1-1-1994

Searching for a minimal cost of closed-loop automatic assembly system with the genetic algorithm

Yisheng Hsiao
Iowa State University

Searching for a minimal cost of closed-loop automatic assembly system with the genetic algorithm

by

Yisheng Hsiao

A Thesis Submitted to the

Graduate Faculty in Partial Fulfillment of the

Requirements for the degree of

MASTER OF SCIENCE

Department: Industrial and Manufacturing Systems Engineering
Major: Industrial Engineering

Iowa State University
Ames, Iowa

1994

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGEMENTS

I would like to thank my advisor, Dr. Doug Gemmill, for all his assistance, guidance and support throughout this research. His valuable suggestions and comments were greatly appreciated. His instruction made the process of completing my thesis finish smoothly.

I would also like to thank my parents for their constant encouragement, love, and support. You were there whenever I needed your support. I will always grateful for all you have given for me.

1. INTRODUCTION

The assembly process plays an significant role in U.S. manufacturing and the economy. According to statistical reports of the Bureau of Labor, there are about eighteen million people employed in manufacturing in the U.S. (Schloemer, 1992). Manufacturing is regarded as a competitive weapon in the marketplace and it is recommended that each company include in their business plans specific goals in the area of achieving manufacturing excellence (Hays and Wheelright, 1984). Manufacturing consists of three major stages; design, fabrication, and assembly. The resulting activity of manufacturing can be either fabrication or assembly. It was estimated that approximately eight million people work in the area of manufacturing processes associated with product assembly (Liu and Sanders, 1988). Usually, when the product is more complex, assembly is a larger concern. The assembly cost can often account for more than 50% of the completed product (Boothroyd, 1992). Obviously, the assembly process is an important area for U.S. manufacturing and the economy.

An assembly line can be classified as manual or automatic. In manual assembly, people work in a line each contributing something to the assembly of a certain product. In automatic assembly, automatic assembly devices, such as robots, spot-welding, etc., have been used to replace manual workers. The automation of assembly has many advantages: reduction in the cost of assembly, increased productivity, a more consistent product, and avoidance of dangerous operations for operators. However, only 5% of products are produced by automatic assembly, others are still produced by manual assembly (Boothroyd, 1992). Although only a small portion of products are produced by automatic assembly, it is expected that automatic assembly will grow in batch manufacturing. Batch production represents more than 35% of the US manufacturing base and makes up 36% of manufacturing's share of the GNP (Browne et al. 1988).

An automatic assembly system contains a series of workstations which are formed in a predetermined order. A transport mechanism is used to transfer the assemblies between workstations. An automatic assembly system could be labeled as an intermittent or continuous transfer system (Boothroyd, 1992). In the intermittent system, the pallets are moved intermittently and the workheads, structures used to assemble parts, remain fastened. In the continuous system, the workheads index back and forth as the pallets are transferring at a constant speed. The intermittent assembly system is the more common system used in industrial automatic assembly.

Automatic assembly systems can also be grouped as synchronous systems or asynchronous systems according to the type of transfer system installed. The synchronous system transfers all the assemblies simultaneously; consequently, the entire assembly system stops if any one of the workstations is shutdown. Thus, a synchronous assembly system with several workstations will have high downtime if any one of the station's reliability is not high. However, an asynchronous assembly system is separated by buffer units. When a workstation has finished its operations, the assembly is moved to another workstation by a continuous operating carrier/conveyor. It will not affect other fixtures or workstations when assemblies are removed from the conveyor of the system for assembly tasks.

According to the configuration of the transport mechanism, an automatic assembly system can be classified as open or closed system. A closed system starts and finishes assemblies at the same area; however, an open assembly system starts assemblies at one end and completes at the other. Since an open system does not have any space limit, it may have a variable number of pallets in the system. On the other hand, a closed system has a fixed number of assemblies in the system at all times.

This study will deal exclusively with a closed asynchronous automatic assembly system (Figure 1.1). This kind of automatic assembly system exists in a variety of areas such as

manufacturing and packaging. For this type system, the workstations are designed around a closed loop. Assemblies are built on pallets and the pallets are moved by conveyor. All pallets will be occupied when the system is running. As soon as a completed assembly is unloaded, a new workpiece is installed on the pallet. Consequently, the loading area becomes the first station in the system, and the unloading area becomes the last station. Both the space between any two adjoining workstations and the pallet dimensions decide the number of pallets that can be formed in a queue between any two adjacent workstations.

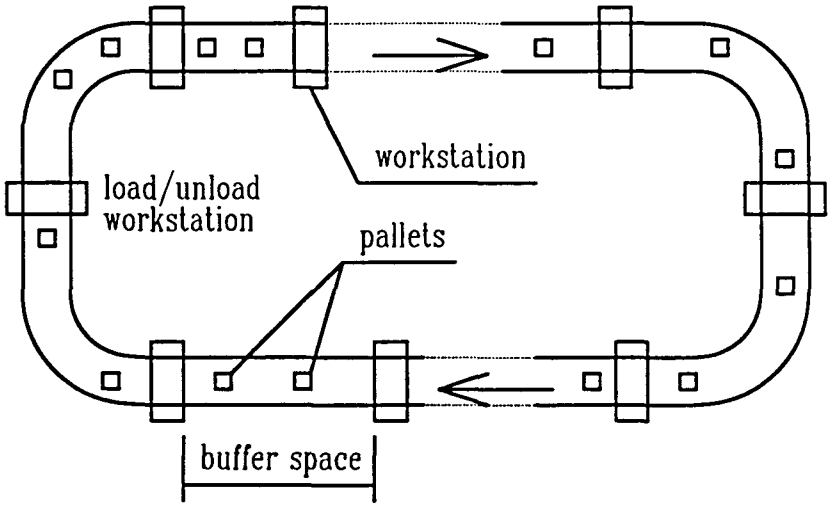

Figure 1.1. Closed-Loop Asynchronous Automatic Assembly System

The analysis of automatic assembly systems has been classified into deterministic models, analytical models or stochastic process models, and simulation methods. Deterministic models use equations to predict the behavior of the system. Deterministic modeling works well in the analysis of deterministic systems. However, most systems are actually not deterministic. It is difficult to use deterministic modeling to model a system

containing stochastic elements. Analytical approaches only work well on small and less complex systems. This method can be used effectively to model systems containing two or three stations. But, it is difficult to incorporate the transportation delays, blocking and starvation aspects of an automatic assembly system by applying queuing theory. Recently some effort has been made to model the transportation delay and its effect on performance evaluation of transfer lines (Commault and Semery, 1990). Most queuing models assume that the buffer space is infinite. Thus, the blocking effect is ignored. However, buffer spaces are usually small and the blocking effect can be considerable. For a more complex system, simulation is the most effective of the three modeling techniques. Simulation can properly incorporate the complicated effects of transportation delay, blocking and starving in stochastic systems. However, simulation is only an evaluative approach. It is necessary to combine a search technique in order to find the optimal solution.

The search techniques used to search for optimal solutions are categorized into gradient methods, enumerative methods, and random search methods (Goldberg, 1989). Gradient methods use slopes or derivatives to improve performance towards minimum or maximum point or optimal variable set. These methods are effective for well-behaved functions, or continuous functions, having few local optimum. Enumerative methods are used to evaluate all possible combinations of system variables. Clearly, this approach is feasible only when systems are small. Random search methods randomly generate values for the system variables in order to search for the optimal solution. One of the most well-known of the random search techniques is the genetic algorithm.

The genetic algorithm is a randomized algorithm that searches for a globally or near globally optimal solution. This algorithm uses bit strings as the method of representing complicated structures. The genetic algorithm uses three transformations, reproduction, crossover, and mutation to generate new configurations. This algorithm generates new

configurations in a manner similar to the mechanics of natural selection and "survival of the fittest."

The main objective of this research is to investigate the effectiveness of the genetic algorithm to the optimization of a closed-loop asynchronous automatic assembly system with a stochastic cost model. This research applies the genetic algorithm method to search for the solution which minimizes the manufacturing cost. The decision variables consist of the total number of pallets and the number of buffer spaces between stations. In this research, the optimal number of buffer spaces and pallets, and the production rate are obtained by linking a simulation model to the genetic algorithm. A cost model is used to calculate the cost required for this system. Comparison of the solutions between the maximum throughput problem and the minimum cost problem are made.

2. LITERATURE REVIEW

Automatic assembly systems are a member of stochastic systems. When an automatic assembly system is designed, the optimization of the system's performance is always a serious concern. Methods applied for stochastic optimization are normally referred to as Monte Carlo methods. A lot of different optimization techniques have been developed for solving stochastic problems. Glynn (1986) surveyed various approaches, all continuous parameter stochastic optimization, in simulation optimization. These approaches include Response Surface Methodology, Stochastic Quasi-Gradient Methods (SQG), Kiefer-Wolfowitz Algorithm, and Robbins-Monro Algorithm. These approaches may not work well when the parameters are discrete. Meketon (1987) also surveyed certain other approaches. He classified the approaches into three categories: standard non-linear programming techniques, response surface methodologies, and stochastic approximation techniques. However these approaches can only guarantee a local optimum solution, not a global solution. In addition to these stochastic optimization techniques, some discrete parameter deterministic optimization techniques, such as simulated annealing and genetic algorithms, have been applied to stochastic problems with good results. Simulated annealing and genetic algorithm are random search algorithms based on processes found in thermodynamics and natural selection respectively (Davis and Steenstrup, 1987).

This literature review collects two areas of the automatic assembly process analysis. The first section will review the research on genetic algorithms. The second section will discuss the efforts performed on the optimization of assembly systems. The final section provides a general summary of the literature as related to this research.

2.1 Genetic Algorithms in Optimization

This section will present the literature of genetic algorithms in two categories: algorithm development and applications. The performance of the algorithm is presented in chapter four.

2.1.1. Literature on the Development of Genetic Algorithms

The Genetic Algorithm is a discrete parameter probabilistic method for searching the solution space. Holland (1975) first applied this algorithm in artificial systems. He presented the concepts of reproduction, crossover, and mutation to demonstrate how the genetic algorithm can offer an effective search in a complicated solution space.

Davis (1987) edited a book that contains 13 papers. These papers discuss various issues that use genetic algorithms and simulated annealing as searching tools. In this book, Davis and Steenstrup reviewed and supplied a brief description of both simulated annealing and genetic algorithms. In the same book, Grefenstette (1987) considered the incorporation of problem-specific information into genetic algorithms. Grefenstette implied that genetic algorithms are not particularly useful for fine local searches. He suggested that one could apply genetic algorithms to specify "promising" regions, and then invoke a local search method to search for the optimum solution. He used several heuristic methods for population initialization to evaluate the traveling salesman problem. Grefenstette inferred that if heuristic information is utilized with caution to avoid causing premature convergence of the solution, the heuristic information should be effective. In the same book, Booker (1987) considered the premature convergence problem of the genetic algorithm. He showed that carefully selected modifications of search operators, such as using two crossover points instead of one and changing the crossover rate dynamically to make up for imbalances, can dramatically improve the performance of genetic algorithms. Goldberg discussed the behavior of simple genetic

algorithms when applied to the minimal, deceptive problem (MDP). The MDP is built to misguide the simple genetic algorithm away from the global optimum solution and toward sub-optimal solutions. In Davis' book, Goldberg (1987) concluded that the simple genetic algorithm converged across a broad range of initial parameters, thus, the MDP could not distract the genetic algorithm.

Pettey et al. (1987) tried to use a parallel genetic algorithm to decrease the search time in problems with large population size. The authors show when the population size is exceedingly large, genetic algorithms can take an excessive amount of running time. In contrast, when the population is small, genetic algorithms can be limited in terms of the wrong solution space. The authors introduce a class of parallel genetic algorithms (PGA) to defeat the problem of redundant running time and illustrate them in the traveling salesman problem. The results indicate that the PGA can allow for an larger population size of a genetic search.

Richardson et al. (1989) showed some steps to accomplish penalty functions in genetic algorithms. In this penalty scheme, some infeasible or illegal combinations are given a strong penalty. Historical recommendations for using penalty functions suggested applying harsh penalties for infeasible solutions. Infeasible solutions would be forced out of the current population when a large penalty is assigned to the performance measure of the improper solutions. The authors recommended that a well chosen, ranged penalty is more desirable than harsh penalties. The authors inferred that these types of penalties maintain the information for all strings; however, the harsh penalties do not.

Goldberg (1989) published a book that focuses on the issues of genetic algorithms. This book introduces the history and operation of genetic algorithms. Goldberg discusses the simple operators such as reproduction, crossover, and mutation in detail, several advanced genetic operators containing dominance and abeyance, and mathematical foundations. Goldberg also introduces several knowledge-based techniques that involve genetic algorithms,

such as knowledge-augmentation and hybridization. The techniques of knowledge-augmentation involve improving a genetic algorithm with some "problem-specific" data. Hybrid schemes mix the crossing of a genetic algorithm with a problem-specific optimization or search technique. This book also discusses parallel genetic algorithms which indicate that a single master would simultaneously directe several different, but parallel, generations. Moreover, this book provides Pascal code for a simple genetic algorithm (SGA) that combines the three fundamental genetic operators, reproduction, crossover, and mutation.

Davis (1991) published the Handbook of Genetic Algorithms. Unlike any previous publications, this book discusses the coding of genetic algorithms from an object-oriented point of view. In fact, Davis refers to his code as the Objective-Oriented Genetic Algorithm (OOGA). Overall, Davis' main contribution to genetic algorithm research was twofold: a lot of case studies implicating the application of genetic algorithms and using an object-oriented method to implement genetic algorithms.

2.1.2. Literature on the Application of Genetic Algorithms

There are a lot of publications describing the application of genetic algorithms to theoretical and "real-world" problems. The following section reviews some of the applications of genetic algorithms, and relates to this study.

Davis and Ritter (1987) applied genetic algorithms to optimize the performance of a simulated annealing algorithm that was used to optimize student class schedules. The authors concluded that the application of genetic algorithms in this area enabled them to obtain better annealing parameter settings than humans found.

Glover (1987) used genetic algorithms to solve a complex keyboard configuration problem. Since it is difficult to produce language-to-keyboard mapping for the Eastern Asian languages, Glover tried to use genetic algorithms to find the optimal solution. Glover claimed

that the genetic algorithms provide a robust search technique when applied with different operators and representations than those used in the standard genetic algorithm.

Cohoon et al. (1988) applied distributed genetic algorithms on the floor plan design problem. This particular application is used to design the placement of modules in the VLSI cycle. The objective of the placement is to find the minimum of the wire lengths and weighted sum of the area. The authors apply multiple processors (called GAPE) to implement a distributed genetic algorithm. After developing several fit sub-populations, these sub-groups are combined by GAPE into one large generation. GAPE then goes forward to evolve this single population. They found that GAPE worked consistenty better than using genetic algorithms in a serial procedure.

Falkenauer and Bouffouix (1991) used the genetic algorithm to optimize the job shop problem with many tasks, many machines, and precedence constraints. The job shop problem also is a combinational problem. They showed the difficulties in solving the job shop problem. They presented an encoding of the job shop problem to conquer these difficulties. Then, they applied the genetic algorithm to demonstrate the performance on job shop scheduling problems with examples of real-world size.

Wellman (1991) transferred a simple genetic algorithm to the optimization of buffer space and pallet allocation in a closed-loop asynchronous automatic assembly system. Wellman's research concentrated on the application issues of a simple genetic algorithm and the relative performance of this algorithm compared to the results of Liu and Sanders' (1988) work. Wellman claimed that the simple genetic algorithm did not get better results in comparison to Liu and Sanders' SQG method. However, Wellman's research results showed that the genetic algorithm could obtain reasonable results.

Huntley and Brown (1991) used a parallel heuristic to solve the quadratic assignment problem. Huntley and Brown developed an algorithm, SAGA, with the idea of combining

decentralized characteristics of genetic algorithms and centralized characteristics of simulated annealing methods. A genetic algorithm is used to create populations and then simulated annealing "matures" these populations. The authors concluded SAGA performed well on two standard problems found in the related literature; however, SAGA's running time was longer than some less complicated algorithms.

Fujita et al. (1993) combined a genetic algorithm and a local optimal algorithm to optimize layout design problems including blank nesting. They used the genetic algorithm to handle the combinations (solutions), and the local minimization algorithm to determine the embodiment layout based on the fixed combinations so as to minimize the volume that corresponds to the fitness in the genetic algorithm. They concluded that this hybrid approach could produced an effective nesting result.

Genetic algorithms have been successfully applied to many optimization problems. In addition to the above applications, genetic algorithms have been applied in optimal control problems, process design and optimization problems, database query optimization, neural networks, and machine learning. The applications of genetic algorithms mentioned above just scratch the surface of what is available.

2.2 Optimization of Automatic Assembly Systems

In the last few years, many researchers have developed different techniques and methods for performance evaluation of automatic assembly systems.

Leung and Sanders (1986) presented a special form of parallel workstation, called tunnel-gated station, to solve the problem when some operations may have relatively long cycle time in an automatic assembly system. The tunnel-gated station is a kind of transfer machine used to seize an assembly from the transfer and lift it to an elevated position. Thus,

an additional pallet can pass under the station, when the tunnel-gated station is working. This type of system could be applied to carry out parallel station configurations.

Kamath and Sanders (1987) developed analytical methods to evaluate the performance of an automatic assembly system. They applied the Renewal Approximations (RA) approach and the Product -Form Analysis (PFA) approach to a closed-loop automatic assembly system with a queuing network model. The results show that both approaches worked well. However, in the large automatic assembly system, the PFA approach does not work better than the RA approach.

Kamath et al. (1988) extended their own research of analytical performance analysis models to optimize a closed-loop flexible assembly system. They used an approximate factor to improve the general arrival and service time at each queue. They assumed that this closed-loop flexible assembly system model did not have any transport delay or blocking. The simulation results showed that the approximate approach could improve the accuracy for steady-state performance measures. The results also showed that the approximate approach performed well for a broad range of parameter values and system sizes.

Liu and Sanders (1988) combined a queuing network model and a stochastic quasi-gradient (SQG) method to the performance optimization of asynchronous flexible assembly systems to search for maximum system throughput. First, a queuing network was used to determine the number of pallets in the system. Then, under the fixed number of pallets, a SQG algorithm is applied to minimize the number of buffer spaces. They test this Network-SQG method with a variety of assembly systems which include ten workstations and are subject to blocking and starvation effects. They concluded that this hybrid algorithm performed well and could obtain a near optimal solution in this discrete problem, even though the SQG algorithm was designed for solving continuous problems.

Bulgak and Sanders (1988) presented an approximate analytical method to optimize automatic assembly systems with statistical process control and repair loop. They pointed out that quality and productivity improvement are of vital importance due to the competitive world market. Statistical process control is applied to produce a high percentage of acceptance of quality. They concluded that this analytical model could accurately predict the performance of the automatic assembly system. Bulgak and Sanders (1991) extended their model to asynchronous flexible assembly systems. This system considered starvation and blocking effects. They applied a hybrid approach for the stochastic optimization of asynchronous flexible assembly systems with statistical process control and repair loops. First, they used analytical models to set the number of pallets to meet a certain throughput. They then used Monte Carlo optimization methods associated with discrete event simulation to evaluate the number of buffer spaces that attempts to maximize the throughput. The stochastic quasi-gradient and a modified simulated annealing algorithms were used to implement the Monte Carlo optimization. The authors claimed that both methods perform reasonably well in designing this system to obtain a maximum production rate.

Liu and Chiou (1989) developed a heuristic method based on a queuing network model for the design optimization of a closed automatic assembly system. They used this heuristic method to determine the total number of pallets and the number of buffer spaces between workstations to meet the optimal system utilization. Then, they applied a regression model and simulation experiments to evaluate the performance of the heuristic method. They found that the heuristic method could obtain a near optimal solution.

Tandiono (1991) extended a study of Liu and Sanders (1988) mentioned previously. In this study, she attempted to use the manufacturing cost as the goal of performance of the asynchronous assembly system. In contrast, Liu and Sanders evaluated the performance of asynchronous automatic systems based on system throughput. Tandiono presented a cost

model that considered the cost of the pallets, buffer spaces, conveyors, holding inventory, and a penalty cost. Tandiono combined the cost model, stochastic quasi-gradient method, and a discrete event simulation model to search for the minimum cost. She found that by considering the economic factor, the system could optimize the number of pallets and the number of buffer spaces at the same time and find a solution which is more suitable in the real world.

Wellman (1991) applied genetic algorithms in a study of Liu and Sanders (1988) to optimize the performance of an asynchronous automatic assembly system. The genetic algorithm performs reasonably well in getting good solutions when compared with results of SQG of the same system, even though genetic algorithms were built for application to deterministic systems. However, it does not show up to be superior to SQG. Since the response surface for the buffer and pallet allocation problem using throughput as the objective function is basically smooth. The SQG method performs well on this type of surface due to its search algorithm depending on estimates of gradients.

2.3 Summary of Literature

Genetic Algorithms are designed for solving deterministic objective functions. They have been used to deal with many kinds of theoretical and practical problems. However, the genetic algorithms have not been used on an automatic assembly system except Wellman's effort.

In the last few years, many researchers have developed analytical methods for performance evaluation of automatic assembly systems. These studies are mentioned previously. The difficulties in modeling the automatic assembly system are the vast number of decision variables involved and the complicated interactions among them. Thus, many unrealistic simplifying assumptions are usually required for these analytical models of

automatic assembly systems. In order to avoid these assumptions, simulation is used to model automatic assembly systems. Moreover, faster computers and cheaper computing cost lead to computer simulation becoming more powerful and popular.

There are two important factors that will affect the performance of an automatic assembly system. One is the number of buffer spaces between stations. The other is the number of pallets in the system. If the number of buffer spaces is too large, more work-in-process inventories will be loaded in the system. If the number of buffer spaces are too small, the frequency of blocking will increase and a high probability of starvation will occur because of small work-in-process inventories. The number of pallets in the system has a big impact on the effects of changes in number of buffer spaces. So, the performance of automatic assembly system could be improved by setting up an appropriate number of buffer spaces and an appropriate number of pallets in the system.

Liu and Sanders (1988) used the stochastic quasi-gradient (SQG) algorithms for performance optimization of asynchronous flexible assembly systems based on system throughout or production rate. Tandiono (1991) extends the study of Liu and Sanders to minimize the manufacturing cost. She also applied the SQG method to search for the optimal solutions, the number of pallets and buffer spaces. The SQG method is revealed to be a viable option; however, the algorithm tends to be a "greedy" algorithm in that it will find local optimal rapidly. Wellman (1991) applies genetic algorithms to the performance optimization of an asynchronous automatic assembly system. The genetic algorithm performs reasonably well in getting good solutions when compared with results of SQG, even though genetic algorithms were built for application to deterministic systems. However, it does not appear to be superior to SQG. If a cost function is used, the response surface could become quite "ragged" depending on the nature of the cost function. In this situation the genetic algorithm

which does not depend on gradient search, may be able to find better results. This application of the genetic algorithm has not been investigated.

This research attempts to combine genetic algorithms, a cost model, and a discrete event simulation to optimize the performance of a closed-loop asynchronous automatic assembly system.

3. SYSTEM DESCRIPTION

This research applies the genetic algorithm method to search for the optimal solutions by minimizing the manufacturing cost. The decision variables consist of the total number of pallets and the number of buffer spaces between each pair of workstations. This chapter discusses the systems that were used to fulfill the goal. The simulation model is discussed in section one. The cost model is presenteded in section two.

3.1. Simulation Model

3.1.1. Model Development

In order to enable the performance analysis of stochastic optimization techniques, a model similar to that of Liu and Sanders' study (1988) was developed. The simulation was developed using SIMAN (SIMulation ANalysis) simulation language. SIMAN from Systems Modeling Corporation, like other general simulation languages, such as GPSS, SLAM, and SIMSCRIPT, is a commercial simulation package. The simulation model considers the transport delays, the blocking, and starvation effects. When the downstream buffer is full, the current station becomes blocked. When the upstream buffer is empty, the current station becomes starved. It is assumed that no direct supply is available at the upstream buffer. So, transport delay is considered in this model. The transport time required to fully leave the station and arrive at the next buffer is also considered in this model.

The station cycle time is assumed to be deterministic. The randomness depends on the random times between station jams and on the random times expected to clear the jams. Part jams may happen whenever there is a bad positioning of the part being assembled, or a defective part is being assembled. When this condition occurs, the part will be repositioned, or the defective part will be taken away. It is presumed that the jams do not destroy the base

part, therefore, it is not necessary to move the base part away from the pallet. Also, it is assumed that there is only one operator available to clear the jams. Other required information in this model is the jam rate, the mean clear time and its distribution, the station cycle time, and the transport time.

3.1.2. Batch Size and Steady-State Simulation

Simulations may be classified as terminating or nonterminating, depending on whether there is an obvious way to determine run length. A terminating simulation is one that specifies some finite length of time of each run, while a nonterminating simulation is one that does not specify the length of a run, or at least runs over a very long period of time. This simulation model of an asynchronous automatic assembly system utilizes a nonterminating simulation.

A nonterminating simulation starts at simulation time zero and runs for a specified period of time. To avoid the influence of the initial condition of the model, steady-state simulation is used for such simulation. Its objective is to study long-run, or steady-state, behavior of a nonterminating simulation. One way to observe the steady-state behavior is called warming up the model or initial-data deletion. This method divides each simulation run into two phases: an initialization phase from time zero to time T_s, and a data collection phase from time T_s to the finishing time T_f. It is important to choose the appropriate value for the warm-up period (or deletion length) T_s to delete the effects of initial transients.

One of the approaches for analyzing the output of a nonterminating simulation is the batch means method (Banks and Carson, 1984). This study uses this batch means method to decide the warm-up period. In order to apply this batch means method, the batch size should be predetermined. A procedure that decides the batch size is stated as follows:

(1). The procedure starts with an instinctively chosen batch size (subrun length) l and an initial number of batches n. Each batch results in a single response x_i, with $i = 1,\ldots,n$. The number n must be greater than or equal to 100.

(2). Next, it is tested whether the batch responses x_i are indeed independent or not. The Von Neumann statistic is used to confirm this problem. The Von Neumann equation is defined as follows:

$$q = \frac{\sum_{i=1}^{n-1}(x_i - x_{i+1})^2}{\sum_{i=1}^{n}(x_i - \bar{x})^2}$$

where x_i = response from subrun i,

$\bar{x}$ = average response from all n subruns.

When all the x_i are independent, the value of q will be equal to 2. A z-test is used to test this statistical problem, the null hypothesis H_0: $q = 2$ (independent x_i) vs. the alternative hypothesis H_a: $q \neq 2$. The null hypothesis will be rejected when

$$\left|\frac{q-2}{\sigma_q}\right| < Z_{\alpha/2}$$

where $\sigma_q = \sqrt{\frac{4(n-2)}{(n-1)(n+1)}}$

(3). If the result shows independence, then the process stops. This batch size will be applied to the batch means method. If the null hypothesis is rejected, the procedure returns to step (1) and the value of the batch size is increased.

The input parameters for the batch size procedure and batch means method were derived from the examples of the maximum throughput model of Liu and Sanders (1988). Those input parameters are stated as follows.

Problem type 1:

Station Cycle Time = 5 seconds
Number of Pallets = 20
Number of Buffer Spaces at stations 1 through 10 = (3, 3, 3, 3, 3, 3, 3, 3, 3, 3)
The Jam Rates (%) at stations 1 through 10 = (1, 1, 1, 1, 1, 1, 1, 1, 1, 1)
Transportation Time = 1 second per buffer space
Jam Clear Time = Geometric distribution (mean 18 seconds)

Problem type 2:

Station Cycle Time = 5 seconds
Number of Pallets = 40
Number of Buffer Spaces at stations 1 through 10 = (5, 5, 17, 4, 4, 4, 4, 5, 5, 5)
The Jam Rates (%) at stations 1 through 10 = (0, 3, 3, 0, 0, 0, 3, 0, 0, 0)
Transportation Time = 1 second per buffer space
Jam Clear Time = Geometric distribution (mean 36 seconds)

Problem type 3:

Station Cycle Time = 5 seconds
Number of Pallets = 40
Number of Buffer Spaces at stations 1 through 10 = (4, 4, 10, 10, 12, 12, 4, 4, 4, 4)
The Jam Rates (%) at stations 1 through 10 = (0, 3, 0, 3, 0, 3, 0, 0, 0, 0)
Transportation Time = 1 second per buffer space
Jam Clear Time = Geometric distribution (mean 36 seconds)

Problem type 4:

Station Cycle Time = 5 seconds
Number of Pallets = 20
Number of Buffer Spaces at stations 1 through 10 = (2, 3, 4, 4, 4, 2, 2, 2, 3, 3)
The Jam Rates (%) at stations 1 through 10 = (0, 3, 0, 0, 2, 0, 0, 2, 0, 0)
Transportation Time = 1 second per buffer space
Jam Clear Time = Geometric distribution (mean 36 seconds)

Problem type 5:

Station Cycle Time = 5 seconds
Number of Pallets = 50
Number of Buffer Spaces at stations 1 through 10 = (4, 11, 15, 6, 6, 6, 12, 11, 6, 7)
The Jam Rates (%) at stations 1 through 10 = (0.5, 3, 0.5, 0.5, 0.5, 0.5, 3, 0.5, 0.5, 0.5)
Transportation Time = 1 second per buffer space
Jam Clear Time = Geometric distribution (mean 18 seconds)

With these five types of system parameters, the Von Neumann statistic showed that a batch size 1000 seconds is an appropriate batch size under testing at $\alpha = 0.05$ and 100 batches. In this situation, the value of q fell in the acceptance interval (1.61192, 2.38808). Table 3.1 shows the results for the five problems with ten different runs of the simulation for each type.

When the batch size is decided, the next step is to determine the warm-up period. Ten independent runs were obtained to determine a truncation point where all observations before this point are abandoned and observations after this point are kept. Each run consisted of 100

Table 3.1. The Results of Von Neumann Test

Number of Run	Type 1 q-value	Type 2 q-value	Type 3 q-value	Type 4 q-value	Type 5 q-value
1	2.138303	2.037759	2.351688	1.761249	2.006332
2	2.368414	2.181268	1.842971	1.917113	1.931841
3	2.172705	1.92972	1.940507	1.767926	2.071625
4	1.96486	2.094937	1.838898	2.119209	2.121761
5	2.14064	1.843073	2.153594	1.91101	1.941406
6	1.837485	1.97146	2.313608	1.927583	2.117418
7	1.957847	1.932803	2.313048	1.654419	2.144834
8	2.216105	1.775853	2.012118	2.022636	1.746736
9	1.756472	2.518383	1.869536	2.242487	2.104225
10	2.072204	2.285505	1.994886	2.264007	2.051299

batches and each batch was 1000 seconds long. The results are shown in Figure 3.1 to Figure 3.5. Checking these five figures, deleting the first three batches would eliminate most of the initial transient bias. Therefore, the initial 3000 seconds of simulation time was selected as the warm-up period.

3.1.3. Model Validation

To validate this simulation model, five output results with different input parameters from this simulation model are compared to those from Liu and Sanders (1988). A 95% confidence interval is used to measure the statistical accuracy by running the simulation for a specific length of time that generates about 20000 assemblies. In Liu and Sanders' simulation system, they used exactly 20000 assemblies, not a specific length of time, to obtain the confidence interval. In this study, problem type 1 and type 5 use 150000 seconds as the simulation time, and problem type 2, 3, and 4 use 160000 seconds as the simulation time. In

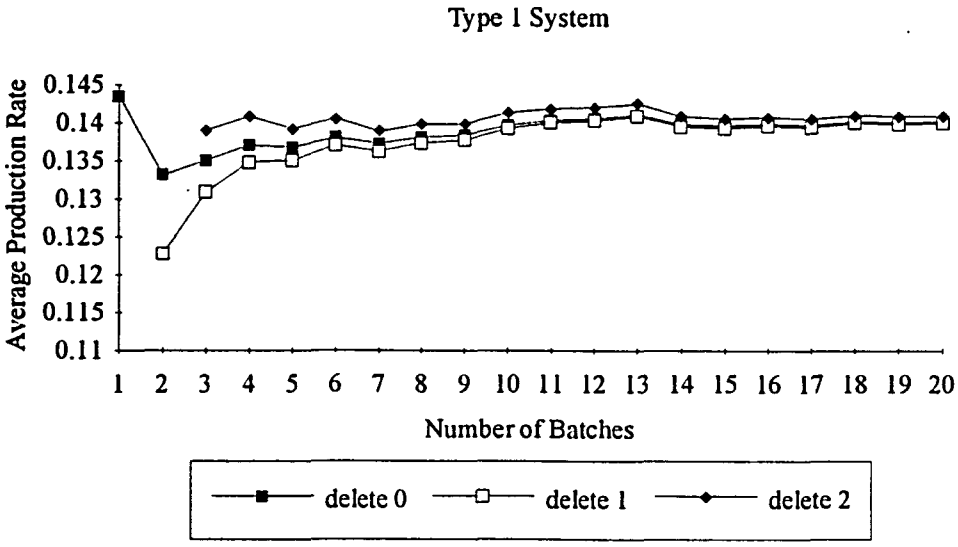

Figure 3.1. Cumulative Averages with Deletions of Type 1 System

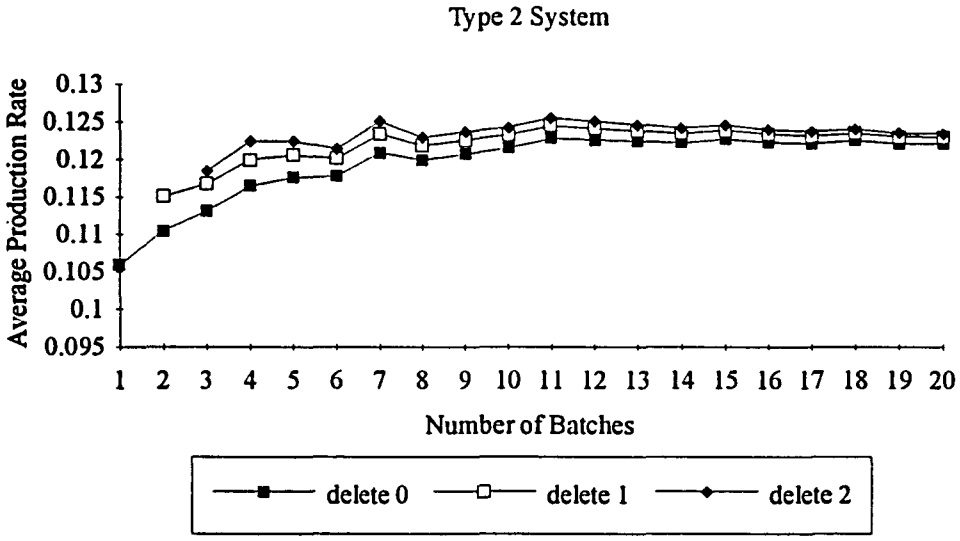

Figure 3.2. Cumulative Averages with Deletions of Type 2 System

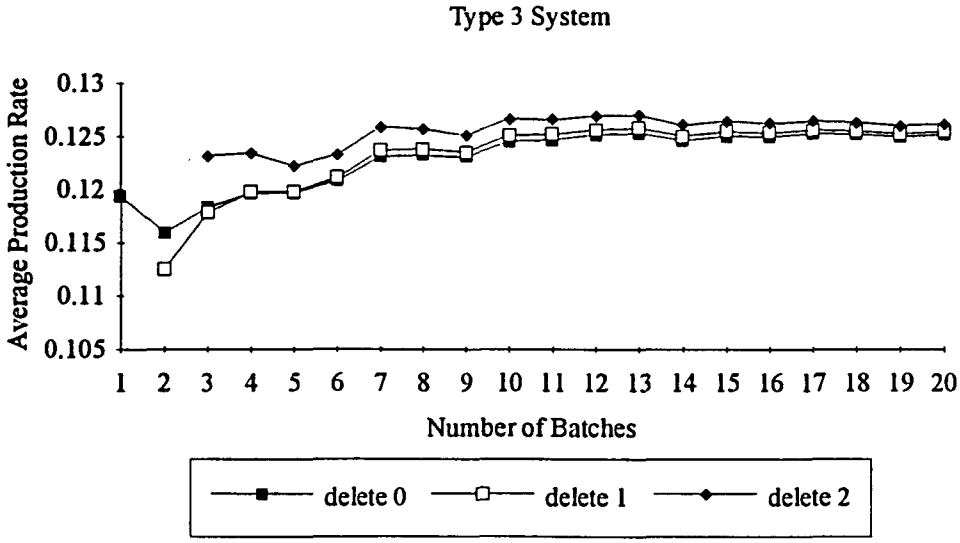

Figure 3.3. Cumulative Averages with Deletions of Type 3 System

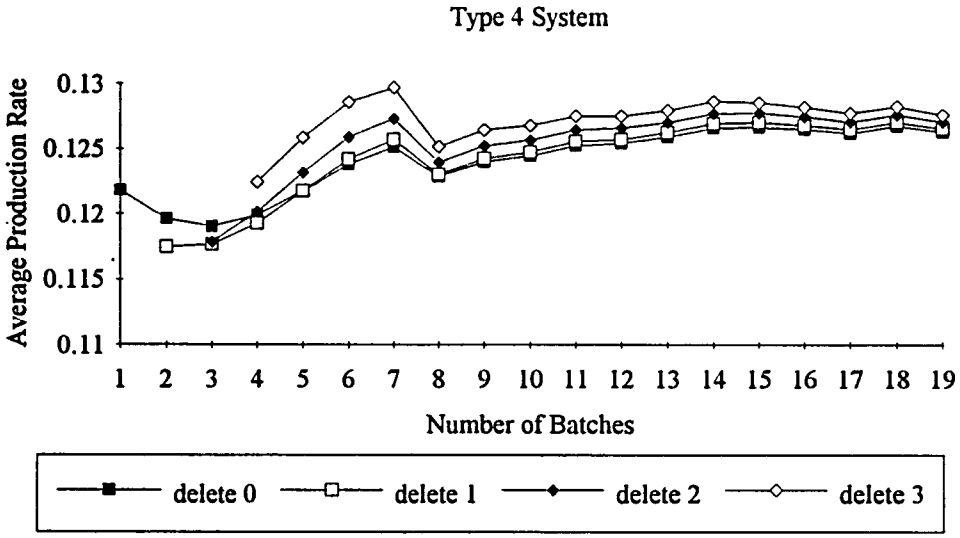

Figure 3.4. Cumulative Averages with Deletions of Type 4 System

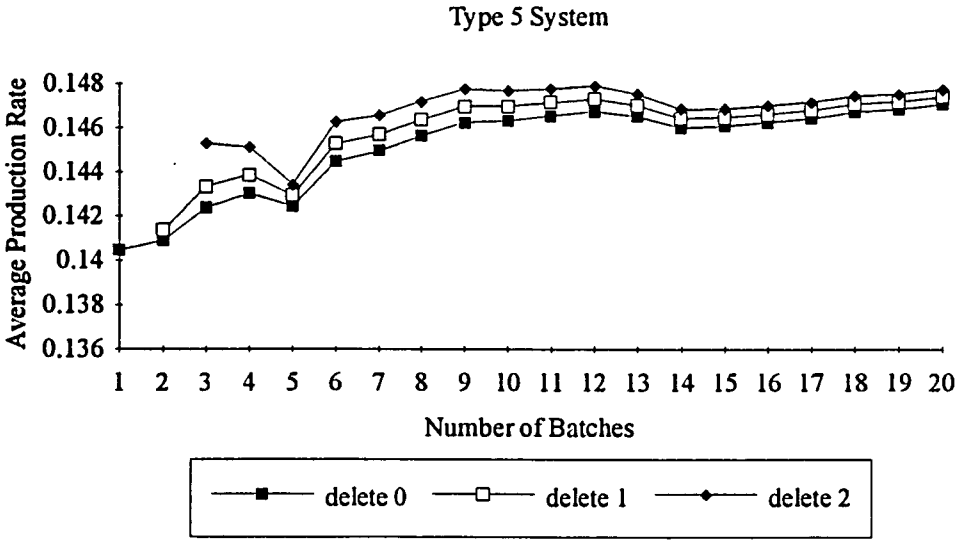

Figure 3.5. Cumulative Averages with Deletions of Type 5 System

each case, the first 10% of each run time was used as a warm-up period. Ten independent replications were made to obtain the confidence intervals.

The input parameters for the five different types of problems were mentioned previously. The confidence interval for each problem is shown in Table 3.2.

Table 3.2. 95% Confidence Intervals of the Production Rate

Problem Type	Liu & Sanders' Model	This Research's Model
1	(0.1426, 0.1454)	(0.1395, 0.1411)
2	(0.1270, 0.1302)	(0.1205, 0.1225)
3	(0.1277, 0.1325)	(0.1225, 0.1245)
4	(0.1243, 0.1297)	(0.1236, 0.1254)
5	(0.1478, 0.1512)	(0.1458, 0.1468)

In order to compare the above results between Liu and Sanders' model and the SIMAN model, an F-test is used to check whether the variances are equal or not. Then, based on the results, a formula is used to calculate the difference between these two systems.

(1) F-test: The procedures of hypothesis testing are

a. Null hypothesis $H_o: \sigma_1^2 = \sigma_2^2$

b. Alternative hypothesis $H_a: \sigma_1^2 \neq \sigma_2^2$

c. Test statistic = $S^2_{larger} / S^2_{smaller}$. The results are shown in table 3.3.

d. The null hypothesis is rejected if the test statistic is larger than the F-statistic. The F-statistic ($F_{\alpha/2, n_1-1, n_2-1}$)= 4.03 under $\alpha = 0.05$ and sample size for both hypotheses is equal to 10.

Table 3.3. Test Statistics

Problem Type	Standard Deviation for Liu & Sanders' Model	Standard Deviation for This Research 's Model	Test Statistic = $\frac{S^2_{larger}}{S^2_{smaller}}$
1	0.001957	0.001154	2.875875
2	0.002237	0.001334	2.812032
3	0.003355	0.001320	6.460069
4	0.003775	0.001279	8.711498
5	0.002377	0.000754	9.938382

e. Conclusion: The hypothesis that the variances of type 1 and 2 are equal cannot be rejected. However, for type 3, 4, and 5, the null hypotheses are rejected.

(2) Calculate the difference: The 95% confidence intervals for the differences on production rate were calculated using the formulas:

a. When the variances are same, $\sigma_1^2 = \sigma_2^2$

$$\overline{Y_2} - \overline{Y_1} \pm t_{\alpha/2,\nu} s.e.(\overline{Y_2} - \overline{Y_1})$$

where t is the t-statistic, degrees of freedom $\nu = n_1 + n_2 - 2$,

$$s.e.(\overline{Y_2} - \overline{Y_1}) = \sqrt{\frac{(n_1-1)s_1^2 + (n_2-1)s_2^2}{n_1+n_2-2}} \sqrt{\frac{1}{n_1} + \frac{1}{n_2}}$$

b. When the variances are different, $\sigma_1^2 \neq \sigma_2^2$

$$\overline{Y_2} - \overline{Y_1} \pm t_{\alpha/2,\nu} s.e.(\overline{Y_2} - \overline{Y_1})$$

where t is the t-statistic,

$$\text{degrees of freedom } \nu = \frac{\left(\frac{s_1^2}{n_1} + \frac{s_2^2}{n_2}\right)^2}{\left[\frac{\left(s_1^2/n_1\right)^2}{(n_1 - 1)}\right] + \left[\frac{\left(s_2^2/n_2\right)^2}{(n_2 - 1)}\right]}$$

$$\text{s.e.}(\overline{Y_2} - \overline{Y_1}) = \sqrt{\frac{s_1^2}{n_1} + \frac{s_2^2}{n_2}}$$

The results are presented in table 3.4. The results show that the confidence intervals of this research are very close to those of Liu and Sanders (1988). The confidence intervals of the differences reveal that there are small differences for all five types of problems. These small differences are tolerable, considering the fragmentary knowledge of their model. Thus, the SIMAN simulation model was evaluated to be acceptable.

Table 3.4. The Difference of Two Systems

Problem Type	Difference
1	(-0.0052, -0.0022)
2	(-0.0088, -0.0054)
3	(-0.0091, -0.0041)
4	(-0.0043, -0.0007)
5	(-0.0049, -0.0015)

3.2. Cost Model

Most of the research related to automatic assembly systems have evaluated the performance by system throughput. In practice, it is not necessary to obtain the maximum production rate as long as the system can reach the production goal. The more utilities work in the system, the higher probability of generating higher production rate will be obtained.

However, it can be too costly and cause a low utilization of the system. Thus, Tandiono (1991) introduced a cost model to evaluate the performance of the closed-loop asynchronous automatic assembly system. She claimed that when the designer uses the manufacturing cost as the performance measure, the number of buffer spaces and the number of pallets can be optimized simultaneously. However, the simultaneous optimization can not be achieved when using system throughput as the performance measure. She assumed that the basic configurations of the automatic assembly system were fixed. Therefore, the cost factors involved in this model included cost of buffer spaces (floor space), cost of pallets, cost of conveyor, cost of work-in-process holding inventory, and penalty cost when the production rate does not meet the required production goal. Since the first three cost factors are initial costs, it is necessary to uniformly annualize the cost function.

In this study, the cost model is almost the same model as that developed by Tandiono. The only difference is the cost of the conveyors is modified due to the practical situation. The total annual cost is defined by the following equation:

$$TTC = P_c + B_c + C_c + H_c + F_c$$

where

TTC = total evaluated annual cost of assembly system,

P_c = cost of pallet per year,

B_c = cost of buffer space per year,

C_c = cost of conveyor per year,

H_c = cost of work-in-process holding inventory per year,

F_c = cost of the penalty per year.

3.2.1. Cost of Pallets

The cost of pallets relies on the number of pallets prepared for the system. Since, the cost of pallets is an initial total cost, an average annual cost is required to compute throughout the life of the pallets. Therefore, an (A/P) factor is used to annualize the cost of the pallets. The cost equation can be written as follows:

$$P_c = N_p * C_p * f$$

where

N_p = Number of pallets,

C_p = the cost of per pallet,

$f = (A/P)^i_n$ factor, a ratio of annual cost and present cost under a certain return rate i and total life n.

3.2.2. Cost of Buffer Space

The cost of the buffer space depends on the total area of buffer space used by the system. The annual cost of buffer space is defined as following equation:

$$B_c = C_{fb} * A_b$$

where

C_{fb} = annual cost of buffer spaces per buffer unit,

A_b = total area of buffer space in buffer units.

As mentioned before, this is a closed system. Therefore, when one buffer unit is added, it does not necessarily cause an increase in the total area equal to the size of one buffer space. Moreover, the buffer space required in the real situation is affected by the actual arrangement. In order to determine the relationship between the total buffer space area and the number of buffer spaces, Tandiono (1991) introduced a multiplication factor called 'Area/Buffer Ratio'. She assumed that the size of a workstation is the same as one buffer unit and the system is arranged in a rectangular shape. For example, for a system including ten workstations and twenty buffer spaces, it can be arranged from 'one by fourteen buffer units' to 'seven by eight buffer units'. After evaluating the data with the statistical linear regression, she found the estimated 'Area/Buffer Ratio' equation as follows:

$$R = 0.2259 + 0.0314(A_b)$$
$$= 0.2259 + 0.0314(B_s + N_s)$$

In this research, it is also assumed that the size of a workstation is the same as one buffer unit. Then, a new equation for annual cost of buffer space is rewritten from the above two equations:

$$B_c = C_{fb} * R * (B_s + N_s)$$
$$= C_{fb} * [0.2259 + 0.0314(B_s + N_s)] * (B_s + N_s)$$

where

B_s = Number of total buffer space,

N_s = Total number of stations,

C_{fb} = Annual cost of buffer space per buffer unit.

3.2.3. Cost of Conveyor

Four factors, namely, number of workstations, number of buffer spaces, size of a buffer space, and type of conveyor, will affect the cost of the conveyor. When the number of workstations and buffer spaces increase, the length of conveyor increases. In Tandiono's (1991) model, she did not consider the length of the conveyor within the workstation. However, in practical situations, it should not be ignored. Also, the cost of conveyor is an initial cost. An (A/P) factor is used to uniformly annualize the cost of the conveyor. Therefore, the annual cost of conveyor is computed as follows:

$$C_c = (B_s + N_s) * C_b * f$$

where

C_b = Cost of conveyor per buffer unit.

3.2.4. Cost of Holding Inventory

The holding cost in this system focuses on the average amount of work-in-process inventory. The number of buffer spaces and pallets in the system will affect the amount of work-in-process inventory. The annual cost of holding inventory is stated as follows:

$$H_c = WIP * H_c$$
$$= WIP * R_h * V_a$$

where

WIP = Average amount of work-in-process inventory,

H_c = Cost of holding inventory per year,

R_h = Holding cost rate per year,

V_a = Value of each assembly.

3.2.5. Penalty Cost

The penalty cost relies on whether the production goal is met or not. It is assumed that the company will stop producing assemblies when the actual number of assemblies meets the required number of assemblies. Therefore, when the actual production rate is larger than the expected production rate, there is no penalty cost. The penalty cost equation is shown as follows:

$$F_c = (N_e - N_a) * C_u$$

where

N_e = Expected number of assemblies required per year,

N_a = Actual number of assemblies produced by the system per year,

C_u = Penalty cost of underproducing per assembly.

According to all above terms, the total cost function could be combined as follows:

$$TTC = N_p * C_p * f + (B_s + N_s)(R * C_{fb} + C_b * f) + WIP * (R_h * V_a) + (N_e - N_a) * C_u$$

This cost function is used as the objective function expected to be minimized in this research. It will be combined with the simulation model and genetic algorithm to evaluate the performance of the closed-loop asynchronous automatic assembly system.

4. THE GENETIC ALGORITHM

This chapter discusses genetic algorithms in detail. Section one introduces the basic mechanisms that consist of genetic algorithms and how these mechanisms work in searching the potential solution space. Section two describes how a simple genetic algorithm was applied to optimize the decision variables of the number of pallets and buffer spaces, as mentioned previously. Moreover, a modified genetic algorithm will be presented.

4.1. General Description of Genetic Algorithms

Genetic algorithms use a simple representation of bit strings (strings of 0s and 1s) to demonstrate a method of representing complicated structures, and the power of simple transformations to improve these bit strings (chromosomes). Each genotype (a single chromosome) would represent a potential solution to a problem, and an evolution process run on a population of chromosomes corresponds to a search through a space of potential solutions. These transformations, based on the mechanics of natural selection, are reproduction, crossover, and mutation. Natural selection is a multi-directional search by keeping a population of potential solutions and promotes information formation and exchange between these directions. The process of natural selection happens in natural systems by which the fittest individuals dominate in the mating pools. Given an objective function, at each generation the relatively "good" solutions reproduce, while the relatively "bad" solutions die.

Genetic algorithms include three basic operators: reproduction, crossover, and mutation. These operators process the population of individuals from generation to generation. The reproduction operator generates a mating pool of individuals by copying their current existing chromosomes with respect to the probability distribution based on fitness

values. Each chromosome has a fitness value and each generation has a total fitness. When the new populations are selected, the crossover operator is applied. There are two major steps for crossover operation. First, random pairs of individuals in the mating pool are mated. This step involves the specified probability of crossover. Second, if crossover is expected to occur, the number that indicates the position of the crossing point is randomly chosen. For example, assume there are two chromosomes being mated as follows:

chromosome 1: 0101|11011

chromosome 2: 1011|00101

Suppose the number that indicates the position of crossing point is 4. Then, the results after crossover are:

new chromosome 1: 0101|00101

new chromosome 2: 1011|11011

If the crossover does not occur, the parents are copied into the new generation. When the crossover is finished, the mutation operator is performed on a bit-by-bit basis. According to the specified mutation probability, every bit has an equal chance to undergo mutation, i.e., change from 0 to 1 or vice versa.

Following selection, crossover, and mutation, the new population is prepared for its next evaluation. This evaluation is applied to construct the probability distribution (for the next selection process). All processes of a simple genetic algorithm are summarized in Figure 4.1.

(1) Initialize a population (from 1 to the specified number of pop_size).

The procedures of reproduction:
(2) Calculate the fitness value f_i for each chromosome c_i (i = 1, ..., pop_size)
(3) Find the total fitness of the population

$$F = \sum_{i=1}^{pop\_size} f_i$$

(4) Calculate the probability p_i of a selection for each chromosome c_i (i = 1, ..., pop_size)

$$p_i = f_i / F$$

(5) Calculate a cumulative probability q_i for each chromosome c_i (i = 1, ..., pop_size)

$$q_i = \sum_{k=1}^{i} p_k$$

(6) Generate a random float number n from the range [0, 1].
(7) Suppose $n < q_1$ then choose the first chromosome (c_1); otherwise choose the $i-th$ chromosome c_i ($2 \le i \le pop\_size$) such that $q_{i-1} < n \le q_i$.

The procedures of crossover:
(8) Generate a random float number n from the range [0, 1].
(9) If $n < p_c$ (p_c is specified crossover probability), choose given chromosome for crossover.

The procedures of mutation:
(10) Generate a random float number n from the range [0, 1].
(11) $n < p_m$ (p_m is specified mutation probability), mutate the bit (change from 0 to 1 or vice versa).

Figure 4.1. The Processes of a Simple Genetic Algorithm

From the abovementioned discussion, it is clear that a genetic algorithm applied to evaluate a particular problem should have the following five components (Michalewicz 1992):

(1) a genetic representation for the potential solutions,

(2) a method to generate initial populations,

(3) an objective function,

(4) genetic operators to alter the formation of children during reproduction,

(5) parameters, such as population size, probabilities of applying genetic operators, etc..

4.2. The Genetic Algorithm in this Research

The theoretical basis of genetic algorithms depend on a binary string representation of solutions, and on the notion of a schema (Goldberg 1989, Michalewicz 1992). Although genetic algorithms explain the reasons of convergence to the optimal solution for a given problem formulation, unfortunately, it does not always follow the theory in practical applications. The reasons, mentioned by Michalewicz (1992) are:

(1) the genetic algorithm assumes an unlimited number of iterations,
(2) the genetic algorithm assumes an unlimited population size, and
(3) the coding of the problem usually shifts the genetic algorithm to operate in a different solution space.

In practical applications, it is impossible to apply the genetic algorithm with infinite population size and number of iterations. In the attempt to find a more effective approach, some researchers combined specific knowledge about their particular problem into a genetic algorithm.

In this research, a classical genetic algorithm and a modified genetic algorithm are both used to find the optimal solution. The classical genetic algorithm implemented for the optimization of the asynchronous automatic assembly system is a simple genetic algorithm presented by Goldberg (1989). This genetic algorithm includes three operators, reproduction, crossover, and mutation, all described previously. The main differences between the classical genetic algorithm used in this research and that presented by Goldberg are the handling of the individual structure and the computer language code. The algorithms used in this research

were written in C language; however, Goldberg used PASCAL to perform the genetic algorithm.

Let P(t) denote populations of potential solutions at generation t, $P(t) = \{f_1^t, f_2^t, \ldots, f_n^t\}$. As mentioned before, the objective in this research is to minimize the objective function. The structure of the modified genetic algorithm is shown in Figure 4.2. The modification with respect to classical genetic algorithm is that in the modified genetic algorithm the best and second best chromosomes of the previous generation are forced into the following generation when the best chromosome is larger than that of the previous generation. If this condition happens, two distinctive chromosomes whose fitness is larger than the average fitness in current generation are randomly selected to die simultaneously.

```
procedure of modified genetic algorithm
begin
t: = 0;
initialize P(t);
evaluate P(t);
find the minimum fitness f(parent) in P(t);
while termination condition is not met do
    begin
        t: = t + 1;
        select parents from P(t-1) according to the fitness;
        produce the offspring P(t) from these parents using genetic operators;
        evaluate P(t);
        find the minimum fitness f(offspring) in P(t);
        while f(offspring) is larger than f(parent)
                select two different chromosomes with the minimum and
                second minimum fitness of P(t-1) to replace randomly two
                different chromosomes having larger fitness  than the average
                value in P(t);
    end;
end
```

Figure 4.2. The Algorithm of Modified Genetic Algorithm

The decision variables applied in this research are the number of pallets and buffer spaces. The number of pallets and buffer spaces are easily encoded into a chromosome. The decimal value of the number of pallets and buffer spaces are converted to their binary equivalent. For instance, the maximum number of pallets in this system is 63 pallets; this can be encoded by a bit chromosome of size 6. The maximum number of buffer spaces for each workstation is 31. This can be encoded as a bit chromosome of size 5. Suppose there are 10 workstations in the system, then each chromosome consists of a bit string of size 56.

In order to perform the genetic algorithm, some initial population of individuals is required. There are several ways to implement this initial population. Some authors suggest randomly generating the initial population, while others propose some special approaches. In this research, the chosen method is to create the initial population randomly.

5. COST OPTIMIZATION

This chapter presents the results of cost optimization from combining the classic or modified genetic algorithm, a cost model, and a simulation model. Before showing the results, it is necessary to predetermine proper operating parameter settings for the algorithms, cost model, and simulation model. Therefore, section one discusses the relative parameters in this research. Section two presents the results.

5.1. Determine Parameters

The optimal solution is to minimize the total annual cost resulting from the sum of pallets and the number of buffer spaces between each pair of workstations. Five different types of closed-loop asynchronous assembly systems with different parameters were evaluated. In order to achieve the objective, it was required to predetermine three types of parameters. Those are: the parameters of the algorithms, the parameters of the cost model, and the parameters in the system and simulation model.

5.1.1. Setting the Cost Parameters

The cost model includes nine parameters: the unit cost of pallet, the annual cost of buffer space per buffer unit, the cost of conveyor per buffer unit, the rate of return, the total life of the system, the annual holding cost rate, the value of each assembly, the expected production rate, and the penalty cost. It is assumed that the life of the system is ten years. The lower bound of the confidence interval of the production rate obtained from the simulation model in chapter 3 is selected as the required production rate for each category. The value of each assembly is $100. Both annual holding cost rate and rate of return are set

at a level of 10%. The other cost parameters, summarized by Tandiono (1991), are presented as follows:

Cost per pallet = $500
Annual cost per unit of buffer space = $1,500
Cost of conveyor per unit of buffer space = $15,000

Now, all parameters are determined except for the unit penalty cost. From the cost model, the value of penalty cost may become the major factor in the objective function when the unit penalty cost is higher. However, when the unit penalty cost is smaller, the cost of buffer spaces or conveyors may become the significant element. Thus, two kinds of unit penalty cost, 10% and 1% of the value of an assembly, were used in the objective function. It was expected that the variety of the total number of pallets and the number of buffer spaces between each pair of workstations under these two different unit penalty costs would be sizable.

5.1.2. Setting the Parameters in System and Simulation Model

This automatic assembly system includes four parameters: the station cycle time, the jam rate, the jam clear time, and the conveyor speed. It is assumed that each workstation has 5 seconds to perform the assembly task. The conveyor transports the pallets at a buffer unit distance every second. Thus, the variables of the system are only the jam rate and the clear time. The jam clear time is a random number with exponential distribution and a mean of 18 or 36 seconds. Both unique jam rate and clear times were combined differentlly to form five categories of asynchronous automatic assembly systems. The type 1 automatic assembly system is a system with uniform stations; that is, all the stations have the same system parameters. Except for the type 1 automatic assembly system, the other four systems have

more complicated combinations in each workstation. The varied combinations of jam rate and clear time in each workstation will be presented in the section 5.2.

Every combination of the total number of pallets and the number of buffer spaces between each pair of workstations generate individual production rates after the SIMAN simulation procedure. Every production rate is used to estimate the penalty cost of that system. Each production rate obtained from the SIMAN simulation is based on a single run length of 13000 seconds with the first 3000 seconds as the warm-up period.

5.1.3. Setting the Genetic Algorithm Operating Parameters

Before running the classical and modified genetic algorithms, four operating parameters, population size, crossover probability, mutation probability, and the run length, need to be determined. For the first three parameters, Goldberg (1989) recommend that a high crossover probability, a low mutation probability (inversely proportional to the population size), and a moderate population size are expected for a good performance of the genetic algorithm. Wellman (1991) tested several different combinations of the parameters: population size, crossover probability, and mutation probability. He concluded that the following values of crossover and mutation probabilities perform well and applied these parameters in a genetic algorithm to evaluate the decision variables of an asynchronous automatic assembly system.

crossover probability: 0.6 or 0.8
mutation probability: 0.001 or 0.005

However, for the parameter of population size, results were very noisy. He used two different population sizes to perform the closed-loop asynchronous automatic assembly system. One consists of population size of 50 and run length of 50 generations. The other

consists of a population size of 100 and run length of 25 generations. He claimed that the genetic algorithm run with a population size of 100 is better than the population size of 50 and performs reasonably well in obtaining good solutions. Since he claimed that the asynchronous automatic assembly system which he used to evaluate the performance is statistically indifferent with those Liu and Sanders (1988) and the system in this research also is very close to those of Liu and Sanders, population size of 100 is selected to perform genetic algorithm in this research. Moreover, considering the computer calculating time, run length of 25 generations is also selected as the parameter to perform the optimization algorithms.

In short, the operating parameters for classic or modified genetic algorithm applied to the closed-loop asynchronous automatic assembly system are summarized as follows:

population size: 100
crossover probability: 0.6
mutation probability: 0.005
run length (number of generations): 25

Another operating parameter used in genetic algorithms is the length of the encoded string. As mentioned previously, the decimal values of the number of pallets and buffer spaces are necessarily converted to their binary equivalent. It was assumed that the maximum total number of pallets in this system is 63 pallets; this can be encoded by a bit chromosome of size 6. The maximum number of buffer spaces for each pair of workstations is 31. This can be converted by a bit chromosome of size 5. For the type 1 automatic assembly system, all workstations have the same characteristics. Therefore, it is expected that the buffer sizes for all stations will be the same. It is enough that only one buffer size and the total number of pallets are encoded into a bit string form. Thus, each chromosome has a bit string of size 11. However, considering the other four systems, there are 10 unique buffers between

workstations in each system. All ten buffer spaces and the total number of pallets are encoded into a bit string form. Thus, each chromosome consists of a bit string of size 56.

5.2. The Results of Cost Optimization

Two kinds of objective functions are used in this research. These two functions are almost identical. The only difference between these two functions is the value of penalty cost. One considered 10% of the value of per assembly as the penalty cost; however, the other considered only 1%. Five different categories of closed-loop asynchronous automatic assembly systems are evaluated. Both objective functions are applied to evaluate the five categories of automatic assembly systems. Four replications are used in both classical and modified genetic algorithm for each category. Each replication is executed under identical conditions. The difference between each replication is the use of different random number seeds. There are four kinds random number seeds. Two of them, called program seeds, are used to perform the genetic algorithm program. The other two, called simulation seeds, are used to run the SIMAN simulation model. Each replication should have one program seed and one simulation seed. Thus, the number of permutations of the two kinds of seeds is four. The remarks of GA1, mGA1, GA2, and mGA2 are used to represent these conditions in the next section. Using the same simulation seed, each pair of GA1 and mGA1, or GA2 and mGA2 performs classic and modified genetic algorithm in identical conditions. The only difference between GA1 and GA2, or mGA1 and mGA2 is the program seed used to perform the algorithms.

5.2.1. Type 1 Asynchronous Automatic Assembly System

For the type 1 asynchronous automatic assembly system, all the workstations in the

system have the same jam rate. The jam clear time is exponentially distributed with a mean of 18 seconds. The parameters used for this system are listed as follows:

Station cycle time = 5 seconds
Transport speed = 1 buffer unit per second
Jam rate (%) = (1, 1, 1, 1, 1, 1, 1, 1, 1, 1)
Exponential mean clear time = 18 seconds
The expected production rate = 0.1395 assemblies per second.

First, the penalty cost of 10% of value per assembly is applied in the objective function to evaluate this type of system. Figure 5.1 and Figure 5.2, classified according to the simulation seed, present the optimal solutions of each generation of the classic genetic algorithm (GA) and modified genetic algorithm (mGA). The details of the cost of each generation are listed in Appendix A. The results of the optimization are shown in Table 5.1.

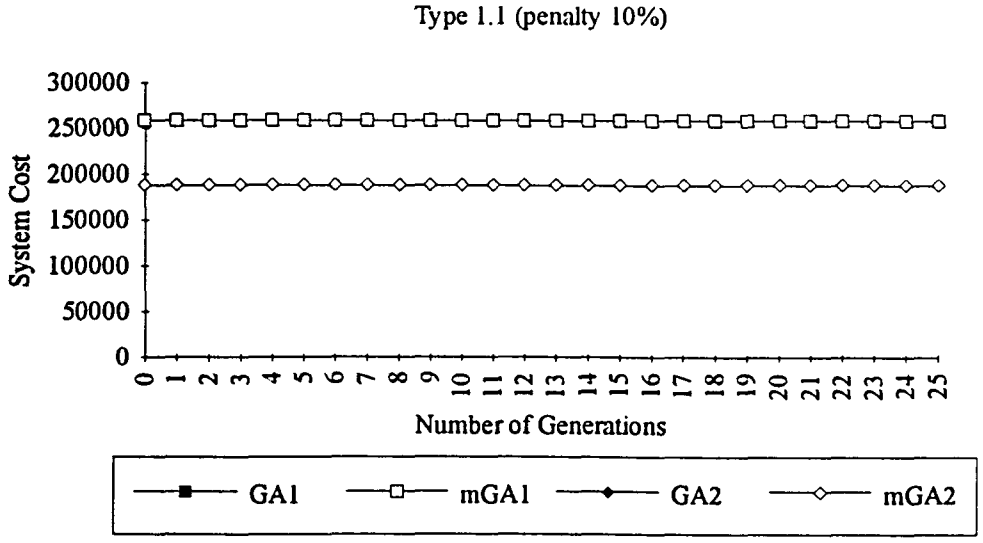

Figure 5.1. Performance of GA and mGA for type 1 system under simulation seed type 1

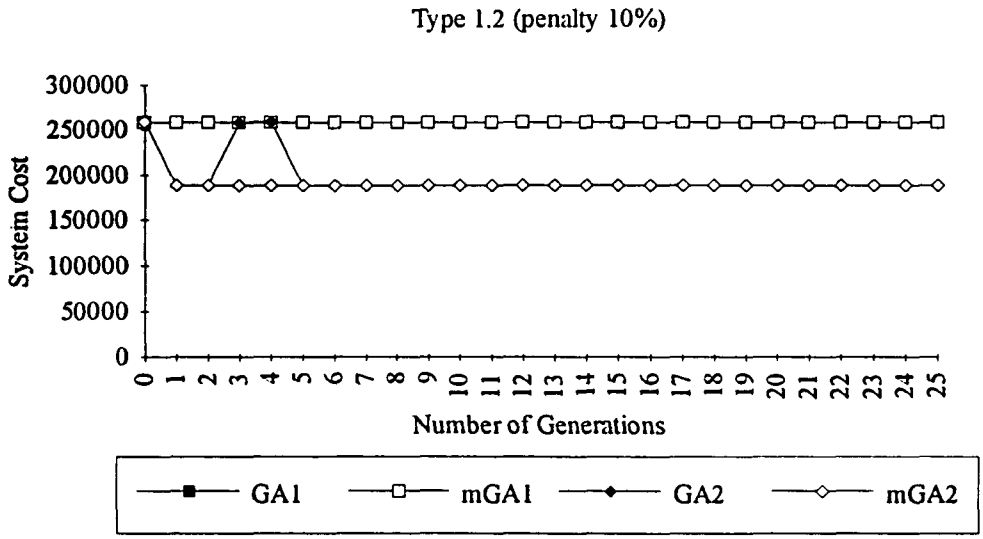

Figure 5.2. Performance of GA and mGA for type 1 system under simulation seed type 2

Table 5.1. Optimization results of type 1 system (penalty 10%)

Type of simulation seed	Run	Buffer sizes 1	2	3	4	5	6	7	8	9	10	Pallets	Prod. Rate	Cost
1	GA1	4	4	4	4	4	4	4	4	4	4	20	0.1396	258582.5
	mGA1	4	4	4	4	4	4	4	4	4	4	20	0.1396	258582.5
	GA2	3	3	3	3	3	3	3	3	3	3	18	0.1400	188208.8
	mGA2	3	3	3	3	3	3	3	3	3	3	18	0.1400	188208.8
2	GA1	4	4	4	4	4	4	4	4	4	4	21	0.1399	258673.9
	mGA1	4	4	4	4	4	4	4	4	4	4	21	0.1399	258673.9
	GA2	3	3	3	3	3	3	3	3	3	3	21	0.1400	188482.9
	mGA2	3	3	3	3	3	3	3	3	3	3	21	0.1400	188482.9

As shown in Figure 5.1 and 5.2, the performance of each pair of the classic and modified genetic algorithms is almost the same. This is the reason why the representative lines of GAs do not appear in these figures. From table 5.1, it appears that the seed parameter significantly affected the results. The results were better with type 2 program seed than with type 1. Since the penalty cost acts as an main factor in the objective function, the combination of the total number of pallets and the number of buffer spaces of each pair of workstations always tends to produce a production rate, higher than or near to the expected production rate.

Then, considering another type of penalty cost, 1% of value of per assembly, the results are presented in Figure 5.3, Figure 5.4 and Table 5.2. Also, Figure 5.3 and Figure 5.4, classified according to the simulation seed, illustrate the solutions of each generation of both the GA and mGA. Table 5.2 summarizes the optimization results. The detailed results are also presented in Appendix A.

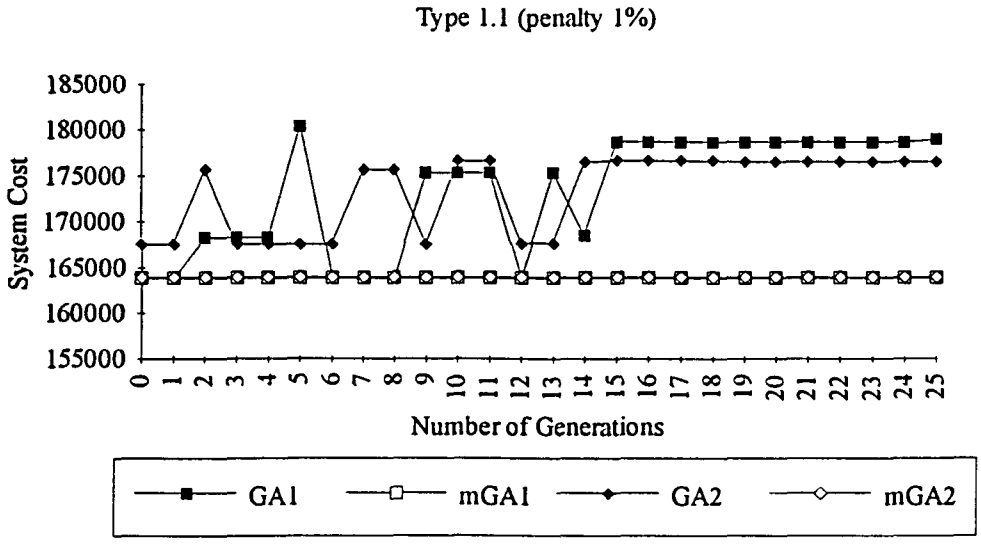

Figure 5.3. Performance of GA and mGA for Type 1 System Under Simulation Seed Type 1 and Penalty Cost 1%

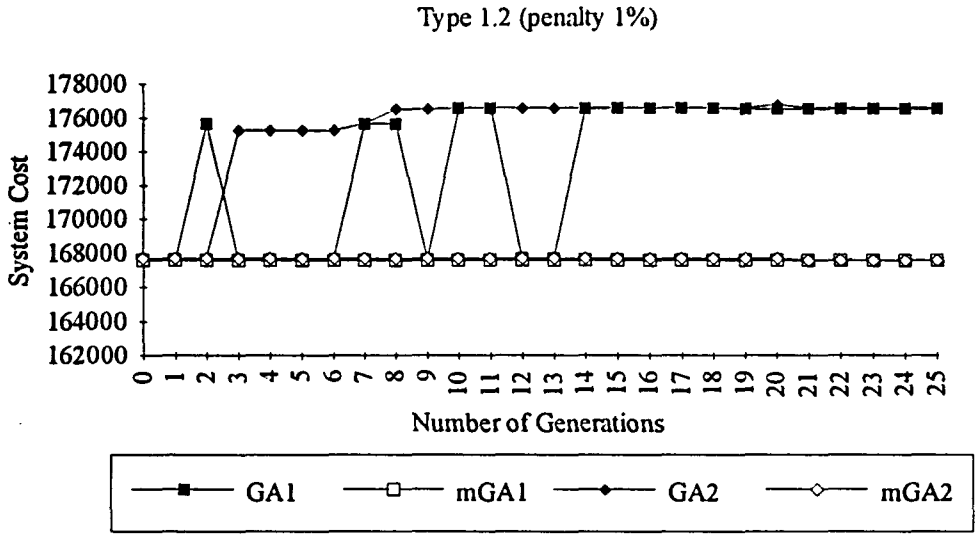

Figure 5.3. Performance of GA and mGA for Type 1 System Under Simulation Seed Type 2 and Penalty Cost 1%

Table 5.2. Optimization results of type 1 system (penalty 1%)

Type of simulation seed	Run	Buffer sizes 1	2	3	4	5	6	7	8	9	10	Pallets	Prod. Rate	Cost
1	GA1	1	1	1	1	1	1	1	1	1	1	13	0.1257	178964.4
	mGA1	2	2	2	2	2	2	2	2	2	2	15	0.1346	163854.9
	GA2	1	1	1	1	1	1	1	1	1	1	10	0.1257	195824.2
	mGA2	2	2	2	2	2	2	2	2	2	2	16	0.1346	163946.3
2	GA1	1	1	1	1	1	1	1	1	1	1	11	0.1260	176535.2
	mGA1	2	2	2	2	2	2	2	2	2	2	15	0.1341	167598.8
	GA2	1	1	1	1	1	1	1	1	1	1	12	0.1260	176626.5
	mGA2	2	2	2	2	2	2	2	2	2	2	15	0.1341	167598.8

As shown in Figure 5.3 and 5.4, the performance between each pair of the classical and modified genetic algorithms is significantly different. Since the performance of mGA1

and mGA2 is similar under the same simulation seed, both representative lines in the figure are duplicate. Comparing the cost result of the last generation, clearly, the performance of mGA is better than that of GA under identical conditions. Since the penalty cost was changed to 1% value of per assembly, the penalty cost is not the main factor in the objective function. The results show that the cost of buffer spaces and conveyor replace the penalty cost as the major elements in the objective function. The genetic algorithm tends to find the number of buffer spaces as small as possible under some reasonable penalty cost. The actual production rates of the final generation are always smaller than the expected production rates.

5.2.2. Type 2 Asynchronous Automatic Assembly System

In the type 2 automatic assembly system, it is assumed that some workstations have positive jam rates and the others have zero jam rates. The jam clear time is exponentially distributed with a mean 36 seconds. The parameters involved in this system are listed as follows:

Station cycle time = 5 seconds
Transport speed = 1 buffer unit per second
Jam rate (%) = (0, 3, 3, 0, 0, 0, 3, 0, 0, 0)
Exponential mean clear time = 36 seconds
The expected production rate = 0.1205 assemblies per second.

Also, the penalty cost of 10% of value per assembly is first applied in the objective function to evaluate this type system. Figure 5.5 and 5.6, classified according to the simulation seed, show the optimal solutions of each generation of the GA and mGA. The detail of the cost of each generation for both algorithms is listed in Appendix A. The results of the optimization are shown in Table 5.3.

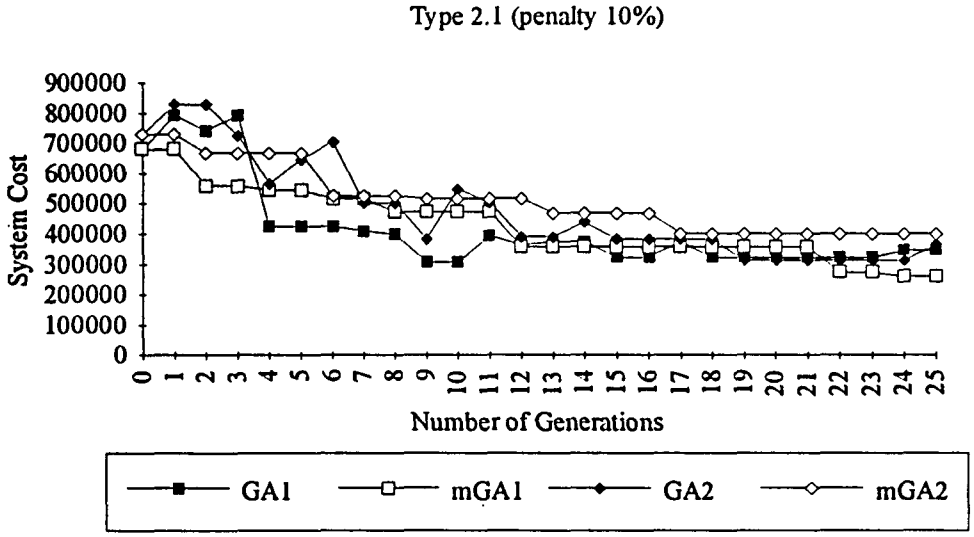

Figure 5.5. Performance of GA and mGA for Type 2 System Under Simulation Seed Type 1 and Penalty Cost 10%

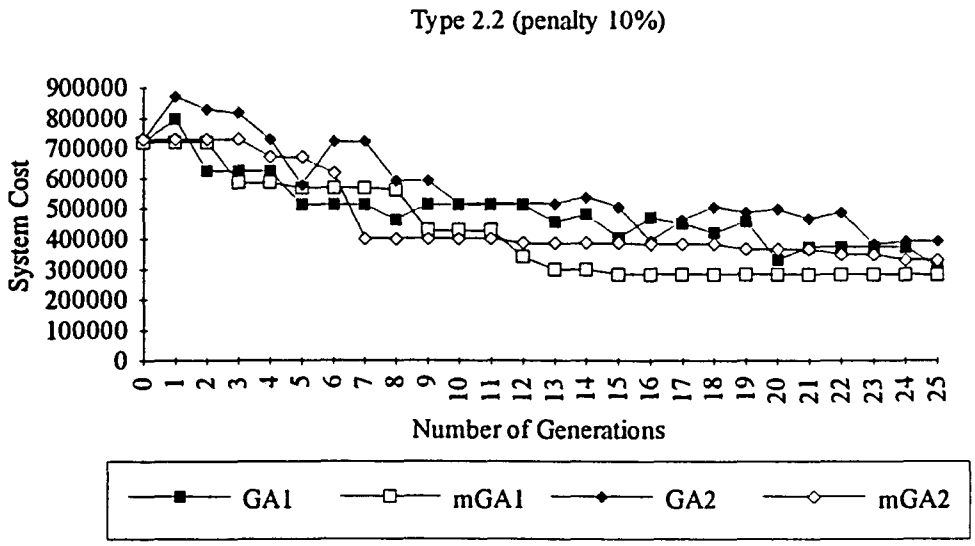

Figure 5.6. Performance of GA and mGA for Type 2 System Under Simulation Seed Type 2 and Penalty Cost 10%

Table 5.3. Optimization Results of Type 2 System (penalty 10%)

Type of simulation seed	Run	Buffer sizes 1	2	3	4	5	6	7	8	9	10	Pallets	Prod. Rate	Cost
1	GA1	2	8	18	9	9	1	1	1	1	1	36	0.1216	348134.7
	mGA1	3	6	10	1	1	2	2	6	2	7	48	0.1212	261141.0
	GA2	4	9	13	11	1	1	1	1	2	10	38	0.1221	365558.4
	mGA2	1	7	9	4	1	15	4	1	3	12	47	0.1209	401993.2
2	GA1	1	1	11	4	1	3	17	1	3	4	46	0.1210	307594.5
	mGA1	3	3	1	1	5	4	6	6	3	5	41	0.1199	283383.0
	GA2	5	1	23	3	1	7	1	5	1	3	60	0.1198	394264.5
	mGA2	2	11	12	3	1	1	4	1	1	13	44	0.1209	332001.5

As shown in Figure 5.5 and 5.6, the performance of each pair of the GA and mGA is different. Comparing the results, the solutions obtained from the mGA are not always superior to those from GA. However, the mGA generally performed better than the GA did. Again, since the penalty cost plays an important part in the objective function, the production rate is usually higher than or near the expected production rate.

Next, considering 1% of value per assembly as the penalty cost in objective function, the results are presented in Figure 5.7, Figure 5.8 and Table 5.4. Also, Figure 5.7 and Figure 5.8, grouped under the simulation seed, illustrate the solutions of each generation for both the GA and mGA. Table 5.4 shows the optimization results. The detailed results are collected in Appendix A.

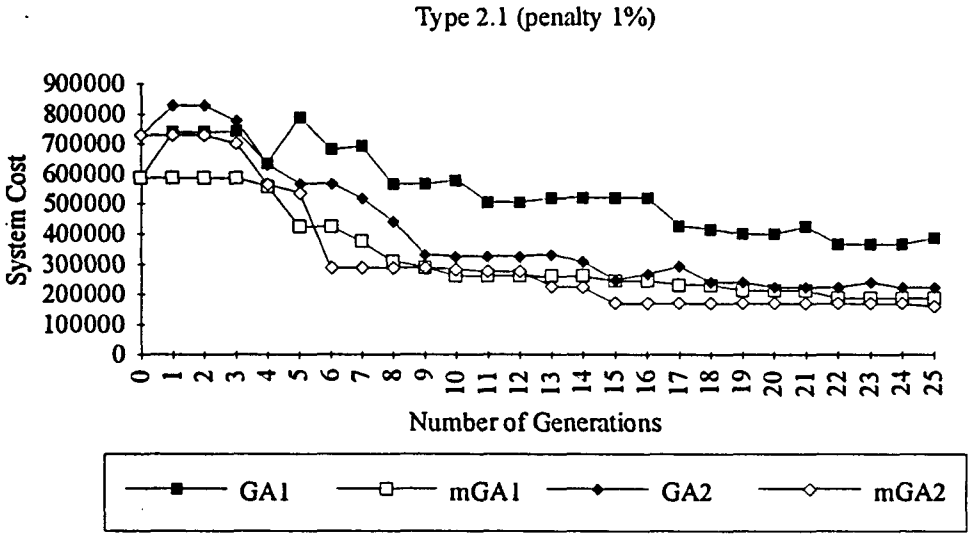

Figure 5.7. Performance of GA and mGA for Type 2 System Under Simulation Seed Type 1 and Penalty Cost 1%

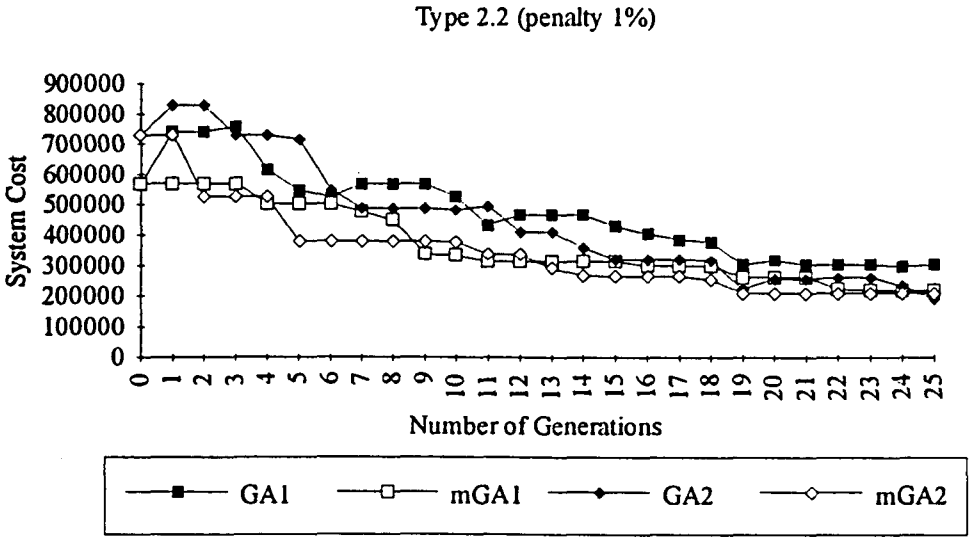

Figure 5.8. Performance of GA and mGA for Type 2 System Under Simulation Seed Type 2 and Penalty Cost 1%

Table 5.4. Optimization Results of Type 2 System (penalty 1%)

Type of simulation seed	Run	Buffer sizes 1	2	3	4	5	6	7	8	9	10	Pallets	Prod. Rate	Cost
1	GA1	1	1	2	13	11	5	6	8	3	2	51	0.1166	387281.9
	mGA1	1	2	3	1	8	2	3	3	2	1	30	0.1172	188576.8
	GA2	3	2	1	1	1	8	6	1	1	1	32	0.1118	223070.6
	mGA2	7	3	3	2	1	3	1	1	1	1	18	0.1185	159655.9
2	GA1	3	1	1	10	7	1	1	5	9	5	53	0.1176	303960.9
	mGA1	3	1	3	6	1	1	4	1	1	7	28	0.1144	219645.1
	GA2	2	1	2	2	1	5	1	1	3	3	18	0.1125	190748.6
	mGA2	3	5	3	2	1	1	4	1	3	1	34	0.1124	210484.1

As shown in Figure 5.7, Figure 5.8 and Table 5.4, the performance of mGA is generally better than that of GA under identical conditions. Again, the results show that the cost of buffer spaces and conveyor replace the penalty cost as the major elements in the objective function. The genetic algorithms can optimize the number of buffer spaces as small as possible under some reasonable penalty cost. The actual production rates of the final generation are always smaller than the expected production rate.

5.2.3. Type 3 Asynchronous Automatic Assembly System

In the type 3 automatic assembly system, some workstations have positive jam rates and the others have zero jam rates. The jam clear time is exponentially distributed with a mean of 36 seconds. The parameters involved in this system are collected as follows:

Station cycle time = 5 seconds
Transport speed = 1 buffer unit per second
Jam rate (%) = (0, 3, 0, 3, 0, 3, 0, 0, 0, 0)

Exponential mean clear time = 36 seconds
The expected production rate = 0.1225 assemblies per second.

Also, the penalty cost of 10% of value per assembly is applied to optimize the cost of this type system. Figure 5.9 and 5.10, classified under simulation seed, show the optimal solutions of each generation of the GA and mGA. The details of the cost of each generation of both algorithms are listed in Appendix A. The results of the optimization are arranged in Table 5.5.

Comparing the final results, the solutions obtained from the mGA are superior to that acquired from GA. Also, since the penalty cost plays a major role in the objective function, the production rates are higher than or near the expected production rate.

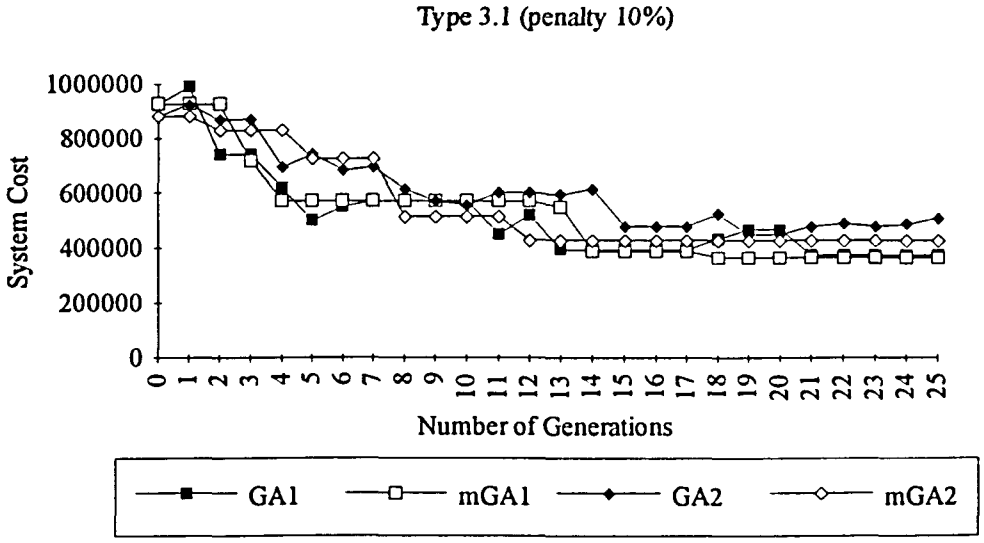

Figure 5.9. Performance of GA and mGA for Type 3 System Under Simulation Seed Type 1 and Penalty Cost 10%

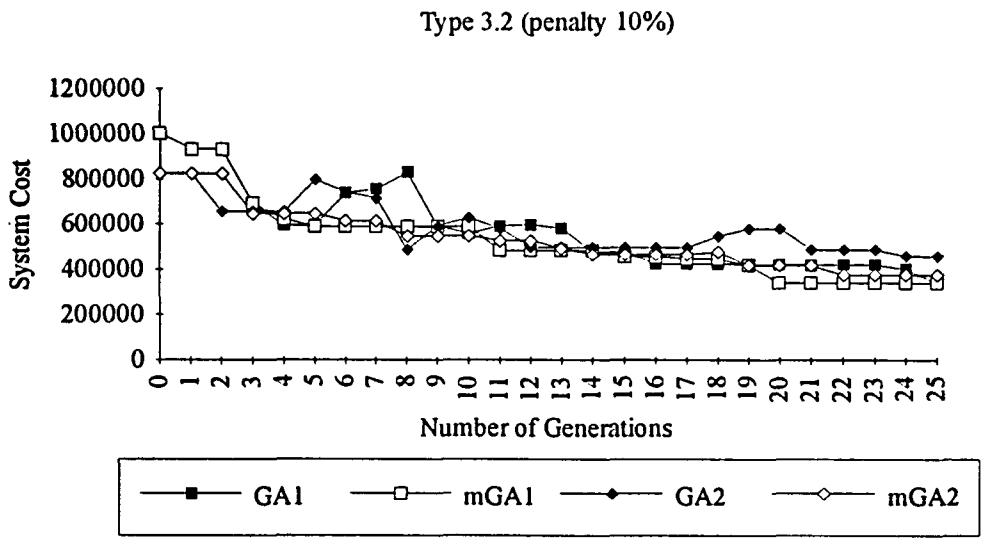

Figure 5.10. Performance of GA and mGA for Type 3 System Under Simulation Seed Type 2 and Penalty Cost 10%

Table 5.5. Optimization Results of Type 3 (penalty 10%)

Type of simulation seed	Run	Buffer sizes 1	2	3	4	5	6	7	8	9	10	Pallets	Prod. Rate	Cost
1	GA1	6	1	2	15	8	2	5	5	3	7	50	0.1244	375416.8
	mGA1	2	6	11	6	9	1	6	3	2	7	45	0.1267	366198.1
	GA2	1	7	1	10	8	15	2	4	13	1	57	0.1217	509446.1
	mGA2	1	5	6	12	1	15	4	3	4	9	28	0.1267	427955.5
2	GA1	2	4	9	1	8	7	11	3	1	3	44	0.1224	339489.6
	mGA1	2	1	1	10	9	3	5	5	4	9	44	0.1224	339489.6
	GA2	5	5	18	1	16	1	4	5	1	7	54	0.1226	458877.5
	mGA2	1	9	12	7	8	1	4	1	1	5	40	0.1219	376564.2

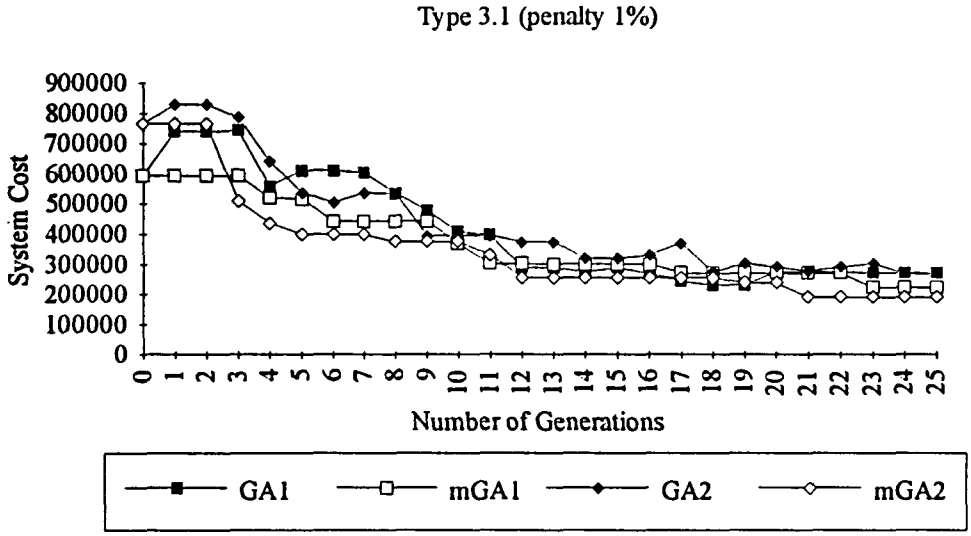

Figure 5.11. Performance of GA and mGA for Type 3 System Under Simulation Seed Type 1 and Penalty Cost 1%

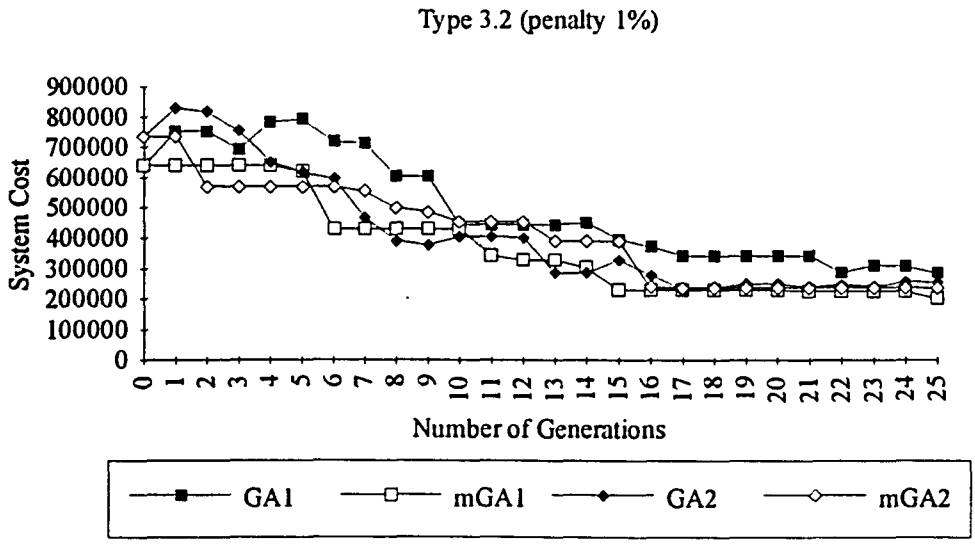

Figure 5.12. Performance of GA and mGA for Type 3 System Under Simulation Seed Type 2 and Penalty Cost 1%

Next, 1% of value per assembly is considered as the penalty cost in the objective function. Figure 5.11 and 5.12, grouped under the simulation seed, illustrate the solutions of each generation of both the GA and mGA. The optimization results are shown in Table 5.6. The detailed results are collected in Appendix A.

Table 5.6. Optimization Results of Type 3 System (penalty 1%)

Type of simulation seed	Run	Buffer sizes 1	2	3	4	5	6	7	8	9	10	Pallets	Prod. Rate	Cost
1	GA1	1	1	5	1	2	10	1	1	1	3	36	0.1086	268497.9
	mGA1	1	2	1	1	7	1	6	2	2	3	36	0.1149	221323.5
	GA2	3	4	1	1	1	11	1	1	2	4	39	0.1109	270487.4
	mGA2	2	2	6	3	1	1	1	3	1	1	31	0.1153	188192.5
2	GA1	2	3	6	10	1	4	4	1	3	7	28	0.1201	284821.9
	mGA1	1	4	1	1	9	4	1	3	1	1	22	0.1174	201324.3
	GA2	2	9	2	2	5	1	1	5	1	6	26	0.1172	255572.2
	mGA2	1	11	1	3	1	3	1	2	7	1	35	0.1172	236043.7

Comparing the cost results, the performance of mGA is better than that of GA under identical conditions. From Table 5.6, the program seed used in GA and mGA only slightly affected the final results under identical conditions. Again, the results show that the cost of buffer spaces and conveyor replace the penalty cost as the major elements in the objective function. The genetic algorithms tend to find the number of buffer spaces as small as possible under some reasonable production rates. Thus, the production rates are always smaller than the required production rate.

5.2.4. Type 4 Asynchronous Automatic Assembly System

The type 4 asynchronous automatic assembly system also consists of some workstations have positive jam rates and the others have zero jam rates. However, the positive jam rates are not always the same value. The jam clear time is exponentially distributed with a mean 36 seconds. The relative parameters are listed as follows:

Station cycle time = 5 seconds
Transport speed = 1 buffer unit per second
Jam rate (%) = (0, 3, 0, 0, 2, 0, 0, 2, 0, 0)
Exponential mean clear time = 36 seconds
The expected production rate = 0.1236 assemblies per second.

Also, the penalty cost of 10% of value per assembly is applied to evaluate the system cost. Figure 5.13 and 5.14, classified under simulation seed, show the optimal solutions of each generation of the GA and mGA. The detail of the optimal cost of each generation of both algorithms are listed in Appendix A. The results of the optimization are shown in Table 5.7.

From Figure 5.13, Figure 5.14, and Table 5.7, it is clear that the performance of the mGA is better than that GA in this system. According to Table 5.7, the program seed only slightly affected the results obtained from both algorithms. Comparing the final results, the solutions obtained from the mGA are much better than that attained from GA. Once more, since the penalty cost is a principal factor in the objective function, the higher production rates are always generated to avoid the penalty cost of underproducing.

Subsequently, the penalty cost of 1% of value per assembly is used in the objective function. Figure 5.15 and 5.16, grouped under the simulation seed, illustrate the solutions of each generation of both the GA and mGA. The optimization results are presented in Table 5.8. The detailed results are collected in Appendix A.

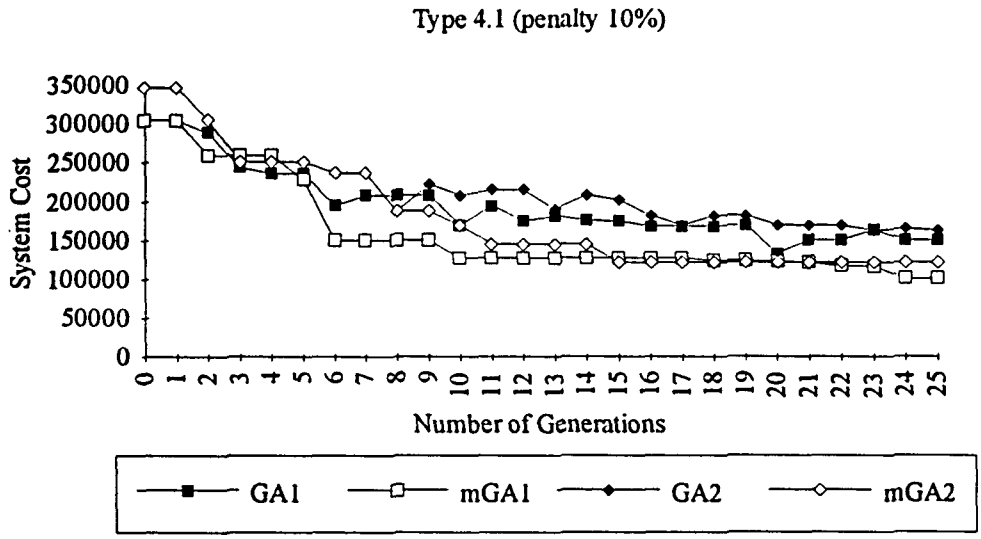

Figure 5.13. Performance of GA and mGA for Type 4 System Under Simulation Seed Type 1 and Penalty Cost 10%

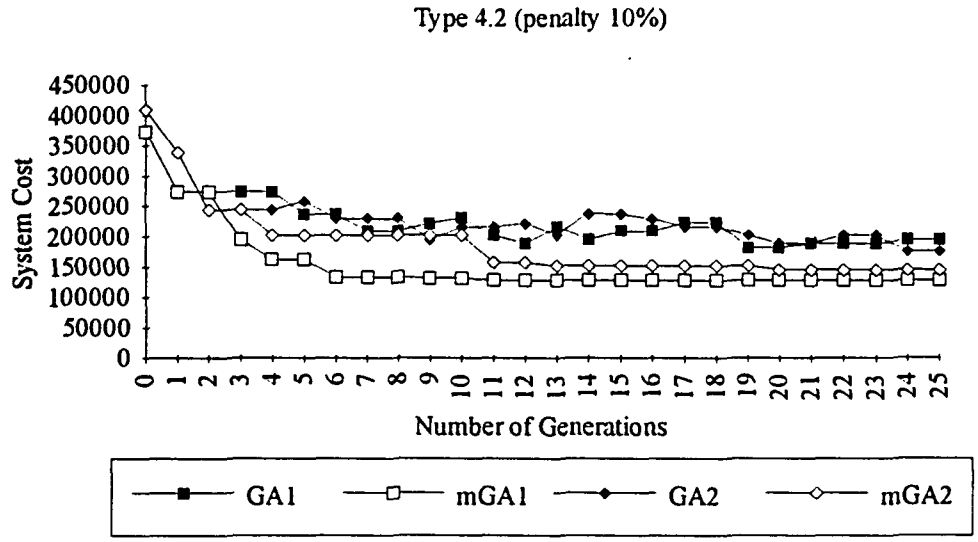

Figure 5.14. Performance of GA and mGA for Type 4 System Under Simulation Seed Type 2 and Penalty Cost 10%

Table 5.7. Optimization Results of Type 4 System (penalty 10%)

Type of simulation seed	Run	Buffer sizes 1	2	3	4	5	6	7	8	9	10	Pallets	Prod. Rate	Cost
1	GA1	3	3	4	1	2	4	4	1	1	1	17	0.1252	150524.4
	mGA1	1	1	2	4	2	1	1	1	1	1	25	0.1289	101224.4
	GA2	2	4	1	5	5	4	1	2	1	1	23	0.1265	163226.8
	mGA2	2	1	1	1	4	2	3	1	1	3	23	0.1237	122335.6
2	GA1	1	3	1	1	2	9	2	3	2	7	24	0.1249	195352.2
	mGA1	1	1	1	3	7	2	1	2	1	1	17	0.1241	127346.4
	GA2	1	2	3	2	4	4	1	6	4	1	28	0.1265	176214.7
	mGA2	2	1	1	2	1	4	1	2	7	2	20	0.1267	144862.7

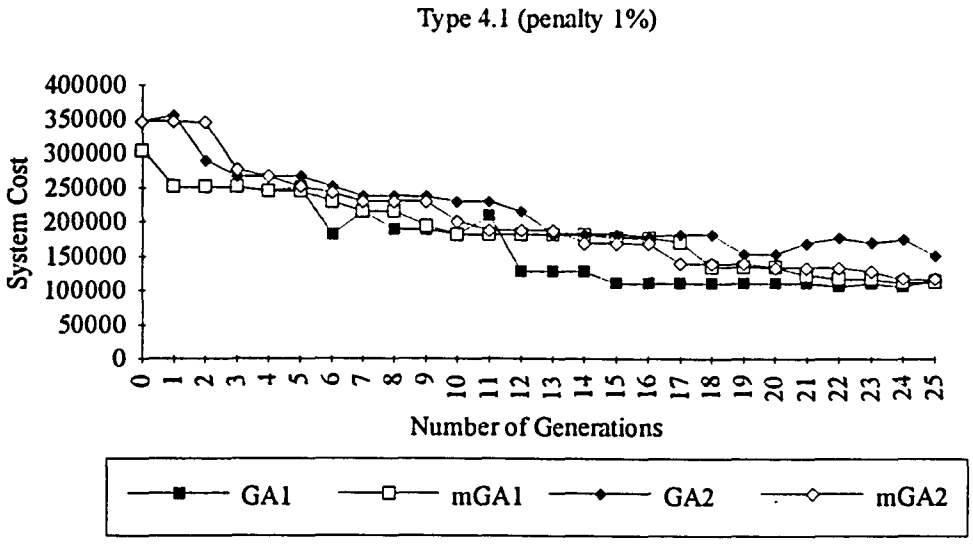

Figure 5.15. Performance of GA and mGA for Type 4 System Under Simulation Seed Type 1 and Penalty Cost 1%

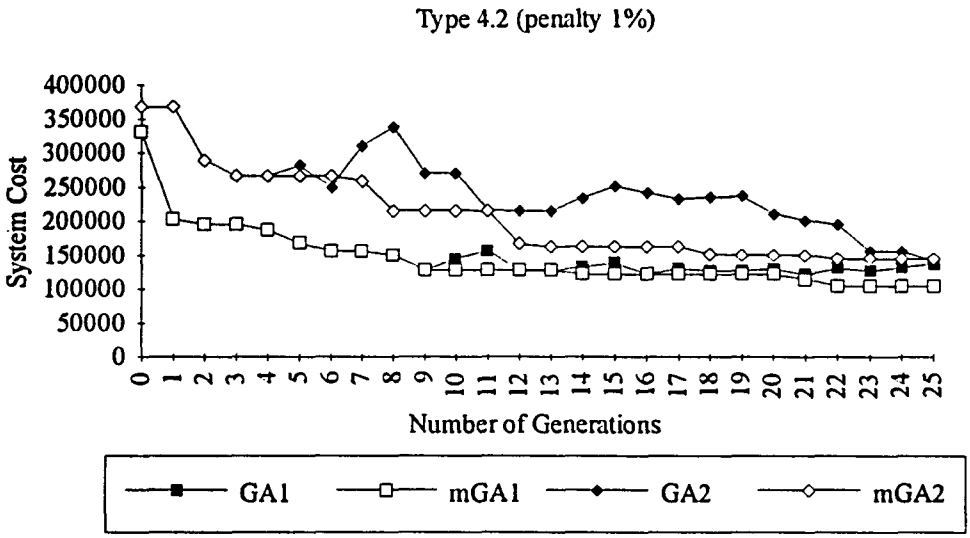

Figure 5.16. Performance of GA and mGA for Type 4 System Under Simulation Seed Type 2 and Penalty Cost 1%

Table 5.8. Optimization Results of Type 4 System (penalty 1%)

Type of simulation seed	Run	Buffer sizes 1	2	3	4	5	6	7	8	9	10	Pallets	Prod. Rate	Cost
1	GA1	2	1	3	3	3	1	1	2	1	1	16	0.1268	116231.2
	mGA1	3	1	4	1	1	1	1	1	1	2	26	0.1228	112488.3
	GA2	3	1	1	2	3	5	3	1	4	1	31	0.1278	151803.6
	mGA2	1	1	2	1	1	4	3	2	1	2	28	0.1282	117327.7
2	GA1	3	4	4	1	2	1	2	3	1	1	17	0.1247	138747.0
	mGA1	3	1	1	1	1	3	1	2	1	1	16	0.1230	104894.8
	GA2	2	1	1	1	1	5	2	3	4	2	27	0.1233	141907.1
	mGA2	1	2	5	2	4	2	1	1	4	1	24	0.1242	145228.2

Comparing the cost results, the performance of mGA is generally better than that of GA under identical conditions. From Table 5.8, it is apparent that the type of program seed can affect the results. The results indicate that type 1 program seed performs better than type 2 program seed. Since the number of buffer spaces in this system is small, the cost of the buffer spaces and the conveyor should not be too large. Although the penalty cost is only 1% of value per assembly, it still acts as a primary factor in the objective function. Therefore, the production rates that are higher than or near to expected production rate are expected. However, there exists a larger potential solution space than the system that uses 10% of value per assembly as the penalty cost. Comparing the production rate of the 10% penalty cost system, a much lower production rate could be generated to optimize this 1% penalty cost system.

5.2.5. Type 5 Asynchronous Automatic Assembly System

In the type 5 asynchronous automatic assembly system, it was considered that all workstations have positive jam rates of 0.5% or 3%. The jam clear time is exponentially distributed with a mean 18 seconds. The parameters required in this system are listed as follows:

Station cycle time = 5 seconds
Transport speed = 1 buffer unit per second
Jam rate (%) = (0.5, 3, 0.5, 0.5, 0.5, 0.5, 3, 0.5, 0.5, 0.5)
Exponential mean clear time = 18 seconds
The expected production rate = 0.1458 assemblies per second.

Once more, the penalty cost of 10% of value per assembly is applied to optimize the cost of this type system. Figure 5.17 and 5.18, classified under simulation seed, show the solutions of each generation of the GA and mGA. The details of the cost of each generation

of both algorithms are listed in Appendix A. The results of the optimization are presented in Table 5.9.

As shown in Figure 5.17, Figure 5.18 and Table 5.9, performance of mGA is better than that of GA. Again, since the penalty cost is the weightiest factor in the objective function, the production rates that are higher than or near to expected production rate are usually produced.

Following, the 1% of value per assembly is considered as the penalty cost in the objective function. Figure 5.19 and 5.20, grouped under the simulation seed, illustrate the solutions of each generation of both the GA and mGA. The optimization results are shown in Table 5.10. The detailed results are collected in Appendix A.

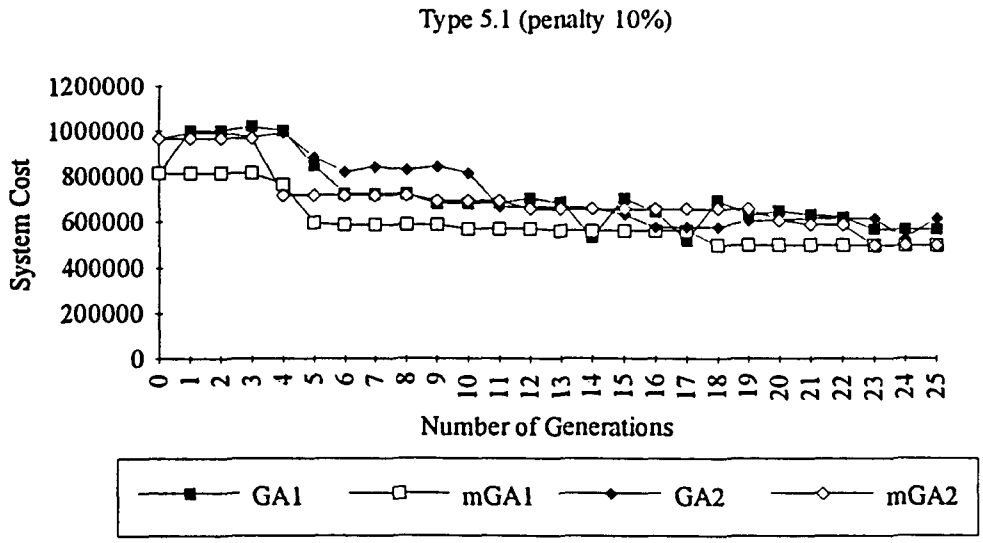

Figure 5.17. Performance of GA and mGA for Type 5 System Under Simulation Seed Type 1 and Penalty Cost 10%

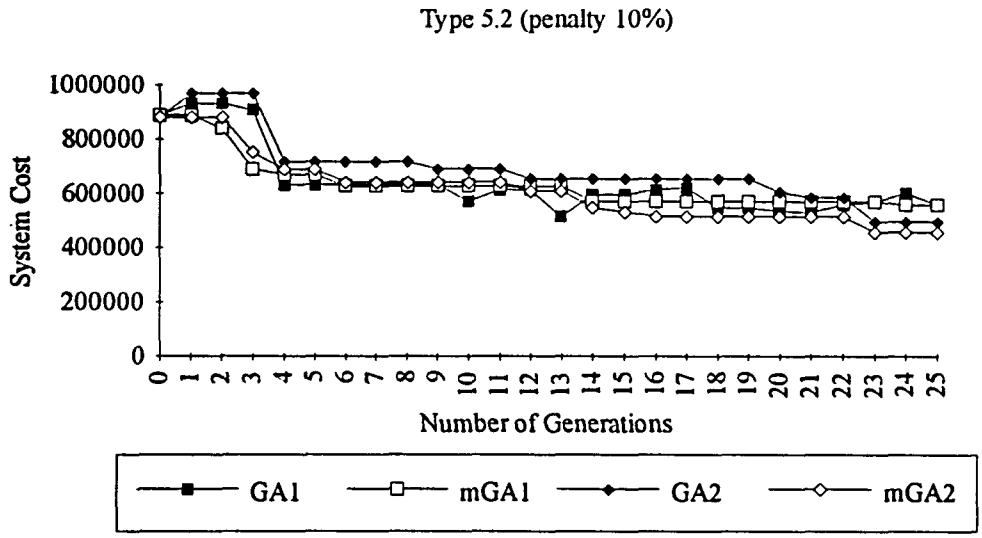

Figure 5.18. Performance of GA and mGA for Type 5 System Under Simulation Seed Type 2 and Penalty Cost 10%

Table 5.9. Optimization Results of Type 5 system (penalty 10%)

Type of simulation seed	Run	Buffer sizes 1 2 3 4 5 6 7 8 9 10	Pallets	Prod. Rate	Cost
1	GA1	1 14 14 5 11 1 6 8 7 6	48	0.1457	567094.2
	mGA1	4 12 13 2 7 1 9 11 7 1	44	0.1459	497344.1
	GA2	6 9 12 10 1 7 5 10 7 5	47	0.1449	616355.2
	mGA2	9 15 10 1 16 1 5 6 3 1	42	0.1460	497161.3
2	GA1	12 10 6 5 6 8 6 5 6 9	54	0.1467	560154.4
	mGA1	2 11 4 5 11 8 6 3 9 5	50	0.1446	558071.8
	GA2	3 17 23 11 2 3 4 5 13 2	50	0.1478	670485.9
	mGA2	7 2 23 4 1 7 3 1 3 12	42	0.1461	457781.0

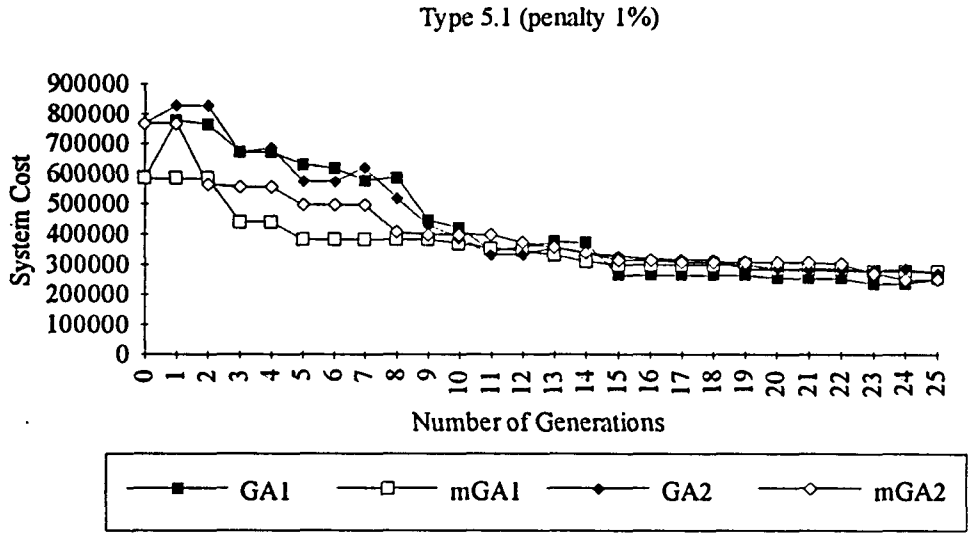

Figure 5.19. Performance of GA and mGA for Type 5 System Under Simulation Seed Type 1 and Penalty Cost 1%

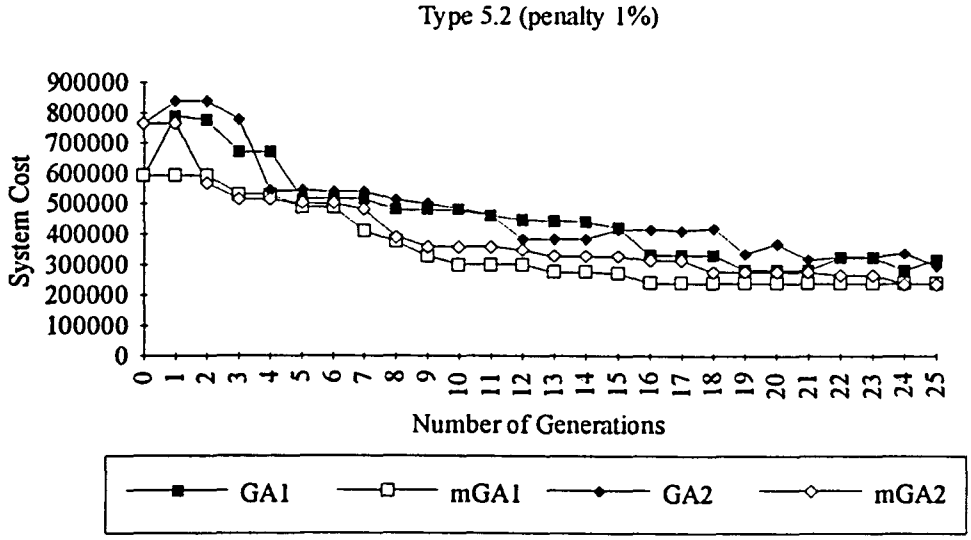

Figure 5.20. Performance of GA and mGA for Type 5 System Under Simulation Seed Type 2 and Penalty Cost 1%

Table 5.10. Optimization Results Of Type 5 System (penalty 1%)

Type of simulation seed	Run	Buffer sizes 1	2	3	4	5	6	7	8	9	10	Pallets	Prod. Rate	Cost
1	GA1	3	1	1	1	5	6	1	5	5	3	21	0.1380	253484.5
	mGA1	3	2	1	1	5	4	2	1	3	9	41	0.1349	278524.8
	GA2	1	1	5	2	4	3	4	1	1	8	31	0.1346	273262.2
	mGA2	2	3	1	3	3	1	4	1	3	6	20	0.1351	249292.6
2	GA1	2	1	1	1	7	1	8	4	6	7	48	0.1361	318982.8
	mGA1	3	6	1	1	4	1	9	1	2	2	23	0.1391	238835.2
	GA2	4	1	5	7	1	10	1	1	1	8	27	0.1396	298204.7
	mGA2	2	1	1	1	1	1	4	2	3	3	29	0.1306	236670.5

Comparing the cost results in Table 5.10, the performance of mGA is better than that of GA under identical conditions except for the first solution. The results show that the cost of buffer spaces and conveyor are the major elements in the objective function. The genetic algorithms try to find the number of buffer spaces as small as possible under some reasonable lower production rates. Thus, the production rates on Table 5.10 are smaller than the required production rate to reduce the number of buffer spaces.

5.3. Results Between Minimum Cost and Maximum Throughput

Liu and Sanders (1988) applied a hybrid algorithm including an analytical model and the SQG algorithm to find the optimal number of pallets and buffer spaces under the goal of maximizing the system throughput. They first used the analytical model to determine the total number of pallets in each system. By using different numbers of pallets in each system, they then utilized the SQG algorithm to search for the number of buffer spaces to maximize the throughput. Since a system with more pallets and buffer spaces results in a higher production

rates, the optimal solutions of the number of pallets and buffer spaces will be indefinite when try to optimize the number of pallets and buffer spaces simultaneously. However, considering a cost model, we are able to optimize both parameters simultaneously. Although more pallets reduce the starvation effects and more buffer spaces decrease the blocking effects, the size of the system is limited by the economic factors. Thus, the optimization processes can be accomplished concurrently.

In order to validate the results performed by genetic algorithms, the cost of those solutions in this research is compared with the cost of Liu and Sanders' solutions. The solutions of Liu and Sanders is put into the same SIMAN simulation model to evaluate their production rate. Also, the cost is calculated through the same cost model. The comparisons are made for all five categories of asynchronous automatic assembly systems. The results are summarized in Table 5.11 and 5.12.

Table 5.11. The Results of Minimum Cost Model and Maximum Throughput (penalty 10%)

Penalty 10%	Simulation Seed Type 1			Simulation Seed Type 2		
Type of System	Solution of GA1	Solution of mGA1	Result Using Liu and Sanders' Solution	Solution of GA2	Solution of mGA2	Result of Liu and Sanders' Solution
1	188208.8	188208.8	188391.5	188482.9	188482.9	285735.5
2	348134.7	261141.0	410492.2	307594.5	283383.0	410492.2
3	375416.8	366198.1	623937.7	339489.6	339489.6	975873.7
4	150524.4	101224.4	256770.5	176214.7	127346.4	181890.5
5	567094.2	497161.3	727001.8	560154.4	457781.0	682073.8

Table 5.12. The Results of Minimum Cost Model and Maximum Throughput (penalty 1%)

Penalty 1%	Simulation Seed Type 1			Simulation Seed Type 2		
Type of System	Solution of GA1	Solution of mGA1	Result Using Liu and Sanders' Solution	Solution of GA2	Solution of mGA2	Result Using Liu and Sanders' Solution
1	178690.2	163854.9	188391.5	176535.2	167598.8	198125.9
2	225903.2	159655.9	410492.2	190748.6	210484.1	410492.2
3	268497.9	190294.1	516110.5	255572.2	201324.3	551304.1
4	116231.2	112488.3	189378.5	138747.0	104894.8	181890.5
5	253484.5	249292.6	686566.6	298204.7	236670.5	682073.8

From Table 5.11 and 5.12, it is clear that the costs of all five problems in this research are lower than the costs using Liu and Sanders' solutions. In order to obtain the maximum system production rate, Liu and Sanders' algorithm overestimated the buffer sizes and resulted in larger systems with higher costs. When considering a cost model, it is able to design a system that not only has a lower system cost but also could reach the production goal.

5.4. Summary

Although the genetic algorithm is created to solve deterministic objective function, it has been demonstrated that it is able to search for the optimal solution to a stochastic system. Generally speaking, the performance of modified genetic algorithm (mGA) is better than that of classic genetic algorithm (GA).

Since asynchronous automatic assembly systems have a very low cycle time, small decreases in the production rate will result in a large shortage of assemblies over a year. Therefore, using different penalty costs in the objective function generates largely different

results. Insufficient pallets and buffers spaces can result in a low production rate. Excessive pallets and buffer spaces lead to high cost of facilities. Considering both production rate and the cost of facility simultaneously, a less expensive system can be found compared to the maximum-throughput problem.

Although the results reveal that the GA and mGA perform well for asynchronous automatic assembly systems, the computational requirements for each GA or mGA run were very large. Liu and Sanders reported that they spent a time of 45 minutes to fulfill ten iterations of the stochastic quasi-gradient algorithm. However, the typical running times for the GA and mGA implementation were approximately 24 hours (on Digital DEC 5000/240 workststion computer).

6. CONCLUSION

In this research, two major issues are presented: modeling of an asynchronous automatic assembly system utilizing SIMAN simulation and comparing to the model of Liu and Sanders; and the application of genetic algorithms. The SIMAN simulation model was developed to simulate the asynchronous automatic assembly system and compared to Liu and Sanders' model for validation. The simulation model considered the the transport delay, blocking and starvation effect in the system.

The genetic algorithms are based on the natural selection principles that guide the generation of new chromosomes in living entities via genetic recombination. A genetic algorithm consists of a reproductive plan that provides organizational structure (i.e. bit strings) for representing the pool of genotypes of a generation, for selecting surviving genotypes to create the offspring of the next generation, and a set of genetic operators (i.e., crossover and mutation) used to generate the new offspring. A cost model was used as an objective function to distinguish "bad" or "good" results. The cost model considered the cost of pallets, buffer spaces, conveyor, holding inventory and penalty for underproducing.

In general, the objective function of the asynchronous automatic assembly system is stochastic. Genetic algorithms are created for optimization of deterministic objective functions. However, by integrating the SIMAN simulation model, cost model and genetic algorithms, the results in this research have shown that the GA and mGA perform well for the asynchronous automatic assembly system. The optimal solution consists of the total number of pallets and the number of buffer spaces between each pair of workstations. These variables were optimized concurrently to minimize the total annual system cost. Comparing to the cost of the best solutions of Liu and Sanders, it is clear that both genetic algorithms are able to find a cheaper system, while still reaching the production goal. This research also compared the

performance of GA and mGA under identical conditions. Although the solutions obtained from mGA are not always better than that from GA, the results showed that the performance of mGA is generally better than that of GA.

In this research, two different kinds of penalty cost functions were applied to evaluate the performance of the solutions. One considered a higher penalty cost (10% of the assembly value), while the other considered a lower penalty cost (1% per assembly). From the results, if the penalty is large, the algorithms will find a system with a large number of pallets and buffer spaces which usually generates a production rate high enough to avoid penalty. If the penalty is small, the algorithms will tend to converge to a system with a smaller number of pallets and buffer spaces which generally produces a smaller production rate than production goal. However, if the optimal number of buffer spaces is small (i.e. type 4 system), the penalty cost of the value of 1% per assembly still acts as a major factor and the system generates a high production to minimize the system cost. This is the reason that each type system generates a smaller system cost but type 4 system does not when the penalty cost is reduced.

Since different simulation seeds generate different production rates under identical system conditions, using the average of several simulation runs with different simulation seeds to an objective function might be advantageous. But, considering the huge computational requirement for a genetic algorithm run, the computer running cost would be increased dramatically. A huge execution time is the main drawback when the genetic algorithm is used in combination with a simulation model. Future research can be done to use computers with parallel processor architectures. This might decrease dramatically the execution time of genetic algorithms. Since the genetic algorithms search potential solution spaces in parallel, the algorithm could perform well on a parallel processor with a reasonable running time.

As mentioned in this study, the penalty cost depends on the actual production rate. Slight shortage in the production rate can result in a big difference in the annual cost when the unit penalty cost is high. The results also showed that the system accepted the combination of the smaller number of pallets and buffer spaces which generates a lower production under a lower penalty cost. These results are based on the assumption of paying the penalty cost only when underproducing happens. It does not need to produce more assemblies after deadline to fulfill the contract. For future research, a different penalty cost model which considers a penalty cost per day after deadline may be reasonable. Future research areas also can extend the asynchronous automatic assembly system to a more complex system which involves inspection and repair loops.

REFERENCES

Banks, J., and Carson, J. S., 1984. *Discrete-Event System Simulation*, Prentice-Hall, New Jersey.

Booker, Lashon, 1987 "Improving Search in Genetic Algorithms." *Genetic Algorithms and Simulated Annealing* (Edited by Lawrence Davis). Morgan Kaufmann Publishers, Los Altos, CA. pp. 61-73.

Boothroyd, Geoffrey, 1986. Economics of Assembly Systems. *Automated Assembly*, 2nd Edition (Edited by Lane, J. D.), Marcel Dekker, Inc., New York. pp. 34-48.

Boothroyd, Geoffrey, 1992. *Assembly Automation and product Design*. Marcel Dekker, New York.

Browne, Jimmie, Harhen, J. and Shivnan, J., 1988. Production Management Systems, Addison-Wesley, Great Britain.

Bulgak, A. A., and Sanders, J. L., 1991. "Approximate Analytical Performance Model for Automatic Assembly Systems with Statistical Process Control and Automated Inspection." *Journal of Manufacturing Systems*. Vol. 10, No. 2. pp. 121-133.

Bulgak, A. A., and Sanders, J. L., 1989. "Hybrid Algorithms for Design Optimization of Asynchronous Flexible Assembly Systems with Statistical Process Control and Repair." *Proceedings of Third ORSA/TIMS Conference on Flexible Manufacturing Systems*. Cambridge, MA. August 14-16. pp. 275-281.

Bulgak, A. A., and Sanders, J. L., 1988. "Integrating a Modified Simulated Annealing Algorithm with the Simulation of a Manufacturing System to Optimize Buffer Sizes in Automatic Assembly Systems." *Proceedings of the 1988 Winter Simulation Conference*. December 12-14. San Diego, C.A. pp. 684-690.

Bulgak, A. A., and Sanders, J. L., 1991. "Modeling and Design Optimization of Asynchronous Flexible Assembly Systems with Statistical Process and Repair." *The International Journal of Flexible Manufacturing Systems*. Vol. 3. pp. 251-274.

Cohoon, J.P., Hegde, S. U., Martin, W. N., Richards, D., 1988. "Floorplan Design using Distributed Genetic Algorithms." *Proceedings of IEEE International Conference on Computer-AidedDesign: ICCAD 88 a Conference for the EE CAD. IEEE*, New York. pp. 452-455.

Commult, C. and Semery, A., 1990. "Taking Into Account Delays in Buffers For Analytical Performance Evaluation of Transfer Lines." *IIE Transactions.* Vol 22 No. 2. pp. 133-142.

Davis, Lawrence, and Ritter, Frank., 1987. "Schedule Optimization with Probabilistic Search." *Proceedings from The Third Conference on Artificial Intelligence Applications. IEEE Computer Society Press,* New York. pp. 231-236.

Davis, Lawrence (Editor), 1987, *Genetic Algorithms and Simulated Annealing.* Morgan Kaufmann Publishers, Los Altos, CA.

Davis, Lawrence (Editor)., 1991. *Handbook of Genetic Algorithms.* Van Nostrand Reinhold Company, New York.

Davis, Lawrence, and Steenstrup, Martha., 1987. "Genetic Algorithms and Simulated Annealing: An Overview." *Genetic Algorithms and Simulated Annealing* (Edited. Lawrence Davis). Morgan Kaufmann Publishers, Los Altos, CA. pp. 1-11.

Falkenauer, E. and Bouffouix, S., 1991. "A genetic algorithm for job shop." *Proceedings - IEEE International Conference on Robotics and Automation.* Vol. 1. pp. 824-829.

Fujia, K., Akagji, S. and Kirokawa, N., 1993, "Hybrid Approach for Optimal Nesting Using a Genetic Algorithm and a Local Minimization Algorithm." *Advances in Design Automation American Society of Mechanical Engineers, Design Engineering Division* DE V 65 PT 1, ASME, New York. pp. 477-484.

Glover, D. E., 1987. "Solving a Complex Keyboard Configuration Problem Through Generalized Adaptive Search." *Genetic Algorithms and Simulated Annealing* (Edited by Lawrence Davis) Morgan Kaufmann Publishers, Los Altos, CA. pp. 13-31.

Glynn, P. W., 1986. "Optimization of Stochastic Systems." *Winter Simulation Conference Proceedings.* Washington, D.C. Dec. 8-10. pp. 52-59.

Goldberg, D. E., 1989. *Genetic Algorithms in Search, Optimization, and Machine Leaning,* Addison-Wesley, New York.

Goldberg, D. E., 1987. "Simple Genetic Algorithms and the Minimal, Deceptive Problem." *Genetic Algorithms and Simulated Annealing* (Edited by Lawrence Davis). Morgan Kaufmann Publishers, Los Altos, CA. pp. 75-88.

Grefenstette, J. J., 1987. "Incorporating Problem Specific Knowledge into Genetic Algorithms." *Genetic Algorithms and Simulated Annealing* (Edited by Lawrence Davis). Morgan Kaufmann Publishers, Los Altos, CA. pp. 43-60.

Hays, R. and Wheelright, S., 1984. *Restoring our competitive Edge: Computing Through Manufacturing*. John Wiley and Sons, New York.

Holland, John H., 1975. *Adaptation in Natural and Artificial Systems*. The University of Michigan Press, Ann Arbor, Michigan.

Huntley, Christorpher L., and Browns, D. E., 1991 "A Parallel Heuristic for Quadratic Assignment Problems." *Computers Operations Research*. Vol. 18, No. 3. pp. 275-289.

Kamath, M., and Sanders, J. L., 1987, "Analytical Methods for Performance Evaluation of Large Asynchronous Automatic Assembly Systems." *Lasge Scale Systems, Theory and Applications*, Vol 12, No. 2, pp. 143-154.

Kamath, M., Suri, R., and Sanders, J. L., 1988 "Analytic Performance Models for Closed-Loop Flexible Assembly Systems." *The International Journal of Flexible Manufacturing Systems*. Vol. 1. pp. 51-84.

Kraig, A, D., 1993 "Tandom application of a genetic algorithm and stochastic quasigradient method to the optimization of an assembly systemn," M.S. Thesis Iowa State University.

Law, A. M., and Kelton, W. D., 1991. *Simulation Modeling and Systems Analysis*. Second Edition. McGraw-Hill, New York.

Leung, W. K., and Sanders, J. L., 1986. "Simulation Analysis of the Performance of Tunnel-Gated Stations for Free-Transfer Assembly Systems." *Journal of Manufacturing Systems*. Vol. 5, No. 3. pp. 191-202.

Liu, C. M., and Chiou, J. M., 1989. "Design and Performance Evaluation of Closed Automatic Assembly Systems." *International Journal of Production Research*. Vol. 28, No. 9. pp. 1577-1593.

Liu, C. M., and Sanders, J. L., 1989. "Approximate Design Optimization of Asynchronous Assembly Systems." *International Journal of Computer Applications in Technology*. Vol. 2. No. 1. pp. 30-37.

Liu, C. M., and Sanders, J. L., 1988. "Stochastic Design Optimization of Asynchronous Flexible Assembly Systems." *Annals of Operations Research*. Vol. 15. pp. 131-154.

Meketon, M. S., 1987. "Optimization in Simulation: A Survey of Recent Results." *Proceedings of the 1987 Winter Simulation Conference*. pp. 58-67.

Michalewicz, Z.,1992. *Genetic Algorithms + Data Structure = Evolution Programs*, Springer-Verlag, Berlin, New York.

Pegden, C. D., Shannon, R. E., and Sadowski, R. P., 1990. *Introduction to simulation Using SIMAN*. McGraw-Hill, New York.

Pettey, C. B., Leuze, M. R., and Grefenstette, J. J., 1987. "A Parallel Genetic Algorithm." Genetic Algorithms and Their Applications: *Proceedings of the Second International Conference on Genetic Algorithms*. MIT, Cambridge, MA, July 28-31, pp. 155-161.

Richardson, J. T., Palmer, M. R., Liepins, G., Hilliard, M., 1989. "Some Guidelines for Genetic Algorithms with Penalty Functions." *Proceedings of the third International Conference on Genetic Algorithms*. George Mason University, June 4-7, 1989. pp. 191-197.

Schloemer, P. G., 1992. "Let's Get America Back To Business." *Industry Week*, April 6, 1992: 34.

Tandiono, Elly. 1991. "Stochastic Optimization of Cost of Automatic Assembly Systems." M.S. Thesis. Iowa State University.

Wellman, M. A. 1991. "A genetic Algorithm Approach to optimization of Asynchronous automatic Assembly Systems." M.S. Thesis. Iowa State University.

APPENDIX A

DETAILED RESULTS OF EACH GENERATION

Table 1. The Minimum Results of Each Generation for System Type 1 (penalty 10% of Value of Per Assembly)

System 1	Simulation Seed Type 1				Simulation Seed Type 2			
penalty 10%	Program Seed Type 1		Program Seed Type 2		Program Seed Type 1		Program Seed Type 2	
Number of Generation	GA1	mGA1	GA2	mGA2	GA1	mGA1	GA2	mGA2
0	258765.3	258765.3	188300.1	188300.1	258765.3	258765.3	259313.5	259313.5
1	258765.3	258765.3	188208.8	188208.8	258765.3	258765.3	188482.9	188482.9
2	258582.5	258582.5	188208.8	188208.8	258765.3	258765.3	188482.9	188482.9
3	258582.5	258582.5	188208.8	188208.8	258673.9	258673.9	258765.3	188482.9
4	258582.5	258582.5	188208.8	188208.8	258673.9	258673.9	259770.4	188482.9
5	258582.5	258582.5	188208.8	188208.8	258673.9	258673.9	188482.9	188482.9
6	258582.5	258582.5	188208.8	188208.8	258673.9	258673.9	188482.9	188482.9
7	258765.3	258582.5	188208.8	188208.8	258673.9	258673.9	188482.9	188482.9
8	258582.5	258582.5	188208.8	188208.8	258673.9	258673.9	188482.9	188482.9
19	258582.5	258582.5	188208.8	188208.8	258673.9	258673.9	188482.9	188482.9
10	258582.5	258582.5	188208.8	188208.8	258673.9	258673.9	188482.9	188482.9
11	258582.5	258582.5	188208.8	188208.8	258673.9	258673.9	188482.9	188482.9
12	258582.5	258582.5	188208.8	188208.8	258673.9	258673.9	188482.9	188482.9
13	258582.5	258582.5	188208.8	188208.8	258673.9	258673.9	188482.9	188482.9
14	258582.5	258582.5	188208.8	188208.8	258673.9	258673.9	188482.9	188482.9
15	258582.5	258582.5	188208.8	188208.8	258673.9	258673.9	188482.9	188482.9
16	258582.5	258582.5	188208.8	188208.8	258673.9	258673.9	188482.9	188482.9
17	258582.5	258582.5	188208.8	188208.8	258673.9	258673.9	188482.9	188482.9
18	258582.5	258582.5	188208.8	188208.8	258673.9	258673.9	188482.9	188482.9
19	258582.5	258582.5	188208.8	188208.8	258673.9	258673.9	188482.9	188482.9
20	258582.5	258582.5	188208.8	188208.8	258673.9	258673.9	188482.9	188482.9
21	258582.5	258582.5	188208.8	188208.8	258673.9	258673.9	188482.9	188482.9
22	258582.5	258582.5	188208.8	188208.8	258673.9	258673.9	188482.9	188482.9
23	258582.5	258582.5	188208.8	188208.8	258673.9	258673.9	188482.9	188482.9
24	258582.5	258582.5	188208.8	188208.8	258673.9	258673.9	188482.9	188482.9
25	258582.5	258582.5	188208.8	188208.8	258673.9	258673.9	188482.9	188482.9

Table 2. The Minimum Results of Each Generation for System Type 1 (penalty 1% of Value of Per Assembly)

System 1	Simulation Seed Type 1				Simulation Seed Type 2			
penalty 1%	Program Seed Type 1		Program Seed Type 2		Program Seed Type 1		Program Seed Type 2	
Number of Generation	GA1	mGA1	GA2	mGA2	GA1	mGA1	GA2	mGA2
0	163854.9	163854.9	163946.3	163946.3	167598.8	167598.8	167690.2	167690.2
1	163854.9	163854.9	163946.3	163946.3	167598.8	167598.8	167690.2	167690.2
2	168256.3	163854.9	163946.3	163946.3	175695.0	167598.8	167690.2	167690.2
3	168256.3	163854.9	163946.3	163946.3	167598.8	167598.8	175269.6	167690.2
4	180426.4	163854.9	163946.3	163946.3	167598.8	167598.8	175269.6	167690.2
5	163946.3	163854.9	163946.3	163946.3	167598.8	167598.8	175269.6	167690.2
6	163946.3	163854.9	168530.4	163946.3	167598.8	167598.8	175269.6	167690.2
7	163946.3	163854.9	168530.4	163946.3	175695.0	167598.8	175695.0	167690.2
8	175361.0	163854.9	178873.0	163946.3	175695.0	167598.8	176535.2	167690.2
19	175361.0	163854.9	175361.0	163946.3	167598.8	167598.8	176535.2	167690.2
10	175361.0	163854.9	178781.6	163946.3	176626.5	167598.8	176626.5	167690.2
11	163946.3	163854.9	178873.0	163946.3	176626.5	167598.8	176626.5	167690.2
12	175361.0	163854.9	178873.0	163946.3	167690.2	167598.8	176626.5	167690.2
13	168530.4	163854.9	178873.0	163946.3	167690.2	167598.8	176626.5	167690.2
14	178690.2	163854.9	178873.0	163946.3	176535.2	167598.8	176626.5	167690.2
15	178690.2	163854.9	178873.0	163946.3	176626.5	167598.8	176626.5	167690.2
16	178690.2	163854.9	181157.4	163946.3	176626.5	167598.8	176626.5	167690.2
17	178690.2	163854.9	178873.0	163946.3	176626.5	167598.8	176626.5	167690.2
18	178690.2	163854.9	178873.0	163946.3	176535.2	167598.8	176626.5	167690.2
19	178690.2	163854.9	178873.0	163946.3	176535.2	167598.8	176626.5	167690.2
20	178690.2	163854.9	178873.0	163946.3	176535.2	167598.8	176809.3	167690.2
21	178690.2	163854.9	178690.2	163946.3	176535.2	167598.8	176626.5	167598.8
22	178690.2	163854.9	178690.2	163946.3	176535.2	167598.8	176626.5	167598.8
23	178690.2	163854.9	178781.6	163946.3	176535.2	167598.8	176626.5	167598.8
24	178690.2	163854.9	179055.7	163946.3	176535.2	167598.8	176626.5	167598.8
25	178964.4	163854.9	178690.2	163946.3	176535.2	167598.8	176626.5	167598.8

Table 3. The Minimum Results of Each Generation for System Type 2
(penalty 10% of Value of Per Assembly)

System 2	Simulation Seed Type 1				Simulation Seed Type 2			
penalty 10%	Program Seed Type 1		Program Seed Type 2		Program Seed Type 1		Program Seed Type 2	
Number of Generation	GA1	mGA1	GA2	mGA2	GA1	mGA1	GA2	mGA2
0	680154.9	680154.9	729092.9	729092.9	720969.2	720969.2	729092.9	729092.9
1	792984.9	559606.2	828749.1	729092.9	996136.3	720969.2	870591.2	729092.9
2	742248.1	559606.2	828749.1	668567.1	625076.8	720969.2	827652.6	729092.9
3	792984.8	544550.6	728179.1	668567.1	625076.8	585923.2	816670.5	729092.9
4	425408.2	544550.6	565298.8	668567.1	625076.8	585923.6	729641.1	670211.8
5	425408.2	518056.4	645400.0	668567.1	514248.1	570922.1	580992.0	669937.7
6	425408.2	518056.4	706445.7	527207.4	514248.1	570922.1	721985.4	619472.8
7	410126.7	473490.5	503929.6	527207.4	514248.1	570922.1	721985.4	402815.6
8	400988.1	473490.5	503929.6	527207.4	465583.7	563275.7	592922.7	402815.6
19	306680.7	473490.5	382628.0	517965.0	514248.1	427955.5	592922.7	402815.6
10	306680.7	473490.5	549237.4	517965.0	514248.1	427955.5	516503.0	402815.6
11	392948.8	357804.6	503929.6	517965.0	514248.1	427955.5	517051.3	402815.6
12	365375.7	357804.6	391030.0	517965.0	514248.1	341300.3	517051.3	386391.7
13	374137.5	357804.6	391030.0	468489.9	457324.1	341300.3	517051.3	386391.7
14	374137.5	357804.6	438199.1	468489.9	480488.6	298717.4	539145.4	386391.7
15	322979.7	357804.6	383084.9	468489.9	405876.7	298717.4	507516.1	386391.7
16	322979.7	357804.6	383084.9	468489.9	470490.3	283931.2	394136.7	384272.8
17	374137.5	357804.6	383084.9	401993.1	450090.3	283931.2	462902.4	384272.8
18	322979.7	357804.6	383084.9	401993.1	421461.1	283931.2	507516.1	384272.8
19	322979.7	357804.6	314783.1	401993.1	458146.5	283931.2	487174.9	367203.2
20	322979.7	357804.6	314783.1	401993.1	330258.7	283931.2	498714.7	367203.2
21	322979.7	357804.6	314783.1	401993.1	373954.8	283931.2	467758.9	367203.2
22	322979.7	274939.0	314783.1	401993.1	373954.8	283931.2	487905.9	350419.1
23	322979.7	274939.0	314783.1	401993.1	373954.8	283931.2	384272.8	350419.1
24	348134.7	261141.0	314783.1	401993.1	373954.8	283931.2	394136.7	332001.5
25	348134.7	261141.0	365558.4	401993.1	307594.5	283383.0	394264.5	332001.5

Table 4. The Minimum Results of Each Generation for System Type 2 (penalty 1% of Value of Per Assembly)

System 2	Simulation Seed Type 1				Simulation Seed Type 2			
Penalty 1%	Program Seed Type 1		Program Seed Type 2		Program Seed Type 1		Program Seed Type 2	
Number of Generation	GA1	mGA1	GA2	mGA2	GA1	mGA1	GA2	mGA2
0	586636.6	586636.6	729092.9	729092.9	570911.8	570911.8	729092.9	729092.9
1	740561.1	586636.6	828200.9	729092.9	740561.1	570911.8	828200.9	729092.9
2	740561.1	586636.6	828200.9	729092.9	740561.1	570911.8	828121.2	525670.1
3	742613.6	586636.6	778883.6	702750.3	759836.1	570911.8	729092.9	525670.1
4	636561.9	559697.6	634094.8	567080.3	614142.7	504112.6	729092.9	525670.1
5	790422.3	425736.5	569612.4	538534.5	550059.8	504112.6	716580.0	379840.7
6	684594.3	425736.5	569612.4	288289.5	524520.1	504112.6	548415.0	379840.7
7	694943.6	376056.4	520117.8	288289.5	570708.9	477031.0	487631.8	379840.7
8	568943.4	309931.6	444279.6	288289.5	570708.9	449412.6	487484.7	379840.7
19	568943.4	292150.8	333179.9	288289.5	570708.9	337054.0	487484.7	379840.7
10	580352.4	261411.5	325831.2	283931.3	526438.1	333637.5	485016.9	377753.4
11	507900.9	261411.5	325831.2	276675.1	433292.7	313240.3	496623.3	337503.5
12	507900.9	261411.5	325831.2	276675.1	467886.7	313240.3	410583.6	337503.5
13	518711.6	260589.1	333179.9	228184.8	467147.7	313240.3	410583.6	292037.7
14	520958.0	260223.6	307432.7	221693.2	467147.7	313240.3	359191.2	267042.4
15	520958.0	245454.4	248716.8	169079.7	429359.5	313240.3	319985.6	266494.2
16	520958.0	245454.4	268481.8	169079.7	403455.2	300087.6	319985.6	266494.2
17	426644.6	231757.7	292147.2	169079.7	384692.7	300087.6	319985.6	266494.2
18	416783.5	231757.7	240044.4	169079.7	378136.2	300087.6	317624.3	255654.3
19	400144.9	213088.7	240044.4	169079.7	304219.5	263042.1	222961.0	210484.1
20	400144.9	213088.7	224611.1	169079.7	317631.0	263042.1	259587.6	210484.1
21	425730.8	213088.7	224611.1	169079.7	303960.9	263042.1	257525.6	210484.1
22	367202.4	188576.8	224611.1	169079.7	303960.9	221793.3	261501.8	210484.1
23	367202.4	188576.8	241020.3	169079.7	303960.9	220376.1	261501.8	210484.1
24	367202.4	188576.8	223070.6	169079.7	296471.5	219645.1	234763.4	210484.1
25	387281.9	188576.8	223070.6	159655.9	303960.9	219645.1	190748.6	210484.1

Table 5. The Minimum Results of Each Generation for System Type 3
(penalty 10% of Value of Per Assembly)

System 3	Simulation Seed Type 1				Simulation Seed Type 2			
penalty 10%	Program Seed Type 1		Program Seed Type 2		Program Seed Type 1		Program Seed Type 2	
Number of Generation	GA1	mGA1	GA2	mGA2	GA1	mGA1	GA2	mGA2
0	926530.0	926530.0	880563.9	880563.9	1003571.6	1003571.6	823719.9	823719.9
1	988803.8	926530.0	925202.3	880563.9	933246.3	933246.3	828200.9	823719.9
2	741517.1	926530.0	868199.9	829114.6	933246.3	933246.3	657073.5	823719.9
3	741517.1	717676.5	871503.8	829114.6	694121.3	694121.3	657073.5	647318.9
4	617480.3	571074.4	694669.5	829114.6	595134.1	623797.5	657073.5	647318.9
5	499080.2	571074.4	741011.9	725534.9	595134.1	591095.2	800684.4	647318.9
6	549602.9	571074.4	682439.3	725534.9	741608.5	591095.2	742789.3	613685.8
7	570434.8	571074.4	694029.9	725534.9	757436.3	591095.2	718590.3	613685.8
8	570434.8	571074.4	614234.1	513684.9	831620.0	591095.2	487723.2	548323.6
19	570434.8	571074.4	569612.4	513684.9	592008.9	591095.2	593393.4	548323.6
10	448354.1	571074.4	555532.8	513684.9	560154.4	591095.2	629814.6	548323.6
11	518513.3	569245.8	601855.8	513684.9	592100.3	482917.9	580992.0	527503.0
12	394008.7	569245.8	601855.8	427955.5	596722.6	482917.9	497526.8	527503.0
13	394008.7	548780.5	592374.4	427955.5	581540.3	482917.9	497526.8	493457.3
14	394008.7	389431.9	614508.2	427955.5	478196.5	482917.9	497526.8	467302.0
15	394008.7	389431.9	478653.4	427955.5	477648.3	457781.0	497709.6	467302.0
16	394008.7	389431.9	478653.4	427955.5	422809.4	457781.0	497709.6	467302.0
17	434075.3	389431.9	478653.4	427955.5	422809.4	448354.1	497709.6	467302.0
18	467376.7	366198.1	522551.8	427955.5	422809.4	448354.1	549054.6	477648.3
19	467376.7	366198.1	448080.0	427955.5	422626.7	420090.5	580809.3	419085.4
20	375782.3	366198.1	448080.0	427955.5	422626.7	340037.9	580809.3	419085.4
21	375782.3	366198.1	479018.9	427955.5	420821.5	340037.9	487540.4	419085.4
22	375782.3	366198.1	488545.6	427955.5	422626.7	340037.9	487540.4	376564.2
23	375782.3	366198.1	478653.4	427955.5	422626.7	340037.9	487540.4	376564.2
24	375416.8	366198.1	487357.7	427955.5	402450.1	339489.6	459517.1	376564.2
25	375416.8	366198.1	509446.0	427955.5	339489.6	339489.6	458877.5	376564.2

Table 6. The Minimum Results of Each Generation for System Type 3 (penalty 1% of Value of Per Assembly)

System 3	Simulation Seed Type 1				Simulation Seed Type 2			
penalty 1%	Program Seed Type 1		Program Seed Type 2		Program Seed Type 1		Program Seed Type 2	
Number of Generation	GA1	mGA1	GA2	mGA2	GA1	mGA1	GA2	mGA2
0	592627.0	592627.0	765035.4	765035.4	640480.6	640550.2	734083.6	734083.6
1	740561.1	592627.0	829114.6	765035.4	752873.0	640550.2	828200.9	734083.6
2	740561.1	592627.0	828931.9	765035.4	752873.0	640550.2	816507.9	570746.9
3	742613.6	592627.0	786945.6	508176.3	692273.6	640550.2	753423.5	570746.9
4	560140.5	519025.0	641659.2	437179.6	781948.2	640550.2	651261.0	570746.9
5	613063.4	518476.8	537957.6	400436.8	791926.1	620799.4	617275.4	570746.9
6	613063.4	444877.0	506145.5	400436.8	718291.1	430605.4	596410.6	570746.9
7	603991.2	444877.0	537957.6	400436.8	713687.4	430605.4	466705.6	554004.3
8	535717.9	444877.0	534467.8	375470.2	604125.3	430605.4	389956.1	498925.5
19	478416.0	444877.0	394956.8	375470.2	603120.1	430605.4	378100.8	486261.2
10	407995.8	368932.7	394956.8	375470.2	444341.7	430605.4	403438.4	454672.7
11	396592.5	300641.1	394956.8	330391.5	444341.7	341572.5	403438.4	454672.7
12	289733.7	300641.1	375211.9	255389.2	444341.7	326607.7	401538.1	454672.7
13	289733.7	300641.1	375211.9	255389.2	444341.7	326607.7	284259.4	390450.8
14	274376.7	300641.1	321148.3	255389.2	451596.8	302320.4	284259.4	390450.8
15	289733.7	300641.1	321148.3	255389.2	394757.5	229558.5	327994.6	390450.8
16	269197.8	300641.1	330477.4	255389.2	373058.5	229558.5	280756.0	237797.1
17	242624.0	270781.0	371201.2	255389.2	340860.1	229558.5	229894.4	237797.1
18	230778.9	270781.0	274760.6	255389.2	340860.1	229558.5	239404.4	237797.1
19	230778.9	270781.0	304567.8	237540.3	340860.1	229558.5	250070.4	237797.1
20	274194.0	270781.0	291287.0	237540.3	340860.1	229558.5	250070.4	237797.1
21	274194.0	270781.0	277870.0	188192.5	340860.1	225605.2	239404.4	237797.1
22	273514.3	270781.0	291711.2	188192.5	286515.9	225605.2	246627.4	237797.1
23	269591.8	221323.5	303217.3	188192.5	311448.9	225605.2	240043.5	237705.8
24	269591.8	221323.5	270487.4	188192.5	306662.8	225605.2	256035.8	237705.8
25	268497.9	221323.5	270487.4	188192.5	284821.9	201324.3	255572.2	236043.7

Table 7. The Minimum Results of Each Generation for System Type 4 (penalty 10% of Value of Per Assembly)

System 4	Simulation Seed Type 1				Simulation Seed Type 2			
penalty 10%	Program Seed Type 1		Program Seed Type 2		Program Seed Type 1		Program Seed Type 2	
Number of Generation	GA1	mGA1	GA2	mGA2	GA1	mGA1	GA2	mGA2
0	305036.0	305036.0	346581.3	346581.3	372492.8	372392.8	409395.7	409395.7
1	305310.1	305036.0	346581.3	346581.3	274299.3	274299.3	338741.8	338471.8
2	289022.4	259496.3	305584.2	305584.2	274390.7	274299.3	243973.5	243973.5
3	243882.1	259496.3	251322.3	251322.3	274482.1	194712.6	243973.5	243973.5
4	236627.5	259404.9	251322.3	251322.3	274482.1	162678.6	243973.5	202589.9
5	236627.5	229558.5	251322.3	251322.3	236901.6	162678.6	259313.5	202498.5
6	195078.1	150615.8	237267.1	237267.1	236901.6	132999.6	229773.9	202498.5
7	208551.1	150615.8	237267.1	237267.1	209373.5	132999.6	229773.9	201858.9
8	208551.1	150615.8	188665.6	188665.6	209373.5	132999.6	229773.9	201858.9
19	208368.3	150615.8	22309.5	188665.6	222583.6	131502.2	195260.8	201858.9
10	170176.2	127437.8	208459.7	169445.2	230254.5	131502.2	216251.3	201858.9
11	194804.0	127437.8	215611.6	145593.7	202133.0	127346.4	216159.9	157559.5
12	175666.5	127437.8	215611.6	145593.7	188848.4	127346.4	222492.3	157559.5
13	181707.8	127437.8	190201.0	145593.7	216404.7	127346.4	201858.9	151529.5
14	175940.6	127437.8	208551.1	145593.7	195352.2	127346.4	237541.2	151529.5
15	175209.6	127437.8	201858.9	122335.6	208825.2	127346.4	236627.5	151529.5
16	168897.0	127437.8	182164.6	122335.6	208825.2	127346.4	229467.1	151529.5
17	168897.0	127437.8	169262.5	122335.6	223040.5	127346.4	216251.3	151346.8
18	168897.0	124998.6	182164.6	122335.6	222766.4	127346.4	216251.3	151346.8
19	170166.7	124998.6	182530.1	122335.6	182347.4	127346.4	202315.7	151346.8
20	133365.1	122883.9	170084.8	122335.6	182347.4	127346.4	188482.9	145593.7
21	150889.9	122244.3	169262.5	122335.6	188757.0	127346.4	188482.9	145593.7
22	150889.9	117601.8	169262.5	122335.6	188757.0	127346.4	201767.5	145593.7
23	163775.1	117053.6	163044.1	122335.6	188848.4	127346.4	202407.1	145593.7
24	150524.4	101224.4	165774.2	122335.6	195352.2	127346.4	175849.2	145593.7
25	150524.4	101224.4	163226.8	122335.6	195352.2	127346.4	176214.7	144862.7

Table 8. The Minimum Results of Each Generation for System Type 4 (penalty 1% of Value of Per Assembly)

System 4	Simulation Seed Type 1				Simulation Seed Type 2			
penalty 1%	Program Seed Type 1		Program Seed Type 2		Program Seed Type 1		Program Seed Type 2	
Number of Generation	GA1	mGA1	GA2	mGA2	GA1	mGA1	GA2	mGA2
0	305036.0	305036.0	346581.3	346581.3	330495.2	330495.2	330495.2	367409.6
1	250774.0	250774.0	355428.8	346581.3	202915.1	202915.1	367409.6	367409.6
2	250774.0	250774.0	289479.3	344366.7	195335.8	195335.8	289479.3	289479.3
3	250774.0	250774.0	266850.7	275836.3	195335.8	195335.8	266850.7	266850.7
4	245453.3	245453.3	266485.2	267033.4	187297.6	187297.6	266850.7	266850.7
5	250774.0	245453.3	266485.2	252053.3	168805.6	168805.6	281842.2	266850.7
6	181616.4	230015.3	251505.0	242653.5	156463.0	156463.0	250180.1	266850.7
7	215063.4	215720.8	237267.1	229741.2	156463.0	156463.0	310570.7	259587.6
8	189213.9	215720.8	237267.1	229741.2	150798.5	150798.5	338102.1	216159.9
19	189305.3	194202.6	236901.6	229741.2	128691.5	128691.5	270821.1	216159.9
10	182712.9	181600.0	230106.7	200840.0	144862.7	128691.5	270821.1	216159.9
11	208872.2	181600.0	230106.7	188208.8	157303.2	128691.5	215794.4	215703.0
12	128077.4	181600.0	215703.0	188208.8	127255.0	128691.5	215703.0	166770.3
13	128077.4	181600.0	182256.0	188208.8	127255.0	128691.5	215703.0	162495.8
14	128442.9	181600.0	182256.0	169536.6	133000.0	121787.4	234605.7	162495.8
15	110860.6	176553.6	182256.0	169536.6	138838.4	121787.4	251170.7	162495.8
16	110860.6	176553.6	178844.4	169033.3	122883.9	121787.4	242509.3	162495.8
17	110860.6	170084.8	181981.9	139295.2	130341.6	121787.4	233101.9	162495.8
18	110860.6	135444.0	181981.9	139295.2	127255.0	121787.4	235354.5	151255.4
19	110860.6	135444.0	153793.7	139295.2	127255.0	121787.4	236993.0	151164.0
20	110860.6	134187.5	153793.7	133822.0	130341.6	121787.4	211776.0	151164.0
21	110860.6	122569.3	169810.7	133822.0	121787.4	115444.7	202041.6	151164.0
22	106406.6	117236.3	176214.7	133822.0	131090.4	104986.1	195535.0	145228.2
23	110952.0	117236.3	170084.8	127894.6	127255.0	104894.8	156408.5	145228.2
24	106406.6	112488.3	174781.9	117327.7	132908.2	104894.8	156408.5	145228.2
25	116231.2	112488.3	151803.6	117327.7	138747.0	104894.8	141907.1	145228.2

Table 9. The Minimum Results of Each Generation for System Type 5 (penalty 10% of Value of Per Assembly)

System 5	Simulation Seed Type 1				Simulation Seed Type 2			
penalty 10%	Program Seed Type 1		Program Seed Type 2		Program Seed Type 1		Program Seed Type 2	
Number of Generation	GA1	mGA1	GA2	mGA2	GA1	mGA1	GA2	mGA2
0	814655.6	814655.6	968975.3	968975.3	890666.0	890666.0	880563.9	880563.9
1	992018.3	814655.6	992583.7	968975.3	933246.3	890666.0	944423.6	880563.9
2	992018.3	814655.6	992583.7	968975.3	933246.3	840921.9	944423.6	880563.9
3	992018.3	814655.6	973066.8	968975.3	907036.5	692385.1	880746.6	751771.6
4	992018.3	766015.7	992583.7	717585.1	634081.1	670917.8	894484.1	690448.9
5	847411.0	594703.4	886889.5	717585.1	634081.1	670917.8	948182.9	690448.9
6	722814.1	588316.2	820074.2	717585.1	634081.1	627214.6	821671.9	642340.4
7	722814.1	588316.2	845241.2	717585.1	634081.1	627214.6	821671.9	642340.4
8	722814.1	588316.2	828322.9	717585.1	634081.1	627214.6	607528.4	642340.4
19	681982.4	588316.2	845241.2	692842.0	634081.1	627214.6	607528.4	642340.4
10	681982.4	569521.0	815319.1	692842.0	569927.8	627214.6	626185.4	642340.4
11	681982.4	569521.0	664940.3	692842.0	614427.3	627214.6	591048.1	642340.4
12	704031.4	569521.0	664940.3	654475.0	611547.1	627214.6	644939.1	611216.9
13	682623.8	560428.6	664940.3	654475.0	515626.7	627214.6	631163.3	611216.9
14	531716.6	560428.6	659083.8	654475.0	597202.6	570434.8	631163.3	548323.6
15	704031.4	560428.6	632846.2	654475.0	597202.6	570434.8	647593.0	532121.1
16	643634.1	560428.6	576066.9	654475.0	614325.4	570434.8	611252.4	517508.1
17	518056.4	520428.6	576066.9	654475.0	622242.4	570434.8	571646.6	517508.1
18	694395.4	497344.1	576066.9	654475.0	547589.3	570434.8	571646.6	517508.1
19	625533.6	497344.1	603898.1	654475.0	547589.3	570434.8	571646.6	517508.1
20	646222.4	497344.1	607431.1	606358.6	534660.5	570434.8	571646.6	517508.1
21	629590.4	497344.1	613685.8	587656.1	530743.9	570434.8	571646.6	517508.1
22	621303.9	497344.1	613685.8	587656.1	560366.2	570434.8	678114.4	517508.1
23	567094.2	497344.1	614478.5	497161.3	570617.5	570434.8	588030.2	457781.0
24	567094.2	497344.1	524694.7	497161.3	602754.6	558071.8	549237.4	457781.0
25	567094.2	497344.1	614478.5	497161.3	560154.4	558071.8	670485.9	457781.0

Table 10. The Minimum Results of Each Generation for System Type 5 (penalty 1% of Value of Per Assembly)

System 5	Simulation Seed Type 1				Simulation Seed Type 2			
penalty 1%	Program Seed Type 1		Program Seed Type 2		Program Seed Type 1		Program Seed Type 2	
Number of Generation	GA1	mGA1	GA2	mGA2	GA1	mGA1	GA2	mGA2
0	585994.2	585887.8	770330.3	770276.8	593595.3	593595.3	764718.1	764718.1
1	780605.0	585887.8	829114.6	770276.8	789982.3	593595.3	839966.1	764718.1
2	766361.1	585887.8	829114.6	565582.6	776025.9	593595.3	839966.1	568354.6
3	673714.0	441509.1	670760.1	557560.1	670257.5	532633.0	780593.9	515718.6
4	673714.0	441509.1	688553.0	557560.1	670257.5	532633.0	543608.1	515718.6
5	633334.4	382284.2	576278.1	498472.5	517811.5	488596.4	543608.1	501389.1
6	619176.7	382284.2	576278.1	498472.5	517811.5	488596.4	540455.5	501389.1
7	581054.8	382284.2	621299.2	497848.4	517811.5	410363.4	540455.5	481771.0
8	588244.1	382284.2	518716.1	407217.1	480823.7	375400.3	514291.9	391738.8
19	448565.6	382284.2	430787.1	399546.3	480823.7	329695.0	501611.9	360017.3
10	421323.3	370851.7	388451.0	399546.3	480823.7	301725.7	484103.1	360017.3
11	344594.4	353964.2	332878.2	398882.6	461074.4	301725.7	460113.2	360017.3
12	359388.6	346310.6	332878.2	373817.6	446341.9	301725.7	383856.4	348026.2
13	376017.3	330554.0	356948.6	356375.6	443796.1	274070.2	383856.4	326707.9
14	372704.2	310691.3	337220.2	339328.5	439518.8	274070.2	383856.4	326707.9
15	263310.2	300384.7	328555.0	313243.1	422376.4	272880.3	415077.2	326707.9
16	263310.2	300384.7	318389.9	313243.1	331647.1	238939.7	415077.2	314341.2
17	263310.2	300384.7	316896.3	306758.4	331647.1	238939.7	413137.7	314341.2
18	263310.2	300384.7	316896.3	306758.4	331647.1	238939.7	419788.1	276495.6
19	263310.2	300384.7	285522.9	306758.4	283619.0	238939.7	335384.3	276495.6
20	255364.2	286048.3	285522.9	306758.4	283801.8	238939.7	369485.6	276495.6
21	253484.5	286048.3	283595.9	306758.4	283801.8	238939.7	315745.9	276495.6
22	253484.5	286048.3	281597.2	302151.0	325635.8	238939.7	326907.0	265863.3
23	236745.8	278890.2	281597.2	271746.8	325227.8	238939.7	326907.0	265863.3
24	236745.8	278890.2	284316.0	249292.6	283344.9	238939.7	338929.5	238282.0
25	253484.5	278890.2	273262.2	249292.6	318825.3	238939.7	298114.7	238282.0

APPENDIX B

IMPLEMENTATION OF A GENETIC ALGORITHM

WITH C SOURCE CODE

```
/*      The Simple Genetic Algorithm

        The genetic algorithm used in this study is
        the simple genetic algorithm (SGA) which
        was presented by
                        David E. Goldberg (1989)

        The program is part of this research done to
        achieve the requirements for a Master of
        Science degree in Industrial Engineering.
                Programming by:
                                Yisheng Hsiao
                                IMSE Department
                                Iowa State University
                                1994                    */


#include<stdio.h>
#include<stdlib.h>
#include<time.h>

int num_bistring = 56;
int num_works = 10;

struct individual
  {
        int chromo[57]; /* genotype=bistring */
        int x[12];  /* phenotype = 11 integers */
        float fitness;  /* objective function value */
        int parent1;  /* parent number 1 */
        int parent2;  /* parent number 2 */
        int cross_site;  /* cross-over site */
        float rate;  /* production rate */
  }     oldpop[110],newpop[110];

float r1num(void);  /* random number generator #1 */
float r2num(void);  /* random number generator #2 */

int popsize, gen, maxgen;
float pcross, pmutation, sumfitness;
```

```
int nmutation, ncross, jcross;
float avg, max, min;
long int seed1;  /* global rng seeds */

FILE *fpout, *fpout1;

/***************************/
/*                         */
/*      main()             */
/*                         */
/***************************/

void main(void)
 {
   void copy_new_into_old();
   int select_individual();
   int flip();
   int find_x_site();
   void crossover();
   int mutation();
   void generation();
   int decode_num_of_buffers();
   int decode_num_of_pallets();
   void set_fitness_value();
   void write_siman_exp();
   void create_init_population();
   void generate_init_report();
   void statistics();
   void buffer_np_out();
   void report();

   int i;
   double tused;

   time_t tstart, tstop;

   printf("\n\n    Input a random seed --> ");
   scanf("%li", &seed1);
   popsize = 100;  /* This must be an even number */
   maxgen = 25;
   pcross = 0.6;  /* Crossover probability */
   pmutation = 0.005;  /* mutation probabiity */
```

```
  nmutation = 0;
  ncross = 0;
  gen = 0;

  /* open global output file */
  if((fpout = fopen("hyst1515.21","w")) == NULL)
  {
    printf("Unable to open HYST1515.21 file !\n\n");
    exit(1);
  }

  /* Time Start */
  time(&tstart);

  system("model hyst362.mod hyst362.m");
  printf("Finished compiling model file\n");
  create_init_population();
  statistics(oldpop);
  generate_init_report();
  copy_new_into_old(newpop, oldpop)
  for(i=1; i<=maxgen; i++)
  {
    gen++;
    generation();
    statistics(newpop);
    report();
    copy_new_into_old(oldpop, newpop);
  }
  buffer_np_out();  /* outputs final number of buffers and the number of pals */

  /* Time end */
  time(&tstop);
  tused = difftime(tstop, tstart);
  fprintf(fpout,"The running time is %10.2f seconds\n\n", tused);
  fclose(fpout);
}

/*************************************/
/*                                   */
/*      Copy new into old population */
/*                                   */
/*************************************/
```

```
void copy_new_into_old( oldpop, newpop)

 struct individual oldpop[110];
 struct individual newpop[110];

 {
   int i,j;

   for( i=1; i<= popsize; i++)
   {
     for(j=1; j<=num_bistring; j++)
        oldpop[i].chromo[j]=newpop[i].chromo[j];
     for(j=1; j<=num_works + 1; j++)
        oldpop[i].x[j]=newpop[i].x[j];
     oldpop[i].fitness= newpop[i].fitness;
     oldpop[i].parent1 = newpop[i].parent1;
     oldpop[i].parent2 = newpop[i].parent2;
     oldpop[i].cross_site = newpop[i].cross_site;
     oldpop[i].rate = newpop[i].rate;
   }
 }

/*************************/
/*                       */
/*      Select individual */
/*                       */
/*************************/

int select_individual(work_pop)

 struct individual work_pop[110];
 {
   int i, pop_index;  /* population index */
   float rpw, partial_sum;   /* random point on wheel, partial sum */
   float tsf;   /* transformed sumfitness for min problem */
   float tfv;   /* transformed fitness value for min problem */

   partial_sum = 0.0;
   pop_index = 1;
   tsf = 0.0;
   for( i=1; i <= popsize; i++)
        tsf = tsf + (sumfitness/work_pop[i].fitness);
   rpw = r1num() * tsf;
```

```
   tfv = (sumfitness/work_pop[pop_index].fitness);
   partial_sum = partial_sum + tfv;
   while(( rpw >= partial_sum) && (pop_index < popsize))
   {
        pop_index++;
        tfv = (sumfitness/work_pop[pop_index].fitness);
        partial_sum = partial_sum + tfv;
   }
   return pop_index;
 }

/********************/
/*                  */
/*      flip()      */
/*                  */
/********************/

int flip(pcross)

 float pcross;
 {
   float rndnum;
   rndnum = r1num();
   if(rndnum <= pcross)
     return 1;
   else
     return 0;
 }

/********************/
/*                  */
/*      find x site */
/*                  */
/********************/

int find_x_site()
 {
   int num, randint;

   (int)randint = r1num()*32767;
   num = (randint % 55) +1;
   return num;
 }
```

```
/*******************/
/*                 */
/*     mutation()  */
/*                 */
/*******************/

int mutation(alleleval)
  int alleleval;
 {
   int mutate;

   mutate = flip(pmutation);  /* mutate with pmutation probability */
   /* change the allele value */
   if(mutate)
   {
     nmutation = nmutation + 1;
     if(alleleval)
          return 0;
     else
          return 1;
   }
   else
     return alleleval;  /* No change occurred */
 }


/*******************/
/*                 */
/*     crossover() */
/*                 */
/*******************/

void crossover(parent1, parent2, child1, child2)
  int parent1[57], parent2[57], child1[57], child2[57];

 {
   int j;

   if(flip(pcross))
   {
     jcross = find_x_site();   /* assumes constant chromosome length */
     ncross = ncross +1;
   }
```

```
    else
      jcross = num_bistring;

    for( j=1; j<= jcross ; j++)
    {
          child1[j] = mutation(parent1[j]);
          child2[j] = mutation(parent2[j]);
    }

    if( jcross != num_bistring)
    {
      for( j = jcross + 1; j <= num_bistring; j++);
      {
           child1[j] = mutation(parent2[j]);
           child2[j] = mutation(parent1[j]);
      }
    }
  }


//*************************/
/*                       */
/*      Set fitness value      */
/*                       */
/*************************/

void set_fitness_value(work_pop, index)
   struct individual work_pop[110];
   int index;

  {
    FILE *fp;
    float prod_rate, avg_prod_rate, req_prod_rate;
    float Cp, Hr, f, Cb, Cfb, Rab, Va, Cu, total_cost;
    int pals, Bs, Ns,WIP, i;

    Bs = 0;
    req_prod_rate = 0.1205
    Cp = 500.0;
    f = 0.16275;
    Cfb = 1500.0;
    Cb = 15000.0;
    Hr = 0.1;
```

```
   Va = 100.0;
   Cu = 0.1 * Va;
   Ns = 10;
   pals = work_pop[index].x[11];
   WIP = pals;

   for(i=1; i<= num_works; i++)
     Bs = Bs + work_pop[index].x[i];


   Rab = 0.2259 + 0.0314*(Bs + Ns);

   if((fp=fopen("hyst021.rat","r")) == NULL)
   {
         printf("Cannot open HYST021.RAT file !\n\n");
         exit(1);
   }
   /* Find average production rate */

   fscanf( fp,"%f\n", &prod_rate);
   fclose(fp);
   avg_prod_rate = prod_rate;
   work_pop[index].rate = prod_rate;

   if(req_prod_rate > avg_prod_rate)
     total_cost = pals*Cp*f+(Bs+Ns)*(Rab*Cfb + Cb*f) + WIP*Hr*Va + (req_prod_rate-
avg_prod_rate)*Cu*52*5*8*3600;
   else
     total_cost = pals * Cp * f + (Bs + Ns) * (Rab * Cfb + Cb * f) + WIP * Hr * Va;
   printf("Total Cost = %12.3f\n",total_cost);
   work_pop[index].fitness = total_cost;

 }

/*************************************/
/*                                   */
/*      write siman experiment file  */
/*                                   */
/*************************************/

void write_siman_exp(work, i)
 struct individual work[110];
 int i;
```

```
{
    int j;

    if((fpout1=fopen("hyst021.exp","w")) == NULL)
    {
        printf("Unable to open HYST021.EXP file !!\n\n");
        exit(2);
    }
    fprintf(fpout1,"BEGIN;\n");

    fprintf(fpout1,"PROJECT,       Thesis assembly model,HYS;\n\n");

    fprintf(fpout1,"ATTRIBUTES:    WkStation;\n\n");

    fprintf(fpout1,"STATIONS:      WorkStation1:\n");
    for(j=2; j < num_works; j++)
        fprintf(fpout1,"               WorkStation%d:\n",j);
    fprintf(fpout1,"               Unload;\n\n");

    fprintf(fpout1,"VARIABLES:     Pallets,          %d:\n",work[i].x[num_works+1]);
    fprintf(fpout1,"               Jam(10),          .0,.03,.03,.0,.0,\n");
    fprintf(fpout1,"                                 .0,.03,.0,.0,.0:\n");
    fprintf(fpout1,"               Buf(10),          %d,",work[i].x[1]);
    for(j=2; j < num_works; j++)
        fprintf(fpout1,"%d,",work[i].x[j]);
    fprintf(fpout1,"%d;\n\n",work[i].x[num_works]);

    fprintf(fpout1,"SEEDS:         1,153,NO:\n");
    fprintf(fpout1,"               2,1515,NO;\n\n");

    fprintf(fpout1,"QUEUES:        Machine1Q:\n");
    for(j=2; j < num_works; j++)
        fprintf(fpout1,"               Machine%dQ:\n",j);
    fprintf(fpout1,"               UnloadoperQ:\n");
    fprintf(fpout1,"               UnloadQ:\n");
    fprintf(fpout1,"               OperatorQ:\n");
    fprintf(fpout1,"               OperLoadQ;\n\n");

    fprintf(fpout1,"RESOURCES:     Machine(9):\n");
    fprintf(fpout1,"               Unloadoper:\n");
    fprintf(fpout1,"               Operator;\n\n");
```

```
    fprintf(fpout1,"SEQUENCES:      1,WorkStation1 &\n");
    for(j=2; j < num_works; j++)
        fprintf(fpout1,"              WorkStation%d &\n",j);
    fprintf(fpout1,"              Unload;\n\n");

    fprintf(fpout1,"CONVEYORS:      Conveyor, 1, 1, 1, A, 1, A, 1;\n\n");

    fprintf(fpout1,"SEGMENTS:       1, Unload,\n");
    for(j=1; j < num_works; j++)
        fprintf(fpout1,"              WorkStation%d - %d,\n",j, work[i].x[j]);
    fprintf(fpout1,"              Unload - %d;\n\n", work[i].x[num_works]);

    fprintf(fpout1,"TALLIES:        Average Cycle Time;\n\n");
    fprintf(fpout1,"FILES:         Rate,\"hyst021.rat\",SEQ,\"(F10.8)\",IGN;\n\n");
    fprintf(fpout1,"REPLICATE,      1, 0, 10000, NO, YES, 3000;\n\n");
    fprintf(fpout1,"END;\n");

    fclose(fpout1);

}

/***************************/
/*                         */
/*      execute_siman()    */
/*                         */
/***************************/

void execute_siman()
{

    system("expmt hyst021.exp hyst021.e");
    printf("finished compiling experiment file\n");
    system("linker  hyst362.m hyst021.e hyst021.p");
    printf("Finished linking model file and experiment file\n");
    system("siman hyst021.p");
    printf("Finished executing siman file\n");
}

/*************************/
/*                       */
/*      Generation()     */
/*                       */
/*************************/
```

```
void generation()
 {
   int i, j, mate1, mate2;

   j = 1;

   while( j <= popsize)
   {
     /* Select a pair of mates */
     mate1 = select_individual(oldpop);
     mate2 = select_individual(oldpop);

     /* crossover and mutations achieved by crossover() */
     crossover( oldpop[mate1].chromo, oldpop[mate2].chromo, newpop[j].chromo,
newpop[j+1].chromo);

     /* Decode string, evaluate fitness, and record parentage date on both children */
     for(i=1; i <= num_works; i++)
     {
          newpop[j].x[i] = decode_num_of_buffers(newpop[j].chromo, i);
       if(newpop[j].x[i] <= 0)
       {
         newpop[j].chromo[5*i] = 1;
         newpop[j].x[i] = 1;
       }
     }

        newpop[j].x[num_works+1]= decode_num_of_pallets(newpop[j].chromo);
        if(newpop[j].x[num_works+1] <= 0)
        {
          newpop[j].chromo[5*(num_works+1)+1] = 1;
          newpop[j].x[num_works +1] = 1;
        }

         /* write siman experiment file */
         write_siman_exp(newpop, j);
         /* Call siman to find prodution rate */
         execute_siman();
         set_fitness_value( newpop, j);

         newpop[j].parent1 = mate1;
         newpop[j].parent2 = mate2;
         newpop[j].cross_site = jcross;
```

```
        for(i=1; i <= num_works; i++)
            {
            newpop[j+1].x[i] = decode_num_of_buffers(newpop[j+1].chromo, i);
              if(newpop[j+1].x[i] <= 0)
                  {
                    newpop[j+1].chromo[5*i] = 1;
                    newpop[j+1].x[i] = 1;
                  }
             }

           newpop[j+1].x[num_works +1] = decode_num_of_pallets(newpop[j+1].chromo);
           if(newpop[j+1].x[num_works +1] <= 0)
            {
              newpop[j+1].chromo[5*(num_works +1) +1] = 1;
              newpop[j+1].x[num_works +1] = 1;
            }
           /* write siman experiment file */
           write_siman_exp(newpop, j+1);
           /* Call siman to find prodution rate */
           execute_siman();
           set_fitness_value(newpop, j+1);

           newpop[j+1].parent1 = mate1;
           newpop[j+1].parent2 = mate2;
           newpop[j+1].cross_site = jcross;

           j = j+2;  /* Increment population index */
    }
  }


/*************************************/
/*                                   */
/*      Decode number of buffers     */
/*                                   */
/*************************************/

int decode_num_of_buffers( num_buffer,i)
  int num_buffer[57], i;

 {
   int j, b32, b16, b8, b4, b2, b1, bsum;
```

```
  j = (i-1) * 5 + 1;
  b16 = num_buffer[j]*16;
  b8 = num_buffer[j+1]*8;
  b4 = num_buffer[j+2]*4;
  b2 = num_buffer[j+3]*2;
  b1 = num_buffer[j+4]*1;
  bsum = b16 + b8 +b4 +b2 +b1;
  return bsum;
}


/*********************************/
/*                               */
/*      Decode num of pallets    */
/*                               */
/*********************************/

int decode_num_of_pallets(station_config)
  int station_config[57];

{
  int i, b32, b16, b8, b4, b2, b1, bsum;

  i = 51;
  b32 = station_config[i]*32;
  b16 = station_config[i+1]*16;
  b8 = station_config[i+2]*8;
  b4 = station_config[i+3]*4;
  b2 = station_config[i+4]*2;
  b1 = station_config[i+5]*1;
  bsum = b32 + b16 + b8 +b4 +b2 +b1;
  return bsum;
}


/*********************************/
/*                               */
/*      create initial population */
/*                               */
/*********************************/

void create_init_population()
{
```

```
  int x1, x2, i;
  /* int sum_buffer=0,value; */

  for( x1 = 1; x1 <= popsize; x1++)
  {
    for( x2=1; x2<=num_bistring; x2++)
        oldpop[x1].chromo[x2] = flip(0.5);
    for(i=1; i <= num_works; i++)
    {
        oldpop[x1].x[i]= decode_num_of_buffers(oldpop[x1].chromo,i);
      if(oldpop[x1].x[i] <= 0)
      {
        oldpop[x1].chromo[5*i] = 1;
        oldpop[x1].x[i] = 1;
      }
    }

    oldpop[x1].x[num_works+1]= decode_num_of_pallets(oldpop[x1].chromo);
    if(oldpop[x1].x[num_works +1] <= 0)
    {
      oldpop[x1].chromo[5*(num_works+1)+1] = 1;
      oldpop[x1].x[num_works +1] = 1;
    }

        /* write siman experiment file */
        write_siman_exp(oldpop, x1);

        /* Call siman to find prodution rate */
        execute_siman();

        set_fitness_value(oldpop, x1);

        oldpop[x1].parent1 = 0;
        oldpop[x1].parent2 = 0;
        oldpop[x1].cross_site = 0;
  }
 }

/*********************************/
/*                               */
/*      Generate initial report  */
/*                               */
/*********************************/
```

```
void generate_init_report()
 {
   fprintf(fpout,"     Searching for Minimal Cost of Closed-Loop\n");
   fprintf(fpout,"Automatic Assembly System with the Genetic Algorithm.\n\n");
   fprintf(fpout,"          Yisheng Hsiao\n");
   fprintf(fpout,"          Thesis Work\n");
   fprintf(fpout,"          Spring 1994\n\n\n");
   fprintf(fpout,"Summary of Parameters\n");
   fprintf(fpout,"   Population Size: %d\n",popsize);
   fprintf(fpout,"   Chromosome Length is fixed at 56.\n");
   fprintf(fpout,"   Maximum number of generations: %d\n",maxgen);
   fprintf(fpout,"   Crossover probability: %6.4f\n",pcross);
   fprintf(fpout,"   Mutation probability: %6.4f\n\n\n",pmutation);
   fprintf(fpout,"Initial Population Statistics\n");
   fprintf(fpout,"   Initial population minimum fitness: %12.3f\n",min);
   fprintf(fpout,"   Initial population maximum fitness: %12.3f\n",max);
   fprintf(fpout,"   Initial population average fitness: %12.3f\n",avg);
   fprintf(fpout,"   Initial population sum of fitness: %12.3f\n",sumfitness);
   fprintf(fpout,"\n\n\n\n\n\n\n\n\n");
 }


/********************/
/*                  */
/*      Statistics()    */
/*                  */
/********************/

void statistics(work_pop)
  struct individual work_pop[110];


 {
   int i;

   sumfitness = work_pop[1].fitness;
   min = work_pop[1].fitness;
   max = work_pop[1].fitness;
   for(i=2; i<=popsize; i++)
   {
     sumfitness = sumfitness + work_pop[i].fitness;
     if(work_pop[i].fitness > max)
          max = work_pop[i].fitness;   /* set new max */
```

```
    if(work_pop[i].fitness < min)
        min = work_pop[i].fitness;   /* set new min */
  }
  avg = sumfitness/popsize;   /* calculation of average */
}


/***************************/
/*                         */
/*      buffer_np_out()    */
/*                         */
/***************************/

void buffer_np_out()
{
  int i, j;
  fprintf(fpout,"popsize= %d\n",popsize);
  for(i=1; i<= popsize; i++)
  {
    for(j=1; j<= num_works; j++)
        fprintf(fpout," %d",newpop[i].x[j]);
    fprintf(fpout," pals= %2d rate=%10.8f
%12.3f\n",newpop[i].x[num_works+1],newpop[i].rate,newpop[i].fitness);
  }
}


/*******************/
/*                 */
/*      Report()   */
/*                 */
/*******************/

void report()
{
  int i, j;

  for(i=1;i <= popsize; i++)
  {
    fprintf(fpout,"O %3d: ",i);
    for(j=1; j<= 56; j++)
        fprintf(fpout,"%1d",oldpop[i].chromo[j]);
    fprintf(fpout,"\n");
```

```
      for(j=1; j<= num_works; j++)
           fprintf(fpout," %2d",oldpop[i].x[j]);
      fprintf(fpout," Pals= %2d",oldpop[i].x[num_works+1]);
      fprintf(fpout," rate= %10.8f %12.3f ][\n",oldpop[i].rate,oldpop[i].fitness);

      fprintf(fpout,"N %3d:[(%2d,%2d) %2d]
",i,newpop[i].parent1,newpop[i].parent2,newpop[i].cross_site);
      for(j=1; j<= 56; j++)
           fprintf(fpout,"%1d",newpop[i].chromo[j]);
      fprintf(fpout,"\n");
      for(j=1; j<= num_works; j++)
           fprintf(fpout," %2d",newpop[i].x[j]);
      fprintf(fpout," Pals= %2d",newpop[i].x[num_works+1]);
      fprintf(fpout," rate= %10.8f %12.3f ][\n",newpop[i].rate,newpop[i].fitness);

   }
   fprintf(fpout,"-------------------------------------------------\n");
   fprintf(fpout,"Generation 1 Stats: max=%12.3f, min=%12.3f, avg=%12.3f,
sumfit=%12.3f\n",max,min,avg,sumfitness);
   fprintf(fpout,"Accumulated Stats: nmutation=%5d, ncross=%5d\n",nmutation,ncross);
   fprintf(fpout,"-------------------------------------------------\n\n\n\n\n");
 }

/********************/
/*                  */
/*      r2num()     */
/*                  */
/********************/

float r2num()
 {
   float ran_num;
   seed1 = (seed1*3993) + 1;
   if(seed1 < 0)
   {
     seed1 += 2147483647;
     seed1 += 1;
   }
   ran_num = seed1/2147483647.0;
   return ran_num;
 }
```