

**ASSEMBLY LINE BALANCING USING  
GENETIC ALGORITHMS**

**A THESIS  
SUBMITTED TO THE DEPARTMENT OF INDUSTRIAL  
ENGINEERING  
AND THE INSTITUTE OF ENGINEERING AND SCIENCE  
OF BILKENT UNIVERSITY  
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS  
FOR THE DEGREE OF  
MASTER OF SCIENCE**

**By  
Muzaffer Tanyer  
September, 1997**

Thesis  
QA  
402.5  
.T36  
1997

# ASSEMBLY LINE BALANCING USING GENETIC ALGORITHMS

A THESIS  
SUBMITTED TO THE DEPARTMENT OF INDUSTRIAL  
ENGINEERING  
AND THE INSTITUTE OF ENGINEERING AND SCIENCE  
OF BILKENT UNIVERSITY  
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS  
FOR THE DEGREE OF  
MASTER OF SCIENCE

By

~~Muzaffer Tanyer~~

September, 1997

QA  
402.5  
.T36  
1997

B638981

I certify that I have read this thesis and that in my opinion it is fully adequate, in scope and in quality, as a thesis for the degree of Master of Science.



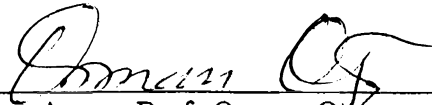
Assoc. Prof. İhsan Sabuncuoğlu (Advisor)

I certify that I have read this thesis and that in my opinion it is fully adequate, in scope and in quality, as a thesis for the degree of Master of Science.



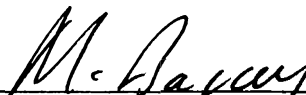
Assoc. Prof. Erdal Erel

I certify that I have read this thesis and that in my opinion it is fully adequate, in scope and in quality, as a thesis for the degree of Master of Science.



Assoc. Prof. Osman Oğuz

Approved for the Institute of Engineering and Sciences:



Prof. Mehmet Baray

Director of Institute of Engineering and Science

# ABSTRACT

## ASSEMBLY LINE BALANCING USING GENETIC ALGORITHMS

Muzaffer Tanyer

M.S. in Industrial Engineering

Supervisor: Assoc. Prof. İhsan Sabuncuoğlu

September, 1997

For the last few decades, the genetic algorithms (GAs) have been used as a kind of heuristic in many areas of manufacturing. Facility layout, scheduling, process planning, and assembly line balancing are some of the areas where GAs are already popular. GAs are more efficient than traditional heuristics and also more flexible as they allow substantial changes in the problem's constraints and in the solution approach with small changes in the program. For this reason, GAs attract the attention of both the researchers and practitioners.

Chromosome structure is one of the key components of a GA. Therefore, in this thesis, we focus on the special structure of the assembly line balancing problem and design a chromosome structure that operates dynamically. We propose a new mechanism to work in parallel with GAs, namely dynamic partitioning. Different from many other GA researchers, we particularly compare different population revision mechanisms and the effect of elitism on these mechanisms. Elitism is revised by the simulated annealing idea and various levels of elitism are created and their effects are observed. The proposed GA is also compared with the traditional heuristics.

*Key words:* Genetic Algorithms, Assembly Line Balancing, Simulated Annealing.

# ÖZET

## GENETİK ALGORİTMALAR İLE HAT DENGELEME

Muzaffer Tanyer

Endüstri Mühendisliği Bölümü Yüksek Lisans

Tez Yöneticisi: Doç. İhsan Sabuncuoğlu

Eylül, 1997

Son yıllarda genetik algoritmalar üretimin pek çok alanında bir çeşit sezgisel yöntem olarak kullanılmaya başlanmıştır. Yerleşim planlama, sıralama, süreç planlama ve hat dengeleme, genetik algoritmaların şimdiden popüler olduğu alanlardandır. Genetik algoritmalar geleneksel sezgisel yöntemlerden daha etkili ve problemin zorlamalarında ve çözüm yaklaşımında yapılacak önemli değişiklikleri programda yapılacak küçük değişikliklerle halledebileceklerinden dolayı da daha esnekler. Bu sebeple, genetik algoritmalar hem araştırmacıların hem de pratisyenlerin ilgisini çekmektedir.

Kromozom yapısı genetik algoritmaların en önemli yapı taşlarından biridir. Bu sebeple, bu tezde hat dengeleme probleminin özel yapısını inceliyoruz ve dinamik olarak değişen bir kromozom yapısı tasarlıyoruz. Dinamik bölmeleme adımı verdiğimiz, genetik algoritmalarla paralel olarak çalışan yeni bir mekanizma öneriyoruz. Diğer birçok genetik algoritma araştırmacısından farklı olarak, özellikle değişik nüfus yenileme mekanizmalarını karşılaştırıyoruz ve seçkinlik kuralının bu mekanizmalar üzerindeki etkisini araştırıyoruz. Seçkinlik kuralı, yumuşatma benzetimi fikri ile yenilenmiş ve çeşitli seçkinlik düzeyleri yaratılıp etkileri gözlenmiştir. Önerilen genetik algoritma geleneksel sezgisel yöntemlerle de karşılaştırılmıştır.

*Anahtar sözcükler:* Genetik Algoritmalar, Hat Dengeleme, Yumuşatma Benzetimi.

## ACKNOWLEDGEMENT

I am indebted to Assoc. Prof. İhsan Sabuncuođlu for his invaluable guidance, encouragement and understanding for bringing this thesis to an end.

I am also indebted to Assoc. Prof. Erdal Erel due to his kind concern, supervision, and suggestions during this study.

I would like to express my gratitude to Assoc. Prof. Osman Ođuz for accepting to read and review this thesis.

I would like to state my most sincere greetings to my dear advisor in Bosphorus University, Assoc. Prof. Kadri Özçaldıran. I must have bewildered him by accomplishing my graduate study successfully.

I would like to express my special thanks to my dearest friends Kemal Kılıç, Özgür Tüfekçi, Alev Kaya. and Savaş Dayanık for their great help and morale support throughout my studies.

And I would like to thank to Bilkent University for having taught me that one should consistently work hard to succeed in life instead of always taking short-cuts.

Finally, I would like to thank to my parents and my brother Alper for their love and support throughout my life.

# Contents

<b>1</b>	<b>INTRODUCTION</b>	<b>1</b>
<b>2</b>	<b>LITERATURE REVIEW</b>	<b>3</b>
2.1	Assembly Line Balancing . . . . .	3
2.2	Genetic Algorithms . . . . .	6
2.2.1	Basic Structure of a GA Process . . . . .	8
2.2.2	Coding . . . . .	8
2.2.3	Fitness Function	9
2.2.4	Reproduction . . . . .	9
2.2.5	Genetic Algorithm Applications . . . . .	11
2.3	ALB and GA . . . . .	11
2.4	Motivation and Organization . . . . .	18
<b>3</b>	<b>THE PROPOSED GENETIC ALGORITHMS</b>	<b>20</b>
3.1	The Characteristics of the Proposed GAs . . . . .	21
3.2	Classical GA vs Modern GA . . . . .	24



3.3	Dynamic Partitioning . . . . .	24
3.4	Elitism With Simulated Annealing	26
<b>4</b>	<b>DYNAMIC PARTITIONING</b>	<b>27</b>
4.1	Motivation . . . . .	27
4.2	Implementation . . . . .	28
4.3	Experimentation	30
4.3.1	ANOVA Results . . . . .	33
4.4	Major Findings . . . . .	38
<b>5</b>	<b>ELITISM WITH SIMULATED ANNEALING</b>	<b>41</b>
5.1	Introduction and Motivation . . . . .	41
5.2	Integration of SA to Elitism . . . . .	42
5.3	Experimentation	45
<b>6</b>	<b>CLASSICAL GA vs MODERN GA</b>	<b>48</b>
6.1	Motivation . . . . .	49
6.2	Experimentation	51
<b>7</b>	<b>COMPARISON WITH TRADITIONAL HEURISTICS</b>	<b>54</b>
7.1	Genetic Algorithms versus Traditional Heuristics . . . . .	54
7.2	Comparison with Leu et al.'s GA (1994) . . . . .	57
7.3	Comparison with Baybars' LBHA-1 (1986) . . . . .	62

<i>CONTENTS</i>	viii
<b>8 CONCLUSION</b>	<b>66</b>
8.1 Contributions . . . . .	66
8.2 Future Research Directions . . . . .	68

# List of Figures

2.1	Classification of Assembly Line Balancing Literature (Taken from Ghosh and Gagnon (1989)) . . . . .	4
2.2	Uniform Crossover Mechanism . . . . .	10
2.3	Crossover Operator of Leu et al. (1994) . . . . .	14
2.4	Stochastic Universal Sampling . . . . .	15
3.1	Coding the Chromosome Representation of an Assembly Line	22
4.1	Illustration of Dynamic Partitioning . . . . .	30
7.1	The Max-Task Time Heuristic Solution to the Kilbridge-Wester 45-Task Problem . . . . .	59
7.2	Modern GA Solution for the Kilbridge-Wester 45-Task Problem	61
7.3	The 70-Task Problem of Tonge (1961) . . . . .	62

# List of Tables

4.1	<i>ANOVA results for fitness scores</i>	34
4.2	<i>Bonferroni and Duncan grouping of fitness scores due to DPC.</i>	35
4.3	<i>ANOVA results for CPU times . . . . .</i>	36
4.4	<i>Bonferroni and Duncan grouping of CPU times due to DPC . .</i>	37
4.5	<i>Bonferroni and Duncan grouping of fitness scores due to F-Ratio</i>	38
5.1	<i>ANOVA results for fitness scores</i>	46
5.2	<i>Bonferroni and Duncan grouping of fitness scores due to <math>\alpha</math>, F-Ratio, population size . . . . .</i>	47
6.1	<i>Optimum parameters for Classical GA and Modern GA . . . . .</i>	52
6.2	<i>ANOVA results for the comparison of two algorithms . . . . .</i>	53
6.3	<i>Bonferroni and Duncan grouping of fitness scores due to algorithm.</i>	53
7.1	<i>The heuristic methods to solve the Kilbridge-Wester problem . .</i>	58
7.2	<i>Comparison of non-GA heuristics, Leu et al.'s GA and the proposed GA . . . . .</i>	60
7.3	<i>Factor levels at which the optimum solution is found . . . . .</i>	60

7.4 *Comparison of eight methods on the 70-task problem of Tonge  
(1961) in terms of number of stations . . . . .* 64

# Chapter 1

## INTRODUCTION

An assembly line consists of a sequence of work stations which are connected by a conveyor belt. Each station has to perform repeatedly a specified set of tasks on consecutive product units moving along the line at constant speed. Because of the uniform movement of the line, each product unit spends the same fixed time interval, called the cycle time, in every work station. As a consequence, the cycle time determines the production rate which is equal to the reciprocal of the cycle time. Tasks or operations are indivisible elements of work which have to be performed to assemble a product. The execution of each task is assumed to require a fixed amount of time. Due to technological restrictions, precedence constraints partially specifying the sequence of tasks have to be considered. These constraints can be represented by a precedence graph containing nodes for all tasks and arcs  $(i, j)$  if task  $i$  has to be completed before task  $j$  can be started. The Assembly Line Balancing (ALB) problem is to allocate the tasks equally to a minimum possible number of stations such that each task is assigned to exactly one station and no precedence constraint is violated.

The ALB problem has been first introduced by Helgeson et al. in 1954 [3], and has become an important research area since then. However, Artificial Intelligence (AI) techniques such as Genetic Algorithms (GAs) have been introduced to the ALB problem very recently (i.e., Leu et al, 1994).

Assuming that no precedence relationship exists, a modest assembly line consisting of 30 tasks has  $30!$  ( $2.6 \times 10^{32}$ ) possible schedules. If one can develop a system using heuristic rules which can limit this explosion, the search space can be reduced to a reasonable size. The main idea of AI is based on the concept of this intelligent search. The search methods used to find the optimum solution to the ALB problem require unreasonable computational effort that increases exponentially as the problem size gets larger. This necessitates the adoption of different strategies which apply heuristic information to the search technique.

The basic notion in all the heuristic search methodologies is to use the problem specific knowledge intelligently to reduce the search efforts. GAs are intelligent random search mechanisms that can easily be applied to optimization problems. Provided that standard GA operators are modified to work effectively in the specific problem domain, GA can be a very powerful search mechanism. This has already been proved by Leu et al. [29], Suresh et al. [37], and Anderson and Ferris [1] in the ALB problem domain. In this study, we take a further step and make use of a specific characteristic of the ALB problem to adopt the GA structure as well as its operators to work more effectively than the above mentioned GA attempts to the ALB problem.

The rest of the thesis is organized as follows. After a comprehensive literature review that reveals our motivation for this study in Chapter 2, we explain our algorithms in Chapter 3. The proposed approach that exploits the special characteristics of the ALB problem is presented in Chapter 4. We also integrate two AI tools, GA and simulated annealing (SA), working together to achieve a better performing search in Chapter 5. Then we compare the performance of two major types of basic GA structures on the ALB problem in Chapter 6. We then demonstrate the performance of our algorithm on ALB problems reported in the literature and compare it with the best performing heuristics. Finally, we summarize our study and discuss our major findings in the conclusion chapter.

# Chapter 2

## LITERATURE REVIEW

### 2.1 Assembly Line Balancing

An assembly line consists of a finite set of work elements or tasks, each having an operation processing time and a set of precedence relations, which specify the permissible orderings of the tasks. The line balancing problem is assigning the tasks to an ordered sequence of stations, such that the precedence relations are satisfied and some measure of effectiveness is optimized (e.g. minimize the balance delay or minimize the number of work stations) [14].

Since the assembly line balancing (ALB) problem was first formulated by Helgeson et al. in 1954 [3], many solution approaches have been devised, starting with Salveson (1955) [33]. Salveson is the first to publish it in mathematical form and propose a linear programming (LP) solution. Since the ALB problem falls into the NP hard class of combinatorial optimization problems, it has not been possible to develop efficient algorithms for obtaining optimal solutions [14]. Hence, numerous research efforts have been directed towards the development of computer efficient approximation algorithms or heuristics.



Ghosh and Gagnon (1989) classify the ALB problem and the accompanying research and literature into four categories, as shown in Figure 2.1: Single Model Deterministic (SMD), Single Model Stochastic (SMS), Multi/Mixed Model Deterministic (MMD), and Multi/Mixed Model Stochastic (MMS).

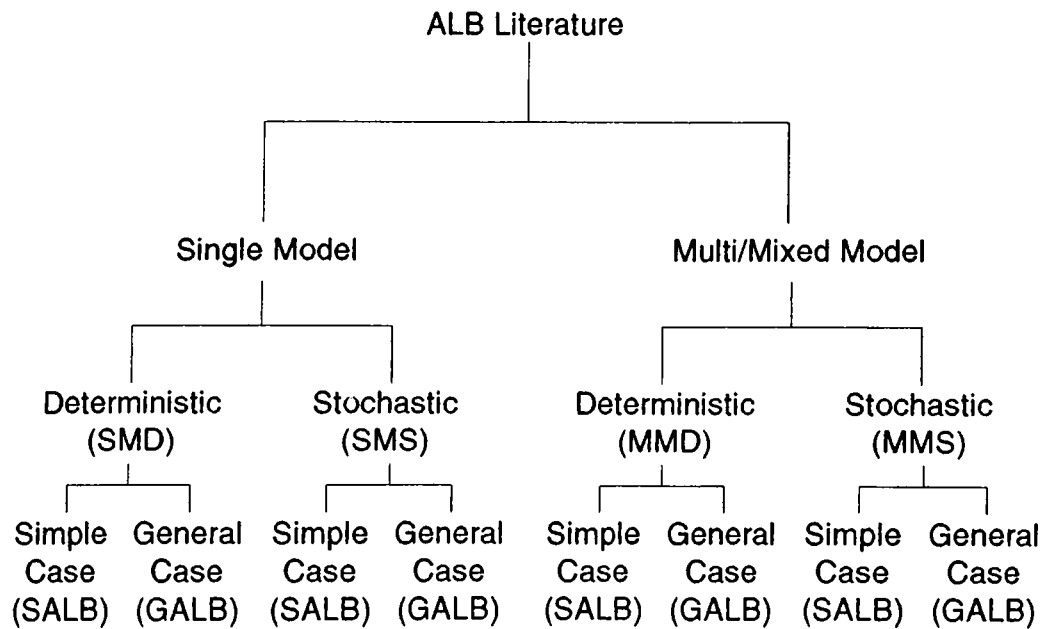


Figure 2.1: Classification of Assembly Line Balancing Literature (Taken from Ghosh and Gagnon (1989))

The SMD version of the ALB problem assumes dedicated, single-model assembly lines where the task times are known deterministically and an efficiency criterion is optimized. This is the original and simplest form of the assembly line balancing problem (SALB). If other restrictions or factors (e.g. parallel stations, zoning restrictions) are introduced into the model, it becomes the General ALB problem (GALB). Our research area is the SMD category's SALB subcategory. It is also known as type-1 assembly line balancing problem since the cycle time is fixed and we aim to minimize the number of stations. The variation in which the number of stations is fixed to minimize the cycle

time is referred to as the type-2 assembly line balancing problem. The SMD-SALB category has been the most researched, as evidenced by the number of articles published in the literature, i.e. 64 articles since 1983 [14]. A summary of the previous research in this category is as follows:

Salveson (1955) formulated the SALB version of the SMD problem as an LP problem. Bowman (1960) (later modified by White 1961) came up with an integer programming (IP) solution, describing task assignments to stations with binary variables. IP formulations were contributed by Klein (1963), Thangavelu and Shetty (1971), Patterson and Albracht (1975), and Talbot and Patterson (1984). The formulation provided by Patterson and Albracht (1975), a general integer program without binary variables, significantly reduced the size of the problem formulation. Dynamic programming (DP) formulations were contributed by Jackson (1956), Held et al. (1963), Kao and Queyranne (1982), Held and Karp (1962), and Schrage and Baker (1978). Specialized branch and bound approaches (those not based on general IP theory) were contributed by Jackson (1956), Hu (1961). Van Assche and Herroelen (1979), Johnson (1981), and Wee and Magazine (1981).

Besides the reasonable progress in the development of optimal seeking approaches, considerable advancement has been achieved in the development of heuristic approaches to solve the SMD problem as well. According to a study by Talbot et al. (1986), Hoffman's Precedence Matrix approach (1963), Dar-El's MALB (1973), and Dar-El and Rubinovitch's MUST (1979) are the most promising of the heuristic techniques for the SALB problem. Baybars' LBHA, devised more recently than the heuristics mentioned before, is an efficient heuristic, as well [4]. It consists of several reduction phases, i.e. reduction via node elimination, determining the sets of tasks that are likely to be in the same station, decomposing the network, and determining feasible sub-sequences of tasks. After pre-processing the problem by the reduction phases the heuristic solution phase starts, which is a backward procedure that starts with the last tasks in the precedence diagram and bases the assignment decisions on the principle that last tasks are likely to be assigned to the last stations along the line. Baybars presents a comparison of his heuristic with Tonge's (1965),

Moodie and Young's (1965), and Nevins' (1972) heuristics on Tonge's problems. We will also present our results on the same problem set, and compare with the other heuristics in the later chapters.

The SMS problem formulation introduces the task time variability. The MMD problem category assumes deterministic task times, but introduces the concept of an assembly line producing multiple products. Multi-model lines assemble two or more products separately in batches. In mixed-model lines, single units of different models can be introduced in any order or mix to the line. Since multi-model lines are equivalent to mixed-model lines for batch size equals to one, they are classified in one category. MMS differs from MMD in that stochastic task times are allowed.

## 2.2 Genetic Algorithms

In this section, we give a brief review of genetic algorithms (GAs) together with their recent applications to manufacturing problems.

GAs are adaptive methods which may be used to solve search and optimization problems. They are based on genetic processes of biological organisms. Over many generations, natural populations evolve according to the principles of natural selection and survival of the fittest. By mimicking this process, GAs are able to evolve solutions to real life problems, if they have been suitably encoded.

In nature, individuals who are most successful in surviving will have relatively a large number of offsprings. Poorly performing individuals, on the other hand, will produce less number of offsprings, or even none after some point in time. This means that the genes from the highly adapted, or fit individuals will spread to an increasing number of individuals in each successive generation. The strong characteristics from different ancestors can sometimes produce super-fit offspring, whose fitness is greater than that of either parent. In this way, species evolve to become more and more well suited to their

environment.

GAs use a direct analogy to natural behavior. They work with a population of individuals, each representing a possible solution to the given problem. Actually, individuals are represented by their "chromosomes" that carry their characteristic specifications on the "genes" which are ordered on chromosomes. Each chromosome is assigned a "fitness score" according to the quality of the solution it provides to the problem. The highly fit chromosomes are given opportunity to reproduce by cross breeding or "recombining" with other individuals in the population. This produces new individuals as offspring, which share some features taken from each parent. The least fit members are less likely to get selected for reproduction, so they die out. The new generation contains a higher proportion of the characteristics possessed by the superior members of the previous generation. In this way, over many generations, superior characteristics are preserved and individuals of the population are enhanced on the average due to their fitness score. Hence, if the GA is well designed, the population will converge to an optimal or near-optimal solution at the end. Holland (1975) showed that a computer simulation of this process of natural adaptation could be employed for solving optimization problems. Goldberg (1989) provides a comprehensive introduction to the theory, operation, and application of GAs in search, optimization and machine learning [15].

The power of GAs comes from the fact that the technique is robust, and can deal with a wide range of problem areas. GAs are not guaranteed to find the optimal solution but they are generally successful at finding acceptable good solutions to problems acceptable quickly. If specialized techniques exist for solving particular problems, they are likely to outperform GAs in both speed and the accuracy of the final result. The main ground for GAs is then, is in difficult areas where no such techniques exist. On the contrary, even where existing techniques work well, improvements have been made by hybridizing them with GAs.

### 2.2.1 Basic Structure of a GA Process

The standard (or classical) GA algorithm can be represented as follows.

The following notations are used in the algorithm:

$R_x$  denotes the crossover rate,

$R_m$  denotes the mutation rate, and

$N_p$  denotes the population size.

**Algorithm 2.1** : *Classical GA*

**begin**

*Generate initial population*

*Compute fitness of each individual*

**while** *Termination\_Criteria not reached* **do**

*Select  $0.5 \times R_x \times N_p$  pairs of parents from old generation  
for mating*

*Recombine the selected pairs to give offsprings*

*Mutate  $R_m \times R_x \times N_p$  offsprings chosen at random*

*Compute the fitness of the offsprings*

*Insert offsprings in the new generation*

**end**

*Choose the best-fit chromosome and the corresponding solution*

**end.**

### 2.2.2 Coding

Before a GA can be run, a suitable coding for the problem must be devised. It is assumed that a potential solution to the problem may be represented as

a set of parameters. These parameters (genes) are joined together to form a string of values (chromosomes). The ideal is to use a binary alphabet for the string but there are other possibilities, too.

In genetic terms, the set of parameters represented by a particular chromosome is referred as a genotype. The genotype contains the information required to construct an organism which is referred to as the phenotype. The same terms are used in GAs. The fitness of an individual depends on the performance of the phenotype. This can be inferred from genotype.

### 2.2.3 Fitness Function

GAs require a fitness function, which assigns a figure of merit to each coded solution. Given a particular chromosome, the fitness function returns a single numerical fitness which is supposed to be proportional to the utility or ability of the individual represented by that chromosome.

### 2.2.4 Reproduction

Parents are selected randomly from the population using a scheme which favors the more fit individuals. Having selected two parents, their genes are recombined on a new chromosome, typically by using the mechanisms of crossover and mutation.

#### Crossover

We will explain the crossover operator by means of an example: One popular crossover mechanism is the uniform crossover technique. In this technique, each gene in the offspring is created by copying the corresponding gene from one or the other parent, chosen according to a randomly generated crossover mask. The crossover mask consists of ones and zeros. If the number on the mask

corresponding to a gene is zero, then that gene is transported from the first parent, otherwise the corresponding gene from the second parent is transported to the offspring chromosome. This process is presented in Figure 2.3. The process is repeated with the parents exchanged to produce another offspring. A new crossover mask is randomly generated every time the crossover process is repeated.

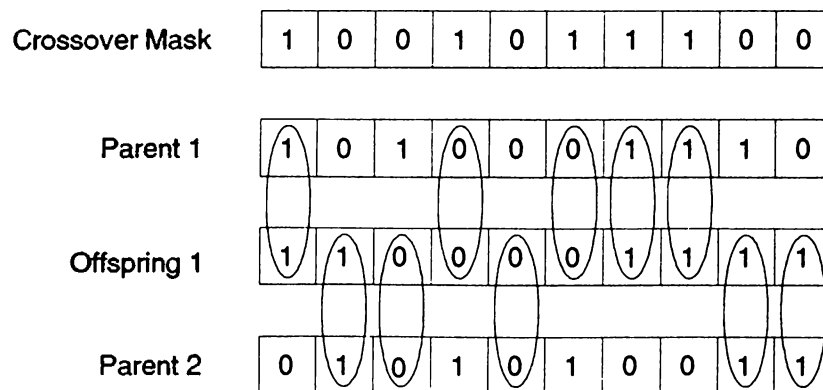


Figure 2.2: Uniform Crossover Mechanism

## Mutation

Mutation is applied to each individual after crossover. It randomly alters each chromosome with a small probability, referred to as the mutation rate. Despite its generally low probability of use, mutation is a very important operator. Its optimum probability is much more critical than that of crossover. Mutation provides a small amount of random search, and helps ensure that no point inside the search space has zero probability of being examined. An example to mutation is exchanging the places of two randomly selected genes on a particular chromosome. For instance, if this mutation operator was applied to

Parent 1 in Figure 2.3, assuming that the fourth and eighth genes are randomly selected for exchanging their places, the mutated chromosome would be 1011001010.

### 2.2.5 Genetic Algorithm Applications

Genetic algorithms are applied to various kinds of manufacturing problems. Some very recent examples from different fields of manufacturing are as follows: Suresh et al. (1995) devised a GA for facility layout, and showed that the population maintained by the GA for facility layout should consist of feasible solutions only [36]. Bullock et al. (1995) underlined the potential of the genetic algorithms both as a high-level decision support technique during the preliminary stages of the design process and as a detail design of complex components [7]. Chen and Tseng (1996) presented the planning of a near-optimum path and location of a workpiece (i.e. robot arm) by genetic algorithms [9]. Kamhawi et al. (1996) addressed the feature sequencing problem in the Rapid Design System, that is a feature-design system that integrates product design and process planning [25]. Gupta et al. (1996) proposed a GA based solution approach to address the machine cell-part grouping problem [16]. Gupta et al. (1995) used a GA approach to solve a problem formulated which minimizes the intercell and intracell part movements in cellular manufacturing [17]. Wellman and Gemmill (1995) applied GAs to the performance optimization of asynchronous automatic assembly systems [46]. Starkweather et al. (1991) devised a genetic recombination operator for the traveling salesman problem, which is proved to be superior to previous genetic operators for this problem [35]. Davis (1985), devised a GA for job shop scheduling [13].

## 2.3 ALB and GA

There are only three articles in literature which deals with ALB using GAs. While one of them deals with the deterministic (SMD) SALB problem, the



other two are concerned about the stochastic (SMS) case. We present a comprehensive review of these articles in chronological order.

Leu et al. (1994) has introduced the concept of GAs to the SALB problem [29]. In this study, the authors used heuristic solutions in the initial population to obtain better results than the heuristics. They also demonstrated the possibility of balancing assembly lines with multiple criteria and side constraints. According to the authors, the GA approach has the following advantages: i) GAs search a population rather than a single point and this increases the odds that the algorithm will not be trapped in a local optimum since many solutions are considered concurrently, and ii) GA fitness functions may be of any form (i.e., unlike gradient methods that have differentiable evaluation functions) and several fitness functions can be utilized simultaneously.

The coding of the solution to the ALB problem is done by representing the number of the tasks on a chromosome in the order that they take place in the assembly line. For example, "1 3 6 5 2 7 4" can be a chromosome for a 7 task ALB problem. Then, stations are formed such that the first station is filled with the tasks on the chromosome, starting with the first task and proceeding with the next ones until the station time reaches the cycle time. This procedure is repeated in the same way for the other stations until every task on the chromosome is placed in a station, e.g. "1 3 6" (station time = 25) and "5 2 7 4" (station time = 19) are the two stations for a cycle time of 25, for the above example. The fitness functions used by Leu et al. (1994) are: i)  $z_1 = \text{smoothness index} = \sum_{k=1}^n (C - S_k)^2 / n$ , where  $n$  is the number of stations,  $S_k$  is the  $k$ th station time, and  $C$  is the cycle time, ii)  $z_2 = \sum_{k=1}^n (C - S_k) / n$ , iii)  $z = 2\sqrt{z_1} + z_2$  to demonstrate balancing with multiple criteria, and iv)  $\text{efficiency} = 1 - \frac{\sum_{k=1}^n (C - S_k)}{n \times C}$ .

The population revision mechanism used by the authors is similar to Whitley and Kauth's (1988) GENITOR [48]. The steps of the algorithm of Leu et al. (1994) are as follows:

**Algorithm 2.2 : Modern GA**

*Generate initial population*  
**repeat**  
    *Choose two parents by roulette wheel selection*  
    *Decide whether to recombine or mutate*  
    *Form two offsprings by recombination or one by mutation*  
    *Replace parents with offsprings if they are outperformed by offsprings*  
**until** *Stopping-Criterion is reached*

The initial population is generated randomly by assuring feasibility of precedence relations. Roulette wheel selection is a procedure that selects a chromosome from a population with a probability directly proportional to its fitness (i.e., the best-fit chromosome has the greatest probability of being selected amongst a population). The decision between recombining or mutating depends on a certain probability, i.e., if the probability of recombining is 98% then the probability of mutating is 2%. Replacing a parent with an offspring only if the offspring is better than the parent is called *elitism*. Elitism rule will be discussed in detail in Chapter 4. The crossover (recombination) operator is a variant of Davis' (1985) *order crossover* operator [12]. The two parents that are selected for crossover are cut from two random cut-points. The offspring takes the same genes outside the cut-points at the same location as its parent and the genes in between the cut-points are scrambled according to the order that they have in the other parent. This procedure is demonstrated in Figure 2.3. The major reason that makes this crossover operator a very suitable one for ALB is that it assures feasibility of the offspring. Since both parents are feasible and both offsprings are formed without violating the feasibility sequence of either parent, both children must also be feasible. Keeping a feasible population is a key to ALB problem since preserving feasibility drastically reduces computational effort.

The mutation operator of Leu et al. (1994) is *scramble mutation*, that is, a random cut-point is selected and the genes after the cut-point are randomly

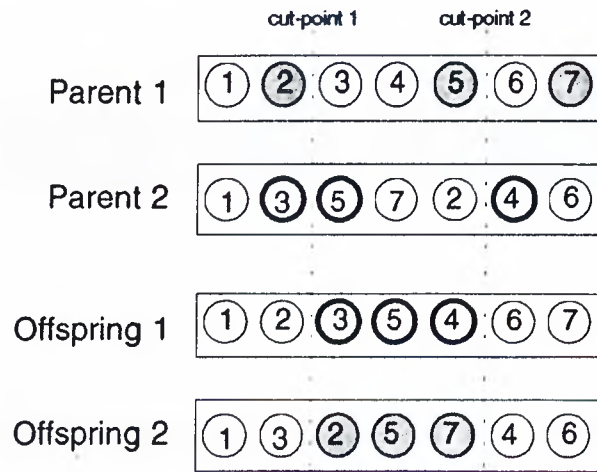


Figure 2.3: Crossover Operator of Leu et al. (1994)

replaced (scrambled), assuring feasibility. For example, if parent 1 in Figure 2.3 is mutated with the same cutpoint as cut-point 1 then tasks 1 and 2 stay in their current places but tasks 3, 4, 5, 6, and 7 are randomly replaced by assuring feasibility of precedence constraints. Elitism is applied to the mutation procedure as well, i.e., a chromosome is mutated only if mutation improves its fitness value.

Anderson and Ferris (1994) demonstrated that GAs can be effective in the solution of combinatorial optimization problems, working specifically on the ALB (SALB) problem [1]. In the first part of the paper, they describe a fairly standard implementation for the ALB problem. Experiments are reported to indicate the relative importance of crossover and mutation mechanisms and the scaling of fitness values. In the second part, an alternative parallel version of the algorithm for use on a message passing system is introduced. The authors note that they did not expect a GA to be as effective as some of the special purpose heuristic methods for the ALB problem. Their aim is not to demonstrate the superiority of a GA for ALB problem, but rather to give some indication for the potential use of this technique for combinatorial optimization problems.

They coded the problem in a different way than Leu et al. (1994). The

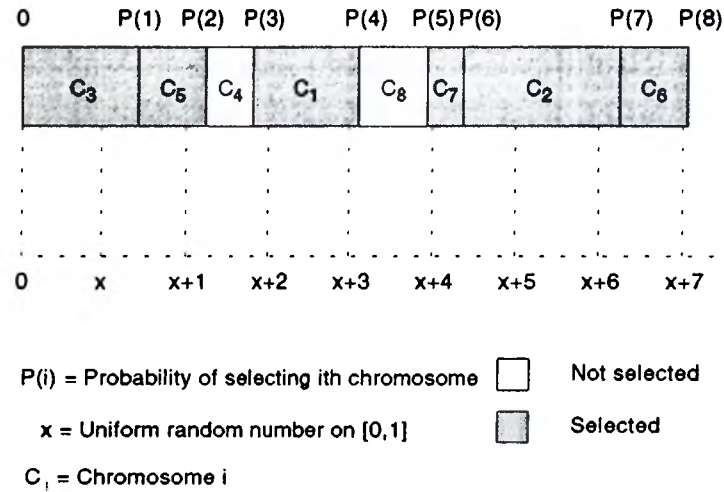


Figure 2.4: Stochastic Universal Sampling

solution is represented by a string of numbers such that the number in the  $i$ th place of the string is the station to which the  $i$ th operation is to be assigned. Stochastic universal sampling is used as the selection procedure. That is, assigning each chromosome an interval proportional to its fitness (i.e., as in roulette wheel selection procedure) such that the total length of the intervals is  $N$ , and connecting all the intervals side by side in random order such that they form an imaginary line. Then a random number  $x$  is chosen uniformly on  $[0,1]$  and the line of intervals is marked at points  $x, x + 1, \dots, x + N$ . Finally, the chromosomes corresponding to the marked intervals are put in the mating pool. This procedure guarantees that the more-than-average-fit chromosomes are selected for recombination, whereas roulette wheel selection does not. Stochastic random sampling is demonstrated by means of an example in Figure 2.4. The eight tasks of the assembly line are shuffled and arranged in random order as 3, 5, 4, 1, 8, 7, 2, 6. They correspond to an interval proportional to their probability of selection,  $P(1), P(2), \dots, P(8)$ , respectively. If the total length of the intervals is 8 then the average length of the intervals is 1, that is equal to the sampling interval. Hence, the tasks that have selection probabilities above the

average selection probability, i.e., tasks 1, 2, 3, are guaranteed to be selected. On the other hand, some of the tasks that have selection probabilities below the average are not selected, i.e., tasks 4 and 8. The tasks that are selected by stochastic random sampling are placed in the mating pool in the order that they appear in Figure 2.4 (i.e., 3, 5, 1, 7, 2, 6). For instance, if four chromosomes are required to be placed in the mating pool, then the tasks 3, 5, 1, 7 are placed. The authors compared it with another commonly used method called "remainder stochastic sampling without replacement" and found that this procedure is similar in performance to stochastic universal sampling. However, a comparison with roulette wheel selection or any other selection procedure is not included in this paper.

Infeasible solutions (chromosomes) are allowed in the population but the population is forced to feasibility by assigning high penalty costs to the infeasible chromosomes. The standard single point crossover operator is used, that is, two offsprings are obtained from two parents by choosing a random point along the chromosome, both chromosomes are split at that point, and then the front part of one parent is joined to the back part of the other parent and vice versa. The mutation operator randomly increases or decreases the value of one gene of a chromosome by one unit (i.e., the task that is represented by that gene is transferred to a neighbor station). Elitism is used in this study, but in two different ways: i) the best fit chromosome is transferred to the next population so that the final generation is guaranteed to contain the best solution ever found, and ii) if any of the offsprings perform worse than the worst individual in the previous generation then that offspring is not retained and, instead, one of the parents is allowed to continue in the next generation. While the first type of elitism is desirable from the point of view of assessing the relative performance of different versions of the GA, the second type is effective in speeding up the convergence of the algorithm. The algorithm stops after a certain number of iterations (i.e., 350).

Suresh et al. (1996) used GAs to solve the SMS type ALB problem [37]. The ability of GAs to consider a variety of objective functions is imposed as the major feature of GAs. A modified GA working with two populations, one

of which allows infeasible solutions, and exchange of specimens at regular intervals is proposed for handling irregular search spaces, i.e., the infeasibility problem due to precedence relations. The authors claim that a population of only feasible solutions would lead to a fragmented search space, thus increasing probability of getting trapped in a local minima. They also state that infeasible solutions can be allowed in the population only if genetic operators can lead to feasible solutions from an infeasible population. Since a purely infeasible population may not lead to a feasible solution in this particular problem, two alternative populations, one purely feasible and one allowing a fixed percentage of infeasible chromosomes, is combined in a controlled pool to facilitate the advantages of both of them. Certain chromosomes are exchanged at regular intervals between the two populations. The two chromosomes that are exchanged have the same rank of fitness value in its own population. Experimental results on large sized problems showed that the GA working with two populations gives better results than the GA with one feasible population.

The coding scheme used by Suresh et al. (1996) is the same as Leu et al.'s (1994). The SMS type problem is converted into SMD by assuming deterministic station times calculated as follows:  $ST = S_{mean} + \sigma\sqrt{S_{var}}$ , where  $ST$  is the station time for each station,  $S_{mean}$  is the sum of the means of tasks assigned to that station,  $\sigma$  is the confidence coefficient for normally distributed task times, and  $S_{var}$  is the sum of the variances of tasks assigned to that station. Moodie and Young's (1965) smoothness index ( $SI = \sum_{k=1}^n (S_{max} - S_k)^2 / n$ , where  $n$  is the number of stations,  $S_{max}$  is the maximum station time, and  $S_k$  is the  $k$ th station time), and Reeve's (1971) objective to minimize the probability of line stopping are used as the objective functions. The probability of a station not exceeding the cycle time is denoted by  $P_s$ . It is the area under the normal curve corresponding to the value of  $z$  given by  $z = \frac{C - S_{mean}}{\sqrt{S_{var}}}$ . The probability of line stopping is given by  $P = (1 - \prod_{s=1}^n P_s)$ .

Single point conventional crossover mechanism is used. Then a correction algorithm is applied to the infeasible offsprings in the purely feasible population to make them feasible. Two kinds of mutation is used: i) swapping two tasks in different stations, and ii) interchanging all the tasks of two different stations.

Feasibility is assured in the mutation operation by checking if the cycle time or the precedence constraints are violated.

## 2.4 Motivation and Organization

As stated above, there are many heuristics and exact methods proposed for the ALB problem. Because of the NP hard characteristic of the ALB problem, it has not been possible to develop efficient algorithms for obtaining optimal solutions [14]. Hence, numerous research efforts have been directed towards the development of computer efficient heuristics. Intelligent heuristic approaches (i.e., different than the traditional heuristics that work basically with priority rules) such as GAs, started to emerge recently, i.e., since 1994. The limited number of GA studies on the ALB problem have been helpful in demonstrating that GA is a promising heuristic for the ALB problem and that GAs are superior to traditional heuristics in certain respects such as multi objective optimization and flexibility due to change in problem's constraints. These properties of GAs are not limited to only the ALB problem. In this study, we direct our research towards exploiting the characteristics of the ALB problem to improve the GA structure specially designed for the ALB problem. After providing comprehensive information about our initial GA structure in Chapter 3, we explain the proposed GA approach in Chapter 4.

We make use of another characteristic of the ALB problem in Chapter 5. The ratio of the number of precedence relationships are compared to the maximum possible number of precedence relationships, i.e., flexibility ratio (F-Ratio). We observe that the use of elitism rule contributes to the GA, but in some problems with high ratios, strict elitism causes early convergence. Hence, we redirect our study towards relaxing the elitism rule with the simulated annealing idea.

There are two major ways of moving from one population to its neighbor population in literature, that can be classified as the classical GA and the

modern GA structures. We initially use the modern GA structure, but we code the classical structure as well, and test which structure is more appropriate for the ALB problem in Chapter 6. This comparison has not been done before in GA literature either for the ALB problem or for any other problems.

Finally, we solve two well known ALB problems (i.e., Kilbridge and Wester (1961) and Tonge (1961)) and compare our results to the best performing heuristics in literature in Chapter 7.



## Chapter 3

# THE PROPOSED GENETIC ALGORITHMS

As it has been mentioned in Chapter 2, there are two kinds of GA structures in the literature: i) classical GA (i.e., Algorithm 1.1), and GENITOR type or modern GA (i.e., Algorithm 1.2). In this study, we first use a modern GA structure which forms the skeleton of our GA. While the classic structure performs many crossovers to generate the consecutive population, our GA performs only one recombination operation between two iterations. Because of the reason that our first algorithm resembles GENITOR's structure, we will refer to it as GENITOR type (or modern GA) from now on. Hence, the GA that is mentioned in Chapter 4 and Chapter 5 has the GENITOR type structure.

Although both structures have been used before in many studies, they have not been compared. We make this comparison and discuss the advantages and drawbacks of both structures in Chapter 6. We present the algorithms as follows:

### **Algorithm 3.1** : *Classical GA*

*Generate initial population*

**repeat**

*Transfer the best  $(1 - R_x) \times 100$  chromosomes to the next population*

**repeat**

*Choose two parents from the current population for crossover*

*Include the offsprings in the next population*

**until** *Next population is full*

*Apply mutation to one of the chromosomes with  $R_m$  probability*

**until** *Stopping-condition is reached*

*Take the best-fit chromosome of the final population as the solution*

**Algorithm 3.2** : *Modern GA*

*Generate initial population*

**repeat**

*Choose two parents for recombination*

*Apply mutation with  $R_m$  probability or crossover with  $1 - R_m$  probability*

*Replace parents with offsprings*

**until** *Stopping-condition is reached*

*Take the best-fit chromosome of the final population as the solution*

### 3.1 The Characteristics of the Proposed GAs

The specific characteristics of the two algorithms like the crossover operator, mutation operator, fitness function, etc. are the same. Some of these characteristics are devised with the inspiration taken from current examples in literature that are proved to be successful. We describe these characteristics as follows:

1. *Coding*: Each task is represented by a number that is placed on a string of numbers (i.e., chromosome), such that the string size is the number of tasks. The tasks are ordered on the chromosome relative to their order of processing. Then the tasks are divided into stations such that the

total of the task times in each station does not exceed the cycle time. For example, the first task on the chromosome is assigned to the first station, and if the total of the first and the second task's times does not exceed the cycle time then the second task is assigned to the first station, otherwise it is assigned to the second station. The task to station assignment procedure goes on like this until the last task is assigned to the final station. The coding scheme is demonstrated in Figure 3.1.

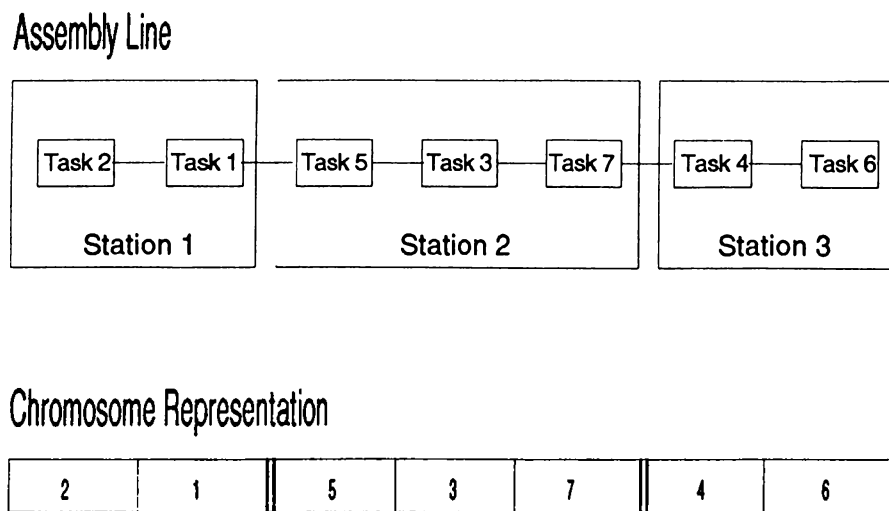


Figure 3.1: Coding the Chromosome Representation of an Assembly Line

2 *Fitness function:* The objective in a type-1 ALB problem is clearly to minimize the number of stations, but given two different solutions with the same number of stations, one may be "better balanced" than the other. For example, a line with three stations may have stations times as 30-50-40 or 50-50-20. We consider the 30-50-40 solution to be superior (better balanced) to the 50-50-20 solution. Hence, we used a fitness

function that combines the two objectives (i.e., minimizing the number of stations and obtaining the best balanced station):

$$\text{Fitness Function} = 2\sqrt{\frac{\sum_{k=1}^n (S_{\max} - S_k)^2}{n} + \frac{\sum_{k=1}^n (S_{\max} - S_k)}{n}}$$

where  $n$  is the number of stations,  $S_{\max}$  is the maximum station time, and  $S_k$  is the  $k$ th station time. The first part of our fitness function aims to find the best balance among the solutions that have the same number of stations while the second part only aims to minimize number of stations in the solution.

- 3 *Crossover & Mutation:* We use a variant of "order crossover operator", and scramble mutation operator. Both of these operators create feasible offsprings and they are the same as Leu et al.'s [29] crossover and mutation operators. (Refer to Chapter 2 for detailed explanation.)
- 4 *Scaling:* The fitness scores need to be scaled such that the total of the scaled fitness scores are equal to 1, in order to activate the selection procedure (i.e., roulette wheel selection). Since our objective is to minimize the fitness scores, we need to assign the highest scaled fitness score to the lowest fitness score and vice versa, to assign a probability of selection that is proportional to the fitness of chromosomes. We achieve this by subtracting each fitness value from the double of the highest (worst) fitness value in the population and assigning the subtrahend as the new fitness value of that chromosome. Then, by dividing each new fitness score by the total of new fitness scores, we scale the fitness scores such that their total equals to 1.
- 5 *Selection Procedure:* We use a well known selection procedure called "roulette wheel selection". Fitness scores are scaled as described above, and each chromosome is assumed to consist of an interval proportional to its scaled fitness score, all intervals placed next to each other on the  $[0,1]$  interval. Then, a uniform random number in the  $[0,1]$  interval is generated, and the chromosome which is assigned to the interval corresponding to the random number is selected. This procedure selects chromosomes proportional to their fitness scores.

6 *Stopping Condition:* The algorithm stops after a certain number of iterations. We used 500, 1000, and 2000 values for the number of iterations parameter, but we only use the 500 value in Chapters 5 and 6.

## 3.2 Classical GA vs Modern GA

The basic difference between the classical and modern GA structures is the number of crossovers at each iteration. While the classical GA performs a number of crossovers between a fixed proportion of the members of its population, the modern GA performs only one crossover between two of its members. Additionally, a group of best performing chromosomes of the current population is transferred to the next population in classical GA. Although the classical approach seems to provide a more comprehensive search by definition, we observed that the modern approach is able to compete with the classical approach, besides it requires much less CPU time. The details of a comparison between the classical and modern GA structures is given in detail in Chapter 6.

## 3.3 Dynamic Partitioning

Although there are many attempts to improve the performance of GAs for ALB (i.e., working with two populations [37], including heuristic solutions in the initial population [29], working with multiple evaluation criteria [29, 37], controlling the convergence speed by adjusting the scaling parameter and parallel implementation of the algorithm [1]) in the literature, no attempts have been taken to reduce the problem size prior to or during the solution procedure. Reduction of the problem size prior to the solution procedure have been done in some heuristics (i.e., [4]), but reduction during the solution procedure has not been implemented in any of the heuristics or random search algorithms before. Therefore, we introduce the idea of dynamic partitioning, that is, reducing

the problem size while the algorithm is running, which is a new development in both the ALB and the GA literature. We applied dynamic reduction by partitioning the chromosomes of the GA population, and freezing some parts of the chromosomes due to some freezing criteria (i.e., see Chapter 4) and continuing the remaining iterations with the unfrozen part. Hence, we call this reduction process as "Dynamic PARTitioning (DPA)". We applied DPA on the modern GA structure and we achieved improved results. DPA process and its experimentation are described in detail in Chapter 4. DPA has been applied to the modern GA in Chapter 4, but its implementation on the classical GA is discussed as well in Chapter 6. The modern GA is modified due to the integration with DPA as follows:

**Algorithm 3.3** : *Modern GA with DPA*

```

Generate initial population
repeat
    Choose two parents for recombination
    Apply mutation with  $R_m$  probability or crossover with  $1 - R_m$  probability
    Replace parents with offsprings
    if the DPA criteria is satisfied then
        Freeze a set of tasks (genes)
        Deducer the frozen tasks from all the chromosomes in the population
until Stopping-condition is reached
Take the best-fit chromosome of the final population as the solution

```

The classical GA is modified in a similar manner, so we do not include the modified algorithm here.

### 3.4 Elitism With Simulated Annealing

The elitism rule has been applied in many GAs before, but its effect on the performance of the algorithm has not been discussed. We do a comprehensive study of the elitism rule in Chapter 5, and we also introduce the concept of relaxing the elitism rule by using the Simulated Annealing (SA) idea. Just like the fitness score scaling factor, elitism is a factor that affects the convergence of the GA population. If elitism is applied without any control parameter, the algorithm may be induced by the negative effects of early or late convergence. Hence, we start with no elitism and then increase the elitism level iteration by iteration, controlling its level by SA. The concept of SA and its application to elitism is discussed in detail in Chapter 5. Thus, we only present the basic modifications on the modern GA structure in this chapter as follows:

**Algorithm 3.4** : *Modern GA with SA controlled elitism*

```

Generate initial population
Set  $P_e$  equals to 1
repeat
    Choose two parents for recombination
    Apply mutation with  $R_m$  probability or crossover with  $1 - R_m$  probability
    Obtain two offsprings by crossover or one by mutation
    Apply following to each offspring
        if the offspring is better than its parent then
            Replace parent with its offspring
        else Replace parent with its offspring with  $R_e$  probability
    Reduce  $P_e$ 
until Stopping-condition is reached
Take the best-fit chromosome of the final population as the solution

```

## Chapter 4

# DYNAMIC PARTITIONING

Dynamic PArtitioning (DPA) is a method that modifies chromosome structures of Genetic Algorithms (GAs) in order to save from CPU time and to achieve improved results, if possible. DPA modifies the chromosome structure by freezing the tasks that are allocated in certain stations that satisfy some criteria, and continues with the remaining iterations without the frozen tasks. Hence, DPA allows the GA to focus on the allocation of the remaining tasks during the search, and saves a considerable amount of computation time. In what follows, we use "without DPA" to refer to the traditional GA.

### 4.1 Motivation

Although a typical GA developed for assembly line balancing (ALB) problem is a fast problem solver (our GA solves a 50 task problem after 500 iterations in approximately 1.5 seconds on a pentium 133 PC), it needs an experimental design of several factors in order to tune the parameters for each type of ALB problem. Hence, it has to be run a lot of times, in the order of ten thousands, for parameter tuning, and this requires a significant amount of CPU time in the order of days. Therefore, our motivation for devising the DPA methodology is to save a considerable amount of CPU time even if we have to sacrifice some



from the GA's performance, i.e. the final fitness score. In fact, we found out that the performance improves significantly as well, while a significant amount of CPU time saving is achieved. We claim that the underlying reason for the performance improvement is that DPA activates the GA to work out more effectively with the remaining "a fewer number of tasks" after each freezing or partitioning.

## 4.2 Implementation

For the sake of continuity between the remaining tasks, we consider freezing the tasks that are allocated at the first and the last stations, (i.e., the genes at the beginning and at the end of the chromosome are considered as potentially freezable). The second criteria for freezing is to achieve an optimal station time at the potentially freezable stations. This optimality condition depends on the fitness function. The freezing criteria that best fits to our fitness function,

$$z = 2\sqrt{\frac{\sum_{k=1}^n (S_{\max} - S_k)^2}{N} + \frac{\sum_{k=1}^n (S_{\max} - S_k)}{N}},$$

$$\text{is } \frac{|S^* - S_i|}{S^*} < DPC, \quad i = 1, n; \quad DPC = 0.01, 0.02, 0.03, \dots$$

where

$n$  = number of stations

$S^* = \frac{\sum_{i=1}^n S_i}{n^*}$  where  $n^*$  is the minimum available number of stations, i.e.

$$n^* = \left\lceil \frac{\sum_{i=1}^n S_i}{CT} \right\rceil$$

The DPC (*Dynamic-Partitioning-Constant*) parameter enables us to fine-tune our algorithm. In other words, DPC adjusts the accuracy of the station freezing criteria. When it increases, the average number of partitionings per run also increases. Hence, we save more in computation time but we may end up with a poorer solution (i.e. worse final fitness scores) due to the freezing criteria.

As described above, the two criteria for DPA are checked at the end of each iteration. If the first or the last station satisfies the criteria, then that (those) station(s) is (are) frozen and the GA goes on to the next iteration with the unfrozen tasks only. Since the length of the chromosome decreases after each freezing (partitioning), the GA program spends less time per iteration for the remaining iterations.

The population size, i.e. the number of chromosomes in the GA population, stay fixed at the starting population size throughout each run, until the last iteration. At each iteration, one of the chromosomes in the population gives the best solution, thus the best fitness score. This chromosome is called the *best-fit chromosome*. After each iteration, the best-fit chromosome is checked if it satisfies the DPA criteria. If it does, DPA is applied to the best-fit chromosome and the frozen genes (tasks) are deduced from all the other chromosomes of the population. This does not create any infeasibility for the precedence constraints because the frozen tasks are either at the beginning or at the end of the partitioned chromosome. DPA mechanism is illustrated by means of an example in Figure 4.1. In this example, DPA criteria are satisfied for both the first and the last stations at the 45th iteration. Hence, tasks 1, 2, 13, 15, and 16 are frozen. Then, the GA balances the remaining eleven tasks, disregarding the frozen five. At the 136th iteration, only the first station satisfies the DPA criteria, and hence the tasks belonging to this station (i.e., tasks 7, 11) are frozen. The frozen tasks are then added on to the best-fit chromosome of the final iteration in the order that they were frozen.

During the early stage of the research, it is presumed that if we apply DPA starting with the first iteration then we might do some early freezing which would bind us to a local optima which is not the optimal solution. Therefore, we use a warm-up period that allows the initial random population to achieve a considerable fitness score before partitioning starts.

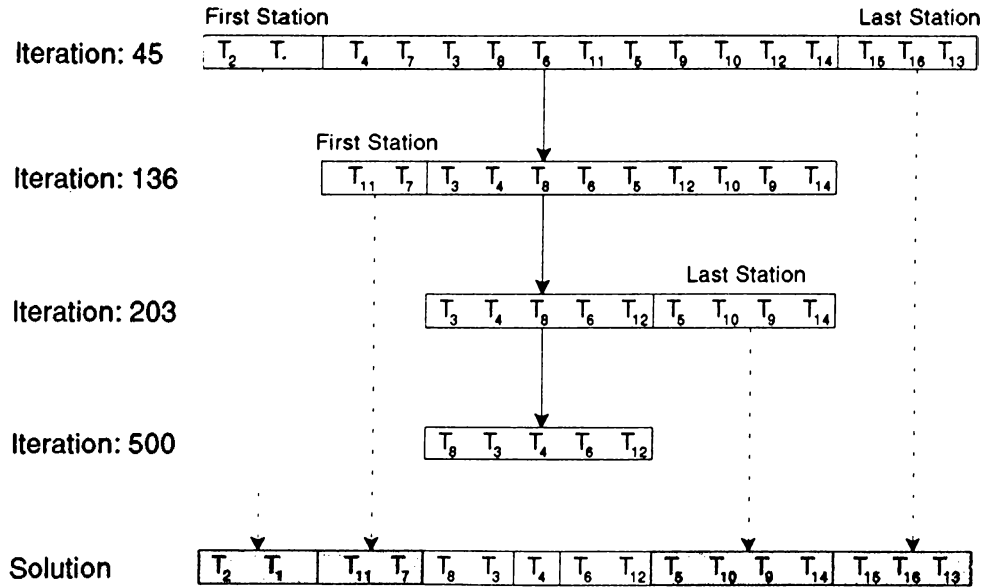


Figure 4.1: Illustration of Dynamic Partitioning

### 4.3 Experimentation

To investigate the utility of DPA, we solve 30 different ALB problems that are generated the same way as in previous studies in literature (i.e., Leu et al. (1994)). In addition, we measure the effects of different DPA and GA parameters.

Thirty problems which consist of 50 tasks are randomly generated in three sets, each set generated around a different F-Ratio. The first set is generated at approximately 10% F-Ratio, the second one at approximately 50%, and the third at approximately 90%. F-Ratio is a measure of the precedence relations among the tasks, that may take a value between 0 and 1, indicating how complicated the ALB problem is, according to the number of existing precedence relations compared to the total of available precedence relations in a problem.

The formula to calculate F-Ratio for an n-task problem is as follows:

$$F - Ratio = \frac{2 \times (\text{number of 1's in the precedence matrix})}{n(n - 1)}$$

where the precedence matrix is an upper triangular binary matrix with  $(i, j)$ th entry equals to one if task  $j$  is a follower of task  $i$  on the precedence diagram, zero otherwise.

The task times of all thirty problems are generated from the binomial distribution ( $n=30, p=0.25$ ). Zero duration task times are increased to one time unit. This choice has a foundation from the fact that this particular distribution models the task times of the actual ALB problems. These parameters are also the choices of Leu et al. [29] and Talbot et al. [40]. In fact, Talbot et al. [[40], pp 438-439] states that "an investigation of actual line balancing problems appearing in the open literature suggests [the above] parameters for generating task times." Note that this particular distribution is not symmetric; this is not surprising considering that the binomial is skewed in such a way as to give a few "long" task times, relative to other task times. Leu et al's (1994) comment on this choice is as follows: "Any choice may affect problem difficulty considerably. Although we do not claim generalizability of our results beyond the problems studied, we think that the effect of distribution and parameters would be more pronounced with conventional heuristics than with the GA approach developed here." Finally, we choose the cycle time as 56, which is approximately twice the average of the maximum task times of the thirty problems.

We examine four DPA and GA parameter settings, namely DPC, warm up period (WU), number of iterations (ITER), and population size (POPSIZE). DPC and warm up period are the two DPA parameters. Number of iterations and population size are the two GA parameters that are included in the analysis to see if they affect the results. "GA with DPA" and "GA without DPA" are the two abbreviations we use to refer to the GA program which does not use the dynamic partitioning function and the GA program which uses the dynamic partitioning function, respectively.

The first factor, DPC, has four levels: 0, 0.01, 0.02, and 0.03. When the DPC is at 0 level, the GA works as if without DPA. As we change the value of DPC from 0 to any other number (between 0 and 1) we turn on the DPA function, but as we switch between any two numbers other than zero, this affects the fine tuning of the DPA criteria, or the trade off between the computation time savings and the performance of the GA with DPA.

The second factor is the warm up period. This factor has four levels: 0, 25, 50, 75, and 500. DPA is applied with no warm up period at the 0 level. We use 500 as the DPA level to observe the effects of a very long warm-up period. Obviously, when the number of iterations factor is also at the 500 level, we observe no DPA, that is. this factor has no effect when DPC is at 0 level.

The third factor is the number of iterations. The following three levels are used: 500, 1000, and 2000. Finally, the fourth factor is the population size with four levels at 20, 30, 40, and 50. These two factors were initially expected to affect both the GA with DPA and the GA without DPA performances. Although mutation rate could also be selected as a parameter, we prefer to keep it fixed at a reasonable level (0.05) to save from additional computation time. According to our previous experiment, 0.05 level is the best mutation level among 0, 0.025, 0.05, 0.075, and 0.1 levels, for GA without DPA. Hence, we assume that it is a reasonable level for all levels of the DPC factor, and we do not observe the effect of mutation factor on DPA.

As we stated earlier, the 30 problems are generated at three F-Ratio levels (i.e., 10%, 50%, and 90%). We were planning to include F-Ratio as a fifth factor in this experimental design, but since the average fitness scores observed at different F-Ratio levels differ a lot, we do not include it as a factor in this design. Hence, we observe the effects of the other four factors at three F-Ratio levels separately from each other. However, we also perform another experiment afterwards, to observe the effect of F-Ratio factor on the fitness scores.

In the experiments, we take ten replications of each problem at each combination of factor levels, by using the same set of ten random seeds. Therefore,

we solve  $30$  (problems)  $\times$   $10$  (replications)  $\times$   $4$  (DPC levels)  $\times$   $5$  (warm-up levels)  $\times$   $3$  (iteration levels)  $\times$   $4$  (population size levels) =  $72000$  problems. The Anova test results in terms of the fitness score and the computation time are as follows:

### 4.3.1 ANOVA Results

As it can be observed in Table 4.1, all four factors, and some of their two and three way interactions are significant at the 5% level in 10% F-Ratio case. The warm-up period factor is not significant in 50% F-Ratio case. DPC and warm-up period factors are both insignificant in 90% F-Ratio case.

The Bonferroni and Duncan grouping of the fitness scores due to DPC is also reported in Table 4.2. We used Duncan as an alternative to Bonferroni which is a rather conservative method, but both methods gave the same groupings in all the experiments in this chapter. In general, DPA performs significantly better than the GA without DPA, (DPC at 0), at two levels of the DPC, 0.01 and 0.02, in 10% F-Ratio case. In 50% F-Ratio case, DPA is significantly better only for 0.01 level of the DPC. Additionally, the improvement of the average fitness score when DPC is at the optimum level compared to GA without DPA is 7.69%, which is lower than the improvement in 10% F-Ratio case, that is 16.43%. In 90% F-Ratio case, DPA is not significantly better than GA without DPA, but it is slightly better at all levels of the DPC. Other observations are as follows:

We observe that the performance improves significantly as the number of iterations increases, at all levels of DPC. This observation was expected since we used elitism, i.e. the fitness score is not allowed to get worse than the value obtained at a prior iteration. Exceptionally, there is no significant difference between the 500 and 1000 levels in the 90% F-Ratio case. This finding is not surprising because it is harder to improve the solution quality when the feasible set is small as in 90% F-Ratio case.

<i>Fitness Scores</i>					
Source	DF	Sum of Sq	F Value	Pr > F	Sig. at 0.05?
<i>F-Ratio = 10%</i>					
Model	167	14378.71	34.68	0.0001	yes
Error	23832	59159.74			
DPC	3	1966.58	264.07	0.0001	yes
ITER	2	9772.55	1968.39	0.0001	yes
WU	4	74.48	7.50	0.0001	yes
POPSIZE	3	793.68	106.58	0.0001	yes
ITER*DPC	6	663.74	44.56	0.0001	yes
DPC*WU	12	305.00	10.24	0.0001	yes
DPC*POPSIZE	9	88.10	3.94	0.0001	yes
ITER*WU	8	229.55	11.56	0.0001	yes
ITER*POPSIZE	6	283.70	19.05	0.0001	yes
WU*POPSIZE	12	18.37	0.62	0.8302	no
DPC*WU*POPSIZE	36	31.24	0.35	0.9999	no
ITER*DPC*WU	24	113.32	1.90	0.0049	yes
ITER*DPC*POPSIZE	18	30.41	0.68	0.8340	no
ITER*WU*POPSIZE	24	7.99	0.13	1.0000	no
<i>F-Ratio = 50%</i>					
Model	167	2809.20	6.50	0.0001	yes
Error	23832	61721.60			
DPC	3	761.25	97.98	0.0001	yes
ITER	2	1128.53	217.87	0.0001	yes
WU	4	7.02	0.68	0.6074	no
POPSIZE	3	165.46	21.30	0.0001	yes
ITER*DPC	6	130.21	8.38	0.0001	yes
DPC*WU	12	167.56	5.39	0.0001	yes
DPC*POPSIZE	9	65.94	2.83	0.0025	yes
ITER*WU	8	40.64	1.96	0.0471	yes
ITER*POPSIZE	6	183.56	11.81	0.0001	yes
WU*POPSIZE	12	42.11	1.35	0.1797	no
DPC*WU*POPSIZE	36	28.66	0.31	1.0000	no
ITER*DPC*WU	24	22.70	0.37	0.9981	no
ITER*DPC*POPSIZE	18	45.92	0.99	0.4735	no
ITER*WU*POPSIZE	24	19.63	0.32	0.9994	no
<i>F-Ratio = 90%</i>					
Model	167	4940.09	1.35	0.0001	yes
Error	23832	520551.62			
DPC	3	0.27	0.00	0.9996	no
ITER	2	1250.52	28.63	0.0001	yes
WU	4	0.36	0.00	1.0000	no
POPSIZE	3	3379.81	51.58	0.0001	yes
ITER*DPC	6	0.06	0.00	1.0000	no
DPC*WU	12	0.84	0.00	1.0000	no
DPC*POPSIZE	9	0.75	0.00	1.0000	no
ITER*WU	8	0.01	0.00	1.0000	no
ITER*POPSIZE	6	305.90	2.33	0.0296	yes
WU*POPSIZE	12	0.50	0.00	1.0000	no
DPC*WU*POPSIZE	36	0.64	0.00	1.0000	no
ITER*DPC*WU	24	0.02	0.00	1.0000	no
ITER*DPC*POPSIZE	18	0.36	0.00	1.0000	no
ITER*WU*POPSIZE	24	0.06	0.00	1.0000	no

Table 4.1: ANOVA results for fitness scores

<i>Bonferroni Grouping</i>	<i>Duncan Grouping</i>	<i>Mean</i>	<i>N</i>	<i>DPC</i>
10% F-Ratio				
A	A	3.733	6000	0.03
B	B	3.548	6000	0
C	C	3.323	6000	0.02
D	D	2.965	6000	0.01
50% F-Ratio				
A	A	3.743	6000	0.03
B	B	3.510	6000	0
B	B	3.480	6000	0.01
C	C	3.240	6000	0.02
90% F-Ratio				
A	A	10.623	6000	0
A	A	10.620	6000	0.03
A	A	10.618	6000	0.01
A	A	10.614	6000	0.02

Table 4.2: *Bonferroni and Duncan grouping of fitness scores due to DPC.*

There is not a significant performance difference between the levels of the warm-up period. Hence, we omit this factor in the further analysis.

In 10% F-Ratio case, 20 and 30 levels of the population size factor perform significantly better than 40 level, while there is no significant difference between 40 and 50 levels. Roughly, performance improves as the population size increases in 10% F-Ratio case, but the opposite relation is observed in 50% and 90% F-Ratio cases. In 50% F-Ratio case, 40 and 50 levels perform significantly better than the 20 and 30 levels. Similarly, in 90% F-Ratio case, 50 level performs significantly better than the 30 and 40 levels, while the worst performance is shown by the 20 level. Hence, we conclude that the optimum population size is inversely proportional with the number of all feasible solutions, in our problem.

Table 4.4 confirms our expectations that the CPU time savings would increase as the level of DPC is increased. But, as we note in Table 4.2, the performance of the algorithm decreases if we slacken our DPA criteria (i.e., increase DPC). The CPU time saving for DPC at 0.01 (i.e., at the optimum



<i>CPU Time</i>					
Source	DF	Sum of Sq	F Value	Pr > F	Sig. at 0.05?
<b>F-Ratio = 10%</b>					
Model	167	317886844.56	423.09	0.0001	yes
Error	23832	107222212.91			
DPC	3	39965797.158	2961.03	0.0001	yes
ITER	2	221475236.12	24613.36	0.0001	yes
WU	4	12618309.58	701.16	0.0001	yes
POPSIZE	3	14709022.85	1089.78	0.0001	yes
ITER*DPC	6	19417914.67	719.33	0.0001	yes
DPC*WU	12	4735933.37	87.72	0.0001	yes
DPC*POPSIZE	9	89602.23	2.21	0.0185	yes
ITER*WU	8	3341638.94	92.84	0.0001	yes
ITER*POPSIZE	6	172506.21	6.39	0.0001	yes
WU*POPSIZE	12	93323.25	1.73	0.0544	no
DPC*WU*POPSIZE	36	49551.79	0.31	1.0000	no
ITER*DPC*WU	24	1165622.78	10.79	0.0001	yes
ITER*DPC*POPSIZE	18	26556.49	0.33	0.9966	no
ITER*WU*POPSIZE	24	25829.11	0.24	1.0000	no
<b>F-Ratio = 50%</b>					
Model	167	338013613.13	883.86	0.0001	yes
Error	23832	54575204.98			
DPC	3	47259999.22	47974.24	0.0001	yes
ITER	2	219721716.08	47974.24	0.0001	yes
WU	4	22186867.76	2422.15	0.0001	yes
POPSIZE	3	10170984.90	1480.49	0.0001	yes
ITER*DPC	6	20175884.62	1468.41	0.0001	yes
DPC*WU	12	8385227.70	305.14	0.0001	yes
DPC*POPSIZE	9	37068.83	1.80	0.0631	no
ITER*WU	8	7002495.84	382.23	0.0001	yes
ITER*POPSIZE	6	57952.60	4.22	0.0003	yes
WU*POPSIZE	12	205032.25	7.46	0.0001	yes
DPC*WU*POPSIZE	36	106336.83	1.29	0.1144	no
ITER*DPC*WU	24	2554821.89	46.49	0.0001	yes
ITER*DPC*POPSIZE	18	45259.33	1.10	0.3464	no
ITER*WU*POPSIZE	24	103965.27	1.89	0.0053	yes
<b>F-Ratio = 90%</b>					
Model	167	472137987.51	3671.42	0.0001	yes
Error	23832	18351793.51			
DPC	3	1894134.57	819.92	0.0001	yes
ITER	2	459903043.76	99999.99	0.0001	yes
WU	4	654259.41	212.41	0.0001	yes
POPSIZE	3	7903184.28	3421.08	0.0001	yes
ITER*DPC	6	845397.93	182.98	0.0001	yes
DPC*WU	12	467938.31	50.64	0.0001	yes
DPC*POPSIZE	9	27598.34	3.98	0.0001	no
ITER*WU	8	229897.42	37.32	0.0001	yes
ITER*POPSIZE	6	6303.33	1.36	0.2246	yes
WU*POPSIZE	12	17151.22	1.86	0.0346	yes
DPC*WU*POPSIZE	36	12427.69	0.45	0.9982	no
ITER*DPC*WU	24	159493.27	8.63	0.0001	yes
ITER*DPC*POPSIZE	18	11582.80	0.84	0.6591	no
ITER*WU*POPSIZE	24	5572.18	0.30	0.9936	yes

Table 4.3: ANOVA results for CPU times

<i>Bonferroni Grouping</i>	<i>Duncan Grouping</i>	<i>Mean</i>	<i>N</i>	<i>DPC</i>
<b>10% F-Ratio</b>				
A	A	3.289	6000	0
B	B	2.605	6000	0.01
C	C	2.340	6000	0.02
D	D	2.241	6000	0.03
<b>50% F-Ratio</b>				
A	A	3.246	6000	0
B	B	2.510	6000	0.01
C	C	2.235	6000	0.02
D	D	2.096	6000	0.03
<b>90% F-Ratio</b>				
A	A	3.237	6000	0
B	B	3.167	6000	0.01
C	C	3.103	6000	0.02
D	D	2.996	6000	0.03

Table 4.4: *Bonferroni and Duncan grouping of CPU times due to DPC*

level) is 22.67% in 10% F-Ratio case, 20.80% in 50% F-Ratio case, and 2.16% in 90% F-Ratio case. The effects of other factors on CPU time are as follows: CPU time increases as the number of iterations or the population size or the warm-up period increases. The effect of each level of these factors differs significantly from each other (see Table 4.3).

After performing Anova tests for the three sets of problems having different F-Ratios, we combined the three sets of data and observed F-Ratio's effect on the overall. We observed that F-Ratio is a significant factor at the 0.05 level. The Bonferroni and Duncan grouping of the fitness scores due to F-Ratio is presented in Table 6.1. Although the task times of each set of problems are generated by the same generator, as the F-Ratio increases (i.e., the number of precedence relationships between the tasks increases), the fitness scores increase exponentially.

<i>Bonferroni Grouping</i>	<i>Duncan Grouping</i>	<i>Mean</i>	<i>N</i>	<i>F-Ratio</i>
A	A	10.619	24000	90%
B	B	3.493	24000	50%
C	C	3.392	24000	10%

Table 4.5: *Bonferroni and Duncan grouping of fitness scores due to F-Ratio*

## 4.4 Major Findings

The major motivation that led us to devise the DPA procedure was to achieve a significant amount of CPU time reduction. As presented above, we have achieved that objective. Even though we were expecting some deterioration in the performance of fitness scores due to DPA, fortunately we had an improvement. In other words, we have achieved a better performance with dynamic partitioning than the traditional application of GA without dynamic partitioning, while also saving from the CPU time. This counter-intuitive result can be explained as follows: The stations that we freeze by DPA already have station times that are very close to the optimal station time in order to minimize the fitness function, as explained earlier. Hence, by freezing some of the tasks without straying too much from optimal balancing, the GA concentrates more on the remaining tasks. If we did not freeze the stations that satisfy the DP criteria, the mutation and crossover mechanisms would waste time on working on these already balanced stations as well, instead of focusing on the poorly balanced tasks. Therefore, given the same number of iterations, a GA with DPA is able to work (try alternative combinations) on balancing the poorly balanced stations more than a GA without DPA. Consequently, we achieved a significantly better performance with DPA than the GA without DPA. Since the GA without DPA has many structural and operational similarities between Leu et al.'s (1994) GA (i.e., as stated in Chapter 3) and that our problem generation schemes are also very similar, we claim that the proposed GA (i.e., GA with DPA) is better than Leu et al.'s (1994) GA. We prove this by achieving a better solution on the Kilbridge-Wester [27] problem than Leu et al.'s (1994) solution on the same problem in Chapter 7.

Another interesting observation is that the improvement effect of DPA decreases as the F-Ratio increases, i.e. as the search space gets narrower. The reason is that the possibility of partitioning at the same level of DPC decreases due to the fact that the number of feasible solutions decreases because of the large number of precedence relationships. Besides, even if the GA is allowed to focus on the poorly balanced stations by DPA, it is less likely that this focus will lead to a better result since GA without DPA is already able to perform a sufficient search in a narrow search space.

Like every other random search algorithm, DPA has several tuning factors as well. A brief summary of our observations on these factors are as follows:

DPC seems to be the major factor affecting the performance of the proposed algorithm. DPC performs usually better at its nearest to 0 level (i.e., 0.01), but we observed in the problem set with 90% F-Ratio that DPC at 0.02 level gives better results on the average than DPC at 0.01 level. This suggests that DPC requires fine-tuning to achieve the best performance of DPA, however we shall note that the optimum value of this factor is likely to be a positive number that is very close to zero.

It can be noted in Table 4.2 and Table 4.4 that there is a payoff between the score and the CPU time as we change the value of DPC. In case (a), the improvement of the average fitness score is about 16% (8% in case (b)) while the CPU time saving is about 21% (23% in case (b)) when DPC is at 0.01. When DPC is at 0.02 level, the improvement of the fitness score in case (a) decreases to 6% (1% in case (b)) and the CPU time saving increases to 29% (31% in case (b)). However, we cannot observe this behavior in case (c). But 90% F-Ratio is a very high value and the possible number of solutions in an ALB problem with such a high F-Ratio is very small. Hence, it is not surprising that we do not observe a significant difference between the levels of the DPC (in Table 4.2).

Warm-up period is also effective but it is not possible to observe a linear effect of this factor on the score. Therefore, this factor should also be fine tuned to achieve a better performance. We also note that this factor needs a

different tuning at each level of the DPC. The number of iterations factor shows a linear effect, GA performing better at all levels of the DPC when this factor level increases. But the increase in performance of the algorithm is exponential, i.e. the increase in performance gets less as the number of iterations increases.

The population size factor can be tuned for obtaining the optimum performance of the algorithm as well. From the three sets of problems that have 10%, 50%, and 90% F-ratios, we observed that a larger population size yields to a better score on problems with higher F-Ratio (i.e. 50% and 90%). On the other hand, it may be found counter-intuitive, at the first sight, that smaller population sizes performed better than the larger ones on problems with 10% F-Ratio. Our reasoning for this observation stems from the fact that the search space is wider at low F-Ratios. Therefore, a large population cannot concentrate on local minimum search. The special recombination mechanism that is used in our GA is responsible for this finding, i.e. only one pair of chromosomes are selected for recombination at each iteration. Assuming that the selection of the best-fit chromosome as one of the parents is potentially more advantageous for local minimum search than recombining two other chromosomes, the performance is expected to decrease as the population size increases, in a wide search space, since the probability of selecting the best-fit chromosome for recombination in a large population is less than in a small population.

Finally, we observe that F-Ratio is a significant factor. This factor's increase causes an exponential increase in the fitness score. This observation was expected since the increase in the number of precedence relations reduces the allocation alternatives of the tasks, hence may increase the number of stations.

## Chapter 5

# ELITISM WITH SIMULATED ANNEALING

### 5.1 Introduction and Motivation

Simulated Annealing (SA) is a well known global search algorithm. In local search algorithms, we start with an initial solution, and a neighbor to the initial solution is selected from the set of feasible solutions. The difference between SA and the other local search algorithms is that in local search algorithms, we move to the neighbor point (or solution) only if the solution is improved. However, in SA, we can move to the neighbor solution even if it is not better. The probability of accepting such a transition is calculated by a function usually called as the acceptance function. We can calculate this probability as  $P(x) = \min(1, \exp(-\frac{\nabla c_{ij}}{T}))$  where  $\nabla c_{ij}$  is the change of cost between the solution on hand and the neighbor, and  $T$  is a control parameter that corresponds to temperature.

We note here that the "elitism rule" used in our algorithm basically resembles the "other local search algorithms" mentioned above. Hence, by using the characteristic of SA that differs from the other local search algorithms, we intend to widen the myopic view of elitism. In other words, strict elitism may be

a reason to be trapped in a local minimum, hence, our motivation to apply SA to elitism is to decrease the possibility of getting trapped in a local minimum in our search for the global minimum.

The SA algorithm usually starts with a relatively high value of  $T$ , to have a better chance to avoid being prematurely trapped in a local minimum. Then  $T$  is lowered in steps until it approaches zero. After termination, the final configuration is used as the solution of the problem. There are two different ways of decreasing the control parameter,  $T$ : i) inhomogeneous algorithm where  $T$  is decreased after each transition, and ii) homogeneous algorithm where  $T$  is decreased after a number of transitions,  $L$ . We use the first approach in our algorithm, because it provides a smooth trajectory for  $P(x)$ . (SA resembles the annealing process of metals, where the temperature decreases gradually, as in the first approach.) The following parameters and strategies are the parts of decision in the SA algorithm: initial temperature,  $T_0$ ; number of transitions required for decreasing  $T$ ,  $L_k$ ; temperature function,  $T_k$ ; and the stopping criteria. The problem specific decision elements are the initial solution, neighborhood generation, and evaluation of  $\nabla c_{ij}$ .

## 5.2 Integration of SA to Elitism

Obviously,  $\nabla c_{ij}$  is the difference between the fitness scores of the offspring and its parent. We accept the offspring if its fitness score is smaller than its parent's, since our objective is to minimize the fitness function. In case the offspring's fitness score is larger than its parent's, we calculate  $\nabla c_{ij}$ , and then evaluate the probability function,  $P(x)$ . We decrease the temperature at each iteration whether we evaluate  $P(x)$  or not.  $T$  is decreased exponentially with respect to  $T_{k+1} = T_k \times \alpha$ , where  $k$  is the iteration number, and  $\alpha$  is the scaling factor that is a positive number smaller than 1 and usually very close to 1, i.e. 0.98. Hence,  $T = T_0 \times \alpha^k$ , at the  $k$ th iteration. We decrease  $T$  until 0.01, at which  $P(x)$  takes a value smaller than 0.0001 at the 500th iteration, when  $\nabla c_{ij}$  takes a value greater than 0.1. We do not explicitly define a stopping criteria other

than the 0.01 limit for  $T$ , and that limit is to prevent exponential overflows that cause run-time errors in computers. We keep the iteration number fixed at 500 but  $P(x)$  starts to take values that are almost to zero after  $T$  reaches the 0.01 limit, hence this limit can be thought of as the stopping criteria of SA where strict elitism takes over again. According to this stopping criteria,  $P(x)$  reaches approximately zero at different iteration numbers due to different  $\alpha$  levels. For example,  $P(x)$  is approximately zero after the 50th iteration at 0.80  $\alpha$  level; and after the 220th iteration at 0.95  $\alpha$  level. In our experimental setup, we used 7 different levels of alpha, 0, 0.8, 0.95, 0.96, 0.97, 0.98, and 1. The level 0 means "strict elitism", i.e. no SA, and the level 1 means "no elitism" where our crossover mechanism (neighborhood generation mechanism) turns out to be a random search mechanism instead of a local search mechanism. The problem specific decision elements of SA are replaced by GA decision elements in our application. Whereas the initial solution is the best-fit chromosome of the initial population, neighborhood generation is simply the crossover and mutation mechanisms, and evaluation of  $\nabla c_{ij}$  is the difference between the fitness scores of the offspring and its parent.

Other decision to be made in SA is the movement policy. There are three policies, i.e. first wins, best wins, and random wins. We use the first policy. Specifically, we generate a neighbor by crossover and mutation mechanisms and decide whether to replace the current solution with that neighbor. The best wins policy is rather more time consuming, because all possible permutations of a simple neighbor generator are tried, and the ones that are randomly selected with the probability generated by the probability function are considered as neighbors. If the number of all possible permutations is too large, then a number of these can be tried instead of all. Among these neighbors, the best neighbor replaces the existing solution. Random wins is quite similar to best wins, but among the neighbors, one of them is chosen randomly instead of the best one. First wins policy is rather considered as a myopic mechanism but it saves time compared to best wins or random wins. Best wins policy is actually very similar to the strict elitism policy, used previously, because as long as there is a better solution among the neighbors at a certain iteration, the solution will



always be replaced by a better neighbor. The random wins policy prevents this continuous improvement policy, that has the risk of getting trapped in a local minimum, by choosing randomly among the neighbors. But this random choice may also be similar to the roulette wheel selection that favors better neighbors. Since our neighborhood generator is a random generator, the first wins policy is not a myopic policy at all in our algorithm. If our neighbor generator was deterministic then we would try the best wins or the random wins policies as well. Suresh et al. (1996) also use the first wins policy and their neighbor generator is also a random generator, i.e. randomly shifting a job from one workstation to another or interchanging the positions of the two jobs. A summary of the steps of our algorithm, that are adapted from Vidal [44], is as follows:

**Step 1.** Select the best-fit chromosome of the initial population as the initial solution,  $\varphi_0$ , and the starting temperature,  $T_0$ , as 1000. Set  $\varphi_{\min} = \varphi_0$ .

**Step 2.** Evaluate the cost function,  $C(\varphi_{\min}) : S \rightarrow R$ , where  $S$  is the search space.

**Step 3.** Select a neighbor by crossover/mutation,  $\bar{\varphi} \in S$ .

**Step 4.** If  $\nabla = C(\bar{\varphi}) - C(\varphi) < 0$  then  $\varphi_{r+1} = \bar{\varphi}$ , otherwise  $\varphi_{r+1} = \bar{\varphi}$  with probability  $p = e^{-\frac{\nabla}{T_r}}$ . If  $C(\varphi_{r+1}) < C(\varphi_{\min})$  then  $\varphi_{\min} = \varphi_{r+1}$ .

**Step 5.** Set  $r=r+1$  and evaluate, if  $T_r > 0.01$  then  $T_{r+1} = T_r \times \alpha$ , otherwise  $T_{r+1} = T_r$ .

**Step 6.** If  $T_r < T_{mem}$  then  $T_{mem} = T_r$ . (The best solution is kept in the memory since the solution at the last iteration may not be the best.)

**Step 7.** If  $k \leq$  maximum iteration number then stop, the solution is  $T_{mem}$ ; otherwise go to Step 2.

### 5.3 Experimentation

Our experimental setup consists of the same 30 problems that were also used in the previous chapter, i.e., dynamic partitioning, and the same 10 random number generator seeds and 4 population size factors, i.e. 20, 30, 40, 50. In addition, 7 alpha levels are used, i.e. 0, 0.80, 0.95, 0.96, 0.97, 0.98, 1. This makes a total of  $30 \times 10 \times 4 \times 7 = 8400$  problems.

The Anova results of this experimental design is given in Table 5.1. We observe that alpha levels are significantly different at 0.05 level. We grouped the factors by Bonferroni and Duncan methods and the results are presented in Table 5.2. According to Bonferroni, that is a conservative test, the alpha levels are not significantly different from each other, but Duncan test groups 1 and 0.98 levels separately from the other levels. Thus, we conclude that elitism is better than no elitism according to Duncan grouping. We then increase the number of iterations factor to 1000 to see if we could observe a significant difference in the Bonferroni grouping as well. We observed again that the Bonferroni grouping of the alpha levels do not show any significant difference, but the Duncan test groups 0 and 0.8 levels together as the best, 0.95, 0.96, and 0.97 levels as the second best, 0.98 level as the third best, and 1 level as the worst performing group. Since the performance increases as the level of elitism is increased, our previous conclusion that elitism is better than no elitism is confirmed. The Anova results and the Bonferroni and Duncan grouping of the factors at 1000 number of iterations level are also presented in Tables 5.1 and 5.2. Even though we change the initial temperature from 1000 to 10,000,000 in order to avoid early cooling, the combined effect of alpha on population size is insignificant.

Later, we enlarged our experimental design by including three different DPC levels, i.e. 0.01, 0.02, and 0.03, in addition to the other factors. In this case, we observe that the combined effect of alpha and DPC is significant at 0.05 level, but overlapping of the levels of alpha is observed in the Bonferroni groupings.

<i>Fitness Scores</i>					
Source	DF	Sum of Squares	F Value	Pr > F	Significant at 0.05?
<i>Number of Iterations = 500</i>					
Model	41	64954.54	171.63	0.0001	yes
Error	8358	77151.19			
ALPHA	6	801.64	14.47	0.0001	yes
F-RATIO	2	63714.08	3421.91	0.0001	yes
POPSIZE	3	90.48	3.27	0.0204	yes
F-RATIO*ALPHA	12	801.65	7.24	0.0001	yes
POPSIZE*ALPHA	18	86.68	0.52	0.9499	no
<i>Number of Iterations = 1000</i>					
Model	41	72736.44	220.21	0.0001	yes
Error	8358	67334.30			
ALPHA	6	1108.85	22.94	0.0001	yes
F-RATIO	2	70000.85	4344.50	0.0001	yes
POPSIZE	3	273.72	11.33	0.0001	yes
F-RATIO*ALPHA	12	1292.04	13.36	0.0001	yes
POPSIZE*ALPHA	18	60.97	0.42	0.9844	no

Table 5.1: ANOVA results for fitness scores

Although we could not achieve any significant improvement by relaxing the elitism rule, we observe that strict elitism ( $\alpha = 0$ ) is significantly better than no elitism ( $\alpha = 1$ ). Considering that our reproduction mechanism is a special one which is different from the classical approach, i.e. only one or two chromosomes are replaced by new offsprings, we claim that elitism should be used in order to obtain a better performance with this kind of a reproduction mechanism. Leu et al. (1995) used elitism without testing if it's better than no elitism or not, but our research makes it clear that elitism is significantly better than no elitism. Our further research will reveal if elitism yields a better performance in classical reproduction mechanisms, as well as if SA in elitism is able to contribute to the performance. We will also evaluate the performance difference between the classical reproduction mechanism compared to our reproduction mechanism in the next chapter.

<i>Bonferroni</i>	<i>Duncan</i>	<i>Mean</i>	<i>N</i>		<i>Bonferroni</i>	<i>Duncan</i>	<i>Mean</i>	<i>N</i>	
<i>Grouping</i>	<i>Grouping</i>			$\alpha$	<i>Grouping</i>	<i>Grouping</i>			$\alpha$
Iter = 500					Iter = 1000				
A	A	7.368	1200	1	A	A	6.925	1200	1
B A	B	7.018	1200	0.98	B A	B	6.608	1200	0.98
B C	C	6.712	1200	0.97	B C	C	6.315	1200	0.97
C	C	6.596	1200	0.95	D C	C	6.209	1200	0.96
C	C	6.585	1200	0.96	D C E	C	6.123	1200	0.95
C	C	6.494	1200	0.8	D E	D	5.876	1200	0
C	C	6.440	1200	0	E	D	5.825	1200	0.8
F-Ratio					F-Ratio				
A	A	10.609	2800	90%	A	A	10.351	2800	90%
B	B	5.091	2800	10%	B	B	4.271	2800	50%
C	C	4.534	2800	50%	B	B	4.184	2800	10%
Pop. size					Pop. size				
A	A	6.900	2100	20	A	A	6.554	2100	20
B A	B A	6.762	2100	30	B	B	6.287	2100	30
B A	B	6.700	2100	50	B	C B	6.153	2100	50
B	B	6.616	2100	40	B	C	6.080	2100	40

Table 5.2: *Bonferroni and Duncan grouping of fitness scores due to  $\alpha$ , F-Ratio, population size*

## Chapter 6

# CLASSICAL GA vs MODERN GA

In this chapter we propose a new classical algorithm, as an alternative to the modern or GENITOR type structure that was used in the previous chapters. The basic structure of both the classical and modern approaches to GAs have been already explained in Chapter 3. We will provide a more extensive review and comparison of these two approaches in this chapter.

All the genetic characteristics, such as coding, crossover operator, mutation operator, and selection policy are the same for both algorithms. These characteristics are explained in Chapter 3, hence it will not be repeated here. The difference between these two algorithms is due to the sequence in which the genetic operators are activated. In the classical GA, a proportion of the population which is the best group of chromosomes in the current population is transferred to the next population. The remaining individuals of the next population are formed by the offsprings that are generated by necessary number of crossover operations between the chromosomes of the current population.

## 6.1 Motivation

While the modern approach performs only one recombination operation at any iteration, the classical approach involves a large part of the population in the recombination process, performing multiple recombinations. Hence, the difference between the two consecutive populations of classical GA is expected to be more than in modern GA. Even though the modern GA seems to perform a rather myopic search than the classical GA at the first sight, it is an issue to be resolved whether the time spent on performing many recombinations could be used more effectively by increasing the number of iterations while performing only one recombination at each iteration. However, the performance of the algorithm is more important than the time it takes for the ALB problem unless this time can be kept in reasonable limits. Thus, we focus on the performance of the algorithms while comparing them.

We let both algorithms perform the same number of iterations, hence, the classical GA performs more recombination operations than the modern GA. Does performing more recombination operations make a GA more advantageous? The answer to this question lies beneath the selection policy and the convergence of the population issues. The fundamental theorem of genetic algorithms state that the qualities (genes) of the superior individuals (chromosomes) are preserved over generations while the qualities of the weak individuals evade. A corollary to this theorem is that if the more-than-average-fit chromosomes of a population gets involved in as many crossovers as possible, then its superior qualities are more likely to be distributed over the population. In the classical GA, the offspring that is generated by the first crossover operation in an iteration is likely to be better than the average of the current population, since its parents are likely to be over the average because of roulette wheel selection and it cannot be worse fit than its parents because of elitism. But it has to wait until all crossover operations are completed to be able to get selected. On the other hand, a superior offspring that results from a crossover operation is eligible to be selected for the next crossover operation in modern GA. Since every crossover operation considers the offspring that resulted from

the previous crossover operation a more efficient selection policy is applied in modern GA. This is an advantage of the modern GA compared to the classical approach. The advantage of the classical GA is that it allows more diversity and sharing of the good qualities of chromosomes at each iteration.

Elitism is a very important factor which may affect the diversity and hence the rate of convergence of the algorithm. We expected that this factor would be even more effective on classical GA since a greater number of crossover operations takes place in classical GA than in the modern GA. We have presented explicitly how the elitism rule is integrated to the structure of modern GA in Algorithm 3.4, and now we present the structure of the classical GA with DPA and with SA controlled elitism as follows:

**Algorithm 6.1** : *Classical GA with DPA and SA controlled elitism*

*Generate initial population*

*Set  $P_e$  equals to 1*

**repeat**

*Transfer the best  $(1 - R_x) \times N_p \times 100$  chromosomes to the next population*

**repeat**

*Choose two parents from the current population for crossover*

*Apply the following to each offspring*

**if** *the offspring is better than its parent* **then**

*Include the offspring in the next population*

**else** *Include the parent in the next population with  $1 - P_e$  probability*

*or Include the offspring in the next population with  $P_e$  probability*

**until** *Next population is full*

**if** *the DPA criteria are satisfied* **then**

*Freeze a set of tasks (genes)*

*Deduce the frozen rasks from all the chromosomes in the population*

*Apply mutation to one of the chromosomes with  $R_m$  probability*

*Decrease the value of  $P_e$*

*until Stopping-condition is reached*

*Take the best-fit chromosome of the final population as the solution*

## 6.2 Experimentation

We choose the same set of 30 problems that was used in Chapter 4 to compare the performance of the algorithms. We first optimize the parameters of each algorithm. The key parameters of the modern GA are the cooling rate, DPC, mutation rate, and population size. Each of these parameters are explained in Chapter 4 but we remind them as follows: The cooling rate parameter belongs to the simulated annealing terminology and it is used for controlling the level of elitism. DPC is the dynamic partitioning constant which adjusts the level of accuracy of freezing stations due to dynamic partitioning criteria. Mutation rate determines the level of probability of activating the mutation operator. Finally, population size is the number of chromosomes in the population. The warm-up period factor which was introduced in Chapter 4 is not taken into consideration because it was observed in Chapter 4 that it is not significant at the 0.05 level. Hence, the level of warm-up period factor is taken as 0, i.e., no warm-up period is used in any experiments in this chapter.

All the parameters that are used for the modern GA apply to the classical GA as well, additionally another factor is used for classical GA: crossover rate. In the modern GA, the mutation rate parameter subtracted from 1 gives the crossover rate, hence we did not need to define another parameter to set the crossover rate. Since the mutation and crossover operations occur independently in classical GA, we use the crossover rate parameter as well as the mutation rate parameter.

The levels experimented for the cooling rate parameter are 0, 0.95, 0.97, 0.99, and 1. The initial temperature associated with 500 iterations is 1000. The levels of DPC are 0, 0.01, 0.02, 0.03. The levels used for the mutation rate parameter are 0.01, 0.025, 0.05, 0.075, 0.1, 0.125, 0.15, 0.175, 0.2. We



	Classical GA	Modern GA
Cooling rate	*	*
DPC	*	0.03
Population size	50	50
Mutation rate	0.175	0.05
Crossover rate	0.9	N/A

Table 6.1: *Optimum parameters for Classical GA and Modern GA*

used 0.1, 0.25, 0.5, 0.75, 0.9 levels for the crossover rate parameter of classical GA, and 20, 30, 40, and 50 levels for the population size factor. The optimum parameters for both algorithms are as in Table 6.1.

The '\*' sign in Table 6.1 indicates that there is no significant difference between the levels of the corresponding parameter. Cooling rate is one such parameter and it is not significant in both the classical GA and modern GA. An interesting observation is that DPC is not effective with classical GA. We took 0 levels of the '\*' marked factors when comparing the two algorithms. We use the same 10 seeds that was used in Chapter 4. Hence, our experimental design is consisted of 30 problems, 10 seeds, and 2 different solution approaches.

The Anova results of the comparison experiment is given in Table 6.2, and the Bonferroni and Duncan groupings in Table 6.3. Results show that the classic GA is clearly superior to the modern GA, if we disregard the CPU times spent by each algorithm. However, there's a large difference between the CPU time of the algorithms. The classical GA takes approximately 22 times as much time as the modern GA. This difference in time is because of the large number of crossovers at each iteration in classical GA. On the other hand, as we said at the beginning of this chapter, we are concerned about improving the performance of the algorithm although we may have to spend more time because of two reasons: i) while the average CPU time for classical GA is about 44 seconds which is still not a very long time, and ii) ALB is a problem to be solved only once before the assembly line is being designed, hence time is not a measure which is as important as the performance of the solution to this problem. We improve the average solution of 30 problems solved with 10 different seeds from 6.372 to 5.444, as presented in Table 6.2,

<i>Fitness Scores</i>					
Source	DF	Sum of Squares	F Value	Pr > F	Significant at 0.05?
<i>Number of Iterations = 500</i>					
Model	1	129.99	9.46	0.0022	yes
Error	598	8213.20			
ALGORITHM	1	129.99	9.46	0.0022	yes

Table 6.2: ANOVA results for the comparison of two algorithms

<i>Bonferroni Grouping</i>	<i>Duncan Grouping</i>	<i>Mean</i>	<i>N</i>	<i>Algorithm</i>
A	A	6.372	300	MODERN
B	B	5.441	300	CLASSIC

Table 6.3: Bonferroni and Duncan grouping of fitness scores due to algorithm.

which is a significant improvement. Hence, we recommend the classical GA for the solution of ALB problems unless it will not be used in a very flexible manufacturing system that needs to be balanced frequently.

## Chapter 7

# COMPARISON WITH TRADITIONAL HEURISTICS

First, we compare genetic algorithms with traditional heuristics in general. Then we compare the proposed GAs' performance with Leu et al.'s (1994) GA on the Kilbridge-Wester's (1961, [27]) 45-task ALB problem and with Baybars' (1986) heuristic and other heuristics on Tonge's (1961, [42]) 70-task problem in this chapter.

### 7.1 Genetic Algorithms versus Traditional Heuristics

The central theme of research on genetic algorithms has been *robustness*, the balance between efficiency and efficacy necessary for survival in many different environments [15]. The implications of robustness for search schemes are manifold. If search schemes can be more robust, costly redesigns can be reduced or eliminated. Genetic algorithms are theoretically and empirically proven to provide robust search in complex spaces. The primary monograph on the topic is Holland's (1975) *Adaptation in Natural and Artificial Systems* [21]. We will

discuss the robustness of traditional optimization and search methods, then list the properties of genetic algorithms that enable them to surpass the traditional heuristics in the quest for robustness in the rest of this section.

The current literature identifies three main types of search methods: calculus based, enumerative, and random. We will examine each type to see what conclusions may be drawn without formal testing.

Calculus-based methods have been studied heavily. These subdivide into two main classes: indirect and direct. Indirect methods seek local optima by solving the usually nonlinear set of equations resulting from setting the gradient of the objective function equal to zero. Given a smooth, unconstrained function, finding a possible peak starts by restricting search to those points with slopes of zero in all directions. On the other hand, direct (search) methods seek local optima by hopping on the function and moving in a direction related to the local gradient. While both of these calculus-based methods have been improved and extended, some simple reasoning shows their lack of robustness. First, both methods are local in scope; the optima they seek are the best in a neighborhood of the current point. Clearly, starting the search in the neighbor of a low peak will cause us to miss the highest peak. Furthermore, once the lower peak is reached, further improvement must be sought through random restart or other trickery. Second, calculus-based methods depend upon the existence of derivatives (well-defined slope values). Hence, these methods that depend upon the restrictive requirements of continuity and derivative existence are unsuitable for all but a very limited problem domain. For this reason and because their inherently local scope of search, we must reject calculus-based methods.

Enumerated schemes have been considered in many shapes and sizes. The idea is fairly straightforward. Within a finite search space, or a discretized infinite search space, the algorithm starts looking at objective function values at every point in the space, one at a time. Although the simplicity of this type of algorithm is attractive, and enumeration is a very human kind of search (when the number of possibilities is small), such schemes must ultimately be

discounted in the robustness race for one simple reason: lack of efficiency. Many practical spaces are simply too large to search one at a time and still have a chance of using the information to some practical end. Even the highly praised enumerative scheme dynamic programming breaks down on problems of moderate size and complexity, suffering from a malady labeled as "the curse of dimensionality" by its creator (Bellman, 1961, [5]).

Random search algorithms have achieved increasing popularity as researchers have recognized the shortcomings of calculus-based and enumerative schemes. Yet, random walks and random schemes that search and save the best must also be discounted because of the efficiency requirement. Random searches, in the long run, can be expected to do no better than enumerative schemes. Having said that we should discount strictly random search methods, we must be careful to separate them from randomized techniques. The genetic algorithm is an example of a search procedure that uses random choice as a tool to guide a highly exploitative search through a coding of a parameter space. Another popular search technique, simulated annealing, uses random processes to help guide its form of search for minimal energy states. The important thing to recognize at this juncture is that randomized search does not necessarily imply directionless search.

While we conclude that conventional search methods are not robust, this does not imply that they are not useful. The schemes mentioned and countless hybrid combinations and permutations have been used successfully in many applications, however, as more complex problems are attacked, other methods will be necessary. And it would be worthwhile sacrificing peak performance on a particular problem, provided by a conventional search method, to achieve a relatively high level of performance across a spectrum of problems, that can be provided by a robust scheme like genetic algorithms.

In order for genetic algorithms to surpass the traditional optimization and search methods in the quest for robustness, GAs must differ in some very fundamental ways. Genetic algorithms are different from normal optimization and search procedures in four ways:

1. GAs work with a coding of the parameter set, not the parameters themselves. Hence, they are largely unconstrained by the limitations of other methods (continuity, derivative existence, unimodality, and so on).
2. GAs search from a population of points, not a single point. Since they work from a rich database of points simultaneously, climbing many peaks in parallel, the probability of finding a false peak is reduced over methods that move from a single point in the decision space to the next by using some transition rule to determine the next point.
3. GAs use payoff (objective function) information, not derivatives or other auxiliary knowledge. Many search techniques require much auxiliary information in order to work properly. For example, gradient techniques need derivatives and other search procedures like the greedy techniques of combinatorial optimization require access to most if not all tabular parameters. By contrast, GAs have no need for all this auxiliary information, i.e., GAs are blind. To perform an effective search for better structures, they only require payoff (objective function) values associated with individual strings. This characteristic makes a GA a more canonical method than many search schemes.
4. GAs use probabilistic transition rules, not deterministic rules. The use of probability does not suggest that the method is some simple random search like decision making at the toss of a coin. Genetic algorithms use random choice as a tool to guide a search towards regions of the search space with likely improvement.

Taken together, these four differences contribute to a GA's robustness and resulting advantage over other techniques.

## 7.2 Comparison with Leu et al.'s GA (1994)

Leu et al. solved Kilbridge-Wester's (1961) problem by their GA and compared it with five other heuristics that are also available in the QS software package

Heuristic No	Primary Heuristic	Heuristic Used to Break Ties
1	Maximum task time	Maximum total number of follower tasks
2	Maximum total number of follower tasks	Maximum task time
3	Minimum number of immediate-follower tasks	Minimum total number of follower tasks
4	Maximum number of immediate-follower tasks	Random task assignment
5	Maximum task time	Minimum total number of follower tasks

Table 7.1: *The heuristic methods to solve the Kilbridge-Wester problem*

[8]. These five non-GA heuristics use a single pass heuristic that is accompanied by another heuristic to break ties, as shown in Table 7.1. The heuristics that are used as primary or as tie-breaking are selected as the best performing heuristics due to Leu et al.'s (1994) literature survey [29]. Figure 7.1 presents the Kilbridge-Wester (1961) problem and also shows the solution of Leu et al. (1994), which is superior to the solutions of the other five heuristics.

The cycle time of the original problem is 55, but Leu et al. (1994) slightly changes this value to 56 to observe the sensitivity of non-GA heuristics to changes in problem's constraints. Leu et al. (1994) compare the heuristics with their GA by means of four different measures: i) mean-squared idle time, ii) square root of mean squared idle time, iii) efficiency (utilization), and iv) maximum station time. If the maximum station time is less than the given cycle time then it becomes the new cycle time, i.e., the cycle time is reduced. Hence, it is desirable to minimize the maximum task time in a type-1 ALB problem if the number of stations is already minimized (i.e., utilization is maximized). The other three measures are already explained in Chapter 2. We evaluated the performance of our GAs in terms of these measures as well. We present our results in comparison with Leu et al.'s (1994) and other heuristics' in Table 7.2.

We solved the Kilbridge-Wester problem by both the modern GA and the classical GA. We solved the problem using 1200 different factor level combinations. The factors used and their levels are: DPC (0, 0.01, 0.03, 0.05),

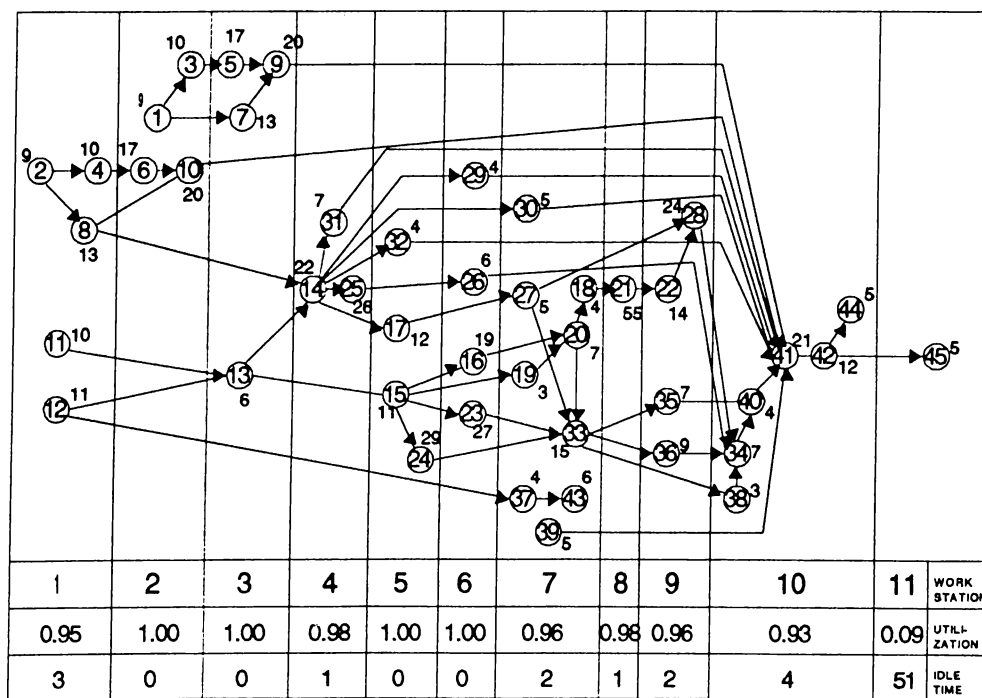


Figure 7.1: The Max-Task Time Heuristic Solution to the Kilbridge-Wester 45-Task Problem

population size (20, 50), cooling rate (0, 0.95, 0.97, 0.99, 1), mutation rate (0.02, 0.05, 0.1), and 10 random seeds. The same experimental design is applied for both the classical GA and the modern GA. The other factors that we used only at one level are as follows: number of iterations = 500, warm-up period = 0, crossover rate = 0.90.

As shown in Table 7.2, our modern GA found the optimum number of stations, leaving Leu et al.'s GA (1994) and other heuristics behind. The solution that provides the optimum number of stations was found at 13 different factor level combinations. These factor levels are presented in Table 7.3 to demonstrate the significantly positive effect of dynamic partitioning and our modification of the elitism rule by applying SA methodology to the performance of GAs. It can also be observed in Table 7.3 that the optimum number of stations (i.e., 10) could be found by modern GA only when dynamic partitioning was



Solution Method	Mean-Squared Idle Time	Sqr. Root (Mean-Sqrd Idle Time)	Efficiency	Maximum Workload
Heuristic 1	239.64	15.48	0.8961	56
Heuristic 2	239.27	15.47	0.8961	56
Heuristic 3	67.45	8.21	0.8961	56
Heuristic 4	124.91	11.17	0.8961	56
Heuristic 5	239.64	15.48	0.8961	56
Leu et al.'s GA	51.81	7.20	0.8961	55
Modern GA	1.20	1.10	0.9855	56
Classical GA	38.73	6.22	0.8961	55

Table 7.2: Comparison of non-GA heuristics, Leu et al.'s GA and the proposed GA

No	DPC	Population Size	Random Seed	Cooling Rate	Mutation Rate	CPU Time	Mean Sqrd Idle Time
1	0.03	20	14567	0.97	0.05	0.76	1.40
2	0.03	20	97665	0.99	0.02	0.76	1.40
3	0.03	20	97665	0.99	0.05	0.82	1.20
4	0.03	20	77943	0.99	0.10	1.04	1.40
5	0.03	20	47729	0.99	0.10	0.93	1.40
6	0.03	20	77943	1.00	0.10	1.04	1.40
7	0.03	50	84521	0.97	0.10	1.37	1.20
8	0.03	50	76421	0.97	0.10	1.59	1.20
9	0.03	50	60013	0.99	0.10	1.70	1.40
10	0.03	50	14567	1.00	0.02	1.53	1.40
11	0.05	20	14567	0.97	0.05	0.76	1.40
12	0.05	20	77943	0.99	0.10	1.04	1.40
13	0.05	20	47729	0.99	0.10	0.93	1.40

Table 7.3: Factor levels at which the optimum solution is found

activated (i.e.,  $DPC \neq 0$ ) together with SA in elitism (i.e., cooling rate  $\neq 0$ ). The two different mean squared idle time measures in Table 7.3 indicate that there are two alternative solutions with 10 stations. The CPU times in seconds is also presented in Table 7.3. The difference in CPU times are because of the difference in the iteration number when the algorithm starts dynamic partitioning. For example, if it starts partitioning early then the CPU time will be reduced due to the reduced chromosome size. We present the solution with the lowest mean squared idle time (i.e., 1.20), which gives the best balanced assembly line, in Figure 7.2.

Although the classical GA does not find the optimum number of stations at the given factor levels, it performs better than Leu et al.'s GA (1994) and

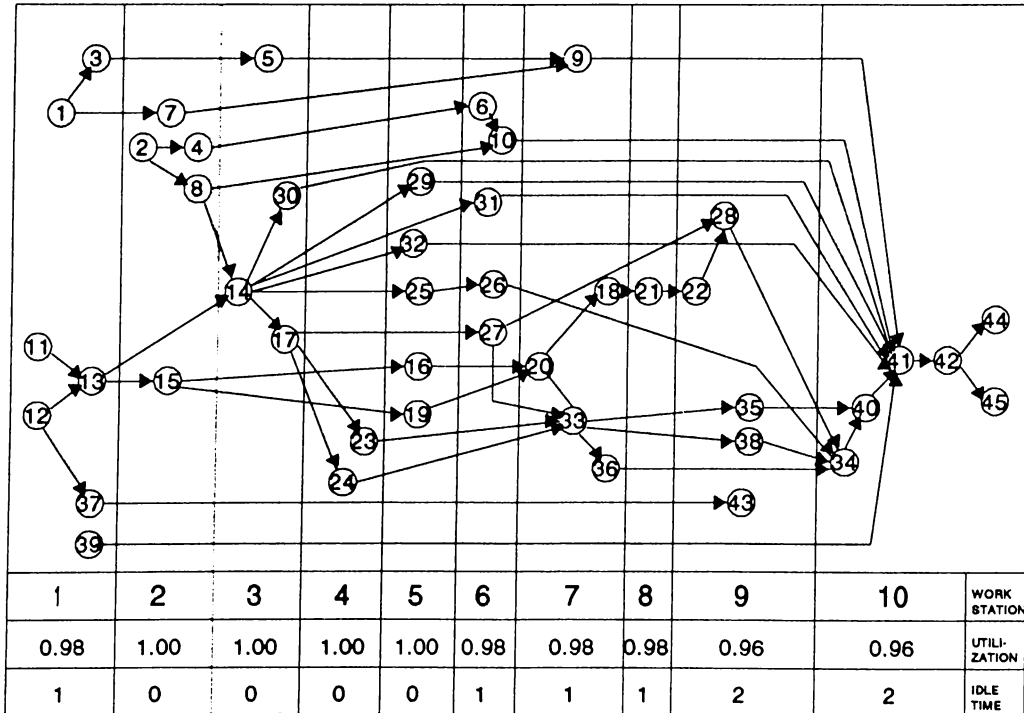


Figure 7.2: Modern GA Solution for the Kilbridge-Wester 45-Task Problem

the other five non-GA heuristics due to mean squared idle time measure, as can be seen in Table 7.2.

The classical GA performs worse than the modern GA in the Kilbridge-Wester problem. Contrarily, we observe that the classical GA performs better than the modern GA on the average in Chapter 6. This contradiction can be explained by the following facts: (i) the modern GA is better than the classical GA in 35 of the 300 problems solved hence, the classical GA is not always better than the modern GA, (ii) 26 of these 35 problems are generated between 10% and 50% F-Ratio levels, whereas the Kilbridge-Wester problem is also between these levels with its 39.70% F-Ratio, (iii) if we used only the optimum level of the cooling rate factor (i.e., zero) as in Chapter 6, we would not be able to find the optimum solution of this problem by the modern GA because all the optimum solutions are found at cooling rate levels that are not zero.

### 7.3 Comparison with Baybars' LBHA-1 (1986)

Baybars solved Tonge's (1961) 70-task problem with a heuristic called LBHA-1. We present Tonge's (1961) problem in Figure 7.3. This problem is a real life application that comes from the electronics industry. Since 1965, numerous attempts have been made to solve the Tonge (1961) problem for 13 different cycle times. The cycle time ranges from 83 to 364 in these versions, hence some versions contain tasks that have larger task times than the cycle time. For those versions, parallel stations are needed. For example, 15 parallel stations are needed for the first version where the cycle time is 83. In this version, task 13, that has a task time of 134 units, needs one parallel station and its task time is revised as  $134 - 83 = 51$ , after the parallel station has been used. The cycle times of the other versions are given in Table 7.4.

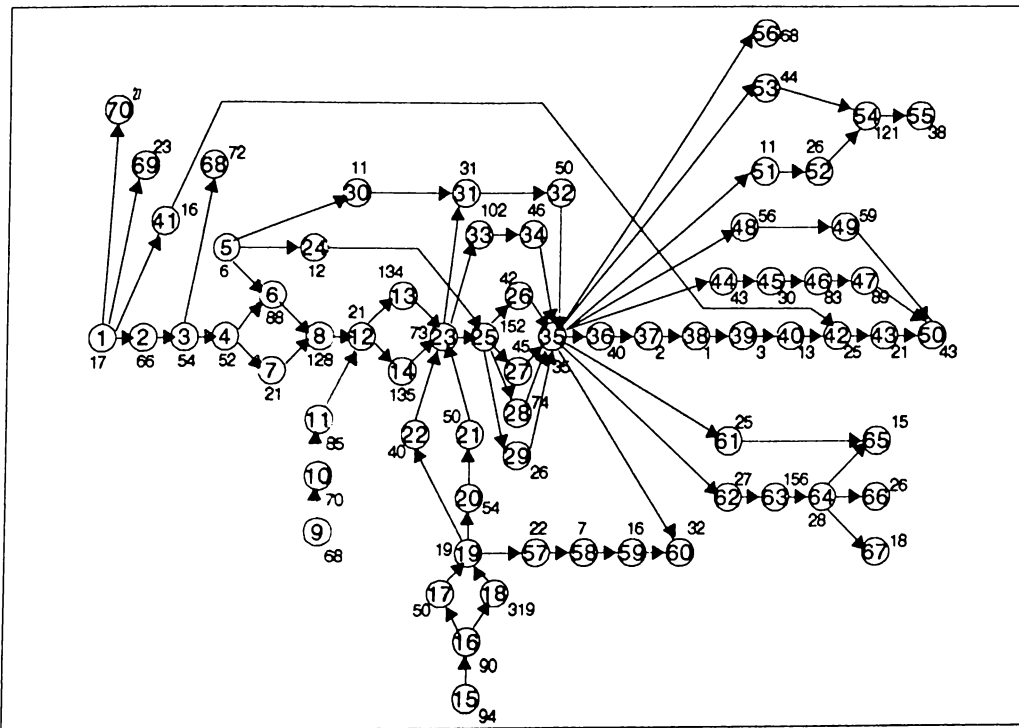


Figure 7.3: The 70-Task Problem of Tonge (1961)

We solve the 13 versions of Tonge's (1961) problem with both the modern

GA and the classical GA. Our algorithms have five parameters, i.e., DPC, number of iterations, cooling rate, mutation rate, and population size, that need to be optimized for each problem. First, we experimented the effect of each parameter on the performance for the first version of Tonge's (1961) problem. Because of the similarity of the versions (i.e., all precedence relations are the same but some of the task times change due to parallel stations), we eliminated some levels of some of the factors due to our experimental results on the first version. We fixed the number of iterations factor to the 500 level because we did not observe any significant improvement in higher levels. We observed that the best performing level of DPC is 0.05, hence we eliminated all other levels except the 0 level, which we kept to observe without DPA performance. The mutation parameter's levels are 0.01, 0.03, 0.05, 0.10, 0.20, and the cooling rate parameter takes 0, 0.95, 0.97, 0.99, 1 values for all versions. Additionally, we use the same 10 seeds that we used in the previous chapters. Hence, we solved each version of the problem 500 times, i.e.  $5$  (mutation)  $\times$   $5$  (cooling rate)  $\times$   $2$  (DPC)  $\times$   $2$  (modern/classical)  $\times$   $10$  (seeds) = 500, with both the modern GA and the classical GA. We took the best solution, i.e., the minimum number of stations, among these 500 solutions as our solution to the problem in Table 7.4. The optimal solutions to these problems as well as the results of previous studies that have targeted this problem are also given in Table 7.4.

It can be observed from Table 7.4 that the modern GA performs better than all heuristics except Nevins' (1972) and Baybars' (1986). However, there is no significant difference between the performance of Nevins' (1972) or Baybars' (1986) heuristics and the modern GA. The modern GA solutions match those of Baybars except for four cases in which we exceed the optimum solution by one and for one case in which we find the optimum solution while Baybars' LBHA-1 does not. Considering that the modern GA found five of the thirteen optimal solutions and found solutions to the other cases with only one more station than the optimum, it performs quite well on the versions of Tonge's (1961) problem. The classical GA also performs well, but the modern GA performs better than the classical GA in three cases. This result contradicts with the experiment

Cycle time	Optimal solution	Moodie and Young (1965)	Tonge (1965)			Nevins (1972)	Baybars (1986)	Modern GA	Classical GA
			MIF	RC	BPC				
83	47	48	50	50	49	47	47	48	48
86	46	47	47	48	47	46	46	46	47
89	43	44	45	46	44	43	43	44	44
92	?	43	43	44	43	42	42	43	43
95	40	42	43	43	41	40	40	41	42
170	22	24	24	24	23	23	23	23	23
173	22	24	24	24	23	22	23	23	23
176	22	22	24	23	22	22	23	22	23
179	21	22	23	23	21	21	22	22	22
182	21	22	23	22	21	21	22	22	22
346	11	11	11	12	11	11	11	11	11
349	11	11	11	11	11	11	11	11	11
364	11	11	11	11	11	11	11	11	11

Table 7.4: Comparison of eight methods on the 70-task problem of Tonge (1961) in terms of number of stations

in Chapter 6, in which we observe that the classical GA performs better than the modern GA on the average. First of all, the performance measure is "the number of stations" in Tonge's (1965) problem. Therefore, even if the classical GA provides a better balanced solution with the same number of stations, this improvement is not noticeable since it is not reflected on the performance measure. Additionally, we observe that the modern GA performed better than the classical GA in 35 of the 300 problems in Chapter 6, hence it is not very unlikely that it exceptionally performs better than the classical GA in three versions of Tonge's (1965) problem.

Although GAs are applicable to any kind of ALB problem regardless of the F-Ratio, we observe that they perform worse in problems with high F-Ratio, as in Tonge's (1961) problem with 59.42% F-Ratio. If the number of precedence relations increases, the possibility of generating offsprings that are better than their parents decreases. In such a case, another crossover operator that provides more substantial changes on the parents' genes may be used instead of a moderate crossover operator like the one we used. Our crossover operator is the two point crossover operator, as explained in Chapter 3. The purpose of the two point crossover is to conduct a neighborhood search that is done by keeping the head and the tail of each offspring the same as its parent. The offspring should be close in fitness to its parent because only its

middle genes have changed. Conversely, a one point crossover would change on the average the half of the entire chromosome of each offspring, and such a change could be too drastic and might move the offspring out of the local search neighborhood. Similarly, more-than-two-point crossover could result in changes in fitness functions that are either too small or too large depending on how the swapping is done. Hence, if we used a one point crossover we might have achieved better results compared to our two point crossover operator, for Tonge's problem that has a high F-Ratio.

On the other hand, Baybars' LBHA-1 consists of reduction phases that reduces the problem size by eliminating tasks, determining mutually exclusive task sets, and decomposing the network, while no reduction phases are applied to the problem before our GAs. Hence, if the same reduction phases were implemented before our GA, we might have achieved better results on Tonge's (1961) problem. It has been shown by Leu et al. (1994) that starting the GA with a better initial population significantly improves the solution quality.

# Chapter 8

## CONCLUSION

This chapter provides a brief summary of the contributions of this thesis and addresses some possible extensions of this study for future research. In this study, we have studied genetic algorithms (GAs) and their application to the assembly line balancing problem (ALB) for the deterministic and single model case (SALB). We proposed new solution methodologies to find near-optimal balances by making use of the authentic characteristics of the ALB problem in the GA structure. In the next section, we will make a short summary of our contributions.

### 8.1 Contributions

We showed that the chromosome structure of GAs can be changed dynamically to provide an effective search in the ALB problem domain. We have reduced the chromosome size during the search procedure, when certain conditions are satisfied, by freezing the stations at the beginning or the end of the assembly line and the tasks that are assigned to these stations. We improved both the solution quality and the computational time by this reduction technique, i.e., dynamic partitioning (DPA), compared to the GA without DPA. Over a set of 30 randomly generated problems, we tested our GA with DPA and have

seen that the objective value is improved by 16.43% for the problems with 10% F-Ratio and 7.69% for the problems with 50% F-Ratio. However, we did not observe a significant improvement even in the extreme case of problems with 90% F-Ratio, because these problems have a small number of possible solutions due to the large number of precedence relations that they consist of.

We also studied elitism, which is a rule that accepts the offspring only if it is better than its parent. We expanded the elitism rule to create levels of elitism by using the simulated annealing (SA) idea. Our contribution to revise the elitism rule extends to showing how any binary decision rule can be expanded to have a continuous range. We observed that elitism contributes significantly to the performance of the GA. This observation was not made in any other study, although elitism was used before in literature. We have seen that the levels of elitism does not significantly differ from each other on the average in our experiment with 30 randomly generated problems. We also observed that the optimum solution can be found in some problems at some of the other levels while it is not possible to find it at the strict elitism or no elitism levels. Thus, elitism with SA contributes to the search by providing multiple levels of elitism rather than restricting it with only strict elitism.

We compared the two kinds of GAs that can be classified according to their organizational structure as the classical GA and the modern GA. The difference between the two classes is basically the number of crossovers at each iteration, i.e., only two offsprings can replace their parents after one crossover operation at each iteration in the modern GA while a big proportion of the next population is regenerated by multiple crossover operations in the classical GA. We compared the two kinds of GAs and observed that the classical GA requires significantly more amount of computational time than the modern GA, but the classical GA performed significantly better on the average over a set of 30 randomly generated problems. Thus, we recommend the classical GA for solving ALB problems unless it will not be used in a very flexible manufacturing system that needs to be balanced frequently.



## 8.2 Future Research Directions

There are several future research directions originating from this research study as such:

- Prior to dynamic partitioning, a static partitioning procedure that divides the ALB problem into smaller subproblems can be applied to problems that consist of a very large number of tasks. If the problem is divided into sub-problems with this technique, then the chromosome size of the GA for each subproblem can be reduced to a reasonable size in order to prevent intolerable computational time requirements and to provide a more extensive search.
- Dynamic partitioning can be revised such as it freezes not only the stations at the beginning or at the end of the chromosome but also the other stations. We expect that such a revision would improve the performance of the GA with DPA especially in problems that consist of a large number of tasks.
- In this study, we considered only the single model and deterministic case of ALB problems (SALB), however the scope of the study can be extended to multi/mixed model and/or stochastic cases as well.
- Effects of DPA and elitism with SA may be observed in GAs with different crossover and mutation operators and with different coding representations.

# Bibliography

- [1] E. J. Anderson and M. C. Ferris. Genetic algorithms for combinatorial optimization: the assembly line balancing problem. *ORSA Journal on Computing*, pages 161–173, 1994.
- [2] F. V. Assche and W. S. Herroelen. An optimal procedure for the single model deterministic assembly line balancing problem. *International Journal of Operational Research*, 3, 2, 1979.
- [3] H. W. B, S. M. E, and S. W. W. How to balance an assembly line. Technical report, New Caraan, Conn.: Carr Press, Division for Advanced Management, 1954.
- [4] I. Baybars. An efficient heuristic method for the simple assembly line balancing problem. *International Journal of Production Research*, pages 149–166, 1986.
- [5] R. Bellman. *Adaptive control processes: a guided tour*. Princeton, NJ: Princeton University Press, 1961.
- [6] E. H. Bowman. Assembly line balancing by linear programming. *Operations Research*, pages 8, 3, 1960.
- [7] G. N. Bullock, M. J. Denham, I. C. Parmee, and J. G. Wade. Developments in the use of the genetic algorithm in engineering design. *Design Studies*. pages 507–524, 1995.
- [8] Y.-L. Chang and R. S. Sullivan. *QS: Quant Systems, version 2*. Englewood Cliffs, NJ: Prentice Hall, 1991.

- [9] C.-J. Chen and C.-S. Tseng. The path and location planning of workpieces by genetic algorithms. *Journal of Intelligent Manufacturing*, pages 69–76, 1996.
- [10] E. M. Dar-El. Malb - a heuristic technique for balancing large scale single-model assembly lines. *AIIE Transactions*, 5, 4, December 1973.
- [11] E. M. Dar-El and Y. Rubinovitch. Must - a multiple solutions technique for balancing single model assembly lines. *Management Science*, 25, 11, 1979.
- [12] L. Davis. Applying adaptive algorithms to epistatic domains. In *Proc. International Joint Conference on Artificial Intelligence*, 1985.
- [13] L. Davis. Job shop scheduling with genetic algorithms. In *Proc. International Joint Conference on Artificial Intelligence*, 1985.
- [14] S. Ghosh and R. J. Gagnon. A comprehensive literature review and analysis of the design, balancing and scheduling of assembly systems. *International Journal of Production Research*, pages 637–670, 1989.
- [15] D. E. Goldberg. *Genetic algorithms in search, optimization and machine learning*. Addison-Wesley, 1989.
- [16] Y. Gupta, M. Gupta, A. Kumar, and C. Sundaram. A genetic algorithm-based approach to cell composition and layout design problems. *International Journal of Production Research*, pages 447–482, 1996.
- [17] Y. P. Gupta, M. C. Gupta, A. Kumar, and C. Sundram. Minimizing total intercell and intracell moves in cellular manufacturing: a genetic algorithm approach. *International Journal of Computer Integrated Manufacturing*, pages 92–101, 1995.
- [18] M. Held and R. M. Karp. A dynamic programming approach to sequencing problem. *Journal of the Society of Industrial and Applied Mathematics*, 10, 1, 1962.

- [19] M. Held, R. M. Karp, and R. Shareshian. Assembly line balancing - dynamic programming with precedence constraints. *Operations Research*, 11, 3, 1963.
- [20] T. R. Hoffman. Assembly line balancing with a precedence matrix. *Management Science*, 9, 4, 1963.
- [21] J. H. Holland. *Adaptation in natural and artificial systems*. The University of Michigan Press, Ann Arbor, MI, 1975.
- [22] T. C. Hu. Parallel sequencing and assembly line problems. *Operations Research*, 9, Nov.-Dec. 1961.
- [23] J. R. Jackson. A computing procedure for a line balancing problem. *Management Science*, 2, 3, 1956.
- [24] R. V. Johnson. Assembly line balancing algorithms: computational comparisons. *International Journal of Production Research*, 19, 3, 1981.
- [25] H. N. Kamhawi, S. R. Leclair, and C. L. P. Chen. Feature sequencing in the rapid design system using a genetic algorithm. *Journal of Intelligent Manufacturing*, pages 55-67, 1996.
- [26] E. P. C. Kao and M. Queyranne. On dynamic programming methods for assembly line balancing. *Operations Research*, 30, 2, 1982.
- [27] M. D. Kilbridge and L. Wester. A heuristic method of assembly line balancing. *The Journal of Industrial Engineering*, pages 292-298, 1961.
- [28] M. Klein. On assembly line balancing. *Operations Research*, 11, 2, 1963.
- [29] Y. Y. Leu, L. A. Matheson, and L. P. Rees. Assembly line balancing using genetic algorithms with heuristic-generated initial populations and multiple evaluation criteria. *Decision Sciences*, pages 581-606, 1995.
- [30] C. L. Moodie and H. H. Young. A heuristic method of assembly line balancing for assumptions of constant or variable work element times. *Journal of Industrial Engineering*, 16, 1, 1965.

- [31] A. J. Nevins. Assembly line balancing using best bud search. *Management Science*, 18, 9, 1972.
- [32] J. H. Patterson and J. J. Albracht. Assembly-line balancing: zero-one programming with fibonacci search. *Operations Research*, 23:166–172, 1975.
- [33] M. E. Salveson. The assembly line balancing problem. *The Journal of Industrial Engineering*, 8, 3, 1955.
- [34] L. Schrage and K. R. Baker. Dynamic programming solution of sequencing problems with precedence constraints. *Operations Research*, 26, May-June 1978.
- [35] T. Starkweather, D. Whitley, K. Mathias, and S. McDaniel. Sequence scheduling with genetic algorithms. Technical report, Colorado State University, 1991.
- [36] G. Suresh, V. V. Vinod, and S. Sahu. A genetic algorithm for facility layout. *International Journal of Production Research*, pages 3411–3423, 1995.
- [37] G. Suresh, V. V. Vinod, and S. Sahu. A genetic algorithm for assembly line balancing. *Production Planning and Control*, pages 38–46, 1996.
- [38] F. B. Talbot and J. H. Patterson. An integer programming algorithm with network cuts for solving the assembly line balancing problem. *Management Science*, 30, 1, 1984.
- [39] F. B. Talbot, J. H. Patterson, and W. V. Gehrlein. A comparative evaluation of heuristic line balancing techniques. *Management Science*, 32, 4, 1986.
- [40] F. B. Talbot, J. H. Patterson, and W. V. Gehrlein. A comparative evolution of heuristic line balancing techniques. *Management Science*, pages 430–454, 1986.
- [41] S. R. Thangavelu and C. M. Shetty. Assembly line balancing by zero-one programming. *AIIE Transactions*, 3, 1, 1971.

- [42] F. M. Tonge. *A heuristic program of assembly line balancing*. Englewood Cliffs, NJ: Prentice-Hall, 1961.
- [43] T. M. Tonge. Assembly line balancing using probabilistic combinations of heuristics. *Management Science*, 11, 7, 1965.
- [44] R. V. V. Vidal. *Applied simulated annealing*. Springer-Verlag, 1993.
- [45] T. S. Wee and M. J. Magazine. *An efficient branch and bound algorithm for an assembly line balancing problem - part I: minimize the number of work stations*. University of Waterloo, Ontario, Canada, June 1981.
- [46] M. A. Wellman and D. D. Gemmill. A genetic algorithm approach to optimization of asynchronous automatic assembly systems. *The International Journal of Flexible Manufacturing Systems*, pages 27–46, 1995.
- [47] W. W. White. Comments on a paper by Bowman. *Operations Research*, 9, March-April 1961.
- [48] D. Whitley and J. Kauth. Genitor: A different genetic algorithm. In *Rocky Mountain Conference on Artificial Intelligence*, 1988.

## Vitae

Muzaffer Tanyer was born on August 24, 1973 in Samsun, Turkey. He studied secondary and high school at Tarsus Amerikan College. He attended to the Department of Electical and Electronics Engineering, Bogazici University, in 1991 and graduated from the same department in July 1995. In October 1995, he joined to the Department of Industrial Engineering at Bilkent University as a research assistant. Until September 1997, he worked with Assoc. Prof. İhsan Sabuncuoğlu for his graduate study at the same department.