u> <u>P4</u> <u>P5</u> P2 P6

   Algorithm:

   Step1: Randomly select an integer $i \leq n$

   Step2: Random select an integer $j \leq n$

   Step3: If $P(i) \neq P(j)$, go to Step4; otherwise, return to step2.

   Step4: Randomly select an integer k in n excluding i to j

   Step5: Insert $P(i)$ to $P(j)$ at position $k+1$

5. Reversing a randomly selected subsequence of jobs

P1 P2 <u>P3</u> <u>P4</u> <u>P5</u> P6

P1 P2 <u>P5</u> <u>P4</u> <u>P3</u> P6

Algorithm:

Step1: Randomly select an integer $i \leq n$

Step2: Random select an integer $j \leq n$

Step3: If $P(i) \neq P(j)$, go to Step4; otherwise, return to step2.

Step4: Reverse job sequence from $P(i)$ to $P(j)$

6. Reversing a randomly selected subsequence of jobs and then inserting the entire selected jobs to a randomly selected position

   P1 P2 <u>P3</u> <u>P4</u> <u>P5</u> P6

   P1 <u>P5</u> <u>P4</u> <u>P3</u> P2 P6

   Algorithm:

   Step1: Randomly select an integer $i \leq n$

   Step2: Random select an integer $j \leq n$

   Step3: If $P(i) \neq P(j)$, go to Step4; otherwise, return to step2.

   Step4: Reverse job sequence from $P(i)$ to $P(j)$

   Step5: Randomly select an integer k in n excluding i to j

   Step6: Insert $P(j)$ to $P(i)$ at position $k+1$

An appropriate perturbation scheme leads to a good performance of the SA algorithm. According to the research done on SA, it appears that the perturbation scheme is problem related, i.e., one perturbation scheme may be the best for one type of the problem, but may be the worst for another type of the problem. Tian (1999) found that for their given flow shop scheduling problem, scheme number 4 is the most efficient scheme. However, they stated that the generation mechanisms currently adopted by the flow shop scheduling problem are mainly schemes 1, 2 and 3.

In order to find a proper perturbation scheme for solving the proposed CONWIP line models, all above six schemes are used in our proposed SA based heuristic algorithm.

- **Cooling schedule**

  A cooling schedule or sometimes called annealing schedule refers to a temperature control scheme that determines the solving movement permitted during the search, and therefore it is critical to the SA algorithm's performance. The cooling schedule normally includes:

  1. Initial temperature T_1
  2. Final temperature T_f or stopping criterion
  3. The number of iterations to be performed at each temperature level N_{max}
  4. Rule of changing temperature

  The commonly used rule of changing temperature of SA is temperature decrement rule. There are basically two temperature decrement rules:

  – Exponential cooling scheme

  $$T_{k+1} = CT_k$$

  where C is called cooling rate and is a constant less than 1; T_{k+1}, T_k are temperatures at step $k+1$ and step k, respectively.

  – Linear cooling scheme

$$T_{k+1} = T_k - \Delta T$$

where T_{k+1} and T_k are temperatures at step $k+1$ and step k, respectively; ΔT is a small temperature change for each step.

In addition to the above commonly used temperature decrement rules, Alrefaei et al. (1999) presented a SA algorithm that used a constant temperature instead of decrementing temperature. Due to the popularity of the exponential cooling scheme, and since it has been shown to be an effective cooling scheme by many researchers, the exponential cooling scheme is used in our proposed SA algorithm. In the exponential cooling scheme, the selection of the cooling rate C is essential to the SA algorithm. If the parameter C is too small, then the algorithm may end up at a local optimal solution, and if it is too large then the algorithm may take a long time to converge. There is always a tradeoff between the quality of the SA solutions and the computation effort. The SA based heuristic algorithm's performance is evaluated based on the different selection of cooling rate C.

In the next section, the steps of the proposed SA based heuristic algorithm are explained in detail.

5.1. SA Based Heuristic Algorithm Steps

There are 7 steps in our proposed SA based heuristic algorithm:

Step 1: Initialization

The SA heuristic algorithm parameters are specified. The parameters include initial and final values of cooling temperature, T_1 and T_f, respectively; the cooling rate C and the desired number of iterations N_{max} for each level of the current temperature T.

A randomly generated sequence is selected as initial solution. The objective function value of this initial solution becomes the objective function values for both the current solution F_c and the best solution F_b. The iteration number N is set to 1.

Step 2: Objective Functions

In this step, the objective functions of the proposed models are selected as objective functions for the proposed SA algorithm.

For the single CONWIP line model:

$$F = f_{max}$$

For the multiple CONWIP assembly-type line model:

$$F = f_{max} - f_{min}$$

Step 3: Generate a feasible neighboring solution

This can be done by using one of the six perturbation schemes presented in the previous section. The new sequence obtained by the perturbation scheme is referred to as the test solution and its objective function value is represented by F_t.

Step 4: Compare F_t of the test solution with F_c of the current solution

If the objective function value of the test solution is greater than the objective function value of the current solution, i.e., $F_t \geq F_c$, go to Step 5. Otherwise, if $F_t < F_c$, the test solution will replace the current solution. Then, compare this test solution's objective function value with the objective function value of the best solution found so far F_b. If $F_t < F_b$, then replace the best solution with that of the test solution. Regardless whether the best solution is replaced with the test solution, go to step 6.

Step 5: Examine the Metropolis criterion

Determine the percent difference between the objective function values of the test solution and the current solution. The difference is denoted by ΔF and it is calculated by:

$$\Delta F = 100(F_t - F_c)/F_c$$

The Metropolis criterion is then used to determine the probability at which the relatively inferior test solution should be accepted. This probability is calculated by:

$$P(A) = exp(-\Delta F/(K_b T)$$

The value K_b is called the Boltzman constant. The Boltzman constant allows us to control the probability of inferior solutions being accepted. A random number then is generated in the interval $(0, 1)$. If this random number is less than $P(A)$, then the test solution replaces the current solution. Regardless of acceptance status, go to Step 6.

Step 6: Increment N

Increase the increment counter N by one. If the value of N is less than or equal to the desired number of iterations N_{max} for each temperature level, return to Step 3. Otherwise, go to Step 7.

Step 7: Adjust temperature

Decrease the temperature by its cooling rate as follow:

$$T = CT$$

If the new value of T is greater than or equal to the stopping value of T_f, i.e. if $T \geq T_f$, then reset N to one and return to Step 3. Otherwise, stop.

5.2. Pseudo Code of Simulated Annealing Based Heuristic Algorithm

The pseudo code of the developed algorithm is shown in Figure 5.1. The notations used in the code are given below:

T_1	-	Initial Temperature
T_f	-	Final Temperature
C	-	Cooling Rate
ΔF	-	Percent difference between the objective function values of the test solution and the current solution
K_b	-	Boltzman constant
PA	-	Probability for accepting an inferior value
F_t	-	New overall completion time of production line
F_c	-	Current overall completion time of production line
F_b	-	Best overall completion time of production line
N_{max}	-	Maximal iteration for each temperature level
N	-	Iteration number

SA based Heuristic Algorithm

```
begin
    Initialize T1, Tf, Nmax, Kb, C;
    Randomly generate a solution;
    Calculate Ft, Fc and Fb;
    repeat
        repeat
            if ( Ft < Fc)
            {
                Fc = Ft;
                If ( Ft < Fb)
                    Fb = Ft;
            }
            else
            {
                ΔF = ( Ft  –  Fc) * 100 / Fc;
                PA = exp (- ΔF / ( Kb * T ));
                Generate a random number from (0, 1);
                If (random # < PA)
                {
                    Fc = Ft;
                }
            }
            N ++;
            Generate a feasible neighboring solution by using perturbation schemes;
            Calculate Ft;
        until N > Nmax
        T = T1 * C;
    until ( T > Tf)
end
```

Figure 5.1: SA Based Heuristic Algorithm Pseudo Code

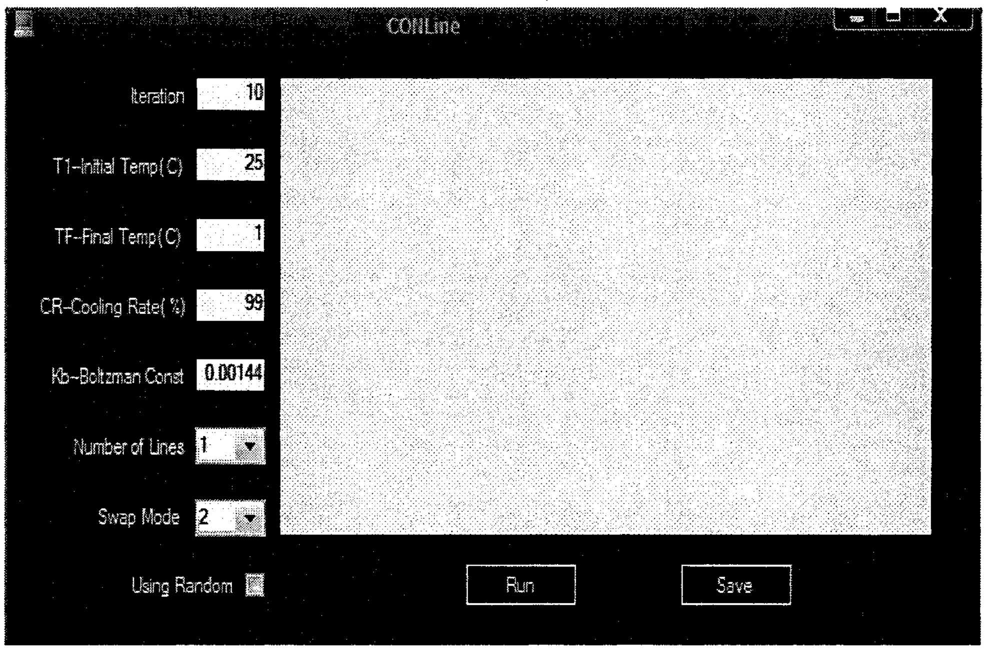
5.3. Implementing the Heuristic Algorithm

The proposed SA based heuristic algorithm is used to solve model CONL2 and MCONL presented in the previous chapters. The SA based heuristic algorithm is implemented using C# in a software package called CONLine. CONLine is designed to have a friendly GUI (Graphic User Interface) as shown in Figure 5.2. The GUI enables the user to easily enter both system parameters and SA parameters. The main functionalities of CONLine are given below.

1. CONLine is designed to solve the models for both single CONWIP line and multiple CONWIP assembly lines. In the case of multiple CONWIP assembly-type systems, the maximum number of fabrication lines is 15. The user can select the number of fabrication lines via the GUI. A single CONWIP line is chosen if 1 is selected from the dropdown list of the number of lines.

2. CONWIP system parameters, such as number of parts, number of machines and desired WIP levels can either be entered via the GUI or input from a data file. The processing times of parts can either be generated randomly from a uniform distribution (0,100) by CONLine or input from a data file. If the user wants to generate processing times randomly, the Random checkbox should be selected via the GUI. Once this checkbox is selected, a new GUI allows the user to enter the number of parts, the number of machines and the desired WIP levels. The new GUI is shown in Figure 5.3. If the user selects to randomly generate part processing time data, the initial sequence of these parts will also be generated randomly. CONLine also allows the user to input a pre-determined initial part sequence through a data file.

3. SA control parameters can also be entered via GUI. These control parameters include the number of iterations required for each temperature level; initial



Iteration 10

T1–Initial Temp(C) 25

TF–Final Temp(C) 1

CR–Cooling Rate(%) 99

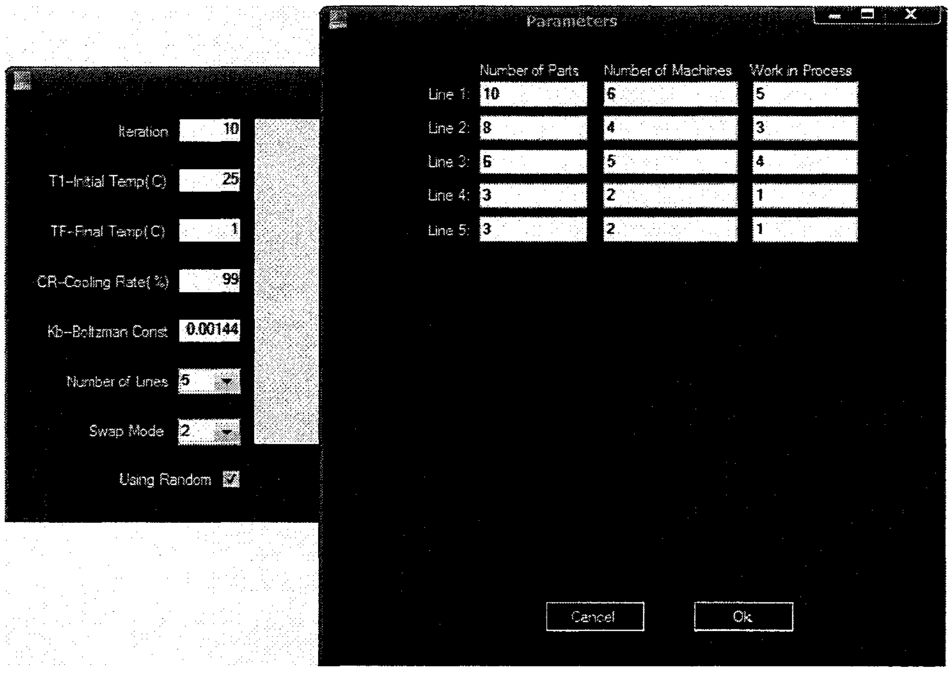
Kb–Boltzman Const 0.00144

Number of Lines 1

Swap Mode 2

Using Random

Figure 5.2: CONLine User Interface 1



Iteration	10
T1-Initial Temp(C)	25
TF-Final Temp(C)	1
CR-Cooling Rate(%)	99
Kb-Boltzman Const	0.00144
Number of Lines	5
Swap Mode	2
Using Random	✓

Parameters

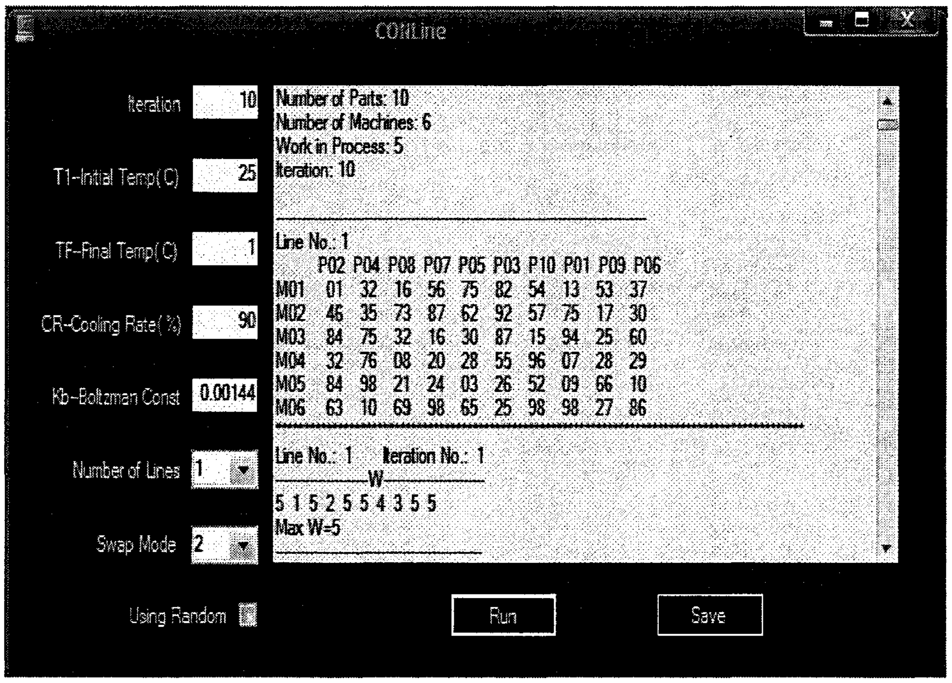
	Number of Parts	Number of Machines	Work in Process
Line 1:	10	6	5
Line 2:	8	4	3
Line 3:	6	5	4
Line 4:	3	2	1
Line 5:	3	2	1

Figure 5.3: CONLine User Interface 2

temperature; final temperature; cooling rate and Boltzman constant.

4. Perturbation schemes can also be selected from the GUI for a specific system. There are up to 6 different schemes offered to be selected in SWAP MODE dropdown list. All 6 scheme algorithms were discussed in the previous section.

5. With all inputs entered, the RUN button on the GUI will let CONLine start the algorithm and solve the selected model (single CONWIP line model or multiple CONWIP line model). Once the computation is complete, results are shown in the result window. The result includes part finishing times at each machine on each fabrication line for each iteration. The minimal makespan of all parts is also shown in the result. Moreover, the WIP levels at each time period are also displayed in the result window. The WIP level at each time period does not exceed the desired WIP level entered. The user can browse the results in the result window or use the SAVE button to save it into a data file. Figure 5.4 shows an example of the CONLine result window.



Iteration 10

T1-Initial Temp(C) 25

TF-Final Temp(C) 1

CR-Cooling Rate(%) 90

Kb-Boltzman Const 0.00144

Number of Lines 1

Swap Mode 2

```
Number of Parts: 10
Number of Machines: 6
Work in Process: 5
Iteration: 10

______________________________________
Line No.: 1
     P02 P04 P08 P07 P05 P03 P10 P01 P09 P06
M01  01  32  16  56  75  82  54  13  53  37
M02  46  35  73  87  62  92  57  75  17  30
M03  84  75  32  16  30  87  15  94  25  60
M04  32  76  08  20  28  55  96  07  28  29
M05  84  98  21  24  03  26  52  09  66  10
M06  63  10  69  98  65  25  98  98  27  86
******************************************************

Line No.: 1    Iteration No.: 1
_________W_________
5 1 5 2 5 5 4 3 5 5
Max W=5
___________________
```

Figure 5.4: CONLine Result Window

Chapter 6

Example Problems and Computational Results

6.1. CONL1 Numerical Examples

In this section, a small sized problem is presented to illustrate the CONL1 model developed in Chapter 3. Consider a production line with four serial workstations that produces three products. This production line is similar to that shown in Figure 3.1. Product demands and processing times for products A, B, and C are shown in Table 6.1. Setup times required by the items at the workstations are shown in Table 6.2. Workstation 2 is the bottleneck station in this example, since it requires significant longer setup times and processing times for all the items than other workstations in the line. Production control follows the hybrid pull/push CONWIP strategy.

In solving this example problem, we considered several factors as elaborated below:

1. Weighted setup costs and workload imbalance costs vs. job sequence, lot size and the length of the part list

Table 6.1: Demand and Processing Time

		Processing Time			
Parts	Demands	Station 1	Station 2	Station 3	Station 4
A	2	2	2	2	2
B	1	9	10	7	5
C	4	3	6	4	2

Table 6.2: Setup Time

	Station 1			Station 2			Station 3			Station 4		
	A	B	C	A	B	C	A	B	C	A	B	C
A	0.00	0.10	0.20	0.00	1.00	1.00	0.00	0.05	0.03	0.00	0.05	0.20
B	0.10	0.00	0.05	2.00	0.00	3.00	0.02	0.00	0.05	0.02	0.00	0.05
C	0.10	0.05	0.00	3.00	5.00	0.00	0.20	0.05	0.00	0.10	0.03	0.00

Based on the data shown in Tables 6.1 and 6.2, the mathematical programming model CONL1 is utilized to find the optimal production sequence and optimal lot size for each item with various given part list length K. The optimization is achieved in terms of best setup times and smooth production. The example problem was tested for different K and three different scenarios. If production smoothness is considered insignificant compared to setup costs, then β is set to 0 and the solutions will be generated with minimized total setup cost only. Optimal solutions of the model with $\beta = 0$ and different part list lengths K are presented in Table 6.3.

In fact, if we assume that production smoothness is not a concern in controlling the production process, then different part list lengths K will have no impact on the optimal solution in minimizing the setup cost. The best sequence is always $B-A-C$ and the best lot sizes are 1, 2 and 4 for these three products, respectively. In the second scenario, we assume that setup cost is insignificant compared to the requirement of production smoothness. In this case, we let $\alpha = 0$ and let $\beta = 1$.

Table 6.3: Computational Results with Setup Cost Only ($\alpha = 1, \beta = 0$)

K	Sequence	Lot Size	Objective
3	$B - A - C$	$1, 2, 4$	3
4	$B - A - A - C$	$1, 2, 0, 4$	3
5	$B - B - B - A - C$	$0, 0, 1, 2, 4$	3
6	$B - B - B - B - A - C$	$0, 1, 0, 0, 2, 4$	3
7	$B - B - B - B - B - A - C$	$0, 0, 1, 0, 0, 2, 4$	3

Table 6.4: Computational Results without Setup Cost Only ($\alpha = 0, \beta = 1$)

K	Sequence	Lot Sizes	Objective
3	$C - B - A$	$4, 1, 2$	18
4	$C - C - A - B$	$2, 2, 2, 1$	9
5	$B - C - A - C - C$	$1, 1, 2, 1, 2$	8
6	$B - C - A - C - C - C$	$1, 1, 2, 1, 1, 1$	4
7	$B - C - A - C - C - C - B$	$1, 1, 2, 1, 1, 1, 0$	5

Optimal solutions of the model for different K are shown in Table 6.4. The results in Table 6.4 show that $K = 6$ corresponds to the minimized objective function value and the job order is $B - C - A - C - C - C$. The lot sizes are 1, 1, 2, 1, 1, 1, respectively, for the products in the above sequence. If this solution is implemented, there should be 6 entries on the part list. In the third and more general scenario, neither setup cost nor production smoothness requirement is excluded from the optimization model. In this situation, with $\alpha = 1$, a number of β values were used to generate different solutions. Solutions corresponding to $\beta = 0.8, 1.0, 2.0$ and 3.0 and different K values are presented in Tables 6.5, 6.6, 6.7 and 6.8, respectively.

The results in Table 6.5 to Table 6.8 show that when $\beta = 0.8$ or 1.0, the best part list length K is 6, the best sequence is $B - A - C - C - C - C$ and the lot sizes are 1, 2, 1, 1, 1 and 1, respectively, for the products in the above

Table 6.5: Computational Result 1 with $\alpha = 1.0, \beta = 0.8$

K	Sequence	Lot Sizes	Objective
3	$C-A-B$	$4,2,1$	20.8
4	$C-C-A-B$	$2,2,2,1$	11.2
5	$B-A-C-C-C$	$1,2,1,1,2$	12.6
6	$B-A-C-C-C-C$	$1,2,1,1,1,1$	7.8
7	$B-C-C-C-C-A-A$	$1,1,1,1,1,1,1$	12.4

Table 6.6: Computational Result 2 with $\alpha = 1.0, \beta = 1.0$

K	Sequence	Lot Sizes	Objective
3	$C-B-A$	$4,1,2$	25
4	$C-C-A-B$	$2,2,2,1$	13
5	$B-A-C-C-C$	$1,2,1,1,2$	15
6	$B-A-C-C-C-C$	$1,2,1,1,1,1$	9
7	$B-C-C-C-C-A-A$	$1,1,1,1,1,0,2$	14

sequence. When $\beta = 2.0$ or 3.0, the best part list length K is also 6, the best sequence will be $C-C-C-C-A-B$, and the lot sizes are 1, 1, 1, 1, 2 and 1, respectively. These results indicate that different weights associated with setup costs and smoothness requirement affect the CONWIP control plan significantly. In reality, such weight differences can be estimated from different cost and accounting figures and the mathematical programming model can be run for different scenarios as shown in this example. The best solution, closest to the real situation, can be evaluated, selected and implemented.

2. Number of containers

Once the part list, job sequence and lot size have been determined, the number of containers can be obtained from the formula presented in Section 3.2.3.

As shown in Table 6.6 when $\alpha = 1$, $\beta = 1$, the length of the part list K is

Table 6.7: Computational Result 3 with $\alpha = 1.0, \beta = 2.0$

K	Sequence	Lot Sizes	Objective
3	$C-B-A$	$4,1,2$	43
4	$C-C-A-B$	$2,2,2,1$	22
5	$B-C-A-C-C$	$1,1,2,1,2$	23
6	$C-C-C-C-A-B$	$1,1,1,1,2,1$	14
7	$B-C-C-C-C-A-A$	$1,1,1,1,1,1,1$	22

Table 6.8: Computational Result 4 with $\alpha = 1.0, \beta = 3.0$

K	Sequence	Lot Sizes	Objective
3	$C-B-A$	$4,1,2$	61
4	$C-C-A-B$	$2,2,2,1$	31
5	$B-C-A-C-C$	$1,1,2,1,2$	31
6	$C-C-C-C-A-B$	$1,1,1,1,2,1$	19
7	$B-C-A-C-C-C-B$	$1,1,2,1,1,1,0$	27

5; job sequence is $B-A-C-C-C$ and lot sizes are 1, 2, 1, 1, 2, respectively. The number of containers = 110/38= 2.89. The number of the containers is then rounded to the nearest integer. In this case, the number of the containers is 3.

3. Makespan vs. the length of the part list and lot sizes

The makespan is defined as the time required to complete all the products from the time the first part starts in the system for processing until the last part leaves the system. The objective function of CONL1 has two portions. The first portion is the setup costs and the second portion is the costs associated with the job production smoothness. The setup costs or the setup times will affect the overall completion time as the setup times discussed in model CONL1 are sequence dependent. However, it is not very obvious that the second portion of the objective function would affect the part overall completion time. Below we will show how the job production smoothness factor will affect the overall part completion time.

Take two cases from Table 6.6 with $\alpha = 1.0$ and $\beta = 1.0$. The two cases are shown below in Table 6.9:

Table 6.9: Two cases with Weighted Objective Function ($\alpha = 1.0, \beta = 1.0$)

Cases	K	Sequence	Lot Sizes	Objective
Case 1	5	$B-A-C-C-C$	$1,2,1,1,2$	15
Case 2	6	$B-A-C-C-C-C$	$1,2,1,1,1,1$	9

From the above table, we can see that the setup costs or the setup times for the two sequences are the same since only different part type switching incurs setup costs while switching between entries representing the same parts does not incur any setup cost. Because the costs associated with job production smoothness on the bottleneck are different, the length of the part list and the lot sizes are different in these two cases.

Table 6.10 below shows the makespan for the two cases with the number of containers set to 2, 3, 4 and 5.

Table 6.10: Makespans vs. Number of Containers for Case 1 and Case 2

Number of	Makespans	
Containers	Case 1	Case 2
2	87	79
3	73	67
4	68	61
5	66	61

Two observations can be made from the information in Table 6.10 and Figure 6.1. First, in terms of CONWIP system performance measured by the makespan, it performs better in case 2 than in case 1. In other words, the CONWIP system with smaller lot size and longer part list has better performance than the system with larger lot size and shorter part list. Secondly, with the same overall part completion time for the two cases, case 2 has smaller number of containers than that in case 1. Since the system WIP levels are limited by the number of containers, smaller number of containers indicate lower WIP levels. The result shows that smaller lot size results in smaller number of containers and thus, lower WIP levels. From these two observations, it shows that both processing lot size and transfer lot size are important for system performance. The processing lot size is determined by the setup time but the transfer lot size can be reduced to decrease the overall part completion time thus to increase the throughput and to reduce the WIP level. These observations also confirm that the second portion of the objective function of CONL1 is of the same importance as the setup costs and is essential to determine the optimal transfer lot size.

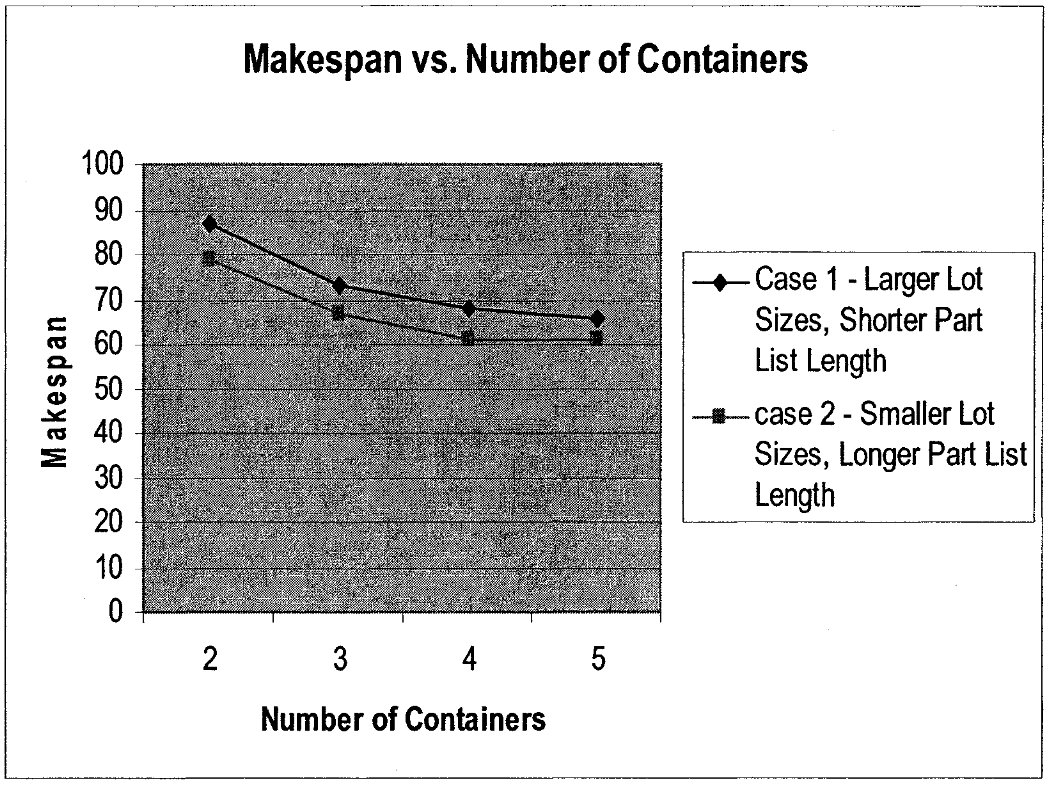


Figure 6.1: CONWIP System Performance for Different Lot Sizes

6.2. CONL2 Numerical Examples

In Section 3.3.2, we presented the model CONL2 for a single serial CONWIP production line. In Chapter 5, a simulated annealing based heuristic algorithm was presented to solve the model. In this section, we will present several example problems to illustrate this model. The example problems are solved by the proposed SA based heuristic algorithm. Computational result analyzes are given to evaluate the CONWIP system performance. The system performance includes makespan and WIP levels.

6.2.1. Example 1

In Example 1, a small sized problem is considered to validate the model CONL2 and evaluate the proposed simulating annealing based heuristic algorithm.

Specific information for this small sized single CONWIP serial line is:

- The production system has 3 machines and is a single serial CONWIP production line
- 6 types of products or parts are processed in this CONWIP line.
- All 6 parts are processed by all 3 machines sequentially.
- Processing times on the 3 machines for the six products were generated randomly from uniform distribution (0,100). For comparison purpose, once they are generated, they are kept the same for the calculation of all WIP levels. The processing times for the six products are listed in Table A.1 of Appendix A.

- WIP levels from 1 to 6 are tested.

- Different sets of parameters for simulated annealing based heuristic algorithm are tested to evaluate its performance. The parameters selected are listed in Table 6.11.

Table 6.11: SA Parameteres

SA Para. Sets	Cooling Rate (%)	Iterations for Each Temperature Level	Initial $P(A)$ (%)	Inferiority Base (%)	Initial Temperature	Final Temperature
1	95	10	25	5	25	1
2	95	20	25	5	25	1
3	95	20	50	10	25	1
4	99	20	50	10	25	1

$P(A)$ and Inferiority Base in Table 6.11 give the probability of inferior solutions being accepted to become the current solution. These values are used to compute the Boltzman constant K_b. At the initial temperature T_1 a solution inferior to the current solution by "Inferiority Base" percent has an "Initial $P(A)$" percent of being accepted. For example, Set 1 in Table 6.11 indicates that a solution 5% inferior to the current solution in terms of the objective function value has a 25% probability of being accepted at initial temperature T_1.

As discussed in Chapter 5, six different perturbation schemes are used in our proposed SA based heuristic algorithm. These perturbation schemes are:

1. Interchanging two adjacent jobs

2. Interchanging two jobs that are randomly selected

3. Inserting a randomly selected job to a randomly selected position

4. Inserting a randomly selected subsequence of jobs to a randomly selected position

5. Reversing a randomly selected subsequence of jobs

6. Reversing a randomly selected subsequence of jobs and then inserting the entire selected jobs to a randomly selected position

Tables 6.12, 6.13, 6.14, 6.15, 6.16 and 6.17 show minimal makespans found by the SA algorithm and optimal makespans found by the off-the-shelf software package LINGO.

Table 6.12: Example 1 - Results of Perturbation Scheme 1

SA Para. Sets	WIP Levels	Minimal Makespans (SA)	Optimal Makespans	Deviation (%)	Total Solutions Evaluated	CPU Times (seconds)
1	1	1021	1021	0.0	693	3
	2	548	538	1.9		3
	3	438	438	0.0		3
	4	421	417	1.0		2
	5	417	417	0.0		3
	6	417	417	0.0		3
2	1	1021	1021	0.0	1323	5
	2	538	538	0.0		4
	3	452	438	3.2		4
	4	417	417	0.0		4
	5	418	417	0.2		5
	6	417	417	0.0		4
3	1	1021	1021	0.0	1323	4
	2	538	538	0.0		4
	3	447	438	2.1		5
	4	418	417	0.2		7
	5	417	417	0.0		5
	6	417	417	0.0		5
4	1	1021	1021	0.0	6741	31
	2	538	538	0.0		32
	3	438	438	0.0		29
	4	417	417	0.0		31
	5	417	417	0.0		29
	6	417	417	0.0		33

Table 6.13: Example 1 - Results of Perturbation Scheme 2

SA Para. Sets	WIP Levels	Minimal Makespans (SA)	Optimal Makespans	Deviation (%)	Total Solutions Evaluated	CPU Times (seconds)
1	1	1021	1021	0.0	693	2
	2	538	538	0.0		2
	3	449	438	2.5		2
	4	438	417	5.0		2
	5	418	417	0.2		3
	6	417	417	0.0		3
2	1	1021	1021	0.0	1323	5
	2	538	538	0.0		4
	3	438	438	0.0		4
	4	417	417	0.0		5
	5	417	417	0.0		4
	6	417	417	0.0		4
3	1	1021	1021	0.0	1323	5
	2	538	538	0.0		4
	3	447	438	2.1		5
	4	417	417	0.0		6
	5	417	417	0.0		6
	6	417	417	0.0		6
4	1	1021	1021	0.0	6741	30
	2	538	538	0.0		30
	3	438	438	0.0		31
	4	417	417	0.0		30
	5	417	417	0.0		30
	6	417	417	0.0		29

Table 6.14: Example 1 - Results of Perturbation Scheme 3

SA Para. Sets	WIP Levels	Minimal Makespans (SA)	Optimal Makespans	Deviation (%)	Total Solutions Evaluated	CPU Times (seconds)
1	1	1021	1021	0.0	693	2
	2	538	538	0.0		2
	3	447	438	2.1		2
	4	426	417	2.2		2
	5	417	417	0.0		3
	6	426	417	2.2		2
2	1	1021	1021	0.0	1323	5
	2	538	538	0.0		5
	3	438	438	0.0		5
	4	417	417	0.0		6
	5	417	417	0.0		4
	6	417	417	0.0		4
3	1	1021	1021	0.0	1323	5
	2	538	538	0.0		6
	3	447	438	2.1		4
	4	417	417	0.0		6
	5	417	417	0.0		5
	6	417	417	0.0		5
4	1	1021	1021	0.0	6741	30
	2	538	538	0.0		31
	3	438	438	0.0		31
	4	417	417	0.0		30
	5	417	417	0.0		30
	6	417	417	0.0		31

Table 6.15: Example 1 - Results of Perturbation Scheme 4

SA Para. Sets	WIP Levels	Minimal Makespans (SA)	Optimal Makespans	Deviation (%)	Total Solutions Evaluated	CPU Times (seconds)
1	1	1021	1021	0.0	693	2
	2	538	538	0.0		2
	3	438	438	0.0		2
	4	417	417	0.0		2
	5	417	417	0.0		2
	6	426	417	2.2		2
2	1	1021	1021	0.0	1323	5
	2	538	538	0.0		4
	3	438	438	0.0		5
	4	426	417	2.2		4
	5	417	417	0.0		4
	6	417	417	0.0		5
3	1	1021	1021	0.0	1323	5
	2	538	538	0.0		6
	3	438	438	0.0		7
	4	417	417	0.0		5
	5	417	417	0.0		5
	6	417	417	0.0		5
4	1	1021	1021	0.0	6741	32
	2	538	538	0.0		33
	3	438	438	0.0		30
	4	417	417	0.0		30
	5	417	417	0.0		31
	6	417	417	0.0		30

Table 6.16: Example 1 - Results of Perturbation Scheme 5

SA Para. Sets	WIP Levels	Minimal Makespans (SA)	Optimal Makespans	Deviation (%)	Total Solutions Evaluated	CPU Times (seconds)
1	1	1021	1021	0.0	693	2
	2	538	538	0.0		3
	3	438	438	0.0		3
	4	417	417	0.0		2
	5	417	417	0.0		2
	6	418	417	0.2		2
2	1	1021	1021	0.0	1323	5
	2	538	538	0.0		5
	3	452	438	3.2		4
	4	417	417	0.0		5
	5	417	417	0.0		4
	6	417	417	0.0		4
3	1	1021	1021	0.0	1323	5
	2	538	538	0.0		6
	3	438	438	0.0		5
	4	417	417	0.0		5
	5	417	417	0.0		5
	6	418	417	0.2		6
4	1	1021	1021	0.0	6741	29
	2	538	538	0.0		31
	3	438	438	0.0		32
	4	417	417	0.0		31
	5	417	417	0.0		31
	6	417	417	0.0		31

Table 6.17: Example 1 - Results of Perturbation Scheme 6

SA Para. Sets	WIP Levels	Minimal Makespans (SA)	Optimal Makespans	Deviation (%)	Total Solutions Evaluated	CPU Times (seconds)
	1	1021	1021	0.0		2
	2	538	538	0.0		3
	3	438	438	0.0		3
1	4	418	417	0.2	693	3
	5	417	417	0.0		3
	6	417	417	0.0		2
	1	1021	1021	0.0		6
	2	538	538	0.0		4
	3	438	438	0.0		4
2	4	417	417	0.0	1323	4
	5	417	417	0.0		4
	6	417	417	0.0		5
	1	1021	1021	0.0		7
	2	538	538	0.0		7
	3	438	438	0.0		4
3	4	417	417	0.0	1323	5
	5	417	417	0.0		5
	6	417	417	0.0		5
	1	1021	1021	0.0		32
	2	538	538	0.0		30
	3	438	438	0.0		31
4	4	417	417	0.0	6741	33
	5	417	417	0.0		33
	6	417	417	0.0		30

Table 6.18 gives both minimal makespans and the job sequences found by solving CONL2 using the developed simulated annealing algorithm.

For each SA parameter set, six WIP levels were computed.

The relationship between the WIP levels and the makespans is shown in Figure 6.2.

Table 6.18: Results of Example 1 - Perturbation Scheme 2

SA Para. Sets	WIP Levels	Minimal Makespans	Best Job Sequences	Total Solutions Evaluated	Computational Times (seconds)
1	1	1021	P2-P5-P6-P1-P3-P4	693	2
	2	538	P2-P3-P5-P4-P1-P6		2
	3	449	P2-P4-P1-P5-P3-P6		2
	4	438	P5-P2-P4-P1-P3-P6		2
	5	418	P2-P5-P4-P1-P3-P6		3
	6	417	P2-P4-P5-P1-P3-P6		3
2	1	1021	P2-P5-P6-P1-P3-P4	1323	5
	2	538	P2-P3-P5-P4-P1-P6		4
	3	438	P5-P2-P1-P4-P3-P6		4
	4	417	P2-P4-P5-P1-P3-P6		5
	5	417	P2-P4-P5-P1-P3-P6		4
	6	417	P2-P4-P5-P1-P3-P6		4
3	1	1021	P2-P5-P6-P1-P3-P4	1323	5
	2	538	P2-P3-P5-P4-P1-P6		5
	3	447	P5-P2-P1-P4-P3-P6		5
	4	417	P2-P4-P5-P1-P3-P6		6
	5	417	P2-P4-P5-P1-P3-P6		6
	6	417	P2-P4-P5-P1-P3-P6		6
4	1	1021	P2-P5-P6-P1-P3-P4	6741	30
	2	538	P2-P3-P5-P4-P1-P6		30
	3	438	P5-P2-P4-P1-P3-P6		31
	4	417	P2-P4-P5-P1-P3-P6		30
	5	417	P2-P4-P5-P1-P3-P6		30
	6	417	P2-P4-P5-P1-P3-P6		29

By looking at the results in Tables 6.12 to 6.18, we have the following observations:

1. CONWIP system performance:

The makespan decreases as the WIP level increases. However, once the WIP reaches certain point, adding more containers or increasing WIP levels will not improve the system makespan. In our particular example, the best WIP level or the number of the containers for this CONWIP production line is 4. This provides an important message to factory managers that more containers beyond a certain number cannot further reduce the makespan. Also it shows that the proposed model and the heuristic method can help managers to find the best WIP level to improve the production performance and at the same time to avoid having no-value added but costly extra containers circulating in the system.

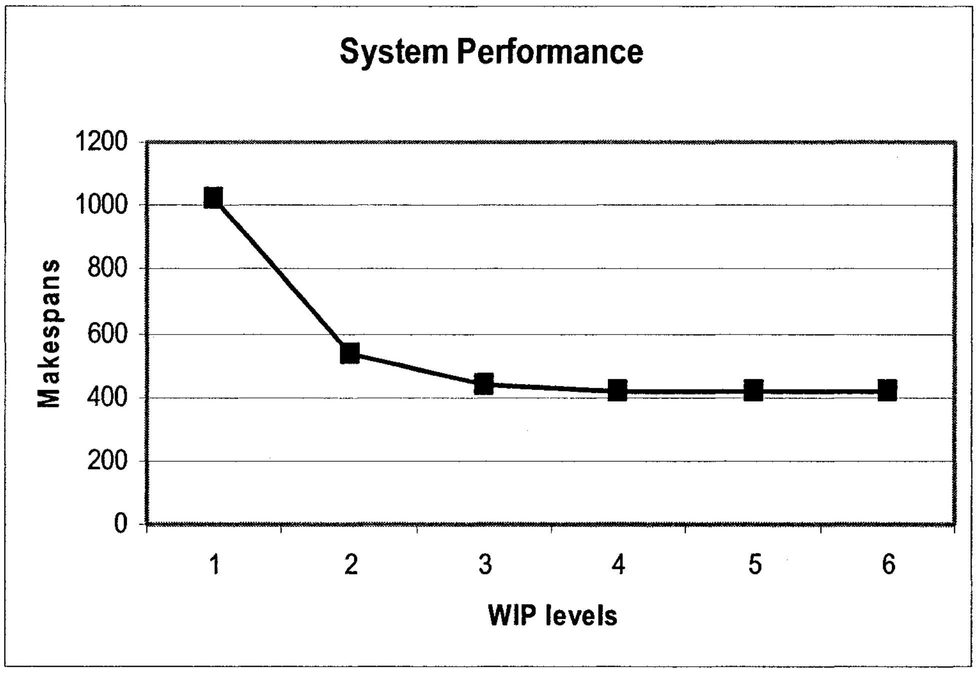


Figure 6.2: Makespan vs. WIP Level in Single CONWIP Line Example 1

2. Bottleneck Machines

As we discussed earlier, the system reaches its maximum throughput level with 4 containers. We know that there are only 3 machines in the system, this indicates that there is no idle machine in the system at any point of time and there is a waiting queue in front of the bottleneck machine. Table 6.19 gives the total waiting time for different WIP levels. The total waiting time is the waiting time of all parts in front of each machine during the time of processing.

Table 6.19: Waiting Times (Scheme 4)

WIP Level	Total waiting Times		
	Machine 1	Machine 2	Machine 3
2	160	0	0
3	0	144	21
4	0	125	102

Table 6.19 shows that with different WIP levels, the total waiting time is different. Also the location of the bottleneck machine changes when the WIP level varies. With 2 containers, machine 1 appears as the constraint of the system as parts are waiting in front of machine 1 for a free container to enter the system. On the plant floor, a waiting queue appearing in front of the first machine of the product line could indicate that more containers need to be added to the system. With 3 containers, machine 2 becomes the bottleneck in the system while with 4 containers, both machines 2 and 3 are shown as bottlenecks. In the case of 4 containers, no dominant bottleneck can be determined. In conclusion, when the number of containers are varied, the bottleneck machine can shift from one to another. Moreover, different WIP levels can have more than one bottleneck in the system.

3. Simulated Annealing Based Algorithm Performance

Tables 6.12 to 6.17 show that with 99% cooling rate, the proposed SA based heuristic algorithm found the optimal solutions for all the tested WIP levels. While with a 95% cooling rate as shown in set 3, SA based heuristic algorithm failed to find the optimal solutions for all the WIP levels. So selecting a higher cooling rate may result in better results. This is understandable as with a higher cooling rate, the temperature in the annealing process decreases slower than that with the lower cooling rate. Therefore, the annealing process will take a longer time. And this longer annealing process avoids having the system being cooled too fast without reaching its steady state. So the cooling rate should be carefully selected when the SA algorithm is applied in order to obtain an optimal or near-optimal solution and to avoid being trapped in a local optimum.

By comparing set 2 and 3, the results do not show much difference. In terms of the algorithm parameters, the differences between these 2 sets are initial $P(A)$ and the inferior base. As we mentioned earlier, these values were used to determine the Boltzman constant. The Boltzman constant is 0.00577 and 0.00144 for set 2 and set 3, respectively. From this experiment, we conclude that the Boltzman constant does not have a clear effect on the results for the particular tested case.

4. Perturbation Schemes vs. SA Solutions

Tables 6.12 to 6.17 show that when the iteration number is relatively small, as shown in set 1, schemes 4, 5 and 6 were able to find the optimal makespan while other schemes failed to find the optimal solution with the WIP level set as 4. With same computation time and same initial job sequence, schemes 4, 5 and 6 are more efficient than all other schemes. When the iteration number is large, we do not observe significant differences among the six different perturbation schemes in terms of solution quality.

In fact, it is necessary to find the most efficient scheme when considering tradeoffs between solution quality and computation time, especially for large sized problems where it is impossible to choose sufficient large iteration numbers within an acceptable computation time.

5. Computation Efforts Observation

Computation time depends on the selection of the cooling rate; the iterations for each temperature level and the WIP level. It is obvious that the CPU time or computation time increases when the cooling rate and the iterations for each temperature level increase. However, Table 6.18 does not show a clear trend of CPU time or computation time with WIP level increase.

6.2.2. Example 2

In this section, we present another example for a small sized single serial CONWIP line. In this example, the number of machines is larger than the number of parts. This is different from the example 1 where the number of machines is smaller than the number of parts.

Below is a summary of the system parameters for example 2:

- The production system has 10 machines and is a single serial CONWIP production line
- 5 types of products or parts are processed in this CONWIP line.
- All 5 parts are processed by all 10 machines sequentially.
- Processing times on the 10 machines for the 5 products were generated randomly from uniform distribution (0,100). For comparison purpose, once they are generated, they are kept the same for the calculation of all WIP levels. The processing times for the 5 products are listed in Table A.2 of Appendix A.
- WIP levels from 1 to 8 are tested.

Example 2 was also solved by the SA based heuristic algorithm, but only scheme 4 was employed. SA parameters chosen in this example are the the same as in set 1 in example 1, i.e., cooling rate is 95%, the initial temperature is 25, the final temperature is 1 and the iteration for each temperature level is 10. The Boltzman constant is 0.00144. Table 6.20 shows the minimal makespan for 8 WIP levels.

Table 6.20: Results of Example 2 - Perturbation Scheme 4

WIP Levels	Minimal Makespan (SA)	Optimal Makespan	Deviation (%)	Total Solutions Evaluated	CPU Times (seconds)
1	2588	2588	0.0	693	4
2	1390	1390	0.0	693	5
3	1020	1020	0.0	693	5
4	827	827	0.0	693	5
5	810	810	0.0	693	4
6	810	810	0.0	693	4
7	810	810	0.0	693	4
8	810	810	0.0	693	4

Results of example 2 show the same observations as those of example 1, except the following:

In example 2 problem, the system reaches its maximum capacity when the container number is 5. Since there are 10 machines in the system, there are always 5 machines at idle at any point of time. While in example 1, there is no idle machine. This is the result of less parts than machines and the imbalance in processing times of the different parts. This observation suggests that in order to obtain a better overall CONWIP system performance and to better utilize machine capacities, processing times of all parts may need to be balanced if possible.

6.2.3. Example 3

The third example was constructed to solve a relatively large problem of a single serial CONWIP production line. Details of this example are given below:

- The system has 10 machines in a single serial CONWIP production line.
- 30 products or parts are manufactured in this CONWIP line.
- All 30 parts are processed by all 10 machines sequentially.
- Processing times on the 10 machines for the 30 products were generated randomly from (0,100) using uniform distribution and they are kept as the same for all WIP level computations. The processing times for the 30 products are listed in Table A.3 of Appendix A.
- WIP levels from 1 to 30 are tested.
- Different sets of parameters for the simulated annealing heuristic algorithm were tested to evaluate the heuristic algorithm. The parameters selected are listed in Table 6.21

Table 6.21: SA Parameters

SA Para. Sets	Cooling Rate (%)	Iterations for Each Temperature Level	Initial $P(A)$ (%)	Inferiority Base (%)	Initial Temperature	Final Temperature
1	99	50	25	5	25	1
2	99.5	50	25	5	25	1

The six perturbation schemes presented in Chapter 5 were also used in the SA algorithm to solve this example problem. Tables 6.23, 6.24, 6.25, 6.26, 6.27 and

6.28 show the minimal makespans found by the SA based heuristic algorithm for schemes 1, 2 , 3 ,4, 5 and 6, respectively.

Similar observations were obtained from the computational results of this larger example problem.

1. System Performance

In terms of system performance, we have similar observations to those from Example 1. Tradeoffs between makespan and WIP level are shown clearly from Figure 6.3 when the number of containers is varied. A smaller container number to keep the WIP level low limits the jobs in the system and thus reduces production throughput. The results show that larger number of containers can reduce the makespan. However, the results also show that 14 containers are maximum in terms of reducing the makespan for this problem. If there are more than 14 containers circulating in the system, the extra containers will not help to further reduce the makespan.

2. Perturbation Schemes

As shown in Figure 6.3, different schemes have no significant impact on makespan and WIP levels for this particular case.

3. Simulated Annealing Heuristic Algorithm Performance

In terms of parameters of simulated annealing, from Table 6.23 to 6.28, the same conclusion in solving the small sized problem can be reached. Faster cooling rate results longer computation time since there are more iterations. Also different levels of WIP may not affect much on the computation time.

4. Computation Efforts Observations

The model was solved using the SA heuristic algorithm on a Intel Centrino Duo 1.83Hz PC computer. By comparing the results shown in Table 6.22, it is obvious that with the number of parts increasing, the computation time increases.

Table 6.22: Computation Times with Different Number of Parts

Number of Parts	Cooling Rate (%)	Iterations for Each Temperature Level	Initial $P(A)$ (%)	Inferiority Base (%)	Initial Temp	Final Temp	CPU Time
6	99	50	25	5	25	1	101
30	99	50	25	5	25	1	244

In solving these examples, when part number increases from 6 to 30 with the same WIP level at 6 and the same perturbation scheme 1, the computation time increases from about 101 seconds to 244 seconds with the same SA parameters.

Table 6.23: Example 3 - Results of Perturbation Scheme 1

SA Para. Sets	WIP Levels	Minimal Makespans (SA)	Total Solutions Evaluated	CPU Times (seconds)
1	1	15732	16371	248
	2	7897		243
	4	4322		242
	6	3201		244
	8	2753		239
	10	2562		251
	12	2487		252
	14	2452		254
	16	2455		249
	18	2427		250
	20	2405		252
	22	2426		255
	24	2428		268
	26	2453		250
	28	2360		253
	30	2390		251
2	1	15732	32793	927
	2	7902		898
	4	4259		922
	6	3225		923
	8	2784		972
	10	2556		949
	12	2469		962
	14	2406		967
	16	2423		943
	18	2379		931
	20	2380		922
	22	2390		957
	24	2384		947
	26	2423		946
	28	2444		953
	30	2382		946

Table 6.24: Example 3 - Results of Perturbation Scheme 2

SA Para. Sets	WIP Levels	Minimal Makespans (SA)	Total Solutions Evaluated	CPU Times (seconds)
1	1	15732	16371	255
	2	7910		246
	4	4278		242
	6	3191		242
	8	2763		243
	10	2572		250
	12	2451		252
	14	2343		250
	16	2402		251
	18	2398		250
	20	2368		251
	22	2350		248
	24	2400		249
	26	2380		250
	28	2381		250
	30	2364		250
2	1	15732	32793	980
	2	7899		894
	4	4267		953
	6	3187		896
	8	2747		895
	10	2528		924
	12	2410		942
	14	2383		929
	16	2361		942
	18	2372		933
	20	2395		959
	22	2389		968
	24	2391		956
	26	2371		1032
	28	2370		992
	30	2395		951

Table 6.25: Example 3 - Results of Perturbation Scheme 3

SA Para. Sets	WIP Levels	Minimal Makespans (SA)	Total Solutions Evaluated	CPU Times (seconds)
1	1	15732	16371	255
	2	7896		245
	4	4239		250
	6	3168		250
	8	2776		246
	10	2523		248
	12	2375		253
	14	2400		250
	16	2392		247
	18	2415		249
	20	2422		244
	22	2402		243
	24	2382		249
	26	2388		246
	28	2381		247
	30	2391		258
2	1	15732	32793	916
	2	7890		899
	4	4218		886
	6	3207		890
	8	2720		891
	10	2521		915
	12	2453		923
	14	2398		925
	16	2391		927
	18	2385		923
	20	2408		932
	22	2366		929
	24	2377		926
	26	2355		921
	28	2395		929
	30	2342		932

Table 6.26: Example 3 - Results of Perturbation Scheme 4

SA Para. Sets	WIP Levels	Minimal Makespans (SA)	Total Solutions Evaluated	CPU Times (seconds)
1	1	15732	16371	247
	2	7899		240
	4	4261		237
	6	3196		238
	8	2724		245
	10	2540		246
	12	2432		262
	14	2393		251
	16	2414		248
	18	2400		251
	20	2401		249
	22	2398		254
	24	2366		244
	26	2393		262
	28	2398		254
	30	2392		247
2	1	15732	32793	917
	2	7896		904
	4	4223		887
	6	2702		884
	8	2701		884
	10	2502		918
	12	2382		930
	14	2391		909
	16	2373		925
	18	2355		913
	20	2370		922
	22	2384		926
	24	2356		930
	26	2380		926
	28	2342		916
	30	2378		934

Table 6.27: Example 3 - Results of Perturbation Scheme 5

SA Para. Sets	WIP Levels	Minimal Makespans (SA)	Total Solutions Evaluated	CPU Times (seconds)
1	1	15732	16371	245
	2	7898		238
	4	4228		238
	6	3195		237
	8	2768		237
	10	2537		243
	12	2438		245
	14	2374		245
	16	2394		259
	18	2363		248
	20	2402		250
	22	2360		250
	24	2396		259
	26	2368		244
	28	2368		245
	30	2340		251
2	1	15732	32793	908
	2	7895		892
	4	4262		883
	6	3166		879
	8	2752		881
	10	2547		912
	12	2434		923
	14	2384		915
	16	2385		915
	18	2376		917
	20	2318		913
	22	2345		915
	24	2384		919
	26	2373		915
	28	2330		917
	30	2370		917

Table 6.28: Example 3 - Results of Perturbation Scheme 6

SA Para. Sets	WIP Levels	Minimal Makespans (SA)	Total Solutions Evaluated	CPU Times (seconds)
	1	15732		254
	2	7902		240
	4	4238		240
	6	3188		241
	8	2702		240
	10	2561		245
	12	2435		247
	14	2358		247
1	16	2392	16371	250
	18	2392		248
	20	2406		255
	22	2379		249
	24	2387		249
	26	2372		250
	28	2376		249
	30	2390		255
	1	15732		948
	2	7899		891
	4	4189		886
	6	3177		884
	8	2741		881
	10	2538		916
	12	2400		938
	14	2399		929
2	16	2363	32793	957
	18	2370		967
	20	2384		1002
	22	2387		927
	24	2354		919
	26	2385		916
	28	2344		923
	30	2350		924

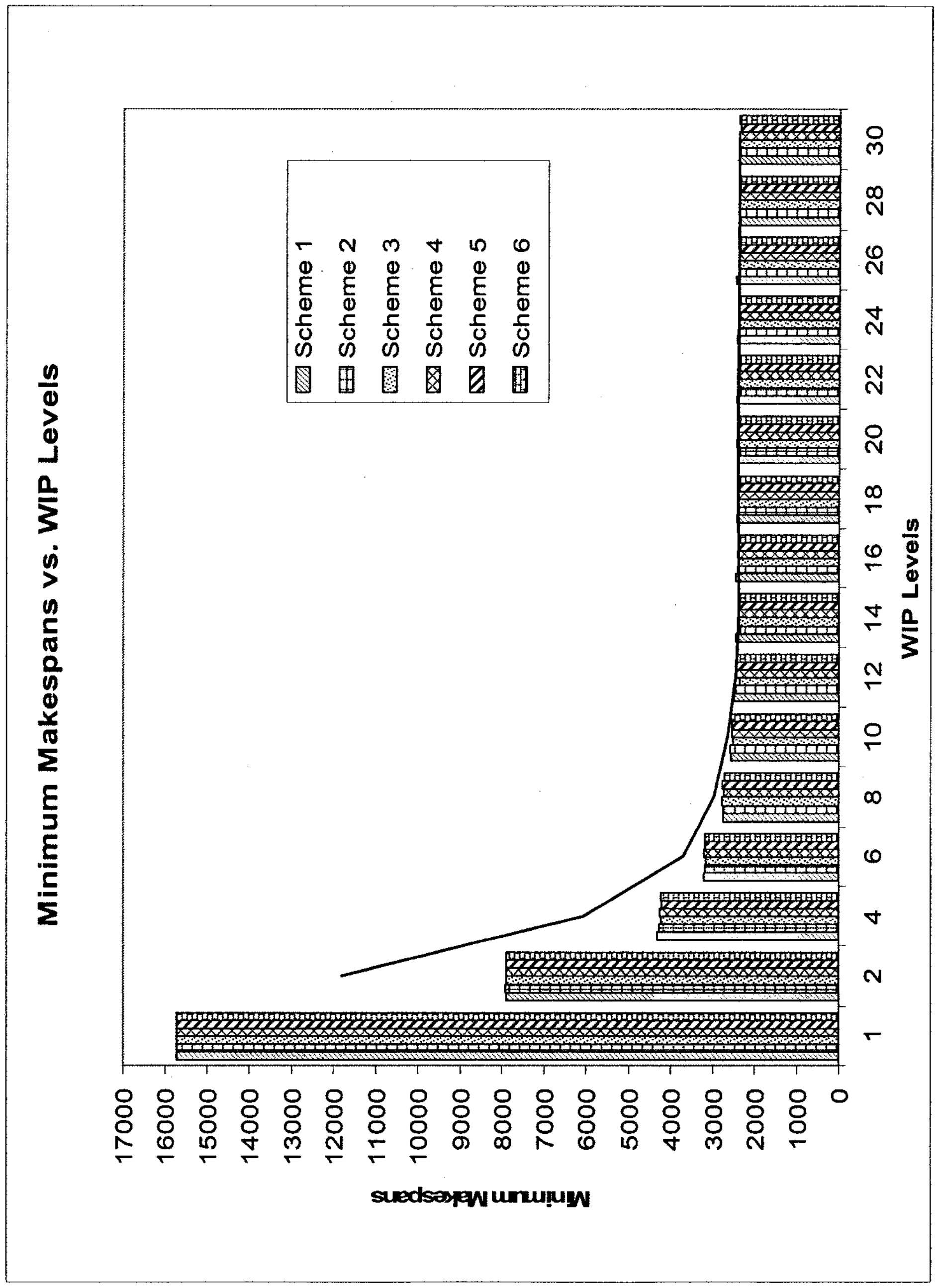


Figure 6.3: Example 3 - Single Serial Line CONWIP System Performance

6.3. MCONL Numerical Examples

The assembly-type multiple line CONWIP model MCONL is solved using the same heuristic method - simulated annealing based heuristic algorithm presented in Chapter 5. In this section, we present two examples of CONWIP based assembly-type systems with several fabrication lines. The layout of the system is similar to the system shown in Figure 4.1.

6.3.1. Example 1

Example 1 problem is an assembly-type multiple line CONWIP system with three fabrication lines feeding an assembly station. In each of the fabrication lines, a number of parts are processed sequentially on a number of machines. The first fabrication line has 4 machines processing 6 parts. The second fabrication line has 6 machines processing 10 parts. The third fabrication line has 3 machines processing 5 parts. The processing time for each part at each machine is listed in Tables B.1, B.2 and B.3 of Appendix B. The processing times are randomly generated from uniform distribution (0, 100). Once they are generated, they are kept the same for the calculation of all WIP inventory levels.

As discussed earlier, the major issue in an assembly-type multiple CONWIP line system is to synchronize the fabrication lines by minimizing the makespan difference among the fabrication lines. To reach this goal, the WIP inventory level and job sequence for each of the fabrication lines are two essential parameters. The purpose of this numerical example is to show how these two parameters can be determined using our presented model MCONL and SA based algorithm.

As presented in Chapter 5, the model MCONL and the proposed SA based heuristic algorithm are implemented in a software package CONLine. The example

Table 6.29: SA Parameters for Multiple Line Example 1

SA Para. Set	Cooling Rate (%)	Iterations for Each Temperature Level	Initial $P(A)$ (%)	Inferiority Base (%)	Initial Temperature	Final Temperature
1	99.5	50	25	5	25	1

1 problem is solved using the CONLine software package. To evaluate the efficiency of different perturbation schemes, all six perturbation schemes are used to compute the makespan difference for this example. For all six perturbation schemes, the SA parameters used in this example are listed in Table 6.29. The WIP inventory levels for fabrication lines 1, 2 and 3 are chosen as 2, 6 and 1, respectively for the six perturbation schemes. Table 6.30 shows the results of the job sequence and the makespan differences for all six perturbation schemes. Figure 6.4 compares the makespan differences for the six perturbation schemes.

The results shown in Figure 6.4 are in favor of scheme 4 and scheme 6 as the least makespan differences are found by schemes 4 and 6 within approximately the same computation time.

As shown in Table 6.30, when the WIP inventory levels of all fabrication lines are set, the job sequence of the fabrication line 2 plays an important role on synchronization of the three fabrication lines and it determines the throughput of the system. This is because line 2 is the slowest line among three fabrication lines. The fastest line in this example problem is the line 3. To have a synchronized CONWIP system, the slowest line should be assigned with large number of containers or set with a higher WIP inventory level; the fastest line should have a small number of

Table 6.30: Results of Multiple Line CONWIP System Example 1

Schemes	Fab - Lines	WIP Levels	Minimal Makespans	Minimal Different Times	Best Job Sequences	CPU Times (seconds)
	1	2	786		P5 P6 P3 P2 P4 P1	
1	2	6	828	168	P9 P2 P8 P4 P3 P5 P7 P1 P6 P10	1755
	3	1	660		P2 P4 P3 P1 P5	
	1	2	745		P6 P5 P2 P3 P4 P1	
2	2	6	816	156	P9 P4 P2 P1 P8 P7 P3 P5 P6 P10	1795
	3	1	660		P3 P1 P5 P4 P2	
	1	2	737		P6 P1 P4 P5 P2 P3	
3	2	6	832	172	P9 P2 P10 P8 P3 P7 P5 P4 P6 P1	1762
	3	1	660		P5 P2 P1 P3 P4	
	1	2	731		P6 P3 P1 P2 P5 P4	
4	2	6	802	142	P9 P4 P2 P8 P7 P1 P3 P5 P6 P10	1781
	3	1	660		P1 P2 P5 P4 P3	
	1	2	779		P4 P2 P6 P3 P5 P1	
5	2	6	821	161	P9 P4 P1 P7 P8 P3 P2 P5 P6 P10	1782
	3	1	660		P1 P4 P2 P5 P3	
	1	2	711		P1 P5 P6 P2 P4 P3	
6	2	6	802	142	P9 P4 P2 P8 P1 P7 P3 P5 P6 P10	1800
	3	1	660		P4 P5 P1 P2 P3	

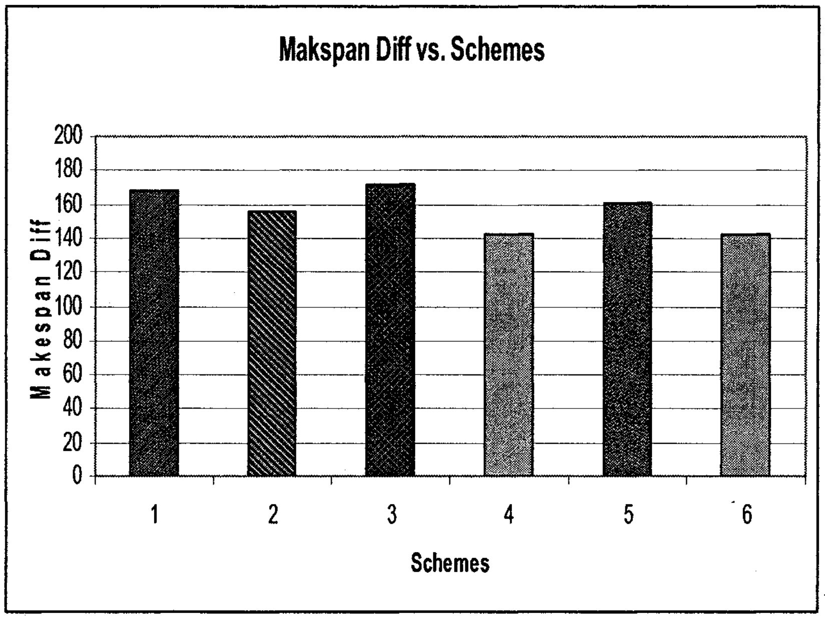


Figure 6.4: CONWIP System with 3 Fabrication Lines

containers or lower WIP inventory level setting.

As mentioned earlier, the WIP inventory level settings for fabrication lines 1, 2 and 3 are chosen as 2, 6 and 1 respectively. But we have not given the reason why these WIP inventory levels were chosen as above. There could be many possible choices of the WIP inventory level setting for the three fabrication lines. Trying all the possibilities is time consuming and not necessary. Below we define a relatively simple way to determine the WIP inventory levels for all fabrication lines.

First of all, we compute the job sequence and the corresponding WIP levels for each individual fabrication line, i.e., each fabrication line is treated as a single serial CONWIP line excluding the final assembly station.

As shown in Figure 6.4, perturbation schemes 4 and 6 give the same best result within an acceptable amount of computation time. So in this case, we simply use scheme 4 to calculate makespans with different WIP inventory levels for each individual line. The SA parameters used here are the same as the one used earlier in Table 6.29.

Figure 6.5 illustrates the system performance for all three fabrication lines. The system performance reflects the relationship between WIP levels and minimum makespan. Since fabrication line 2 has the longest makespan, we call it the critical line. The critical line determines the production rate of the entire assembly system. Figure 6.5 shows that fabrication line 2 reaches its production capacity with six containers circulating in the line. To synchronize the three fabrication lines, the best WIP inventory level for each fabrication line needs to be decided. To do so, we start from critical line 2 and draw a horizonal line as shown in Figure 6.5 from line 2's minimum makespan. The line has intersections with both line 1 and line 3. From the intersection of line 2 and line 1, a vertical line can be drawn to find corresponding WIP level for line 1. From the intersection of line 2 and line 3, we can find the WIP level for line 3 in the same way. The WIP level found by the vertical line may not be an integer. Round the WIP level to its closest integer if needed. In this case, the WIP level or the number of containers for line 1 and line 3 are 2 and 1, respectively. Once the combination of WIP levels for line 1, line 2 and line 3 is determined, this combination can then be feed into the SA based heuristic algorithm to calculate the minimal makespan difference. The makespan difference should be minimized for a synchronized production of the 3 fabrication lines. In this case, the best WIP levels are 2 containers for line 1, 6 containers for line 2, and 1 container for line 3. The resulting WIP levels, part sequence for each fabrication line and the minimum makespan difference are shown in Table 6.30.

The method described above to find the best combination of WIP levels for each fabrication line is able to provide a factory manager with an initial container distribution for the entire system. The combination can be adjusted based on experience in the plant. Using the method presented above, it is easy to identify the critical line and then based on the critical line to find the WIP levels of the other fabrication lines.

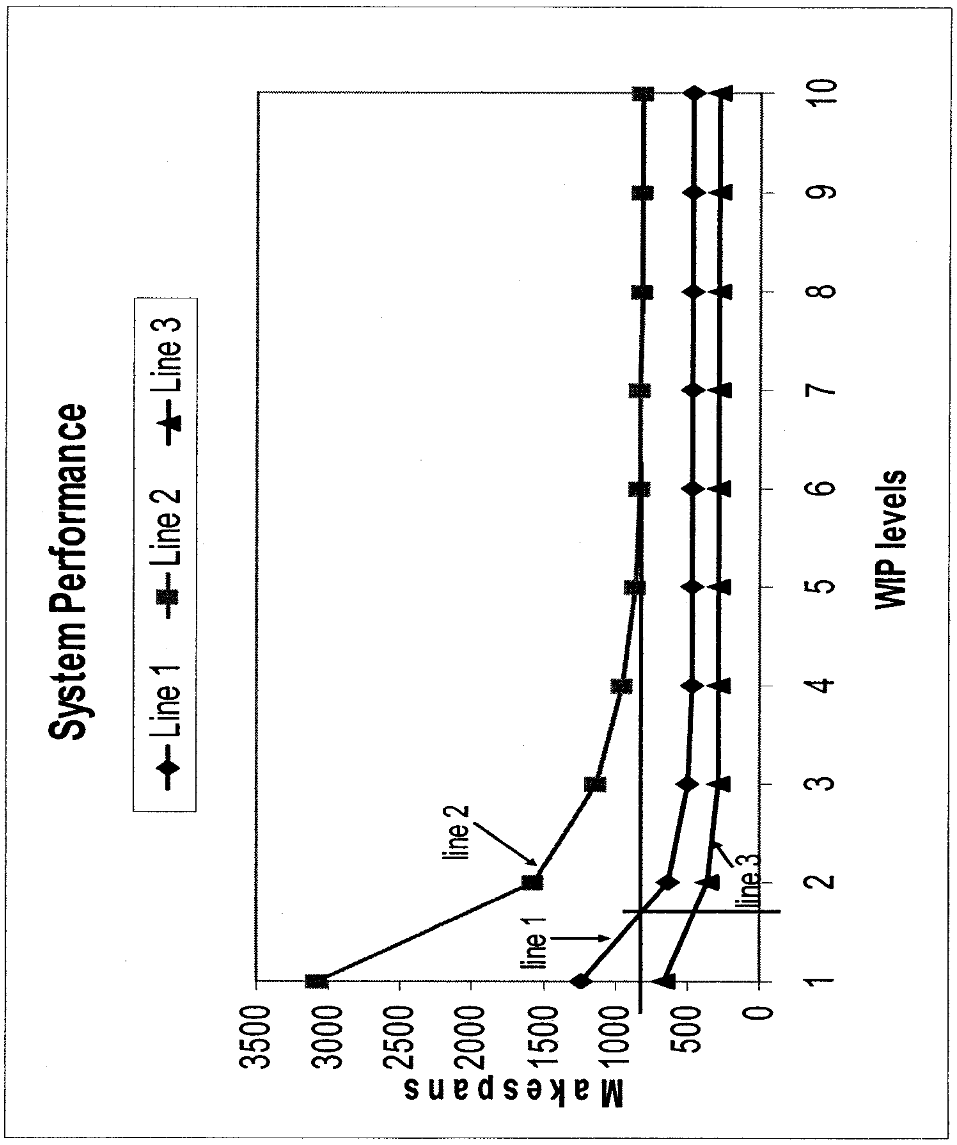


Figure 6.5: Example 1 - CONWIP System Performance

6.3.2. Example 2

In this section, we present another example with more fabrication lines for an assembly-type multiple line CONWIP system. In this example, there are 5 fabrication lines in the system. In each of the fabrication lines, a number of parts are processed sequentially on a number of machines. Fabrication line 1 has 3 machines processing 4 parts; line 2 has 4 machines processing 5 parts; line 3 has 3 machines processing 3 parts; line 4 has 4 machines processing 4 parts; line 5 has 5 machines processing 3 parts. The processing time data for each part in each fabrication line are listed in Tables B.4, B.5, B.6, B.7 and B.8 of Appendix B.

Same as example 1, all six perturbation schemes are used to solve this example for minimizing the makespan difference among the lines. For all six perturbation schemes, the SA parameters used in this example is listed in Table 6.31. The WIP inventory levels for fabrication lines 1, 2, 3, 4 and 5 are chosen as 3, 3, 1, 2 and 2, respectively for the six perturbation schemes.

The result shown in Figure 6.6 does not favor any scheme. Different schemes do not have any impact on determining the makespan difference in this case. The reason behind this is that the total computation iteration numbers determined by the SA parameters are sufficiently large enough for this example problem. There are less parts processed by each of the fabrication lines compared with the example 1 problem even though there are more fabrication lines in this case.

Using perturbation scheme 4 and the SA parameters shown in Table 6.31, makespan versus WIP levels for each fabrication line are calculated by the SA based heuristic algorithm. The results are shown in Figure 6.7.

Critical line and the number of containers or WIP levels are determined in the same way as described in example 1. The critical line in this example is line 1. The

Table 6.31: SA Parameters for Multiple Line Example 2

SA Para. Set	Cooling Rate (%)	Iterations for Each Temperature Level	Initial $P(A)$ (%)	Inferiority Base (%)	Initial Temperature	Final Temperature
1	99	10	25	5	25	1

number of containers are $3, 3, 1, 2, 2$ for line 1 to line 5.

Comparing this example with example 1, we can see in this example that the number of containers among fabrication lines are closer. In this example, among all 5 lines, the maximum number of containers is 3 and the minimum number of containers is 1. This is because workloads among fabrication lines in example 2 system are more balanced. The imbalance of workloads among fabrication lines in example 1 results in a larger difference in the number of containers and larger makespan difference in the system. Thus, balancing workloads among the fabrication lines is also essential in order to synchronize the production and to minimize the overall WIP inventory level in the system. In conclusion, in order to have a better performing assembly-type CONWIP system, not only the WIP level and job sequence for each fabrication line need to be well determined, but also the workloads among the lines need to be well balanced.

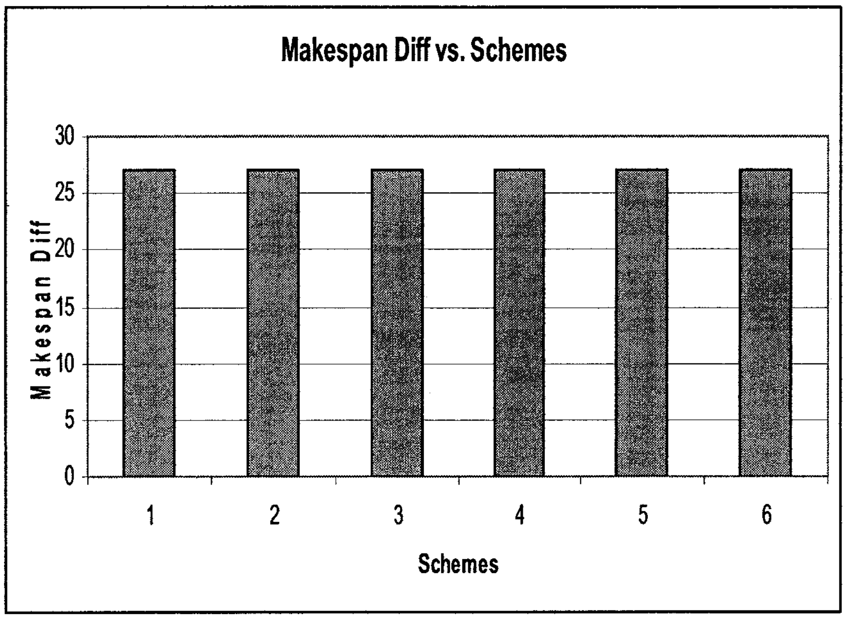


Figure 6.6: CONWIP System with 5 Fabrication Lines

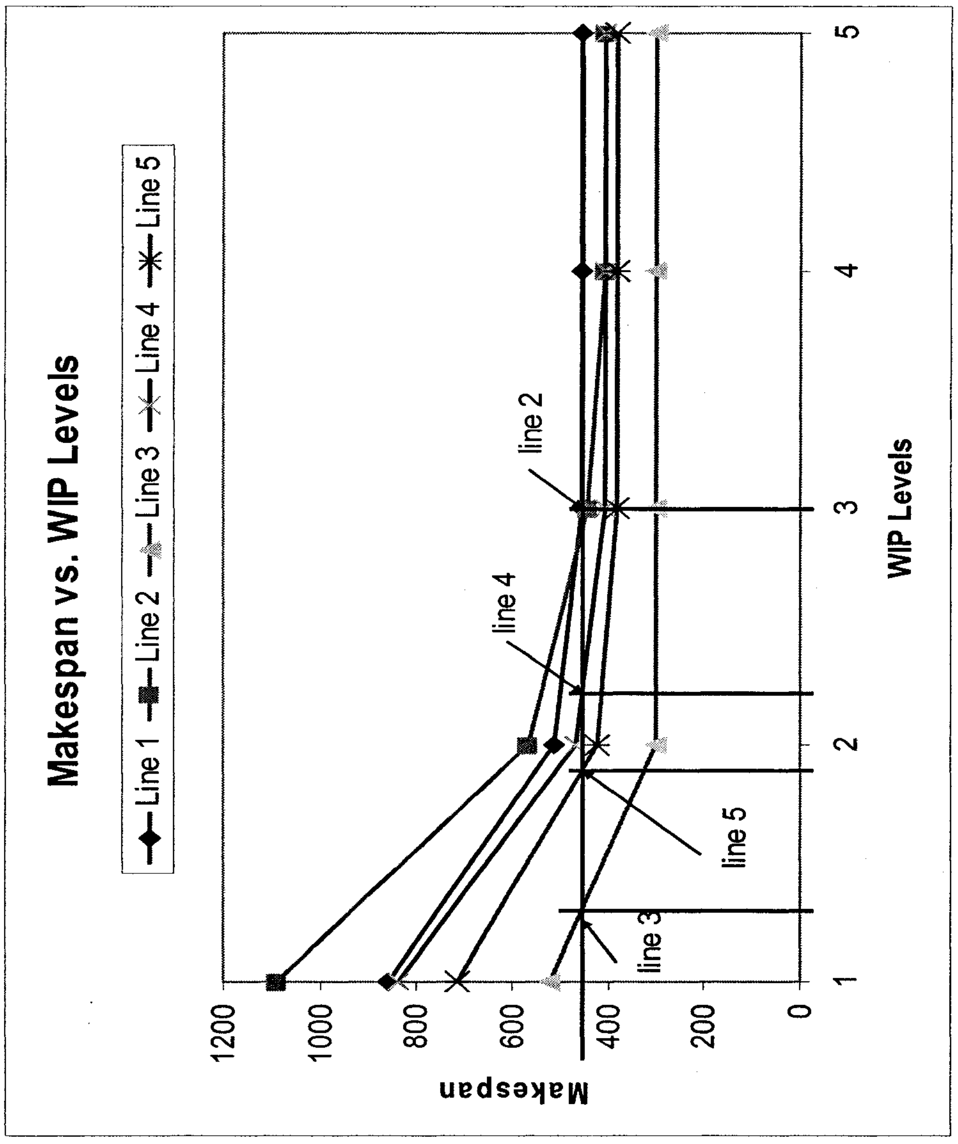


Figure 6.7: Example 2 - CONWIP System Performance

Chapter 7

Conclusions and Future Research

7.1. Overview

Facing ever increasing international competition and pressure for profitability, the manufacturing industry has made profound and fundamental changes to cope with such challenges. Lean manufacturing has become a must for many manufacturing companies. On the other hand, the cost of change and the resistance to change by labor forces in the manufacturing industry are also tremendous. To have a lean operation and to ease the pain of change to a minimum, hybrid production control systems, such as CONWIP systems should be considered by many manufacturing companies in converting their push-based MRP systems to pull-based Just-In-Time systems. Push-based systems have been widely used by many manufacturing companies, but they do not respond well to today's rapid product development and may create difficulties in keeping a manufacturing company competitive. Pull systems are typically lean systems, but may be difficult to implement in many existing manufacturing systems. The CONWIP control approach takes advantages of both push and pull systems and is easy to implement in most existing manufacturing systems.

That is why CONWIP systems have attracted many practitioners and researchers since it was introduced a decade ago. In order to successfully implement a CONWIP system in an existing manufacturing system, the following issues need to be addressed:

1. Job sequence in the backlog list
2. Number of containers used in the system
3. System performance forecast
4. Transfer lot size

To effectively address the above issues, many factors must be considered. These factors include:

1. Single production line or multiple production lines
2. Single product or multiple products
3. Workload balancing
4. Lot splitting
5. Job sequencing
6. Setup time/cost
7. Work-In-Process levels
8. Bottleneck machines

7.2. Summary and Concluding Remarks

In an effort to address the factors listed in the previous section, we developed three mathematical programming models for CONWIP system analysis in this research. Our first model CONL1 presented in Chapter 3.2 was developed for a single serial CONWIP production line. The solution of that model determines the job sequence and the lot sizes for different items processed in a CONWIP line. The model considered sequence dependent setup cost and the cost incurred by workload imbalance. Our review shows that this is the first CONWIP mathematical programming model that considers both sequence dependent setup cost and workload balancing. Also this is the first CONWIP mathematical programming model to address the transfer lot size problems. A numerical example based on a small sized problem was solved by LINGO to demonstrate the model and its potential benefits. Computational experience on such a small problem showed that a significant amount of cost savings can be achieved by considering bottleneck machine workload balancing and lot splitting. Our computational results also showed that there are significant differences on WIP level and makespan if workload balancing is not attempted.

Model CONL1 uses a hybrid control philosophy. It is a hybrid of TOC and CONWIP. In such a control policy, the bottleneck machine has to be identified first. In some manufacturing environments, the bottleneck machine can be identified based on past experience. However, in many manufacturing systems and their operations, identifying bottleneck machines may be difficult. Moreover, bottleneck machines may depend on product mix and can shift from time to time. In addition, in complicated manufacturing systems, there could be more than one bottleneck machine. To overcome this difficulty of identifying bottleneck machines, model CONL2 was developed in Chapter 3. No bottleneck machines need to be identified beforehand in solving the model CONL2 and the number of containers can be determined directly in solving the model. The objective function of CONL2 is to minimize makespan. The focus of model CONL2 is to meet demands in time and at the same time to achieve a minimal WIP level. Numerical examples using small sized and medium sized problems were presented to demonstrate the model. Computational results show a clear relationship between WIP levels and makespan. Our results also show that production makespan cannot be decreased further if the WIP level is over certain limit.

Both model CONL1 and model CONL2 are developed for single serial CONWIP production lines. In practice, multiple line production systems are more prevalent in manufacturing. Our review shows that no mathematical programming model for multiple line CONWIP production system has been seen in published research so far. To this end, a model dealing with an assembly-type CONWIP system with multiple fabrication lines was developed in Chapter 4. The advantage of using CONWIP in assembly-type multiple line systems is that the WIP can be controlled at different levels for each fabrication line in order to synchronize the production of all fabrication lines. The synchronization allows the parts from all fabrication lines to arrive at the assembly station at around the same time. With proper WIP setting and job sequence in each fabrication line, the system makespan is minimized while the WIP level of the entire assembly system is kept at the lowest level. Model MCONL was developed to minimize the overall system makespan difference among fabrication lines and to decide on the number of containers or the WIP setting for each fabrication line. Numerical examples using small sized problems were solved by the proposed SA based heuristic algorithm to illustrate the model. Computational results showed that the performance of an assembly-type CONWIP system with multiple fabrication lines is mainly determined by a critical fabrication line. A critical fabrication line is a line that has the largest WIP setting.

As we mentioned above, three mathematical programming models were developed for CONWIP systems in this research. All these models are NP-hard. In solving such mathematical programming models for large sized problems, a branch and bound based general search algorithm cannot give optimal or near optimal solutions within acceptable computational times. To this end, we developed an efficient heuristic algorithm based on simulated annealing that is presented in Chapter 5. Six different perturbation schemes were incorporated in the algorithm. The impact of different SA parameters on CONWIP system performance and computational efforts were investigated.

7.3. Future Research

7.3.1. CONWIP System Mathematical Programming Models

For future research in terms of modeling, there are some interesting areas where the current models can be extended:

a) This research focused on a single period time planning. So the natural extension of the study would be multi-period planning.

b) In this research, processing times and setup times were assumed known and deterministic. The future study can be extended to have deterministic processing time with random machine failure and repair times.

c) In this research, the WIP level was set to a limited level for a production line or the whole production system. It will be interesting to move the WIP levels down for individual type of parts. This can be modeled by having CONWIP constraints for each part.

7.3.2. Heuristic Algorithm

An effective simulated annealing based heuristic algorithm was developed in this research to solve CONWIP production planning problems. The algorithm can be effectively used in solving large sized and close to real world problems. It would be interesting to develop other heuristic algorithms to solve the models and to compare them with the heuristic algorithm developed in this research. Therefore, developing other efficient and robust solution methods is another interesting topic for future research.

7.4. Publications from This Thesis Research

This section lists several papers published or in preparation for possible publication based on different aspects of this thesis research. They include international journals and conference publications.

- Zhang, W., Chen, M., 2001. A mathematical programming model for production planning using CONWIP, *International Journal of Production Research*, **39**, 2723–2734.

- Zhang, W., Chen, M., 2000. Job Sequencing in CONWIP Production Lines, INFORMS 2000 Annual Meeting, San Antonio, TX, November, 2000.

- Zhang, W., Chen, M., 2007. Simulated Annealing based Heuristic Algorithm for Multiple Production Line CONWIP System, In preparation, to be submitted to the *International Journal of Production Economics*, April 2007.

Bibliography

[1] Alrefaei, M. H. and Andradottir, S., 1999. A simulated annealing algorithm with constant temperature for discrete stochastic approximation. *Management Science*, **45**, 748–764.

[2] Altiok, T. and Shiue, G. A., 2000. Pull-type manufacturing systems with multiple product types. *IIE Transactions*, **32**, 115–124.

[3] Anwar, M. F. and Nagi, R., 1998. Integrated scheduling of material handling and manufaturing activities for just-in-time production of complex assemblies. *International Journal of Production Research*, **36**, 653–681.

[4] Arreola-Risa, A. and DeCroix, G. A., 1998. Make-to-order versus make-to-stock in a production-inventory system with general production lines. *IIE Transactions*, **30**, 705–713.

[5] Battaglia, D. A. and Stella, L., 2006. Optimization through quantum annealing: theory and some applications. *Contemporary Physics*, **47**, 195–208.

[6] Beamon, B. M. and Bermudo, J. M., 2000. A hybrid push/pull control algorithm for multi-stage, multi-line production systems. *Production Planning & Control*, **11**, 349–356.

[7] Billington, P. J., McClean, J. O., and Thomas, L. J., 1983. Mathematical programming approaches to capacity-constrainted MRP systems: review, formulation and problem reduction. *Management Science*, **29**, 1126–1141.

[8] Bitran, G. R., and Chang, L., 1987. A mathematical programming approache to a deterministic Kanban system. *Management Science*, **33**, 427–441.

[9] Bonvik, A. M., and Gershwin, S. B., 1996. Beyond Kanban: creating and analyzing lean shop floor control policies. *1996 Manufacturing and Service Operations Management Conference*, Tuck School of Management, Dartmouth College.

[10] Bonvik, A. M., Dallery, Y., and Gershwin, S. B., 2000. Approximate analysis of production systems operated by a CONWIP finite buffer hybrid control policy. *International Journal of Production Research*, **38**, 2845–2869.

[11] Cao, D. and Chen, M. Y., 2005. A mixed integer programming model for a two line CONWIP-based production and assembly system. *International Journal of Production Economics*, **95**, 317–326.

[12] Celikbas, M., Shanthikumar, J. G. and Swaminathan, J. M., 1999. Coordinating production quantities and demand forecasts through penalty schemes. *IIE Transactions*, **31**, 851–864.

[13] Chang, T. M., and Yin, Y., 1994. Generic kanban system for dynamic environments. *International Journal of Production Research*, **32**, 889–902.

[14] Cheng, T. C., Gupta, J. N. and Wang, G., 2000. A review of flowshop

scheduling research with setup times. *Production and Operations Management*, **9**, 262–279.

[15] Cheng, T. C., and Wang, G., 1998. Batching and scheduling to minimize the makespan in the two-machine flowshop. *IIE Transaction*, **30**, 447–453.

[16] Collins, N. E., Eglese, R. W. and Golden, B. L., 1988. Simulated annealing - an annotated bibliography. *American Journal of Mathematical Management Sciences*, **8**, 209–308.

[17] Dallery, Y., and Liberopoulos, G., 2000. Extented Kanban control system: combining kanban and base stock. *IIE Transactions*, **32**, 369–386.

[18] Dar-el, E. M., Herer, Y. T., and Masin, M., 1999. CONWIP-based production lines with multiple bottlenecks: performance and design implications. *IIE Transactions*, **31**, 99–111.

[19] Deleersnyder, J., Hodgson, T. J., King, R. E., and O'Grady, P. J., 1992. Integrating kanban type pull systems and MRP type push systems: insights from a markovian model. *IIE Transactions*, **31**, 99–111.

[20] Dolgui, A., Finel, B., Guschinsky, N. N., Levin, G. M. and Vernadat, F. B., 2006. MIP approach to balancing transfer lines with blocks of parallel operations. *IIE Transactions*, **38**, 869–882.

[21] Duenyas, I., 1994. Estmating the throuput of a cyclic assembly system. *International Journal of Production Research*, **32**, 1403–1419.

[22] Duenyas, I. and Hopp, W, J., 1992. CONWIP assembly with deterministic processing and random outrages. *IIE Transactions*, **24**, 97–109.

[23] Duenyas, I. and Keblis, M, F., 1995. Release policies for assembly systems. *IIE Transactions*, **27**, 507–518.

[24] Duri, C., Frein, Y. and Lee, H., 2000. Performance evaluation and design of a CONWIP system with inspections. *International Journal of Production Economics*, **64**, 219–229.

[25] Framinan, J. M., Gonzalez, P. L, and Ruiz-Usano, R., 2003. The CONWIP production control system: review and research issues. *Production Planning & Control*, **14**, 255–265.

[26] Gaafar, L. K. and Masoud, S. A., 2005. Genetic algorithm and simulated annealing for scheduling in agile manufacturing. *International Journal of Production Research*, **43**, 3069–3085.

[27] Garey, M. R. and Johnson, D. S., 1979. *Computers and intractability: A guide to the theory of NP-completeness.* Freeman, San Francisco, CA, USA.

[28] Gillard, W. G., 2002. A simulation study comparing performance of CONWIP and bottleneck-based release rules. *Production Planning & Control*, **13**, 211–219.

[29] Glover, F. and Laguna, M., 1986. *Tabu Search.* Kluwer, Boston.

[30] Golany, B., Dar-EL, E., M. and Zeev, N., 1999. Controlling shop floor operations in a multi-family, multi-cell manufacturing environment through constant work-in-process. *IIE Transactions*, **31**, 771–781.

[31] Goldratt, E. M. and Cox, J., 1984. *The Goal.* North River, New York.

[32] Grabowski, J. and Wodecki, M., 2004. A very fast tabu search algorithm for the permutation flow shop problem with makspan criterion. *Computers & Operations Research*, **31**, 1891–1909.

[33] Gstettner, S. and Kuhn, H., 1996. Analysis of production control systems Kanban and CONWIP. *International Journal of Production Research*, **34**, 3253–3273.

[34] Gupta, S. and Krishnan, V., 1998. Product family-based assembly sequence design methodology. *IIE Transactions*, **30**, 933–945.

[35] Haddock, J. and Mittenthal, J., 1992. Simulation optimization using simulated annealing. *Computers & Industrial Engineering*, **22**, 387–395.

[36] Hall, R. W., 1981. *Driving the Produtivity Machine: Production Planning and Control in Japan.* Amer Production & Inventory, Falls Church, VA.

[37] Hajek, B., 1988. Cooling schedules for optimal annealing. *Mathematics of Operations Research*, **13**, 311–329.

[38] Harris, J. H. and Powell, S. G., 1999. An algorithm for optimal buffer placement in reliable serial lines. *IIE Transactions*, **31**, 287–302.

[39] Hasija, S. and Rajendran, C., 2003. Scheduling in flowshops to minimize total tardiness of jobs. *International Journal of Production Research*, **42**, 2289–2301.

[40] He, Q. and Jewkes, E. M., 2000. Performance measures of a make-to-order inventory-production system. *IIE Transactins*, **32**, 409–419.

[41] Hejazi, S. R. and Saghafian, S., 2005. Flowshop-scheduling problems with makespan criterion: a review. *Internaltional Journal of Production Research*, **43**, 2895–2929.

[42] Herer, Y. T., Masin, M., 1997. Mathematical programming formulation of CONWIP-based production lines and relationships to MRP. *International Journal of Production Research*, **35**, 1067–1076.

[43] Hillier, M. S., 2000. Characterizing the optimal allocation of storage space in production line systems with variable processing times. *International Journal of Production Research*, **35**, 1067–1076.

[44] Holland, J. H., 1975. *Adaptation in natural and artificial systems.* The university in Michigan Press, MI, USA.

[45] Hopp, W.J., Roof, M.L., 1998. Setting WIP levels with statistical throughput control (STC) in CONWIP production lines. *International Journal of Production Research*, **36**, 867–882.

[46] Hopp, W. J., Spearman, M. L., 1991. Throughput of a constant work in process manufacturing line subject to failures. *International Journal of Production Research*, **29**, 635–655.

[47] Huang, C., and Kusiak, A., 1998. Manufacturing control with a push-pull approach. *International Journal of Production Economics*, **36**, 251–275.

[48] Huang, M., Wang, D. and Ip, W. H., 1998. Simulation study of CONWIP for a cold rolling plant. *International Journal of Production Economics*, **54**, 257–266.

[49] Ignizio, J. P., 2003. The implementation of CONWIP in semiconductor fabrication facilities. *Future Fab Int.*, 14.

[50] Ishibuchi, M., Misaki, S. and Tanaka, H., 1995. Modified simulated annealing for the flow shop sequencing problems. *European Journal of Operational Research*, **81**, 388–398.

[51] Jans, R. and Degraeve, Z., 2007. Meta-heuristics for dynamic lot sizing: A review and comparison of solution approaches. *European Journal of Operational Research*, **177**, 1855–1875.

[52] Khouja, M., 2000. The economic lot and delivery scheduling problem: common cycle, rework, and variable production rate. *IIE Transactions*, **32**, 715-725.

[53] Kirkpatrick, S., Gelatt, C. D. and Vecchi, M. P., 1983. Optimization by Simulated Annealing. *Science*, **220**, 671-680.

[54] Knolmayer, G., Mertens, P. and Zeier A., 2002. *Supply Chain Management Based on SAP Systems.* Springer, New York, NY 10036.

[55] Knutson, K., Kempf, K., Fowler, J. and Carlyle, M., 1999. Lot-to-order matching for a semiconductor assembly and test facility. *IIE Transactions*, **31**, 1103–1111.

[85] Koh, S. G. and Bulfin, R. L., 2004. Comparison of DBR with CONWIP in an unbalanced production line with three stations. *International Journal of Production Research*, **42**, 391–404.

[57] Kouvelis, P. and Karabati, S., 1999. Cyclic scheduling in synchronous production lines. *IIE Transactions*, **31**, 709–719.

[58] Laarhoven, P., Aarts, E., Lenstra, J., 1992. Job shop scheduling by simulated annealing. *Operations Research*, **40**, 113–125.

[59] Laguna, M., 1999. A heuristic for production scheduling and inventory control in the presence of sequence-dependent setup times. *IIE Transactions*, **31**, 125–134.

[60] Leu, B., 2000. Generating a backlog list for a CONWIP production line: A simulation study. *Production Planning & Control*, **11**, 409–418.

[61] Liaw, C., 1999. Applying simulated annealing to open shop scheduling problem. *IIE Transactions*, **31**, 457–465.

[62] Luh, P. B., Zhou, X. and Tomastik, R. N., 2000. An effective method to reduce inventory in job shops. *IEEE Transactions on Robotics and Automation*, **16**, 420-424.

[63] Lutz, C. M., Davis, K. R. and Sun, M. H., 1998. Determining buffer location and size in production lines using tabu search. *European Journal of Operational Research*, **106**, 301–316.

[64] Macdonald, D., 2006. Lean and mean might not be enough. *The Montreal Gazette*, Saturday, October 14, 2006.

[65] Marek, R. P., Elkins, D. A. and Smith D. R., 2001. Understanding the fundamentals of kanban and CONWIP pull systems using simulation. *Proceedings of the 2001 Winter Simulation Conference*, Arlington, VA, U.S.A.

[66] Martin, A. D., Chang, T. M., Yih, Y. and Kincaid, R. K., 1998. Using tabu search to determine the number of kanbans and lotsizes in a generic kanban system. *Annals of operations Research*, **78**, 201–217.

[67] Matanachai, S. and Yano, C. A., 2001. Balancing mixed-model assembly lines to reduce work overload. *IIE Transactions*, **33**, 29–42.

[68] McMullen, P. R. and Frazier, G. V, 2000. A simulated annealing approach to mixed-model sequencing with multiple objectives on a just-in-time line. *IIE Transactions*, **32**, 679–686.

[69] Moden, Y., 1983. *Toyota Production System.* Industrial Engineering and Management Press, Norcross, GA, USA.

[70] Ogbu, F. A. and Smith, D. K., 1990. The application of the simulated annealing algorithm to the solution of the n/m/Cmax flowshop problem. *Computer & Operations Research*, **17**, 243–253.

[71] Osman, L. H. and Potts, C. N., 1989. Simulated annealing for permutation flow-shop scheduling. *OMEGA*, **17**, 551–557.

[72] Ozdamar, L. and Bozyel, M. A., 2000. The capacitated lot sizing problem with overtime decisions and setup times. *IIE Transactions*, **32**, 1043–1057.

[73] Palaka, K., Erlebacher, S. and Kropp, D. H., 1998. Lead-time setting, capacity utilization, and pricing decisions under lead-time dependent demand. *IIE Transactions*, **30**, 151–163.

[74] Peters, B. A. and McGinnis, L. F., 2000. Modeling and analysis of the product assignment problem in single stage electronic assembly systems. *IIE Transactions*, **32**, 21–31.

[75] Philipoom, P. R., Rees, L. P., and Taylor, B. W., 1996. Simultaneously

determining the number of kanbans, container sizes and final-assembly sequence of products in a just-in-time shop. *International Journal of Production Research*, **34**, 51–69.

[76] Philipoom, P. R., Rees, L. P., Taylor, B. W. and Huang, P. Y., 1990. A mathematical programming approach for determining workcenter lotsizes in a just-in-time system with signal Kanbans. *International Journal of Production Research*, **28**, 1–15.

[77] Powell, S. G., and Pyke, D. F., 1998. Buffering unbalanced assembly systems. *IIE Transactions*, **30**, 55–65.

[78] Price, W., Gravel, M., and Nsakanda, A. L., 1994. A review of optimization models of Kanban-based production systems. *European Journal of Operational Research*, **75**, 1–12.

[79] Qiu, M. M., and Burch, E. E., 1997. Hierarchical production planning and scheduling in a multi-product, multi-machine environment. *International Journal of Production Research*, **35**, 3023–3042.

[80] Rao, P. C. and Suri, R., 2000. Performance analysis of an assembly station with input from maultiple fabrication lines. *Production and Operational Management*, **9**, 283–302.

[81] Reeve, C., 1995. A genetic algorithm for flowshop scheduling. *Computers & Operations Research*, **22**, 5–13.

[82] Rios-Mercado, R. Z. and Bard, J. F., 1999. A branch-and-bound algorithm for permutation flow shops with sequence-dependent setup times. *IIE Transactions*, **31**, 721–731.

[83] Roderick, L. M., Toland, J. and Rodriguez, F. P., 1994. A simulation study of CONWIP versus MRP at Westinghouse. *Computers and Industrial Engineering*, **26**, 237–242

[84] Ryan, S. M. and Choobineh, F. F., 2003. Total WIP and WIP mix for a CONWIP controlled job shop. *IIE Transactions*, **35**, 405-418.

[85] Schragenheim, E. and Ronen, B., 1990. DrumCbufferCrope shop floor control. *Production and Inventory Management Journal*, **31**, 18C22.

[86] Shtub, A., 1999. *Enterprise Resource Planning (ERP): The Dynamics of Operations Management.* Kluwer Academic Publishers, Norwell, MA 02061, USA.

[87] Solimanpur, M., Vrat, P. and Shankar, R., 2004. A neuro-tabu search heuristic for the flow shop scheduling problem. *Computers & Operations Research*, **31**, 2151–2164.

[88] Sox, C. and Gao, Y., 1999. The capaciated lot sizing problem with setup carry-over. *IIE Transactions.* **31**, 173–181.

[89] Spearman, M. L., 1992. Customer service in pull production systems. *Operation Research*, **40**, 948–958.

[90] Spearman, M. L., Hopp, W. J. and Woodruff, D. L., 1996. *A hierarchical control architecture for constant work-in-process (CONWIP) production systems.* Elsevier, New York, NY10010, USA.

[91] Spearman, M.L., Woodruff, D.L., Hopp, W. J., 1990. CONWIP: A pull alternative to Kanban. *International Journal of Production Research*, **28**, 879–894.

[92] Spearman, M.L., Hopp, W. J., Woodruff, D.L., 1989. A hierachical control architecture for constant work-in-process (CONWIP) production system. *Journal of Manufacturing and Operations Management*, **2**, 147–171.

[93] Spearman, M. L., and Zazanis, M. A., 1992. Push and pull production systems: Issues and comparisons. *Operations Research*, **40**, 521–532.

[94] Sridhar, J. and Rajendran, C., 1993. Scheduling in a cellular manufacturing system: A simulated annealing approach. *Internal Journal of Production Research*, **31**, 2927–2945.

[95] Stevenson, M., Hendry, L. C., and Kingsman, B. G., 2005. A review of production planning and control: the applicablility of key concepts to the make-to-order industry. *International Journal of Production Research*, **43**, 869–898.

[96] Swaminathan, J. M., and Tayur, S. R., 1999. Managing design of assemly sequences for product lines that delay product differentiation. *IIE Transactions*, **31**, 1015–1026.

[97] Tekin, E. and Sabuncuoglu, I., 2004. Simulation optimization: A comprehensive review on theory and applications. *IIE Transactions*, **36**, 1067–1081.

[98] Tian, P., Ma, J. and Zhang, D. M., 1999. Application of the simulated annealing algorithm to the combinatorial optimization problem with permutation property: An investigation of generation mechanism. *European Journal of Operational Research*, **118**, 81–94.

[99] Trigeiro, W. W., Thomas, L. J. and McClean, J. O., 1989. Capacited lot sizing with setup times. *Management Science*, **35**, 353–366.

[100] Vallada, E., Ruiz, R. and Minella, G., 2006. Minimizing total tardiness in the m-machine flowshop problem: A review and evaluation of heuristics and metaheuristics. *Computers & Operations Research.*

[101] Wang, C., Xu, L., Liu, X. and Qin, X., 2005. ERP research, development and implementation in China: an overview. *International Journal of Production Research*, **43**, 3915–3932.

[102] Wang, L. and Zheng, D.-Z., 2003. An effective hybrid heuristic for flow shop scheduling. *International Journal of Advanced Manufacturing Technology*, **21**, 38–44.

[103] Weitzman, R. and Rabinowitz, G., 2003. Sensitivity of 'Push' and 'Pull' strategies to information updating rate. *International Journal of Production Research*, **41**, 2057–2074.

[104] Weng, Z. K., 1999. Strategies for integrating lead time and customer-order decisions. *IIE Transactions*, **31**, 161–171.

[105] Yalaoui, F. and Chu, C, 2003. An efficient heuristic approach for parallel machine scheduling with job splitting and sequence-dependent setup times. *IIE Transactions*, **35**, 183–190.

[106] Younes, N., Santos, D. and Maria, A, 1998. A simulated annealing approach to scheduling in a flow shop with multi processors. *Industrial Engineering Research Conference Proceesdings*, Banff, Canada.

[107] Zhang, W. and Chen, M., 2001. A mathematical programming model for production planning using CONWIP. *International Journal of Production Research*, **39**, 2723–2734.

[108] Zhang, Y., Luh, P. B., Yoneda, K., Kano, T. and Kyoya, Y., 2000. Mixed-model assembly line scheduling using Lagrangian relaxation technique. *International Journal of Production Research*, **39**, 2723–2734.

Appendix A

Processing Time Data for Model CONL2 Examples

Table A.1: CONL2 Example 1: Processing Times

Machines	Processing Times					
	Part 1	Part 2	Part 3	Part 4	Part 5	Part 6
Mach 1	88	16	62	59	37	96
Mach 2	85	79	81	99	3	6
Mach 3	49	77	27	53	80	7

Table A.2: CONL2 Example 2: Processing Times

Machines	Processing Times				
	Part 1	Part 2	Part 3	Part 4	Part 5
Mach 1	63	94	52	95	31
Mach 2	52	54	68	55	56
Mach 3	55	7	29	93	24
Mach 4	53	3	84	62	95
Mach 5	85	10	92	63	24
Mach 6	31	76	64	43	3
Mach 7	51	15	39	92	13
Mach 8	71	25	80	40	36
Mach 9	24	37	45	64	72
Mach 10	8	19	99	78	15

Table A.3: CONL2 Example 3: Processing Times

Part	Processing Times on Machines									
Parts	M1	M2	M3	M4	M5	M6	M7	M8	M9	M10
P1	40	22	94	69	9	33	82	92	89	54
P2	51	55	31	54	9	84	59	84	14	45
P3	20	49	12	52	25	50	41	26	16	1
P4	68	93	35	47	38	71	46	60	25	24
P5	37	13	99	33	6	51	92	29	85	57
P6	40	90	52	76	26	89	1	48	8	56
P7	61	18	42	84	75	94	51	48	14	30
P8	11	67	14	94	95	8	52	73	59	88
P9	17	26	41	52	67	99	35	42	90	98
P10	63	9	74	68	94	79	32	42	7	44
P11	82	85	27	23	43	83	34	94	85	66
P12	75	59	99	57	67	67	5	70	59	95
P13	44	43	11	94	83	53	65	31	48	35
P14	69	33	12	70	51	68	73	55	81	58
P15	96	15	99	37	91	46	48	93	25	29
P16	1	29	71	32	2	94	50	27	84	8
P17	61	49	4	53	27	2	71	27	50	70
P18	86	13	82	5	53	26	77	15	81	56
P19	50	98	69	63	2	47	55	6	39	55
P20	43	41	23	84	76	98	92	68	33	21
P21	28	11	78	17	69	74	31	42	74	46
P22	30	8	54	85	50	97	47	22	95	53
P23	85	89	67	60	11	1	65	59	74	98
P24	43	75	52	56	51	46	16	74	57	28
P25	80	14	76	46	62	74	57	98	42	5
P26	79	85	35	52	33	94	36	10	59	19
P27	75	18	24	27	22	94	69	94	2	22
P28	36	36	2	6	75	83	45	14	44	15
P29	92	31	74	12	95	52	78	98	10	44
P30	78	77	70	57	13	25	24	58	78	64

Appendix B

Processing Time Data for for Model MCONL Examples

Table B.1: Example 1: Fabrication Line 1 Parts Processing Times

Machines	Processing Times					
	Part 1	Part 2	Part 3	Part 4	Part 5	Part 6
Mach 1	53	19	7	30	18	70
Mach 2	42	9	80	90	55	39
Mach 3	29	64	31	93	98	29
Mach 4	84	68	82	44	62	14

Table B.2: Example 1: Fabrication Line 2 Parts Processing Times

Machines	Processing Times									
	P1	P2	P3	P4	P5	P6	P7	P8	P9	P10
Mach 1	64	44	46	31	96	99	34	65	17	30
Mach 2	23	77	98	58	43	3	58	16	26	59
Mach 3	94	46	57	4	99	52	30	36	83	41
Mach 4	72	52	97	60	82	27	35	54	50	64
Mach 5	48	91	70	59	36	37	62	84	2	24
Mach 6	18	31	41	36	22	47	32	55	88	17

Table B.3: Example 1: Fabrication Line 3 Parts Processing Times

Machines	Processing Times				
	Part 1	Part 2	Part 3	Part 4	Part 5
Mach 1	8	51	68	31	78
Mach 2	94	70	48	2	46
Mach 3	50	10	10	5	75

Table B.4: Example 2: Fabrication Line 1 Parts Processing Times

Machines	Processing Times			
	Part 1	Part 2	Part 3	Part 4
Mach 1	85	85	77	98
Mach 2	10	68	91	52
Mach 3	68	93	50	70

Table B.5: Example 2: Fabrication Line 2 Parts Processing Times

Machines	Processing Times				
	P1	P2	P3	P4	P5
Mach 1	20	69	39	30	58
Mach 2	5	39	34	57	95
Mach 3	94	94	65	71	34
Mach 4	53	7	95	16	97

Table B.6: Example 2: Fabrication Line 3 Parts Processing Times

Machines	Processing Times		
	Part 1	Part 2	Part 3
Mach 1	70	43	72
Mach 2	79	62	29
Mach 3	76	51	30

Table B.7: Example 2: Fabrication Line 4 Parts Processing Times

Machines	Processing Times			
	P1	P2	P3	P4
Mach 1	98	96	66	57
Mach 2	32	36	64	68
Mach 3	24	56	28	54
Mach 4	27	21	45	50

Table B.8: Example 2: Fabrication Line 5 Parts Processing Times

Machines	Processing Times		
	Part 1	Part 2	Part 3
Mach 1	94	60	54
Mach 2	14	88	7
Mach 3	96	35	53
Mach 2	19	44	14
Mach 3	24	61	40

Appendix C

C# Source code of SA based heuristic algorithm

```
/****************************************************************
 * CONLine: Simulated Annealing Based Heuristic Algorithm to   *
 * calculate job sequence for different WIP levels for         *
 * multi-products single and multiple CONWIP lines.            *
 * *************************************************************/
using System;

using System.Collections.Generic;

using System.ComponentModel;

using System.Data;

using System.Drawing;

using System.Text;

using System.Windows.Forms;

using System.IO;
```

```
namespace CONLine {
    public partial class SquenceForm : Form
    {
        DateTime startTime=new DateTime();
        DateTime endTime=new DateTime();

        /* Class ProcessTime*/
        class ProcessTime
        {
            private int _Line;
            private string _mID;
            private string _pID;
            private int _PTime;


            public ProcessTime()
            {
                _Line= 1;
                _mID = "";
                _pID = "";
                _PTime = 0;
            }

            public ProcessTime(int line,string pid, string mid,
               int pt)
            {
                _Line = line;
                _mID = mid;
                _pID = pid;
                _PTime = pt;
            }
            public int Line
            {
                get
                {
                    return _Line;
                }
                set
                {
                    _Line = value;
                }
            }
```

```
public int PTime
{
    get
    {
        return _PTime;
    }
    set
    {
        _PTime = value;
    }
}
public string sMID
{
    get
    {
        return _mID;
    }
    set
    {
        _mID = value;
    }
}
public int iMID
{
    get
    {
        return (int.Parse(this._mID.Remove(0, 1)) -
           1);
    }
    set
    {
        _pID = "M" + value;
    }
}

public string sPID
{
    get
    {
        return _pID;
    }
    set
    {
        _pID = value;
    }
}
```

```
        public int iPID
        {
            get
            {
                return (int.Parse(this._pID.Remove(0, 1)) -
                    1);
            }
            set
            {
                _pID="P"+value;
            }
        }

    }

    /* Class ListItem */
    public class ListItem
    {
        private int _Line;
        private string _pID;
        public ListItem()
        {
            _Line = 1;
            _pID = "";
        }
        public ListItem(int line, string pid)
        {
            _Line = line;
            _pID = pid;
        }

        public int Line
        {
            get
            {
                return _Line;
            }
            set
            {
                _Line = value;
            }
        }
        public string sPID
        {
            get
            {
                return _pID;
            }
            set
            {
                _pID = value;
            }
        }
        public int iPID
        {
            get
            {
                return (int.Parse(this._pID.Remove(0, 1)) -
                   1);
            }
            set
            {
                _pID = "P" + value;
            }
        }
    }

    public SquenceForm()
    {
        InitializeComponent();
    }

    /* Load main form */
    private void Form1_Load(object sender, EventArgs e)
    {

    }

    /* Save results into a data file */
    private void btnSave_Click(object sender, EventArgs e)
    {
        DialogResult dr;
        dr = dlgSave.ShowDialog();
        if (dr == DialogResult.OK)
        {
            StreamWriter wtr = new
               StreamWriter(dlgSave.FileName, false);
            try
            {
                for (int i = 0; i < listBox1.Items.Count;
                    i++)
                {
                    wtr.WriteLine(listBox1.Items[i]);
                }
            }
            catch (Exception ex)
            {
                MessageBox.Show(ex.Message);
            }
            finally
            {
                wtr.Close();
            }
        }
    }

    /* Main */
    private void btnRun_Click(object sender, EventArgs e)
    {
        if (checkBox1.Checked)
        {
            DialogResult drQuit;
            drQuit = MessageBox.Show("Do you want take
                    Random DATA?", "Confirmation",
                    MessageBoxButtons.YesNo,
                    MessageBoxIcon.Question,
                    MessageBoxDefaultButton.Button2);
            if (drQuit == DialogResult.No)
                return;
        }
        else
        {
            DialogResult drQuit;
            drQuit = MessageBox.Show("Do you want take
                    DATA from file?", "Confirmation",
                    MessageBoxButtons.YesNo,
                    MessageBoxIcon.Question,
                    MessageBoxDefaultButton.Button2);
            if (drQuit == DialogResult.No)
                return;
        }
```

```
checkBox1.Checked = false;
startTime = DateTime.Now;
listBox1.Items.Clear();

/* Initialization */
int   numberLines=15;
int[] numberParts = new int[numberLines];
int[] numberMachines = new int[numberLines];
int[] Ctt = new int[numberLines];
int[] WIP = new int[numberLines];
int[] MinTime = new int[numberLines];
int[] position_MinTime = new int[numberLines];
double T1 = 25;
double TF = 1;
double CR = 90.0;
double Kb = 0.00144;
int timesLoop=2;
int count = 0;
int Ct = 999999;
int Cc = 999999;
int result=999999;
int position_result = 0;
string squ="";

const double EPSILON=0.00001;

for (int i = 0; i < numberLines; i++)
{
    WIP[i] = 100000;
    Ctt[i] = 999999;
    MinTime[i] = 999999;
}

#region Collect Parameters

try
{
    timesLoop = int.Parse(txttimesLoop.Text);
}
catch (Exception ex)
{
    MessageBox.Show(ex.Message);
    txttimesLoop.Focus();
}
```

```
try
{
    T1 = double.Parse(textBoxT1.Text);
}
catch (Exception ex)
{
    MessageBox.Show(ex.Message);
    textBoxT1.Focus();
}
try
{
    TF = double.Parse(textBoxTF.Text);
}
catch (Exception ex)
{
    MessageBox.Show(ex.Message);
    textBoxTF.Focus();
}
try
{
    Kb = double.Parse(textBoxKb.Text);
}
catch (Exception ex)
{
    MessageBox.Show(ex.Message);
    textBoxKb.Focus();
}
try
{
    CR = (double.Parse(textBoxCR.Text) / 100.0);
}
catch (Exception ex)
{
    MessageBox.Show(ex.Message);
    textBoxCR.Focus();
}
int swap = int.Parse(SwapMode.Text);
numberLines =  int.Parse(cboxLines.Text);
#endregion

//=new ProcessTime[numberLines][][];
ProcessTime[,,] P;
int[,,] F;          // = new int[numberLines][][];
int[,,] S;          // = new int[numberLines][][];
int[,,] R;          // = new int[numberLines][][];
```

```
int[,] W;
string[][] minList = new string[numberLines][];
string[][] minList1 = new string[numberLines][];
List<ListItem>[] partsList = new
   List<ListItem>[numberLines];

Random rm = new Random();

#region read data from files
for (int i = 0; i < numberLines; i++)
{
    string FileName = "ProcessTime_" + i + ".txt";
    FileInfo theSource = new FileInfo(FileName);
    try
    {
        StreamReader reader = theSource.OpenText();

        string fileText;

        #region read Number of parts, machines
        try
        {
            fileText = reader.ReadLine();
            numberParts[i] = int.Parse(fileText);

            fileText = reader.ReadLine();
            numberMachines[i] = int.Parse(fileText);

            fileText = reader.ReadLine();
            WIP[i] = int.Parse(fileText)-1;
        }
        catch (Exception ex)
        {
            MessageBox.Show(ex.Message);
            //textBoxDI.Focus();
        }
        finally
        {
            reader.Close();
        }

        #endregion
    }
```

```
          catch (Exception ex)
          {
               MessageBox.Show(ex.Message);
          }

     }
     int MaxtempM=0;
     int MaxtempP=0;
     for (int i = 0; i < numberLines; i++)
     {

          minList[i] = new string[numberParts[i]];
          minList1[i] = new string[numberParts[i]];
          if (numberParts[i] > MaxtempP)
               MaxtempP = numberParts[i];
          if (numberMachines[i] > MaxtempM)
               MaxtempM = numberMachines[i];
     }

     P = new ProcessTime[numberLines,MaxtempP,MaxtempM];
     F = new int[numberLines,MaxtempP,MaxtempM];
     S = new int[numberLines, MaxtempP, MaxtempP];
     R = new int[numberLines, MaxtempP, MaxtempP];
     W = new int[numberLines, MaxtempP];

     for (int i = 0; i < numberLines; i++)
     {
          string FileName = "ProcessTime_" + i + ".txt";
          FileInfo theSource = new FileInfo(FileName);
          try
          {
               StreamReader reader = theSource.OpenText();
               try
               {
                    int j = 0;
                    string fileText;
                    fileText = reader.ReadLine();     //read
                       numberParts
                    fileText = reader.ReadLine();     //read
                       numberMachines
                    fileText = reader.ReadLine();     //read
                       WIP

                    fileText = reader.ReadLine();
                    List<ListItem> temp = new
                       List<ListItem>();
          foreach (string spid in
                  fileText.Split(' ', '\n'))
          {
              if ((spid != "") && (spid != "\n"))
              {
                  ListItem item = new ListItem();
                  item.Line = i;
                  item.sPID = spid;
                  temp.Add(item);
              }
          }
          partsList[i] = temp;

          do
          {
              fileText = reader.ReadLine();
              if ((fileText != "") &&
                  (fileText != null))
              {
                  //char[] split = new char[] {};
                  int ki = 0;
                  string
                     smid=string.Format("{0:D2}",
                  (j + 1));
                  smid = "M" +smid;
                  foreach (string s in
                  fileText.Split(' ', '\n'))
                  {
                      if ((s != "")&&(s != "\n"))
                      {
                          int x = int.Parse(s);
                          P[i, ki, j] =
                          new ProcessTime(i,
                          partsList[i][ki].sPID,
                          smid, x);
                          ki++;
                      }
                  }
                  j++;
              }
          } while (fileText != null);
      }
```

```
            catch (Exception EX)
            {
                MessageBox.Show(EX.Message);
            }
            finally
            {
                reader.Close();
            }
        }
        catch (Exception EX)
        {
            MessageBox.Show(EX.Message);
        }

    }
    #endregion

    #region output data
    for (int i = 0; i < numberLines; i++)
    {
        listBox1.Items.Add("Number of Parts: " +
           numberParts[i] + "\n");
        listBox1.Items.Add("Number of Machines: " +
           numberMachines[i] + "\n");
        listBox1.Items.Add("Work in Process: " +
           (WIP[i]+1) + "\n");
        listBox1.Items.Add("Iteration: " + timesLoop +
           "\n");
        listBox1.Items.Add("\n");
        listBox1.Items.Add("\n----------------------
        ------------------------------------------
        ----------------------\n");

        squ ="Line No.: " +(i+1)+" \n\n";
        listBox1.Items.Add(squ);

        squ="             ";
        for (int ki = 0; ki < numberParts[i]; ki++)
        {
            squ += string.Format("{0:S3}   ",
            P[i, ki, 0].sPID);
        }
        squ += "\n";
        listBox1.Items.Add(squ);
        squ = "";
        for (int m = 0; m < numberMachines[i]; m++)
        {
            squ = P[i, 1, m].sMID + "    ";
            for (int ki = 0; ki < numberParts[i]; ki++)
            {
                string spt=string.Format("  {0:D2}",
                   P[i, ki, m].PTime);
                squ += string.Format("{0:S3}  ", spt);
            }
            squ += "\n";
            listBox1.Items.Add(squ);

        }
        listBox1.Items.Add("\n***********************
        *********************************************
        ***********************\n");
    }
    squ = "";
    #endregion

    count = 0;
    int iterate = 0;
    #region

    /* Simulated Annealing Algorithm */
    while (count <= timesLoop)
    {
        for (int i = 0; i < numberLines; i++)
        {
            for (int j = 0; j < numberParts[i]; j++)
            {
                for (int k = 0; k < numberMachines[i];
                   k++)
                {
                    F[i, j, k] = 0;
                }
            }
            // First Part
            F[i, 0, 0] = P[i, 0, 0].PTime;

            for (int m = 1; m < numberMachines[i]; m++)
            {
                F[i, 0, m] = F[i, 0, m - 1] + P[i,0,
                   m].PTime+1;
            }
```

```
int Max_W = 0;

int kkki = partsList[i][0].iPID;
foreach (ListItem pj in partsList[i])
{
    int kj = pj.iPID;
    if (kkki != kj)
        S[i, kkki, kj] = 0;
    if ((F[i, kkki, numberMachines[i] - 1] <
        (F[i, partsList[i].IndexOf(pj), 0] -
        P[i, partsList[i].IndexOf(pj),
        0].PTime))
        && (partsList[i].IndexOf(pj) != 0))
        R[i, kkki, kj] = 1;
    else
        R[i, kkki, kj] = 0;
}

Max_W = 1;

//other parts except the first one
for (int ki = 1; ki < partsList[i].Count;
   ki++)
{
    if ((Max_W >= WIP[i]) && (ki > WIP[i]))
    {
        F[i, ki, 0] =
        Math.Max(F[i, ki - WIP[i] - 1,
        numberMachines[i] - 1],
        F[i, ki - 1, 0]) + P[i, ki, 0].PTime
            + 1;
    }
    else
    {
        F[i, ki, 0] = F[i, ki - 1, 0] +
                      P[i, ki, 0].PTime + 1;
    }

    for (int m = 1; m < numberMachines[i];
       m++)
    {
        F[i, ki, m] = Math.Max(F[i, ki - 1,
           m],
                      F[i, ki, m - 1]) +
                      P[i, ki, m].PTime + 1;
    }
```

```
int kki = partsList[i][ki].iPID;
foreach (ListItem pj in partsList[i])
{
    int kj = pj.iPID;
    if ((ki > partsList[i].IndexOf(pj))
       && (kki != kj))
        S[i, kki, kj] = 1;
    else
        S[i, kki, kj] = 0;

    if ((F[i, partsList[i].IndexOf(pj),
         numberMachines[i] - 1] <
        (F[i, ki, 0] - P[i, ki,
           0].PTime)) &&
        (ki !=
           partsList[i].IndexOf(pj)) &&
        ((F[i, partsList[i].IndexOf(pj),
         numberMachines[i] - 1]) != 0))
        R[i, kki, kj] = 1;
    else
        R[i, kki, kj] = 0;
}

int[] sum_S = new int[numberLines];
int[] sum_R = new int[numberLines];

sum_S[i] = 0;
sum_R[i] = 0;

int kki1 = partsList[i][ki].iPID;
for (int kj = 0; kj <
   partsList[i].Count; kj++)
{
    int kkj = partsList[i][kj].iPID;
    if (kkj != kki1)
    {
        sum_S[i] += S[i, kki1, kkj];
        sum_R[i] += R[i, kki1, kkj];
    }
}
```

```
        W[i,kki1] = sum_S[i] - sum_R[i] + 1;
        W[i,kkki] = 1;

        if (Max_W < W[i,kki1])
            Max_W = W[i,kki1];
}

squ = "Line No.:  " + (i+1) + "       \n";
//listBox1.Items.Add(squ);
squ += "Iteration No.:  " + (iterate + 1) +
    "  \n";
listBox1.Items.Add(squ);
#region output R S

listBox1.Items.Add("----------------------
S[" + (i + 1) +
    "]----------------------\n");
for (int ci = 0; ci < partsList[i].Count;
    ci++)
{
    string squ0 = "";
    for (int j = 0; j < partsList[i].Count;
        j++)
    {
        squ0 += S[i, ci, j].ToString() +
            "   ";
    }
    squ0 += "\n";
    listBox1.Items.Add(squ0);
    squ0 = "";
}

listBox1.Items.Add("\n");
listBox1.Items.Add("----------------------
R[" + (i + 1) + "]
----------------------\n");

string squ2 = "";
for (int ci = 0; ci < partsList[i].Count;
    ci++)
{
    string squ1 = "";
```

```
              for (int j = 0; j < partsList[i].Count;
                 j++)
              {
                  squ1 += R[i, ci, j].ToString() + "
                      ";
              }
              squ1 += "\n";
              listBox1.Items.Add(squ1);
              squ1 = "";
              squ2 += W[i, ci] + "  ";
          }
          squ2 += "\n";
          listBox1.Items.Add("\n----------------------
          W------------------------\n");
          listBox1.Items.Add(squ2);
          listBox1.Items.Add("Max W=" + Max_W);
          listBox1.Items.Add("\n----------------------
          ---------------------------\n");

          listBox1.Items.Add("\n");
          #endregion

          Ctt[i] = F[i,numberParts[i] - 1,
             numberMachines[i] - 1];
          if (MinTime[i] > Ctt[i])
          {
              MinTime[i] = Ctt[i];
              for (int j = 0; j < numberParts[i]; j++)
              {
                  minList1[i][j] =
                     partsList[i][j].sPID;
              }
              position_MinTime[i] = iterate+1;
          }
      }
      int max = Ctt[0];
      int min = Ctt[0];
      for (int ii = 1; ii < numberLines; ii++)
      {
          if (max < Ctt[ii])
              max = Ctt[ii];
          if (min > Ctt[ii])
              min = Ctt[ii];
      }
```

```
Ct = max - min;

if (result > Ct)
{
    for (int i = 0; i < numberLines; i++)
    {
        for (int j = 0; j < numberParts[i]; j++)
        {
            minList[i][j] =
               partsList[i][j].sPID;
        }
    }
    result = Ct;
    position_result = iterate+1;
}

squ  = "Time Difference Ct=" + Ct + "  \n";
squ += "Minimum Time Difference: " + result + "
   \n";
listBox1.Items.Add(squ);

/* if the test value is smaller than the current
    value */
if (Ct < Cc)
{
    #region

    /* Replace the current value with the
       testing value */
    Cc = Ct;

    /* if the testing value is smaller than the
       best value */
    if (Ct < result)
    {
        /* replace the best result with the
           testing value */
        result = Ct;
    }

    count++; /* increment the counter */
```

```
        /* if the counter >= NMAX */
        if (count > timesLoop)
        {
            T1 = (T1 * CR);
            if (T1 >= TF)
            {
                count = 0;
            }
            else
                count = timesLoop + 1;
        }
        for (int l = 0; l < numberLines; l++)
        {
            squ = "Line No.: "+ (l+1)+ "  Complete
               time of all parts in all machine:
               \n";
            listBox1.Items.Add(squ);
            squ = "";
            for (int i = 0; i < numberParts[l]; i++)
            {
                squ += string.Format("{0:S3}
                   ",P[l, i, numberMachines[l] -
                   1].sPID);
            }
            squ += "\n";
            listBox1.Items.Add(squ);
            squ = "";
            for (int i = 0; i < numberParts[l]; i++)
            {
                squ += string.Format(" {0:D3}    ",
                   F[l, i, numberMachines[l] - 1]);
            }
            squ += "\n";
            listBox1.Items.Add(squ);
            squ = "";
            //listBox1.Items.Add("\n*************
            *************************************
            *******************\n");
            listBox1.Items.Add("\n");
        }
    #endregion
    }
```

```
else
{
    #region
    double DI = (Ct - Cc) / (Cc + EPSILON);
    double PA = -DI / (Kb * T1);
    PA = Math.Exp(PA);
    double rmp = rm.Next(10000) / 10000.0 +
        0.01;
    if (rmp < PA)
    {
        Cc = Ct;
    }

    for (int l = 0; l < numberLines; l++)
    {
        squ ="Line No.: "+ (l+1)+ "  Complete
            time of all parts in all machine:
            \n";
        listBox1.Items.Add(squ);
        squ = "";
        for (int i = 0; i < numberParts[l]; i++)
        {
            squ += string.Format("{0:S3}     ",
                P[l, i, numberMachines[l] -
                1].sPID);
        }
        squ += "\n";
        listBox1.Items.Add(squ);
        squ = "";
        for (int i = 0; i < numberParts[l]; i++)
        {
            squ += string.Format(" {0:D3}    ",
                F[l, i, numberMachines[l] - 1]);
        }
        squ += "\n";
        listBox1.Items.Add(squ);
        squ = "";
        listBox1.Items.Add("\n-----------------
        --------------------------------------
        --------------------------------\n");
        listBox1.Items.Add("\n");
    }
    count++;
    if (count > timesLoop)
    {
        T1 = (T1 * CR);
        if (T1 >= TF)
            count = 0;
        else
            count = timesLoop + 1;
    }
    #endregion
}
listBox1.Items.Add("\n**************************
*************************************************
********************\n");

switch (swap)
{
    #region swap Mode 1*************************
    *******************************************
    case 1:
        {
            for (int i = 0; i < numberLines;
                i++)
            {
                int xi, xj;
                ProcessTime temp = new
                    ProcessTime();
                ListItem item1 = new ListItem();

                xi = rm.Next(numberParts[i]);
                xj = xi + 1;
                if (xj >= numberParts[i])
                    xj = 0;

                item1 = partsList[i][xi];

                partsList[i][xi] =
                    partsList[i][xj];

                partsList[i][xj] = item1;

                for (int k = 0; k <
                    numberMachines[i]; k++)
                {
```

```
                temp = P[i, xi, k];
                P[i, xi, k] = P[i, xj, k];
                P[i, xj, k] = temp;
            }
        }
        break;
    }
#endregion

#region swap Mode 2**********************
****************************************

case 2:
    {
        for (int i = 0; i < numberLines;
           i++)
        {
            int xi, xj;
            ProcessTime temp = new
               ProcessTime();
            ListItem item1 = new ListItem();

            xi = rm.Next(numberParts[i]);
            xj = rm.Next(numberParts[i]);
            while (xi == xj)
            {
                xj =
                   rm.Next(numberParts[i]);
            }

            item1 = partsList[i][xi];
            partsList[i][xi] =
               partsList[i][xj];
            partsList[i][xj] = item1;

            for (int k = 0; k <
               numberMachines[i]; k++)
            {
                temp = P[i, xi, k];
                P[i, xi, k] = P[i, xj, k];
                P[i, xj, k] = temp;
            }
        }
        break;
    }
```

```
#endregion

#region swap Mode 3**********************
****************************************
case 3:
    {
        for (int i = 0; i < numberLines;
           i++)
        {
            int xi, xj;
            ProcessTime temp = new
               ProcessTime();
            ListItem item1 = new ListItem();

            xi = rm.Next(numberParts[i]);
            xj = rm.Next(numberParts[i]);
            while (xi == xj)
            {
                xj =
                   rm.Next(numberParts[i]);
            }

            item1 = partsList[i][xi];
            partsList[i].RemoveAt(xi);
            partsList[i].Insert(xj, item1);

            for (int k = 0; k <
               numberMachines[i]; k++)
            {
                if(xi>xj)
                {
                    temp = P[i, xi, k];
                    for (int x = xi; x > xj;
                        x--)
                        P[i, x, k] =
                        P[i, x - 1, k];
                }
                else
                {
                    temp = P[i, xi, k];
                    for (int x = xi; x < xj;
                        x++)
                        P[i, x, k] = P[i, x
                           + 1, k];
                }
```

```
                        P[i, xj, k] = temp;
                    }
                }
                break;
            }
        #endregion
        #region swap Mode 4**********************
        ****************************************
        case 4:
            {
                for (int i = 0; i < numberLines;
                    i++)
                {
                    int xi, xj, xx;
                    List<ProcessTime>[] temp =
                    new List<ProcessTime>
                    [numberMachines[i]];

                    List<ListItem> item = new
                        List<ListItem>(); ;

                    xi = rm.Next(numberParts[i]);
                    xj = rm.Next(numberParts[i]);

                    while (xi == xj)
                    {
                        xj =
                            rm.Next(numberParts[i]);
                    }

                    int l = 0;
                    if (xi > xj)
                    {
                        l = numberParts[i] - xi +
                            xj+1;
                    }
                    else
                        l = xj - xi+1;

                    for (int x = 0; x < l; x++)
                    {
                        int ix=xi+x;
        if (ix >= numberParts[i])
        {
            ix = ix -
                numberParts[i];
        }
        item.Add(partsList[i][ix]);
    }

    int xii = xi;
    for (int x = 0; x < l; x++)
    {
        if (xii >=
            partsList[i].Count)
        {
            xii = xii -
                partsList[i].Count;
        }
        partsList[i].RemoveAt(xii);
    }

    xx =
       rm.Next(partsList[i].Count);

    int xxx = xx;
    for (int x=0; x<l; x++)
        partsList[i].Insert(xxx++,
            item[x]);

    for (int k = 0; k <
       numberMachines[i]; k++)
    {
        List<ProcessTime> temp0 =
            new List<ProcessTime>();
        for (int x = 0; x < l; x++)
        {
            int ix = x + xi;
            if (ix >=
                numberParts[i])
                ix = ix -
                    numberParts[i];
            temp0.Add(P[i, ix, k]);
        }
        temp[k] = temp0;
    }
```

```
if (xi > xj)
{
    for (int k = 0; k <
       numberMachines[i]; k++)
    {
        for (int x = 0; x <
           numberParts[i] - 1;
           x++)
        {
            int ixj = xj + x+1;
            P[i, x, k] = P[i,
               ixj, k];
        }
    }
}
else
{
    for (int k = 0; k <
       numberMachines[i]; k++)
    {
        for (int x = 0; x <
           numberParts[i]-xj;
           x++)
        {
            int ixi = xi + x;
            int ixj = xj + x+1;
            if (ixj <
               numberParts[i])
                P[i, ixi, k] =
                   P[i, ixj, k];
        }
    }
}

for (int k = 0; k <
   numberMachines[i]; k++)
{
    for (int x = numberParts[i]
       - 1-1; x >=xx ; x--)
    {
        int ixj = x + 1;
        P[i, ixj, k] = P[i, x,
           k];
    }
}
```

```
                    for (int k = 0; k <
                       numberMachines[i]; k++)
                    {
                        for (int x = 0; x < l; x++)
                        {
                            int ix = x + xx;
                            P[i, ix, k] =
                               temp[k][x];
                        }
                     }
                }
                break;
            }
        #endregion
        #region swap Mode 5*********************
        ****************************************
        case 5:
            {
                for (int i = 0; i < numberLines;
                   i++)
                {
                    int xi, xj;
                    List<ProcessTime> temp=new
                       List<ProcessTime>();
                    List<ListItem> item=new
                       List<ListItem>();;

                    xi = rm.Next(numberParts[i]);
                    xj = rm.Next(numberParts[i]);

                    while (xi == xj)
                    {
                        xj =
                           rm.Next(numberParts[i]);
                    }

                    int l = 0;
                    if (xi > xj)
                    {
                        l = numberParts[i] - xi +
                           xj;
                    }
                    else
                        l = xj - xi;
```

```
for (int x = 0; x < l; x++)
{
    int ix = x + xi;
    if (ix >= numberParts[i])
    {
        ix = ix -
           numberParts[i];
    }
    item.Add(partsList[i][ix]);
}

item.Reverse();
for (int x = 0; x < l; x++)
{
    int ix = x + xi;
    if (ix >= numberParts[i])
    {
        ix = ix -
           numberParts[i];
    }
    partsList[i][ix] = item[x];
}
for (int k = 0; k <
   numberMachines[i]; k++)
{
    for (int x = 0; x < l; x++)
    {
        int ix = x + xi;
        if (ix >=
           numberParts[i])
            ix = ix -
               numberParts[i];
        temp.Add(P[i, ix, k]);
    }

    temp.Reverse();
    for (int x = 0; x < l; x++)
    {
        int ix = x + xi;

        if (ix >=
           numberParts[i])
            ix = ix -
               numberParts[i];
        P[i, ix, k] = temp[x];
    }
```

```
                    }
                }
                break;
            }
        #endregion
        #region swap Mode 6**********************
        ****************************************

        case 6:
            {
                for (int i = 0; i < numberLines;
                    i++)
                {
                    int xi, xj, xx;
                    List<ProcessTime>[] temp =
                    new List<ProcessTime>
                    [numberMachines[i]];

                    List<ListItem> item = new
                        List<ListItem>(); ;

                    xi = rm.Next(numberParts[i]);
                    xj = rm.Next(numberParts[i]);

                    while (xi == xj)
                    {
                        xj =
                            rm.Next(numberParts[i]);
                    }

                    int l = 0;
                    if (xi > xj)
                    {
                        l = numberParts[i] - xi + xj
                            + 1;
                    }
                    else
                        l = xj - xi + 1;

                    for (int x = 0; x < l; x++)
                    {
                        int ix = xi + x;
                        if (ix >= numberParts[i])
```

```
    {
        ix = ix -
           numberParts[i];
    }
    item.Add(partsList[i][ix]);
}
item.Reverse();
int xii = xi;
for (int x = 0; x < l; x++)
{
    if (xii >=
       partsList[i].Count)
    {
        xii = xii -
           partsList[i].Count;
    }
    partsList[i].RemoveAt(xii);
}

xx =
   rm.Next(partsList[i].Count);

int xxx = xx;
for (int x = 0; x < l; x++)
    partsList[i].Insert(xxx++,
       item[x]);

for (int k = 0; k <
   numberMachines[i]; k++)
{
    List<ProcessTime> temp0 =
       new List<ProcessTime>();
    for (int x = 0; x < l; x++)
    {
        int ix = x + xi;
        if (ix >=
           numberParts[i])
            ix = ix -
               numberParts[i];
        temp0.Add(P[i, ix, k]);
    }
    temp0.Reverse();
    temp[k] = temp0;
}
```

```
if (xi > xj)
{
    for (int k = 0; k <
       numberMachines[i]; k++)
    {
        for (int x = 0; x <
           numberParts[i] - 1;
           x++)
        {
            int ixj = xj + x +
               1;
            P[i, x, k] = P[i,
               ixj, k];
        }
    }
}
else
{
    for (int k = 0; k <
       numberMachines[i]; k++)
    {
        for (int x = 0; x <
           numberParts[i] - xj;
           x++)
        {
            int ixi = xi + x;
            int ixj = xj + x +
               1;
            if (ixj <
               numberParts[i])
                P[i, ixi, k] =
                   P[i, ixj, k];
        }
    }
}

for (int k = 0; k <
   numberMachines[i]; k++)
{
    for (int x = numberParts[i]
       - 1 - 1; x >= xx; x--)
    {
        int ixj = x + 1;
        P[i, ixj, k] =
        P[i, x, k];
```

```
                                        }
                                    }

                                    for (int k = 0; k <
                                        numberMachines[i]; k++)
                                    {
                                        for (int x = 0; x < l; x++)
                                        {
                                            int ix = x + xx;
                                            P[i, ix, k] =
                                                temp[k][x];
                                        }
                                    }
                                }
                                break;
                            }
                        #endregion
                    }
                    iterate++;
                }
                #endregion
                endTime = DateTime.Now;
                #region
                squ = "Minimum Completed Time of Processing in Each
                    Line";
                listBox1.Items.Add(squ);
                for (int l = 0; l < numberLines; l++)
                {
                    squ = "Line No.: " + (l + 1).ToString() + "
                            \n";
                    //listBox1.Items.Add(squ);
                    squ += "Iteration No.: " +
                        position_MinTime[l]+"\n";
                    listBox1.Items.Add(squ);
                    squ = "";
                    for (int i = 0; i < numberParts[l]; i++)
                    {
                        squ += minList1[l][i].ToString() + "      ";
                    }

                    listBox1.Items.Add(squ);

                    squ = "";
                    squ += "\n";
```

```
            squ += "Minimum Completed Time of Processing: ";
            squ +=MinTime[l].ToString();
            squ += "\n";
            listBox1.Items.Add(squ);
            listBox1.Items.Add("\n------------------------
            -----------------------------------------------
            ----------------\n");
            listBox1.Items.Add("\n");
        }

        squ = position_result.ToString() + ".) Minimum Time
           Difference.";
        listBox1.Items.Add(squ);
        for (int l = 0; l < numberLines; l++)
        {
            squ = "Line No.: " +(l+1).ToString()+"\n";
            listBox1.Items.Add(squ);
            squ = "";
            for (int i = 0; i < numberParts[l]; i++)
            {
                squ += minList[l][i].ToString() + "      ";
            }

            listBox1.Items.Add(squ);
        }
        squ = "";
        squ += "\n";

        squ += "Minimum Time Difference: ";
        squ += result.ToString();
        squ += "\n";
        listBox1.Items.Add(squ);
        listBox1.Items.Add("\n-----------------------------
        -----------------------------------------------------
        ------\n");
        listBox1.Items.Add("\n");
        #endregion
        listBox1.Items.Add("Start Time: " + startTime + "
            " + "End Time: " + endTime);
        partsList = null;
        minList = null;
        P = null;
        F = null;
    }
```

```
        private void checkBox1_CheckedChanged(object sender,
            EventArgs e)
        {
            if (checkBox1.Checked)
            {
                int line = int.Parse(cboxLines.Text);
                Parameters frm = new Parameters(line);
                frm.MdiParent = this.ParentForm;
                frm.Show();
            }
        }
    }
}
```