1993

Tandem application of a genetic algorithm and stochastic quasigradient method to the optimization of an assembly system

Kraig Alan Downs
Iowa State University

Tandem application of a genetic algorithm and stochastic quasigradient method to the optimization of an assembly system

by

Kraig Alan Downs

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

iversity
Ames, Iowa

1993

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGEMENTS

I would like to thank Dr. Doug Gemmill for his guidance throughout this research. His help made the process of completing my thesis go very smoothly.

I would like to extend my thanks to the staff at Ford Refrigeration and Electronics for their cooperation. I especially thank John Marlin and Al Kleave for their time and effort during my data collection phase.

This research would not have been possible it were not for my parents. You have always provided me with everything I have ever needed and have always been there during the rough times. I will always be grateful for all you have done for me.

I purposely left the final thanks for the most deserving, my wife Joni. Nearly four years ago you made a vow to stand beside me while I pursue my dreams and aspirations. You not only managed to be supportive, but you also made real strides in your own career. I am eternally in your debt for all your sacrifices and support during this research and throughout my college years.

1. INTRODUCTION

Manufacturing is an important part of the U.S. economy. The bureau of Labor Statistics reports that about 18 million Americans are employed in manufacturing (Schloemer 1992). It was estimated that in the late 1980's manufacturing-linked U.S. employment fell in the 40 to 50 million range (Schloemer 1992). These figures clearly show the importance of manufacturing in the U.S. economy.

The resulting activity of manufacturing can be classified into either fabrication or assembly. All manufacturing firms perform fabrication, assembly, or both. It was estimated that in the U.S. about eight million are employed in the area of manufacturing processes associated with product assembly (Liu and Sanders 1988). The degree to which assembly occurs can vary tremendously. For example, an automobile requires considerable more assembly than an office chair. In any case, a great number of products produced by manufacturing firms require assembly. Usually, the more complex the product the more that assembly becomes a process issue. The importance of assembly is clearly illustrated in a recently proposed approach to product design (Nevins and Whitney 1989). Nevins and Whitney propose a strategic approach to product design in which all concurrent design activities are centered around the determination of assembly sequences and the choice of assembly systems. Nevins and Whitney's approach parallels the hot topic of design for assembly (DFA). The important point to keep in mind about DFA is it recognizes that assembly must be considered at an early stage to insure proper product function, quality, and manufacturability. The discussion above clearly points out the importance of assembly in manufacturing.

Assembly systems can be described by classifying their main components. An assembly system is generally composed of two main parts: the assembly process and the

transfer mechanism. The assembly process can be further be divided into the three distinct areas listed below (Groover 1987):

1. Manual single-station assembly
2. Manual assembly line
3. Automated assembly

Manual single-station assembly involves one workplace with one or more workers assembling a product. This method is common among low-volume or highly complex products. Manual assembly line arrangements have workers in a line each contributing something to the assembly of a given product. Automotive assembly is often performed using this assembly process. The third assembly process classification, automated assembly, is simply assembly performed by automatic equipment.

The transfer mechanism refers to the means by which assemblies or subassemblies are moved to, between, or away from assembly stations. Transfer mechanisms are nothing more than material handling systems specifically setup to accommodate a given assembly process. These mechanisms can be further classified as either manual or automated. Manual transfer requires that workers physically move the assemblies or subassemblies from one location to another. Automated transfer is accomplished by using some type of conveyor system. Automated transfer mechanisms are even further subdivided into continuous, synchronous, or asynchronous. In continuous transfer, the worker or the automatic machine must perform the proper operation while the assemblies or subassemblies are moving. Synchronous transfer is characterized by the flow of assemblies or subassemblies occurring simultaneously at specific points in time. Asynchronous transfer occurs when an assembly or subassembly is moved as soon as processing is finished. The continuous and synchronous methods have maximum production rates equal to the rate of the transfer mechanism. The asynchronous method's production rate is not as easily determined. The production rate

in the asynchronous transfer case can be affected by blocking and starving. Asynchronous transfer can allow more effective use of the stations where operations occur; however, this added flexibility can introduce additional stochastic effects, not found in synchronous systems.

To better understand transfer mechanisms it is necessary to make a distinction between open and closed systems. An open system is one in which items travel along some path that has definite beginning and ending points. The closed system is analogous to a loop: products, carriers, and/or fixtures travel along a closed path. One key feature of the closed loop configuration is that the components, subassemblies, or assemblies are introduced into the loop at some point and usually exit before traveling the entire distance of the closed path. The closed loop configuration is especially useful for recirculating fixtures or carriers and also for recirculating items to specific locations in the system.

Analysis of assembly systems has traditionally been divided into three different categories. These categories are deterministic models, queueing theory models, and simulation. Deterministic modeling involves extensive data collection and analysis: the goal is to develop equations which predict the behavior of the system. Queueing theory concerns itself with the mathematical analysis of customer-server type relationships. Some major issues in queueing theory are service times, queue lengths, and server utilization (Gross and Harris 1985). Simulation involves the development of a model having similar characteristics as the system of interest. Simulation is often performed on a computer, allowing the collection of queueing theory type statistics. Deterministic modeling works well in the analysis of deterministic systems. However, very few systems are actually deterministic. It is possible to use deterministic modeling on systems that contain stochastic elements, but model development can be extremely

difficult and always produces "case specific" models. Queueing theory is limited in application by the complexity of the system of interest. If the system being modeled has a simple server-customer relationship, queueing theory can often be applied quickly and effectively. Also, queueing theory allows for stochastic elements (e.g. probabilistic service and customer arrival times). Obviously, many systems involve much more than simple server-customer relationships. Blocking and starving effects are examples of factors that make queueing theory analysis somewhat ineffective. Simulation is the most effective of the three modeling techniques to properly incorporate the complicated effects of blocking and starving in stochastic systems. A major reason for using simulation to model complex systems, is to enable the application of optimization techniques.

Optimization can be thought of as the a process which seeks to improve performance towards some optimal point or optimal parameter set (Goldberg 1989). For the purposes of this research, optimization implies the search for the set of operating parameters which gives the most desirable performance measure value, while not violating given system constraints. There are three basic approaches to the search for an optimum solution: gradient methods, enumerative methods, and random search methods (Goldberg 1989). Gradient methods involve finding minimum or maximum values using slopes or derivatives. These methods are reasonably effective for well-behaved functions or systems. Well-behaved refers to a continuous function having relatively few local minima or maxima. Enumerative methods can simply be described as the evaluation of all possible combinations of systems parameters. For some smaller systems this approach is feasible; however, in many situations this is not the case. Often, time and cost discourage the use of enumerative approaches to optimization. Random search methods, or stochastic search methods, are intended to successfully optimize a parameter or set of parameters in a stochastic environment. Random search methods are best

represented by techniques such as simulated annealing and the genetic algorithm. These techniques are meant to overcome the problems that gradient methods have with "hanging up" on local minima or maxima.

This research investigates the tandem application of a genetic algorithm and a gradient method (specifically the stochastic quasigradient method) to the optimization of an asynchronous semi-automatic assembly system. The intent of the tandem application is to utilize the strengths of each optimization technique. A genetic algorithm will be used to identify the interesting features (peaks or valleys) of the response surface and the stochastic quasigradient method will investigate the local areas around these features. A simulation of an actual assembly system is used to obtain performance measures for different sets of input parameters. A detailed description of the assembly system and the simulation model is provided in Chapter 3.

2. REVIEW OF RELEVANT LITERATURE

The performance or behavior of many systems is not entirely deterministic. This includes systems for manufacturing, assembly, transportation, service, communications, etc. Each system has unique aspects which can introduce stochastic elements at different stages. Optimizing the performance of systems influenced by the effects of stochastic elements has been studied in detail. Methods used for stochastic optimization are also commonly referred to as Monte Carlo methods. The first section of this chapter briefly describes the different techniques used in the optimization of stochastic systems. The next two sections will review the research on genetic algorithms and stochastic quasigradient methods respectively. The fourth section will review the work performed on the optimization of assembly systems. The fifth section of this chapter will look into research on hybrid optimization techniques. The final section is a general summary of the literature as related to the research objectives of this thesis.

2.1. Optimization Techniques for Stochastic Systems

The following list provides some of the more common methods used in the attempt to optimize stochastic systems; however, this list is by no means exhaustive. Stochastic optimization techniques include stochastic quasigradient methods (SQG), Robbins-Monro Algorithm, Kiefer-Wolfowitz Algorithm, response surface methodology, and optimization homotopy. Deterministic optimization techniques that have been adapted for use with stochastic problems include genetic algorithms and simulated annealing. In their purest form, SQG methods, Robbins-Monro Algorithm, Kiefer-Wolfowitz Algorithm, and response surface methodology are continuous parameter stochastic optimization techniques (Glynn 1986), while genetic algorithms and simulated annealing are considered to be discrete parameter deterministic optimization techniques.

Optimization homotopy can be used for both discrete and continuous parameter optimization; however, it is actaully better for continuous parameter optimization because it assumes a continuous "path." All of these continuous parameter optimization techniques involve gradient calculations (Glynn 1986). Working descriptions of each of these gradient type algorithms are provided by Glynn (1986) and Gemmill (1988). Even though the aforementioned optimization techniques were intended for continuous functions, there has been considerable application of these methods to discrete parameter or discrete function problems. A later section of this chapter will specifically describe the application of SQG to discrete parameter problems.

As previously mentioned, simulated annealing and genetic algorithms are designed for discrete parameter optimization problems. Both of these optimization techniques are random search algorithms based on processes found in nature: simulated annealing - thermodynamics, genetic algorithms - natural selection (Davis 1987). The simulated annealing algorithm recognizes a connection between statistical mechanics and combinatorial optimization. This technique was first suggested by Kirkpatrick *et al.* (1983). Statistical mechanics is the study of the behavior of large systems of interacting components. This includes the atomic behavior of a solid in thermal equilibrium at a finite temperature (Davis 1987). Simulated annealing has been applied to a variety of problems including computer design (Kirkpatrick *et al.* 1983), the traveling salesman problem (Bonomi and Lutton 1984), the portfolio problem (Gemmill 1988), and flexible manufacturing systems design (Lie 1991).

Genetic algorithms were formally introduced in *Adaptation in Natural and Artificial Systems* (Holland 1975). These algorithms implement simple genetic operations such as reproduction, crossover, and mutation to optimize a given set of parameters. Genetic algorithms are based on the "survival of the fittest" concept. In

nature, the most fit individuals tend to have a higher survival rate, and thus are larger contributors to the gene pool of a given generation.

Three optimization methods have been given considerably more attention in recent research: SQG, simulated annealing, and genetic algorithms. All three of these methods have been successfully applied to various problems. As pointed out in the introduction, this research involves the tandem application of a random search method and a gradient method. Both simulated annealing and the genetic algorithms are random search techniques, and SQG is a gradient technique. For the specific assembly system being researched, the parameter encoding process for genetic algorithms is more logical and intuitive than that of simulated annealing. Therefore, a detailed review of pertinent literature involving genetic algorithms and SQG is given. The reader is reminded that heuristic versions of both genetic and SQG algorithms are used in this research. If these two algorithms are referenced as stochastic optimization techniques, the reference is aimed at the heuristic versions.

2.2. Genetic Algorithms in Optimization

As discussed previously, genetic algorithms operate in a similar manner to the natural selection process. The actual mechanics of the algorithm are presented in chapter 4. The literature on genetic algorithms can be divided into two distinct categories: algorithm development/description and applications.

2.2.1. *Literature on the development and description of genetic algorithms*

Holland (1975) established the application of the adaptive characteristics of natural systems to artificial systems. For all practical purposes, he can be considered the founder of genetic algorithms. The mathematical foundation of genetic algorithms is laid

forth in this book. He generalized the concepts of reproduction, crossover, and mutation to explain how genetic algorithms provide a robust search in a complex solution space. Holland made a significant contribution in the area of stochastic optimization by formally introducing genetic algorithms.

Davis (1987) compiled a set of 13 papers into a book. These papers discuss various issues concerning genetic algorithms and simulated annealing. There are several relevant articles presented in this book. Davis and Steenstrup (1987) provided a concise description of both genetic algorithms and simulated annealing. John Grefenstette (1987) discussed the incorporation of problem-specific information into genetic algorithms. He suggested that since genetic algorithms are not especially good for fine local searches, one could use genetic algorithms to identify "promising" regions, and then invoke a local search method to hone in on the optimum solution. This same point was also mentioned by Goldberg (1989). Grefenstette used the traveling salesman problem to illustrate several heuristic methods for population initialization. Most genetic algorithms begin with a random initial population. The research proposed to begin with a "good" initial population rather than a completely random one. The "good" initial population is selected using cost information. Grefenstette concluded that heuristic information is effective, but must be applied with caution to refrain from causing premature convergence of the solution. Another significant paper presented in this same book is one written by Goldberg (1987) about the behavior of simple genetic algorithms when applied to the minimal, deceptive problem (MDP). The MDP is designed to mislead the simple genetic algorithm away from the global optimum solution and toward sub-optimal solutions. Goldberg concluded that the simple genetic algorithm converged across a wide range of initial parameters; therefore, eluding the distractions presented by the MDP.

Pettey *et al.* (1987) suggested the use of a parallel genetic algorithm to improve the search time in problems with large populations. In cases when the population is small, genetic algorithms can be incorrectly constrained, in terms of the search space. Conversely, if the population is overly large, genetic algorithms can take an inordinate amount of time to run. The authors present a class of parallel genetic algorithms (PGA) to overcome this problem of excessive run time. The Traveling Salesman Problem is used as an example to illustrate a PGA. The algorithm described simultaneously processes several generations of strings (individuals): to accomplish this, multiprocessor technology is utilized. The authors' findings indicate that a PGA can allow for an increased population size of a genetic search.

Richardson *et al.* (1989) presented some steps to implementing penalty functions in genetic algorithms. After a specific problem has been coded into a genetic algorithm, it is not uncommon to have particular combinations of bits in the bit string be infeasible or illegal. In this penalty scheme, these infeasible or illegal combinations are given a substantial penalty. It is explained that historical recommendations for applying penalty functions advised using harsh penalties for illegal solutions. By assigning a large penalty to the performance measure of the inappropriate solution, it would be forced out of the population. Richardson *et al.* (1989) advised that a well chosen, graded penalty is more desirable than harsh penalties. He claimed that these types of penalties preserve the information for all strings where harsh penalties do not. This concept will be revisited in a later chapter.

Goldberg (1989) published a textbook addressing the issues of genetic algorithms. This text provided a straightforward introduction to the history and operation of genetic algorithms. The simple genetic operators such as crossover, reproduction, and mutation used in the algorithm are described in detail. Goldberg included a chapter discussing the

mathematical foundations, for the theorists. This text also discussed several advanced genetic operators including dominance and abeyance. Goldberg also presented several knowledge-based techniques that incorporate genetic algorithms: hybridization and knowledge-augmentation are two such techniques. Hybrid schemes involve the crossing of a genetic algorithm with a problem-specific optimization or search technique. Knowledge-augmented techniques involve enhancing a genetic algorithm with some "problem-specific" information. Parallel genetic algorithm schemes are also described. Parallel genetic algorithms imply that there are several different, but parallel, generations operating simultaneously being directed by a single master. This is analogous to a computer network having a server. The text also spends considerable time discussing genetic algorithms in machine learning. Goldberg's main contribution to genetic algorithm research was threefold. First, the text provided Pascal computer code for a simple genetic algorithm (SGA). The SGA incorporates the three fundamental genetic operators, which are reproduction, crossover, and mutation. Second, Goldberg's text thoroughly discussed all aspects of genetic algorithms. Finally, this text compiled a nearly exhaustive list of references pertaining to genetic algorithms.

In 1991, Lawrence Davis published the *Handbook of Genetic Algorithms* (Davis 1991). Davis presented a clear description of genetic algorithms, including history, explanation of operation, and variations. These variations discuss hybridization of and parameterizing genetic algorithms. Davis dedicated an extensive portion of the text to application studies of genetic algorithms. Applications are given for aircraft design, neural network architecture design, schedule optimization, and robot trajectory generation, to name a few. Unlike any previous publications, Davis presented the coding of genetic algorithms from an object-oriented point of view. In fact, Davis refers to his code as the Object-Oriented Genetic Algorithm (OOGA). Overall, this book provides

two significant contributions to genetic algorithm research: a large pool of case studies involving the application of genetic algorithms and an object-oriented approach to genetic algorithms.

2.2.2. *Literature on the application of genetic algorithms*

There are numerous 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 mainly concentrates on research relevant to this thesis.

One of the first applications of genetic algorithms was given by Albert D. Bethke at the University of Michigan (Bethke 1978). Bethke used genetic algorithms as function optimizers. The traditional way of optimizing some multiple variable stochastic functions was to use calculus-based methods. This article proposed the use of genetic algorithms instead of these traditional methods. The mechanics of genetic algorithms are presented in the context of function optimization. Bethke concluded that the genetic algorithms are far less sensitive to noise than gradient or calculus-based methods. This is illustrated with an example of finding the maximum value of a bi-modal objective function using a genetic algorithm and a gradient method.

Davis and Ritter (1987) presented an interesting optimization application using genetic algorithms. They used genetic algorithms to optimize the performance of a simulated annealing algorithm. The simulated annealing algorithm was used to optimize student class schedules. Specifically, Davis and Ritter applied a genetic algorithm to optimize the annealing parameters. The research concluded that the application of genetic algorithms in this capacity was able to find better annealing parameter settings

than humans found. Davis and Ritter's contribution lies in their integration of two optimization techniques.

Glover (1987) presented an article on solving a keyboard configuration problem using genetic algorithms. The motivation of this research is the configuration difficulties brought about when attempting to map keyboards for the Eastern Asian languages. The article recognized that it is difficult to apply expert systems to large combinatorial type problems. A prototype algorithm meant to illustrate the robustness of the genetic algorithm was proposed and tested. Glover concluded by stating that genetic algorithms provide a robust search technique when applied with representation and operator constraints.

Cohoon *et al.* (1988) tested distributed genetic algorithms on the floor plan design problem. The floor plan design problem involves determining the optimal arrangement of rectangular features in a given area. The particular application given is the placement of modules in the VLSI design cycle; the objective of the placement is to minimize the wire lengths and the weighted sum of the area. They implemented a distributed genetic algorithm using multiple processors (referred to as GAPE). After evolving several fit sub-populations, GAPE combines these groups into one large generation. The algorithm then proceeds to evolve this single population. Cohoon *et al.* observed that GAPE performed consistently better than applying genetic algorithms in a sequential manner.

Wellman (1991) applied a simple genetic algorithm to the optimization of buffer allocation in a closed-loop asynchronous automatic assembly system. Wellman's research focused on the application issues of a simple genetic algorithm and the algorithm's relative performance compared to the work of others. Wellman explored the simple genetic algorithm's performance at various settings of the population size, the crossover probability, and the mutation probability. The research compares the results of

a simple genetic algorithm to the results of Liu and Sanders' (1988) work: Liu and Sanders (1988) applied a stochastic quasigradient method to the same problem. Wellman found that the simple genetic algorithm did not perform well in comparison to Liu and Sander's SQG method. The simple genetic algorithm consumed much more computer time than the SQG method. However, he showed that the genetic algorithm does produce reasonable results. Wellman's work provided two important contributions to genetic algorithm research. His first contribution was the application of genetic algorithms to the optimization of assembly systems. The other contribution was in the comparison of the performance of two different stochastic optimization methods.

Huntley and Brown (1991) applied a parallel heuristic to the quadratic assignment problem. Their algorithm (SAGA) operated by cascading a genetic algorithm and a simulated annealing method. The heuristic is considered parallel because of its intended implementation using parallel computers or processors. Huntley and Brown developed SAGA with the idea of combining decentralized characteristics of genetic algorithms and centralized characteristics of simulated annealing methods. A genetic algorithm is used to generate populations and then simulated annealing "matures" these populations. Huntley and Brown concluded that SAGA performed favorably on two standard problems found in the related literature. However, SAGA had a longer runtime than some less complex algorithms. The main contribution of their work is the cascading configuration of SAGA. They recognized that a genetic algorithm was good for general searches and that simulated annealing was better for local searches.

The applications of genetic algorithms given above only scratch the surface of what is available. Some additional applications include control systems, process design and optimization, neural networks, and machine learning.

2.3. Stochastic Quasigradient Methods in Optimization

Formal stochastic quasigradient (SQG) methods were introduced by Ermoliev (1969). They were intended to be applied to stochastic and nonlinear programming problems. The fundamental idea of these methods is to implement statistical estimates of function values (e.g. gradients) instead of exact function values. The published research on SQG is reviewed in two different sections: theoretical development work and applications.

2.3.1. *Theoretical development of SQG*

Ermoliev (1969) provided the foundations for SQG in an article discussing stochastic gradients and quasi-feyer sequences. This early research considered problems of random search and adaptive minimization. Kushner (1974) further developed SQG by showing convergence theorems for stochastic approximation methods in finding local minima. Gupal and Norkin (1977) presented a stochastic finite-difference method to be used in the minimization of discontinuous functions.

Yuri Ermoliev (1983) presented a paper on using SQG methods for the optimization of systems. SQG methods are made for solving complex stochastic functions. Ermoliev pointed out that stochastic processes are important due to their common existence. He reviewed recent work involving SQG methods. One significant issue Ermoliev discussed is use of penalty functions. The minimization of a penalty function can be a viable objective function. The author cautioned that this method of using a penalty function may not converge under certain conditions.

Ermoliev and Gaivoronski (1984) presented several SQG methods and their respective computer implementations. They clearly illustrated each step required in the implementation of a SQG method. Discussion of step size choice and step direction

calculations were given. Several suggestions were made for determining step direction when observations of the gradient are not available. The methods given were central and forward finite difference approximations. They provided some guidelines on choosing step sizes at different stages in the optimization process. They recommended the application of SQG using an interactive approach. Ermoliev and Gaivoronski also described a computer software package to perform SQG (STO). The STO program is applied to practical problems; this included examples involving facility location and the control law problem. This paper's significant contribution is in the step-by-step discussion of the operation and implementation of SQG. Many recent applications of SQG cite this article as providing the key explanation of the technique.

Liu (1987) provided a reasonably thorough history of the development of SQG methods. He also provided a detailed description of the mechanics of SQG. Other references describing the operation of SQG algorithms include (Tandiono 1991) and (Gemmill 1988). A description of the mechanics of SQG methods is provided in Chapter 4.

2.3.2. *Applications of SQG*

The applications discussed in this section are relevant to this research and the industrial engineering field in general. Unlike genetic algorithms, SQG has been applied to the optimization of several different assembly systems. Many of the references cited below will be discussed from an algorithm viewpoint now, and in a later section will be presented from an assembly system viewpoint.

Liu (1987) implemented SQG to the design optimization of an asynchronous automatic assembly system. Liu used the SQG methods as described by Ermoliev and Gaivoronski (1984). The research indicated that using forward finite difference methods

provided an acceptable alternative to central finite difference methods. Liu pointed out several disadvantages of the SQG method. One major weakness of SQG is the difficulty in choosing a "good" initial solution set.

Gemmill (1988) applied SQG to the portfolio problem. His research found that SQG tends to converge to local optima instead of global optima; the starting point tends to significantly affect the probability of converging to a local optima. Gemmill also found the algorithm to converge rather slowly.

Liu and Sanders (1988) used the SQG method, as described by Ermoliev and Gaivoronski (1984), for the performance optimization of an asynchronous flexible assembly system. Both starving and blocking effects were introduced into the assembly system model. Their research used a queueing network model to set the number of pallets and then used an SQG algorithm to determine buffer spacing for optimal system throughput. Liu and Sanders' method can be considered a hybrid technique. They concluded that their queueing network/SQG algorithm produced reasonable results in a difficult area which is serviced by very few techniques.

Tandiono (1991) presented research involving the optimization of an automatic assembly system (AAS) from a cost perspective. She implemented SQG methods to optimize a simulation of a given AAS. Tandiono found that by using an objective function involving cost, SQG could simultaneously optimize the number of pallets and buffer sizes. From the research, she discovered that SQG performance was somewhat dependent upon step size choice. A penalty function was integrated into the optimization procedure. Discussion warned that the penalty should be severe enough so that inappropriate parameter sets will not be accepted as optima.

Bulgak and Sanders (1991) proposed a technique for the design optimization of asynchronous flexible assembly systems with statistical process control and repair. Their

algorithm uses a queueing network model to determine the number of pallets needed to achieve a desired system throughput, and then applies a SQG method. They also test their algorithm using a modified simulated annealing method instead of SQG. Bulgak and Sanders concluded that both of their hybrid algorithms worked well in producing optimal, or near optimal, design parameters for flexible assembly systems with automated SPC and repair.

2.4. Optimization of Assembly Systems

This section reviews the literature involving the optimization of assembly systems. Various pieces of literature already reviewed in this chapter have mentioned assembly systems in the context of stochastic optimization techniques. This section will revisit some of those same articles; however, the focus will be on the assembly system itself.

The assembly systems used in the articles discussing stochastic optimization can be divided into two general categories: asynchronous automatic assembly systems and asynchronous flexible assembly systems. Several researchers used special variations of these two general categories, and those will be discussed individually.

2.4.1. *Asynchronous automatic assembly systems*

Asynchronous automatic assembly systems can be thought of as either open or closed. This review focuses on closed AASs. These systems generally consist of a series of workstations in which each perform partial assembly of an object. One key aspect of these systems is the in-line or series structure. The object being assembled is attached to a pallet, and the pallet is transported from station to station via a conveyor system. Each station houses some automated assembly process. In many cases, the operation times are

considered to be constant and there are assigned jam probabilities at each station. The most common decision variables used in the analysis of AASs are number of pallets and buffer sizes. Those researchers that have used a basic asynchronous automatic assembly system include Kamath and Sanders (1986), Kamath and Sanders (1987), Liu (1987), Bulgak and Sanders (1988), Liu and Sanders (1989), Liu and Chiou (1990), Wellman (1991), and Tandiono (1991).

2.4.2. *Asynchronous flexible assembly systems*

Asynchronous flexible assembly systems (AFAS) are manufacturing systems designed to provide a high degree of automation and a high degree of flexibility. One way to describe AFASs is as a controlled process which can assemble various parts and products according to a determined schedule (Andreasen and Ahm 1988). The general components of AFASs are assembly operations, material handling systems and components/component types (Lie 1991). The most common decision variables are number of assembly cells, buffer spaces, and batch sizes. Individuals having done research on the optimization of AFASs include Kamath *et al.* (1988), Liu and Sanders (1989), Bulgak and Sanders (1989), Lie (1991), and Bulgak and Sanders (1991).

2.4.3. *Special cases of asynchronous assembly systems*

There are two special cases of assembly systems found in the literature that are relevant. Bulgak and Sanders (1989, 1991, 1991) implemented an asynchronous automatic assembly system which had the unique features of an automatic test station and a repair loop. Their system configuration consisted of a double loop arrangement: one loop contained all the assembly operations while the other loop contained a repair station. Bulgak and Sanders used this system in three separate publications.

The other special case of AASs involved a system using a tunnel-gated station (Leung and Sanders 1986). Tunnel-gated stations were designed to lift an assembly up off of the transfer line. Operations are performed on assemblies in the raised position, while other assemblies are allowed to pass underneath. This type of system could be used to implement parallel station configurations.

2.5. Hybrid Stochastic Optimization Methods

Hybrid optimization algorithms can be thought of as the integration of two different algorithms. Hybrid schemes can also be knowledge enhanced algorithms. The hybrid designs are created to provide better performance than either of its constituent algorithms. The goal of hybridization is to find an algorithm that is more robust than current techniques (Davis 1991). Both Goldberg (1989) and Davis (1991) suggested hybrid schemes, described the rationale behind them, and discussed the general approach to their creation.

As mentioned previously, Davis and Ritter (1987) constructed a hybrid stochastic optimization technique for student scheduling. Davis and Ritter described their technique as a probabilistic search routine combining the genetic algorithm and simulated annealing. Their hybrid technique used simulated annealing to optimize the class scheduling problem, while concurrently applying a genetic algorithm to optimize the simulated annealing parameters. They determined that their algorithm was more successful and faster at student scheduling than the people who currently perform it. Davis and Ritter also suggested that their algorithm can be applied to other problems of the same type.

As mentioned earlier, Liu and Sanders (1988) presented a hybrid algorithm to optimize the system performance of flexible assembly systems. They used a queueing

network model to determine the number of pallets in the system and then applied a SQG method to determine the buffer sizes. They referred to their technique as the combined Network-SQG method. Liu and Sanders found their algorithm performed reasonably well on the complex problem of AFAS optimization.

Bulgak and Sanders (1991) implemented a two stage algorithm for the stochastic optimization of asynchronous flexible assembly systems with SPC and repair. They used a queueing network model in the first stage to predict the number of pallets required to meet a desired throughput. The second stage of their hybrid algorithm utilizes a stochastic optimization technique to set buffer sizes. Bulgak and Sanders experimented with two different stochastic optimization techniques for use in the second stage: SQG and simulated annealing. They concluded that both versions of their hybrid algorithm worked; however, the SQG version was computationally more efficient than the simulated annealing version. It was also found that simulated annealing had more potential for application to a wider variety of problem types.

The last hybrid method discussed in this review is the cascaded algorithm proposed by Huntley and Brown (1991). As mentioned before, their heuristic used a genetic algorithm cascaded with a simulated annealing algorithm. The genetic algorithm is used to create good populations and the simulated annealing procedure is applied to mature the populations. Their intention was to implement this algorithm using parallel processors. The cascading application of two stochastic optimization techniques, as presented by Huntley and Brown, is similar to some of the research in this thesis.

2.6. Summary of Literature

Genetic algorithms have been applied to numerous practical and theoretical problems. Some researchers found genetic algorithms to be successful in their

applications, others did not. In the attempt to find a more robust approach, some researchers integrated specific knowledge about their particular problem into a genetic algorithm.

In their purest form, SQG methods are intended for locating optima in a continuous solution space. In the search for stochastic optimization techniques, researchers found heuristic ways to apply SQG to discrete or discontinuous solution spaces. This opened up the application of SQG methods to a much wider field of practical problems. Even with the ability to be applied to discrete parameter problems, SQG methods have not been widely applied.

The research done on the stochastic optimization of assembly systems has been almost exclusively focused on asynchronous automatic assembly systems and asynchronous flexible assembly systems, with a few variations such as repair loops and tunnel-gated stations. The assembly systems described in the literature were theoretical rather than actual. However, some system features were based on actual equipment (e.g. tunnel-gated stations).

In the absence of desired performance, researchers developed hybrid stochastic optimization algorithms. Hybrid algorithms were created by integrating two optimization techniques or augmenting a single technique with problem specific knowledge. These algorithms strive to be more robust than their constituent algorithms.

The research presented in this thesis provides two new contributions to the field of stochastic optimization. First, the simulated assembly system is a detailed representation of an actual assembly system. This assembly system has parallel operations and also has both automatic and manual stations. To the best of our knowledge, this is the first research aimed at optimizing an assembly system with both

manual and automatic stations. Second, the optimization technique involves the tandem application of a genetic algorithm and a SQG method.

3. SYSTEM DESCRIPTION

The assembly system described in this research is an actual system used by Ford Refrigeration and Electronics (Connersville, IN) to assemble plate/fin style evaporators. Evaporators are a vital component in the climate control system of the automobile. The evaporator is a component which is not readily familiar to most people. Their unfamiliarity is due to the fact that when assembled in the automobile, the evaporator is completely enclosed in a plastic housing. This housing is located in the rear of the engine compartment, usually in close proximity with the fire wall. The evaporator is responsible for removing heat from the air used to cool the passenger compartment. Refrigerant enters the evaporator in a low pressure liquid state. As the refrigerant passes through the plate/fin network it picks up heat and changes from a low pressure liquid to a low pressure gas. Blowers in the automobile force warm air across the plate/fin network. The boiling refrigerant removes heat from the warm air, and this air cools the passenger compartment (Dwiggins 1978).

3.1. Description of Actual System

This assembly system is classified as a palletized asynchronous semi-automated build line. A general overhead view is illustrated in Figure 3.1. There are four major components to this assembly system: build stations, the vision system, core banding, and core unloading. The pallets are designed with fixtures attached to them. The fixtures assist the workers in properly assembling evaporators. The network of transport track is at a single elevation: all pallets move in the same x-y plane. The complete transport network structure is defined by several connected segments. The conveyor speed is essentially constant at 10.38 in/sec. This is the average speed of a loaded pallet determined by direct observation. The transportation network is arranged in a

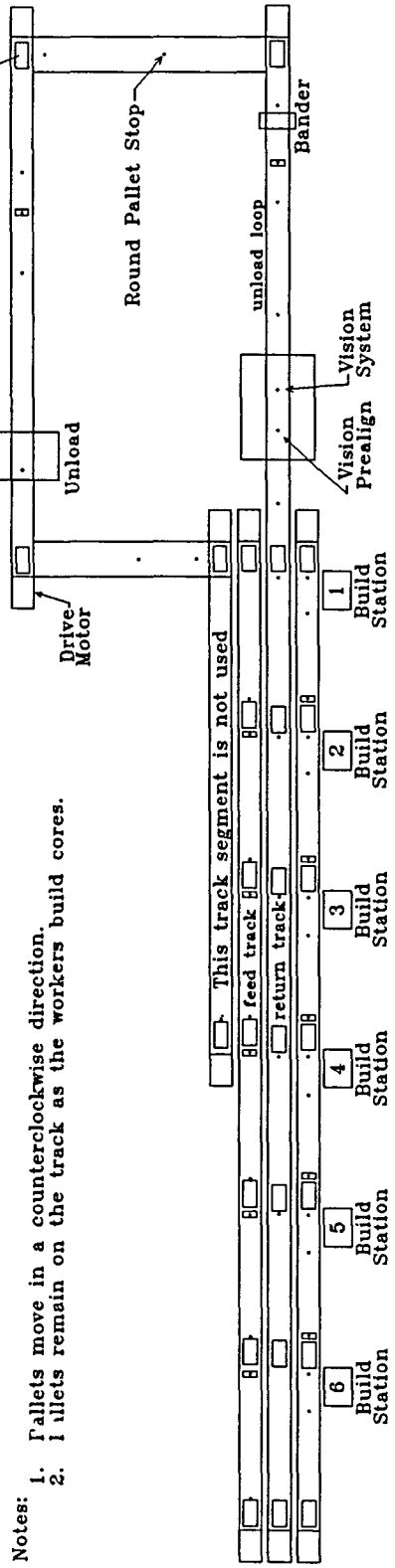

Figure 3.1 Top view of evaporator assembly system

rectangular configuration, as shown in Figure 3.1. The structure is such that pallets move throughout the system in a counterclockwise direction. Pallets move strictly in straight-line motion, there are no arc type motions. As a pallet is being transported by a conveyor, it will eventually reach the end of the current segment. When this occurs, pallets are raised by mechanisms referred to in this research as lifts. These lifts transport pallets from one conveyor segment to another. Pallets are never rotated: when they turn 90° corners, the pallets remain oriented in the same direction. Upon completion of assembly, the pallet (with attached evaporator core) is released. The pallet moves from the build station onto the return track. Once on the return track, the pallet moves towards the unload loop. During the pallet's stay on the unload loop, it must visit four different automatic stations: vision prealign, vision inspection, bander, and core unload. These stations are visited in the order given, without exception. During the first stop, vision prealign straightens the assembled evaporator core, readying it for the vision inspection. Upon arrival at vision inspection, the evaporator core is compressed. The vision system then takes several fixed perspective snapshots of the core. This station's purpose is to identify any problems due to incorrect assembly, damaged components, or missing components. The system determines whether the evaporator core is a reject and then encodes this information on a programmable chip located on the pallet. The next station is banding. The evaporator core is banded only if the vision system accepted it. The banding process begins by first compressing the evaporator core and then securing it in the compressed form using a pair of metal bands. This compressed form is necessary for a joining process used in the next stage of manufacturing. After the banding station, the pallets (with assembled evaporator cores) travel to the unload station. There are proximity sensors at the unload station which detect the presence of the metal bands. If the metal bands are present, a robotic arm removes the evaporator core from the fixtured

pallet. If the proximity sensors do not detect metal bands, the evaporator core will remain on the pallet. The empty pallets are circulated back to the build stations. The pallets containing rejected evaporator cores are sent back to the station in which they were assembled. Pallets, both empty and those containing rejected evaporator cores, travel back to the build stations via the feed track. The feed track is analogous to a multiple-opening gravity feed bin; it is a one-way non-recirculating conveyor. As pallets move down the feed track, they may be sent into a build station if needed. Otherwise, the pallets continue down the feed track towards build station 6.

The conveyors in this system are set up using zones. Since pallets are moderately heavy, their travel around the system must be regulated to prevent both excess weight in concentrated areas and pallet collisions. Zones are established by holding pallets at designated stops according to a pre-established set of rules. Another important feature of the conveyor system is located at each 90° bend in the track system. When a pallet encounters a 90° bend, a lift raises the pallet up and transfers it to the next conveyor segment. Zone logic is such that pallets will not collide on a lift. Collision on a lift could cause damage to the conveyor system. There are many intersections in the track network structure. These intersections are carefully controlled to prevent pallet collisions. Most intersections are governed on first-come-first-serve basis, except in the case when a rejected evaporator is being routed back to its builder. In this special case, the rejected core has priority at intersections.

3.2. The Simulation Model

To enable performance analysis of stochastic optimization techniques, a simulation model was developed for the system described in section 3.1. It was impossible to use the actual evaporator core build system to test the stochastic

optimization methods. The simulation was developed using SIMAN, a commercial simulation package from Systems Modeling Corporation. Observation of the actual system was used to determine system parameters for the simulation model. A considerable amount of effort was put into collecting system parameter data. Since all necessary information was provided or could be collected, the simulation model incorporated as much detail as possible. A modified version of the approach to conducting a simulation study, as presented by Law and Kelton (1991), was used to create the simulation model. The steps are listed below.

1. Formulation of problem and objectives.
2. Clearly define the data to be collected.
3. Collect system parameter data and validation data.
4. Building the simulation model.
5. Verification of the simulation model.
6. Validation of the simulation model.

Some of the steps in the above list required several iterations to complete.

3.2.1. *Formulation of problem and objectives*

The objective of this study was to create a simulation model of adequate detail so that stochastic optimization techniques could be tested for their effectiveness in setting the decision variables at optimum levels. As mentioned earlier, a high level of detail is used so the simulation is as representative of the actual system as possible. This assembly system has many places where stochastic elements can directly affect performance. This makes the system an ideal candidate for the application of stochastic optimization methods. It was also at this stage of the study that Ford Refrigeration and Electronics provided the authorization to study the system and freely collect any necessary data.

3.2.2. *Clearly define the data to be collected*

The system had to be studied carefully to completely understand all the data that needed to be collected. A clear picture of the starting and stopping points of various data being collected was essential. The length of each operation is defined by the time the pallet stops at the station until the time the pallet is released. The necessary data included core building times, vision prealign times, vision system times (both good and rejected cores), banding times, core unload times, and conveyor speed.

3.2.3. *Collect system parameter data and validation data*

The data for the system were collected by direct observation using a stopwatch. All data were collected and analyzed in seconds. The number of observations for each specific parameter varies due to some parameters occurring infrequently. The ultimate goal of data collection was to determine the distributions for each of the system parameters. The advantages of using distribution functions for operation times, rather than empirical data is twofold: values for the data are not restricted to only those of the observations and there are no minimum or maximum values as in empirical distributions (Law and Kelton 1991). Having collected large enough samples from which to draw conclusions, Kolmogorov-Smirnov tests were then applied to each data set. The tests were conducted at an α level of 20% (Mullin 1990). The resulting distributions are shown in Table 3.1.

As illustrated in Table 3.1, there are five different stochastic variables within the simulation model and several are found in more than one place. These stochastic elements along with starving and blocking effects support the initial decision to model the system using simulation.

Table 3.1 System time parameter distributions and values (all times in seconds)

Parameter Description	Distribution	Distribution Parameters	Shift to be Added	Mean Value
New Core Build	Erlang	k = 3, β = 6.013	49.77	67.81
Rejected Core Fix	Exponential	β = 50.461	18.88	50.46
Vision Prealign	Erlang	k = 3, β = 0.111	3.31	3.64
Core Banding	Constant	μ = 5.94, s = 0.061	--	5.94
Vision System (accepted core)	Erlang	k = 8, β = 0.072	5.26	5.84
Vision System (rejected core)	Uniform	a = 8.21, b = 9.00	--	8.61
Accepted Core Unload	Constant	μ = 4.98, s = 0.090	--	4.98

The data needed for validation were not collected at the same time as parameter data. This was not a wise choice, but was inescapable due to time and distance constraints. The actual data used in the validation of the simulation model will be presented in a later section.

3.2.4. *Building the simulation model*

The system is modeled using a terminating, discrete-event simulation. There are definite starting and stopping points in time that define the system. These points are events such as the start of the shift, the beginning of break periods, the ending of break periods, and the ending of the shift.

The simulation of the system is divided into five general parts: the feed track section, the build station section, the return track section, the unload loop section, and the daily schedule section. When an entity (pallet) is in the feed track section, it will enter one of the build stations. The exact station a given entity will enter is completely dependent on the specific situation. Once an entity enters a specific build station section, it is delayed to represent the transport time. Eventually the entity will be delayed by

either a new build time or a rejected core fix time. The next stage of the simulation is the return track section. This section handles the motion of the entities from the exit of the build station section to the beginning of the unload loop section. The return track section is analogous to an interstate with several on-ramps where the build stations represent these on-ramps. Entities travel out of the return track section into the unload loop section. In the unload loop, entities experience a series of delays. Entities are delayed for transport time, vision prealign, vision inspection, banding, and unloading. After finishing with the unload loop section, entities reenter the feed track section.

The daily schedule section of the simulation is responsible for causing events to occur at specific times during the shift. The entire simulation is based on seconds; therefore, the simulated shift is defined by events occurring at some number of seconds from time zero. Time zero is referred to as the start of the shift. The schedule of events that the daily schedule section handles, as referenced from time zero, is given in Table 3.2.

The simulation model uses basic SIMAN structures to model the evaporator assembly system. As mentioned previously, round pallet stops provide holding functions

Table 3.2 Time of day event schedule (all times in seconds)

Event Description	Beginning Time	Ending Time
Initialization	0	1,000
Working	1,000	6,400
Break time	6,400	7,180
Working	7,180	13,600
Break time	13,600	14,380
Working	14,380	18,700
Lunch time	18,700	20,800
Working	20,800	26,200
Break time	26,200	26,980
Working	26,980	29,500
Initialization for next shift	29,500	31,500

to establish the zoning effect. These round pallet stops are modeled using Queue blocks. Each build station, the vision prealign, the vision inspection system, the bander, and the core unload are all represented in the simulation model as resources. To simulate a pallet entering the station, an entity must seize the specific resource. Delay statements are used to represent travel times and process times. Branch statements perform the transfer of control from section to section. The lifts are treated as variables. The variable for a specific lift must be equal to zero in order for an entity to access it. Scan statements establish the control logic. In order to induce the occurrence of an event at a particular time, a dummy entity is created, used to set some variables, and then disposed. This is the process that takes place to indicate to the simulated system that the workers are on break. As shown, the simulation is constructed of basic SIMAN blocks and elements.

The simulation model was built using a direct approach. This meant trying create a one-to-one correspondence between the actual system and the simulation model. This approach seemed to be the most effective and efficient method at the outset of the research; however, it proved to be quite inefficient in terms of computer run time. There are a number of Scan statements used in the simulation. Scan statements combined with the large size of the model, causes the run time to be longer than desired. In the worst case, one replication (the simulation of one shift) can take about 4.5 minutes on a 486DX33 class microcomputer.

3.2.5. *Verification of the simulation model*

The process of verifying the model was much easier, since the simulation was developed in sections. By developing the simulation in sections, entity flow could be easily followed. The key to proper model verification is to have a complete understanding of the system being simulated. The evaporator assembly system being

simulated in this research is governed by an extensive array of ladder logic. Understanding the resultant series of events when an entity encounters a busy or a vacant intersection is absolutely necessary. Verification of the simulation model also requires that the programmer become quite familiar with the simulation language.

3.2.6. *Validation of the simulation model*

In the ideal case, the data used for validation of the simulation model are collected simultaneously with the system parameter data. As mentioned before, this was not possible. Therefore, the validation data were collected about 10 months after that of system parameters. To compare the simulation of the evaporator assembly system with the actual system, several different statistics were utilized. The statistics used in the verification data by no means represent all possibilities; these statistics were perceived as being important to this particular system. Table 3.3 summarizes the results of model validation. The entries in Table 3.3 describing the system configurations are interpreted as (st6,st5,st4,st3,st2,st1,np) where st variables refer to the status of the respective build stations and np stands for the number of pallets. A st variable of 1 means that a worker is assembling evaporator cores at that station, and a st value of 0 implies to no worker is present.

Note in the Table 3.3 that production rate (on a shift basis) is not used as a validation statistic. The workers building evaporator cores have standards to meet. Once a given worker meets his or her standard, they will stop building evaporator cores. This feature was not included in the simulation. The simulation portrays the workers building cores until the end of the shift.

As shown in Table 3.3, the mean value from the simulation is contained within the 95% confidence interval for six out the nine statistics. However, items 5, 6, and 8

Table 3.3 Model validation data (all times in seconds)

Statistic	System Config.	Simulation Mean	95% CI for Actual Data
Pallet cycle time	(1,1,1,1,0,0,22)	16.94	(14.01,17.13)
Pallet cycle time	(0,1,1,1,0,0,19)	22.79	(18.63,23.89)
Pallet cycle time	(0,1,1,1,0,0,19)	22.79	(19.42,23.72)
Pallet cycle time	for station 3 only	300.38	(175.48,494.08)
Pallet cycle time	for station 4 only	334.94	(254.45,311.75)
Pallet cycle time	for station 5 only	376.56	(274.04,320.38)
Pallet cycle time	for station 6 only	526.16	(372.96,557.50)
Pallet travel time	in unload loop	85.28	(83.29,83.77)
Pallet cycle time (for specific pallets)	(1,1,1,1,0,0,22)	372.66	(366.2,404.74)

show that the simulated mean is not contained within the 95% confidence interval of the actual validation data. There are several explanations for this problem. One possible explanation could be that the validation data were collected 10 months after the system parameter data. During validation data collection time, no major changes were observed. However, to be sure operation times did not change over the 10 month gap, the system parameters would have to be retimed. This is the exact reason why validation data should be collected at the same time as system parameter data. Another possible explanation for the differences is the fact that the workers during the two data collection times were completely different people. The average evaporator build time could have changed because of this.

Due to time constraints, the validation data were collected in a small period of time. This could provide another possible explanation for the difference between statistics. Since the time period in which the validation data were collected was very

small, it is quite possible that the validation data is not representative of what one would observe over the course of an entire shift. One last possible reason for differences could be attributed to problems in the simulation model; however, the model was checked thoroughly and was operating as intended.

Table 3.3. also indicates a tendency for the mean simulation data to be larger than the mean validation data. This tendency is probably explained by the previously mentioned 10 month data collection gap. It could also be explained by the possibility of the actual system having faster conveyor speeds than the simulation. In any case, the validation data shows that the simulation model is representative enough to make analysis and optimization meaningful.

3.3. Jam and Reject Rates

Station jams were experienced during the data collection phase of the simulation study. Jams occurred at both the build stations and the automatic stations. These jam rates were not included in the simulation because they were found to be very low. Also, since the system has actual people tending the machines, the jam time was short and insignificant. If a jam occurs at a build station, the resident worker almost immediately rectifies the problem. If a jam occurs at an automatic station, there is also a worker assigned to take care of it. Jams at automatic stations do not cause any significant blocking or starving effects if they are corrected in a timely manner.

The reject rate of the vision system was observed during both the primary and the validation data collection periods. After collection of some data and discussions with the process engineers, the reject rate was set at 1.5%. The reject rate is an important factor to keep in mind due to the effect it can have on starvation; however, since the reject rate was somewhat controllable, this factor was not used as a decision variable.

3.4. Simulation Output

In order to measure the performance of different combinations of decision variables, it is necessary to define the quantity to be obtained from the simulation model; this quantity is the number of good evaporator cores assembled in a given shift. Along with this value, it is also necessary to keep record of the associated combination of decision variables. These decision variables include the station configuration and the number of pallets. A single number between 1 and 63, inclusive, is used to represent the station configuration. The set of build stations is viewed as a six element array, 0 meaning no worker is present and 1 meaning the station is occupied by a worker. The single number representing the station configuration is simply the decimal equivalent of the binary number created by the six element array. The simulation outputs the following three pieces of information to a file: the quantity of good evaporator cores produced, the station configuration, and the number of pallets.

Up to this point, the issue of performance measures has been avoided. This research looks at two different measures for relative performance evaluation. Complete discussions of these performance measures are given in Chapter 4.

4. DESCRIPTION OF ALGORITHMS

This chapter contains descriptions of genetic algorithms and stochastic quasigradient methods. General descriptions of both algorithms are given followed by explanations of each of the modified versions tested in this research. Related issues including performance measures and penalty function are also discussed.

4.1. The Genetic Algorithm

As mentioned previously, genetic algorithms are patterned after the process of natural selection. Natural selection is a process that ocurrs in natural systems by which the fittest individuals dominate in the mating pools. This tends to perpetuate characteristics of the more fit individuals. This chapter descibes the basic operations that make up genetic algorithms and how a simple genetic algorithm was used to optimize the decision parameters of the simulated evaporator core build system in chapter 3. The genetic algorithm described in this research is a modified version of that presented in Goldberg's (1989) text.

4.1.1. *A general description of genetic algorithms*

In order to describe how the genetic algorithm works, it is first necessary to present the basic components. The genetic algorithm operates on a population. This population is composed of a number of individuals. These individuals are simply bit arrays or bit strings (arrays or strings of 0s and 1s). The decision variables of the system or function being optimized are coded into these bit strings. Each bit string represents a certain set of values for the decision variables. These sets of decision variables each have some associated performance measure. These performance measures are used to evaluate the fitness of the associated bit string.

The genetic algorithm described in this research uses three basic operators: reproduction, crossover, and mutation. These operators process the population of individuals from generation to generation. The reproduction operator creates a mating pool of individuals by copying existing strings according their fitness value. Each string has a fitness value and each generation has a total fitness or sumfitness value. The fitness value of an individual string divided by the sumfitness gives the probability of that string being part of the mating pool for the next generation (for maximization problems). A similar procedure is used to calculate the probability of a string being part of the mating pool for minimization problems. It is this weighting scheme that allows the more fit individuals to be dominant contributors to the mating pool. Once the mating pool has been created, the crossover operator is applied. There are two steps to the crossover operation. First, random pairs of individuals in the mating pool are mated. This mating involves the possibility of crossover and/or mutation. Crossover and mutation occur with some specified probability. Second, if crossover is slated to occur, a crossover site is randomly chosen. For example, suppose we have two strings being mated and crossover is to occur between bits 4 and 5.

string 1: 0100|101011 $\Rightarrow$ 0100101100

Crossover

string 2: 0111|101100 $\Rightarrow$ 0111101011

This clearly shows how the respective strings retain their original identity up to the point of the crossover site and then they exchange the remaining portions of their strings with each other. If crossover is not to occur, the parents are copied into the next generation. The mutation operator is applied to each bit of each string. Mutation is simply the toggling of a bit according to the mutation probability. Through these three operators, generations of individuals are processed. This idea is summarized in Figure 4.1.

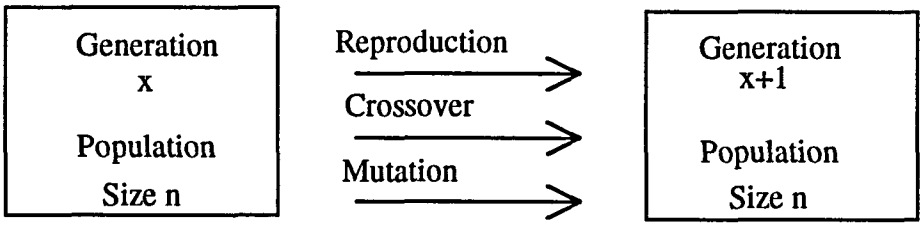

Figure 4.1 The operation of the genetic algorithm (Goldberg 1989)

The final generation should contain more fit individuals on average than the initial generation. The formal mathematical foundations for the genetic algorithm are presented by Holland (1975). Goldberg (1989) also presents these foundations, but in a more example oriented manner.

4.1.2. *The genetic algorithm in this research*

The genetic algorithm implemented for the optimization of the evaporator assembly system uses the three basic operators discussed previously. The main differences between the genetic algorithm used in this research and the simple genetic algorithm presented by Goldberg (1989) is the handling of the individual structure and the determination of the performance measure. The algorithms used in this research were written in C language and Goldberg used PASCAL to implement his simple genetic algorithm. This made it necessary to change the specific structuring of the code. Through this restructuring, Goldberg's simple genetic algorithm was transformed to specifically accommodate the system being studied in this research.

The decision variables used in this research encode into the genetic algorithm very well. These decision variables are the number of workers and the number of pallets. The number of workers must be identified by both quantity and distribution. Knowledge of only the number of workers is insufficient; the placement of those workers is also

important. Recall that the evaporator assembly system contains six build stations. Each build station either contains zero or one worker. Describing the number of workers and their respective placements naturally encodes into bit string form. The number of pallets are also easily encoded into a bit string form. The decimal value of the number of pallets is simply converted to its binary equivalent. The maximum number of pallets in the system varies according to the placement and number of workers. The overall maximum value is 31 pallets; this can be encoded by a bit string of of size 5. Throughout this research, the number and placement of workers will be referred to as the station configuration. An individual in this research is defined by a specific station configuration number and a specific number of pallets. Since the individuals must be in the form of a bit string, the separate bit strings formed from the encoded decision variables are concatenated. For example, suppose there are four workers assembling evaporators: one at station 6, one at station 5, one at station 3, and one at station 1. Also suppose there are 17 pallets. The individual would appear as follows:

Encoded Station Configuration: 110101
Encoded Number of Pallets: 10001
Associated Individual: 11010110001

The algorithm also keeps track of the performance measure for each individual. Discussion of the performance measures used to evaluate individuals is presented in a later section.

In order to apply the genetic algorithm, some initial population of individuals is required. There are different ways of obtaining this group, some authors suggest randomly creating the initial generation, while others recommend that some insight be used. The choice of this research is to select an initial population (generation 0) randomly, to purposely deny any special advantage to the genetic algorithm.

From the description in chapter 3, it can be observed that this assembly system has a finite set of decision variables. The nature of the decision variable encoding process can cause problems in terms of remaining inside of the feasible region. As the genetic algorithm operates it is probable that some individual will be introduced into the population that is not in the feasible decision variable space. This problem is alleviated by using a penalty function. A later section in this chapter is dedicated specifically to the discussion of the penalty functions used.

4.2. Stochastic Quasigradient Methods (SQG)

SQG methods use statistical estimates of the gradient of a function to determine which direction to step in the solution space. Using an SQG method involves objective function formulation, choice of step direction, choice of step size, a projection operation, and some stopping criteria.

4.2.1. *Objective function formulation*

The objective function should accurately reflect the performance of the system being studied. Since performance measures are discussed in a later section, we will simply refer to this quantity as PM for purposes of this description. The objective function for the optimization of the evaporator assembly decision variables can be shown as follows :

$$\min : \mathrm{PM} = F(x) \; ; \; x \text{ in } X$$
$$F(x) = \mathrm{E}_{\omega} f(x, \omega)$$

The variable x above represents a vector of decision variables constrained within set X. The ω denotes a random variable belonging to some space. This randomness can enter the problem through blocking effects, starving effects, evaporator build times, and vision

inspection times. For a given set of decision variables, simulation can provide the expected value of *F(x)*. Based on this expected value and a step direction, the value of x can be forced towards an optimal solution.

4.2.2. *The operation of SQG*

The stochastic quasigradient algorithm progresses from one feasible point to another using the following algorithm :

$$x_{s+1} = \pi_s(x_s - \rho_s \upsilon_s)$$

where x_s is the current approximation to the optimal solution, ρ_s is a step size, and υ_s is the current step direction. The π_s symbol simply represents a projection operator. This insures when the algorithm steps, it does not step out of bounds; it keeps the solution in the feasible region. The general procedure for applying this projection operation is to project the solution back into the feasible region when an out of bounds condition is encountered. The closest feasible solution is usually chosen. The final variable in the SQG algorithm equation, x_{s+1}, is the next approximation of the optimal solution. To apply this algorithm we need to choose a step size, a step direction, and a stopping criteria.

There are three approaches in the choice of step size: fixed, variable, and modified. The fixed step size method should be applied with caution. If the initial solution is not very close to the optimal solution, a fixed step size can restrict the SQG algorithm from finding the global optimal. In many cases a sub-optimal or local minima will be chosen instead.

The variable step size method adjusts the step size according to changes in solution estimates over some period of time. The dynamic behavior of this method tends

to make step sizes large when the current solution is expected to be distant from the optimal solution, and small when the optimal solution is expected to be near.

In the modified step size method, the size of the step may or may not change during a given iteration. The decision to modify the step size is made according to some criteria, such as the moving average of the estimated function value (Liu 1987). If the step size is to be modified, it is usually decreased by some fixed percentage.

We can use a statistical estimate of the gradient function *F(x)* as the step direction. This makes υ_s equivalent to ξ_s such that

$$E(\xi_s | x_1, x_2, \ldots, x_s) = F_x(x_s) + \alpha_s = \upsilon_s.$$

In this equation, ξ_s is a statistical estimate of υ_s and α_s decreases as the number of iterations increases. In this equation, υ_s is the stochastic quasigradient of function *F(x)*. There are several methods available to estimate the gradient direction. Some of the more common methods include finite difference approximations. These approximations come in two varieties: forward finite difference(FFD) and central finite difference(CFD). The FFD can be expressed in the following form:

$$\xi_s = \sum_{i=1}^{n} \frac{f(x_s + \delta_s e_i, \omega_{s,i,1}) - f(x_s, \omega_{s,i,2})}{\delta_s} e_i$$

where *s* is the iteration number, δ_s is the step, $\omega_{s,i,1}$ and $\omega_{s,i,2}$ are stochastic random values generated in iteration *s*, and e_i represents unit basis vectors from R_n. The CFD can be expressed in the following form:

$$\xi_s = \sum_{i=1}^{n} \frac{f(x_s + \delta_s e_i, \omega_{s,i,1}) - f(x_s - \delta_s e_i, \omega_{s,i,2})}{2\delta_s} e_i.$$

By comparing the two finite difference equations, it is observed that the CFD method requires twice as many function evaluations as does the FFD method. Depending on how

the function evaluations are performed, the extra calculations required for the CFD method can significantly increase the computer run time required.

The final issue involving the application of an SQG algorithm is the stopping criteria. Several different schemes for determining the iteration in which to stop the algorithm have been proposed. Methods include stopping when the function value reaches some prescribed goal, stopping the algorithm when the step size has been reduced to a certain value, stopping after a certain number of iterations have occurred, or stopping after a certain period of time.

4.2.3. *SQG algorithms used in this research*

This research uses four different versions of the SQG algorithm to optimize the simulated evaporator assembly system. The binary nature of the station configuration decision variable made encoding difficult. This required the experimentation of several different encoding schemes. The forward finite difference (FFD) method is used in each of the SQG implementations to determine the gradient direction. A variable step size is used for the "number of pallets" decision variable in each heuristic. In these cases, a constant reduction multiplier is applied. The step size for other decision variables is specific to the particular implementation. All the systems implementing a heuristic form of the SQG method are initialized with a randomly chosen set of decision variables. The determination of appropriate step sizes, reduction multiplier percentages, and stopping criteria for each algorithm are presented in Chapter 5.

4.2.3.1. SQG1 SQG1 is the first version of the stochastic quasigradient method. In this heuristic, there are two decision variables: a station configuration and the number of pallets. In each iteration, the decision variables are changed according to the

SQG method. The step in the station configuration context is defined as the toggling of a randomly chosen station bit; if the gradient determined that toggling the chosen bit is beneficial, then it is toggled, otherwise the bit remains the same. The number of pallets is an integer between 1 and max_pallets, inclusive. Max_pallets is the maximum number of pallets allowed by the given station configuration. The step size for the station configuration is constant a one. The step size for the number of pallets is varied according to the modified method.

4.2.3.2. SQG2 SQG2 is the second heuristic implementing the stochastic quasigradient method. SQG2 has two decision variables: number of workers and number of pallets. The number of workers variable has a fixed step size of one. SQG2 looks at adding a worker to the current configuration using FFD. A vacant station from the current configuration is randomly chosen. If the gradient indicates that we must step forward, that bit is added to the station configuration. If the gradient says we must step the opposite direction, a randomly chosen occupied build station is vacated (changed from 1 to 0). The number of pallets variable is handled in the usual manner.

4.2.3.3. SQG3 SQG3 is the third implementation of the stochastic quasigradient method. This version also has two decision variables: station configuration and number of pallets. In this particular variant the station configuration is converted from a 6 digit binary number to the equivalent decimal value. This decimal number is then used as a decision variable. In SQG3, both the decimal equivalent of the station configuration and the number of pallets have modified step sizes. The starting step sizes for the decision variables do not have to be the same; however, the reduction multiplier

percentages are the same. Both variables are handled in the usual manner by the SQG method.

4.2.3.4. SQG4 This version of the SQG method is used as part of the tandem algorithm. This variation uses the SQG method to only determine the number of pallets. SQG4 takes sets of decision variables already determined to be good by a genetic algorithm (GA1) and specifically looks for solution sets with the same station configurations but a different number of pallets. SQG4 is intended to do the fine tuning work for the tandem algorithm. SQG4 also uses a modified step size method for the number of pallets.

4.3. The Tandem Algorithm

The tandem algorithm is a hybrid heuristic employing versions of both the genetic algorithm and the SQG method. The tandem algorithm optimizes a set of decision variables by first applying the genetic algorithm (GA1) for a specified number of generations. Each of the individuals in the last generation are then used as starting configurations for a version of the SQG method (SQG4). SQG4 does not change any of the station configurations found by GA1, but searches for more desirable performance measure values by optimizing the number of pallets decision variable. Since the problem formulation in this research calls for minimization, the idea behind the tandem heuristic is to let the genetic algorithm locate the big valleys in the response surface and then allow the SQG method to find the bottoms of these valleys. In this research, the genetic algortihm component (GA1) outputs its results into a file where the SQG method portion accesses the data.

4.4. Production Data from Simulation

A simulation of the evaporator assembly system was used to obtain the production data needed for the optimization algorithms. The simulation outputs the number of accepted evaporator cores produced in a given shift. As mentioned in chapter 3, the simulation of this assembly system takes quite a while to run due to the size of the model and the number of Scan statements used. The initial intention in this research was to tie all the optimization programs directly to the simulation model to obtain the production information needed to calculate the performance measure. Given all the different algorithms and replications tested in this research, it would have have taken months of computer time to obtain the results. To overcome this problem, the simulation program was run for 5 replications of every possible combination of station configurations and number of pallets. There are 1551 different combinations. This task took about 180 hours of 486DX33 computer run time. The results from all these combinations were placed into a single data file. This data file contains 1551 lines, one for each possible combination. Each line contains five "good cores produced" values, a station configuration number, and the number of pallets. This complete set of data was collapsed into two different files. SIMAVG.DAT contains 1551 lines of information, but only 3 values per line: the average of the 5 replicates, the station configuration number, and the number of pallets. The other collapsed data file, SIMNORM.DAT, contains 1551 lines with 4 pieces of information on each line: the average value of the 5 replicates, the standard deviation of the 5 replicates, the station configuration, and the number of pallets. The file SIMNORM.DAT is used in place of actually linking the simulation program with the optimization programs. When an optimization algorithm needs a production value for a specific station configuration, it simply accesses SIMNORM.DAT and obtains the associated average production value and the standard

deviation. The optimization program then generates a normal random variate using this mean and standard deviation.

This method of using data from a file as opposed to running the simulation model each time production data is needed, allowed for expedient testing of numerous algorithms and multiple replications. The standard deviations for the production value, within a given station configuration and a given number of pallets, were relatively small. Only 5 replications of production values were collected for each configuration; however, they tended to be consistent.

4.5. Performance Measures

This research compares several different heuristics intended to optimize the decision variables of the evaporator assembly system. To make these comparisons, measures of performance were defined. Two different measures of performance were investigated for possible comparison use. These two measures will be referred to as performance measure 1 (pm1) and performance measure 2 (pm2). Pm1 is defined by the following:

$$pm1 = \frac{(op\_rate)(hrs/shift)(number\ of\ builders)}{production\ value}$$

where *op_rate* is the average hourly wage of the operator, *hrs/shift* is fixed at 8, the *number of builders* is as described, and the *production value* is the number of good cores produced in the shift. Pm1 values were viewed for the 1551 line data set. The response surface produced was rather flat and uninteresting. To make the response surface more interesting, pm2 was created. To determine pm2, we must first define some other quantities. Pm2 is based on a required production rate, referred to as *req*. This was set at 1368 evaporator cores. This is about four times the standard for one builder. This

required value can be easily adjusted according to production needs. We use *req* to define the requirement ratio *rr* as follows:

$$rr = \frac{production\ value}{req}.$$

We must also define a penalty cost to production values above and below *req*. Underproduction and overproduction are penalized at different rates. The penalty is defined as follows:

$$penalty = \begin{cases} if\ rr \leq 1 & penalty = (upc1(req - prod)) / prod \\ else & penalty = (upc2(prod - req)) / prod \end{cases}$$

where *upc1* is the unit penalty cost for underproduction and *upc2* is the unit cost for overproduction. Given these quantities, pm2 is defined as the following:

$$pm2 = pm1 + penalty.$$

The two parameters, upc1 and upc2, can be adjusted to scale the terrain of the response surface. Penalizing for underproduction and overproduction makes sense from a manufacturing standpoint. Underproduction can impede the assembly of automobiles while overproduction must occupy valuable floor space. Since overproduction must be stored as work-in-process, it runs the chance of being damaged during the extra handling involved.

As mentioned before, the response surface created by using pm1 is basically flat. There are too many vastly different solutions that have similar performance measures. This would make it very difficult to judge the performance of the optimzation heuristics; therefore, only pm2 was used for comparison purposes.

4.6. Penalty Functions

In the discussion of genetic algorithms and SQG methods, the notion of some solution sets being out of bounds was mentioned. This condition can occur when there are zero workers or too many pallets. These conditions are very undesirable. To insure that these infeasible solutions do not remain in the solution set, their respective performance measures are purposely inflated. The penalty function causes the performance measure value of the infeasible solution to be 100 to 1000 times larger than average feasible pm values. This drives these infeasible solutions out of the solution set.

5. RESULTS FROM ALGORITHM COMPARISONS

This chapter presents the results from the comparison of the five heuristics described in Chapter 4. Before comparing, it was necessary to choose proper operating parameter settings for each of the algorithms. The determination of the best parameter settings was accomplished using design of experiments techniques. The concept of replication arises throughout the this section; five replications simply means that the algorithm was run five times with a different random number seed for each run.

5.1. Setting Algorithm Parameters

Each of the algorithms have a specific set of operating parameters that need to be determined prior to algorithm comparisons. The parameters are set using factorial and single factor designed experiments. Throughout all the experiments, an alpha (α) level of .05 is used. The SIMAVG.DAT was the data file used to set all the operating parameters. The reason behind this is that the response surface for this data file is more stable than that of SIMNORM.DAT; however, the general shape of the two response surfaces should be the same. The performance measure pm2 is used for setting all operating parameters. The underproduction and overproduction unit costs were 1.0 and 0.25, respectively.

Each of the SQG based heuristics require several operating parameters. One of those parameters is the stopping criteria. In this research, the stopping criteria will be some fixed number of iterations. Therefore, every SQG based algorithm will need to have a designated stopping iteration.

5.1.1. *Determining the GA's operating parameters*

To run the genetic algorithm we must determine four different operating parameters: crossover probability, mutation probability, population size, and run length. Some previous research indicated values for the first three of these parameters. Goldberg (1989) provided results from a function optimization problem which suggested that the crossover probability be set relatively high, the mutation probability be set low (inversely proportional to the population size), and use a moderate population size. The actual settings proposed by Goldberg were as follows:

crossover probability : 0.6
mutation probability : 0.02
population size : 30

Wellman (1991) tested several different combinations of these three parameters. Recall that Wellman used a genetic algorithm to optimize the decision variables of an asynchronous automatic assembly system. Wellman recommended the following set of operating parameters:

crossover probability : 0.6 *or* 0.8
mutation probability : 0.001 *or* 0.005
population size : 30

Since both authors mentioned had different systems they were optimizing, it was necessary to test different GA parameters for use with the simulated evaporator assembly system. Two separate experiments were performed. The first experiment was a 3^3 factorial experiment and was used to test the crossover probability, the mutation probability, and the population size. This first test implemented a run length of 10 generations. The results are given in Table 5.1. Table 5.1 shows that there is only one significant factor, and that is the population size. With this result, a single factor

Table 5.1 GA parameter factorial experiment results

Parameter	Levels Tested	F_0	$F_{crit}(\alpha=.05)$
(A) Crossover Prob.	0.4, 0.6, 0.8	0.23	3.35
(B) Mutation Prob.	0.01, 0.02, 0.03	1.38	3.35
(C) Population Size	30, 50, 70	40.76	3.35
AxB interaction	-	0.78	2.73
AxC interaction	-	0.14	2.73
BxC interaction	-	1.38	2.73
ABC interaction	-	0.73	2.31

Experimental Conditions:

1. Two replicates for each combination.
2. Response = sum of top 20 pm2's in generation 10.

experiment was performed to explicitly test different population sizes. The crossover probability and the mutation probability were set at 0.6 and 0.02, respectively. Table 5.2 provides the ANOVA results for the population size experiment.

The results in Table 5.2 show that the population size has a significant effect on the response in the experiment. To select a specific population size, we must first compare several different sizes. Figure 5.1 provides the population size averages for different levels. From this figure it can be seen that the average response ranges from about 7.41 to 6.96. There appears to be an approximate inverse relationship between the population size and the response value. Also, the chart shows a definite drop in average response up to a population size of 50, and then it tends to level off. The appropriate method of choosing a population size is to compare treatment levels using a test such as a Scheffe' test, a Least Significant Difference (LSD) test, or a Bonferroni test. The LSD method is too risky because of the possibility of making a Type I error. There are

Table 5.2 ANOVA summary from population size experiment

Source of Variation	Levels Tested	F_0	$F_{crit}(\alpha=.05)$
Population size	30, 40, 50, 60, 70, 80, 90, 100	7.27	2.31

Experimental Conditions:

1. Five replicates for each combination.
2. Response = sum of top 20 pm2's in generation 10.

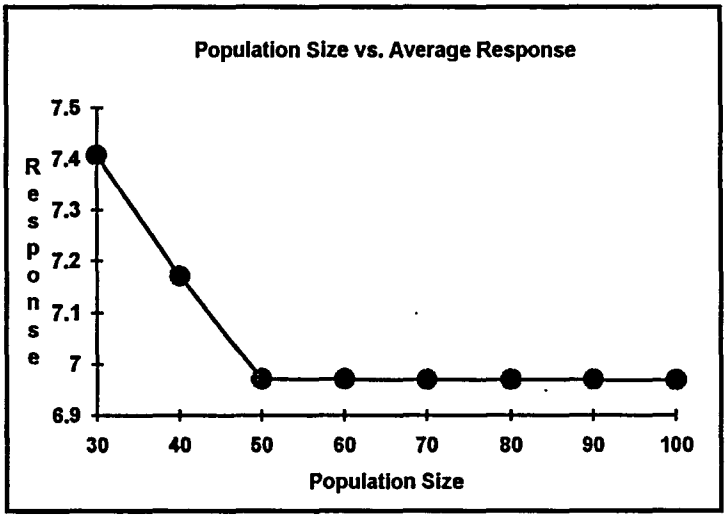

Figure 5.1 Population size vs. average response

k(k-1) / 2 different pair-wise comparisons, where k is the number of treatments. Figure 5.1 contains 8 different treatments which translates to 28 existing comparisons. If an α level of .05 is used for each of the 28 comparisons, it is almost guaranteed that a Type I error will occur. Both the Scheffe′ and Bonferroni tests are designed to overcome this problem. We will use the Bonferroni test whenever treatment levels are being compared. The Bonferroni test is essentially an LSD test with an α level adjusted for the number of comparisons.

Using a Bonferroni test, the 8 different population sizes were compared. Figure 5.2 presents the significance groupings for these population sizes. The reader is reminded that the objective of this research is to minimize the performance measure. Figure 5.2 clearly illustrates that population sizes of 30 are significantly different than sizes of 50, 60, 70, 80, 90, and 100. A population of 40 is not significantly different than any of the other sizes. With the given objective function being a minimization problem, population sizes of 50, 60, 70, 80, 90, and 100 provide the best performance of those tested.

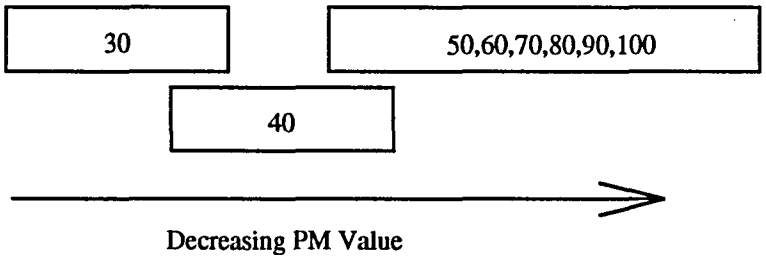

Figure 5.2 Significance groupings for population sizes

The final operating parameter for the genetic algorithm is the number of generations. This parameter was determined by using a single factor experiment testing the following levels: 5, 10, 15, 20, and 25 generations. The results from ANOVA are provided in Table 5.3.

Table 5.3 shows that the levels of the number of generations tested were not significantly different. To get a better picture of how the number of generations run using the genetic algorithm affects the average value of the 20 best performance measures, a plot was created. Figure 5.3 is a plot of the average performance measure versus the number of generations run. From this figure, we see that average response is rather high for the first three generations, but levels off after that. This graph shows why

Table 5.3 ANOVA summary from run length experiment

Source of Variation	Levels Tested	F_0	$F_{crit}(\alpha=.05)$
Number of generations	5, 10, 15, 20, 25	0.93	2.87

Experimental Conditions:

1. Five replicates for each combination.
2. Response = average of top 20 pm2's.

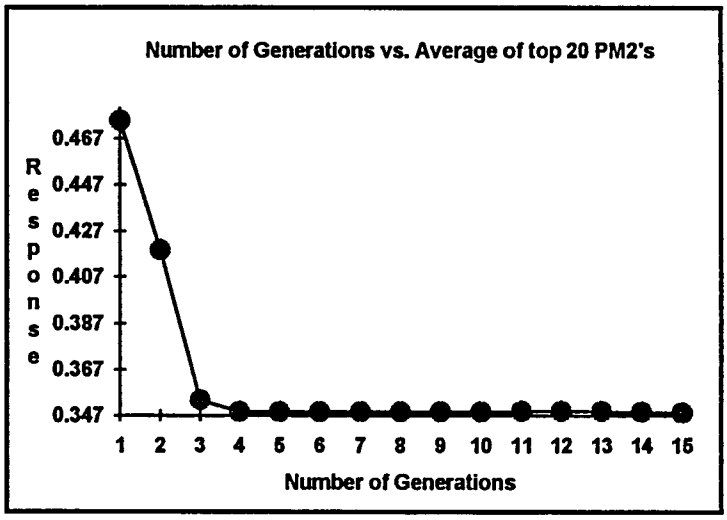

Figure 5.3 Number of generations vs. average response

the ANOVA results in Table 5.3 indicated that the number of generations did not have a significant effect on the response. Since the number of generations of 10 is well into the level response area, but not so large that it takes an unreasonable amount of time to run in our experiments, it was chosen as the parameter setting.

5.1.2. *Determining SQG1's operating parameters*

There are three different operating parameters that need to be set to run SQG1: reduction multiplier percentage, initial pallet step size, and stopping iteration. There

were not any clear recommendations from previous work to assist in choosing these parameters. To check different levels of the operating parameters a 3^3 factorial experiment was used. The results from this experiment are given in Table 5.4. This experiment indicated that the stopping iteration is the only significant factor. To

Table 5.4 SQG1 operating parameter experiment results

Parameter	Levels Tested	F_0	$F_{crit}(\alpha=.05)$
(A) Reduction %	75%, 85%, 95%	2.79	3.35
(B) Init. Pallet Step Size	1, 8, 15	2.20	3.35
(C) Stopping Iteration	10, 20, 30	22.73	3.35
AxB interaction	-	1.01	2.73
AxC interaction	-	1.57	2.73
BxC interaction	-	0.90	2.73
ABC interaction	-	0.59	2.31

Experimental Conditions:

1. Two replicates for each combination.
2. Response = optimal value in the last iteration.

determine which stopping iteration size to use, we must perform an additional experiment focusing only on this parameter. To test various levels of the stopping iteration parameter, it is necessary to fix the reduction multiplier percentage and the initial pallet step size. The reduction multiplier percentage and the initial pallet step size were set at 95% and 8, respectively. Both of these levels were chosen because they had the lowest totals for their respective variables. The results from this single factor experiment are displayed in Table 5.5. The results prove that the stopping iteration has a significant effect on the response. The next step was to see which levels of the stopping

iteration provide the most desirable effect. This can be accomplished by using a Bonferroni test. The results of the Bonferroni test performed on the stopping iteration variable are pictured in Figure 5.4. Figure 5.4 indicates that the there are two significant groups of stopping iterations: 5 and the others. This figure shows that choosing a stopping iteration of 10, 15, 20, 25, or 30 is better than choosing 5.

Table 5.5 ANOVA summary for stopping iteration experiment

Source of Variation	Levels Tested	F_0	$F_{crit}(\alpha=.05)$
Stopping iteration	5, 10, 15, 20, 25, 30	5.89	2.62

Experimental Conditions:

1. Five replicates for each combination.
2. Response = optimal value in the last iteration.

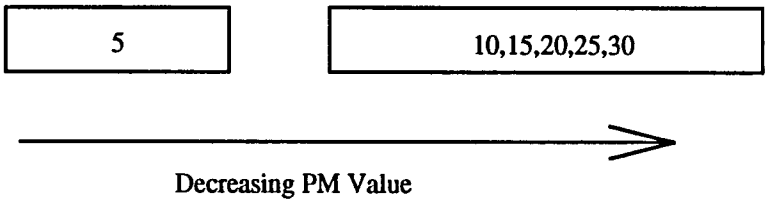

Figure 5.4 Significance groupings for stopping iterations

5.1.3. *Determining SQG2's operating parameters*

As in the case of SQG1, there are also three operating parameters that need to be set before SQG2 can be applied to the optimization of the simulated evaporator assembly system. These factors are reduction multiplier percentage, initial pallet step size, and number of iterations. As before, a 3^3 factorial experiment was used to determine how to set these three operating parameters. The results of this experiment are provided in Table 5.6. The outcome of this experiment shows that given the levels of the factors tested,

none of the main effects or any interactions have a significant effect on the response. Since no factors were found to be significant, the levels of the factors were set by choosing the ones which supplied the minimum totals in the ANOVA. These settings were 75% for the reduction multiplier percentage, 8 for the initial pallet step size, and 20 for the number of iterations.

Table 5.6 SQG2 operating parameter experiment results

Parameter	Levels Tested	F_0	$F_{crit}(\alpha=.05)$
(A) Reduction %	75%, 85%, 95%	1.59	3.35
(B) Init. Pallet Step Size	1, 8, 15	1.15	3.35
(C) Stopping Iteration	10, 20, 30	2.75	3.35
AxB interaction	-	2.36	2.73
AxC interaction	-	0.64	2.73
BxC interaction	-	0.62	2.73
ABC interaction	-	0.97	2.31

Experimental Conditions:

1. Two replicates for each combination.
2. Response = optimal value in the last iteration.

5.1.4. *Determining SQG3's operating parameters*

The operation of SQG3 differs slightly from the two previous versions discussed. SQG3 has four different operating parameters that need to be set prior to its application to the optimization problem in this research: reduction multiplier percentage, initial pallet step size, initial station configuration step size, and the stopping iteration. To set all four of these operating parameters, a 3^3 factorial experiment was implemented followed by a

single factor experiment. This allowed for the statistical testing of all four parameters. The results from the 3 factor factorial experiment are given in Table 5.7. This table plainly shows that none of the parameters, at the levels tested, had a significant effect on the response. Since no significant effects were detected, these three parameters were set

Table 5.7 SQG3 operating parameter experiment results

Parameter	Levels Tested	F_0	$F_{crit}(\alpha=.05)$
(A) Reduction %	75%, 85%, 95%	0.07	3.35
(B) Init. Pallet Step Size	1, 8, 15	0.10	3.35
(C) Init. SC Step Size	1, 8, 15	0.38	3.35
AxB interaction	-	1.16	2.73
AxC interaction	-	1.56	2.73
BxC interaction	-	0.77	2.73
ABC interaction	-	0.66	2.31

Experimental Conditions:

1. Two replicates for each combination.
2. Response = optimal value in the last iteration.

according to minimum total values from the ANOVA output. The reduction multiplier percentage was set at 95%, the initial pallet step size was set at 8, and the initial station configuration step size was set at 15. The other required operating parameter is the stopping iteration.

As mentioned before, a single factor experiment was used to analyze different levels of the stopping iteration parameter. The findings from this experiment are shown in Table 5.8. This experiment shows that none of the various levels of the stopping

Table 5.8 ANOVA summary for stopping iteration of SQG3 experiment

Source of Variation	Levels Tested	F_0	$F_{crit}(\alpha=.05)$
Stopping iteration	5, 10, 15, 20, 25, 30	1.25	2.62

Experimental Conditions:

1. Five replicates for each combination.
2. Response = optimal value in the last iteration.

iteration had a significant effect on the response, so the level of the parameter was set using the minimum total. A stopping iteration of 25 was chosen.

5.1.5. *Determining SQG4's operating parameters*

As mentioned before, SQG4 is the second half of the tandem algorithm. This heuristic has three operating parameters: the reduction multiplier percentage, the initial pallet step size, and the stopping iteration. To set these parameters a 3^3 factorial experiment was utilized. The results from this experiment are presented in Table 5.9.

The response for the experiment in Table 5.9 is the sum of the best 20 performance measures. The reader is reminded that SQG4 uses a data set found by GA1. This data set contains the configurations of the 20 best individuals in the final generation of a genetic algorithm. SQG4 uses each one of these configurations as a starting point and attempts to optimize the number of pallets. This is the reason the performance measure for this experiment is a sum of the best 20 values. Table 5.9 shows that none of the factors tested provided a significant effect on the response. As in several previous cases, the factor level settings were chosen according the minimum total values found in the ANOVA. The reduction multiplier percentage, the initial pallet step size, and the stopping iteration were established at 85%, 8, and 15, respectively.

Table 5.9 SQG4 operating parameter experiment results

Parameter	Levels Tested	F_0	$F_{crit}(\alpha=.05)$
(A) Reduction %	75%, 85%, 95%	2.24	3.35
(B) Init. Pallet Step Size	1, 8, 15	2.64	3.35
(C) Stopping Iteration	10, 15, 20	0.96	3.35
AxB interaction	-	2.04	2.73
AxC interaction	-	0.60	2.73
BxC interaction	-	0.98	2.73
ABC interaction	-	1.11	2.31

Experimental Conditions:

1. Two replicates for each combination.
2. Response = Sum of the pm2's of top 20 individuals.

5.1.6. *A summary of chosen operating parameters*

The operating parameters for each algorithm tested in this research were established using design of experiments techniques. In some cases 3 factor factorial designs were utilized, and in other situations single factor experiments were employed. Tables 5.10 and 5.11 present summaries of the settings of all the various operating parameters determined in this chapter. The parameters are listed by their associated optimization heuristic.

Table 5.10 A summary of operating parameter settings for GA based algorithms

Algorithm	Crossover Probability	Mutation Probability	Population Size	Stopping Generation
GA	0.60	0.02	60	10
Tandem (GA1)	0.60	0.02	60	10

Table 5.11 A summary of operating parameter settings for SQG based algorithms

Algorithm	Reduction Multiplier %	Initial Pallet Step Size	Stat. Config. Step Size	Stopping Iteration
SQG1	95%	8	-	15
SQG2	75%	8	-	20
SQG3	95%	8	15	25
Tandem (SQG4)	85%	8	-	15

5.2. Comparing the Optimization Algorithms

To compare the five optimization heuristics given in this research, we must first define the entire set of experimental conditions. The operating parameters in Tables 5.10 and 5.11 along with the following define the exact experimental conditions:

1. SIMNORM.DAT was used as the data file.
2. An α level of .05 was used for all testing, except Bonferroni.
3. The performance measure definition pm2 was used.
4. Penalty costs: *upc1* = 2.00, *upc2* = 0.40.
5. Required production was set at 1368 (*req*).

As listed, SIMNORM.DAT was used as the input data file. Recall that this file contains a mean value for "good cores produced" in a shift, the standard deviation for the number of "good cores produced" in a shift, the station configuration number, and the number of pallets. There is a set containing these four pieces of information for every possible combination of station configuration and number of pallets. Each optimization program produces a normal random variate using the mean and standard deviation when a value for "good evaporators produced" is needed.

The optimization heuristics were compared using three different approaches. The first comparison involved a single factor experiment with each of the five algorithms being a treatment level. The response for this experiment was the best performance measure. The second comparison category is the optimal system configurations found by each of the algorithms. The third comparison classifies the efficiencies of the algorithms by their respective computer run times.

5.2.1. *Algorithm comparison by the best performance measure*

To compare the five algorithms, a logical response had to be selected. Remember that the genetic algorithm based heuristics produce several different sets of station configurations and associated number of pallets, whereas the SQG based heuristics produce a single system configuration. So that comparisons would make sense, a response of the best performance measure was chosen. The ANOVA results from this experiment are displayed in Table 5.12.

This table tells us that the different optimization algorithms provide significantly different effects on the performance measure. The mean values of the best performance measures over ten replications for each optimization algorithm are given in Table 5.13. To determine which optimization algorithms provide more desirable response values, a Bonferroni test was applied to each of the treatment means. The value of the Bonferroni test statistic was 0.1936. This implies that any difference in the mean responses between heuristics of greater than 0.1936 is significant. Table 5.13 clearly shows that the mean values of the best performance measures for 10 replications are very close, except in the case of SQG? Since a data file containing the production information was used, it was possible to load this data into a spreadsheet and scan for minimum and maximum values. Having this data set in a spreadsheet, it was simple to calculate the performance measure.

Table 5.12 ANOVA summary for the comparison of the 5 optimization heuristics

Source of Variation	Algorithms Tested	F_0	$F_{crit}(\alpha=.05)$
Optimization heuristic	GA, SQG1, SQG2, SQG3, Tandem	5.67	2.58

Experimental Conditions:

1. Ten replicates for each combination.
2. Response = performance measure of optimal solution.

Table 5.13 Mean values of the best performance measures for 10 replications

Algorithm	Mean Value ($\overline{Y_{i\cdot}}$)
GA	0.351126
SQG1	0.351522
SQG2	0.351639
SQG3	0.582505
Tandem	0.350993

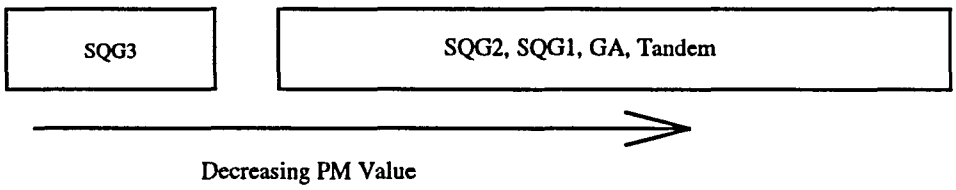

Figure 5.5 Significance groupings of algorithm performance according to the best performance measure

Pm2 values were calculated using average production per shift value. The global minimum for pm2 using the average production per shift value was 0.348123. Four of the five heuristics provided optimal solutions reasonably close to this value. As mentioned before, the Bonferroni test was then applied. The groupings from this test are illustrated in Figure 5.5. These groupings in Figure 5.5 indicate that SQG2, SQG1, GA, and Tandem provide significantly better (lower) performance measure values than SQG3.

5.2.2. *Algorithm comparison by configuration of optimal solution*

Another important way of comparing the performance of the five heuristics proposed in this research, is to look at the station configuration and the number of pallets in the optimal solution. Table 5.14 presents the statistics concerning the station configuration and the number of pallets in the optimal solution. These results reveal some interesting facts about the optimal solutions picked by different heuristics. All the algorithms, except SQG3, provided optimal solutions consisting of four workers. Even SQG3 found four workers to be the optimal in 7 out of the 10 replications. This means that with the chosen required production level set at 1368, the unit cost of underproduction set at 2.0, and the unit cost of overproduction set at 0.4, four workers is the best choice. This outcome was expected when the value of 1368 was chosen for *req*. There are 15 different combinations of four workers possible in this system (6 choose 4). Most of these 15 station configurations were chosen by at least one of the algorithms at one time or another; however, none of the algorithms honed in on any particular station configuration.

Table 5.14 System configuration information from optimal solutions

Algorithm → Replication ↓	GA (sc , np)	SQG1 (sc, np)	SQG2 (sc , np)	SQG3 (sc , np)	Tandem (sc , np)
1	011011 , 16	100111 , 19	111100 , 17	011101 , 13	001111 , 14
2	010111 , 16	110011 , 21	101101 , 23	110101 , 24	110110 , 16
3	101101 , 15	111001 , 23	101110 , 22	110011 , 21	110110 , 15
4	100111 , 19	001111 , 20	010111 , 15	001011 , 20	110011 , 15
5	110110 , 16	001111 , 14	101101 , 17	000111 , 13	101110 , 15
6	101101 , 16	010111 , 15	101101 , 15	110011 , 15	110011 , 15
7	001111 , 19	010111 , 15	001111 , 17	010111 , 13	101110 , 15
8	111100 , 17	111010 , 16	111001 , 17	010111 , 18	011101 , 15
9	110011 , 15	011101 , 16	111010 , 18	110110 , 25	101011 , 15
10	111100 , 16	011101 , 15	110101 , 20	100011 , 10	100111 , 15
# of workers (min,max,avg)	4, 4, 4	4, 4, 4	4, 4, 4	3, 4, 3.7	4, 4, 4
# of Pallets (min,max,avg)	15, 19, 16.5	14, 20, 17.4	15, 23, 18.1	10, 25, 17.2	14, 16, 15

Note: **sc** denotes station configuration and **np** denotes number of pallets

Another interesting aspect of the data in Table 5.14 is the number of pallets given as optimal solutions. As previously mentioned, the Tandem algorithm was designed to first find optimal station configurations and then the optimal number of pallets for each of those configurations. The optimal solution for the Tandem algorithm contained 15 pallets in 8 of the replications and 14 and 16 in the other two. GA and Tandem tended to have more consistent values for the optimal number of pallets than SQG1, SQG2, or SQG3.

5.2.3. *Algorithm comparison by computer run time*

Comparing the five different algorithms using computer run time as a point of interest can be broken down into two categories: run time using SIMAN to directly

obtain the production values and using a data file such as SIMNORM.DAT. As mentioned before, the direct link to SIMAN takes significantly more computer run time than using a data file like SIMNORM.DAT. Each of the heuristic programs have a certain amount of overhead, but a majority of the run time is due to the retrieval of production data. Table 5.15 lists the number of simulation runs or data file look ups required and the approximate average time to obtain one replication. The time measurements were made on a 486DX33 class microcomputer. Table 5.15 points out the tremendous time saved by using the data file SIMNORM.DAT to generate production data rather than directly using SIMAN. This table also shows that the heuristics implementing a genetic algorithm (GA and Tandem) require significantly more production information, and thus more computer run time. In terms of computer run time requirements, SQG1 and SQG2 are much more efficient than either GA1 or Tandem.

Table 5.15 Comparison of algorithms according to computer run time

Algorithm	Number of configs. checked	Approx. hours / repl. using SIMNORM.DAT	Approx. hours / repl. using SIMAN directly
GA	660	0.0480	15.2
SQG1	48	0.0042	1.1
SQG2	84	0.0069	1.9
SQG3	104	0.0078	2.4
Tandem	1300	0.2000	29.9

5.3. General Observations Concerning the Optimization Algorithms

The comparisons provided in this chapter have demonstrated that four of the five proposed heuristics arrive at good solutions for the optimization of the system configuration of the simulated evaporator assembly system. The four best heuristics were GA, SQG1, SQG2, and Tandem. There are some other general considerations when comparing these four acceptable heuristics.

SQG1 exhibited very good performance in the categories of performance measure optimization and computer run time; however, this algorithm has a potentially dangerous flaw. When the initial number of workers is far from the optimal value, SQG1 can get hung up on some distant local minima. The required production was set at 1368 in the definition of performance measure 2 (pm2). This value was set anticipating that one or more cases with four workers would be optimal. If the required production had been set at some value where the optimal number of workers would be for example one or six, there is a much greater chance that SQG1 would have problems.

The genetic based algorithms (GA and Tandem) both provided very good results but took inordinate amounts of time to run relative to the SQG based algorithms. One must not overlook the flexibility of the genetic algorithms. The genetic based algorithms supply multiple solutions. Approximately 25 to 35% of the solutions contained in the final generation are not significantly different than the declared optima. This adds a degree of flexibility to the optimization of the system. Imagine a situation where a specific build station or some combination of build stations cannot be used for one reason or another, with a genetic based algorithm there are several acceptable alternative solutions available.

Chapter 6 will summarize the findings of this research and offer some topics for future work.

6. CONCLUSION

The work presented in this research can be divided into two distinct categories: development of the simulation model and the application of optimization algorithms. A simulation model was developed to represent an existing evaporator assembly system. This assembly system has six parallel manual evaporator build stations and several automatic stations. The system can be officially classified as a palletized semi-automatic asynchronous build line. The evaporator assembly system is subject to effects from several stochastic variables. These stochastic elements make deterministic analysis of the system nearly impossible; therefore, techniques involving stochastic optimization were implemented. Five different optimization heuristics were applied to the simulated assembly system. These heuristics were based on genetic algorithms, stochastic quasigradient methods (SQG), or both. The decision variables used in this optimization problem were station configuration and number of pallets. The measure of performance utilized was essentially a unit cost considering the number of workers assembling evaporators, the required number of evaporators per shift, the amount of underproduction, and the amount of overproduction.

The simulation model was used to obtain production information for the evaporator assembly system. Two methods of utilizing production information were proposed: a direct link with SIMAN to obtain "good cores produced" values and creating a production data file and simply performing sequential accesses to obtain "good cores produced" values. Due to enormous time savings, the "good cores produced" values were assumed normal and a data file containing the mean, standard deviation, station configuration, and number of pallets for all possible system configurations was generated.

This research compared the performance of the five proposed heuristics: GA1, SQG1, SQG2, SQG3, and Tandem. GA1 applies a heuristic form of a genetic algorithm, while SQG1, SQG2, and SQG3 employ a form of a stochastic quasigradient method. The Tandem algorithm exploits both GAs and SQG methods. Before directly comparing the optimization algorithms, factorial experiments were utilized to set each heuristic's operating parameters. The five heuristics were then compared using three different criteria: the best performance measure, station configuration of optimal solution(s), and required computer run time.

The results in this research showed that GA1, SQG1, SQG2, and Tandem were able to find one or more solutions sporting near-optimal performance measures. SQG3 did a poor job of finding a near-optimal solution. In every replicate, GA1, SQG1, SQG2, and Tandem declared a four worker case optimal. This implies that one of the configurations containing four workers is probably the global optima. The genetic based algorithms were more consistent on their choice of the optimal number of pallets than the SQG based heuristics. The last comparison issue was computer run time. SQG1, SQG2, and SQG3 were superior to the genetic algorithms in terms of run time because the GAs required far more production data.

It was stated in previous chapters that the Tandem algorithm was designed to use a genetic algorithm to identify the main "valleys" on the response surface and then apply a SQG method to hone in on the lowest point in that "valley." On the average, the Tandem algorithm provided solutions possessing the smallest performance measure; however, there was not a statistically significant difference between Tandem, GA1, SQG1, and SQG2. Even so, the Tandem algorithm was successful in finding near-optimal solutions for the simulated evaporator assembly system. Another piece of evidence supporting the positive performance of the Tandem algorithm is the consistency

of the number of pallets in the solutions declared optimal. In 8 out of 10 replicates, the number of pallets was 15. In the other two replicates the number of pallets declared optimal was 14 and 16. The ranges of the numbers of pallets chosen as optimal for the other heuristics encompassed 15; however, the other heuristics were not nearly as consistent as the Tandem algorithm. This consistency presents strong evidence that the actual optimal solution contains 15 pallets. The performance of the Tandem algorithm was very favorable in all aspects except relative computer run time. The fact that genetic algorithms require a lot of time to run is no surprise; Wellman (1991) came to this same conclusion.

Up to this point, no direct recommendation has been made as to which of the five heuristics to use. We will now address this issue. The behavior of SQG1 may not be reliable in a larger solution space. Considering their construction, SQG2 should be more robust in a wider range of solution spaces than SQG1. A larger solution space implies an expanded evaporator assembly system, or another entirely different system. The behavior of these algorithms would most likely change in the solution space of a different problem. The two genetic algorithm based heuristics, GA1 and Tandem, provided good performance measure results but were very computationally inefficient. If one were to blindly apply one of the algorithms discussed to some similar but different optimization problem, use Tandem, GA1, or SQG2. If the time required to obtain a performance measure value for a given set of decision variables is small, then application of the Tandem algorithm is recommended. The time to evaluate a set of decision variables is the key factor in deciding whether to use a genetic algorithm based heuristic.

The operation of genetic algorithms and SQG methods is highly dependent on the choice of the operating parameters. Optimal parameter settings will vary according to the solution space. There are two important considerations when using GAs or SQG

methods. First, test various combinations of operating parameters using design of experiments or other appropriate methods. Second, carefully choose the different operating parameter settings to test. This involves obtaining a thorough understanding of the system being studied so knowledgeable choices can be made.

This research has presented two new ideas. The first is the application of optimization heuristics to parallel server assembly systems. The second new idea is the tandem application of a genetic algorithm and a stochastic quasigradient method. Another important aspect of this research is the application of optimization techniques on a "real world" system.

There are several topics for future research that arise from this study. The heuristics presented in this research could be applied to larger parallel server systems. Another possible topic would be to investigate the behavior of other heuristics (e.g. simulated annealing or optimization homotopy) on the optimization of the assembly system presented in this research.

REFERENCES

Andreasen, M. M., and Ahm T. *Flexible Assembly Systems.* IFS Publications / Springer-Verlag, London. 1988.

Bethke, Albert D. "Genetic Algorithms as Function Optimizers." *Logic of Computers Group, Computer and Communication Sciences Department.* Technical Report No. 212. NASA Grant No. NGG-1176. NSF Grant No. MCS76-04297. April 1978.

Bonomi, E. and Lutton, J. "The N-city Travelling Salesman Problem: Statistical Mechanics and the Metropolis Algorithm." *SIAM Review.* Vol. 26, No. 4, Oct. 1984. pp. 551-568.

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

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

Bulgak, A. A., and Sanders, J. L. "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, 1988. San Diego, CA. pp. 684-690.

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

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

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

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

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

Dwiggins, Boyce H. *Automotive Air Conditioning.* 4th edition. Van Nostrand Reinhold Company, New York. 1978.

Ermoliev, Yuri. "Facility Location Problem." *Numerical Techniques for Stochastic Optimization* (Edited by Ermoliev and Wets). Springer-Verlag, New York. 1988. pp. 413-434.

Ermoliev, Yuri M. "On the Method of Generalized Stochastic Gradients and Quasi-Feyer Sequences." *Kibernetika* (English Translation). Vol. 5, No. 2. 1969. pp. 208-220.

Ermoliev, Yuri. "Stochastic Quasigradient Methods and their Application to System Optimization." *Stochastics.* Vol. 9. 1983. pp. 1-36.

Ermoliev, Yuri, and Gaivoronski, Alexei. *Stochastic Quasigradient Methods and their Implementation.* Working Paper, WP-84-55. IIASA, Laxenburg, Austria. 1984.

Gemmill, Douglas D. "Optimization Appraoches to the Portfolio Problem." Doctoral Dissertation. University of Wisconsin, Madison. 1988.

Glover, David E. "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. 1987. pp. 13-31.

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

Glynn, Peter W., and Sanders, Jerry L. *Monte Carlo Optimization of Stochastic Systems: Two New Approaches.* Proceeding of the 1986 ASME Computers in Engineering Conference.

Goldberg, David E. *Genetic Algorithms in Search, Optimization, and Machine Learning.* Addison-Wesley Publishing Company, Reading, Mass. 1989.

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

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

Groover, Mikell P. *Automation, Production Systems, and Computer Integrated Manufacturing*. Prentice-Hall, Inc., Englewood Cliffs, NJ. 1987.

Gross, Donald, and Harris, Carl M. *Fundamentals of Queueing Theory*. 2nd edition. John Wiley & Sons, New York. 1985.

Gupal, A. M., and Norkin, V. I. "Algorithm for the Minimization of Discontinuous Functions." *Kibernetika* (English Translation). Vol. 2. 1977. pp. 73-75.

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

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

Kamath, M., and Sanders, J. L. "Analysis of Asychronous Automatic Assembly Systems with Bottleneck Stations." *Proceedings of the SME Systems 1 Conference*. Chicago, IL. March 24-26, 1986.

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

Kirkpatrick, S., Gelatt, C. D. Jr.,and Vecchi, M. P. "Optimization by Simulated Annealing." *Science*. Vol. 220, No. 4598. May 13, 1983. pp.671-680.

Kushner, Harold J. "Stochastic Approximation Algorithms for Constrained Optimization Problems." *The Annals of Statistics*. Vol. 2, No. 4. 1974. pp. 713-723.

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

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

Lie, Laurensius. "Simulated annealing algorithm applied in the flexible manufacturing systems design." M.S. Thesis. Iowa State University. 1991.

Liu, Chi-ming. "Stochastic design optimization of asynchronous automaic assembly systems." Doctoral Dissertation. University of Wisconsin, Madison. 1987.

Liu, C. M., and Chiou, J. M. " Design and performance evaluation of closed automatic assembly systems." *International Journal of Production Research*. Vol. 28, No. 9. 1990. pp. 1577-1593.

Liu, C. M., and Sanders, J. L. "Approximate design optimization of asynchronous assembly systems." *International Journal of Computer Applications in Technology*. Vol. 2, No. 1. 1989. pp. 30-37.

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

Montgomery, Douglas C. *Design and Analysis of Experiments*. Third edition. John Wiley & Sons, New York. 1991.

Mullin, John P. *Some Practical on Data Analysis for Queueing Systems Analysis*. Working Paper. IMSE Department. Iowa State University. March 29, 1990.

Nevins, James L., and Whitney, Daniel E. *Concurrent Design of Products & Processes : A Strategy for the Next Generation in Manufacturing*. McGraw-Hill Publishing Company, New York. 1989.

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

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

Richardson, J. T., Palmer, M. R., Liepins, G., Hilliard, M. "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.

Schildt, Herbert. *Turbo C : The Complete Reference*. McGraw-Hill Publishing Company, Berkeley, CA. 1988.

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

Tandiono, Elly. "Stochastic optimization of cost of automatic assembly systems." M.S. Thesis. Iowa State University. 1991.

Wellman, Mark A. "A genetic algorithm approach to optimization of asynchronous automatic assembly systems." M.S. Thesis. Iowa State University. 1991.

APPENDIX A

EVAPORATOR ASSEMBLY SYSTEM SIMULATION,

SIMAN MODEL FILE

```
BEGIN;
        CREATE,    1,1000:0,1;    ! Time 0 to 1000 is init time
        BRANCH,    1:
                ALWAYS,pres;
        CREATE,    1,6400:0,1;    ! These create statements simulate
        BRANCH,    1:             ! the working day.
                ALWAYS,npres;
        CREATE,    1,7180:0,1;
        BRANCH,    1:
                ALWAYS,pres;
        CREATE,    1,13600:0,1;
        BRANCH,    1:
                ALWAYS,npres;
        CREATE,    1,14380:0,1;
        BRANCH,    1:
                ALWAYS,pres;
        CREATE,    1,18700:0,1;
        BRANCH,    1:
                ALWAYS,npres;
        CREATE,    1,20800:0,1;
        BRANCH,    1:
                ALWAYS,pres;
        CREATE,    1,26200:0,1;
        BRANCH,    1:
                ALWAYS,npres;
        CREATE,    1,26980:0,1;
        BRANCH,    1:
                ALWAYS,pres;
        CREATE,    1,29500:0,1;   ! builders leave at end of shift
        BRANCH,    1:
                ALWAYS,npres;
        CREATE,    1,31499.9:0,1;
        WRITE,     PM_OUTPUT,"(F6.1)":NC(Gcores):DISPOSE;

npres   ASSIGN:    op1gone = 1;       ! builders on break
        ASSIGN:    op2gone = 1;
        ASSIGN:    op3gone = 1;
        ASSIGN:    op4gone = 1;
        ASSIGN:    op5gone = 1;
        ASSIGN:    op6gone = 1:DISPOSE;

pres    ASSIGN:    op1gone = 0;       ! builders working
        ASSIGN:    op2gone = 0;
        ASSIGN:    op3gone = 0;
        ASSIGN:    op4gone = 0;
        ASSIGN:    op5gone = 0;
        ASSIGN:    op6gone = 0:DISPOSE;

        CREATE,    1,0:1,1;   ! Check the number of the last station
        BRANCH,    1:
                IF,laststa .EQ. 6,l6:
```

```
            IF,laststa .EQ. 5,l5:
            IF,laststa .EQ. 4,l4:
            IF,laststa .EQ. 3,l3:
            IF,laststa .EQ. 2,l2:
            ELSE,l1;
l6      ASSIGN:   laststa = 6:DISPOSE;
l5      ASSIGN:   cs12x12a = 5:DISPOSE;
l4      ASSIGN:   cs11x12 = 5:DISPOSE;
l3      ASSIGN:   cs10x11 = 5:DISPOSE;
l2      ASSIGN:   cs9x10 = 5:DISPOSE;
l1      ASSIGN:   cs8x9 = 5:DISPOSE;

; ******* TOP OF THE MODEL *******

top     ASSIGN:   cs7ax7b = 0;
        QUEUE,    rps7b;
        SCAN:     ((NQ(rps8)+cs7bx8) .LT. 3) .AND. (lift4 .EQ. 0);
        ASSIGN:   cs7bx8 = 1;
        ASSIGN:   lift4 = 1;        ! indicate lift4 is busy
        DELAY:    2.86+0.42+1.54;   ! P to lift4, lift, and clear
        ASSIGN:   lift4 = 0;        ! indicate lift4 is clear
        DELAY:    3.95;             ! lift4 proxy to rps8
        ASSIGN:   cs7bx8 = 0;

        QUEUE,    rps8;
        SCAN:  ((builtby .EQ. 1) .AND. (NQ(sb1) .EQ. 0) .AND.
               (lift5 .EQ. 0) .AND. (lift15 .EQ. 0) .AND.
               (lift18 .EQ. 0)) .OR. ((builtby .GT. 1) .AND.
               ((NQ(rps9)+cs8x9) .LT. 2)) .OR. ((sta1stat .EQ.0)
               .AND. ((NQ(rps9)+cs8x9) .LT. 2)) .OR. ((
               sta1stat .EQ. 2) .AND. (lift5 .EQ. 0) .AND.
               (builtby .EQ. 0) .AND. (lift15 .EQ. 0) .AND.
               (lift18 .EQ. 0) .AND. (NQ(sb1) .EQ. 0)) .OR.
               ((sta1stat .EQ. 1) .AND. ((NQ(rps9)+cs8x9).LT.3)
               .AND. (builtby .EQ. 0));

! This is evaporator core build station 1

        BRANCH,   1:
               IF,builtby .EQ. 1,ToSta1:
               IF,builtby .GT. 1,Pass1:
               IF,sta1stat .EQ. 0,Pass1:
               IF,sta1stat .EQ. 1,Pass1:
               IF,sta1stat .EQ. 2,ToSta1;

ToSta1  ASSIGN:   sta1stat = 1;
        ASSIGN:   lift5 = 1:
        ASSIGN:   lift15 = 1;
        ASSIGN:   lift18 = 1;
        DELAY:    3.19;           ! lift5 to lift15
        ASSIGN:   lift5 = 0;      ! clear lift5
```

```
        DELAY:    1.31;          ! lift15 to lift18
        ASSIGN:   lift15 = 0;     ! clear lift15
        DELAY:    2.04;          ! lift18 to proxy
        ASSIGN:   lift18 = 0;     ! clear lift18
        DELAY:    2.18;          ! proxy to sb1
        QUEUE,    sb1;
        SCAN:   (NQ(sta1buf) .EQ. 0) .AND. (NR(sta1) .EQ. 0) .AND.(NQ(o1ghold) .EQ. 0);
        ASSIGN:   sta1stat = 2;
        QUEUE,    s1dbuf;
        SEIZE:    sta1;
        QUEUE,    o1ghold;
        SCAN:     (op1gone .EQ. 0);
        BRANCH,   1:
                  IF,builtby .EQ. 0,s1new:
                  ELSE,s1rej;
s1new   DELAY:    ERLANG(6.013,3)+49.77;  ! build time for new core
        BRANCH,   1:
                  ALWAYS,ctn1;
s1rej   DELAY:    EXPO(50.4607)+18.88;    ! fix rejected core
ctn1    RELEASE:  sta1;
        ASSIGN:   builtby = 1;
        QUEUE,    sta1buf;
        SCAN: (lift17 .EQ. 0) .AND. (lift16 .EQ. 0) .AND.(NQ(rps1) .EQ. 0);
        COUNT:    Sta1Job;
        ASSIGN:   lift16 = 1;
        ASSIGN:   lift17 = 1;
        ASSIGN:   cs1 = 1;
        DELAY:    1.61+0.42+1.31+0.42+1.55;
        ASSIGN:   lift16 = 0;
        ASSIGN:   lift17 = 0;
        BRANCH,   1:
                  ALWAYS,Rt1;     ! branch to return track at 1

!   ******  HEADED TO BUILD STATION 2  ******

Pass1   ASSIGN:   cs8x9 = 1;
        DELAY:    7.15;          ! travel from rps8 to rps9
        ASSIGN:   cs8x9 = 0;
        QUEUE,    rps9;
        SCAN:   ((builtby .EQ. 2) .AND. (NQ(sb2) .EQ. 0) .AND.
                  (lift6 .EQ. 0) .AND. (lift14 .EQ. 0) .AND.
                  (lift19 .EQ. 0)) .OR. ((builtby .GT. 2) .AND.
                  ((NQ(rps10)+cs9x10) .LT. 2)) .OR. ((sta2stat .EQ.
                  0) .AND. ((NQ(rps10)+cs9x10) .LT. 2)) .OR. ((
                  sta2stat .EQ. 2) .AND. (lift6 .EQ. 0) .AND.
                  (builtby .EQ. 0) .AND. (lift14 .EQ. 0) .AND.
                  (lift19 .EQ. 0) .AND. (NQ(sb2) .EQ. 0)) .OR.
                  ((sta2stat .EQ. 1) .AND. ((NQ(rps10)+cs9x10)
                  .LT. 3) .AND. (builtby .EQ. 0));

        BRANCH,   1:
```

```
                IF,builtby .EQ. 2,ToSta2:
                IF,builtby .GT. 2,Pass2:
                IF,sta2stat .EQ. 0,Pass2:
                IF,sta2stat .EQ. 1,Pass2:
                IF,sta2stat .EQ. 2,ToSta2;

ToSta2  ASSIGN:   sta2stat = 1;
        ASSIGN:   lift6 = 1;
        ASSIGN:   lift14 = 1;
        ASSIGN:   lift19 = 1;
        DELAY:    1.55+0.42+1.31;   ! lift6 to lift14
        ASSIGN:   lift6 = 0;        ! clear lift6
        DELAY:    1.31;             ! lift14 to lift19
        ASSIGN:   lift14 = 0;       ! clear lift14
        DELAY:    0.42+1.62;        ! lift19 to proxy
        ASSIGN:   lift19 = 0;       ! clear lift19
        DELAY:    2.18;             ! proxy to sb2
        QUEUE,    sb2;
        SCAN:   (NQ(sta2buf) .EQ. 0) .AND. (NR(sta2) .EQ. 0) .AND. (NQ(o2ghold) .EQ. 0);
        ASSIGN:   sta2stat = 2;
        STATION,  station2;
        QUEUE,    s2dbuf;
        SEIZE:    sta2;
        QUEUE,    o2ghold;
        SCAN:     (op2gone .EQ. 0);
        BRANCH,   1:
                IF,builtby .EQ. 0,s2new:
                ELSE,s2rej;
s2new   DELAY:    ERLANG(6.013,3)+49.77;   ! build time for new core
        BRANCH,   1:
                ALWAYS,ctn2;
s2rej   DELAY:    EXPO(50.4607)+18.88;     ! fix rejected core
ctn2    RELEASE:  sta2;
        ASSIGN:   builtby = 2;
        QUEUE,    sta2buf;
        SCAN:   (lift18 .EQ. 0) .AND. (lift15 .EQ. 0) .AND.((NQ(rps17)+rt2x1) .LT. 2);
        COUNT:    Sta2Job;
        ASSIGN:   rt2x1 = 1;
        ASSIGN:   lift15 = 1;
        ASSIGN:   lift18 = 1;
        DELAY:    1.62+0.42+1.31+0.42+1.55;
        ASSIGN:   lift15 = 0;
        ASSIGN:   lift18 = 0;
        BRANCH,   1:
                ALWAYS,Rt2;     ! branch to return track at 2

!   ******  HEADED TO BUILD STATION 3  ******

Pass2   ASSIGN:   cs9x10 = 1;
        DELAY:    7.04;         ! travel from rps9 to rps10
        ASSIGN:   cs9x10 = 0;
```

```
          QUEUE,     rps10;
          SCAN:    ((builtby .EQ. 3) .AND. (NQ(sb3) .EQ. 0) .AND.
                   (lift7 .EQ. 0) .AND. (lift13 .EQ. 0) .AND.
                   (lift20 .EQ. 0)) .OR. ((builtby .GT. 3) .AND.
                   ((NQ(rps11)+cs10x11) .LT. 1)) .OR. ((sta3stat .EQ.
                   0) .AND. ((NQ(rps11)+cs10x11) .LT. 1)) .OR. ((
                   sta3stat .EQ. 2) .AND. (lift7 .EQ. 0) .AND.
                   (builtby .EQ. 0) .AND. (lift13 .EQ. 0) .AND.
                   (lift20 .EQ. 0) .AND. (NQ(sb3) .EQ. 0)) .OR.
                   ((sta3stat .EQ. 1) .AND. ((NQ(rps11)+cs10x11)
                   .LT. 2) .AND. (builtby .EQ. 0));

          BRANCH,    1:
                   IF,builtby .EQ. 3,ToSta3:
                   IF,builtby .GT. 3,Pass3:
                   IF,sta3stat .EQ. 0,Pass3:
                   IF,sta3stat .EQ. 1,Pass3:
                   IF,sta3stat .EQ. 2,ToSta3;

ToSta3    ASSIGN:    sta3stat = 1;
          ASSIGN:    lift7 = 1;
          ASSIGN:    lift13 = 1;
          ASSIGN:    lift20 = 1;
          DELAY:     1.55+0.42+1.31;   ! lift7 to lift13
          ASSIGN:    lift7 = 0;        ! clear lift7
          DELAY:     1.31;             ! lift13 to lift20
          ASSIGN:    lift13 = 0;       ! clear lift13
          DELAY:     0.42+1.79;        ! lift20 to proxy
          ASSIGN:    lift20 = 0;       ! clear lift20
          DELAY:     2.03;             ! proxy to sb3
          QUEUE,     sb3;
          SCAN:    (NQ(sta3buf) .EQ. 0) .AND. (NR(sta3) .EQ. 0)
                   .AND. (NQ(o3ghold) .EQ. 0);
          ASSIGN:    sta3stat = 2;
          STATION,   station3;
          QUEUE,     s3dbuf;
          SEIZE:     sta3;
          QUEUE,     o3ghold;
          SCAN:      (op3gone .EQ. 0);
          BRANCH,    1:
                   IF,builtby .EQ. 0,s3new:
                   ELSE,s3rej;
s3new     DELAY:     ERLANG(6.013,3)+49.77;   ! build time for new core
          BRANCH,    1:
                   ALWAYS,ctn3;
s3rej     DELAY:     EXPO(50.4607)+18.88;     ! fix rejected core
ctn3      RELEASE:   sta3;
          ASSIGN:    builtby = 3;
          QUEUE,     sta3buf;
          SCAN:    (lift19 .EQ. 0) .AND. (lift14 .EQ. 0) .AND.((NQ(rps16)+rt3x2) .LT. 2);
          COUNT:     Sta3Job;
```

```
        ASSIGN:   rt3x2 = 1;
        ASSIGN:   lift14 = 1;
        ASSIGN:   lift19 = 1;
        DELAY:    1.62+0.42+1.31+0.42+1.55;
        ASSIGN:   lift14 = 0;
        ASSIGN:   lift19 = 0;
        BRANCH,   1:
                  ALWAYS,Rt3;     ! branch to return track at 3

!   ******  HEADED TO BUILD STATION 4  ******

Pass3   ASSIGN:   cs10x11 = 1;
        DELAY:    7.04;           ! travel from rps10 to rps11
        ASSIGN:   cs10x11 = 0;
        QUEUE,    rps11;
        SCAN:   ((builtby .EQ. 4) .AND. (NQ(sb4) .EQ. 0) .AND.
                  (lift8 .EQ. 0) .AND. (lift12 .EQ. 0) .AND.
                  (lift21 .EQ. 0)) .OR. ((builtby .GT. 4) .AND.
                  ((NQ(rps12)+cs11x12) .LT. 1)) .OR. ((sta4stat .EQ.
                  0) .AND. ((NQ(rps12)+cs11x12) .LT. 1)) .OR. ((
                  sta4stat .EQ. 2) .AND. (lift8 .EQ. 0) .AND.
                  (builtby .EQ. 0) .AND. (lift12 .EQ. 0) .AND.
                  (lift21 .EQ. 0) .AND. (NQ(sb4) .EQ. 0)) .OR.
                  ((sta4stat .EQ. 1) .AND. ((NQ(rps12)+cs11x12)
                  .LT. 2) .AND. (builtby .EQ. 0));

        BRANCH,   1:
                  IF,builtby .EQ. 4,ToSta4:
                  IF,builtby .GT. 4,Pass4:
                  IF,sta4stat .EQ. 0,Pass4:
                  IF,sta4stat .EQ. 1,Pass4:
                  IF,sta4stat .EQ. 2,ToSta4;

ToSta4  ASSIGN:   sta4stat = 1;
        ASSIGN:   lift8 = 1;
        ASSIGN:   lift12 = 1;
        ASSIGN:   lift21 = 1;
        DELAY:    1.55+0.42+1.31;   ! lift8 to lift12
        ASSIGN:   lift8 = 0;        ! clear lift8
        DELAY:    1.31;             ! lift12 to lift21
        ASSIGN:   lift12 = 0;       ! clear lift12
        DELAY:    0.42+1.79;        ! lift21 to proxy
        ASSIGN:   lift21 = 0;       ! clear lift21
        DELAY:    1.96;             ! proxy to sb4
        QUEUE,    sb4;
        SCAN:     (NQ(sta4buf) .EQ. 0) .AND. (NR(sta4) .EQ. 0) .AND. (NQ(o4ghold) .EQ. 0);
        ASSIGN:   sta4stat = 2;
        STATION,  station4;
        QUEUE,    s4dbuf;
        SEIZE:    sta4;
        QUEUE,    o4ghold;
```

```
        SCAN:     (op4gone .EQ. 0);
        BRANCH,   1:
                  IF,builtby .EQ. 0,s4new:
                  ELSE,s4rej;
s4new   DELAY:    ERLANG(6.013,3)+49.77;  ! build time for new core
        BRANCH,   1:
                  ALWAYS,ctn4;
s4rej   DELAY:    EXPO(50.4607)+18.88;    ! fix rejected core
ctn4    RELEASE:  sta4;
        ASSIGN:   builtby = 4;
        QUEUE,    sta4buf;
        SCAN:     (lift20 .EQ. 0) .AND. (lift13 .EQ. 0) .AND. ((NQ(rps15)+rt4x3) .LT. 2);
        COUNT:    Sta4Job;
        ASSIGN:   rt4x3 = 1;
        ASSIGN:   lift20 = 1;
        ASSIGN:   lift13 = 1;
        DELAY:    1.62+0.42+1.31+0.42+1.55;
        ASSIGN:   lift13 = 0;
        ASSIGN:   lift20 = 0;
        BRANCH,   1:
                  ALWAYS,Rt4;     ! branch to return track at 4

!   ******  HEADED TO BUILD STATION 5  ******

Pass4   ASSIGN:   cs11x12 = 1;
        DELAY:    7.05;           ! travel from rps11 to rps12
        ASSIGN:   cs11x12 = 0;
        QUEUE,    rps12;
        SCAN:   ((builtby .EQ. 5) .AND. (NQ(sb5) .EQ. 0) .AND.
                (lift9 .EQ. 0) .AND. (lift11 .EQ. 0) .AND.
                (lift22 .EQ. 0)) .OR. ((builtby .GT. 5) .AND.
                ((NQ(rps12a)+cs12x12a) .LT. 2)).OR.((sta5stat .EQ.
                0) .AND. ((NQ(rps12a)+cs12x12a) .LT. 2)) .OR. ((
                sta5stat .EQ. 2) .AND. (lift9 .EQ. 0) .AND.
                (builtby .EQ. 0) .AND. (lift11 .EQ. 0) .AND.
                (lift22 .EQ. 0) .AND. (NQ(sb5) .EQ. 0)) .OR.
                ((sta5stat .EQ. 1) .AND. ((NQ(rps12a)+cs12x12a)
                .LT. 3) .AND. (builtby .EQ. 0));

        BRANCH,   1:
                  IF,builtby .EQ. 5,ToSta5:
                  IF,builtby .GT. 5,Pass5:
                  IF,sta5stat .EQ. 0,Pass5:
                  IF,sta5stat .EQ. 1,Pass5:
                  IF,sta5stat .EQ. 2,ToSta5;

ToSta5  ASSIGN:   sta5stat = 1;
        ASSIGN:   lift9 = 1;
        ASSIGN:   lift11 = 1;
        ASSIGN:   lift22 = 1;
        DELAY:    1.55+0.42+1.31;   ! lift9 to lift11
```

```
          ASSIGN:    lift9 = 0;        ! clear lift9
          DELAY:     1.31;             ! lift11 to lift22
          ASSIGN:    lift11 = 0;       ! clear lift11
          DELAY:     0.42+1.79;        ! lift22 to proxy
          ASSIGN:    lift22 = 0;       ! clear lift22
          DELAY:     1.96;             ! proxy to sb5
          QUEUE,     sb5;
          SCAN:      (NQ(sta5buf) .EQ. 0) .AND. (NR(sta5) .EQ. 0).AND. (NQ(o5ghold) .EQ. 0);
          ASSIGN:    sta5stat = 2;
          STATION,   station5;
          QUEUE,     s5dbuf;
          SEIZE:     sta5;
          QUEUE,     o5ghold;
          SCAN:      (op5gone .EQ. 0);
          BRANCH,    1:
                     IF,builtby .EQ. 0,s5new:
                     ELSE,s5rej;
s5new     DELAY:     ERLANG(6.013,3)+49.77;   ! build time for new core
          BRANCH,    1:
                     ALWAYS,ctn5;
s5rej     DELAY:     EXPO(50.4607)+18.88;     ! fix rejected core
ctn5      RELEASE:   sta5;
          ASSIGN:    builtby = 5;
          QUEUE,     sta5buf;
          SCAN:      (lift21 .EQ. 0) .AND. (lift12 .EQ. 0) .AND. ((NQ(rps14)+rt5x4) .LT. 2);
          COUNT:     Sta5Job;
          ASSIGN:    lift12 = 1;
          ASSIGN:    lift21 = 1;
          ASSIGN:    rt5x4 = 1;
          DELAY:     1.62+0.42+1.31+0.42+1.55;
          ASSIGN:    lift12 = 0;
          ASSIGN:    lift21 = 0;
          BRANCH,    1:
                     ALWAYS,Rt5;      ! branch to return track at 5

!   ******  HEADED TO BUILD STATION 6  ******

Pass5     ASSIGN:    cs12x12a = 1;
          BRANCH,    1:
                     IF,(NQ(rps12a)+cs12x12a) .EQ. 0,nonein:
                     ELSE,onein;
nonein    DELAY:     7.24;            ! rps12 to rps12a empty queue
          BRANCH,    1:
                     ALWAYS,cnt6;
onein     DELAY:     5.71;            ! rps12 to rps12a one in queue
cnt6      ASSIGN:    cs12x12a = 0;
          QUEUE,     rps12a;
          SCAN:      (lift10 .EQ. 0) .AND. (NQ(sb6) .EQ. 0);
          ASSIGN:    lift10 = 1;
          ASSIGN:    sta6stat = 1;
          DELAY:     1.41+0.42+2.61+0.42+3.83;    ! rps12a to sb6
```

```
        ASSIGN:   lift10 = 0;       ! clear lift10
        QUEUE,    sb6;
        SCAN:     (NQ(sta6buf) .EQ. 0) .AND. (NR(sta6) .EQ. 0).AND. (NQ(o6ghold) .EQ. 0);
        ASSIGN:   sta6stat = 2;
        STATION,  station6;
        QUEUE,    s6dbuf;
        SEIZE:    sta6;
        QUEUE,    o6ghold;
        SCAN:     (op6gone .EQ. 0);
        BRANCH,   1:
                  IF,builtby .EQ. 0,s6new:
                  ELSE,s6rej;
s6new   DELAY:    ERLANG(6.013,3)+49.77;  ! build time for new core
        BRANCH,   1:
                  ALWAYS,ctn6;
s6rej   DELAY:    EXPO(50.4607)+18.88;    ! fix rejected core
ctn6    RELEASE:  sta6;
        ASSIGN:   builtby = 6;
        QUEUE,    sta6buf;
        SCAN:     (lift22 .EQ. 0) .AND. (lift11 .EQ. 0) .AND. ((NQ(rps13)+rt6x5) .LT. 3);
        COUNT:    Sta6Job;

!  **** This is the Return Track  ****

        ASSIGN:   rt6x5 = 1;
        ASSIGN:   lift11 = 1;
        ASSIGN:   lift22 = 1;
        DELAY:    1.62+0.42+1.31+0.42+1.53;  ! sta6 to proxy
        ASSIGN:   lift11 = 0;
        ASSIGN:   lift22 = 0;
        DELAY:    4.01;          ! proxy to rps13
        ASSIGN:   rt6x5 = 0;

        QUEUE,    rps13;
        SCAN:     (lift12 .EQ. 0) .AND. ((NQ(rps14)+rt5x4) .LT. 3);
        ASSIGN:   rt5x4 = 1;
        ASSIGN:   lift12 = 1;
        DELAY:    3.12;          ! rps13 to past lift12
        ASSIGN:   lift12 = 0;
Rt5     DELAY:    3.93;          ! just past lift12 to rps14
        ASSIGN:   rt5x4 = 0;

        QUEUE,    rps14;
        SCAN:     (lift13 .EQ. 0) .AND. ((NQ(rps15)+rt4x3) .LT. 3);
        ASSIGN:   rt4x3 = 1;
        ASSIGN:   lift13 = 1;
        DELAY:    3.12;          ! rps14 to past lift13
        ASSIGN:   lift13 = 0;
Rt4     DELAY:    4.02;          ! just past lift13 to rps15
        ASSIGN:   rt4x3 = 0;
```

```
        QUEUE,    rps15;
        SCAN:     (lift14 .EQ. 0) .AND. ((NQ(rps16)+rt3x2) .LT. 3);
        ASSIGN:   rt3x2 = 1;
        ASSIGN:   lift14 = 1;
        DELAY:    3.12;          ! rps15 to past lift14
        ASSIGN:   lift14 = 0;
Rt3     DELAY:    4.03;          ! just past lift14 to rps16
        ASSIGN:   rt3x2 = 0;

        QUEUE,    rps16;
        SCAN:     (lift15 .EQ. 0) .AND. ((NQ(rps17)+rt2x1) .LT. 3);
        ASSIGN:   rt2x1 = 1;
        ASSIGN:   lift15 = 1;
        DELAY:    3.12;          ! rps16 to past lift15
        ASSIGN:   lift15 = 0;
Rt2     DELAY:    4.02;          ! just past lift15 to rps17
        ASSIGN:   rt2x1 = 0;

        QUEUE,    rps17;
        SCAN:     (lift16 .EQ. 0) .AND. (NQ(rps1) .EQ. 0) .AND. (cs1 .EQ. 0);
        ASSIGN:   cs1 = 1;
Rt1     DELAY:    3.12;          ! clear lift 16

! ***** Entering the Unload Loop *****

        DELAY:    1.71;
        ASSIGN:   cs1 = 0;

        QUEUE,    rps1;
        SCAN:     (NR(vpa) .EQ. 0) .AND. (NQ(vpabuf) .EQ. 0);
        DELAY:    3.26;
        QUEUE,    dbvpa;          ! dummy buffer for vision prealign
        SEIZE:    vpa;            ! seize vision prealign
        DELAY:    ERLANG(0.1110897,3) + 3.31;
        RELEASE:  vpa;            ! release vision prealign
        QUEUE,    vpabuf;         ! hold pallet for vision system
        SCAN:     (NR(vision) .EQ. 0) .AND. (NQ(visbuf) .EQ. 0);
        DELAY:    1.80;

        QUEUE,    dbvision;       ! dummy buffer for vision system
        SEIZE:    vision;         ! seize vision system
        ASSIGN:   status = 0;     ! clear pallet status
        BRANCH,   1:
                  WITH,0.985,Accept:
                  WITH,0.015,Reject;
Reject  DELAY:    UNIFORM(8.21,9.00);
        ASSIGN:   status = 1;     ! set rejected pallet status
        COUNT:    Bcores;
        BRANCH,   1:
                  ALWAYS,Cont;
Accept  DELAY:    ERLANG(0.072442,8) + 5.26;
```

```
          ASSIGN:    builtby = 0;
          COUNT:     Gcores;

Cont      RELEASE:   vision;          ! release vision system station
          QUEUE,     visbuf;          ! hold pallet for vision system
          SCAN:      (NQ(rps2buf) .EQ. 0) .AND. (NR(rps2) .EQ. 0);
          DELAY:     3.33;

          QUEUE,     dbrps2;          ! dummy buffer for chip read
          SEIZE:     rps2;            ! seize rps2 chip read
          DELAY:     1.08;            ! read chip info from pallet
          RELEASE:   rps2;            ! release rps2 chip read
          QUEUE,     rps2buf;         ! hold for clear rps3
          SCAN:      NQ(rps3) .EQ. 0;
          DELAY:     5.00;

          QUEUE,     rps3;
          SCAN:      NQ(sps1) .EQ. 0;   ! see if sps1 is clear
          DELAY:     1.61;            ! go to sps1

          QUEUE,     sps1;
          SCAN:      (NR(band) .EQ. 0) .AND. (NQ(bandbuf) .EQ. 0);
          DELAY:     2.94;            ! sps delay time + travel time

          QUEUE,     dband;           ! dummy buffer for bander
          SEIZE:     band;            ! seize banding station
          BRANCH,    .1:
                     IF,status .EQ. 1,Bad1:
                     ELSE,Good1;
Good1     DELAY:     5.94;            ! constant banding time
          BRANCH,    1:
                     ALWAYS,Cont2;
Bad1      DELAY:     1.08;            ! regular chip read time
Cont2     RELEASE:   band;
          QUEUE,     bandbuf;         ! Used in initialization also
          SCAN:      (lift1 .EQ. 0) .AND. (NQ(rps4) .LT. 2);
          ASSIGN:    lift1 = 1;
          DELAY:     4.93;            ! H to lift1 proxy w/ lift time
          ASSIGN:    lift1 = 0;
          DELAY:     3.13;            ! Just past lift1 to rps4
          QUEUE,     rps4;
          SCAN:      NQ(rps5) .LT. 2;

          DELAY:     5.36;            ! Travel from rps4 to rps5
          QUEUE,     rps5;
          SCAN:      (lift2 .EQ. 0) .AND. (NQ(rps6) .EQ. 0);
          ASSIGN:    lift2 = 1;
          DELAY:     5.04;            ! J to lift2 proxy w/ lift time
          ASSIGN:    lift2 = 0;
          DELAY:     1.43;            ! proxy switch to K
          QUEUE,     rps6;
```

```
        SCAN:     NQ(sps2) .EQ. 0;    ! wait for sps2 to be clear
        DELAY:    1.65;               ! move to sps2
        QUEUE,    sps2;
        SCAN:     (NQ(rps7) .EQ. 0);
        DELAY:    2.82;
        QUEUE,    rps7;
        SCAN:     (NQ(ulbuf) .EQ. 0) .AND. (NR(unload) .EQ. 0);
        DELAY:    8.72;

        QUEUE,    duload;             ! dummy buffer for unload
        SEIZE:    unload;             ! seize unload station
        BRANCH,   1:
                  IF,status .EQ. 1,Bad2:
                  ELSE,Good2;
Good2   DELAY:    4.98;               ! constant unload time
        BRANCH,   1:
                  ALWAYS,Cont3;
Bad2    DELAY:    1.08;               ! regular chip read time
Cont3   RELEASE:  unload:
        QUEUE,    ulbuf;
        SCAN:     (lift3 .EQ. 0) .AND. ((NQ(rps7a)+cs7x7a) .LT. 3);
        ASSIGN:   cs7x7a = 1;
        ASSIGN:   lift3 = 1;
        DELAY:    6.65;       ! unload to lift3 proxy w/ lift time
        ASSIGN:   lift3 = 0;
        DELAY:    3.32;       ! lift3 proxy to rps7a
        ASSIGN:   cs7x7a = 0;
        QUEUE,    rps7a;
        SCAN:     (NQ(rps7b)+cs7ax7b) .LT. 3;
        ASSIGN:   cs7ax7b = 1;
        DELAY:    2.58;
        BRANCH,   1:
                  ALWAYS,top;     ! Enter the station feeder loop
END;
```

APPENDIX B

EVAPORATOR ASSEMBLY SYSTEM SIMULATION,

SIMAN EXPERIMENT FRAME

```
BEGIN;

ATTRIBUTES:  1,builtby,0:
             2,StartTime:
             3,status;

VARIABLES:   lift1,0:
             lift2,0:
             lift3,0:
             lift4,0:
             lift5,0:
             lift6,0:
             lift7,0:
             lift8,0:
             lift9,0:
             lift10,0:
             lift11,0:
             lift12,0:
             lift13,0:
             lift14,0:
             lift15,0:
             lift16,0:
             lift17,0:
             lift18,0:
             lift19,0:
             lift20,0:
             lift21,0:
             lift22,0:
             sta1stat,2:
             sta2stat,2:
             sta3stat,2:
             sta4stat,2:
             sta5stat,2:
             sta6stat,2:
             laststa,6:
             cs1,0:
             cs7x7a,0:
             cs7ax7b,0:
             cs7bx8,0:
             cs8x9,0:
             cs9x10,0:
             cs10x11,0:
             cs11x12,0:
             cs12x12a,0:
             rt6x5,0:
             rt5x4,0:
             rt4x3,0:
             rt3x2,0:
             rt2x1,0:
             op1gone,1:
             op2gone,1:
```

op3gone,1:
op4gone,1:
op5gone,1:
op6gone,1;

STATIONS: station1:
station2:
station3:
station4:
station5:
station6:
vision_prealign:
vision_system:
chip_read:
core_bander:
unload_core;

QUEUES: 1,rps8, FIFO:
2,rps9, FIFO:
3,rps10, FIFO:
4,rps11, FIFO:
6,rps12, FIFO:
7,rps12a, FIFO:
8,rps13, FIFO:
9,rps14, FIFO:
10,rps15, FIFO:
11,rps16, FIFO:
12,rps17, FIFO:
13,sta1buf, FIFO:
14,sta2buf, FIFO:
15,sta3buf, FIFO:
16,sta4buf, FIFO:
17,sta5buf, FIFO:
18,sta6buf, FIFO:
19,s1dbuf, FIFO:
20,s2dbuf, FIFO:
21,s3dbuf, FIFO:
22,s4dbuf, FIFO:
23,s5dbuf, FIFO:
24,s6dbuf, FIFO:
25,sb1, FIFO:
26,sb2, FIFO:
27,sb3, FIFO:
28,sb4, FIFO:
29,sb5, FIFO:
30,sb6, FIFO:

31,rps1, FIFO:
32,dbvpa, FIFO:
33,vpabuf, FIFO:
34,dbvision,FIFO:

```
                35,visbuf,  FIFO:
                36,dbrps2,  FIFO:
                37,rps2buf, FIFO:
                38,rps3,    FIFO:
                39,sps1,    FIFO:
                40,dband,   FIFO:
                41,bandbuf, FIFO:
                42,rps4,    FIFO:
                43,rps5,    FIFO:
                44,rps6,    FIFO:
                45,sps2,    FIFO:
                46,rps7,    FIFO:
                47,duload,  FIFO:
                48,ulbuf,   FIFO:
                49,rps7a,   FIFO:
                50,rps7b,   FIFO:
                51,o1ghold, FIFO:
                52,o2ghold, FIFO:
                53,o3ghold, FIFO:
                54,o4ghold, FIFO:
                55,o5ghold, FIFO:
                56,o6ghold, FIFO;

RESOURCES:      1,sta1:
                2,sta2:
                3,sta3:
                4,sta4:
                5,sta5:
                6,sta6:
                7,vpa:
                8,vision:
                9,rps2:
                10,band:
                11,unload;

ARRIVALS:
        1,QUEUE(bandbuf),0,1:
        2,QUEUE(bandbuf),10,1:
        3,QUEUE(bandbuf),20,1:
        4,QUEUE(bandbuf),30,1:
        5,QUEUE(bandbuf),40,1:
        6,QUEUE(bandbuf),50,1:
        7,QUEUE(bandbuf),60,1:
        8,QUEUE(bandbuf),70,1:
        9,QUEUE(bandbuf),80,1:
        10,QUEUE(bandbuf),90,1:
        11,QUEUE(bandbuf),100,1:
        12,QUEUE(bandbuf),110,1:
        13,QUEUE(bandbuf),120,1:
        14,QUEUE(bandbuf),130,1:
        15,QUEUE(bandbuf),140,1:
```

```
        16,QUEUE(bandbuf),150,1:
        17,QUEUE(bandbuf),160,1:
        18,QUEUE(bandbuf),170,1:
        19,QUEUE(bandbuf),180,1:
        20,QUEUE(bandbuf),190,1:
        21,QUEUE(bandbuf),200,1:
        22,QUEUE(bandbuf),210,1;

COUNTERS:   1,Sta1Job:
            2,Sta2Job:
            3,Sta3Job:
            4,Sta4Job:
            5,Sta5Job:
            6,Sta6Job:
            7,Bcores:
            8,Gcores;

FILES:      PM_OUTPUT,"PM.OUT",SEQ,FRE;

REPLICATE,  5,0,31500;

END;
```

APPENDIX C

TANDEM ALGORITHM MASTER PROGRAM,

C SOURCE CODE

```
/* tandem.c -- This is the driver file for the tandem algorithm.
              This algorithm calls GA1.EXE and SQG4.EXE.

              GA1.EXE implements a simple genetic algorithm to optimize
              the station configurations in the assembly system.

              SQG4.EXE takes the station configurations which GA1.EXE
              wrote into TAND.DAT and then optimizes the number of
              pallets.  The SQG method is applied to each of the station
              configurations found by the simple genetic algorithm.

              Masters Thesis Work
              Kraig A. Downs
              IMSE, Iowa State University
              Spring 1993
*/

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

FILE *fpout,*fpin1,*fpin2;

main()
{
 intro_to_screen();
 printf("\n\n       Genetic Algorithm executing . . . ");
 spawnl(P_WAIT,"ga1.exe",NULL);
 printf("\n\n       SQG Algorithm executing . . . ");
 spawnl(P_WAIT,"sqg4.exe",NULL);
 printf("\n\n\n The program has finished, \n");
 printf(" check TAOUT.DAT for the results.\n\n");
}


intro_to_screen()
  /* This functions prints an introduction to the screen */
{
 clrscr();
 printf("\n                THE TANDEM ALGORITHM\n\n");
 printf("                By: Kraig Downs\n");
 printf("           Master of Science - Thesis, 1993\n\n\n");
 printf("\n  GA1.EXE will place data in TAND.DAT\n");
 printf("  The final output data will be placed in TAOUT.DAT\n");
 printf("\n\n  Please wait while the program is operating . . .");
}
```

APPENDIX D

TANDEM ALGORITHM SLAVE PROGRAM,

IMPLEMENTATION OF A GENETIC ALGORITHM,

C SOURCE CODE

```
/* ga1.c -- a simple genetic algorithm
            This genetic algorithm is the simple genetic algorithm (SGA)
            which is described in the following text :

                    "Genetic Algorithms in Search, Optimization,
                      and Machine Learning"
                    David E. Goldberg
                    Addison-Wesley Publishing
                    1989

   This program is the first portion of the tandem algorithm.  This program
   is designed to supply good station configurations.  The SQG part of the
   tandem algorithm, looks at number of pallets.

   This program is part of the research done to fulfill the requirements
   for a Master of Science degree in Industrial Engineering.
         Programming by :
                          Kraig Downs
                          IMSE Dept.
                          Iowa State University
                          1993                    */

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

struct individual
  {
   int chromo[12];  /* genotype = bitstring */
   int x[3];        /* phenotype = 2 integers  */
   float fitness;   /* objective function value */
   int parent1;     /* parent number 1 */
   int parent2;     /* parent number 2 */
   int cross_site;  /* cross-over site */
  } oldpop[80],newpop[80];


/* variable declarations */

/*  The four random number generators used in this program (r1num(),
   r2num(), r3num(), r4num()) are derived from the published work listed
   below.
      "Some Efficient Random Number Generators for Micro-Computers"
      Thesen, Sun, and Wang
      Department of Industrial Engineering, University of Wisconsin-Madison
      Proceedings of the 1984 Winter Simulation Conference
*/

float r1num(void);   /* random number generator #1 */
float r2num(void);   /* random number generator #2 */
float r3num(void);   /* random number generator #3 */
```

```
float r4num(void);   /* random number generator #4 */

int popsize,gen,maxgen;
float pcross,pmutation,sumfitness;
int nmutation,ncross,jcross;
float avg,max,min;
long seed1,seed3,seed4;        /* global rng seeds */
int seed2;
float req,upc1,upc2;

FILE *fpout;

main()
{
int i;

upc1 = 1.0;
upc2 = 0.25;
req = 1368.0;

printf("\n\n     Input a random seed --> ");
scanf("%li",&seed1);
popsize = 60;   /* WARNING!! This must be an even number. */
maxgen = 10;
pcross = 0.6;          /* crossover probability */
pmutation = 0.02;      /* mutation probability */

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

/* open global output file */
if((fpout=fopen("tand.dat","w")) == NULL)
  {
    printf("Unable to open TAND.DAT file !!\n\n");
    exit(1);
  }
create_init_population();
statistics(oldpop);
/* generate_init_report(); */
/*debuggit(oldpop);*/
/*fclose(fpout);*/
/*exit(1);*/
for (i=1; i<=maxgen; i++)
  {
    gen++;
    generation();
    statistics(newpop);
    /* report(); */
    copy_new_into_old(oldpop,newpop);
  }
```

```
sc_np_out();  /* outputs final station configs and respective # of pals */
fclose(fpout);
}


copy_new_into_old(oldpop,newpop)
  struct individual oldpop[80];
  struct individual newpop[80];
    /* This function copies the newpop into the oldpop, thus readying
          the population for the next generation.
    */
{
   int i,j;

   for (i=1; i<=popsize; i++)
     {
       for (j=1; j<=11; j++)
            oldpop[i].chromo[j] = newpop[i].chromo[j];
       oldpop[i].x[1] = newpop[i].x[1];
       oldpop[i].x[2] = newpop[i].x[2];
       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;
     }
}


int select_individual(work_pop)
  struct individual work_pop[80];
          /* This function is responsible for the selection of a
             single individual using a modified roulette wheel
             selection method.  This function is transformed to
             accommodate a minimization problem.  */
{
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;
}


int flip(pcross)
   float pcross;
         /*  returns 1 with probability pcross, zero otherwise. */
{
 float rndnum;
 rndnum = r1num();
 if (rndnum <= pcross)
   return 1;
 else
   return 0;
}


int find_x_site()
    /* Selects a random integer between 1 and 10 inclusive.  We are
          looking for the gap between 2 alleles.  */
{
int num,randint;

(int)randint = r1num()*32767;
num = (randint%10) + 1;
return num;
}


crossover(parent1,parent2,child1,child2)
   int parent1[12],parent2[12],child1[12],child2[12];
         /* This function determines whether a cross is going to occur
            and then performs the cross.
         */
{
int j;

if (flip(pcross))
  {
   jcross = find_x_site();    /* assumes constant chromosome length */
   ncross = ncross + 1;
  }
else
   jcross = 11;

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

```
    child2[j] = mutation(parent2[j]);
  }

if (jcross != 11)
  {
    for (j=jcross+1;j<=11;j++)
      {
          child1[j] = mutation(parent2[j]);
          child2[j] = mutation(parent1[j]);
      }
  }
}



int mutation(alleleval)
    int alleleval;
              /* This function mutates an allele with pmutation probability
                 and updates mutation counter if a mutation occurs. */
{
int mutate;
mutate = flip(pmutation); /* mutate with pmutation probability */
if (mutate)             /* Change the allele value */
  {
    nmutation = nmutation + 1;
    if (alleleval)
      return 0;
    else
      return 1;
  }
else
  return alleleval;    /* No change occurred */
}



generation()
    /* This function creates a new generation using select, crossover,
       and mutation.  This function ( generation() ) assumes an even
       numbered population size.  */
{
int 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 */
    newpop[j].x[1] = decode_station_config(newpop[j].chromo);
    newpop[j].x[2] = decode_num_of_pallets(newpop[j].chromo);
    set_fitness_value(newpop,j);
    newpop[j].parent1 = mate1;
    newpop[j].parent2 = mate2;
    newpop[j].cross_site = jcross;

    newpop[j+1].x[1] = decode_station_config(newpop[j+1].chromo);
    newpop[j+1].x[2] = decode_num_of_pallets(newpop[j+1].chromo);
    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 */
   }
}



int decode_station_config(station_config)
   int station_config[12];
     /* This function decodes the station configuration */
{
int b32,b16,b8,b4,b2,b1,bsum;

b32 = station_config[1]*32;
b16 = station_config[2]*16;
b8  = station_config[3]*8;
b4  = station_config[4]*4;
b2  = station_config[5]*2;
b1  = station_config[6]*1;
bsum = b32+b16+b8+b4+b2+b1;
return bsum;
}



int decode_num_of_pallets(station_config)
   int station_config[12];
     /* This function decodes the number of pallets */
{
int b16,b8,b4,b2,b1,bsum;

b16 = station_config[7]*16;
b8  = station_config[8]*8;
b4  = station_config[9]*4;
b2  = station_config[10]*2;
```

```
b1  = station_config[11]*1;
bsum = b16+b8+b4+b2+b1;
return bsum;
}


set_fitness_value2(work_pop,index)
  struct individual work_pop[80];
  int index;
   /* This function scans the data file for the number of good
      cores produced by a given system configuration. It also
      calculates the performance measure using the average value
      for production rates. */
{
FILE *fp;
float avg_gc,pm,oprate,hrs_per_shift;
float number_of_operators;
int config,pals,legal,mp,np;

oprate = 15.00;
hrs_per_shift = 8.0;

mp = max_pallets(work_pop,index);
np = decode_num_of_pallets(work_pop[index].chromo);
if ((np > mp)||(work_pop[index].x[2] == 0)||(work_pop[index].x[1]))
  {
   legal=0;
   avg_gc = 1;     /* illegal number of pallets gets low production rate */
  }
else
  legal=1;
if (legal)
 {
  if((fp=fopen("simavg.dat","r")) == NULL)
    {
         printf("Cannot open SIMAVG.DAT file !!\n\n");
         exit(1);
    }
  fscanf(fp,"%f%d%d\n",&avg_gc,&config,&pals);
  while((config != work_pop[index].x[1])||(pals != work_pop[index].x[2]))
    fscanf(fp,"%f%d%d\n",&avg_gc,&config,&pals);
  fclose(fp);
 }
number_of_operators = work_pop[index].chromo[1] + work_pop[index].chromo[2] +
                      work_pop[index].chromo[3] + work_pop[index].chromo[4] +
                      work_pop[index].chromo[5] + work_pop[index].chromo[6];
if ((number_of_operators == 0)||(work_pop[index].x[2] == 0))
 {
  number_of_operators = 10;  /* big penalty for no operator case */
  avg_gc = 1;         /* minimum production for no operator case */
 }
```

```
pm = (oprate * hrs_per_shift * number_of_operators)/avg_gc;
work_pop[index].fitness = pm;
}


set_fitness_value(work_pop,index)
  struct individual work_pop[80];
  int index;
   /*  This function scans the data file for the number of good
         cores produced by a given system configuration. It also
         calculates the performance measure using a Normal
         random variate using the parameters created from the
         5 replications.  */
{
FILE *fp;
float mean,sdev,prod,pm,oprate,hrs_per_shift;
float number_of_operators,mp,np,nrv,sum;
int config,pals,legal,i;

oprate = 15.00;
hrs_per_shift = 8.0;

mp = max_pallets(work_pop,index);
np = decode_num_of_pallets(work_pop[index].chromo);
if ((np > mp)||(work_pop[index].x[2] == 0)||(work_pop[index].x[1] == 0))
  {
   legal=0;
   nrv=1;   /* illegal number of pallets gets low production rate */
  }
else
  legal=1;
if (legal)
 {
  if((fp=fopen("simnorm.dat","r")) == NULL)
    {
        printf("Cannot open SIMNORM.DAT file !!\n\n");
        exit(1);
    }
  fscanf(fp,"%f%f%d%d\n",&mean,&sdev,&config,&pals);
  while((config != work_pop[index].x[1])||(pals != work_pop[index].x[2]))
    fscanf(fp,"%f%f%d%d\n",&mean,&sdev,&config,&pals);
  fclose(fp);
  /* create a normal random variate */
  sum = -6.0;
  for (i=1; i<=12; i++)
    sum = sum + r1num();
  nrv = (sum * sdev) + mean;
 }
number_of_operators = work_pop[index].chromo[1] + work_pop[index].chromo[2] +
                      work_pop[index].chromo[3] + work_pop[index].chromo[4] +
                      work_pop[index].chromo[5] + work_pop[index].chromo[6];
```

```
if ((number_of_operators == 0)||(work_pop[index].x[2] == 0))
  {
   number_of_operators = 10;   /* big penalty for no operator case */
   nrv = 1;          /* minimum production for no operator case */
  }
pm = (oprate * hrs_per_shift * number_of_operators)/nrv;
work_pop[index].fitness = pm;
}


set_fitness_value1(work_pop,index)
  struct individual work_pop[80];
  int index;
    /*  This function scans the data file for the number of good
          cores produced by a given system configuration. It also
          calculates the performance measure using a Normal
          random variate using the parameters created from the
          5 replications.  */
{
FILE *fp;
float mean,sdev,prod,pm,oprate,hrs_per_shift;
float number_of_operators,mp,np,nrv,sum;
int config,pals,legal,i;
float penalty,rr;

oprate = 15.00;
hrs_per_shift = 8.0;

mp = max_pallets(work_pop,index);
np = decode_num_of_pallets(work_pop[index].chromo);
if ((np > mp)||(work_pop[index].x[2] == 0)||(work_pop[index].x[1] == 0))
   {
    legal=0;
    nrv=1;    /* illegal number of pallets gets low production rate */
   }
else
   legal=1;
if (legal)
  {
   if((fp=fopen("simnorm.dat","r")) == NULL)
     {
          printf("Cannot open SIMNORM.DAT file !!\n\n");
          exit(1);
     }
   fscanf(fp,"%f%f%d%d\n",&mean,&sdev,&config,&pals);
   while((config != work_pop[index].x[1])||(pals != work_pop[index].x[2]))
     fscanf(fp,"%f%f%d%d\n",&mean,&sdev,&config,&pals);
   fclose(fp);
   /* create a normal random variate */
   sum = -6.0;
   for (i=1; i<=12; i++)
```

```
    sum = sum + r1num();
   nrv = (sum * sdev) + mean;
  }
number_of_operators = work_pop[index].chromo[1] + work_pop[index].chromo[2] +
                      work_pop[index].chromo[3] + work_pop[index].chromo[4] +
                      work_pop[index].chromo[5] + work_pop[index].chromo[6];
if ((number_of_operators == 0)||(work_pop[index].x[2] == 0))
  {
   number_of_operators = 10;   /* big penalty for no operator case */
   nrv = 1;          /* minimum production for no operator case */
  }
rr = nrv/req;
if (rr<=1.0)  penalty = (upc1 * (req - nrv))/nrv;
else  penalty = (upc2 * (nrv - req))/nrv;
pm = (oprate * hrs_per_shift * number_of_operators)/nrv;
pm = pm + penalty;
work_pop[index].fitness = pm;
}


set_fitness_value4(work_pop,index)
  struct individual work_pop[80];
  int index;
  /*  This function scans the data file for the number of good
      cores produced by a given system configuration. It also
      calculates the performance measure using the average value
      for production rates. */
{
FILE *fp;
float avg_gc,pm,oprate,hrs_per_shift;
float number_of_operators,rr,penalty;
int config,pals,legal,mp,np;

oprate = 15.00;
hrs_per_shift = 8.0;

mp = max_pallets(work_pop,index);
np = decode_num_of_pallets(work_pop[index].chromo);
if ((np > mp)||(work_pop[index].x[2] == 0)||work_pop[index].x[1] == 0)
  {
   legal=0;
   avg_gc = 1;    /* illegal number of pallets gets low production rate */
  }
else
  legal=1;
if (legal)
  {
   if((fp=fopen("simavg.dat","r")) == NULL)
     {
          printf("Cannot open SIMAVG.DAT file !!\n\n");
          exit(1);
```

```
    }
   fscanf(fp,"%f%d%d\n",&avg_gc,&config,&pals);
   while((config != work_pop[index].x[1])||(pals != work_pop[index].x[2]))
     fscanf(fp,"%f%d%d\n",&avg_gc,&config,&pals);
   fclose(fp);
  }
number_of_operators = work_pop[index].chromo[1] + work_pop[index].chromo[2] +
                      work_pop[index].chromo[3] + work_pop[index].chromo[4] +
                      work_pop[index].chromo[5] + work_pop[index].chromo[6];
if ((number_of_operators == 0)||(work_pop[index].x[2] == 0))
  {
   number_of_operators = 10;   /* big penalty for no operator case */
   avg_gc = 1;          /* minimum production for no operator case */
  }
rr = avg_gc/req;
if (rr<=1.0)  penalty = (upc1 * (req - avg_gc))/avg_gc;
else  penalty = (upc2 * (avg_gc - req))/avg_gc;
pm = (oprate * hrs_per_shift * number_of_operators)/avg_gc;
pm = pm + penalty;
work_pop[index].fitness = pm;
}


create_init_population()
   /*  This function creates the initial population of strings.
         Strings are randomly created.
   */
{
  int x1,x2;

  for (x1=1; x1<=popsize; x1++)
    {
     for (x2=1; x2<=11; x2++)
          oldpop[x1].chromo[x2] = flip(0.5);
     oldpop[x1].x[1] = decode_station_config(oldpop[x1].chromo);
     oldpop[x1].x[2] = decode_num_of_pallets(oldpop[x1].chromo);
     set_fitness_value(oldpop,x1);
     oldpop[x1].parent1 = 0;
     oldpop[x1].parent2 = 0;
     oldpop[x1].cross_site = 0;
    }
}


int max_pallets(work_pop,index)
  struct individual work_pop[80];
  int index;
   /* This function calculates the maximum number of pallets allowed
         for a given station configuration.  The absolute maximum number
         of pallets is 31.
   */
```

```
{
int x1,x2,i,num_sta_op,laststa,maxpals;

num_sta_op = work_pop[index].chromo[1] + work_pop[index].chromo[2]+
             work_pop[index].chromo[3] + work_pop[index].chromo[4]+
             work_pop[index].chromo[5] + work_pop[index].chromo[6];
laststa = 0;
laststa = work_pop[index].chromo[6]==1 ? 1 : laststa;
laststa = work_pop[index].chromo[5]==1 ? 2 : laststa;
laststa = work_pop[index].chromo[4]==1 ? 3 : laststa;
laststa = work_pop[index].chromo[3]==1 ? 4 : laststa;
laststa = work_pop[index].chromo[2]==1 ? 5 : laststa;
laststa = work_pop[index].chromo[1]==1 ? 6 : laststa;
x2 = 10;
switch(laststa)
 {
  case 6:   x1 = 10;
            break;
  case 5:   x1 = 8;
            break;
  case 4:   x1 = 7;
            break;
  case 3:   x1 = 6;
            break;
  case 2:   x1 = 4;
            break;
  case 1:   x1 = 2;
            break;
  case 0:   x1 = 1;
            break;
 }
maxpals = (2*num_sta_op) + x1 + x2;
if (maxpals == 32) maxpals=31;
return maxpals;
}



generate_init_report()
  /* This function is used to generate a header in the output file
     which gives all the initial system parameters.
  */
{
fprintf(fpout,"The Application of a Genetic Algorithm to the Optimization\n");
fprintf(fpout,"of an Asynchronous Semi-Automatic Assembly System.\n\n");
fprintf(fpout,"                Kraig A. Downs\n");
fprintf(fpout,"                Thesis Work\n");
fprintf(fpout,"                Spring 1993\n\n\n");
fprintf(fpout,"Summary of Parameters\n");
fprintf(fpout,"    Population Size : %d\n",popsize);
```

```
fprintf(fpout,"    Chromosome Length is fixed at 11.\n");
fprintf(fpout,"    Maximum number of generations : %d\n",maxgen);
fprintf(fpout,"    Crossover probability : %f\n",pcross);
fprintf(fpout,"    Mutation probability : %f\n\n\n",pmutation);
fprintf(fpout,"Initial Population Statistics\n");
fprintf(fpout,"    Initial population minimum fitness : %f\n",min);
fprintf(fpout,"    Initial population maximum fitness : %f\n",max);
fprintf(fpout,"    Initial population average fitness : %f\n",avg);
fprintf(fpout,"    Initial population sum of fitness  : %f\n",sumfitness);
fprintf(fpout,"\n\n\n\n\n\n\n\n\n");
}


statistics(work_pop)
  struct individual work_pop[80];
  /*  This function calculates population statistics for a generation. */
{
 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 */
}


sc_np_out()
   /* This function outputs the station configurations and the number
        of pallets of the individuals in the final population.
   */
{
int i;
fprintf(fpout,"%d\n",popsize);
for (i=1; i<=popsize; i++)
  fprintf(fpout,"%d    %d\n",newpop[i].x[1],newpop[i].x[2]);
}   .



report()
   /* This function creates the generation reports used to view
        the results from the genetic algorithm.
   */
```

```
{
int i;    .

fprintf(fpout,"-------------------------------------------------------------------------------\n");
fprintf(fpout,"                    Population Report\n");
fprintf(fpout,"        Generation %d                  Generation %d\n",gen-1,gen);
fprintf(fpout," #    Individual  SC NP Fitness    #  Parents XS  Individual SC NP Fitness\n");
fprintf(fpout,"-------------------------------------------------------------------------------\n");
for (i=1; i<=popsize; i++)
 {
  fprintf(fpout,"%3d: %1d%1d%1d%1d%1d%1d%1d%1d%1d%1d%1d  %2d,%2d %10.6f ][ ",
            i,oldpop[i].chromo[1],oldpop[i].chromo[2],oldpop[i].chromo[3],oldpop[i].chromo[4],
            oldpop[i].chromo[5],oldpop[i].chromo[6],oldpop[i].chromo[7],oldpop[i].chromo[8],
            oldpop[i].chromo[9],oldpop[i].chromo[10],oldpop[i].chromo[11],oldpop[i].x[1],
            oldpop[i].x[2],oldpop[i].fitness);
  fprintf(fpout,"%3d:(%2d,%2d) %2d %1d%1d%1d%1d%1d%1d%1d%1d%1d%1d%1d %2d,%2d
%10.6f\n",
            i,newpop[i].parent1,newpop[i].parent2,newpop[i].cross_site,
            newpop[i].chromo[1],newpop[i].chromo[2],newpop[i].chromo[3],newpop[i].chromo[4],
            newpop[i].chromo[5],newpop[i].chromo[6],newpop[i].chromo[7],newpop[i].chromo[8],
            newpop[i].chromo[9],newpop[i].chromo[10],newpop[i].chromo[11],newpop[i].x[1],
            newpop[i].x[2],newpop[i].fitness);
 }
fprintf(fpout,"-------------------------------------------------------------------------------\n");
fprintf(fpout,"Generation 1 Stats: max=%10.6f, min=%10.6f, avg=%10.6f, sumfit=%10.6f\n",max,min,
            avg,sumfitness);
fprintf(fpout,"Accumulated Stats: nmutation=%5d, ncross=%5d\n",nmutation,ncross);
fprintf(fpout,"-------------------------------------------------------------------------------\n\n\n\n\n\n");
}


/* ********************************************************************** */

int uniform(min,max)
 float min,max;
{
 float urn;
 int uprod;
 urn = (min + (((max+1) - min)*r1num()));
 uprod = (int)urn;
 return uprod;
}
float r1num()
{
 float ran_num;
 seed1 = (seed1*2125) + 1;
 if (seed1 < 0)
  seed1 = seed1 + 2147483647 + 1;
 ran_num = seed1/2147483647.0;
 return ran_num;
                                        }
```

APPENDIX E

TANDEM ALGORITHM SLAVE PROGRAM, IMPLEMENTATION OF A SQG METHOD, C SOURCE CODE

```
/* SQG4.C -- This is the program file for the implementation of a
            stochastic quasigradient method to the optimization of
            an evaporator assembly system at Ford Refrigeration and
            Electronics in Connersville, IN.

                        Kraig A. Downs
                        Masters Thesis Work
                        February 1993

            This is the second half of the tandem algorithm.  The tandem
            algorithm is described and programmed in TANDEM.c.  TANDEM.c
            simply calls GA1.exe and SQG4.exe in order to implement the
            tandem application of a genetic algorithm and a SQG method.

            This variation of the SQG method uses station configurations
            produce by GA1.exe.  It then optimizes the number of pallets
            for each of these configurations.

            This implementation of SQG uses a forward finite difference
            equation to estimate the gradient.  It also uses a modified
            step size.
*/

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

float r1num();
int maxpals();
int find_config_num();
float set_obj_func_val();    /* uses pm1, simnorm.dat */
float set_obj_func_val2();   /* uses pm2, simnorm.dat */
float set_obj_func_val3();   /* uses pm2, simavg.dat */

struct solution
  {
    int np;     /* number of pallets */
    int s6;     /* station 6 */
    int s5;     /* station 5 */
    int s4;     /* station 4 */
    int s3;     /* station 3 */
    int s2;     /* station 2 */
    int s1;     /* station 1 */
    float pm;   /* performance measure */
  } csol,tsol;    /* current solution, old solution */

FILE *fp1;       /* output file */
long seed1;      /* random number generator seed */
float pm;        /* performance measure */
int pss;         /* pallet step size */
int iter_num;    /* the iteration number */
float g1;        /* gradient 1 */
```

```
int htc[2];     /* how to change */
int popsize;    /* number of configurations from GA results */
int gar[80][3];   /* GA results from TAND.DAT file */
float pm_array[80][4]; /* Contains the final performance measure for
                            each config. Final pm found using SQG method. */
float req;      /* required production rate */
float rp;       /* reduction percentage */
float upc1;     /* unit cost for underproduction */
float upc2;     /* unit cost for overproduction */

main()
{
int i,j,ipc;

if ((fp1=fopen("taout.dat","w"))==NULL)
  {
   puts("ERROR !  Unable to open TAOUT.DAT\n\n");
   exit(1);
  }

seed1 = 92383;
req = 1368.0;
rp = 0.85;
ipc = 15;  /* the number of iterations per configuration */
upc1 = 2.0;
upc2 = 0.40;

init_rand_gen();  /* initializes random number generator */
load_ga_results();

for (i=1; i<=popsize; i++)
  {
   iter_num = 1;     /* initialize iteration counter */
   fprintf(fp1,"\n\n\n");
   fprintf(fp1," >>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>\n");
   fprintf(fp1,"     Station configuration # %d\n",i);
   fprintf(fp1," <<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<\n");
   pss = 8;
   fill_csol(i);
   for (j=1; j<=ipc; j++)
     {
         calculate_gradients(csol);
         set_next_solution();
         /* write_stats(); */
         iter_num++;
         modify_pss();
     }
   if (tsol.pm < csol.pm)  set_pm_array(tsol,i);
   else  set_pm_array(csol,i);
  }
sort_by_pm();  /* sorts the pm_array[][] in ascending order by pm */
```

```
output_pm_array();  /* writes pm_array[][] to a file */
fclose(fp1);
}


fill_csol(index)
 int index;
 /* This function properly fills the current solution csol with
    the station configuration and number of pallets.
 */
{
 encode_sc(gar[index][1]);
 csol.np = gar[index][2];
 csol.pm = set_obj_func_val(csol);
}


set_pm_array(wsol,index)
  struct solution wsol;
  int index;
        /* This function adds the optimal solution to the pm_array[][] */
{
 pm_array[index][1] = (find_config_num(wsol) * 1.0);
 pm_array[index][2] = (wsol.np * 1.0);
 pm_array[index][3] = (wsol.pm * 1.0);
}



load_ga_results()
   /* This function loads the results from TAND.DAT into the two
        dimensional array gar[80][3].
   */
{
 FILE *fpx;
 int i;

 if ((fpx=fopen("tand.dat","r"))==NULL)
   {
    puts("ERROR !  Unable to open TAND.DAT \n\n");
    exit(1);
   }
 fscanf(fpx,"%d\n",&popsize);
 for (i=1; i<=popsize; i++)
   fscanf(fpx,"%d     %d\n",&gar[i][1],&gar[i][2]);
 fclose(fpx);
}


calculate_gradients(wsol)
```

```
 struct solution wsol;
   /* This function determines the quasigradients for the number of
      pallets decision variable. Note: this function uses a FORWARD
      FINITE DIFFERENCE equation to estimate the gradient.
   */
{
struct solution tempsol;
float x,y;

tempsol.s6=wsol.s6; tempsol.s5=wsol.s5; tempsol.s4=wsol.s4;
tempsol.s3=wsol.s3; tempsol.s2=wsol.s2; tempsol.s1=wsol.s1;
tempsol.np=wsol.np;

/* determine gradient direction for number of pallets */
x = wsol.pm;
if ((tempsol.np+pss) > max_pallets(tempsol))
   tempsol.np = max_pallets(tempsol) + 1;
else
   tempsol.np = tempsol.np + pss;
y = set_obj_func_val(tempsol);
g1 = (y - x)/pss;
if (((y - x)/pss) < 0.00)
 htc[1] = 1;     /* set "how to change" to step forward */
else
 htc[1] = -1;    /* set "how to change" to step backward */
}



encode_sc(config)
 int config;
   /* This function encodes the decimal version of the station
      configuration into a binary number.
   */
{
 int temp;
 temp = config;
 if ((temp/32) == 1) { csol.s6 = 1; temp = temp - 32; }
 else  csol.s6 = 0;
 if ((temp/16) == 1) { csol.s5 = 1; temp = temp - 16; }
 else  csol.s5 = 0;
 if ((temp/8) == 1) { csol.s4 = 1; temp = temp - 8; }
 else  csol.s4 = 0;
 if ((temp/4) == 1) { csol.s3 = 1; temp = temp - 4; }
 else  csol.s3 = 0;
 if ((temp/2) == 1) { csol.s2 = 1; temp = temp - 2; }
 else  csol.s2 = 0;
 if ((temp/1) == 1) { csol.s1 = 1; temp = temp - 1; }
 else  csol.s1 = 0;
}
```

```
set_next_solution()
   /* This function makes the changes to the current solution
      vector according to what is in htc[1].
            htc[1] ---> 1 up, -1 down
   */
{
save_csol();  /* copies csol into tsol */

/* set number of pallets variable for next iteration */
if (htc[1] > 0)
  {
   if (max_pallets(csol) < (csol.np+pss))
      csol.np = max_pallets(csol);
   else
      csol.np = csol.np + pss;
  }
else
  {
   if ((csol.np-pss).< 0)
      csol.np = 0;
   else
      csol.np = csol.np - pss;
  }
csol.pm = set_obj_func_val(csol);  /* set pm of new config */
}



save_csol()
  /* This function saves the current solution for statistics purposes. */
{
  tsol.s6 = csol.s6;
  tsol.s5 = csol.s5;
  tsol.s4 = csol.s4;
  tsol.s3 = csol.s3;
  tsol.s2 = csol.s2;
  tsol.s1 = csol.s1;
  tsol.np = csol.np;
  tsol.pm = csol.pm;
}


write_stats()
  /* This function writes out the necessary statistics to report the
     history of the SQG algorithm.
  */
{
  int cfg;
  fprintf(fp1,"-------------------------------------------------------\n");
```

```
 cfg = find_config_num(tsol);
 fprintf(fp1,"Iteration %d\n",iter_num);
 fprintf(fp1,"       old : %d%d%d%d%d%d%d, cfg# = %d, np = %d, pm = %f\n",
          tsol.s6,tsol.s5,tsol.s4,tsol.s3,tsol.s2,tsol.s1,cfg,tsol.np,
          tsol.pm);
 fprintf(fp1,"       htc[1]=%d,g1=%f,pss=%d\n",htc[1],g1,pss);
 cfg = find_config_num(csol);
 fprintf(fp1,"       new : %d%d%d%d%d%d%d, cfg# = %d, np = %d, pm = %f\n\n",
          csol.s6,csol.s5,csol.s4,csol.s3,csol.s2,csol.s1,cfg,csol.np,
          csol.pm);
}


init_rand_gen()
 /* This function initializes the random number generator */
{
int i;
float x;
x = 0.0;
for (i=1; i<=20; i++)
 x = x + r1num();
}


int find_config_num(wsol)
 struct solution wsol;
{
int cfg_num;
cfg_num = (((wsol.s6)*32)+((wsol.s5)*16)+((wsol.s4)*8)+((wsol.s3)*4)+
          ((wsol.s2)*2)+(wsol.s1));
return cfg_num;
}



float set_obj_func_val(wsol)
 struct solution wsol;
  /* This function sets the objective function value.  It references
     a data file containing the parameters so a normal random variate
     can be generated.
  */
{
FILE *fpin;
float mean,sdev,oprate,hrs_per_shift;
float prod_rate,sum;
int config,pals,iconl,i,mp,cfgnum,num_ops;

oprate = 15.00;
hrs_per_shift = 8.0;

mp = max_pallets(wsol);
```

```
num_ops = wsol.s1+wsol.s2+wsol.s3+wsol.s4+wsol.s5+wsol.s6;
if ((wsol.np > mp)||(wsol.np == 0)||(num_ops == 0))
  {
   legal = 0;       /* illegal or infeasible number of pallets */
   prod_rate = 1;       /* penalty production rate */
  }
else
  legal = 1;
if (legal)
  {
   if ((fpin=fopen("simnorm.dat","r"))==NULL)
     {
          printf("Cannot open SIMNORM.DAT file !!\n\n");
          exit(1);
     }
   cfgnum = find_config_num(wsol);
   fscanf(fpin,"%f%f%d%d\n",&mean,&sdev,&config,&pals);
   while((config != cfgnum)||(pals != wsol.np))
     fscanf(fpin,"%f%f%d%d\n",&mean,&sdev,&config,&pals);
   fclose(fpin);
   /* create a normal random variate */
   sum = -6.0;
   for (i=1; i<=12; i++)
     sum = sum + r1num();
   prod_rate = (sum * sdev) + mean;
  }
if ((num_ops == 0)||(wsol.np == 0))
  {
   num_ops = 10;   /* big penalty for no operator case */
   prod_rate = 1;   /* another addition to the penalty */
  }
pm = (oprate * hrs_per_shift * num_ops)/prod_rate;
return pm;
}


float set_obj_func_val2(wsol)
 struct solution wsol;
   /* This function sets the objective function value.  It references
      a data file containing the parameters so a normal random variate
      can be generated.  This returns the performance measure with
      the pm2 definition.
   */
{
FILE *fpin;
float mean,sdev,oprate,hrs_per_shift;
float prod_rate,sum,rr,penalty;
int config,pals,legal,i,mp,cfgnum,num_ops;

oprate = 15.00;
hrs_per_shift = 8.0;
```

```
mp = max_pallets(wsol);
num_ops = wsol.s1+wsol.s2+wsol.s3+wsol.s4+wsol.s5+wsol.s6;
if ((wsol.np > mp)||(wsol.np == 0)||(num_ops == 0))
  {
   legal = 0;       /* illegal or infeasible number of pallets */
   prod_rate = 1;      /* penalty production rate */
  }
else
  legal = 1;
if (legal)
  {
   if ((fpin=fopen("simnorm.dat","r"))==NULL)
     {
          printf("Cannot open SIMNORM.DAT file !!\n\n");
          exit(1);
     }
   cfgnum = find_config_num(wsol);
   fscanf(fpin,"%f%f%d%d\n",&mean,&sdev,&config,&pals);
   while((config != cfgnum)||(pals != wsol.np))
     fscanf(fpin,"%f%f%d%d\n",&mean,&sdev,&config,&pals);
   fclose(fpin);
   /* create a normal random variate */
   sum = -6.0;
   for (i=1; i<=12; i++)
    sum = sum + r1num();
   prod_rate = (sum * sdev) + mean;
  }
if ((num_ops == 0)||(wsol.np == 0))
  {
   num_ops = 10;  /* big penalty for no operator case */
   prod_rate = 1;  /* another addition to the penalty */
  }
rr = prod_rate/req;    /* set requirement ratio */
if (rr<=1.0)  penalty = (upc1 * (req - prod_rate))/prod_rate;
else  penalty = (upc2 * (prod_rate - req))/prod_rate;
pm = (oprate * hrs_per_shift * num_ops)/prod_rate;
pm = pm + penalty;
return pm;
}


float set_obj_func_val3(wsol)
 struct solution wsol;
   /* This function sets the objective function value.  It references
      a data file containing the average production rate.  This returns
      the performance measure with the pm2 definition.
   */
{
FILE *fpin;
float oprate,hrs_per_shift;
```

```
float prod_rate,rr,penalty;
int config,pals,legal,i,mp,cfgnum,num_ops;

oprate = 15.00;
hrs_per_shift = 8.0;

mp = max_pallets(wsol);
num_ops = wsol.s1+wsol.s2+wsol.s3+wsol.s4+wsol.s5+wsol.s6;
if ((wsol.np > mp)||(wsol.np == 0)||(num_ops == 0))
 {
  legal = 0;       /* illegal or infeasible number of pallets */
  prod_rate = 1;      /* penalty production rate */
 }
else
 legal = 1;
if (legal)
 {
  if ((fpin=fopen("simavg.dat","r"))==NULL)
    {
        printf("Cannot open SIMAVG.DAT file !!\n\n");
        exit(1);
    }
  cfgnum = find_config_num(wsol);
  fscanf(fpin,"%f%d%d\n",&prod_rate,&config,&pals);
  while((config != cfgnum)||(pals != wsol.np))
    fscanf(fpin,"%f%d%d\n",&prod_rate,&config,&pals);
  fclose(fpin);
 }
if ((num_ops == 0)||(wsol.np == 0))
 {
  num_ops = 10;  /* big penalty for no operator case */
  prod_rate = 1;  /* another addition to the penalty */
 }
rr = prod_rate/req;   /* set requirement ratio */
if (rr<=1.0)  penalty = (upc1 * (req - prod_rate))/prod_rate;
else  penalty = (upc2 * (prod_rate - req))/prod_rate;
pm = (oprate * hrs_per_shift * num_ops)/prod_rate;
pm = pm + penalty;
return pm;
}



int max_pallets(wsol)
 struct solution wsol;
  /* This function calculates the maximum number of pallets allowed
     for a given configuration number.  The absolute maximum number
     of pallets is 31.
  */
{
int x1,x2,i,num_sta_op,laststa,maxpals;
```

```
num_sta_op = wsol.s6 + wsol.s5 + wsol.s4 + wsol.s3 + wsol.s2 + wsol.s1;
laststa = 0;
laststa = wsol.s1 == 1 ? 1 : laststa;
laststa = wsol.s2 == 1 ? 2 : laststa;
laststa = wsol.s3 == 1 ? 3 : laststa;
laststa = wsol.s4 == 1 ? 4 : laststa;
laststa = wsol.s5 == 1 ? 5 : laststa;
laststa = wsol.s6 == 1 ? 6 : laststa;
x2 = 10;
switch(laststa)
 {
  case 6:   x1 = 10;
            break;
  case 5:   x1 = 8;
            break;
  case 4:   x1 = 7;
            break;
  case 3:   x1 = 6;
            break;
  case 2:   x1 = 4;
            break;
  case 1:   x1 = 2;
            break;
  case 0:   x1 = 1;
            break;
 }
maxpals = (2*num_sta_op) + x1 + x2;
if (maxpals == 32) maxpals = 31;
return maxpals;
}

modify_pss()
 /* This function modifies the pallet step size */
{
 float mpss;
 mpss = (rp * pss);
 if (mpss < 1.00) mpss=mpss+1.0;
 pss = (int)mpss;     /* truncates the float to an int */
}
sort_by_pm()
 /* This function sorts pm_array[][] in ascending order
    by performance measure (pm).  A simple bubble sort
    algorithm is used to sort pm_array[][].  */
{
 int done,swaps,i;
 float temp_sc,temp_np,temp_pm;

 done = 0;  /* not done sorting yet */
 while (done != 1)
  {
```

```
    swaps = 0;
    for (i=1; i<=(popsize - 1); i++)
        {
         if (pm_array[i+1][3] < pm_array[i][3])
           {
            swaps++;
            temp_sc = pm_array[i][1];
            temp_np = pm_array[i][2];
            temp_pm = pm_array[i][3];
            pm_array[i][1] = pm_array[i+1][1];
            pm_array[i][2] = pm_array[i+1][2];
            pm_array[i][3] = pm_array[i+1][3];
            pm_array[i+1][1] = temp_sc;
            pm_array[i+1][2] = temp_np;
            pm_array[i+1][3] = temp_pm;
           }
        }
    if (swaps == 0)  done = 1;
   }
}

output_pm_array()
  /* This function outputs the pm_array[][] contents into
     TAOUT2.DAT. Note that the contents of pm_array[][]
     have been sorted.  */
{
 FILE *fpo2;

 int i;
 float sumfit;

 if ((fpo2=fopen("taout2.dat","w"))==NULL)
   {
    printf("Cannot open TAOUT2.DAT file !!\n\n");
    exit(1);
   }
 sumfit = 0.0;
 /*for (i=1; i<=popsize; i++)*/
 for (i=1; i<=20; i++)
   {
    fprintf(fpo2,"sc = %3.0f,  np = %3.0f,  pm = %10.6f\n",
                    pm_array[i][1],pm_array[i][2],pm_array[i][3]);
    sumfit = sumfit + pm_array[i][3];
   }
 fprintf(fpo2,"sum of top 20 PMs = %15.6f\n",sumfit);
 fclose(fpo2);
}


float r1num()     /* random number generator */
{
```

```
 float ran_num;
 seed1 = (seed1*2125) + 1;
 if (seed1 < 0)
   seed1 = seed1 + 2147483647 + 1;
 ran_num = seed1/2147483647.0;
 return ran_num;
}
```