A Genetic Algorithm For
Robust Simulation Optimization

A thesis presented to

The Faculty of the

Fritz J. and Dolores H. Russ
College of Engineering and Technology
Ohio University

In Partial Fulfillment
of the Requirement for the Degree
Master of Science

by

Steven C. Harris
June, 1996
# Table of Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Approval Page</td>
<td>ii</td>
</tr>
<tr>
<td>Table of Contents</td>
<td>iii</td>
</tr>
<tr>
<td>List of Tables</td>
<td>viii</td>
</tr>
<tr>
<td>List of Figures</td>
<td>xii</td>
</tr>
<tr>
<td><strong>Chapter</strong></td>
<td></td>
</tr>
<tr>
<td>1. Introduction</td>
<td>1</td>
</tr>
<tr>
<td>2. Literature Review</td>
<td>5</td>
</tr>
<tr>
<td>2.1 Simulation</td>
<td>5</td>
</tr>
<tr>
<td>2.2 Optimization Techniques Applied to Simulated Systems</td>
<td>6</td>
</tr>
<tr>
<td>2.2.1 Considerations for Optimization of Simulated Systems</td>
<td>6</td>
</tr>
<tr>
<td>2.2.2 Applications of Optimization Techniques to Simulation</td>
<td>7</td>
</tr>
<tr>
<td>2.2.3 Classification of Optimization Techniques</td>
<td>9</td>
</tr>
<tr>
<td>2.2.3.1 Stochastic Techniques</td>
<td>9</td>
</tr>
<tr>
<td>2.2.3.2 Local Techniques</td>
<td>9</td>
</tr>
<tr>
<td>2.2.3.3 Global Techniques</td>
<td>10</td>
</tr>
<tr>
<td>2.3 Simulated Annealing</td>
<td>11</td>
</tr>
<tr>
<td>2.4 Genetic Algorithms</td>
<td>12</td>
</tr>
<tr>
<td>2.4.1 Genetic Algorithm Representations</td>
<td>15</td>
</tr>
<tr>
<td>2.4.2 Creating the Initial Population</td>
<td>18</td>
</tr>
<tr>
<td>2.4.3 Population Size</td>
<td>19</td>
</tr>
</tbody>
</table>
2.4.4 Population Replacement ........................................ 20
2.4.5 Selection Strategies ............................................. 20
2.4.6 Scaling Evaluations .............................................. 22
2.4.7 Crossover ......................................................... 22
2.4.8 Mutation ......................................................... 24
2.4.9 Stopping Criteria and Convergence Measure .................. 25
2.4.10 Increasing Replications ......................................... 26
2.4.11 Constraints ....................................................... 27

3. Design of a Generic Simulation Genetic Algorithm .............. 29
   3.1 Genetic Algorithm Design from Literature ..................... 29
   3.2 Genetic Algorithm Design by Experiment .................... 33

4. Experiments .......................................................... 35
   4.1 Genetic Algorithm Implementation and Simulation Interface .. 35
   4.2 Test Problems .................................................... 42
      4.2.1 Mathematical Equation Problem .......................... 42
      4.2.2 Restaurant Problem ........................................ 44
      4.2.3 Distribution Problem ...................................... 46
      4.2.4 Standard Buffer Problem .................................. 51
      4.2.5 Warehouse Storage and Retrieval Problem ............... 53
      4.2.6 Job-shop Problem ......................................... 56
   4.3 Genetic Operator Rate Experiment ............................ 68
A.2 Simulation Genetic Algorithm Initialization Program (GAINIT.CPP) . 122
A.3 Simulation Simulated Annealing Algorithm Program (ANJS.CPP) . 127

B. Basic Test Problems in Coded for SIMAN ......................... 133

B.1.1 Restaurant Problem Model File ............................... 133
B.1.2 Restaurant Problem Experiment File ......................... 135
B.2.1 Distribution Problem 1 Model File ............................. 136
B.2.2 Distribution Problem 1 Experiment File ..................... 138
B.3.1 Distribution Problem 6 Model File ............................. 140
B.3.2 Distribution Problem 6 Experiment File ..................... 142
B.4.1 Standard Buffer Problem Model File .......................... 144
B.4.2 Standard Buffer Problem Experiment File ................... 147
B.5.1 Warehouse Storage and Retrieval Problem 1 Model File ..... 149
B.5.2 Warehouse Storage and Retrieval Problem 1 Experiment File .... 154
B.5.3 ASRS.EXE for use with AS/RS Problems (ASRS.C) .......... 156
B.6.1 Job-shop Problem 1 Model File ............................... 167
B.6.2 Job-shop Problem 1 Experiment File ......................... 171
B.7.1 Job-shop Problem 8 Model File ............................... 173
B.7.2 Job-shop Problem 8 Experiment File ......................... 176
B.8.1 Job-shop Problem 11 Model File .............................. 178
B.8.2 Job-shop Problem 11 Experiment File ....................... 182

C. Results of Experiments ............................................. 184
D. Results of Comparison

Abstract
<table>
<thead>
<tr>
<th>Table</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.1 Mathematical equation problem 1</td>
<td>44</td>
</tr>
<tr>
<td>4.2 Mathematical equation problem 2</td>
<td>44</td>
</tr>
<tr>
<td>4.3 Restaurant problem description</td>
<td>45</td>
</tr>
<tr>
<td>4.4 Restaurant problem optimization parameters</td>
<td>45</td>
</tr>
<tr>
<td>4.5 Distribution problem 1 description</td>
<td>47</td>
</tr>
<tr>
<td>4.6 Distribution optimization parameters (problems 1,3-5)</td>
<td>47</td>
</tr>
<tr>
<td>4.7 Distribution problem 2 description</td>
<td>48</td>
</tr>
<tr>
<td>4.8 Distribution optimization parameters (problem 2)</td>
<td>48</td>
</tr>
<tr>
<td>4.9 Distribution problem 3 description</td>
<td>48</td>
</tr>
<tr>
<td>4.10 Distribution problem 4 description</td>
<td>49</td>
</tr>
<tr>
<td>4.11 Distribution problem 5 description</td>
<td>49</td>
</tr>
<tr>
<td>4.12 Distribution problem 6 description</td>
<td>50</td>
</tr>
<tr>
<td>4.13 Distribution optimization parameters (problem 6)</td>
<td>51</td>
</tr>
<tr>
<td>4.14 Standard buffer problem description</td>
<td>52</td>
</tr>
<tr>
<td>4.15 Standard buffer problem optimization parameters</td>
<td>52</td>
</tr>
<tr>
<td>4.16 Standard buffer problem constraint</td>
<td>52</td>
</tr>
<tr>
<td>4.17 AS/RS problem requirements</td>
<td>53</td>
</tr>
<tr>
<td>4.18 Representation of rack dimension variables</td>
<td>54</td>
</tr>
<tr>
<td>4.19 AS/RS problem optimization parameters</td>
<td>55</td>
</tr>
</tbody>
</table>
4.20 AS/RS problem optimization representation ........................................... 55
4.21 Job-shop problem operation sequences. .................................................. 57
4.22 Job-shop problem distance between stations. .......................................... 57
4.23 Maximum cutting speeds. ......................................................................... 58
4.24 Revenue from sales. .................................................................................. 59
4.25 Costs for job-shop (problem 1). ................................................................. 59
4.26 Surface feet removed for each operation (problems 1-4). ....................... 60
4.27 Scheduling rules for input buffers. ............................................................. 61
4.28 Job-shop optimization parameters (problems 1-7). .................................. 61
4.29 Constraints for job-shop (problems 1-7). .................................................. 62
4.30 Costs (problem 2). ................................................................................... 62
4.31 Costs (problems 3-4). .............................................................................. 63
4.32 Revenue (problems 4&7). ........................................................................ 63
4.33 Costs (problem 5). ................................................................................... 64
4.34 Surface feet removed for each operation (problems 5-7). ....................... 64
4.35 Costs for job-shop (problems 6&7). .......................................................... 65
4.36 Costs (problems 8&9). .............................................................................. 65
4.37 Job-shop optimization parameters (problems 8-10). ............................... 66
4.38 Constraints for job-shop (problems 8-11). ................................................. 66
4.39 Costs (problem 9). ................................................................................... 66
4.40 Revenue (problem 9). .............................................................................. 67
D.6 Results of job-shop problems 5 - 7. ......................... 191

D.7 Results of job-shop problems 8 - 11. .......................... 192
List of Figures

Figure

4.1 Graph of one dimension of mathematical equation problem. ............... 43
4.2 Distribution problem 1. .................................................. 46
4.3 Distribution problem 6. .................................................. 50
4.4 Standard buffer problem. ................................................ 51
4.5 Crossover rate. ......................................................... 72
4.6 Creep rate. ......................................................... 72
4.7 Jump rate. ......................................................... 72
5.1 Algorithms' solution ratings. ........................................... 83
5.2 Comparison of algorithms on various problem sizes. ....................... 85
5.3 Genetic algorithm's percent difference from simulated annealing. ...... 87
6.1 Portion of genetic algorithm's optimization of distribution problem 4. ... 92
Chapter 1
Introduction

Discrete-event simulation is a very powerful and is widely used tool for system analysis and design. Simulation basically models a real-life system as stochastic simulated events in time using a computer for all of the necessary calculations. These systems can come from many different fields, including computer systems, material-handling systems, automated production and storage facilities, military tactics, and economic/social systems (Pegden, Shannon, and Sadowski 1990). From a set of system inputs, simulation produces an estimate of the output of the system. Simulation does not contain in itself the ability to search for the set of inputs that would yield the optimal system output.

Therefore, to optimize a simulated system, two components are required: the simulation of the system and an optimization procedure. This optimization procedure can be automated in the form of a computer program that selects system inputs based on a specified optimization technique and then runs the simulation using these inputs to produce an estimate of the system output. This output is then fed back into the optimization program where it is used by the optimization technique to select further system inputs.

The goal of this research is to find a robust, effective optimization technique for simulation. Robustness refers to the technique's capacity to function on a wide variety of simulation problems without the modification of optimization parameters. This property would allow for use by non-experts on potentially any problem. Effectiveness refers to the technique's ability to find optimal or near-optimal solutions
in the least amount of time. These two criteria will be used to judge the algorithm’s feasibility.

Such an automated optimization tool for simulation would have great benefit for system design. Utilizing the computational power and speed of a computer, the optimization tool would allow for an in-depth search of the solution space in a reasonable amount of time. Also, the system designer would be allowed to focus on developing an accurate model of the system and realistic system constraints without having to also be an expert in optimization.

Many different optimization techniques exist in the literature. One of these techniques is called a genetic algorithm. Genetic algorithms imitate biological natural selection by maintaining a "population" of solutions where better solutions are favored for the "reproduction" of new solutions. These new solutions are created by applying "genetic" operators to selected existing solutions. Chapter 2 discusses optimization techniques including genetic algorithms and why genetic algorithms seem well-suited for use in simulation optimization.

The purpose of this thesis was to investigate the feasibility of using a genetic algorithm as a robust simulation optimization tool. A steady-state genetic algorithm was developed from literature and experimentation to be used as a robust simulation optimization tool. The algorithm uses real-encoded variables to represent a population of fifty non-identical solutions. Tournament selection, one-point real crossover, a creep operator, and jump-mutation are used to create one new solution per iteration.
The new solution replaces the worst solution in the population if the new solution is better than the worst.

In the literature, genetic algorithms are often fine-tuned to work well on a specific problem. In this thesis, a genetic algorithm which seemed well-suited for simulation optimization was intentionally not fine-tuned for each simulation problem so that the robustness of the algorithm could be tested. To measure the algorithm's robustness, it was applied to twenty simulation problems of different sizes and characteristics. The algorithm successfully functioned on each of the problems. To measure the effectiveness of this algorithm, another optimization technique called simulated annealing was also applied to the same test problems. The genetic algorithm found significantly better solutions than simulated annealing on nine test problems and insignificantly different solutions than simulated annealing on eight problems. On the three problems where simulated annealing's solutions were significantly better than the genetic algorithm's solutions, the genetic algorithm averaged within 9.8% of simulated annealing's solutions. Also, the genetic algorithm found as good of a solution as a gradient technique applied to a benchmark problem taken from literature (Ho, Eyler, and Chien 1979). Overall, the genetic algorithm proved to be a promising candidate for a robust optimization tool worthy of further research.

Chapter 2 presents the pertinent information from literature on simulation optimization, various types of optimization techniques, simulated annealing, and genetic algorithms. Chapter 3 develops a basic design for the simulation genetic algorithm from suggestions and examples found in the literature. Chapter 4 discusses how the genetic algorithm will interface with the simulator, explains the details of the
test problems, and presents the design and results of the two experiments used to finalize the genetic algorithm's design. Chapter 5 presents the details of the simulated annealing algorithm used and the comparison of it with the genetic algorithm as applied to the test problems. Chapter 6 draws conclusions from the comparisons and suggests areas of future research.
Chapter 2
Literature Review

2.1 Simulation

"Simulation is one of the most powerful analysis tools available to those responsible for the design and operation of complex processes or systems" (Pegden, Shannon, and Sadowski 1990). Discrete-event simulation basically involves modeling a process or system, usually with a computer program, so that the model mimics the actual system (Schriber 1987). The simulation keeps track of a list of stochastic simulated events and collects statistics on these events. These statistics provide the output of the simulation which attempts to predict the behavior of the actual system. From this property comes a major benefit of simulation: a model of a system can be easily and inexpensively modified and tested to predict the result of the same modification to the real system. For this reason, simulation is said to be descriptive in that it "describes" the behavior of a system or process. Simulation is especially useful for modeling complex systems which would be either very difficult if not impossible to model analytically.

Simulation experiments are conducted for either one of two reasons (Law and Kelton 1982). One reason may be to simply determine the output of a given system configuration. The alternate reason is to design a system by selecting system inputs that optimize the system's outputs. Obviously, the first reason is readily available with simulation because simulation is by nature descriptive. Each simulation run describes the performance of that particular system configuration. Therefore, the second alternative, optimization of system inputs, is impossible for simulation alone. If simulation is to be used to optimize a system, it must be coupled with some type of
optimization technique which will coordinate the search of the system's configuration space. The optimization technique will select the system inputs, conduct a simulation run with those inputs, and then use the simulation outputs to further direct the search.

2.2 Optimization Techniques Applied to Simulated Systems

To date, optimization in simulation has been widely practiced using various optimization strategies. These applications of optimization techniques have addressed only specific problems. Simulation optimization is not widely used because no standard, efficient method has been recommended by researchers. The need exists for research to suggest an effective and robust simulation optimization technique for simulation users who need to optimize but are not optimization experts. This thesis seeks to determine the feasibility of an optimization technique called a genetic algorithm as a robust simulation optimization tool. It then will provide a comparison of this technique to a similar technique called simulated annealing.

2.2.1 Considerations for Optimization of Simulated Systems

For a technique to be effective for simulation optimization, it must take into account several important issues relevant to simulation. First, modeling a real system assumes that there are realistic constraints on the system's input parameters. Therefore, the optimization technique must be capable of handling variable bounds and constraints. Also, these system variables may be integers, real values, or non-numeric. The solution technique should at least be able to approximate these different variable types if it is not able to handle them directly. Also, outputs from simulations are usually stochastic, so the technique must take this into consideration by running multiple replications at each variable configuration to improve estimates on simulation output.
Finally, since simulation replications are time consuming, an effective optimization technique should strive to run as few simulation replications as possible without sacrificing too much solution quality.

2.2.2 Applications of Optimization Techniques to Simulation

Various applications of optimization techniques to simulation have been conducted in industry. Some of these applications include buffer sizing (Ho and Cassandras 1983), maintenance logistics (Gecan and Chrissis 1985), inventory control (Bengu and Haddock 1986), furnace optimization (Lefrancois, L'Esperance, and Turmel 1991), and manufacturing capacity (Mansuri and Czajkiewicz 1991). These published cases show the feasibility of various optimization techniques in conjunction with simulation, but little work has been published applying genetic algorithms to simulation optimization.

A genetic algorithm was developed to handle a variety of variable types for application to the optimization of simulated manufacturing systems (Tautou and Pierreval 1995). This algorithm was applied to a deterministic simulation as an example problem with successful results. Specific issues in this paper will be discussed in greater depth in Section 2.4 on genetic algorithms.

An as of yet unpublished paper presented at an ORSA/TIMS conference in Alaska discussed the use of a genetic algorithm in conjunction with a simulation (Mollaghasemi and Pet-Edwards 1994). The genetic algorithm was used to optimize weights of scheduling heuristics for simulated project and machine scheduling problems.
One paper compares a classical genetic algorithm to a Hooke-Jeeves pattern search and a response surface method search (Yunker and Tew 1994). One simulation problem of a time shared computer system consisting of two real-valued decision variables was used for the comparison. The genetic algorithm found much better solutions over 50 searches than the other two algorithms.

Another paper compares a classical genetic algorithm to a random search over the same number of simulation evaluations (Azadivar and Tompkins 1994). Two slightly different algorithms gave similar results on two simple manufacturing process simulations. The first problem consisted of five machines in series. All five non-numeric decision variables had four alternatives consisting of different machines of various capacity and reliability. The second problem was the first problem expanded to eleven machines in series. The algorithm used very small population sizes and performed a very limited number of simulation evaluations. The conclusions reported the genetic algorithm significantly out-performed the random search only on the larger problem.

These papers begin to show the feasibility of genetic algorithms for use in simulation optimization, but more research is required to support these claims of feasibility. No research has been published which examines the robustness of a simulation-specific genetic algorithm by testing such an algorithm on a wide variety of stochastic simulation problems.
2.2.3 Classification of Optimization Techniques

Optimization techniques can be classified as local or global, depending on the scope of their search of the solution space. These two categories contain stochastic approaches which explicitly address the stochastic nature of simulation output. These make up three basic categories:

1. Stochastic techniques
2. Local techniques
3. Global techniques

2.2.3.1 Stochastic Techniques

Stochastic techniques are based on response surface methodologies (Biles 1974, Smith 1973). They use fractional factorial experiments to determine the improving direction at each iteration. Another stochastic approach is stochastic approximation (Glynn 1986, Gaivoronski 1993). These techniques applied to simulation optimization are quite slow in reaching an optimal solution because of the large number of replications that are required at each iteration to perform the factorial experiments. This thesis ignores stochastic techniques, assuming that other global techniques will provide solutions of comparable quality in less time.

2.2.3.2 Local Techniques

Non-linear programming techniques for finding optimum values have been applied to simulation output and make up most of the local optimization techniques. These include one-at-a-time direct search (Gecan and Chrissis 1985), Hooke-Jeeves pattern search (Bengu and Haddock 1986; Nandkeolyar and Christy 1989; Lefrancois,

These local techniques basically search around a current best solution to find a better solution. This risks finding a solution which is locally optimal but which is not the global optimum. Also, some of the local techniques require that a derivative of the objective function be estimated. For these reasons, this thesis ignores local techniques preferring to investigate more effective global techniques.

2.2.3.3 Global Techniques

Global techniques are relatively new and were created to overcome the weakness of local techniques getting stuck in local optima. These techniques use different mechanisms for avoiding local optima. In general, this is done by providing a stochastic element in the search procedure.

Genetic algorithms, simulated annealing, and Bayesian sampling are three global techniques. Genetic algorithms and simulated annealing algorithms generally produce similar results in non-simulation optimization situations (Stuckman, Evans, and Mollahsemi 1991). For this reason, this thesis will evaluate the genetic algorithm designed for simulation optimization by comparing it with a simulated annealing algorithm applied to the same test simulations. Genetic algorithms and simulated annealing will therefore be discussed in greater depths in the following sections. Bayesian sampling deals directly with the stochastic output of simulation but will not be considered because it requires large numbers of simulation runs.
2.3 Simulated Annealing

Simulated annealing was introduced by Kirkpatrick et al. (1983) and is analogous to the annealing of solids. As in the annealing of metals, simulated annealing also uses controlled "cooling" operations on nonphysical optimization variables to produce an optimized solution from poorer solutions. This is done through the iterative improvement of current solutions similar to other optimization techniques. But unlike other techniques, simulated annealing can occasionally accept a less desirable point as the new current solution with a specified probability. This provides the search with a chance of escaping local optima by "jumping" from a less-promising region to one that is more likely to contain the global optimum. The basic simulated annealing algorithm is shown in the following steps:

**Step 1:** Initialization. Randomly select an initial set of variables \( X_C \) representing a point in the solution space. This point is then evaluated, \( f(X_C) \).

**Step 2:** Random moves. A new set of parameters \( X_A \) is selected from the neighborhood of \( X_C \). The neighborhood consists of all solutions within some determined distance from \( X_C \). \( X_A \) cannot be identical to \( X_C \). This new point is evaluated \( f(X_A) \). The difference between the evaluations of these two points \( \Delta E \) is evaluated as:

\[
\Delta E = f(X_A) - f(X_C). \quad \text{(for minimization)}
\]

If the new point is better than the current solution, then \( X_A \) replaces \( X_C \) as the new current solution. If the new point is worse than the current solution, then \( X_A \) replaces \( X_C \) as the new current solution with the probability of \( e^{-\Delta E/T} \), where \( T \) is the current "temperature" of the annealing process. The initial value of this variable is
predetermined. The current temperature $T_{NOW}$ remains unchanged for $M$ iterations and is then decreased at the cooling rate $\alpha$. After $M$ iterations, the new value for $T_{NOW}$ is calculated as:

$$T_{NOW} = \alpha T_{NOW} \quad \text{(where } 0 < \alpha < 1)$$

Both $M$ and $\alpha$ are predetermined. This gradual decrease in temperature gradually decreases the probability that a worse solution will be accepted. As the algorithm nears its final temperature, the algorithm is less likely to jump into a neighboring region and more likely to settle into the local optimum, which by this point hopefully is the global optimum.

**Step 3: Termination.** The algorithm stops when the current temperature goes below a predetermined final temperature. The final current solution is reported as the optimal solution.

### 2.4 Genetic Algorithms

Developed by John Holland (1975), genetic algorithms are general purpose, adaptive global search techniques whose creation was inspired by biological evolution. They are based on Darwin's theory survival of the fittest (Hajela 1990). Individuals in a population of a particular species which are more fit to survive in their current environment have a greater likelihood of reproducing. Over generations, weaker individuals tend to die out and stronger individuals flourish, increasing the overall fitness of the population. This same notion is used by genetic algorithms to find solutions to problems (Davis 1987).
A genetic algorithm maintains a "population" of candidate solutions to a particular problem (Grefenstette 1986). Each candidate solution is considered an individual in the population and is traditionally an encoding of the decision variables (Davis 1987). Individuals are evaluated for their "fitness". For simulation optimization, this would be a single performance measure, namely the output of the simulation. The dependence of genetic algorithms on only a single numeric performance measure makes it a very robust search technique. For example, different requirements for different applications can be programmed into the evaluation function. This allows the genetic algorithm to be left unchanged (Davis 1987). A population's overall fitness is improved by using simple, stochastic "genetic operators". These operators reproduce new individuals by "mating" existing individuals and/or applying stochastic modifications to these new individuals. By favoring individuals with higher performance measures for reproduction, the overall fitness of the population is improved over time. After a number of generations, the genetic algorithm will tend to converge at the global optimum or at least a near optimal solution (Davis 1987). For a more in-depth look at the theory behind genetic algorithms, see Genetic Algorithms in Search, Optimization, and Machine Learning (Goldberg 1989).

The basic properties of genetic algorithms give them several advantages. First, it is simple to develop and code a genetic algorithm that works. The challenge lies in making the genetic algorithm as efficient as possible. Second, they make no assumptions about the problem space (Whitley, Starkweather, and Bogart 1990). For example, a genetic algorithm does not require prior knowledge of the continuity, convexity, or any derivatives of the solution surface since it works solely from the performance measures. The genetic algorithm is only concerned with the value of the
performance measure, not how an individual's variable values are turned into that performance measure (Ackley 1987). Using a computer simulation to determine fitness measures is therefore a fairly straight-forward endeavor. Finally, the very nature of genetic algorithms give them the ability to handle a variety of variable types. Unlike other optimization techniques, genetic algorithms make no assumptions about the continuity of the decision variables making them especially adept at handling integer and non-numeric variables.

The basic genetic algorithm which Holland developed is often referred to as a "classical genetic algorithm." From that basic form, an unlimited number of variations can be developed by modifying one or many of the classical genetic algorithm's characteristics. These many characteristics will be discussed later in attempts to develop a genetic algorithm well-suited for simulation optimization. The following is a list of the basic steps of a classical genetic algorithm:

1. Create an initial population of binary strings.
2. Evaluate all members of the population.
3. Stochastically select members for the next generation favoring the more fit.
4. Pair those selected and apply crossover at a given rate.
5. Mutate bits at a given rate.
6. Repeat steps 2-5 until convergence or stopping rule is reached.

From the basic steps of the classical genetic algorithm, it is fairly straight-forward to extract the genetic algorithm design parameters which must be determined. Genetic algorithms do not have to be coded into binary strings, so the internal representation or parameter encoding must be determined. A method for creating the initial population
must also be selected (Davis 1987). In genetic algorithms, population size is normally kept constant, so this size must be determined. In classical genetic algorithms, each generation replaces all individuals from the previous generation with the new individuals created. This full replacement is not necessary; so the type of replacement, whether full or partial must be established. The fitness evaluations may need to be scaled to create a more efficient genetic algorithm, so a scaling strategy might be necessary. Genetic operators need to be selected that are consistent with the parameter encoding scheme. These include a selection strategy to decide which individuals will reproduce to create new individuals for the next generation, a type of crossover and mutation, and the probabilities of performing these operations (AI Expert 1990). A stopping criteria must be established to end the iterating of the genetic algorithm. This stopping can either reflect a convergence amongst the population or an upper time limit. A way of limiting the genetic search to reflect real-life constraints and variable bounds must be included. Finally, since the genetic algorithm will be applied to stochastic computer simulations, an effective method must be determined for increasing the number of simulation replications per individual fitness evaluation. Each of these will be discussed in more depth in the following sections to direct the design of a genetic algorithm that will work well for simulation optimization.

2.4.1 Genetic Algorithm Representations

A genetic algorithm representation (or encoding) must be able to represent every possible combination of decision parameters for the problem (Pham and Yang 1993). In Artificial Intelligence terminology, this is called the "internal representation" of the external information. There are many different ways to encode parameters into a genetic algorithm. Nygard recognized that many different representation schemes
have been tried, from the most common encoding into binary strings, to lists of integers or lists of real values, lists of rules, or a combination of any of the above (Nygard, Ficek, and Sharda 1992). These seem to break into two basic strategies, abstract and real encoding. The abstract representation is most often a binary string which can represent any kind of data type using the right encoder and decoder. The greater the length of the binary string, the finer-grained the search becomes. In a real-encoding representation, basically the genetic algorithm works on the actual parameter values. This can avoid the encoding and decoding process, but as it will be shown, this does not guarantee that a real-encoding scheme is better. Whichever representation is chosen, it will dramatically affect the rest of the genetic algorithm design. The advantages and disadvantages of both encodings will now be discussed.

The major advantage to using a binary representation such has been proposed in the classical genetic algorithm design is its strong theoretical basis (Michalewicz 1992). Much study has been done on the structure of binary strings as they progress through the generations that can prove how the genetic operators cause them to converge. Genetic operators on other representations have not been studied as thoroughly (Davis 1987). The study of similarity subsets in the strings is called schemata. A binary string can represent a larger number of these schemata than can a real encoding (Goldberg 1991). These genetic operators have a certain elegance about them that makes them intuitively attractive (Michalewicz 1992). Overall, there is some evidence that supports that a binary encoding is optimal (AI Expert 1990).

Unfortunately, there are some disadvantages that also must be recognized when considering using a binary representation. In a binary encoding, usually the entire
combination of parameters is encoded into one long string of 1's and 0's. In multi-
dimensional searches, these binary strings can become very large (Michalewicz 1992).
The genetic algorithm requires more memory to store the strings and slows the actual
processing of the strings. This large length of binary strings might be the reason that a
binary encoding might tend to not work as well as a real representation on larger
problems (Whitley, Starkweather, and Bogart. 1990). Another problem is called
hamming cliffs (Goldberg 1991). In binary encodings, two points that are close
together as real parameters may not be close together in the encoding scheme. For
example, using binary strings to encode integers, a "0111" string represents a seven
and a "1000" string represents an eight. In real space, the distance of these two
parameters is relatively close. But in hamming space, the distance is 4 which is the
same hamming distance as between 0 and 15. This characteristic may allow the
genetic algorithm to be "deceived" and kept from reaching an optimal solution near a
hamming cliff. Finally, binary encoded genetic algorithms are more difficult to
program.

There are a few advantages of using a real representation instead of a binary one.
Using this encoding, the dimensionality of the problem is reduced from what it would
be in binary form, and this fact reduces the opportunity for the genetic algorithm to be
"deceived." Since one binary string encodes all the parameters, there is a good chance
that a crossover will disrupt the parameters in a way that does not represent a logical
adjusting of parameters. In real encoding, this problem is eliminated. There is a one-
to-one correspondence between the real parameter and the parameter used in the
genetic algorithm (Goldberg 1991). This is intuitively more straightforward. Real
encodings are, in general, more precise than binary representations. They are also
better at handling parameter bounds and non-trivial constraints (Michalewicz 1992). Real genetic algorithms seem to converge more quickly than binary (Goldberg 1991). Simple to program and efficient genetic operators may be used. On some tests that have been conducted, real genetic algorithms seemed to perform better than binary encodings (Michalewicz 1992). A genetic algorithm using real encoding was successfully applied to a deterministic simulation (Tautou and Pierreval 1995).

Real encoding has its pitfalls just as does binary encoding. Though convergence is usually faster with real genetic algorithms, sometimes this comes at the cost of decreasing the solution quality. "Blocking" is a possible obstacle to converging at the optimum. This occurs when the optimum solution lies outside of above average regions which are more likely to be searched (Goldberg 1991). Even "fancy" genetic operators do not show much promise of solving problems that are blocked. Real genetic algorithms may forfeit some of the low-order knowledge that are maintained in binary strings. Also, genetic operators on real genetic algorithms are not well studied (Goldberg 1991). At this point in genetic algorithm research, there is no clear-cut best representation. Goldberg's advice is simply to pick one without agonizing over it too much (Goldberg 1991). For our genetic algorithm applied to simulation optimization, a real-valued representation scheme was used as it appeared to be as effective as a binary scheme, less difficult to program, and it has been successfully applied to deterministic simulation.

2.4.2 Creating the Initial Population

A method must be chosen to create an initial population. A genetic algorithm's performance can be improved by seeding the population with high fitness candidate
solutions. Heuristics or intuition can be used to select these good individuals. A problem may arise when seeding the initial population because what are thought of as being good solutions may actually lead to only locally optimal solutions, resulting in deception. Another method is simply to create individuals by randomly assigning values within the bounded region of each variable. This may perform worse than a seeded population, but it avoids having to determine what are good candidate solutions which may be a difficult task. Selecting an initial population at random also makes the genetic algorithm more robust to various problems and allows for a more accurate measure of the genetic algorithm's performance (Davis 1987). This random creation of an initial population, therefore, was used in testing and evaluating our genetic algorithm applied to various simulation problems.

2.4.3 Population Size

A trade off is made when choosing the population size. Selecting a larger population will probably cause the genetic algorithm to converge to a better solution by providing more "genetic" diversity, but it will also require more fitness evaluations (i.e. simulation replications) which will require more computational time (Whitley, Starkweather, and Bogart. 1990). The smallest population size found in literature was 15 individuals (Tautou and Pierreval 1995) to reduce computational time. Population sizes of 50 or more are much more common in the literature. Some experimentation was required to determine what would be a good population size for our genetic algorithm to be effective at optimizing a wide variety of simulations.
2.4.4 Population Replacement

In classical genetic algorithms, each iteration consists of generating a new population by applying genetic operators to the existing population. The existing population is then disposed of and the newly created population takes its place. The generation size is the number of individuals in the existing population that are replaced by the next generation in one iteration. For classical genetic algorithms, the generation size is equal to the population size. An alternative to this level of population replacement is called a steady-state genetic algorithm. In this algorithm, the generation size is set equal to one (Montana 1991). In other words, an iteration consists of one new individual being created and added to the population. The individual in the population with the poorest fitness is then discarded to maintain a constant population size (Syswerda 1991). No duplicates are allowed within the population, which improves genetic diversity and allows crossover to have a greater impact on the search (Whitley, Starkweather, and Bogart. 1990). An advantage to this steady-state genetic algorithm is that it immediately incorporates better individuals into the reproduction process (Montana 1991). It finds as good or better solutions than standard genetic algorithms in much less time and is especially more effective on larger, more difficult optimization problems (Whitley, Starkweather, and Bogart. 1990). For application to simulation optimization, a steady state genetic algorithm was used as it appeared to be both an effective and efficient design choice.

2.4.5 Selection Strategies

Selection strategies are a very important component of any genetic algorithm because they implement the survival of the fittest strategy to improve the overall population fitness. Selection dominates early genetic algorithm performance and will restrict
subsequent search to better individuals (Goldberg 1991). This strategy will tell the genetic algorithm how to stochastically select individuals from the current generation to be used to create the next generation. It must ensure that better individuals have a better chance of contributing to the next generation (AI Expert 1990). If this selective pressure is too strong, it may cause premature convergence to a less-than-optimal solution. If the selective pressure is too weak, the genetic algorithm becomes an inefficient search (Michalewicz 1992).

There are many different selection strategies. The kind of solution required may determine which method should be used. If only a good solution is needed, a more elitist selection may be used which will cause the genetic algorithm to converge more quickly. If the globally optimum solution is desired, then a less elitist strategy should be used. Finding this absolute optimum will cost more in terms of computational time. In this thesis, a tournament selection strategy was chosen. Since most simulations are stochastic, no solution can be absolutely proven to be globally optimal. A near optimal solution is the most that can be hoped for. Therefore, in this thesis, a tournament selection strategy was chosen which might seem too elitist. But because of the steady-state nature of the genetic algorithm which prevents identical individuals from being in the population at the same time, this apparent elitism has a limited effect. In a simple tournament selection, a non-uniform random integer is generated between one and three, with two being twice as likely as one or three. This number determines the number of individuals which are selected at random to enter the "tournament." The individual with the highest fitness evaluation among individuals in the tournament is chosen to be a parent for the next generation (Nygard, Ficek, and Sharda 1992). This selection strategy is easy to implement. It also eliminates the need
for scaling because it is purely rank-based. Finally, it is less elitist than using a "biased roulette wheel" (Gulsen, Smith, and Tate 1994).

2.4.6 Scaling Evaluations
In many genetic algorithm applications, it may be necessary to scale fitness evaluations to make the genetic algorithm sensitive enough to converge. The problem lies in having fitness values that are relatively close together (i.e., 103, 97, 101, 92, etc.). It would be difficult for the genetic algorithm to give priority to the 103 over the 92 because they are proportionally very close together. If individuals with higher fitness evaluations are given too much priority in selection, the genetic algorithm may converge prematurely to a less than optimal solution. If individuals with higher fitness evaluations are not given enough priority in selection, the genetic algorithm may fail to converge in a reasonable amount of time (Davis 1987). A good scaling strategy is to divide all fitness values of a population by the minimum fitness value not equal to zero in that population (Ackley 1987). This scaling is recalculated every generation as fitness values may change from one generation to the next. Since the genetic algorithm being designed for simulation optimization uses tournament selection, no scaling of fitness values is needed. This is due to the fact that tournament selection gives priority to individuals on the basis of their rank in the population instead of on the basis of their fitness evaluation.

2.4.7 Crossover
Crossover is a key to the power of genetic algorithms (Davis 1987). It is the primary way in which new combinations of parameters are created. The idea of crossover is that two individuals selected from the current generation are paired up to "mate."
They produce children by swapping sections of their variable encodings with each other. The type of crossover used is dependent on the representation scheme used.

Since our genetic algorithm applied to simulation uses real encoding, crossover for real-encoded genetic algorithms must be considered separately from binary-encoded genetic algorithms. In real-valued crossover, one or two children can be created. Most real genetic algorithm crossover techniques either average the parents parameter values or take some linear combination of them (Goldberg 1991). This produces only one child. This is severely limiting for application to simulation optimization, as not all variables are numeric. Another crossover for real-encoded genetic algorithms is directly borrowed from binary crossover as variable values are exchanged whole between parents to produce two children (Goldberg 1991). One-point crossover was selected since no examples of two-point or uniform crossover for real-representation genetic algorithms were found in the literature. In one-point crossover, two children are normally created. Since in steady-state genetic algorithms only one child is needed, one of the two children created by the crossover operator will be randomly discarded. This type of crossover does not change variable values, so it is seen as slightly less significant to the optimization process than binary crossover. The crossover rate specifies how often crossover is performed to create new individuals from parent individuals. In one application, a crossover rate of 25% was used which caused approximately 25% of the new individuals to be created by crossover (Michalewicz 1992). Another application has crossover rates ranging from 10 to 20% (Tautou and Pierreval 1995). But due to the lack of current research in real-encoded crossover applied to steady-state genetic algorithms, the crossover rate was determined experimentally for the simulation genetic algorithm.
2.4.8 Mutation

Mutation injects new genetic material into a biological population. It does the same for a genetic algorithm's population. It is seen as more important for real-coded genetic algorithms than for binary-coded genetic algorithms. As the genetic algorithm proceeds, genetic material is removed from the population. Mutation prevents the permanent removal of genetic material from the population (Nygard, Ficek, and Sharda 1992). This gives the genetic algorithm its global capability as mutation can free a search from a less-than-optimal local basin (Pham and Yang 1993).

Real-encoded genetic algorithm mutation has two major categories, creeping and jumping. Creeping acts a little like stochastic hill-climbing in that it perturbs the current parameter slightly around its current value. Every parameter within an individual has a small probability of being increased or decreased by either a fixed or changing amount (Goldberg 1991). This is only applicable to numeric variables. Jumping acts like binary mutation except that it is much more disruptive. A variable is chosen at random and then assigned a random value within the range of that variable's bounds. Jumping is helpful for "jumping" out of a less-than-optimal local basin. Since it is disruptive, it should have a small rate of occurrence (Goldberg 1991). An application of a genetic algorithm to a simulation used jump rates ranging from 3.3 to 6.6% (Tautou and Pierreval 1995). For best results, both creeping and jumping should be used. In general, creeping should have a much higher probability of occurring than jumping. Both rates at which these operators are applied to a variable in the simulation genetic algorithm were determined experimentally. To avoid negating the possible benefit of mutation or crossover, only one or the other is used in creating a each new individual.
2.4.9 Stopping Criteria and Convergence Measure

There are a number of different ways to stop a genetic algorithm's search. The most optimistic is testing the genetic algorithm for convergence (Nygard, Ficek, and Sharda 1992). In classical genetic algorithms, this occurs when most of the best individuals in the population are all identical. They are reported to be optimal. In our steady-state genetic algorithm, no identical individuals are allowed to exist in the population simultaneously, so this convergence measure can not be used. Another way to halt a genetic algorithm is to set an acceptable fitness level (Nygard, Ficek, and Sharda 1992). When the genetic algorithm finds an individual that meets this level, it stops. This is useful when finding a good solution is more important than finding the optimal solution or if a desired optimal or near-optimal solution is known. A final stopping criteria is to set a maximum number of replications that can be performed (Ackley 1987). This method may be seen as a safety net to keep the genetic algorithm from performing an unreasonably large number of replications.

Using a steady-state genetic algorithm on simulation problems where the optimum is unknown, a method must be used to measure assumed convergence in the population. One possible method is to assume that the population has converged if there is no improvement in the best 10% of the population over a specified number of iterations. This specified number of iterations is related to the population size. If the population is very large, then more iterations will be required to ensure that no improvement in the top 10% really indicates that the optimal or near-optimal solution has been reached. Another method to measure convergence is to assume the optimal or near-optimal solution has been reached if the worst fitness value in the population is within three standard deviations from the best. First, the variance of the simulation
evaluations are estimated and averaged. If the fitness value of the worst individual in
the population is within $3\sigma$ of the best individual, then there is no way to say with
much confidence that the individual with the worst fitness estimate is, in reality, not
the optimal. In other words, if the genetic algorithm continues, there is an ever­
increasing chance that the truly optimal solution could be discarded. Either one of
these convergence measures or a combination of the two could be used in the
simulation genetic algorithm. After either of these stopping criteria are met, the
genetic algorithm either increases the number of replications per individual's fitness
evaluation and resumes the genetic search or the optimization process ends. The
convergence measure used in this robust simulation genetic algorithm was determined
experimentally.

2.4.10 Increasing Replications
Because simulation output is stochastic, multiple replications of a simulation at a
particular variable setting are executed and their outputs averaged to produce an
estimate of the fitness of that individual. This is a null issue when applying genetic
algorithms to deterministic problems, so it has not been discussed in genetic algorithm
literature. If only one replication is executed for each individual, the fitness value
generated might be a poor estimate. When more replications are executed for each
individual, the fitness estimate becomes better but the cost in computational time also
increases. Therefore, a trade-off exists between improving fitness estimates and
decreasing computational time. Because of the relatively large amount of
computational time required for complex simulation replications and the large number
of iterations required for a genetic algorithm to converge, a ceiling of seven was set as
the most replications that would be run per evaluation of one individual. Intuitively
though, it would be inefficient to execute every fitness evaluation at a level of seven replications. At the beginning of a genetic search, the entire population consists of randomly generated individuals. It seems more efficient, therefore, to evaluate individuals at the beginning of the search for fewer replications each to weed out weaker individuals and improve the overall population more quickly. As the optimal solution is approached, the number of replications should be increased to improve the fitness estimates. For the simulation genetic algorithm, this increase occurs when the convergence measure is triggered. The increase in replications at each convergence and the number of times the genetic algorithm was re-executed was determined experimentally.

2.4.11 Constraints

A genetic algorithm applied to simulation must be able to handle a number of constraining factors. Simulations usually model real-life or proposed real-life systems. The genetic algorithm must be able to manage these limitations so that it only allows the simulation to execute replications on feasible system configurations. The first of these limitations is simply the variable bounds. In real-encoded genetic algorithms, these are easy to maintain. If all variables are within their bounds, there still can be deterministic constraints which are violated by the combination of variables. The simplest way to handle both variable bounds and deterministic constraints is to destroy the infeasible individual (Adeli and Cheng 1994). This will keep the simulation from executing time-consuming replications on infeasible candidates. Finally, an individual's evaluation may fail to meet a stochastic constraint. This kind of constraint is usually handled by assigning a penalty function to the individual's fitness. This can be done by setting the fitness to zero or by subtracting some multiple of the absolute
difference between the fitness and the constraint level from the fitness evaluation (Hajela 1990). In the simulation genetic algorithm, the variable bounds and deterministic constraints are maintained within the algorithm. These constraints can be either integer or real-valued linear constraints. Stochastic constraints, which are less common, are maintained within the simulation by assigning a penalty function to infeasible simulation outputs.
Chapter 3
Design of a Robust Simulation Genetic Algorithm

In this chapter, the design of the genetic algorithm for application to simulation optimization will be discussed. First, qualitative design issues will be summarized. These designs were made from literature suggestions, literature examples, or intuitive reasoning based on the needs of a simulation optimization tool. Second, experiments to determine design parameters which were not well justified in the literature will be briefly described.

3.1 Genetic Algorithm Design from Literature
The majority of the issues involved in designing a robust genetic algorithm for simulation optimization were resolved by qualitative means. The literature was consulted to give guidance in the elements of design. The following is a summary of the simulation genetic algorithm design.

The genetic algorithm uses a real-valued representation scheme to encode optimization parameters, as opposed to using binary encoding. For this representation, integer variables in the simulation are represented by integers in the genetic algorithm. Likewise, real-valued variables are represented by real values in that they can have digits to the right of the decimal. But in a sense, these real-valued variables are treated as integers because they are assigned smallest increments which prevent the genetic algorithm from searching insignificant decimal variations. Non-numeric alternative variables are represented by integers, but these integers are treated as non-ordered alternatives instead of incremental values.
The initial population of a size to be determined experimentally will be created randomly. (For each individual in the population, each of their variables will be randomly assigned a value within the variable's range.) If an individual violates any constraint or is identical to any individual already in the population, it is discarded and another individual is created to take its place. A simulation is executed for each individual and the simulation output is used as the individual's fitness measure. The population is then sorted by fitness in order of best to worst.

In the steady-state genetic algorithm, each iteration consists of the creation of one new individual, evaluation of the individual, and possible replacement of the worst member of the population. To create a new individual, one or two parents are selected from the existing population. Crossover requires two parents, otherwise only one parent is needed. Whether one or two parents are called for, they are selected using tournament selection with random tournament sizes between one and three. A tournament size of two occurs 50% of the time, one and three each occur 25% of the time. A number of individuals equal to the tournament size are selected at random from the population and the individual with the best fitness estimate (highest rank in the population sorting) "wins" the tournament and proceeds to either be mated in a crossover operation or altered in a mutation operation.

The rate at which crossover occurs was determined experimentally. If crossover is called for, one point in between adjacent variables in the variable list is selected randomly. The crossover occurs by creating a new individual using all the variable values to the left of the crossover point from the first parent and splicing that with all the variable values to the right of the crossover point from the second parent.
Mutation mode is entered at a rate of one minus the crossover rate. In other words, if crossover is not called for, then a single parent proceeds to be mutated by either creep and/or jump-mutation operators. The new individual is created by mutating some of the variable values of the single parent. For each variable, a random number between zero and one is generated for the jump-mutation operation. If the number is below the jump-mutation rate, then that variable value is discarded and replaced by a random value from within that variable's bounds. The jump-mutation rate was determined experimentally. If the jump-mutation operation does not occur, then another random number between zero and one is generated for the creep operation for each numeric parameter. (The creep operation is invalid for non-numeric variables and therefore not applied to those variables. It will also have no effect on integer variables which have a small range of permitted values.) If this number is below the creep rate, then the parameter value is "creeped" by adding or subtracting a random value from the existing value. This random value can be no larger than 10% of the range of the variable's bounds. The creep step-size is decreased as the population converges. The equation for this step-size is:

$$\text{step-size} = 0.10 \times U[0, 1] \times R \times (c_p - c_c)/c_p$$

where:
- $U[0,1] = \text{uniform random floating point number between 0 and 1}$
- $R = \text{variable's range}$
- $c_p = \text{convergence point (i.e. 100 iterations of the algorithm)}$
- $c_c = \text{convergence counter}$

The convergence point and convergence counter are used for the stopping rule of the genetic algorithm. The counter increments every iteration if the newly created individual is not within the best 10% of the solutions in the population. If the individual does fall within the best 10%, the counter is reset to zero. When the counter reaches the specified convergence point value, the algorithm is stopped. The creep
rate was determined experimentally. This process of possibly jumping or creeping continues for each variable until what remains is a new individual which is most likely different from the parent that created it.

After the new individual is created by either the crossing-over of two parents or jumping and/or creeping one parent, this new individual is checked for feasibility. First, the individual is compared to each member of the current population to ensure that it is not identical to any existing individual. Second, each of the individual's variables are checked to ensure that they are all within each variable's bounds. (The creep operator is the only operation that could possibly create a parameter out-of-bounds.) Third, the individual is checked to ensure that its variables meet all deterministic constraints. If any one of these tests fail, the infeasible individual is simply discarded and the process of creating a new individual begins again with the selection of new parents. Because the creep operator may produce an infeasible new individual while crossover can never, the actual percentage of individuals produced by crossover will most likely be slightly higher than the specified crossover rate.

If the individual is found to be feasible, it is evaluated using the SIMAN simulator running the simulation model for a specified number of replications. This number of replications will vary between one and seven and is dependent on the method selected for increasing replications. This method was determined experimentally. The simulation creates one output per replication which is used by the genetic algorithm as the fitness measure for that individual. When more than one replication is conducted, the fitness measure of the individual is the arithmetic mean of the outputs for each replication.
After the new individual is evaluated, it is compared with the worst (lowest ranked) individual in the current population. If the new individual is not better than the worst, the new individual is discarded. If the new individual is better than the worst, the new individual enters the population and the worst member is discarded to maintain a constant population size. The population is once again sorted by fitness values, in order of best to worst.

At this stage in the genetic algorithm, the convergence measurement / stopping rule is checked. The method of measuring convergence was determined experimentally. If convergence is not yet detected, the algorithm iterates back to the creation of another new individual. If convergence is detected, the algorithm stops. Depending on the method of increasing simulation replications, the number of replications per fitness evaluation is increased. The evaluations of the existing population are averaged into additional evaluations of the population at the new replication level minus the previous replication level to provide fitness estimates of the entire population at this higher number of replications. The optimization then returns to the genetic algorithm iterating process. If the number of replications has reached seven replications, the optimization process ends and the highest ranked individual is presented as the genetic algorithm's solution.

3.2 Genetic Algorithm Design by Experiment

Issues in designing the robust genetic algorithm for simulation which were not sufficiently discussed in the literature were determined experimentally. Two different experiments were executed for different design issues. The first experiment set out to determine rates at which genetic operators should be applied for optimal performance of the algorithm. The second experiment resolved more difficult design issues such as
population size, convergence measure, and method of increasing replications. Design issues were divided into separate experiments by their assumed interactions with each other. The genetic operators seemed to be inter-related so their rates were determined simultaneously. Population size, convergence measure, and method of increasing replications also intuitively seemed inter-related so they were investigated in the same experiment. These experiments will be discussed in detail in Chapter 4.
Chapter 4
Experiments

In this chapter, the final genetic algorithm design issues are resolved by experimentation. First, the implementation of the genetic algorithm and the subsequent interfacing with the SIMAN simulation program are discussed. Second, the test problems that are used in the experiments as well as in the comparison of the genetic algorithm with simulated annealing are described. Third, the preliminary experiments to determine the rates of genetic operators, the population size, the stopping criteria, and the replication increase will be presented.

4.1 Genetic Algorithm Implementation and Simulation Interface

The genetic algorithm that was developed for this thesis was coded in Borland C++ version 2.0. The optimization process was coded into two separate executable files. The first, GA.EXE, is the main program and contains the entire optimization process and simulation calls. The second, GAINIT.EXE is an initialization program that is called from within GA.EXE and serves as an interface with the user. The source code for both are listed in Appendix A. GA.EXE executes GAINIT.EXE at the beginning of its execution. GAINIT.EXE prompts the user to enter the necessary details of the optimization problem. Details include:

- minimize or maximize
- number, names, and bounds of integer variables
- number, names, bounds, and smallest increment of real variables
- number, names, and number of alternatives of non-numeric variables
- number of integer and real linear constraints
- coefficients, signs, and right-hand-sides of constraints
A smallest increment is required for real variables as the genetic algorithm could not efficiently optimize real-valued variables out to many decimal places. After this information has been keyed in by the user, GAINIT.EXE writes all this data to a GASPECS.DAT file to be used in GA.EXE and ends.

After control is returned, GA.EXE reads the problem specifications from GASPECS.DAT and proceeds to the optimization process. The genetic algorithm creates a random initial population and evaluates each individual in the population at one replication using a system command to execute SIMAN. The specific SIMAN *.P file to run depends on the number of replications. Other considerations for these simulation model files will be discussed later. The genetic algorithm then begins, using genetic operators to create new individuals and evaluating them using SIMAN at one replication each. When the genetic algorithm's stopping rule is reached, the replication level is increased. The program runs each individual in the population for more simulation replications to improve their evaluation estimates. The genetic algorithm is then continued, but individuals are now simulated for this higher number of replications. When the genetic algorithm's stopping rule is reached, the optimization process is completed and the results are written to the screen and to a file.

Simulation models were created using the educational version of SIMAN 4.0. To create models that are compatible with this genetic algorithm optimization program, a number of considerations were made in coding, compiling, and naming the files. The GA.EXE program communicates with the SIMAN simulator through data files. When GA.EXE needs to evaluate a particular individual, it writes the value for each variable of that individual to a data file called INPUT.DAT. These values are within the ranges specified by the user at the initialization stage of the optimization. For integer and
real-valued variables, their actual values are written to the data file. For non-numeric variables, an integer is written to the file which corresponds to a specific alternative. When the specified number of simulation replications conclude, SIMAN writes the replications' outputs (performance measures) to a data file called OUTPUT.DAT which is read by GA.EXE. Averaging over the number of replications is done within GA.EXE. Therefore, the genetic algorithm is very robust and problem non-specific as it merely writes simulation inputs to a data file and then reads simulation outputs from a data file. The genetic algorithm program never has any knowledge or dealings with the internal workings of the simulation. (Because the genetic algorithm is "blind" to how the input parameters produce a performance measure, this genetic algorithm could be used for other optimization besides simulation. This can be done by using another method of calculating a performance measure from the variables instead of simulation and modifying the necessary "system" commands within the genetic algorithm code.)

Since the genetic algorithm is blind to how the problem variables produce a performance measure, the simulation code must account for all of the problem specific details of the system which is being simulated. The simulation must be carefully coded to be able to function with any feasible combination of parameter values. Special notice must be taken to the limitations of the particular version of SIMAN being used. The simulation may not function properly under extreme variable configurations causing the optimization process to crash, to produce an incorrect answer, or not to converge. For example, one error encountered while creating test simulations for this thesis was the simulation running out of available entities. This was partially due to using an educational version of SIMAN which had limited
memory capacity. Therefore, all simulation models were modified and stream-lined to reduce the maximum number of entities in the simulated system at any one time.

The simulation code must be equipped with the ability to read and write to data files. This is easily done in SIMAN using a FILES command in the experiment file to specify the details of which files should be opened and how they should be read or written to. The following is an example of such a command:

```
FILES:INVALUE, "INPUT.DAT", SEQ, FRE:
OUTVALUE, "OUTPUT.DAT", SEQ, FRE;
```

where INVALUE and OUTVALUE are the identifiers used in READ and WRITE blocks in the model file to specify a specific file, INPUT.DAT or OUTPUT.DAT in this case. SEQ specifies that the data files are sequential-access files, which must be read or written to in sequential order. FRE specifies that the values in the files are separated from other values either by commas or, as in this case, a space or spaces. (Pegden, Shannon, Sadowski 1990) In the model file, READ and WRITE statements are used to read and write data to a specific file. For example:

```
READ, INVALUE: MR(1), MR(2);
```

will read the next two values in sequence from INPUT.DAT and assign them to the 'number of resources' variables for resource 1 and resource 2, respectively. WRITE statements have similar syntax.

The simulation should read the values from INPUT.DAT at the beginning of each simulation replication to initialize the simulated system. To do this, one dummy entity is created at time 0 and passes through the necessary READ statements. Any other variable assignments based on the values just read can then be made before the dummy entity is disposed of. Care must be taken to ensure that the values are read
from the file in the same order GA.EXE wrote them to the file. Integer and real-valued variables can usually be directly assigned to their corresponding variables in the simulation. Non-numeric variables are integers which correspond to a specific alternative. These integers must be assigned to a variable which will be used within the simulation to direct the simulation to use a specific alternative. This can be fairly cumbersome from a programming perspective and will probably result in logic for various alternatives existing in the code simultaneously with the integer variable directing system flow to the specified code.

At the end of each simulation replication, one output value is written to the OUTPUT.DAT file. This value represents a final performance measure of the simulated system to be optimized. This was accomplished by creating a dummy entity at the specified finishing time minus a small increment. The model statement:

```
CREATE, 1, TFIN - 0.001;
```

creates a dummy entity at 0.001 time units before the simulation replication ends. This dummy entity is then passed through any blocks necessary to calculate the replication's performance measure, and then this value is written to OUTPUT.DAT using a WRITE block. This dummy entity is then disposed.

Since this genetic algorithm requires that different numbers of replications be run at different points in the optimization, a different experiment files must be made for each number of replications. A REPLICATE statement can ensure that old data in OUTPUT.DAT from previous SIMAN calls will be over-written, but that previous output data from earlier replications in this SIMAN call are not over-written. For example, the statement:

```
REPLICATE, 6, 0, 300, YES, YES, 0;
```
calls for six replications of three hundred time units long which begin at time 0. System status is re-initialized between each replication and previous observations are discarded. A warm-up period of zero time units is specified.

Different random number seeds must be specified in the experiment files of differing replications to keep the outputs of earlier replications from being repeated when running the same simulation configuration for additional replications. To reduce the variance between simulation replications, common random numbers should be used (Nelson 1987). To implement common random numbers, every distribution in the simulation should have its own unique random number seed. This is very easily implemented in SIMAN using various random number seeds in the SEEDS element in the experiment file and then specifying which seed should be used for particular stochastic processes in the model file.

For example, if the genetic algorithm runs each solution for one simulation replication initially and then switches to running them for seven replications, an intermediate step must exist which will improve the one-replication estimates up to seven-replication estimates. This is done by re-running all the solutions in the existing population for six additional replications. To prevent the output of the first replication of the set of six from being identical to the output of the previously run replication, the experiment file for six replications must use different random number seeds than the experiment file for one replication. After all the fitness measures have been updated, the algorithm evaluates all future solutions for seven replications. The experiment file for seven replications has the same random number seeds as the one replication experiment file. Therefore, the solutions that exist during this updating process will no longer have random numbers in common with newer solutions. But assuming
the algorithm runs long enough at the seven-replication level to replace all the solutions that were updated, the reduction of variance by common random numbers would again hold true.

Since the number of replications is only specified in the experiment files, only one model file needs to exist. The different experiment files can be compiled and all linked to the same model file. Care must be taken to name the *.P files to represent the number of replications called for in that particular simulation. GA.EXE specifies how many replications are to be run by calling for specific *.P files. GA.EXE calls MOD1.P for one replication, MOD6.P for six replications, and MOD7.P for seven replications, and so on.

Genetic algorithms usually require a large number of evaluations, which in this case are time-consuming simulation runs. One area where time can be saved is in the loading of SIMAN from the hard drive. Every time GA.EXE calls for replications to be run, the SIMAN program must be loaded into RAM from the hard drive, and then the particular *.P file must be loaded before the replications can begin. When the replications are completed, SIMAN is removed from RAM and control is returned to GA.EXE. The next time replications need to be run, the SIMAN program must be reloaded into RAM and the process repeats. All this loading and re-loading of SIMAN from the hard drive is very time-consuming and also taxing on the hard drive. A way to eliminate this wear and tear on the hard drive and to speed up this process is to use a RAM drive, a virtual disk drive that uses the computer's RAM as virtual disk space. It is easily created by adding the following line to the DOS CONFIG.SYS file:

```
DEVICEHIGH = C:\DOS\RAMDRIVE.SYS 1024 /E
```
This statement will assign 1,024 kilobytes of RAM to function as a virtual disk drive. All the files necessary for the optimization process should be copied to the RAM drive. Change the current drive to the RAM drive and proceed to run GA.EXE the same as if it were running from the hard drive.

4.2 Test Problems

For the design and testing of the simulation genetic algorithm, many test problems were borrowed from the literature and/or developed. The problems can be broken down into six basic problems from which variations were derived to increase the number of test problems. The first problem is a simple mathematical equation instead of a simulation model. The second problem is a simple simulation of a restaurant. The third is a distribution problem. The fourth is a standard buffer problem taken from literature. The fifth is a warehouse storage and retrieval problem and sixth is a job shop problem. Appendix B lists the SIMAN code for the five basic simulation problems. In the following sections, these basic problems will be described along with their variations.

4.2.1 Mathematical Equation Problem

In the preliminary experiment for determining the rates of genetic operators, a simple mathematical equation was used to reduce the computational time of the experiment. The equation used integer, real, and non-numeric variables. A parabolic function is applied to each variable to force the maximum evaluation to occur at the variable's upper and lower bounds and the minimum to occur at the mean of the variable's bounds. For example, the integer variables range from 1 to 11, so the maxima occur at 1 and 11 and the minimum occurs at 6 as shown in Figure 4.1.
The non-numeric variables are treated as integer variables. The overall expression is the summation of the evaluations of these parabolic functions over all the integer, real, and non-numeric variables:

$$f(X) = \sum_{\text{ints}} (6 - x_{\text{int}})^2 + \sum_{\text{reals}} (6 - x_{\text{real}})^2 + \sum_{\text{nons}} (6 - x_{\text{non}})^2.$$  

This same equation was used for two different problems. The specifications for problem 1 is shown in Table 4.1. The number of combinations for each variable type is found by taking the number of possibilities for one variable raised to the power of the quantity of that type of variable. For example, for the integer variables, the number of possibilities for one variable is simply the upper bound minus the lower bound plus one. This value is raised by the power of three for the number of combinations of integer variables. The total number of combinations possible is the product of the number of combinations for each variable type.
Table 4.1 Mathematical equation problem 1.

<table>
<thead>
<tr>
<th>Variable type</th>
<th>Quantity</th>
<th>Lower bound</th>
<th>Upper bound</th>
<th>Smallest increment</th>
<th>Number of combinations</th>
</tr>
</thead>
<tbody>
<tr>
<td>integer</td>
<td>3</td>
<td>1</td>
<td>11</td>
<td>1</td>
<td>131</td>
</tr>
<tr>
<td>real</td>
<td>3</td>
<td>1</td>
<td>11</td>
<td>0.1</td>
<td>1030301</td>
</tr>
<tr>
<td>non-numeric</td>
<td>2</td>
<td>1</td>
<td>11</td>
<td>1</td>
<td>121</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>total number of combinations 1.6593E+11</td>
</tr>
</tbody>
</table>

Problem 2 is the same as problem 1 except that the objective is to maximize $f(X)$ (Table 4.2).

Table 4.2 Mathematical equation problem 2.

<table>
<thead>
<tr>
<th>Objective</th>
<th>maximize $f(X)$</th>
</tr>
</thead>
</table>

The maximum evaluation of the equation is 200 and occurs when all variables have a value of either their lower or upper bound. The minimum evaluation of the equation is 0 and occurs when all variables have a value of 6 which is the mean of their bounds.

4.2.2 Restaurant Problem

The experiment to determine rates of genetic operators used a simple simulation of a restaurant. This problem is a slight modification of the restaurant simulation from *Introduction to Simulation Using SIMAN* (Pegden, Shannon, and Sadowski 1990, 116-20).

The restaurant is simulated to be open from 5 to 9pm (240 minutes). Party sizes range from two to five patrons and arrive according to an exponential distribution. Each table in the restaurant can seat two people so tables are pulled together to accommodate parties larger than two. Cashiers also act as hosts and seat parties when
tables become available. If a new party arrives and there are more than five parties waiting to be seated, the new party leaves. For the sake of simplicity, a waiter only waits on one table at a time and stays with that table from when the party is seated until the party is finished dining. For details on the restaurant model, see Table 4.3.

Table 4.3 Restaurant problem description.

<table>
<thead>
<tr>
<th>Processes</th>
<th>Minutes</th>
</tr>
</thead>
<tbody>
<tr>
<td>time between party arrivals (5 - 6pm)</td>
<td>exponential (1.6)</td>
</tr>
<tr>
<td>time between party arrivals (6 - 7pm)</td>
<td>exponential (1.0)</td>
</tr>
<tr>
<td>time between party arrivals (7 - 9pm)</td>
<td>exponential (1.6)</td>
</tr>
<tr>
<td>party size</td>
<td>discrete (0.4, 2, 0.7, 3, 0.9, 4, 1.0, 5)</td>
</tr>
<tr>
<td>time to be seated</td>
<td>constant (1.0)</td>
</tr>
<tr>
<td>service time</td>
<td>triangular (14, 19, 24)</td>
</tr>
<tr>
<td>dining time</td>
<td>normal (25, 5)</td>
</tr>
<tr>
<td>time to pay cashier</td>
<td>normal (1.5, 5)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Income</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>average price of dinner</td>
<td>$5 / patron</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Costs</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>cashier wages</td>
<td>$25 / cashier</td>
</tr>
<tr>
<td>waiter wages</td>
<td>$40 / waiter</td>
</tr>
<tr>
<td>tables over 50</td>
<td>$10 / extra table</td>
</tr>
</tbody>
</table>

The objective of the problem is to maximize profit by selecting the optimal number of tables, waiters, and cashiers. The restaurant can have fifty tables at no cost, but a cost exists for each extra table. Optimization details are given in Table 4.4.

Table 4.4 Restaurant problem optimization parameters.

<table>
<thead>
<tr>
<th>Objective</th>
<th>Objective function</th>
</tr>
</thead>
<tbody>
<tr>
<td>maximize</td>
<td>profit = income - total costs</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Variables</th>
<th>Variable type</th>
<th>Lower bound</th>
<th>Upper bound</th>
<th>Smallest increment</th>
<th>Number of alternatives</th>
</tr>
</thead>
<tbody>
<tr>
<td>tables</td>
<td>integer</td>
<td>50</td>
<td>75</td>
<td>1</td>
<td>26</td>
</tr>
<tr>
<td>waiters</td>
<td>integer</td>
<td>5</td>
<td>25</td>
<td>1</td>
<td>21</td>
</tr>
<tr>
<td>cashiers</td>
<td>integer</td>
<td>1</td>
<td>3</td>
<td>1</td>
<td>3</td>
</tr>
</tbody>
</table>

number of combinations 1,638
4.2.3 Distribution Problem

The distribution problem was used in the preliminary and primary experiments and in comparison with simulated annealing. A total of six problems were generated by modifying the basic distribution problem (problem 1). The basic problem consists of two workstations in series. Each workstation has an input and an output buffer. A shipping operation transports parts from workstation 1 to workstation 2 in quantities specified by a ship-size. The system operates as a pull system. Desired buffer levels "pull" parts through the system. For example, workstation 1 only produces parts when buffer 2 falls below its desired buffer level. These desired buffer levels can be thought of as reorder points. Customer demand "pulls" parts from buffer 4 at an exponential arrival rate. A sale is made when a customer arrives and removes a part from buffer 4. A sale is lost when a customer arrives and buffer 4 is empty. The simulation is simulated for a 9600 minute month with a 1000 minute warm-up period to move the simulation to steady-state. A diagram of the system is shown in Figure 4.2.

![Diagram of the Distribution Problem](image)

**Figure 4.2 Distribution problem 1.**

Customer demand, transportation and operation times, and costs for distribution problem 1 are shown in Table 4.5.
Table 4.5 Distribution problem 1 description.

<table>
<thead>
<tr>
<th>Processes</th>
<th>Minutes</th>
</tr>
</thead>
<tbody>
<tr>
<td>customer demand</td>
<td>exponential (20)</td>
</tr>
<tr>
<td>time from order to receipt of raw material</td>
<td>triangular (20,30,100)</td>
</tr>
<tr>
<td>workstation 1 process time</td>
<td>triangular (20,30,50)</td>
</tr>
<tr>
<td>shipping time</td>
<td>triangular (100,200,300)</td>
</tr>
<tr>
<td>workstation 2 process time</td>
<td>triangular (20,30,40)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Costs</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>lost sales</td>
<td>$100 / part</td>
</tr>
<tr>
<td>raw material in buffer 1 and workstation 1</td>
<td>$5 / part / month</td>
</tr>
<tr>
<td>work-in-process in buffer 2 and being shipped</td>
<td>$10 / part / month</td>
</tr>
<tr>
<td>work-in-process in buffer 3 and workstation 2</td>
<td>$15 / part / month</td>
</tr>
<tr>
<td>finished goods in buffer 4</td>
<td>$30 / part / month</td>
</tr>
<tr>
<td>shipping cost</td>
<td>$10 / shipment</td>
</tr>
</tbody>
</table>

The objective of the problem is to minimize total cost by finding desired buffer levels and shipping size that best reduce the costs of lost sales, inventory, and shipping. The optimization details are shown in Table 4.

Table 4.6 Distribution optimization parameters (problems 1,3-5).

<table>
<thead>
<tr>
<th>Objective</th>
<th>minimize total costs</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>Variables</th>
<th>Variable type</th>
<th>Lower bound</th>
<th>Upper bound</th>
<th>Smallest increment</th>
<th>Number of alternatives</th>
</tr>
</thead>
<tbody>
<tr>
<td>buffer 1</td>
<td>integer</td>
<td>0</td>
<td>40</td>
<td>1</td>
<td>41</td>
</tr>
<tr>
<td>buffer 2</td>
<td>integer</td>
<td>0</td>
<td>40</td>
<td>1</td>
<td>41</td>
</tr>
<tr>
<td>buffer 3</td>
<td>integer</td>
<td>0</td>
<td>40</td>
<td>1</td>
<td>41</td>
</tr>
<tr>
<td>buffer 4</td>
<td>integer</td>
<td>0</td>
<td>40</td>
<td>1</td>
<td>41</td>
</tr>
<tr>
<td>ship-size</td>
<td>integer</td>
<td>1</td>
<td>50</td>
<td>1</td>
<td>50</td>
</tr>
</tbody>
</table>

number of combinations: 141,288,050

Distribution problem 2 is slightly simpler than problem 1. This simplification was done to speed up the optimization process so this problem could more readily be run many times in the genetic operator rates experiment. The ship-size is forced to be one and shipping cost is eliminated. This difference from problem 1 is shown in Table 4.7.
Table 4.7 Distribution problem 2 description.

<table>
<thead>
<tr>
<th>Cost</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>shipping cost</td>
<td>$0 / shipment</td>
</tr>
</tbody>
</table>

The optimization problem is also changed as the bounds on the buffer sizes are reduced. These reductions, constant ship-size, and the consequent reduction in combinations are shown in Table 4.8.

Table 4.8 Distribution optimization parameters (problem 2).

<table>
<thead>
<tr>
<th>Objective</th>
<th>minimize total costs</th>
</tr>
</thead>
<tbody>
<tr>
<td>Variables</td>
<td>Variable type</td>
</tr>
<tr>
<td>buffer 1</td>
<td>integer</td>
</tr>
<tr>
<td>buffer 2</td>
<td>integer</td>
</tr>
<tr>
<td>buffer 3</td>
<td>integer</td>
</tr>
<tr>
<td>buffer 4</td>
<td>integer</td>
</tr>
<tr>
<td>ship-size</td>
<td>integer</td>
</tr>
<tr>
<td>number of combinations</td>
<td>160,000</td>
</tr>
</tbody>
</table>

Distribution problem 3 is identical to problem 1 except that the mean of the customer arrivals has been changed from 20 to 10 minutes. The new customer demand is shown in Table 4.9.

Table 4.9 Distribution problem 3 description.

<table>
<thead>
<tr>
<th>Process</th>
<th>Minutes</th>
</tr>
</thead>
<tbody>
<tr>
<td>customer demand</td>
<td>exponential (10)</td>
</tr>
</tbody>
</table>

Distribution problem 4 is also identical to problem 1 except that the process time distributions have been tripled. The new distributions are shown in Table 4.10.
Table 4.10 Distribution problem 4 description.

<table>
<thead>
<tr>
<th>Processes</th>
<th>Minutes</th>
</tr>
</thead>
<tbody>
<tr>
<td>time from order to receipt of raw material</td>
<td>triangular (60,90,300)</td>
</tr>
<tr>
<td>workstation 1 process time</td>
<td>triangular (60,90,150)</td>
</tr>
<tr>
<td>shipping time</td>
<td>triangular (100,200,300)</td>
</tr>
<tr>
<td>workstation 2 process time</td>
<td>triangular (60,90,120)</td>
</tr>
</tbody>
</table>

Distribution problem 5 uses the same process times as problem 1. The costs for this problem have been changed. The cost of lost sales has been cut in half and the costs of work-in-process and shipping have been doubled. These changes are shown in Table 4.11.

Table 4.11 Distribution problem 5 description.

<table>
<thead>
<tr>
<th>Costs</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>lost sales</td>
<td>$50/part</td>
</tr>
<tr>
<td>raw material in buffer 1 and workstation 1</td>
<td>$10/part/month</td>
</tr>
<tr>
<td>work-in-process in buffer 2 and being shipped</td>
<td>$20/part/month</td>
</tr>
<tr>
<td>work-in-process in buffer 3 and workstation 2</td>
<td>$30/part/month</td>
</tr>
<tr>
<td>finished goods in buffer 4</td>
<td>$60/part/month</td>
</tr>
<tr>
<td>shipping cost</td>
<td>$20/shipment</td>
</tr>
</tbody>
</table>

The final distribution problem, problem 6, basically doubles the system from problem 1. Four workstations, each with an input and output buffer, are placed in series with a shipping operation between each pair of workstations. A diagram of this enlarged system is shown in Figure 4.3. Table 4.12 shows the details of process times and costs.

The objective is still to minimize total costs. There are now eight desired buffer levels and three shipping-sizes to be optimized. The bounds on the buffer levels have been decreased from those in problem 1 to prevent the simulation from crashing due to too many entities. The optimization parameters are shown in Table 4.13.
Figure 4.3 Distribution problem 6.

Table 4.12 Distribution problem 6 description.

<table>
<thead>
<tr>
<th>Processes</th>
<th>Minutes</th>
</tr>
</thead>
<tbody>
<tr>
<td>customer demand</td>
<td>exponential (20)</td>
</tr>
<tr>
<td>time from order to receipt of raw material</td>
<td>triangular (20,30,100)</td>
</tr>
<tr>
<td>workstation 1 process time</td>
<td>triangular (20,30,50)</td>
</tr>
<tr>
<td>shipping time 1</td>
<td>triangular (100,200,300)</td>
</tr>
<tr>
<td>workstation 2 process time</td>
<td>triangular (20,30,40)</td>
</tr>
<tr>
<td>shipping time 2</td>
<td>triangular (50,100,150)</td>
</tr>
<tr>
<td>workstation 3 process time</td>
<td>triangular (60,90,150)</td>
</tr>
<tr>
<td>shipping time 3</td>
<td>triangular (200,400,600)</td>
</tr>
<tr>
<td>workstation 4 process time</td>
<td>triangular (60,90,120)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Costs</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>lost sales</td>
<td>$100 / part</td>
</tr>
<tr>
<td>raw material in buffer 1 and workstation 1</td>
<td>$5 / part / month</td>
</tr>
<tr>
<td>work-in-process in buffer 2 and shipping 1</td>
<td>$10 / part / month</td>
</tr>
<tr>
<td>work-in-process in buffer 3 and workstation 2</td>
<td>$15 / part / month</td>
</tr>
<tr>
<td>work-in-process in buffer 4 and shipping 2</td>
<td>$20 / part / month</td>
</tr>
<tr>
<td>work-in-process in buffer 5 and workstation 3</td>
<td>$30 / part / month</td>
</tr>
<tr>
<td>work-in-process in buffer 6 and shipping 3</td>
<td>$40 / part / month</td>
</tr>
<tr>
<td>work-in-process in buffer 7 and workstation 4</td>
<td>$50 / part / month</td>
</tr>
<tr>
<td>finished goods in buffer 8</td>
<td>$30 / part / month</td>
</tr>
<tr>
<td>shipping cost</td>
<td>$10 / shipment</td>
</tr>
</tbody>
</table>
Table 4.13 Distribution optimization parameters (problem 6).

<table>
<thead>
<tr>
<th>Variables</th>
<th>Variable type</th>
<th>Lower bound</th>
<th>Upper bound</th>
<th>Smallest increment</th>
<th>Number of alternatives</th>
</tr>
</thead>
<tbody>
<tr>
<td>buffer 1</td>
<td>integer</td>
<td>0</td>
<td>25</td>
<td>1</td>
<td>26</td>
</tr>
<tr>
<td>buffer 2</td>
<td>integer</td>
<td>0</td>
<td>20</td>
<td>1</td>
<td>21</td>
</tr>
<tr>
<td>buffer 3</td>
<td>integer</td>
<td>0</td>
<td>20</td>
<td>1</td>
<td>21</td>
</tr>
<tr>
<td>buffer 4</td>
<td>integer</td>
<td>0</td>
<td>20</td>
<td>1</td>
<td>21</td>
</tr>
<tr>
<td>buffer 5</td>
<td>integer</td>
<td>0</td>
<td>25</td>
<td>1</td>
<td>26</td>
</tr>
<tr>
<td>buffer 6</td>
<td>integer</td>
<td>0</td>
<td>20</td>
<td>1</td>
<td>21</td>
</tr>
<tr>
<td>buffer 7</td>
<td>integer</td>
<td>0</td>
<td>20</td>
<td>1</td>
<td>21</td>
</tr>
<tr>
<td>buffer 8</td>
<td>integer</td>
<td>0</td>
<td>25</td>
<td>1</td>
<td>26</td>
</tr>
<tr>
<td>ship-size 1</td>
<td>integer</td>
<td>1</td>
<td>20</td>
<td>1</td>
<td>20</td>
</tr>
<tr>
<td>ship-size 2</td>
<td>integer</td>
<td>1</td>
<td>20</td>
<td>1</td>
<td>20</td>
</tr>
<tr>
<td>ship-size 3</td>
<td>integer</td>
<td>1</td>
<td>20</td>
<td>1</td>
<td>20</td>
</tr>
</tbody>
</table>

number of combinations: 5.7426E+14

4.2.4 Standard Buffer Problem

As a benchmark to judge the performance of the genetic algorithm, a standard buffer problem was borrowed from literature. It is the basic problem of setting buffer sizes to optimize part flow in a serial production line. From the literature, a gradient technique was applied to this problem (Ho, Eyler, and Chien 1979) with results that will be compared to the genetic algorithm's performance.

The problem consists of a serial production line consisting of five machines separated by buffers. Each machine performs a distinct operation on the part. Machines being down or buffers being full may cause blocking in earlier machines. A diagram of the system is shown in Figure 4.4.

Figure 4.4 Standard buffer problem.
The system being modeled is said to have infrequent shifting because of the relatively long times between a machine shifting from one state to another. Details of the system elements are shown in Table 4.14.

Table 4.14 Standard buffer problem description.

<table>
<thead>
<tr>
<th>Processes for all machines</th>
<th>Minutes</th>
</tr>
</thead>
<tbody>
<tr>
<td>cycle time</td>
<td>1</td>
</tr>
<tr>
<td>mean time between failure</td>
<td>1000</td>
</tr>
<tr>
<td>mean time to repair</td>
<td>200</td>
</tr>
</tbody>
</table>

The objective of the problem is to find the optimal buffer sizes for the system to produce 32,000 parts in the shortest amount of time. A size constraint ensures that the sum of all buffer sizes does not exceed 400. Optimization parameters are shown in Table 4.15. The constraint is shown in Table 4.16.

Table 4.15 Standard buffer problem optimization parameters.

<table>
<thead>
<tr>
<th>Variables</th>
<th>Variable type</th>
<th>Lower bound</th>
<th>Upper bound</th>
<th>Smallest increment</th>
<th>Number of alternatives</th>
</tr>
</thead>
<tbody>
<tr>
<td>buffer 1</td>
<td>integer</td>
<td>1</td>
<td>300</td>
<td>1</td>
<td>300</td>
</tr>
<tr>
<td>buffer 2</td>
<td>integer</td>
<td>1</td>
<td>300</td>
<td>1</td>
<td>300</td>
</tr>
<tr>
<td>buffer 3</td>
<td>integer</td>
<td>1</td>
<td>300</td>
<td>1</td>
<td>300</td>
</tr>
<tr>
<td>buffer 4</td>
<td>integer</td>
<td>1</td>
<td>300</td>
<td>1</td>
<td>300</td>
</tr>
</tbody>
</table>

number of combinations 8.100E+09

Table 4.16 Standard buffer problem constraint.

<table>
<thead>
<tr>
<th>Constraint</th>
</tr>
</thead>
<tbody>
<tr>
<td>buffer 1 + buffer 2 + buffer 3 + buffer 4 &lt;= 400</td>
</tr>
</tbody>
</table>
4.2.5 Warehouse Storage and Retrieval Problem

A more complex problem which uses integer, real, and non-numeric variables involved the design of an automated storage and retrieval system (AS/RS). The system requires the selection of the number of bays and levels, the size of equipment motors, and the depth of the storage racks. The number of levels and the number of bays are combined into one variable called the rack dimension. The size of the motors are represented by two variables, the speed of the horizontal motor and the speed of the vertical motor. Finally, a non-numeric depth variable indicates whether the racks are single or double deep. Three problems are used which are slight variations of each other. Table 4.17 shows the minimum system requirements for each problem.

Table 4.17 AS/RS problem requirements.

<table>
<thead>
<tr>
<th>Problem</th>
<th>Open storage spaces</th>
<th>Minimum rate of storage (cycles / hour)</th>
<th>Minimum rate of retrieval (cycles / hour)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2000</td>
<td>20</td>
<td>25</td>
</tr>
<tr>
<td>2</td>
<td>1000</td>
<td>28</td>
<td>35</td>
</tr>
<tr>
<td>3</td>
<td>400</td>
<td>48</td>
<td>60</td>
</tr>
</tbody>
</table>

The representations for the rack dimension variable and their corresponding values are shown in Table 4.18.
Table 4.18 Representation of rack dimension variables.

<table>
<thead>
<tr>
<th>Problem</th>
<th>Rack dimension variable</th>
<th>Bays</th>
<th>Levels (single rack)</th>
<th>Levels (double rack)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>4</td>
<td>250</td>
<td>125</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>5</td>
<td>200</td>
<td>100</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>6</td>
<td>167</td>
<td>84</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>7</td>
<td>143</td>
<td>72</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>8</td>
<td>125</td>
<td>63</td>
</tr>
<tr>
<td></td>
<td>5</td>
<td>9</td>
<td>112</td>
<td>56</td>
</tr>
<tr>
<td></td>
<td>6</td>
<td>10</td>
<td>100</td>
<td>50</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>4</td>
<td>125</td>
<td>63</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>5</td>
<td>100</td>
<td>50</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>6</td>
<td>84</td>
<td>42</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>7</td>
<td>72</td>
<td>36</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>8</td>
<td>63</td>
<td>32</td>
</tr>
<tr>
<td></td>
<td>5</td>
<td>9</td>
<td>56</td>
<td>28</td>
</tr>
<tr>
<td></td>
<td>6</td>
<td>10</td>
<td>50</td>
<td>25</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>4</td>
<td>50</td>
<td>25</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>5</td>
<td>40</td>
<td>20</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>6</td>
<td>34</td>
<td>17</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>7</td>
<td>29</td>
<td>15</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>8</td>
<td>25</td>
<td>13</td>
</tr>
<tr>
<td></td>
<td>5</td>
<td>9</td>
<td>23</td>
<td>12</td>
</tr>
<tr>
<td></td>
<td>6</td>
<td>10</td>
<td>20</td>
<td>10</td>
</tr>
</tbody>
</table>

The objective of the problem is to select the minimum cost configuration that will meet the specified requirement. One of these requirements is the total number of open storage spaces. Open storage spaces are calculated by the following equation:

$$\text{Open spaces} = \text{bays} \times \text{levels} \times \text{depth} \times 2.$$  

Each bay has two sides so the above equation is multiplied by two. Table 4.19 shows the optimization parameters for this problem.
Table 4.19 AS/RS problem optimization parameters.

<table>
<thead>
<tr>
<th>Variables</th>
<th>Variable type</th>
<th>Lower bound</th>
<th>Upper bound</th>
<th>Smallest increment</th>
<th>Number of alternatives</th>
</tr>
</thead>
<tbody>
<tr>
<td>rack dimension</td>
<td>integer</td>
<td>0</td>
<td>6</td>
<td>1</td>
<td>7</td>
</tr>
<tr>
<td>horiz. max speed (fpm)</td>
<td>real</td>
<td>360</td>
<td>740</td>
<td>20</td>
<td>20</td>
</tr>
<tr>
<td>vert. max speed (fpm)</td>
<td>real</td>
<td>10</td>
<td>200</td>
<td>10</td>
<td>20</td>
</tr>
<tr>
<td>rack depth</td>
<td>non-numeric</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>number of combinations</td>
</tr>
</tbody>
</table>

The simulation models use a representation of the above parameters. This representation is shown in Table 4.20.

Table 4.20 AS/RS problem optimization representation.

<table>
<thead>
<tr>
<th>Variables</th>
<th>Variable type</th>
<th>Lower bound</th>
<th>Upper bound</th>
<th>Smallest increment</th>
<th>Number of alternatives</th>
</tr>
</thead>
<tbody>
<tr>
<td>rack dimension</td>
<td>integer</td>
<td>0</td>
<td>6</td>
<td>1</td>
<td>7</td>
</tr>
<tr>
<td>horiz. max speed (fpm)</td>
<td>integer</td>
<td>0</td>
<td>19</td>
<td>1</td>
<td>20</td>
</tr>
<tr>
<td>vert. max speed (fpm)</td>
<td>integer</td>
<td>0</td>
<td>19</td>
<td>1</td>
<td>20</td>
</tr>
<tr>
<td>rack depth</td>
<td>integer</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>number of combinations</td>
</tr>
</tbody>
</table>

This problem has no deterministic constraints on the variables besides their bounds. It does, however, have stochastic constraints on the simulation output. These constraints are the system requirements. If the storage cycles per hour or the retrieval cycles per hour are below system requirements, a penalty cost is added to the overall system cost. For example, if the storage cycles per hour requirement is not met, the penalty cost is calculated by the following equation:

\[
PC = $1,000,000 \times (\text{Required Cycles/hour} - \text{Achieved Cycles/hour})
\]

The penalty cost for not meeting the required retrieval rate is calculated using the same equation. These penalty costs cause the genetic algorithm to highly favor only those systems that meet the minimal system requirements of each problem.
The stochastic simulation output shows whether the system configuration is feasible or not. The fitness measure of each system is the deterministic cost of the system based on its configuration plus any penalties incurred for being infeasible. Therefore, unless a system configuration is on the border of feasibility, its actual fitness measure will be deterministic. Of all the test problems, only the AS/RS problems have this deterministic feature.

When the AS/RS simulations were coded for SIMAN, event routines were required. Therefore SIMAN.EXE cannot be used to execute these problems. Instead, the event routines were coded in a C file called ASRS.C listed in Appendix B.5.3. This C file was linked with SIMAN to produce ASRS.EXE which must be used instead of SIMAN.EXE for the AS/RS problems.

4.2.6 Job-shop Problem
A basic job-shop problem was created to test the genetic algorithm's ability to solve a complex problem with many decision variables. This basic problem was modified to create a total of eleven different test problems. The job-shop produces three different parts, each requiring a different sequence of machining operations. The job-shop consists of three types of machines: mills, lathes, and drills. These machines are grouped into workstations depending on machine type. For example, all mills are located in the milling workstation. Raw materials are located in the "Enter" station and finished goods are moved to the "Exit" station. Parts are transported from station to station by automatic guided vehicles (AGVs) one part at a time. Table 4.21 shows the operation sequence for each part. Table 4.22 shows the distances between workstations.
Table 4.21 Job-shop problem operation sequences.

<table>
<thead>
<tr>
<th>Part</th>
<th>Operation 1</th>
<th>Operation 2</th>
<th>Operation 3</th>
<th>Operation 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>mill</td>
<td>lathe</td>
<td>drill</td>
<td>-</td>
</tr>
<tr>
<td>2</td>
<td>lathe</td>
<td>mill</td>
<td>drill</td>
<td>-</td>
</tr>
<tr>
<td>3</td>
<td>drill</td>
<td>mill</td>
<td>lathe</td>
<td>mill</td>
</tr>
</tbody>
</table>

Table 4.22 Job-shop problem distance between stations.

<table>
<thead>
<tr>
<th>(in feet)</th>
<th>mill</th>
<th>lathe</th>
<th>drill</th>
<th>exit</th>
</tr>
</thead>
<tbody>
<tr>
<td>enter</td>
<td>200</td>
<td>250</td>
<td>250</td>
<td>125</td>
</tr>
<tr>
<td>mill</td>
<td>-</td>
<td>150</td>
<td>150</td>
<td>200</td>
</tr>
<tr>
<td>lathe</td>
<td>-</td>
<td>-</td>
<td>300</td>
<td>200</td>
</tr>
<tr>
<td>drill</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>175</td>
</tr>
</tbody>
</table>

The shop uses a CONWIP (CONstant Work-In-Process) strategy. The rule states that no new job can begin (be released from the Enter station) until the Work-In-Process in the system has fallen below a fixed level (Donohue and Spearman 1993). In other words, when one part is finished and leaves the system, a raw part is released into the system for production.

Regarding the machining operations, it will be assumed that all identical machines will operate at the same percentage of their maximum cutting speed. The maximum cutting speeds and the percentage of the cutting speed used for the standard deviation of the machining operations are shown in Table 4.23.
Table 4.23 Maximum cutting speeds.

<table>
<thead>
<tr>
<th>Machine</th>
<th>Maximum cutting speed (surface ft. / minute)</th>
<th>Standard deviation percentage</th>
</tr>
</thead>
<tbody>
<tr>
<td>mill (low-end)</td>
<td>500</td>
<td>7.50%</td>
</tr>
<tr>
<td>mill (med)</td>
<td>570</td>
<td>5.00%</td>
</tr>
<tr>
<td>mill (high-end)</td>
<td>670</td>
<td>2.50%</td>
</tr>
<tr>
<td>lathe (low-end)</td>
<td>530</td>
<td>7.50%</td>
</tr>
<tr>
<td>lathe (med)</td>
<td>620</td>
<td>5.00%</td>
</tr>
<tr>
<td>lathe (high-end)</td>
<td>730</td>
<td>2.50%</td>
</tr>
<tr>
<td>drill (low-end)</td>
<td>670</td>
<td>6.00%</td>
</tr>
<tr>
<td>drill (high-end)</td>
<td>800</td>
<td>3.00%</td>
</tr>
</tbody>
</table>

Tool-life is a function of cutting speed. Higher cutting speeds means shorter tool-lives. Raw material and tooling material are assumed to be identical for every operation. The following equation is used to calculate tool-life from cutting speed:

\[
\text{Tool-life} = \frac{C}{(\text{cutting speed})^\alpha}
\]

where \( C \) and \( \alpha \) are constants based on the material of the part and the tool (Chang, Wysk, and Wang 1991). For this problem, \( C \) equals 80,000 and \( \alpha \) equals 1.2.

All machining operation process times are normally distributed with a mean of the surface feet divided by the cutting speed and a standard deviation equal to the standard deviation percentage of the cutting speed from Table 4.23.

The eleven job-shop problems are all very similar. The first problem will be discussed in detail and then how the next ten problems differ from the first will simply be stated.

The objective in this problem is to maximize profit. Profit is the revenue brought in from the sale of the manufactured parts less the total cost of manufacturing. The revenue is the price of each part times the number of that part produced in a specific time frame. The price of each part is given in Table 4.24.
Table 4.24 Revenue from sales

<table>
<thead>
<tr>
<th>Part</th>
<th>Revenue / part</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>$200</td>
</tr>
<tr>
<td>2</td>
<td>$175</td>
</tr>
<tr>
<td>3</td>
<td>$250</td>
</tr>
</tbody>
</table>

The purchase costs of machines have been amortized over twelve years to produce an estimate of the daily cost of owning and operating each machine (Donohue and Spearman 1993). Costs are given for high speed machines (high-end), low speed machines (low-end), and mid-range speed machines. The cost of tools is dependent on the machine on which it is used. It is assumed that there is one operator per machine. Also included are the daily costs of storing inventory (Work-In-Process) and the daily cost of owning and operating AGVs. Table 4.25 shows all the costs for job-shop problem 1.

Table 4.25 Costs for job-shop (problem 1).

<table>
<thead>
<tr>
<th>Machine Costs</th>
<th>Tool Costs</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mills</td>
<td>Mills</td>
</tr>
<tr>
<td>High-end</td>
<td>High-end</td>
</tr>
<tr>
<td>Mid-range</td>
<td>Mid-range</td>
</tr>
<tr>
<td>Low-end</td>
<td>Low-end</td>
</tr>
<tr>
<td>Lathes</td>
<td>Lathes</td>
</tr>
<tr>
<td>High-end</td>
<td>High-end</td>
</tr>
<tr>
<td>Mid-range</td>
<td>Mid-range</td>
</tr>
<tr>
<td>Low-end</td>
<td>Low-end</td>
</tr>
<tr>
<td>Drills</td>
<td>Drills</td>
</tr>
<tr>
<td>High-end</td>
<td>High-end</td>
</tr>
<tr>
<td>Low-end</td>
<td>Low-end</td>
</tr>
<tr>
<td>Inventory Costs</td>
<td>Labor Costs</td>
</tr>
<tr>
<td>Part 1</td>
<td>Operator</td>
</tr>
<tr>
<td>Part 2</td>
<td>Mat. Handling Costs</td>
</tr>
<tr>
<td>Part 3</td>
<td>AGVs</td>
</tr>
</tbody>
</table>

- Mills High-end: $1525 / day
- Mills Mid-range: $1200 / day
- Mills Low-end: $950 / day
- Lathes High-end: $1200 / day
- Lathes Mid-range: $925 / day
- Lathes Low-end: $700 / day
- Drills High-end: $900 / day
- Drills Low-end: $650 / day
- Part 1 Inventory: $0.12 / part / day
- Part 2 Inventory: $0.11 / part / day
- Part 3 Inventory: $0.15 / part / day
- Operator Labor: $124 / day
- Mat. Handling Costs: AGVs $12 / day
The amount of material removed during each operation and the cutting speed determine the process time of that operation assuming a constant in-feed. The surface feet removed during each operation is shown in Table 4.26.

Table 4.26 Surface feet removed for each operation (problems 1-4).

<table>
<thead>
<tr>
<th>Part</th>
<th>Operation 1</th>
<th>Operation 2</th>
<th>Operation 3</th>
<th>Operation 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>6000</td>
<td>3700</td>
<td>2500</td>
<td>-</td>
</tr>
<tr>
<td>2</td>
<td>4500</td>
<td>3200</td>
<td>1700</td>
<td>drill</td>
</tr>
<tr>
<td>3</td>
<td>1500</td>
<td>7000</td>
<td>3200</td>
<td>latex</td>
</tr>
</tbody>
</table>

There are 21 optimization variables that the genetic algorithm will change in attempts to optimize the profit of the job-shop. Since a CONWIP strategy is used, Work-In-Process is a variable. Number of AGVs and the number of each type of machine are also variables. In problem 1, any combination of machine types is permitted. Cutting speeds for each machine type are given as a percentage of the maximum cutting speed for that particular machine type. The lower bounds for percentage cutting speeds of mid-range and high-end machines are higher than that of low-end machines. It is wasteful to spend more money on a higher speed machine only to operate it at a lower speed which could be achieved with a less expensive machine. Part mix is specified as the percentage of part 1 and the percentage of part 2. The percentage of part 3 is then calculated as 100% minus the sums of the percentages of parts 1 and 2. Finally, scheduling rules for how parts are taken from input buffers is a non-numeric variable. The four alternative scheduling rules are shown in Table 4.27. The optimization parameters are shown in Table 4.28.
Table 4.27 Scheduling rules for input buffers.

<table>
<thead>
<tr>
<th>Rule</th>
<th>Integer representation</th>
</tr>
</thead>
<tbody>
<tr>
<td>First In-First Out</td>
<td>0</td>
</tr>
<tr>
<td>Most Profitable-First Out</td>
<td>1</td>
</tr>
<tr>
<td>Most Completed-First Out</td>
<td>2</td>
</tr>
<tr>
<td>Longest in System-First Out</td>
<td>3</td>
</tr>
</tbody>
</table>

There are five constraints on the input variables. The first three constraints ensure that there is at least one mill, one lathe, and one drill in the job-shop. The fourth constraint is a space or economic constraint which limits the total number of machines in the job-shop to 12. Finally, the sum of the part mix for parts 1 and 2 must be less than or equal to 100%. If the sum of the part mix for parts 1 and 2 equals 100%, the percentage of part 3 will be 0%. Table 4.29 summarize these constraints.

Table 4.28 Job-shop optimization parameters (problems 1-7).

<table>
<thead>
<tr>
<th>Objective</th>
<th>maximize profit = total revenue - total cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>Variables</td>
<td>Variable type</td>
</tr>
<tr>
<td>Work-In-Process level</td>
<td>integer</td>
</tr>
<tr>
<td>number of AGVs</td>
<td>integer</td>
</tr>
<tr>
<td>number of high-end mills</td>
<td>integer</td>
</tr>
<tr>
<td>number of mid-range mills</td>
<td>integer</td>
</tr>
<tr>
<td>number of low-end mills</td>
<td>integer</td>
</tr>
<tr>
<td>number of high-end lathes</td>
<td>integer</td>
</tr>
<tr>
<td>number of mid-range lathes</td>
<td>integer</td>
</tr>
<tr>
<td>number of low-end lathes</td>
<td>integer</td>
</tr>
<tr>
<td>number of high-end drills</td>
<td>integer</td>
</tr>
<tr>
<td>number of low-end drills</td>
<td>integer</td>
</tr>
<tr>
<td>% cutting speed of high-end mills</td>
<td>real</td>
</tr>
<tr>
<td>% cutting speed of mid-range mills</td>
<td>real</td>
</tr>
<tr>
<td>% cutting speed of low-end mills</td>
<td>real</td>
</tr>
<tr>
<td>% cutting speed of high-end lathes</td>
<td>real</td>
</tr>
<tr>
<td>% cutting speed of mid-range lathes</td>
<td>real</td>
</tr>
<tr>
<td>% cutting speed of low-end lathes</td>
<td>real</td>
</tr>
<tr>
<td>% cutting speed of high-end drills</td>
<td>real</td>
</tr>
<tr>
<td>% cutting speed of low-end drills</td>
<td>real</td>
</tr>
<tr>
<td>% part 1</td>
<td>real</td>
</tr>
<tr>
<td>% part 2</td>
<td>real</td>
</tr>
<tr>
<td>scheduling rules</td>
<td>non-numeric</td>
</tr>
<tr>
<td>number of combinations</td>
<td>1.355E+22</td>
</tr>
</tbody>
</table>
Table 4.29 Constraints for job-shop (problems 1-7).

<table>
<thead>
<tr>
<th>Description</th>
<th>Type</th>
<th>Constraint</th>
</tr>
</thead>
<tbody>
<tr>
<td>mills</td>
<td>integer</td>
<td>high-end + mid-range + low-end &gt;= 1</td>
</tr>
<tr>
<td>lathes</td>
<td>integer</td>
<td>high-end + mid-range + low-end &gt;= 1</td>
</tr>
<tr>
<td>drills</td>
<td>integer</td>
<td>high-end + low-end &gt;= 1</td>
</tr>
<tr>
<td>total machines</td>
<td>integer</td>
<td>number of all machines &lt;= 12</td>
</tr>
<tr>
<td>part mix</td>
<td>real</td>
<td>% part 1 + % part 2 &lt;= 100%</td>
</tr>
</tbody>
</table>

The job-shop problem is simulated for two 8-hour days. Job-shop problems 2 through 11 are identical to problem 1 except where noted in the following descriptions.

Job-shop problem 2 assigns tool costs that are independent of the machine's maximum cutting speed. These tool costs are shown in Table 4.30.

Table 4.30 Costs (problem 2).

<table>
<thead>
<tr>
<th>Tool Costs</th>
<th>Mills</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>High-end</td>
<td>$30 / tool</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Mid-range</td>
<td>$30 / tool</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Low-end</td>
<td>$30 / tool</td>
<td></td>
</tr>
<tr>
<td>Lathes</td>
<td>High-end</td>
<td>$20 / tool</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Mid-range</td>
<td>$20 / tool</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Low-end</td>
<td>$20 / tool</td>
<td></td>
</tr>
<tr>
<td>Drills</td>
<td>High-end</td>
<td>$15 / tool</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Low-end</td>
<td>$15 / tool</td>
<td></td>
</tr>
</tbody>
</table>

Job-shop problem 3 uses the tool costs from problem 2. Other costs that differ from problem 1 are inventory costs and labor costs. All costs that are different from problem 1 are shown in Table 4.31.
Table 4.31 Costs (problems 3-4).

<table>
<thead>
<tr>
<th>Tool Costs</th>
<th>Mills</th>
<th></th>
<th>Lathes</th>
<th></th>
<th>Drills</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>High-end</td>
<td>$30 / tool</td>
<td>$20 / tool</td>
<td>High-end</td>
<td>$15 / tool</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Mid-range</td>
<td>$30 / tool</td>
<td>$20 / tool</td>
<td>Mid-range</td>
<td>$15 / tool</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Low-end</td>
<td>$30 / tool</td>
<td>$20 / tool</td>
<td>Low-end</td>
<td>$15 / tool</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Inventory Costs

<table>
<thead>
<tr>
<th>Part</th>
<th>Revenue / part</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>$22 / part/day</td>
</tr>
<tr>
<td>2</td>
<td>$20 / part/day</td>
</tr>
<tr>
<td>3</td>
<td>$25 / part/day</td>
</tr>
</tbody>
</table>

Labor Costs

| Operator | $400 / day |

Job-shop problem 4 differs from problem 1 in that it uses the same costs as problem 3 shown in Table 4.31. It also assigns a different level of revenue for each finished part. The revenue for each part is shown in Table 4.32.

Table 4.32 Revenue (problems 4&7).

<table>
<thead>
<tr>
<th>Part</th>
<th>Revenue / part</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>$220</td>
</tr>
<tr>
<td>2</td>
<td>$240</td>
</tr>
<tr>
<td>3</td>
<td>$250</td>
</tr>
</tbody>
</table>

Job-shop problem 5 differs from problem 1 in that the costs of high-end machines are lower and the cost of each AGV is higher. These costs are shown in Table 4.33. Also, the surface feet machined for each operation is different and shown in Table 4.34.
Table 4.33 Costs (problem 5).

<table>
<thead>
<tr>
<th>Machine Costs</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Mills</td>
<td></td>
<td></td>
</tr>
<tr>
<td>High-end</td>
<td>$1400/day</td>
<td></td>
</tr>
<tr>
<td>Lathes</td>
<td></td>
<td></td>
</tr>
<tr>
<td>High-end</td>
<td>$1100/day</td>
<td></td>
</tr>
<tr>
<td>Drills</td>
<td></td>
<td></td>
</tr>
<tr>
<td>High-end</td>
<td>$800/day</td>
<td></td>
</tr>
<tr>
<td>Mat. Handling Costs</td>
<td></td>
<td></td>
</tr>
<tr>
<td>AGVs</td>
<td>$50/day</td>
<td></td>
</tr>
</tbody>
</table>

Table 4.34 Surface feet removed for each operation (problems 5-7).

<table>
<thead>
<tr>
<th>Part</th>
<th>Operation 1</th>
<th>Operation 2</th>
<th>Operation 3</th>
<th>Operation 4</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>surf. ft.</td>
<td>surf. ft.</td>
<td>surf. ft.</td>
<td>surf. ft.</td>
</tr>
<tr>
<td>1</td>
<td>5000</td>
<td>3500</td>
<td>2500</td>
<td>-</td>
</tr>
<tr>
<td>2</td>
<td>7000</td>
<td>2300</td>
<td>3500</td>
<td>-</td>
</tr>
<tr>
<td>3</td>
<td>2500</td>
<td>7000</td>
<td>3200</td>
<td>2500</td>
</tr>
</tbody>
</table>

Job-shop problem 6 uses higher inventory costs than problem 1. It uses the lower costs for high-end machines from problem 5. Costs for each AGV are more than problem 1 but not as high as problem 5. Surface feet removed for each operation is the same as problem 5. Table 4.35 shows the costs in problem 6 which differ from the costs in problem 1.
Table 4.35 Costs for job-shop (problems 6&7).

<table>
<thead>
<tr>
<th>Machine Costs</th>
<th>Mills</th>
<th></th>
<th>Lathes</th>
<th></th>
<th>Drills</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>High-end</td>
<td>$1400 / day</td>
<td>High-end</td>
<td>$1100 / day</td>
<td>High-end</td>
<td>$800 / day</td>
</tr>
<tr>
<td>Mat. Handling Costs</td>
<td>AGVs $24 / day</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Inventory Costs</td>
<td>Part 1 $13 / part / day</td>
<td></td>
<td>Part 2 $14 / part / day</td>
<td></td>
<td>Part 3 $12 / part / day</td>
<td></td>
</tr>
</tbody>
</table>

Job-shop problem 7 uses the same costs as problem 6 shown in Table 4.35. Revenue for finished goods are the same as problem 4 shown in Table 4.32.

Job-shop problems 8 through 11 differ significantly from problem 1 in that only one machine type is used per workstation. For problem 8, only high-end machines are used. The optimization parameters and constraints are therefore different. Costs which differ from problem 1 are shown in Table 4.36. Optimization parameters are shown in Table 4.37 and constraints are shown in Table 4.38.

Table 4.36 Costs (problems 8&9).

<table>
<thead>
<tr>
<th>Machine Costs</th>
<th>Mill $1525 / day</th>
<th>Lathe $1200 / day</th>
<th>Drill $900 / day</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tool Costs</td>
<td>Mill tool $30 / tool</td>
<td>Lathe tool $20 / tool</td>
<td>Drill tool $15 / tool</td>
</tr>
</tbody>
</table>
Table 4.37 Job-shop optimization parameters (problems 8-10).

<table>
<thead>
<tr>
<th>Objective</th>
<th>maximize profit = total revenue - total cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>Variables</td>
<td>Variable type</td>
</tr>
<tr>
<td>Work-In-Process level</td>
<td>integer</td>
</tr>
<tr>
<td>number of AGVs</td>
<td>integer</td>
</tr>
<tr>
<td>number of mills</td>
<td>integer</td>
</tr>
<tr>
<td>number of lathes</td>
<td>integer</td>
</tr>
<tr>
<td>number of drills</td>
<td>integer</td>
</tr>
<tr>
<td>% cutting speed of mills</td>
<td>real</td>
</tr>
<tr>
<td>% cutting speed of lathes</td>
<td>real</td>
</tr>
<tr>
<td>% cutting speed of drills</td>
<td>real</td>
</tr>
<tr>
<td>% part 1</td>
<td>real</td>
</tr>
<tr>
<td>% part 2</td>
<td>real</td>
</tr>
<tr>
<td>scheduling rules</td>
<td>non-numeric</td>
</tr>
<tr>
<td>number of combinations</td>
<td>5.008E+13</td>
</tr>
</tbody>
</table>

Table 4.38 Constraints for job-shop (problems 8-11).

<table>
<thead>
<tr>
<th>Description</th>
<th>Type</th>
<th>Constraint</th>
</tr>
</thead>
<tbody>
<tr>
<td>total machines</td>
<td>integer</td>
<td>number of all machines &lt;= 12</td>
</tr>
<tr>
<td>part mix</td>
<td>real</td>
<td>% part 1 + % part 2 &lt;= 100%</td>
</tr>
</tbody>
</table>

Job-shop problem 9 uses the same machine and tool costs as problem 8. Inventory and AGV costs are higher than problem 8 and are shown in Table 4.39. Revenue from finished parts is different from problem 1 and shown in Table 4.40.

Table 4.39 Costs (problem 9).

<table>
<thead>
<tr>
<th>Mat. Handling Costs</th>
<th>Inventory Costs</th>
</tr>
</thead>
<tbody>
<tr>
<td>AGVs $50 / day</td>
<td>Part 1 $22 / part / day</td>
</tr>
<tr>
<td>Part 2 $25 / part / day</td>
<td>Part 3 $30 / part / day</td>
</tr>
</tbody>
</table>
Table 4.40 Revenue (problem 9).

<table>
<thead>
<tr>
<th>Part</th>
<th>Revenue / part</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>$200</td>
</tr>
<tr>
<td>2</td>
<td>$210</td>
</tr>
<tr>
<td>3</td>
<td>$220</td>
</tr>
</tbody>
</table>

Job-shop problem 10 uses lower machine costs for high-end machines. Tool costs are higher. Inventory costs are the same as problem 9, but AGV costs are back to those in problem 1. Revenue from finished parts is once again different from problem 1.

Costs are shown in Table 4.41 and revenue is shown in Table 4.42.

Table 4.41 Costs (problem 10).

<table>
<thead>
<tr>
<th>Machine Costs</th>
<th>Tool Costs</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mill</td>
<td>Mill tool</td>
</tr>
<tr>
<td>$1400 / day</td>
<td>$40 / tool</td>
</tr>
<tr>
<td>Lathe</td>
<td>Lathe tool</td>
</tr>
<tr>
<td>$1100 / day</td>
<td>$35 / tool</td>
</tr>
<tr>
<td>Drill</td>
<td>Drill tool</td>
</tr>
<tr>
<td>$800 / day</td>
<td>$60 / tool</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Inventory Costs</th>
<th>Labor Costs</th>
</tr>
</thead>
<tbody>
<tr>
<td>Part 1</td>
<td>Operator</td>
</tr>
<tr>
<td>$22 / part / day</td>
<td>$124 / day</td>
</tr>
<tr>
<td>Part 2</td>
<td>Mat. Handling Costs</td>
</tr>
<tr>
<td>$25 / part / day</td>
<td>AGVs</td>
</tr>
<tr>
<td>Part 3</td>
<td>$12 / day</td>
</tr>
<tr>
<td>$30 / part / day</td>
<td></td>
</tr>
</tbody>
</table>

Table 4.42 Revenue (problem 10).

<table>
<thead>
<tr>
<th>Part</th>
<th>Revenue / part</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>$260</td>
</tr>
<tr>
<td>2</td>
<td>$240</td>
</tr>
<tr>
<td>3</td>
<td>$250</td>
</tr>
</tbody>
</table>

Job-shop 11 is exactly the same as problem 1 except that it maintains the constraint explained in problem 8. Each workstation may only have one type of machine. Selection of each machine type creates three new non-numeric variables in the optimization problem. For mills and lathes, 1, 2, and 3 represent low-end, mid-range, and high-end, respectively. For drills, 1 and 2 represent low-end and high-end,
respectively. Optimization parameters are shown in Table 4.43. Constraints are shown in Table 4.38.

Table 4.43 Job-shop optimization parameters (problem 11).

<table>
<thead>
<tr>
<th>Variables</th>
<th>Variables type</th>
<th>Lower bound</th>
<th>Upper bound</th>
<th>Smallest increment</th>
<th>Number of alternatives</th>
</tr>
</thead>
<tbody>
<tr>
<td>Work-In-Process level</td>
<td>integer</td>
<td>1</td>
<td>50</td>
<td>1</td>
<td>50</td>
</tr>
<tr>
<td>number of AGVs</td>
<td>integer</td>
<td>1</td>
<td>5</td>
<td>1</td>
<td>5</td>
</tr>
<tr>
<td>number of mills</td>
<td>integer</td>
<td>1</td>
<td>10</td>
<td>1</td>
<td>10</td>
</tr>
<tr>
<td>number of lathes</td>
<td>integer</td>
<td>1</td>
<td>10</td>
<td>1</td>
<td>10</td>
</tr>
<tr>
<td>number of drills</td>
<td>integer</td>
<td>1</td>
<td>10</td>
<td>1</td>
<td>10</td>
</tr>
<tr>
<td>% cutting speed of mills</td>
<td>real</td>
<td>0.25</td>
<td>1.00</td>
<td>0.025</td>
<td>31</td>
</tr>
<tr>
<td>% cutting speed of lathes</td>
<td>real</td>
<td>0.25</td>
<td>1.00</td>
<td>0.025</td>
<td>31</td>
</tr>
<tr>
<td>% cutting speed of drills</td>
<td>real</td>
<td>0.25</td>
<td>1.00</td>
<td>0.025</td>
<td>31</td>
</tr>
<tr>
<td>% part 1</td>
<td>real</td>
<td>0.00</td>
<td>1.00</td>
<td>0.025</td>
<td>41</td>
</tr>
<tr>
<td>% part 2</td>
<td>real</td>
<td>0.00</td>
<td>1.00</td>
<td>0.025</td>
<td>41</td>
</tr>
<tr>
<td>mill type</td>
<td>non-numeric</td>
<td>1</td>
<td>3</td>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>lathe type</td>
<td>non-numeric</td>
<td>1</td>
<td>3</td>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>drill type</td>
<td>non-numeric</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>scheduling rules</td>
<td>non-numeric</td>
<td>0</td>
<td>3</td>
<td>1</td>
<td>4</td>
</tr>
</tbody>
</table>

4.3 Genetic Operator Rate Experiment

Because of the lack of definitive literature regarding the rates of genetic operators for real-encoded genetic algorithms, a simple experiment was conducted to identify rate parameters to be used. As set out in Chapter 3, this genetic algorithm for simulation will use three operators: one-point crossover, creep, and jump-mutation. The probabilities that any of these operators will actually occur can be thought of as the genetic operator's rate. This experiment was done to give some insight and guidelines into what combination of rates would make the genetic algorithm the most robust. To do this, the experiment utilized a few different test problems.
4.3.1 Experiment Test Problems

Four different test problems were used in this experiment. Simple problems were selected fitting the preliminary nature of the experiment. The test problems are thoroughly described in Section 4.2. Mathematical equation problems 1 and 2 were both used and coded directly into the genetic algorithm code. The restaurant problem and distribution problem 2 were also used and coded for the SIMAN simulator.

4.3.2 Experimental Design

A full-factorial experiment was run with three factors made up of the genetic operator rates. Each factor had three levels. The levels were chosen based loosely on rates and recommendations from literature (Michalewicz 1992; AI Expert 1990; Nygard, Ficek, and Sharda 1992) and are shown in Table 4.44.

<table>
<thead>
<tr>
<th>Factor</th>
<th>Levels</th>
<th>Low</th>
<th>Medium</th>
<th>High</th>
</tr>
</thead>
<tbody>
<tr>
<td>crossover rate</td>
<td>0.250</td>
<td>0.500</td>
<td>0.750</td>
<td></td>
</tr>
<tr>
<td>creep rate</td>
<td>0.100</td>
<td>0.250</td>
<td>0.400</td>
<td></td>
</tr>
<tr>
<td>jump rate</td>
<td>0.001</td>
<td>0.005</td>
<td>0.010</td>
<td></td>
</tr>
</tbody>
</table>

The basic genetic algorithm used for this experiment made some assumptions about design issues that were to be resolved in the next experiment. A population size of fifty individuals was used. The genetic algorithm executed each simulation at one replication each until the stopping rule was reached. Then the entire population was evaluated for another six replications to improve fitness estimates. The stopping rule consisted of a maximum number of simulation evaluations. This number varied depending on the test problem being run and was held constant for each problem to
allow for more meaningful comparison (Forrest and Mitchell 1993). Table 4.45 shows the number of replications run for each problem.

Table 4.45 Replications for test problems.

<table>
<thead>
<tr>
<th>Problem</th>
<th>no. of replications</th>
</tr>
</thead>
<tbody>
<tr>
<td>mathematical equation problem 1</td>
<td>800</td>
</tr>
<tr>
<td>mathematical equation problem 2</td>
<td>800</td>
</tr>
<tr>
<td>restaurant problem</td>
<td>450</td>
</tr>
<tr>
<td>distribution problem 2</td>
<td>800</td>
</tr>
</tbody>
</table>

Each combination of genetic operator rates produced a slightly different genetic algorithm. Each of these algorithms was executed on all four test problems. The best simulation output found by the end of the algorithm was used as the measure of that algorithm on that problem. The results of the different test problems were scaled to a simple rating between zero and one by the following method. An estimate of the mean of the solution space for each problem was found by averaging a large number of simulation outputs produced from random variable combinations. For each genetic algorithm, the absolute difference was calculated between the best simulation output and the problem's average. The rating for each algorithm was found by dividing each absolute difference by the problem's maximum absolute difference. This produced ratings in which the best algorithm has a rating of 1.0 and lesser algorithms have lower ratings. The overall rating for each genetic algorithm was found by averaging the individual ratings over all four test problems.

4.3.3 Experiment Results and Conclusions

The output of each genetic algorithm run, the ratings of each rate combination for each problem and the overall rating are shown in Appendix C, Table C.1. Table 4.46 shows
the Analysis of Variance (ANOVA) for this experiment. To find the F-values from tables, 120 was used as a conservative replacement for the error's degrees of freedom.

Table 4.46 ANOVA for the genetic operator rate experiment.

<table>
<thead>
<tr>
<th>Source of Variation</th>
<th>Degrees of Freedom</th>
<th>Sum of Squares</th>
<th>Mean Square</th>
<th>F-test (alpha=.05)</th>
<th>F (alpha=.10)</th>
</tr>
</thead>
<tbody>
<tr>
<td>crossover rate (A)</td>
<td>2</td>
<td>0.0044511</td>
<td>0.0022255</td>
<td>2.9706</td>
<td>3.070</td>
</tr>
<tr>
<td>creep rate (B)</td>
<td>2</td>
<td>0.0009847</td>
<td>0.0004924</td>
<td>0.6572</td>
<td>3.070</td>
</tr>
<tr>
<td>AB</td>
<td>4</td>
<td>0.0024922</td>
<td>0.0006231</td>
<td>0.8316</td>
<td>2.450</td>
</tr>
<tr>
<td>jump rate (C)</td>
<td>2</td>
<td>0.0012991</td>
<td>0.0006495</td>
<td>0.8670</td>
<td>3.070</td>
</tr>
<tr>
<td>AC</td>
<td>4</td>
<td>0.0006212</td>
<td>0.0001553</td>
<td>0.2073</td>
<td>2.450</td>
</tr>
<tr>
<td>BC</td>
<td>4</td>
<td>0.0002616</td>
<td>0.0000654</td>
<td>0.0873</td>
<td>2.450</td>
</tr>
<tr>
<td>ABC</td>
<td>8</td>
<td>0.0023167</td>
<td>0.0002896</td>
<td>0.3865</td>
<td>2.020</td>
</tr>
<tr>
<td>error</td>
<td>81</td>
<td>0.0606838</td>
<td>0.0007492</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

For the F-values from the tables, From the ANOVA, the crossover rate appears to almost be significant at an alpha value of 0.05. At an alpha value of 0.10, the crossover rate is significant. The creep rate, jump rate, and all interactions do not appear to be significant even at the higher alpha value. Based on these results, the crossover rate was selected based on the experiment's numeric results. The creep and jump rates were selected intuitively as the experiment showed them to be insignificant.

The ratings were grouped into categories by one factor at a time and the average ratings for each factor level were calculated. Figure 4.5 shows the average ratings for each crossover level, Figure 4.6 shows the ratings for each creep level, and Figure 4.7 shows the ratings for each jump level.
For the crossover rate, 0.75 has a better average rating than 0.50 and 0.25. For the creep rate, there is no significant difference between 0.10 and 0.25. Similarly, there is no significant difference between the jump rate of 0.005 and 0.010. Based on this experiment, a crossover rate of 0.75 was selected. As for creep and jump-mutation operators, using either of the two higher rates seemed equally promising. From an intuitive perspective, a creep rate of 0.25 threatened to be too disruptive on the population so a creep rate of 0.10 was selected. As for the jump rate, there was nothing to suggest that using a rate of 0.010 would be too disruptive. It was selected for the simulation genetic algorithm. Table 4.47 summarizes which rates were selected.

<table>
<thead>
<tr>
<th>Genetic operator</th>
<th>Rate</th>
</tr>
</thead>
<tbody>
<tr>
<td>crossover</td>
<td>0.75</td>
</tr>
<tr>
<td>creep</td>
<td>0.10</td>
</tr>
<tr>
<td>jump</td>
<td>0.01</td>
</tr>
</tbody>
</table>
4.4 Final Design Issues Experiment

For application of a genetic algorithm to simulation, the algorithm had to take into account the stochastic nature of simulation. In simulation, the estimate of the actual mean performance of a system is improved as more simulation replications of that system are performed. This creates a difficulty for the optimization process as each replication is very time intensive. To keep the algorithm's performance within a reasonable time frame, the number of replications done for each system configuration can be lower at earlier stages of the optimization process without sacrificing solution quality. At early stages of the algorithm, fewer replications will be run per configuration. This will hopefully locate the algorithm in a promising portion of the solution space in a relatively short amount of time. Later in the algorithm, the number of replications will be increased to improve the fitness estimates of individuals to produce a better solution. There is no suggested method from the literature of how much or when to increase these replications. For that reason, this experiment investigated different methods of increasing replications to find which method worked best over a variety of simulation problems. It also investigated how the algorithm measures population convergence. This so-called convergence does not guarantee that the optimal solution has been reached, but it marks the point when it does not appear that the algorithm will find a better solution. When convergence is reached, the replication level is increased per the specified method. The final issue this experiment determined was the population size that seemed to work best over a variety of problems. All of these issues are inter-related so they were investigated in the same experiment.
4.4.1 Experiment Test Problems

Three different simulation test problems were used in this experiment. More complex problems were selected than for the genetic operator rate experiment. The test problems are thoroughly described in Section 4.2. The SIMAN simulator was used for distribution problem 1, warehouse storage and retrieval problem 1 and job-shop problem 11. For this experiment, the job-shop problem was simulated for one 8-hour day and altered to exclude the scheduling rule as an optimization variable. Instead, a first-in/first-out rule was used throughout.

4.4.2 Experimental Design

A full-factorial experiment was conducted with three factors made up of convergence measure, replication increase, and population size. Each factor had three levels. These different levels will now be discussed.

The first convergence measure level signaled convergence if 100 iterations passed without an improvement in the top ten percent of the population. The second level used an estimate of the solution space standard deviation, calculated by averaging the standard deviations of the three best individuals in the population over the specified replication level. When the replication level was one, four replications were done over the five best individuals, as a standard deviation could not be calculated from one replication. Convergence is signaled when the fitness estimate of the worst member of the population is within three solution space standard deviations of the fitness estimate of the best. The third level uses both measures from level one and level two. When either of the convergence measures signal, convergence is signaled.
All of the levels for the replication increase factor begin running the genetic algorithm at one replication. For the first level, the number of replications is increased to seven when the algorithm converges. The population is simulated for an additional six replications to improve the fitness estimate for each member of the population to a seven replication confidence level. The algorithm is then restarted using this existing population, but now evaluating each new individual for seven replications. When convergence is reached, the optimization process concludes. For the second level, the number of replications is increased to four when the first convergence occurs. The entire existing population is evaluated for another three replications before the algorithm restarts using the existing population but now evaluating new individuals for four replications. When the next convergence occurs, the entire population is evaluated for an additional three replications to boost the fitness estimates up to a total replication level of seven. The optimization process then ends. For the third level, after the first convergence occurs, the entire population is evaluated for an additional six replications to improve the fitness estimates to a level of seven replications. The optimization process then concludes, without re-running the algorithm.

The population size levels were chosen loosely from population sizes used in the literature. Population sizes of 50, 100, and 200 individuals were used for the three levels. Table 4.48 summarizes the levels for each factor.

Table 4.48 Experiment factors and levels.

<table>
<thead>
<tr>
<th>Factor</th>
<th>First (1)</th>
<th>Levels</th>
<th>Third (3)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>no improvement in top 10% of population after 100 iters.</td>
<td>worst within 3-sigma of best population</td>
<td>combination of first and second levels</td>
</tr>
<tr>
<td>replication increase</td>
<td>GA at 1 rep. =&gt; GA at 7 reps.</td>
<td>GA at 1 rep. =&gt; GA at 4 reps. =&gt; evaluate population at 7 reps.</td>
<td>GA at 1 rep. =&gt; evaluate pop. at 7 reps.</td>
</tr>
<tr>
<td>population size</td>
<td>50</td>
<td>100</td>
<td>200</td>
</tr>
</tbody>
</table>
The basic genetic algorithm used the genetic operator rates selected from the preliminary experiment shown in Table 4.47. Each combination of factors produced a different genetic algorithm. Each of these algorithms was applied to all three test problems. The best simulation output found by the end of the algorithm was used as the measure of that algorithm on that problem. The results of the different test problems were scaled to a simple rating between zero and one by the same method described in Section 4.3.2.

4.4.3 Experiment Results and Conclusions

The output of each genetic algorithm from this experiment, the ratings of each factor combination for each problem and the overall rating is shown in Appendix C, Table C.2. Table 4.49 shows the Analysis of Variance for this experiment. To find the F-values from tables, 60 was used as a conservative replacement for the error's degrees of freedom.

<table>
<thead>
<tr>
<th>Source of Variation</th>
<th>Degrees of Freedom</th>
<th>Sum of Squares</th>
<th>Mean Square</th>
<th>F-test</th>
<th>F (alpha=.05)</th>
</tr>
</thead>
<tbody>
<tr>
<td>convergence measure (A)</td>
<td>2</td>
<td>2.0006489</td>
<td>1.0003245</td>
<td>9.7909</td>
<td>3.150</td>
</tr>
<tr>
<td>replication increase (B)</td>
<td>2</td>
<td>0.0038624</td>
<td>0.0019312</td>
<td>0.0189</td>
<td>3.150</td>
</tr>
<tr>
<td>AB</td>
<td>4</td>
<td>0.0012154</td>
<td>0.0002538</td>
<td>0.0025</td>
<td>2.530</td>
</tr>
<tr>
<td>population size (C)</td>
<td>2</td>
<td>0.0056783</td>
<td>0.0028392</td>
<td>0.0278</td>
<td>3.150</td>
</tr>
<tr>
<td>AC</td>
<td>4</td>
<td>0.0036346</td>
<td>0.0009086</td>
<td>0.0089</td>
<td>2.530</td>
</tr>
<tr>
<td>BC</td>
<td>4</td>
<td>0.0007302</td>
<td>0.0001825</td>
<td>0.0018</td>
<td>2.530</td>
</tr>
<tr>
<td>ABC</td>
<td>8</td>
<td>0.0014889</td>
<td>0.0001861</td>
<td>0.0018</td>
<td>2.100</td>
</tr>
<tr>
<td>error</td>
<td>54</td>
<td>5.5171333</td>
<td>0.1021691</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

From the ANOVA, only the methods of measuring convergence proved to differ significantly from each other at an alpha value of 0.05. None of the other factors or interactions showed significant differences. For this reason, the method of
convergence was chosen based on the analysis of the experimental data. The other factors were selected based on purely intuitive reasons.

As seen in Table C.2, the ratings for the warehouse storage and retrieval problem are all either zero or one. The zero ratings for all of the $3\sigma$ stopping rule algorithms are due to the fact that their exists no standard deviation for that problem's solution space. The output of the problem, the cost of the system configuration, is deterministic causing no variation between replications. For this reason, the $3\sigma$ method failed on this test problem. The results from this experiment were grouped by the convergence measure used and the ratings were averaged over all the test problems. These overall averages are shown in Table 4.50.

Table 4.50 Convergence measure results.

<table>
<thead>
<tr>
<th>Method</th>
<th>Overall Averages</th>
</tr>
</thead>
<tbody>
<tr>
<td>1) No Improvement</td>
<td>0.971</td>
</tr>
<tr>
<td>2) 3-Sigma</td>
<td>0.632</td>
</tr>
<tr>
<td>3) Both 1 &amp; 2</td>
<td>0.958</td>
</tr>
</tbody>
</table>

The method that measures standard deviation is less robust than the no improvement method, so it being used as the only method was quickly discarded. As seen from the overall averages, there is no benefit from using both methods together, so no improvement in the top ten percent of the population after 100 iterations was selected as the convergence measure for the simulation genetic algorithm.

The ANOVA showed no significant difference between the different methods of increasing the replications. Only one of the methods executed the genetic algorithm at seven replications. The other two simply re-evaluated the population at that level after
the algorithm had concluded. Intuitively, it seemed that having better estimates during the genetic algorithm's execution would improve the algorithms performance at a cost of a few more replications. This cost seemed small compared to the cost of reaching poorer solutions. For this reason, running the genetic algorithm for one replication and then re-running it for seven replications was chosen for the simulation genetic algorithm.

Since the ANOVA showed that there was no significant difference between the algorithms in terms of population size, a population size of 50 was selected for the purpose of lowering the total number of replications. The entire population is evaluated for one replication at the beginning of the optimization process. When the first genetic algorithm converges, the entire population is re-evaluated for an additional six replications each. With a population size of 50, these overall population evaluations come out to 350 simulation replications. If the size were the largest, 200, these evaluations would constitute 1400 replications, showing a considerable increase over using a population size of 50.

Table 4.51 summarizes these final design selections. The following chapter will apply this fully designed genetic algorithm to the test problems and compare those results to simulated annealing applied to the same test problems.

Table 4.51 Summary of final design selections

<table>
<thead>
<tr>
<th>Factor</th>
<th>Level selected</th>
</tr>
</thead>
<tbody>
<tr>
<td>convergence measure</td>
<td>no improvement in top 10%</td>
</tr>
<tr>
<td>replication increase</td>
<td>GA at 1 rep.=&gt;GA at 7 reps.</td>
</tr>
<tr>
<td>population size</td>
<td>50</td>
</tr>
</tbody>
</table>
Chapter 5
Genetic Algorithm Application to Test Problems

After the completion of the preliminary experiments, the genetic algorithm for simulation optimization was fully designed. To test the algorithm's robustness and effectiveness it was applied to the standard buffer problem, all but one of the distribution problems, and all of the AS/RS and job-shop problems. Since the genetic algorithm is being tested for its robustness on a variety of problems for potential use by non-experts in optimization, the algorithm's parameters were not altered from their initial design to better fit any particular test problem and the algorithm is only run once on each problem. This seemed the best way to imitate how a non-expert would use this genetic algorithm if it were provided as a robust optimization tool. In the literature, the genetic algorithms' parameters are continually being modified to find the best solution to a particular problem. Instead of focusing on finding the absolute optimal solution to one problem, this thesis focuses on evaluating the robustness of the optimization algorithm.

To provide a meaningful comparison to another optimization technique, a simulated annealing algorithm was also applied to these test problems. As stated in Chapter 2, both algorithms generally produce similar results in non-simulation applications. An overview of simulated annealing was given in Section 2.3. Section 5.1 presents the details of the simulated annealing algorithm as applied to the test problems. Section 5.2 gives the results of the application of both algorithms to the test problems and Section 5.3 analyzes those results.
5.1 Simulated Annealing Algorithm

The simulated annealing algorithm used in this thesis was developed by Anussornnitisarn (1995) and the code is listed in Appendix A. This algorithm's neighborhood is defined as one unit in each direction along every dimension of the solution space from the current solution. For all test problems, an initial state was set up so that a specified maximum difference between the current best solution and any new solution would be accepted at a probability of fifty percent. To control this, the initial temperature was estimated by the following equation:

\[ \text{Initial temperature} = -\Delta E / \ln(0.5) \]

where \( \Delta E \) is the specified maximum difference and is dependent on the specific test problem. The initial temperature is the calculated initial temperature rounded to the next round number. The final temperature is set to 0.2% of the initial temperature.

The temperature is steadily decreased as the algorithm proceeds at a specified cooling rate, \( \alpha \). For each temperature level, a specific number of random new solutions are evaluated, \( M \). This number is calculated by adding three to the number of optimization variables. Table 5.1 summarizes all of these parameters for each test problem.

<table>
<thead>
<tr>
<th>Problem</th>
<th>Problem Number</th>
<th>delta-E</th>
<th>Initial Temperature calculated</th>
<th>Final Temp.</th>
<th>Cooling Rate (alpha)</th>
<th>Iterations / temp. (M)</th>
</tr>
</thead>
<tbody>
<tr>
<td>standard buffer</td>
<td>1</td>
<td>300 min.</td>
<td>432.8</td>
<td>500</td>
<td>1</td>
<td>0.75</td>
</tr>
<tr>
<td>distribution</td>
<td>1, 3-5</td>
<td>$350</td>
<td>504.9</td>
<td>500</td>
<td>1</td>
<td>0.75</td>
</tr>
<tr>
<td>distribution</td>
<td>6</td>
<td>$350</td>
<td>504.9</td>
<td>500</td>
<td>1</td>
<td>0.75</td>
</tr>
<tr>
<td>AS/RS</td>
<td>1-3</td>
<td>$10,000</td>
<td>14426.9</td>
<td>15000</td>
<td>30</td>
<td>0.80</td>
</tr>
<tr>
<td>job-shop</td>
<td>1-7</td>
<td>$3,000</td>
<td>4328.1</td>
<td>5000</td>
<td>10</td>
<td>0.80</td>
</tr>
<tr>
<td>job-shop</td>
<td>8-10</td>
<td>$3,000</td>
<td>4328.1</td>
<td>5000</td>
<td>10</td>
<td>0.80</td>
</tr>
<tr>
<td>job-shop</td>
<td>11</td>
<td>$3,000</td>
<td>4328.1</td>
<td>5000</td>
<td>10</td>
<td>0.80</td>
</tr>
</tbody>
</table>
5.2 Results of Test Problems

Both the genetic algorithm described in Chapter 4 and the simulated annealing algorithm described above were applied to the test problems. The variable combinations that each algorithm returned as their best solution are listed in Appendix D.

The first test problem is slightly different from the other test problems in that it was taken from literature. The standard buffer problem was solved by a gradient technique (Ho, Eyler, and Chien 1979) and the result is used as a benchmark for comparison. The genetic algorithm and simulated annealing results for the problem are shown in Table 5.2. Three results are shown in the following table to evaluate each algorithm's performance. The first is the total number of replications required by each algorithm. The second is the evaluation of what each algorithm found to be the optimal combination of optimization variables. (This value is not guaranteed to be the true optimum.) The third result is a relative measure for comparing one algorithm to another, called the rating. The rating of algorithm $i$ on problem $j$ is calculated by the following equation:

$$\text{rating}_{ij} = 1 - \frac{\text{abs}(x_{ij} - x_{0j})}{x_{0j}}$$

where:

- $x_{ij} = \text{solution to problem } j \text{ produced by algorithm } i$
- $x_{0j} = \text{best solution to problem } j \text{ out of all algorithms}$

The algorithm with the best solution to a test problem will receive a rating of one. The other algorithm(s) will receive a rating less than one depending on how much worse its solution is compared to the best solution.
Table 5.2 Results of standard buffer problem.

<table>
<thead>
<tr>
<th>Problem</th>
<th>Gradient Technique</th>
<th>Genetic Algorithm</th>
<th>Simulated Annealing</th>
</tr>
</thead>
<tbody>
<tr>
<td>replications</td>
<td>800</td>
<td>2098</td>
<td>651</td>
</tr>
<tr>
<td>optimum (minutes)</td>
<td>54855.7</td>
<td>54654.3</td>
<td>56892.8</td>
</tr>
<tr>
<td>rating</td>
<td>0.996</td>
<td>1.000</td>
<td>0.959</td>
</tr>
</tbody>
</table>

All the test problems are listed in Table 5.3 with their results from both the genetic algorithm and the simulated annealing algorithm. The results include the number of replications, the mean value of the solution over seven replications, the 95% confidence interval for the solution, and the rating. No confidence intervals exist for the AS/RS problems because the system's capital costs are deterministic. For a graphical representation of the algorithms' ratings, see Figure 5.1.

Table 5.3 Results of algorithms applied to test problems.

<table>
<thead>
<tr>
<th>Problem</th>
<th>Genetic Algorithm</th>
<th>Simulated Annealing</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>lower C.I. mean</td>
<td>upper C.I. rating</td>
</tr>
<tr>
<td>std. buffer</td>
<td>reps. n/a</td>
<td>2098</td>
</tr>
<tr>
<td>distribution</td>
<td>2136</td>
<td>1367</td>
</tr>
<tr>
<td>distribution</td>
<td>59853</td>
<td>54654.3</td>
</tr>
<tr>
<td>AS/RS 1</td>
<td>1312</td>
<td>n/a</td>
</tr>
<tr>
<td>AS/RS 2</td>
<td>1432</td>
<td>n/a</td>
</tr>
<tr>
<td>AS/RS 3</td>
<td>2235</td>
<td>n/a</td>
</tr>
<tr>
<td>job-shop 1</td>
<td>2365</td>
<td>34332</td>
</tr>
<tr>
<td>job-shop 2</td>
<td>2828</td>
<td>48024</td>
</tr>
<tr>
<td>job-shop 3</td>
<td>2498</td>
<td>31868</td>
</tr>
<tr>
<td>job-shop 4</td>
<td>3172</td>
<td>41601</td>
</tr>
<tr>
<td>job-shop 5</td>
<td>3059</td>
<td>32466</td>
</tr>
<tr>
<td>job-shop 6</td>
<td>3452</td>
<td>31737</td>
</tr>
<tr>
<td>job-shop 7</td>
<td>2806</td>
<td>38335</td>
</tr>
<tr>
<td>job-shop 8</td>
<td>2875</td>
<td>41866</td>
</tr>
<tr>
<td>job-shop 9</td>
<td>2228</td>
<td>33336</td>
</tr>
<tr>
<td>job-shop 10</td>
<td>4580</td>
<td>45299</td>
</tr>
<tr>
<td>job-shop 11</td>
<td>3206</td>
<td>40903</td>
</tr>
</tbody>
</table>
Figure 5.1 Algorithms' solution ratings.
5.3 Analysis of Results

The genetic algorithm reached a better objective value than the simulated annealing algorithm on 17 out of 20 test problems (85%). The genetic algorithm found a slightly better solution for the standard buffer problem taken from the literature than the gradient technique. This was achieved because the gradient technique used a statistical test to halt the algorithm. The genetic algorithm appears to find a better solution, but it is not statistically significantly better than the gradient technique's solution based on the confidence interval. When simulated annealing out-performed the genetic algorithm, the genetic algorithm's average percent difference from the best was 9.8%, with its worst performance being 18.2%. On the rest of the test problems, simulated annealing had an average percent difference from the best of 15.8%, with its worst performance being 49.3%. Overall, the genetic algorithm's solutions averaged 17.1% better than simulated annealing's solutions.

Since simulation is stochastic, the confidence intervals for the various solutions must be considered. One algorithm which produces a slightly better solution than another algorithm might seem to be the better algorithm, but if the variance of the simulation is large enough at the solution points, it would not be possible to say with any statistical confidence that either solution is better. It is therefore necessary to investigate the significant difference, if any, between the solutions given by each algorithm for each problem. Using a graphical method of comparison (Pegden, Shannon, and Sadowski 1990, 202), a confidence interval lower limit from one algorithm less than the confidence interval upper limit from the other algorithm prevents one from concluding that a significant difference exists between the two solutions. In such a case, the solutions are too close together to suggest that either algorithm out-performed the other on that particular problem. By examining the
confidence intervals given in Table 5.3, the algorithms reached significantly different solutions on all the AS/RS problems and on job-shop problems 1 through 9. The genetic algorithm significantly out-performs simulated annealing on 9 out of 20 problems (45%).

To look at the impact of the size of the test problems on the algorithms' performance, a single performance measure was calculated for each test problem as the percentage of how much better the genetic algorithm's solution was compared to simulated annealing's solution. This value was calculated by the following equation:

\[
\begin{align*}
\% \text{ GA better than SA} &= \left( \frac{x_{SA} - x_{GA}}{x_{SA}} \right) \times 100 & \text{(for minimization)} \\
&= \left( \frac{x_{GA} - x_{SA}}{x_{SA}} \right) \times 100 & \text{(for maximization)}
\end{align*}
\]

where:

- \( x_{SA} \) = the simulated annealing algorithm's solution to the test problem
- \( x_{GA} \) = the genetic algorithm's solution to the test problem

Negative values represent test problems where simulated annealing out-performed the genetic algorithm. Figure 5.2 graphs this measure versus the number of optimization variables in each problem.

![Effect of Problem Size on Performance](image)

**Figure 5.2** Comparison of algorithms on various problem sizes.
It appears that there is no way to predict from the problem size whether the genetic algorithm will out-perform simulated annealing. But from the above graph, it appears that the genetic algorithm out-performs simulated annealing by a much smaller margin in smaller problems. There appears to be little difference in solution between the two algorithms on the smaller problems. There seems to be a greater opportunity for the genetic algorithm to overwhelmingly out-perform simulated annealing as the problem size increases, but again there is no guarantee that it will.

Even though the genetic algorithm out-performs the simulated annealing algorithm on most of the test problems, there is a cost in using the genetic algorithm in terms of computational time. The genetic algorithm, on average, uses about 112% more replications than simulated annealing. In terms of number of replications, the genetic algorithm requires an average of about 1,100 more replications. Figure 5.3 illustrates the relationship between the genetic algorithm's results and required replications as compared to simulated annealing. As above, the genetic algorithm's percentage better than simulated annealing was used along with the genetic algorithm's percentage of replications more than simulated annealing. The percentage of replications is given by the following equation:

\[
\% \text{GA more replications than SA} = \left( \frac{r_{GA} - r_{SA}}{r_{SA}} \right) \times 100
\]

where:
- \( r_{GA} \) = replications required by the genetic algorithm for the test problem
- \( r_{SA} \) = replications required by simulated annealing for the test problem

Intuition predicts that more iterations, which equates to more replications, through an optimization process should produce a better solution. Therefore, seeing a better solution from more replications lends little to supporting a claim that one algorithm is better than another. On the other hand, on six of the test problems the genetic
Figure 5.3 Genetic algorithm's percent difference from simulated annealing.
algorithm found solutions over 15% and up to 97% better than the simulated annealing algorithm but remained within 75% more replications than simulated annealing. This suggests that for simulation optimization, the genetic algorithm has the potential of reaching drastically better solutions without incurring a much higher cost in terms of computational time, though this cannot be guaranteed.

The standard buffer problem illustrates how significant the computational cost can be. On a 486DX2-66 PC, about 3.5 minutes were required per replication. Therefore, it took a little over 5 days to run the genetic algorithm for the 2,098 replications. Simulated annealing applied to the same problem only took a day and a half for the 651 replications.

From the comparison of this genetic algorithm to the simulated annealing algorithm, it appears that genetic algorithm is a good algorithm to use for simulation optimization. It found better solutions than simulated annealing on 85% of the problems and significantly better solutions on 45%. Using the genetic algorithm to improve the probability of finding a better solution than simulated annealing comes at the cost of possibly many more simulation replications.
Chapter 6
Conclusion

6.1 Summary
An automated, robust optimization tool coupled with simulation would provide a powerful tool for the analysis and design of systems. A system designer would be free to focus on developing more accurate simulation models without having to be an expert in optimization techniques. A computerized optimization for simulation is many times faster and much more likely to produce a near-optimal solution than a simulation user doing a manual search.

Of the many optimization techniques in the literature, genetic algorithms are particularly suited for simulation optimization. It is a global search method which means it has mechanisms built into it to prevent it from becoming trapped in local optima. They can handle multiple variable bounds and constraints. Genetic algorithms, by their very nature, handle integer, real and non-numeric variables equally well. Most other optimization techniques were designed to optimize mathematical equations involving real variables and must be substantially altered to work with integer and non-numeric variables.

The purpose of this thesis was to investigate the feasibility of using a genetic algorithm as a robust simulation optimization tool. Feasibility was measured based on two criteria: robustness and effectiveness. Robustness addresses the algorithm's capacity to function on wide variety of problems. Effectiveness addresses the algorithm's capacity to find optimal or a near-optimal solutions to problems in the least amount of time. To investigate these criteria, a genetic algorithm was developed to be
a robust simulation optimization tool. This algorithm was tested on 20 simulation test problems. The algorithm was intentionally not fine-tuned for any particular problem to allow for a less-biased measure of robustness and effectiveness. One test problem solved by a gradient technique in the literature provided a benchmark to compare to the genetic algorithm's solution. The other test problems were developed to represent a wide variety of problem types. While trying to maintain realistic system design scenarios, the number, type, and range of variables were varied to produce problems of different sizes and characteristics. Some of the problems also had deterministic and/or stochastic constraints. A simulated annealing algorithm was applied to the same test problems to provide further comparison with the genetic algorithm's solutions.

The basic elements of the simulation genetic algorithm were developed from suggestions and examples in the literature. A real representation was used instead of a the more traditional binary representation. The initial population was created randomly. A steady-state genetic algorithm model was followed for population replacement at each iteration. A simple tournament selection was used to select parents for reproduction. One-point real crossover, creep, and jump-mutation genetic operators were used to create new individuals from parents. Deterministic constraints on input configurations were implemented by discarding any infeasible individual before it is simulated and then creating a new individual. Stochastic constraints on the simulation's output were implemented by assessing a penalty cost on any infeasible system.

Other elements of the genetic algorithm were determined by experiment. Based on the results of the genetic operator rate experiment, the crossover rate was set to 0.75, the creep was set to 0.10 and the jump-mutation rate was set to 0.01. Based on the results
of the final design issues experiment, the population size was set to 50, the stopping rule of no improvements in the top 10% of the population for 100 iterations was selected, and the algorithm evaluated individuals for one replication at first and then for seven replications after the stopping rule was met for the one replication level.

6.2 Conclusions

The algorithm proved to be robust to all the test problems to which it was applied because it was able to handle all of the different features of the different problems. It was also able to produce solutions to the problems which were much better than any of the random solutions from the initial populations. An example of this can be seen in Figure 6.1 which shows a sub-section of the genetic algorithm's optimization process on distribution problem 4 to minimize cost. The graph shows a snap-shot of the genetic algorithm's population at each iteration. This snap-shot includes the best fitness estimate in the population, the worst fitness estimate, and the average fitness estimate over the entire population. The dramatic jump in solution quality between replications 855 and 1156 is due to the stopping rule being reached at the one replication level and the entire population being re-evaluated at a replication level of seven. As seen from this graph, the best individual and the overall average of the initial population are substantially worse than the algorithm's final solution.

Concerning the effectiveness of the simulation genetic algorithm, the algorithm's solution quality was given highest priority. The number of replications required was also considered but given less priority simply because in real-life applications, finding a significantly better solution will usually be well worth the relatively small cost of more computational time.
Figure 6.1 Portion of genetic algorithm's optimization of distribution problem 4.
The genetic algorithm found a solution as good as the gradient technique's solution on the benchmark test problem taken from literature. When comparing the genetic algorithm's solutions to simulated annealing's, the genetic algorithm produced better solutions on 85% of the 20 test problems. Based on the confidence intervals, the genetic algorithm significantly out-performed simulated annealing on 45% of the test problems. The average improvement of the genetic algorithm's solutions over simulated annealing's solutions was 17.1%.

When considering problem size in relation to solution quality, it appeared that the genetic algorithm out-performed the simulated annealing algorithm by a much smaller margin on smaller problems. This suggests that the genetic algorithm has a greater potential of finding overwhelmingly better solutions on larger problems.

As would be predicted of a genetic algorithm applied to simulation, the cost in terms of computational time due to replications is high. But on a few problems, the genetic algorithm found substantially better solutions with relatively few more replications than simulated annealing. This suggests that for simulation optimization, the genetic algorithm has the potential of reaching drastically better solutions without incurring a much higher cost in terms of computational time.

Concluding from these results, the genetic algorithm should be used on any simulation optimization problem if time allows or if reaching as good a solution as possible is critical. If time is very limited, the simulated annealing algorithm may produce good solutions, especially on smaller problems (approximately 8 decision variables or less).
Overall, the comparison demonstrated the simulation genetic algorithm to be a promising candidate for a robust simulation optimization tool. The genetic algorithm functioned on all the test problems and produced better solutions on most problems and competitive solutions on the rest. Therefore, it seems reasonable to suggest more research in this area. The following section makes some suggestions on where this future research should go.

6.3 Future Work

The results of this thesis support the claim that a genetic algorithm may make a suitable robust optimization tool for simulation. This claim could be bolstered by further research using this genetic algorithm on other test problems or further tests on the same test problems. More tests of the genetic algorithm could be made on the same test problems provided the algorithm is seeded with different random numbers for each observation to produce a different initial population resulting in a different search. However, the results of this thesis were convincing enough to the author about the strengths of a genetic algorithm applied to simulation optimization that the following research suggestions will focus on improvements to the genetic algorithm that should be investigated before more extensive testing is done. A genetic algorithm with improved search logic and reduced replications would be both an effective and robust simulation optimization tool.

The genetic algorithm used in this thesis was developed to be a "good," robust algorithm, but it has much room for improvement as applied to stochastic simulation. These improvements can be divided into two basic categories. The first is in the area of improving the algorithm's search to improve its probability of finding better solutions. This will always be an issue in optimization because it is unlikely that any
optimization procedure will ever be found that can guarantee that the optimal solution has been found for a stochastic system. The second category of improvements is in the area of replication reduction. The genetic algorithm most often found better solutions to the test problems, but usually at a cost of more replications which equates to more computational time.

6.3.1 Search Improvements

Assuming that using a steady-state genetic algorithm with a real representation of variables is reasonable for application to simulation, more research can be done into the details of the genetic algorithm's design. The genetic operators used here, variants of these, and others should be investigated as more research is done in the area of real-representation genetic operators. For example, two-point and uniform crossover should also be considered as possible improvements in the crossover method.

Intuitively, it seems that different problems with different variable types and sizes of solution space might be more efficiently searched using different genetic operator rates, population sizes, and selection strategies. To prevent the user from having to be an expert in genetic algorithms, it would be useful for the genetic algorithm itself to have the ability to automatically set these search parameters based on the details of the problem being optimized. To do this, much research would be needed to determine functions or heuristics that would set population size, selection strategy and genetic operators rates based on the particular problem's number of variables, range of variables, and combinatorial size of the solution space (Miller and Goldberg 1994).

As evidenced in Figure 6.1, the genetic algorithm is very powerful at quickly narrowing the search to a promising sub-section of the solution space. By 600
replications, the entire population is in a "good" neighborhood which is then slowly
explored over the next 3,000 replications before the final stopping rule is reached.
This property of the genetic algorithm leads to two suggestions for further research.

First, "adaptive" genetic algorithm strategies should be investigated. These strategies
gradually changes the search parameters, such as genetic operator rates, as the
optimization proceeds to change the focus of the genetic algorithm from a global
search to a local search. For example, crossover is a powerful global search technique
while the creep operator acts more as a local search technique. In an adaptive genetic
algorithm, the crossover rate would be gradually decreased as the population narrows
into a promising neighborhood while the creep rate would be gradually increased to
strengthen the local search of that neighborhood.

Second, using another search technique in conjunction with a genetic algorithm might
also prove promising. A genetic algorithm could be used first to globally search the
solution space. When a promising neighborhood is located, the optimization could
switch over to a faster and more powerful local search technique. This would exploit
the genetic algorithm's strength as a global tool and minimize its weakness as a local
search technique.

A final recommendation at improving the genetic algorithm's search would be to
divide the population into sub-populations (Whitley, Starkweather and Bogart 1990).
This "distributed" genetic search can aggressively pursue better solutions within each
sub-population without exhausting the overall genetic diversity in the population. At
certain intervals, some members of the sub-populations are exchanged with other sub-
populations to keep the sub-populations from stagnating. These fine-grained parallel
genetic algorithms were found to be superior to more typical serial genetic algorithms and deserve more research for application to simulation.

6.3.2 Replication Reduction

As mentioned above, the large number of fitness evaluations required by the genetic algorithm is a major concern. This is an even greater priority when applying genetic algorithms to simulation rather than to a mathematical function because simulation replications are so time consuming. An obvious weakness of the genetic algorithm used in this thesis is its stopping rule. At the final replication level of 7, the algorithm does not conclude until 100 iterations (700 replications) have passed without producing an individual in the top 10% of the population. Even the slightest, statistically insignificant improvement in the top 10% resets the counter to zero. This feature led the genetic algorithm to require substantially more replications than simulated annealing. Therefore, a better stopping rule in conjunction with how and when the replication level is increased could greatly reduce the number of replications.

One strategy would be to modify the existing stopping rule. Instead of resetting the counter to zero every time a new individual entered the top 10%, the counter could be decreased by a fixed value, 10 for example, but not allowed to go negative. At the early stages of the optimization, when improvements are more frequent, the counter would practically remain at zero, but as improvements happened less often, the counter would gradually increase. If a new individual leads the search into a new direction, an increase in better new individuals will most likely occur which will quickly lower the counter. In other words, insignificant improvements will allow the counter to gradually increase but a number of significant improvements will continue
to decrease the counter until the new, improved neighborhood has been more thoroughly searched.

A number of variations on this theme are possible. Instead of decreasing the counter by a fixed value, the counter could be decreased by a value based on the position the new individual enters the population. For example, if it enters in the bottom half of the population, the counter would still be increased. If it enters slightly above half, the counter would not be changed. If it enters higher in the population, the counter would be decreased based on its distance from the top. A new individual which proved to be the best in the population would therefore decrease the counter by the largest amount. This amount would be predetermined.

A similar strategy would be to change the counter based on the difference between the new individual's solution and the best solution in the population. If a solution is close to or better than the best, the counter is decreased proportionately. If the solution is far from the best, the counter is still only increased by one.

Any of these new counting strategies could be combined with a paired t-test which tests for significant difference between two systems. For example, a paired t-test could be performed between a new individual entering the top 10% of the population and the individual it is bumping out of the top 10%. If there is no significant difference between the new individual and the one just bumped, then the counter should not be reset or decreased, depending on the counting strategy being used. A number of possibilities exist for using this paired t-test to determine if an improvement is significant or not.
The level of significance is dependent on the sample size, which in this case is number of replications. If the t-test is being used in the stopping rule, it would make sense to also use that test to determine the next replication level. This is a more dynamic approach than the one used in this thesis which set the algorithm to execute the simulations at one replication and then at seven, regardless of any significant difference or the variance of the simulated system. For simulations with a low variance, fewer than seven replications may be required to achieve good estimates and measures of significant difference. For simulations with a high variance, seven replications may not be enough to determine any statistical certainty.

The following is an example of a stopping rule and replication increase strategy that makes use of paired t-tests. The genetic algorithm would proceed running each simulation for one replication until a minimum number of iterations had passed to narrow the search to a promising neighborhood. This minimum number of iterations could be based on the size of the solution space or an average variance of the solution space. (A paired t-test is impossible with a sample size of one.) The replication level would then be increased by one and the entire population would be re-evaluated for better estimates. At this point, a paired t-test could be done between the best member of the population and an individual in an arbitrary position in the population (i.e. the fifth best.) If the arbitrary position is too high in the population, the stopping rule will be reached to quickly. Sufficient room in the top of the population must be maintained for slight variations of the best solutions to exist simultaneously. If the position is too low, the opposite will be true of the stopping rule. In that case, the algorithm would have too many indistinguishable solutions and would waste time searching for infinite, insignificant improvements. If no significant difference can be found between the best and the arbitrary individual, then the replication level should be increased by one again.
and the process should repeat. If a significant difference is found, the genetic
algorithm should be executed at this new replication level using this same t-test as the
stopping rule. This rule will only need to be checked when a new individual enters the
population at or above the arbitrary position. When the stopping rule is reached, the
replication level is increased and the process repeats. This continues until a maximum
replication level is reached. This maximum level is determined ahead of time and
based on the variance of the simulation problem. This or a similar process would
allow the algorithm to dynamically set the sample size (number of replications) based
on its need to prove whether one solution is statistically any better than another. This
is an improvement to the $3\sigma$ convergence measure that was investigated in the final
design issues experiment in Section 4.4. This strategy would have to be modified to
function on problems with deterministic or partially deterministic objective functions
such as the AS/RS test problems.

A final suggestion for reducing replications would be to prevent the final replications
of a new individual from being executed if the first replications strongly suggest that
the individual is not better than the worst member of the population. For example, at a
replication level of 7, an individual could be evaluated for 3 replications. If the
average of those replications is worse than the worst member of the population, the
individual is discarded before running its remaining replications. Even if the
discarded individual would have had a slightly higher average after more replications,
it is likely that such an individual would enter the very bottom of the population and
contribute nothing beneficial to overall optimization process. Such a procedure could
dramatically reduce the number of replications required by eliminating frivolous extra
replications on poor candidates.
Reference List


Appendix A

Genetic Algorithm and Simulated Annealing C++ Code

A.1 Simulation Genetic Algorithm Main Program (GA.CPP)

// Steady State Genetic Algorithm for Simulation Optimization
// population size = 50
// increase Replications-> run GA at 1 replication until convergence,
// then run GA at 7 reps. until convergence.
// convergence measure = 100 iterations with no improvement in
// top 10% of population

// Requires the following files:
// GAINIT.EXE initialization program
// SIMAN.EXE, .CFG, .MSG SIMAN simulation program
// MOD1.P compiled SIMAN file running 1 replication.
// MOD6.P compiled SIMAN file running 6 replications.
// MOD7.P compiled SIMAN file running 7 replications.

// Developed by Steve Harris

#include "stdio.h"
#include "conio.h"
#include "stdlib.h"
#include "math.h"
#include "process.h"
#include "dos.h"

#define maxints 11 //max # of integer dec vars
#define maxreals 11 //max # of real dec vars
#define maxnons 6 //max # of non-numeric dec vars
#define maxcis 5 //max # of integer constraints
#define maxcrs 5 //max # of real constraints
#define maxpop 201 //population size

int max_or_min; //+1 for maximize, -1 for minimize
int i, j, h, k; //counting indices
int choice; //for user input
int different, anysame; //preserve no duplicates
int numints, numreals, numnons; //# of int, real and non-numeric dec vars
int numci, numcr; //of int and real constraints
int intlb[maxints], intub[maxints]; //int. upper and lower bounds
int nonalts[maxnons]; //array of non-numeric of # of alternatives
int cisign[maxcis], crsign[maxcrs]; //constraint signs, -2:<, -1:<=, 0:==, 1:=, 2:>, 3:!=
int pint[maxpop][maxints]; //individual's integer variables
int pnon[maxpop][maxnons]; //individual's non-numeric variables
int popsize; //population size
int toppop; //of individs in top 10% of population
int simreps; //count of total number of simulation reps.
int newaddition; //1 if last individ is better than worst
int converge; //counts no improvements until assumes convergence
const int count = 100; //convergence point
int tempint; //temporary integer for swapping in sort
int parent[2]; //pop index to mark selected parents
int numparents; //number of parents to be selected (2 for crossover)
int direction; // -1 or +1 creep directions
int intrange; //range of particular int dec variables
int intcreep; //creeping integer
int pass; //counts number of passes through the GA
int location; //where in program when INTERRUPT.DAT written
int interrupt_count; //counts replications for writing INTERRUPT.DAT
int numreps; //number of reps simulation to be run
int numtrials; //number of trials per combination for std. dev. meas.
int first, last, totalreps; //first through last individ to evaluate
float preal[maxpop][maxreals]; //individs reals
int name[15][maxints];
char *simancall; //which siman model to run
char *resultfile; //which results file to write to
float garbage; //dummy float var
float std_dev; //std. dev. for simulation model
float variance; //variance
float mean[5]; //5 sample means of n reps. for std.dev. estimation
float avgmean; //avg of 5 sample means for std. estimation
float realub[maxreals], realub[maxreals]; //real hi & lo bounds
float realub[maxreals]; //same as above
float smallinterval[maxreals]; //smallest interval for real variables
float ci[maxcisl][maxints]; //int constraint coefficients
float cr[maxcrs][maxreals]; //real constraint coefficients
float rhi[maxcisl]; //int constraint right-hand-sides
float rhis[maxcrs]; //real constraint right-hand-sides
float lincomb; //sums left hand side of constraints
float pvalue[maxpop]; //value of individ
float lastavg; //top pop avg from last iteration
float summation, topaverage; //for calculating averages
float popaverage; //avg. for entire population
float realrange; //range of real variables for creep
float crossover; //random value (0-1) compared to cross rate
float crossrate; //rate at which crossover will occur
float creep; //random value (0-1) for creep
float creepRate; //rate at which creep will occur
float mutate; //random value (0-1) for mutate
float mutaterate; //rate at which mutation will occur
int bestindivid; //best individ from tournament selection
FILE *fileptr;
FILE *file2;

void read_gaspecs(void); //reads GASPECS.DAT file
void initial_pop(void); //creates initial population
void evaluate(void); //evaluates an individual (runs SIMAN)
void sort_pop(void); //sorts individuals in population, best first
void find_popavg(void); //finds mean simulation outputs of pop.
void find_topavg(void); //finds mean sim. outputs of best 10% of pop.
void converge_counter(void); //increments & monitors convergence
void new_individ(void);  //generates new individual using genetic ops.
void pop2file(void);    //writes entire population to a file
void results2file(void); //writes algorithm results to a file
void write_interupt(void); //writes current state of GA to INTERRUPT.DAT
void read_interupt(void);  //reads INTERRUPT.DAT to begin GA in-progress
void calc_std_dev(void);  //performs extra reps. to calc. std. dev.

void main()
{
   // printf("Test: Enter a floating point number: ");
   // scanf("%f", &garbage);

c1rscr();
printf("Genetic Algorithm for Simulation Optimization\n\n");
printf("Choose one:\n1. Initialize for new optimization\n");
printf("\t2. Continue running an interrupted optimization\n");
printf("\t3. (Requires GASPECS.DAT, INTERRUPT.DAT, & PROGRESS.DAT)\n");
printf("\t4. Quit\n");
scanf("%d", &choice);
interrupt_count = 0;
simancall = "siman mod1";
popsize = 50;

if (choice == 3)
   exit (0);

if (choice == 2)    //Resume GA from files
{
   read_gaspecs();
   read_interupt();
   file2 = fopen ("progress.dat", "a");
}
else                  //New GA Initialization
{
   clrscr();
   printf("Choose one:\n1. Create new GASPECS.DAT file\n");
   printf("\t2. Use existing GASPECS.DAT\n");
   scanf("%d", &choice);
   if (choice == 1)
      system("gainit"); //Invokes GAINIT program
   read_gaspecs();

   //Initialize variables
   simreps = 0;          //Zeros total reps. counter
   converge = 0;         //Zeros convergence counter
   lastavg = 0;          //Last top 10% zero
   if (max_or_min==-1) lastavg = 999999.99; //If minimizing, set
                        // last top 10%
                        // to a very large number.
   pass = 1;             //First pass through
   location = 1;         //To resume an interrupted GA
                        // at the correct step.

   //Create initial population
   initial_pop();

   //Evaluate population
   first = 0;            //First individ. to evaluate
last = popsize - 1; //Last individ. to evaluate
numreps = 1; //Evaluate for 1 replication.
totalreps = 1; //Total reps. for each individ.
    // equals 1 rep.
evaluate(); //Evaluate pcp. from first to last.
newaddition = 1; //tells to sort entire pop.
sort_pop(); //sort population

resultfile = "initial.dat"; //INITIAL.DAT contains initial pop.
pop2file(); //Writes initial population to file.
file2 = fopen("progress.dat", "w"); //Opens progress file

//Establish Rates for Genetic Operators for creating new individuals
crossrate = 0.75; //GA uses crossover 75% of the time
creeprate = 0.10; //When no crossover, GA will creep 10% of variables
mutaterate = 0.01; //When no crossover, GA will mutate 1% of variables
toppop = int((float)popsize*.1); // # of individuals in top 10% of pop.

//Begin Optimization, first running GA @ 1 replication, then @ 7 reps.
do{
    newaddition = 1; //Tells to sort entire population
    
    //If optimization is at another location besides 1, then skip the GA.
    if (location == 1)
        //Begins GA
        do{
            if (newaddition > 0) //If pop. has changed since last
                // iteration, sort population
                {
                    sort_pop(); //Sort populations
                    find_popavg(); //Calculates population average
                }

            fprintf (file2, "%d %f %f %f\n", simreps, pvalue[0], popaverage, pvalue[popsize-1]); //writes continuous data to PROGRESS.DAT

            //Checks if new top 10% avg is better than last top 10%.
            find_topavg(); //Calculates average of top 10% of pop.
            converge_counter(); //Increments convergence counter if not
                // in top 10% of population

            //Creates a new individual from existing ones
            new_individ();

            summation = 0; //Zeros summation counter
            newaddition = 0; //Assumes new individual is worse

            //Evaluate new individual
            pvalue[popsize] = 0; //Zeros evaluation value for new individ.
            first = popsize; //First and last (only) individual to be
            last = popsize; //evaluated is the new individual
            evaluate(); //Runs SIMAN reps and calculates avg outputs

            //Check if new individual is better than worst individual in pop.
            if (pvalue[popsize] * max_or_min > pvalue[popsize-1] * max_or_min)
                {
                    //New individual is better than worst

                    }
newaddition = popsize - 1; // Since population is already sorted,
  // new individual is only individ.
  // that must be moved through pop.

// Insert new value into population as the last memmber of population.
// Sorting will be done at the beginning of the next iteration.
for (h = 0; h <= numints - 1; ++h)
  pint[popsize-1][h] = pint[popsize][h]; // replace worst's ints
for (h = 0; h <= numreals - 1; ++h)
  preal[popsize-1][h] = preal[popsize][h]; // replace worst's reals
for (h = 0; h <= numnons - 1; ++h)
  pnon[popsize-1][h] = pnon[popsize][h]; // replace worst's non-nums
pvalue[popsize-1] = pvalue[popsize]; // replace worst's fitness
}
while (converge < conpt); // If convergence counter is less than
  // convergence point, then perform
  // another iteration,
  // Else, GA stops.

clrscr();
printf("GA CONVERGENCE! 
");

if (pass == 1)
{
  resultfile = \"results1.dat\"; // After first GA iteration at 1 rep.
  results2file(); // Write results to a file

  // incr. rep. alt. #1
  // Evaluate pop. with 6 more reps. each to improve evaluation estimates
  if (location != 2)
    first = 0; // First individ. to evaluate is first in pop.
  location = 2; // If GA interrupted here, resuming will skip
    // running GA at 1 rep. and resume evaluating
    // at 6 reps.
  last = popsize - 1; // Last individ. to evaluate is last in pop.
  simancall = \"siman mod6\"; // Use MOD6.P for 6 replications
  numreps = 6; // Number of reps. to be run
  totalreps = 7; // 6 reps plus the existing evaluation at 1 rep.
    // gives a total of 7 replications each
  evaluate(); // Runs SIMAN reps and calculates avg outputs

  // Set up to run GA at 7 reps. each individual
  numreps = 7; // Number of reps. to be run
  location = 1; // If GA interrupted here, resuming will skip
    // to running GA at 7 reps.
  newaddition = 1; // Tells to sort entire pop.
  sort_pop(); // Sort population
  simancall = \"siman mod7\"; // Run MOD7.P file with SIMAN for 7 reps.
}

pass = pass + 1; // Counts number of passes through GA
  converge = 0; // Resets convergence counter
} while (pass < 3); // incr. rep. alt. #1 & 2
newaddition = 1; // Sort entire population
sort_pop(); // Sorts population
find_popavg(); // Calculates population average
find_topavg();  //Calculates average of top 10% of pop.
fprintf (file2, "$d %f %f $n", simreps, pvalue[0], popaverage,
pvalue[popsize-1]);
fclose(file2);  //Writes to and closes PROGRESS.DAT

//Create final report to screen and file.
for (i=0; i<=9; ++i)
    printf("individ #d evaluated to: %f $n", i+1, pvalue[i]);
for (i=0; i<=9; ++i)
    printf("%d simulation replications $n", simreps);
printf("Average of Top Ten Percent = $f $n", topaverage);

resultfile = "results7.dat";  //Results after eval pop. at 7 reps.
results2file();  //Writes results to a file
sound(1000);  //Beep to indicate GA completed
delay(250);
nosound();
exit (0);  //End of program

void read_gaspecs()
{
    //Read GA specifics from a file
    fileptr = fopen("gaspecs.dat","r");
    fscanf(fileptr, "%d", &max_or_min);  //Reads if the objective is to
     Min. or Max
    fscanf(fileptr, "%d", &numints);  //Reads number of integer vars.
    for (i=0; i<=numints-1; i++)
        {
            fscanf(fileptr, "%s", &intname[i]);  //Reads names of int. vars.
            fscanf(fileptr, "%d %d", &intlb[i], &intub[i]);  //int. bounds
        }
    fscanf(fileptr, "%d", &numreals);  //Reads number of real vars.
    for (i=0; i<=numreals-1; i++)
        {
            fscanf(fileptr, "%s", &realname[i]);  //Reads names of real vars.
            fscanf(fileptr, "%f %f", &reallb[i], &realub[i]);  //real bounds
            fscanf(fileptr, "%f", &smallinterval[i]);  //smallest increment
                // for real vars.
        }
    fscanf(fileptr, "%d", &numnons);  //Reads number of non-numeric vars.
    for (i=0; i<=numnons-1; i++)
        {
            fscanf(fileptr, "%s", &nonname[i]);  //names of non-numerics
            fscanf(fileptr, "%d", &nonalts[i]);  //number of alternatives
        }
    fscanf(fileptr, "%d", &numci);  //number of linear int. constraints
    for (i=0; i<=numci-1; i++)
        {
            for (j=0; j<=numints-1; j++)
                fscanf(fileptr, "%f", &ci[i][j]);  //constraint coefficients
            fscanf(fileptr, "%f", &cisign[i]);  //Reads int. corresponding
                // to constraint sign(i.e. <= )
            fscanf(fileptr, "%f", &rhs[i]);  //Reads Right-Hand-Side value
fscanf(fileptr, "%d", &numcr); //Reads # of linear real constraints
for (i=0; i<=numcr-1; i++)
{
    for (j=0; j<=numreals-1; j++)
        fscanf(fileptr, "%f", &cr[i][j]); //constraint coefficients
    fscanf(fileptr, "%d", &crsign[i]); //Reads int. corresponding to constraint sign (i.e. <=)
    fscanf(fileptr, "%f", &rhsr[i]); //Reads Right-Hand-Side value
}
fclose(fileptr); //Close GASPECS.DAT

void initial_pop()
{
    printf("Creating random initial population.\n"); //Create initial pop.
    for (i=0; i<=popsize-1; ++i)
        do
        { //Randomly create individs.
            for (j=0; j<=numints-1; ++j) //within variable bounds.
                pint[i][j] = random(intlb[j]-intlb[j]+1)+intlb[j];
            for (j=0; j<=numreals-1; ++j)
                preal[i][j] = reallb[j] + random((realub[j]-
                                           reallb[j]+smallinterval[j])) *
                                           smallinterval[j];
            for (j=0; j<=numnons-1; ++j)
                pnon[i][j] = random(nonalt[j]) + 1;
            anysame=0; //Zeros "identical individs" var.
            //Checks new individual for identical to any existing individs.
            // in the population. If identical, re-create the new individ.
            for (h=0; h<i-1; ++h)
                { //Check if new individual meets all constraints. If not, re-create it.
                    different=0; //Zeros "differences" counter
                    for (j=0; j<=numints-1; ++j)
                        if (pint[i][j] != pint[h][j]) ++different;
                    for (j=0; j<=numreals-1; ++j)
                        if (preal[i][j] != preal[h][j]) ++different;
                    for (j=0; j<=numnons-1; ++j)
                        if (pnon[i][j] != pnon[h][j]) ++different;
                    if (different==0) ++anysame; //If no differences
                                           //found, they are ident.
                }
            for (h=0; h<numci-1; ++h) //Check int. constraints met
                { //lincomb=0.0;
                    lincomb = 0.0;
                    for (j=0; j<=numints-1; ++j)
                        lincomb = lincomb + pint[i][j] * ci[h][j];
                    switch (crsign[h]+3) //Do test that corresponds to
                                         //constraint sign.
                        { case 1: if (lincomb>rhsi[h]) ++anysame;
                           break;
                          case 2: if (lincomb<rhsi[h]) ++anysame;
                           break;
                          case 3: if (lincomb=rhsi[h]) ++anysame;
                           break;
                          case 4: if (lincomb=rhsi[h]) ++anysame;
                           break;
                          case 5: if (lincomb<=rhsi[h]) ++anysame;
                           break;
                      }}}}
case 6: if (lincomb=rhsi[h]) ++anysame;
break;
}
for (h=0; h<numcr-1; ++h) //Check real constraints met
{
    lincomb=0.0;
    for (j=0; j<numreals-1; ++j)
        lincomb = lincomb + preal[i][j] * cr[h][j];
    switch (crsign[h]+3)
    {
    case 1: if (lincomb>=rhsr[h]) ++anysame;
            break;
    case 2: if (lincomb>rhsr[h]) ++anysame;
            break;
    case 3: if (lincomb!=rhsr[h]) ++anysame;
            break;
    case 4: if (lincomb<rhsr[h]) ++anysame;
            break;
    case 5: if (lincomb<=rhsr[h]) ++anysame;
            break;
    case 6: if (lincomb==rhsr[h]) ++anysame;
            break;
    }
}
while (anysame>0); //If the new individ. is identical
     // to an existing member of pop. or
     // if any constraint not met, discard
     // and create a new individ.

void evaluate()
{
    //Evaluate individuals (run simulation)
    for (i = first; i <= last; ++i)
    {
        //Write individual's parameter values to INPUT.DAT for SIMAN to use.
        fileptr = fopen("input.dat", "w");
        for (j=0; j<numints-1; j++)
            fprintf(fileptr, "%d\n", pint[i][j]);
        for (j=0; j<numreals-1; j++)
            fprintf(fileptr, "%f\n", preal[i][j]);
        for (j=0; j<numnons-1; j++)
            fprintf(fileptr, "%d\n", pnon[i][j]);
        fclose(fileptr);

        printf("Eval. Ind. %d: %d reps. for a total of %d\n", i+1,
               numreps, totalreps);
        system(simancall); //SIMAN runs whatever *.P file is specified
        // by simancall variable. SIMAN reads INPUT.
        // DAT file and incorporates parameter set-up
        // into the simulation replications. It then
        // writes the simulation output to OUTPUT.DAT
        // which corresponds to the evaluation of the
        // objective function at that particular
        // parameter configuration.
        //Read individual's evaluation from OUTPUT.DAT
        fileptr = fopen("output.dat", "r");
        for (j=0; j<numreps-1; ++j)
        {
            //Do stuff with fileptr
        }
        fclose(fileptr);
fscanf(fileptr, "%f", &garbage);
pvalue[i]=pvalue[i]+garbage;  //Add to existing fitness value
}
fclose(fileptr);
pvalue[i] = pvalue[i] / totalreps;  //Find mean over total reps.
printf("%d: %f at %d reps. for a total of %d
", i+1,
pvalue[i], numreps, totalreps);

simreps = simreps + numreps;  //Increments total reps. counter

interrupt_count = interrupt_count + numreps;
     //Increments Interrupt counter
if (interrupt_count > 300)  //If counter > 300, then write
     // INTERRUPT.DAT file
    {
        if (i+1 < last)
            first = i+1;
        else
            first = last;
        write_interupt();
        interrupt_count = 0;  //Zeros interrupt counter
    }
}

void sortyop()
{
    //Sort population
    for (i = newaddition; i <= popsize-1; ++i)
    {
        j=i;
        do {
            if (pvalue[j]*max_or_min>pvalue[j-1]*max_or_min)
            {
                for(h=0; h<=numints-1; ++h)
                {
                    tempint=pint[j][h];
pint[j][h]=pint[j-1][h];
pint[j-1][h]=tempint;
                }
                for(h=0; h<=numreals-1; ++h)
                {
                    garbage=preal[j][h];
preal[j][h]=preal[j-1][h];
preal[j-1][h]=garbage;
                }
                for(h=0; h<=numnons-1; ++h)
                {
                    tempint=pnon[j][h];
pnon[j][h]=pnon[j-1][h];
pnon[j-1][h]=tempint;
                }
                garbage=pvalue[j];
pvalue[j]=pvalue[j-1];
pvalue[j-1]=garbage;
                j=j-1;
            } else j=0;
        } while (j>0);
void find_topavg()
{
    summation=0;
    for (i=0; i<=toppop-1; ++i)
        summation=summation+pvalue[i];
    topaverage = summation / toppop;
}

void find_popavg()
{
    summation = 0;
    for (i=0; i<=popsize-1; ++i)
        summation = summation + pvalue[i];
    popaverage = summation / popsize;
}

void converge_counter()
{
    // Checks for a significant improvement in the top 10% of the pop.
    // Report GA convergence if 100 GA iterations w/out sig. improvement
    printf("topAvg: %4.3f ", topaverage);
    printf("%d %d ", simreps, converge);
    printf("%4.3f - %4.3f\n", pvalue[0], pvalue[toppop]);
    if (fabs(topaverage-lastavg)<fabs(lastavg/1000)) ++converge;
        // Only an improvement of over 1/1000
        // of the lastavg will be considered
        // as a significant improvement.
        // If no sig. improvement, increment
        // convergence counter.
    else
    {
        converge=0; // If sig. improvement occurred
        lastavg=topaverage; // Assign new top avg to last avg
    }
}

void new_individ()
{
    // Creates new individual using existing individuals and genetic ops.
    do {
        numparents=1; // Default is use only one parent
        crossover=(float)rand()/32767; // Random number (0-1)
        if (crossover<crossrate) numparents=2;
            // If number is less
            // than crossover rate, then
            // crossover will occur so 2
            // parents will be needed.
        // Select parent(s) by tournament selection w/ variable tournament size
        for (i=0; i<numparents-1; ++i)
        {
            // Generate tournament size with 25% prob. size = 1 individuals,
            // 50% prob. size = 2 individuals,
            // 25% prob. size = 3 individuals.
            // ...
numtourn=random(4)+1;
if (numtourn>2) numtourn=numtourn-1;

bestindivid=popsize;
   //Start with worst in pop. being
   //the one to beat in tournament
for(j=1; j<=numtourn; ++j) //Select individuals
   //for tournament
   {
      individ=random(popsize);//int.(0 to popsize-1)
      if (individ<bestindivid) //If individ is better
         //than current best,
         //then replace best.
         {
            parent[i]=individ;
            bestindivid=individ;
         }
   }

//Default crossover-point at the end of parameter (no crossover)
crosspoint=numints+numreals+numnons;

if (numparents==2) //2 parents means do crossover, so
   {
      //randomly select crossover point
crosspoint=random(numints+numreals+numnons-1)+1;
   }

//If no crossover, the crossover of one parent with crossover point
//at the end of the variable list assigns all the parents values
//to the new individual.
currentpoint=0; //Zeros location along variable list
whichparent=0; //Take first values from first parent.
for (i=0; i<=numints-1; ++i)
   {
      if (currentpoint==crosspoint) whichparent=1;
      //If current point = crossover point, take
      //the rest of parameters from second parent.
pint[popsize][i]=pint[parent[whichparent]][i];
currentpoint++;
      //increment location along variable list
   }

for (i=0; i<=numreals-1; ++i)
   {
      if (currentpoint==crosspoint) whichparent=1;
preal[popsize][i]=preal[parent[whichparent]][i];
currentpoint++;
   }

for (i=0; i<=numnons-1; ++i)
   {
      if (currentpoint==crosspoint) whichparent=1;
pnon[popsize][i]=pnon[parent[whichparent]][i];
currentpoint++;
   }

//If no crossover occurred (1 parent), perform creep & mutate ops.
if (numparents==1)
   {
      for (i=0; i<=numints-1; ++i)
{mutate=(float)rand()/32767; //random number(0-1)
if (mutate<mutaterate) //If mutation, randomly
    // generate new value within
    // variable bounds.
{
pint[popsize][i]=random(intub[i]-
    intlb[i]+1)+intlb[i];
} else
{
    creep=(float)rand()/32767; //random number(0-1)
    if (creep<creeprate)
    {
        intrange=intub[i]-intlb[i]+1; //range of var.
        // bounds.
        creep=(float)rand()/32767; //random number(0-1)
        creep=creep*(intrange*.1)*(conpt-
        converge)/conpt+1;
        //Amount of creep can at max be 10% of range and
        // creep distance decreases as GA nears
        // convergence.
        intcreep=int(creep); //Transform creep to int.
    }
    else
    {
        creep=(float)rand()/32767; //random number(0-1)
        if (creep<creeprate)
        {
            intrange=intub[i]-intlb[i]+1; //range of var.
            // bounds.
            creep=(float)rand()/32767; //random number(0-1)
            creep=creep*(intrange*.1)*(conpt-
            converge)/conpt+1;
            //Amount of creep can at max be 10% of range and
            // creep distance decreases as GA nears
            // convergence.
            intcreep=int(creep); //Transform creep to int.
        }
        else
        {
            creep=(float)rand()/32767; //random number(0-1)
            if (creep<creeprate)
            {
                realrange=realub[i]-reallb[i];
                creep=random((realrange*.1*(conpt-
                converge)/conpt+smallinterval[i])
                /smallinterval[i])*smallinterval[i];
                direction=random(2);
                if (direction==0) direction=-1;
                preal[popsize][i]=real[popsize][i]+creep* direction;
                preal[popsize][i]=(float) int(preal[popsize][i]/
                smallinterval[i])*smallinterval[i];
            }
            else
            {
                creep=(float)rand()/32767; //random number(0-1)
                if (creep<creeprate)
                {
                    realrange=realub[i]-reallb[i];
                    creep=random((realrange*.1*(conpt-
                    converge)/conpt+smallinterval[i])
                    /smallinterval[i])*smallinterval[i];
                    direction=random(2);
                    if (direction==0) direction=-1;
                    preal[popsize][i]=real[popsize][i]+creep* direction;
                    preal[popsize][i]=(float) int(preal[popsize][i]/
                    smallinterval[i])*smallinterval[i];
                }
            }
        }
    }
} //Only preform mutation on non-numeric variables. Creep only for
// incremental variables and is not suited for un-ranked alternatives.
for (i=0; i<=numnons-1; ++i)
{
    mutate=(float)rand()/32767;
    if (mutate<creepRate)
    {
        pnon[popsize][i]=random(nonals[i])+1;
    }
}

//Check newly created individual for being identical and infeasible.
// Similar to checking procedure in INITIAL_POP function.
for (h=0; h<i-1; ++h) //Check for identical individs
{
    different=0;
    for (j=0; j<numints-1; ++j)
    {
        if (pint[i][j]!=pint[h][j]) ++different;
    }
    if (different==0) ++anysame;
}

for (h=0; h<numreals-1; ++h) //Check w/in variable bounds
{
    if (pint[i][h]>intub[h]) ++anysame;
    if (pint[i][h]<intlb[h]) ++anysame;
}

for (h=0; h<numnons-1; ++h)
{
    if (pnon[i][h]>nonals[h]) ++anysame;
    if (pnon[i][h]<1) ++anysame;
}

for (h=0; h<numci-1; ++h) //Check int constraints
{
    lincomb=0.0;
    for (j=0; j<numints-1; ++j)
        lincomb = lincomb + pint[i][j] * ci[h][j];
    switch (cisign[h]+3)
    {      case 1: if (lincomb>=rhsi[h]) ++anysame; break;
        case 2: if (lincomb>rhsi[h]) ++anysame; break;
        case 3: if (lincomb!=rhsi[h]) ++anysame; break;
        case 4: if (lincomb<rhsi[h]) ++anysame; break;
    }  

case 5: if (lincomb<=rhsi[h]) ++anysame;
    break;
case 6: if (lincomb==rhsi[h]) ++anysame;
    break;
}
for (h=0; h<=numcr-1; ++h) //Check real constraints
{
    lincomb=0.0;
    for (j=0; j<=numreals-1; ++j)
        lincomb = lincomb + preal[i][j] * cr[h][j];
    switch (crsign[h]+3)
    {
        case 1: if (lincomb>=rhsr[h]) ++anysame;
            break;
        case 2: if (lincomb>rhsr[h]) ++anysame;
            break;
        case 3: if (lincomb!=rhsr[h]) ++anysame;
            break;
        case 4: if (lincomb<rhsr[h]) ++anysame;
            break;
        case 5: if (lincomb<=rhsr[h]) ++anysame;
            break;
        case 6: if (lincomb==rhsr[h]) ++anysame;
            break;
    }
} while (anysame>0); //If new individual is identical or
// infeasible, discard and create a
// new individual.

void pop2file()
{
    //Writes entire population to a file to use if an interruption occurs.
    file2 = fopen(resultfile, "w");
    for (i=0; i<=popsize-1; ++i)
    {
        fprintf(file2, "%9.3f ", pvalue[i]);
        for (j=0; j<=numints-1; j++)
            fprintf(file2, "%d ", pint[i][j]);
        for (j=0; j<=numreals-1; j++)
            fprintf(file2, "%7.3f ", preal[i][j]);
        for (j=0; j<=numnons-1; j++)
            fprintf(file2, "%d ", pnon[i][j]);
        fprintf(file2, "\n");
    }
    fclose (file2);
}

void results2file()
{
    //Writes GA results to a file
    fileptr = fopen(resultfile, "w");
    for (i=0; i<=popsize-1; ++i)
    {
        fprintf(fileptr, "%f: ", pvalue[i]);
        for (j=0; j<=numints-1; ++j)
            fprintf(fileptr, "%d ", pvalue[i][j]);
        for (j=0; j<=numreals-1; ++j)
            fprintf(fileptr, "%7.3f ", preal[i][j]);
        for (j=0; j<=numnons-1; ++j)
            fprintf(fileptr, "%d ", pnon[i][j]);
        fprintf(fileptr, "\n");
    }
    fclose (fileptr);
}
fprintf(fileptr, "%d Simulation replications.\n", simreps);
for (j=0; j<=numreals-1; ++j)
    fprintf(fileptr, "best => %f\n", pvalue[j]);
for (j=0; j<numreals-1; ++j)
    fprintf(fileptr, " %7.3f ", preal[j]);
for (j=0; j<=numnons-1; ++j)
    fprintf(fileptr, "non-num%d %d\n", j, pnon[j]);
fclose(fileptr);

void write_interupt()
{
    // Writes INTERUPT.DAT file to resume an interupted optimization.
    fileptr = fopen("interrupt.dat", "w");
    for (h=0; h<pops-1; ++h)
    {
        fprintf(fileptr, "%9.3f pvalue[h]);
        for (j=0; j<=numints-1; j++)
            fprintf(fileptr, "%d ", pint[h][j]);
        for (j=0; j<=numreals-1; j++)
            fprintf(fileptr, "%7.3f ", preal[h][j]);
        for (j=0; j<=numnons-1; j++)
            fprintf(fileptr, "%d ", pnon[h][j]);
        fprintf(fileptr, "\n");
    }
    fprintf(fileptr, "%d \t %d \
", location, pass);
    fprintf(fileptr, "%d \t %d \t %d \t %d \
", converge, comp, simreps, popsize);
    fprintf(fileptr, "%d \t %d \t %d \t %d \
", first, last, numreps, totalreps);
    fprintf(fileptr, "%f \t %f \t %f \
", popaverage, topaverage, lastavg);
    fclose(fileptr);
}

void read_interupt()
{
    // Reads INTERUPT.DAT to resume an optimization in-progress.
    fileptr = fopen("interrupt.dat", "r");
    for (h=0; h<pops-1; ++h)
    {
        fscanf(fileptr, "%f", &pvalue[h]);
        for (j=0; j<=numints-1; j++)
            fscanf(fileptr, "%d ", &pint[h][j]);
for (j=0; j<numreals-1; j++)
    fscanf(fileptr, "%f", &preal[h][j]);
for (j=0; j<numnons-1; j++)
    fscanf(fileptr, "%d", &pnon[h][j]);
}
    fscanf(fileptr, "%d %d", &location, &pass);
    fscanf(fileptr, "%d %d %d %d", &converge, &conpt, &simreps, &popsize);
    fscanf(fileptr, "%d %d %d", &first, &last, &numreps, &totalreps);
    fscanf(fileptr, "%f %f %f", &popaverage, &topaverage, &lastavg);
    fscanf(fileptr, "%f", &std_dev);
fclose(fileptr);
void calc_std_dev()
{
    //For use with GA convergence based on std. deviations and confidence
    // intervals. This function only used for the 3-sigma stopping rule.
    std_dev = 0;
    for (i = first; i <= last; ++i)
    {
        fileptr = fopen("input.dat","w");
        for (j=0; j<numints-1; j++)
            fprintf(fileptr, "%d\n", pint[i][j]);
        for (j=0; j<numreals-1; j++)
            fprintf(fileptr, "%f\n", preal[i][j]);
        for (j=0; j<numnons-1; j++)
            fprintf(fileptr, "%d\n", pnon[i][j]);
        fclose(fileptr);
        system(simancall);
        fileptr = fopen("output.dat","r");
        mean[0] = pvalue[i];
        avgmean = mean[0];
        for (j=1; j<=numtrials-1; ++j)
        {
            mean[j] = 0;
            for (k=1; k<=numreps; ++k)
            {
                fscanf(fileptr, "%f", &garbage);
                mean[j] = mean[j] + garbage;
            }
            mean[j] = mean[j] / numreps;
            avgmean = avgmean + mean[j];
        }
        fclose(fileptr);
        avgmean = avgmean / numtrials; //find mean over total reps.
        printf("Individual %d evaluated to %f.\n\n", i+1, pvalue[i]);
        variance = 0;
        for (j=0; j<=numtrials-1; ++j)
        {
            printf("%f\n", mean[j]);
            variance = variance + (mean[j]-avgmean)*(mean[j]-avgmean);
        }
        fclose(fileptr);
        variance = variance / (numtrials-1);
```
printf("variance = \%f\n", variance);
printf("std_dev = \%f\n\n", sqrt(variance));
getch();
std_dev = std_dev + sqrt(variance);
simreps = simreps + numtrials*numreps - numreps;
interrupt_count = interrupt_count + numtrials*numreps - numreps;
}
std_dev = std_dev / (last + 1);
printf("std_dev = \%f\n\n", std_dev);
getch();
```
A.2 Simulation Genetic Algorithm Initialization Program (GAINIT.CPP)

// Steady-State Genetic Algorithm for Simulation Optimization
// Initialization Program
// This program is executed from within GA.EXE
// Developed by Steve Harris

#include "stdio.h"
#include "conio.h"
#include "stdlib.h"
#include "math.h"
#include "process.h"

#define maxints 11 //max # of integer dec vars
#define maxreals 11 //max # of real dec vars
#define maxnons 6 //max # of non-numeric dec vars
#define maxcis 5 //max # of integer constraints
#define maxcrs 5 //max # of real constraints

int max_or_min; // +1 for maximize, -1 for minimize
int i, j; // counting indices
int numints, numreals, numnons; // # of int, real, & non-numeric vars.
int numci, numcr; // # of int and real constraints
int intlb[maxints], intub[maxints]; // int upper and lower bounds
int nonalts[maxnons]; // array of non-numeric of # of alternatives
int cisign[maxcis], crsign[maxcrs]; // constraint signs, -2:<,-1:<=,0:==,1:>=,2:,3:=
char intname[15][maxints], realname[15][maxreals];
char nonname[15][maxnons];
char okay; // for y and n choices
float garbage; // dummy float var
float reallb[maxreals], realub[maxreals]; // real up and low bounds
float smallinterval[maxreals]; // smallest interval for real variables
float ci[maxcis][maxints]; // int constraint coefficients
float cr[maxcrs][maxreals]; // real constraint coefficients
float rhsi[maxcis]; // int constraint right-hand-sides
float rhr[maxcrs]; // real constraint right-hand-sides
FILE *fileptr;

void main()
{
    // GA Initialization
    printf("TEST: enter a float point number: ");
    scanf("%f", &garbage);
    okay='y';
    clrscr();
    printf("Steady State GA for Simulation Optimization

    " );
    do {
        printf("Maximize(+1) or Minimize(-1) Simulation Output? ");
        scanf("%d", &max_or_min);
        } while(abs(max_or_min)!=1);
    do {
        clrscr();
        printf("\nDECISION VARIABLES\n");
        printf("***************\n");
        do {
printf("How many integer variables: ");
scanf("%d", &numints);
do
{
    printf("How many integer constraints: ");
    scanf("%d", &numci);
} while (numci>maxcis);
do
{
    printf("How many non-numeric variables: ");
    scanf("%d", &numnons);
} while (numnons>maxnons);
clrscr();
printf("DECISION VARIABLE SETTINGS\n");
printf("- - - - - - - - - - - - - - - - - - - - - - - - - -\n");
printf("%d into vars. with %d constraints.\n",numints,numci);
printf("%d real vars. with %d constraints.\n",numreals,numcr);
printf("%d non-numeric vars.\n",numnons);
printf("Is this correct (y or n)? ");
scanf("%s", &okay);
do
{
clrscr();
for (i=0; i<=numints-1; i++)
{
    printf("Name of integer variable %d: ",i);
    scanf("%s", &intname[i]);
do
    {
        printf(" lower bound: ");
        scanf("%d", &intlb[i]);
        printf(" upper bound: ");
        scanf("%d", &intub[i]);
        printf("\n");
    } while(intlb[i]>intub[i]);
}
printf("\n");
for (i=0; i<=numreals-1; i++)
{
    printf("Name of real-valued variable %d: ",i);
    scanf("%s", &realname[i]);
do
    {
        printf(" lower bound: ");
        scanf("%f", &reallb[i]);
        printf(" upper bound: ");
        scanf("%f", &realub[i]);
        printf("\n");
    } while (reallb[i]>realub[i]);
do
    {
        printf(" smallest interval: ");
        scanf("%f", &smallinterval[i]);
    } while ((realub[i]-reallb[i]+smallinterval[i])
}
for (i=0; i<=numnons-1; i++)
{
    printf("\nName of non-numeric variable %d: ", i);
    scanf("%s", &nonname[i]);
    printf(" number of alternatives: ");
    scanf("%d", &nonalts[i]);
}
clsscr();
printf("VARIABLES AND RANGES\n-----------------
\n")
for (i=0; i<=numints-1; i++)
    printf("Integer variable %d: %s from %d to %d
", i, intname[i], intlb[i], intub[i]);
for (i=0; i<=numreals-1; i++)
    printf("Real-valued variable %d: %s from %f to %f
", i, realname[i], reallb[i], realub[i]);
for (i=0; i<=numnons-1; i++)
    printf("Non-numeric variable %d: %s with %d
alternatives\n", i, nonname[i], nonalts[i]);
printf("\nIs this correct (y or n)? ");
scanf("%s", &okay);
} while (okay!='y');
if (numci>0)
{
    do {
        for (i=0; i<=numci-1; i++)
        {
            clrsr();
            printf("CONSTRAINTS\n-------------\n\n")
            printf("Integer Constraint %d\n",i);
            for (j=0; j<=numints-1; j++)
            {
                printf("Coefficient of %s:",intname[j]);
                scanf("%f", &ci[i][j]);
            }
            printf("\nConstraint Sign (-2 for <, -1 for <=, 0 for =, 1 for >=, 2 for >, 3 for !=): ");
            scanf("%d", &cisign[i]);
            printf("\nRight-hand-side of constraint: ");
            scanf("%f", &rhsi[i]);
        }
    clrsr();
    printf("CONSTRAINTS\n-------------\n")
    printf("Integer Constraints\n")
    for (i=0; i<=numci-1; i++)
    {
        for (j=0; j<=numints-1; j++)
            printf("+(%5.2f)i%d ", ci[i][j], j);
        switch (cisign[i]+3)
        {
        case 1 :printf("< ");
            break;
        case 2 :printf("<= ");
            break;
        case 3 :printf("= ");
            break;
        case 4 :printf("> ");
            break;
        case 5 :printf(">= ");
            break;
        case 6 :printf("!= ");
            break;
        case 7 :printf("== ");
            break;
        case 8 :printf("!= ");
            break;
        default: printf("Invalid sign\n");
        
        printf("\n\nRight-hand-side of constraint: ");
        scanf("%f", &rhsi[i]);
    }
case 4 : printf(">= ");
break;
case 5 : printf("> ");
break;
case 6 : printf("!= ");
break;
}
printf("%5.2f\n", rhs1[i]);
}
printf("\nIs this correct (y or n)? ");
scanf("%s", &okay);
} while (okay!='y');
if (numcr>0)
{
  do {
    clrscr();
    printf("CONSTRAINTS\n------------\n")
    printf("Real-Valued Constraint \dn", i);
    for (j=0; j<=numreals-1; j++)
    {
      printf("Coefficient of %s:", realname[j]);
      scanf("%f", &cr[i][j]);
    }
    printf("\nConstraint Sign (-2 for <, -1 for <=, 0 for =, 1 for >=, 2 for >, 3 for !=): ");
    scanf("%d", &crsign[i]);
    printf("\nRight-hand-side of constraint: ");
    scanf("%f", &rhrs[i]);
  }
    clrscr();
    printf("CONSTRAINTS\n------------\n")
    printf("Real-Valued Constraints\n")
    printf("\n")
    for (i=0; i<=numcr-1; i++)
    {
        for (j=0; j<=numreals-1; j++)
        printf("+(%5.2f)r%d", cr[i][j], j);
        switch (crsign[i]+3)
        {
        case 1 : printf("< ");
                  break;
        case 2 : printf("<= ");
                  break;
        case 3 : printf("= ");
                  break;
        case 4 : printf(">= ");
                  break;
        case 5 : printf("> ");
                  break;
        case 6 : printf("!= ");
                  break;
        }
    printf("%5.2f\n", rhrs[i]);
    }
    printf("\nIs this correct (y or n)? ");
    scanf("%s", &okay);
while (okay!='y');

// Write all initialization data to GASPECS.DAT file
fileptr = fopen("gaspecs.dat","w");
fprintf(fileptr, "%d\n", max_or_min);
fprintf(fileptr, "%d\n", numints);
for (i=0; i<=numints-1; i++)
{
    fprintf(fileptr, "%s\n", intname[i]);
    fprintf(fileptr, "%d\n", intlb[i], intub[i]);
}
fprintf(fileptr, "\n%d\n", numreals);
for (i=0; i<=numreals-1; i++)
{
    fprintf(fileptr, "%s\n", realname[i]);
    fprintf(fileptr, "%f %f\n", reallb[i], realub[i]);
    fprintf(fileptr, "%f \n", smallinterval[i]);
}
fprintf(fileptr, "\n%d\n", numnons);
for (i=0; i<=numnons-1; i++)
{
    fprintf(fileptr, "%s\n", noname[i]);
    fprintf(fileptr, "%d\n", nonalts[i]);
}
fprintf(fileptr, "\n%d\n", numci);
for (i=0; i<=numci-1; i++)
{
    for (j=0; j<=numints-1; j++)
        fprintf(fileptr, "%f \n", ci[i][j]);
    fprintf(fileptr, "%d\n", cisign[i]);
    fprintf(fileptr, "%f\n", rhsi[i]);
}
fprintf(fileptr, "\n%d\n", numcr);
for (i=0; i<=numcr-1; i++)
{
    for (j=0; j<=numreals-1; j++)
        fprintf(fileptr, "%f \n", cr[i][j]);
    fprintf(fileptr, "%d\n", crsign[i]);
    fprintf(fileptr, "%f\n", rshr[i]);
}
fclose(fileptr);
exit (0);
A.3 Simulation Simulated Annealing Algorithm Program (ANJS.CPP)

// ANJS.cpp --> simulated annealing V1.0
// For Jobshop problem
// Modified Date: 2/1/95
// Developed by Pornthep Annussornnitisarn

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

#define max 21
#define numint 10
#define numreal 10
#define numnon 1
#define maxrep 10

float tempi, tempf;
float alpha;
int ubc[max], lbc[max];
float breal[numreal];
float y, yran;
int x[max], xran[max];
int runnum = 0;
int repreun = 1;
int k;
float prob1, prob2;
int accept(float e, float t);
int ekconst(void);
void genmv(void);
float energy(void);
//float f(int *xx);
void copyxy(void);

void main()
{
    int i;
    float temp, delta;
    float result;
    void getinitv(void);
    void sendinfo(int *xx);
    float output(void);
    void report(int step, float t);
    void stop(void);

    ubc[20] = 3;
    lbc[15] = 0; lbc[16] = 0; lbc[17] = 0; lbc[18] = 0; lbc[19] = 0;
}
lbc[20]= 0;

randomize();
getinitv();
temp = tempi;
sendinfo(x);

system("siman jpl1");
result = output();
y = -1*result;
report(1,temp);
runnum++;

runnum = runnum + (reprun - 1);
//y = 3358.2;
do {
    i = 1;

do {
    genmv();
sendinfo(xran);
    printf("runnum = %d, Temp = %f\n",runnum,temp);
    if (reprun == 1) system("siman jpl1");
    else if (reprun == 4) system("siman jpl4");
    else if (reprun == 7) system("siman jpl7");
    if (kbhit() != 0) stop();
    runnum = runnum + (reprun - 1);
    yran = -1*output();
    delta = energy();
    if (delta < 0)
        {
        copyxy();
        report(2,temp);
        }
    else
        {
        if (accept(delta, temp) == 1)
            {
            copyxy();
            report(3,temp);
            }
        else report(4,temp);
        }
    if (kbhit() != 0) stop();
    i++;
}while (i <= k);

temp = alpha*temp;
if (temp <= tempi*0.2) reprun = 4;
if (temp <= tempi*0.2*0.1) reprun = 7;
}while (temp > tempf);

void getinitv(void)
{
    int i;
FILE *fp;
fp = fopen("control.dat","r");
scanf(fp,"%f", &tempi);
scanf(fp,"%f", &tempf);
scanf(fp,"%f", &alpha);
scanf(fp,"%d", &k);
fclose(fp);
fp = fopen("anjs1.dat","r");
for (i = 0; i < max; i++)
  fscanf(fp,"%d", &x[i]);
fclose(fp);
}

void genmv(void)
{
  int i, zero;
  int step, mv, ok;
  ok = 0;
do
  {
    zero = 0;
    for (i = 0; i < max; i++)
      {
        step = 3;
        mv = random(step);
        if (mv == 0)  xran[i] = x[i] - 1;
        if (mv == 1)
            {
              xran[i] = x[i];
              zero++;
            }
        if (mv == 2)  xran[i] = x[i] + 1;
        if (xran[i] > ubc[i])  xran[i] = ubc[i];
        if (xran[i] < lbc[i])  xran[i] = lbc[i];
        if (zero >= max)
          {
            i = -1;
            zero = 0;
          }
      }
    ok = ckconst();
  }while(ok != 0);
}

float energy(void)
{
  float diff;
  diff = yran - y;
  return(diff);
}

int accept(float e, float t)
{
  float a,b;
  int i;
  a = random(100);
  b = 100 * exp((-1*e)/t);
  if (b < 1) b = 0;
  printf("\n! ! ! ! ! prob accept = %5.2f ; a = %5.2f", b, a);
printf(" at TEMP %5.1f\n", t);
    if (a < b) i = 1;
    else i = 0;
    prob1 = b;
    prob2 = a;
    return(i);
}

/*float f(int *xx)
{
    int xl, x2;
    float y;
    xl = *xx;
    xx = xx + 1;
    x2 = *xx;
    y = -1*((2*x1*x2) + (2*x2) - (x1*x1) - (2*x2*x2));
    //printf("\n!! %d %d %f\n",xl,x2,yy);
    return(yy);
} */

void copyxy(void)
{
    int i,
    y = yran;
    for (i = 0; i < max; i++) x[i] = xran[i];
}

void sendinfo(int *xx)
{
    int i;
    FILE *fp;
    breal[0] = 0.25; breal[1] = 0.75; breal[2] = 0.75; breal[3] = 0.25;
    breal[4] = 0.75; breal[5] = 0.75; breal[6] = 0.75; breal[7] = 0.75;
    breal[8] = 0.00; breal[9] = 0.00;
    fp = fopen("input.dat","w");
    for (i = 0; i < numint; i++)
    {
        fprintf(fp,"%2d ", *xx);
        xx = xx + 1;
    }
    fprintf(fp, "\n");
    for (i = 0; i < numreal; i++)
    {
        fprintf(fp,"%5.3f ",breal[i] + (*xx)*0.025);
        xx = xx + 1;
    }
    fprintf(fp, "\n");
    for (i = 0; i < numnon; i++)
    {
        fprintf(fp,"%2d ", *xx);
        xx = xx + 1;
    }
    fclose(fp);
}

float output(void)
{
    float out[maxrep], sum, result;
}
int i;
FILE *fp;
fp = fopen("output.dat", "r");
sum = 0.0;
for (i = 0; i < reprun; i++)
{
    fscanf(fp, "%f", &out[i]);
    sum = sum + out[i];
}
fclose(fp);
result = sum/reprun;
return(result);
}

int ckconst(void)
{
    int sum;
    int coni1, coni2, coni3, coni4;
    float conr1;
    float r1, r2;
    sum = 0;
    coni3 = xran[8] + xran[9];
    r1 = break[8] + xran[18] * 0.025;
    r2 = break[9] + xran[19] * 0.025;
    conr1 = r1 + r2;
    if (coni1 < 1) sum++;
    if (coni2 < 1) sum++;
    if (coni3 < 1) sum++;
    if (coni4 > 12) sum++;
    if (conr1 > 1) sum++;
    if (sum > 0)
    {
        int i;
        printf("\n");
        for (i = 0; i < max; i++) printf("%2d," , xran[i]);
        printf("\n");
        printf("%d, %d, %d, %d, %f\n", coni1, coni2, coni3, coni4, conr1);
    }
    return(sum);
}

void report(int step, float t)
{
    int i;
    FILE *fp;
    if (runnum < 1)
    {
        fp = fopen("report.an4", "w");
        fprintf(fp, "Simrun#	Solution		Obj		Temp\n");
        fprintf(fp, "--------------------------\n");
        fprintf(fp, "\n");
        fprintf(fp, "%3d\t\tO\t\t\t", runnum);
        for (i = 0; i < max; i++)
        {

fprintf(fp,"%2d, ",x[i]);
}  
fprintf(fp,"\t%7.1f\t\t%4.1f\t\n",-1*y,t);
fclose(fp);
}
else  
{
fp = fopen("report.an4","a");
if (step == 2)
{
fprintf(fp,"%3d\t\(\",runnum);
for (i = 0; i < max; i++)
{
fprintf(fp,"%2d, ",x[i]);
}
fprintf(fp,"\t%7.1f\t\t%4.1f\t\n",-1*y,t);
}
if (step == 3)
{
fprintf(fp,"%3d\t\(\",runnum);
for (i = 0; i < max; i++)
{
fprintf(fp,"%2d, ",x[i]);
}
fprintf(fp,"\t%7.1f\t\t%4.1f\t%3.0f %3.0f\n",-1*y,t,probl,prob2);
}
if (step == 4)
{
fprintf(fp,"%3d\t\(\",runnum);
for (i = 0; i < max; i++)
{
fprintf(fp,"%2d, ",xran[i]);
}
fprintf(fp,"\t%7.1f\t\t%4.1f\t%3.0f %3.0f\n",-1*yran,t,probl,prob2);
}
fclose(fp);
}
}

void stop(void)
{ char ch;
  ch = getch();
  if (ch == 'y') exit(1);
}
Appendix B
Basic Test Problems in Coded for SIMAN

B.1.1 Restaurant Problem Model File

BEGIN;
;READ RESOURCE DATA FROM DATA FILE
CREATE;
READ, INVALUE: MR(1), MR(2), MR(3);
CLOSE, INVALUE: DISPOSE;

;WRITE OUTPUT TO A FILE
CREATE, 1, TFIN-.001;
BRANCH, 1: IF, NQ(Waiterq) > 0, notenough:
          ELSE, enough;
notenough ASSIGN: profit=0;
enough WRITE, OUTVALUE: profit: DISPOSE;

;SAMPLE PROBLEM 3.4

CREATE: EXPONENTIAL(enter, 1), Door; Create parties
BRANCH, 1:
          IF, TNOW <= 240, checktime:
          ELSE, Closed; Open or Closed?
Closed ASSIGN: Door = 0: DISPOSE; Set MaxBatches to 0

checktime BRANCH, 1:
          IF, TNOW < 60, 5to6:
          IF, TNOW < 120, 6to7:
          ELSE, 7to9;
5to6 ASSIGN: enter=1.6; NEXT(Open); sets 5-6 entry dist
6to7 ASSIGN: enter=1.0; NEXT(Open); "6-7" " "
7to9 ASSIGN: enter=1.6; "7-9" " 
Open ASSIGN:
          Ps=DISCRETE(.4, 2,
          .7, 3,
          .9, 4,
          1, 5, 2): !party size 2 (40%)
          !party size 3 (30%)
          !party size 4 (20%)
          !party size 5 (10%)
income=income+Ps*5:
profit=income-(25*MR(3))-(40*MR(2))-(MR(1)-50)*10;
QUEUE, TableQ, 5, Leave; wait for table
SEIZE, 2: Table,(Ps+1)/2: !get tables
          . Cashier; and host(cashier)
DELAY: 1.0; time to be seated
RELEASE: Cashier; release host
QUEUE, WaiterQ; wait for avail. waiter
SEIZE: Waiter;  get waiter
DELAY: TRIANGULAR(14,19,24,3) + !delay by service
       NORMAL(24,5,4);  delay by dining
RELEASE: Table,(Ps+1)/2: !release tables
          Waiter;

QUEUE,  CashierQ;
SEIZE,1: Cashier;  wait for cashier
DELAY: NORMAL(1.5,.5,5);  seize cashier
delay to pay bill
RELEASE: Cashier;  release cashier
COUNT: Served Parties:DISPOSE;  count served cust.

COUNT: Lost Parties:DISPOSE;  count balked cust.
B.1.2 Restaurant Problem Experiment File

BEGIN;
RESOURCES:  Table,63:
           Waiter,25:
           Cashier,2;

VARIABLES:  enter,1.6:
            income:
            profit:
            Door,1.e4;

QUEUES:    TableQ:
           WaiterQ:
           CashierQ;

COUNTERS:  Lost Parties:
           Served Parties;

ATTRIBUTES:  Ps;

OUTPUTS:   profit,,Profit_ $;

REPLICATE,  1,0,300,Yes,Yes,0;

FILES:INVALUE, "INVALUE.DAT", SEQ,FRE:
    OUTVALUE,"OUTVALUE.DAT",SEQ,FRE;

SEEDS:1, 54321:
    2, 76543:
    3, 98765:
    4, 10987:
    5, 32109;
END;
B.2.1 Distribution Problem 1 Model File

; 1DP.MOD
; Date modified: 11/1/94
; Note: Original problem with 5 decision variables

BEGIN;
;READ DATA FROM INPUT.DAT
CREATE;
READ, INVALUE: LEVEL1, LEVEL2, LEVEL3, LEVEL4,
SHIPSIZE;
CLOSE, INVALUE: DISPOSE;

;WRITE COST TO A OUTPUT.DAT
CREATE, 1, TFIN;
ASSIGN: C1 = 100 * NC(LOST_SALES) + 10 * NC(SHIPPED):
C2 = 5 * (DAVG(1)+DAVG(2));
C3 = 10 * (DAVG(3)+DAVG(4)+DAVG(5));
C4 = 15 * (DAVG(6)+DAVG(7));
C5 = 30 * DAVG(8);
WRITE, OUTVALUE: C1+C2+C3+C4+C5: DISPOSE;

;CUSTOMER DEMAND SUBMODEL
CREATE: EXPO(20,5);
QUEUE, DEMAND;
BRANCH,1: IF, NQ(BUF4).LE.0,LOST:
IF, NQ(BUF4).GT.0,GAIN;

LOST COUNT: LOST_SALES:DISPOSE;
GAIN REMOVE: 1,BUF4,LEAVE:DISPOSE;
LEAVE COUNT: SALES:DISPOSE;

;MAIN MODEL
CREATE;
SUPP QUEUE,
SCAN: (NQ(BUF1) + ORDER) .LE. LEVEL1;
ASSIGN: ORDER = ORDER + 1;
BRANCH,2: ALWAYS,SUPP:
ALWAYS,BUFF1;

BUFF1 DELAY: TRIA(20,30,100,1);
ASSIGN: ORDER = ORDER - 1;
QUEUE, BUFF1;
SCAN:
(NQ(BUF2) + FAC1+NQ(SHIPPQ)) .LE. (LEVEL2+SHIPSIZE);
ASSIGN: FAC1 = FAC1 + 1;
DELAY: TRIA(20,30,50,2);
ASSIGN: FAC1 = FAC1 - 1;
QUEUE, BUF2;
SCAN:  (NQ(BUF3) + SHIP) .LE. LEVEL3;
QUEUE,  SHIPQ;
GROUP:  SHIPSIZE, FIRST;
ASSIGN:  SHIP = SHIP + SHIPSIZE;
COUNT:  SHIPPED;
DELAY:  TRIA(100, 200, 300, 3);
ASSIGN:  SHIP = SHIP - SHIPSIZE;
SPLIT;

QUEUE,  BUF3;
SCAN:  (NQ(BUF4) + FAC2) .LE. LEVEL4;
ASSIGN:  FAC2 = FAC2 + 1;
DELAY:  TRIA(20, 30, 40, 4);
ASSIGN:  FAC2 = FAC2 - 1;

QUEUÉ,  BUF4: DETACH;
END;
B.2.2 Distribution Problem 1 Experiment File

; 1DP1.EXP
; Date modified: 11/1/94
; Note: Original problem with 5 decision variables

BEGIN;
; PROJECT, dist#1, PORNTHEP A. AND STEVE H.;

VARIABLES: 1,SHIP,0:
2,ORDER,0:
3,FAC1,0:
4,FAC2,0:
5, C1:
6, C2:
7, C3:
8, C4:
9, C5:
11,LEVEL1,10:
12,LEVEL2,10:
13,LEVEL3,10:
14,LEVEL4,10:
15,SHIPSIZE,10;

COUNTERS: SALES:
LOST_SALES:
SHIPPED:

QUEUES: 1,BUF1:
2,BUF2:
3,BUF3:
4,BUF4:
5,DEMAND:
6,VENDOR:
7,SHIPQ;

DSTATS: 1,NQ(BUF1), BUFFER 1:
2,FAC1, PARTS IN FACTORY1:
3,NQ(BUF2), BUFFER 2:
4,NQ(SHIPQ), SHIPPING Q:
5,SHIP, PARTS IN SHIPPING:
6,NQ(BUF3), BUFFER 3:
7,FAC2, PARTS IN FACTORY2:
8,NQ(BUF4), BUFFER 4;

FILES: INVALUE, "INPUT.DAT", SEQ, FRE:
OUTVALUE, "OUTPUT.DAT", SEQ, FRE;

SEEDS: 1, 12345:
2, 34567:
3, 56789:
4, 78901;
5, 90123;

REPLICATE, 1,0,10600,,1000;
END;
B.3.1 Distribution Problem 6 Model File

; 6DP.MOD
; Date modified: 12/1/94
; Note: 11 decision variables

BEGIN;
; READ DATA FROM INPUT.DAT
CREATE;
READ, INVALUE: LEVEL1, LEVEL2, LEVEL3, LEVEL4;
READ, INVALUE: LEVEL5, LEVEL6, LEVEL7, LEVEL8;
READ, INVALUE: SHIP1SIZE, SHIP2SIZE, SHIP3SIZE;
CLOSE, INVALUE: DISPOSE;

; WRITE COST TO OUTPUT.DAT
CREATE, 1, TFIN;
ASSIGN: C1 = 100 * NC(LOST_SALES) + 10 * NC(SHIPPED1)
+ 10 * NC(SHIPPED2) + 10 * NC(SHIPPED3);
C2 = 5*(DAVG(1)+DAVG(2));
C3 = 10*(DAVG(3)+DAVG(4)+DAVG(5));
C4 = 15*(DAVG(6)+DAVG(7));
C5 = 20*(DAVG(8)+DAVG(9)+DAVG(10));
C6 = 30*(DAVG(11)+DAVG(12));
C7 = 40*(DAVG(13)+DAVG(14)+DAVG(15));
C8 = 50*(DAVG(16)+DAVG(17));
C9 = 60*DAVG(18);
WRITE, OUTVALUE: C1+C2+C3+C4+C5+C6+C7+C8+C9: DISPOSE;

; CUSTOMER DEMAND SUBMODEL
CREATE: EXPO(20, 9);
QUEUE, DEMAND;
BRANCH, 1:
IF, NQ(BUF8) .LE. 0, LOST;
IF, NQ(BUF8) .GT. 0, GAIN;

LOST COUNT: LOST_SALES: DISPOSE;
GAIN REMOVE: 1, BUF8, LEAVE: DISPOSE;
LEAVE COUNT: SALES: DISPOSE;

; MAIN MODEL
CREATE: VENDOR;
QUEUE, SCAN:
(NQ(BUF1) + ORDER) .LE. LEVEL1;
ASSIGN: ORDER = ORDER + 1;
BRANCH, 2:
ALWAYS, SUPP;
ALWAYS, BUFF1;

BUFF1 DELAY: TRIA(20, 30, 100, 1);
ASSIGN: ORDER = ORDER - 1;
QUEUE, SCAN: (NQ(BUF2) + FAC1 + NQ(SHIP1Q)) .LE. (LEVEL2 + SHIP1SIZE);
ASSIGN: FAC1 = FAC1 + 1;
DELAY: TRIA(20, 30, 50, 2);
ASSIGN: FAC1 = FAC1 - 1;

QUEUE, BUF2;
SCAN: (NQ(BUF3) + SHIP1) .LE. LEVEL3;
QUEUE, SHIP1Q;
GROUP: SHIP1SIZE, FIRST;
ASSIGN: SHIP1 = SHIP1 + SHIP1SIZE;
COUNT: SHIPPED1;
DELAY: TRIA(100, 200, 300, 3);
ASSIGN: SHIP1 = SHIP1 - SHIP1SIZE;
SPLIT;

QUEUE, BUF3;
SCAN: (NQ(BUF4) + FAC2 + NQ(SHIP2Q)) .LE. (LEVEL4 + SHIP2SIZE);
ASSIGN: FAC2 = FAC2 + 1;
DELAY: TRIA(20, 30, 40, 4);
ASSIGN: FAC2 = FAC2 - 1;

QUEUE, BUF4;
SCAN: (NQ(BUF5) + SHIP2) .LE. LEVEL5;
QUEUE, SHIP2Q;
GROUP: SHIP2SIZE, FIRST;
ASSIGN: SHIP2 = SHIP2 + SHIP2SIZE;
COUNT: SHIPPED2;
DELAY: TRIA(200, 400, 600, 5);
ASSIGN: SHIP2 = SHIP2 - SHIP2SIZE;
SPLIT;

QUEUE, BUF5;
SCAN: (NQ(BUF6) + FAC3 + NQ(SHIP3Q)) .LE. (LEVEL6 + SHIP3SIZE);
ASSIGN: FAC3 = FAC3 + 1;
DELAY: TRIA(60, 90, 150, 6);
ASSIGN: FAC3 = FAC3 - 1;

QUEUE, BUF6;
SCAN: (NQ(BUF7) + SHIP3) .LE. LEVEL7;
QUEUE, SHIP3Q;
GROUP: SHIP3SIZE, FIRST;
ASSIGN: SHIP3 = SHIP3 + SHIP3SIZE;
COUNT: SHIPPED3;
DELAY: TRIA(50, 100, 150, 7);
ASSIGN: SHIP3 = SHIP3 - SHIP3SIZE;
SPLIT;

QUEUE, BUF7;
SCAN: (NQ(BUF8) + FAC4) .LE. LEVEL8;
ASSIGN: FAC4 = FAC4 + 1;
DELAY: TRIA(60, 90, 120, 8);
ASSIGN: FAC4 = FAC4 - 1;
QUEUE, BUF8:DETACH;

END;
B.3.2 Distribution Problem 6 Experiment File

; 6DP1.EXP
; Date modified: 12/1/94
; Note: 11 decision variables

BEGIN;
; PROJECT, buffer#5, PORNTHEP A. AND STEVE H.;

VARIABLES: 1, ORDER, 0:
2, FAC1, 0:
3, FAC2, 0:
4, FAC3, 0:
5, FAC4, 0:
6, SHIP1, 0:
7, SHIP2, 0:
8, SHIP3, 0:
11, C1:
12, C2:
13, C3:
14, C4:
15, C5:
16, C6:
17, C7:
18, C8:
19, C9:
21, LEVEL1, 10:
22, LEVEL2, 10:
23, LEVEL3, 10:
24, LEVEL4, 10:
25, LEVEL5, 10:
26, LEVEL6, 10:
27, LEVEL7, 10:
28, LEVEL8, 10:
31, SHIP1SIZE, 10:
32, SHIP2SIZE, 10:
33, SHIP3SIZE, 10;

COUNTERS: SALES:
LOST_SALES:
SHIPPED1:
SHIPPED2:
SHIPPED3;

QUEUES: 1, BUF1:
2, BUF2:
3, BUF3:
4, BUF4:
5, BUF5:
6, BUF6:
7, BUF7:
DSTATS:

1, NQ(BUF1), BUFFER 1:
2, FAC1, PARTS IN FACTORY1:
3, NQ(BUF2), BUFFER 2:
4, NQ(SHIP1Q), SHIPPING Q1:
5, SHIP1, PARTS IN SHIPPING:
6, NQ(BUF3), BUFFER 3:
7, FAC2, PARTS IN FACTORY2:
8, NQ(BUF4), BUFFER 4:
9, NQ(SHIP2Q), SHIPPING Q2:
10, SHIP2, PARTS IN SHIPPING2:
11, NQ(BUF5), BUFFER 5:
12, FAC3, PARTS IN FACTORY3:
13, NQ(BUF6), BUFFER 6:
14, NQ(SHIP3Q), SHIPPING Q3:
15, SHIP3, PARTS IN SHIPPING3:
16, NQ(BUF7), BUFFER 7:
17, FAC4, PARTS IN FACTORY4:
18, NQ(BUF8), BUFFER 8:

FILES:

INVALUE, "INPUT.DAT", SEQ, FRE:
OUTVALUE, "OUTPUT.DAT", SEQ, FRE;

SEEDS:

1, 12345:
2, 34567:
3, 56789:
4, 78901:
5, 90123:
6, 54321:
7, 32109:
8, 19876:

REPLICATE, 1, 0, 10600, , 1000;

END;
B.4.1 Standard Buffer Problem Model File

; STD.MOD
; Test problem taken from literature (Ho, Eyler and Chien 1979)
; part. c
; uses vars. to track buffer volumes to avoid entity overflow

BEGIN;
; READ DATA FROM INPUT.DAT
CREATE;
READ, INVALUE: CAP1, CAP2, CAP3, CAP4;
CLOSE, INVALUE: DISPOSE;

; MAIN MODEL
CREATE;

START QUEUE, STARTBUFFER;
SEIZE: MACHINE1;
DELAY: 1;
BRANCH, 1:
    IF, BUF1 >= CAP1, BLOCK1:
    ELSE, CONT11;

BLOCK1 QUEUE, BLOCKED1;
WAIT: 1, 1;
CONT11 BRANCH:
    IF, DISCRETE (0.001, 1, 1, 0, 1) == 1, FAIL1:
    ALWAYS, CONT12:
    ALWAYS, START;

FAIL1 ALTER: MACHINE1, -1;
DELAY: EXPO (200, 2);
COUNT: MACH1FAIL;
ALTER: MACHINE1, 1: DISPOSE;

CONT12 RELEASE: MACHINE1;

BRANCH, 1:
    IF, NQ(BUFFER1) == 0, INCRE1:
    ELSE, DISP1;

DISP1 ASSIGN: BUF1 = BUF1 + 1: DISPOSE;
INCRE1 ASSIGN: BUF1 = BUF1 + 1;
RETURN1 QUEUE, BUFFER1;
SEIZE: MACHINE2;
ASSIGN: BUF1 = BUF1 - 1;
SIGNAL: 1, 1;
BRANCH, 2:
    IF, BUF1 > 0, RETURN1:
    ALWAYS, CONT13;

CONT13 DELAY: 1;
BRANCH, 1:
    IF, BUF2 >= CAP2, BLOCK2:
    ELSE, CONT21;

BLOCK2 QUEUE, BLOCKED2;
WAIT: 2, 1;

CONT21 BRANCH:
  IF, DISCRETE (0.001, 1, 1, 0, 3) == 1, FAIL2:
    ALWAYS, CONT22;
FAIL2 ALTER: MACHINE2, -1;
DELAY: EXPO (200, 4);
COUNT: MACH2FAIL;
ALTER: MACHINE2, 1: DISPOSE;
CONT22 RELEASE: MACHINE2;

BRANCH, 1:
  IF, NQ(BUFFER2) == 0, INCRE2:
    ELSE, DISP2;
DISP2 ASSIGN: BUF2 = BUF2 + 1: DISPOSE;
INCRE2 ASSIGN: BUF2 = BUF2 + 1;
RETURN2 QUEUE, BUFFER2;
SEIZE: MACHINE3;
ASSIGN: BUF2 = BUF2 - 1;
SIGNAL: 2, 1;
BRANCH, 2:
  IF, BUF2 > 0, RETURN2:
    ALWAYS, CONT23;
CONT23 DELAY: 1;
BRANCH, 1:
  IF, BUF3 >= CAP3, BLOCK3:
    ELSE, CONT31;
BLOCK3 QUEUE, BLOCKED3;
WAIT: 3, 1;
CONT31 BRANCH:
  IF, DISCRETE (0.001, 1, 1, 0, 5) == 1, FAIL3:
    ALWAYS, CONT32;
FAIL3 ALTER: MACHINE3, -1;
DELAY: EXPO (200, 6);
COUNT: MACH3FAIL;
ALTER: MACHINE3, 1: DISPOSE;
CONT32 RELEASE: MACHINE3;

BRANCH, 1:
  IF, NQ(BUFFER3) == 0, INCRE3:
    ELSE, DISP3;
DISP3 ASSIGN: BUF3 = BUF3 + 1: DISPOSE;
INCRE3 ASSIGN: BUF3 = BUF3 + 1;
RETURN3 QUEUE, BUFFER3;
SEIZE: MACHINE4;
ASSIGN: BUF3 = BUF3 - 1;
SIGNAL: 3, 1;
BRANCH, 2:
  IF, BUF3 > 0, RETURN3:
    ALWAYS, CONT33;
CONT33 DELAY: 1;
BRANCH, 1:
  IF, BUF4 >= CAP4, BLOCK4:
ELSE, CONT41;

BLOCK4 QUEUE, BLOCKED4;
WAIT: 4, 1;

CONT41 BRANCH:  
    IF, DISCRETE (0.001, 1, 1, 0, 7) == 1, FAIL4:  
        ALWAYS, CONT42;

FAIL4 ALTER: MACHINE4, -1;
DELAY: EXPO (200, 8);
COUNT: MACH4FAIL;
ALTER: MACHINE4, 1: DISPOSE;

CONT42 RELEASE: MACHINE4;

BRANCH, 1:  
    IF, NQ(BUFFER4) == 0, INCRE4:  
        ELSE, DISP4;

DISP4 ASSIGN: BUF4 = BUF4 + 1: DISPOSE;
INCRE4 ASSIGN: BUF4 = BUF4 + 1;

RETURN4 QUEUE, BUFFER4;
SEIZE: MACHINES;
ASSIGN: BUF4 = BUF4 - 1;
SIGNAL: 4, 1;
BRANCH, 2:  
    IF, BUF4 > 0, RETURN4:  
        ALWAYS, CONT43;

CONT43 DELAY: 1;
BRANCH:  
    IF, DISCRETE (0.001, 1, 1, 0, 9) == 1, FAIL5:  
        ALWAYS, CONT52;

FAIL5 ALTER: MACHINE5, -1;
DELAY: EXPO (200, 10);
COUNT: MACH5FAIL;
ALTER: MACHINE5, 1: DISPOSE;

CONT52 RELEASE: MACHINE5;
COUNT: NUMPARTS;
BRANCH, 1:  
    IF, NC(NUMPARTS) == 32000, WRITE;

;WRITE COST TO A OUTPUT.DAT
WRITE WRITE, OUTVALUE: TNOW: DISPOSE;
END;
B.4.2 Standard Buffer Problem Experiment File

; STD1.EXP
; Standard buffer problem

BEGIN;

PROJECT, LITERATURE TEST PROBLEM, STEVE HARRIS;
VARIABLES: CAP1, 100:
CAP2, 100:
CAP3, 100:
CAP4, 100:
BUF1, 0:
BUF2, 0:
BUF3, 0:
BUF4, 0;

QUEUES: STARTBUFFER:
BLOCKED1:
BUFFER1:
BLOCKED2:
BUFFER2:
BLOCKED3:
BUFFER3:
BLOCKED4:
BUFFER4;

RESOURCES: MACHINE1:
MACHINE2:
MACHINE3:
MACHINE4:
MACHINE5;

COUNTERS: NUMPARTS, 32001:
MACH1FAIL:
MACH2FAIL:
MACH3FAIL:
MACH4FAIL:
MACH5FAIL;

DSTATS: BUF1:
BUF2:
BUF3:
BUF4:
NQ(BLOCKED1):
NQ(BLOCKED2):
NQ(BLOCKED3):
NQ(BLOCKED4):
NQ(BUFFER1):
NQ(BUFFER2):
NQ(BUFFER3):
NQ(BUFFER4);

FILES: INVALUE, "INPUT.DAT", SEQ, FRE:
OUTVALUE, "OUTPUT.DAT", SEQ, FRE;
SEEDS:  
1, 96583:  
2, 67929:  
3, 54425:  
4, 52123:  
5, 57541:  
6, 4341:  
7, 43763:  
8, 62687:  
9, 12017:  
10, 73099;  
END;
B.5.1 Warehouse Storage and Retrieval Problem 1 Model File

; AS.MOD Last modified 11/25/94
; Note: - No tray pick-up
; - Include DBDEEP

BEGIN;
CREATE;
;Read variables from FILE
READ, infile: NOASL, NOSID, NOBAY, NOLVL;
READ, infile: INBAY(1), INLVL(1);
READ, infile: INBAY(2), INLVL(2);
READ, infile: OBAY(1), OLVL(1);
READ, infile: OBAY(2), OLVL(2);
READ, infile: HVM, HVI, HVC, HA, HD, HSI, HSC;
READ, infile: VVM, VVI, VVC, VA, VD, VSI, VSC;
CLOSE, infile;

;Units conversion
ASSIGN: HA = -1*(HA*12);
ASSIGN: HD = HD*12;
ASSIGN: HVM = (HVM*12)/60;
ASSIGN: HVI = (HVI*12)/60;
ASSIGN: HTI = HSI/HVI;
ASSIGN: HVC = (HVC*12)/60;
ASSIGN: HTC = HSC/HVC;

ASSIGN: VA = -1*(VA*12);
ASSIGN: VD = VD*12;
ASSIGN: VVM = (VVM*12)/60;
ASSIGN: VVI = (VVI*12)/60;
ASSIGN: VTI = VSI/VVI;
ASSIGN: VVC = (VVC*12)/60;
ASSIGN: VTC = VSC/VVC;

ASSIGN: HGAM = (1.0/HD)-(1.0/HA);
ASSIGN: VGAM = (1.0/VD)-(1.0/VA): DISPOSE;

; STORE COMMAND
; Note ABAY, ALVL --> input bay&level
; BBAY, BLVL --> stored bay&level
CREATE;
ASSIGN: CURBAY = 1:
CURLVL = 5:
TASK = 1;
ASSIGN: INPBAY = CURBAY:
INPLVL = CURLVL;
POINTS1 ASSIGN: ABAY = DISCRETE(0.5, INBAY(1), 1.0, INBAY(2), 1):
ALVL = DISCRETE(0.5, INLVL(1), 1.0, INLVL(2), 1);
BRANCH, 1:
IF, (ABAY.EQ.INPBAY).AND.(ALVL.EQ.INPLVL), POINTS1:
ELSE, POINTS2;

POINTS2 ASSIGN: BBAY = ANINT(UNIF(0.5, (NOBAY+0.5), 1)):
BLVL = ANINT(UNIF(0.5, (NOLVL+0.5), 1));
ASSIGN: INDEX = 1;

POINTS3 BRANCH, 1:
IF, (BBAY.EQ.INBAY(INDEX)).AND.(BLVL.EQ.INLVL(INDEX)), POINTS2:
ELSE, CONS1;

CONS1 ASSIGN: INDEX = INDEX + 1;
BRANCH, 1:
IF, INDEX.LE.INSTA, POINTS3:
ELSE, CONS2;

CONS2 ASSIGN: INDEX = 1;
POINTS4 BRANCH, 1:
IF, (BBAY.EQ.OBAY(INDEX)).AND.(BLVL.EQ.OLVL(INDEX)), POINTS2:
ELSE, CONS3;

CONS3 ASSIGN: INDEX = INDEX + 1;
BRANCH, 1:
IF, INDEX.LE.OUTSTA, POINTS4:
ELSE, CONS4;

CONS4 ASSIGN: INPBAY = ABAY:
INPLVL = ALVL;
QUEUE, STCMDQ:DETACH;

; RETRIEVE COMMAND
; Note ABAY, ALVL --> retrieved bay&level
; BBAY, BLVL --> output bay&level
CREATE,, 1;

POINTR0 BRANCH, 1:
WITH, SFPROB, SFC1:
WITH, NOPROB, SFC2;

SFC1 ASSIGN: TASK = 3:NEXT(POINTR1);

SFC2 ASSIGN: TASK = 2:NEXT(POINTR1);

POINTR1 ASSIGN: BBAY = DISCRETE(0.5, OBAY(1), 1.0, OBAY(2), 2):
BLVL = DISCRETE(0.5, OLVL(1), 1.0, OLVL(2), 2);

POINTR2 ASSIGN: ABAY = ANINT(UNIF(0.5, (NOBAY+0.5), 2)):
ALVL = ANINT(UNIF(0.5, (NOLVL+0.5), 2));
BRANCH, 1:
IF, (ABAY.EQ.BBAY).AND.(ALVL.EQ.BLVL), POINTR2:
ELSE, CONR1;

CONR1 ASSIGN: INDEX = 1;

POINTR3 BRANCH, 1:
IF, (ABAY.EQ.INBAY(INDEX)).AND.(ALVL.EQ.INLVL(INDEX)),
POINTS2:
ELSE, CONR2;
CONR2
ASSIGN: INDEX = INDEX + 1;
BRANCH, 1:
IF, INDEX.LE.INSTA, POINTR3:
ELSE, CONR3;
CONR3
ASSIGN: INDEX = 1;
POINTR4
BRANCH, 1:
IF, (ABAY.EQ.OBAY(INDEX)).AND. (ALVL.EQ.OLVL(INDEX)),
POINTR2:
ELSE, CONR4;
CONR4
ASSIGN: INDEX = INDEX + 1;
BRANCH, 1:
IF, INDEX.LE.OUTSTA, POINTR4:
ELSE, CONR5;
CONR5
QUEUE, RTCMDQ: DETACH;

; AS/RS CONTROL SYSTEM
CREATE;
BACK
BRANCH, 1:
IF, (NQ(STCMDQ).GT.0).AND. (NQ(RTCMDQ).GT.0), POINTC1:
ELSE, CONC1;
CONC1
BRANCH, 1:
IF, NQ(STCMDQ).GT.0, POINTC2:
ELSE, CONC2;
CONC2
BRANCH, 1:
IF, NQ(RTCMDQ).GT.0, POINTC3:
ELSE, BACK;
POINTC1
BRANCH, 1:
WITH, STPROB, POINTC2:
WITH, RTPROB, POINTC3;
POINTC2
REMOVE: 1, STCMDQ, START;
ASSIGN: DONE = 0:NEXT(POINTC4);
POINTC3
REMOVE: 1, RTCMDQ, START;
ASSIGN: DONE = 0:NEXT(POINTC4);
POINTC4
QUEUE, WAITDONE;
SCAN: DONE.EQ.1: NEXT(BACK);

; AS/RS SYSTEM
START
BRANCH, 1:
IF, TASK.GE.2, RETRIEVE:
ELSE, CONSYS1;
CONSYS1
DUPLICATE: 1, POINTS1;
SEIZE: SR:MARK(TRVALTIME);
EVENT: 1;
DELAY: MAXTIM;
ASSIGN: CURBAY = ABAY:
CURLVL = ALVL;
ASSIGN: INPUTB = ABAY:
INPUTL = ALVL;
EVENT: 2;
DELAY: MAXTIM;
ASSIGN: CURBAY = BBAY:
   CURLVL = BLVL;
RELEASE: SR;
BRANCH, 1:
   IF, NQ(RTCDMQ).LT.1, DISP:
   ELSE, CONSYS2;
CONSYS2 REMOVE: 1, RTCDMQ, RETRIEVE;
TALLY: STORETIME, INT(TRAVALTIME);
COUNT: STORENO, 1: DISPOSE;
DISP ASSIGN: DONE = 1;
TALLY: STORETIME, INT(TRAVALTIME);
COUNT: STORENO, 1: DISPOSE;

RETRIEVE DUPLICATE: 1, POINTRO;

BRANCH, 1:
   IF, NOSID.GE.1, MODE2:
   ELSE, MODE1;

MODE1 ASSIGN: TASK = 2: NEXT(MODE2);

MODE2 SEIZE: SR: MARK(TRAVALTIME);
EVENT: 1;
DELAY: MAXTIM;
ASSIGN: CURBAY = ABAY:
   CURLVL = ALVL;

BRANCH, 1:
   IF, TASK.EQ.2, OUTBAY:
   ELSE, SHUF;L

SHUF L BRANCH, 1:
   IF, (CURBAY+3).LE. (NOBAY), GORHT:
   ELSE, GOLFT;
GORHT ASSIGN: CBAY = CURBAY + 3: NEXT(GOLVL);
GOLFT ASSIGN: CBAY = CURBAY - 3: NEXT(GOLVL);

GOLVL BRANCH, 1:
   IF, (CURLVL+2).LE. (NOLVL), Goup:
   ELSE, GODWN;
Goup ASSIGN: CLVL = CURLVL + 2: NEXT(GOOUT);
GODWN ASSIGN: CLVL = CURLVL - 2: NEXT(GOOUT);

GOOUT EVENT: 3;
DELAY: MAXTIM;
ASSIGN: CURBAY = CBAY:
   CURLVL = CLVL;
EVENT: 1;
DELAY: MAXTIM;
ASSIGN: CURBAY = ABAY:
OUTBAY EVENT: 2;
DELAY: MAXTIM;
ASSIGN: CURBAY = BBAY:
        CURLVL = BLVL;
RELEASE: SR;
ASSIGN: DONE = 1;
TALLY: RETRIEVETIME, INT(TRAVALTIME);
COUNT: RETRIEVENO, 1: DISPOSE;
END;
B.5.2 Warehouse Storage and Retrieval Problem 1 Experiment File

; 1AS1.EXP Last modified 11/25/94
; Note: - No pick-up tray
;      - Included DBDEEP

BEGIN;
;PROJECT, P1, PORNTHEP;

ATTRIBUTES: 1,TASK:
2,ABAY:
3,ALVL:
4,BBAY:
5,BLVL:
6,CBAY:
7,CLVL:
11,HRZTIM:
12,VRTTIM:
13,MAXTIM:
14,TRAVALT;

VARIABLES: 1,NOASL:
2,NOSID:
3,NOBAY:
4,NOLVL:
5,HVM:
6,HVI:
7,HVC:
8,HA:
9,HD:
10,HSI:
11,HSC:
12,VVM:
13,VVI:
14,VVC:
15,VA:
16,VD:
17,VSI:
18,VSC:
19,HTI:
20,HTC:
21,VTI:
22,VTC:
23,HGAM:
24,CURBAY:
25,CURLVL:
26,CURSMV:
27,CURSLD:
28,VGAM:
30,INSTA, 2:
31-32, INBAY (2):
QUEUES:

<table>
<thead>
<tr>
<th>Queue</th>
<th>Description</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>STCMDQ</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>RTCMDQ</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>ASRSQ</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>WAITDONE</td>
<td></td>
</tr>
</tbody>
</table>

RESOURCES:

<table>
<thead>
<tr>
<th>Resource</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SR</td>
<td></td>
</tr>
</tbody>
</table>

COUNTERS:

<table>
<thead>
<tr>
<th>Counter</th>
<th>Description</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>STORENO</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>RETRIEVENO</td>
<td></td>
</tr>
</tbody>
</table>

TALLIES:

<table>
<thead>
<tr>
<th>Tally</th>
<th>Description</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>STORETIME</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>RETRIEVETIME</td>
<td></td>
</tr>
</tbody>
</table>

SEEDS:

<table>
<thead>
<tr>
<th>Seed</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>5041</td>
</tr>
<tr>
<td>2</td>
<td>50221</td>
</tr>
</tbody>
</table>

DSTATS:

<table>
<thead>
<tr>
<th>Stat</th>
<th>Description</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>NR(1)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>ASRS_UTILIZATION</td>
<td></td>
</tr>
</tbody>
</table>

FILES:

<table>
<thead>
<tr>
<th>File</th>
<th>Description</th>
<th>SEQ</th>
<th>FREE</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>infile</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>&quot;input.dat&quot;</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

REPLICATE:

<table>
<thead>
<tr>
<th>Replication</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0, 14400, 600</td>
</tr>
</tbody>
</table>

END;
B.5.3 ASRS.EXE for use with AS/RS Problems (ASRS.C)

/ *****************************************************************************
* Revision History:                                                      *
* Date  Who  Modifications                                                *
* -------------------------------------------------------------------------- *
* 11/26/94  Pornthep A.  AS/RS                                         *
* Copyright 1992, Systems Modeling Corporation.  All Rights Reserved.   *
*****************************************************************************/

#ifdef SUN_
   /*
   * SUN requires underscore appended to C functions called by
   * FORTRAN*
   *
   */
#else
   /*
   * Define cprime  cprime_
   * define cwrap  cwrap_
   * define cevent  cevent_
   * define cuf  cuf_
   * define cur  cur_
   * define cstate  cstate_
   * define cuclea  cuclea_
   * define cusav  cusav_
   * define curst  curst_
   * define ckeyhi  ckeyhi_
   * define cuerr  cuerr_
   */
#endif

#ifndef _MSC_VER
   /*
   * Microsoft C 6.0 or later.
   *
   * Adjust for incompatibility between FORTRAN 5.1 and C6.0
   * Note: C6 is ANSI, C5 is K&R, FORTRAN 5.1 expects K&R
   *
   * Note2: Another C6.0 bug requires compiling with /Gh option
   *
   */
#else
   # define ms_float double
#endif

#include "c:\siman\lib\simlib.h"
#include "c:\c600\include\stdio.h"

#define maxrep 10

float horztm;
float verttm;
float maxtim;
smint cno1 = 1, cno2 = 2;
float result1, result2;

/* Routine : cprime
   * Description : Called from SIMAN at the beginning of each
   *   simulation
   *   * schedule the initial event for the model.
   */
void cprime ( sim )
    simstr *sim;
{

    FILE *fp;
    smint index,i,j;
    smint temp1,temp2;
    float temp1,temp2;
    smint bay,level;
    smint Nvar,Nvar2;

    index = sim->numrep;
    if (index >= 1)
    {
        fp = fopen("input.dat","r");
        fscanf(fp,"%d %d",&temp1,&temp2);
        fscanf(fp,"%d %d",&bay,&level);
        fscanf(fp,"%d %d",&temp1,&temp2);
        fscanf(fp,"%d %d",&temp1,&temp2);
        fscanf(fp,"%d %d",&temp1,&temp2);
        fscanf(fp,"%d %d",&temp1,&temp2);
        for (i = 0; i < 2; i++)
            for (j = 0; j < 7; j++)
                fscanf(fp,"%f",&temp1);
        for (i = 0; i < (bay+1); i++)
            fscanf(fp,"%f",&temp1);
        for (i = 0; i < level+1; i++)
            fscanf(fp,"%f",&temp1);
        Nvar = i+75;
        setv(&Nvar,&temp1);
    }
    for (i = 2; i <= (bay+1); i++)
    {
        Nvar = i+75;
        /* CALCULATE ABSOLUTE DIMENSIONS FOR BAYS */
        for (j = 0; j < level; j++)
            setv(&Nvar,&temp1);
    }
Nvar2 = Nvar-1;
    templ = v(&Nvar2) + v(&Nvar);
    setv(&Nvar,&templ);
}
/
* ------------------------------------------------------------- */
/* CALCULATE ABSOLUTE DIMENSIONS FOR LEVELS */
for (i = 2; i <= (level+1); i++)
{
    Nvar = i+375;
    Nvar2 = Nvar-1;
    templ = v(&Nvar2) + v(&Nvar);
    setv(&Nvar,&templ);
}
/
* ------------------------------------------------------------- */
fclose(fp);

return;
}

/***************************************************************************/
* Routine : cwrap
* Description : Called from SIMAN at the end of each simulation
*               replication to process the logic associated with the
*               end of the replication.
***************************************************************************/
void cwrap ( sim )
simstr *sim;
{
    smint index;
    float stout, rtout;
    FILE *fp;
    index = sim->numrep;
    stout = nc(&cnol); rtout = nc(&cno2);
    if (index == 1) {
        fp = fopen("output.dat","w");
        fprintf(fp, "%4.0f %4.0f\n",stout,rtout);
        fclose(fp);
    } else {
        fp = fopen("output.dat","a");
        fprintf(fp, "%4.0f %4.0f\n",stout,rtout);
        fclose(fp);
    }
    return;
}
/***************************************************************************/
* Routine : cevent
* Description : Maps the event number n to a call to the appropriate
  event subroutine containing the logic for the event.
* Usage :

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

void cevent (1, n, sim)
smint *1;
smint *n;
simstr *sim;

FILE *fp;
smint Ntab,Nvar;
smint curbay(curlvl,desbay,deslvl,task);
static smint index = 0;

void getdist(smint *1, simstr *sim,
smint curbay, smint curlvl,
smint desbay, smint deslvl);

/*if (index < 1)
{
  fp = fopen ("trace.dat","w");
  fprintf(fp,"TaskNo From BayNo LevelNo To BayNo, LevelNo ");
  fprintf(fp,"Vtime Htime Maxtime\n");
  fprintf(fp,"--------------------------
") ;
}
else fp = fopen("trace.dat","a"); */
if (*n == 1)
{
  Nvar = 24;
curbay = v(&Nvar);
  Nvar = 25;
curlvl = v(&Nvar);
  Ntab = 2;
desbay = a(1,&Ntab);
  Ntab = 3;
deslvl = a(1,&Ntab);
  Ntab = 1;
task = a(1,&Ntab);
  getdist(1, sim, curbay, curlvl, desbay, deslvl);
  /* fprintf(fp,"%3d",task);
  fprintf(fp,"%3d",curbay);
  fprintf(fp,"%3d",curlvl);
  fprintf(fp,"%3d",desbay);
  fprintf(fp,"%3d",deslvl);
  fprintf(fp,"%5.1f",horztm);
  fprintf(fp,"%5.1f",verttm);
  fprintf(fp,"%5.1f\n",maxtim); */
}
if (*n == 2)
{
  Nvar = 24;
curbay = v(&Nvar);
Nvar = 25;
curlvl = v(&Nvar);
Ntab = 4;
desbay = a(1,&Ntab);
Ntab = 5;
deslvl = a(1,&Ntab);
Ntab = 1;
task = a(1,&Ntab);
getdist(1, sim, curbay, curlvl, desbay, deslvl);
/* fprintf(fp,"%3d",task);*/
cprintf(fp,"%3d",curbay);
cprintf(fp,"%3d",curlvl);
cprintf(fp,"%3d",desbay);
cprintf(fp,"%3d",deslvl);
cprintf(fp,"%5.1f",horztm);
cprintf(fp,"%5.1f",verttm);
cprintf(fp,"%5.1f\n",maxtim);*/
}

if (*n == 3)
{
Nvar = 24;
curbay = v(&Nvar);
Nvar = 25;
curlvl = v(&Nvar);
Ntab = 6;
desbay = a(1,&Ntab);
Ntab = 7;
deslvl = a(1,&Ntab);
Ntab = 1;
task = a(1,&Ntab);
getdist(1, sim, curbay, curlvl, desbay, deslvl);
/* fprintf(fp,"%3d",task);*/
cprintf(fp,"%3d",curbay);
cprintf(fp,"%3d",curlvl);
cprintf(fp,"%3d",desbay);
cprintf(fp,"%3d",deslvl);
cprintf(fp,"%5.1f",horztm);
cprintf(fp,"%5.1f",verttm);
cprintf(fp,"%5.1f\n",maxtim);*/
}
/*fclose(fp);*/
index++;
}

return;
}

void getdist(l, sim, cb, cl, db, dl)
{
smint *l;
simstr *sim;
smint cb,cl,db,dl;

smint Ntab,Nvar,bay1,bay2,lvl1,lvl2;
float ha,hd,hvm,hvi,hti,hvc,htc;
float va,vd,vvm,vvi,vti,vvc,vtc;
float hgam,vgam;
float hs1,ht1,hs2,ht2,hs3,ht3,hs4,ht4,hs5,ht5;
float vs1, vt1, vs2, vt2, vs3, vt3, vs4, vt4, vs5, vt5;
float horzdt, vertdt;

Nvar = 8; ha = v(&Nvar);
Nvar = 9; hd = v(&Nvar);
Nvar = 5; hvm = v(&Nvar);
Nvar = 6; hvi = v(&Nvar);
Nvar = 19; hti = v(&Nvar);
Nvar = 7; hvc = v(&Nvar);
Nvar = 20; htc = v(&Nvar);
Nvar = 15; va = v(&Nvar);
Nvar = 16; vd = v(&Nvar);
Nvar = 12; vvm = v(&Nvar);
Nvar = 13; vvi = v(&Nvar);
Nvar = 21; vti = v(&Nvar);
Nvar = 14; vvc = v(&Nvar);
Nvar = 22; vtc = v(&Nvar);
Nvar = 23; hgam = v(&Nvar);
Nvar = 28; vgam = v(&Nvar);

hs1 = (hgam/2.0)*(hvc*hvc);
ht1 = hgam*hvc;
hs2 = hs1 + (htc*hvc);
ht2 = ht1 + htc;
hs3 = (htc*hvc)+((hgam/2.0)*(hvi*hvi));
ht3 = htc + (hvi*hgam);
hs4 = hs3 + (hti*hvi);
ht4 = htc + hti;
hs5 = (htc*hvc)+(hti*hvi)+((hgam/2.0)*(hvc*hvm));
ht5 = htc + hti + (hgam*hvm);

vs1 = (vgam/2.0)*(vvc*vvc);
vt1 = vgam*vvc;
vs2 = vs1 + (vtc*vvc);
vt2 = vt1 + vtc;
vs3 = (vtc*vvc)+((vgam/2.0)*(vvi*vvi));
vt3 = vtc + (vvi*vgam);
vs4 = vs3 + (vti*vvi);
vt4 = vt3 + vti;
vs5 = (vtc*vvc)+(vti*vvi)+((vgam/2.0)*(vvm*vvm));
vt5 = vtc + vti + (vgam*vvm);

bay1 = cb + 1 + 74; bay2 = db + 1 + 74;
horzdt = abs(v(&bay1)-v(&bay2));

if (horzdt < hs1) horztm = sqrt((2.0*hgam*horzdt));
else if (horzdt <= hs2) horztm = ht1+(horzdt-hs1)/hvc;
else if (horzdt < hs3)
    horztm = htc + sqrt((2.0*hgam*(horzdt-(htc*hvc))));
else if (horzdt <= hs4)
    horztm = htc + (horzdt-hs3)/hvi;
else if (horzdt < hs5)
    horztm = htc + hti + sqrt((2.0*hgam*(horzdt-(htc*hvc)-(hti*hvi))));
else
horztm = ht5 + ((horzdt-hs5)/hvm);

lvl1 = cl + 1 + 374; lvl2 = dl + 1 + 374;
vertdt = abs(v(&lvl1)-v(&lvl2));
if (vertdt < vs1) verttm = sqrt((2.0*vgam*vertdt));
else if (vertdt <= vs2) verttm = vt1+((vertdt-vs1)/vvc);
   else if (vertdt < vs3)
         verttm = vtc + sqrt((2.0*vgam*(vertdt-(vtc*vvc))));
   else if (vertdt <= vs4)
         verttm = vt3 + ((vertdt-vs3)/vvi);
   else if (vertdt < vs5)
         verttm = vtc + vt1 +
                     sqrt((2.0*vgam*(vertdt-(vtc*vvc)-(vti*vvi))));
   else
         verttm = vt5 + ((vertdt-vs5)/vvm);
if (horztm > verttm) maxtim = horztm;
else maxtim = verttm;
Ntab = 13;
seta(l,&Ntab,&maxtim);
return;
}

/***************************************************************************/
**  *  * Routine : cuf *
**  * Description : Gets index computed in user rule NUF for queue, *
**  * resource, and transporter selection rules. *
**  *
**  * Inputs :   smint *l - index of the current entity *
**  *            smint *n - user rule *
**  *            simstr *sim - common block pointer *
**  *
**  * Returns : index computed in user rule NUF *
**  */
/***************************************************************************/

ms_float cuf ( l, n, sim )

smint *l;
smint *n;
simstr *sim;
{
    smint i;
    float result = 0.0;
    /**
      printf ( "\n\nEntered cuf\n" );
      printf ( "\n1 =%d, *l \n");
      printf ( "\nn =%d, *n \n");
      printf ( "\ntnow =%f, sim->tnow \n");

      i = 1;
      result = tavg(&i);
      **/
printf ("\n\ntavg(l) = %f", result);
result = (float)(*n);
**/
    return((ms_float)result);
}

/**
* Routine :      cur
* Description : Gets index computed in user rule NUR for queue,
*                resource, and transporter selection rules.
* Inputs :      smint *1 - index of the current entity
*                smint *n - user rule
*                simstr *sim - common block pointer
* Returns :      index computed in user rule NUR
*
***************************************************************************/
ms_float cur ( l, n, sim )
smint *1;
smint *n;
simstr *sim;
{
    float result = 0.0;
    /**
        printf ("\nCalled function CUR(%d", *n );
        printf ("\n   =%d", *1 );
        printf ("\n   =%d", *n );
        printf ("\nnow =%f", sim->tnow );
    **/
        return((ms_float)result);
    }

/**
* Routine :      cstate
* Description : Contains the state and differential equations used in
*                a continuous simulation
* Usage :        *
* Inputs :      simstr *sim - common block pointer
* Returns :      none
*
*******************************************************************************/
void cstate ( sim )

simstr *sim;
{
/**
 printf ( "\n\nEntered cstate\n" );
 printf ( "\ntnow =%f", sim->tnow );
**/
 return;
}

/*******************************************************************************/
/* Routine : cuclea */
/* Description : Clears the user-defined statistics. A call to */
cuclea
/* is made automatically by subroutine CLEAR when SIMAN */
/* system variable statistics. */
/* Inputs : simstr *sim - common block pointer */
/* Returns : none */
/*******************************************************************************/
void cuclea ( sim )

simstr *sim;
{
/**
 printf ( "\n\nEntered cuclea\n" );
 printf ( "\ntnow =%f", sim->tnow );
 printf ( "\numrep =%d", sim->numrep );
**/
 return;
}

/*******************************************************************************/
/* Routine : cusav */
/* Description : Saves the current system status */
/* Inputs : struct simstr *sim - common block pointer */
/* Returns : none */
/*******************************************************************************/
void cusav ( sim )
simstr *sim;
{
  /**
   * printf
   * printf
   * printf
   **/
   return;
}

/***************************************************************
**
* Routine :       curst
* Description :   Restores the information saved by cusav
* Inputs :        simstr *sim - common block pointer
* Returns :       none
***************************************************************
*/
void curst ( sim )

simstr *sim;
{
  /**
   * printf
   * printf
   * printf
   **/
   return;
}

/***************************************************************
**
* Routine :       ckeyhi
* Description :   Called from SIMAN each time the user presses a key
during simulation execution and sets ICODE to the
ASCII value of the key pressed.
* Inputs :        smint *key - ASCII value of the key pressed
* simstr *sim - common block pointer
* Returns :       none
***************************************************************
*/
void ckeyhi ( key, sim )

smint *key;
simstr *sim;
{
/**
 printf ( "\n\nEntered ckeyhi\n" );
 printf ( "\nkey =\%d", *key );
 printf ( "\ntnow =\%f", sim->tnow );
/**/ return;
}

/***************************************************************/
/* Routine : cuerr */
/* Description : Called by WWTMSG to overwrite SIMAN error messages */
/* Inputs : char sname - name of routine in which error occurred */
/* smint *mstyp - message type */
/* smint *msnum - message number */
/* simstr *sim - common block pointer */
/* */
/* Returns : none */
/***************************************************************/
void cuerr (sname,mstyp,msnum,sim )
char sname;
smint *mstyp;
smint *msnum;
simstr *sim;
{
/**
 simint i;
 float result;
 printf ( "\n\nEntered cuerr\n" );
 printf ( "\n1 =\%d", *1 );
 printf ( "\ntavg(l)\n" );
 printf ( "\ntnow =\%f", sim->tnow );
 printf ( "\nnnumrep =\%d", sim->numrep );
 i = 1;
 result = tavg(&i);
 printf ( "\n\ntavg(1) = \%f", result );
/**/ return;
}
B.6.1 Job-shop Problem 1 Model File

; JOBSHOP PROBLEM #1
; 1 rep. = 2 days

BEGIN;
; INITIALIZE SYSTEM VARIABLES

; Read data from INPUT.DAT
CREATE;
MR(6), MR(7), MR(8);
READ, INVALUE: PERCS(1), PERCS(2), PERCS(3), PERCS(4), PERCS(5),
PERCS(6), PERCS(7), PERCS(8), PT1, PT2;
READ, INVALUE: RSEL;
CLOSE, INVALUE;

; Set Number of AGVs
BRANCH, 1:
IF, NUMAGV.LT.5, 4AGVS:
IF, NUMAGV.LT.4, 3AGVS:
IF, NUMAGV.LT.3, 2AGVS:
IF, NUMAGV.LT.2, 1AGV:
ELSE, CNT;
4AGVS ASSIGN: IT(AGV,5) = 2: NEXT(CNT);
3AGVS ASSIGN: IT(AGV,4) = 2: NEXT(CNT);
2AGVS ASSIGN: IT(AGV,3) = 2: NEXT(CNT);
1AGV ASSIGN: IT(AGV,2) = 2: NEXT(CNT);

; Set Cutting Speeds, Part mix, Tool Lifes, CNT
ASSIGN: CS(1) = MAXCS(1)*PERCS(1):
CS(2) = MAXCS(2)*PERCS(2):
CS(3) = MAXCS(3)*PERCS(3):
CS(4) = MAXCS(4)*PERCS(4):
CS(5) = MAXCS(5)*PERCS(5):
CS(6) = MAXCS(6)*PERCS(6):
CS(7) = MAXCS(7)*PERCS(7):
CS(8) = MAXCS(8)*PERCS(8):
ASSIGN: PT3 = 1 - PT1 - PT2:
TOOLLIFE(1) = 80000/(CS(1)**1.2):
TOOLLIFE(2) = 80000/(CS(2)**1.2):
TOOLLIFE(3) = 80000/(CS(3)**1.2):
TOOLLIFE(4) = 80000/(CS(4)**1.2):
TOOLLIFE(5) = 80000/(CS(5)**1.2):
TOOLLIFE(6) = 80000/(CS(6)**1.2):
TOOLLIFE(7) = 80000/(CS(7)**1.2):
TOOLLIFE(8) = 80000/(CS(8)**1.2): DISPOSE;

; END OF REPLICATION
; Write profit to OUTPUT.DAT
CREATE, 1, TFIN;
ASSIGN: PROF = 200*NC(PART1S) + 175*NC(PART2S) + 250*NC(PART3S):
   PROF = PROF - (950*MR(1) + 1200*MR(2) + 1525*MR(3)) * 2:
   PROF = PROF - (700*MR(4) + 925*MR(5) + 1200*MR(6)
                  + 650*MR(7) + 900*MR(8)) * 2:
   PROF = PROF - (12*NUMAGV*2 + 26*NC(1) + 28*NC(2) + 30*NC(3) + 18*NC(4)):
   PROF = PROF - (19*NC(5) + 20*NC(6) + 13*NC(7) + 15*NC(8)):
   PROF = PROF - 124*2*(MR(1) + MR(2) + MR(3) + MR(4) + MR(5)
                  + MR(6) + MR(7) + MR(8)):
   PROF = PROF - (.12*wip*pt1 + .11*wip*pt2 + .15*wip*pt3) * 2;
WRITE, OUTVALUE: PROF :DISPOSE;

; JOB-SHOP MANUFACTURING SYSTEM

; Parts enter system, assigned part type, wait for AGV
CREATE, WIP;
NEWPART ASSIGN: IS = 0:MARK(TIMEIN);
ASSIGN: NS = DISCRETE(P1, P2, P1 + P2, 1, 1, 3, 9):
   TIMEIN = (-1)*TIMEIN;
ROUTE: 0.0, SEQ;

; ENTER SUBMODEL, AGV request
STATION, ENTER;
BRANCH, 1: IF, RSEL.EQ.1, RE1:
   IF, RSEL.EQ.2, RE2:
   IF, RSEL.EQ.3, RE3:
   ELSE, AGVSTA;
RE1 ASSIGN: RULE = NS:NEXT(AGVSTA);
RE2 ASSIGN: RULE = IS:NEXT(AGVSTA);
RE3 ASSIGN: RULE = TIMEIN:NEXT(AGVSTA);
AGVSTA QUEUE, AGVQ;
REQUEST: AGV(SDS);
DELAY: AGVLOAD;
TRANSPORT: AGV, SEQ;

; MILL WORKSTATIONS
; Send piece to type of mill with smallest queue
STATION, STA1;
DELAY: AGVLOAD;
FREE: AGV;
BRANCH, 1: IF, RSEL.EQ.2, RM2:
   ELSE, MILLSTA;
RM2 ASSIGN: RULE = IS;
MILLSTA QUEUE, MILLQ;
SELECT, POR:
   MHI:
      MMED:
      MLO;
   MHI SEIZE: MACHINE(3);
   ASSIGN: MIND = 3: NEXT(GENERIC);
   MMED SEIZE: MACHINE(2);
   ASSIGN: MIND = 2: NEXT(GENERIC);
   MLO SEIZE: MACHINE(1);
; LATE WORKSTATIONS
; Send piece to type of lathe with smallest queue
STATION, STA2;
DELAY: AGVLOAD;
FREE: AGV;
BRANCH, 1: IF, RSEL.EQ.2, RL2:
   ELSE, LATHSTA;
RL2 ASSIGN: RULE = IS;
LATHSTA QUEUE, 2;
SELECT, POR: LHI:
   LMED:
   LLO;
LHI SEIZE: MACHINE(6);
ASSIGN: MIND = 6: NEXT(GENERIC);
LMED SEIZE: MACHINE(5);
ASSIGN: MIND = 5: NEXT(GENERIC);
LLO SEIZE: MACHINE(4);
ASSIGN: MIND = 4: NEXT(GENERIC);

; DRILL WORKSTATIONS
; Send piece to type of lathe with smallest queue
STATION, STA3;
DELAY: AGVLOAD;
FREE: AGV;
BRANCH, 1: IF, RSEL.EQ.2, RD2:
   ELSE, DRILSTA;
RD2 ASSIGN: RULE = IS: NEXT(DRILSTA);
DRILSTA QUEUE, 3;
SELECT, POR: DHI:
   DLO;
DHI SEIZE: MACHINE(8);
ASSIGN: MIND = 8: NEXT(GENERIC);
DLO SEIZE: MACHINE(7);
ASSIGN: MIND = 7: NEXT(GENERIC);

; GENERIC MACHINING SUBMODEL
generic ASSIGN: INDEX = 1;
C1 BRANCH, 1: IF, T(MIND, INDEX).EQ.0, C3:
   ELSE, C2;
C2 ASSIGN: INDEX = INDEX + 1: NEXT(C1);
C3 ASSIGN: T(MIND, INDEX) = 1;
   DELAY: NORM(SFT/CS(MIND), STD(MIND)*SFT/CS(MIND), MIND) + 1:
   MARK(T1);
   ASSIGN: T(MIND, INDEX) = 0: MARK(T2);
   ASSIGN: TOOLTIME(MIND, INDEX) = TOOLTIME(MIND, INDEX) + (T2-T1);
   BRANCH, 1: IF, TOOLTIME(MIND, INDEX).GE.TOOLIFE(MIND), CHG:
   ELSE, C4;
C4 RELEASE: MACHINE(MIND): NEXT(AGVSTA);

; EXIT SUBMODEL
STATION, EXITSYS;
DELAY: AGVLOAD;
FREE: AGV;
BRANCH, 1: IF, NS.EQ.1, PARTA:
   IF, NS.EQ.2, PARTB:
      ELSE, PARTC;
   PARTA COUNT: PART2S:NEXT (NEWPART);
   PARTB COUNT: PART1S:NEXT (NEWPART);
   PARTC COUNT: PART3S:NEXT (NEWPART);
; CHANGE TOOL SUBMODEL
CHG  DUPLICATE: 1, AGVSTA;
   ASSIGN: T(MIND, INDEX) = 1;
   DELAY: CHGTIME(MIND);
   ASSIGN: T(MIND, INDEX) = 0;
   ASSIGN: TOOLTIME(MIND, INDEX) = 0;
   RELEASE: MACHINE(MIND);
   COUNT: MIND:DISPOSE;
END;
B.6.2 Job-shop Problem 1 Experiment File

BEGIN;
PROJECT, JP1, SH and PA;

ATTRIBUTES: 1, INDEX:
  2, T1:
  3, T2:
  4, MIND:
  5, SFT:
  6, TIMEIN:
  7, RULE;

VARIABLES: 1, PT1:
  2, PT2:
  3, PT3:
  4, MAXCS (8) , 500, 570, 670, 530, 620, 730, 670, 800:
  12, PERCS (8):
  20, CS (8):
  28, STD (8) , .075, .05, .025, .075, .05, .025, .06, .03:
  36, TOOLLIFE (8):
  44, CHGTIME (8) , 5, 4, 3, 4, 3, 2, 3, 2:
  52, AGVLOAD, .50:
  53, TOOLTIME (8, 5), 0:
  93, T (8, 5), 0:
  133, NUMAGV:
  134, RSEL:
  135, WIP:
  136, PROF;

STATIONS: 1, ENTER:
  2, STA1:
  3, STA2:
  4, STA3:
  5, EXITSYS;

RESOURCES: MACHINE (8);

QUEUES: 1, MILLQ, HVF (RULE):
  2, LATHEQ, HVF (RULE):
  3, DRILLQ, HVF (RULE):
  4, AGVQ, HVF (RULE);

TRANSPORTERS: 1, AGV, 5, 1, 100;

SEQUENCES:
  1, ENTER&STA2, SFT=4500&STA1, SFT=3200&STA3, SFT=1700&EXITSYS:
  2, ENTER&STA1, SFT=6000&STA2, SFT=3700&STA3, SFT=2500&EXITSYS:
  3, ENTER&STA3, SFT=1500&STA1, SFT=7000&STA2, SFT=3200&
STA1, SFT=1500 & EXITSYS;

DISTANCES: 1, 1-5, 200, 250, 250, 125/
            150, 150, 200/
            300, 200/
            175;

COUNTERS: 8:
            9, PART1S:
            10, PART2S:
            11, PART3S;

OUTPUTS: 1, PROF, PROFIT;

SEEDS: 1, 12345:
       2, 34567:
       3, 79961:
       4, 20583:
       5, 15719:
       6, 81039:
       7, 49879:
       8, 63333:
       9, 45039;

FILES: INVALUE, "INPUT1.DAT", SEQ, FRE:
       OUTVALUE, "OUTPUT.DAT", SEQ, FRE;

REPLICATE, 1, 0, 1200, , 240;
END;
B.7.1 Job-shop Problem 8 Model File

; JOBSHOP PROBLEM #8
; 1 rep. = 2 day

BEGIN;
;INITIALIZE SYSTEM VARIABLES

;Read data from INPUT.DAT
CREATE;
READ, INVALUE: WIP, NUMAGV, MR(1), MR(2), MR(3);
READ, INVALUE: PERCS(1), PERCS(2), PERCS(3), PT1, PT2;
READ, INVALUE: RSEL;
CLOSE, INVALUE;

;Set Number of AGVs
BRANCH, I:
IF, NUMAGV.LT.5, 4AGVS:
  IF, NUMAGV.LT.4, 3AGVS:
    IF, NUMAGV.LT.3, 2AGVS:
      IF, NUMAGV.LT.2, 1AGV:
      ELSE, CNT;

4AGVS ASSIGN: IT(AGV, 5) = 2: NEXT (CNT);
3AGVS ASSIGN: IT(AGV, 4) = 2: NEXT (CNT);
2AGVS ASSIGN: IT(AGV, 3) = 2: NEXT (CNT);
1AGV ASSIGN: IT(AGV, 2) = 2: NEXT (CNT);

;Set Cutting Speeds, Part mix, Tool Lifes,
CNT ASSIGN: CS(1) = MAXCS(1) * PERCS(1):
  CS(2) = MAXCS(2) * PERCS(2):
  CS(3) = MAXCS(3) * PERCS(3);
ASSIGN: PT3 = 1 - PT1 - PT2:
  TOOLLIFE(1) = 80000 / (CS(1)**1.2):
  TOOLLIFE(2) = 80000 / (CS(2)**1.2):
  TOOLLIFE(3) = 80000 / (CS(3)**1.2):
END OF REPLICATION

;Write profit to OUTPUT.DAT
CREATE, I, TFIN;
ASSIGN: PROF = 200 * NC(PART1S) + 175 * NC(PART2S) + 250 * NC(PART3S):
  PROF = PROF - (1525 * MR(1) + 1200 * MR(2) + 900 * MR(3)) * 2:
  PROF = PROF - (12 * NUMAGV* 2 + 26 * TC(1) + 28 * TC(2) + 30 * TC(3)):
  PROF = PROF - 124 * (MR(1) + MR(2) + MR(3)) * 2:
  PROF = PROF - (.12*wip*pt1+.11*wip*pt2+.15*wip*pt3)*2;
WRITE, OUTVALUE: PROF :DISPOSE;

;JOB-SHOP MANUFACTURING SYSTEM

;Parts enter system, assigned part type, wait for AGV
CREATE, WIP;
NEWPART ASSIGN: IS = 0:MARK(TIMEIN);
ASSIGN: NS = DISCRETE(P1,2,P1+P2,1,1,3,9);
ASSIGN: TIMEIN = (-1)*TIMEIN;
ROUTE: 0.0,SEQ;

;ENTER SUBMODEL, AGV request
STATION, ENTER;
BRANCH, 1: IF, RSEL.EQ.1, RE1:
   IF, RSEL.EQ.2, RE2:
   IF, RSEL.EQ.3, RE3:
   ELSE, AGVSTA;
RE1 ASSIGN: RULE = NS:NEXT(AGVSTA);
RE2 ASSIGN: RULE = IS:NEXT(AGVSTA);
RE3 ASSIGN: RULE = TIMEIN:NEXT(AGVSTA);
AGVSTA QUEUE, AGVQ;
   REQUEST: AGV(SDS);
   DELAY: AGVLOAD;
   TRANSPORT: AGV, SEQ;

;MILL WORKSTATIONS
;Send piece to type of mill with smallest queue
STATION, STA1;
DELAY: AGVLOAD;
FREE: AGV;
BRANCH, 1: IF, RSEL.EQ.2, RM2:
   ELSE, MILLSSTA;
RM2 ASSIGN: RULE = IS;
MILLSSTA QUEUE, MILLQ;
   SEIZE: MACHINE(1);
   ASSIGN: MIND = 1: NEXT(GENERIC);

;LATHE WORKSTATIONS
;Send piece to type of lathe with smallest queue
STATION, STA2;
DELAY: AGVLOAD;
FREE: AGV;
BRANCH, 1: IF, RSEL.EQ.2, RL2:
   ELSE, LATHSTA;
RL2 ASSIGN: RULE = IS;
LATHSTA QUEUE, 2;
   SEIZE: MACHINE(2);
   ASSIGN: MIND = 2: NEXT(GENERIC);

;DRILL WORKSTATIONS
;Send piece to type of lathe with smallest queue
STATION, STA3;
DELAY: AGVLOAD;
FREE: AGV;
BRANCH,1: IF, RSEL.EQ.2, RD2:
ELSE, DRILSTA;
RD2 ASSIGN: RULE = IS:NEXT(DRILSTA);
DRILSTA QUEUE, 3;
SEIZE: MACHINE(3);
ASSIGN: MIND = 3: NEXT(GENERIC);

;GENERIC MACHINING SUBMODEL
generic ASSIGN: INDEX = 1;
C1 BRANCH,1: IF, T(MIND,INDEX).EQ.0, C3:
ELSE, C2;
C2 ASSIGN: INDEX = INDEX + 1: NEXT(C1);
C3 ASSIGN: T(MIND,INDEX) = 1;
DELAY: NORM(SFT/CS(MIND),STD(MIND)*SFT/CS(MIND),MIND)+1:
                      MARK(T1);
ASSIGN: T(MIND, INDEX) = 0: MARK(T2);
ASSIGN: TOOLTIME(MIND,INDEX)=TOOLTIME(MIND,INDEX)+(T2-T1);
BRANCH,1: IF, TOOLTIME(MIND,INDEX).GE.TOOLLIFE(MIND), CHG:
ELSE, C4;
C4 RELEASE:MACHINE(MIND): NEXT (AGVSTA);

;EXIT SUBMODEL
STATION,EXITSYS;
DELAY: AGVLOAD;
FREE: AGV;
BRANCH,1: IF, NS.EQ.1,PARTA:
          IF, NS.EQ.2,PARTB:
ELSE, PARTC;
PARTA COUNT: PARTAS:NEXT (NEWPART);
PARTB COUNT: PART2S:NEXT (NEWPART);
PARTC COUNT: PART3S:NEXT (NEWPART);

;CHANGE TOOL SUBMODEL
CHG DUPLICATE: 1, AGVSTA;
ASSIGN: T(MIND,INDEX) = 1;
DELAY: CHGTIME(MIND);
ASSIGN: T(MIND,INDEX) = 0;
ASSIGN: TOOLTIME(MIND,INDEX) = 0;
RELEASE:MACHINE(MIND);
COUNT: MIND:DISPOSE;
END;
B.7.2 Job-shop Problem 8 Experiment File

BEGIN;
PROJECT, 8JSP, SH;

ATTRIBUTES: 1, INDEX:
  2, T1:
  3, T2:
  4, MIND:
  5, SFT:
  6, TIMEIN:
  7, RULE;

VARIABLES: 1, PT1:
  2, PT2:
  3, PT3:
  4, MAXCS(3), 670, 730, 800:
  8, PERCS(3):
  11, CS(3):
  14, STD(3), .025, .025, .03:
  17, TOOLLIFE(3):
  20, CHGTIME(3), 5, 4, 3:
  23, AGVLOAD, .50:
  38, TOOLTIME(3, 10), 0:
  68, TC(3), 0:
  71, T(3, 10), 0:
  101, NUMAGV:
  102, RSEL:
  103, WIP:
  104, PROF;

STATIONS: 1, ENTER:
  2, STA1:
  3, STA2:
  4, STA3:
  5, EXITSYS;

RESOURCES: MACHINE(3);

QUEUES: 1, MILLQ, HVF(RULE):
  2, LATHEQ, HVF(RULE):
  3, DRILLQ, HVF(RULE):
  4, AGVQ, HVF(RULE);

TRANSPORTERS: 1, AGV, 5, 1, 100;

SEQUENCES:
1, ENTER & STA2, SFT=4500 & STA1, SFT=3200 & STA3, SFT=1700 & EXITSYS:
2, ENTER & STA1, SFT=6000 & STA2, SFT=3700 & STA3, SFT=2500 & EXITSYS:
3, ENTER&STA3, SFT=1500&STA1, SFT=7000&STA2, SFT=3200&STA1, SFT=1500&EXITSYS;

DISTANCES: 1, 1-5, 200, 250, 250, 125/
            150, 150, 200/
            300, 200/
            175;

COUNTERS:  3:
            4, PART1S:
            5, PART2S:
            6, PART3S;

OUTPUTS:   1, PROF, PROF;

SEEDS:      1, 32345:
            2, 54567:
            3, 79961:
            4, 95583:
            5, 115719:
            6, 321039:
            7, 549879:
            8, 764333:
            9, 985039;

FILES:     INVALUE, "INPUT.DAT", SEQ, FRE:
            OUTVALUE, "OUTPUT.DAT", SEQ, FRE;

REPLICATE, 1, 0, 1200, , 240;
END;
B.8.1 Job-shop Problem 11 Model File

;JOB SHOP PROBLEM #2
; 1 rep. = 1 day

BEGIN;
;INITIALIZE SYSTEM VARIABLES

;Read data from INPUT.DAT
CREATE;
READ, INVALUE: WIP, NUMAGV, MR(1), MR(2), MR(3);
READ, INVALUE: PERCS(1), PERCS(2), PERCS(3), PT1, PT2,;
READ, INVALUE: milltype, lathetype, drilltype;
CLOSE, INVALUE;

;Set Number of AGVs
BRANCH, S: IF, NUMAGV.LT.5, 4AGVS:
   IF, NUMAGV.LT.4, 3AGVS:
   IF, NUMAGV.LT.3, 2AGVS:
   IF, NUMAGV.LT.2, 1AGV:
   ALWAYS, CONT;
4AGVS ASSIGN: IT(AGV, 5) = 2: DISPOSE;
3AGVS ASSIGN: IT(AGV, 4) = 2: DISPOSE;
2AGVS ASSIGN: IT(AGV, 3) = 2: DISPOSE;
1AGV ASSIGN: IT(AGV, 2) = 2: DISPOSE;

;Assign machine-related variables
CONT1 BRANCH, L1: IF, milltype.EQ.1, m1:
   IF, milltype.EQ.2, m2:
   ELSE, m3;

   m1 ASSIGN: MAXCS(1) = 500:
   STD(1) = .075:
   CHGTIME(1) = 5:
   millcost = 950:
   mtoolcost = 26: NEXT (CONT1);

   m2 ASSIGN: MAXCS(1) = 570:
   STD(1) = .05:
   CHGTIME(1) = 4:
   millcost = 1200:
   mtoolcost = 28: NEXT (CONT1);

   m3 ASSIGN: MAXCS(1) = 670:
   STD(1) = .025:
   CHGTIME(1) = 3:
   millcost = 1525:
   mtoolcost = 30;

CONT1 BRANCH, L1: IF, lathetype.EQ.1, L1:
   IF, lathetype.EQ.2, L2:
   ELSE, L3;

   L1 ASSIGN: MAXCS(2) = 530:
   STD(2) = .075:
CHGTIME(2) = 4:
lathecost = 700:
ltoolcost = 18: NEXT (CONT2);
L2 ASSIGN:
MAXCS(2) = 620:
STD(2) = .05:
CHGTIME(2) = 3:
lathecost = 925:
ltoolcost = 19: NEXT (CONT2);
L3 ASSIGN:
MAXCS(2) = 730:
STD(2) = .025:
CHGTIME(2) = 2:
lathecost = 1200:
ltoolcost = 20;

CONT2 BRANCH, 1:
IF, drilltype.EQ.1, d1:
ELSE, d2;
d1 ASSIGN:
MAXCS(3) = 670:
STD(3) = .06:
CHGTIME(3) = 3:
drillcost = 650:
dtoolcost = 13: NEXT (CONT3);
d2 ASSIGN:
MAXCS(3) = 800:
STD(3) = 0.03:
CHGTIME(3) = 2:
drillcost = 900:
dtoolcost = 15;

;Set Cutting Speeds, Part mix, Tool Lifes,
CONT3 ASSIGN: CS(1) = MAXCS(1) * PERCS(1):
CS(2) = MAXCS(2) * PERCS(2):
CS(3) = MAXCS(3) * PERCS(3):
ASSIGN: PT3 = 1 - PT1 - PT2:
TOOLLIFE(1) = 80000/(CS(1)**1.2):
TOOLLIFE(2) = 80000/(CS(2)**1.2):
TOOLLIFE(3) = 80000/(CS(3)**1.2): DISPOSE;

;END OF REPLICATION
;Write profit to OUTPUT.DAT
CREATE, 1, TFIN;
ASSIGN: PROF = 200*NC(PART1S) + 175*NC(PART2S) + 250*NC(PART3S):
PROF = PROF - (millcost*MR(1) + lathecost*MR(2) + drillcost*MR(3)):
PROF = PROF - 12*NUMAGV:
PROF = PROF - (mtoolcost*TC(1) + ltoolcost*TC(2) + dtoolcost*TC(3)):
PROF = PROF - 124*(MR(1) + MR(2) + MR(3)):
PROF = PROF -.12*wip*pt1 + .11*wip*pt2 + .15*wip*pt3:
WRITE, OUTVALUE: PROF : DISPOSE;

;JOB-SHOP MANUFACTURING SYSTEM

;Parts enter system, assigned part type, wait for AGV
CREATE, wip;
nearpart ASSIGN: IS = 0;
ASSIGN: NS = DISCRETE(PT1,1,PT1+PT2,2,1,3,9);
ROUTE: 0.0,SEQ;

; ENTER SUBMODEL, AGV request
STATION, ENTER;
agvq QUEUE, 4;
REQUEST: AGV(SDS);
DELAY: AGVLOAD;
TRANSPORT: AGV,SEQ;

; MILL WORKSTATIONS
; Send piece to type of mill with smallest queue
STATION, STA1;
DELAY: AGVLOAD;
FREE: AGV;
QUEUE, 1;
SEIZE: MACHINE(1);
ASSIGN: mind = 1: NEXT(generic);

; LATHE WORKSTATIONS
; Send piece to type of lathe with smallest queue
STATION, STA2;
DELAY: AGVLOAD;
FREE: AGV;
QUEUE, 2;
SEIZE: MACHINE(2);
ASSIGN: mind = 2: NEXT(generic);

; DRILL WORKSTATIONS
; Send piece to type of lathe with smallest queue
STATION, STA3;
DELAY: AGVLOAD;
FREE: AGV;
QUEUE, 3;
SEIZE: MACHINE(3);
ASSIGN: mind = 3: NEXT(generic);

generic ASSIGN: INDEX = 1;
C1 BRANCH, 1: IF, T(mind, INDEX) .EQ.0, C3:
ELSE, C2;
C2 ASSIGN: INDEX = INDEX + 1: NEXT(C1);
C3 ASSIGN: T(mind, INDEX) = 1;
DELAY: NORM(sft/CS(mind),STD(mind)*sft/CS(mind),mind)+1:
MARK(T1);
ASSIGN: T(mind, INDEX) = 0: MARK(T2);
ASSIGN: TOOLTIME(mind, INDEX)=TOOLTIME(mind, INDEX)+(T2-T1);
BRANCH, 1: IF, TOOLTIME(mind, INDEX) .GE. TOOLLIFE(mind), CHG:
ELSE, C4;
C4 RELEASE: MACHINE(mind): NEXT (agvq);

; EXIT SUBMODEL
STATION, EXITSYS;
DELAY: AGVLOAD;
FREE: AGV;
BRANCH, 1: IF, NS.EQ.1, PARTA:
    IF, NS.EQ.2, PARTB:
    ELSE, PARTC;
PARTA COUNT: PART1S:NEXT (newpart);
PARTB COUNT: PART2S:NEXT (newpart);
PARTC COUNT: PART3S:NEXT (newpart);

;CHANGE TOOL SUBMODEL
CHG DUPLICATE: 1, agvq;
ASSIGN: T(mind, INDEX) = 1;
DELAY: CHGTIME(mind);
ASSIGN: T(mind, INDEX) = 0;
ASSIGN: TOOLTIME(mind, INDEX) = 0;
RELEASE: MACHINE(mind);
ASSIGN: TC(mind) = TC(mind) + 1 : DISPOSE;
END;
BEGIN;
PROJECT, 2JS1, SH;

ATTRIBUTES: 1, INDEX:
  2, T1:
  3, T2:
  4, mind:
  5, sft;

VARIABLES: milltype:
lathetype:
drilltype:
millcost:
lathecost:
drillcost:
mtoolcost:
ltoolcost:
dtoolcost:
PT1:
PT2:
PT3:
MAXCS(3):
PERCS(3):
CS(3):
STD(3):
TOOLLIFE(3):
CHGTIME(3):
AGVLOAD, .50:
TOOLTIME(3,10), 0:
TC(3), 0:
T(3,10), 0:
NUMAGV:
wip, 20:
PROF;

STATIONS: 1, ENTER:
  2, STA1:
  3, STA2:
  4, STA3:
  5, EXITSYS;

RESOURCES: MACHINE(3);

QUEUES: 4;

TRANSPORTERS: 1, AGV, 7, 1, 100;

SEQUENCES:
  1, ENTER&STA1, sft=6000&STA2, sft=3700&STA3, sft=2500&EXITSYS:
DISTANCES: 1, 1-5, 200, 250, 250, 125/
150, 150, 200/
300, 200/
175;

COUNTERS: 1, PART1S:
2, PART2S:
3, PART3S;

DSTATS: 1, NQ(4), WAIT_FOR_AGV:
2, NQ(1), WAIT_MILL:
3, NQ(2), WAIT_LATHE:
4, NQ(3), WAIT_DRILL:
5, NT(AGV), AGV_USED:
6, NR(1), MILL:
7, NR(2), LATHE:
8, NR(3), DRILL;

OUTPUTS: 1, PROF., PROFIT;

SEEDS: 1, 14615:
2, 10019:
3, 41673:
4, 25863:
5, 19123:
6, 92479:
7, 9903:
8, 66637:
9, 44713;

FILES: INVALUE, "INPUT.DAT", SEQ, FRE:
OUTVALUE, "OUTPUT.DAT", SEQ, FRE;

REPLICATE, 1, 0, 480;
END;
Appendix C
Results of Experiments

The following two tables show the detailed results of the two experiments discussed in Chapter 4. The solution, number of replications, and ratings are given for each genetic algorithm applied to each test problem used in the experiments.
Table C.1 Results of genetic operator rates experiment.

<table>
<thead>
<tr>
<th>Genetic Algorithm Description Crossover Rate</th>
<th>Equation Problem 1</th>
<th>Equation Problem 1</th>
<th>Restaurant Problem</th>
<th>Distribution Problem 2</th>
<th>Overall average rating</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>f(X) min. rating</td>
<td>f(X) max. rating</td>
<td>profit($) rating</td>
<td>cost($) rating</td>
<td>rating</td>
</tr>
<tr>
<td>0.25 0.10 0.001</td>
<td>800 4.81 0.937</td>
<td>800 193.29 0.969</td>
<td>450 1920 1.000</td>
<td>800 770.77 0.999</td>
<td>0.976</td>
</tr>
<tr>
<td>0.25 0.10 0.005</td>
<td>800 5.02 0.934</td>
<td>800 194.12 0.976</td>
<td>450 1920 1.000</td>
<td>800 763.02 1.000</td>
<td>0.977</td>
</tr>
<tr>
<td>0.25 0.10 0.010</td>
<td>800 1.62 0.979</td>
<td>800 193.17 0.968</td>
<td>450 1920 1.000</td>
<td>800 764.16 1.000</td>
<td>0.987</td>
</tr>
<tr>
<td>0.25 0.25 0.001</td>
<td>800 2.73 0.964</td>
<td>800 190.82 0.949</td>
<td>450 1920 1.000</td>
<td>800 866.32 0.992</td>
<td>0.976</td>
</tr>
<tr>
<td>0.25 0.25 0.005</td>
<td>800 1.33 0.983</td>
<td>800 195.13 0.984</td>
<td>450 1920 1.000</td>
<td>800 793.21 0.998</td>
<td>0.991</td>
</tr>
<tr>
<td>0.25 0.25 0.010</td>
<td>800 1.21 0.984</td>
<td>800 185.01 0.901</td>
<td>450 1920 1.000</td>
<td>800 789.15 0.998</td>
<td>0.971</td>
</tr>
<tr>
<td>0.25 0.40 0.001</td>
<td>800 3.70 0.951</td>
<td>800 180.71 0.865</td>
<td>450 1890 0.976</td>
<td>800 773.54 0.999</td>
<td>0.948</td>
</tr>
<tr>
<td>0.25 0.40 0.005</td>
<td>800 1.89 0.975</td>
<td>800 187.57 0.922</td>
<td>450 1890 0.976</td>
<td>800 788.49 0.998</td>
<td>0.968</td>
</tr>
<tr>
<td>0.25 0.40 0.010</td>
<td>800 0.27 0.997</td>
<td>800 184.57 0.897</td>
<td>450 1890 0.976</td>
<td>800 795.08 0.997</td>
<td>0.967</td>
</tr>
<tr>
<td>0.50 0.10 0.001</td>
<td>800 1.80 0.977</td>
<td>800 188.72 0.931</td>
<td>450 1920 1.000</td>
<td>800 794.02 0.998</td>
<td>0.976</td>
</tr>
<tr>
<td>0.50 0.10 0.005</td>
<td>800 2.49 0.967</td>
<td>800 189.49 0.938</td>
<td>450 1920 1.000</td>
<td>800 780.68 0.999</td>
<td>0.976</td>
</tr>
<tr>
<td>0.50 0.10 0.010</td>
<td>800 2.54 0.967</td>
<td>800 191.45 0.954</td>
<td>450 1920 1.000</td>
<td>800 774.51 0.999</td>
<td>0.980</td>
</tr>
<tr>
<td>0.50 0.25 0.001</td>
<td>800 4.53 0.940</td>
<td>800 185.01 0.901</td>
<td>450 1920 1.000</td>
<td>800 774.51 0.999</td>
<td>0.960</td>
</tr>
<tr>
<td>0.50 0.25 0.005</td>
<td>800 1.52 0.980</td>
<td>800 192.32 0.961</td>
<td>450 1920 1.000</td>
<td>800 774.51 0.999</td>
<td>0.985</td>
</tr>
<tr>
<td>0.50 0.25 0.010</td>
<td>800 1.02 0.987</td>
<td>800 193.21 0.968</td>
<td>450 1920 1.000</td>
<td>800 774.51 0.999</td>
<td>0.989</td>
</tr>
<tr>
<td>0.50 0.40 0.001</td>
<td>800 4.81 0.937</td>
<td>800 193.21 0.968</td>
<td>450 1920 1.000</td>
<td>800 826.56 0.995</td>
<td>0.975</td>
</tr>
<tr>
<td>0.50 0.40 0.005</td>
<td>800 2.09 0.973</td>
<td>800 193.17 0.968</td>
<td>450 1890 0.976</td>
<td>800 786.73 0.998</td>
<td>0.979</td>
</tr>
<tr>
<td>0.50 0.40 0.010</td>
<td>800 1.96 0.974</td>
<td>800 193.21 0.968</td>
<td>450 1890 0.976</td>
<td>800 789.53 0.998</td>
<td>0.979</td>
</tr>
<tr>
<td>0.75 0.10 0.001</td>
<td>800 1.26 0.984</td>
<td>800 191.53 0.954</td>
<td>450 1890 0.976</td>
<td>800 764.16 1.000</td>
<td>0.978</td>
</tr>
<tr>
<td>0.75 0.10 0.005</td>
<td>800 1.50 0.980</td>
<td>800 197.05 1.000</td>
<td>450 1890 0.976</td>
<td>800 770.77 0.999</td>
<td>0.989</td>
</tr>
<tr>
<td>0.75 0.10 0.010</td>
<td>800 1.02 0.987</td>
<td>800 193.21 0.968</td>
<td>450 1890 0.976</td>
<td>800 765.15 1.000</td>
<td>0.983</td>
</tr>
<tr>
<td>0.75 0.25 0.001</td>
<td>800 0.53 0.993</td>
<td>800 195.11 0.984</td>
<td>450 1920 1.000</td>
<td>800 771.86 0.999</td>
<td>0.994</td>
</tr>
<tr>
<td>0.75 0.25 0.005</td>
<td>800 1.46 0.981</td>
<td>800 195.11 0.984</td>
<td>450 1920 1.000</td>
<td>800 771.86 0.999</td>
<td>0.991</td>
</tr>
<tr>
<td>0.75 0.25 0.010</td>
<td>800 1.81 0.976</td>
<td>800 197.03 1.000</td>
<td>450 1920 1.000</td>
<td>800 780.95 0.999</td>
<td>0.994</td>
</tr>
<tr>
<td>0.75 0.40 0.001</td>
<td>800 0.05 1.000</td>
<td>800 193.21 0.968</td>
<td>450 1920 1.000</td>
<td>800 786.31 0.998</td>
<td>0.992</td>
</tr>
<tr>
<td>0.75 0.40 0.005</td>
<td>800 0.02 1.000</td>
<td>800 193.21 0.968</td>
<td>450 1920 1.000</td>
<td>800 786.73 0.998</td>
<td>0.992</td>
</tr>
<tr>
<td>0.75 0.40 0.010</td>
<td>800 0.37 0.995</td>
<td>800 191.41 0.953</td>
<td>450 1920 1.000</td>
<td>800 799.09 0.997</td>
<td>0.986</td>
</tr>
</tbody>
</table>
### Table C.2 Results of final design issues experiment.

<table>
<thead>
<tr>
<th>Genetic Algorithm Description</th>
<th>Replication Increase</th>
<th>Pop. Size</th>
<th>Distribution Problem 1</th>
<th>AS/RS Problem 1</th>
<th>Job-shop Problem 11</th>
<th>overall average rating</th>
</tr>
</thead>
<tbody>
<tr>
<td>Convergence Measure</td>
<td></td>
<td></td>
<td>Distribution Problem 1</td>
<td>cost($) rating</td>
<td>cost($) rating</td>
<td>profit($) rating</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>reps. cost($) min. rating</td>
<td>reps. cost($) min. rating</td>
<td>reps. profit($) max. rating</td>
</tr>
<tr>
<td>no improv.</td>
<td>GA @ 1 =&gt; GA @ 7</td>
<td>50</td>
<td>3749</td>
<td>901.31 0.996</td>
<td>1762</td>
<td>188844 1.000</td>
</tr>
<tr>
<td>no improv.</td>
<td>GA @ 1 =&gt; GA @ 7</td>
<td>100</td>
<td>5653</td>
<td>887.12 1.000</td>
<td>1705</td>
<td>188911 1.000</td>
</tr>
<tr>
<td>no improv.</td>
<td>GA @ 1 =&gt; GA @ 7</td>
<td>200</td>
<td>7183</td>
<td>887.12 1.000</td>
<td>2507</td>
<td>189030 1.000</td>
</tr>
<tr>
<td>no improv.</td>
<td>GA @ 1 =&gt; re-eval. @ 7</td>
<td>50</td>
<td>1719</td>
<td>937.69 0.987</td>
<td>1188</td>
<td>188844 1.000</td>
</tr>
<tr>
<td>no improv.</td>
<td>GA @ 1 =&gt; re-eval. @ 7</td>
<td>100</td>
<td>2705</td>
<td>922.56 0.991</td>
<td>1396</td>
<td>188911 1.000</td>
</tr>
<tr>
<td>no improv.</td>
<td>GA @ 1 =&gt; re-eval. @ 7</td>
<td>200</td>
<td>4412</td>
<td>922.56 0.991</td>
<td>2168</td>
<td>189030 1.000</td>
</tr>
<tr>
<td>no improv.</td>
<td>GA @ 1 =&gt; re-eval. @ 7</td>
<td>50</td>
<td>879</td>
<td>971.06 0.978</td>
<td>712</td>
<td>188911 1.000</td>
</tr>
<tr>
<td>no improv.</td>
<td>GA @ 1 =&gt; re-eval. @ 7</td>
<td>100</td>
<td>1845</td>
<td>971.06 0.978</td>
<td>984</td>
<td>188979 1.000</td>
</tr>
<tr>
<td>no improv.</td>
<td>GA @ 1 =&gt; re-eval. @ 7</td>
<td>200</td>
<td>3088</td>
<td>971.90 0.978</td>
<td>1716</td>
<td>189030 1.000</td>
</tr>
<tr>
<td>3-sigma</td>
<td>GA @ 1 =&gt; GA @ 7</td>
<td>50</td>
<td>1362</td>
<td>948.30 0.984</td>
<td>n/a</td>
<td>n/a 0.000</td>
</tr>
<tr>
<td>3-sigma</td>
<td>GA @ 1 =&gt; GA @ 7</td>
<td>100</td>
<td>7178</td>
<td>887.12 1.000</td>
<td>n/a</td>
<td>n/a 0.000</td>
</tr>
<tr>
<td>3-sigma</td>
<td>GA @ 1 =&gt; GA @ 7</td>
<td>200</td>
<td>7588</td>
<td>913.66 0.993</td>
<td>n/a</td>
<td>n/a 0.000</td>
</tr>
<tr>
<td>3-sigma</td>
<td>GA @ 1 =&gt; GA @ 7</td>
<td>50</td>
<td>826</td>
<td>976.63 0.977</td>
<td>n/a</td>
<td>n/a 0.000</td>
</tr>
<tr>
<td>3-sigma</td>
<td>GA @ 1 =&gt; GA @ 7</td>
<td>100</td>
<td>2223</td>
<td>950.88 0.984</td>
<td>n/a</td>
<td>n/a 0.000</td>
</tr>
<tr>
<td>3-sigma</td>
<td>GA @ 1 =&gt; GA @ 7</td>
<td>200</td>
<td>5449</td>
<td>935.48 0.988</td>
<td>n/a</td>
<td>n/a 0.000</td>
</tr>
<tr>
<td>3-sigma</td>
<td>GA @ 1 =&gt; re-eval. @ 7</td>
<td>50</td>
<td>774</td>
<td>1007.85 0.969</td>
<td>n/a</td>
<td>n/a 0.000</td>
</tr>
<tr>
<td>3-sigma</td>
<td>GA @ 1 =&gt; re-eval. @ 7</td>
<td>100</td>
<td>1347</td>
<td>1013.99 0.967</td>
<td>n/a</td>
<td>n/a 0.000</td>
</tr>
<tr>
<td>3-sigma</td>
<td>GA @ 1 =&gt; re-eval. @ 7</td>
<td>200</td>
<td>1897</td>
<td>1072.31 0.952</td>
<td>n/a</td>
<td>n/a 0.000</td>
</tr>
<tr>
<td>both 1 &amp; 2</td>
<td>GA @ 1 =&gt; GA @ 7</td>
<td>50</td>
<td>2034</td>
<td>916.21 0.993</td>
<td>1762</td>
<td>188844 1.000</td>
</tr>
<tr>
<td>both 1 &amp; 2</td>
<td>GA @ 1 =&gt; GA @ 7</td>
<td>100</td>
<td>1942</td>
<td>981.62 0.976</td>
<td>1705</td>
<td>188911 1.000</td>
</tr>
<tr>
<td>both 1 &amp; 2</td>
<td>GA @ 1 =&gt; GA @ 7</td>
<td>200</td>
<td>10602</td>
<td>887.12 1.000</td>
<td>2507</td>
<td>189030 1.000</td>
</tr>
<tr>
<td>both 1 &amp; 2</td>
<td>GA @ 1 =&gt; re-eval. @ 7</td>
<td>50</td>
<td>3680</td>
<td>935.48 0.988</td>
<td>1188</td>
<td>188844 1.000</td>
</tr>
<tr>
<td>both 1 &amp; 2</td>
<td>GA @ 1 =&gt; re-eval. @ 7</td>
<td>100</td>
<td>1785</td>
<td>954.15 0.983</td>
<td>1396</td>
<td>188911 1.000</td>
</tr>
<tr>
<td>both 1 &amp; 2</td>
<td>GA @ 1 =&gt; re-eval. @ 7</td>
<td>200</td>
<td>6981</td>
<td>946.24 0.985</td>
<td>2168</td>
<td>189030 1.000</td>
</tr>
<tr>
<td>both 1 &amp; 2</td>
<td>GA @ 1 =&gt; re-eval. @ 7</td>
<td>50</td>
<td>676</td>
<td>1031.32 0.963</td>
<td>712</td>
<td>188911 1.000</td>
</tr>
<tr>
<td>both 1 &amp; 2</td>
<td>GA @ 1 =&gt; re-eval. @ 7</td>
<td>100</td>
<td>1697</td>
<td>980.66 0.976</td>
<td>984</td>
<td>188979 1.000</td>
</tr>
<tr>
<td>both 1 &amp; 2</td>
<td>GA @ 1 =&gt; re-eval. @ 7</td>
<td>200</td>
<td>1845</td>
<td>1046.36 0.959</td>
<td>1716</td>
<td>189030 1.000</td>
</tr>
</tbody>
</table>
Appendix D

Results of Comparison

The following tables show the results of the genetic algorithm and the simulated annealing algorithm applied to the 20 test problems. The solutions given by each algorithm are presented with the input variable values that produced those solutions.

Table D.1 Results of standard buffer problem.

<table>
<thead>
<tr>
<th>Variables</th>
<th>Standard Buffer Problem</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>GA</td>
</tr>
<tr>
<td>buffer 1</td>
<td>28</td>
</tr>
<tr>
<td>buffer 2</td>
<td>172</td>
</tr>
<tr>
<td>buffer 3</td>
<td>137</td>
</tr>
<tr>
<td>buffer 4</td>
<td>59</td>
</tr>
<tr>
<td>Replications</td>
<td>2098</td>
</tr>
<tr>
<td>Lower C.I.</td>
<td>49456</td>
</tr>
<tr>
<td>Cost (min.)</td>
<td>54654</td>
</tr>
<tr>
<td>Upper C.I.</td>
<td>59853</td>
</tr>
<tr>
<td>Rating</td>
<td>1.000</td>
</tr>
</tbody>
</table>
Table D.2 Results of distribution problems 1, 3, 4, and 5.

<table>
<thead>
<tr>
<th>Variables</th>
<th>Distribution Problem 1</th>
<th>Distribution Problem 3</th>
<th>Distribution Problem 4</th>
<th>Distribution Problem 5</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>GA</td>
<td>SA</td>
<td>GA</td>
<td>SA</td>
</tr>
<tr>
<td>buffer 1</td>
<td>4</td>
<td>16</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>buffer 2</td>
<td>0</td>
<td>3</td>
<td>0</td>
<td>14</td>
</tr>
<tr>
<td>buffer 3</td>
<td>8</td>
<td>14</td>
<td>25</td>
<td>31</td>
</tr>
<tr>
<td>buffer 4</td>
<td>10</td>
<td>6</td>
<td>11</td>
<td>9</td>
</tr>
<tr>
<td>ship size</td>
<td>22</td>
<td>14</td>
<td>33</td>
<td>19</td>
</tr>
</tbody>
</table>

Results

<table>
<thead>
<tr>
<th>replications</th>
<th>3638</th>
<th>776</th>
<th>2617</th>
<th>776</th>
<th>3703</th>
<th>776</th>
<th>1807</th>
<th>776</th>
</tr>
</thead>
<tbody>
<tr>
<td>lower C.I.</td>
<td>919</td>
<td>978</td>
<td>1236</td>
<td>1469</td>
<td>950</td>
<td>1084</td>
<td>1635</td>
<td>1789</td>
</tr>
<tr>
<td>cost ($)</td>
<td>974</td>
<td>1069</td>
<td>1367</td>
<td>1547</td>
<td>1020</td>
<td>1145</td>
<td>1784</td>
<td>2049</td>
</tr>
<tr>
<td>upper C.I.</td>
<td>1028</td>
<td>1161</td>
<td>1497</td>
<td>1624</td>
<td>1090</td>
<td>1205</td>
<td>1933</td>
<td>2309</td>
</tr>
<tr>
<td>rating</td>
<td>1.000</td>
<td>0.902</td>
<td>1.000</td>
<td>0.868</td>
<td>1.000</td>
<td>0.878</td>
<td>1.000</td>
<td>0.851</td>
</tr>
</tbody>
</table>

Table D.3 Results of distribution problem 6.

<table>
<thead>
<tr>
<th>Variables</th>
<th>Distribution Problem 6</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>GA</td>
</tr>
<tr>
<td>buffer 1</td>
<td>10</td>
</tr>
<tr>
<td>buffer 2</td>
<td>0</td>
</tr>
<tr>
<td>buffer 3</td>
<td>0</td>
</tr>
<tr>
<td>buffer 4</td>
<td>0</td>
</tr>
<tr>
<td>buffer 5</td>
<td>14</td>
</tr>
<tr>
<td>buffer 6</td>
<td>4</td>
</tr>
<tr>
<td>buffer 7</td>
<td>1</td>
</tr>
<tr>
<td>buffer 8</td>
<td>14</td>
</tr>
<tr>
<td>ship size 1</td>
<td>13</td>
</tr>
<tr>
<td>ship size 2</td>
<td>13</td>
</tr>
<tr>
<td>ship size 3</td>
<td>13</td>
</tr>
</tbody>
</table>

Results

<table>
<thead>
<tr>
<th>replications</th>
<th>4357</th>
<th>1358</th>
</tr>
</thead>
<tbody>
<tr>
<td>lower C.I.</td>
<td>3623</td>
<td>4386</td>
</tr>
<tr>
<td>cost ($)</td>
<td>4086</td>
<td>4900</td>
</tr>
<tr>
<td>upper C.I.</td>
<td>4549</td>
<td>5413</td>
</tr>
<tr>
<td>rating</td>
<td>1.000</td>
<td>0.801</td>
</tr>
</tbody>
</table>
Table D.4 Results of AS/RS problems 1 - 3.

<table>
<thead>
<tr>
<th>Variables</th>
<th>AS/RS Problem 1</th>
<th>AS/RS Problem 2</th>
<th>AS/RS Problem 3</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>GA</td>
<td>SA</td>
<td>GA</td>
</tr>
<tr>
<td>bays</td>
<td>50</td>
<td>72</td>
<td>36</td>
</tr>
<tr>
<td>levels</td>
<td>10</td>
<td>7</td>
<td>7</td>
</tr>
<tr>
<td>horiz. max. speed</td>
<td>420</td>
<td>400</td>
<td>360</td>
</tr>
<tr>
<td>vert. max. speed</td>
<td>30</td>
<td>40</td>
<td>40</td>
</tr>
<tr>
<td>rack depth</td>
<td>double</td>
<td>double</td>
<td>double</td>
</tr>
<tr>
<td>Results</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>replications</td>
<td>1312</td>
<td>826</td>
<td>1432</td>
</tr>
<tr>
<td>lower C.I.</td>
<td>n/a</td>
<td>n/a</td>
<td>n/a</td>
</tr>
<tr>
<td>cost ($)</td>
<td>188844</td>
<td>189190</td>
<td>96735</td>
</tr>
<tr>
<td>upper C.I.</td>
<td>n/a</td>
<td>n/a</td>
<td>n/a</td>
</tr>
<tr>
<td>rating</td>
<td>1.000</td>
<td>0.998</td>
<td>1.000</td>
</tr>
</tbody>
</table>
Table D.5 Results of job-shop problems 1 - 4.

<table>
<thead>
<tr>
<th>Variables</th>
<th>Job-shop Problem 1</th>
<th>Job-shop Problem 2</th>
<th>Job-shop Problem 3</th>
<th>Job-shop Problem 4</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>GA</td>
<td>SA</td>
<td>GA</td>
<td>SA</td>
</tr>
<tr>
<td>Work-In-Process level</td>
<td>50</td>
<td>28</td>
<td>48</td>
<td>44</td>
</tr>
<tr>
<td>number of AGVs</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>number of high-end mills</td>
<td>4</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>number of mid-range mills</td>
<td>1</td>
<td>3</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>number of low-end mills</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>number of high-end lathes</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>number of mid-range lathes</td>
<td>2</td>
<td>2</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>number of low-end lathes</td>
<td>0</td>
<td>0</td>
<td>4</td>
<td>0</td>
</tr>
<tr>
<td>number of high-end drills</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>number of low-end drills</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>% cut. speed, high mills</td>
<td>0.850</td>
<td>0.950</td>
<td>0.925</td>
<td>0.800</td>
</tr>
<tr>
<td>% cut. speed, mid mills</td>
<td>0.750</td>
<td>0.925</td>
<td>0.925</td>
<td>0.900</td>
</tr>
<tr>
<td>% cut. speed, low mills</td>
<td>0.425</td>
<td>0.375</td>
<td>0.850</td>
<td>0.250</td>
</tr>
<tr>
<td>% cut. speed, high lathes</td>
<td>0.925</td>
<td>0.850</td>
<td>0.775</td>
<td>0.800</td>
</tr>
<tr>
<td>% cut. speed, mid lathes</td>
<td>0.825</td>
<td>1.000</td>
<td>0.925</td>
<td>0.975</td>
</tr>
<tr>
<td>% cut. speed, low lathes</td>
<td>0.900</td>
<td>0.325</td>
<td>0.850</td>
<td>0.375</td>
</tr>
<tr>
<td>% cut. speed, high drills</td>
<td>0.900</td>
<td>0.825</td>
<td>0.950</td>
<td>0.925</td>
</tr>
<tr>
<td>% cut. speed, low drills</td>
<td>0.825</td>
<td>0.800</td>
<td>0.775</td>
<td>0.925</td>
</tr>
<tr>
<td>% part 1</td>
<td>0.050</td>
<td>0.075</td>
<td>0.075</td>
<td>0.150</td>
</tr>
<tr>
<td>% part 2</td>
<td>0.050</td>
<td>0.400</td>
<td>0.900</td>
<td>0.300</td>
</tr>
<tr>
<td>scheduling rules</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>Results</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>replications</td>
<td>2365</td>
<td>2832</td>
<td>2828</td>
<td>2832</td>
</tr>
<tr>
<td>lower C.I.</td>
<td>34332</td>
<td>31322</td>
<td>48024</td>
<td>36508</td>
</tr>
<tr>
<td>profit ($ )</td>
<td>35098</td>
<td>32228</td>
<td>48526</td>
<td>37432</td>
</tr>
<tr>
<td>upper C.I.</td>
<td>35864</td>
<td>33134</td>
<td>49028</td>
<td>38357</td>
</tr>
<tr>
<td>rating</td>
<td>1.000</td>
<td>0.918</td>
<td>1.000</td>
<td>0.771</td>
</tr>
<tr>
<td>Variables</td>
<td>Job-shop Problem 5</td>
<td>Job-shop Problem 6</td>
<td>Job-shop Problem 7</td>
<td></td>
</tr>
<tr>
<td>---------------------------------</td>
<td>-------------------</td>
<td>-------------------</td>
<td>-------------------</td>
<td></td>
</tr>
<tr>
<td></td>
<td>GA</td>
<td>SA</td>
<td>GA</td>
<td>SA</td>
</tr>
<tr>
<td>Work-In-Process level</td>
<td>29</td>
<td>39</td>
<td>28</td>
<td>42</td>
</tr>
<tr>
<td>number of AGVs</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>number of high-end mills</td>
<td>0</td>
<td>3</td>
<td>0</td>
<td>3</td>
</tr>
<tr>
<td>number of mid-range mills</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>number of low-end mills</td>
<td>5</td>
<td>0</td>
<td>4</td>
<td>0</td>
</tr>
<tr>
<td>number of high-end lathes</td>
<td>0</td>
<td>2</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>number of mid-range lathes</td>
<td>2</td>
<td>0</td>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>number of low-end lathes</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>number of high-end drills</td>
<td>2</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>number of low-end drills</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>% cut. speed of high-end mills</td>
<td>0.800</td>
<td>0.975</td>
<td>0.950</td>
<td>0.750</td>
</tr>
<tr>
<td>% cut. speed of mid-range mills</td>
<td>0.900</td>
<td>0.850</td>
<td>0.975</td>
<td>0.775</td>
</tr>
<tr>
<td>% cut. speed of low-end mills</td>
<td>0.975</td>
<td>0.425</td>
<td>0.950</td>
<td>0.350</td>
</tr>
<tr>
<td>% cut. speed of high-end lathes</td>
<td>0.775</td>
<td>0.950</td>
<td>0.850</td>
<td>0.850</td>
</tr>
<tr>
<td>% cut. speed of mid-range lathes</td>
<td>0.900</td>
<td>0.850</td>
<td>0.925</td>
<td>0.875</td>
</tr>
<tr>
<td>% cut. speed of low-end lathes</td>
<td>0.975</td>
<td>0.475</td>
<td>0.400</td>
<td>0.325</td>
</tr>
<tr>
<td>% cut. speed of high-end drills</td>
<td>0.875</td>
<td>0.800</td>
<td>0.900</td>
<td>0.900</td>
</tr>
<tr>
<td>% cut. speed of low-end drills</td>
<td>0.825</td>
<td>0.800</td>
<td>0.925</td>
<td>0.850</td>
</tr>
<tr>
<td>% part 1</td>
<td>0.150</td>
<td>0.250</td>
<td>0.800</td>
<td>0.100</td>
</tr>
<tr>
<td>% part 2</td>
<td>0.075</td>
<td>0.425</td>
<td>0.025</td>
<td>0.425</td>
</tr>
<tr>
<td>scheduling rules</td>
<td>0</td>
<td>0</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>Results</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>replications</td>
<td>3059</td>
<td>2832</td>
<td>3452</td>
<td>2832</td>
</tr>
<tr>
<td>lower C.I.</td>
<td>32466</td>
<td>27502</td>
<td>31737</td>
<td>26389</td>
</tr>
<tr>
<td>profit ($)</td>
<td>32917</td>
<td>28322</td>
<td>32853</td>
<td>27236</td>
</tr>
<tr>
<td>upper C.I.</td>
<td>33369</td>
<td>29142</td>
<td>33970</td>
<td>28084</td>
</tr>
<tr>
<td>rating</td>
<td>1.000</td>
<td>0.860</td>
<td>1.000</td>
<td>0.829</td>
</tr>
</tbody>
</table>
Table D.7 Results of job-shop problems 8 - 11.

<table>
<thead>
<tr>
<th>Variables</th>
<th>Job-shop Problem 8</th>
<th>Job-shop Problem 9</th>
<th>Job-shop Problem 10</th>
<th>Job-shop Problem 11</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>GA</td>
<td>SA</td>
<td>GA</td>
<td>SA</td>
</tr>
<tr>
<td>Work-In-Process level</td>
<td>67</td>
<td>18</td>
<td>65</td>
<td>34</td>
</tr>
<tr>
<td>number of AGVs</td>
<td>5</td>
<td>2</td>
<td>5</td>
<td>1</td>
</tr>
<tr>
<td>number of mills</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>number of lathes</td>
<td>2</td>
<td>2</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>number of drills</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>% cutting speed of mills</td>
<td>0.975</td>
<td>0.650</td>
<td>0.675</td>
<td>0.700</td>
</tr>
<tr>
<td>% cutting speed of lathes</td>
<td>0.875</td>
<td>1.000</td>
<td>0.725</td>
<td>0.650</td>
</tr>
<tr>
<td>% cutting speed of drills</td>
<td>0.800</td>
<td>0.500</td>
<td>0.550</td>
<td>0.700</td>
</tr>
<tr>
<td>% part 1</td>
<td>0.000</td>
<td>0.300</td>
<td>0.100</td>
<td>0.100</td>
</tr>
<tr>
<td>% part 2</td>
<td>0.550</td>
<td>0.575</td>
<td>0.775</td>
<td>0.500</td>
</tr>
<tr>
<td>mill type</td>
<td>n/a</td>
<td>n/a</td>
<td>n/a</td>
<td>n/a</td>
</tr>
<tr>
<td>lathe type</td>
<td>n/a</td>
<td>n/a</td>
<td>n/a</td>
<td>n/a</td>
</tr>
<tr>
<td>drill type</td>
<td>n/a</td>
<td>n/a</td>
<td>n/a</td>
<td>n/a</td>
</tr>
<tr>
<td>scheduling rules</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>0</td>
</tr>
<tr>
<td>Results</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>replications</td>
<td>2875</td>
<td>1652</td>
<td>2228</td>
<td>1652</td>
</tr>
<tr>
<td>lower C.I.</td>
<td>41866</td>
<td>20696</td>
<td>33336</td>
<td>18577</td>
</tr>
<tr>
<td>profit ($)</td>
<td>42982</td>
<td>21801</td>
<td>34278</td>
<td>19794</td>
</tr>
<tr>
<td>upper C.I.</td>
<td>44099</td>
<td>22906</td>
<td>35220</td>
<td>21011</td>
</tr>
<tr>
<td>rating</td>
<td>1.000</td>
<td>0.507</td>
<td>1.000</td>
<td>0.577</td>
</tr>
</tbody>
</table>
Abstract

An automated, robust optimization algorithm coupled with simulation would provide a powerful tool for the analysis and design of systems. Such a technique would function on a wide variety of problems and find optimal or near-optimal solutions in the least amount of time possible. The technique would search the problem's solution space by selecting system inputs based on simulation output. The purpose of this thesis was to investigate the feasibility of using a genetic algorithm for robust simulation optimization.

A genetic algorithm was developed from literature consultation and two experiments. It was applied to 20 simulation test problems. To measure the algorithm's robustness, its parameters were not fine-tuned for the different problems. The test problems were developed to represent a wide variety of realistic problems of various sizes and characteristics.

The algorithm proved to be robust to all the test problems because it was able to handle all of their different characteristics. The genetic algorithm found a solution as good as a gradient technique's solution on a benchmark test problem taken from literature, better solutions than a simulated annealing algorithm's solutions on 85% of the 20 test problems, and significantly better solutions than simulated annealing on 45% of the problems. The genetic algorithm out-performed simulated annealing by a larger margin on large problems, but required more replications on most of the problems. From the results, this genetic algorithm was shown to be a feasible and promising candidate for robust simulation optimization.