The Optimization of Line Cycle Time in Printed Circuit Board Assembly

By

WAN Yuk Fong, Solar

A Thesis Submitted for the Degree of Master of Philosophy

DEPARTMENT OF MANUFACTURING ENGINEERING

THE HONG KONG POLYTECHNIC UNIVERSITY

2001

Abstract of thesis entitled **The Optimization of Line Cycle Time in Printed Circuit Board Assembly**'
submitted by **Wan Yuk Fong, Solar**
for the degree of Master of Philosophy
at The Hong Kong Polytechnic University in November 2000

This research addresses the problem of Printed Circuit Board (PCB) assembly in an electronics manufacturing system. In the electronic industry, an assembly line normally has several non-identical component placement machines and the placement times of the different machines for the same component are different. Faced with the global competition, an efficient component placement operation is essential.

This research attempts to determine the allocation of components to placement machines for the optimization of the line cycle time. A mathematical model was constructed to represent the mechanism for determining the optimal line cycle time. Initially the mathematical model was in a non-linear integer minimax type formulation. It was then converted into an integer linear programming format. The Branch-and-Bound (B&B) algorithm was applied to solve the integer linear programming model in this research project. However, the B&B alogrithm was found to have taken a very long time to get the optimal solution and hence a heuristic method, the Tabu Search (TS) heuristic was proposed to solve the problem. The performances of both the B&B algorithm and the TS heuristic were compared. The result showed that the TS heuristic can achieve an acceptable solution with a shorter computational time and less number of iterations while the B&B algorithm can guarantee to arrive at the optimal solution.

Both the B&B algorithm and the Tabu Search procedure are found applicable to determine the optimal line cycle time in PCB assembly efficiently. Moreover, the

cycle time of PCB assembly can be reduced as well as the cost of production by the use of the model and methods presented in the project. A further study is required in order to implement the model and the algorithms developed in this project in a real industrial situation, such as, a graphical user interface.

ACKNOWLEDGMENTS

First of all, the author would like to take this opportunity to express her sincere gratitude to her chief supervisor, Dr. P. Ji, for his strong guidance, invaluable advice and continuous encouragement that made this project to be completed successfully. The gratitude also extends to her colleagues and friends, Mr. C. Y. Cheung, Mr. K. H. Tang and Mr. S. L. Mok, for their thoughtful discussions and on-going supports.

The author would also like to express her acknowledgement to The Hong Kong Polytechnic University for the funding of this research project (Project No.: G-V737)

Last but not least, special thanks must go to her father and mother, for their continuous encouragement and patience during the past two years.

TABLE OF CONTENTS

ABSTRACT i

ACKNOWLEDGMENTS iii

TABLE OF CONTENTS iv

LIST OF FIGURES ix

LIST OF TABLES xi

CHAPTER ONE

INTRODUCTION 1

1.1 PRINTED CIRCUIT BOARD (PCB) ASSEMBLY 1

1.1.1 PCB assembly process *1*

1.1.2 Production Planning of PCB assembly *5*

1.1.3 Component grouping *6*

1.2 OBJECTIVES 8

1.3 RESEARCH SCOPE 9

CHAPTER TWO

LITERATURE REVIEW 11

2.1 INTRODUCTION 11

2.2 REVIEW ON COMPONENT GROUPING AND SETUP STRATEGIES 13

2.2.1 Component grouping in a single machine *13*

2.2.2 Component grouping in multiple machines *14*

2.2.3 Setup strategies for PCB machines *16*

2.2.4 Machine assignment *18*

2.3 LINEAR PROGRAMMING AND INTEGER LINEAR PROGRAMMING 18

2.3.1 Linear programming *19*

2.3.1.1 Interior point algorithm 20

2.3.2 Integer linear programming *21*

2.3.3 Algorithms for integer linear programming *22*

2.3.3.1 Exact solution 22

2.3.3.2 Cutting plane algorithm 23

2.3.3.3 Meta-heuristics 23

2.3.3.3.1 Lagrangian relaxation 24

2.3.3.3.2 Genetic and evolutionary algorithm 25

2.3.3.3.3 Simulated annealing 27

2.3.3.3.4 Artificial neural network 29

2.4 CONCLUSION 30

CHAPTER THREE

MATHEMATICAL MODEL 32

3.1 INTRODUCTION 32

3.2 MODELING OF PCB ASSEMBLY 33

3.2.1 Machine-component relationship *33*

3.2.2 Minimizing the cycle time for the component insertion operation *34*

3.3 CASE STUDY 38

3.4 CONCLUSION 42

CHAPTER FOUR

A BRANCH-AND-BOUND ALGORITHM 43

4.1 INTRODUCTION 43

4.2 COMPUTATIONAL COMPLEXITY IN INTEGER LINEAR PROGRAMMING 44

4.3 LINEAR PROGRAMMING RELAXATION 44

4.4 THE BRANCH-AND-BOUND (B & B) ALGORITHM 46

4.4.1 A survey on the branch-and-bound algorithm 47

4.4.1.1 Depth-first search 51

4.4.1.2 Breath-first search 54

4.4.2 Performance evaluation 57

4.5 NUMERICAL CASE STUDY 59

4.5.1 Solution procedure 60

4.5.2 Result analysis 63

4.6 CONCLUSION 64

CHAPTER FIVE

A TABU SEARCH HEURISTIC 65

5.1 INTRODUCTION 65

5.2 A SURVEY ON THE TABU SEARCH HEURISTIC 66

5.3 THE TABU SEARCH TECHNIQUE 68

5.3.1 Short term memory *69*

5.3.2 Operation parameters *71*

5.3.2.1 Candidate list strategy 71

5.3.2.2 Tabu list 71

5.3.2.3 Tabu tenure 72

5.3.2.4 Aspiration criteria 72

5.3.2.5 Strategic oscillation 73

5.3.3 Long term memory *73*

5.3.4 Significance of the Tabu Search heuristic *74*

5.4 ALGORITHM FOR THE LINE CYCLE TIME DETERMINATION PROBLEM 74

5.4.1 Initialization *75*

5.4.2 The mechanism of tabu search with branching strategy *76*

5.4.3 The mechanism of diversification *77*

5.4.4 Algorithm *77*

5.4.5 Function evaluation *78*

5.5 NUMERICAL CASE STUDY 78

5.5.1 Initialization *79*

5.5.2 Result analysis *80*

5.6 IDENTIFICATION OF CHANGING THE RANGE OF NUMERICAL SETTING IN TS .. 83

5.7 THE TABU SEARCH HEURISTIC VS. THE BRANCH-AND-BOUND ALGORITHM . 88

5.8 CONCLUSION 90

CHAPTER SIX

CONCLUSION 91

6.1 IMPLEMENTATION 91

6.2 CONCLUSIONS 91

6.3 FURTHER INVESTIGATIONS 94

REFERENCES 96

APPENDIX I Line layout for an electronic company 107

APPENDIX II Line configuration 109

APPENDIX III Component placement times 111

APPENDIX IV Source codes for the B&B algorithm in C++ 113

APPENDIX V Source codes for the B&B algorithm in Matlab 130

APPENDIX VI Source codes for the tabu search heuristic in Matlab 147

LIST OF FIGURES

Figure 1.1 Type I SMT manufacturing process sequence (without conventional components) 2

Figure 1.2 Type II SMT manufacturing sequence (with conventional components and devices) 3

Figure 1.3 A typical PCB assembly line 4

Figure 1.4 An example for different component types to be assembled 7

Figure 2.1 The relationship in a PCB assembly system 12

Figure 2.2 (a) Sequence-dependent scheduling (b) Group Setup production method 17

Figure 4.1 The B&B algorithm [Tah75] 50

Figure 4.2 Graphical representation for DFS 51

Figure 4.3 Branching strategy with the DFS in the Branch-and-Bound algorithm 53

Figure 4.4 The branch-and-bound solution tree with DFS 54

Figure 4.5 Graphical representation of BFS 55

Figure 4.6 The branch-and-bound solution tree with BFS 56

Figure 4.7 Computational Implementation of the Branch-and-Bound Algorithm . 58

Figure 5.1 The tabu search short-term memory component [Glo90b] 70

Figure 5.2 General structure of tabu search for the line cycle time determination problem 75

Figure 5.3 Graphical representation for the distribution of the feasible solution occurrence 81

Figure 5.4 Graphical distribution for various feasible solutions 86

Figure 5.5 Range of iterations in obtaining the optimum solution with different settings 88

LIST OF TABLES

Table 1.1 Two combinations to component-machine relationship 8

Table 2.1 Mapping of physical parameters in simulation to combinatorial optimization 28

Table 3.1 The relationship between machines and component types for a PCBA line 33

Table 3.2 Tableau for double-sided PCBA relationship 38

Table 3.3 Tableau form of the numerical case study 39

Table 5.1 Distribution of the feasible solution in 1000 running cycles 81

Table 5.2 Combination of component type-machine relationship to different objective values 82

Table 5.3 Numerical setting range for the backtracking level 83

Table 5.4 Percentage of resulted feasible solution in each numerical setting range 85

Table 5.5 Findings on number of iterations required to obtain the optimal solution 87

CHAPTER ONE

INTRODUCTION

1.1 Printed Circuit Board (PCB) Assembly

The manufacturing of printed circuit boards (PCBs) is a primal activity in many electronics manufacturing companies. Printed circuit board assembly (PCBA) is a process-oriented operation that includes mounting of various types of components on a board with an associated production volume. In order to stay competitive, minimization of the line cycle time is required to increase productivity and responsiveness to the market.

1.1.1 PCB assembly process

In PCB manufacturing, two main electronic assembly technologies have been developed, which are Plated-Through Hole (PTH) Technology, and Surface Mounting Technology (SMT). In most cases, a combination of both technologies is utilized in a single PCB. Several advantages can be achieved with SMT: (1) Smaller component can be made, with leads closer together. (2) Packing densities can be increased. (3) Components can be mounted on both sides of the board. (4) Smaller PCBs can be used for the same electronic system. (5) Drilling of many through holes during board fabrication can be eliminated, but via holes are still required for layers interconnected.

In order to accomplish greater packaging densities and functionality, the

PCB assembly system has an absolute necessity in packing more surface mount devices (SMDs). The type of surface mount design being assembled dictates the sequence of steps in the manufacturing process. There are two major types of surface mount design. Type I has SMDs only without any conventional components on both sides (Figure 1.1). When conventional components coexist with SMDs in Type II, the manufacturing sequence can proceed in two different ways depending on the type of SMDs (Figure 1.2).

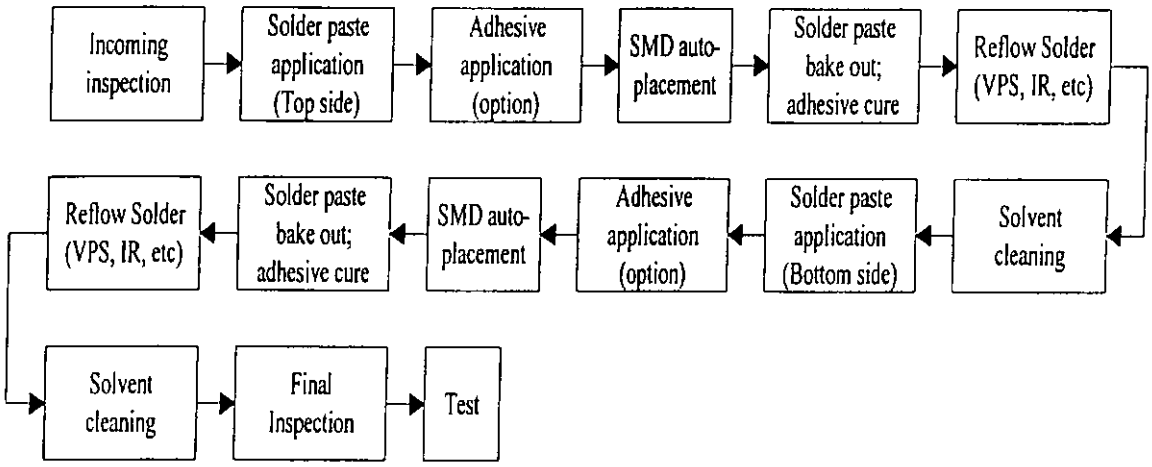


FIGURE 1.1 TYPE I SMT MANUFACTURING PROCESS SEQUENCE (WITHOUT CONVENTIONAL COMPONENTS)

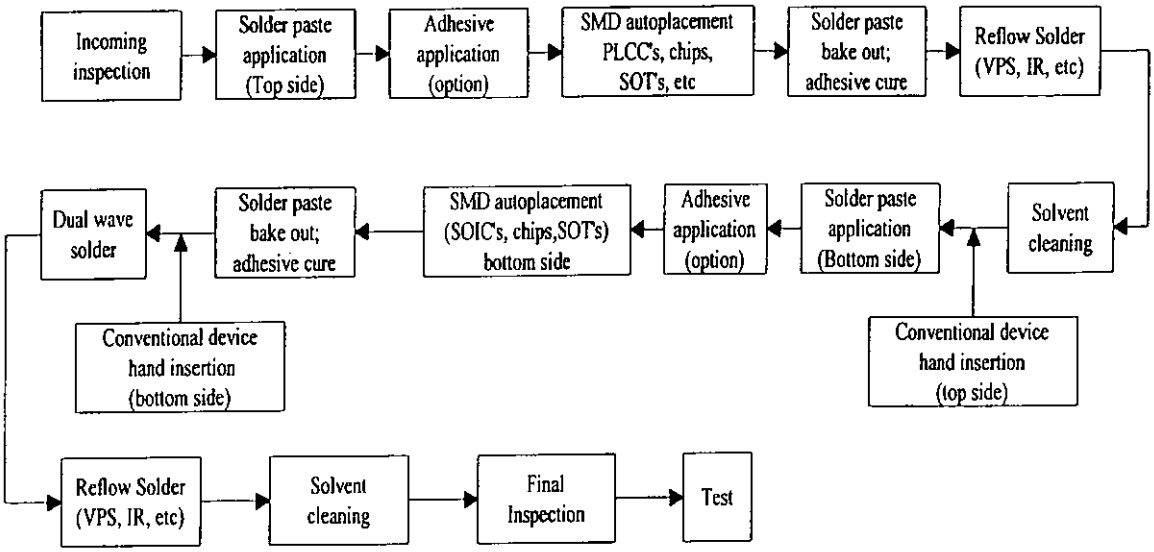


FIGURE 1.2 TYPE II SMT MANUFACTURING SEQUENCE (WITH CONVENTIONAL COMPONENTS AND DEVICES)

A typical PCB assembly line (Figure 1.3) generally consists of six manufacturing operations. (1) Application of solder paste: it can be performed by several techniques: screen (stencil) printing, syringe dispensing and pin transfer techniques. The stencil printing is the most extensively used technique for solder paste application, whereas the syringe dispensing and pin transfer technique are commonly used for adhesive cure application. An accurate decomposition of the solder paste can prevent the solder bridging effect during the reflow soldering. (2) Component placement: The component placement operation becomes significant among the PCB assembly operations for two reasons: it usually proceeds with the most expensive equipment in the PCB assembly line. And it is typically the process that determines the output in term of productivity. (3) Reflow soldering: A reflow soldering allows melting of the soldering and forms the bond between the devices and the board. There are several types of reflow solder techniques to be used: vapor

phase reflow, infrared reflow, dual-wave reflow and laser reflow soldering. (4) Cleaning: It is one of the final assembly stages in PCB assembly. It removes contaminants during the fabrication and previous assembly processes. And these contaminants may prevent electrical contact between the probes on a bed-of-nails test fixture and the electrical assembly. (5) Testing: In-circuit and function level tests are required to verify the package configuration and check whether the board operates in correspondence with the customer's design specifications. (6) Final inspection: This can maintain the high quality and reliability of the PCB before the line releases.

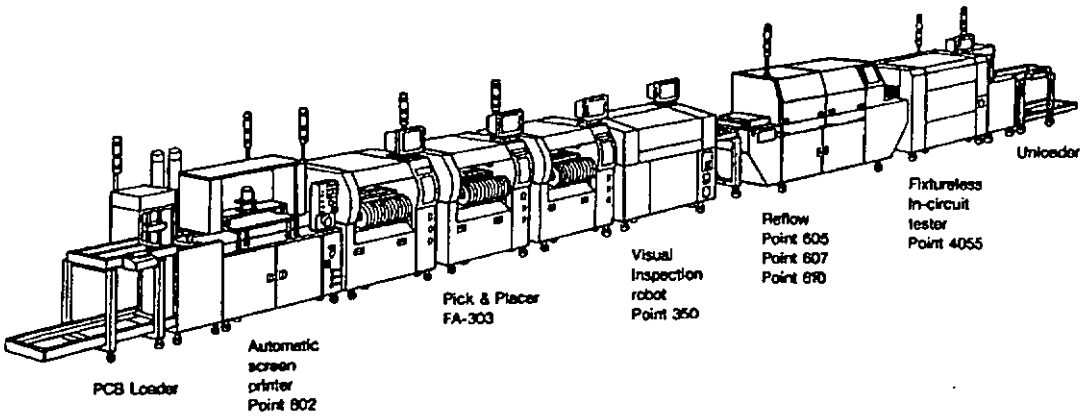


FIGURE 1.3 A TYPICAL PCB ASSEMBLY LINE

Among the assembly operations mentioned above, the component placement process is the most crucial part in enhancing the productivity. This is undeniable that the optimization of the component placement process can elevate not only the productivity but also the utilization of the resources in the factory floor. The occasion for this can be claimed in the following:

- Cost intensive and capacity limitation of the automatic component placement machines may cause the bottleneck of the line. Thus, high utilization of the

machines should be carried.

- The reduction of the production cost can be obtained by the increment of the production volume.

1.1.2 Production planning of PCB assembly

The electronics manufacturing industry is experiencing rapid technological change that causes increasing the logistical complexity on process planning. The task of allocating different types of components to the machines together with balancing the assembly line becomes hard to perform in a good manner concurrently. As a result, the placement time and the operation cost are the critical consideration. Therefore, the optimization on both the product arrangement (which line assembles which board) and line performance (minimum cycle time) are necessary.

The electronics manufacturing factory with low-mix and high-volume and on the surface resembles a flow shop in which the basic production problem is to put the right items together with the right time. Basically, there are two successive tasks in the production planning system. One is the determination of the optimal cycle time and the other one is the decision making on which product to be assembled on which line associated with its quantities to be allocated on the line. The former task is what this research project is going to take into investigation and the latter one has been studied by Ji *et al.*, where the problem was formulated as a generalized transportation problem (GTP) and a new algorithm on the dual of the model was presented [Jip94].

Associated with the interaction between the process planning, production

planning and scheduling, there are different levels of related problems to be tackled. The problems include the grouping of machines and components as well as feeder arrangement and sequencing.

1.1.3 Component grouping

The component grouping problem in a PCB assembly system is a special case of the mixed-model assembly line-balancing problem. It involves the assignment of different components to machines in order to achieve specific production objectives. In reality, a single PCB consists of a number of different types of components, varying from several to several hundreds, in different sizes, shapes and patterns according to the specifications. The decision regarded on how to group different types of components to the machines in the assembly line is made. Assuming that the line has three non-identical placement machines CP I, IP I & IP II (Figure 1.4) and six different components to be assembled. There are many possible combinations of component grouping associated with the machines with different cycle times. The decision making on forming the component group to each machine such that the line cycle time can be minimized, is the key problem to be coped with in this research.

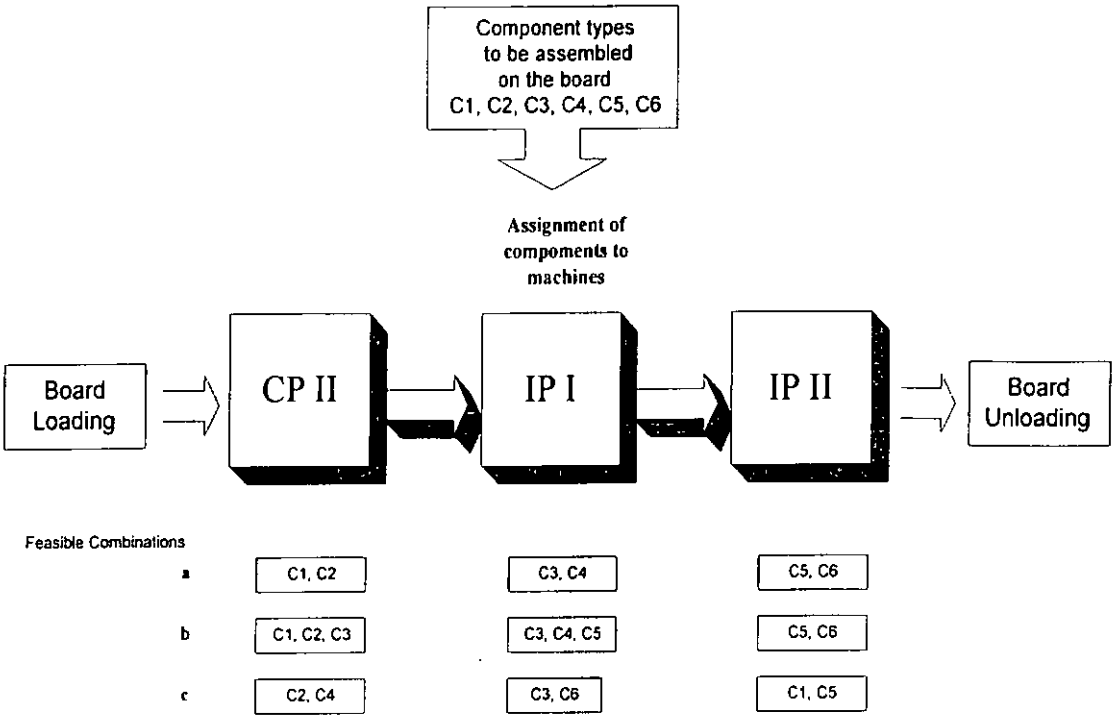


FIGURE 1.4 AN EXAMPLE FOR DIFFERENT COMPONENT TYPES TO BE ASSEMBLED

There are many different combinations of component-machine relationship, which can also obtain the optimal cycle time. The following tables (Table 1.1) shown below demonstrate two of these combinations:

Machine	Component type C_j					
	C1	C2	C3	C4	C5	C6
CP I	20	12	0	0	0	0
IP I	0	0	26	23	0	0
IP II	0	0	0	0	35	38

(a)

Machine	Component type C_j					
	C1	C2	C3	C4	C5	C6
CP I	20	12	5	0	0	0
IP I	0	0	21	23	7	0
IP II	0	0	0	0	28	38

(b)

TABLE 1.1 TWO COMBINATIONS TO COMPONENT-MACHINE RELATIONSHIP

1.2 Objectives

The research aimes at grouping different component types to the existing placement machines, in order to either minimize the line cycle time or balance the workload among machines, which will result in enhancing the productivity.

In the previous research work, a revised simplex method was applied onto

the component placement problem to attain the optimal line cycle time [Sze98]. The rounding method was used to fulfil the integer constraints in the integer linear programming model. However, the solution obtained in this way may not be optimal. Therefore, in this research, a branch-and-bound (B&B) algorithm and a tabu search (TS) heuristic will be studied in order to determine an optimal line cycle time for a large-scale PCB manufacturing company to solve this line cycle time determination problem.

The main objectives of this research study can be summarized as shown below:

- To study the performance of the B&B algorithm with integer programming on determining line cycle time in PCBA.
- To study the performance of Tabu Search heuristic towards the line cycle time determination problem on PCBA.
- To compare the effectiveness of these algorithms.

1.3 Research Scope

With a high production volume, the reduction of line cycle time becomes a critical task in PCB manufacturing in order to increase the productivity. In production planning of PCB assembly, approaches to increasing the productivity are board grouping, component grouping, and sequencing. The board grouping strategy tries to reduce setup time, while component grouping strategy aims at minimizing the assembly cycle time in the line. Therefore, the study on how to group components in order to obtain an efficient line with high throughput rate has to be made.

This thesis is organized on five additional chapters. Chapter 2 reviews the literature concerning component grouping and setup strategies on a PCB assembly system. The review also focuses on algorithms for integer programming as well as mathematical and heuristics techniques developed for those optimization problems. Chapter 3 presents a mathematical model formulated with an objective of minimizing the line cycle time.

The analysis phase of the research begins on Chapter 4, where the integer-programming model solved by the Branch-and-Bound Algorithm will be presented. Based on the mathematical model developed in Chapter 3, the performance evaluation will be made with a case study. A heuristic technique, Tabu Search, will be developed on Chapter 5. This search heuristic technique is revised from the idea of the general tabu search and branching strategy. The performance of this heuristic approach is also discussed in this chapter. A concise conclusion is drawn on the last chapter, which concludes this project and points out some possible work for further investigation.

Chapter Two

Literature Review

2.1 Introduction

For years, vast efforts have been made on the global optimization problem of the throughput rate on printed circuit board assembly (PCBA) systems. Considering a typical component placement operation, there are three basic categories of problems that can be taken into account in improving productivity of a PCBA system. They are (1) determination of which board(s) to be produced on which assembly line, or which boards to be grouped into which assembly line; (2) determination of which machine in the assembly line to assemble which components; (3) sequencing the insertion operations and feeder arrangement of different component types. The hierarchy between these categories was described in Figure 2.1 [Amm97]. Problems in each of these categories are rather complex, so it is difficult to solve all of them simultaneously. A great deal of literature can be found on Category (1) such as [Mai91], [Sht92], [Des95], [San95], [Gar96], [Ask94], and Category (3) such as [Bal88], [Jiz91], [Jiz93], [Cha89], [Nel95] and [Bar94]. However a few studies address problem category (2). Consequently, this research will focus on the line cycle time determination problem, and a detailed review of which will be given in the next sections.

This chapter is divided into two main sections. A detailed review of the component grouping problems on both single and multiple machines, together with several setup strategies to achieve the goal of minimizing the cycle time or

maximizing the throughput rate is described in Section 2.2. Section 2.3 provides a general review on the algorithms used in solving integer linear programming (ILP) models. Two main parts are presented, exact solution algorithms and heuristics.

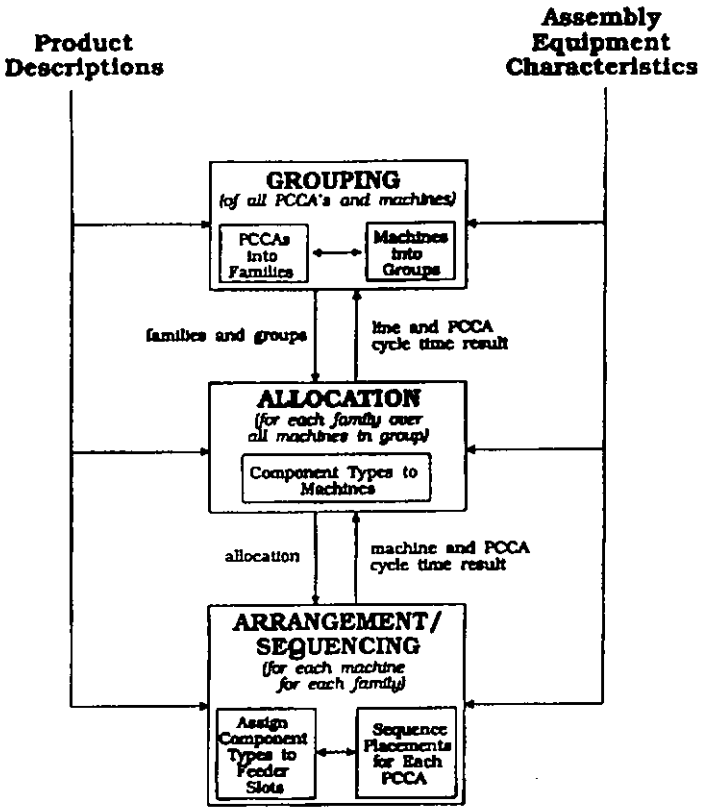


FIGURE 2.1 THE RELATIONSHIP IN A PCB ASSEMBLY SYSTEM

2.2 Review on Component Grouping and Setup Strategies

In the past few years, various research studies have been made on how to group different component types on either single or multiple PCB assembly machines. The major decisions made at the assembly system are the allocation of the components to machine families in order to either minimize the manufacturing cycle time or maximize the throughput rate.

2.2.1 Component grouping in a single machine

When the assembly system includes only a single machine in the assembly line, PCBs may require more than one pass on the machine. Several setups are needed for the insertion of all the required components. Sadiq *et al.* developed a rule-based approach to find a near optimal solution for sequencing a group of PCBs on a single placement machine [Sad93]. Referring to the historical record on the component usage in the database, boards are firstly sequenced on the machine, and then feeders are arranged to these boards to minimize the number of component changes. Maimon and Shtub examined several approaches for partitioning various component types into sets (families) that can be simultaneously loaded on a single machine when machine set-up time and board-loading costs were known [Mai91].

Hiller and Brandeau explored a branch-and-bound (B&B) algorithm in both single machine and multiple machines, using a linear programming (LP) relaxation to obtain lower bounds, and the Lagrangian relaxation to obtain feasible solutions as well as the upper bound [Hil98]. The results from the Lagrangian relaxation heuristic yielded solutions that are very close to the optimal component assignment solution.

Günther *et al.* sub-divided the problem of sequencing PCB assembly jobs into two sub-problems of job sequencing and component set-up, and solved them heuristically [Gün98]. For job sequencing, a heuristic construction was made by choosing the job with the minimum changeover time to each stage of the initial job sequence. The heuristic terminated when no further improvement could be achieved. For the component setup problem, a so-called "keep component needed soonest" (KCNS) policy was applied for the tool exchange problem in a flexible manufacturing system, which sorts those components needed soon for future jobs and kept in the magazine and removed later. The heuristic solution procedure adopted component commonality between PCB types.

Daskin *et al.* utilized the B&B algorithm to minimize a weighted sum of total number of times that PCB types were switched to give the total subset cardinalities on a single-machine assembly system with limited component staging capacity [Das97]. The branching in the B&B algorithm was based on whether two PCBs have to be in the same group or different groups. The heuristic algorithm had the capability of performing single PCB moves and pair-wise swaps of PCBs, based on cost effectiveness.

2.2.2 Component grouping in multiple machines

When the assembly system includes multiple machines, a decision regarding the allocation of component types to an individual machine must be made. Chang and Young explored a new PCB assembly mechanism by using multiple components simultaneously [Cha90]. This mechanism was proposed through the analysis of motion relationship in placing multiple surface mount components simultaneously. A

mathematical model was formulated as a cardinality set covering problem. A heuristic algorithm was proposed to obtain a near optimal component placement strategy. Another heuristic, called GRASP, was presented by Klinecwicz and Rajan in allocation of placement operations to the machines as well as the sequencing of boards in a line [Kli94].

Günther *et al.* delivered the component knitting problem in the semi-automated PCB assembly, concerned about the allocation of the components among various identical assembly stations and took production time and component magazine capacity constraints into account [Gün96]. A heuristic solution containing both job and machine selection was explored. It was stated that the knitting could provide a considerable improvement on overall productivity due to reduced setup time in the assembly shop.

Watkins and Cochran rendered a heuristic-based decision tool to rebalance several product groups on a line by selecting and moving the components from the bottleneck machines to non-bottleneck machines [Wat95]. Consequently, there was a reduction of the component relocation cost.

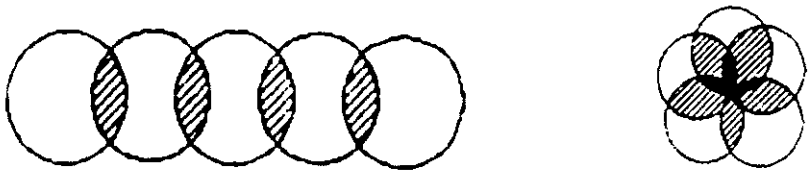
Ammons *et al.* discussed the component allocation problem for an electronic assembly system with multiple, non-identical placement machines in an effort to balance each PCB type with combined placement and setup times across the machines [Amm97]. An integer programming (IP) formulation of the problem was developed, and two alternative solution approaches were presented. One approach is list-processing-based heuristic, which assumes that every machine has the identical processing time in placing any components. However, the actual component placement time is different, depending on the component's configuration. Another approach is the linear programming-based B&B procedure. A conventional LP-based

B&B software package, called MINTO, was used to solve the problem. For the large-scale integer linear programming model, MINTO is not an effective problem-solving tool.

2.2.3 Setup strategies for PCB machines

As the production efficiency is related to the setup time, one of the main approaches employed for reducing the overall set-up time for production is Group Technology (GT). Applying Group Technology (GT) to PCBA gives an advantage of saving setup time and maximizing machine utilization, as no component setup is required when changing from one PCB type to another on boards that share the same components. The concept of GT explores product similarity to minimize the impact of changeover on system performance. Carmon *et al.* firstly introduced a group setup (GSU) (Figure. 2.2(b)) method to reduce the setup time for the PCB assembly [Car89]. The proposed GSU method is somewhat a clustering heuristic based on the group technology approach. The boards were divided into groups, each of which was produced in two stages. At the first stage, the common components were set up on the machines and assembled onto their respective PCBs. At the second stage, the set-up and the assembly of the remaining components were made on each product. By comparing this method with the traditional production approach, the GSU method gave better throughput rate and production makespan. The implementation of the GSU was said to be beneficial in the most high-mix, low volume production environment. A comparison of Sequence Dependent Scheduling (SDS) (Figure 2.2(a)) with the above two methods was made by Maimon *et al.* [Mai91]. The idea underlying the SDS was that PCB types should be scheduled in a way that the

subsequent PCB should have a maximum number of common components as the current PCB. The ultimate goal for the SDS is to minimize the component changes required during the sequence. The two scheduling methods outperformed the traditional production in PCB assembly in terms of setup time and average work in progress (WIP) level. It was found that the GSU scheduling method performed better in line throughput, whereas SDS performed better in terms of average WIP level. The SDS method was found to be superior to the others when the common components are evenly distributed among the PCB types such that the difference in set-up time between the GSU and the SDS is small.



(a) (b)

FIGURE 2.2 (a) SEQUENCE-DEPENDENT SCHEDULING (b) GROUP SETUP PRODUCTION METHOD

McGinnis *et al.* imposed a multi-setup strategy, aiming at minimizing the production time lost due to setup and avoiding excessive work in progress (WIP) [Mcg92]. Two multi-setup strategies were proposed: decompose and sequence (DAS), and partition and repeat (PAR). The first strategy broke the PCB family into smaller subsets and looked for common components to pairs of subsets, and then sequenced the subsets to minimize the incremental setups between them. The second strategy partitioned the components required by the family into subsets such that a group had enough staging capacity for each subset.

With medium-volume and medium variety manufacturing, a partial setup

strategy was proposed by Leon and Peters [Leo96]. The strategy was applied to the optimization of a single-placement machine producing multiple products. Another kind of setup strategy presented by Leon *et al.* was group setup strategy [Leo98]. The group setup strategy was developed on the basis of board grouping and component sequencing. The similar PCBs were grouped into families using hierarchical clustering algorithm, where similar boards were added to the family within the feeder constraint of machines. Component sequencing was another consideration for this strategy.

2.2.4 Machine assignment

Ben-Arieh and Dror examined the two-machine assignment problem with an objective of maximizing the output [Ben92]. Two cases were stated on this problem. The first case assumed that no component type could be inserted by more than one machine. The second was that each type of components can be assigned to both machines. Both cases of the problem were tested with real life industrial setting and the resulting solutions were within 0.5% of optimality.

2.3 Linear Programming and Integer Linear Programming

After reviewing the optimization problem for the PCBA, different techniques were found and a review on several techniques will be provided later in this section. The optimization problem in PCBA is to find an optimal or close to optimal solution for a number of decision variables so that an objective could be minimized (e.g. setup time, number of setups, makespan, line cycle time) or maximized (e.g. throughput,

profit) under certain constraints, such as limited resources. In fact, for the line cycle time determination problem to be studied in this project, the model will be formulated as an integer linear programming (the mathematical model will be described in Chapter 3). So this section discusses several optimization techniques for solving an integer linear programming model.

2.3.1 Linear programming

The development of linear programming (LP) has been amongst the most important scientific advances in recent decades. The LP is the most common type problem in allocating limited resources to activities in an optimal way. Linear programming uses a mathematical model to describe this kind of problems with linear functions. For instance, the optimality conditions put emphasis on the extreme points. The solution methods are typically the simplex method, and the interior point method. The simplex method follows a pathway to a solution through extreme points. It is available for solving LP problems of enormous size. Given a set of m linear inequalities or equations in n variables, that is, the constraints, linear programming (LP) is utilized to find non-negative values of these variables. It can either satisfy all the constraints or optimize the linear function of the variables. The general form of LP model [Bea88]:

$$\min \text{ or } \max \sum_{j=1}^{n} c_j x_j + c_0$$

subject to the constraints:

$$\sum_{j=1}^{n} a_{ij} x_j (\leq, =, \geq) b_i \qquad i = 1, \ldots, m$$

where c_0, c_j, b_i and a_{ij} are constants or known data while x_j are non-negative variables to be determined.

The idea of the simplex method is to make a trip on the polyhedron underlying a linear programming, from vertex to vertex along edges, until an optimal vertex is reached. Two variants of the simplex method are the revised simplex and the dual simplex method.

A revised simplex method applied on the component grouping problem was studied [Sze98]. The resulting solution was found to be continuous instead of integer decision variables. Fixing the non-integer variables by the rounding method, the solution acquired becomes feasible.

2.3.1.1 Interior point algorithm

The introduction of the interior point algorithm shifted the interest towards the interior point or more precise continuous interior trajectories. The basic idea of an interior point algorithm is to enable the method to take long steps, by choosing directions that do not immediately run into the boundary. Karmarkar proposed this polynomial-time interior point algorithm, which produces a procedure that cuts across the interior of the solution space [Kar84]. The effectiveness of the algorithm appears to be in the solution of extremely large LP problems. Comparing both the simplex method and the interior-point algorithm, the latter one is more likely to perform well if certain conditions are met, such as no good initial solution is available, or the problem is degenerate [Nas96].

2.3.2 Integer linear programming

The mathematical model for integer linear programming is the linear programming model with one additional restriction that the value of the decision variables must be integer. In the real-life situation almost all the component assembly problems are integer-programming problems since no fractional part of a single unit of component can be assembled. The integer programming problem is very difficult to solve, in contrast to LP problems, since the computational time for an LP problem is fairly predictable. For an LP, the time increases approximately proportionally with the number of variables and with the number of constraints squared. As the number of integer variables is increased, the computational time for the ILP may increase dramatically. In fact, a general ILP problem is NP-complete and hence generally believed not to be polynomially solvable and confirmed that solving ILP problems were difficult and time consuming [Gar79].

Ammons *et al.* presented a bi-criterion integer-programming model with bin packing heuristic, in allocation of placement operations to machines and sequencing of boards through the shops [Amm85]. Later, Crama *et al.* set up integer programming models on single board type, multiple machines with multiple sequential placement head in a line for throughput rate optimization in PCBA [Cra90]. An IP-based heuristic on optimization of robotic component placement with single board type and sequential single-head machine was developed by Broad *et al.* [Bro96]. Ammons *et al.* discussed the component allocation for an electronic assembly system with multiple, non-identical machines, such that the workload of PCB assembly of each machine for all boards can be balanced [Amm97]. The problem of allocating components to machines was formulated into a large-scale

integer linear programming model.

In some previous research, ILP was solved by LP relaxation, which neglected the integral constraints, and then followed by a rounding solution to acquire the variables with their nearest integer values. However, the resulting solution may or may not be optimal to the problem. Therefore, some algorithms need to be explored in ILP, such as [Kum95], [Cap95].

2.3.3 Algorithms for integer linear programming

2.3.3.1 Exact solution

Various exact solution algorithms were designed to find an optimal solution in a more efficient way than the complete enumeration of the past. Linear programming became the first formulation of scheduling problems with the invention of the Simplex algorithm by Dantzig in 1947 that provided efficient computation [Wil93]. Other important early Operations Research methodologies for Integer Programming are the branch-and-cut and the B&B algorithm [Lan60], [Bar85].

The B&B algorithm directly divides a problem into several subproblems and calculates the lower bound for each. This procedure usually generates a huge tree. The computational complexity of the B&B algorithm is also exponential. Branch and bound methods are therefore limited to less than one hundred activities. Other enumerative methods also suffer from the exponential computational complexity for reasonably large problems.

Unfortunately, most scheduling problems belong to the class of NP-complete problems which are intractable since nobody has shown that a polynomial bounded

algorithm exists for these problems. Exact solution methods are thus of limited practical relevance in obtaining best solutions. Several algorithms are lied within the exact solution, such as the cutting plane algorithm and the B&B algorithm. The review of the former algorithm is presented below. A more detailed review of the B&B algorithm will be given in Chapter 4.

2.3.3.2 Cutting plane algorithm

The cutting plane method is another common approach for ILP, and it is the first systematic technique, developed by Gomory, for the pure integer-programming problem [Gom58]. The algorithm generates extra constraints to "cut out" part of the feasible region in an LP model. These extra constraints can also satisfy all the feasible solutions but violate the optimal solution by the LP relaxation. A family of facet defining inequalities is known for many classes of integer programming such as the travelling salesman problem [Gro91], and the linear ordering problem [Gro84]. However, Taha suggested that when the choice is between the cutting plane method and branch-and-bound method, the latter is generally superior [Tah97]. Nemhauser and Wolsey gave more background on cutting plane methods for integer programming methods [Nem88].

2.3.3.3 Meta-heuristics

Meta-heuristics are techniques which seek good (i.e. near optimal) solutions at a reasonable computational cost without being able to guarantee optimality. Since an exact integer solution needs a relative long computational time, many heuristic techniques such as simulated annealing, genetic algorithms, tabu search and artificial

neural networks have been applied to the operations research field to obtain a near-optimal solution. They are the most recent development of approximate search methods for solving complex optimization problems that arise in manufacturing and business sectors. They have achieved a widespread success in tackling a variety of practical and difficult combinatorial optimization problems, where they are the mathematical study of finding an optimal arrangement, grouping, ordering, or selection of discrete objects with a finite number. In fact, the advantages over an exact algorithm are that they are much faster to execute (i.e. shorter computational time), and have a higher flexibility to deal with many complex problems. As tabu search is selected as a heuristic algorithm to solve the line cycle time determination problem in this research project, it will be discussed separately in Chapter 5.

2.3.3.3.1 Lagrangian relaxation

In the early 1970s, Lagrangian Relaxation was utilized in solving the travelling salesman problem and became a useful technique in generating lower bounds for combinatorial problems. It was defined with respect to the constraint set $Ax \geq b$ by introducing a Lagrange multiplier vector $\lambda \geq 0$ which is attached to the constraint set and brought into the objective function. Ahmadi and Matsuo presented a method to deal with the line segmentation problem in allocating the machines in a multi-stage production line to a number of different 'families' of items [Ahm91a]. A quadratic integer programming using the Lagrangian relaxation, sub-gradient optimization and a Lagrangian heuristic was formulated with the large-scale circuit board manufacturing problem. Noon and Bean applied a Lagrangian heuristic to an asymmetric generalized travelling salesman problem [Noo91]. The problem of assignment operations was examined by Ahmadi and Tang using the Lagrangian

relaxation to minimize the total movement of jobs between machines [Ahm91b]. Campbell and Mabert explored the batch production planning of a number of items on a single machine as a mixed integer programming [Cam91]. Gavish and Pirkul presented an approach to deal with the multi-resources generalized assignment problem with the objective of minimizing the total cost involved in assigning tasks to agents, with each possible task/agent assignment being a vector of resources, and with limited resources available to each agent [Gav91].

2.3.3.3.2 Genetic and evolutionary algorithm

The idea of a genetic algorithm (GA) can be defined as the intelligent exploitation of a random search. The name of the genetic algorithm originates from the analogy between the representation of a complex structure by means of a vector of components and the genetic structure of a chromosome.

A simple genetic algorithm can be described as follows [Gol89]:

Step 1. *Initialization.* Create an initial random population of chromosomes and evaluate each chromosome. Set the current population as an initial population.

Step 2. *Reproduction.* Select two parent chromosomes from the current population. The selection process is stochastic, so that a chromosome with a higher fitness is more likely to be selected.

Step 3. *Crossover.* Generate two offspring from two parent chromosomes by exchanging bit strings (crossover).

Step 4. *Mutation.* Apply a random mutation to each offspring (with a small probability).

Step 5. Repeat steps 2, 3 and 4 until the number of offspring in the new population is the same as the number of chromosomes in the old population.

Step 6. Evaluate each offspring. Set the current population as the new population of offspring and go back to Step 2.

The procedure is repeated for a fixed number of generations, or until no more improvement is observed. In the context of obvious relevance in finding the optimum solution to a large combinatorial problem, a genetic algorithm works by maintaining a population of a number of chromosomes of potential parents, whose fitness values have been calculated.

Major characteristics encountered in the GA are population-based selection, crossover and mutation. The population size directly affects the performance of the algorithm and gives a spatial dimension for selection. Small population size may take the risk of serious under-covering the solution space, while large population may incur severe computational penalty. Meanwhile, practical performance in the real world with extremely large populations, may reduce the competitiveness with other methods such as simulated annealing and tabu search. A population size as small as 30 was suggested in order to have a good output solution. The mutation helps to preserve a reasonable level of population diversity for solving functional optimization problems [Ree93].

By recombining two parent solutions, offspring usually inherit those variables for which both parents share the same value. Thus the recombination essentially results in a reduction of the size of the search space [Ree96].

The algorithm was applied to minimize the movement of the feeder rack (as a surrogate of makespan) in the component sequencing problem by Dikos *et al.* [Dik97]. Other applications of combinatorial problems were bin packing [Fal92], [Ree93], machine sequencing [Ree95], and travelling salesman problem [Whi91].

Some relevant research based on the genetic algorithm was studied by [Kho98a], [Kho98b], [Mai98].

Schaffer et al. derived the application of the genetic algorithm to balance an assembly line of robots that place surface mount devices (SMDs) on printed circuit boards [Sch96]. A component grouping problem using a GA was explored [Sze98].

It was shown that a GA is closer to a neighborhood search and it reduced the neighborhood and then searched in random fashion. Adding local optimization as an extra 'operator' has been found to improve the GA's performance albeit it may have a cost in terms of computational requirements.

2.3.3.3.3 Simulated annealing

Simulated annealing has been used for discrete optimization since early 80s. In the early research work, the implementation of simulated annealing was characterized as a simple and widely applicable heuristic approach. This approach can be regarded as a variant of the well-known heuristic technique of neighborhood search, in which a subset of the feasible solutions is explored by repeatedly moving from the current solution to a neighborhood solution. However, the strategy, in which the search moves in the direction of improvement, resulted in convergence to a local optimum rather than global optimal [Ree93]. Any local optimization algorithm can be converted into an annealing algorithm by random sampling the neighborhoods and allowing the acceptance of an inferior solution according to the probability as follows:

$$P(\delta E) = \exp(-\delta E/kt)$$

where

k is a physical constant known as Boltzmann's constant

t is temperature

δE is an increase in magnitude of energy

The mapping of elements of the physical cooling process onto the elements of a combinatorial optimization problem is shown in Table 2.1:

Thermodynamic simulation	Combinatorial optimization
System states	Feasible solution
Energy	Cost
Change of state	Neighborhood solution
Temperature	Control parameter
Frozen state	Heuristic solution

TABLE 2.1 MAPPING OF PHYSICAL PARAMETERS IN SIMULATION TO COMBINATORIAL OPTIMIZATION.

Many applications using simulated annealing reported in the Operational Research fields involve scheduling or time-tabling problems - particularly production scheduling, for example, determining an optimum sequence for a given set of jobs through a set of machines in order to minimize the makespan. Kuik and Solomon tackled a multi-level problem for the reduction of setup time [Kui90]. Other research using stimulated annealing on a PCBA system was done by Larrhovden and Zijm [Lar93].

Simulated annealing was capable of providing good solutions to some very difficult problems. However, long computational time even to approximate convergence to the optimum, combined with the realization of fine-tuning of the cooling schedule and a judicious choice of neighborhood structure, is needed to get the best out of annealing.

2.3.3.3.4 Artificial neural network (ANN)

Many combinatorial optimization problems were NP-complete. Different heuristic approaches were therefore used to find reasonably good solutions. The artificial neural networks (ANNs) offered a promising solution in the area of mapping the manufacturing features of a component to a sequence of machining operation [Kap92]. In addition, this methodology would ease the knowledge acquisition bottleneck. It can also characterize by their learning ability, providing a promising approach for automated knowledge acquisition. This approach deals better with a non-linear model and creates its own relationship amongst information without the presence of equations. It can also handle the noisy and missing data with good predictive accuracy. The most widely used ANN is the Back Propagation ANN. This type of ANN is excellent in dealing with prediction and classification tasks. Another one is the Kohonen, or Self-Organizing Map, which is excellent at finding relationships amongst complex sets of data. The Kohonen self-organizing neural network is based on somatotopical mapping [Koh89] and has been successfully used for the travelling salesman problem [Ang88]. Since the two-dimensional implementation of the mapping is very similar to the two-dimensional placement problem, the Kohonen's network has been applied to the circuit placement problem by several researchers [Her90], [Rao92].

Placement sequence identification using artificial neural networks in surface mount PCBA was explored by Su and Srihari [SuY96]. Detailed information on ANN can be found on [Zur92]. For a mixed integer combinatorial optimization problem in a power system, an approach combining the feedforward neural network and the simulated annealing method to solve unit commitment was presented by Nayak *et at.* [Nay00]. The type of neural network used in this method is a multi-layer

perception trained by the back-propagation algorithm. The goal is to achieve reduction of the computational time with an optimal generation schedule.

2.4 Conclusion

This chapter provided an extensive literature review on component grouping problems in PCBA together with a brief review on the exact algorithms and heuristic techniques for integer linear programming. Based on the review made above a conclusion can be drawn as follows:

1. A wide range of research studies have been conducted on the optimization problems in PCB assembly under the Category (1) and (3), but only a few studies have addressed the line cycle time determination problem. This, therefore, deserves further study.
2. Computational time is an important efficiency measurement of an algorithm, especially when the problem size becomes large in an integer linear programming.
3. Heuristic techniques provide alternatives to have a near-optimal solution faster in integer programming. The trends for meta-heuristics become significant in solving a combinatorial problem. Several meta-heuristic techniques are presented in the review.

Among the various heuristic techniques mentioned above and the tabu search heuristic which will be discussed later in Chapter 5, they are useful methods for discrete problems around strategies for transcending local optimality. However, they usually accomplish in a problem-specific design in the heuristics. Such that a method, which works well on one problem, may not work well on another. The random start

approach used in simulated annealing, which injects a randomizing element into the generation of an initial starting point, results in longer running time to reach the approximate solution. In addition, the choice of certain parameters like cooling rate and the neighborhood structure, needs to be fine-tuned in order to acquire a better solution. For genetic algorithms, the choice of parents to be matched in each generation is also based on random or biased random sampling of population. It may also need some control mechanism such as crossover and mutation rate, population size, selection mechanism and so on. Furthermore, there is no prescription to indicate how solutions might be combined systematically to achieve such exploitation. The tabu search heuristic approach, by contrast, utilizes penalties and incentives to induce the attributes of a good solution in the neighborhood structure. It is also shown to offer an advantage in ease of implementation and in flexibility to handle additional constraints during the optimization. In this research project, Tabu Search (TS) will be used as a tool to solve the line cycle time determination problem, and a detail survey will be presented later in Chapter 5.

In the next chapter, the line cycle time determination problem is formulated as a mathematical model with the objective of minimizing the total cycle time for machines. Meanwhile, a numerical case study will be demonstrated with the optimal solution, solved by the integer linear programming model.

CHAPTER THREE

MATHEMATICAL MODEL

3.1 Introduction

Constructing a mathematical model is the most effective way to explain real-life optimization problems. The mathematical models exhibit these problems in terms of mathematical representation by translating verbal descriptions of these problems into equivalent mathematical formulation. During the model formulation of the line cycle time determination problem, it is found that certain variables should take the integer value in practice, it is not reasonable to process with a fractional value. Problems in this case are called integer programming (LP).

Integer linear programming (ILP) occurs frequently because many decisions are essentially discrete, having one or more options within a finite set of alternatives. Although there are number of standard "tricks" available to cope with the situations that often arise in formulating IPs, it is probably true to say that formulating IPs is a much harder task than formulating LPs.

In this chapter, the line cycle time determination problem in PCBA is formulated as a mathematical model. The formulation addresses the line cycle time determination problem on assembling n types of components by m machines in the production line. Furthermore, a numerical case study is presented as an integer linear programming. A comparison is made between the rounding LP solution and the pure IP solution.

3.2 Modeling of PCB assembly

3.2.1 Machine-component relationship

In modeling of PCB assembly, suppose n different component types are assembled by m machines (they may or may not be identical) in an assembly line, where n is greater than m. The placement time for machine i to assemble component j is t_{ij} and the set up time (loading and unloading) for machine i is s_i, and the requirements for different component type j are defined by c_j.

The relationship between the machines and different component types can be generalized as follows (Table 3.1):

		Component type (j)					Set up time (s_i)
	t_{ij}	1	2	3	.....	n	
Machine (i)	1	t_{11}	t_{12}	t_{13}	.....	t_{1n}	s_1
	2	t_{21}	t_{22}	t_{23}	.....	t_{2n}	s_2
	3	t_{31}	t_{32}	t_{33}	.....	t_{3n}	s_3
	.	.	.	.	.....	.	.
	.	.	.	.	.....	.	.
	.	.	.	.	.....	.	.
	m	t_{m1}	t_{m2}	t_{m3}	.....	t_{mn}	s_m
Quantity requirement of component j (c_j)		c_1	c_2	c_3	.....	c_n	

TABLE 3.1 THE RELATIONSHIP BETWEEN MACHINES AND COMPONENT TYPES FOR A PCBA LINE

3.2.2 Minimizing the cycle time for the component insertion operation

Primarily, the line cycle time determination problem is in a form of min-max nonlinear type programming. With the introduction of decision variable x_{ij} to indicate how many component j are assigned to machine i, the model is transferred into an Integer Linear Programming (ILP) model. In detail, the problem of maximizing the throughput or minimizing the line cycle time can be formulated as follows:

$$\textit{Minimize} \left[\max \left(\sum_{j=1}^{n} \left(t_{ij} x_{ij} + s_i \right) \middle| i = 1, 2, \ldots, m \right) \right] \tag{3.1}$$

Subject to

$$\sum_{i=1}^{m} x_{ij} = c_j \qquad \text{for } j = 1, 2, \ldots, n \tag{3.2}$$

$$x_{ij} \geq 0 \text{ and integers} \tag{3.3}$$

(LP3-1)

The model LP3-1 is formulated as the minimax type mathematical model of minimizing the cycle time as the objective function (3.1). The constraint set (3.2) guarantees that all components are assembled by machines. However, the type of this model is non-linear and it is difficult to solve. Therefore, a revised model should be constructed to convert the minimax type model into an integer linear programming model as follows:

Minimize

$$T \tag{3.4}$$

Subject to:

$$T - \sum_{j=1}^{n} t_{ij}\, x_{ij} \geq s_i \qquad \text{for } i = 1, 2, \ldots, m \tag{3.5}$$

$$\sum_{i=1}^{m} x_{ij} = c_j \qquad \text{for } j = 1, 2, \ldots, n \tag{3.6}$$

$$x_{ij} \geq 0 \text{ and integers} \tag{3.7}$$

(LP3-2)

In the formulation of LP3-2, a new decision variable T is introduced, which simplifies the objective function of the problem in LP3-1 as well as the model. The new model should meet the requirement for each component type in (3.6) and the integrality constraint, which constrains the entire decision variables to be integer values. Mathematically, LP3-1 and LP3-2 are equivalent, but LP3-2 is an integer linear programming model, and can be solved by general algorithms, such as, the cutting plane method or the branch-and-bound (B&B) algorithm. Besides the objective function, the major consideration for the model LP3-2 is the line cycle time as well as identifying the bottleneck machine.

Model LP3-2 has $nm + 1$ variables and $n + m$ constraints. In general, n is large, for example, 300 component types in a board. If there are 3 machines in a line to assemble these components, LP3-2 will have 303 constraints and 901 variables. So, the formulation is bulky and difficult to handle. Fortunately, however, many components on a board are of the same type.

In order to satisfy the increase in packaging density of the PCB in the recent

electronics manufacturing, different component types are designed to be mounted on both the top and bottom side of a single PCB. For a model with double-sided PCBA, this can be simply decomposed into two single-sided PCB problems with two sets of machine groups.

If n component types on the topside and q component types on the bottom side are required to be mounted, while $j = 1, 2, \ldots, n, n+1, n+2, \ldots, n+q$ is for the component types for both the top and bottom sides. There are two placement stations in a production line (See Appendix I). Station 1 processes the components on the bottom side, while Station 2 processes the components on the top. Station 1 and 2 possess m and p machines, respectively, (i.e. $i = 1, 2, \ldots m, m+1, \ldots, m+p$). The notation for all other decision variables, placement time, setup time as well as the component requirement is the same as above in a single-sided PCB. The model for double-sided PCBA can be generalized as follows:

Minimize

$$\{\max (T_1, T_2)\} \tag{3.8}$$

Subject to

$$T_1 - \sum_{j=1}^{n} t_{ij}\, x_{ij} \geq s_i \qquad \text{for } i = 1, 2, \ldots, m \tag{3.9}$$

$$T_2 - \sum_{j=n+1}^{n+p} t_{ij} x_{ij} \geq s_i \qquad \text{for } i = m+1, m+2, \ldots, m+p \tag{3.10}$$

$$\sum_{i=1}^{m} x_{ij} = c_j \qquad \text{for } j = 1, 2, \ldots, n \tag{3.11}$$

$$\sum_{i=m+1}^{m+p} x_{ij} = c_j \qquad \text{for } j = n+1, n+2, \ldots, n+q \qquad (3.12)$$

$$x_{ij} \geq 0 \text{ and integers} \qquad (3.13)$$

(LP3-3)

Obviously, Model LP3-3 points out that the optimal objective value can be determined by the decomposition method. With two workstations on the line, we can also generate the following machine-component relationship (Table 3.2). The placement times for Station 1 to assemble the components on the top side and for Station 2 to assemble the components on the bottom side are assigned to be infinite (∞). This means that the machines in Station 1 are unable to process any components at the top side, and vice versa.

		Components on bottom side				*Components on top side*				*set up*
		1	2	.....	*n*	*n*+1	*n*+2	.....	*n*+*q*	*time*, s_i
Machines in station one	1	t_{11}	t_{12}	.....	t_{1n}	∞	∞	.....	∞	s_1
	2	t_{21}	t_{22}	.....	t_{2n}	∞	∞	.....	∞	s_2
	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮
	m	t_{m1}	t_{m2}	.....	t_{mn}	∞	∞	.....	∞	s_m
Machines in station two	*m*+1	∞	∞	.....	∞	$t_{(m+1)(n+1)}$	$t_{(m+1)(n+2)}$	.....	$t_{(m+1)(n+q)}$	s_{m+1}
	m+2	∞	∞	.....	∞	$t_{(m+2)(n+1)}$	$t_{(m+2)(n+2)}$	.....	$t_{(m+2)(n+q)}$	s_{m+2}
	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮
	m + *p*	∞	∞	.....	∞	$t_{(m+p)(n+1)}$	$t_{(m+p)(n+2)}$	.....	$t_{(m+p)(n+q)}$	s_{m+p}
Component Quantity c_j		c_1	c_2	.....	c_n	c_{n+1}	c_{n+2}	.....	c_{n+q}	

TABLE 3.2 TABLEAU FOR DOUBLE-SIDED PCBA RELATIONSHIP

3.3 Case Study

Since a double-sided PCBA model can be simplified as two single-sided models, a case study exemplifying the mathematical formulation of a single-sided PCB is presented. The numerical case study illustrates how the model LP3-2 is implemented in grouping the component types into a machine such that workload among the machines can be balanced in the line. In this case, the line consists of 3 machines and 6 different component types associated with their total quantity of 478 to be assembled (i.e. $m = 3$ and $n = 6$). The relative placement times and setup times are illustrated in Table 3.3 (in 0.1 seconds). The infinity (∞) is given to machine i, if it is unable to process component j. In practice, a significant large value (e.g. 90000) is assigned in the mathematical model.

Machine i	Component type j						Setup time s_i
	1	2	3	4	5	6	
1	3	7	7	∞	∞	∞	110
2	7	12	17	24	17	24	147
3	23	38	35	38	38	36	147
Quantity requirement of Component c_j	321	67	35	12	31	12	

TABLE 3.3 TABLEAU FORM OF THE NUMERICAL CASE STUDY

The line cycle time determination problem in the form of LP3-2 can be represented as follows:

Minimize

T

Subject to

1) $T - 3x_{11} - 7x_{12} - 7x_{13} - 90000x_{14} - 90000x_{15} - 90000x_{16} >= 110$
2) $T - 7x_{21} - 12x_{22} - 17x_{23} - 24x_{24} - 17x_{25} - 24x_{26} >= 147$
3) $T - 23x_{31} - 38x_{32} - 35x_{33} - 38x_{34} - 38x_{35} - 36X_{36} >= 147$
4) $X_{11} + X_{21} + X_{31} = 321$
5) $X_{12} + X_{22} + X_{32} = 67$
6) $X_{13} + X_{23} + X_{33} = 35$
7) $X_{14} + X_{24} + X_{34} = 12$
8) $X_{15} + X_{25} + X_{35} = 31$
9) $X_{16} + X_{26} + X_{36} = 12$
10) T, $x_{ij} \geq 0$ and integers. for i = 1, 2, 3 , j = 1, 2,.., 5, 6

The model is then solved by relaxing integer requirement as a linear programming model. By using a commercial software package, LINDO, the result is generated as follows:

LP OPTIMUM FOUND AT STEP 14

OBJECTIVE FUNCTION VALUE

1) 1328.562

VARIABLE	VALUE	REDUCED COST
T	1328.562378	0.000000
X11	321.000000	0.000000
X12	1.508918	0.000000
X13	35.000000	0.000000
X14	0.000000	48793.671875
X15	0.000000	48793.671875
X16	0.000000	48793.953125
X21	0.000000	0.587396
X22	65.491081	0.000000
X23	0.000000	1.581451
X24	0.000000	2.214031
X25	23.274673	0.000000
X26	0.000000	2.497027
X31	0.000000	1.627824
X32	0.000000	1.581451
X33	0.000000	1.156956
X34	12.000000	0.000000
X35	7.725327	0.000000
X36	12.000000	0.000000

With the rounding method (i.e. the value decision variables are rounded to the nearest integer) used in obtaining the optimum solution in this example, a comparison is made between the result obtained in this way and the solution from the pure integer programming. The pure IP solution is also obtained from LINDO with the integer requirement. The results are as follows:

Variables	Value (Rounding method)	Value (Pure integer programming)
x_{11}	321	319
x_{12}	2	3
x_{13}	35	35
x_{14}	0	0
x_{15}	0	0
x_{16}	0	0
x_{21}	0	1
x_{22}	65	64
x_{23}	0	0
x_{24}	0	0
x_{25}	23	24
x_{26}	0	0
x_{31}	0	1
x_{32}	0	0
x_{33}	0	0
x_{34}	12	12
x_{35}	8	7
x_{36}	12	12
T	1339	1333

The optimum cycle time in the pure integer-programming model for the above formulation is 133.3 seconds. However, there was a 0.6 seconds difference for the rounding method. For a high volume production on this PCB (e.g. 10000 units produced in a batch), there is 6000 seconds difference. The production line will become inefficient and the cost of production will be high and some profit will be lost.

3.4 Conclusion

In order to present the line cycle time determination problem in PCBA in the form of mathematical representation, a mathematical model of the problem was formulated in this chapter. A model based on the minimax type was developed initially. This model was the evolution from an integer non-linear programming into a general integer linear programming (ILP). Later, a general ILP model with an additional decision variable T was presented, making it possible to reduce the problem size. In addition, an optimization model for the double-sided PCB assembly was further developed. The model can then be sub-divided into two independent sub-problems with less computational time required than with one model formulation. In this research, the techniques proposed in solving the line cycle time determination problem in PCBA are based on the simplified model, i.e. LP3-2. In the next chapter, a branch-and-bound algorithm will be applied to solving the model developed.

CHAPTER FOUR

A Branch-and-Bound Algorithm

4.1 Introduction

An integer linear programming (ILP) model involves maximizing or minimizing a linear expression subject to a set of linear constraints. Either pure or mixed integer variables are restricted to take integer values. If variables are considered as representing the coordinates of the points in the multi-dimensional space, then the constraints correspond to the region of the space. Many ILP algorithms rely on solving the LP relaxation by neglecting the integer constraints on the variables at the beginning. Different ILP representations of the same problem may result in different LP relaxation.

So far, the strategy to be adopted in integer programming is in a sense of minimizing the degradation in objective value between successive LP relaxations. Three possible, but conflicting, aims are:

1) To obtain the proven optimal solution as quickly as possible.
2) To obtain a desirable integer (feasible) solution and terminate the tree search within a reasonable computational time.
3) To obtain a large number of integer solutions at a reasonable price.

To increase the efficiency of the tree search some useless branches should be fathomed. The higher the solution tree is done, the less the computation effort will be needed. To do that, the cutting off method for the objective function should be

used. Once the solution value of the LP relaxation at a branch becomes worse than the bound, the branching operation will be terminated. This may either cause backtracking or termination of the optimization process.

The intention of this chapter is to determine the optimal cycle time for the line cycle time determination problem by using the branch-and-bound (B&B) procedure, and the decision is made on how to group different types of components into multiple non-identical machines in an assembly line.

4.2 Computational complexity in integer linear programming

To be more precise about the solvability issue of integer linear programming, definitions from the theory of NP-completeness are described [Gar79]. This theory deals with so-called recognition problems, also referred to as feasibility or decision problems. Given an optimization problem, the corresponding recognition problem, requiring a yes/no answer, can be constructed in the following manner. Suppose we have a minimization problem, Min f(x), then the recognition problem is asking for the existence of a solution x such that $f(x) \leq z$ for some threshold value z.

4.3 Linear programming relaxation

LP relaxation neglects the integer constraint when solving an integer programming model. The vertices of the feasible region of an LP relaxation are in general not integer points. If the solution obtained by LP relaxation is not feasible, neither is the original IP model, since both share the same solution space. It gives a bound on the model (lower bound for a minimization model, upper bound for a

maximization model). In addition, a feasible integer solution resulted from the LP relaxation gives a bound on the full IP model (upper bound for minimization model and lower bound for maximization).

The B&B algorithm requires the simplex method to acquire the initial solution as well as the solution at each node of the tree structure. In fact, it is a quite efficient pivot method for solving LP models arising in real-world applications. Before applying the simplex method to an LP, all the constraints on which pivot operations are carried out must be transformed into equality constraints by introducing the appropriate slack or artificial variables. If a primal feasible basic vector is not available, it tries to find one by temporarily ignoring the goal in the objective function. The two-phase method is then constructed. The Phase I problem is to prove whether or not the original LP has a feasible solution. If an initial feasible basic vector for the original problem is available, Phase II then directly solves the problem with the simplex method, which is initialized by the specially constructed LP from Phase I. The following steps are used to illustrate the general procedure of a minimization model:

1. Formulate the mathematical model into a standard form.
2. Multiply the "≥" constraint row(s) with the sign "≥", by -1 and add a slack variable on each of these constraints in order to make it into an equality constraint. When the simplex method is completed, the right-hand side (RHS) constant(s) vector will become nonnegative again.
3. Find the first basic feasible solution. (Iterate through these steps with a slightly expanded form of the problem.)
4. Calculate the reduced costs.
5. Test for optimality.

6. Choose the entering variable.
7. Calculate the Search Direction.
8. Test for unboundedness.
9. Choose the leaving variable by the minimum Ratio Test.
10. Update the solution.
11. Change the basis.
12. Go to Step 5.

4.4 The branch-and-bound (B & B) algorithm

The branch-and-bound algorithm is an approach developed for working out discrete and combinatorial optimization problems. In the meantime, searching for an optimum feasible solution is done by a partial enumeration. The idea of imposing the B&B algorithm for integer programming using linear relaxation was proposed by Land and Doig in the early 60's [Lan60]. The process involves keeping a list of solutions from the linear programming relaxation, which relaxes the constraints for integrity of all the decision variables. Denote the optimal solution by z* and L is the list of the current solution for the decision variable(s) that does not satisfy the integrity requirement. The discrete optimization problems are the problems in which the decision variables are assumed to be discrete values from a specific set (i.e. a set of integers). The combinatorial optimization problems, on the other hand, are the question of choosing the best one out of all possible combinations. The effectiveness of the B&B procedure for solving integer linear programming (ILP) problem using LP relaxation has been well documented in past decades.

4.4.1 A survey on branch-and-bound (B & B) algorithm

Aghezzaf *et al.* addressed a balancing problem in order to achieve a given production rate or to optimize the use of workstations with the B&B algorithm [Agh95]. Daskin *et al.* formulated the PCB component grouping problem into an integer-programming model in order to minimize the total component and PCB loading cost subject to a capacity constraint on the number of types of components and obtained the optimal solution with the B&B algorithm [Das95]. An optimization algorithm based on the B&B method was developed by Asano *et al.* to minimize the maximum tardiness in single machine scheduling problem with ready and due time constraints on jobs [Asa95].

A B&B algorithm was proposed by Sprecher for solving the Type I simple assembly line balancing problem (SALB-I) [Spr96]. The algorithm, based on the precedence tree guided enumeration scheme, was proposed for dealing with a board class of resource-constrained project scheduling problem. Another exact B&B algorithm was formulated to find optimal solutions and to provide a guidance on the source of the gap between a heuristic, the pick and rule (PAR) heuristic, and the lower bound results. This PAR heuristic was presented by Kumar et al. to minimize the total number of processors, while determining the number of processors at each type, the sequence of the processor, and the operations to be performed at a flexible assembly system [Kum00].

The main phases for the B&B algorithm used to solve the discrete and combinatorial problems are: selection branching, bounding and fathoming. The selection phase normally uses a backtracking technique to systematically go through all the possible configurations of a space. These configurations may be all possible

arrangements of objects (permutations) or all possible ways of building a collection of them (subsets). There are two backtracking techniques, the depth-first search and the breath-first search. Sections 4.4.2 and 4.4.3 will give a brief description of these two search techniques.

In the branching phase, the solution of a model is partitioned into two mutually exclusive subsets, and each of these is represented by a node connected in the B&B tree. In the bounding phase, in the case of a minimization model, lower bounds on the optimal solutions of the sub-models are determined. Finally, in the fathoming phase, sub-models are excluded for further consideration because a better solution has been found.

The LP-based branch-and-bound algorithm is stated as follow (Figure 4.1):

1. *Initialize*

   z^* = current best solution with LP relaxation, that is the initial node.

2. *Select*

   Choose one of the decision variables (x_j) from L.

3. *Branch*

   Fix x_j by defining it into two possible sub-regions: $\lfloor x_j \rfloor$ and $\lfloor x_j \rfloor + 1$. $\lfloor x_j \rfloor$ denoted as the greatest integer value approaching to x_j - 1. Take an additional constraint into the model and process it with LP. If the improved solution is found, record it.

4. *Optimality checking / terminate*

   If $L = 0$, then terminate, as the optimal solution has been obtained. If L > 0 and the improved solution is found, go to step 2. Otherwise, go to step 5.

5. *Bounding*

   Apply the compliment of current node and solve it with LP. If the solution is not improved, backtrack one level and repeat solving with LP by selecting with compliment of that node. Otherwise, go to step 4.

The major difficulty in the algorithm is the optimality conditions to check if a given (feasible) solution is optimal or not. Given a candidate solution, how to find an "improving feasible direction" for the next move is a challenge.

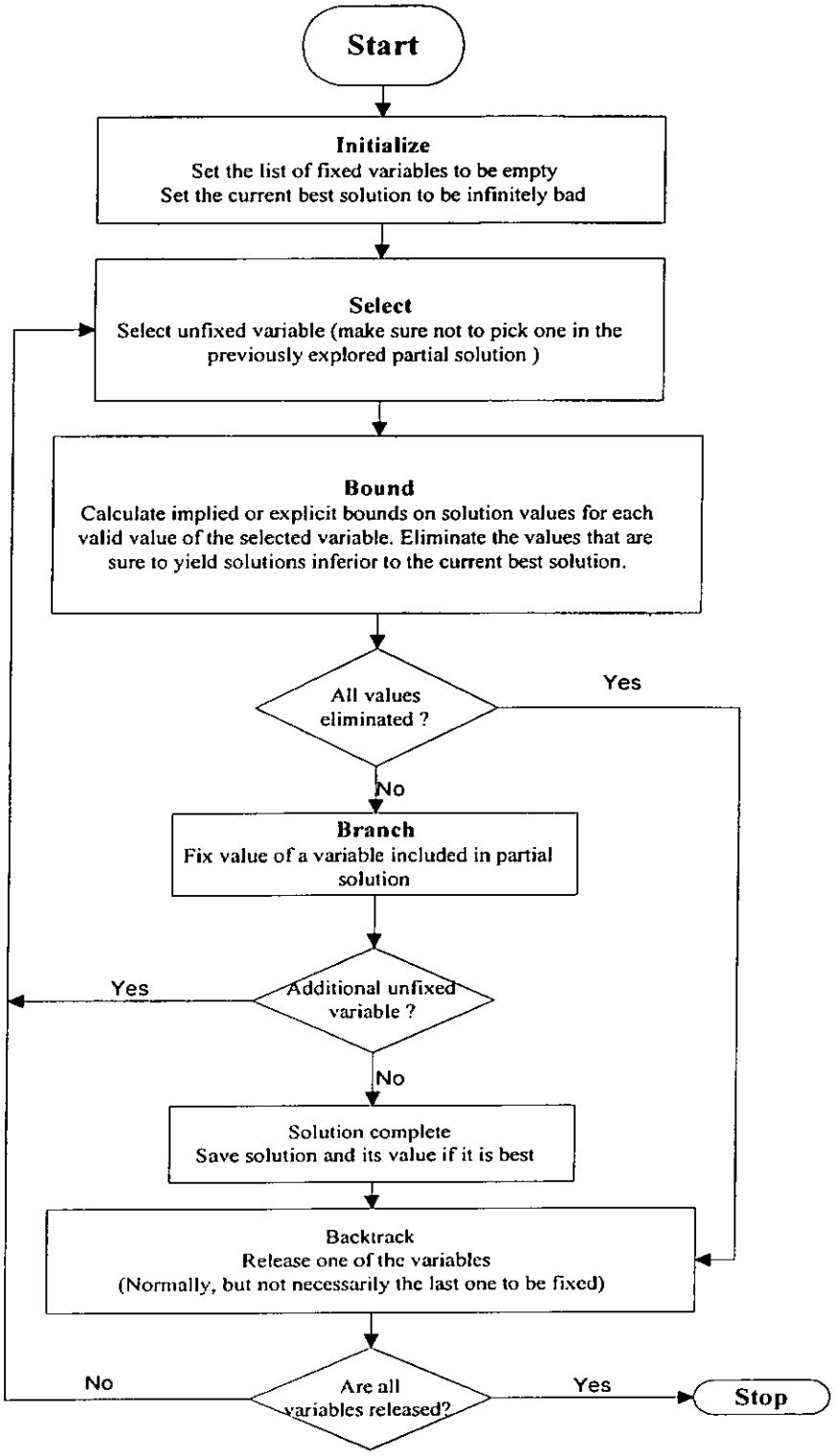


FIGURE 4.1 THE B&B ALGORITHM [TAH75]

4.4.1.1 Depth-first search

Depth-first search (DFS) begins with expanding the initial node and generating its successors. In each subsequent step, the DFS expands one of the most recently generated nodes. If this node has no successors (or cannot lead to any solutions), the DFS backtracks and expands a different node. A major advantage of the DFS is that its storage requirement is linear to the depth of the state space being searched. The graphical representation for the tree structure of the DFS is as follow (Figure 4.2):

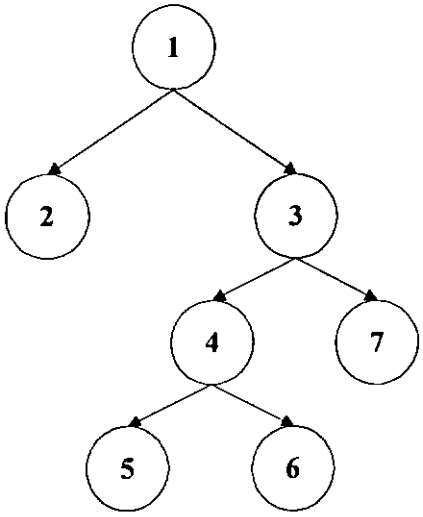


FIGURE 4.2 GRAPHICAL REPRESENTATION FOR DFS

There are two major DFS algorithms: simple backtracking, and depth-first branch-and-bound.

- *Simple Backtracking*

The DFS search method terminates upon finding the first solution (not necessarily optimal). The simple backtracking does not use heuristic information to order the successors of an expanded node. Variant ordered backtracking uses a

heuristic to order the successors of an expanded node.

- *Depth-First Branch-and-Bound*

It exhaustively searches the state space even after finding a solution path. If a solution is found, it will compare with the current value of the solution. If the new one is better than the previous value, the previous value will be replaced by the new value.

Example

A simple example in an ILP model is done to illustrate how DFS performs.

Minimize

$2x1 + 5x2 + 3x3$

Subject to

$3x1 + 2x2 + x3 >= 10$ (4.1)

$x1 + 3x2 + 2x3 >= 12$ (4.2)

$x1 + 2x2 - x3 >= 0$ (4.3)

$x1, x2, x3 \geq 0$ and integer

Step 0 Solve with LP relaxation

In this example, this yields

$x1 = 1.375$, $x2 = 1.125$, $x3 = 3.625$ Objective = 19.25

Step 1 Branch-and-bound algorithm with the DFS

Decision variable x2 is first selected as a branching variable. Since x2 can only take integer values, there is no loss of generality in stipulating that either $x2 \leq 1$ or $x2 \geq 2$. These conditions are appended individually to the original model to create

two new sub-models. For this example with the minimization of the objective function, we start from choosing $x2 \leq 1$ to build the tree with the DFS. The process can be diagrammatically illustrated by using tree structures (Figure 4.3 & 4.4):

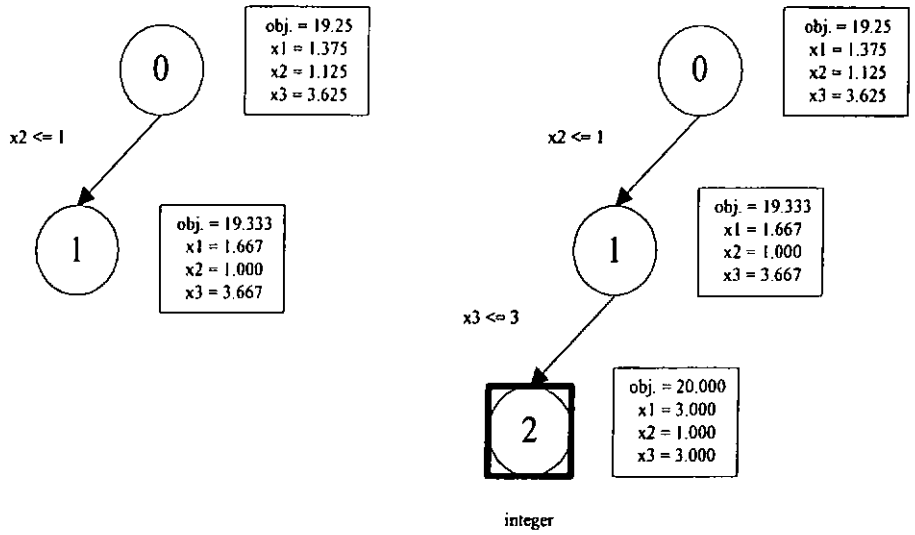


FIGURE 4.3 BRANCHING STRATEGY WITH THE DFS IN THE BRANCH-AND-BOUND ALGORITHM

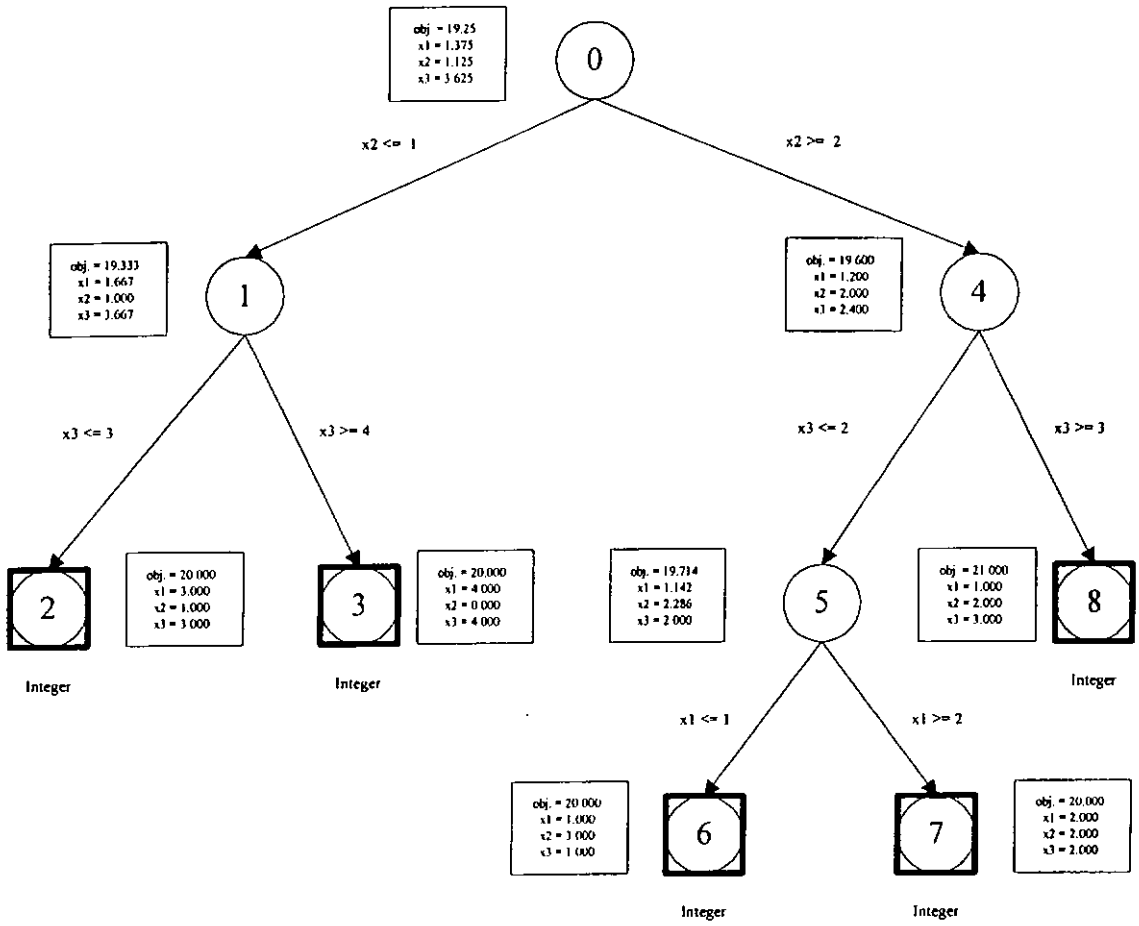


FIGURE 4.4 THE BRANCH-AND-BOUND SOLUTION TREE WITH DFS

The numbers of the node indicate the sequence of the branching operation. It is shown that the complete structure of the tree consists of 8 nodes and an optimal solution is found firstly at node 2.

4.4.1.2 Breath-first search

It is not necessary to terminate branches where the objective value has become worse than that of the best integer solution so far found. The breath-first search (BFS) maintains two lists, open and closed. At the beginning, the initial node

is placed on the open list, then sorted according to a heuristic evaluation function that measures how likely each node is to yield a solution. The graphical representation for the tree structure of the BFS is shown in Figure 4.5.

At each step, the most promising node from the open list is removed. If the node is the goal node, the algorithm terminates, otherwise the node is expanded. The expanded node is placed on the closed list, and the node with the highest heuristic value is deleted.

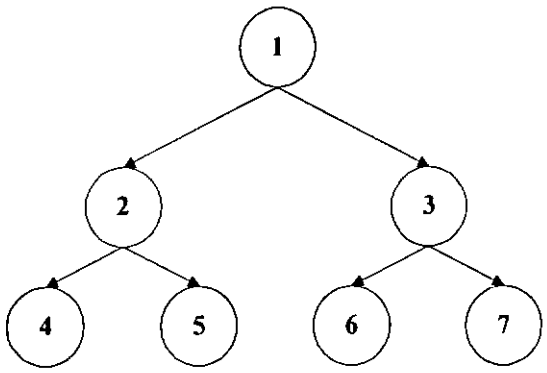


FIGURE 4.5 GRAPHICAL REPRESENTATION OF BFS

The basic idea in the above algorithm is that the nodes are expanded according to the priority (here the priority is the selected heuristic function). The successors of the expanded node are put into the priority queue, while in the DFS, a stack is used. The successors of a newly expanded node will be explored before the old nodes. The main drawback of the BFS is that its memory requirement is linear to the size of the search space explored. With the same example in the previous section, we elaborate it with the BFS. The diagrammatic illustration of the B&B solution tree can be shown as below (Figure 4.6):

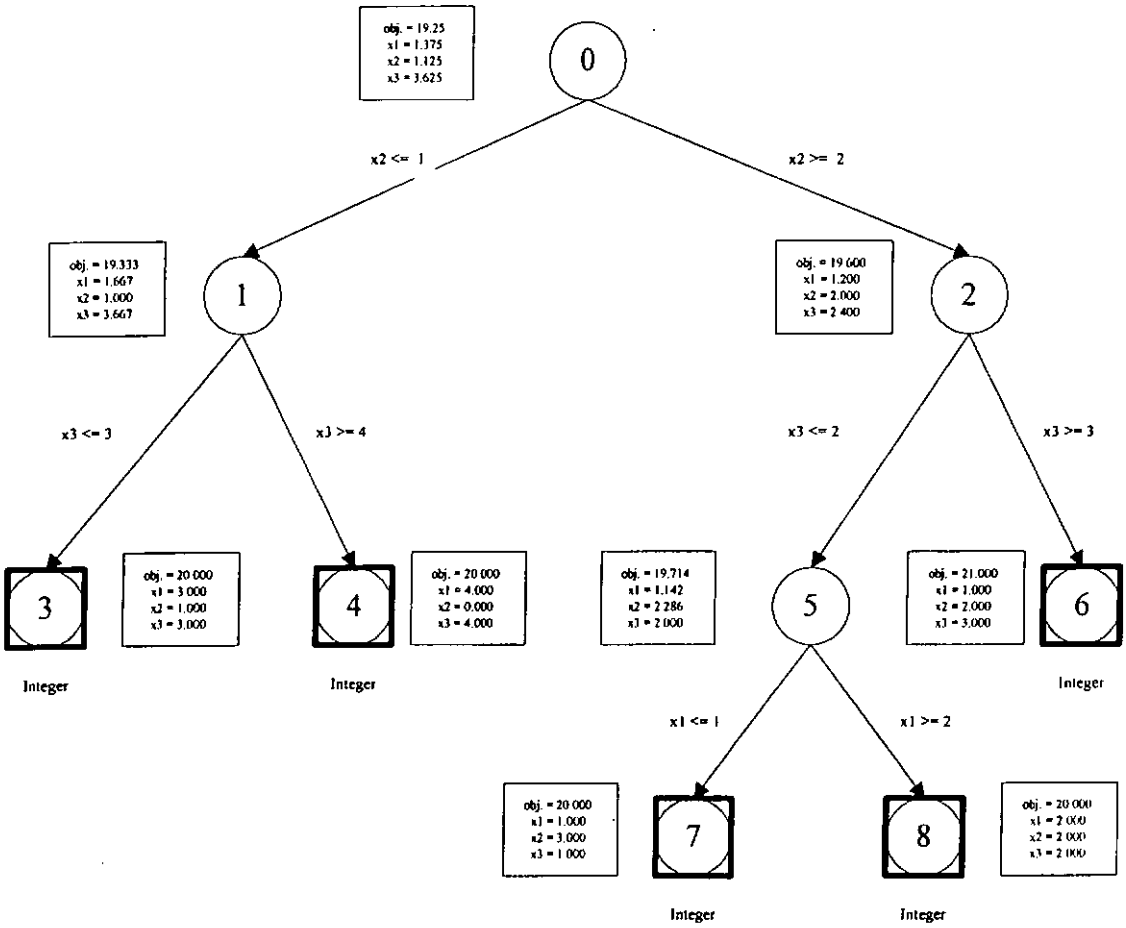


FIGURE 4.6 THE BRANCH-AND-BOUND SOLUTION TREE WITH BFS

The sequence for the branching operation is different compared with the DFS. Although the complete structure of tree still consists of 8 nodes, the optimal solution is found in node 3. It is noticed that it needs one more node or branch before reaching the optimal solution. However, with the increase on the number of decision variables, the solution space will become larger, and may need more computational time to reach the optimal solution.

4.4.2 Performance evaluation

The presentation of the B&B algorithm in this section shows the special data structure of the line cycle determination problem in PCBA. A procedure with the simplex method together with the B&B algorithm is implemented by the computer language C++ (See Appendix IV) and MATLAB (See appendix V). A numerical case study is used to obtain the optimal solution of the algorithm, and compared with the solution from a commercial software package.

In the computational implementation of the B&B algorithm, the depth-first search is used in the branching procedure (Figure 4.7). Both the lower bound and the upper bound are generated at each node in the B&B algorithm. The lower bound of the current node acts as an index to fathom in the usual way when it is greater than the upper bound of the previous node. In fact, in the described branching procedures, the branch is developed on one side only (i.e. left-side branch or right-side branch) until the node is fathomed and backtracking is needed in the algorithm. Moreover, the replacement of the upper bound is taken when the current one is less than the previous or reserves the previous one if the current one is larger. When an additional constraint enters the problem, the problem is repeatedly solved by the simplex method. The following structure is the implementation of the B&B algorithm with the depth first search strategy:

```
DFS_Branch ( Problem T1):
{
  // Fathom the node
  if (T1.L > T1_Previous_U)
  return;

  // Branch for the selected variables with non-integer value

  for  (int i= 1; i <= m; i++)
     for (int j = 1; j <= n; j++)
    {
     if T1.basis [i][j]  != round (T1.basis[i][j])
       continue;

     //Generate an additional constraint to the problem
     newT1 = T1;
     perform the simplex method over new T1;
     if (newT1 is not feasible)
     {
              backtrack to the upper node;
       Set the constraint with another branch;
       modify record and save the new bound;
       return;
     }

     compute the newT1.L;
     DFS_Branch (Problem newT1);
     until all variables are integers or newT1.L = newT1.U
    }
}
```

FIGURE 4.7 COMPUTATIONAL IMPLEMENTATION OF THE BRANCH-AND-BOUND ALGORITHM

4.5 Numerical case study

The B&B algorithm is applied to the case with a single-sided PCBA described in Chapter Three. With the objective of minimizing the line cycle time, the optimal solution acquired by the B&B algorithm is found. Recall the mathematical model for the case in the line cycle time determination problem as stated below:

Minimize

T

Subject to

1) $T - 3x_{11} - 7x_{12} - 7x_{13} - 90000x_{14} - 90000x_{15} - 90000x_{16} >= 110$

2) $T - 7x_{21} - 12x_{22} - 17x_{23} - 24x_{24} - 17x_{25} - 24x_{26} >= 147$

3) $T - 23x_{31} - 38x_{32} - 35x_{33} - 38x_{34} - 38x_{35} - 36X_{36} >= 147$

4) $X_{11} + X_{21} + X_{31} = 321$

5) $X_{12} + X_{22} + X_{32} = 67$

6) $X_{13} + X_{23} + X_{33} = 35$

7) $X_{14} + X_{24} + X_{34} = 12$

8) $X_{15} + X_{25} + X_{35} = 31$

9) $X_{16} + X_{26} + X_{36} = 12$

10) T, $x_{ij} \geq 0$ and integers. for $i = 1, 2, 3$; $j = 1, 2, \ldots, 5, 6$

4.5.1 Solution procedures

- *Step 0 (Initialization)*

$$A=\begin{bmatrix}
3 & 7 & 7 & 90000 & 90000 & 90000 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & -1 \\
0 & 0 & 0 & 0 & 0 & 0 & 7 & 12 & 17 & 24 & 17 & 24 & 0 & 0 & 0 & 0 & 0 & -1 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 23 & 38 & 35 & 38 & 38 & 36 & -1 \\
1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 0
\end{bmatrix}$$

b=[-110 -147 -147 321 67 35 12 31 12]^{T};

c=[0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1];

- *Step 1 (The simplex method)*

In the simplex method, the two-phase method is performed.

Phase I

The basis obtained in Phase I is:

```
bas =
   5   7   13   19   2   3   10   11   12
```

The current basic variable values are:

```
b =
  1.0e+003 *
   0.0000
   0.2093
   0.1117
   2.7151
   0.0670
   0.0350
   0.0120
   0.0310
   0.0120
```

The pivot step is done using the 'Gauss-Jordan' elimination. No special factorizations are used to ensure stability. A final check on the rounding error is made.

Phase II

The solution obtained in Phase II is:

```
bas =

    1    8   17   19    2    3   16   11   18
```

The current basic variable values are:

```
b =

    1.0e+003 *
     0.3210
     0.0655
     0.0077
     1.3286
     0.0015
     0.0350
     0.0120
     0.0233
     0.0120
```

The current objective value is:

```
T =

  1.3286e+003
```

The number of iterations is 6

Final tableau

```
1.0e+004 *
```

Columns 1 through 7

0.0001	0	0	0	0	0	0.0001
0	0	0	-0.5887	-0.5887	-0.5887	0.0001
0	0	0	-0.1285	-0.1285	-0.1285	0
0	0.0001	0	-4.8794	-4.8794	-4.8794	-0.0001
0	0	0.0001	0.5887	0.5887	0.5887	0
0	0	0	0	0	0	0
0	0	0	0.0001	0	0	0
0	0	0	0.1285	0.1285	0.1285	0
0	0	0	0	0	0.0001	0
0	0	0	4.8794	4.8794	4.8794	0.0001

Columns 8 through 14

0	0	0	0	0	0.0001	0
0.0001	0.0001	0	0	0	0.0001	0.0001
0	0	-0.0001	0	0.0001	-0.0001	0.0001
0	-0.0002	0.0002	0	-0.0002	-0.0002	-0.0002
0	-0.0001	0	0	0	-0.0001	0
0	0.0001	0	0	0	0	0
0	0	0.0001	0	0	0	0
0	0	-0.0001	0.0001	0.0001	0.0001	0.0001
0	0	0	0.0001	0	0	0
0	0.0002	0.0002	0	0.0002	0.0002	0.0002

Columns 15 through 21

0	0	0	0	0	0.0001	0
0.0001	0.0001	0	0	0	0.0001	0.0001
0	0	-0.0001	0	0.0001	-0.0001	0.0001
0	-0.0002	0.0002	0	-0.0002	-0.0002	-0.0002
0	-0.0001	0	0	0	-0.0001	0
0	0.0001	0	0	0	0	0
0	0	0.0001	0	0	0	0
0	0	-0.0001	0.0001	0.0001	0.0001	0.0001
0	0	0	0.0001	0	0	0
0	0.0002	0.0002	0	0.0002	0.0002	0.0002

Columns 22 through 23

0	0.0321
0	0.0065

0	0.0008
0	0.1329
0	0.0002
0	0.0035
0	0.0012
0	0.0023
0	0.0012
0	-0.1329

Step 3 (Branch-and-Bound)

Substitute the resulted solution from the simplex method to the initialization of the B&B algorithm. The optimal solution obtained from the algorithm is stated as below:

$$[x_{11}, x_{12}, x_{13}, x_{14}, x_{15}, x_{16}, x_{21}, x_{22}, x_{23}, x_{24}, x_{25}, x_{26}, x_{31}, x_{32}, x_{33}, x_{34}, x_{35}, x_{36}]$$

$$= [319, 3, 35, 0, 0, 0, 1, 64, 0, 0, 24, 0, 1, 0, 0, 12, 7, 12]$$

For machine i, the machine processing time MT_i (in 0.1 sec) is:

i = 1, $MT_1 = 1333$

i = 2 $MT_2 = 1330$

i = 3 $MT_3 = 1324$

Therefore, the line cycle time is equal to:

$\max \{MT_1, MT_2, MT_3\} = \max \{ 1333, 1330, 1324\} = 1333$

The objective function value from the computational program is T = 1333.

4.5.2 Result analysis

The line cycle time determination problem has been formulated as an integer linear programming and solved by the branch-and-bound algorithm. The optimal solutions obtained are all integers. Previously, the task of grouping different

component types can be solved with a linear programming, by rounding the decision variable that have fraction values, into integers. If the rounding method is applied, the solution becomes:

$$[x_{11}, x_{12}, x_{13}, x_{14}, x_{15}, x_{16}, x_{21}, x_{22}, x_{23}, x_{24}, x_{25}, x_{26}, x_{31}, x_{32}, x_{33}, x_{34}, x_{35}, x_{36}]$$

$$= [321, 2, 35, 0, 0, 0, 0, 65, 0, 0, 23, 0, 0, 0, 0, 12, 8, 12]$$

and the objective function value T = 1339.

However, in the integer linear programming, the optimal solution obtained is:

$$[x_{11}, x_{12}, x_{13}, x_{14}, x_{15}, x_{16}, x_{21}, x_{22}, x_{23}, x_{24}, x_{25}, x_{26}, x_{31}, x_{32}, x_{33}, x_{34}, x_{35}, x_{36}]$$

$$= [319, 3, 35, 0, 0, 0, 1, 64, 0, 0, 24, 0, 1, 0, 0, 12, 7, 12]$$

and the objective function value T = 1333

There is a difference of 0.6 second and the relative error is about 0.45%. This percentage is high since the production volume is very large.

4.6 Conclusion

In this chapter, a review on the branch-and-bound algorithm was carried out since it is a widely used optimization technique in integer linear programming. In addition, the line cycle time determination problem was formulated as an ILP. Instead of using the rounding method to obtain an all-integer solution to the problem, the branch-and-bound algorithm was implemented to optimize the line cycle time. This algorithm is based on the tree search strategies. There are two major tree search strategies, the depth-first search and the breath-first search. Obviously, the results obtained are based on depth-first search, since this storage requirement is linear to the depth of the state space being searched.

CHAPTER FIVE

A TABU SEARCH HEURISTIC

5.1 Introduction

Tabu search (TS) is a search heuristic that can be used to guide any search process. It employs a set of moves for transforming one solution to another and provides an evaluation function for measuring the attractiveness of the move. It is designed to cross boundaries of feasibility or local optimality normally treated as a barrier, to impose systematically and release the constraints to permit exploration of the forbidden regions. Restrictions are imposed to guide the search process in negotiating difficult regions. These restrictions operate in several forms, both by direct exclusion of certain search alternatives classified as tabu (forbidden), and also by translation into modified evaluations and probability in selection.

The main problem of a basic local search is that the search space exploration often gets trapped in solution states, called local minima, in which no improving operator applies. To cope with this problem of being trapping in a local minimum, a control tactic in tabu search is implemented. In the computational effort, the issue relative to the speeding up of the execution on each iteration of the search is addressed.

It uses a flexible structure memory to record the varying time span in the intensification and diversification in order to accelerate the searching process.

In section 5.2 of this chapter, some previous applications of tabu search

heuristics are described. Section 5.3 gives a brief description on the tabu search heuristic technique. Sections 5.4 and 5.5, respectively, present the technique used for the line cycle time determination problem and a numerical case study. Finally, in section 5.6, a comparison is made on this heuristic search technique and the branch-and-bound algorithm used for the problem.

5.2 A survey on tabu search heuristic

Tabu search is an iterative procedure for solving discrete combinatorial optimization problems. It was first suggested by Glover in 1977 and since then it has been increasingly used [Glo77]. It has been successfully applied to obtain optimal or sub-optimal solutions to such problems as scheduling, timetabling, travelling salesman and layout optimization. Varieties of applications were made from scheduling to telecommunications. Early applications of tabu search focused on the flow-shop scheduling problem. Daniel *et al.* presented the TS method for the flexible resource flow shop scheduling problem, which employed a nested search strategy based on the decomposition of the problem into three main components such as resources-allocation, job sequencing and operation start time [Dan93]. Dell'Amico and Trubian applied TS to the job shop scheduling problem with a bi-directional method to find feasible starting solutions [Del93]. A partial schedule was initially obtained by this method. Several points were taken into consideration:

- How to choose the initial heuristic to produce the starting point.
- How to modify the starting point into several starting points, to allow multiple search attempts.
- How to find a small enough neighborhood to make computation tractable with

an excellent solution.

- How to filter out most choices in the neighborhood by an approximate evaluation of the interchange.
- How to calculate the remaining interchanges efficiently.
- How to deal with various technical issues such as tabu list size and aspiration criteria.

Woodruff *et al.* presented a TS procedure for production scheduling, addressed a general sequencing problem with objectives of minimizing deadlines and setup times [Woo92]. They used an insertion move to transform one trial solution to another. A candidate list was used as a mean of reducing the computational effort involved in evaluating the neighborhood. The tabu list was based on the concept of hashing function in controlling the function of searching, and supported the contention of long-term memory characteristics. Tabu search was also applied to the quadratic assignment problem [Kap94], a machine-scheduling problem and the clustered travelling salesman problem [Lap96]. Also, Vakharia *et al.* explored the TS method to group technology in the cell formulation [Vak97].

Tsubakitani *et al.* studied the problem of optimizing the size of the tabu list when applying tabu search with a short-term memory function to the symmetric travelling salesman problem [Tsu98]. The study also identified the best tabu list size within a given computational time limit. Moreover, it revealed that a good tabu list size is smaller that generally believed and that smaller neighborhoods require large tabu list sizes in order to be effective.

Baar *et al* illustrated the resource-constrained project-scheduling problem with a tabu search algorithm in order to determine a schedule with a minimal makespan. A schedule scheme consisting of sets of relations, which defined a set of

possible schedule, was introduced on the basis of parallelity [Baa99].

Since the robotics board and magazine simultaneously move at different speeds during a robotic assembly, the routing of robotic travel is based on relative coordinates. Consequently, the coordinates of placement points and the magazine are constantly changing. In one study, a novel tabu search (TS) based approach was presented [Suc98]. The proposed approach arranged the placement sequence and assigned the magazine slots to yield a performance better than the conventional one.

5.3 The Tabu Search technique

The philosophy of tabu search is to derive and exploit a collection of principles of intelligent problem solving. A fundamental element underlying tabu search is the use of flexible memory that embodies the dual processes of creating and exploiting memory structures.

Tabu search has two high-level control strategies, intensification and diversification [Glo89], [Glo90a]. Intensification aims at focusing the search on promising areas of the search space, while diversification directs the search to yet unexplored, but promising, regions.

Initially, tabu search was only used with a short-term memory, which emphasized for escaping local minima [Pir96]. One of the main features of short-term memory is to store attributes of the solution already visited in the recent past. A restriction rule is associated with these attributes in order to forbid the occurrence with their complements.

The memory structures of tabu search are operated by reference to four principle dimensions, consisting of recency, frequency, quality, and influence. These

dimensions are set against a background of logical structure and connectivity. Tabu search is founded in three primary themes:

1. The use of flexible attribute-based memory structures designed to permit evaluation criteria and historical search information to be exploited more thoroughly than by rigid memory structures.
2. Associated mechanism of control – for employing the memory structure – based on the interplay between conditions that constrain and free the search process (embodied in tabu restrictions and aspiration criteria).
3. The incorporation of memory functions of different time spans (from short term to long term) to implement strategies for intensifying and diversifying the search.

5.3.1 Short-term memory

The core of tabu search is embedded in the short-term memory process. The short-term memory of tabu search constitutes a form of aggressive exploration that seeks to make the best possible move, subject to requiring available choices to satisfy certain constraints. These constraints embody the tabu restrictions by rendering selected attributes of these moves forbidden. In general the tabu restrictions are used to prevent the search from repeating swap combinations tried in the recent past, potentially reversing the effects of previous moves by interchanges that might return to a previous position. The tabu will classify all the swaps in the most recent pair of modules. A structure for the short-term memory in TS can be generalized as follows (Figure 5.1):

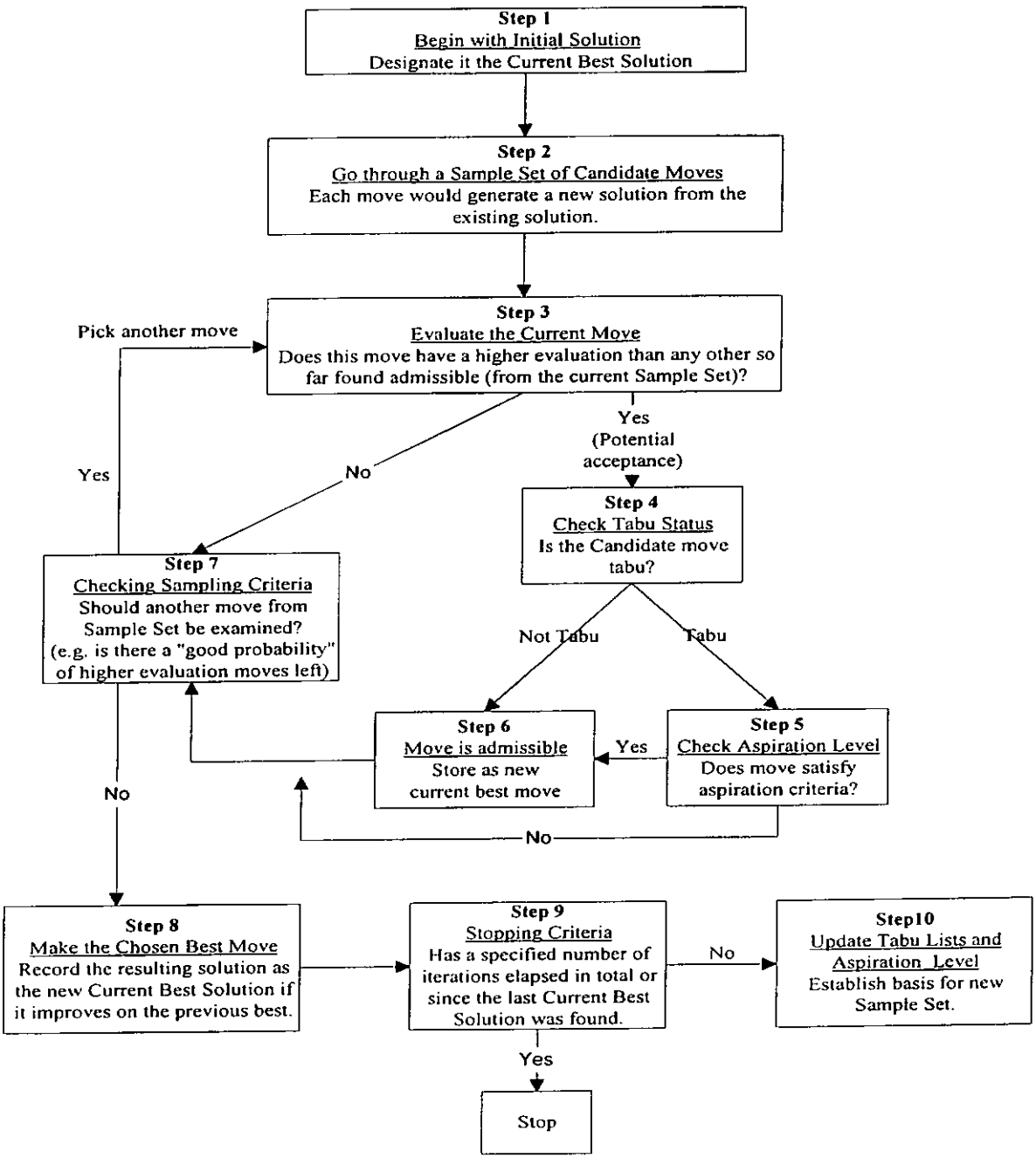


FIGURE 5.1 THE TABU SEARCH SHORT-TERM MEMORY COMPONENT [GLO90b]

5.3.2 Operation parameters

5.3.2.1 Candidate list strategy

Candidate list strategy intends to isolate a candidate subset of moves from a large neighborhood, to reduce the computational expense of evaluating all the moves in the entire neighborhood. A simple form of candidate list strategy is to construct a single element list by sampling from the neighborhood space at random, and to repeat the process if the outcome is deemed unacceptable. Another kind of candidate list strategy periodically examines larger portions of the neighborhood, creating a master list of several best alternatives found. The master list then identifies moves for additional iterations until a threshold of acceptability triggers the creation of a new master list. Moves can be selected by choosing the best candidate from several processes, or each process can execute its own preferred move [Glo93b].

5.3.2.2 Tabu list

Tabu list records a complete description of last visited solutions, in which the number of records kept is previously defined. This serves as the tabu restriction to inhibit the cycling or revisit to the branch and implicitly keep track of moves by recording attributes complementary to the running list. The size for the list is generally within the range $\sqrt{(n)}$ and $2*\sqrt{(n)}$, where n is the number of decision variables. The tabu list management concerns updating the tabu list (i.e. deciding on how many and which moves have to be set to be tabu in an iteration of the search). Tabu list management carries on two different memory structures: recency based and

frequency based. Recency based structure maintains the records individually for different attributes or different kinds of attributes. Frequency based structure is one of the features in long term memory.

5.3.2.3 Tabu tenure

In general, recency-based memory is managed by creating one or several tabu lists, which record the tabu-active attributes that both implicitly and explicitly identify the current status. Tabu tenure can vary for different types or combinations of attributes, and can also vary over different intervals of time and stages of the search. Effective tabu tenure is empirically dependent on the size of the problem. However, no single rule has been designed to yield an effective tenure for all problem classes. Tenures that are too small can be recognized by periodically repeating objective functions or occurrence of cycle. Tenures that are too large can be recognized by resulting deterioration in the quality of the solutions found. There are two major types of tabu tenure: random dynamic tenure and systematic dynamic tenure. Random tabu tenure is to randomly select a range, usually following a uniform distribution. Systematic tenure consists of creating a sequence of tabu search tenure in a range.

Once a good range of tenure values is located, first level improvement generally results by selecting different values from this range on different iterations.

5.3.2.4 Aspiration criteria

An aspiration criteria is used to determine when tabu activation rules can be

overridden to remove a tabu classification, or applied to a move with improved-best and aspiration-by-default criteria. The appropriate use of such criteria can be very important for enabling a TS heuristic to achieve its best performance level in the search strategy.

5.3.2.5 Strategic oscillation

The strategic oscillation in tabu search illustrates an intimate relationship between changes in neighborhood and changes in evaluation. It provides an effective interplay between intensification and diversification over the intermediate to long term. A standard neighborhood that only allows moves among feasible solutions enlarges by this approach to encompass non-feasible solutions. The search is then strategically driven to cross the feasibility boundary into the non-feasible region. The emphasis on guidance differentiates a meta-heuristic from a simple random restart procedure or a random perturbation procedure.

The use of strategic oscillation in some applications maintains this construction at a given level. The applications, including alternating constructive and destructive processes, can be accompanied by exchanging moves. A proximate optimality principle motivates the exchange on either side of the candidate solution at different levels of the spanning tree before proceeding to the adjacent level. The principle roughly states that good constructions at one level are likely to be close to good constructions at another.

5.3.3 Long term memory

For long-term memory, frequency based memory is often used, which is decomposed into subclasses by taking account of the dimension of solution quality and move influence. Its improvement begins to be manifest in a relatively modest length of time. Attributes that have greater frequency measures can trigger a tabu activation rule if they are based on consecutive solutions that end with the current solution.

5.3.4 Significance of the Tabu Search heuristic

Although the global solution of an objective function is not guaranteed to be found, TS always performs a better search than the existing tree search strategy. Since TS processes the searching operation with random neighbor solution, the existence of a tabu list provides an effective way on iterative deepening and avoiding cycling behavior. In addition, this heuristic gives a stable solution and better solution performance for a small problem size. For a large problem, it can also have a better performance if a sufficient computational time is allowed.

5.4 Algorithm for the Line Cycle Time Determination Problem

The general structure of the TS heuristic for the line cycle time determination problem is shown in Figure 5.2. The algorithm starts with an initial solution, which is generated by solving the problem with the simplex method, and then it acts as the lower bound of the objective function in the model LP3-3. The fundamental branching moves, which are already assigned to different values or bounds to integer

variables, generate alternatives that arise in the tree search strategies in the branch-and-bound algorithm, and can be readily embedded in tabu search. When the TS is used to guide branching strategies, various branching moves that are not considered in the usual branch-and-bound algorithm, become natural to include. An approach for guiding branching decisions in a TS procedure is considered in the line cycle time determination problem in PCBA.

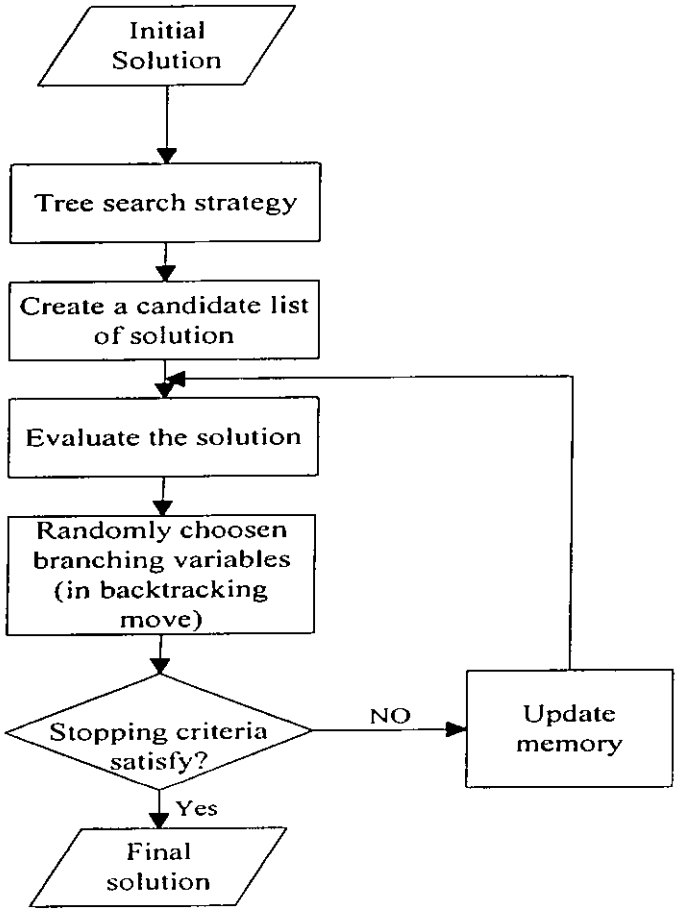


FIGURE 5.2 GENERAL STRUCTURE OF TABU SEARCH OF THE LINE CYCLE TIME DETERMINATION PROBLEM

5.4.1 Initialization

The following initialization procedure is used to generate a feasible solution

for the line cycle time determination problem in PCB assembly:

Step 1: Solve the mathematical model with the LP relaxation where decision variables are allowed to be fractional.

Step 2: Select one of the decision variables, x_{ij}, with a fractional value.

Step 3: Fix x_{ij} by defining it into two possible sub-region: $\lfloor x_{ij} \rfloor$ and $\lfloor x_{ij} \rfloor + 1$.

where $\lfloor x_{ij} \rfloor$ is denoted as the greatest integer value approaching to x_{ij} - 1. Take the additional constraint into the model and process it with LP relaxation.

At this stage, i.e., at each node of the search tree, an analysis is performed to identify which decision variable x_{ij} and which branching alternative for the variable will be selected for immediate exploration. The imposition of the branching inequality that will generate a more constrained linear program, is solved to continue the process, while the alternative branch is saved, to be explored. Tabu search then randomly selects a decision variable to resume the search from the early stages of the search tree.

5.4.2 The mechanism of tabu search with branching strategy

This mechanism is initially based on the branch-and-bound algorithm. Three types of moves are relevant to a tabu search based approach:

Restriction : impose a branch of the form $x_{ij} \leq \lfloor x_{ij} \rfloor$ or $x_{ij} \geq \lfloor x_{ij} \rfloor + 1$.

Relaxation : relax (undo) a branch previously imposed.

Reversal : impose a branch that complements a branch previously imposed.

This set of moves describes the options that differ from those customarily

available to the classic branch-and-bound algorithm. Tabu search, by contrast, can reverse a branch search process at a "full resolution level" of the tree (i.e. at a level where the branches yield integer values for all integer variables).

5.4.3 The mechanism of diversification

The mechanism seeks to drive the search into unexplored regions. The imposed randomization is a means for achieving diversity without reliance on memory. The use of randomization, via assigned probabilities, allows gains in efficiency by obviating extensive record kept so that a more systematic pursuit of diversity is necessary. However, this mechanism entails a loss of efficiency by allowing duplications and potentially unproductive wandering.

5.4.4 Algorithm

Given a feasible solution x^* with the objective function value T^*, let $x := x^*$ with $z(x) = T^*$. The iteration proceeds if the stopping criterion is not fulfilled:

- Randomly select the best admissible move that transforms x into x' with the objective function value z(x') and add its attributes to the running list
- Perform tabu list management, that is, compute moves to be set tabu and update the tabu list.
- Perform exchanges: $x := x'$ and $z(x) = z(x')$; if $z(x) < T^*$.

The elements that underlie the assignment of probabilities to the tabu search heuristic, by contrast, maintain the distinction function between the relative

attractiveness of alternative moves at all stages without giving the non-improving moves higher status at the beginning. In addition, progressiveness diminishing their chances for considerations at the later stages can be eliminated. Furthermore, the tabu search heuristic does not specify that the process should start with a solution far from the optimality or the best solution is to be identified as the local optimal, which can be reached at the conclusion of an eventual undeviating descent.

5.4.5 Function evaluation

Combinatorial optimization problems are not always conveniently structured to assure a feasible path exists between all feasible solutions, and allowing non-feasible solutions to be evaluated and visited. The presentation of the tabu search heuristic in this section shows the data structure to optimize the line cycle time determination problem. The procedure for the tabu search is implemented by the computer language MATLAB (See appendix VI). A numerical case is used to obtain the proof of the performance with the classic branch-and-bound method.

5.5 Numerical Case Study

The TS heuristic is applied to the case with the single-sided PCBA in the sense of the line cycle time determination problem described in Chapter Three. With the objective of minimizing the line cycle time, near optimal or optimal solutions acquired by the TS heuristic can be found. Recall the mathematical model for the case in the line cycle time determination problem:

Minimize

T

Subject to

1) $T - 3x_{11} - 7x_{12} - 7x_{13} - 90000x_{14} - 90000x_{15} - 90000x_{16} >= 110$

2) $T - 7x_{21} - 12x_{22} - 17x_{23} - 24x_{24} - 17x_{25} - 24x_{26} >= 147$

3) $T - 23x_{31} - 38x_{32} - 35x_{33} - 38x_{34} - 38x_{35} - 36X_{36} >= 147$

4) $X_{11} + X_{21} + X_{31} = 321$

5) $X_{12} + X_{22} + X_{32} = 67$

6) $X_{13} + X_{23} + X_{33} = 35$

7) $X_{14} + X_{24} + X_{34} = 12$

8) $X_{15} + X_{25} + X_{35} = 31$

9) $X_{16} + X_{26} + X_{36} = 12$

10) T, $x_{ij} \geq 0$ and all are integers. for i = 1, 2, 3 , j = 1, 2, .., 6

5.5.1 Initialization

The initialization of the TS for the line cycle time determination problem is the same as the simplex method described in the previous chapter. The initial solution is based on the linear programming relaxation and not all decision variables are integers. For the model described as above (LP 3-3):

- *Step 1* Initialization from LP relaxation – the simplex method

  The initial solution resulted from simplex method

	x_{11}	x_{12}	x_{13}	x_{14}	x_{15}	x_{16}	x_{21}	x_{22}	x_{23}	x_{24}	x_{25}	X_{26}	x_{31}	x_{32}	X_{33}	x_{34}	x_{35}	x_{36}	T
Value	321	1.51	35	0	0	0	0	65.49	0	0	23.27	0	0	0	0	12	7.73	12	1328.56

- *Step 2* Tabu search in branching strategy

The probability assigned to different backtracking levels provided a jumping step that can speed the searching process. The example using one set of numerical setting range with the assignment of the probabilities is stated as below:

```
if random_n<=0.875
   stacksize_step=1;
end
if (random_n>0.875 & random_n<=0.925)
   stacksize_step=2;
end
if (random_n>0.925 & random_n<=0.975)
   stacksize_step=3;
end
if random_n>0.975
   stacksize_step=4;
end
```

5.5.2 Result analysis

The line cycle time determination problem has been formulated as an integer linear program and solved by the TS heuristic with the branching strategy. The final solutions obtained are all integers. Provided the backtracking procedure or the move is randomly selected with respect to different levels, the best solution obtained may have variations among the lower and the upper bounds. With the above model, there are several numerical setting ranges presented within the TS heuristic. Moreover, the result is found to be closer to the optimal solution rather than the upper bound of the solution. The distribution of the feasible solution resulted from the computation is shown in Table 5.1 and Figure 5.3:

	Feasible objective value T			
	1333	1335	1336	1339
Solution Frequency	463	332	201	4
Ratio in 1000 running cycles	0.463	0.332	0.201	0.004

TABLE 5.1 DISTRIBUTION OF THE FEASIBLE SOLUTION IN 1000 RUNNING CYCLES

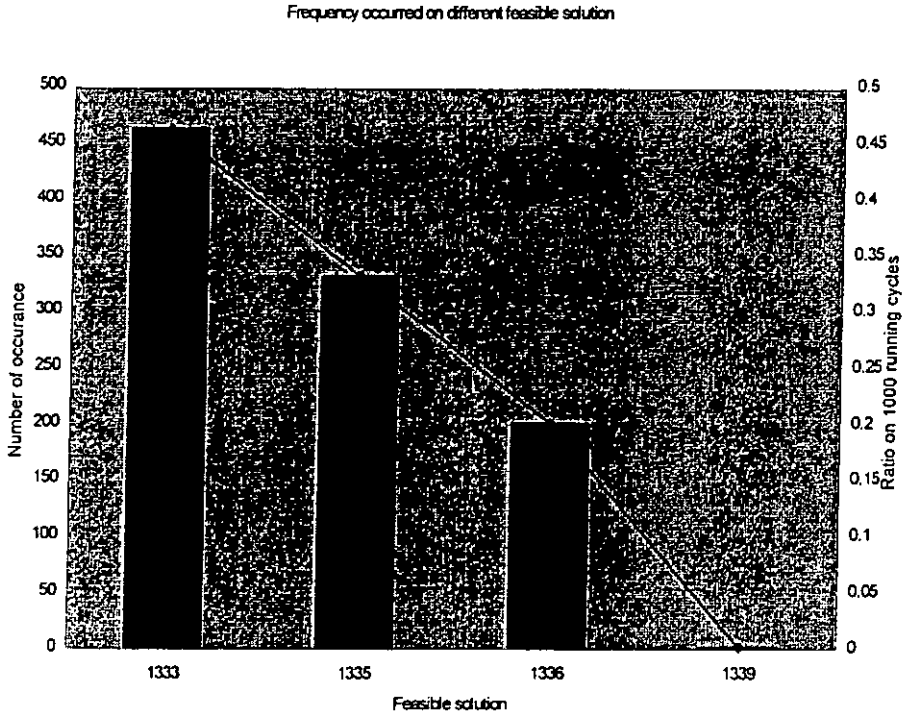


FIGURE 5.3 GRAPHICAL REPRESENTATION FOR THE DISTRIBUTION OF THE FEASIBLE SOLUTION OCCURRENCE

From the resulting solution, there are a total of 4 variations of the feasible objective value including the optimal solution and the upper bound of the numerical case. The optimum solution of this problem is 133.3 seconds, whereas the upper bound is 133.9 seconds and feasible solutions lie within this range.

Upon the feasible objective value as well as the optimal one obtained, general combinations of component type – machine assignment with different feasible objective values are shown as below (Table 5.2):

Decision Variable	Value of decision variables with objective value T			
	1333	1335	1336	1339
x_{11}	319	319	320	321
x_{12}	3	3	4	3
x_{13}	35	35	34	35
x_{14}	0	0	0	0
x_{15}	0	0	0	0
x_{16}	0	0	0	0
x_{21}	1	1	0	0
x_{22}	64	63	63	63
x_{23}	0	0	0	0
x_{24}	0	0	1	0
x_{25}	24	25	24	25
x_{26}	0	0	0	0
x_{31}	1	1	1	0
x_{32}	0	1	0	1
x_{33}	0	0	1	0
x_{34}	12	12	11	12
x_{35}	7	6	7	6
x_{36}	12	12	12	12

TABLE 5.2 COMBINATION OF COMPONENT TYPE-MACHINE RELATIONSHIP TO DIFFERENT OBJECTIVE VALUES

5.6 Identification of changing the range of numerical setting in TS

The backtracking operation performed in tabu branching is based on the randomly generated number in order to determine which level of the nodes is backtracked. This project investigates the effect of this range of numerical settings. Ten sets of numerical setting range are performed as follows (Table 5.3):

Settings(starting probability / range of settings)	Numerical setting range (n) for backtracking level			
	Level 1	Level 2	Level 3	Level 4
1 (0.10/0.30)	$n \leq 0.100$	$n > 0.100$ or $n \leq 0.400$	$n > 0.400$ or $n \leq 0.700$	$n > 0.700$
2 (0.20/0.267)	$n \leq 0.200$	$n > 0.200$ or $n \leq 0.467$	$n > 0.467$ or $n \leq 0.733$	$n > 0.733$
3 (0.30/0.233)	$n \leq 0.300$	$n > 0.300$ or $n \leq 0.533$	$n > 0.533$ or $n \leq 0.767$	$n > 0.767$
4 (0.40/0.20)	$n \leq 0.400$	$n > 0.400$ or $n \leq 0.600$	$n > 0.600$ or $n \leq 0.800$	$n > 0.800$
5 (0.50/0.167)	$n \leq 0.500$	$n > 0.500$ or $n \leq 0.667$	$n > 0.667$ or $n \leq 0.833$	$n > 0.833$
6 (0.60/0.133)	$n \leq 0.600$	$n > 0.600$ or $n \leq 0.733$	$n > 0.733$ or $n \leq 0.867$	$n > 0.867$
7 (0.70/0.10)	$n \leq 0.700$	$n > 0.700$ or $n \leq 0.800$	$n > 0.800$ or $n \leq 0.900$	$n > 0.900$
8 (0.80/0.067)	$n \leq 0.800$	$n > 0.800$ or $n \leq 0.867$	$n > 0.867$ or $n \leq 0.933$	$n > 0.933$
9 (0.85/0.05)	$n \leq 0.850$	$n > 0.850$ or $n \leq 0.900$	$n > 0.900$ or $n \leq 0.950$	$n > 0.950$
10 (0.90/0.033)	$n \leq 0.900$	$n > 0.900$ or $n \leq 0.933$	$n > 0.933$ or $n \leq 0.967$	$n > 0.967$

TABLE 5.3 NUMERICAL SETTING RANGE FOR THE BACKTRACKING LEVEL.

At each numerical setting range in the TS heuristic, 1000 iterations were run.

In the TS heuristic, one of the major operations is the backtracking operation. In general, the backtracking operation is normally retreated to one level, based on one of two available bounds.

If the backtracking operation proceeds, a number is randomly generated to determine which level to be backtracked by falling into the numerical setting ranges. The performance evaluation of these different numerical setting ranges is made on the proportion of various feasible solutions obtained within 1000 program runs and the number of iterations in each run before achieving those feasible solutions.

After 1000 running cycles of each numerical setting range, the percentage of various feasible solution illustrated in Table 5.4 and the graphical representation of the percentage distribution of various feasible solutions is shown in Figure 5.4:

T	Settings for numerical setting ranges									
	1	2	3	4	5	6	7	8	9	10
1333	0%	0%	1%	3%	5%	8%	15%	28%	40%	54%
1334	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%
1335	1%	3%	6%	11%	14%	20%	26%	32%	31%	31%
1336	31%	34%	37%	33%	44%	49%	46%	36%	28%	15%
1337	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%
1338	1%	1%	1%	2%	2%	1%	0%	0%	0%	0%
1339	41%	40%	34%	37%	23%	14%	9%	3%	1%	0%
1340	17%	16%	16%	13%	10%	7%	3%	1%	0%	0%
1341	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%
1342	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%
1343	3%	3%	2%	1%	1%	0%	1%	0%	0%	0%
1344	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%
1345	3%	2%	1%	1%	1%	0%	0%	0%	0%	0%
1346	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%
1347	3%	2%	1%	1%	1%	0%	0%	0%	0%	0%

TABLE 5.4 PERCENTAGE OF RESULTED FEASIBLE SOLUTION IN EACH NUMERICAL SETTING RANGE

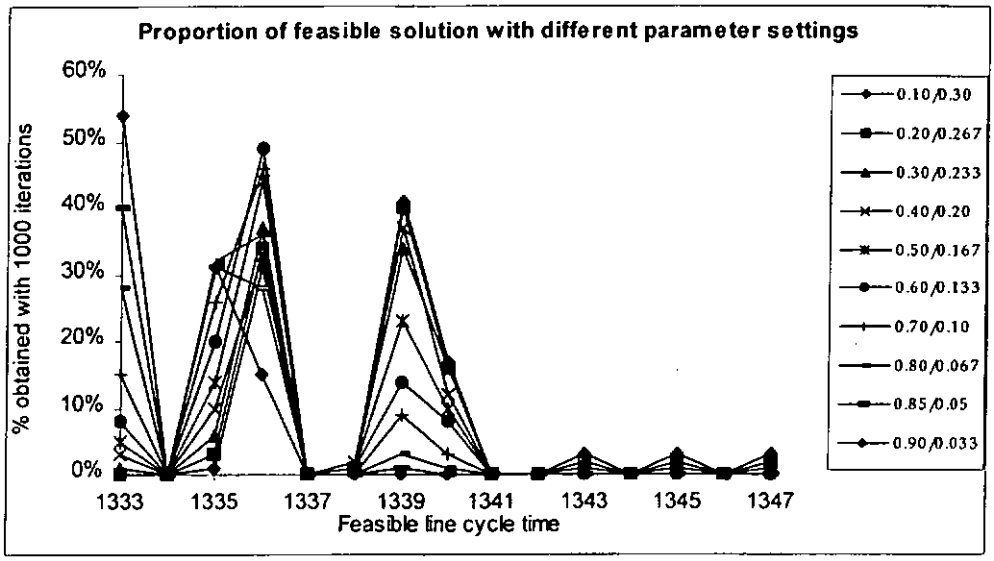


FIGURE 5.4 GRAPHICAL DISTRIBUTION FOR VARIOUS FEASIBLE SOLUTIONS

From the results shown above, there is 5% or less of the runs found to be the optimal solution (i.e. 1333) from setting (1) to (5). There is a trend of increase in the percentage in obtaining the optimal solution. There is also an increase in the percentage in obtaining the feasible solutions between 1335 and 1336. For the feasible solution greater than 1339, there is still a certain proportion in the running samples that are within setting (1) to (5). However, there are almost zero occurrences starting from setting (6). Ranging from the setting (6) towards setting (10), the feasible solutions are focused on 3 key points: 1333, 1335 and 1336. In setting (10), over 50% of the running cycles can obtain the optimal solution. Meanwhile, all other feasible solutions that are greater than 1336 are zero occurrence. This suggests that there is a higher chance to obtain the optimal solution when the numerical setting range tends to be closer to 1. In addition, it will become similar to the classic branch-and-bound algorithm, as the numerical setting

range for one level is closer to one. Moreover, the probability of chance to higher level is reduced onwards.

Investigations are made on how the number of iterations required reaches the optimal solution with different numerical setting ranges. In 1000 running cycles for each numerical setting range, the maximum and minimum number of iterations to obtain the optimum solution (i.e. 1333) are found. Table 5.5 shows the finding on the number of iterations required with graphical representation in Figure 5.5 as below:

Settings	Number of iterations required	
	Minimum	**Maximum**
1	Inf.	Inf.
2	Inf.	Inf.
3	16	32
4	16	35
5	16	47
6	15	55
7	14	91
8	20	127
9	21	133
10	24	170

TABLE 5.5 FINDINGS ON NUMBER OF ITERATIONS REQUIRED TO OBTAIN THE OPTIMAL SOLUTION

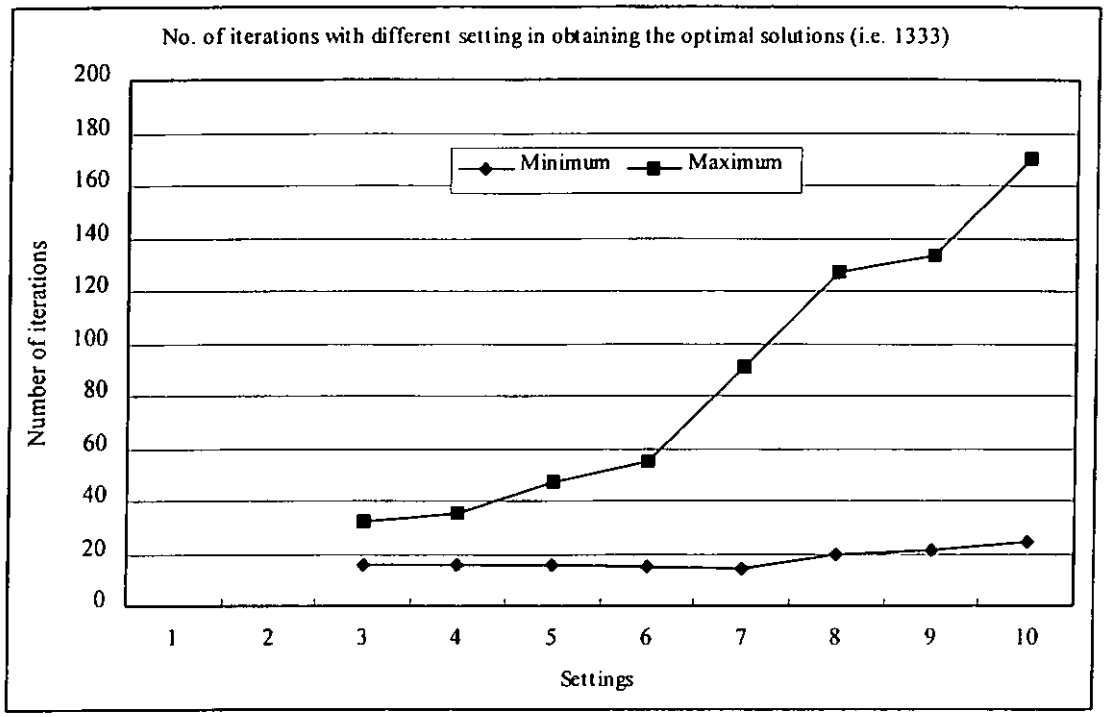


FIGURE 5.5 RANGE OF ITERATIONS IN OBTAINING THE OPTIMUM SOLUTION WITH DIFFERENT SETTINGS

From the graph above, it is found that the maximum numbers of iterations are gradually increased from setting (3) toward (10) and meet the peak of 170 iterations in obtaining the optimum solution. By comparing it with the results obtained from the branch-and-bound algorithm, the number of iterations required to reach the optimum solution is 174 iterations. It is suggested that it is closer to the B&B algorithm when the numerical setting range is narrow and approaches 1.

5.7 The Tabu search heuristic vs. the branch-and-bound algorithm

The fundamental branching moves, which assign different values or bound to integer variables, generate alternatives in the tree search strategy of the branch-and-bound algorithm. The tree search strategy can be readily embedded in the tabu

search heuristic. As the guiding branching decisions in a "TS branching" are considered, the move is represented in two ways: the first one is by leaving the reversed branch in place, and the second one is by shifting the branch to the end of the tree. Since tabu tenures and aspiration criteria may not mean that the latest move is the last to be free from tabu restrictions, unlike the branch-and-bound algorithm, there is no precise meaning to a sequential representation of a tabu search with the branching strategy procedure. The ability to make decisions that are not constrained by sequence, in the more limited sense that sequence is interpreted in the branch-and-bound algorithm, is an important characteristic of TS in this setting. The backtracking operation for the TS heuristic is not one level of nodes only, although the complement branch for the previous node is available. Another difference with the B&B algorithm is the ability to conduct a large part of the search close to a full resolution level, which has implications for the quality of information used to make choices of new moves. The quality of bounds and other information that compose choices made at earlier levels of a tree are not as good as the quality available at deeper levels of the tree. Consequently, a TS trajectory that chooses moves to stay close to full resolution level generally provides better information at each step than that is available from a tree search approach. Moreover, the B&B lies with the decisions made at the earlier level, as it explores descendants of the corresponding nodes. The computation performance can be evaluated by the reduction on the number of iterations in the TS heuristic in comparison with the traditional branch-and-bound algorithm.

5.8 Conclusion

In this chapter, a foregoing application of the tabu search (TS) heuristic demonstrated the usefulness of the approach and the adapting method with the probabilities assigned. The TS heuristic guides the search strategy to continue exploration without becoming confounded by an absence of improving moves, and without falling back into a local optimum. From its ability to diversify the search of solution space, it may gain the computational efficiency in acquiring the optimum or near optimum solutions. The performance in evaluation of the number of iterations in getting the solutions is better than the way of using the traditional branch-and-bound algorithm.

CHAPTER SIX

CONCLUSION

6.1 Implementation

This project studied the line cycle time determination problem by means of grouping different component types to different machine (i.e. Level 2 in the hierarchical relationship of PCBA system). However, in the real-life situation, all problems in the 3 levels should be taken into considerations in order to obtain a complete solution. Furthermore, the placement times used in the case study are only the average times. In a real situation, the placement times vary according to the distance traveled between the feeder and the destination location. Besides, some additional constraints like minimum placement quantity of each component type for machines, should be taken into account to reduce the number of setups by each machine. Finally, a graphical user interface (GUI) should be designed and implemented for end users in practice.

6.2 Conclusions

Optimizing the line cycle time is one of the crucial activities in printed circuit board assembly (PCBA). Appropriate component-machine relationships can lead to shorter line cycle time together with increase in the productivity, which can enhance competitiveness in the market. The determination of the component–machine relationships may be refined to PCB grouping to assembly lines, component grouping to placement machines and sequencing of assembly operations.

This research aimed at exploring the assignment of different component types to multiple non-identical placement machines in order to optimize the line cycle time in PCBA. Various approaches to the determination of optimal or near optimal cycle time including linear programming and heuristics like genetic algorithms were exploited previously. The line cycle time determination problem was initially formulated as a minimax type integer programming. However, due to the non-linearity characteristic of the minimax model, it is difficult to compute such a function with an irregular behavior. A simplified model formulated as an integer linear programming in printed circuit board assembly (PCBA) system is presented. The double-sided PCB assembly model can be sub-divided into two individual ones, being the same as formulating two single-sided PCBA models.

In solving integer linear programming, the branch-and-bound (B&B) algorithm has been widely used. It can be described as corresponding to a tree strategy composed of nodes and branches. The initialization of this algorithm can be done by the resulting solution obtained from the LP relaxation. Previously, the solving of the problem concerned was limited by LP relaxation. The solution was found to be continuous and the integer solution could be acquired by rounding to the nearest integers. However, the rounded solution was feasible but not optimal. By adding extra constraints with bound in the B&B algorithm, it eliminated the current LP relaxed optimum from further considerations. In this project, the depth-first search was chosen as a searching strategy rather than the breath-first search in the B&B algorithm, as it can obtain the optimum solution more efficiently with reduced searching space. Computer programs were written in Matlab and C++. Matlab was selected because it has a large library that has been tested as a powerful tool to solve many engineering problems. Satisfactory results have been obtained during the

algorithm validation process. A comparison between this programming implementation and a commercial software package has been made. It is important to remark that the solution found in the algorithm is the same as the one found in the commercial software with the same mathematical model. However, one of the disadvantages for the B&B algorithm is the length of time required to get the optimal solution. The B&B procedure increases exponentially with the number of variables in the large size problem.

To encounter with the long computational time, a heuristic method, called Tabu search, was designed to seek the optimal solution for the line cycle time determination problem. Tabu search is widely regarded as a high-level heuristic for solving combinatorial optimization problems owing to its ability to overcome the problem of being trapped in a local optimum. The heuristic technique introduces tabu list and restrictions, which attempt to avoid cycling behavior of the algorithm and guide the search process to negotiate in different regions. The commitment of the branching strategy to the TS heuristic with a random assignment in the backtracking procedure has successfully generated the optimal or near optimal solution with less number of iterations and shorter computational time. The implementation of the TS heuristic search takes place with Matlab. The solution is found to be as close as expected. There are almost 50% reduction in both the number of iterations and the computing time, compared with the program of the B&B algorithm. The imperfection for this heuristic in the line cycle time determination problem is that it cannot guarantee the optimal solution in every iteration due to randomly selecting the node in the backtracking operation.

Implementation of the heuristic approach demonstrated that the time required to determine the line cycle time for PCBA can be reduced by comparing with the

B&B algorithm. The solution recommended can contribute to the PCBA system with optimal line cycle time. The major contribution of this research can be summarized as follows:

1. In optimizing the ILP model, the B&B algorithm was found to be a promising approach to guarantee the resulted optimal solution. However, the performance of the B&B algorithm is limited by the number of decision variables. The computational time increases greatly as the number of decision variables grows. In fact, it is not an efficient method for the large size combinatorial problem.
2. This study can be seen as the first attempt to employ the Tabu Search (TS) heuristic approach together with the branching strategy embedded in optimization of a PCBA system. With the use of the TS heuristic approach, the computational time for the optimization problem can be reduced. This can help to generate optimal or close to optimal line cycle time in a more efficient way especially when faced with a large combinatorial problem.

6.3 Further Investigations

Future work related to this project is suggested as follows:

1. Additional constraints to reduce the number of setups.

   Observing the results generated by both the branch-and-bound algorithm and the TS, there were some component type(s) with one or two in quantity being assembled by a machine in the assembly line. It may be impractical to have setup for one unit of that component type, a compromise between the setup times and the optimal cycle time should be is considered. Additional constraints may need to be included to prevent this from happening as well as to reduce the

number of setups.

2. Enriching the TS heuristic's performance.

   In this project, some basic techniques in the TS were performed. It may be worth hybridizing with other heuristic approaches such as genetic algorithms (GA) or artificial neural network (ANN) to enrich the diversification and intensification operations.

REFERENCES

[Ahm91a] Ahmadi, R. H., and Matsuo, H., *The line segmentation problem.* Operations Research, 1991. **39**: pp. 42-55.

[Ahm91b] Ahmadi, R. H., and Tang, C. S., *An operation partitioning problem for automated assembly system design.* Operations Research, 1991. **39**: pp. 824-835.

[Amm85] Ammons, J.C., and Mcginnis, L. F., *CA generation expansion planning model for electric utilities.* Engineering Economist, 1985. **30**(3): pp. 205-226.

[Amm97] Ammons, J.C., Carlyle, M., Cranmer, L., Deepuy, G., Ellis, K., Mcginnis, L. F., Tovey, C. A., and Xu, H., *Component allocation to balance workload in printed circuit card assembly systems.* IIE Transactions, 1997. **29**: pp. 265-275.

[Asa95] Asano, M., and Ohta, H., *Single machine scheduling to minimize weighted earliness subject to ready and due times.* Transactions of the Institue of Systems, Control and Information Engineers, 1995. **8**(10): pp. 523-528.

[Ask94] Askin, R.G., Dror, M., and Varkharia, A. J., *Printed circuit board family grouping and component allocation for a multimachine, open-shop assembly cell.* Naval Research Logistic, 1994. **41**: pp. 587-608.

[Bal88] Ball, M.O., and Magazine, M. J., *Sequencing of insertions in printed circuit board assembly.* Operations Research, 1988. **36**(2): pp. 192-201.

[Bar94] Bard, J.F., Clayton, R. W., and Feo, T. A., *Machine setup and component placement in printed circuit board assembly.* The International Journal of Flexible Manufacturing System, 1994. **6**: pp.

5-31.

[Ben92] Ben-Arieh, D., and Maimon, O., *Annealing method for PCB assembly scheduling on two sequential machines,* International Journal of Computer Integrated Manufacturing, 1992. **5**(6): pp. 361-367.

[Bro96] Broad, K., Mason, A., Ronnqvist, M., and Frater, M., *Optimal robotic component placement.* Journal of the Operation Research Society, 1996. **47**(11): pp. 1343-1354.

[Cam91] Campbell, G. M., and Mabert, V. A., *Cyclical schedules for capacitated lot sizing with dynamic demands.* Management Sciences, **37**: pp. 409-427

[Cap95] Caprara, A. and Fischetti, M.; Maio, D., *Exact and approximate algorithms for the index selection problem in physical database design,* IEEE Transaction on Knowledge and Data Engineering, 1995, **7**(6), pp. 955-967

[Car89] Carmon, T.F., Maimon, O. Z., and Dar-El, E. M., *Group set-up for printed circuit board assembly,* International Journal of Production Research, 1989. **27**(10): pp. 1795-1810.

[Cha89] Chan, D., and Mercier, D., *IC chip insertion: an application of the travelling salesman problem.* International Journal of Production Research, 1989. **27**(10): pp. 1837-1841.

[Cha90] Chang, C.M., and Young, L., *A simultaneous-mounting process for automated printed circuit board assembly.* International Journal of Production Research, 1990. **28**(11): pp. 2051-2064.

[Cra90] Crama, Y., Kolen, A. W. J., Oerlemans, A. G., and Spieksma, F. C. R., *Throughput rate optimization in the automated assembly of*

printed circuit boards. Annals of Operations Research, 1990. **26**: pp. 455-480.

[Dan93] Daniels, R. L., and Mazzola, J. B., *A tabu search heuristic for the flexible-resource flow shop scheduling problem.* Annals of Operations Research, 1993. **41**(1-4): pp. 207-230.

[Das97] Daskin, M.S., Maimon, O., Shtub, A., and Braha, D., *Grouping components in printed circuit board assembly with limited component staging capacity and single card setup: problem characteristics and solution procedures.* International Journal of Production research, 1997. **35**(6): pp. 1617-1638.

[Del93] Dell'Amico, M.A.T., M., *Applying tabu search to the job-shop scheduling problem.* Annals of Operations Research, 1993. **41**: pp. 231-252.

[Des95] Dessouky, M.M., Adiga, S., and Park, K., *Design and scheduling of flexible assembly lines for printed circuit boards.* International Journal of Production Research, 1995. **33**(3): pp. 757-775.

[Dik97] Dikos, A., Nelson, P. C., Tripak, T. M., and Wang, W., *Optimization of high-mix printed circuit card assembly using genetic algorithms.* Annal of Operations Research, 1997. **75**: pp. 303-324.

[Fal92] Falkenauer, E. and Delchambre, A., *A genetic algorithm for bin packing and line balancing*, Proceedings of IEEE 1992 International Conference on Robotics and Automation, Nice, France, 1992. pp.1186-1192.

[Fel93] Feldmann, K., Franke, J., and Rothhaupt, A. *Automated generating and simulation of insertion.* in IEEE/CHMT International Electronic

Manufacturing Technology Symposium. 1993, pp. 206-210.

[Gar70] Garfinkel, R., *An improved algorithm for the bottleneck assignment problem.* Operation Research, 1970. **19**: p. 1747-1751.

[Gar79] Garey, M. R. and Johnson, D. S., *Computer and Intractability.* A Guide to the Theory of NP-Completeness, 1979, San Francisco, CA: W. H. Freeman and Company.

[Gar96] Garetti, M., Pozzetti, A., and Tavecchio, R., *Production scheduling in SMT electronic board assembly.* Production Planning and Control, 1996. 7(2): pp. 197-204.

[Gav91] Gavish, B., and Pirkul, H., *Algorithms for the multi-resource generalized assignment problem.* Management Sciences, **37**: pp. 695-713.

[Glo77] Glover, F., *Heuristics for Integer programming using surrogate constraints.* Decision Sciences, 1977. **8**(1): pp. 156-166.

[Glo89] Glover, F., *Tabu search - Part I.* ORSA Journal of Computing, 1989. **1**: pp. 190-206.

[Glo90] Glover, F., *Tabu search - Part II,* in *Operations Research Society of America*, 1990.

[Glo90] Glover, F., *Tabu search: A tutorial.* Interfaces, 1990. **20**: pp. 74-94.

[Glo93] Glover, F., Wand Laguna, M., *Tabu search. Modern Heuristic Techniques for Combinatorial Problems.* 1993: Blackwell Scientific Publishing, Oxford. pp. 70-141.

[Glo93] Glover, F., *A user's guide to tabu search.* Annals of Operations Research, 1993. **41**: pp. 3-28.

[Gol89] Goldberg, D. E., Genetic Algorithms in Search, Optimization and

Machine Learning, Addison Wesley Published, 1989.

[Gom58] Gomory, R. E., *Outline of an algorithm for integer solution to linear programs.* Bulletin of the American Mathematical Society, 1958. **64**: pp. 275-278.

[Gro84] Grötschel, M. and Junger, M., and Reinelt, G., *A cutting plane algorithm for the linear ordering problem*, Operations Research, 1984. **32**: pp. 1195-1220.

[Gro91a] Grötschel, M. and Holland, O., *Solution of large-scale travelling salesman problems*, Mathematical Programming, 1991. **51**(2): pp. 141-202.

[Gro91b] Grötschel, M. and Jünger, M., and Reinelt, G., *A cutting plane algorithm for the linear ordering problem*, Operations Research, 1984. **32**: pp. 1195-1220.

[Gun96] Gunther, H.O., Gronalt, M., and Piller, F., *Component knitting in semi-automated printed circuit board assembly.* International Journal of Production Economics, 1996. **43**: pp. 213-226.

[Gun98] Gunther, H.O., Gronalt, M., and Zeller, R., *Job sequencing and component set-up on a surface mount placement manchine.* Production, Planning & Control, 1998. **9**(2): pp. 201-211.

[Hil98] Hiller, M.S., and Brandeau, M. L., *Optimal Component Assignment and Board Grouping in Printed Circuit Board Manufacturing.* Operations Research, 1998. **46**(5): pp. 675-689.

[Jiz91] Ji, Z., Leu, M. C., and Wong, H., *Application of linear assignment model for planning of robotic printed circuit board assembly.* ASME Manufacturing Processes and Materials Challenges in Microelectronic

Packaging, 1991. **ADM v131 / EEP v1**: pp. 35-41.

[Jiz93] Ji, Z., Leu, M. C., and Wong, H., *Development and implementation of linear assignment algorithm for assembly of PCB components.* ASME Advances in Electronic Packaging, 1993. **4**(1): pp. 365-371.

[Jip94] Ji, P., Wong, Y. S., Loh, H. T., and Lee, L. C., *SMT production scheduling: a generalized transportation approach.* International Journal of Production Research, 1994. **32**(10): pp. 2323-2333.

[Kap94] Kapov, J.S., *Extensions of a tabu search adaptation to quardratic assignment problem.* Computers and Operations Research, 1994. **21**(8): pp. 855-865.

[Kar84] Karmarkar, N., *A new polynomial-time algorithm for linear programming,* Combinatorica, 1984. **4**: pp. 373-395.

[Kho98a] Khoo, L.P., and Ong, N. S., *PCB assembly planning using genetic algorithms.* The International Journal of Advanced Manufacturing Technology, 1998. **14**: pp. 363-368.

[Kho98b] Khoo, L.P., and Ng, T. K., *A genetic algorithm-based planning system for PCB component placement.* International Journal of Production Economics, 1998. **54**: pp. 321-332.

[Kiu90] Kiu, R., and Salomon, M., *Multi-level lot-sizing problem: evaluation of a simulated annealing heuristic*, European Journal of Operations Research, 1990. **45**: pp. 25-37.

[Kli94] Klincewicz, J. G., and Rajan, A., *Using GRASP to solve the component grouping problem*, Naval Research Logistics, 1994. **41**(7): pp. 893-912.

[Koh95] Kohonen, T., Self Organization Map, 1995. Springer-Verlag, Berlin.

[Kum95] Kumar, R.and Haomin Li., *Integer programming approach to printed circuit board assembly time optimization,* IEEE Transactions, 1995, **18** (4), pp. 720 -727.

[Kum00] Kumar, A., Jacobson, S. H., Sewell, E. C., and Salomon, M., *Computational analysis of a flexible assembly system design problem,* European Journal of Operations Research, 2000. **123**(3): pp. 453-472.

[Laa93] Laarhoven Van, P. J. M., and Zijm, W. H. M., *Production preparation and numerical control in PCB assembly,* International Journal of Flexible Manufacturing System, 1993. **5**(3): pp. 187-207.

[Lan60] Land, A., and Doig, A., *An automatic method of solving discrete programming problems*, Econometric, 1960. **28**(3): pp. 497-520.

[Lap96] Laporte, G., Potvin, J. Y. and Quilleret, F., *A tabu search heuristic using genetic diversification for the clustered traveling salesman problem.* Journal of Heuristics, 1996. **2**: pp. 187-200.

[Leo96] Leon, V.J., and Peters, B. A., *Replanning and analysis of partial setup strategies in printed circuit board assembly systems.* International Journal of Flexible Manufacturing Systems, 1996. **8**(4): pp. 289-412.

[Leo98] Leon, V.J., and Peters, B.A., *A comparison of setup strategies for printed circuit board assembly.* Computers & Industrial Engineering, 1998. **34**(1): pp. 219-234.

[Mai91] Maimon, O.Z., and Shtub, A., *Grouping method for printed circuit board assembly.* International Journal of Production Research, 1991. **29**(7): pp. 1379-1390.

[Mai93] Maimon, O.Z., and Dar-El, E. M., *Set-up schemes for printed circuit boards assembly.* European Journal of Operational Research, 1993.

70: pp. 177-190.

[Mai98] Maimon, O.Z., and Braha, D., *A genetic algorithm approach to scheduling PCBs on a single machine.* International Journal of Production Research, 1996. **36**(3): pp. 761-784.

[Mcg92] McGinnis, L.F., Ammons, J. C., Carlyle, M., Cranmer, L., Depuy, G. W., Ellis, Y. P., Tovey, C. A., and Xu, H., *Automatic process planning for printed circuit card assembly.* IIE Transactions, 1992. **24**(4): pp. 18-30.

[Muk92] Mukai, S., PCB continuous line system proceeds from manufacture to inspection. Journal of Electronics Engineering, May 1994. pp. 34-39.

[Nas96] Nash, S. G. and Sofer, A.: *Linear and Nonlinear Programming.* The McGraw-Hill Company, 1996.

[Nay00] Nayak, A., and Sharma, J.: *A hybrid neural network and simulated annealing approach to the unit commitment problem.* Computers and Electrical Engineering, 2000. **26**(6): pp. 461-477.

[Nel95] Nelson, K.M., and Wille, L. T. *Comparative study of heuristics for optimal printed circuit board assembly.* in *Southcon Conference Record.* 1995.

[Nem88] Nemhauser, G. L. and Wolsey, L. A., Integer and Combinatorial Optimization, John Wiley, New York, 1988.

[Noo91] Noon, C. E. and Bean, J. C., *A Lagrangian based approach for the asymmetric generalized travelling saleman problem.* Operations Research, 1991. **39**: pp. 623-632.

[Pir96] Pirlot, M., *General local search methods.* European Journal of Operational Research, 1996. **92**: pp. 493-511.

[Ree93] Reeves, C.R., *Improving efficiency of tabu search for machine sequencing problme.* Journal of Operations Research Society, 1993. **44**(4): pp. 382-385.

[Ree95] Reeves, C. R., *Genetic algorithms and combinatorial optimization*, in Rayward-Smith, V. J., Applications of Modern Heuristic Methods, 1995. pp. 111-126, Alfred Waller, Henley-on-Thames, UK.

[Ree96] Rayward-Smith, V. J., Osman, I. H., Reeves, C. R. and Smith, G. H., *Modern Heuristic Search Method,* John Wiley & Sons Ltd., 1996

[Sad93] Sadiq, M., Landers, T. L., and Don Taylor, G., *A heuristic algorithm for minimizing total production time for a sequence of jobs on a surface mount placement machine.* International Journal of Production Research, 1993. **31**(6): pp. 1327-1341.

[San95] Santos, D., Kane, J., Caballero, F., and Nagarajan, K., *On the selection of a printed circuit board assembly line system.* Computer Industrial Engineering, 1995. **29**(1-4): pp. 591-595.

[Sch86] Schrijver, A., *Theory of linear and integer programming, Modern Heuristic Search Method,* John Wiley & Sons Ltd., 1986.

[Sch96] Schaffer, J. David and Eshelman, J. Larry, *Combinatorial optimization by genetic algorithms: the value of the genotype / phenotype distinction*, in Rayward-Smith, V. J., Osman, I. H., Reeves, C. R. and Smith, G. H., *Modern Heuristic Search Method,* John Wiley & Sons Ltd., 1996.

[Sht92] Shtub, A., and Maimon, O. Z., *Role of similarity measures in PCB grouping procedures.* International Journal of Production Research, 1992. **30**(5): pp. 973-983.

[Spr98] Sprecher, A., *Competitive branch-and-bound algorithm for the simple assembly line balancing problem.* International Journal of Production Research, 1992. **30**(5): pp. 973-983.

[Suy96] Su, Y. Y., and Srihari, K., *Placement sequence identification using artificial neural networks in surface mount PCB assembly.* The International Journal of Advanced Manufacturing Technology, 1996. **11**(4): pp. 285-299.

[Sze98] Sze, M.T., *Component grouping for printed circuit board assembly*, in *Department of Manufacturing Engineering*. 1998, The Hong Kong Polytechnic University.

[Tah75] Taha, H.A., Integer programming: theory, application and computations. 1975: Academic Press, Inc.

[Vak97] Vakharia, A.J.a.C., Y. L., *Cell formation in-group technology: A combinatorial search approach.* International Journal of Production Research, 1997. **35**(7): pp. 2043.

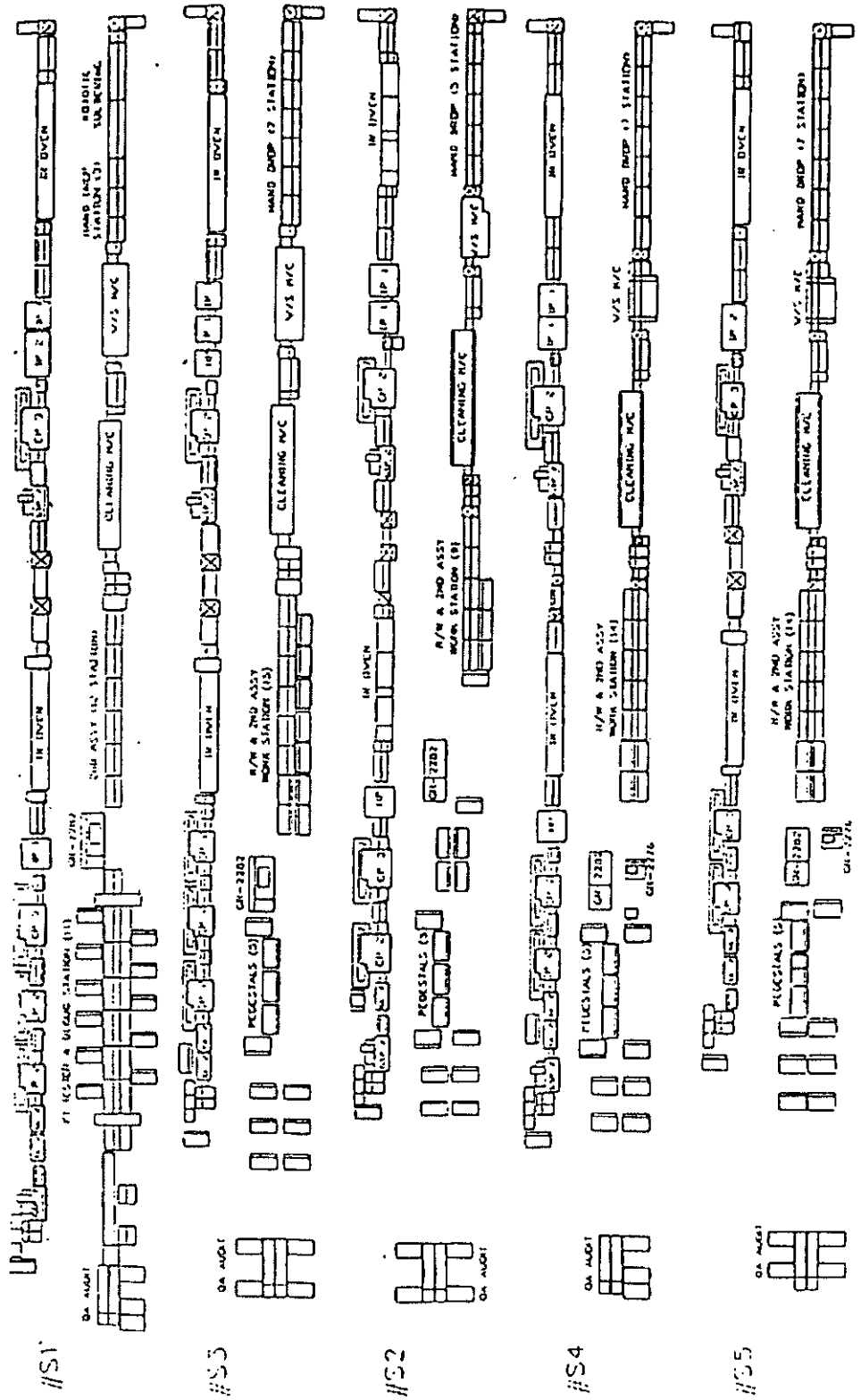
[Wat95] Watkins, R.E., and Cochran, J. K., *A line balancing heuristic case study for existing automated surface mount assembly line setups.* Computers and Industrial Engineering, 1995. **29**(1-4): pp. 681-685.

[Whi91] Whitley, D., Starkweather, T., and Shaner, D., *The travelling salesman and sequence scheduling: quality solutions using genetic edge recombination*, in Davis, L., Handbook of Genetic Algorithms, Van Nostrand Reinhold, New York, 1991

[Woo92] Woodruff, D. L., and Spearman, M. L., *Sequencing and batching for two classes of jobs with deadlines and setup times,* The Journal of Production and Operation Management Society, 1992, pp. 26-34.

APPENDIX I

LINE LAYOUT FOR AN ELECTRONIC COMPANY



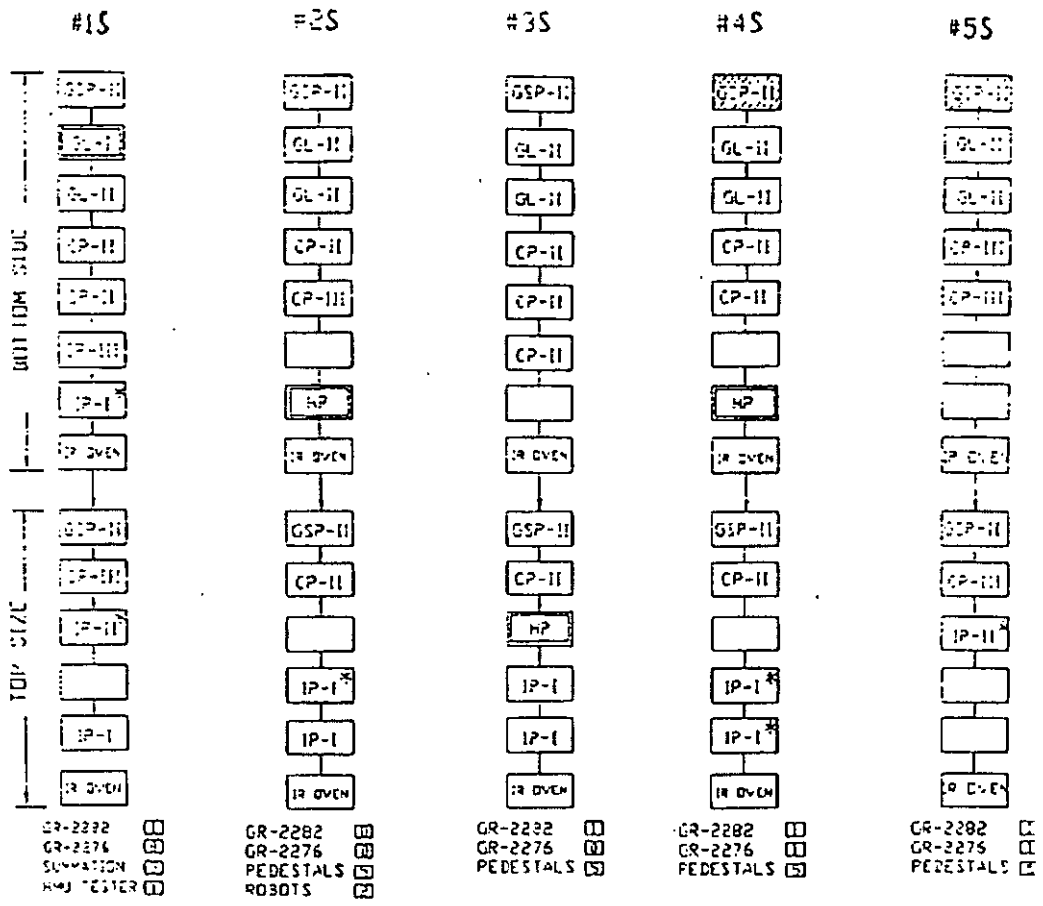
ATTACHMENT 1: Plant Layout

APPENDIX II

LINE CONFIGURATION

MANUFACTURING ENGINEERING XXX Company

SMT LINES CONFIGURATION



LEGEND :-

- UNOCCUPIED MACHINE SPACE
- GSP-II AVAILABLE BY SEP
- DIRECT DRIVE

REVISION : 03
FILE : E:\ONG\SLCI
DATE : .

APPENDIX III

COMPONENT PLACEMENT TIMES

MACHINE BASIC TIME

		TIME
A) CP II & CP III		
1) Chip Components (Resistor, Capacitor, Diode, LED, Transistor)	:	0.3 sec/comp.
2) SOIC (8~28 PINS)	:	0.7 sec/comp.
3) Tantalum	:	0.7 sec/comp.
* Fudicial mark, load & unlod PCB board	:	11.0 sec/panel
B) IP-I		
1) Chip Components (Resistor, Capacitor, Diode, LED, Transistor)	:	1.5 sec/comp.
2) SOIC (8~ 28 PINS)	:	2.5 sec/comp.
3) SOJ IC (24 ~ 28 PINS)	:	3.5 sec/comp.
4) PLCC (20~84 PINS)	:	3.5 sec/comp.
5) PLCC SOCKET	:	3.5 sec/comp.
6) QFP IC (48~160 PINS)	:	3.5 sec/comp.
7) SM CONNECTOR (32~44 x 12~24 size)	:	3.5 sec/comp.
* Fudicial mark, load & unlod PCB board	:	14.67 sec/panel
C) IP-II		
1) Chip Components (Resistor, Capacitor, Diode, LED, Transistor)	:	0.7 sec/comp.
2) SOIC (8~ 28 PINS)	:	1.2 sec/comp.
3) SOJ IC (24 ~ 28 PINS)	:	1.7 sec/comp.
4) PLCC (20~84 PINS)	:	1.7 sec/comp.
5) PLCC SOCKET	:	1.7 sec/comp.
6) QFP IC (48~160 PINS)	:	1.7 sec/comp.
7) SM CONNECTOR (32~44 x 12~24 size)	:	1.7 sec/comp.
* Fudicial mark, load & unlod PCB board	:	14.67 sec/panel
D) HP		
1) Chip Components (Resistor, Capacitor, Diode, LED, Transistor)	:	2.3 sec/comp.
2) SOIC (8~ 28 PINS)	:	3.8 sec/comp.
3) SOJ IC (24 ~ 28 PINS)	:	3.8 sec/comp.
4) PLCC (20~84 PINS)	:	3.8 sec/comp.
5) PLCC SOCKET	:	3.8 sec/comp.
6) QFP IC (48~144 PINS)	:	3.8 sec/comp.
* Fudicial mark, load & unlod PCB board	:	14.67 sec/panel

APPENDIX IV

SOURCE CODES FOR THE B&B ALGORITHM IN C++

```
#include <iostream.h>
#include <stdlib.h>
#include <conio.h>
#include <stdio.h>
#include <math.h>
#include <time.h>
#include <stdio.h>
#include <dos.h>
#include <conio.h>


#define MAX 90

void mcsetup(double Setup[MAX], int machine, double B[MAX])
{
            cout<<"\n"<<"\n";
            for (int i=1; i<=machine; i++)
            { cout << "Enter the set up time for machine M"<<i<<" in second:  ";
     cin >> Setup[i];

     B[i]=Setup[i];
     }
    cout <<"\n"<<"\n";

}

void comptype(double Quantity[MAX], int component, int machine, double B[MAX])
{
        for (int i=1; i<=component; i++)
      {cout << "Enter the quantity for component type C" <<i<<": ";
    cin >> Quantity[i];
    B[machine+i]=Quantity[i];

    }

}

void plactime(double Time[MAX][MAX], int component, int machine)
{
      cout << "\n\nPlease enter the component placement time. \n";
  cout << "Tij refers to component placement time for component j at machine i.\n";
    for (int j=1; j<=component; j++)
  { for (int i=1; i<=machine; i++)
    {cout << "T["<<i<<"]["<<j<<"]:";
     cin >> Time[i][j];}
     cout << endl; }
}

double abstemp(double KKK)
{
      if (KKK<0)
      { KKK=-KKK; }
  return KKK;
}


void INV(double matrix[MAX][MAX], int N, int L)
{
int IS[MAX], JS[MAX];
```

```
double D, T;


for (int K=1; K<=N; K++)
{
      D=0.0;                          //this part gets the maximum volume from the matrix
  for (int I=K; I<=N; I++)
  {
      for (int J=K; J<=N; J++)
    {
    if (abstemp (matrix[I][J])> D)
      {  D= abstemp (matrix[I][J]);
        IS[K]=I;
                JS[K]=J;
      }
    }
  }                                   //end getting the maximum volume from the matrix


  if (D==0)
      {
     cout<<"matrix is sigular"<<endl;
     L=0;

     char addd;
     cin>>addd;
     return;
    }

  for (int J=1; J<=N; J++)    //rows exchange
  {
      T=matrix[K][J];
    matrix[K][J]=matrix[IS[K]][J];
    matrix[IS[K]][J]=T;
  }                                     //end rows exchange

  for (int I=1; I<=N; I++)                    //cols exchange
  {
      T=matrix[I][K];
    matrix[I][K]=matrix[I][JS[K]];
    matrix[I][JS[K]]=T;
  }                     //ends cols exchange


  matrix[K][K]=1/matrix[K][K];        //change the volume of row K

  for (int J=1; J<=N; J++)
  {
      if (J!=K)
     matrix[K][J]= matrix[K][K]*matrix[K][J];
  }



  for (int I=1; I<=N; I++)          //change the volume of cells except Row K and Col K
  {
    if (I!=K)
    {
      for (int J=1; J<=N; J++)
```

```
        {
                if (J!=K)
            matrix[I][J]= matrix[I][J]-matrix[I][K]*matrix[K][J];
          }
      }
    }                       //end changing the volume of cells except Row K and Col K

    for (int I=1; I<=N; I++)
    {
        if (I!=K)
          matrix[I][K]=-matrix[I][K]* matrix[K][K];
    }                       //end changing the volume of row K

}



for (int K=N; K>=1; K--)
{
      for (int J=1; J<=N; J++)
    {
        T=matrix[K][J];
      matrix[K][J]=matrix[JS[K]][J];
      matrix[JS[K]][J]=T;
    }

    for (int I=1; I<=N; I++)
    {
        T=matrix[I][K];
      matrix[I][K]=matrix[I][IS[K]];
      matrix[I][IS[K]]=T;
    }
}


}

void MUL(double A[MAX][MAX], double B[MAX][MAX],int M, int N, int K, double
C[MAX][MAX])
{
  for (int i=1; i<=M; i++)
  {           for (int j=1; j<=K; j++)
      {  C[i][j]=0;
                  {       for (int kk=1; kk<=N; kk++)
          C[i][j]=C[i][j]+A[i][kk]*B[kk][j];
        }
      }
  }

}

void LPLQ(double A[MAX][MAX], double B[MAX], double C[MAX], double X[MAX], int M, int N,
int MN, double S, int PINT[MAX], int L)

{
int JS[MAX];
double P[MAX][MAX],D[MAX][MAX];
int K, J;
```

```
double DD, Z, Y;
int slack, slack1;

slack=N;

 for (int I=1; I<=M; I++ )
 { for (int J=N+1; J<=MAX; J++)
 {A[I][J]=0;}
}

  for (int i=1; i<=M; i++)
  {
      if (PINT[i]==2)
    {
      slack=slack+1;
      A[i][slack]=-1;
      C[slack]=0;
    }
  }

  slack1=slack;

  for (int i=1; i<=M; i++)
  {
      if (PINT[i]==2)
    {
      slack=slack+1;
      A[i][slack]=1;
      C[slack]=1.0E+8;
    }

    if (PINT[i]==1)
    {
      slack=slack+1;
      A[i][slack]=1;
      C[slack]=0;
    }

    if (PINT[i]==0)
    {
      slack=slack+1;
      A[i][slack]=1;
      C[slack]=1.0E+8;
    }
}

MN=slack;

for (int I=1; I<=M; I++)
     {JS[I]=slack1+I;}


Again:

//L=1;
for (int I=1; I<=M; I++)
{
   for (int J=1; J<=M; J++)
   {
```

```
      P[I][J]=A[I][JS[J]];
    }
}

INV(P, M, L);   //outout inverse P

MUL(P, A, M, M, MN, D);   //output D

for (int I=1; I<=MN; I++)
{X[I]=0.0;}

for (int I=1; I<=M; I++)
{
    S=0.0;
  for (int J=1; J<=M; J++)
    {  S=S+P[I][J]*B[J];}

    X[JS[I]]=S;

}


K=0;
DD=1.0E-5;
for (int J=1; J<=MN; J++)
{
    Z=0.0;
 for (int I=1; I<=M; I++)
 {  Z=Z+C[JS[I]]*D[I][J];}
    Z=Z-C[J];

    if (Z>DD)
    {
    DD=Z;
      K=J;
    }

}


if (K==0)
{
    S=0.0;
    for (int J=1; J<=MN; J++)
    {  S=S+C[J]*X[J]; }

 for (int J=1; J<=MN; J++)
    {
    if (X[J]!=0)
    {
      if (abstemp(X[J])<1.0E-10)   // attention 10 is changed into 2
                {X[J]=0;}
          if (abstemp(ceil(X[J])-X[J])<1.0E-10)
                {X[J]=ceil(X[J]);}
          if (X[J]-floor(X[J])<1.0E-10)
                {X[J]=floor(X[J]);}
    }

 }
```

```
    // cout<<"the optimal solution of linear programming is S: "<<S<<endl;
     return ;
    }

    J=0;
    DD=1.0E+20;
    for (int I=1; I<=M; I++)
    {
       if (D[I][K]>=1.0E-10)
     {
       Y=X[JS[I]]/D[I][K];
       if (Y<DD)
       {
       DD=Y;
         J=I;
       }
     }
    }

    if (J==0)
    {
       L=0;
     cout<<"frot testing JJ and L"<<L;
       return;
    }

    JS[J]=K;
    goto Again;
}


void roundoff(double A[MAX][MAX], double XB[MAX][MAX], int N, int machine, int component,
double B[MAX], int counter, double SB[MAX])
{
  double temp[MAX][MAX];
  int KKK;

  for (int J=1; J<=component; J++)
  {
    for (int I=1; I<=machine; I++)
       {
       if (XB[counter][((I-1)*component+J+1)]!=0)
       { KKK=I;
        temp[J+machine][((I-1)*component+J+1)]=ceil(XB[counter][((I-1)*component+J+1)])-
XB[counter][((I-1)*component+J+1)];
        if (temp[J+machine][((I-1)*component+J+1)]>=0.5)
        { XB[counter][((I-1)*component+J+1)]=floor(XB[counter][((I-1)*component+J+1)]);}
        if (temp[J+machine][((I-1)*component+J+1)]<0.5)
        { XB[counter][((I-1)*component+J+1)]=ceil(XB[counter][((I-1)*component+J+1)]);}
       }
    }

    double S=0;
      for (int I=1; I<=machine; I++)
    {
      S=S+XB[counter][((I-1)*component+J+1)];
```

```
    }
    XB[counter][(KKK-1)*component+J+1]=XB[counter][(KKK-1)*component+J+1]+B[J+machine]-
S;

  }

      for (int I=1; I<=machine; I++)
  { double SB1=0;
    for (int J=2; J<=N; J++)
    {
      SB[I]=((-A[I][J])*XB[counter][J]);
      //cout<<"XB["<<counter<<"]["<<J<<"]: "<<XB[counter][J]<<endl;
      //cout<<"(-A["<<I<<"]["<<J<<"]) "<<(-A[I][J]);
      SB1=SB[I]+SB1;

    }

    SB[I]=SB1;
      SB[I]=SB[I]+B[I];


  }
  double SB2=0;
  for (int I=1; I<=machine; I++)
  {
     if (SB2<SB[I])
    { SB2=SB[I]; }
  }

  SB[counter]=SB2;

}

//////////////////////////////////////////////////////////////
//                                                          //
//          The following is the Main Programme             //                    //
//                                                          //
//////////////////////////////////////////////////////////////

int main()
{
int L=1;
int M, N, MN;
int Mleft;
double S=0.0, SS[MAX], SB[MAX];
double A[MAX][MAX], B[MAX], C[MAX], BB[MAX][MAX];
double X[MAX], XB[MAX][MAX],XX[MAX][MAX];
int PINT[MAX];
int counter;
int check;
int machine, component;
double Setup[MAX], Quantity[MAX], Time[MAX][MAX];
time_t first, second;
int iteration=0;
```

```
    cout<<"please enter the number of machine: ";
    cin>>machine;
    cout<<endl<<"please enter the number of components: ";
    cin>>component;
    mcsetup(Setup, machine, B);
    comptype(Quantity, component, machine, B);
    plactime(Time, component, machine);

    first = time(NULL);

        M=machine+component;
        N=machine*component+1;
    MN=M+N;

    for (int I=1; I<=M; I++)
    {
        if (I<=machine)
      {A[I][1]=1;}
      else
      {A[I][1]=0;}
    }
    for (int J=2; J<=N; J++)
    {
        for (int I=1; I<=M; I++)
      {
        A[I][J]=0;
      }
    }

    for (int I=1; I<=machine; I++)
    {
      for (int J=1; J<=component; J++)
      {
        A[I][(I-1)*component+J+1]=-Time[I][J];
        A[J+machine][(I-1)*component+J+1]=1;
      }
    }

    for (int J=2; J<=N; J++)
    {
      C[J]=0;
    }
    C[1]=1;
  for (int I=1; I<=M; I++)
  {
        if (I<=machine)
    {PINT[I]=2;}
    if (I>machine)
    {PINT[I]=0;}

  }

  iteration+=1;
  LPLQ(A, B, C, X, M, N, MN, S, PINT, L);
  for (int I=1; I<=N; I++)
    for (int I=1; I<=N; I++)
    {
        if (X[I]<0.01)
        {X[I]=0;}
```

```
        if ((ceil(X[I])-X[I])<0.01)
        {X[I]=ceil(X[I]);}
        if (X[I]-(floor(X[I]))<0.01)
        {X[I]=floor(X[I]);}
    }
S=0;
for (int I=1; I<=MN; I++)
{S=S+C[I]*X[I];}

int check0=0; //this part to check whether the solution of linear programming is optimal
for (int J=1; J<=N; J++)
{
  if (abstemp(abstemp(ceil(X[J])-abstemp(X[J])))>1.0E-6)
  {check0+=1;}
}

if (check0==0)
{       cout<<endl<<endl<<endl<<endl<<endl;
        cout<<"*****************************************************"<<endl<<endl;

      cout<<"the optimal integer function value: "<<S<<endl;
      cout<<"the optimal solution: "<<endl;

        for (int I=1; I<=machine; I++)
                {
                for (int J=1; J<=component; J++)
                {
                      XX[I][J]=X[(I-1)*component+J+1];

                }
                }

        for (int I=1; I<=machine; I++)
                {
                for (int J=1; J<=component; J++)
                {
           cout.width(5);
           cout<<"XX["<<I<<"]["<<J<<"]: "<<XX[I][J];
                }
         cout<<endl;
                }

        cout<<endl;
        cout<<"*****************************************************"<<endl<<endl;
        cout<<"The iteration time(s): "<<iteration<<endl;
         second = time(NULL);
         printf("The running time is: %f seconds\n",difftime(second,first));
        cout<<"*****************************************************"<<endl<<endl;
        cout<<"Enter y/n to terminate it"<<endl;
        char temppp;
        cin>>temppp;



  return 0;
}       //end checking whether the solution of linear programming is optimal

counter=M;
```

```
SS[counter]=S;  //geting the upper bound SB[0]
for (int J=1; J<=N; J++)
{
      XX[counter][J]=X[J];
      XB[counter][J]=X[J];
}
cout<<"the current SS[0] is: "<<SS[counter]<<endl;
roundoff(A, XB, N, machine, component, B, counter, SB);
cout<<"the current SB["<<counter<<"] is: "<<SB[counter]<<endl;
cout<<"the current XB is: "<<endl;
for (int J=1; J<=N; J++)
{
      if (XB[counter][J]!=0)
  {cout<<"XB["<<counter<<"]["<<J<<"]is: "<<XB[counter][J]<<endl;
  cout<<"";
  cout<<"XX["<<counter<<"]["<<J<<"]is: "<<XX[counter][J]<<endl;
  cout<<"";
  }
}     //geting the upper bound SB[0] and XB[0][J]
cout<<"";


leftside:
int krun=0;

int JX;
double TX=0;
for (int J=2; J <= N ; J++)
{

      double number=0;
      if (XX[counter][J]>=0.001)
  {
      number=ceil(XX[counter][J])-XX[counter][J];
    if (number>=0.01)
    {
      krun=1;
      if (TX<XX[counter][J])
      {
      TX=XX[counter][J];
      JX=J;
      }
    }
  }
}

if (krun==1)
{
      M+=1;
  PINT[M]=1;
  B[M]=floor(XX[counter][JX]);
  BB[M][1]=B[M];
  BB[M][2]=BB[M][1]+1;
  cout<<"\nThe left B["<<M<<"] is: "<<B[M]<<endl;
  cout<<"";
  A[M][JX]=1;
  for (int J1=1; J1<=N; J1++)
  {
      if (J1!=JX)
```

```
    {A[M][J1]=0;}
}

for (int J=1; J<=N; J++)
{X[J]=XX[counter][J];}


iteration+=1;
    LPLQ(A, B, C, X, M, N, MN, S, PINT, L);
cout<<"";
for (int I=1; I<=N; I++)
    {
        if (X[I]<0.01)
            {X[I]=0;}
            if ((ceil(X[I])-X[I])<0.01)
            {X[I]=ceil(X[I]);}
            if (X[I]-(floor(X[I]))<0.01)
            {X[I]=floor(X[I]);}
}
for (int I=1; I<=N; I++)
{
    if (X[I]!=0)
    {cout<<"X["<<I<<"]is:"<<X[I]<<endl; }

}
cout<<"the current N is: "<<N<<endl;
cout<<"";

 if (L==1)
{
  counter+=1;
  for (int J=1; J<=N; J++)
  {
   XX[counter][J]=X[J];
     XB[counter][J]=X[J];
  }
 S=0;
 for (int J=1; J<=N; J++)
        {S=S+C[J]*X[J];}
   check=0;
   for (int J=1; J<=N; J++)
   {
   if (ceil(XX[counter][J])-XX[counter][J]>0.001)
   {check+=1;}
   }

   if (check==0)//if we have got an integer solution
   {
   if (S<=SB[counter-1])
        { cout<<endl<<endl<<endl<<endl<<endl;

   cout<<"*********************************************************"<<endl<<endl;
            cout<<"the optimal integer function value: "<<S<<endl;
   cout<<"the optimal solution: "<<endl;

   for (int I=1; I<=machine; I++)
            {
            for (int J=1; J<=component; J++)
            {
```

```
                    XX[I][J]=X[(I-1)*component+J+1];
            }
        }

    for (int I=1; I<=machine; I++)
                {
                for (int J=1; J<=component; J++)
                {
                cout.width(5);
                cout<<"XX["<<I<<"]["<<J<<"]: "<<XX[I][J];
                }
            cout<<endl;
        }

        cout<<endl;
        cout<<"*********************************************************"<<endl<<endl;
        cout<<"The iteration time(s): "<<iteration<<endl;
        second = time(NULL);
        printf("The running time is: %f seconds\n",difftime(second,first));
        cout<<"*********************************************************"<<endl<<endl;
        cout<<"Enter y/n to terminate it"<<endl;
        cout<<endl<<endl<<endl<<endl;
        char tempp;
        cin>>tempp;



                return 0;
    }


    if (S>SB[counter-1])
    {
    for (int J=M; J>(machine+component);J--)
    {
            if (PINT[J]==1)
            {
                    Mleft=J;
                    cout<<"teh current PINT["<<J<<"] is"<<PINT[J];
                    cout<<"the current Mleft is"<<Mleft;
                    cout<<"";
                    break;
            }
    }
    M=Mleft;
    counter=M-1;
    cout<<"the current M is: "<<M<<endl;
    cout<<"the current counter is "<<counter<<endl;
    cout<<"";
    goto rightside;
    }

}

     if (check!=0)//still fraction solution
     {
                if ((SB[counter-1]-S)<1)
          {  counter=M-1;
```

```
            goto rightside; }

            if ((SB[counter-1]-S)>=1)

          {
            roundoff(A, XB, N, machine, component, B, counter, SB);

            if (SB[counter]>=SB[counter-1])
            { SB[counter]=SB[counter-1];}
           cout<<"the current SB["<<counter<<"]: "<<SB[counter]<<endl;
           cout<<"";
            goto leftside;
             }


        }
      }

      if (L==0)
      {
        counter=M-1;
        goto rightside;
      }
}



if (krun==0)
{       cout<<endl<<endl<<endl<<endl<<endl;
       cout<<"*********************************************************"<<endl<<endl;

     cout<<"the optimal integer function value: "<<S<<endl;
     cout<<"the optimal solution: "<<endl;

       for (int I=1; I<=machine; I++)
                {
                for (int J=1; J<=component; J++)
                {
                      XX[I][J]=X[(I-1)*component+J+1];

                }
                }

       for (int I=1; I<=machine; I++)
                {
                for (int J=1; J<=component; J++)
                {
          cout.width(5);
          cout<<"XX["<<I<<"]["<<J<<"]: "<<XX[I][J];
                }
         cout<<endl;
                }

       cout<<endl;
       cout<<"*********************************************************"<<endl<<endl;
       cout<<"The iteration time(s): "<<iteration<<endl;
        second = time(NULL);
        printf("The running time is: %f seconds\n",difftime(second,first));
       cout<<"*********************************************************"<<endl<<endl;
```

```
        cout<<"Enter y/n to terminate it"<<endl;
        cout<<endl<<endl<<endl<<endl;
        char tempp;
        cin>>tempp;


return 0;
}


rightside:

  PINT[M]=2;
  B[M]=BB[M][2];
  cout<<"\nThe (Right) current B["<<M<<"] is: "<<B[M]<<endl;
  cout<<"";

  /*for (int I=1; I<=M; I++)
  {
      for (int J=1; J<=MN;J++)
    {if (A[I][J]!=0)
    {cout.width(4);
    cout<<"A["<<I<<"]["<<J<<"]"<<A[I][J];}}
    cout<<endl;
  }
  cout<<"";*/

  for (int J=1; J<=N; J++)
  {X[J]=XX[counter][J];}

  iteration+=1;
      LPLQ(A, B, C, X, M, N, MN, S, PINT, L);
  cout<<"";
  for (int I=1; I<=N; I++)
  {
      if (X[I]<0.01)
      {X[I]=0;}
      if ((ceil(X[I])-X[I])<0.01)
      {X[I]=ceil(X[I]);}
      if (X[I]-(floor(X[I]))<0.01)
      {X[I]=floor(X[I]);}
  }
        for (int I=1; I<=N; I++)
      {
      if (X[I]!=0)
      {cout<<"X["<<I<<"]is:"<<X[I]<<endl; }

      }
      cout<<"the current N is: "<<N<<endl;
      cout<<"";
  if (L==1)
  { counter+=1;
    S=0;
            for (int I=1; I<=N; I++)
            {S=S+C[I]*X[I];}
    for (int J=1; J<=N; J++)
    {XX[counter][J]=X[J];
    XB[counter][J]=X[J];}
    check=0;
```

```
for (int J=1; J<=N; J++)
{
 if (ceil(XX[counter][J])-XX[counter][J]>0.01)
 {check+=1;}
}

if (check==0)//if we have got an integer solution
{
  if (S<=SB[counter-1])
  { cout<<endl<<endl<<endl<<endl<<endl;
    cout<<"*****************************************************"<<endl<<endl;

  cout<<"the optimal integer function value: "<<S<<endl;
  cout<<"the optimal solution: "<<endl;

    for (int I=1; I<=machine; I++)
                {
                for (int J=1; J<=component; J++)
                {
                      XX[I][J]=X[(I-1)*component+J+1];

                }
                }

   for (int I=1; I<=machine; I++)
                {
                for (int J=1; J<=component; J++)
                {
        cout.width(5);
        cout<<"XX["<<I<<"]["<<J<<"]: "<<XX[I][J];
                }
     cout<<endl;
                }

   cout<<endl;
   cout<<"*****************************************************"<<endl<<endl;
   cout<<"The iteration time(s): "<<iteration<<endl;
    second = time(NULL);
    printf("The running time is: %f seconds\n",difftime(second,first));
   cout<<"*****************************************************"<<endl<<endl;
   cout<<"Enter y/n to terminate it"<<endl;
   cout<<endl<<endl<<endl<<endl;
   char tempppp;
   cin>>tempppp;

return 0;
}

if (S>SB[counter-1])
{
for (int J=M; J>(machine+component);J--)
  {
   if (PINT[J]==1)
   {
              Mleft=J;

              break;
   }
  }
```

```
            M=Mleft;
            counter=M-1;

            goto rightside;
          }
        }


      if (check!=0)//still fraction solution
      {
        if ((SB[counter-1]-S)>=1)
        {
              roundoff(A, XB, N, machine, component, B, counter, SB);
              if (SB[counter]>=SB[counter-1])
              { SB[counter]=SB[counter-1];}
          cout<<"the current SB["<<counter<<"] is: "<<SB[counter]<<endl;
          cout<<"";
          goto leftside;

        }

        if ((SB[counter-1]-S)<1)
        {
         for (int J=M; J>(machine+component);J--)
         {
          if (PINT[J]==1)
          {
          Mleft=J;

          break;}
         }
         M=Mleft;
         counter=M-1;

         goto rightside;
        }
      }
    }
    if (L==0)
    {

    for (int J=M; J>(machine+component);J--)
    {
      if (PINT[J]==1)
      {
        Mleft=J;
        break;
      }
    }
    M=Mleft;
    counter=M-1;
    goto rightside;

    }

}
```

APPENDIX V

SOURCE CODES FOR THE B&B ALGORITHM IN MATLAB

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%
%%%                       User Interface Input Data
%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

disp('  Select the Input Mode ');
disp('    INTERACTIVE MODE ------------ 1 ');
disp('    FILE MODE FROM DATA FILE ---- 2 ');

  CH =input('Enter 1 or 2 ===>  ');

while (CH ~= 1 & CH ~= 2)== 1
 CH=input('Please Re-enter the choice of input mode - 1 or 2:  ');
   if CH==1 | CH==2
     break;
   end
end

if CH==1
      InputData;
  else
      [filename,path] = uigetfile('*.mat', 'Get File');
      eval(['load' [path filename]]);
end

disp('Time is >')
disp(PT)
disp('Quantity is >')
disp(Q)
disp('Setup time is >')
disp(Setup)


m1=M;
m2=C;

a = zeros(m1+m2, m1*m2+1);
for row = 1:m1+m2
  for col = 1:(m1 * m2) + 1
    if row <= m1
      if col == 1
        a(row, col) = -1;
      else
        row_m1 = floor((col-2)/C +1);
        col_m1 = (mod(col-2, C)) +1;
        a(row_m1, col) =  PT(row_m1, col_m1);
      end
    else
      for col = 2 : (m1*m2) + 1
        if (mod(col-row+m1-1, m2) == 0)
          a(row, col) = 1;
        end
      end
    end
  end
end

b = zeros(m1+m2, 1);
```

```
for i = 1:m1+m2
  if i <= m1
    b(i) = -Setup(i);
  else
    b(i) = Q(i-m1);
  end
end

c = zeros(1, m1*m2+1);
for i = 1:m1*m2+1;
     if i== m1*m2+1
     c(i) = 1;
  end
end
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%
%%%                    User input the required data – info.m
%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

M   = input('Input the number of machine(s) > ');
C   = input('Input the number of Component Type(s) > ');

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%
%%%             Read the Setup Time for different machines
%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

a = 'Please enter the setup time for the Machine';
for i = 1:M
  I = num2str(i);
  P=[a, I, ' > '];
  Setup(i) = input(P);
end

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%
%%%             Read the quantities for different component types
%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

a = 'Please enter the quantity for the Component';
for j = 1:C
  J = num2str(j);
  P=[a, J, ' > '];
  Q(j) = input(P);
end

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%
%%%          Read the Placement Time for different machines to components
%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

a = 'Please enter the placement time PT';
for i = 1:M
  for j = 1:C
    I = num2str(i);
    J = num2str(j);
     P=[a, I, J, ' > '];
    PT(i,j) = input(P);
  end
end

fil = input('Save the data into file, y/n? ', 's');

if (fil == 'Y' | fil == 'y')
  [filename, path] = uiputfile('*.mat','Save As');
     eval('save', [path filename]);
end
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%
%%%                    Start the Program - Start.m
%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

global A;
global B;
global counter;
global ixsep_guess;

templat3;

A=a(1:m1,1:(m1*m2));
B=-b(1:m1);
Aeq=a((m1+1):(m1+m2),1:(m1*m2));
Beq=b((m1+1):(m1+m2));

simplex;

guess_s_t=round(sortrows([bas' b],1));
[n_r,n_c]=size(guess_s_t);
guess_s=guess_s_t(1:(n_r-1),1:n_c);

x0=zeros(m1*m2,1);
for q=1:(n_r-1),
  x0(guess_s(q,1))=guess_s(q,2);
end

xstatus=ones(m1*m2,1);

xstatus(4)=2;
xstatus(5)=2;
xstatus(6)=2;

xlb=zeros(m1*m2,1);                      %
for p=1:m1,                              %  Initial lower and upper bound vectors
  xub((m2*(p-1)+1):(m2*p),1)=Beq;        %
end                                      %

OPTIONS =
optimset('TolPCG',0.1,'TolCon',0.05,'TolX',0.05,'TolFun',0.1,'DiffMaxChange',0.1,'DiffMinChange',0.0
001,'MaxFunEvals','10*numberOfVariables','Display','final','MaxIter',400);

counter=1;           % If counter=0, set the first ixsep with the value of ixsep_guess.
ixsep_guess=18;      % If counter=1, let the BNB20 program selects the first ixsep by itself.

deviation_bas=5;
deviation_nbas=1;

for u=1:(m1*m2),                  %
  deviation=deviation_nbas;       %
  for o=1:(n_r-1),                %
    if (u==guess_s(o,1))          %
      deviation=deviation_bas;    %
    end                           %
  end                             %
  if (x0(u)-deviation) > xlb(u)       %
    xlb(u)=x0(u)-deviation;           %
```

```
  end                                  %Modified lower and upper bound vectors
  if (x0(u)+deviation) < xub(u)        %
    xub(u)=x0(u)+deviation;            %
  end                                  %
end                                    %

Aie=[];
Bie=[];

[errmsg,Z,X,t,c,fail]=BNB20('BnB',x0,xstatus,xlb,xub,Aie,Bie,Aeq,Beq,[],[],OPTIONS);


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%
%%%%                    Objective Function – BnB.m
%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%


function [Tmax] = BnB(x);
global A;
global B;

    T=A*x+B;
    Tmax=max(T);
end
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%
%%%                    Simplex Method – simplex.m
%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%


t1 = cputime;

nbas=[];                          % initialize to avoid compiler definition error
eps4=.00001;                      % accuracy parameter
eps0=10^(-10);                    % numerical zero
eps1=10^(-5);                     % accuracy parameter for optimality check

z=0;                              % initial objective value
[m,n1]=size(a);                   % number of rows and columns of a

if m1 > 0,                        % inequality constraints exist
    a=[a [eye(m1) zeros(m1,m2)]'];
    c=[c zeros(1,m1)];
end

if m2==0,                         % if no equality constraints
    disp(['start phase2 '])
    reg;
    return
else
    corig=c;                      %index for the objective function
    c=[zeros(1,n1+m1) ones(1,m2)];
    a=[a [zeros(m2,m1) eye(m2)]'];
    bas=[n1+1:n1+m1+m2];
    reg;                          % solve phase 1 using the reg.m file
    if z < -eps4,
        disp(['optimal value from phase 1 is: ' num2str(z) ])
        disp(['the above shows that the problem is infeasible'])

  disp(['Final tableau'])
      [a b
      c z]
  return
   else

        a=a(:,1:n1+m1);
        c=c(1:n1+m1);


        while ~all(bas<n1+m1+1),
        disp(['an artificial variable remains in the basis after phase1'])
        disp(['pivot to remove the remaining artificial variables'])
            mto1=[1:m];
            i=mto1(bas>n1+m1);          % pivot row
            i=i(1)
            n1m1to1=[1:n1+m1];
            t=n1m1to1(abs(a(i,:))>eps1);   % pivot column
            t=t(1)
            v=nbas(nbas==t);            % variable entering the basis
            nbas(v)=bas(i);
            bas(i)=t;
```

```
            alpha=a(i,t);                    % pivot element
% Store the data in ap,bp
            ap=a;
            bp=b;
            for k=1:m,
                ratio=ap(k,t)/ap(i,t);
                a(k,:)=ap(k,:)-ap(i,:)*ratio;
                b(k)=bp(k)-bp(i)*ratio;
            end
% Now for the objective row update
            ratio=c(t)/ap(i,t);
            c=c-ap(i,:)*ratio;
            z=z-bp(i)*ratio;
            a(i,:)=ap(i,:)/ap(i,t);
            b(i)=bp(i)/ap(i,t);
          end

            c=corig;
            reg                              % solve the problem using file reg.m
      end

 e = cputime - t1

end
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%
%%%                    Phase II for Simplex Method - reg.m
%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%


rnderr=0;
iterm=500;
stop=1; % use to overcome the bug in the return statement
eps0=10^(-10); % numerical zero
eps1=10^(-5);  % accuracy parameter for optimality check
eps2=10^(-8);  % accuracy parameter pivot element (threshold test)
eps3=10^(-6);  % accuracy parameter for final roundoff error check
a0=a;          % save the matrix a for the final roundoff error test
b0=b;          % save the vector b for the final roundoff error test
c0=c;bas0=bas;
[m,mn]=size(a); % row and column size of a
z=-c(bas)*b;   % initial value for z

% price out the cost vector
z= - c(bas)*b;
%clc
for i=1:m,
    c = c - c(bas(i))*a(i,:);
end

iter=0;                  % initialize the iteration count
n=mn-m;                  % number of nonbasic variables
% nbas - indices of the nonbasic variables
nbas=[];
for j=1:mn,
    if all(j~=bas),
          nbas=[nbas j];
    end
end
% Perform simplex iterations as long as there is a neg cost
while iter<iterm,
% Find a negative reduced cost.
    ctemp=c;      % temporary work vector
    neg=[];
    for j=1:n,
          if ctemp(nbas(j))<-eps1,
               neg=[neg nbas(j)];
          end
    end
    ct=-1;
  if length(neg)==0,
                 disp(['This phase is completed  - current basis is: '])
                  bas=bas
                 disp(['The current basic variable values are : '])
                 b
                 disp(['The current objective value is:'])
                 T = c0(bas)*b
                 disp(['The number of iterations is ' int2str(iter) ])
                  if norm(a0(:,bas)*b-b0,inf)>eps3, % check solution
                  disp(['**WARNING** roundoff error is significant'])
                  end
                  if any(b<-eps0), % check positive final solution
```

```
                disp(['**WARNING** final b not nonnegative'])
            end
            stop=0;
            return
    else
        while ct<-eps1,  % continue till we find a suitable pivot
         [ct,i]=min(ctemp(neg));
         if ct>=-eps1,  % no suitable pivot columns are left
             disp(['a suitable pivot element cannot be found'])
             disp(['probable cause: roundoff error or ill-cond prob'])
             disp(['equilibrate problem before solving'])
             stop=0;
             return
         end
         t=neg(i);     % index of the most neg reduced cost
% Now, let x sub t enter the basis
%
% First, we need to find the variable which leaves the basis
         pos=[];
         ind=[];
         for i=1:m,
             if a(i,t)>eps0,
                 ind=[ind i]; % suitable rows
             end
         end
         if length(ind)==0,
             disp(['The problem is unbounded '])
             stop=0;
             return
         end
         [alpha,i]=min(b(ind)./a(ind,t));
         i=ind(i);     % pivot row
         if a(i,t)>eps2, % a suitable pivot element is found
             ct=0;
         else
             ctemp(t)=0;    % column t is unsuitable pivot col.
         end
        end
        if stop==0,
         return  % Ensure that we return
        end
% Update the basic and nonbasic vectors.
         nbas(nbas==t)=bas(i);
         bas(i)=t;
         alpha=a(i,t);  % pivot element
% Store the data in ap,bp
         ap=a;
         bp=b;
% Now pivot by row
        iter=iter+1;
         for k=1:m,
            ratio=ap(k,t)/ap(i,t);
            a(k,:)=ap(k,:)-ap(i,:)*ratio;
            b(k)=bp(k)-bp(i)*ratio;
         end
% Now for the objective row update
         ratio=c(t)/ap(i,t);
         c=c-ap(i,:)*ratio;
         z=z-bp(i)*ratio;
```

```
            a(i,:)=ap(i,:)/ap(i,t);
            b(i)=bp(i)/ap(i,t);
    end
%
end
if iter>=300,
      text='Iteration bound has been exceeded ****** '
end
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%
%%%                    Branch-and-Bound Algorithm - BNB20.m
%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

function [errmsg,Z,X,t,c,fail] = BNB20(fun,x0,xstat,xl,xu,a,b,aeq,beq,nonlc,setts,opts,varargin);

global maxSQPiter;
global counter;
global ixsep_guess;

% STEP 0 CHECKING INPUT
Z=[]; X=[]; t=0; c=0; fail=0;
if nargin<2, errmsg='BNB needs at least 2 input arguments.'; return; end;
if isempty(fun), errmsg='No fun found.'; return;
elseif ~ischar(fun), errmsg='fun must be a string.'; return; end;
if isempty(x0), errmsg='No x0 found.'; return;
elseif ~isnumeric(x0) | ~isreal(x0) | size(x0,2)>1
  errmsg='x0 must be a real column vector.'; return;
end;
xstatus=zeros(size(x0));
if nargin>2 & ~isempty(xstat)
  if isnumeric(xstat) & isreal(xstat) & all(size(xstat)<=size(x0))
    if all(xstat==round(xstat) & 0<=xstat & xstat<=2)
      xstatus(1:size(xstat))=xstat;
    else errmsg='xstatus must consist of the integers 0,1 en 2.'; return; end;
  else errmsg='xstatus must be a real column vector the same size as x0.'; return; end;
end;
xlb=zeros(size(x0));
xlb(find(xstatus==0))=-inf;
if nargin>3 & ~isempty(xl)
  if isnumeric(xl) & isreal(xl) & all(size(xl)<=size(x0))
    xlb(1:size(xl,1))=xl;
  else errmsg='xlb must be a real column vector the same size as x0.'; return; end;
end;
if any(x0<xlb), errmsg='x0 must be in the range xlb <= x0.'; return;
elseif any(xstatus==1 & (~isfinite(xlb) | xlb~=round(xlb)))
  errmsg='xlb(i) must be an integer if x(i) is an integer variabele.'; return;
end;
xlb(find(xstatus==2))=x0(find(xstatus==2));
xub=ones(size(x0));
xub(find(xstatus==0))=inf;
if nargin>4 & ~isempty(xu)
  if isnumeric(xu) & isreal(xu) & all(size(xu)<=size(x0))
    xub(1:size(xu,1))=xu;
  else errmsg='xub must be a real column vector the same size as x0.'; return; end;
end;
if any(x0>xub), errmsg='x0 must be in the range x0 <=xub.'; return;
elseif any(xstatus==1 & (~isfinite(xub) | xub~=round(xub)))
  errmsg='xub(i) must be an integer if x(i) is an integer variabale.'; return;
end;
xub(find(xstatus==2))=x0(find(xstatus==2));
A=[];
if nargin>5 & ~isempty(a)
  if isnumeric(a) & isreal(a) & size(a,2)==size(x0,1), A=a;
  else errmsg='Matrix A not correct.'; return; end;
end;
B=[];
```

```
if nargin>6 & ~isempty(b)
  if isnumeric(b) & isreal(b) & all(size(b)==[size(A,1) 1]), B=b;
  else errmsg='Column vector B not correct.'; return; end;
end;
if isempty(B) & ~isempty(A), B=zeros(size(A,1),1); end;
Aeq=[];
if nargin>7 & ~isempty(aeq)
  if isnumeric(aeq) & isreal(aeq) & size(aeq,2)==size(x0,1), Aeq=aeq;
  else errmsg='Matrix Aeq not correct.'; return; end;
end;
Beq=[];
if nargin>8 & ~isempty(beq)
  if isnumeric(beq) & isreal(beq) & all(size(beq)==[size(Aeq,1) 1]), Beq=beq;
  else errmsg='Column vector Beq not correct.'; return; end;
end;
if isempty(Beq) & ~isempty(Aeq), Beq=zeros(size(Aeq,1),1); end;
nonlcon='';
if nargin>9 & ~isempty(nonlc)
  if ischar(nonlc), nonlcon=nonlc;
  else errmsg='fun must be a string.'; return; end;
end;
settings = [0 0];
if nargin>10 & ~isempty(setts)
  if isnumeric(setts) & isreal(setts) & all(size(setts)<=size(settings))
    settings(setts~=0)=setts(setts~=0);
  else errmsg='settings should be a row vector of length 1 or 2.'; return; end;
end;
maxSQPiter=1000;
options=optimset('fmincon');
if nargin>11 & ~isempty(opts)
  if isstruct(opts)
    if isfield(opts,'MaxSQPIter')
      if isnumeric(opts.MaxSQPIter) & isreal(opts.MaxSQPIter) & ...
          all(size(opts.MaxSQPIter)==1) & opts.MaxSQPIter>0 & ...
          round(opts.MaxSQPIter)==opts.MaxSQPIter
        maxSQPiter=opts.MaxSQPIter;
        opts=rmfield(opts,'MaxSQPIter');
      else errmsg='options.maxSQPiter must be an integer >0.'; return; end;
    end;
    options=optimset(options,opts);
  else errmsg='options must be a structure.'; return; end;
end;
evalreturn=0;
eval(['z=',fun,'(x0,varargin{:});'],'errmsg="fun caused error."; evalreturn=1;');
if evalreturn==1, return; end;
if ~isempty(nonlcon)
  eval(['[C, Ceq]=',nonlcon,'(x0,varargin{:});'],'errmsg="nonlcon caused error."; evalreturn=1;');
  if evalreturn==1, return; end;
  if size(C,2)>1 | size(Ceq,2)>1, errmsg='C en Ceq must be column vectors.'; return; end;
end;

% STEP 1 INITIALISATION
currentwarningstate=warning;
warning off;
tic;
lx = size(x0,1);
z_incumbent=inf;
x_incumbent=inf*ones(size(x0));
l = ceil(sum(log2(xub(find(xstatus==1))-xlb(find(xstatus==1))+1))+size(find(xstatus==1),1)+1);
```

```
stackx0=zeros(lx,I);
stackx0(:,1)=x0;
stackxlb=zeros(lx,I);
stackxlb(:,1)=xlb;
stackxub=zeros(lx,I);
stackxub(:,1)=xub;
stackdepth=zeros(1,I);
stackdepth(1,1)=1;
stacksize=1;
xchoice=zeros(size(x0));
if ~isempty(Aeq)
  j=0;
  for i=1:size(Aeq,1)
    if Beq(i)==1 & all(Aeq(i,:)==0 | Aeq(i,:)==1)
      J=find(Aeq(i,:)==1);
      if all(xstatus(J)~=0 & xchoice(J)==0 & xlb(J)==0 & xub(J)==1)
        if all(xstatus(J)~=2) | all(x0(J(find(xstatus(J)==2)))==0)
          j=j+1;
          xchoice(J)=j;
          if sum(x0(J))==0, errmsg='x0 not correct.'; return; end;
        end;
      end;
    end;
  end;
end;
errx=optimget(options,'TolX');
handleupdate=[];
if ishandle(settings(2))
  taghandlemain=get(settings(2),'Tag');
  if strcmp(taghandlemain,'main BNB GUI')
    handleupdate=guiupd;
    handleupdatemsg=findobj(handleupdate,'Tag','updatemessage');
    bnbguicb('hide main');
    drawnow;
  end;
end;
optionsdisplay=getfield(options,'Display');
if strcmp(optionsdisplay,'iter') | strcmp(optionsdisplay,'final')
  show=1;
else show=0; end;

% STEP 2 TERMINIATION
while stacksize>0
  c=c+1;

  % STEP 3 LOADING OF CSP
  x0=stackx0(:,stacksize);
  xlb=stackxlb(:,stacksize);
  xub=stackxub(:,stacksize);
  x0(find(x0<xlb))=xlb(find(x0<xlb));
  x0(find(x0>xub))=xub(find(x0>xub));
  depth=stackdepth(1,stacksize);
  if z_incumbent==inf
    stacksize=stacksize-1;
  else
    random_n=rand(1);
    if random_n<=0.85
      stacksize_step=1;
    end
```

```
    if (random_n>0.85 & random_n<=0.925)
      stacksize_step=2;
    end
    if (random_n>0.925 & random_n<=0.975)
      stacksize_step=3;
    end
    if random_n>0.975
      stacksize_step=4;
    end
    stacksize=stacksize-stacksize_step;
    if stacksize<0          %
      stacksize=0;          %If stacksize<0, set stacksize=0 (stacksize can only be positive or zero)
    end                     %
  end
  percdone=round(100*(1-sum(0.5.^(stackdepth(1:(stacksize+1))-1))));

  % UPDATE FOR USER
  if ishandle(handleupdate) & strcmp(get(handleupdate,'Tag'),'update BNB GUI')
    t=toc;
    updatemsg={ ...
        sprintf('searched %3d %% of three',percdone) ...
        sprintf('Z    : %12.4e',z_incumbent) ...
        sprintf('t    : %12.1f secs',t) ...
        sprintf('c    : %12d cycles',c-1) ...
        sprintf('fail : %12d cycles',fail)};
    set(handleupdatemsg,'String',updatemsg);
    drawnow;
  else
    t=toc;
    disp(sprintf('*** searched %3d %% of three',percdone));
    disp(sprintf('*** Z    : %12.4e',z_incumbent));
    disp(sprintf('*** t    : %12.1f secs',t));
    disp(sprintf('*** c    : %12d cycles',c-1));
    disp(sprintf('*** fail : %12d cycles',fail));
  end;

  % STEP 4 RELAXATION
  [x z convflag]=fmincon(fun,x0,A,B,Aeq,Beq,xlb,xub,nonlcon,options,varargin{:});

  % STEP 5 FATHOMING
  K = find(xstatus==1 & xlb~=xub);
  separation=1;
  if convflag<0 | (convflag==0 & settings(1))
    % FC 1
    separation=0;
    if show, disp('*** branch pruned'); end;
    if convflag==0,
      fail=fail+1;
      if show, disp('*** not convergent'); end;
    elseif show, disp('*** not feasible');
    end;
  elseif z>=z_incumbent & convflag>0
    % FC 2
    separation=0;
    if show
      disp('*** branch pruned');
      disp('*** ghosted');
    end;
```

```
elseif all(abs(round(x(K))-x(K))<errx) & convflag>0
  % FC 3
  z_incumbent = z;
  x_incumbent = x;
  separation = 0;
  if show
    disp('*** branch pruned');
    disp('*** new best solution found');
  end;
end;

% STEP 6 SELECTION
if separation == 1 & ~isempty(K)
  dzsep=-1;
  for i=1:size(K,1)
    dxsepc = abs(round(x(K(i)))-x(K(i)));
    if dxsepc>=errx | convflag==0
      xsepc = x; xsepc(K(i))=round(x(K(i)));
      dzsepc = abs(feval(fun,xsepc,varargin{:})-z);
      if dzsepc>dzsep
        dzsep=dzsepc;
        ixsep=K(i);
      end;
    end;
  end;

  if counter==0
    ixsep=ixsep_guess
       counter=counter+1;
  end

  % STEP 7 SEPARATION
  if xchoice(ixsep)==0

    % XCHOICE==0
    branch=1;
    domain=[xlb(ixsep) xub(ixsep)];
    sepdepth=depth;
    while branch==1
      xboundary=(domain(1)+domain(2))/2;
      if x(ixsep)<xboundary
        domainA=[domain(1) floor(xboundary)];
        domainB=[floor(xboundary+1) domain(2)];
      else
        domainA=[floor(xboundary+1) domain(2)];
        domainB=[domain(1) floor(xboundary)];
      end;
      sepdepth=sepdepth+1;
      stacksize=stacksize+1;
      stackx0(:,stacksize)=x;
      stackxlb(:,stacksize)=xlb;
      stackxlb(ixsep,stacksize)=domainB(1);
      stackxub(:,stacksize)=xub;
      stackxub(ixsep,stacksize)=domainB(2);
      stackdepth(1,stacksize)=sepdepth;
      if domainA(1)==domainA(2)
        stacksize=stacksize+1;
        stackx0(:,stacksize)=x;
        stackxlb(:,stacksize)=xlb;
```

```
            stackxlb(ixsep,stacksize)=domainA(1);
            stackxub(:,stacksize)=xub;
            stackxub(ixsep,stacksize)=domainA(2);
            stackdepth(1,stacksize)=sepdepth;
            branch=0;
          else
            domain=domainA;
            branch=1;
          end;
        end;
      else

        % XCHOICE~=0
        L=find(xchoice==xchoice(ixsep));
        M=intersect(K,L);
        [dummy,N]=sort(x(M));
        part1=M(N(1:floor(size(N)/2))); part2=M(N(floor(size(N)/2)+1:size(N)));
        sepdepth=depth+1;
        stacksize=stacksize+1;
        stackx0(:,stacksize)=x;
        O = (1-sum(stackx0(part1,stacksize)))/size(part1,1);
        stackx0(part1,stacksize)=stackx0(part1,stacksize)+O;
        stackxlb(:,stacksize)=xlb;
        stackxub(:,stacksize)=xub;
        stackxub(part2,stacksize)=0;
        stackdepth(1,stacksize)=sepdepth;
        stacksize=stacksize+1;
        stackx0(:,stacksize)=x;
        O = (1-sum(stackx0(part2,stacksize)))/size(part2,1);
        stackx0(part2,stacksize)=stackx0(part2,stacksize)+O;
        stackxlb(:,stacksize)=xlb;
        stackxub(:,stacksize)=xub;
        stackxub(part1,stacksize)=0;
        stackdepth(1,stacksize)=sepdepth;
      end;
    elseif separation==1 & isempty(K)
      fail=fail+1;
      if show
        disp('*** branch pruned');
        disp('*** leaf not convergent');
      end;
    end;
  end;

% STEP 8 OUTPUT
t=toc;
Z = z_incumbent;
X = x_incumbent;
errmsg='';

if ishandle(handleupdate)
  taghandleupdate=get(handleupdate,'Tag');
  if strcmp(taghandleupdate,'update BNB GUI')
    close(handleupdate);
  end;
end;

eval(['warning ',currentwarningstate]);
```

APPENDIX VI

SOURCE CODES FOR THE TABU SEARCH HEURISTIC IN MATLAB

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%
%%%                    User Interface Input Data
%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

disp('  Select the Input Mode ');
disp('    INTERACTIVE MODE ----------- 1 ');
disp('    FILE MODE FROM DATA FILE ---- 2 ');

  CH =input('Enter 1 or 2 ===> ');

while (CH ~= 1 & CH ~= 2)== 1
 CH=input('Please Re-enter the choice of input mode - 1 or 2: ');
  if CH==1 | CH==2
    break;
  end
end

if CH==1
      InputData;
  else
       [filename,path] = uigetfile('*.mat', 'Get File');
       eval(['load' [path filename]]);
end

disp('Time is >')
disp(PT)
disp('Quantity is >')
disp(Q)
disp('Setup time is >')
disp(Setup)


m1=M;
m2=C;

a = zeros(m1+m2, m1*m2+1);
for row = 1:m1+m2
  for col = 1:(m1 * m2) + 1
    if row <= m1
      if col == 1
        a(row, col) = -1;
      else
        row_m1 = floor((col-2)/C +1);
        col_m1 = (mod(col-2, C)) +1;
        a(row_m1, col) = PT(row_m1, col_m1);
      end
    else
      for col = 2 : (m1*m2) + 1
        if (mod(col-row+m1-1, m2) == 0)
          a(row, col) = 1;
        end
      end
    end
  end
end

b = zeros(m1+m2, 1);
```

```
for i = 1:m1+m2
  if i <= m1
    b(i) = -Setup(i);
  else
    b(i) = Q(i-m1);
  end
end

c = zeros(1, m1*m2+1);
for i = 1:m1*m2+1;
     if i== m1*m2+1
     c(i) = 1;
  end
end
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%
%%%                    User input the required data - info.m
%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

M   = input('Input the number of machine(s) > ');
C   = input('Input the number of Component Type(s) > ');

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%
%%%             Read the Setup Time for different machines
%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

a = 'Please enter the setup time for the Machine';
for i = 1:M
  I = num2str(i);
  P=[a, I, ' > '];
  Setup(i) = input(P);
end

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%
%%%             Read the quantities for different component types
%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

a = 'Please enter the quantity for the Component';
for j = 1:C
  J = num2str(j);
  P=[a, J, ' > '];
  Q(j) = input(P);
end

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%
%%%          Read the Placement Time for different machines to components
%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

a = 'Please enter the placement time PT';
for i = 1:M
  for j = 1:C
    I = num2str(i);
    J = num2str(j);
     P=[a, I, J, ' > '];
    PT(i,j) = input(P);
  end
end

fil = input('Save the data into file, y/n? ', 's');

if (fil == 'Y' | fil == 'y')
  [filename, path] = uiputfile('*.mat','Save As');
     eval('save', [path filename]);
end
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%
%%%                    Start the Program - Start.m
%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

global A;
global B;
global counter;
global ixsep_guess;

templat3;

A=a(1:m1,1:(m1*m2));
B=-b(1:m1);
Aeq=a((m1+1):(m1+m2),1:(m1*m2));
Beq=b((m1+1):(m1+m2));

simplex;

guess_s_t=round(sortrows([bas' b],1));
[n_r,n_c]=size(guess_s_t);
guess_s=guess_s_t(1:(n_r-1),1:n_c);

x0=zeros(m1*m2,1);
for q=1:(n_r-1),
  x0(guess_s(q,1))=guess_s(q,2);
end

xstatus=ones(m1*m2,1);

xstatus(4)=2;
xstatus(5)=2;
xstatus(6)=2;

xlb=zeros(m1*m2,1);                    %
for p=1:m1,                            %  Initial lower and upper bound vectors
  xub((m2*(p-1)+1):(m2*p),1)=Beq;      %
end                                    %

OPTIONS =
optimset('TolPCG',0.1,'TolCon',0.05,'TolX',0.05,'TolFun',0.1,'DiffMaxChange',0.1,'DiffMinChange',0.0
001,'MaxFunEvals','10*numberOfVariables','Display','final','MaxIter',400);

counter=1;          % If counter=0, set the first ixsep with the value of ixsep_guess.
ixsep_guess=18;     % If counter=1, let the BNB20 program selects the first ixsep by itself.

deviation_bas=5;
deviation_nbas=1;

for u=1:(m1*m2),                  %
  deviation=deviation_nbas;       %
  for o=1:(n_r-1),                %
    if (u==guess_s(o,1))          %
      deviation=deviation_bas;    %
    end                           %
  end                             %
  if (x0(u)-deviation) > xlb(u)       %
    xlb(u)=x0(u)-deviation;           %
```

```
  end                                  %Modified lower and upper bound vectors
  if (x0(u)+deviation) < xub(u)        %
    xub(u)=x0(u)+deviation;            %
  end                                  %
end                                    %

Aie=[];
Bie=[];

[errmsg,Z,X,t,c,fail]=BNB20('BnB',x0,xstatus,xlb,xub,Aie,Bie,Aeq,Beq,[],[],OPTIONS);


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%
%%%                  Objective Function - BnB.m
%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%


function [Tmax] = BnB(x);
global A;
global B;

    T=A*x+B;
    Tmax=max(T);
end
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%
%%%                    Simplex Method - simplex.m
%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%


t1 = cputime;

nbas=[];                         % initialize to avoid compiler definition error
eps4=.00001;                     % accuracy parameter
eps0=10^(-10);                   % numerical zero
eps1=10^(-5);                    % accuracy parameter for optimality check

z=0;                             % initial objective value
[m,n1]=size(a);                  % number of rows and columns of a

if m1 > 0,                       % inequality constraints exist
    a=[a [eye(m1) zeros(m1,m2)]'];
    c=[c zeros(1,m1)];
end

if m2==0,                        % if no equality constraints
    disp(['start phase2 '])
    reg;
    return
else
    corig=c;                     %index for the objective function
    c=[zeros(1,n1+m1) ones(1,m2)];
    a=[a [zeros(m2,m1) eye(m2)]'];
    bas=[n1+1:n1+m1+m2];
    reg;                         % solve phase 1 using the reg.m file
    if z < -eps4,
        disp(['optimal value from phase 1 is: ' num2str(z) ])
        disp(['the above shows that the problem is infeasible'])

  disp(['Final tableau'])
      [a b
      c z]
  return
  else

      a=a(:,1:n1+m1);
      c=c(1:n1+m1);


      while ~all(bas<n1+m1+1),
      disp(['an artificial variable remains in the basis after phase1'])
      disp(['pivot to remove the remaining artificial variables'])
          mto1=[1:m];
          i=mto1(bas>n1+m1);          % pivot row
          i=i(1)
          n1m1to1=[1:n1+m1];
          t=n1m1to1(abs(a(i,:))>eps1);   % pivot column
          t=t(1)
          v=nbas(nbas==t);            % variable entering the basis
          nbas(v)=bas(i);
          bas(i)=t;
```

```
            alpha=a(i,t);                    % pivot element
% Store the data in ap,bp
            ap=a;
            bp=b;
            for k=1:m,
                ratio=ap(k,t)/ap(i,t);
                a(k,:)=ap(k,:)-ap(i,:)*ratio;
                b(k)=bp(k)-bp(i)*ratio;
            end
% Now for the objective row update
            ratio=c(t)/ap(i,t);
            c=c-ap(i,:)*ratio;
            z=z-bp(i)*ratio;
            a(i,:)=ap(i,:)/ap(i,t);
            b(i)=bp(i)/ap(i,t);
          end

            c=corig;
            reg                              % solve the problem using file reg.m
      end

 e = cputime - t1

end
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%
%%%                    Phase II for Simplex Method - reg.m
%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%


rnderr=0;
iterm=500;
stop=1; % use to overcome the bug in the return statement
eps0=10^(-10); % numerical zero
eps1=10^(-5); % accuracy parameter for optimality check
eps2=10^(-8); % accuracy parameter pivot element (threshold test)
eps3=10^(-6); % accuracy parameter for final roundoff error check
a0=a;        % save the matrix a for the final roundoff error test
b0=b;        % save the vector b for the final roundoff error test
c0=c;bas0=bas;
[m,mn]=size(a); % row and column size of a
z=-c(bas)*b; % initial value for z

% price out the cost vector
z= - c(bas)*b;
%clc
for i=1:m,
    c = c - c(bas(i))*a(i,:);
end

iter=0;                 % initialize the iteration count
n=mn-m;                 % number of nonbasic variables
% nbas - indices of the nonbasic variables
nbas=[];
for j=1:mn,
    if all(j~=bas),
        nbas=[nbas j];
    end
end
% Perform simplex iterations as long as there is a neg cost
while iter<iterm,
% Find a negative reduced cost.
    ctemp=c;      % temporary work vector
    neg=[];
    for j=1:n,
        if ctemp(nbas(j))<-eps1,
            neg=[neg nbas(j)];
        end
    end
    ct=-1;
  if length(neg)==0,
            disp(['This phase is completed  - current basis is: '])
            bas=bas
            disp(['The current basic variable values are : '])
            b
            disp(['The current objective value is:'])
            T = c0(bas)*b
            disp(['The number of iterations is ' int2str(iter) ])
            if norm(a0(:,bas)*b-b0,inf)>eps3, % check solution
            disp(['**WARNING** roundoff error is significant'])
            end
            if any(b<-eps0), % check positive final solution
```

```
                disp(['**WARNING** final b not nonnegative'])
            end
            stop=0;
            return
    else
        while ct<-eps1,  % continue till we find a suitable pivot
         [ct,i]=min(ctemp(neg));
         if ct>=-eps1,  % no suitable pivot columns are left
            disp(['a suitable pivot element cannot be found'])
            disp(['probable cause: roundoff error or ill-cond prob'])
            disp(['equilibrate problem before solving'])
            stop=0;
            return
         end
         t=neg(i);      % index of the most neg reduced cost
% Now, let x sub t enter the basis
%
% First, we need to find the variable which leaves the basis
         pos=[];
         ind=[];
         for i=1:m,
            if a(i,t)>eps0,
                ind=[ind i]; % suitable rows
            end
         end
         if length(ind)==0,
            disp(['The problem is unbounded '])
            stop=0;
            return
         end
         [alpha,i]=min(b(ind)./a(ind,t));
         i=ind(i);      % pivot row
         if a(i,t)>eps2, % a suitable pivot element is found
            ct=0;
         else
            ctemp(t)=0;    % column t is unsuitable pivot col.
         end
        end
        if stop==0,
         return  % Ensure that we return
        end
% Update the basic and nonbasic vectors.
         nbas(nbas==t)=bas(i);
         bas(i)=t;
         alpha=a(i,t);  % pivot element
% Store the data in ap,bp
         ap=a;
         bp=b;
% Now pivot by row
        iter=iter+1;
         for k=1:m,
            ratio=ap(k,t)/ap(i,t);
            a(k,:)=ap(k,:)-ap(i,:)*ratio;
            b(k)=bp(k)-bp(i)*ratio;
         end
% Now for the objective row update
         ratio=c(t)/ap(i,t);
         c=c-ap(i,:)*ratio;
         z=z-bp(i)*ratio;
```

```
            a(i,:)=ap(i,:)/ap(i,t);
            b(i)=bp(i)/ap(i,t);
    end
%
end
if iter>=300,
      text='Iteration bound has been exceeded ******'
end
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%
%%%                    Branch-and-Bound Algorithm - BNB20.m
%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

function [errmsg,Z,X,t,c,fail] = BNB20(fun,x0,xstat,xl,xu,a,b,aeq,beq,nonlc,setts,opts,varargin);
global maxSQPiter;
global counter_wan;
global ixsep_guess;

% STEP 0 CHECKING INPUT
Z=[]; X=[]; t=0; c=0; fail=0;
if nargin<2, errmsg='BNB needs at least 2 input arguments.'; return; end;
if isempty(fun), errmsg='No fun found.'; return;
elseif ~ischar(fun), errmsg='fun must be a string.'; return; end;
if isempty(x0), errmsg='No x0 found.'; return;
elseif ~isnumeric(x0) | ~isreal(x0) | size(x0,2)>1
  errmsg='x0 must be a real column vector.'; return;
end;
xstatus=zeros(size(x0));
if nargin>2 & ~isempty(xstat)
  if isnumeric(xstat) & isreal(xstat) & all(size(xstat)<=size(x0))
    if all(xstat==round(xstat) & 0<=xstat & xstat<=2)
      xstatus(1:size(xstat))=xstat;
    else errmsg='xstatus must consist of the integers 0,1 en 2.'; return; end;
  else errmsg='xstatus must be a real column vector the same size as x0.'; return; end;
end;
xlb=zeros(size(x0));
xlb(find(xstatus==0))=-inf;
if nargin>3 & ~isempty(xl)
  if isnumeric(xl) & isreal(xl) & all(size(xl)<=size(x0))
    xlb(1:size(xl,1))=xl;
  else errmsg='xlb must be a real column vector the same size as x0.'; return; end;
end;
if any(x0<xlb), errmsg='x0 must be in the range xlb <= x0.'; return;
elseif any(xstatus==1 & (~isfinite(xlb) | xlb~=round(xlb)))
  errmsg='xlb(i) must be an integer if x(i) is an integer variabele.'; return;
end;
xlb(find(xstatus==2))=x0(find(xstatus==2));
xub=ones(size(x0));
xub(find(xstatus==0))=inf;
if nargin>4 & ~isempty(xu)
  if isnumeric(xu) & isreal(xu) & all(size(xu)<=size(x0))
    xub(1:size(xu,1))=xu;
  else errmsg='xub must be a real column vector the same size as x0.'; return; end;
end;
if any(x0>xub), errmsg='x0 must be in the range x0 <=xub.'; return;
elseif any(xstatus==1 & (~isfinite(xub) | xub~=round(xub)))
  errmsg='xub(i) must be an integer if x(i) is an integer variabale.'; return;
end;
xub(find(xstatus==2))=x0(find(xstatus==2));
A=[];
if nargin>5 & ~isempty(a)
  if isnumeric(a) & isreal(a) & size(a,2)==size(x0,1), A=a;
  else errmsg='Matrix A not correct.'; return; end;
end;
B=[];
if nargin>6 & ~isempty(b)
```

```
  if isnumeric(b) & isreal(b) & all(size(b)==[size(A,1) 1]), B=b;
  else errmsg='Column vector B not correct.'; return; end;
end;
if isempty(B) & ~isempty(A), B=zeros(size(A,1),1); end;
Aeq=[];
if nargin>7 & ~isempty(aeq)
  if isnumeric(aeq) & isreal(aeq) & size(aeq,2)==size(x0,1), Aeq=aeq;
  else errmsg='Matrix Aeq not correct.'; return; end;
end;
Beq=[];
if nargin>8 & ~isempty(beq)
  if isnumeric(beq) & isreal(beq) & all(size(beq)==[size(Aeq,1) 1]), Beq=beq;
  else errmsg='Column vector Beq not correct.'; return; end;
end;
if isempty(Beq) & ~isempty(Aeq), Beq=zeros(size(Aeq,1),1); end;
nonlcon='';
if nargin>9 & ~isempty(nonlc)
  if ischar(nonlc), nonlcon=nonlc;
  else errmsg='fun must be a string.'; return; end;
end;
settings = [0 0];
if nargin>10 & ~isempty(setts)
  if isnumeric(setts) & isreal(setts) & all(size(setts)<=size(settings))
    settings(setts~=0)=setts(setts~=0);
  else errmsg='settings should be a row vector of length 1 or 2.'; return; end;
end;
maxSQPiter=1000;
options=optimset('fmincon');
if nargin>11 & ~isempty(opts)
  if isstruct(opts)
    if isfield(opts,'MaxSQPIter')
      if isnumeric(opts.MaxSQPIter) & isreal(opts.MaxSQPIter) & ...
          all(size(opts.MaxSQPIter)==1) & opts.MaxSQPIter>0 & ...
          round(opts.MaxSQPIter)==opts.MaxSQPIter
        maxSQPiter=opts.MaxSQPIter;
        opts=rmfield(opts,'MaxSQPIter');
      else errmsg='options.maxSQPiter must be an integer >0.'; return; end;
    end;
    options=optimset(options,opts);
  else errmsg='options must be a structure.'; return; end;
end;
evalreturn=0;
eval(['z=',fun,'(x0,varargin{:});'],'errmsg="fun caused error."; evalreturn=1;');
if evalreturn==1, return; end;
if ~isempty(nonlcon)
  eval(['[C, Ceq]=',nonlcon,'(x0,varargin{:});'],'errmsg="nonlcon caused error."; evalreturn=1;');
  if evalreturn==1, return; end;
  if size(C,2)>1 | size(Ceq,2)>1, errmsg='C en Ceq must be column vectors.'; return; end;
end;

% STEP 1 INITIALISATION
currentwarningstate=warning;
warning off;
tic;
lx = size(x0,1);
z_incumbent=inf;
x_incumbent=inf*ones(size(x0));
l = ceil(sum(log2(xub(find(xstatus==1))-xlb(find(xstatus==1))+1))+size(find(xstatus==1),1)+1);
stackx0=zeros(lx,l);
```

```
stackx0(:,1)=x0;
stackxlb=zeros(lx,I);
stackxlb(:,1)=xlb;
stackxub=zeros(lx,I);
stackxub(:,1)=xub;
stackdepth=zeros(1,I);
stackdepth(1,1)=1;
stacksize=1;
xchoice=zeros(size(x0));
if ~isempty(Aeq)
  j=0;
  for i=1:size(Aeq,1)
    if Beq(i)==1 & all(Aeq(i,:)==0 | Aeq(i,:)==1)
      J=find(Aeq(i,:)==1);
      if all(xstatus(J)~=0 & xchoice(J)==0 & xlb(J)==0 & xub(J)==1)
        if all(xstatus(J)~=2) | all(x0(J(find(xstatus(J)==2)))==0)
          j=j+1;
          xchoice(J)=j;
          if sum(x0(J))==0, errmsg='x0 not correct.'; return; end;
        end;
      end;
    end;
  end;
end;
errx=optimget(options,'TolX');
handleupdate=[];
if ishandle(settings(2))
  taghandlemain=get(settings(2),'Tag');
  if strcmp(taghandlemain,'main BNB GUI')
    handleupdate=guiupd;
    handleupdatemsg=findobj(handleupdate,'Tag','updatemessage');
    bnbguicb('hide main');
    drawnow;
  end;
end;
optionsdisplay=getfield(options,'Display');
if strcmp(optionsdisplay,'iter') | strcmp(optionsdisplay,'final')
  show=1;
else show=0; end;

% STEP 2 TERMINIATION
while stacksize>0
  c=c+1;

  % STEP 3 LOADING OF CSP
  x0=stackx0(:,stacksize);
  xlb=stackxlb(:,stacksize);
  xub=stackxub(:,stacksize);
  x0(find(x0<xlb))=xlb(find(x0<xlb));
  x0(find(x0>xub))=xub(find(x0>xub));
  depth=stackdepth(1,stacksize);
  if z_incumbent==inf
    stacksize=stacksize-1;
  else
    random_n=rand(1);
    if random_n<=0.875
      stacksize_step=1;
    end
    if (random_n>0.875 & random_n<=0.925)
```

```
    stacksize_step=2;
  end
  if (random_n>0.925 & random_n<=0.975)
    stacksize_step=3;
  end
  if random_n>0.975
    stacksize_step=4;
  end
  stacksize=stacksize-stacksize_step;
  if stacksize<0      %
    stacksize=0;      %If stacksize=0, set stacksize=0 (stacksize can only be positive or zero)
  end                 %
end
percdone=round(100*(1-sum(0.5.^(stackdepth(1:(stacksize+1))-1))));

% UPDATE FOR USER
if ishandle(handleupdate) & strcmp(get(handleupdate,'Tag'),'update BNB GUI')
  t=toc;
  updatemsg={ ...
      sprintf('searched %3d %% of three',percdone) ...
      sprintf('Z   : %12.4e',z_incumbent) ...
      sprintf('t   : %12.1f secs',t) ...
      sprintf('c   : %12d cycles',c-1) ...
      sprintf('fail : %12d cycles',fail)};
  set(handleupdatemsg,'String',updatemsg);
  drawnow;
else
  t=toc;
  disp(sprintf('*** searched %3d %% of three',percdone));
  disp(sprintf('*** Z   : %12.4e',z_incumbent));
  disp(sprintf('*** t   : %12.1f secs',t));
  disp(sprintf('*** c   : %12d cycles',c-1));
  disp(sprintf('*** fail : %12d cycles',fail));
end;

% STEP 4 RELAXATION
[x z convflag]=fmincon(fun,x0,A,B,Aeq,Beq,xlb,xub,nonlcon,options,varargin{:});

% STEP 5 FATHOMING
K = find(xstatus==1 & xlb~=xub);
separation=1;
if convflag<0 | (convflag==0 & settings(1))
  % FC 1
  separation=0;
  if show, disp('*** branch pruned'); end;
  if convflag==0,
    fail=fail+1;
    if show, disp('*** not convergent'); end;
  elseif show, disp('*** not feasible');
  end;
elseif z>=z_incumbent & convflag>0
  % FC 2
  separation=0;
  if show
    disp('*** branch pruned');
    disp('*** ghosted');
  end;

elseif all(abs(round(x(K))-x(K))<errx) & convflag>0
```

```
    % FC 3
    z_incumbent = z;
    x_incumbent = x;
    separation = 0;
    if show
      disp('*** branch pruned');
      disp('*** new best solution found');
    end;
  end;

% STEP 6 SELECTION
if separation == 1 & ~isempty(K)
  dzsep=-1;
  for i=1:size(K,1)
    dxsepc = abs(round(x(K(i)))-x(K(i)));
    if dxsepc>=errx | convflag==0
      xsepc = x; xsepc(K(i))=round(x(K(i)));
      dzsepc = abs(feval(fun,xsepc,varargin{:})-z);
      if dzsepc>dzsep
        dzsep=dzsepc;
        ixsep=K(i);
      end;
    end;
  end;

  if counter_wan==0
    ixsep=ixsep_guess
       counter_wan=counter_wan+1;
  end

  % STEP 7 SEPARATION
  if xchoice(ixsep)==0

    % XCHOICE==0
    branch=1;
    domain=[xlb(ixsep) xub(ixsep)];
    sepdepth=depth;
    while branch==1
      xboundary=(domain(1)+domain(2))/2;
      if x(ixsep)<xboundary
        domainA=[domain(1) floor(xboundary)];
        domainB=[floor(xboundary+1) domain(2)];
      else
        domainA=[floor(xboundary+1) domain(2)];
        domainB=[domain(1) floor(xboundary)];
      end;
      sepdepth=sepdepth+1;
      stacksize=stacksize+1;
      stackx0(:,stacksize)=x;
      stackxlb(:,stacksize)=xlb;
      stackxlb(ixsep,stacksize)=domainB(1);
      stackxub(:,stacksize)=xub;
      stackxub(ixsep,stacksize)=domainB(2);
      stackdepth(1,stacksize)=sepdepth;
      if domainA(1)==domainA(2)
        stacksize=stacksize+1;
        stackx0(:,stacksize)=x;
        stackxlb(:,stacksize)=xlb;
        stackxlb(ixsep,stacksize)=domainA(1);
```

```
            stackxub(:,stacksize)=xub;
            stackxub(ixsep,stacksize)=domainA(2);
            stackdepth(1,stacksize)=sepdepth;
            branch=0;
          else
            domain=domainA;
            branch=1;
          end;
        end;
      else

        % XCHOICE~=0
        L=find(xchoice==xchoice(ixsep));
        M=intersect(K,L);
        [dummy,N]=sort(x(M));
        part1=M(N(1:floor(size(N)/2))); part2=M(N(floor(size(N)/2)+1:size(N)));
        sepdepth=depth+1;
        stacksize=stacksize+1;
        stackx0(:,stacksize)=x;
        O = (1-sum(stackx0(part1,stacksize)))/size(part1,1);
        stackx0(part1,stacksize)=stackx0(part1,stacksize)+O;
        stackxlb(:,stacksize)=xlb;
        stackxub(:,stacksize)=xub;
        stackxub(part2,stacksize)=0;
        stackdepth(1,stacksize)=sepdepth;
        stacksize=stacksize+1;
        stackx0(:,stacksize)=x;
        O = (1-sum(stackx0(part2,stacksize)))/size(part2,1);
        stackx0(part2,stacksize)=stackx0(part2,stacksize)+O;
        stackxlb(:,stacksize)=xlb;
        stackxub(:,stacksize)=xub;
        stackxub(part1,stacksize)=0;
        stackdepth(1,stacksize)=sepdepth;
      end;s
    elseif separation==1 & isempty(K)
      fail=fail+1;
      if show
        disp('*** branch pruned');
        disp('*** leaf not convergent');
      end;
    end;
  end;

% STEP 8 OUTPUT
t=toc;
Z = z_incumbent;
X = x_incumbent;
errmsg='';

if ishandle(handleupdate)
  taghandleupdate=get(handleupdate,'Tag');
  if strcmp(taghandleupdate,'update BNB GUI')
    close(handleupdate);
  end;
end;

eval(['warning ',currentwarningstate]);
```