# The Optimization of Simulation Models by Genetic Algorithms: A Comparative Study

by

James M. Yunker

Dissertation submitted to the Faculty of the

Virginia Polytechnic Institute and State University

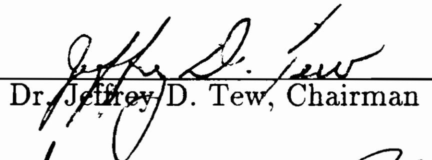
in partial fulfillment of the requirements for the degree of

Doctor of Philosophy

in

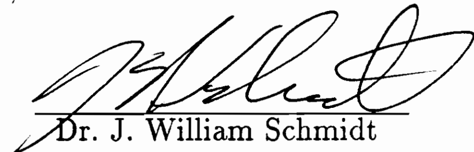
Industrial and System Engineering

Approved

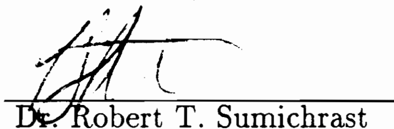


Dr. Jeffrey D. Tew, Chairman

Dr. Osama K. Eyada



Dr. J. William Schmidt



Dr. Robert T. Sumichrast

Dr. Michael P. Deisenroth

December 1993

# The Optimization of Simulation Models by Genetic Algorithms: A Comparative Study

by

James M. Yunker

Dr. Jeffrey Tew, Chairman

Industrial and Systems Engineering

(ABSTRACT)

This dissertation is a comparative study of simulation optimization methods. We compare a new technique, genetic search, to two old techniques: the pattern search and the response surface methodology search. The pattern search uses the Hooke and Jeeves algorithm and the response surface method search uses the code of Dennis Smith. The research compares these three algorithms for accuracy and stability.

In accuracy we look at how close the algorithm comes to the optimum. The optimum having been previously determined from exhaustive testing. We evaluate stability by using the variance of the response function as determined from 50 searches.

The test-bed consists of three simulation models. We took the three simulation models from textbooks and modified them to make them optimization models if that was

required. The first model consists of a big S, little s inventory system with two decision variables: big S and little s. The response is the monthly cost of operating the inventory system. The second model was a university time-sharing computer system with two decision variables: quantum, the amount of time that the computer spends on a job before sending it back to the queue and overhead, that is the time that its takes to execute this routing operation. The response was the cost of operating the system determined from a cost function. The third model was a job-shop with five decision variables: the number of machines at each of the five work stations. The response was the cost of operating the job-shop again determined from a cost function. The decision variables were integer for the inventory system and job-shop, and were real for the computer system.

# Acknowledgements

I would like to thank my graduate committee Chairman, Dr. Jeffrey Tew, for his patience, support, encouragement and desire to produce the best possible dissertation throughout my graduate studies.

I also would like to thank my committee Dr. Osama K. Eyada, Dr. J. William Schmidt, Dr. Michael Deisenroth and Dr. Robert T. Sumichrast for their comments, consideration and help throughout the research and preparation of this dissertation.

I would like to finally thank my mother for her support and help during the effort to create this document. This dissertation is dedicated to the memory of my father.

# Table of Contents

**1. Introduction** ........ 1

1.1. Definition of Simulation ........ 2

1.2. Research Objectives ........ 4

1.3. Organization of Dissertation ........ 6

**2. Literature Review** ........ 7

2.1. Nonderivative Based Algorithms ........ 7

2.1.1. The Pattern Search Method of Hook and Jeeves ........................................ 8

2.2. Derivative Based Algorithms ........................................ 15

2.2.1. Response Surface Methods ........................................ 17

2.3. Shortcomings of Past Approaches ........................................ 22

2.4. Analogy with Artificial Intelligence ........................................ 22

2.4.1. Optimization by Simulated Annealing ........................................ 22

2.4.2. Discussion of Optimization by Simulated Annealing ........................................ 23

2.5. Evolutionary Search ........................................ 25

2.5.1. Genetic Algorithms ........................................ 27

2.5.2. Optimization Using Genetic Algorithms ........................................ 29

2.5.3. Schema ........................................ 30

2.6. Reproduction ........................................ 33

2.6.1. Reproduction Example ........ 35

2.6.2. Reproduction Theory ........ 36

2.7. Crossover ........ 38

2.7.1. Crossover Example ........ 40

2.7.2. Crossover Theory ........ 41

2.8. Mutation ........ 42

2.8.1. Mutation Example ........ 43

2.8.2. Mutation Theory ........ 43

2.9. K-Armed bandit Problem ........ 44

2.10. Alternative Search Methods ........ 47

2.11. Scaling ........ 47

2.12. Discussion of Genetic Coding ........ 48

2.13. Application of Genetic Algorithms ........ 51

2.13.1. Parameter Values in a Genetic Algorithm ..... 52

2.13.2. Genetic Drift ..... 54

**3. Examples** ..... 57

3.1. Example 1 - Inventory System ..... 57

3.2. Example 2 - University Time-Shared Computer System ..... 59

3.3. Example 3 - Job-Shop Model ..... 61

3.4. Example Selection Criteria ..... 62

3.5. Use of Genetic Algorithms in Simulation Optimization ..... 63

3.6. Changing the Domain of Solutions ..... 65

**4. Numerical Results** ..... 67

4.1. Random Number Generator ..... 67

4.2. Experimental Setup ..... 69

4.3. Example 1 - Results ..... 79

4.3.1. Example 1 - Pattern Search ..... 80

4.3.2. Example 1 - Response Surface Search ..... 80

4.3.3. Example 1 - Genetic Search ..... 81

4.4. Example 2 - Results ..... 81

4.4.1. Example 2 - Pattern Search ..... 85

4.4.2. Example 2 - Response Surface Search ..... 85

4.4.3. Example 2 - Genetic Search ..... 86

4.5. Example 3 - Results ..... 87

4.5.1. Example 3 - Pattern Search ..... 87

4.5.2. Example 3 - Response Surface Search ..... 91

4.5.3. Example 3 - Genetic Search ..... 91

**5. Discussion of Results** ..... 96

5.1. Inventory System ..... 96

5.2. University Computer Time-Sharing System ..... 98

5.3. Job-Shop ..... 100

5.4. Statistical Tests ..... 102

5.5. Runs Data ..... 106

5.6. Summary ..... 107

**6. Future Research** ..... 109

**References** ..... 110

**Appendix 1** Figures ..... 117

**Appendix 2** Source Inventory System ..... 165

**Appendix 3** Source Computer System ..... 184

**Appendix 4** Source Job-Shop ..... 223

**Appendix 5** Source Genetic Search Inventory .................................................. 269

**Appendix 6** Source Genetic Search Computer System........................................ 323

**Appendix 7** Source Genetic Search Job-Shop ................................................... 399

# Table of Contents – Figures

**Figure 1** : Initial Points for Inventory System-Pattern Search .................................. 117

**Figure 2** : Final Points for Inventory System-Pattern Search .................................. 118

**Figure 3** : Final Points for Inventory System-Response Surface Search ...................... 119

**Figure 4** : Final Points for Inventory System-Genetic Search .................................. 120

**Figure 5** : Initial Points for Computer System-Pattern Search ................................ 121

**Figure 6** : Final Points for Computer System-Pattern Search .................................. 122

**Figure 7** : Final Points for Computer System-Response Surface Search ...................... 123

**Figure 8** : Final Points for Computer System-Genetic Search .................................. 124

**Figure 9** : Initial Points Job-Shop Pattern Search Machine Groups (1,2) .................. 125

**Figure 10** : Initial Points Job-Shop Pattern Search Machine Groups (1,3) .................. 126

**Figure 11** : Initial Points Job-Shop Pattern Search Machine Groups (1,4) .................. 127

**Figure 12** : Initial Points Job-Shop Pattern Search Machine Groups (1,5) .................. 128

**Figure 13** : Initial Points Job-Shop Pattern Search Machine Groups (2,3) .................. 129

**Figure 14** : Initial Points Job-Shop Pattern Search Machine Groups (2,4) .................. 130

**Figure 15** : Initial Points Job-Shop Pattern Search Machine Groups (2,5) .................. 131

**Figure 16** : Initial Points Job-Shop Pattern Search Machine Groups (3,4) .................. 132

**Figure 17** : Initial Points Job-Shop Pattern Search Machine Groups (3,5) .................. 133

**Figure 18** : Initial Points Job-Shop Pattern Search Machine Groups (4,5) .................. 134

**Figure 19** : Final Points Job-Shop Pattern Search Machine Groups (1,2) .................. 135

**Figure 20** : Final Points Job-Shop Pattern Search Machine Groups (1,3) .................. 136

**Figure 21** : Final Points Job-Shop Pattern Search Machine Groups (1,4) .................. 137

**Figure 22** : Final Points Job-Shop Pattern Search Machine Groups (1,5) .................. 138

**Figure 23** : Final Points Job-Shop Pattern Search Machine Groups (2,3) .................. 139

**Figure 24** : Final Points Job-Shop Pattern Search Machine Groups (2,4) .................. 140

**Figure 25** : Final Points Job-Shop Pattern Search Machine Groups (2,5) .................. 141

**Figure 26** : Final Points Job-Shop Pattern Search Machine Groups (3,4) .................. 142

**Figure 27** : Final Points Job-Shop Pattern Search Machine Groups (3,5) .................. 143

**Figure 28** : Final Points Job-Shop Pattern Search Machine Groups (4,5) .................. 144

**Figure 29** : Final Points Job-Shop Response Surface Search Machine Groups (1,2) .... 145

**Figure 30** : Final Points Job-Shop Response Surface Search Machine Groups (1,3) .... 146

**Figure 31** : Final Points Job-Shop Response Surface Search Machine Groups (1,4) .... 147

**Figure 32** : Final Points Job-Shop Response Surface Search Machine Groups (1,5) .... 148

**Figure 33** : Final Points Job-Shop Response Surface Search Machine Groups (2,3) .... 149

**Figure 34** : Final Points Job-Shop Response Surface Search Machine Groups (2,4) .... 150

**Figure 35** : Final Points Job-Shop Response Surface Search Machine Groups (2,5) .... 151

**Figure 36** : Final Points Job-Shop Response Surface Search Machine Groups (3,4) .... 152

**Figure 37** : Final Points Job-Shop Response Surface Search Machine Groups (3,5) .... 153

**Figure 38** : Final Points Job-Shop Response Surface Search Machine Groups (4,5) .... 154

**Figure 39** : Final Points Job-Shop Genetic Search Machine Groups (1,2) .................. 155

**Figure 40** : Final Points Job-Shop Genetic Search Machine Groups (1,3) .................. 156

**Figure 41** : Final Points Job-Shop Genetic Search Machine Groups (1,4) .................. 157

**Figure 42** : Final Points Job-Shop Genetic Search Machine Groups (1,5) .................. 158

**Figure 43** : Final Points Job-Shop Genetic Search Machine Groups (2,3) .................. 159

**Figure 44** : Final Points Job-Shop Genetic Search Machine Groups (2,4) .................. 160

**Figure 45** : Final Points Job-Shop Genetic Search Machine Groups (2,5) .................. 161

**Figure 46** : Final Points Job-Shop Genetic Search Machine Groups (3,4) .................. 162

**Figure 47** : Final Points Job-Shop Genetic Search Machine Groups (3,5) .................. 163

**Figure 48** : Final Points Job-Shop Genetic Search Machine Groups (4,5) .................. 164

# Table of Contents-Tables

**Table 1** : Inventory System Pattern Search Calculated Values ..................................... 82

**Table 2** : Inventory System Response Surface Search Calculated Values ....................... 83

**Table 3** : Inventory System Genetic Search Calculated Values ..................................... 84

**Table 4** : Computer System Pattern Search Calculated Values ..................................... 88

**Table 5** : Computer System Response Surface Search Calculated Values ...................... 89

**Table 6** : Computer System Genetic Search Calculated Values ..................................... 90

**Table 7** : Job-Shop Pattern Search Calculated Values ................................................. 93

**Table 8** : Job-Shop Response Surface Search Calculated Values .................................. 94

**Table 9** : Job-Shop Genetic Search Calculated Values ................................................ 95

**Table 10** : Inventory System Comparison of Pattern, Response Surface and Genetic Search ........................................................................................................ 97

**Table 11** : Computer System Comparison of Pattern, Response Surface and Genetic Search ........................................................................................................ 99

**Table 12** : Job-Shop Comparison of Pattern Search and Response Surface Search Methods to Genetic Search Method ........................................................................ 101

**Table 13** : Statistical Tests for the Pattern Search and Response Surface Search Methods Compared to Genetic Search Method ........................................................ 103

**Table 14** : Ratio of the Confidence Interval Half-Lengths for Pattern and Response Surface Search Methods to Genetic Search Method .................................................. 105

# Chapter 1. Introduction

Suri (1985) divides the field of modeling into two areas: (a) generative techniques such as linear programming, and (b) evaluative techniques such as simulation. A generative technique will show the indicated optimum when given the input of parameters and constraints, but its application requires certain (sometimes unrealistic) assumptions. With generative techniques the analyst describes a problem, usually employing some coding scheme particular to that problem. The solution is a set of recommended operating parameters for the system that is now optimized. Such techniques remove the analyst from the decision making process. They are not easy to play with to "get a feel" for the system. They suffer from being a "black box" (Suri 1985, p. 15) meaning that we can use them and not be concerned with what is happening internally in the logic and rationale of the code. Simulation is quite different. It is an evaluative technique. It only tells us what the outcome of an operation would be given that certain variables (or factors) are put into the model. We can get a feel for the model's performance by changing these factors and observing the outcome.

The problem is how to take an evaluative technique like simulation and use it to optimize a system. Meketon (1987, p. 66) describes simulation optimization as more of an art than a science. Analysts have attempted to optimize simulation

from several approaches such as a pattern search (Hooke and Jeeves 1961, Pegden and Gately 1980) to response surface methodology (Smith 1978, Daughety and Turnquist 1978), but each had limited success. Previous attempts at simulation optimization have been derivative based or non-derivative based. A derivative based methodology uses some numerical technique to obtain the derivative (usually referred to as the gradient since it involves more than one variable) of the response surface. Using the gradient information, the algorithm searches for an approximated optimum using calculus techniques. Non-derivative based techniques usually use some pattern search heuristic. The analyst uses non-derivative methods when obtaining a derivative is not practical. Both techniques fail in one respect in that they do not use the mathematical information about the entire system. They search a given area and continue when certain criteria are satisfied. They also lack parallelism (the characteristic of searching several directions simultaneously) in their search so they rarely find the indicated optimum.

## 1.1. Definition of Simulation

In this section we formally define the term discrete-event simulation. We use simulation to analyze the stochastic behavior of a system that is not amenable to analytical solutions. A system is a collection of components with each component having distinct characteristics (Law and Kelton 1991, p. 3). An industrial plant with its machines and workers is a system as is a national economy with its producers and consumers. Attributes of the system are the characteristics of the components; they can be either numerical or logical. A numerical attribute would be the interarrival times of customers to a server; whereas a logical attribute would

be the queue discipline, for example first come first served (FCFS). Among the components, relationships exist and the components interact. The environment that the system is in has input to it, and the processes operating in this system transform this input to output (Kleijnen 1974, p. 2).

A simulation model that involves random numbers, but no time evolution is a Monte Carlo simulation. If the behavior of the system over time is of interest, we say that the accompanying model when implemented on a computer is a computer simulation model. Two key components compose simulation models: (a) probabilistic behavior and (b) time evolution.

Of particular interest to us is discrete-event simulation. Discrete-event simulation is a stochastic, time-dependent computer modeling methodology in which events (phenomena that change the state of the system), occur at specific, discrete points in time. Thus, discrete-event simulation is a method of analyzing the dynamic behavior of a stochastic system over time. A bank in which customers arrive randomly, the customers are served by tellers and then the customers leave (i.e., the system changes at discrete points in time) is an example of a system that can be modeled by discrete event simulation. The three discrete events are when a customer arrives, begins service, and completes service and departs. The randomness determines his interarrival times and his service time. The interval between these specific points in time is usually not constant (Banks and Carson 1984, p. 7); a random process determines the intervals.

There are several software packages that are designed specifically to aid in the

construction of discrete event models. Two of the oldest packages are GPSS (General Purpose Simulation Systems) available from several suppliers and SIMSCRIPT available from CACI, Inc. Additional information on GPSS is available in Schriber (1974); additional information on SIMSCRIPT is available in Russell (1976, 1981), Kiviat, Villaneuva, and Markowitz (1973) and the *SIMSCRIPT II.5 Reference Handbook.* The two most popular simulation languages are SLAM (Simulation Language for Alternative Modeling) and SIMAN. Additional information on SLAM is available from Pritsker (1986); and additional information on SIMAN is available from Pegden (1985). Other languages are also available such as SIMULA. Additional information on SIMULA is available from Birtwistle, Dalhl, Myhrhaug and Nygaard (1973).

Naylor, Balintfy, Burdick, and Chu (1967, p. 3) define discrete-event simulation as experimenting with a stochastic model over time. Pritsker (1986, p. 6) defines computer simulation as a process of designing a mathematical and logical model of a real system and experimenting with this model on a computer. Pritsker considers a computer simulation model to be a model building process as well as the experimental testing of that model.

## 1.2. Research Objectives

The research objectives of this dissertation are to compare the search techniques of genetic algorithms with those of pattern search and response surface methodology. These last two techniques are traditional search techniques that have been used in simulation optimization. Analysts have not previously used genetic algorithms in

simulation optimization.

The criterion for comparison of the search algorithms is the proximity to the approximated optimum. If the algorithm being tested does not find the this optimum then which of the three algorithms comes closest to it. The test-bed is a set of three example problems common in simulation with each approximated optimum determined from exhaustive trial searches.

The criterion for comparison of stability is the variance of the response. We evaluate stability by using the variance of the response function as determined from 50 searches; the lower the variance of the response the better the stability.

The amount of CPU time that each algorithm uses is not a criterion. The attainment of sample points in a simulation run is usually a time-consuming process and a genetic search is no exception. It will usually use more CPU time than a pattern or a response surface search. However, a genetic search usually attains something for its additional simulation time – a more desirable indicated optimum.

We determine the calculated optimum for each example from exhaustive trial runs. In each case the calculated optimum is a minimization of an output variable. It is true that based on this method we can never be sure that we have found the true optimum, but there is enough evidence from the simulation and experimental design literature to support the use of trials to find the approximated optimum.

## 1.3. Organization of Dissertation

We now give the organization of the rest of this dissertation. The second chapter is a literature review of previous simulation optimization techniques and it discusses what a genetic search is and how to use it to optimize a simulation model. We first generally discuss the technique and specifically discuss an algorithm associated with the search technique. In each case we offer a general discussion and for the specific algorithm we give a theoretical discussion as well as a practical outline of each step of the algorithm. The third chapter describes in detail the examples that will be used to test all the optimization algorithms. The fourth chapter gives the numerical results of each example and its optimization by each of the three algorithms. The fifth chapter gives the summary and conclusions, and the sixth chapter outlines proposed future work. The reference section follows this and finally seven different appendices. Appendix 1 contains all 48 figures of the experimental computer runs, Appendices 2-4 are the simulation examples in the original coded form, and Appendices 5-7 are the three examples coded simulation models interfaced with the genetic search executive program.

# Chapter 2. Literature Review

In this section we introduce and discuss the various search methodologies used in the simulation optimization programs in this dissertation. They fall into three areas: 1. nonderivative methods; 2. derivative methods; and 3. evolutionary search. In each of these areas we discuss the methodologies, and then specify exactly which technique (one from each of the three general search areas) we chose to use in the example programs.

## 2.1. Nonderivative Based Search

There have historically been two basic approaches to the development techniques of simulation optimization. The first has been to employ search techniques that do not need derivative information. Examples of general optimization techniques that do not require derivative information are the Hooke and Jeeves pattern search (1961), Rosenbrock's (1960) method of rotating coordinates and the simplex method of Nelder and Mead (1965). The methods are all unique, but each has in common the traits of a derivative free search. One obvious quality of this type search is its simplicity in application. This simplicity can also work against it. The methods are direct and simple, but not very perceptive to changes in the evolution of the simulation. In that respect they are dumb. The simulation continues when

certain criteria are met, the simulation stops when these criteria are not met. This is an all or nothing type search.

The obtaining of simulation data points is costly and we want to get as much information from each one as possible. The pattern search does not use as much information as the derivative based search. It does have the quality in that it is easy to interface to an existing simulation program.

The nonderivative method chosen to be used in this dissertation is the pattern search method of Hooke and Jeeves (1961). There are several reasons for this: one is that there exists literature on previous attempts to use it to optimize simulation (Pegden and Gately 1980). The other methods have no such history. The second is that it requires a minimum amount of simulation points to execute a step. Such methods as the simplex method of Nelder and Mead (1965) or the rotating coordinates method of Rosenbrock (1960) require more function evaluations to execute a single step.

### 2.1.1. The Pattern Search Method of Hooke and Jeeves

The basic idea behind this method relies on the optimistic assumption that if any set of moves used in searching for an approximated optimum are successful then they are worth repeating. The method starts with small steps then, if these are successful, the step size increases. Alternatively, when a sequence of steps fails to improve the objective function, this indicates that shorter steps are in order so we may not overlook any promising direction.

We take this analysis from Wilde and Beightler (1967). Consider a function with independent variables $x_i$ ($i$=1, 2, ..., $N$) that we wish to maximize. We establish the pattern during the search itself by stepping $\delta_i$ for variable $x_i$ and checking whether our function has improved – either it is greater or lesser depending on whether we are maximizing or minimizing a function. The new step, designated by $t_{ii}$ ($i$=1, 2, ..., $N$), is a temporary location upon which we test to determine if we have a successful optimization move.

To begin the search, let us choose a base point $b_1$ and a step size $\delta_i$. Let $\delta$ be a vector with its $i$th component being $\delta_i$, and all other components being zero. We measure all criterion at $b_1$, e.g., in simulation optimization we run the simulation the specified number of replications with $b_1$'s decision variables input; and repeat the process again at $b_1 + \delta_1$, with decision variables $b_1 + \delta_1$ for input. If at this new point the function is greater than at $b_1$, then we make $b_1 + \delta_1$ the temporary head $t_{11}$ and consider this a successful step. If at $b_1 + \delta_1$, the function is less than at $b_1$, then we try $b_1 - \delta_1$; and if the function is greater than at $b_1$, then we make $b_1 - \delta_1$ the temporary head $t_{11}$ and consider this a successful step. If neither point gives a greater function than $b_1$, then we keep the optimum for that independent variable at $b_1$ and have no successful steps. Stated formally:

$$t_{11} = \begin{cases} b_1 + \delta_1 & \text{if } y(b_1 + \delta_1) > y(b_1) \\ b_1 - \delta_1 & \text{if } y(b_1 - \delta_1) > y(b_1) > y(b_1 + \delta_1) \\ b_1 & \text{if } y(b_1) > \max[\, y(b_1 + \delta_1), y(b_1 - \delta_1)], \end{cases} \qquad (2.1\text{abc})$$

where $t_{11}$ with its double subscript shows that we are working on the first pattern

and we have perturbed the first variable $x_1$. Note that $t_{10}$ is $b_1$. The next perturbation is for $x_2$ and is the same as (2.1abc) except the perturbation begins around $t_{11}$ not $b_1$. Now putting this into a general formula

$$t_{1j} = \begin{cases} t_{1,\ j-1} + \delta_j & \text{if } y(t_{1,\ j-1} + \delta_j) > y(t_{1,\ j-1}) \\ t_{1,\ j-1} - \delta_j & \text{if } y(t_{1,\ j-1} - \delta_j) > y(t_{1,\ j-1}) > y(t_{1,\ j-1} + \delta_j) \\ t_{1,\ j-1} & \text{if } y(t_{1,\ j-1}) > \max[\, y(t_{1,\ j-1} + \delta_j), y(t_{1,\ j-1} - \delta_j)], \end{cases} \quad (2.2\text{abc})$$

which explains how the $j$th temporary head is obtained from the preceding one $t_{1,\ j-1}$ while trying to establish the first pattern.

When we have had at least one successful step while establishing the first pattern, then the last temporary head point $t_{1_N}$ is the second base point $b_2$. The original base point $b_1$ and the newly determined based point $b_2$ establish the first pattern by the equation

$$t_{20} = b_1 + 2(b_2 - b_1). \quad (2.3)$$

Where $t_{20}$ shows that we are now starting on the second pattern, but have not yet perturbed the variables. Note that repeated successes in this direction causes the pattern to grow as shown by the second term on the right hand side of (2.3). Put more simply

$$b_1 - b_2 = 2(t_{20} - b_2). \quad (2.4)$$

We move all points $2*(b_2 - b_1)$ and again perturb each variable as before, but this

time the perturbations or step sizes are larger after being multiplied by the expansion factor. If the expansion factor is two, then the new steps in the next pattern move are twice as large as before. When we have completed that sequence (and we have had at least one successful step) then

$$t_{30} = b_2 + 2(b_3 - b_2), \tag{2.5}$$

this sequence continues as long as we have at least one successful perturbation out of the $N$ independent variable perturbations.

When there are no successful moves out of the $N$ moves of the independent variables, we then switch to smaller step sizes by dividing the current steps by the expansion factor. This indicates that we are either crossing a ridge or approaching the optimum. In either case we use smaller step sizes for a more careful exploration of the area. However, when a sequence of variable perturbations is carried out and none were successful, and we are at the minimum step size, the search then concludes. The calculated optimum of the function $x_i$ ($i$=1, 2, ..., $N$) is the last base point attained. For example if we are on perturbation for $t_{40}$ and no moves are successful the base point $b_{40}$ is the optimum of the function.

Let us sum up this procedure by formally outlining the steps:

1. Select a function with independent variables $x_i$ ($i$=1, 2, ..., $N$) and choose a step size $\delta_1$ and a base point $b_1$.
2. Run the simulation the required number of replications with $b_1$'s decision

variables input.

3. Step $\delta_1$ and run it again with $b_1+\delta_1$ decision variables input, then record whether the function is greater or lesser than the function evaluated in step 2. at these new points depending on whether we are maximizing of minimizing the function whichever the case may be. If we have a success, then we record it as a success and begin with the next variable in the function. If it is a failure then step $b_1-\delta_1$ and evaluate the new point and determine if the function is greater or lesser than the function evaluated in step 2, just as we did in the previous step. If it is a success then we record it as a success and if it is a failure then we begin with the next variable in the function.

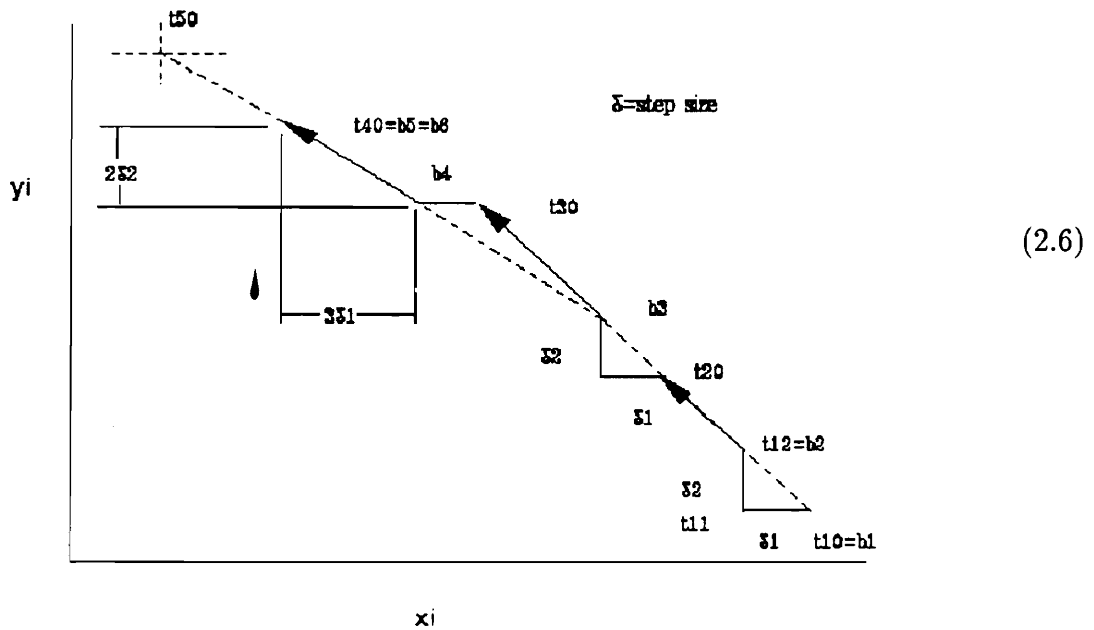
4. Continue with the function variables and using the general formula as in (2.2abc). When completing a sequence and we have at least one successful move then we have established the next base point $b_2$. Calculate the first pattern by $t_{20}$ formula (2.3). This will give $t_{20}$ and using formula (2.2abc) we can begin the sequence again.

5. When we have tried a complete sequence of variables $x_i$ ($i$=1, 2, ..., $N$) and get no successful moves, then decrease the step size by the expansion factor, e.g., if the expansion factor was two then the new step size is one half the old step size. If we are at the minimum step size and have no successful moves in a sequence then we conclude the search.

6. The calculated optimum of the function is the function evaluated at the last base

point.

In conclusion consider (2.6) a diagram of the steps we have just discussed.




(2.6)

The first two sets of solid lines at right angles to each other are steps for variables $x_1$ and $x_2$, for this case $N = 2$. In the first set clearly equation (2.1b) holds when calculating $t_{11}$ and similarly when calculating $t_{21}$. Also notice in the third test (attaining $t_{40}$) there is no improvement in stepping in the $x_2$ direction only in stepping in $x_1$'s direction, hence we only move horizontally. We do not move horizontally and then vertically as in the first two sets. The pattern search veers to the left. The broken line shows the path we delineate and the solid line when either horizontal or vertical is the test steps or the exploration stage, while the solid line

with an arrowhead is a pattern move.

It should be obvious that the Hooke-Jeeves pattern search is a heuristic search that anchors itself at an initial base point $b_1$ (which we select) and then ventures out to a temporary location on recognizance trying to attain a successful move. Wilde and Beightler (1967) use the pedagogical device of an arrow in which the end is the base point $b_1$ and the head is the temporary point $t_{1N}$. When one of these steps is successful we have established a pattern and calculate a new starting point by the general formula:

$$t_{v0} = b_{v-1} + 2(b_v - b_{v-1}). \tag{2.7}$$

Pegden and Gately (1980, p. 24) have modified the above method. When a set of $N$ pattern moves fails to find at least one improvement, the algorithm sets the step sizes to a user-input minimum. We do not reduce them by a previously selected expansion factor in graduated reductions. We do not go from the present step size to the minimum, slowly, but by a one time jump to the minimum step size.

We input a minimum and possibly a maximum value of each of the $N$ variables. The variables can be integer or real. We must use care when selecting these values. If we are simulating physical systems, for example a job-shop, and have a variable for the number of machines in the shop, the minimum value of this variable must always be greater than zero.

Pegden and Gately (1980 p. 20) use two other modifications to the Hooke and

Jeeves search technique, both owing to the difficulty of getting simulation points. In the first modification the algorithm saves old points and their values; and if the pattern search returns to a previously evaluated point (a finite possibility) it uses that previous point and makes no simulation run. This will create some computing overhead, but it is worth incorporating into the simulation program because of the potential number of simulation runs it saves. The second modification also requires a minimal cost in overhead, but it is quite effective in simulation execution: in a sequence of successful variable perturbations we retain the direction of the successful search and use it in the subsequent search.

The Hooke and Jeeves pattern search works well if it finds a ridge on the surface and follows that ridge up the response surface if we are maximizing the model or down if we are minimizing the model. Hooke and Jeeves found that in their search technique the computations increase as the first power of the dimensionality of the variables, whereas in classical minimization techniques the computations increase as the cube of the dimensionality. The reasoning is clear: the Hooke and Jeeves search method is searching for a ridge in order for it to keep searching. A ridge in a multidimensional space is still only one dimensional. By following a ridge we essentially reduce the effective dimensionality of the problem.

## 2.2. Derivative Based Algorithms

The use of derivative information employs more mathematical information than a nonderivative search. It can approximate an optimum by employing traditional mathematical techniques. Each step in this type of search has two purposes: (1)

attain an improved value of the objective function; and (2) give information useful for locating future points that have more desirable values.

One derivative based search method is perturbation theory that uses gradient estimation to perform sensitivity analysis (Ho and Cassandras 1981). It is a perturbation of the sample path of the simulation model. It does not work very well when applied to a rigorous problem, and many investigators question its usefulness as a general simulation optimization tool (Heidleberger, Cao, Zanzanis, and Suri 1988). Perturbation theory seems to have a theoretically sound basis when applied to a simple problem such the $M/M/1$ queue, but when applied to a complex problem (the kind that we would turn to simulation to solve) its accuracy falters.

Schruben (1986) has been more successful in using the gradient approach, but we can use his technique only for screening out unimportant parameters. Schruben achieves this by varying the parameters throughout each run at a given frequency. We can relate input and output spectra with the peaks signaling the important parameters; we drop the unimportant parameters and worry about only the important ones that must have their gradients estimated some other way.

The derivative based approximation is more theoretically rigorous than searches of the nonderivative type. Thus if certain criteria are satisfied then we can be confident that we have obtained a calculated optimum of the system. The amount of research in derivative optimization areas is extensive and thorough. It is definitely not a heuristic approach as many of the nonderivative information approaches are. The ideas originated in theoretical mathematics and statistics.

Such a method that applies these ideas is response surface methodology (RSM).

### 2.2.1. Response Surface Methods

In RSM we are working with a function of several variables; but we do not know the exact form of the function. In more formal terms:

$$y = f(x_1, x_2, \ldots, x_N), \tag{2.8}$$

which is in a $N + 1$ dimensional space. The difference between the number of variables and the number of independent equations that describe the hyperplane we call the degrees of freedom. In (2.8) we have $N + 1$ variables ($x_1$, $x_2$ ..., $x_N$, and $y$) linked by one equation (2.8), thus we have $N$ degrees of freedom. The number of degrees of freedom is the same as the number of dimensions in which the surface lies.

We must use polynomials and their approximations here. The polynomials must be of low order to determine accurately their roots. A polynomial's first derivative is itself a polynomial and only a low order polynomial has roots that can easily be determined from a formula. In all of our surfaces in which we seek to optimize the function, we will use low order polynomials.

In a RSM study we are trying to approximate a surface made up of $N$ independent variables. We want to find the approximated optimum of some objective function that depends on $N$ independent variables ($x_1$, $x_2$, ..., $x_N$). The objective function is

unknown, but experiment determines the value of $y$ for any set of values ($x_1$, $x_2$, ..., $x_N$). Thus the point in the ($x_1$, $x_2$, ..., $x_N$) hyperplane corresponds to a possible experiment, and the point above it on the response surface represents the experimental outcome. The experimental points are confined to a bounded portion of the ($x_1$, $x_2$, ..., $x_N$) hyperplane we call the experimental region. Each experiment gives the elevation of the response surface above a new point in the experimental region. We want to climb as high as possible on the response surface, using historical information to guide the search for the summit. This information derives from the theoretical mathematics of optimization.

In simulation optimization we are trying to find a proxy for the true response values – a response surface. The analyst uses the response instead of the simulation runs in optimization studies; it is the objective function that is analytically unobtainable and hence must be estimated (albeit a simple one) from simulation data. A response surface of a system is the response of the system as a function of the factors; if there are only 2 or 3 factors it can be plotted graphically.

If we assume that the true response surface is expressed as a Taylor series it can be written as

$$y = \beta_0 + \sum_{i=1}^{p} \beta_i x_i + \sum_{i=1}^{p} \sum_{j=1}^{p} \beta_{ij} x_i x_j + \text{(higher order terms)}, \tag{2.9}$$

where $y$ is the independent variable, $x_i$'s are the dependent variables, $\beta_i$'s are the

coefficients and $p$ is the number of independent variables. If first order or linear effects dominate an estimate of the response at $(x_1, x_2, \ldots, x_N)$ is

$$\hat{y} = b_0 + \sum_{i=1}^{p} b_i x_i, \tag{2.10}$$

where $b_i$, obtained by least squares, is an estimate of $\beta_i$.

It is impractical to use a complete factorial on problems with a very large number of factors; we use instead a fractional factorial. In such a design we select a subset of points from the full factorial. This reduces the simulation run length, but does so at a cost.

A good design to use is the $2^{k-p}$ fractional factorial. The estimates from this design are uncorrelated and have a minimum variance. Of course there are other designs that provide similar qualities, but the $2^{k-p}$ fractional factorial is capable of being directly transformed to a second order design with the addition of specific points. The second order design we use to estimate quadratic effects (the $\beta_{ij}$'s) of the controllable factors.

A $2^{k-p}$ fractional factorial consists of $2^{k-p}$ points selected in a specific manner from $2^k$ possible coded points $(\pm 1, \ldots, \pm 1)$ which form a full $2^k$ factorial. Consider a $2^{k-p}$ fractional factorial as a balanced subset of points on the vertices of a $k$-dimensional hypercube. The result is that the first order effects (the $b_i$'s) are

confounded, or mixed up with the estimates of the higher effects. This is acceptable if the first order effects are dominant over higher order effects.

Confounding in the above context means that in a fractional factorial design we may end up with the same algebraic expression for several different effects. In a $2^{4-1}$ fractional factorial it may occur that the formulas for the first order effect $e_1$ and the three way interaction $e_{234}$, are the same so we say that the main effect of factor 1 (main effects are considered as first order effects) is confounded with the three way interaction effect between factors 2, 3 and 4. The common formula for $e_1$ and $e_{234}$ is an unbiased estimator for $\mathrm{E}(e_1) + \mathrm{E}(e_{234})$. If $\mathrm{E}(e_{234})$ is small compared to $\mathrm{E}(e_1)$, then $e_1$ is almost an unbiased estimator of $\mathrm{E}(e_1)$. It frequently happens that higher order interactions are small when compared to main effects (Law and Kelton 1991, p. 671).

One way to insure that we are getting the confounding we want is to employ resolution designs for a particular $2^{k-p}$ fractional factorial. We do not confound two effects if the sum of their order is strictly less than the design's resolution. For example, in a Resolution IV design first order and second order effects are not confounded since $1 + 2 < 4$, while second order effects are confounded since $2 + 2 = 4$.

If $y_j$ denotes the average observed responses of the $m$ iterations of a simulation run corresponding to points $x_j$, then we obtain the estimate $b_i$ of $\beta_i$ from the least squares equation:

$$b = (X'X)^{-1}Xy. \tag{2.11}$$

The estimates of $(b_1, b_2, \ldots, b_m)$ provide an indication of the relative importance of each of the controllable factors, and predict the direction that the maximum response lies. We see this from the fact that it is desired to find, on any hypersphere of a given radius, $R$, the points $(0, 0, \ldots, 0,)$, which are the values of $x$ which maximize:

$$\hat{y} = b_0 + \sum_{i=1}^{p} b_i x_i. \tag{2.12}$$

Where $\hat{y}$ is to be maximized subject to the restriction $\sum x_i^2 = R^2$. From Myers (1971, p. 89) selecting $x_i$ proportional to $b_i$ provides the maximum predicted response on the hypersphere. This direction of maximum predicted response $(b_1, b_2, \ldots, b_m)$ is the steepest ascent path.

We use the first order design in the initial phase of a sequential response surface analysis. It is of course linear and we use it when we are far from the approximated optimum. When the first order design fails to provide a successful search along the steepest ascent path, we augment the existing fractional factorial with additional points to obtain a second order design that allows estimation of the quadratic effects. At this stage in the optimum seeking process, a second order approximation is appropriate:

$$\hat{y} = b_0 + \sum_{i=1}^{p} b_i x_i + \sum_{i=1}^{p} \sum_{j=1}^{p} b_{ij}\, x_i x_j. \tag{2.13}$$

We use a second order design when we are close to the approximated optimum and the first order design is no longer adequate. It takes into account the curvature of the response surface that is critical when we are approaching the approximated optimum.

## 2.3. Shortcomings of Past Approaches

The performance measures for a manufacturing system such as WIP and machine utilization would have trouble being optimized by traditional simulation optimization techniques such as the pattern search and response surface methods. We must turn to a different methodology when we want to optimize something complicated. In this dissertation we discuss the problem of implementing an artificial intelligence methodology in simulation optimization.

## 2.4. Analogy with Artificial Intelligence

Nature is a robust optimizer. It can optimize: either minimize or maximize; and we can find acceptable solutions to intractable optimization problems from analyzing nature's optimization mechanisms. Two concepts that have the most promise are: simulated annealing (SA) and genetic algorithms (GA).

### 2.4.1. Optimization by Simulated Annealing

Metropolis, Rosenbluth, Rosenbluth, Teller and Teller (1953) derived the original idea for SA. They were searching for a method to calculate the equation of state of

a substance on "fast computing machines". They used the method of statistical mechanics, which applies the idea of statistics to molecular phenomenon for determining the aggregate qualities of a substance. There are about $10^{23}$ atoms per cubic centimeter of gas, hence we can only study the most probable behavior of a system in equilibrium at a given temperature. In statistical mechanics there are many methods to obtain the properties of a macroscopic system from microscopic averages.

Kirkpatrick, Gelatt and Vecchi (1983) applied these ideas in combinatorial optimization to the design of mainframe computers. This was the first attempt using the Metropolis algorithm to solve an operations research (OR) problem.

Bulgak and Saunders (1988) also used SA to determine the appropriate buffer sizes for asynchronous assembly systems a type of flexible manufacturing system. Their approach was to employ a discrete event simulation model that would be optimized by a SA algorithm. This is the first application of SA to simulation optimization. Manz, Haddock and Mittenthal (1989) followed where they used SA to optimize the profit (defined as total revenues minus total cost) of an automated manufacturing system. In that project they also attempted to determine an appropriate annealing or cooling schedule of the system to avoid the pitfalls previously discussed.

### 2.4.2. Discussion of Optimization by Simulated Annealing

SA is not without drawbacks. It does have a rigorous mathematical proof

guaranteeing that the algorithm will find an approximated optimum, though the proof tells nothing about the time it takes to reach the approximated optimum. The annealing schedule might require a long time to reach a calculated optimum; it may be infinitely long (Goldberg 1990).

SA is also a serial methodology; it searches one point and moves on from there to another point. SA is not easily parallelizable (Mahfoud 1991, p. 2). As previously stated GA's search a population of points. There is no formal proof that GA's converge. There are mathematical arguments that try to demonstrate that a GA will converge, but no formal proof exists (Goldberg 1990).

Manz, Haddock and Mittenthal (1989, p. 390) have applied SA to simulation optimization with promising results. This scheme borrows its basic ideas from statistical mechanics. A metal cools and the electrons align themselves in an optimal pattern for the transfer of energy. In SA, we construct a model of a system and slowly decrease the "temperature" of this theoretical system and until the system assumes a minimal energy structure. The problem is how to map our particular problem to such an optimizing scheme. This is usually the main problem in optimization strategies that copy ideas from nature.

GA's are optimizers that use the ideas of evolution to optimize a system that is too intractable for traditional optimization techniques. We know that organisms optimize themselves (through generational changes using these three operators) to adapt to their environment. So by taking that simple idea we can optimize artificial systems.

GA's do not have the shortcomings of traditional optimization techniques (Goldberg 1989, p. 7). They can search a response surface with many local optima and find the approximate global optimum with a high probability. There is still no guarantee that they will find the true optimum, however.

The purpose of simulation optimization is to find a calculated optimum of the system under study. GA's are ideal for this purpose. They are quite different from traditional simulation optimization techniques in that they are not dependent on a good mathematical model of the response surface. They will find an indicated optimum. While it may not be the "global optimum", research has shown that it will usually be a better calculated optimum than that traditionally found (Goldberg 1989, p. 5-6).

GA's differ from traditional optimization procedures in four ways (Goldberg 1989, p. 7):

1. GA's work with a coding of the parameter set, not the parameters themselves;
2. GA's search a population of points, not a single point;
3. GA's use payoff (objective function) information, not derivatives or other auxiliary knowledge;
4. GA's use probabilistic transition rules, not deterministic rules.

(2.14)

## 2.5. Evolutionary Search

This title of this section might also be adaptive search or possibly genetic search.

The idea is to mimic the strategies that biologists have learned from evolution and apply them to optimization problems. Clearly an organism's adaptation to its environment is an example of efficient optimization. However, how do we take these ideas and apply them to an industrial optimization problem?

Biologists carried out the first computer simulations of living systems in trying to understand the operation of the living cell (Barricelli 1957, 1962, Fraser 1960). It is Fraser's (1960) writings that strongly suggest these methods of optimizing a living cell may be used outside a natural setting. Holland (1962) attempted this. From this early effort Holland was to elucidate the basic ideas of what we now know as a GA.

Holland was not the only one to attempt to mimic the attributes of life in the solution of optimization problems. The evolutionary operation scheme of Box (1957) was an attempt to do this. Box designed his technique to allow the less technically oriented worker in an industrial plant to optimize its production. It involves the selection of an operating point, and the creation of a hypercube about that point. We then visit the additional points on the hypercube's vertex. If one point improves the plant's operation, we construct a new hypercube and the process continues until no improved points are found. This stretched the imitating life analogy to the point that it was almost useless. By Box's definition almost any optimization scheme was evolutionary. There were several other schemes that used the simplex that worked much better in optimization (Nelder and Mead 1965; Spendley, Hext and Himsworth 1962).

The work that followed Box's was much more specific and much more in line with what we know as a genetic search today (Bledsoe 1961, Bremermann 1962, Friedman 1959). It was here that the idea of a schema or similarity template was introduced as was the idea of populations of possibilities, this will elaborated on later. The ideas of reproduction, and crossover and mutation got their initial start here. The work of Fogel, Owens, and Walsh (1966) then followed, but they made one fundamental error. They tried evolutionary optimization, and they put all of their emphasis on mutation and ignored crossover. This research turned out to be unproductive.

### 2.5.1. Genetic Algorithms

Another phenomenon found in nature is genetic evolution. Nature is an efficient optimizer. Researchers have studied the field and applied its operations to system optimization. They copied the mechanisms of a natural phenomenon and applied them to problems that seemed insolvable by traditional OR methods.

GA's are probabilistic search optimizing algorithms that do not require mathematical knowledge of the response surface of the system that they are optimizing. They borrow the paradigms of genetic evolution in their optimization search, specifically reproduction, crossover and mutation. A simple pseudo-code outline of a GA is:

```
t←0;
initialize P(t); P(t) is the population at time t
evaluate P(t);
while (termination condition not satisfied) do
begin                                                    (2.15)
```

```
        t← t + 1;
        select P(t);
        recombine P(t);
        evaluate P(t);
    end(Grefenstette 1986, p. 123).
```

The first of these operators, reproduction, is the creation of a new generation based on the fittest members of the previous generation. The second operator, crossover, is the creation of a new gene pool from the gene pool of both parents. The genes of the offspring consist of a combination of both parents producing a new set of genes different from either parent. The third operator, mutation, is the random changing of a gene to its opposite number for a binary string. It is the least important of the three operators, but that does not diminish its necessity in the proper operation of a GA. We will discuss these operators more completely below.

For the basic three operators: reproduction, crossover, and mutation to optimize we must have an incentive for each successive generation to improve in some way compared to the previous generation. The Darwin hypothesis: "survival of the fittest" applies here; it helps in understanding this incentive to improve with each generation. An organism must adapt to its environment to survive; if it fails to adapt it will not survive. When an organism can adapt, those traits that allow it to survive pass on to future generations. The next generation's organisms also must survive in their environments and the ones that survive will pass those traits on to the following generation. In surviving it is necessary to adapt competitively along with other organisms in the present generation. That means that in each generation the competition for survival increases and the prevailing organisms are more fit than the previous generations. We achieve increasing fitness using these genetic

operators

## 2.5.2. Optimization Using Genetic Algorithms

In this dissertation we will use only binary strings (representing genes) in GA's; the string consists of the set $V = \{0, 1\}$ (Goldberg 1989, p. 28). Consider, seven-bit string (a chromosome) $X$ that can be symbolized

$$X = x_1 x_2 x_3 x_4 x_5 x_6 x_7, \tag{2.16}$$

and which has one realization

$$X = 0111000. \tag{2.17}$$

Now $x_i$ represents a single binary feature or detector where each feature may take on a value of 0 or 1. In keeping with the genetic analogy, we call $x_i$ a gene.

To have any type of meaningful genetic search requires a population of strings. Consider a population of individual strings $X_j$, $j = 1, 2, \ldots, n$ contained in a population $X(t)$ at time (or generation) $t$. The interest is not in the strings alone, but in their similarity templates or schema. A schema describes a subset of strings that have similarities at certain positions (Holland 1975). The use of schema is to simplify our analysis of GA's.

### 2.5.3. Schema

We now devote a complete section to the concept of schema or similarity template because it is so important in the abstract discussion of genetic algorithms. Without it, it would be almost impossible to discuss why genetic algorithms operate as they do with no information about the surface that they are searching. We pose a rhetorical question: how do genetic algorithms operate only using payoff information and no mathematical information about the response surface?

Consider we have different strings of 1's and 0's. Would all these strings be equal? No, they would not. Consider the string:

$$01101, \tag{2.18}$$

which decodes to 13 and is considerably different from string:

$$01000, \tag{2.19}$$

which decodes to 8. We conclude that having more 1's in the later positions (those that are on the far left) on the strings is preferred to having those locations filled with 0's.

Let us continue and generalize this, and not worry about a specific string, but the general strings. What is it about the most fit strings that make them the most fit? There are two things. The first we have already mentioned in that having 1's in

high positions is a good trait.

The second requires a little more examination. In that as stated above we do have an operator called crossover that cuts two strings at a separate point on each string and then interchanges the pieces to provide two new strings different from the two original strings.

Now back to schema or its plural schemata. If the strings are such:

$$\begin{aligned} A_1 &= 011000 \\ A_2 &= 010001. \end{aligned} \tag{2.20}$$

To disrupt the dominant pattern in $A_1$, we must locate the crossover point at 2 (counting from the left to right). To disrupt the dominant pattern in $A_2$ we must locate the crossover point at 2, 3, 4, or 5.

The dominant pattern in each above example is a schema or a general string with dominant similarities not specific ones. In $A_1$ our interest is in positions 2 and 3, and in $A_2$ our interest is in positions 2 and 6.

A schema is a similarity template that describes a subset of strings with similarities at certain positions. Rewriting the two strings as schema should make things more clear:

$$A_1 \rightarrow H_1 = *1*1** \tag{2.21}$$

$$A_2 \rightarrow H_2 = *1***1.$$

The defining length of $H$ is 2 and the defining length of $H$ is 4, where the defining length is the distance between the first and last specific string position. This will aid in the next discussion. The idea is that a schema with shorter defining length is more fit than a string with a longer defining length because it will more likely survive into the next generation.

Up to now we have considered a schema from the binary alphabet {0, 1}; but we need another symbol to discuss this thoroughly. That is "*" or the "do not care" symbol. It is a metasymbol, which gives an extended alphabet {0, 1, *}. Think of schema as pattern matching: a schema matches a particular string if at each location in the schema a 1 matches a 1 in the string, a 0 matches a 0, or a * matches either. For example *00000 matches two strings {10000, 00000} and *111* matches four strings {01110, 01111, 11110, 11111}. 0*1** matches any of eight strings that are of length 5 and begin with 0 and have a 1 in the third position. The schema concept is merely a tool to discuss similarities among different finite length strings of a finite alphabet.

All schemata are not equal. There are some schemata that are more specific than others. The schema 011*1** is more specific than 0******. The first schema has an order of 4 whereas the latter has an order of 1, with order defined as the number of fixed positions on the string. Schema 1****1* spans more of a string than schema 1*1****. The former schema has a defining length of 5, whereas the latter has a defining length of 2.

## 2.6. Reproduction

Reproduction is the most difficult operation to understand. It is an algorithmic implementation of Darwin's theory of survival of the fittest. We decode each string, calculate its fitness, and collect all the individuals' fitnesses for a population. We know certain rules apply such as the most fit strings tend to survive and move into the next generation. This is a probabilistic rule, not a deterministic rule. They do not automatically move the most fit strings into the next generation.

We now introduce some Turbo Pascal code (Goldberg 1989, p. 63) to show the reproduction algorithm's implementation:

```
function select(popsize : integer, sumfitness : real;
                    var pop : population) : integer;
 {Select a single individual via roulette wheel selection }
var rand, partsum : real; { Random point on wheel selection, partial sum
}
    j : integer;              { population index }
begin
 partsum := 0.0; j := 0;        { Zero out counter and accumulator }
 rand : random * sumfitness;  {Wheel point calculation uses random
number [0,1] }
 repeat { find wheel slot }
 j := j + 1;
 partsum := partsum + pop[j].fitness;                              (2.22)
 until (partsum >= rand) or (j = popsize);
 { Return individual number j }
 select : = j;
end;
```

where

random – is a real number between 0 and 1 or distributed $U(0,1)$,

sumfitness – total sum of the fitnesses,

partsum – partial sum of the fitness values,

pop[j].fitness – each member j of the population's fitness,

with rand given by the following formula:

rand := random * sumfitness.

We have previously summed the population fitnesses and stored them as sumfitness; we then multiply this by the normalized pseudorandom number, random. The repeat until section searches through the weighted roulette wheel until the partial sum is greater than or equal to the stopping point rand. The function then returns the current population index value assigned to select.

A listing of the formal steps for roulette wheel parent selection is:

1. Sum the fitnesses of all the population members and call the result sumfitness;

2. Generate n, a random number between 0 and total fitness; (2.23)

3. Return the first population member whose fitness, added to the fitnesses of the preceding population members, is greater than or equal to n.

While this is the simplest means of implementing selection, it is not the fastest. A binary search would make things more efficient in the search for the correct slot. This is just a simple and direct way of choosing offspring for the next generation, but it is certainly not the only way. Some methods implement a search with a bias towards the best.

### 2.6.1. Reproduction Example

Consider the selection process as a roulette wheel with each slot on the wheel representing a fitness of a member of the population. Each slot's area is in proportion to the fitness value: the higher the fitness the more area the slot has in proportion to the other slots.

We evaluate the binary strings right to left. If the function $x^2$ is to be optimized the following table could apply:

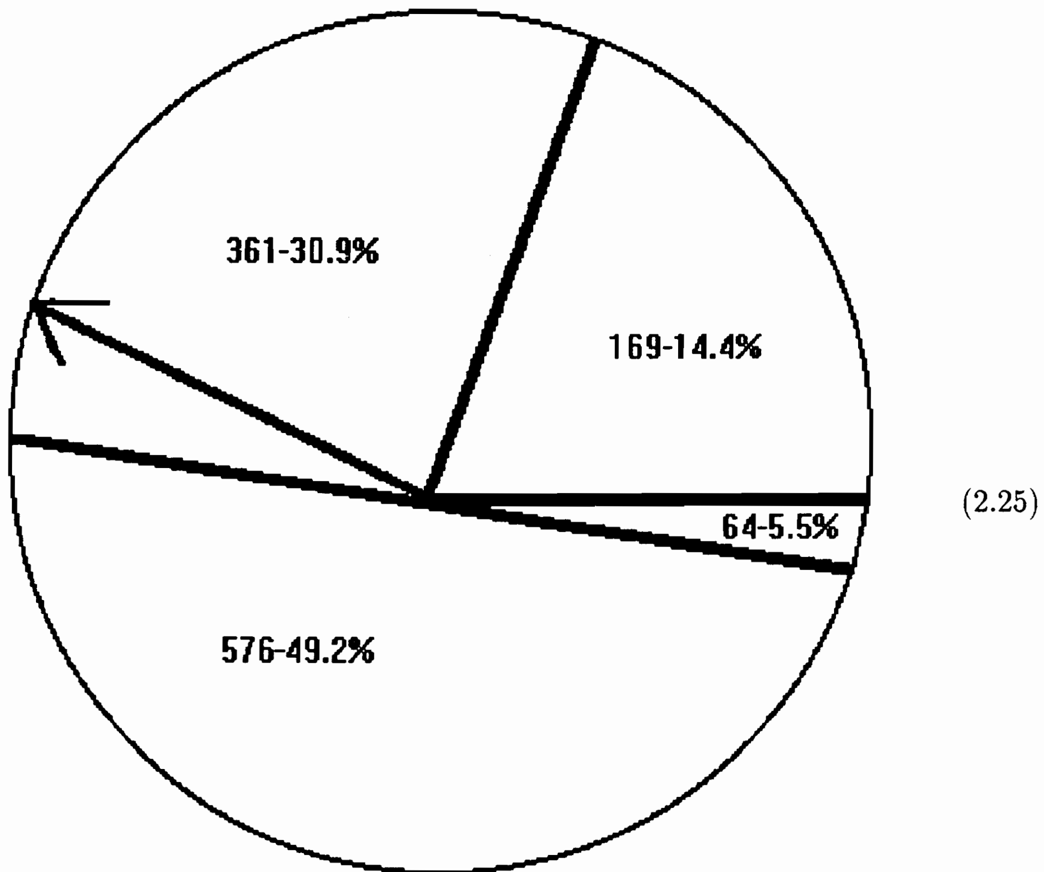
| String | $X$ | Fitness ($X^2$) |
|---|---|---|
| 01101 | 13 | 169 |
| 11000 | 24 | 576 |
| 01000 | 8 | 64 |
| 10011 | 19 | 361. |

(2.24)

Consider the circle with a spinning needle in (2.25). The needle will randomly land on one of four sections. Each of its four sections are in proportion to the values of $x^2$. The higher the value of $x^2$, the greater the chance that string will move to the next generation since its area on the circle is larger, the greater the chance the spinning needle will land on its section. There is no guarantee that the more fit strings will be chosen only a higher probability.

## 2.6.2. Reproduction Theory

At this point a more rigorous mathematical analysis is necessary to discuss the operation of a GA. This analysis is from Goldberg (1989). At any given step $t$ there are $m$ examples of a particular schema $H$ contained in the population $X(t)$, where $m = m(H,\ t)$. This indicates that there are possibly different quantities of different




(2.25)

So with $t = 0$, and with a stationary value for $c$, the following equation applies:

$$m(H,\, t) = m(H{,}0)\; (1 + c)^t, \tag{2.28}$$

which is similar to the compound interest equation in business finance (Goldberg 1989, p. 30). It is a geometric progression. Reproduction allocates exponentially increasing (decreasing) numbers to above- (below-) average schemata.

We note that this expected behavior is carried out with every schema $H$ contained in the population $X$, in parallel. This gives the GA its powerful search capability. We call the phenomenon implicit parallelism. All schemata grow or decay according to their averages under the operation of reproduction alone.

## 2.7. Crossover

The next operation is crossover implemented in a procedure of the same name. The routine takes two parent strings and generates two offspring strings. We pass to the procedure crossover the probabilities of crossover and mutation, along with the string length and crossover accumulator, ncross, and the mutation count accumulator, nmutation.

The Turbo Pascal code is (Goldberg 1989, p. 64):

```
Procedure crossover(var parent1, parent2, child1, child2 : chromosome;
                    var lchrom, ncross, nmutation, jcross : integer;
                    var pcross, mutation : real);
{cross 2 parent strings, place in 2 child strings }
var J;integer;
begin
 if flip(pcross) then begin
   jcross := rnd(1,lchrom − 1);
   ncross := ncross + 1;
  end else
  jcross := lchrom;                                                    (2.29)
{ 1st exchange, 1 to 1, and 2 to 2 }
for j := 1 to jcross do begin
   child1[j] := mutation(parent1[j], pmutation, nmutation);
   child2[j] := mutation(parent1[j], pmutation, nmutation);
end;
if jcross<>lchrom then {skip if cross site is lchrom-no crossover }
 for j := jcross+1 to lchrom do begin
 child1[j] := mutation(parent2, pmutation, nmutation);
 child2[j] := mutation(parent1, pmutation, nmutation);
 end;
end;
```

where the variables have been previously defined in (2.22). The procedure crossover contains a Boolean function called flip. In flip we toss a biased coin and it will come up heads (true) with a probability of pcross (a specified probability). Flip in turn calls on the pseudorandom number routine random. When flip is true a crossover occurs, with the crossing site selected in the function rnd, which returns a pseudorandom integer between specified upper and lower limits (that is between 1 and the chromosome's length less 1). When there is no crossover (flip is false), we select the cross site as the total length of the chromosome. This allows for mutation; mutation can occur whether we have crossover or not.

Remember, as useful as reproduction is, it does nothing to promote the search of new areas in the sample space. Copying old structures without change, will never allow anything new. This is where crossover comes into play. It is a structured yet

randomized interchange of information between two strings.

## 2.7.1. Crossover Example

Using an example from Goldberg (1989, p. 31), take a string of length 7 with two schemata represented in that string:

$$\begin{aligned} X &= 0\,1\,1\,1\,0\,0\,0 \\ H_1 &= *\,1\,*\,*\,*\,*\,0 \\ H_2 &= *\,*\,*\,1\,0\,*\,*. \end{aligned} \tag{2.30}$$

$X$ contains the representation of the two schemata, $H_1$ and $H_2$. Selection of a mate during crossover is random as is the selection of the crossover site. Let the crossover site be 3 (the cross will occur between genes 3 and 4). The result is:

$$\begin{aligned} X &= 0\,1\,1 \mid 1\,0\,0\,0 \\ H_1 &= *\,1\,* \mid *\,*\,*\,0 \\ H_2 &= *\,*\,* \mid 1\,0\,*\,*. \end{aligned} \tag{2.31}$$

Now in this instance we destroy schema $H_1$, but preserve $H_2$. Clearly, schema $H_2$ has a better chance of surviving than schema $H_1$. If $X$'s mate is the same as $X$ then no schema patterns are lost.

## 2.7.2. Crossover Theory

Again this analysis is from Goldberg (1989). The schema survives when the cross site falls outside the defining length. The probability of schema survival under simple crossover is:

$$p_s \geq 1 - \frac{\delta(H)}{l-1}. \tag{2.32}$$

The survival probability with crossover as random probability (i.e., $p_C \leq 1$) is:

$$p_s \geq 1 - p_C \frac{\delta(H)}{l-1}, \tag{2.33}$$

where $l$ is the number of possible sites, $\delta(H)$ is the defining length, $p_s$ is the probability of the schema's survival and $p_C$ is the probability of crossover.

Taking into account both reproduction and crossover, and assuming independence of the operators, an estimate is (Goldberg):

$$m(H,\, t+1) \;\geq\; m(H,\, t)\, \frac{f(H)}{\bar{f}} \left[1 - p_C \frac{\delta(H)}{l-1}\right]. \tag{2.34}$$

This gives an expression for the combined effects of crossover and reproduction on the growth or decline of schema. The survival of the schema depends on two

things: the fitness of the schema compared to the population's average and the defining length of the schema. The best schemata for survival are those that are above average in fitness and have a short defining length. They will grow at exponential rates.

## 2.8. Mutation

Mutation occurs in the function of the same name. This function uses the function flip to determine whether change an allele or not. The Turbo Pascal code implementation is (Goldberg 1989, p. 65):

```
function mutation(alleleval : allele; pmutation : real;
                    var nmutation : integer) : allele;
{ Mutate an allele w/pmutation, count number of mutations }
var mutate : boolean;
begin
 mutate := flip(pmutation); { Flip the biased coin }
 if mutate then begin                                        (2.35)
  nmutation := nmutation + 1;
  mutation : not alleleval; { Change bit value }
 end else;
  mutation := alleleval; { No change }
end;
```

where the variables have been previously defined in (2.22). The function flip will only show heads (true) pmutation of the time as a result of the call to the pseudorandom number generator random within flip itself. In some algorithms that use mutation the mere chance that a mutation occurs, does not mean that a mutation will occur. For example, if flip comes up heads (true) we then flip to see if the binary character in question changes to its opposite member. There is of

course in a fair coin flip, with a 50% chance of this happening. Thus we must be sure of the mutation steps.

## 2.8.1. Mutation Example

Mutation rates are small, about 0.003 for our samples. If we are dealing with say 20 bit positions as in the previous examples we can expect $20*0.003 = 0.06$ positions to be changed by mutation. In the simulation example thus far we can expect no bit positions to change as a result of mutation.

## 2.8.2. Mutation Theory

Again this analysis is from Goldberg (1989). Mutation is the random alteration of a single position that occurs with a probability of $p_m$. Hence, for a schema to survive, the specified positions must survive. An allele (an expression of a gene to its environment) survives with a probability of $(1 - p_m)$. If there are $o(H)$ fixed positions then $(1 - p_m)^{o(H)}$ is the probability of surviving mutation. It is possible to use an approximation to $(1 - p_m)^{o(H)}$ when $p_m$ is small by using the expression $1 - o(H)p_m$.

We put this into (2.34) and after simplifying and ignoring the cross-product terms (which are small), gives the following equation for the survival of schema (Goldberg 1989):

$$m(H, t+1) \geq m(H, t) \frac{f(H)}{\overline{f}} \left[1 - p_c \frac{\delta(H)}{l-1} - o(H)p_m\right]. \tag{2.36}$$

Mutation does little to change the equation, but we must consider it. We call this equation the Schema Theorem or the Fundamental Theorem of Genetic Algorithms. The theorem has far reaching implications in the study of GA's.

## 2.9. The $\kappa$-Armed Bandit Problem

Clearly there is an exponentially increasing number of trials to the observed best schemata. Why is this good? For an answer consider the two-armed bandit problem from statistical decision theory.

It is the two-armed bandit problem or in its general form the $k$-armed bandit problem. A limited amount of money is available to play on two slot machines known as one-armed bandits. The distributions of the two payouts for the two machines are $(\mu_1, \sigma_1^2)$ and $(\mu_2, \sigma_2^2)$; with $\mu_1 \geq \mu_2$ and $\sigma_1^2 = \sigma_2^2$. To maximize winnings we put most of the money into playing the machine with the higher average payoff, $\mu_1$. That information (which bandit is the higher paying machine) is unknown. The logical action is to take a given amount of money that is less than the total and play the two bandits. Then choose a machine based on these trials; and play the rest of the money on that bandit.

This is a tradeoff situation. The longer the two bandits are played, the more information obtained and the more certain is the decision on which of the two bandits to concentrate; also the less money available to play the chosen higher paying machine after having made a selection. The tradeoff is the search for knowledge about the machines, the first stage; and the exploitation of that

knowledge about the machines, the second stage. This balance between gaining knowledge and its exploitation is a recurring paradigm in adaptive theory.

A method given by De Groot (1970, p. 394-404) performs the two separate steps as stated. Given a total of $N$ trials that the limited money will allow; play $n$ trials on each machine before making a decision ($2n < N$). After the first stage, we have a total of $N-2n$ trials to give to the selected higher paying bandit. The expected loss $L$ is (De Jong 1975, p. 25):

$$L(N,\, n) = |\mu_1 - \mu_2| \left[(N-n)\, q(n) + n\, (1 - q(n))\right], \tag{2.37}$$

with $q(n) =$ the probability that we believe the worst arm to be the best arm after $n$ trials. De Groot (1970, p. 394-404) approximated this probability:

$$q(n) \simeq \frac{e^{-x^2/2}}{\sqrt{2\pi}\, x}, \text{ where } x = \frac{y - \mu_1 - \mu_2}{\sqrt{\dfrac{\sigma_1^2}{(n-N)} + \dfrac{\sigma_2^2}{n}}} \tag{2.38}$$

or

$$q(N-n,n) = \frac{1}{\sqrt{2\pi}} \frac{\sqrt{\dfrac{\sigma_1^2}{(N-n)} + \dfrac{\sigma_2^2}{\mathrm{n}}}}{\mu_1 - \mu_2} \quad \exp\, \frac{1}{2}\left[ \frac{-(\mu_1-\mu_2)^2}{\dfrac{\sigma_1^2}{(N-n)} + \dfrac{\sigma_2^2}{n}} \right]. \tag{2.39}$$

Thus $q$ is a function of both variances, both means, $N$ the total number of trials,

and $n$ the number of test trials; it decreases exponentially as a function of $n$. To minimize loss, as $N$ increases, the number of trials allocated to the observed best, $N-n$, should be increased sharply relative to $N$. This should simplify the expression for $x$. To minimize the expected losses from (2.37) we take its first derivative (to find the optimum), and then its second derivative (to check for either a minimum or maximum); and then set the first derivative to zero and solve for $n$. By doing that we minimize the losses and the value for $n^*$ is:

$$n^* \sim b^{-2} \ln\left[\frac{b^4 N^2}{8\pi \ln N^2}\right], \text{ where } b = \frac{\sigma_1}{(\mu_1 - \mu_2)}. \tag{2.40}$$

If we want to know what the trials allocated to the best machine then we can solve for $N - n^*$, which gives:

$$N - n^* \simeq N_1 \simeq \sqrt{8\pi b^4 \ln N^2}\; e^{\,n^*/2b^2}. \tag{2.41}$$

The idea we should learn from these equations is that, to minimize losses, allocate more than exponentially increasing trials to the observed best bandit. In reality there is not enough information to find $n^*$, since we must know about outcomes before they occur. This is true regardless of the distribution defining the bandits. Nevertheless, this does give an upward bound on the strategy. A GA can solve this problem for not just two bandits, but for many or $k$ bandits.

The $k$-armed bandit problem was first advocated by Holland (1975) as a

mathematical explanation to the solution capability of GA's. The minimal expected loss in the allocation of trials to $k$ competing bandits is quite similar to the two-armed bandit. It says that greater than exponentially increasing trials should be given to the observed best of the $k$ bandits.

Recently, some have questioned the $k$-armed bandit problem as an accurate model of the operation of a GA. In a GA we create new schemata and discard them according to the rules discussed above. A $k$-armed bandit problem does not assume that. It assumes that information is kept in the gathering stage and the algorithm uses all of it in the exploitation stage.

## 2.10. Alternative Search Methods

There are more elaborate selection schemes than the one already discussed. In a genetic algorithm it is possible for the function, a probabilistic search, to slide back from an indicated optimum in succeeding generations. This usually is undesirable. The search of course becomes less efficient. We can use a variation on the basic selection model called the elitist model in which the most fit member of each generation is automatically put into the next generation. This insures that the current generation is always going to have the most fit member of the whole genetic search.

## 2.11. Scaling

There is a tendency in genetic algorithms for some super individuals to dominate in

a base model genetic algorithms. A loss of variety in the search occurs when this happens. It could lead to premature convergence to a false calculated optimum. We must scale the objective function back to prevent a takeover of the population by these super individuals. However, on the other end of the search when the population is converging there is a need for increased competition less the search wanders among individuals of equal fitness. It is then that the objective function to be scaled up to accentuate the differences among population members.

Accomplishing this is not so simple as the above explanation. The most fundamental scaling is linear scaling. It requires a linear relationship between $f'$ and the scaled fitness $f'$, which gives a relationship as follows:

$$f' = \mathrm{a}\, f + \mathrm{b}. \tag{2.42}$$

We can choose the coefficients $a$ and $b$ in a number of ways, but the average scaled fitness $f'_{avg}$ must be equal to the average raw fitness $f_{avg}$. This insures that the subsequent use of the selection procedure will insure that each average population member will contribute one expected offspring to the next generation. An appropriate formula is $f'_{max} = \mathrm{C}_{mult} * f_{avg}$, where $\mathrm{C}_{mult}$ is the number of expected copies desired for the best population member. Analysts have successfully used values for $\mathrm{C}_{mult}$ from 1.2 to 2 Goldberg (1989, p.77).

## 2.12. Discussion of Genetic Coding

Goldberg (1989, p. 80) gives two rules of coding a genetic algorithm. The first rule

is:

> The user should select a coding so that short, low order schemata are relevant to the underlying problem and relatively unrelated to the schemata over other fixed positions.

The second rule is

> The user should select the smallest alphabet that permits a natural expression of the problem.

Davis (1991, p. 63) writes that bit string coding is acceptable for performing fundamental research in the filed of GA's, but for "real world problem solving" it is not very practical. Davis adheres to the philosophy that research and problem solving using GA's are two different fields requiring two different approaches. The theoretically appealing binary coding scheme is not necessarily the best scheme for real world problems.

At this point we may wonder why we are using binary codings. We assume that binary codings are the most efficient. A binary string is simply a string of concatenated 0's and 1's. GA's exploit similarities in string codings so that schema of short defining length and high performance survive. We are trying to associate high fitness with similarity among strings in the population. In the example just completed, we are trying to associate similarities and the small cardinality of the alphabet helps this. An example from Goldberg (1989, p. 80) should clarify this point.

Take a binary schema such as:

$$\begin{array}{cc} \text{Binary} & \text{Nonbinary} \\ 0\,0\,0\,0\,0 & A \\ 0\,0\,0\,0\,1 & B \\ \vdots & \vdots \\ 1\,1\,0\,0\,1 & Z \\ 1\,1\,0\,1\,0 & 1 \\ 1\,1\,0\,1\,1 & 2 \\ \vdots & \\ 1\,1\,1\,1\,1 & 6. \end{array} \tag{2.43}$$

In this example there is a mapping of the binary integers and the 32-letter alphabet consisting of the 26 letters and the digits 1-6.

Now the binary and nonbinary codings should code the same number of alternatives, but the length of their strings is different. To get the same number of points in space one should have $2^l = k^{l^m}$, where $l$ is the binary code length and $l^m$ is the nonbinary code length. The number of schemata is $3^l$ for the binary coding and $(k+1)^{l^m}$ in the nonbinary case. The binary alphabet offers the maximum number of schemata per bit of information. These similarities are the essence of the genetic search.

Binary codings lend themselves to the cutting and splicing of chromosomes that is a critical mechanism is the operation of a GA. It facilitates the crossover operator and the mutation operator. Any proof of a theorem, until recently, used a binary coding scheme for simplicity. In practical applications this is not the case since in many cases the GA is combined with an algorithm to give a hybrid algorithm. This

lends itself to a coding scheme based on the number ten and not two.

## 2.13. Application of Genetic Algorithms

It is now apparent that a GA can work quite efficiently to solve optimization problems such as optimizing $x^2$ in (2.24). That function, of course, did not require GA's to maximize it. An application of basic calculus would be simpler. What if there were more than one variable and what if the function was not as well behaved or as well defined as $x^2$? This would most likely be the situation when dealing with a practical optimization problem.

We address the question of dealing with more than one variable several ways; the most obvious is simply to concatenate the strings (or substrings) into one major string. We decide the major string just like the strings for a single variable except that each section of the string represents a different variable. For example, consider the string $C$ below representing three decision variables:

$$C = 011100011000, \tag{2.44}$$

or more clearly:

$$C = 0111|0001|1000. \tag{2.45}$$

We break the string down into its 3 substrings:

| Sub-string | $X_i$ |
|---|---|
| 0111 | 7 |

$$
\begin{array}{ll} 0001 & 1 \\ 1000 & 8. \end{array} \tag{2.46}
$$

So $X_1 = 7$, $X_2 = 1$ and $X_3 = 8$. The major string would reproduce, crossover and mutate as before, except than each string now carries information for a set of decision variables instead of one. That string would need to be decoded into all the values for the decision variables it represents before evaluating the objective function.

In this dissertation we will use only one crossover site per string. There have been studies where there have been two or more, but this dissertation will use one crossover site.

This is not the only method to map a practical problem to a GA for a solution. There are many more. Finding a way to apply GA's to a specific problem is a complete subfield of study for GA's. There are many situations where the method just described would not work; an example would be the traveling salesman problem. A simple concatenation of the strings would not work since it probably would not give a legal tour.

### 2.13.1. Parameter Values in a Genetic Algorithm

A GA requires values for each of its parameters. From the three defined GA operators: reproduction, crossover and mutation, we have three parameters that we must assign values. They are:

$$
\begin{aligned}
&1.\ \text{Population Size,}\\
&2.\ \text{Crossover Rate,}\\
&3.\ \text{Mutation Rate.}
\end{aligned}
\tag{2.47}
$$

For completeness, one more GA parameter needs to be defined: the generation gap. It is strictly a measure of how many members of a population are carried into the next generation. It can be between zero and one and is defined as follows:

$$
\begin{array}{ll}
G = 1 & \text{nonoverlapping populations,} \\
0 < G < 1 & \text{overlapping populations.}
\end{array}
\tag{2.48}
$$

In the present generation we choose $(1 - G)*n$ individuals to be placed in the next generation, where $n$ is the population size we defined earlier.

Various studies (De Jong 1975, Grefenstette 1986) determine that the following values would be appropriate:

$$
\begin{aligned}
&1.\ \text{population size} - 30 \text{ to } 200,\\
&2.\ \text{crossover rate} - 0.60 \text{ to } 1.00,\\
&3.\ \text{mutation rate} - 0.001 \text{ to } 0.003,\\
&4.\ \text{generation gap} - 1.0.
\end{aligned}
\tag{2.59}
$$

## 2.13.2. Genetic Drift

In the science of genetics, genetic drift is the phenomenon that occurs when an evolving population converges to a preliminary sub-optimum. It is a consequence of random selection processes in a finite population. It is the main reason that a large population size is chosen. This illustration employs the terminology of a stochastic process. A population $X(t)$ of $N$ individuals evolves to $X(t+1)$ by a selection of $N$ uniformly random selections with replacement. Assuming a binary genetic representation and a uniform random initial population $X(0)$, then the expected number of 0-alleles $R_1(t)$ for gene $i$ is $N/2$. However, as $t$ increases, the variance of $R(t)$ also increases and deviations from the norm are probable.

Consider the model as a Markov process in which the states are simply the $N+1$ possible values of $R_1(t)$. The transition probability $P_{jk}$ is simply the probability of $k$ successes in $N$ Bernoulli trials given a probability of success on each trial of $j/N$. In equation form:

$$P_{jk} = \binom{N}{k}\left(\frac{j}{N}\right)^k \left(1-\frac{j}{N}\right)^{N-k}. \tag{2.50}$$

The initial state probabilities $P_k$ are simply:

$$P = P_{\frac{N}{2},k} = \binom{N}{k}\left(\frac{1}{2}\right)^k \left(\frac{1}{2}\right)^{N-k} = \binom{N}{k}\left(\frac{1}{2}\right)^N . \tag{2.51}$$

States $R_1(t) = 0$ and $R_1(t) = N$ are absorbing, all other $N-1$ states being transient. The probability of being in either of the absorbing states approaches 1 in

the limit.

Employing a conceptual term from De Jong (1975, p. 54), our interest is in the probability of first entry. The expected number of generations, $f_{jk}$, to first entry into a state $k$ from state $j$ is:

$$f_{jk} = \sum_{n=0}^{\infty} n \, f_{jk}^{n}, \tag{2.52}$$

where $f_{jk}^{n}$ is the probability of first entry to $k$ from $j$ in exactly $n$ steps. The computation of these expected values is difficult. We compute the terms $f_{jk}^{n}$ recursively

$$P_{jk}^{n} = \sum_{i}^{n} f_{jk}^{i} \, f_{jk}^{i} \, P_{kk}^{n-1}, \tag{2.53}$$

with the extended transition probabilities $P_{jk}^{n}$ computed by raising the transition matrix:

$$P = \left[P_{ij}\right] \tag{2.54}$$

to the $n^{th}$ power. A simulation study by De Jong (1975) indicates that a sufficiently large population must be used or else genetic drift will occur. If it does occur then the GA could converge to a false optimum.

An obvious solution is to increase the size of the population. That will increase the time the GA takes to attain an adaptive response. Another way is to introduce mutation; then clearly states $R_1(t) = 0$ and $R_1(0) = N$ are no longer absorbing. If the mutation rate approaches 0.50, then for all practical purposes we have a random search.

We can use the phenomenon of premature convergence or genetic drift to the advantage of the optimization. There is a new type of GA called a micro-GA or $\mu$GA. The whole concept of a $\mu$GA is to use this premature convergence to its advantage and reduce the size of the population to get a very rapid adaptive process.

# Chapter 3. Examples

This dissertation has three examples. All examples are from textbooks and each modified, if necessary, to an optimization problem.

## 3.1. Example 1 - Inventory System

The first example to be optimized is an inventory system (Law and Kelton 1991, p. 75-103 and Banks and Carson 1984, p. 243-251). A company sells a single product and wants to know how many items to have on hand in the next $n$ months. The times between demands are i.i.d. exponential random variables with a mean of 0.1 month. The sizes of the demands, $D$, are i.i.d. random variables with the following distribution:

$$D = \begin{cases} 1 \text{ wp } \frac{1}{6} \\ 2 \text{ wp } \frac{1}{3} \\ 3 \text{ wp } \frac{1}{3} \\ 4 \text{ wp } \frac{1}{6}. \end{cases} \tag{3.1}$$

The inventory review is at the start of each month. The ordering cost is:

$$K + iZ, \tag{3.2}$$

where $K = \$ \ 32.00$ is the setup cost, $i = \$ \ 3.00$ is the incremental ordering cost per item, and the number of items ordered is $Z$. The delivery lag is probabilistic with a uniform distribution of 0.5 to 1.0. The policy on ordering inventory is as follows:

$$Z = \begin{cases} S - I & \text{if } I < s \\ 0 & \text{if } I \geq s, \end{cases} \tag{3.3}$$

where $I$ is the inventory level at the beginning of each month before orders are placed, $S$ is the level of inventory to order up to and $s$ is the level of inventory that triggers the placement of an order. We immediately satisfy a demand when it occurs if the inventory is available; if it is not available, we backlog the excess demand and satisfy it by future deliveries. Each arrival first satisfies backlogged demand and the remainder goes into inventory.

Besides the costs outlined above there are two more: holding costs and shortage costs. The holding costs are:

$$h \frac{\int_0^n I^+(t)\,\mathrm{d}t}{n}, \tag{3.4}$$

where $n$ is the number of months, $\int_0^n I^+(t)\ \mathrm{d}t/n$ is the time-average inventory, and $h = \$ \ 1$ item/month. The shortage costs are:

$$\pi \frac{\int_0^n I^-(t)\,dt}{n}, \tag{3.5}$$

where $n$ is the number of months, $\int_0^n I^-(t)\,dt/n$ is the time-average number of items in backlog and $\pi = \$\ 5$ item/month.

Assume that $I(0) = 60$ (initial inventory is 60 units) and no order is outstanding. Simulate for 10 years and determine what the values of $s$ and $S$, the decision variables, are to attain the lowest operating costs of the inventory system.

The response is:

$$cost = aordc + ahldc + ashrc, \tag{3.6}$$

where $aordc$ is the average holding costs, $ahldc$ is the average ordering costs, and $ashrc$ is the average shortage costs.

## 3.2. Example 2 - University Time-Shared Computer System

The second example is a university computer system (Law and Kelton 1991, p. 157-170 and Bratley, Fox and Schrage 1987, p. 257-259). A university committee plans a computer system consisting of 40 terminals and would like to know how to set the design parameters to minimize the average response time at a given terminal. Each student working at a terminal "thinks" for a time that is exponentially distributed with a mean of 25 seconds. The student then sends a command to the

CPU of the mainframe that has a service time that is an exponentially distributed random variable with a mean of $s$ seconds. The arriving jobs join a queue and are served in round-robin fashion with each job getting a maximum service quantum of $q$ seconds. When the service time of a job, $s$, is less than or equal to $q$, the CPU spends time $q$ plus an overhead $\tau$ processing the job, and then sends it back to the terminal. If the service time is greater than $q$, the CPU spends time $q$ plus $\tau$ processing the job, and then sends the uncompleted job back to the end of the queue, with its service time decreased by $q$ seconds. The job then repeats this process and goes back to the CPU until the service is eventually completed.

The response time is the time that elapses from when the job leaves until it is finished being processed by the CPU; it is the response to be minimized. The decision variables are: 1. quantum of processing time, $q$, and 2. overhead, $\tau$. The response function is:

$$\begin{aligned} cost = {} & average\ response\ time * 1000\ + \\ & average\ number\ in\ queue * 1000\ + \\ & \left(\frac{(0.005 - overhead)}{0.005}\right)^2 * 10000\ + \\ & \left(\frac{(0.50 - quantum)}{0.50}\right)^2 * 10000. \end{aligned} \tag{3.7}$$

## 3.3. Example 3 - Job-Shop Model

The third example is a job-shop model (Law and Kelton 1991, p. 185-200) and consists of a manufacturing facility made up of five groups of machines. Each group consists of identical machines. The jobs arrive at an i.i.d. exponential rate with a mean of 0.125 hour. The distribution of the job types is as follows:

| Job Type | Cumulative Probability | |
|---|---|---|
| 1 | 0.30 | |
| 2 | 0.80 | (3.8) |
| 3 | 1.00. | |

The routing of the job types is as follows:

| Job Type | Machine Groups | |
|---|---|---|
| 1 | 3, 1, 2, 5 | |
| 2 | 4, 1, 3 | (3.9) |
| 3 | 2, 5, 1, 4, 3. | |

There are five machine groups. It is possible for a job to arrive at a group and find all machines busy in which case it will join a FIFO (first in, first out) queue for that machine group. The time to perform a task at a particular machine is an independent 2-Erlang randomly distributed variable whose mean is dependent on the job type and the group in which the machine belongs. The following table gives

the relevant data:

| Job Type | Mean service times for successive tasks, (2-Erlang distribution, hours) |  |
|---|---|---|
| 1 | 0.50, 0.60, 0.85, 0.50 |  |
| 2 | 1.10, 0.80, 0.75 | (3.10) |
| 3 | 1.20, 0.25, 0.70, 0.90, 1.00. |  |

Simulate the job-shop for 365 eight-hour days. All machines at each group are identical so the company wants to minimize the overall average job total delay by optimally placing the machines in the different groups. We calculate the overall average delay by a weighted average of the average delays calculated in each of the five queues for each machine. The weights are the probabilities of a part going to each station. The decision variables are the number of machines in each of the five groups.

The response function is:

$$cost = overall\ average\ delay\ in\ queue * 1000\ + \\ total\ number\ of\ machines * 1000. \tag{3.11}$$

## 3.4 Example Selection Criteria

We now discuss the reason for the selection of these three examples. The inventory

example used integer decision variables. It was also an example in which the calculated optimum was already determined in a textbook (Law and Kelton 1982, p. 379). This was a typical example given in simulation texts. We chose the university computer example because it used real decision variables and again it was an example common in simulation texts. We selected the third example, the job-shop, owing to its complexity and its prevalence in simulation texts. This would be a complex example that would really test the effectiveness of a genetic algorithm.

## 3.5. Use of Genetic Algorithms in Simulation Optimization

A GA can optimize a function. The real utility of a GA comes when we use it to optimize a function that is not as analytically straightforward. A simulation model is a stochastic function (Law and Kelton 1991, p. 679). The complete set of coded instructions constitutes a very complicated function that is usually not analytically tractable.

It is straightforward to apply a GA to a simulation model. Whenever the code in the GA goes to evaluate the objective function it simply makes a run of the simulation model. We should not base the model's response, however, on just one run; that could easily lead to erroneous results. We will replicate the model three times. We base this on a series of tests that started out with five replications as recommended by Law and Kelton (1982, p. 302). We wanted to reduce simulation execution time so we tried running the simulation at five replications and at three replications. There was no significant difference in the correlation present in the

three replications model as compared to the five replications model. Thus we chose to replicate only three times to save time on the simulation runs.

Each performance measure, response, that will be optimized in the three examples above is to be minimized. A GA is a maximizer. We must modify it to use it as a minimizer. For example, if we want to minimize $X_1$ then we can set up a simple relation:

$$Y = X_{max} - X_1. \tag{3.12}$$

Then the GA tries to maximize $Y$, with $X_{max}$ being the highest observed value of $X_1$. The variable of interest is $X_1$, and by maximizing $Y$, (with $X_{max}$ constant) $X_1$ is minimized.

This is a critical point to understand. Genetic algorithms operate on a population by comparisons of each population member's fitness. These comparisons are what create the incentive to probabilistically move population members to the next generation. The choice of $X_{max}$, can not be made casually. If $X_{max}$ is too low then $Y$ may be negative and if $X_{max}$ is too high the search degrades. The degradation is simply because of the fact that the larger $X_{max}$ is the larger $Y$ is. That leads to less striking differences and the search efficiency decreases. A simple example demonstrates this point. Consider the situation when $X_{max}$ and $X_i$ are as follows:

| $Y$ | $X_{max}$ | $X$ |
|---|---|---|
| 90 | 100 | 10 |
| 80 | 100 | 20 |
| 70 | 100 | 30 |
| 60 | 100 | 40 |

(3.13)

| | | |
|---|---|---|
| 50 | 100 | 50. |

The average value for $Y$ the value that we are seeking is 70 with a high of 90 or 128.5% of the average and a low of 50 or 71.4% of the average.

Now consider (3.14) in which the value of $X_{max}$ is 900 greater or 1000:

| $Y$ | $X_{max}$ | $X$ |
|---|---|---|
| 990 | 1000 | 10 |
| 980 | 1000 | 20 |
| 970 | 1000 | 30 |
| 960 | 1000 | 40 |
| 950 | 1000 | 50. |

(3.14)

In this case the average value of $Y$ is 970 with a high of 990 or 102.06% of the average and a low of 950 or 97.9% of the average.

Clearly, the differences in the $Y$ values are sharper in (3.13) than in (3.14). The algorithm uses the differences of the fitness values to choose which strings survive into the next generation. We can expect a better search in (3.13) than we can in (3.14).

## 3.6. Changing the Domain of Solutions

The decision variables in a simulation optimization may not always be integers; they can also be real. In the case of a real variable we must convert from a parameter $x \in [0, 2^l]$ to another interval $[U_{min}, U_{max}]$. An appropriate formula to

map to a different variable interval is:

$$\pi = \frac{U_{min} - U_{max}}{2^l - 1}, \tag{3.15}$$

where $U_{max}$ is the specified maximum that the variable can be, $U_{min}$ is the specified minimum that the variable can be and $l$ is the length of the chromosome.

A example from (Goldberg 1989, p. 82) demonstrates the mapping.

$$\begin{array}{c} \underline{\text{Single Parameter } (l = 4)} \\ 0\,0\,0\,0 \rightarrow U_{min} \\ 1\,1\,1\,1 \rightarrow U_{max} \end{array} \tag{3.16}$$

As we demonstrated above we can construct a multiparameter coding with concatenating as many single parameter codings as the model needs. We must remember that each requires its own specified $U_{max}$ and $U_{min}$ values as shown in (3.16) and we must decode separately each section representing a single variable.

There has been some work on the coding of strings other than binary, but for now binary coding is still the most common method.

# Chapter 4. Numerical Results

In this chapter we examine the numerical results from the computer experiments of the three examples discussed in Chapter 3. We look at each example and its output using the three search techniques, the Hooke and Jeeves pattern search, the response surface method as outlined by the code from Dennis Smith (1978) and the genetic search method.

First we must discuss the method for generating random numbers. Then we will discuss the experimental setup, then how we carried out each experiment and then compare it to the other search methods.

## 4.1. Random Number Generator

The random number generator selection is crucial because the method of random number generation determines the "randomness" and hence the success of a simulation. Moreover, since the genetic search also uses random numbers we must be careful to avoid any unexpected correlation that may improperly influence the simulation's outcome. The most important point in choosing our random number generator is not to exceed the period of the random number generator. Hence, because we need many random numbers we must find one that has a long period.

The random number generator that we have chosen to use for these computer experiments is the one suggested by Marse and Roberts (1983). It uses the source code available in the computer languages FORTRAN, Pascal and C (Law and Kelton 1991). It has a period of $2^{31}-2$. As a precaution in all computer experiments we counted the number of random numbers used to make sure that they never exceeded the period of the random number generator.

The random number generator chosen is a prime modulus multiplicative linear congruential generator (PMMLCG). To explain how this works consider a LCG given by the formula:

$$Z_i = (a\ Z_{i-1} + c)(\text{mod } m) \tag{4.1}$$

where $m$ is the modulus, $a$ is the multiplier and $Z_0$ is the seed, with all variables nonnegative integers. Multiplicative LCG's are similar to LCG's except that $c$ is not added as in (4.1). This violates theorems of LCG's that guarantee a period if certain criteria are met since one of the criteria includes $c$, which is now eliminated. However, careful selection of $m$ and $a$ can guarantee a period of $m-1$.

The selection of $m$ is not simple. We usually select $m$ so $m = 2^b$, where $b$ is the number of binary digits per word on a computer. Let $m$ be the largest prime number that is less than $2^b$. In the case of $b = 31$, the largest prime less than $2^{31}$ is $2^{31}-1$ or 2,147,483,647. For $m$ prime the period is $m-1$ if $a$ is a *primitive element modulo m* or the integer $l$ for which $a^l - 1$ is divisible by $m$ is $l = m-1$. This will guarantee a period of $m-1$; $Z_0$ can be an integer from 1 to $m-1$ and this still

holds. This is a prime modulus multiplicative linear congruential generator (PMMLCG).

## 4.2. Experimental Setup

The experimental setup is as follows. Please note that we have exhaustively run each program to obtain the indicated optimum. We also replicated each search 3 times. We chose the simulation length so that each example problem reached steady state. The simulation length in each example's replication was the same. We conducted a total of 50 searches and made our comparison's of each search technique using the following criteria.

The average response for each example and the standard deviation over the fifty searches of each variable we calculated by the formula:

$$\overline{X}(n) = \frac{\sum_{i=1}^{n} X_i}{n}. \tag{4.2}$$

where $X_i$ is the sample response, $n = 50$ the sample size, and $\overline{X}(\mathrm{n})$ is the average sample response.

The upper and lower 95% confidence limit of the response variable for each example and for each search within that example, we calculated by the formula:

$$\overline{X}(n) \pm t_{1-\frac{\alpha}{2}} \sqrt{\frac{S^2}{n}}, \tag{4.3}$$

where $n = 50$, $t = 1.676$, $\alpha = 0.05$ and $S^2$ is defined as:

$$S^2 = \frac{\sum_{i=1}^{n} (X_i - \overline{X}(n))^2}{n-1}. \tag{4.4}$$

The next four formulas $(4.5 - 4.8)$ concern the average distance from the initial search points to the final search points.

The average distance traversed in each search we calculated by the following formula:

$$\overline{D}beg - end = \frac{\sum_{i=1}^{n} Dbeg - end_i}{n}, \tag{4.5}$$

where $n = 50$. The following formula gives $Dbeg - end_i$ (the distance from the beginning point to the stopping point for the $i$th search):

$$Dbeg - end_i = \sqrt{\sum_{d=1}^{j} (V(d)beg_i - V(d)end_i)^2}, \tag{4.6}$$

where $V(d)beg_i$ is the initial value for the $d$th decision variable, $V(d)end_i$ is the final value for the $d$th decision variable, and $j$ is the number of decision variables in the search. For example, for the inventory system and computer system $j = 2$, in the job-shop $j = 5$.

The following formula gives the variance of the distance traversed:

$$Vbeg - end = \frac{\sum_{i=1}^{n} (Dbeg - end_i - \overline{D}beg - end)^2}{n - 1}, \tag{4.7}$$

with $\overline{D}beg - end$ and $Dbeg - end_i$ defined in (4.5) and (4.6) respectively, and $n = 50$.

The confidence interval formula for $\overline{D}beg - end$ is:

$$\overline{D}beg - end \pm t_{1-\frac{\alpha}{2}} \sqrt{\frac{Vbeg - end}{n}}, \tag{4.8}$$

where $n$, $t$, and $\alpha$ have the same values as in 4.3 and $Vbeg - end$ is defined in (4.7).

The next four formulas (4.9 – 4.12) concern the average distance from the initial search points to the calculated optimum.

The following formula gives the average distance from the 50 initial points to the

calculated optimum in each search:

$$\bar{D}beg-opt = \frac{\sum_{i=1}^{n} Dbeg-opt_i}{n}, \tag{4.9}$$

here again $n = 50$. The following formula gives $Dbeg-opt_i$:

$$Dbeg-opt_i = \sqrt{\sum_{d=1}^{j} (V(d)beg_i - V(d)opt_i)^2}, \tag{4.10}$$

where $V(d)beg_i$ is the initial value for decision variable $d$, $V(d)opt_i$ is the calculated optimum value for decision variable $d$, and $j$ is the number of decision variables in the search.

The following formula gives the variance of the distance

$$Vbeg-opt = \frac{\sum_{i=1}^{n} (Dbeg-opt_i - \bar{D}beg-opt)^2}{n-1}, \tag{4.11}$$

where $n = 50$ and $\bar{D}beg-opt$ and $Dbeg-opt_i$ defined in (4.9) and (4.10), respectively.

The confidence interval formula for $Dbeg-opt$ is:

$$\overline{D}\,beg-opt \pm t_{1-\frac{\alpha}{2}}\sqrt{\frac{Vbeg-opt}{n}}, \qquad (4.12)$$

where $n$, $t$, and $\alpha$ have the same values as in (4.3). Note that formulas 4.5-4.12 cannot be used for the genetic search since we have no single initial point for the fifty searches. So any table entry for a genetic search that requires formulas 4.5-4.12 we mark NA (not available).

The next four formulas $(4.13-4.16)$ concern the average distance from the final search points to the calculated optimum.

The following formula gives the average distance from the final point to the calculated optimum in each search:

$$\overline{D}\,end-opt = \frac{\sum_{i=1}^{n} Dend-opt_i}{n}, \qquad (4.13)$$

where $n = 50$. The following formula for $Dend-opt_i$ (the distance from the beginning point to the optimum for the $i$th search) is:

$$Dend-opt_i = \sqrt{\sum_{d=1}^{j} (V(d)end_i - V(d)opt_i)^2}, \qquad (4.14)$$

where $V(d)end_i$ is the final value for decision variable $d$, $V(d)opt_i$ is the calculated optimum value for decision variable $d$, and $j$ is the number of decision variables in the search.

Formula (4.15) gives the variance of the distance from endpoints to calculated optimum:

$$Vend-opt = \frac{\sum_{i=1}^{n}(Dend-opt_i - \overline{D}end-opt)^2}{n-1}, \tag{4.15}$$

where $Dend-opt_i$ is defined in (4.14) and $\overline{D}end-opt$ is defined in (4.13) and $n = 50$.

The confidence interval for $\overline{D}end-opt$ is:

$$\overline{D}end-opt \pm t_{1-\frac{\alpha}{2}}\sqrt{\frac{Vend-opt}{n}}, \tag{4.16}$$

where $n$, $t$, and $\alpha$ have the same values as in 4.3 and $Vend-opt$ is defined in (4.15). The next set of calculations concerns the runs per search.

The following formulas give the average number of runs per search and its variance:

$$\overline{R} = \frac{\sum_{i=1}^{n} \mathrm{R}_i}{n} \tag{4.17}$$

and

$$Rvar = \frac{\sum_{i=1}^{n} (R_i - \overline{R})^2}{n-1}, \tag{4.18}$$

where $\overline{R}$ is the average number of runs per search, $Rvar$ is the variance of the number of runs per search, $R_i$ is the number of runs for each search and $n = 50$ and is the sample size.

The formula to obtain the confidence interval for the number of runs per search is:

$$\overline{R} \pm t_{1-\frac{\alpha}{2}} \sqrt{\frac{Rvar}{n}}, \tag{4.19}$$

where $n$, $\alpha$ and $t$ have the same values as in 4.3.

We also used statistical tests in the analysis of the data. The test compared the difference of two means one based on either the pattern search or the response surface search compared to the GA search. We used a $t$-test with the assumption of equal but unknown means. The assumption that the means were equal, but unknown, is reasonable since the starting points on each example for a given system were generated from the same statistical distribution. The calculation for

the $t$ value used the following formula:

$$t = \frac{\overline{X}_1 - \overline{X}_2 - d_0}{S_p\sqrt{\frac{1}{n_1} + \frac{1}{n_2}}} \tag{4.20}$$

where $\overline{X}_1$ and $\overline{X}_2$ is the means of samples 1 and 2 respectively, $d_0$ is the difference between the means and $t$ is distributed $t_\nu$, $\nu$ defined as the degrees of freedom calculated by,

$$\nu = n_1 + n_2 - 2 \tag{4.21}$$

with $n_1$ and $n_2$ the sizes of samples 1 and 2 respectively and

$$\sigma_1 = \sigma_2, \tag{4.22}$$

with both variances unknown. Finally, $S_p$ is defined as

$$S_p^2 = \frac{(n_1 - 1)S_1^2 + (n_2 - 1)S_2^2}{n_1 + n_2 - 1}. \tag{4.23}$$

We also considered stability that we related to the variance of the output; the lower the variance the more stable the solution method. The comparison scheme

for the variance was the same as that of the means. We compared the variance of the system optimized by either pattern search of response surface search to the system optimized by a GA. The F-test uses the following formula:

$$F = \frac{S_1^2}{S_2^2}, \tag{4.24}$$

where $S_1^2$ and $S_2^2$ are the variances of samples 1 and 2, respectively and $\nu_1$ and $\nu_2$ the first and second degrees of freedom are given by

$$\nu_1 = n_1 - 1,$$

and (4.25)

$$\nu_2 = n_2 - 1$$

with $n_1$ and $n_2$ are the sample sizes we defined before.

We will simply give the $p$-value. The $p$-value represents the smallest level of significance that would lead to the rejection of the null hypothesis. The null hypothesis is there is no difference between the two search techniques, the alternative hypothesis being the genetic search technique gives a lower value for the response.

The programs' initial code such as Dennis Smith's response surface method and the three original simulation examples were available in FORTRAN. All other code

was also available in FORTRAN, but we translated it using Cobalt Blue's FORTRAN to C translator (FOR_C) to ANSI C. We did this for portability reasons; we tested this code on several platforms.

We ran the pattern search's simulation optimizations and the genetic search's simulation on a SUNSPARC workstation rated at 90 million instructions per second (MIPS). We ran the response surface search's simulation optimizations on an IBM RS/6000 rated at 35 MIPS. The MIPS ratings were different, but owing to a very sophisticated code optimization package on the RS/6000 the execution time of a given program was about the same on both systems.

In all the figures to follow we show the points of interest with $x$'s, be they initial or final points; they will number fifty on each figure. Also, on each figure we show the calculated optimum for the system of interest as an empty square. Note also on Figures 9 to 48 in Appendix 1 there will not necessarily be 50 $x$'s on the graph since some of the points will plot on top of each other. The reason is straight forward: we are plotting integers (in these formulas) in a small sample space (1 to 15 on both the $x$ and the $y$ axis); thus in the plot of these 50 points it is highly probable that some points will be the same and thus plot on top of each other.

For the initial points of the pattern and response surface search we generated the points by a uniform random number generator. The endpoints of the uniform distribution were the range minus the step size on the upper end and the range plus the step size on the lower end. For example, if the range was 10 to 200 with an initial step size of 2, then we generated a $U(12,\ 198)$ distribution.

We setup the initial points this way to be consistent as possible across all search methodologies. The response surface method as outlined by the code of Dennis Smith would not work if the search began at the constraint. Given a that constraint was at $x = 200$ and an initial step size of 2. The code would not work if the search began at say 200 since a step in the positive direction would give a value of 202 and violate the constraint. The initial points were identical for the pattern search and the response surface search. The genetic search's initial points were different since they were generated randomly in each generation of the search.

The response surface search has a limit of 30 trials (with three replications per trial) in an algorithm's run. We rarely reached this limit in practice since the indicated optimum was reached or the algorithm determined that making additional runs would be pointless in providing a better search.

## 4.3. Example 1 - Results

The first example was the inventory system with two decision variables $s$, and $S$. The reorder point and the order up to point, respectively. We gave both decision variables the same range of values, a low of 10 and a high of 200, with the small $s$ value always smaller than the big $S$ value. Both decision variables are only integers, but we used formula (3.15) in the genetic search because of the ranges involved. We used a rounding subroutine to round off to the nearest integer. For the pattern search and the response surface search we generated from a $U(12, 198)$ distribution by a computer program for the fifty initial points.

### 4.3.1. Example 1 - Pattern Search

We now show the initial plot in Figure 1 in Appendix 1. The pattern search used an initial step size of 2, an expansion factor of 2, a lower boundary of 10 and an upper boundary of 200 for both variables. The plot is small $s$ versus big $S$ with the approximated optimum at $s = 25$ and $S = 65$. We show the plot of the fifty simulation optimization results realized after the pattern search in Figure 2 in Appendix 1.

### 4.3.2. Example 1 - Response Surface Search

We now turn to the method of response surface search outlined by the computer code of Dennis Smith for example 1. The response surface method uses more mathematical information than the pattern search method. We would expect a better search. The initial points are identical to those in Figure 1 so there is no need to display them again. We now show the plot after the response surface search in Figure 3 in Appendix 1. The step size for the response surface is 2 for both variables; the following constraints apply:

$$
\begin{gathered}
10 \leq s \leq 200, \\
10 \leq S \leq 200, \\
s \leq S.
\end{gathered}
\tag{4.26}
$$

We used this last constraint to always keep the small $s$ value smaller than the big $S$ value.

### 4.3.3. Example 1 - Genetic Search

We now test the genetic search method on example 1. Restating the methodology we used fifty searches, with each search consisting of a simple genetic search and each generation having a population of 30 individuals. We used seven generations with each individual in a generation replicated three times.

The inputs for the decision variables used formula (3.15) with $U_{min}$ and $U_{max}$ of 10 and 200, respectively. Each chromosome was 30 units long and each variable made up 15 units of that string, thus $l$ equaled 15 in (3.15).

The run itself took about 45 minutes. The data show a very tight pattern close to the calculated optimum. This is also the calculated optimum reported by Law and Kelton (1982, p. 379). This is clearly an improvement over the output for the pattern search and the response surface. We now show the plot of the final points and the calculated optimum of the inventory system in Figure 4 in Appendix 1.

## 4.4. Example 2 - Results

The second example, the university computer system had two decision variables: the first is the quantum or the amount of time that the CPU spends on a job before sending it back to the queue, and the second is the overhead or the amount

## Table 1-Inventory System Pattern Search Calculated Values

| Type of Response | Average | Variance | 95% Confidence Upper Limit | 95% Confidence Lower Limit |
|---|---|---|---|---|
| Optimum Response | 129.91 | 544.48 | 135.45 | 124.38 |
| Distance Traversed | 30.46 | 3005.43 | NA | NA |
| Distance from Initial Points to Optimum | 86.09 | 2,479.59 | NA | NA |
| Distance from Final Points to Optimum | 59.76 | 1,985.00 | NA | NA |
| Runs per Search | 56.04 | 739.26 | 62.48 | 50.00 |

## Table 2-Inventory System Response Surface Search Calculated Values

| Type of Response | Average | Variance | 95% Confidence Upper Limit | 95% Confidence Lower Limit |
|---|---|---|---|---|
| Optimum Response | 144.22 | 989.17 | 151.68 | 136.77 |
| Distance Traversed | 9.06 | 837.83 | NA | NA |
| Distance from Initial Points to Optimum | 86.09 | 2,479.59 | NA | NA |
| Distance from Final Points to Optimum | 80.32 | 2,270.68 | NA | NA |
| Runs per Search | 20.22 | 188.38 | 23.47 | 16.97 |

# Table 3-Inventory System Genetic Search Calculated Values

| Type of Response | Average | Variance | 95% Confidence Upper Limit | 95% Confidence Lower Limit |
|---|---|---|---|---|
| Optimum Response | 106.73 | 2.40 | 107.10 | 106.37 |
| Distance Traversed | NA | NA | NA | NA |
| Distance from Initial Points to Optimum | NA | NA | NA | NA |
| Distance from Final Points to Optimum | 9.88 | 18.83 | NA | NA |
| Runs per Search | 414 | 34,053.06 | 457.74 | 370.26 |

of time it takes for the computer to do this routing. The quantum variable has a low of 0.01 and a high of 1.0, and the overhead has a low of 0.001 and a high of 0.01. We generated the quantum variates by $U(0.14,\ 0.96)$ distribution and the overhead variates by $U(0.0014,\ 0.096)$ distribution both by a computer program to obtain the fifty initial points. The decision variables were real so we used formula (3.15) when performing the genetic search.

## 4.4.1. Example 2 - Pattern Search

The pattern search for the university computer system used an initial step size of 0.04 for the quantum and 0.004 for the overhead, an expansion factor of 2 for both decision variables, a lower boundary of 0.01 and an upper boundary of 1.0 for the quantum variable and an upper boundary of 0.01 and a lower boundary of 0.001 for the overhead decision variable. The minimum step size for the quantum variable is 0.02, the minimum step size of 0.0002 for the overhead. We show the initial fifty points in Figure 5 in Appendix 1. The plot in Figure 6, also in Appendix 1 is of quantum versus overhead with an indicated optimum of quantum $= 0.5$ and overhead $= 0.005$. We show the plot of the fifty simulation optimization results for the pattern search and the optimum in Figure 6 in Appendix 1.

## 4.4.2. Example 2 - Response Surface Search

We again turn to the method of response surface search as outlined by the computer code of Dennis Smith with example 2. The initial points are identical to those in Figure 5. The step size is 0.02 for the quantum decisions variable and

0.0002 for the overhead decision variable. The following four constraints also apply:

$$0.01 \leq \text{quantum} \leq 1.0,$$
$$0.001 \leq \text{overhead} \leq 0.01. \tag{4.27}$$

We now show the plot of the fifty final points after the response surface search in Figure 7 in Appendix 1.

### 4.4.3. Example 2 - Genetic Search

We now test the genetic search method on example 2. The methodology to restate was fifty searches, with each search consisting of a simple genetic search having a population of 30 individuals per generation. We have seven generations with each individual in a generation replicated three times.

We use formula (3.15) again since we are dealing with real variables not integers, with $U_{min}$ and $U_{max}$ equal to 0.01 and 1.0, respectively for the quantum decision variable and $U_{min}$ and $U_{max}$ equal to 0.001 and 0.01, respectively for the overhead decision variable. The chromosome was 30 units long with each variable taking up 15 units. The value of $l$ was 15.

The run itself took about 23.5 hours. We now show the plot of the fifty final points and the calculated optimum in Figure 8 in Appendix 1. The data show a very tight pattern very close to the calculated optimum. Again this is clearly an improvement

over the output for the pattern search and the response surface.

## 4.5. Example 3 - Results

The third example was the job shop with five decision variables. They were the number of machines at each of the five workstations. We gave all decision variables a range from 3 to 13; all decision variables are integers. We generated $U(3, 13)$ distribution by a computer program for the fifty initial points for the pattern search.

### 4.5.1. Example 3 - Pattern Search

The pattern search used an initial step size of 2, a minimum step of 2, an expansion factor of 2, a lower boundary of 1 and an upper boundary of 15 for all five variables. We determined the approximated optimum to be $M_1 = 9$, $M_2 = 5$, $M_3 = 8$, $M_4 = 6$, $\mathrm{M}_5 = 3$.

We now show the plot of the job-shop fifty initial points. There are five decision variables. It is impossible to plot them on a single graph. All data for the job shop will be in the format of ten figures with two decision variables each. That should provide information similar to the previous figures.

We now show the initial points for the job-shop in Figures 9-18 in Appendix 1. The final 50 points and the calculated optimum for the pattern search we now show in Figures 19-28 in Appendix 1.

# Table 4-Computer System Pattern Search Calculated Values

| Type of Response | Average | Variance | 95% Confidence Upper Limit | 95% Confidence Lower Limit |
|---|---|---|---|---|
| Optimum Response | 51,497.53 | 1,394,843 | 51,777.47 | 51,217.60 |
| Distance Traversed | 0.160 | 0.02 | NA | NA |
| Distance from Initial Points to Optimum | 0.21 | 0.02 | NA | NA |
| Distance from Final Points to Optimum | 0.09 | 0.01 | NA | NA |
| Runs per Search | 49.38 | 453.02 | 54.43 | 44.34 |

# Table 5-Computer System Response Surface Search Calculated Values

| Type of Response | Average | Variance | 95% Confidence Upper Limit | 95% Confidence Lower Limit |
|---|---|---|---|---|
| Optimum Response | 17,699.5 | 2,838,000 | 18,098.78 | 17,300.22 |
| Distance Traversed | 0.16 | 0.03 | NA | NA |
| Distance from Initial Points to Optimum | 0.21 | 0.02 | NA | NA |
| Distance from Final Points to Optimum | 0.09 | 0.01 | NA | NA |
| Runs per Search | 40.08 | 630.65 | 46.03 | 34.13 |

## Table 6-Computer System Genetic Search Calculated Values

| Type of Response | Average | Variance | 95% Confidence Upper Limit | 95% Confidence Lower Limit |
|---|---|---|---|---|
| Optimum Response | 16,150.66 | 133,220.98 | 16,237.18 | 16,064.15 |
| Distance Traversed | NA | NA | NA | NA |
| Distance from Initial Points to Optimum | NA | NA | NA | NA |
| Distance from Final Points to Optimum | 0.05 | 0.01 | NA | NA |
| Runs per Search | 365.4 | 39,147.79 | 412.30 | 318.50 |

## 4.5.2. Example 3 - Response Surface

We again turn to the response surface method of Dennis Smith for example 3. The initial points are identical to those in the pattern search so we will not display them again. The following constraints will apply:

$$1 \le M_i \le 15$$
$$\sum_{i=1}^{5} M_i \le 75. \tag{4.28}$$

where $i = 1$, 2, 3, 4 and 5. We now show the final points for the response surface search for example three in Figures 29-38 in Appendix 1.

## 4.5.3. Example 3 - Genetic Search

We now test the genetic search method on example 3. The methodology to restate was fifty searches, with each search consisting of a simple genetic search with each having a population of 30 individuals. Seven generations with each individual in a generation replicated three times. Each chromosome was 30 units long with each of the five variables 4 units long. That made a low of 1 and a high of 15. If the string by chance came out to be zero a possibility since we a dealing with integers, we increased it by one.

The run itself took about approximately 150 hours. The data show a very tight

pattern close to the calculated optimum. This is clearly an improvement over the output for the pattern search and the response surface.

We now show the fifty final points and the calculated optimum for the job-shop in Figures 39-48 in Appendix 1.

## Table 7-Job-Shop Pattern Search Calculated Values

| Type of Response | Average | Variance | 95% Confidence Upper Limit | 95% Confidence Lower Limit |
|---|---|---|---|---|
| Optimum Response | 222,436,453 | $1.927 \times 10^{16}$ | 255,335,993.60 | 189,536,913.30 |
| Distance Traversed | 0.67 | 1.82 | NA | NA |
| Distance from Initial Points to Optimum | 9.02 | 5.39 | NA | NA |
| Distance from Final Points to Optimum | 8.61 | 5.84 | NA | NA |
| Runs per Search | 41.52 | 279.32 | 45.48 | 37.56 |

## Table 8-Job-Shop Response Surface Search Calculated Values

| Type of Response | Average | Variance | 95% Confidence Upper Limit | 95% Confidence Lower Limit |
|---|---|---|---|---|
| Optimum Response | 39,721.31 | 16,998,523 | 40,698.54 | 38,744.09 |
| Distance Traversed | 5.23 | 4.00 | NA | NA |
| Distance from Initial Points to Optimum | 9.02 | 5.39 | NA | NA |
| Distance from Final Points to Optimum | 6.94 | 6.61 | NA | NA |
| Runs per Search | 65.28 | 265.06 | 69.14 | 61.42 |

## Table 9-Job-Shop Genetic Search Calculated Values

| Type of Response | Average | Variance | 95% Confidence Upper Limit | 95% Confidence Lower Limit |
|---|---|---|---|---|
| Optimum Response | 40,082.05 | 8,864,470 | 40,787.74 | 39,376.36 |
| Distance Traversed | NA | NA | NA | NA |
| Distance from Initial Points to Optimum | NA | NA | NA | NA |
| Distance from Final Points to Optimum | 6.57 | 4.82 | NA | NA |
| Runs per Search | 385.2 | 38,694.85 | 431.83 | 338.58 |

# Chapter 5. Discussion of Results

The results of the computer experiments are informative. The trend in the data is that a genetic algorithm executes a superior search considering the calculated optimum and its stability when compared to pattern and response surface search, but uses more computer time. The time taken during a search as we stated in Chapter 1 is not a critical factor in the evaluation of the algorithms.

It is true that at times another method such as the response surface search might execute a roughly equivalent search. Over all three models with both well defined and some not so well defined surfaces, the genetic algorithm search is more reliable and robust.

## 5.1. Inventory System

Consider the inventory model's output as shown in Table 10. The average cost for the genetic search was \$ 106.73 (variance = 2.40) while it was \$ 144.22 (variance = 989.17) for the response surface and \$ 129.91 (variance = 544.48) for the pattern search. The result is: the genetic search method is superior to the other two both in response and stability. The variance also results in a narrower confidence interval as shown in Tables 1, 2 and 3 for the pattern search, response

## Table 10-Inventory System Comparison of Pattern, Response Surface and Genetic Search

| Type of Response | Average | Variance | Upper 95% Conf. Limit | Lower 95% Conf. Limit |
|---|---|---|---|---|
| Optimum Response | 129.91<br>144.22<br>106.73 | 544.48<br>989.17<br>2.40 | 135.45<br>151.68<br>107.10 | 124.38<br>136.77<br>106.37 |
| Distance Traversed | 30.46<br>9.06<br>NA | 3005.43<br>837.83<br>NA | NA<br>NA<br>NA | NA<br>NA<br>NA |
| Distance from Initial Points to Optimum | 86.09<br>86.09<br>NA | 2,479.59<br>2,479.59<br>NA | NA<br>NA<br>NA | NA<br>NA<br>NA |
| Distance from Final Points to Optimum | 59.76<br>80.32<br>9.88 | 1,985.01<br>2,270.68<br>18.83 | NA<br>NA<br>NA | NA<br>NA<br>NA |
| Runs per Search | 56.04<br>20.22<br>414 | 739.26<br>188.38<br>34,053.06 | 62.48<br>23.47<br>457.74 | 49.60<br>16.96<br>370.26 |

surface search and genetic search, respectively. Finally, the average distance and variance from the final points to the approximated optimum is smaller at 9.88 (variance = 18.83). Compared to 59.76, (variance = 1,985.00) and 80.32, (variance = 2,270.68) for the pattern search and response surface search, respectively, the genetic search is superior in value and stability.

The average runs per search is much larger for the genetic search at 414 (variance = 34,053.06) than for the other two search methods. This compares with the average number of runs for the pattern search of 56.04 (variance = 739.26) and for the response surface of 20.22 (variance = 188.38). The larger number of runs seems to be a small tradeoff for a superior search.

## 5.2. University Computer Time-Sharing System

Now consider the computer time-sharing model's output in Table 11. The average cost for the genetic search was $ 16,150.66 (variance = 133,220.98) while it was $ 17,699.5 (variance = 2,838,000) for the response surface and $ 51,497.53 (variance = 1,394,843) for the pattern search. Again the result is: the genetic search method is superior to the other two both in response (lower operating costs) and stability (lower variance). The variance also results in a narrower confidence interval as shown in Tables 4, 5 and 6 for the pattern search, response surface search and genetic search, respectively. Finally, the average distance from the final points to the approximated optimum is smaller at 0.05 (variance = 0.01). Compared to 0.09, (variance = 0.01) and 0.09 (variance = 0.01) for the pattern search and response surface search, respectively.

## Table 11-Computer System Comparison of Pattern, Response Surface and Genetic Search

| Type of Response | Average | Variance | Upper 95% Conf. Limit | Lower 95% Conf. Limit |
|---|---|---|---|---|
| Optimum Response | 51,497.53<br>17,699.50<br>16,150.66 | 1,394,843.85<br>2,837,811.31<br>133,220.98 | 51,777.47<br>18,098.78<br>16,237.18 | 51,217.60<br>17,300.22<br>16,064.15 |
| Distance Traversed | 0.16<br>0.16<br>NA | 0.02<br>0.03<br>NA | NA<br>NA<br>NA | NA<br>NA<br>NA |
| Distance from Initial Points to Optimum | 0.21<br>0.21<br>NA | 0.02<br>0.02<br>NA | NA<br>NA<br>NA | NA<br>NA<br>NA |
| Distance from Final Points to Optimum | 0.09<br>0.09<br>0.05 | 0.01<br>0.01<br>0.01 | NA<br>NA<br>NA | NA<br>NA<br>NA |
| Runs per Search | 49.38<br>40.08<br>365.40 | 453.02<br>630.65<br>39,147.80 | 54.43<br>46.03<br>412.30 | 44.34<br>34.13<br>318.50 |

The average runs per search is much larger for the genetic search at 365.4 (variance = 39,147.79). This compares with the pattern search of 49.38 (variance = 453.02) and for the response surface of 40.08 (variance = 630.65). Again we would conclude that the larger number of runs seems to be a small tradeoff for a superior search.

## 5.3. Job-Shop

Consider the job-shop model's output in Table 12. The average cost determined by the genetic search was $ 40,082.05 (variance = 8,864,470), while the response surface search determined it was $ 39,721.31 (variance = 16,998,523) and the pattern search determined it was $ 222,436,453 (variance = 1.927 x $10^{16}$). The result is that the genetic search method is superior compared to the pattern search both in response and stability and was equal to the response surface search in response and superior to it in stability. The variance also resulted in narrower confidence intervals as shown in Tables 7, 8, and 9. The confidence intervals for the pattern search are very wide while for the response surface search it is narrower and for the genetic search it is the narrowest. Finally, for the genetic search the average distance and variance from the final points to the approximated optimum is smaller at 6.57 (variance = 4.82) which compared to 8.61, (variance = 5.84) and 6.94, (variance = 6.61) for the pattern search and response surface search, respectively. Again, the genetic search's calculated optimum is superior in value and stability.

The average runs per search is much larger for the genetic search at 385.2 (variance = 38,694.85). This compares with the pattern search of 41.52

## Table 12-Job-Shop Comparison of Pattern and Response Surface Search Methods to Genetic Search Method

| Type of Response | Average | Variance | Upper 95% Conf. Limit | Lower 95% Conf. Limit |
|---|---|---|---|---|
| Optimum Response | 222,436,453<br>39,721.31<br>40,082.05 | $1.927 \times 10^{16}$<br>16,998,523<br>8,864,470 | 255,335,993.61<br>40,698.54<br>40,787.74 | 189,536,913.30<br>38,744.09<br>39,376.36 |
| Distance Traversed | 0.67<br>5.23<br>NA | 1.82<br>4.00<br>NA | NA<br>NA<br>NA | NA<br>NA<br>NA |
| Distance from Initial Points to Optimum | 9.02<br>9.02<br>NA | 5.39<br>5.39<br>NA | NA<br>NA<br>NA | NA<br>NA<br>NA |
| Distance from Final Points to Optimum | 8.61<br>6.94<br>6.57 | 5.84<br>6.61<br>4.82 | NA<br>NA<br>NA | NA<br>NA<br>NA |
| Runs per Search | 41.52<br>65.28<br>385.20 | 279.32<br>265.06<br>38,694.85 | 45.48<br>69.14<br>413.83 | 37.56<br>61.42<br>338.58 |

(variance = 279.32) and for the response surface of 65.28 (variance = 265.06). Again the larger number of runs seems to be a small tradeoff for a superior search.

## 5.4. Statistical Tests

We now discuss the statistical comparisons of the means and variances. As we stated above we used a $t$-test for equal, but unknown variances to determine if there was a statistically significant difference between the means of either the pattern search or the response surface search. We also compared the stability or variance of the response of the genetic search compared to the pattern search or the response surface search. In Table 13 we show the results of those tests.

The difference for the inventory system optimized using the pattern search versus the genetic search is statistically significant as it is also for the response surface search versus the genetic search. In both comparisons the $p$-value was essentially zero thus the level of significance was zero.

The difference for the computer system optimized using the pattern search versus the genetic search is statistically significant and it is also for the response surface search versus the genetic search. Again, in both comparisons the $p$-value was essentially zero thus the level of significance was zero.

When we examine the job-shop is when we notice a change in the trend. While it is true that the difference in the pattern search versus the genetic search is significant at the zero level, there is no significant difference (at any low level) between the

## Table 13-Statistical Tests for Pattern Search and Response Surface Methods Compared to Genetic Search Method

| Comparison Scheme | $t_1$ | p value | $F_2$ | p value |
|---|---|---|---|---|
| Inv. Sys. Pattern vs. Genetic Search | 7.01 | 0.00 | 227.27 | 0.00 |
| Inv. Sys. Response vs. Genetic Search | 8.41 | 0.00 | 413.00 | 0.00 |
| Comp. Sys. Pattern vs. Genetic Search | 202.19 | 0.00 | 10.47 | 0.00 |
| Comp. Sys. Response vs. Genetic Search | 6.35 | 0.00 | 21.30 | 0.00 |
| Job-Shop Pattern vs. Genetic Search | 11.33 | 0.00 | $2.2 \times 10^9$ | 0.00 |
| Job-Shop Response vs. Genetic Search | 5.02 | 0.62 | 1.92 | 0.02 |

1. Difference between means

2. Difference between variances

response surface and the genetic search response for the job-shop. Now we turn to discuss stability. In that case the genetic search is superior in every case. The only time the $p$-value is not zero is in the case of the job-shop response surface method versus the genetic search. In that case the $p$-value is only 0.02. The genetic searches show more stability in all comparisons.

Now we consider the 95% confidence interval half-length about the response of each of the two traditional searches as it is compared to the genetic search. We inserted the values into a ratio with the genetic search confidence interval's half-length in the numerator (since it was the smallest) to see the comparison clearly. We show the results in Table 14.

It should be clear that the genetic search produces confidence intervals in all three examples with the smallest ratios for the inventory system (genetic search/pattern search, genetic search/response surface search) and the job-shop (genetic search/pattern search). The largest ratio is the job-shop (genetic search/response surface search). None of these ratios is greater than one; the genetic search is always produces a confidence interval less than one.

The comparison of the genetic search to the pattern search for the inventory system showed the genetic search confidence's interval to be about 4.6% of the pattern search's confidence interval and 4.9% of the response surface's confidence interval.

The comparison of the genetic search to the pattern search for the computer

## Table 14-Ratio of the Confidence Interval Half-Lengths for Pattern and Response Surface Search Methods to Genetic Search Method

| Comparison Scheme | Ratio | Numerical Value |
|---|---|---|
| Inventory System: Genetic Search/ Pattern Search | $\frac{0.367}{5.531}$ | 0.046 |
| Genetic Search/ Response Search | $\frac{0.367}{7.455}$ | 0.049 |
| Computer System: Genetic Search/ Pattern Search | $\frac{86.51}{279.93}$ | 0.103 |
| Genetic Search/ Response Search | $\frac{86.51}{399.28}$ | 0.217 |
| Job-Shop: Genetic Search/ Pattern Search | $\frac{705.69}{32,899,540.15}$ | 0.00 |
| Genetic Search/ Response Search | $\frac{705.69}{977.69}$ | 0.722 |

system showed the genetic search confidence's interval to be about 10% of the pattern search's confidence interval and 22% of the response surface's confidence interval.

In the case of the job-shop the genetic search's confidence interval half-length compared to the pattern search's half-length is essentially zero. Only in the last case do we see a change with the genetic search's confidence interval half-length being about 72% of the response surface half-length.

## 5.5. Runs Data

Clearly, the genetic search uses the most runs compared to the pattern search and the response surface search. Its average number of runs also has the highest variance. The pattern search being a simple search uses the smallest number of runs, followed by the response surface search and the genetic search. We see this order again followed when comparing the variances. The response surface search uses more mathematical information than the pattern so it requires on the average more runs and the variance of that average is greater.

The genetic search requires on the average the most runs and the variance of that average is also greater. This is because the search is probabilistic with no guarantee that as the search progresses from one generation to the next that the optimum will keep improving. The best response may come in an early generation or a late generation. This is a function of the way a genetic search works. We explained earlier in section 2.10 Alternative Search Methods that the way to prevent this is

to modify the genetic algorithm so that it employs an elitist strategy always transfers from the earlier generation to the latter generation the best member of the earlier generation. This will prevent the sliding back of the search during run. We sought to keep this all very simple for the study and avoided the complications that an elitist model would bring to the study. Hence, we must select the best response from the populations of each generation and there is no guarantee that the latest generation would have the lowest response (remember that we a minimizing in all examples of our test bed). We ran the genetic search a specific number of generations and then selected the optimum, not until it did not improve on a selected parameter anymore.

## 5.6. Summary

The comparison of the response surface search and the genetic algorithm search show that the response surface compares quite well with the genetic algorithm in accuracy and of course is superior in speed in the job-shop and is superior in speed and stability for the inventory system and the university time-sharing computer system. This could be just a fortuitous set of events for the response surface algorithm. It might also be a poor choice of parameters for the genetic algorithm. It is quite reasonable that another set of parameters (population size, crossover rate, generations, and mutation rate, etc.) could lead to a better outcome for the genetic algorithm.

The response function values were not much different for the two algorithm search methods, but the graphic plots of both algorithms showed a more concentrated area

around the approximated optimum in the response surface output than in the pattern search output.

The genetic search method always gives a consistent output where the others seemed to work well only in particular circumstances.

# Chapter 6. Future Research

Future research should focus on reducing the time it takes to execute a genetic search. The genetic search produces a superior search as shown earlier in this chapter and in Chapter 5, but the cost is in a longer search time. This is an acceptable tradeoff, but with the advent of micro-genetic algorithms this goal of reducing the search time is a goal that should be seriously considered.

Micro-genetic algorithms attain the optimum much faster, by taking advantage of preliminary convergence discussed in Chapter 3. They work with much smaller populations giving a shorter search time with almost no loss in accuracy. However, research in this area is still very new and there is not much of it available.

The second area to conduct research is in the area of evaluating the simulation model at the approximated optimum. That is once we have determined that the approximated optimum for a simulation model is at a specific point then we test the model itself at those points and in the area close to those points. This means we only use to GA to find an area of potential. Upon reaching this area we may continue to use GA's or resort to another search technique such as response surface to find the optimum.

# References

Banks, J. and J. S. Carson, II. 1984. *Discrete-Event System Simulation.* Prentice-Hall, Englewood Cliffs, NJ.

Barricelli, N. A. 1962. Numerical Testing of Evolution Theories. *ACTA Biotheoretica* **16**, 69-126.

Barricelli, N. A. 1957. Symbiogenetic Evolution Processes Realized by Artificial Methods. *Methodos* **9** No. 35-36, 143-182.

Birtwhistle, G. M. O., O. J. Dahl, B. Myhrhaug, and K. Nygaard. 1973. *SIMULA Begin.* Auerbach, Philadelphia.

Bledsoe, W. W. 1961. *The Use of Biological Concepts in the Analytical Study of Systems.* ORSA-TIMS National Meeting San Francisco.

Box, G. E. P. 1957. Evolutionary Operation: A Method for Increasing Industrial Production. *Journal of the Royal Statistical Society* C **6** No. 2, 81-101.

Bratley, P., B. Fox, and L. E. Schrage. 1987. *A Guide to Simulation.*

Springer-Verlag, New York.

Bremermann, H. J. 1962. Optimization Through Evolution and Recombination. In *Self-Organizing Systems,* eds. M. C. Yovits, G. T. Jacobi, and G. D. Goldstein 93-106. Spartan Books, Washington, D. C.

Bulgak, A. A. and J. L. Saunders. 1988. Integrating a Modified Simulated Annealing Algorithm with the Simulation of a Manufacturing System to Optimize Sizes in Automatic Assembly Systems. *Proceedings of the 1988 Winter Simulation Conference,* 684-690.

CACI, Inc., *SIMSCRIPT II.5 Reference Handbook,* Los Angeles, 1976.

Daughety, A. F. and M. A. Turnquist. 1978. Simulation Optimization Using Response Surfaces Based on Spline Approximations. *Proceedings of the 1978 Winter Simulation Conference,* 183-193.

Davis, L. 1991. *Handbook of Genetic Algorithms.* Van Nostrand Reinhold, New York.

De Groot, M. H. 1970. *Optimal Statistical Decisions.* McGraw-Hill, New York.

De Jong, K. A. 1975. *An Analysis of the Behavior of a Class of Genetic Adaptive Systems.* Ph.D. Dissertation. Department of Computer and Communication Sciences. The University of Michigan. Ann Arbor, MI.

Fogel L. J., A. J. Owens and M. J. Walsh. 1966. *Artificial Intelligence Through Simulated Evolution.* John Wiley & Sons, New York.

Fraser, A. S. 1960. Simulation of Genetic Systems by Automatic Digital Computers 5-linkage, dominance and epistasis. In *Biometrical Genetics,* ed. O. Kempthorne, 77-83. Macmillan, New York.

Friedman, G. J. 1959. Digital Simulation of an Evolutionary Process. *General Systems Yearbook* **4**, 171-184.

Goldberg, D. E. 1990. A Note on Boltzmann Tournament Selection for Genetic Algorithms and Population Oriented Simulated Annealing. *Complex Systems* **4**, 445-460.

Goldberg, D. E. 1989. *Genetic Algorithms in Search, Optimization and Machine Learning.* Addison-Wesley, Reading, MA.

Grefenstette, J. J. 1986. Optimization of Control Parameters for Genetic Algorithms. *IEEE Transactions on Systems, Man, and Cybernetics* **16** No. 1, 122-128.

Heidleberger, P, X. R. Cao, M. A. Zanzanis and R. Suri. 1988. Convergence Properties of Infinitesimal Perturbation Analysis Estimates. *Management Science* **34** no 11, 1281-1302.

Ho, Y. C. and C. Cassandras. 1981. A New Approach to the Analysis of Discrete Event Dynamic Systems. *Automatica* **19** No. 2, 149-167.

Holland, J. H. 1975. *Adaption in Natural and Artificial Systems,* The University of Michigan Press, Ann Arbor, MI.

Holland, J. H. 1962. Concerning Efficient Adaptive Systems. In *Self-Organizing Systems.* eds. M. C. Yovits, G. T. Jacobi and G. D. Goldstein, 215-230. Spartan Books, Washington, D.C..

Hooke, R. and T. A. Jeeves. 1961. Direct Search Solution of Numerical and Statistical Problems. *Journal of the Association for Computing Machinery* **8**, 212-229.

Kirkpatrick, S., C. D. Gelatt and M. P. Vecchi. 1983. Optimization by Simulated Annealing. *Science* **220**, 671-680.

Kiviat, P. J., R. Villaneuva, and H. M. Markowitz. 1973. *SIMSCRIPT II.5 Programming Language.* E. C. Russell (Ed.) CACI, Inc., Los Angeles.

Kleijnen, J. P. C. 1974. *Statistical Techniques in Simulation, Part I.*: Marcel Dekker, Inc, New York.

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

Law, A. M. and W. D. Kelton. 1982. *Simulation Modeling and Analysis.* McGraw-Hill, New York.

Mahfoud, Samir. 1991. *An Analysis of Boltzmann Tournament Selection.* Illinois Genetic Algorithms Laboratory Report No. 91007, The University Of Illinois.

Manz, E. M., J. Haddock, and J. Mittenthal. 1989. Optimization of an Automated Manufacturing System Model Using Simulated Annealing. *Proceedings of the 1989 Winter Simulation Conference,* 390-395.

Marse, K. and S. D. Roberts. 1983 Implementing a Portable FORTRAN Uniform (0,1) Generator. *Simulation* **41**, 135-139.

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

Metropolis, N, A. Rosenbluth, M. Rosenbluth, A. Teller, and E. Teller. 1953. Equation of State Calculations by Fast Computing Machines. *Journal of Chemical Physics* **21**, 1087-1092.

Myers, R. H. 1971. *Response Surface Methodology.* Allyn and Bacon, Boston.

Naylor, T. H., J. L. Balintfy, D. S. Burdick, and K. Chu. 1967. *Computer Simulation Techniques.* John Wiley & Sons, New York.

Nelder, J. A. and R. Mead. 1965. A Simplex Method for Function Optimization. *Computer Journal* **7**, 308-313.

Pegden, C. D. 1985. *Introduction to SIMAN.* Systems Modeling Corporation, State College, PA.

Pegden, C. D. and M. P. Gately. 1980. A Decision Optimization Module for SLAM. *Simulation* **34** No. 1, 18-25.

Pritsker, A. A. B. 1986. *Introduction to Simulation and SLAM.* Halsted Press, New York.

Rosenbrock, H. H. 1960. An Automatic Method for Finding the Greatest or Least Value of a Function. *Computer Journal* **3**, 175-184.

Russell, E. C. 1981. *Building Simulation Models in Simscript 11.5.* CACI, Inc., Los Angeles.

Russell, E. C. 1976. *Simulation and SIMSCRIPT II.5.* CACI, Inc., Los Angeles.

Schriber, T. J. 1974. *Simulation Using GPSS.* John Wiley & Sons, New York.

Schruben, L. W. 1986. Sequential Simulation Using Frequency Domain Methods. *Proceedings of the 1986 Winter Simulation Conference,* 366-369.

Smith, D. E. 1978. *Automated Response Surface Methodology in Digital Computer Simulation-Volume I: Program Description and User's Guide,* Desmatics, Inc, State College, PA.

Spendley, W., G. R. Hext, and F. R. Himsworth. 1962. Sequential Applications of Simplex Designs in Optimization and Evolutionary Operation. *Technometrics* **4**, 441-461.

Suri, R. 1985. An Overview of Evaluative Models for Flexible Manufacturing Systems. *Annals of Operations Research* **3**, 13-21.

Wilde, D. J. and C. S. Beightler. 1967. *Foundations of Optimization.* Prentice-Hall, Englewood Cliffs, N. J.

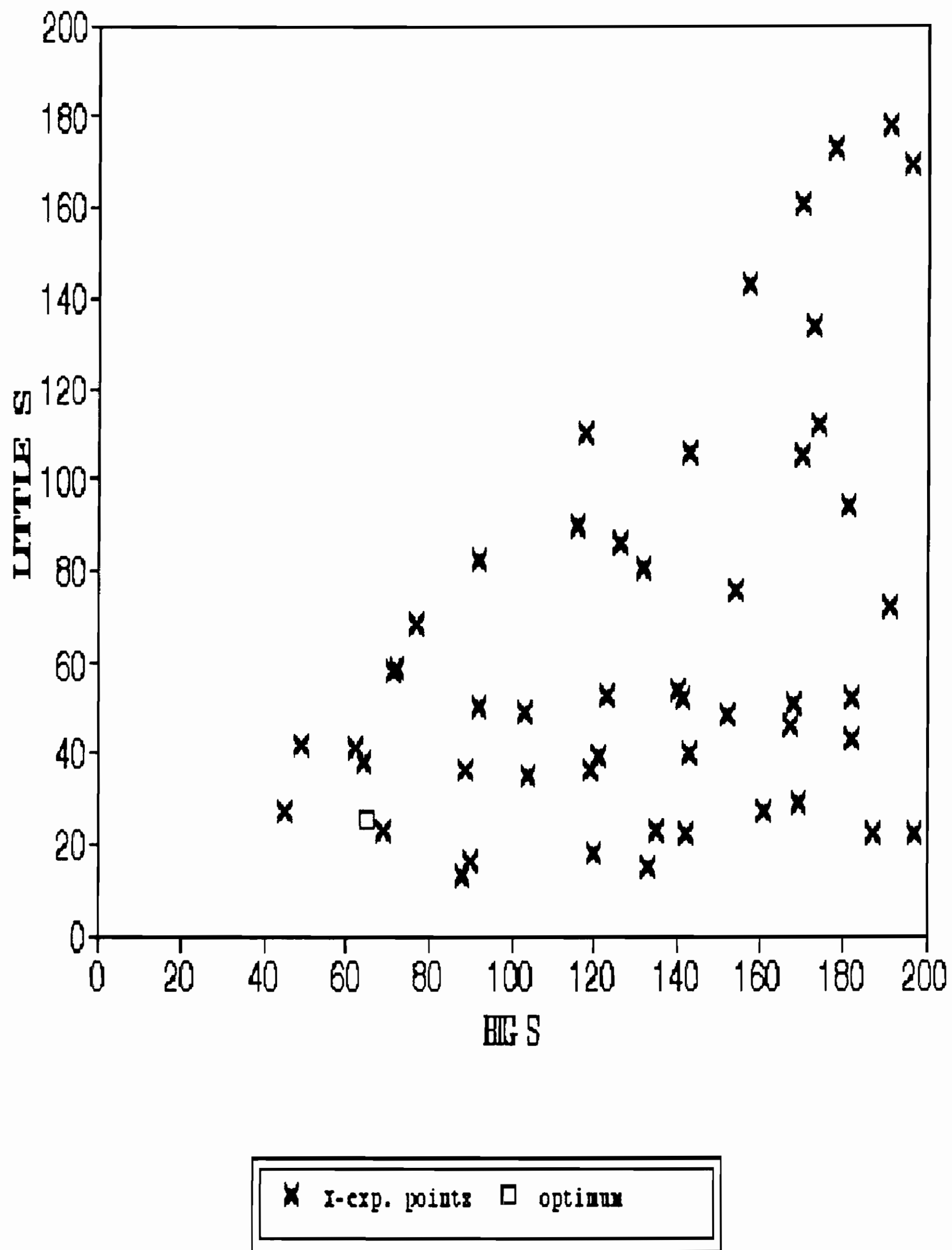



**Figure 1-Initial Points for Inventory System-Pattern Search**

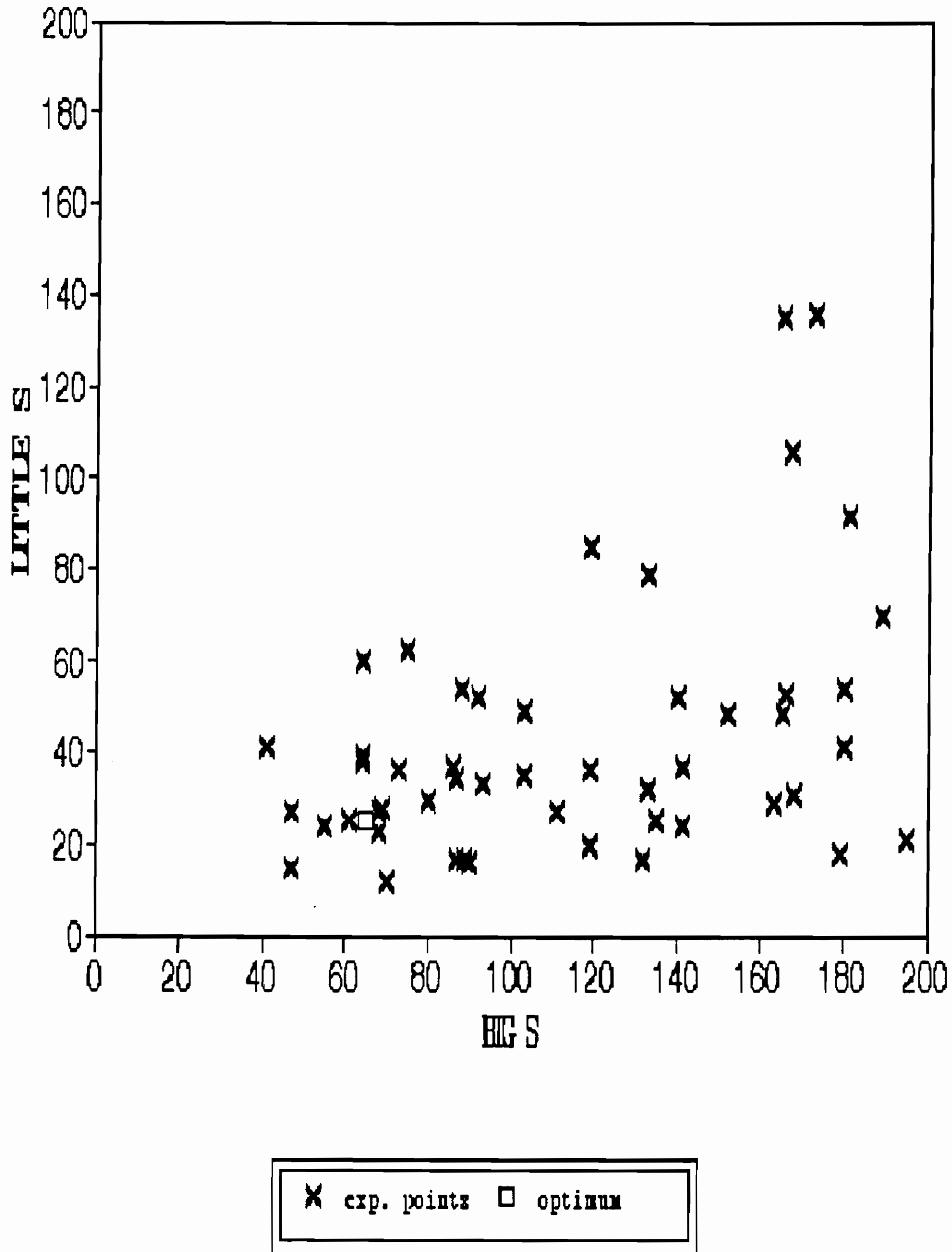



**Figure 2-Final Points for Inventory System-Pattern Search**

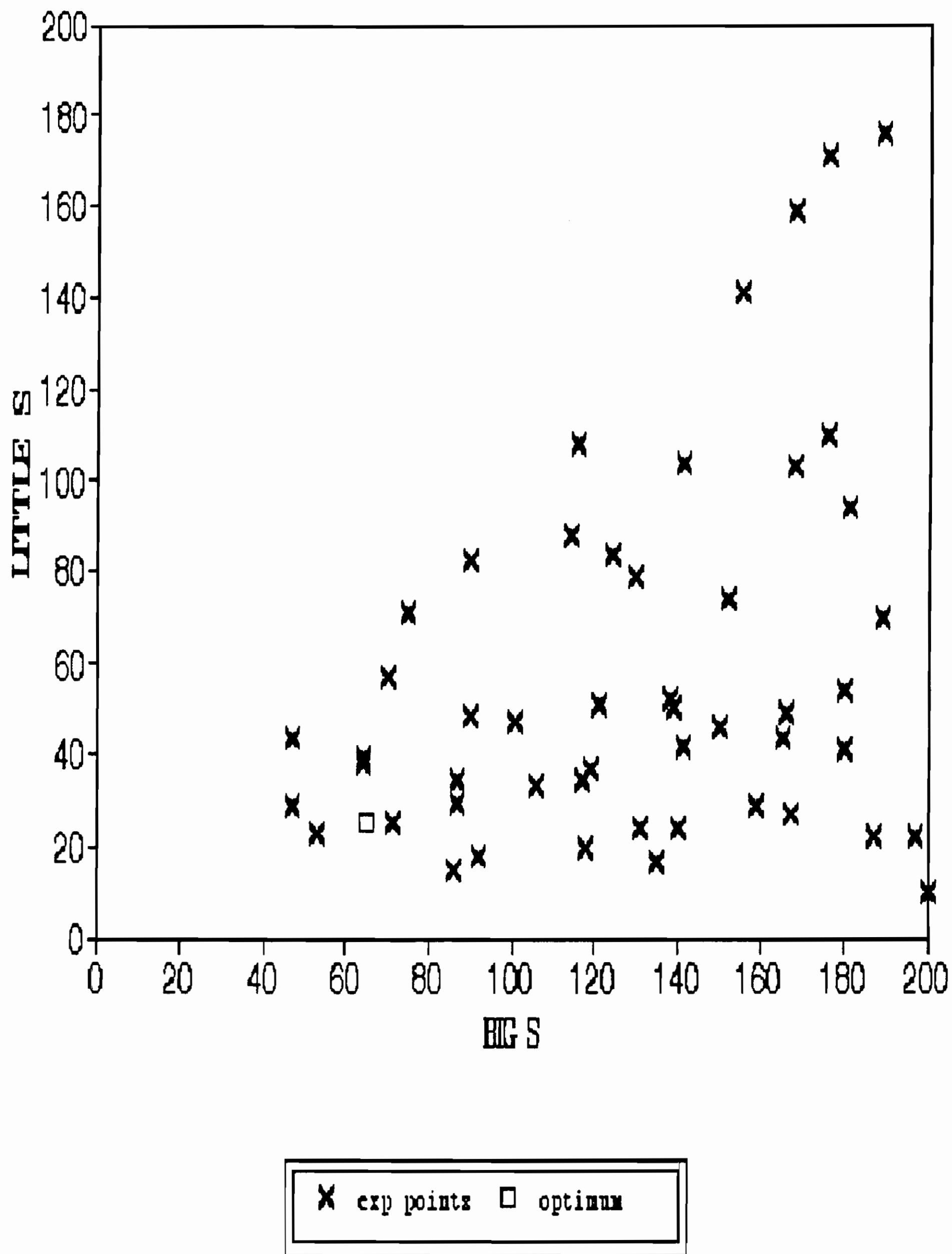



**Figure 3-Final Points for Inventory System-Response Surface Search**

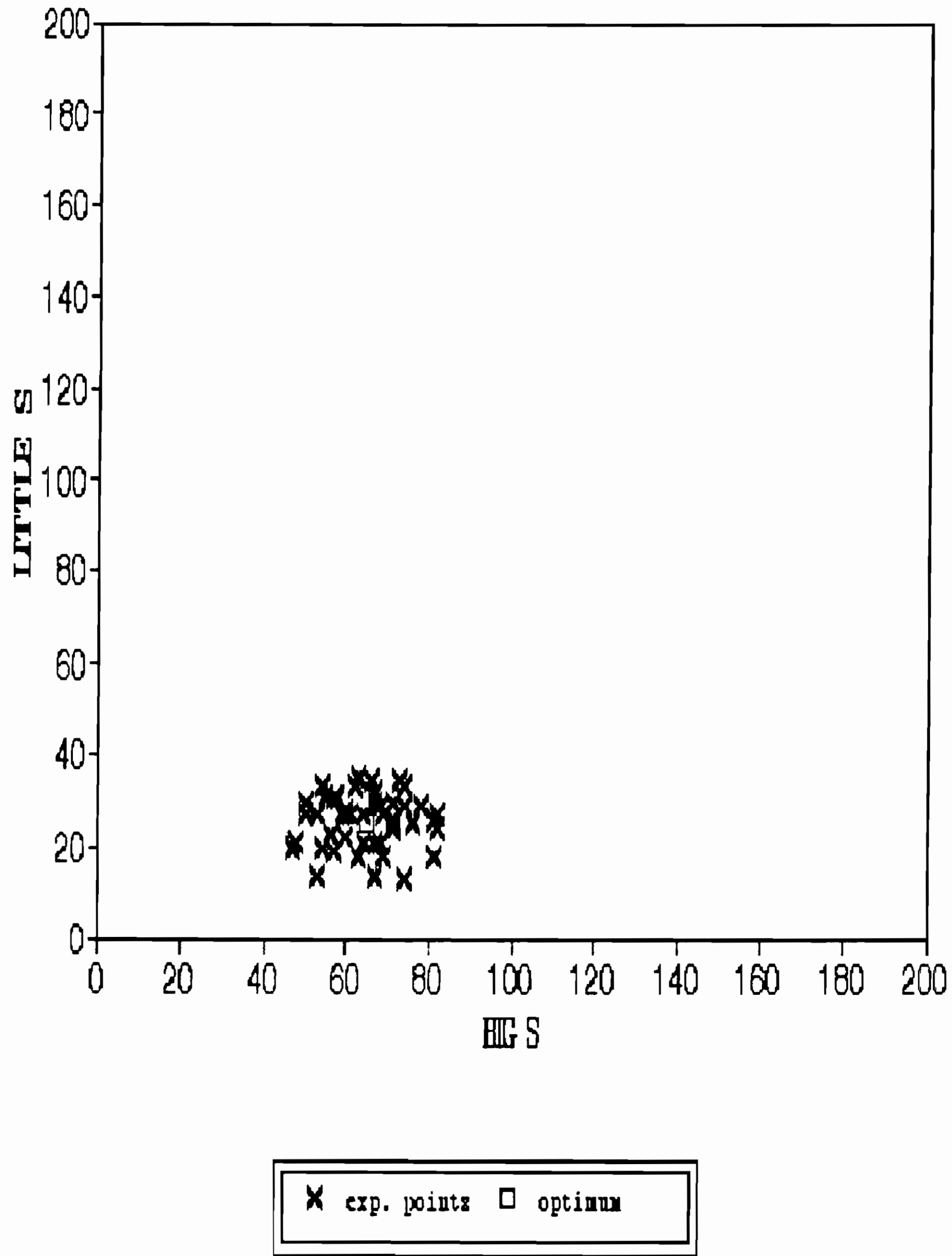



**Figure 4-Final Points for Inventory System-Genetic Search**

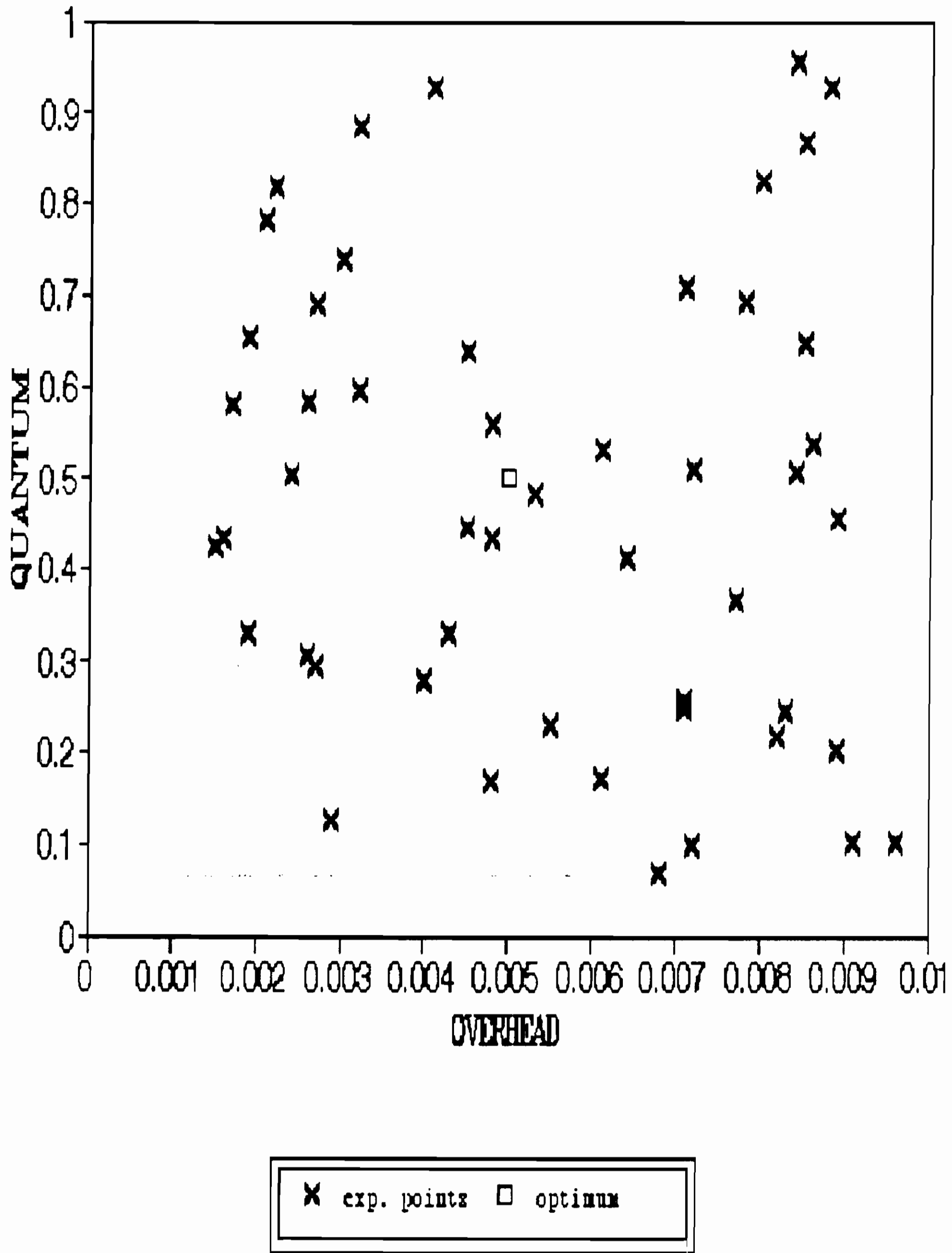



**Figure 5-Initial Points for Computer System-Pattern Search**

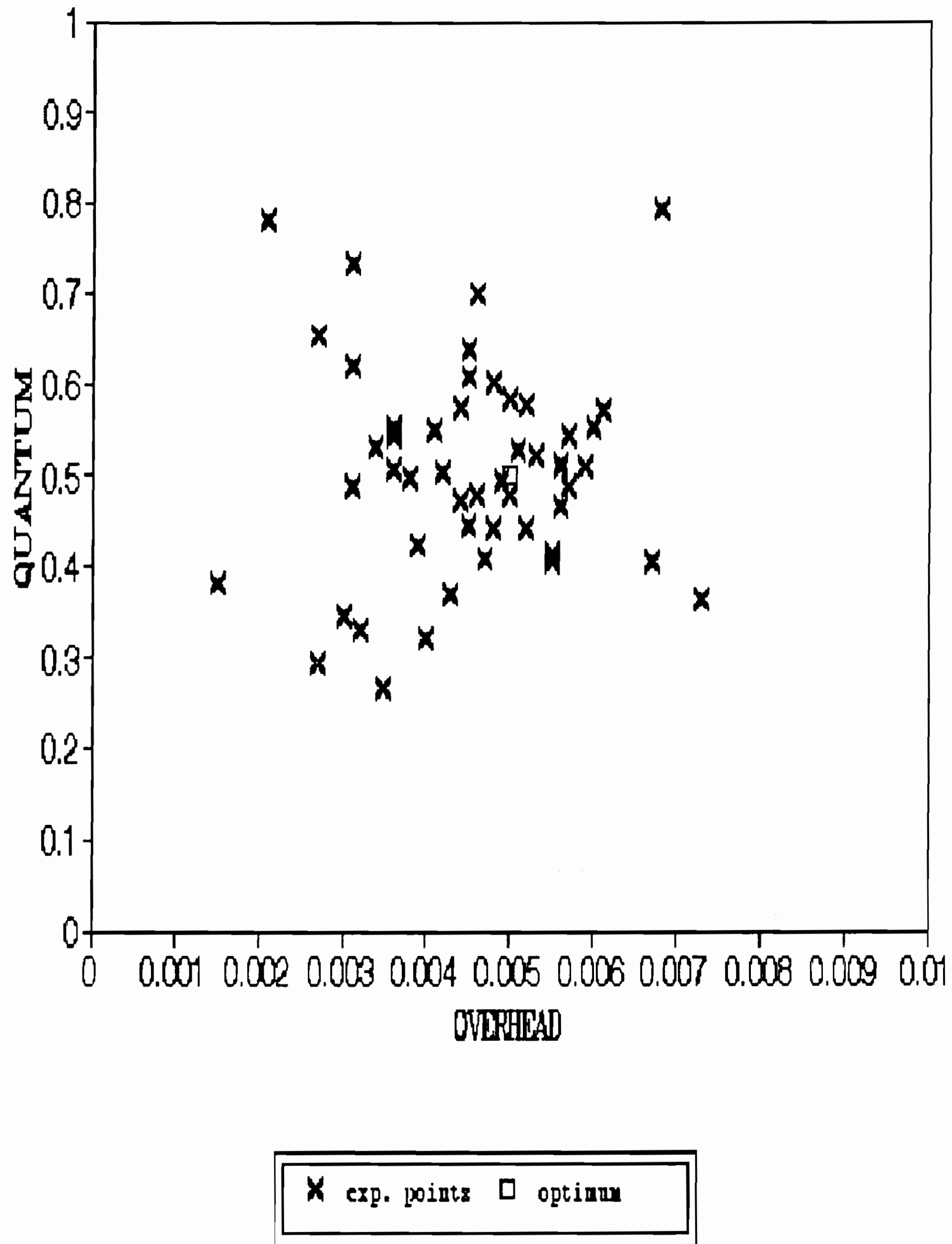



Figure 6-Final Points for Computer System-Pattern Search

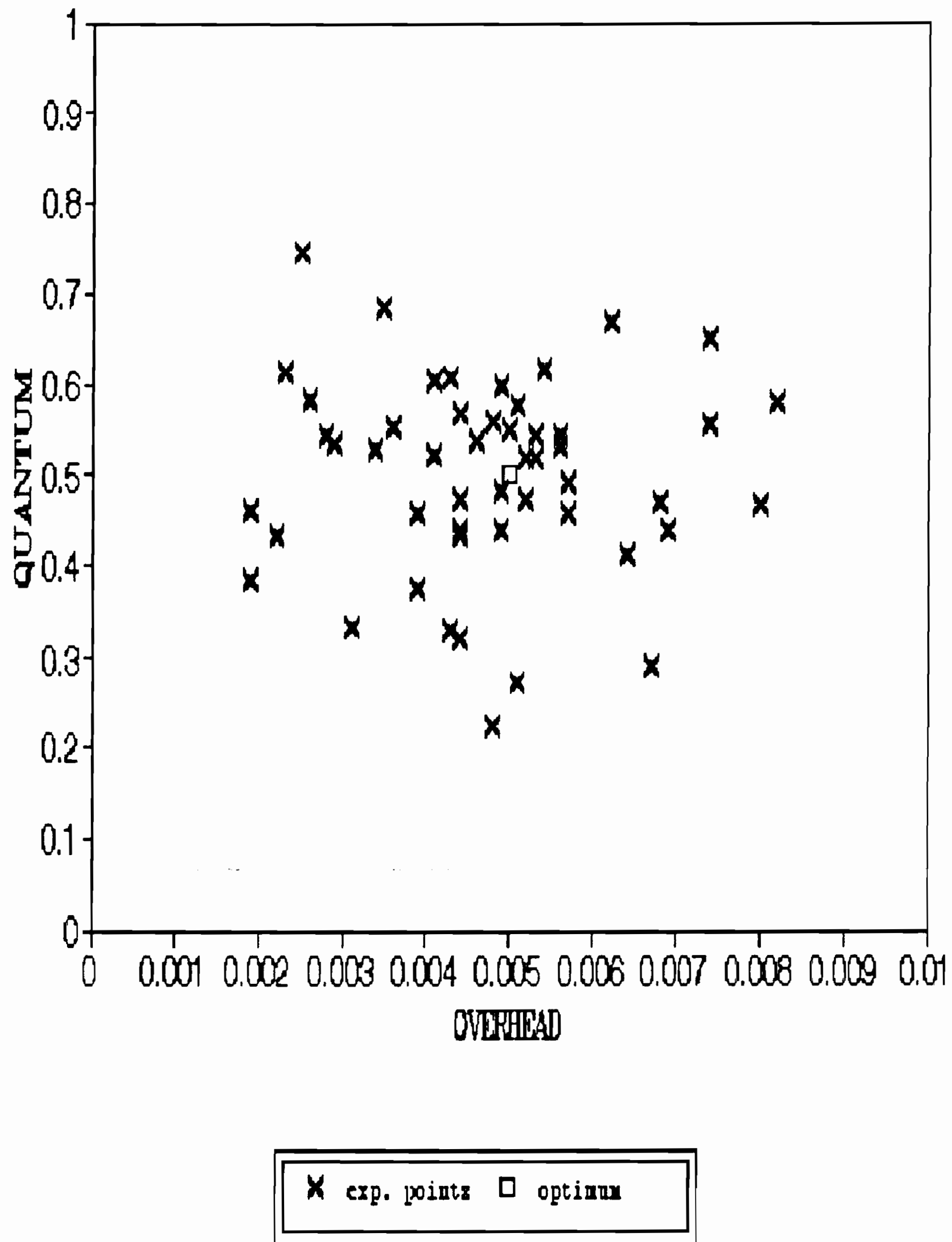



**Figure 7-Final Points for Computer System-Response Surface Search**

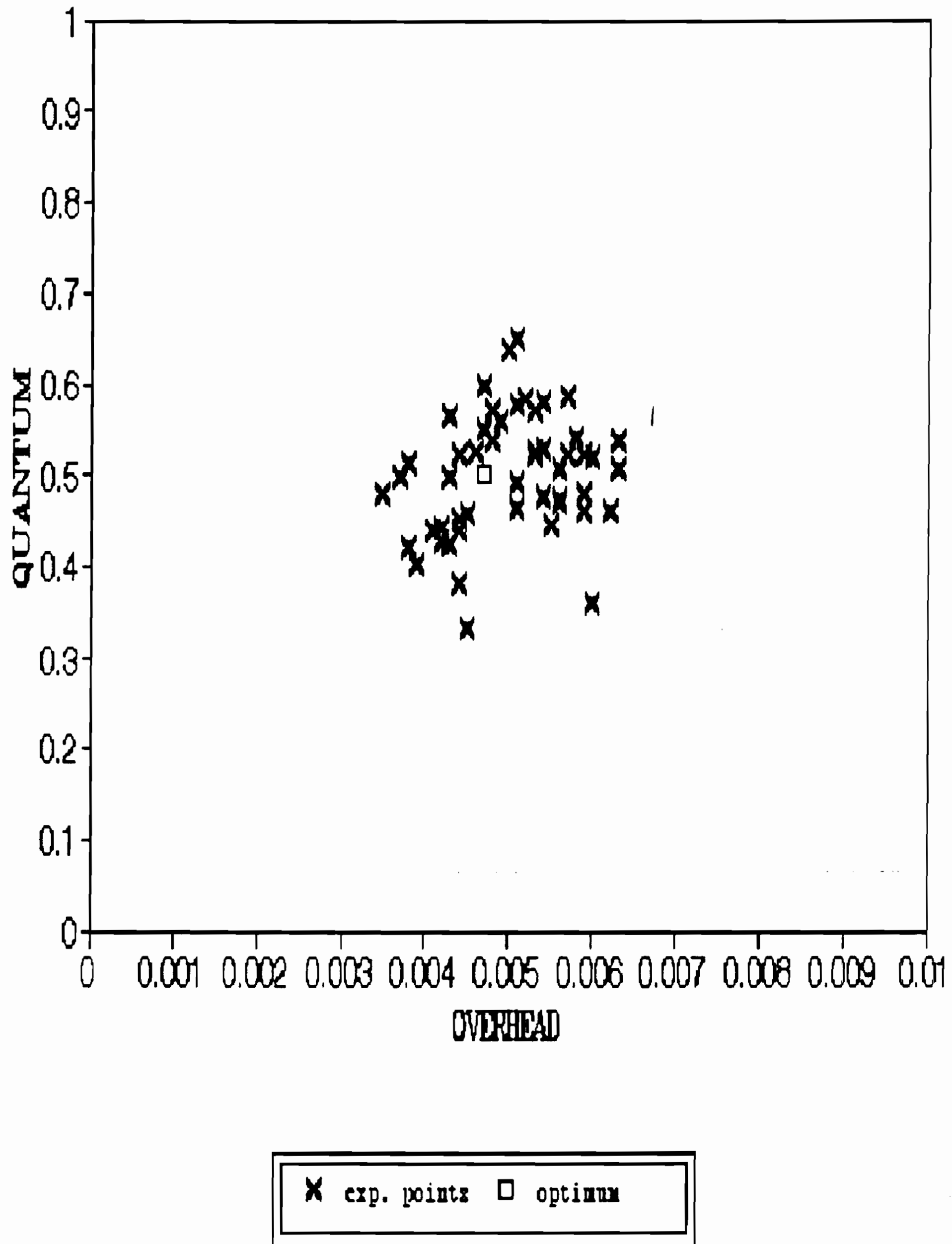



**Figure 8-Final Points for Computer System-Genetic Search**

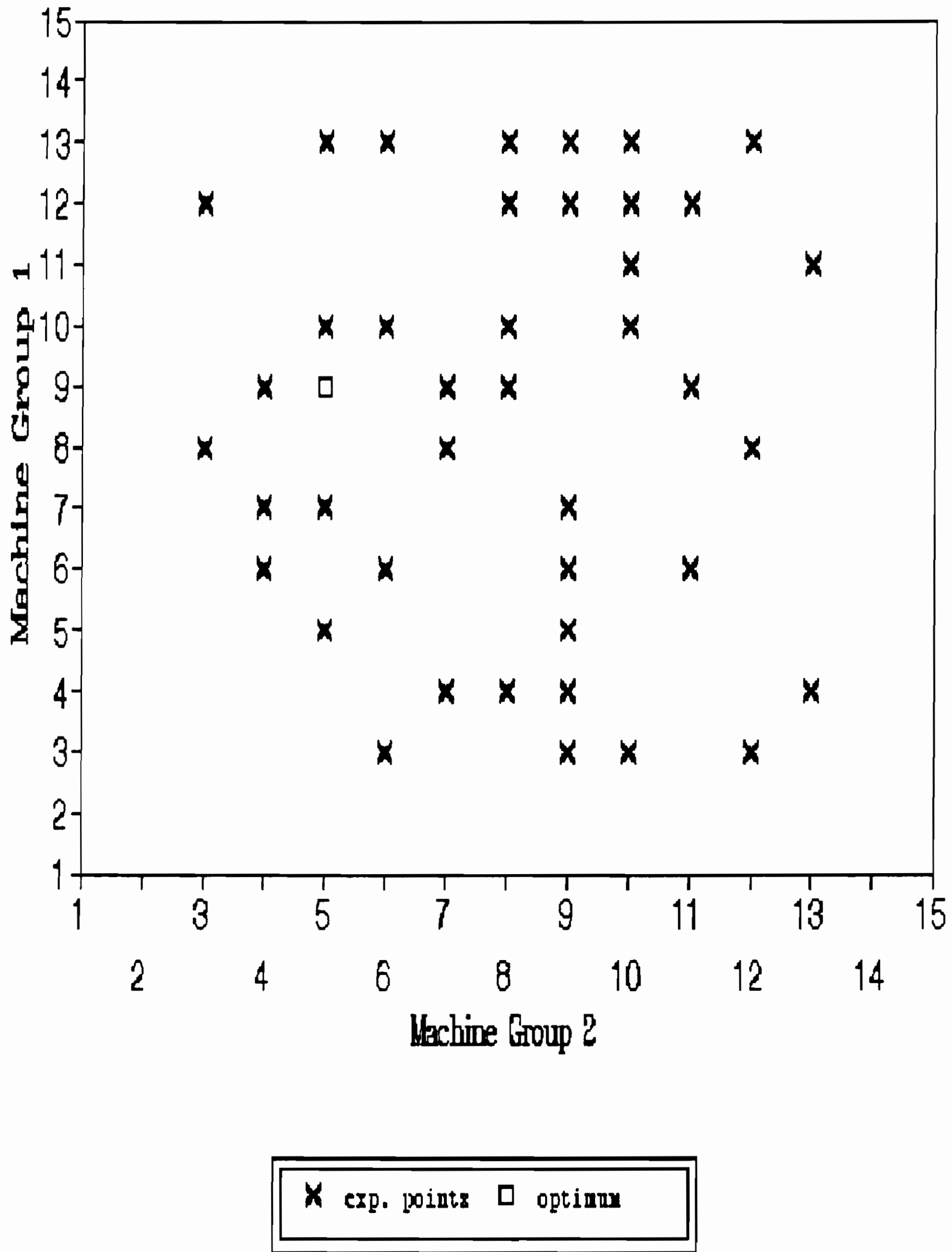



**Figure 9-Initial Points Job-Shop Pattern Search Machine Groups (1,2)**

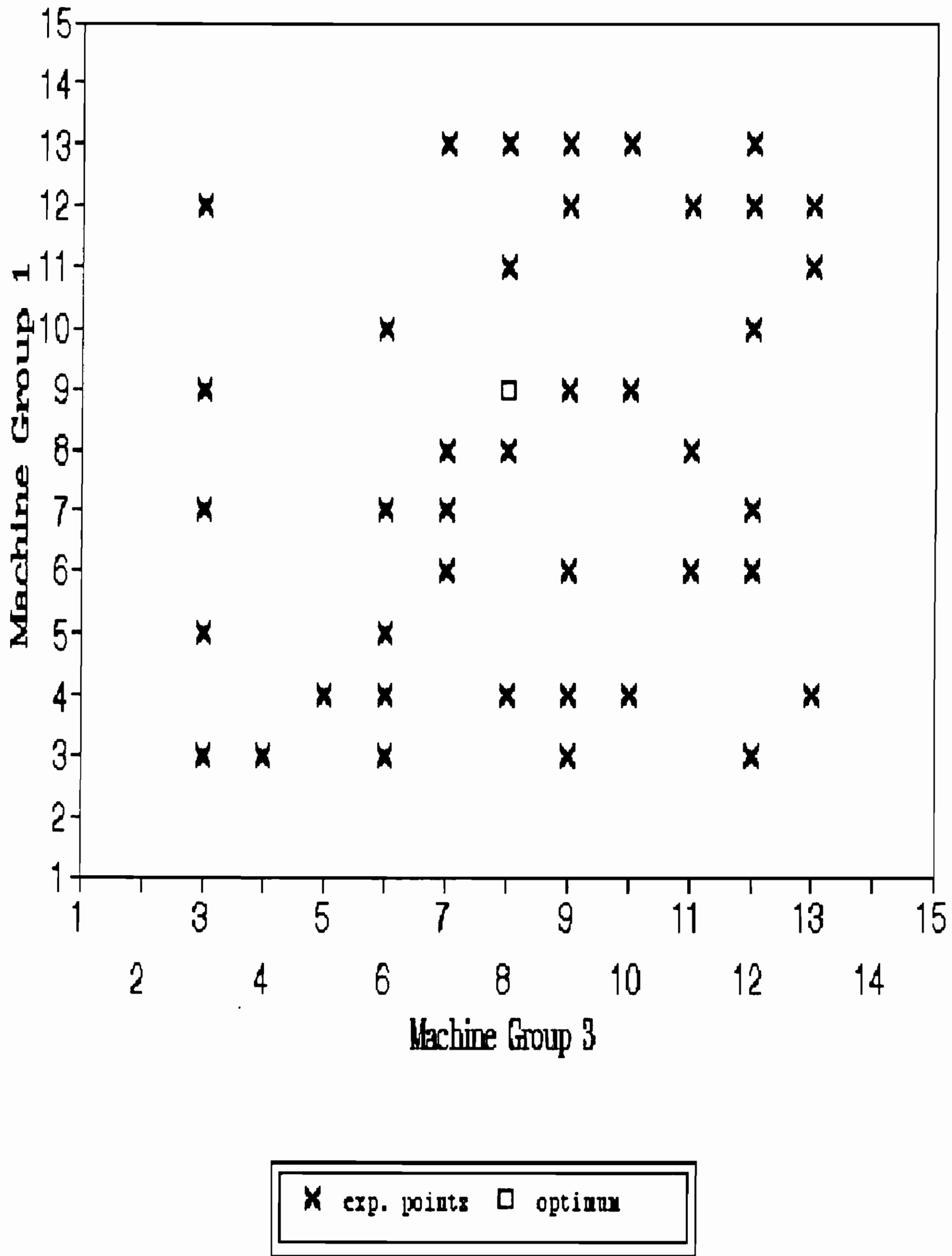



**Figure 10-Initial Points Job-Shop Pattern Search Machine Groups (1,3)**

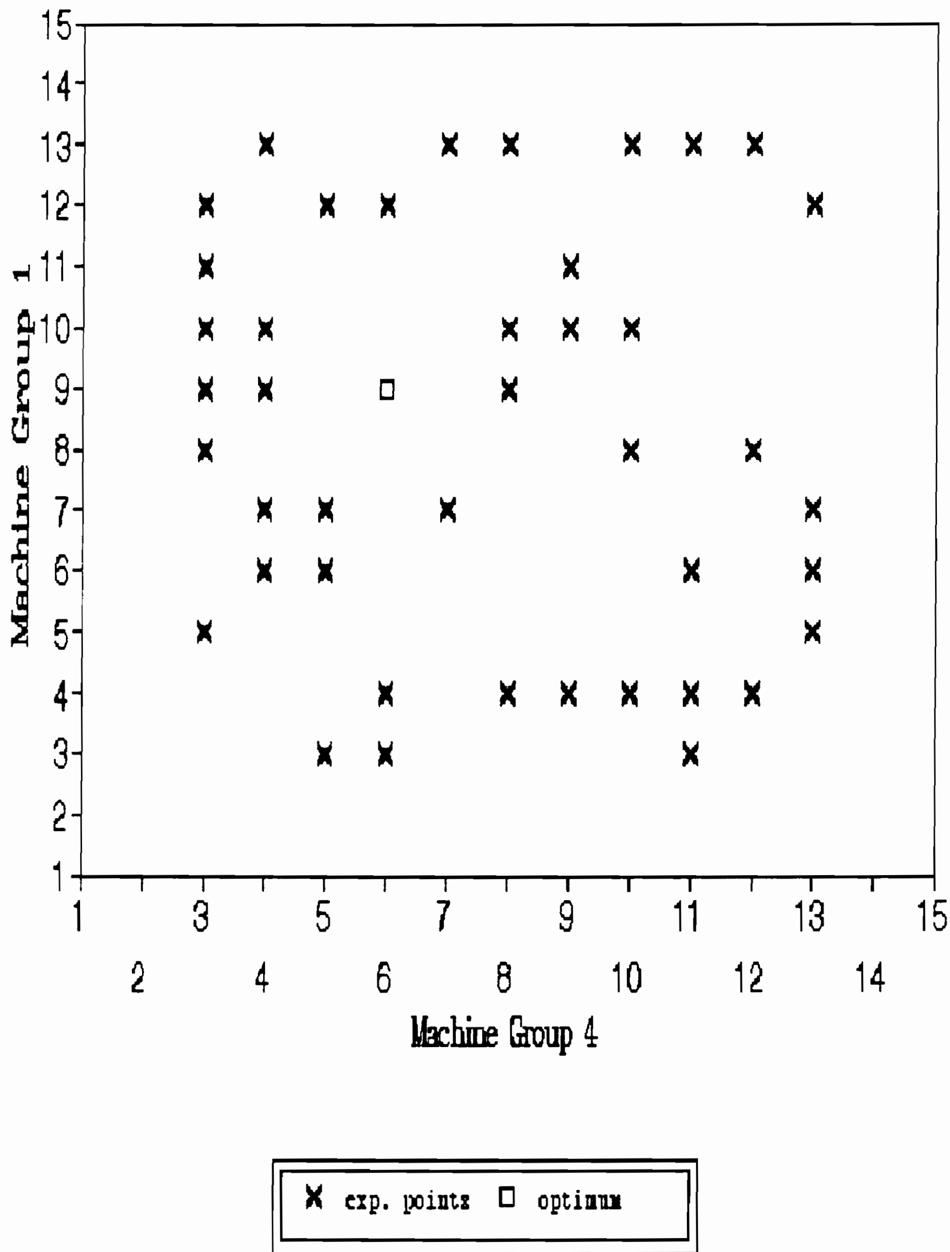
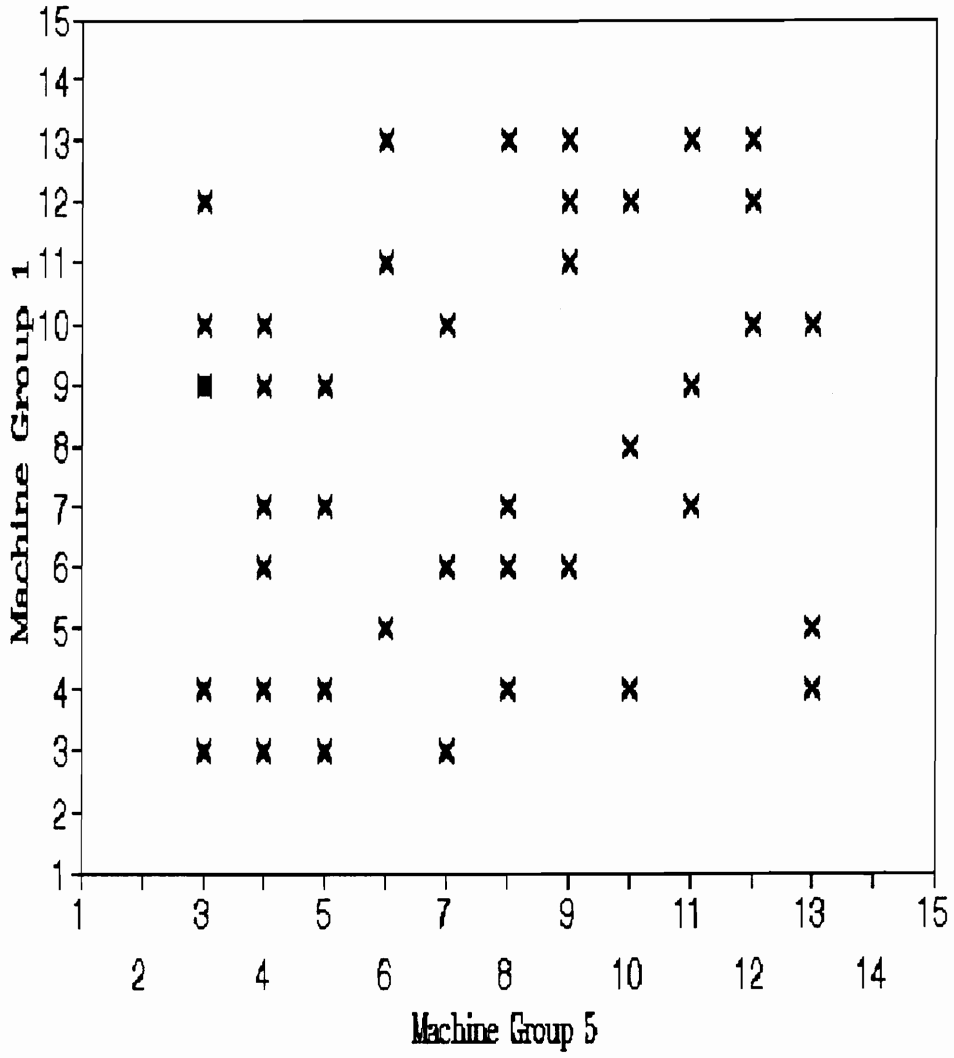



**Figure 11-Initial Points Job-Shop Pattern Search Machine Groups (1,4)**



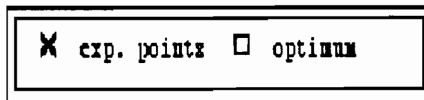




**Figure 12-Initial Points Job-Shop Pattern Search Machine Groups (1,5)**

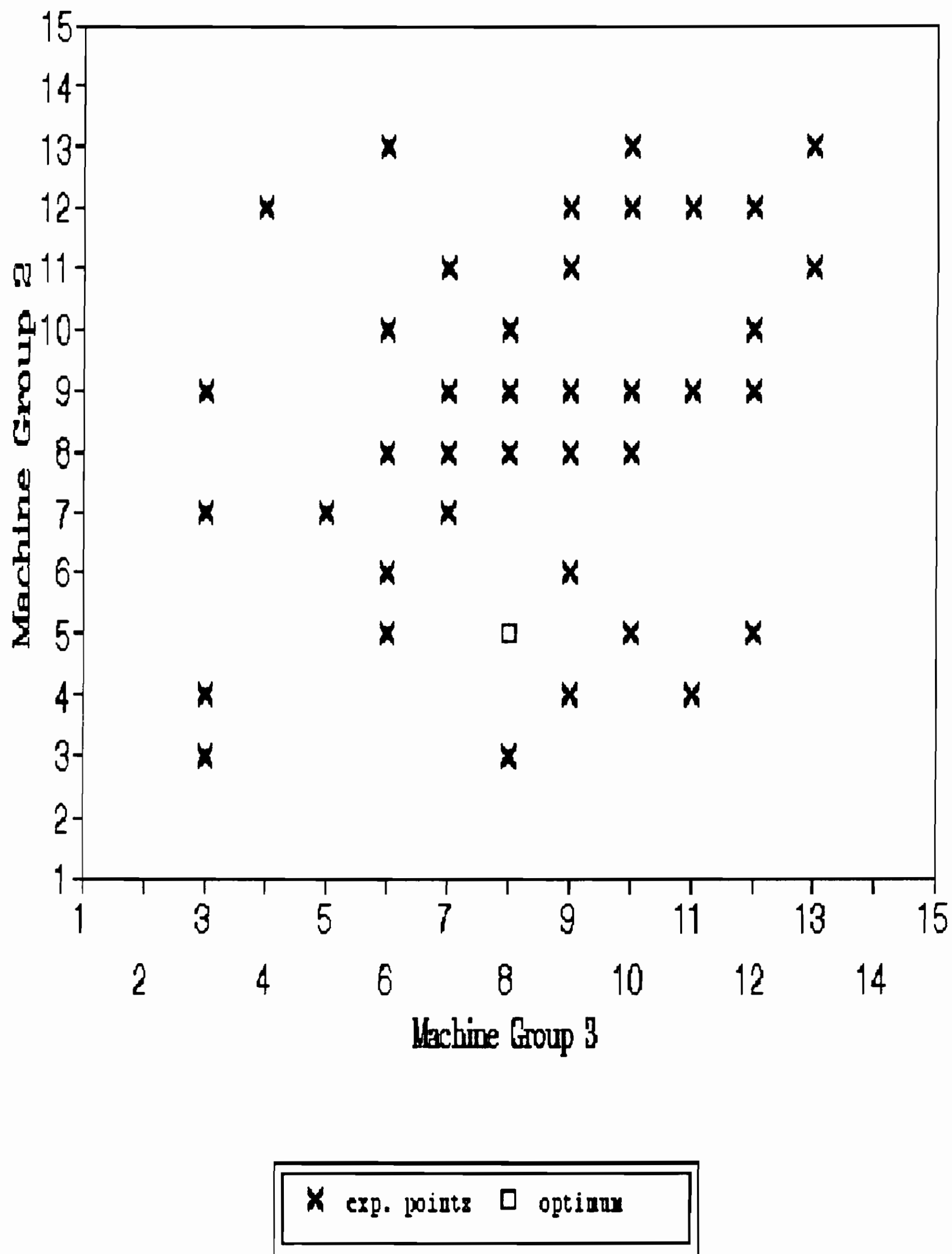



**Figure 13-Initial Points Job-Shop Pattern Search Machine Groups (2,3)**

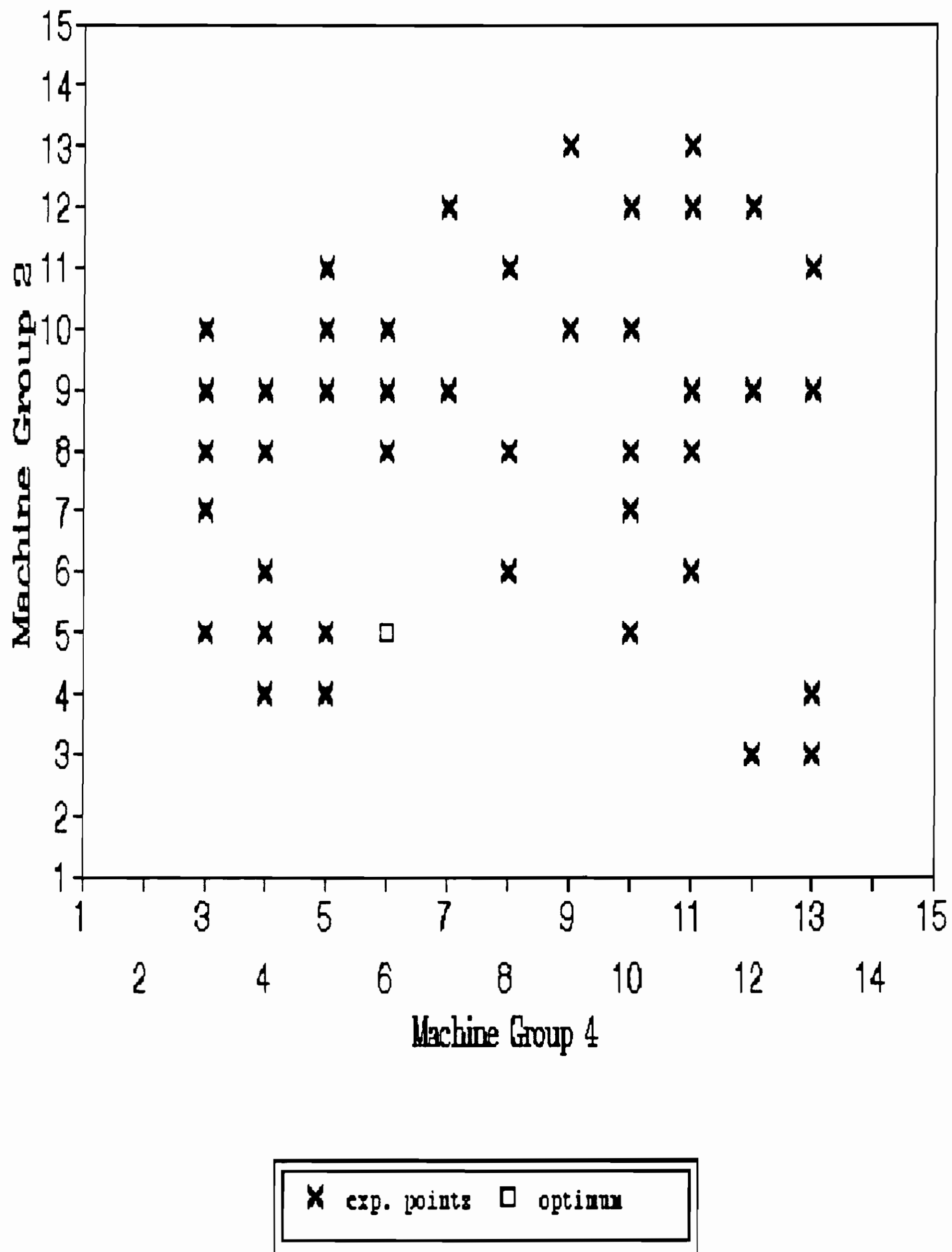



**Figure 14-Initial Points Job-Shop Pattern Search Machine Groups (2,4)**

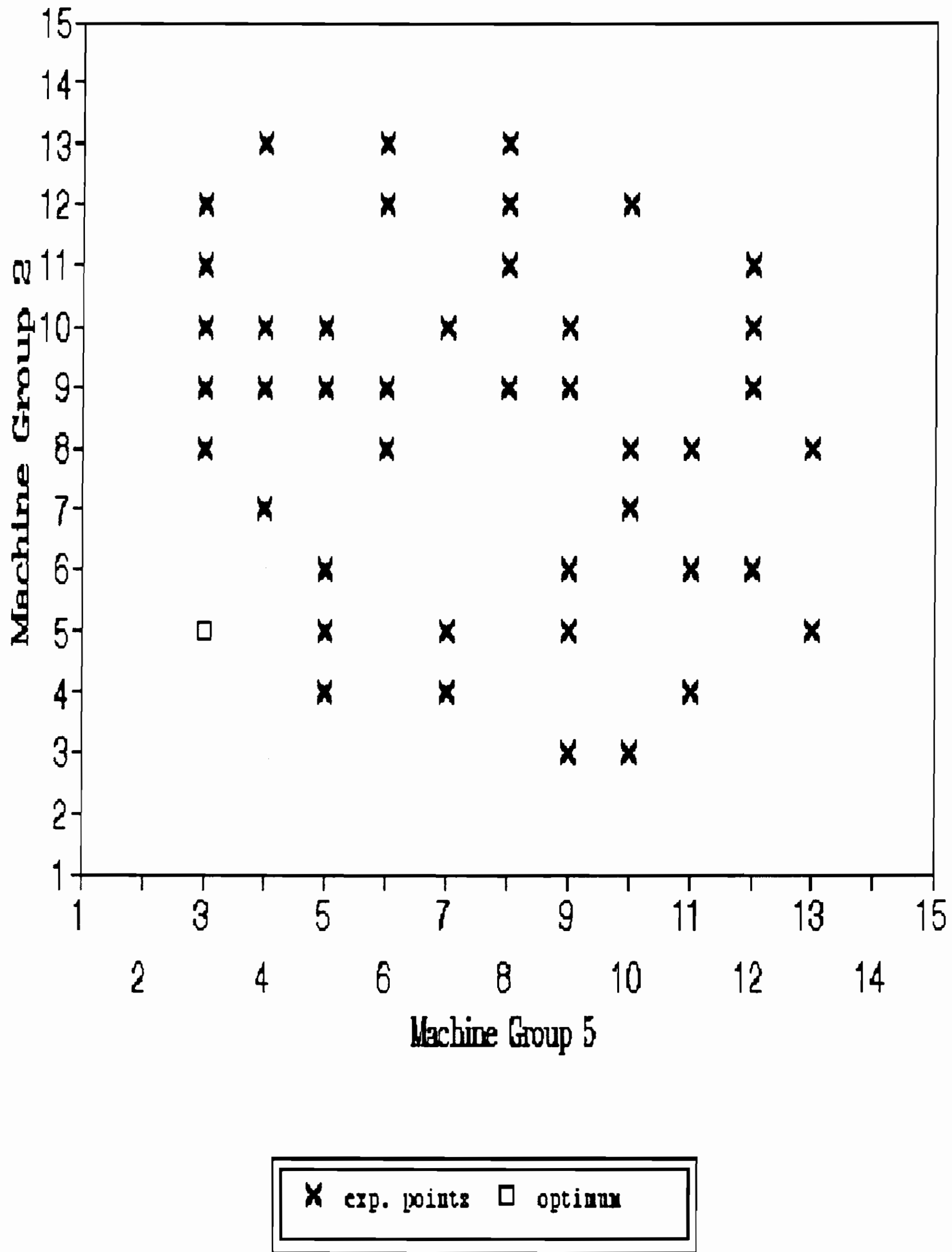



**Figure 15-Initial Points Job-Shop Pattern Search Machine Groups (2,5)**

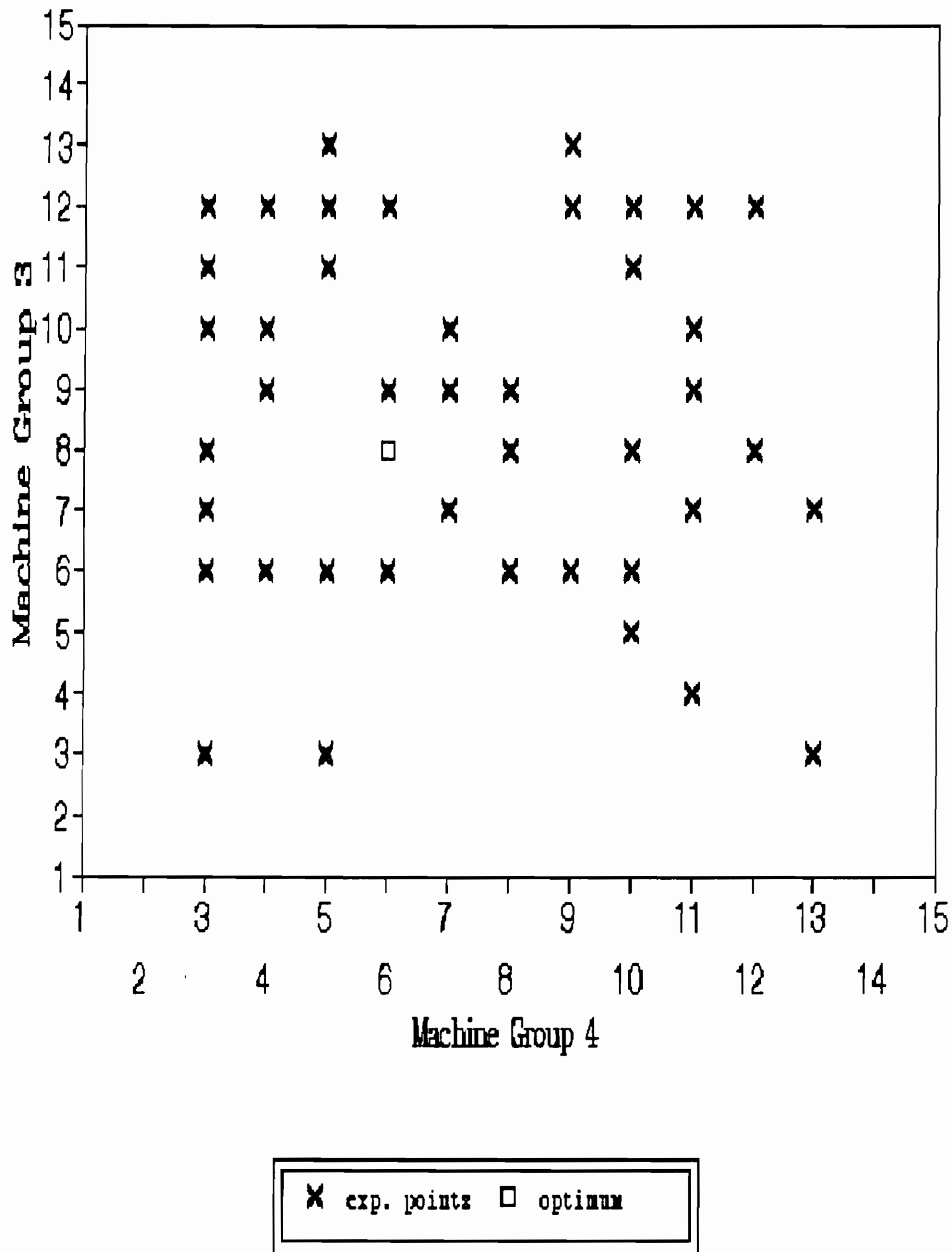



**Figure 16-Initial Points Job-Shop Pattern Search Machine Groups (3,4)**

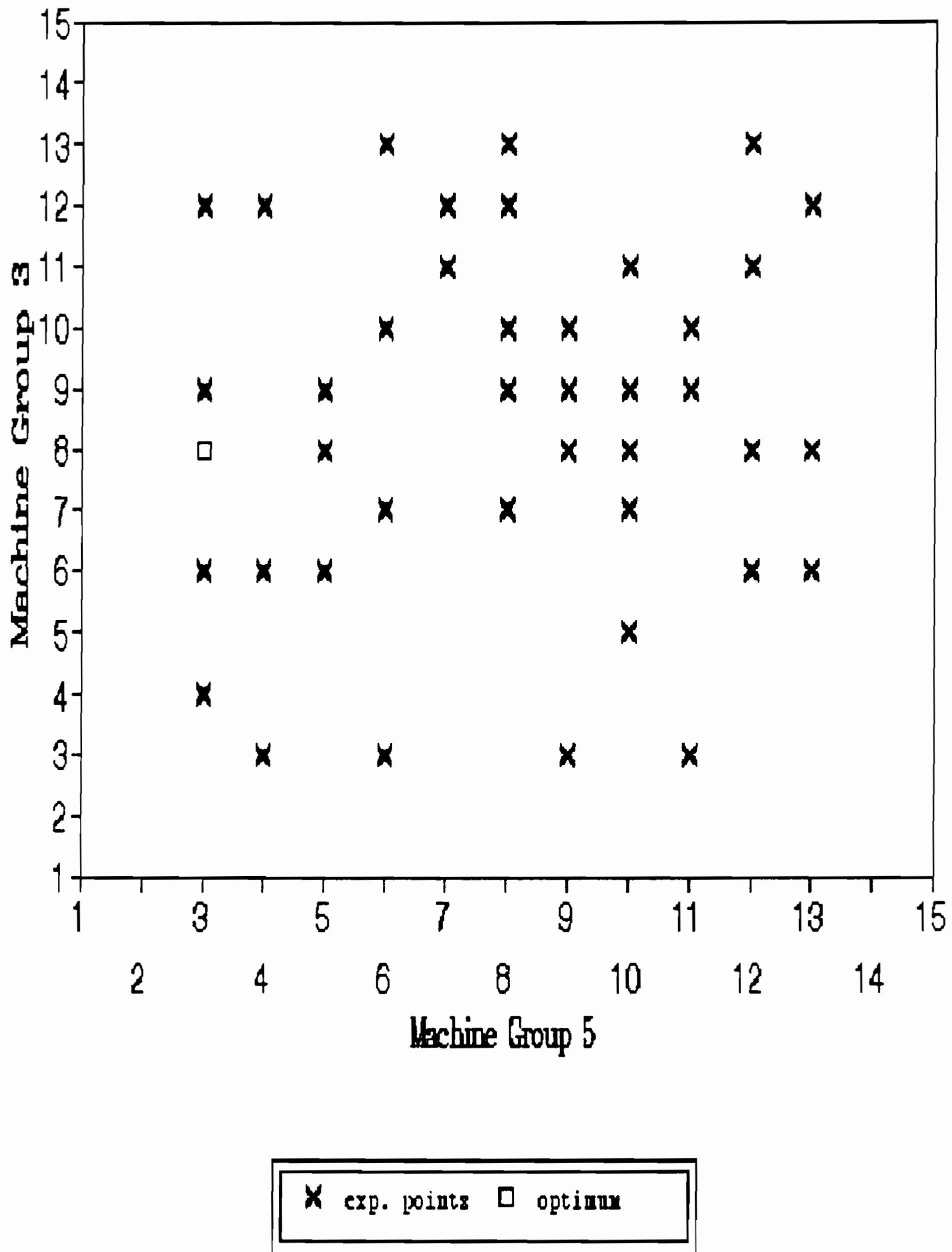



**Figure 17-Initial Points Job-Shop Pattern Search Machine Groups (3,5)**

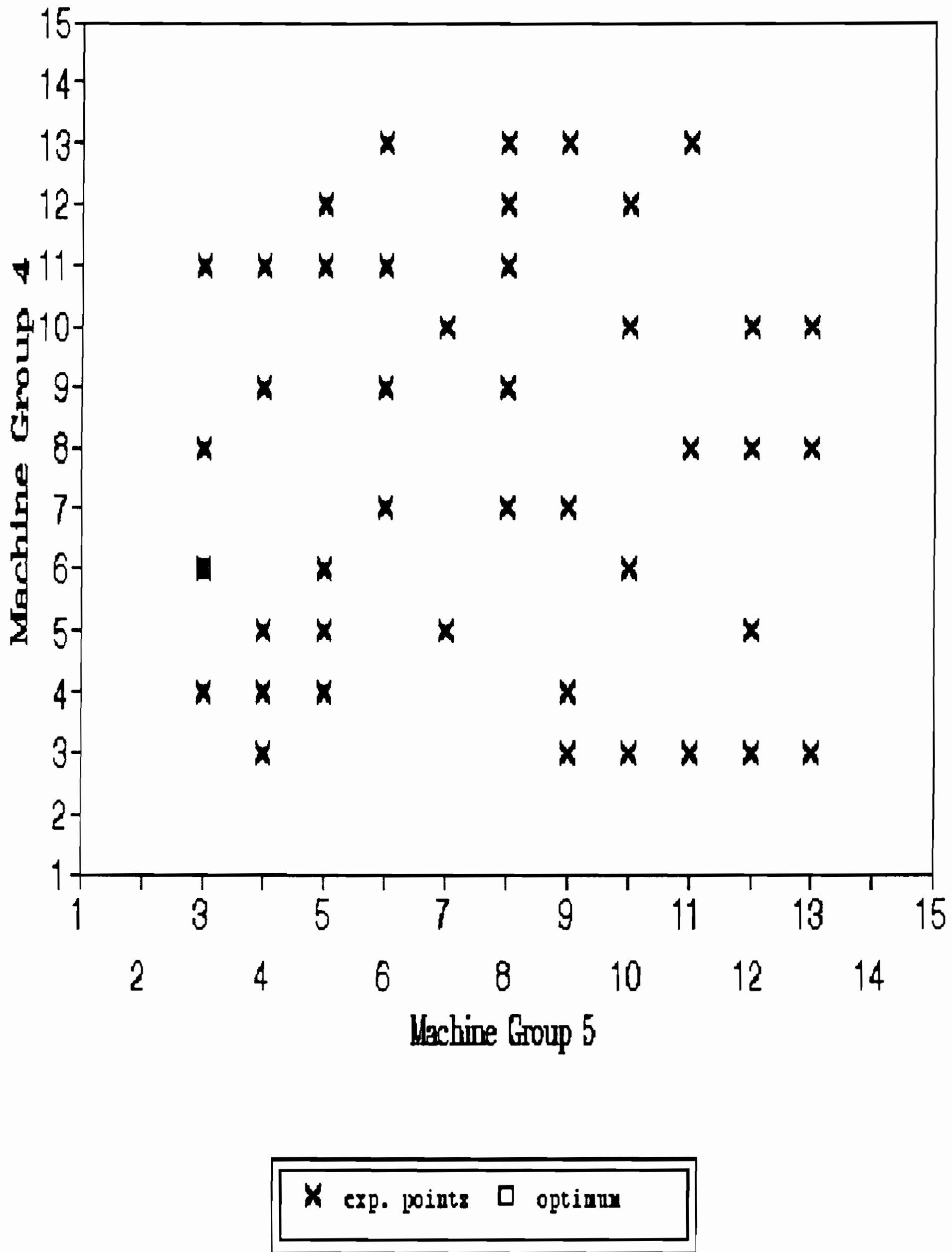
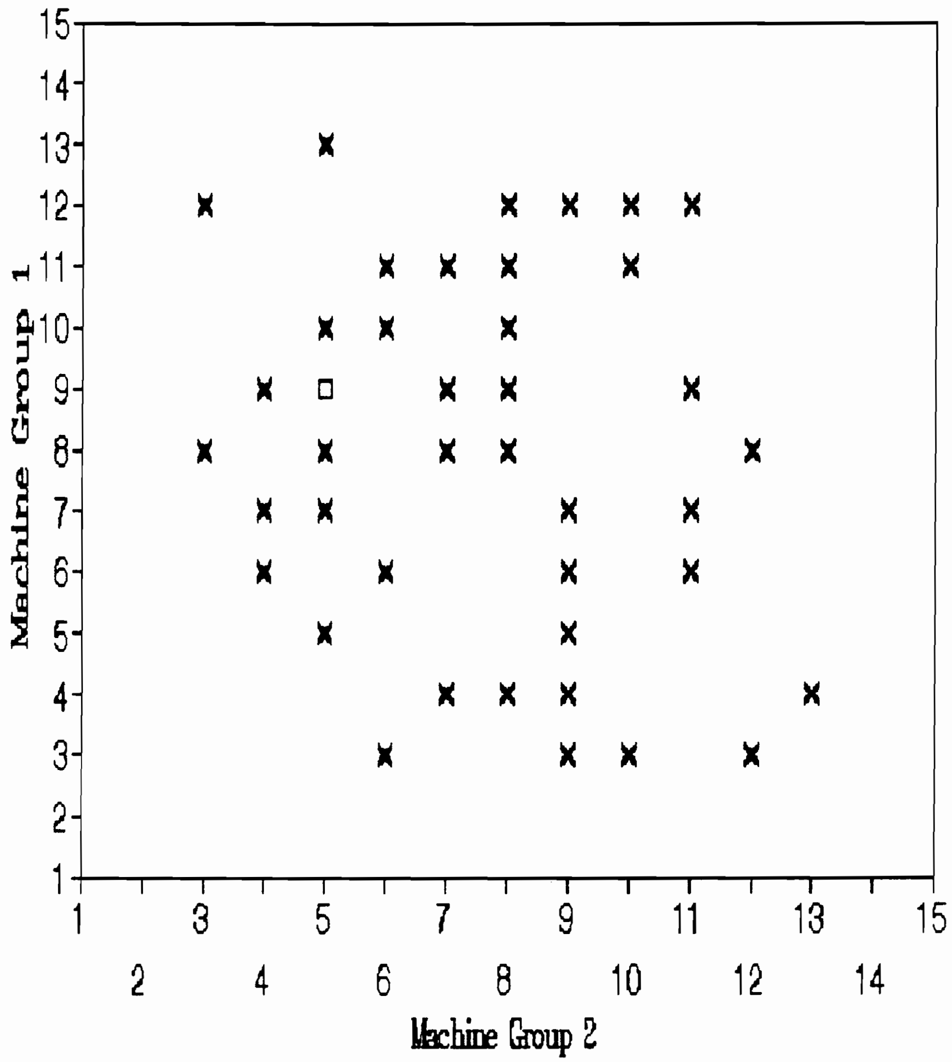



Figure 18-Initial Points Job-Shop Pattern Search Machine Groups (4,5)



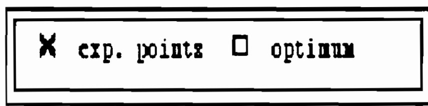
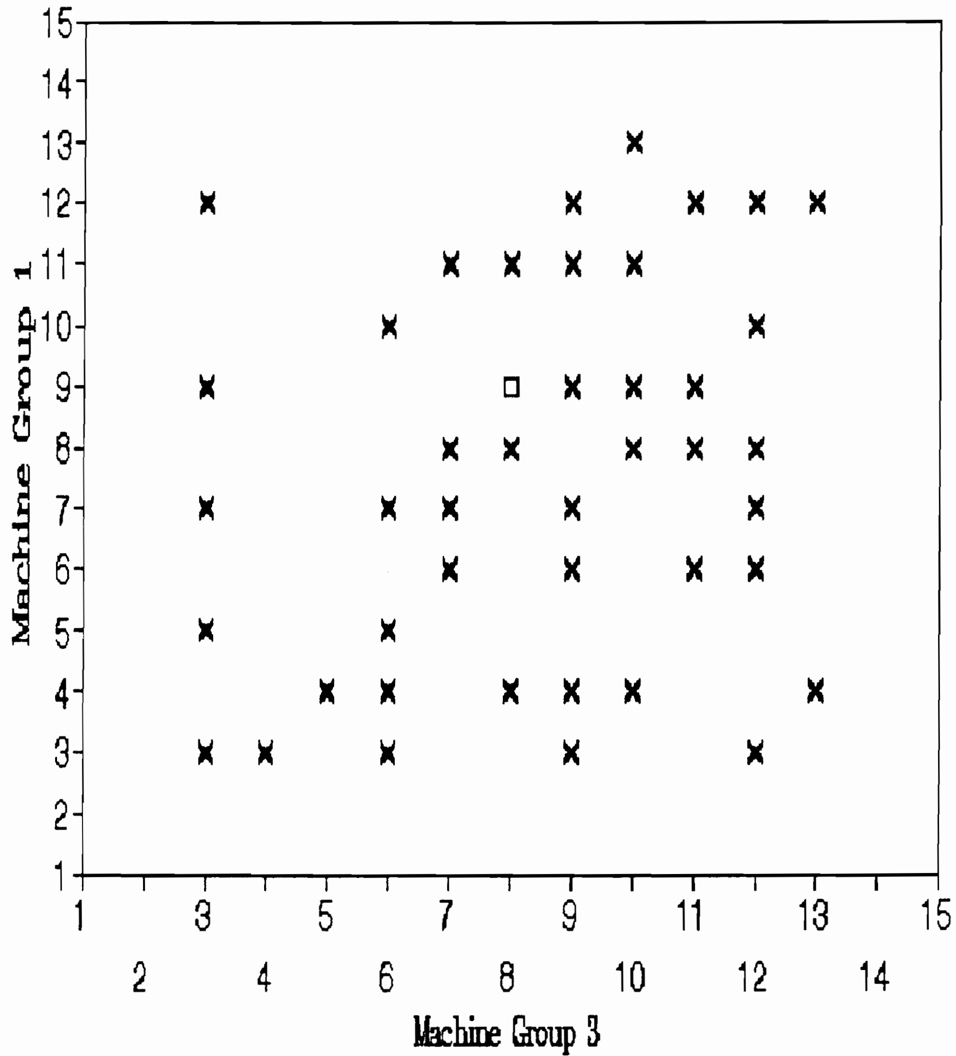




**Figure 19-Final Points Job-Shop Pattern Search Machine Groups (1,2)**



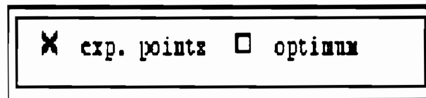




**Figure 20-Final Points Job-Shop Pattern Search Machine Groups (1,3)**

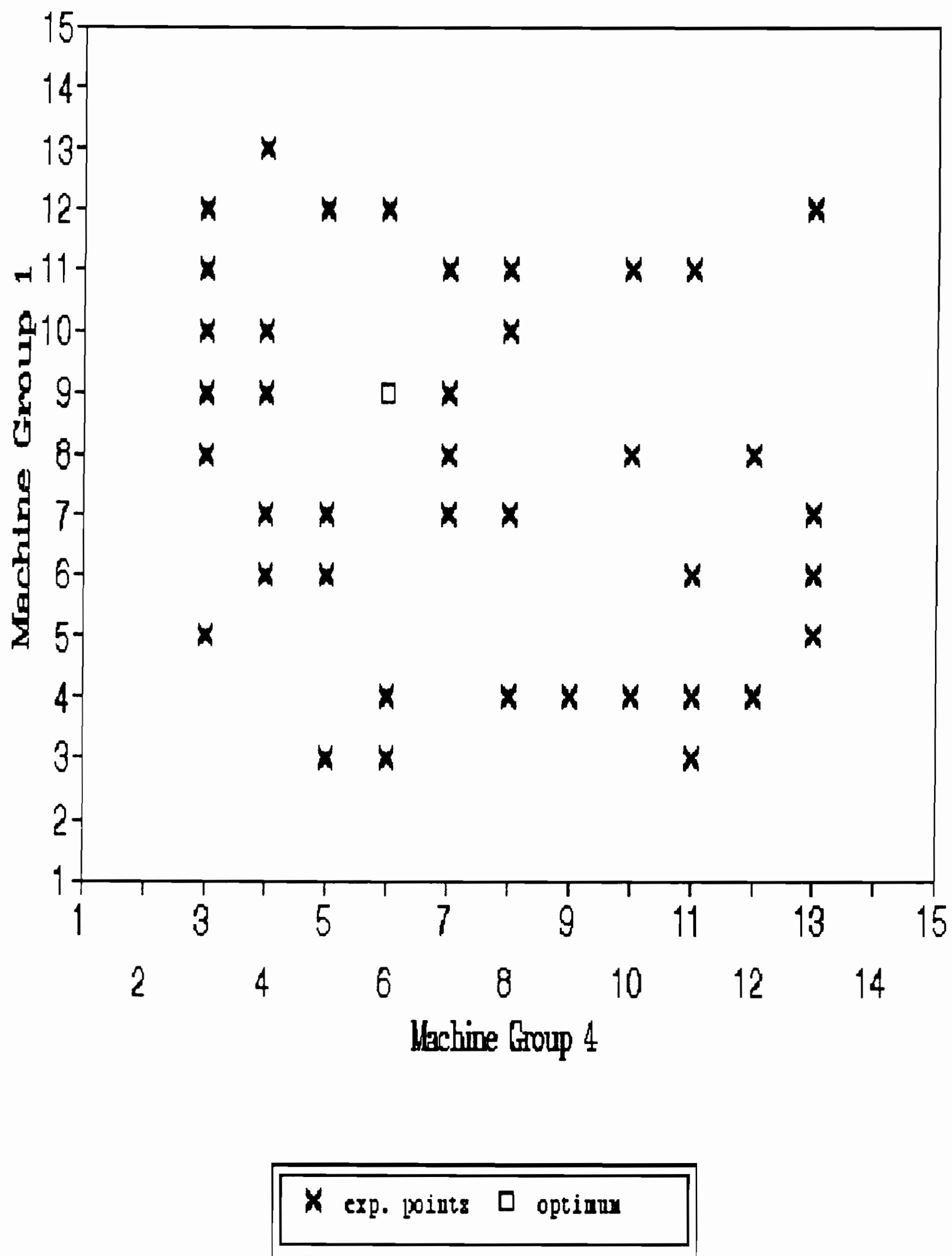
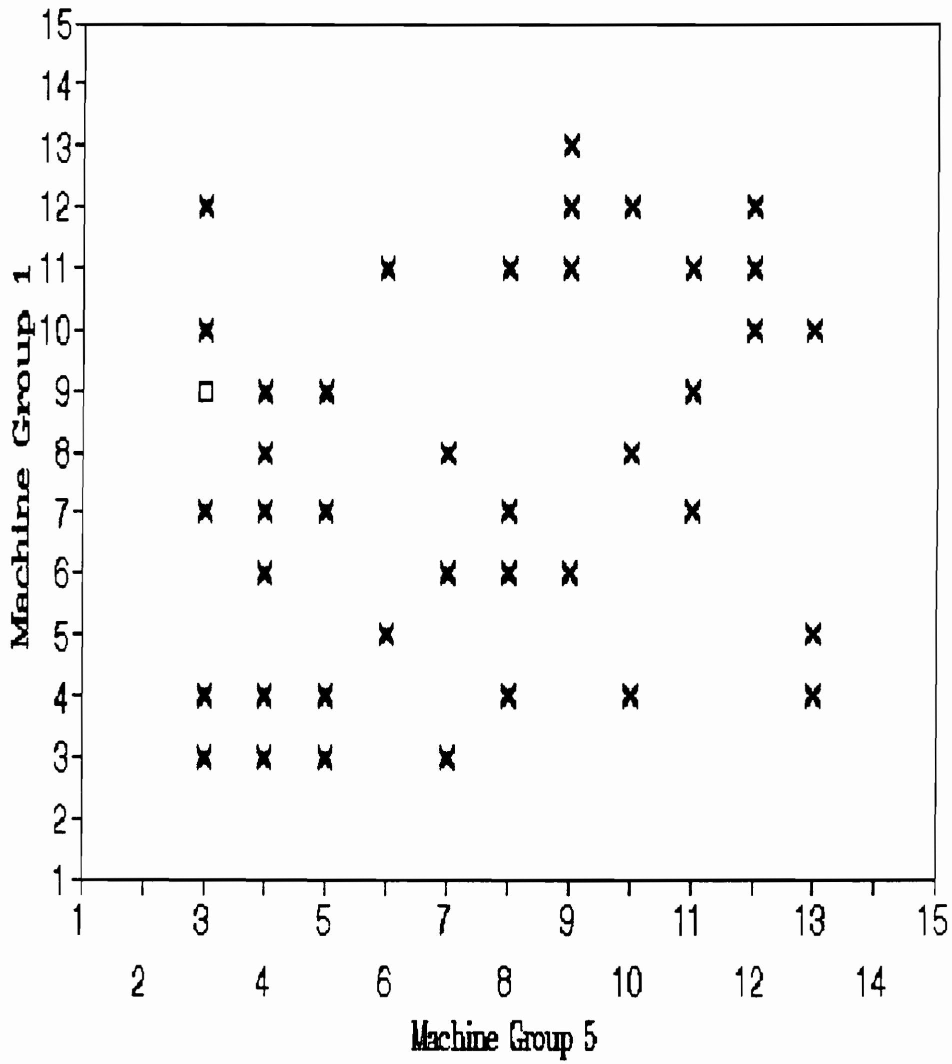



**Figure 21-Final Points Job-Shop Pattern Search Machine Groups (1,4)**



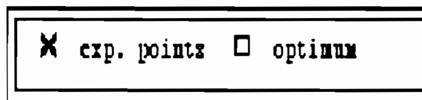




Figure 22-Final Points Job-Shop Pattern Search Machine Groups (1,5)

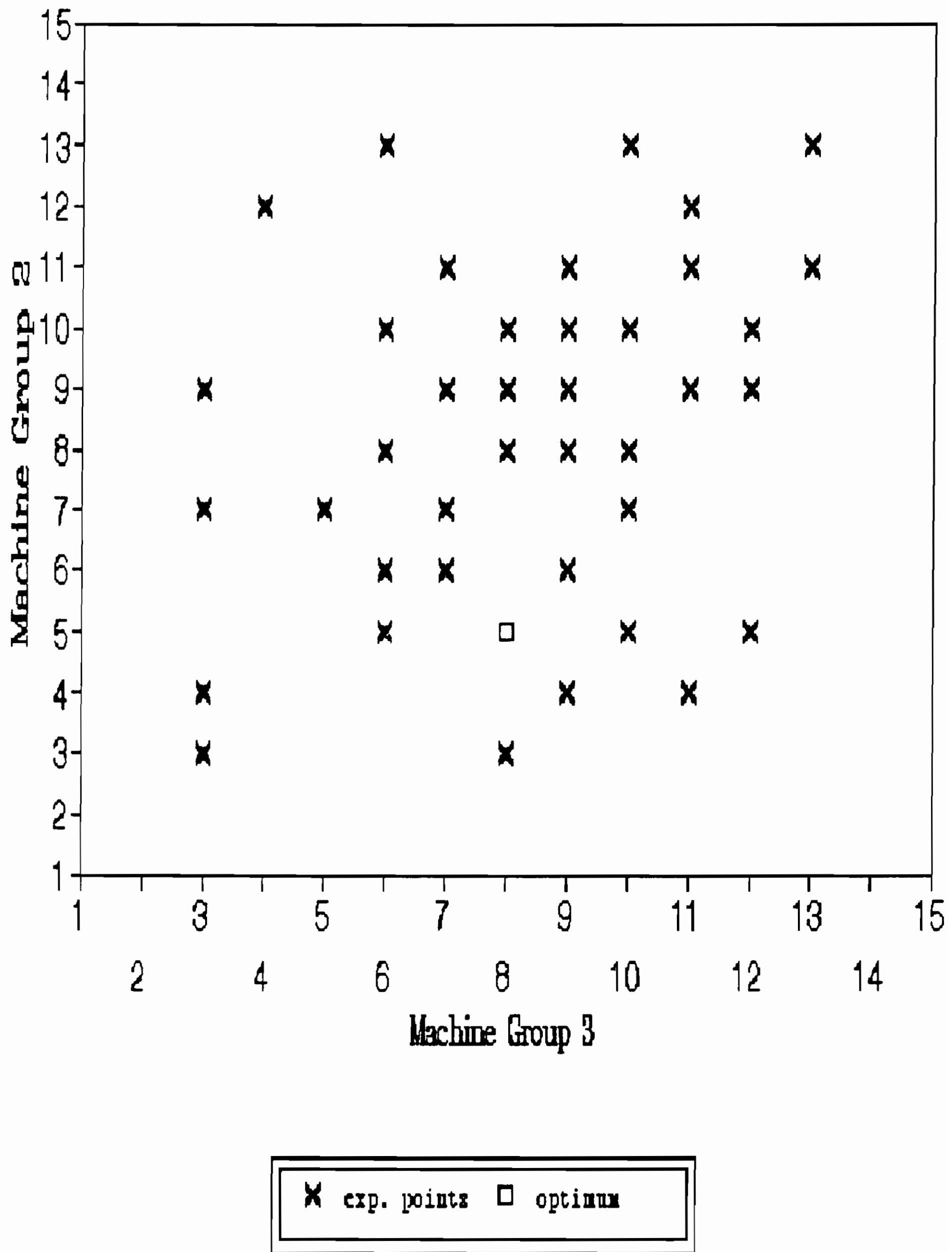



**Figure 23-Final Points Job-Shop Pattern Search Machine Groups (2,3)**

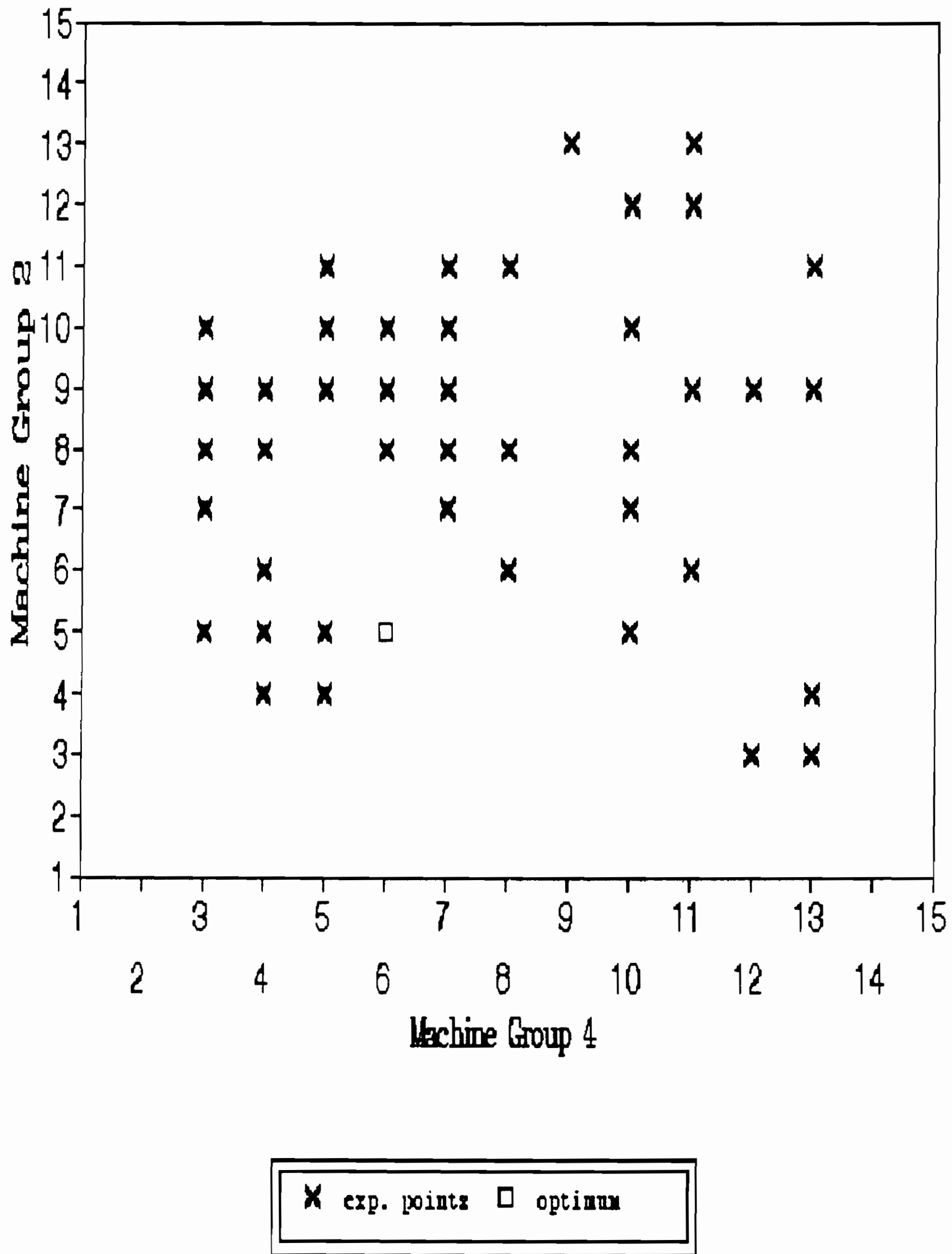



**Figure 24-Final Points Job-Shop Pattern Search Machine Groups (2,4)**

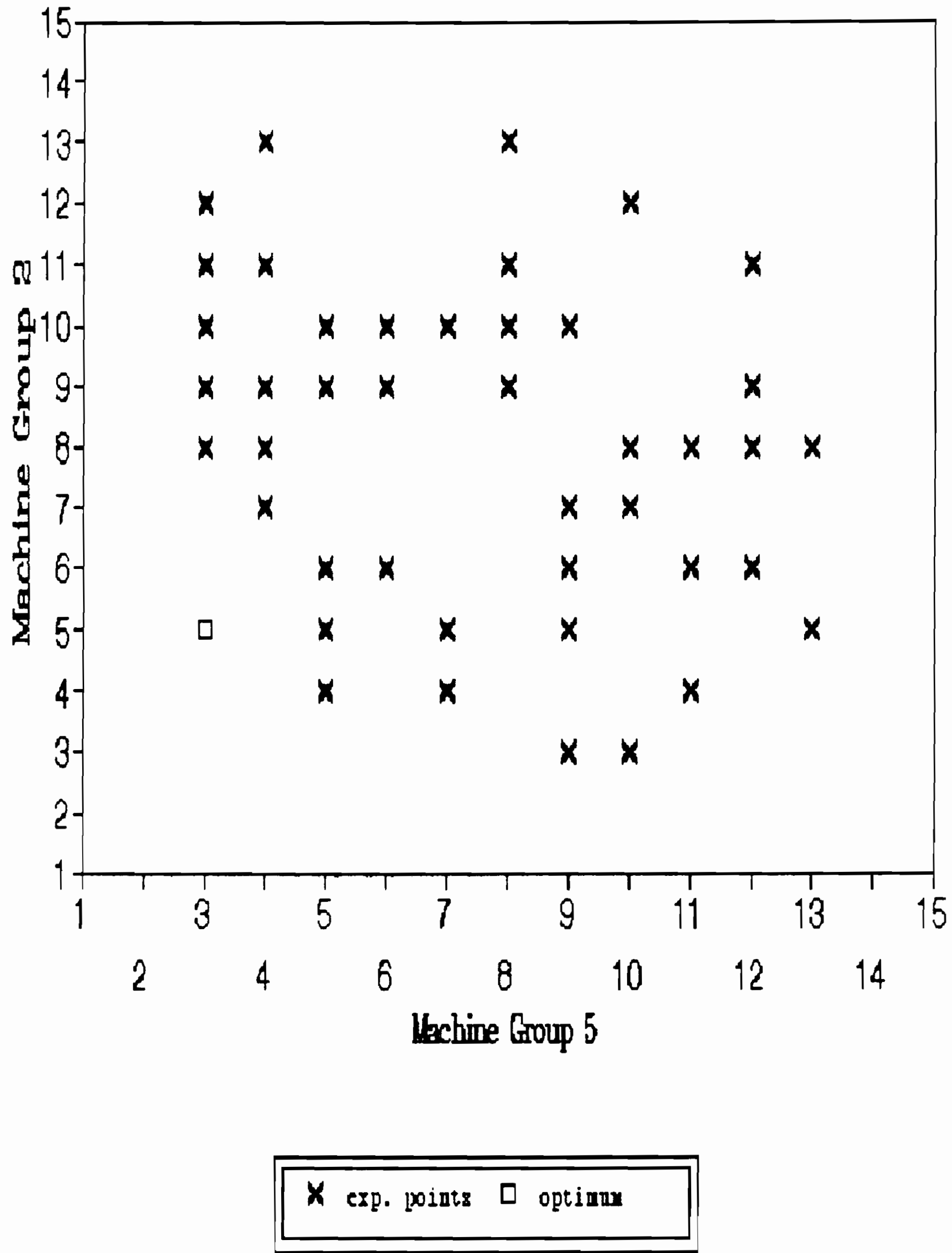



Figure 25-Final Points Job-Shop Pattern Search Machine Groups (2,5)

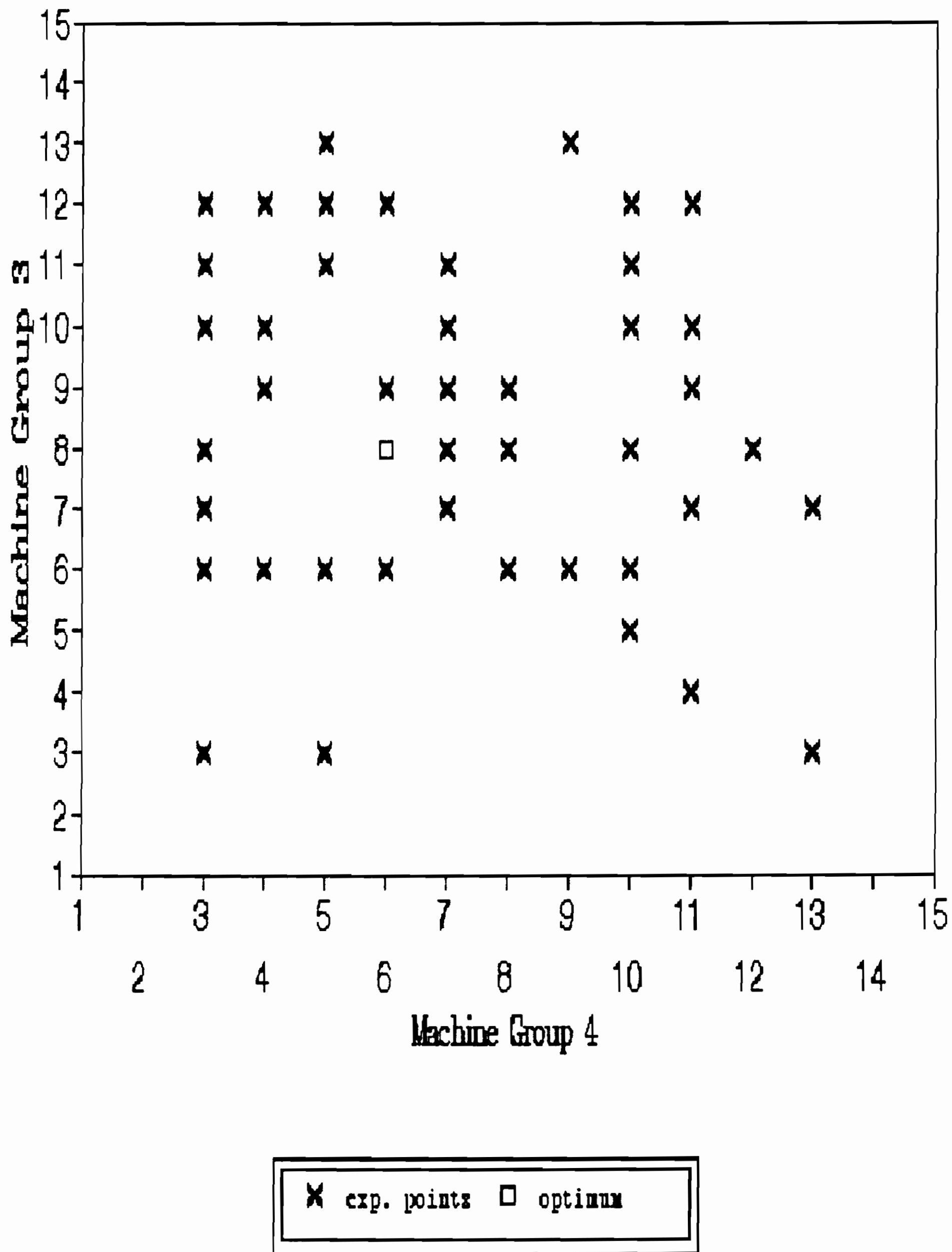



Figure 26-Final Points Job-Shop Pattern Search Machine Groups (3,4)

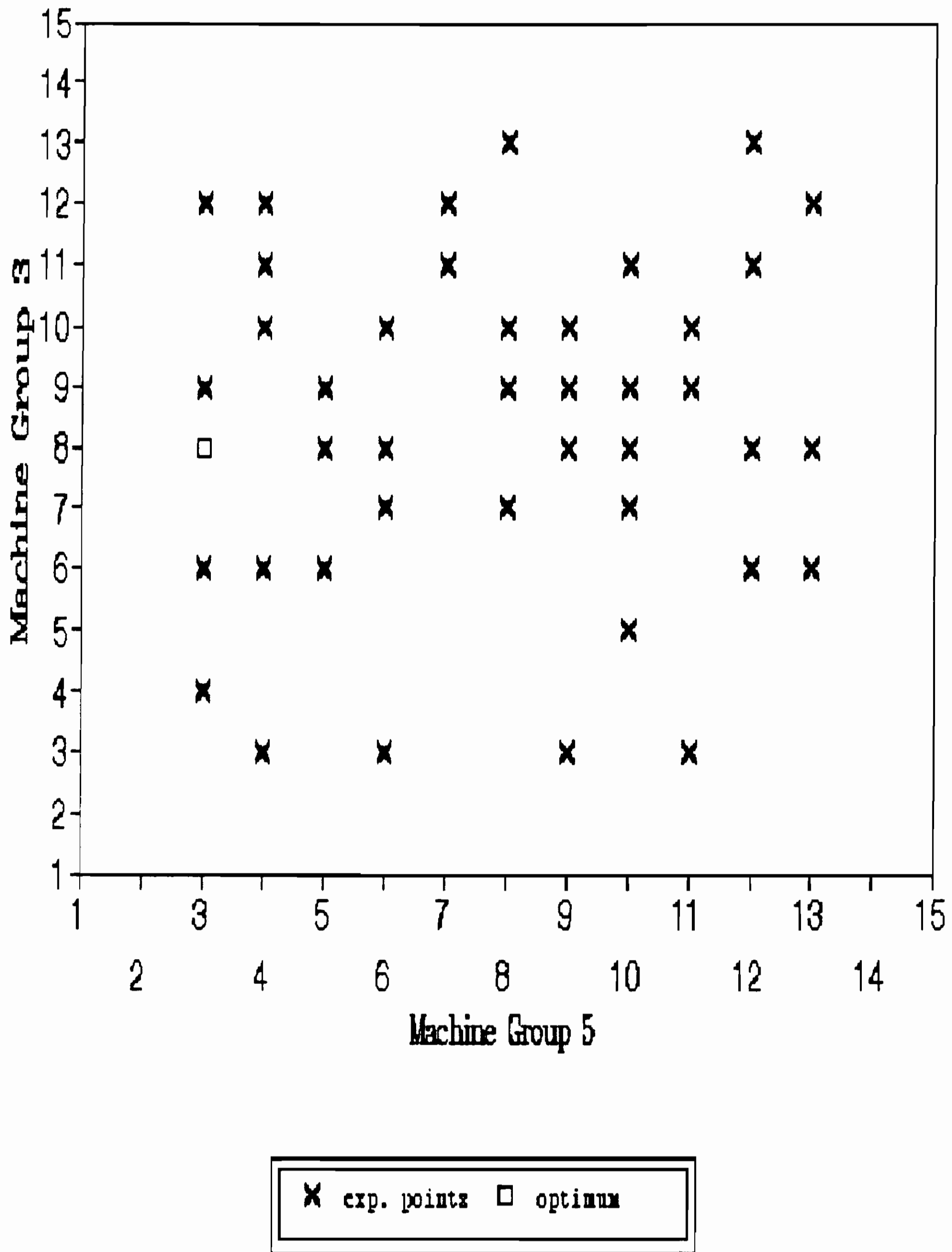
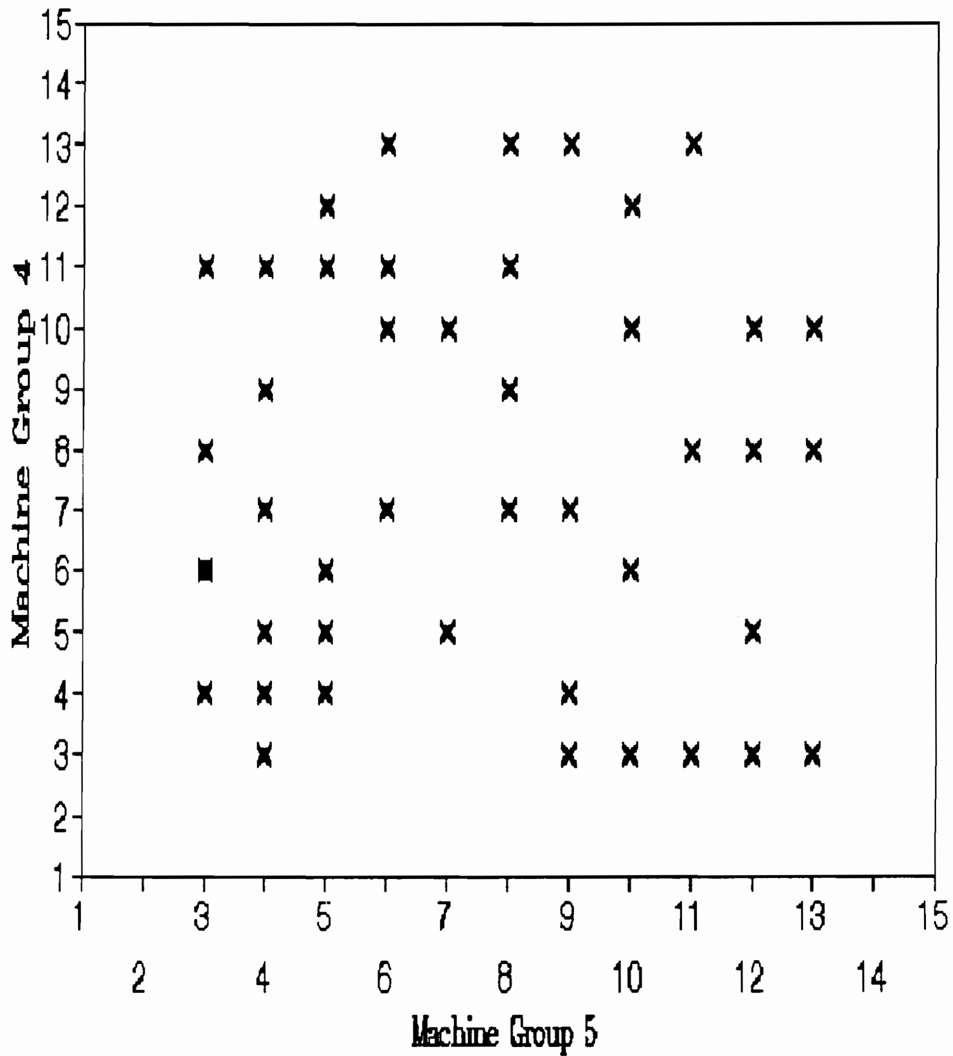



**Figure 27-Final Points Job-Shop Pattern Search Machine Groups (3,5)**



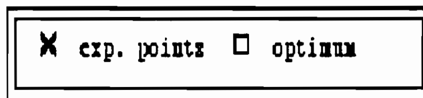




**Figure 28-Final Points Job-Shop Pattern Search Machine Groups (4,5)**

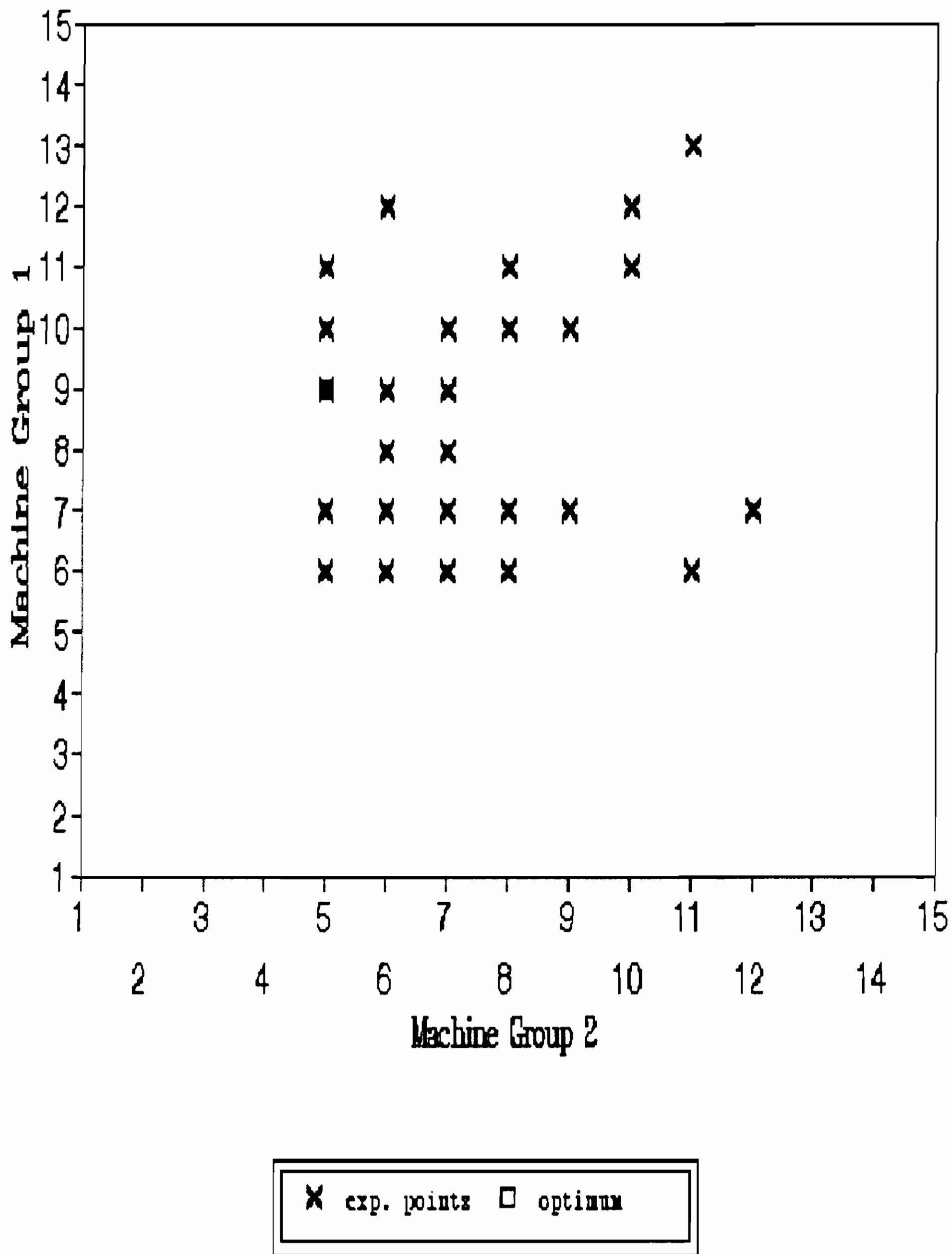



**Figure 29-Final Points Job-Shop Response Surface Search Machine Groups (1,2)**

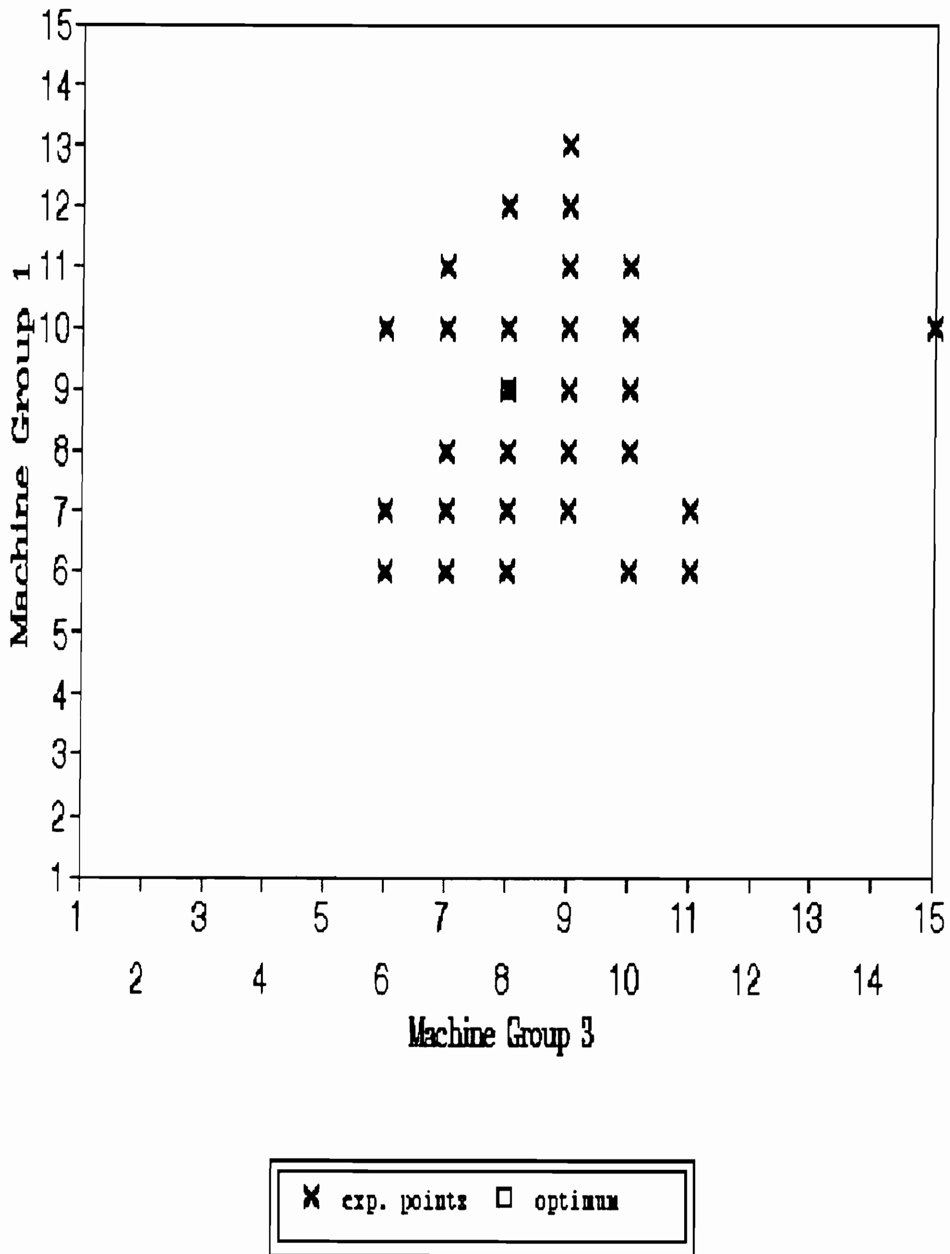



**Figure 30-Final Points Job-Shop Response Surface Search Machine Groups (1,3)**

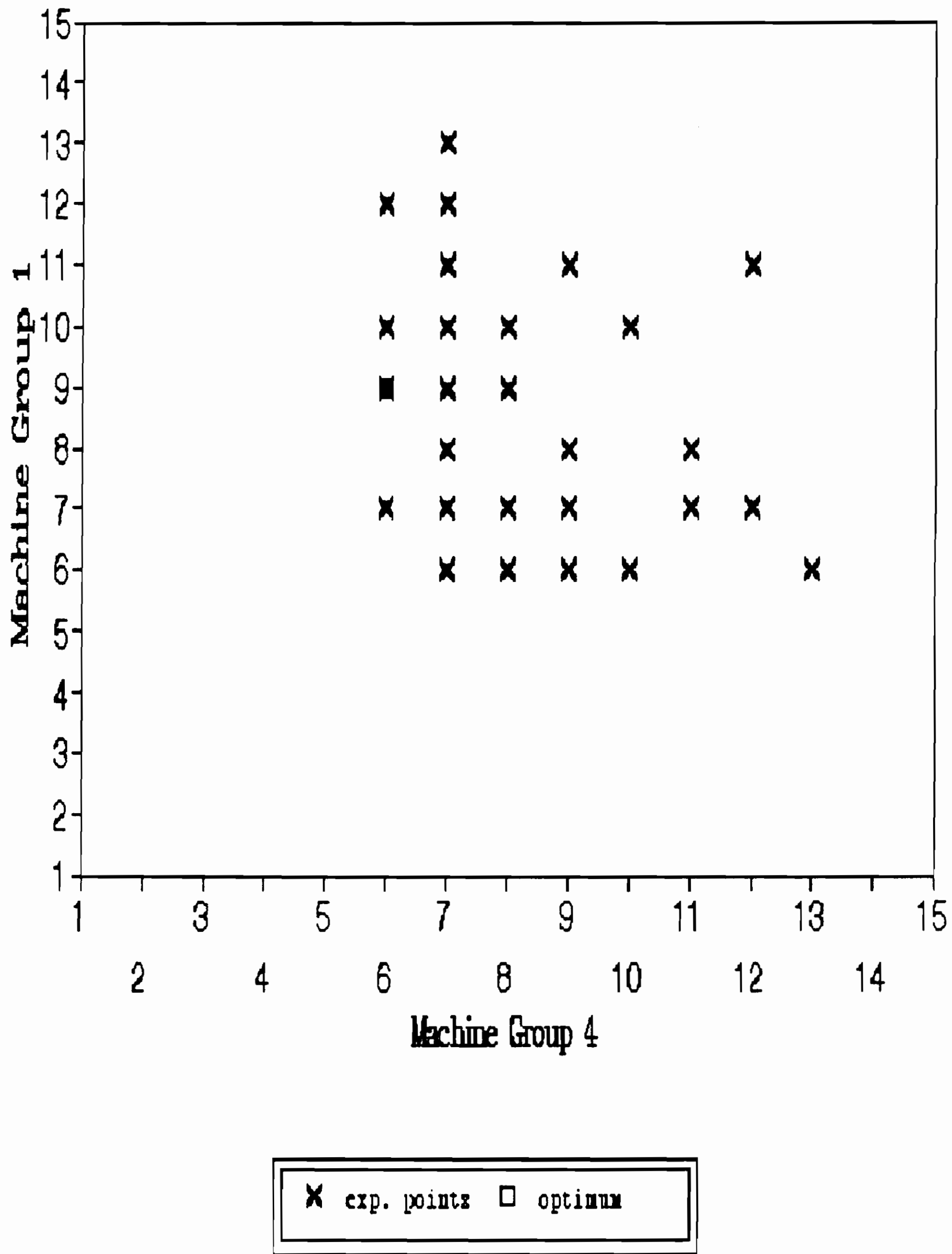



**Figure 31-Final Points Job-Shop Response Surface Search Machine Groups (1,4)**

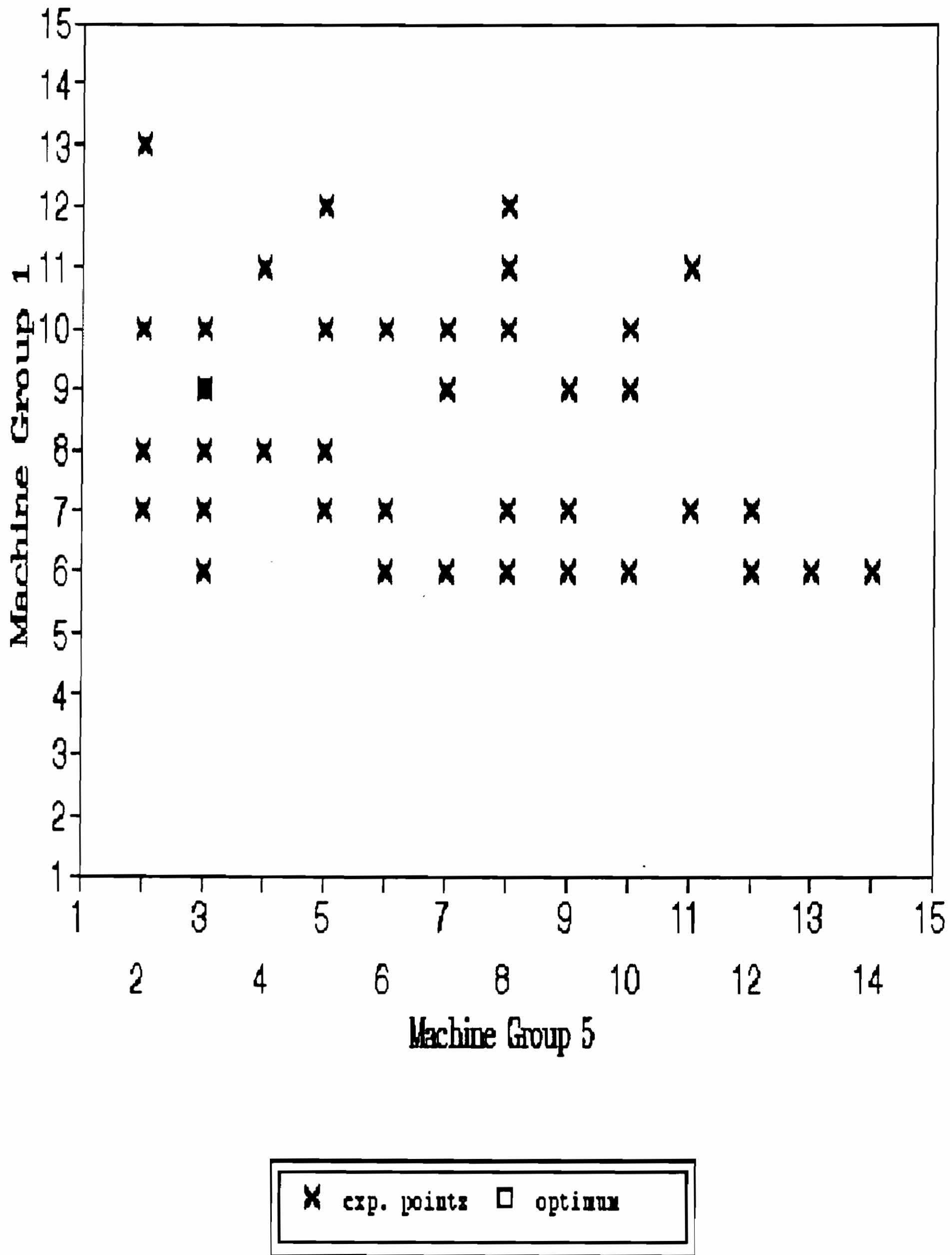



**Figure 32-Final Points Job-Shop Response Surface Search Machine Groups (1,5)**

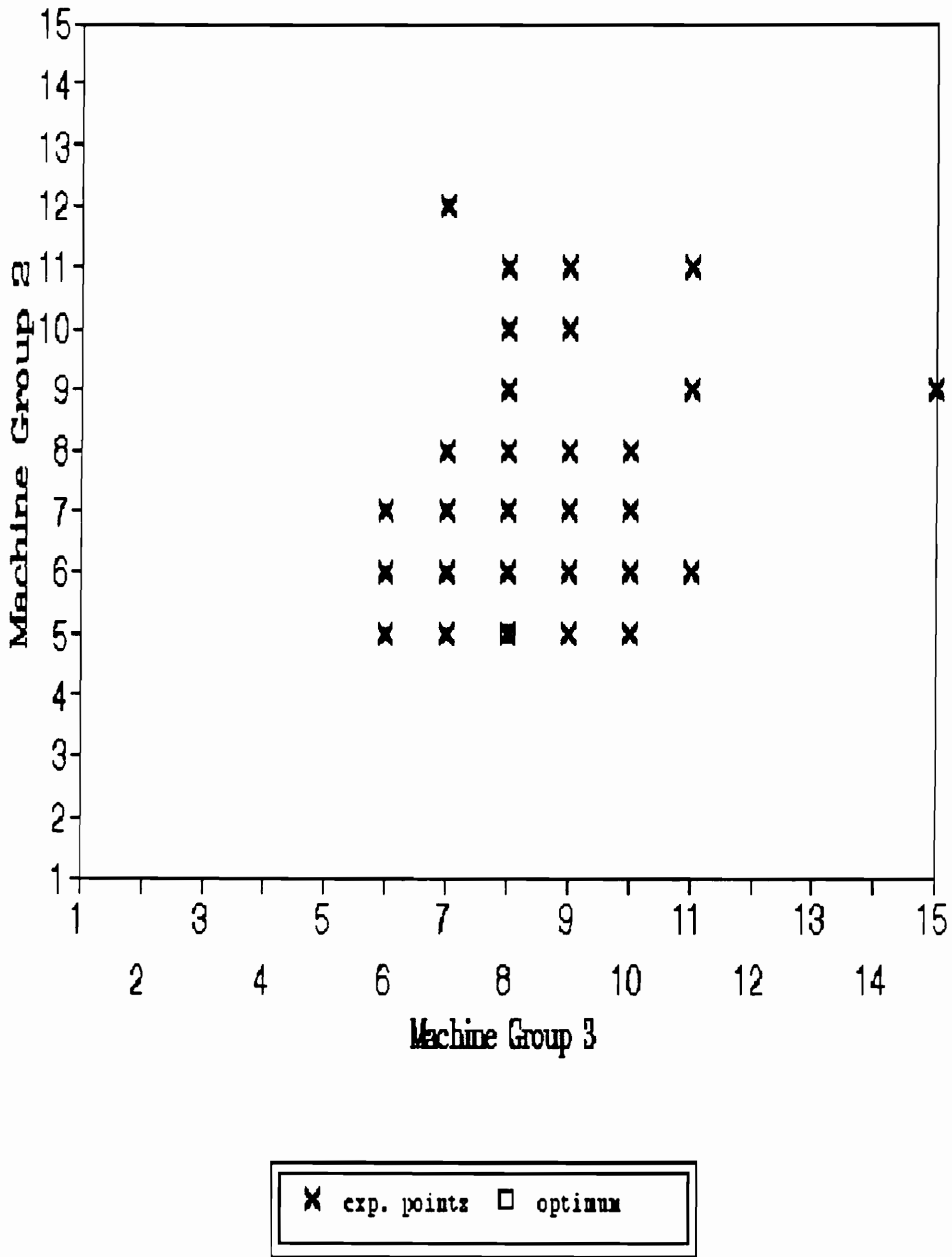



**Figure 33-Final Points Job-Shop Response Surface Search Machine Groups (2,3)**

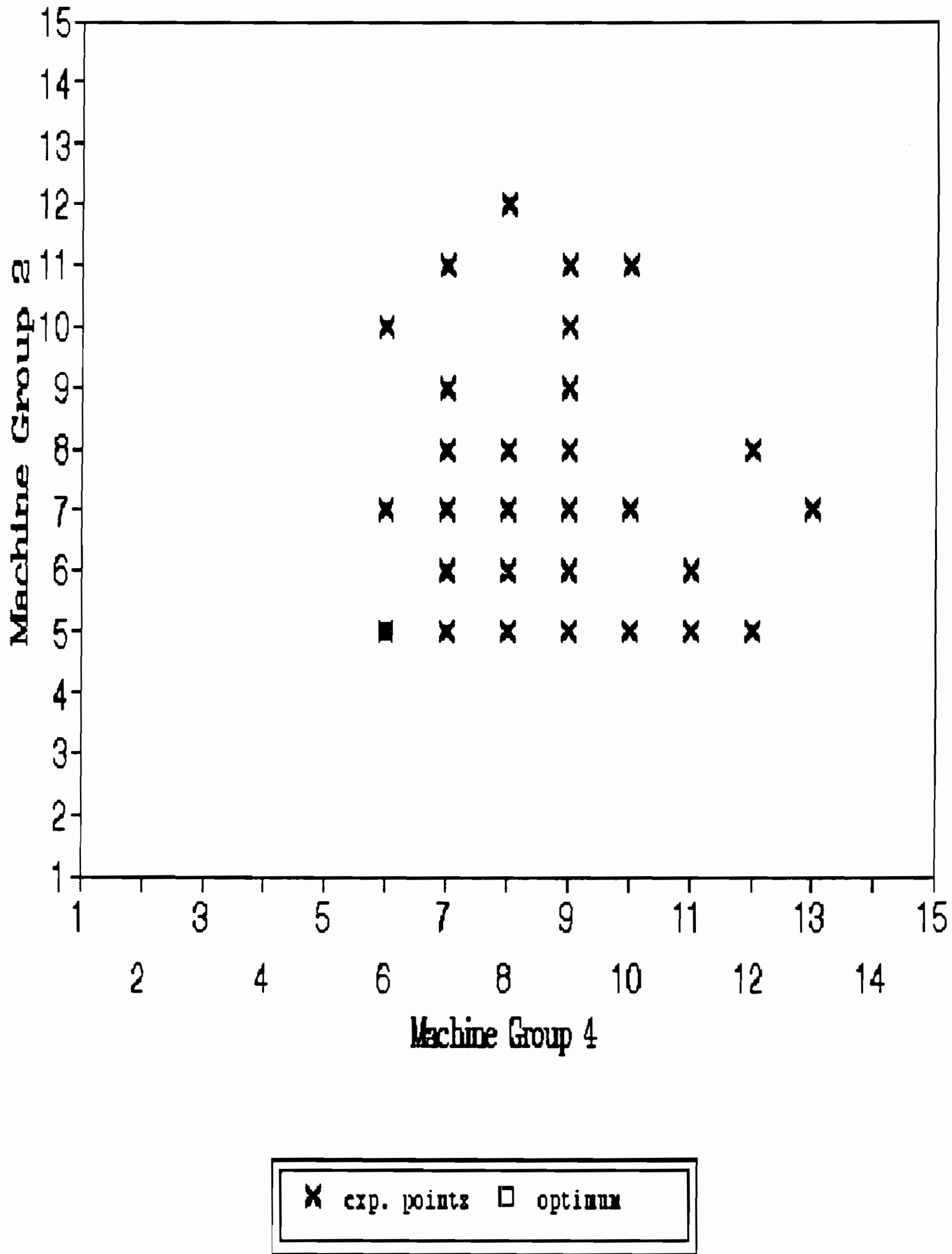



Figure 34-Final Points Job-Shop Response Surface Search Machine Groups (2,4)

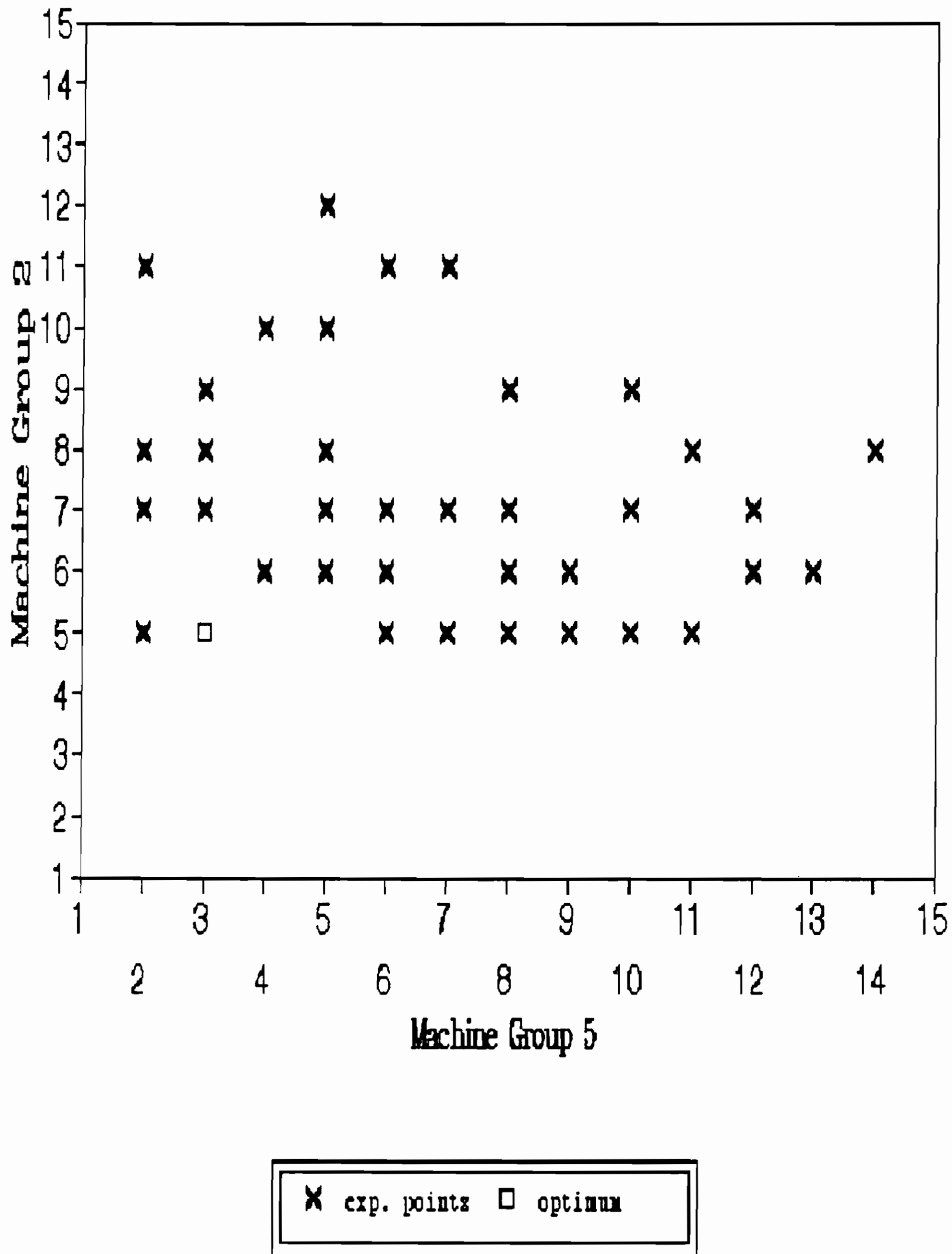



**Figure 35-Final Points Job-Shop Response Surface Search Machine Groups (2,5)**

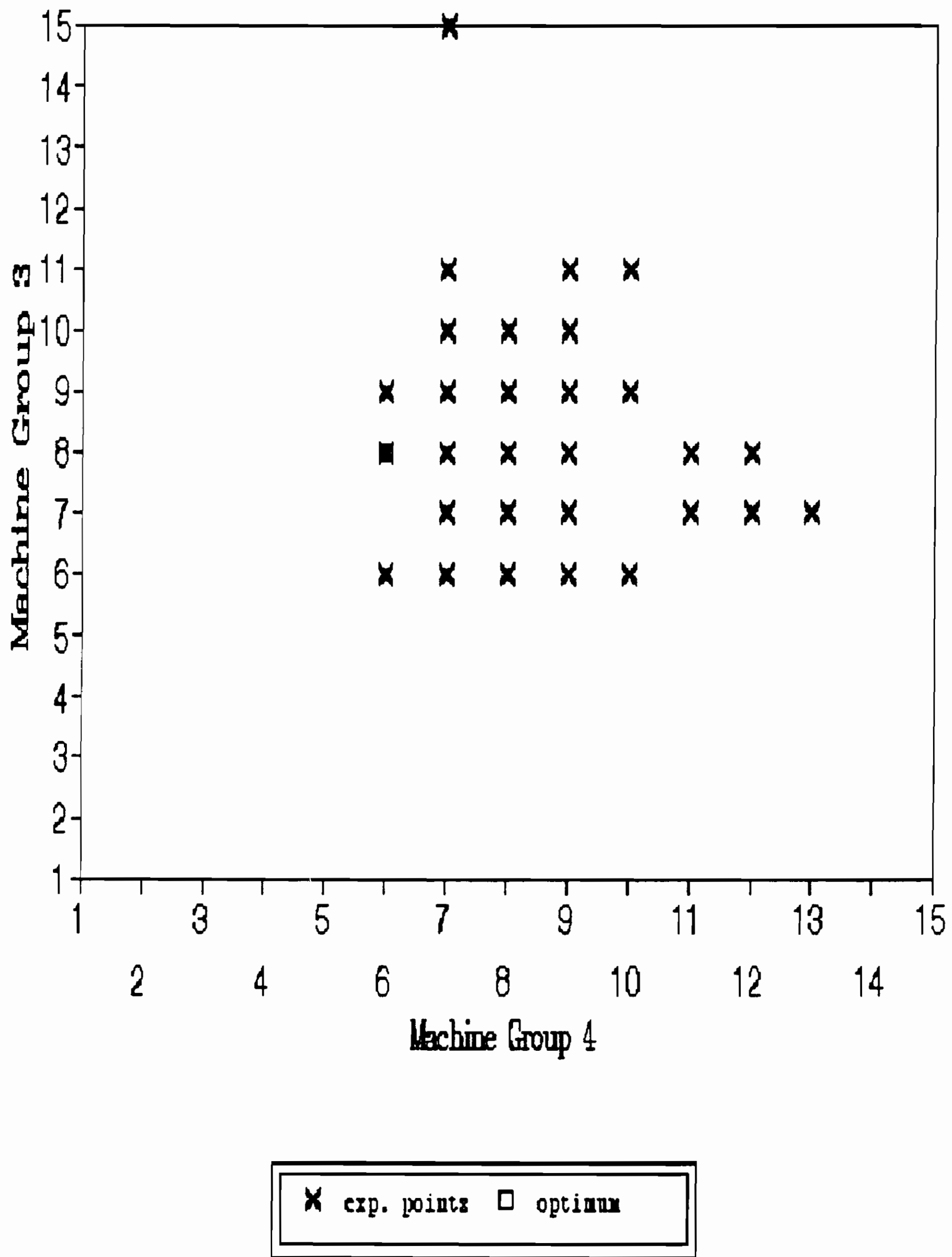



**Figure 36-Final Points Job-Shop Response Surface Search Machine Groups (3,4)**

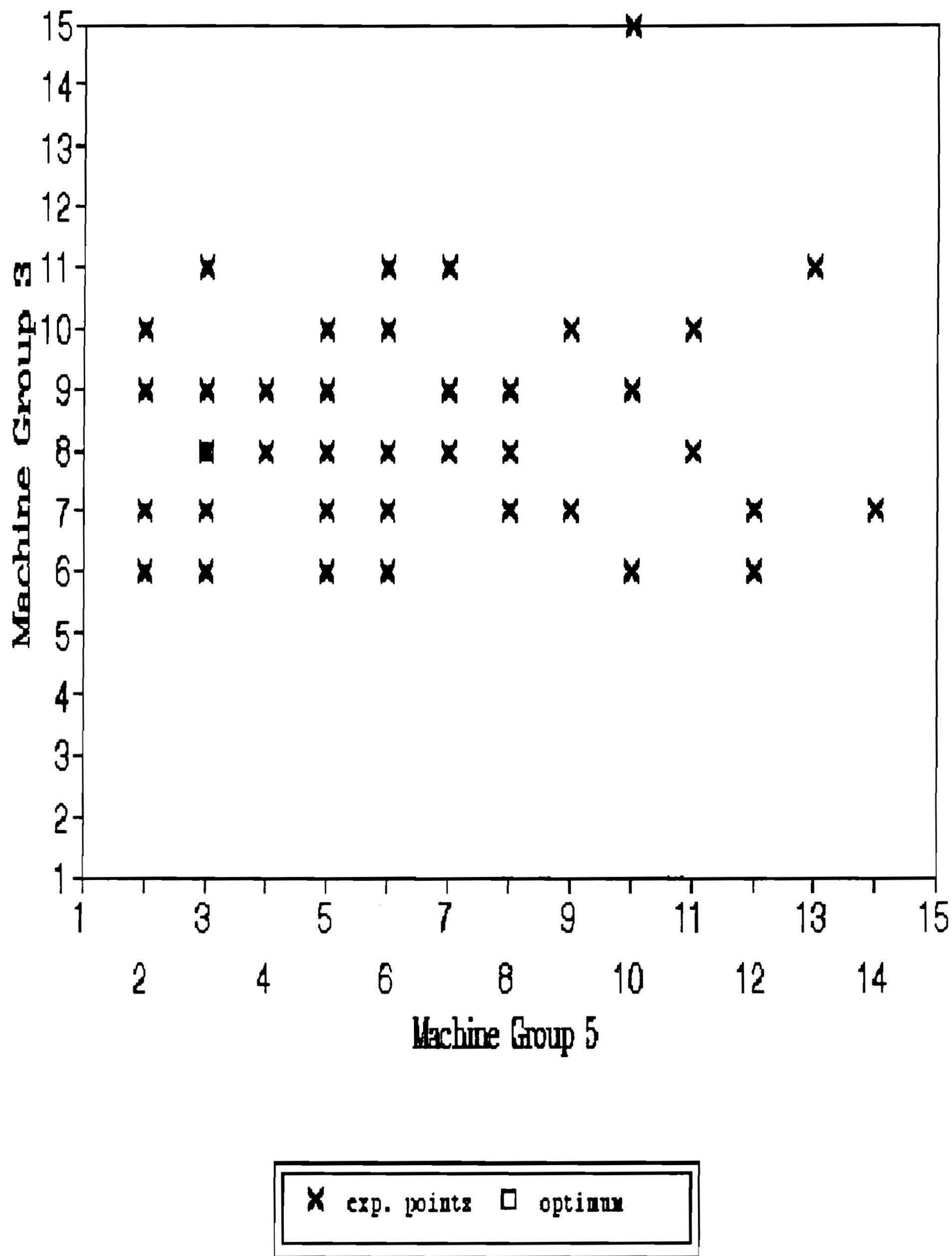



**Figure 37-Final Points Job-Shop Response Surface Search Machine Groups (3,5)**

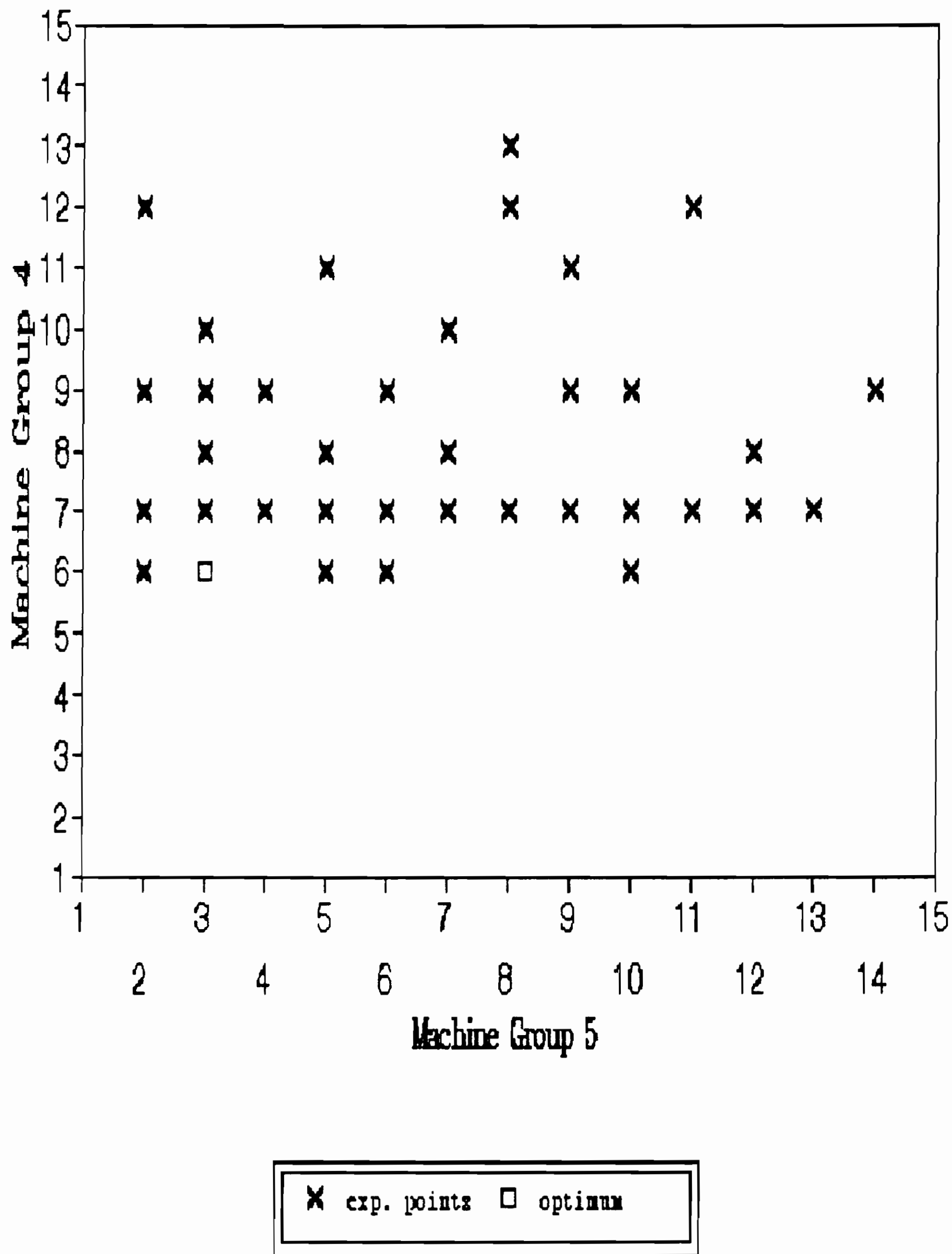



**Figure 38-Final Points Job-Shop Response Surface Search Machine Groups (4,5)**

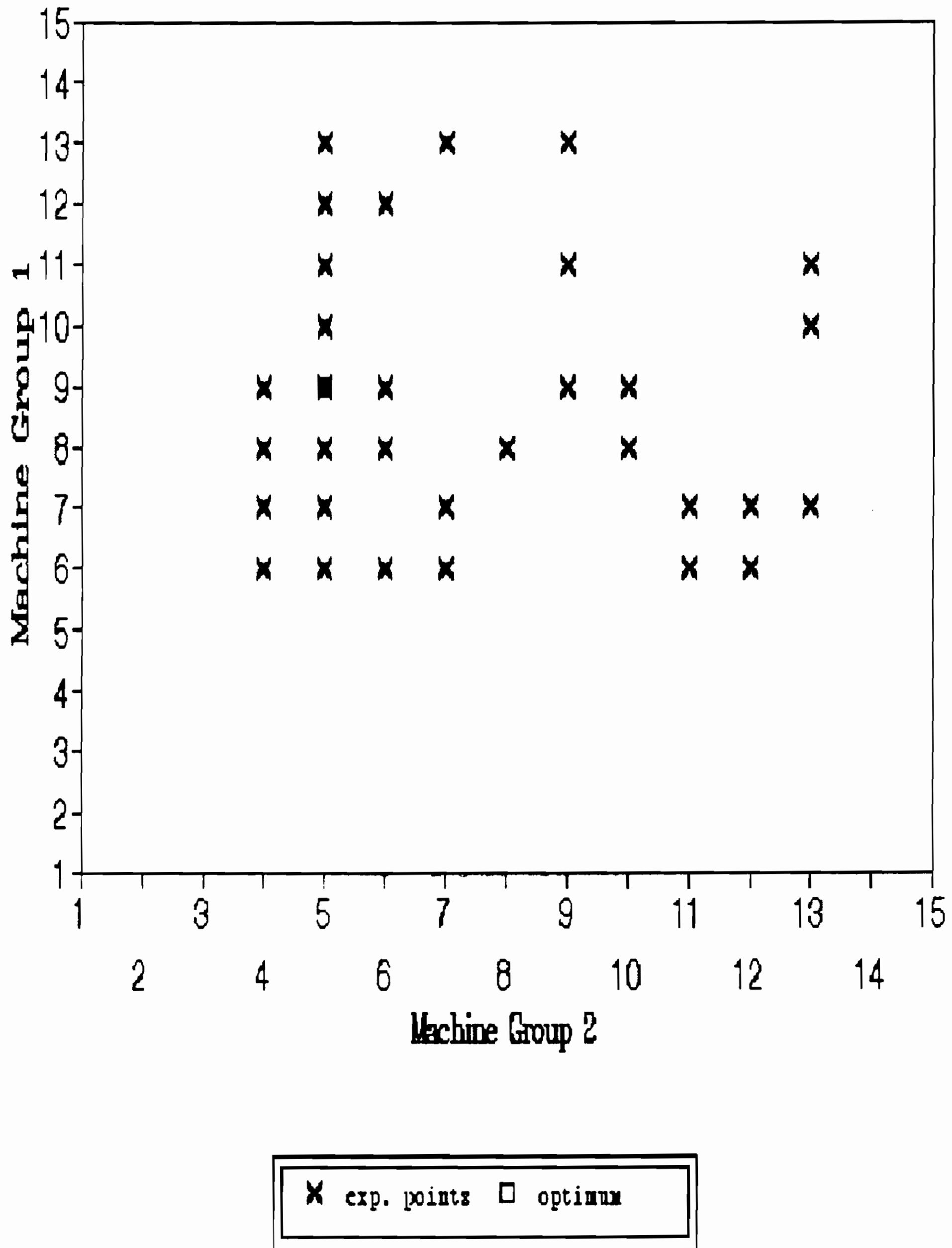



**Figure 39-Final Points Job-Shop Genetic Search Machine Groups (1,2)**

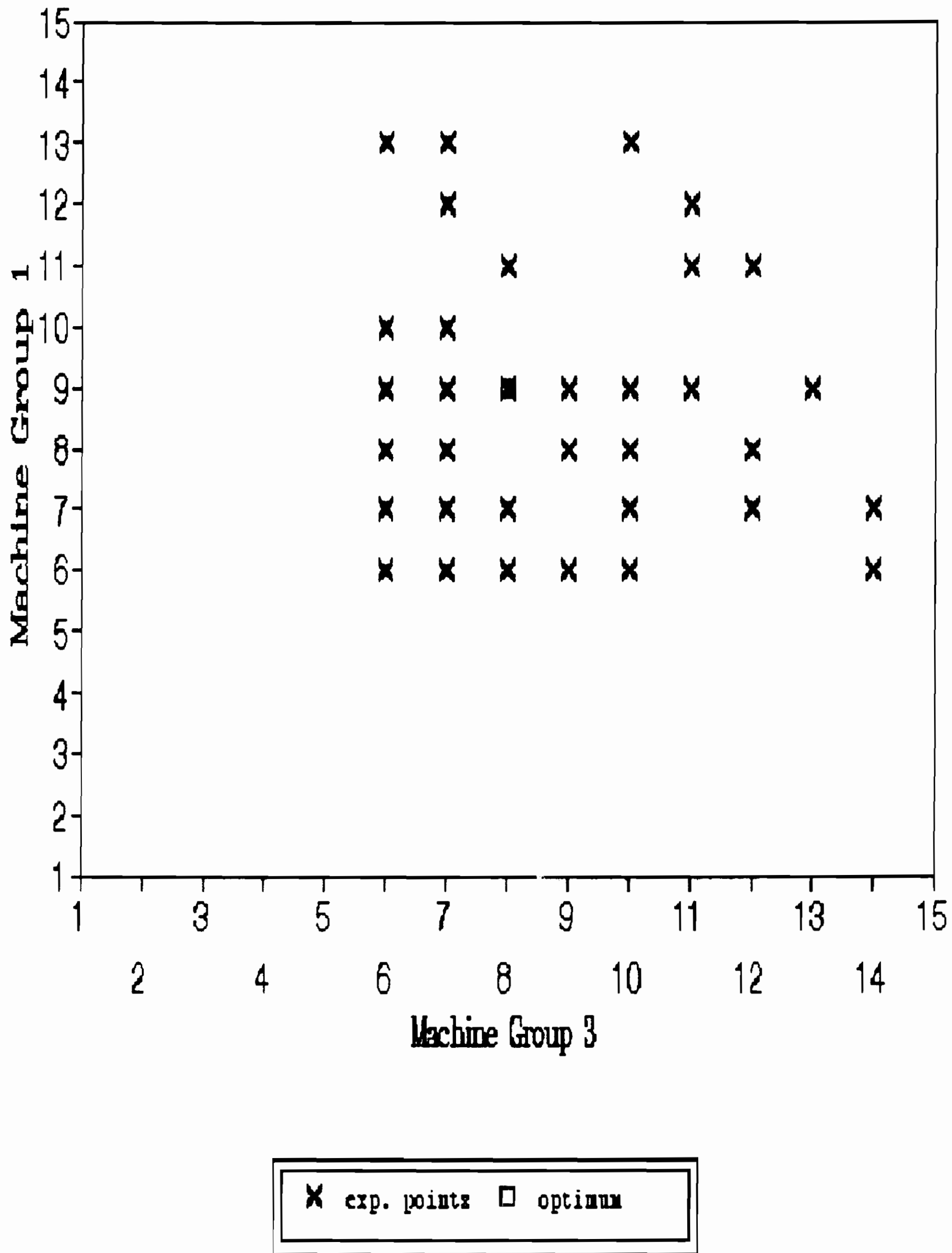



**Figure 40-Final Points Job-Shop Genetic Search Machine Groups (1,3)**

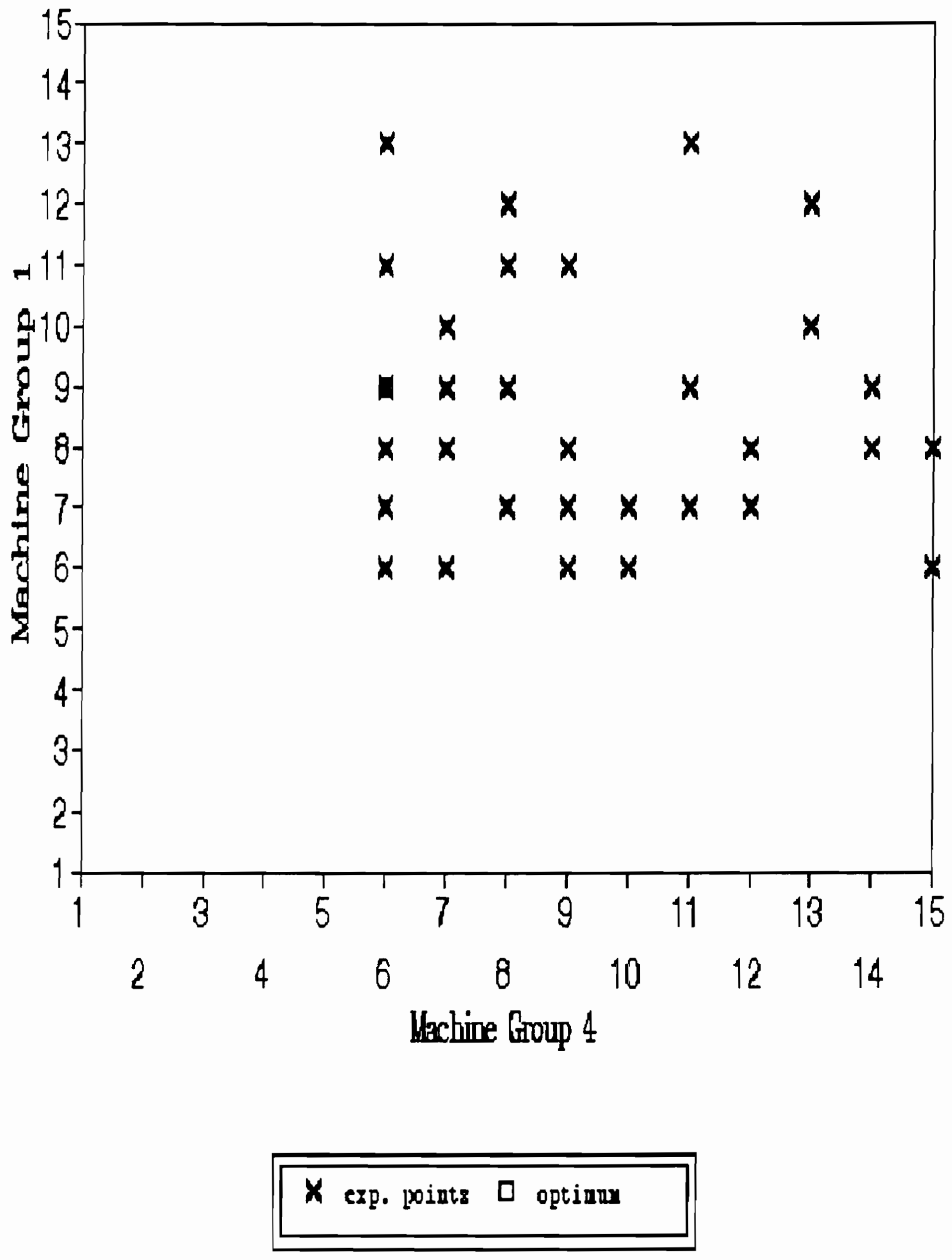



**Figure 41-Final Points Job-Shop Genetic Search Machine Groups (1,4)**

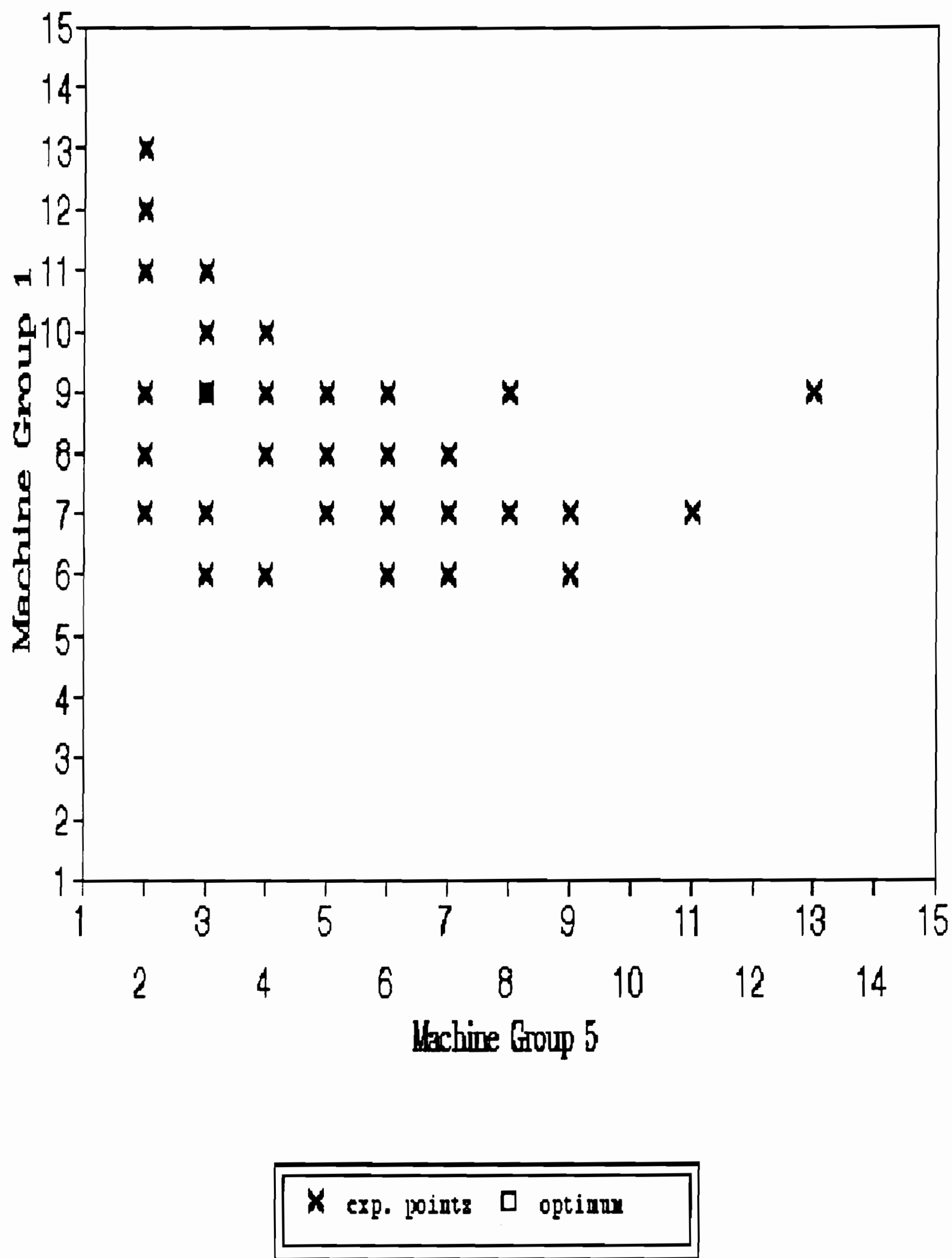



**Figure 42-Final Points Job-Shop Genetic Search Machine Groups (1,5)**

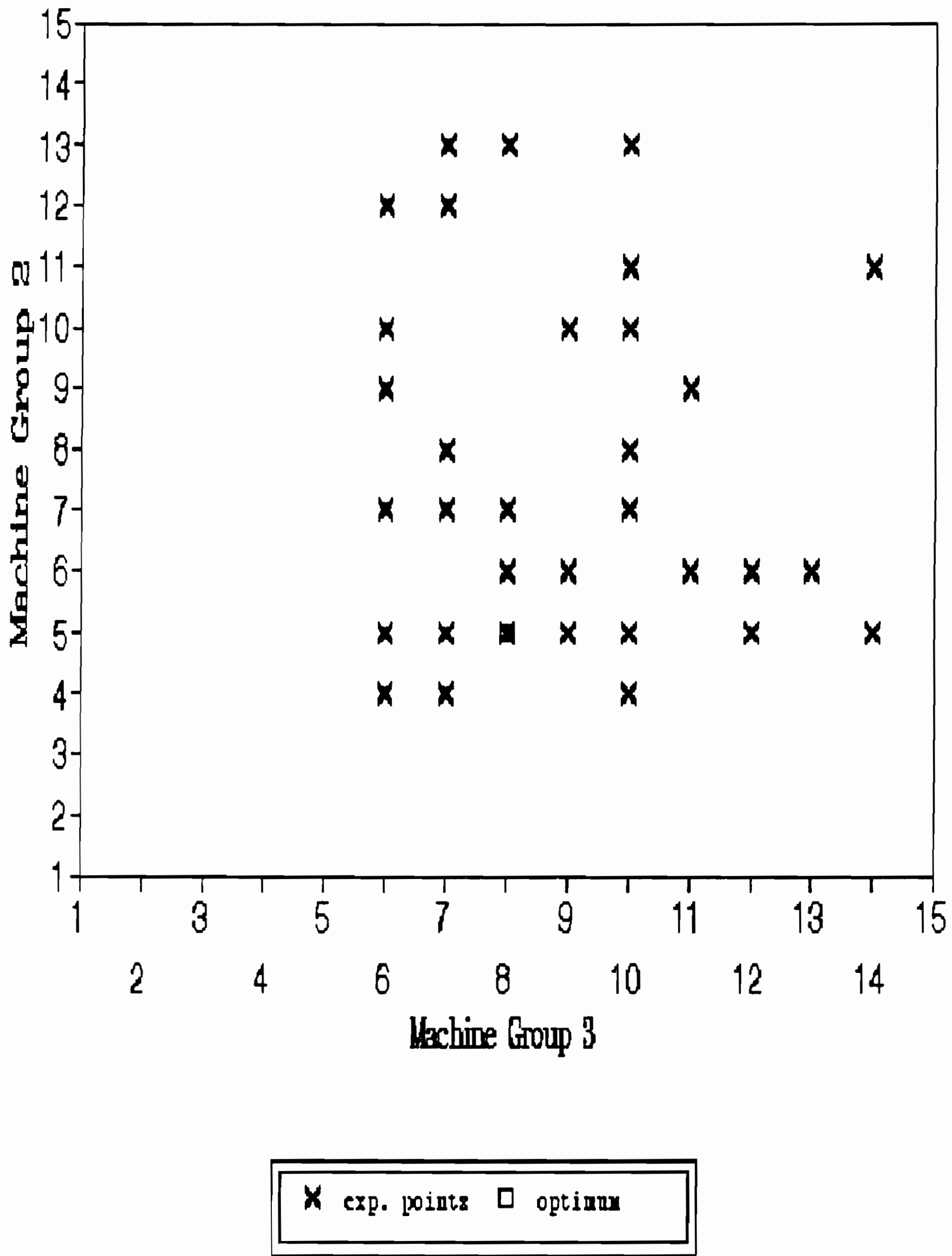



**Figure 43-Final Points Job-Shop Genetic Search Machine Groups (2,3)**

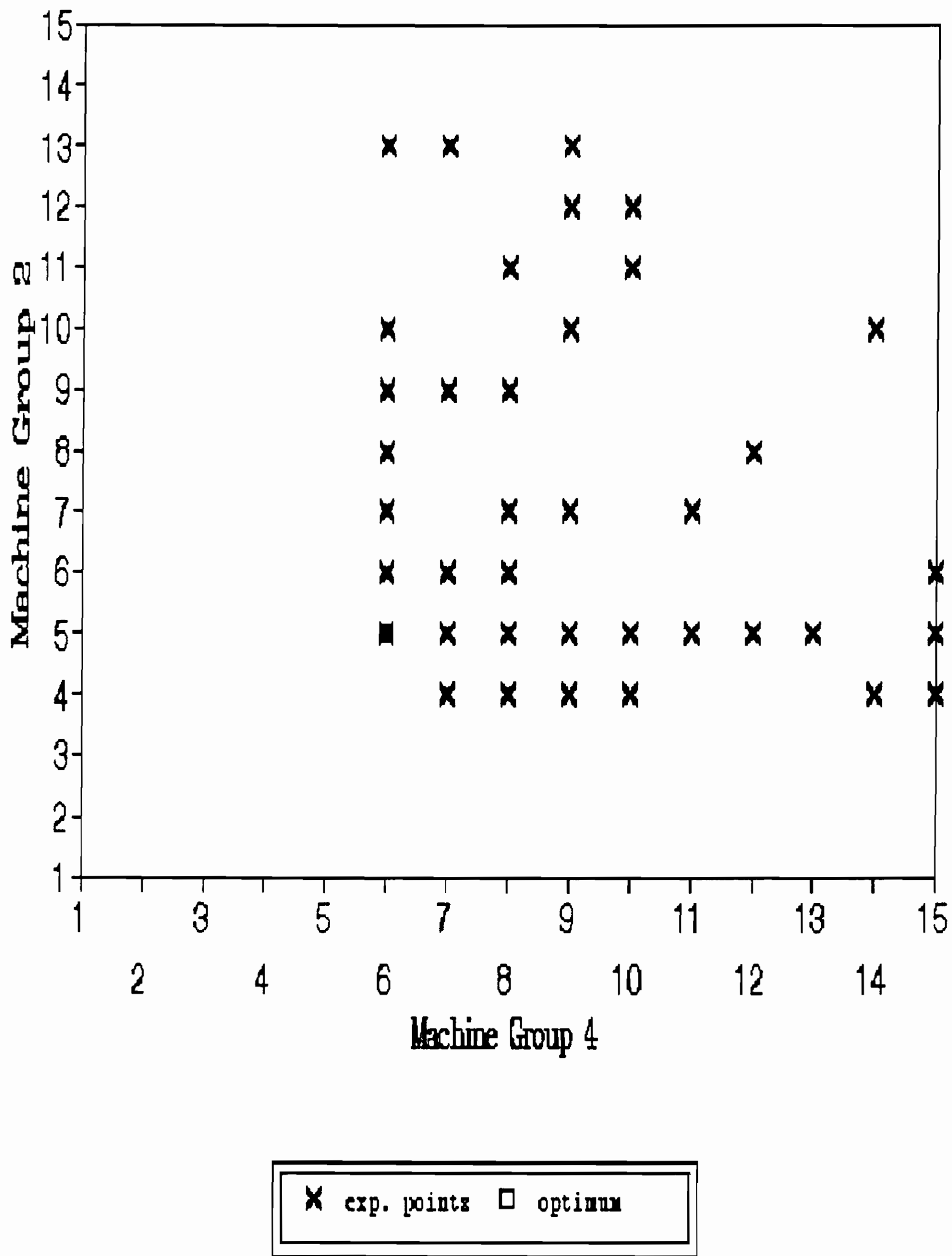



Figure 44-Final Points Job-Shop Genetic Search Machine Groups (2,4)

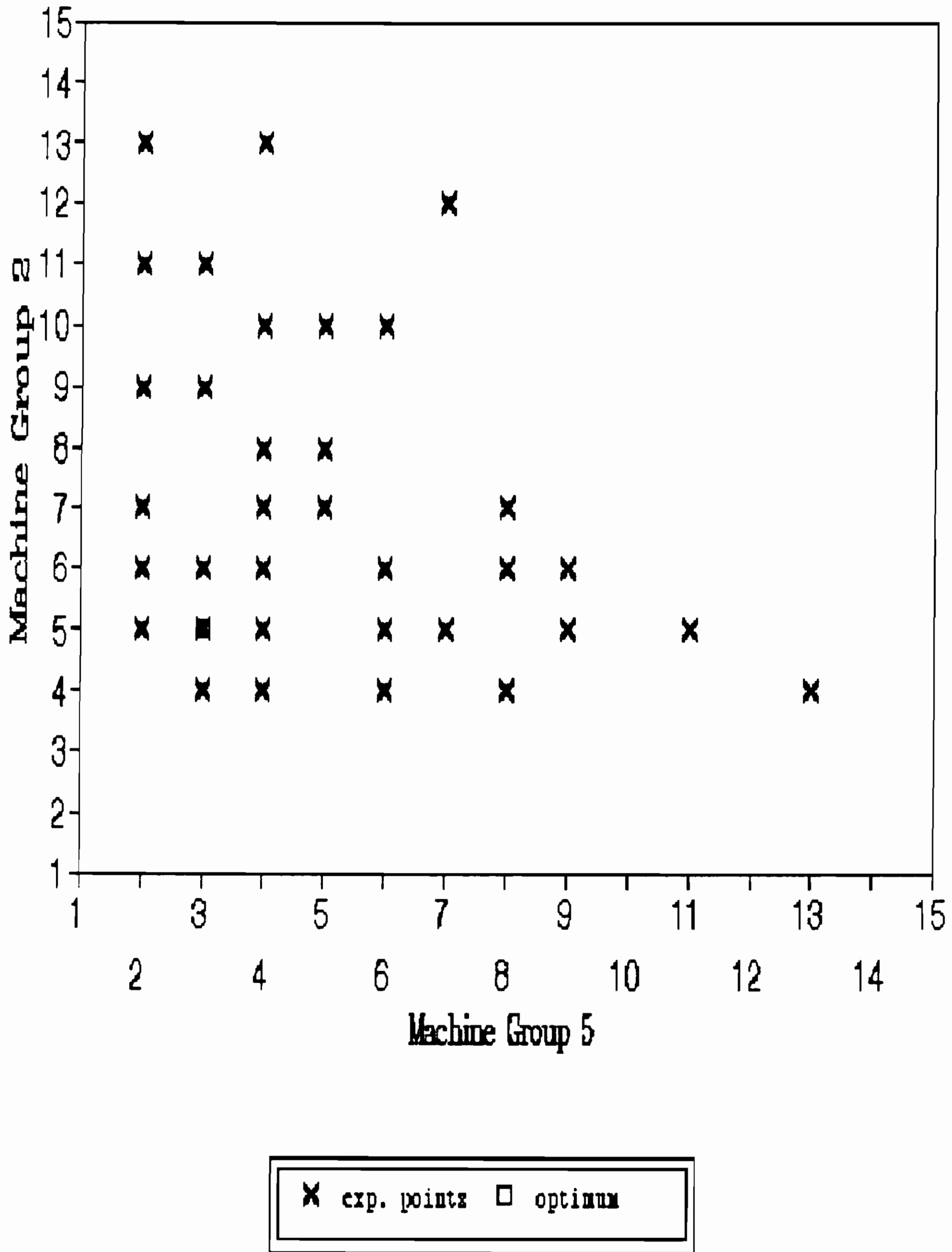



**Figure 45-Final Points Job-Shop Genetic Search Machine Groups (2,5)**

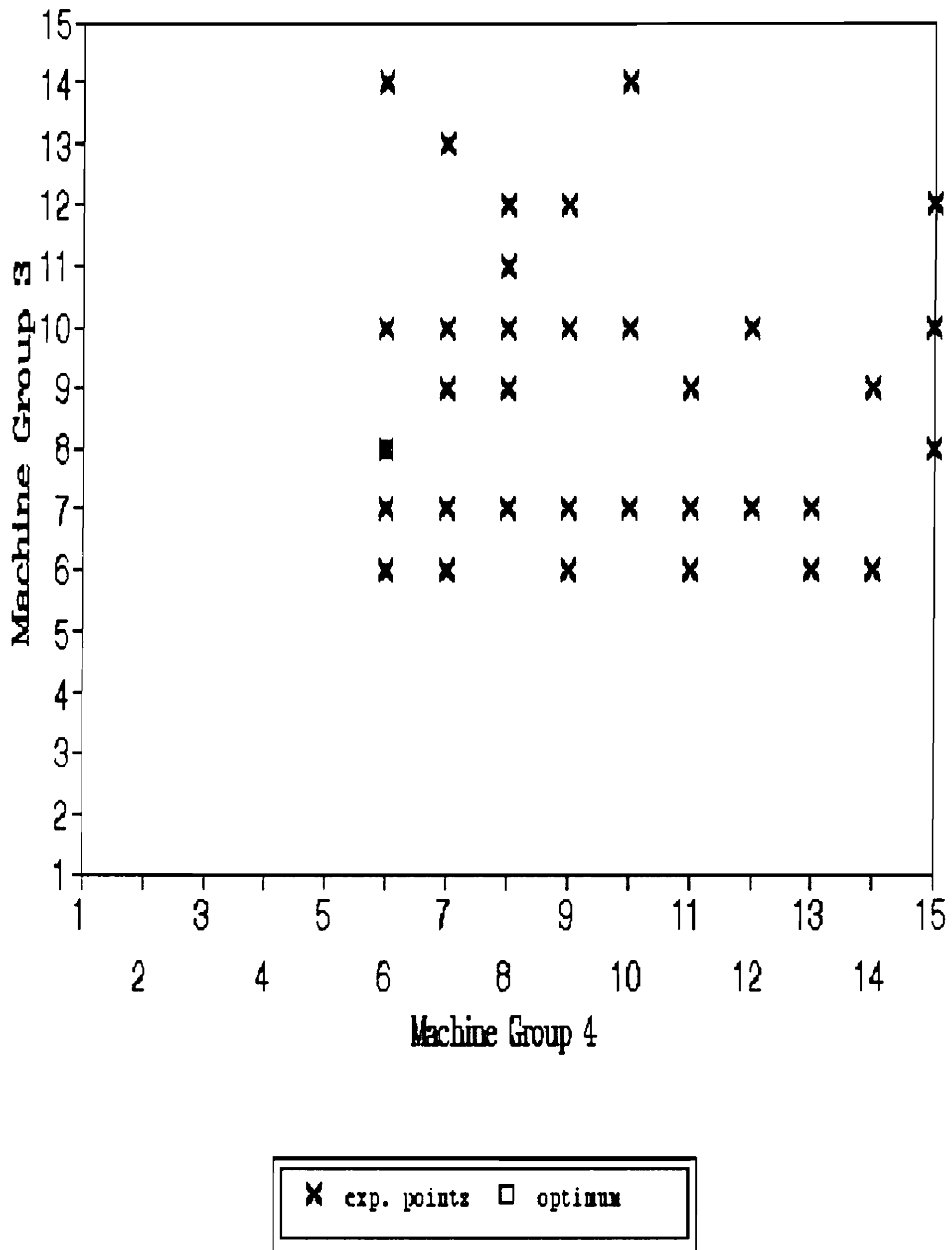



**Figure 46-Final Points Job-Shop Genetic Search Machine Groups (3,4)**

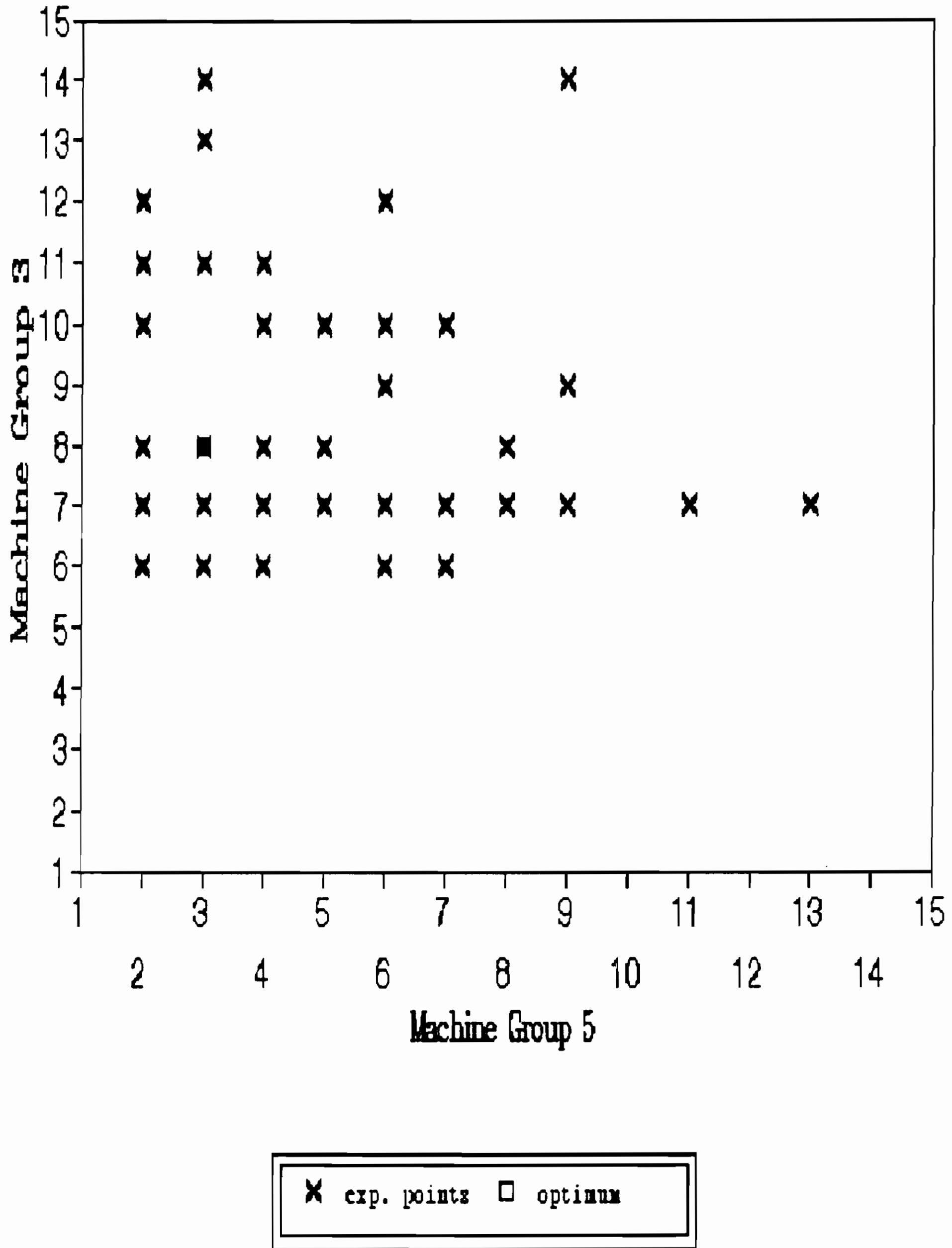



**Figure 47-Final Points Job-Shop Genetic Search Machine Groups (3,5)**

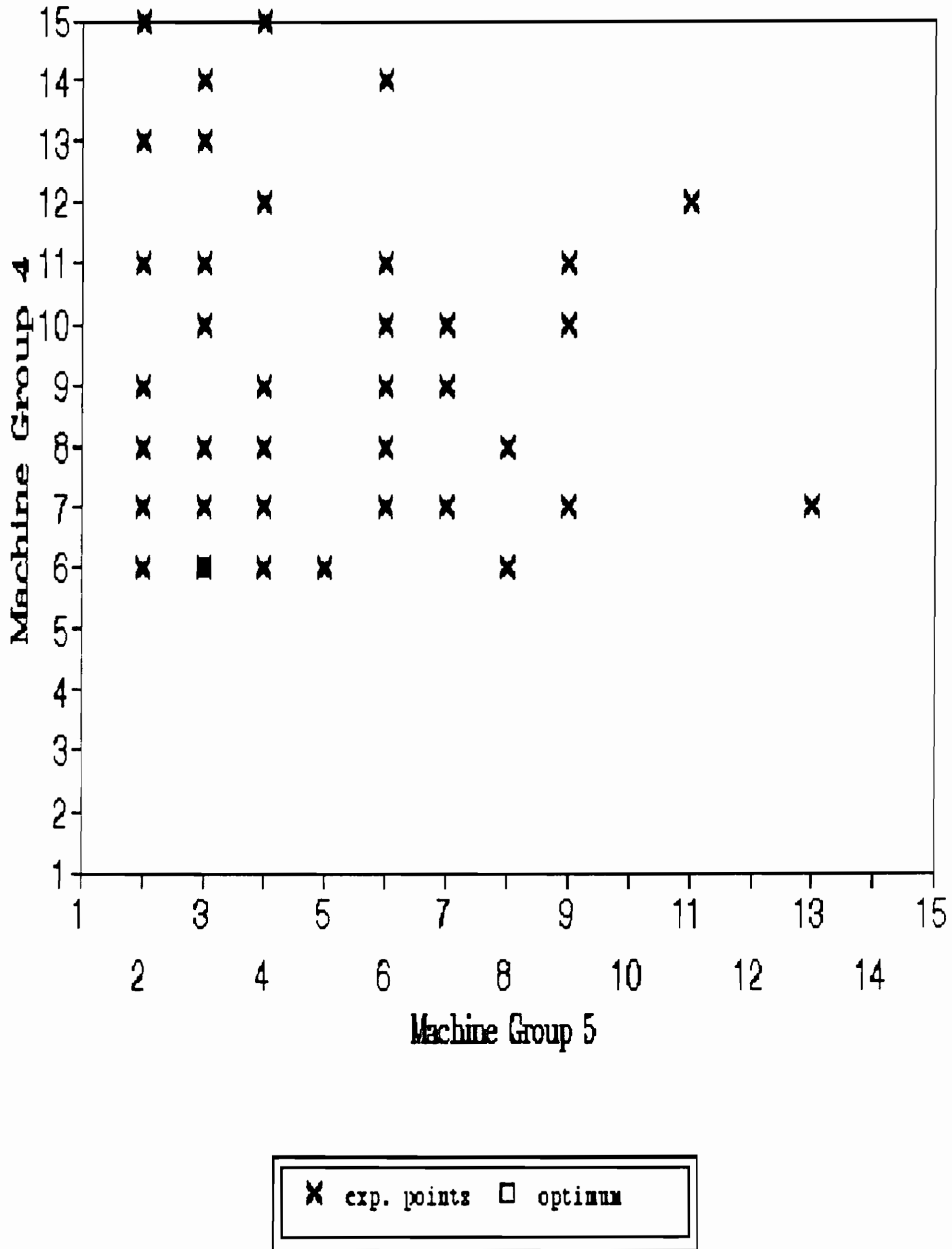



Figure 48-Final Points Job-Shop Genetic Search Machine Groups (4,5)

# Appendix 2

Prototype header file inv.h

```
void demand(void);
void evalu8(void);
double expon(double);
void init(void);
void ordarv(void);
void prgm_(void);
double randi(long*);
double rannd(long*);
void report(void);
void timing(void);
double unifrm(double,double);
void update(void);
```

File inv.in

```
60 120 9 4
0.10 32.0 3.0 1.0 5.0
```

```
0.167 0.500 0.833 1.000
20 40
20 60
20 80
20 100
40 60
40 80
40 100
60 80
60 100
```

File inv.c

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

#define TRUE 1
#define FALSE 0
typedef long LOGICAL32;
typedef char byte;
```

```
#define ADR(t,x)        ( t=(x), &t )
#define IS_ODD(j)       ((j) & 1 )

#include "INV.h"

struct t_model {
float aminus;
long int amount;
float aplus;
long int bigs;
float h, incrmc;
long int initil, invlev;
float mdemdt;
long int nevnts, next, nmnths;
float pi, setupc;
long int smalls;
float time, tlevnt, tne[4], tordc;
}model;
struct t_random {
long int nvalue;
float probd[25];
}random;
FILE *fp;

main( )
```

```
{
static long int i, i_, npolicy;

if ((fp = fopen("inv.in","r"))==NULL) {
 fprintf(stderr, "cannot open file inv.in\n");
  exit(1L);
}

/* *** SPECIFY THE NUMBER OF EVENT TYPES FOR THE TIMING ROUTINE
*/

model.nevnts = 4L;

/* *** READ INPUT PARAMETERS */

fscanf( fp, "%ld %ld %ld %ld ", &model.initil, &model.nmnths,
 &npolicy, &random.nvalue );
fscanf( fp, "%f %f %f %f %f ", &model.mdemdt, &model.setupc,
 &model.incrmc, &model.h, &model.pi );
fscanf( fp, "" );
for( i = 1L; i <= random.nvalue; i++ ){
fscanf( fp, "%f ", &random.probd[i - 1L] );
}

/* ******************************************************************** */
```

```
/* *** PRINT REPORT HEADING */

fprintf( stdout, "    SINGLE-PRODUCT INVENTORY SYSTEM\n" );
fprintf( stdout, "    INITIAL INVENTORY LEVEL                %3ld ITEMS\n",
 model.initil );
fprintf( stdout, "    NUMBER OF DEMAND SIZES                 %3ld\n",
 random.nvalue );
fprintf( stdout, "    DISTRIBUTION FUNCTION OF DEMAND SIZES    " );
for( i = 1L; i <= random.nvalue; i++ ){
fprintf( stdout, "%5.3f", random.probd[i - 1L] );
}
fprintf( stdout, "  \n" );
fprintf( stdout, "      MEAN INTERDEMAND TIME                  %5.2f
MONTHS\n",
 model.mdemdt );
fprintf( stdout, "      LENGTH OF THE SIMULATION               %3ld
MONTHS\n",
 model.nmnths );
fprintf( stdout, "    K = %5.1f   I = %5.1f   H = %5.1f   PI = %5.1f\n",
 model.setupc, model.incrmc, model.h, model.pi );
fprintf( stdout, "\n\n\n" );
fprintf( stdout, "      POLICY      AVERAGE COST      AVERAGE ORDERING
COST    AVERAGE HOLDING COST    AVERAGE SHORTAGE COST\n" );
```

```
/* ********************************************************************** */

/* *** RUN THE SIMULATION VARYING THE INVENTORY POLICY */

for( i = 1L; i <= npolicy; i++ ){
i_ = i - 1;

/* *** READ THE INVENTORY POLICY */

fscanf( fp, "%ld %ld ", &model.smalls, &model.bigs );

/* *** INITIALIZE THE SIMULATION */

init();

/* *** DETERMINE THE NEXT EVENT */

while( TRUE ){
timing();

/* *** CALL THE APPROPRIATE EVENT ROUTINE */

if( model.next == 2L ){
demand();
}
```

```
else if( model.next == 3L ){
goto L_10;
}
else if( model.next == 4L ){
evalu8();
}
else{
ordarv();
}
}
L_10:
report();
}
fprintf( stdout, "1\n" );

fclose(fp);

return 0;
} /* end of function */

void /*FUNCTION*/ init()
{

/* *** INITIALIZE THE SIMULATION CLOCK */
```

```
model.time = 0L;

/* *** INITIALIZE THE STATE VARIABLES */

model.invlev = model.initil;
model.tlevnt = 0.;

/* *** INITIALIZE THE STATISTICAL COUNTERS */

model.tordc = 0.;
model.aplus = 0.;
model.aminus = 0.;

/* *** INITIALIZE THE EVENT LIST. SINCE NO ORDER IS OUSTANDING, THE
TIME
 * *** OF THE NEXT ORDER ARRIVAL IS SET TO INFINITY. */

model.tne[0L] = 1.e30;
model.tne[1L] = expon( model.mdemdt );
model.tne[2L] = model.nmnths;
model.tne[3L] = 0.;
return;
} /* end of function */

void /*FUNCTION*/ timing()
```

```
{
static long int i, i_;
static float rmin;

rmin = 1.e29;
model.next = 0L;

/* *** DETERMINE THE EVENT TYPE OF THE NEXT EVENT TO OCCUR */

for( i = 1L; i <= model.nevnts; i++ ){
i_ = i - 1;
if( model.tne[i_] < rmin ){
rmin = model.tne[i_];
model.next = i;
}
}

/* *** IF THE EVENT IS EMPTY (I.E., NEXT=0), STOP THE SIMULATION.
 * *** OTHERWISE, ADVANCE THE SIMULATION CLOCK */

if( model.next > 0L ){
model.time = model.tne[model.next - 1L];
return;
}
else{
```

```
fprintf( stdout, "1      EVENT LIST EMPTY\n" );
exit(0);
}
return;
} /* end of function */



void /*FUNCTION*/ ordarv()
{


/* *** UPDATE 'APLUS' AND 'AMINUS' */
update();


/* *** INCREMENT THE INVENTORY LEVEL BY TEH AMOUNT ORDERED. */


model.invlev = model.invlev + model.amount;


/* *** SINCE NO ORDER IS NOW OUTSTANDING. SET THE TIME OF THE
NEXT ORDER
 * *** ARRIVAL TO INFINITY */


model.tne[0L] = 1.e30;
return;
} /* end of function */
```

```
void /*FUNCTION*/ demand()
{
static long int dsize, z;

/* *** UPDATE 'APLUS' AND 'AMINUS' */
update();

/* *** GENERATE THE DEMAND SIZE */

dsize = randi( &z );

/* *** DECREMENT THE INVENTORY LEVEL BY THE DEMAND SIZE. */

model.invlev = model.invlev - dsize;

/* *** SCHEDULE THE NEXT DEMAND */

model.tne[1L] = model.time + expon( model.mdemdt );
return;
} /* end of function */

void /*FUNCTION*/ evalu8()
{

/* *** IF THE INVENTORY LEVEL IS LESS THAN 'SMALLS,'PLACE AN ORDER
```

```
FOR
 * *** 'BIGS'-'INVLEV'ITEMS */
if( model.invlev < model.smalls ){
model.amount = model.bigs - model.invlev;
model.tordc = model.tordc + model.setupc + (model.incrmc*model.amount);

/* *** SCHEDULE THE ARRIVAL OF THE ORDER */

model.tne[0L] = model.time + unifrm( .5, 1. );
}

/* *** SCHEDULE THE NEXT INVENTORY EVALUATION */

model.tne[3L] = model.time + 1.;
return;
} /* end of function */


void /*FUNCTION*/ report()
{
static float acost, ahldc, aordc, ashrc;

/* *** UPDATE 'APLUS' AND 'AMINUS' */
update();
```

```
/* *** COMPUTE THE ESTIMATES OF THE DESIRED MEASURES OF
PERFORMANCE */

aordc = model.tordc/model.nmnths;
ahldc = model.h*(model.aplus/model.nmnths);
ashrc = model.pi*(model.aminus/model.nmnths);
acost = aordc + ahldc + ashrc;
fprintf( stdout, "O     (%3ld,%3ld)     %6.2f          %6.2f          %6.2f
%6.2f\n",
 model.smalls, model.bigs, acost, aordc, ahldc, ashrc );
return;
} /* end of function */

void /*FUNCTION*/ update()
{
static float tsle;

/* *** COMPUTE THE TIME SINCE THE LAST EVENT WHICH CHANGED THE
INTVENTORY
 * *** LEVEL */
tsle = model.time - model.tlevnt;
model.tlevnt = model.time;

/* *** DETERMINE WHETHER THE INVENTORY LEVEL DURING THE
PREVIOUS INTERVAL
```

```
 * *** WAS NEGATIVE, ZERO OR POSITIVE */

if( model.invlev != 0L ){
if( model.invlev > 0L ){

/* *** SINCE THE INVENTORY LEVEL DURING THE PREVIOUS INTERVAL
WAS
 * *** POSITIVE, UPDATE 'APLUS' */

model.aplus = model.aplus + (model.invlev*tsle);
}
else{

/* *** SINCE THE INVENTORY LEVEL DURING THE PREVIOUS INTERVAL
WAS
 * *** NEGATIVE, UPDATE 'AMINUS.' */

model.aminus = model.aminus + (-model.invlev*tsle);

/* *** THE INVENTORY LEVEL DURING THE PREVIOUS INTVERVAL WAS
ZERO. */

}
}
return;
```

```
} /* end of function */


double /*FUNCTION*/ randi(z)
long int *z;
{
static long int i, i_, n1;
static float randi_v, u;

/* *** GENERATE A U(0,1) RANDOM VARIABLE. THE FORM OF THIS
STATEMENT
 * *** DEPENDS ON THE COMPUTER USED */
u = rannd( z );

/* *** GENERATE A RANDOM INTEGER BETWEEN 1 AND NVALUE IN
ACCORDANCE WITH
 * *** DISTRIBUTION FUNCTION 'PROBD' */

n1 = random.nvalue - 1L;
for( i = 1L; i <= n1; i++ ){
i_ = i - 1;
if( u < random.probd[i_] )
goto L_10;
}
randi_v = random.nvalue;
```

```
return( randi_v );
L_10:
randi_v = i;
return( randi_v );
} /* end of function */

double /*FUNCTION*/ unifrm(a, b)
double a, b;
{
static long int z;
static float u, unifrm_v;

/* *** GENERATE A U(0,1) RANDOM VARIABLE. THE FORM OF THIS
STATEMENT
 * *** DEPENDS ON THE COMPUTER USED */
u = rannd( &z );

/* *** GENERATE A U(A,B) RANDOM VARIABLE */

unifrm_v = a + (u*(b - a));
return( unifrm_v );
} /* end of function */

double /*FUNCTION*/ expon(rmean)
double rmean;
```

```
{
static long int z;
static float expon_v, u;

/* *** GENERATE A U(0,1) RANDOM VARIABLE. THE FORM OF THE
STATEMENT
 * *** DEPENDS ON THE COMPUTER USED. */
u = rannd( &z );

/* *** GENERATE A EXPONENTIAL RANDOM VARIABLE WITH MEAN
RMEAN */

expon_v = -rmean*log( u );
return( expon_v );
} /* end of function */


double /*FUNCTION*/ rannd(key)
long int *key;
{
static long int fhi, k, leftlo, xalo, xhi;
static float rannd_v;
static long a = 16807;
static long b15 = 32768;
static long b16 = 65536;
```

```
static long p = 2147483647;
static long ix = 12345678;

/* *** Prime modulus multiplicative linear congruential generator
 * *** Z(I) = ((7**5) * Z(I - 1)) (MOD(2**31 - 1))
 * *** based on Schrage's portable random number generator RAND.
 * *** (See Chapter 6 for more datail.) */
/* *** Set the default seed for the random number generator. */
/* *** Determine desired action. */
if( *key == 0L ){

/* *** Generate the next random number. */

xhi = ix/b16;
xalo = (ix - xhi*b16)*a;
leftlo = xalo/b16;
fhi = xhi*a + leftlo;
k = fhi/b15;
ix = (((xalo - leftlo*b16) - p) + (fhi - k*b15)*b16) + k;
if( ix < 0L )
ix = ix + p;
rannd_v = (float)( ix )*4.656612875e-10;
}
else if( *key > 0L ){
```

```
/* *** Set the current IX to KEY. */

ix = *key;
}
else{

/* *** Return current IX in KEY. */

*key = ix;
}
return( rannd_v );
} /* end of function */
```

# Appendix 3

Protoype file comp.h

```
void arrive(double);
void cancel(double);
void endrun(void);
double expon(double);
void file(long,long);
void filest(long);
void initlk(void);
void prgm_(void);
double rannd(long*);
void removve(long,long);
void report(void);
void sampst(double,long);
void start(void);
void timest(double,long);
void timing(void);
double unifrm(double,double);
```

FILE comp.in

```
10 80 10
5000
25 0.80
0.10 0.015
```

File comp.c

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

#define TRUE 1
#define FALSE 0
typedef long LOGICAL32;
typedef char byte;
#define ADR(t,x)        ( t=(x), &t )
#define IS_ODD(j)       ((j) & 1 )

#define DOCNT(i,t,n)    (_d_l=(n), (_d_m=(t-(i)+_d_l)/_d_l) > 0 ? _d_m : 0L )

#include "COMP.h"
```

```
struct t_model {
float mserve, mthink;
long int nterml, numjob;
float overhd, quantm;
long int totjob;
}model;
struct t_simlib {
long int ldecr, levent, lfirst, lincr, llast, lrank[25], lsize[25],
 maxatr, next;
float time, trnsfr[10];
}simlib;
FILE *fp;

main( )
{
static long int _d_l, _d_m, _do0, _do1, increm, maxter, minter, nterml_,
 termnl, termnl_;

if ((fp = fopen("comp.in","r"))==NULL) {
 fprintf(stderr,"cannot open file comp.in\n");
 exit(1);
 }


/* *** READ INPUT PARAMETERS. */
```

```
fscanf( fp, "%ld %ld %ld ", &minter, &maxter, &increm );
fscanf( fp, "%ld ", &model.totjob );
fscanf( fp, "%f %f ", &model.mthink, &model.mserve );
fscanf( fp, "%f %f ", &model.quantm, &model.overhd );

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

/* *** PRINT REPORT HEADING. */

fprintf( stdout, " Time-shared computer model\n\n Number of terminals%9ld to%4ld
by%4ld\n\n Mean think time%13.3f seconds\n\n Mean service time%11.3f seconds\n\n
Quantum%21.3f seconds\n\n Overhead%20.3f seconds\n\n Number of jobs
processed%12ld\n\n\n Number of Average Average Utilization\n
terminals response time number in queue of CPU\n",
 minter, maxter, increm, model.mthink, model.mserve, model.quantm,
 model.overhd, model.totjob );

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

/* *** RUN THE SIMULATION VARYING THE TERMINALS. */

for( model.nterml = minter, _do0=DOCNT(model.nterml,maxter,_do1 = increm);
_do0 > 0; model.nterml += _do1, _do0-- ){
nterml_ = model.nterml - 1;
```

```
model.numjob = 0L;

/* *** INITIALIZE SUPPORT ROUTINES. */

initlk();
simlib.maxatr = 4L;

/* *** SCHEDULE THE FIRST ARRIVAL TO THE CPU FROM EACH
TERMINAL. */

for( termnl = 1L; termnl <= model.nterml; termnl++ ){
termnl_ = termnl - 1;
simlib.trnsfr[0L] = expon( model.mthink );
simlib.trnsfr[1L] = 1.;
simlib.trnsfr[2L] = termnl;
file( 3L, 25L );
}

/* *** DETERMINE THE NEXT EVENT. */

while( TRUE ){
timing();

/* *** CALL THE APPROPRIATE EVENT ROUTINE. */
```

```
if( simlib.next == 2L ){
endrun();
}
else if( simlib.next == 3L ){
goto L_10;
}
else{
arrive( simlib.trnsfr[2L] );
}
}
L_10:
report();
}
fprintf( stdout, "\n" );
fclose (fp);
return 0;
} /* end of function */

void /*FUNCTION*/ arrive(origin)
double origin;
{
static long int int_;
static float jobtim;

/* *** PLACE THE JOB IN THE QUEUE.
```

```
 * *** NOTE THAT THE FOLLOWING DATA ARE STORED FOR EACH JOB:
 * ***      1. TIME OF ARRIVAL TO THE QUEUE.
 * ***      2. THE TERMINAL OF ORIGIN.
 * ***      3. THE NUMBER OF QUANTUM LENGTH CPU RUNS REQUIRED.
 * ***      4. THE REMAINING AMOUNT OF SERVICE. */
jobtim = expon( model.mserve );
simlib.trnsfr[0L] = simlib.time;
simlib.trnsfr[1L] = origin;
int_ = jobtim/model.quantm;
simlib.trnsfr[2L] = int_;
simlib.trnsfr[3L] = jobtim - (int_*model.quantm);
file( 2L, 1L );

/* *** IF THE CPU IS IDLE, START A CPU RUN. */

if( simlib.lsize[1L] == 0L )
start();
return;
} /* end of function */

void /*FUNCTION*/ start()
{
static float runtim;

/* *** REMOVE JOB FROM QUEUE */
```

```
removve( 1L, 1L );

/* *** DETERMINE REQUIRED CPU TIME FOR THIS RUN. */

if( simlib.trnsfr[2L] > 0.5 ){

/* *** A FULL QUANTM IS NEEDED. */

runtim = model.quantm + model.overhd;
simlib.trnsfr[2L] = simlib.trnsfr[2L] - 1.;
}
else{

/* *** LESS THAN A FULL QUANTM IS NEEDED. */

runtim = simlib.trnsfr[3L] + model.overhd;
simlib.trnsfr[2L] = -1.;
}

/* *** PLACE JOB IN THE CPU. */

file( 1L, 2L );

/* *** SCHEDULE THE END OF THE CPU RUN. */
```

```
simlib.trnsfr[0L] = simlib.time + runtim;
simlib.trnsfr[1L] = 2.;
file( 3L, 25L );
return;
} /* end of function */

void /*FUNCTION*/ endrun()
{
static float origin, resptm;

/* *** REMOVE JOB FROM THE CPU */
removve( 1L, 2L );

/* *** IF THIS JOB IS DONE, SCHEDULE ANOTHER ARRIVAL FOR THE SAME
TERMINAL. */

if( simlib.trnsfr[2L] > -0.5 ){

/* *** SINCE THE JOB IS NOT FINISHED, PLACE IT AT THE END OF THE
QUEUE */

file( 2L, 1L );
start();
}
else{
```

```
resptm = simlib.time - simlib.trnsfr[0L];
sampst( resptm, 1L );
origin = simlib.trnsfr[1L];
simlib.trnsfr[0L] = simlib.time + expon( model.mthink );
simlib.trnsfr[1L] = 1.;
simlib.trnsfr[2L] = origin;
file( 3L, 25L );

/* *** INCREMENT THE NUMBER OF COMPLETED JOBS. IF ENOUGH JOBS
ARE DONE,
 * *** SCHEDULE THE END OF THE SIMULATION */

model.numjob = model.numjob + 1L;
if( model.numjob >= model.totjob ){
simlib.trnsfr[0L] = simlib.time;
simlib.trnsfr[1L] = 3.;
file( 1L, 25L );

/* *** IF THE QUEUE IS NOT EMPTY, START ANOTHER JOB. */

}
else if( simlib.lsize[0L] > 0L ){
start();
}
}
```

```
return;
} /* end of function */


void /*FUNCTION*/ report()
{
static float arespt, avgniq, utiliz;


sampst( 0., -1L );
arespt = simlib.trnsfr[0L];
filest( 1L );
avgniq = simlib.trnsfr[0L];
filest( 2L );
utiliz = simlib.trnsfr[0L];
fprintf( stdout, "\n%6ld%16.3f%16.3f%16.3f\n", model.nterml, arespt,
 avgniq, utiliz );
return;
} /* end of function */



struct t_llists {
long int head[25], iout, linkpr[1000], linksr[1000];
float master[10][1000];
long int nar, tail[25];
}llists;
```

```
void /*FUNCTION*/ initlk()
{
static long int i, i_, list, list_, row, row_;

/* *** Initialize links. */
for( row = 1L; row <= 1000L; row++ ){
row_ = row - 1;
llists.linkpr[row_] = 0L;
llists.linksr[row_] = row + 1L;
}
llists.linksr[999L] = 0L;

/* *** Initialize list attributes. */

for( list = 1L; list <= 25L; list++ ){
list_ = list - 1;
llists.head[list_] = 0L;
llists.tail[list_] = 0L;
simlib.lsize[list_] = 0L;
simlib.lrank[list_] = 0L;
}

/* *** Initialize the TRANSFR */

for( i = 1L; i <= 10L; i++ ){
```

```
i_ = i - 1;
simlib.trnsfr[i_] = 0.0;
}

/* *** Initialize mnemonics for record location in lists. */

simlib.lfirst = 1L;
simlib.llast = 2L;
simlib.lincr = 3L;
simlib.ldecr = 4L;

/* *** Initialize mnemonic for event list number. */

simlib.levent = 25L;

/* *** Initialize system attributes. */

simlib.time = 0.;
llists.nar = 1L;
simlib.lrank[simlib.levent - 1L] = 1L;
simlib.maxatr = 10L;

/* *** Initialize statistical routines. */

sampst( 0., 0L );
```

```
timest( 0., 0L );

/* *** Initialize output unit number for SIMLIB error messages. */

llists.iout = 6L;
return;
} /* end of function */

void /*FUNCTION*/ file(option, list)
long int option, list;
{
static long int ahead, behind, ihead, itail, item, item_, row;
static float size;

/* *** IF THE MASTER STORAGE ARRAY IS FULL, STOP THE SIMULATION.
*/
if( llists.nar <= 0L ){
fprintf( stdout, "1          MASTER STORAGE ARRAY OVERFLOW AT TIME
%10.3e\n",
 simlib.time );

/* *** IF LIST VALUE IS IMPROPER, STOP THE SIMULATION. */

}
else if( (list >= 1L) && (list <= 25L) ){
```

```
/* *** INCREMENT THE LIST SIZE. */

simlib.lsize[list - 1L] = simlib.lsize[list - 1L] + 1L;

/* *** IF THE OPTION VALUE IS IMPROPER, STOP THE SIMULATION. */

if( (option >= 1L) && (option <= 4L) ){

/* *** FILE ACCORDING TO THE DESIRED OPTION. */

if( option != 1L ){
if( option != 2L ){

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

/* *** THE LIST IS RANKED, DETERMINE THE ITEM ON WHICH THE LIST IS
TO
 * *** BE RANKED. */

item = simlib.lrank[list - 1L];

/* *** IF AN INVALID ITEM HAS BEEN SPECIFIED, STOP THE SIMULATION.
*/
```

```
if( !((item >= 1L) && (item <= simlib.maxatr))
  ){
fprintf( stdout, "1%10ld IS AN IMPROPER VALUE FOR THE RANK OF LIST
%12ld\n",
 item, list );
exit(0);

/* *** IF THIS IS NOT THE FIRST RECORD IN THIS LIST, CONTINUE. */

}
else if( simlib.lsize[list - 1L] == 1L ){
goto L_40;
}
else{

/* *** SEARCH THE LIST FOR THE PROPER LOCATION. */

row = llists.head[list - 1L];
while( TRUE ){
if( option == 4L ){

/*    RANK THE LIST IN DECREASING ORDER. */

if( simlib.trnsfr[item - 1L] > llists.master[item - 1L][row - 1L] )
goto L_10;
```

```
/* *** RANK THE LIST IN INCREASING ORDER. */

}
else if( simlib.trnsfr[item - 1L] < llists.master[item - 1L][row - 1L] ){

/* *** THE CORRECT LOCATION HAS BEEN FOUND. */

goto L_10;
}

/* *** CONTINUE SEARCHING, CONSIDER THE NEXT ROW. */

behind = row;
row = llists.linksr[behind - 1L];

/* *** IF THE LAST ROW CONSIDERED WAS NOT THE TAIL OF THE LIST,
 * *** CONTINUE. */

if( llists.tail[list - 1L] == behind )
goto L_20;
}

/* *** INSERT BEFORE LAST RECORD EXAMINED. */
```

```
L_10:
if( row == llists.head[list - 1L] )
goto L_30;

/* *** INSERT IN THE PROPER LOCATION BETWEEN THE PRECEEDING
AND
 * *** SUCCEEDING RECORDS (BEHIND AND AHEAD). */

ahead = llists.linksr[behind - 1L];
row = llists.nar;
llists.nar = llists.linksr[row - 1L];
if( llists.nar > 0L )
llists.linkpr[llists.nar - 1L] = 0L;
llists.linkpr[row - 1L] = behind;
llists.linksr[behind - 1L] = row;
llists.linkpr[ahead - 1L] = row;
llists.linksr[row - 1L] = ahead;

/* *** GO TO TRANSFER THE DATA. */

goto L_50;
}
}

/* ********************************************************************** */
```

```
/* *** INSERT AFTER THE LAST RECORD IN THE LIST. */

L_20:
if( simlib.lsize[list - 1L] == 1L )
goto L_40;
row = llists.nar;
llists.nar = llists.linksr[row - 1L];
if( llists.nar > 0L )
llists.linkpr[llists.nar - 1L] = 0L;
itail = llists.tail[list - 1L];
llists.linkpr[row - 1L] = itail;
llists.linksr[itail - 1L] = row;
llists.linksr[row - 1L] = 0L;
llists.tail[list - 1L] = row;

/* *** GO TO TRANSFER THE DATA. */

goto L_50;
}

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

/* *** INSERT BEFORE THE FIRST RECORD IN THE LIST. */
```

```
L_30:
if( simlib.lsize[list - 1L] != 1L ){
row = llists.nar;
llists.nar = llists.linksr[row - 1L];
if( llists.nar > 0L )
llists.linkpr[llists.nar - 1L] = 0L;
ihead = llists.head[list - 1L];
llists.linkpr[ihead - 1L] = row;
llists.linksr[row - 1L] = ihead;
llists.linkpr[row - 1L] = 0L;
llists.head[list - 1L] = row;

/* *** GO TO TRANSFER THE DATA. */

goto L_50;
}

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

/* *** INSERT THE FIRST RECORD IN THE LIST. */

L_40:
row = llists.nar;
llists.nar = llists.linksr[row - 1L];
```

```
if( llists.nar > 0L )
llists.linkpr[llists.nar - 1L] = 0L;
llists.linksr[row - 1L] = 0L;
llists.head[list - 1L] = row;
llists.tail[list - 1L] = row;

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

/* *** TRANSFER THE DATA. */

L_50:
;
for( item = 1L; item <= simlib.maxatr; item++ ){
item_ = item - 1;
llists.master[item_][row - 1L] = simlib.trnsfr[item_];
}

/* *** UPDATE THE AREA UNDER THE NUMBER IN LIST CURVE. */

size = simlib.lsize[list - 1L];
timest( size, 20L + list );
return;
}
else{
fprintf( stdout, "1%10ldIS AN IMPROPER VALUE FOR FILE OPTION AT
```

```
TIME%10.3e\n",
 option, simlib.time );
}
}
else{
fprintf( stdout, "1%10ld IS AN IMPROPER VALUE FOR FILE AT TIME %10.3e\n",
 list, simlib.time );
}
exit(0);
} /* end of function */

void /*FUNCTION*/ removve(option, list)
long int option, list;
{
static long int ihead, itail, item, item_, row;
static float size;

/* *** IF THE LIST IS EMPTY, STOP THE SIMULATION. */
if( !((list >= 1L) && (list <= 25L)) ){
fprintf( stdout, "1%10ld IS AN IMPROPER VALUE FOR REMOVE LIST AT TIME
%10.3e\n",
 list, simlib.time );

/* *** IF THE LIST IS EMPTY, STOP THE SIMULATION. */
```

```
}
else if( simlib.lsize[list - 1L] > 0L ){

/* *** DECREMENT THE LIST SIZE. */

simlib.lsize[list - 1L] = simlib.lsize[list - 1L] - 1L;

/* *** IF THE OPTION VALUE IS IMPROPER, STOP THE SIMULATION. */

if( (option == 1L) || (option == 2L) ){

/* *** IF THERE IS MORE THAN ONE RECORD IN THE LIST, CONTINUE. */

if( simlib.lsize[list - 1L] == 0L ){

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

/* REMOVE THE ONLY RECORD IN THE LIST. */

row = llists.head[list - 1L];
llists.head[list - 1L] = 0L;
llists.tail[list - 1L] = 0L;

/* *** REMOVE ACCORDING TO THE DESIRED OPTION. */
```

```
}
else if( option == 2L ){

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

/* *** REMOVE THE LAST RECORD IN THE LIST. */

row = llists.tail[list - 1L];
itail = llists.linkpr[row - 1L];
llists.linksr[itail - 1L] = 0L;
llists.tail[list - 1L] = itail;

/* *** GO TO TRANSFER THE DATA. */

}
else{

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

/* *** REMOVE THE FIRST RECORD IN THE LIST. */

row = llists.head[list - 1L];
ihead = llists.linksr[row - 1L];
llists.linkpr[ihead - 1L] = 0L;
llists.head[list - 1L] = ihead;
```

```
/* *** GO TO TRANSFER THE DATA. */

}

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

/* *** TRANSFER THE DATA. */

llists.linksr[row - 1L] = llists.nar;
llists.linkpr[row - 1L] = 0L;
llists.nar = row;
for( item = 1L; item <= simlib.maxatr; item++ ){
item_ = item - 1;
simlib.trnsfr[item_] = llists.master[item_][row - 1L];
}

/* *** UPDATE THE AREA UNDER THE NUMBER IN LIST CURVE. */

size = simlib.lsize[list - 1L];
timest( size, 20L + list );
return;
}
else{
fprintf( stdout, "1%10ld IS AN IMPROPER VALUE FOR REMOVE OPTION AT
```

```
TIME %10.3e\n",
 option, simlib.time );
}
}
else{
fprintf( stdout, "1     UNDERFLOW OF LIST %2ld AT TIME %10.3e\n",
 list, simlib.time );
}
exit(0);
} /* end of function */




void /*FUNCTION*/ timing()
{


/* *** Remove the first event from the event list. */
removve( simlib.lfirst, simlib.levent );


/* *** Check for a time reversal. */


if( simlib.trnsfr[0L] >= simlib.time ){


/* *** Advance the simulation clock. */
```

```
simlib.time = simlib.trnsfr[0L];
simlib.next = simlib.trnsfr[1L];
return;
}
else{
fprintf( stdout, " From SIMLIB: Attempt to schedule an event of type %3.0f\n at time
%10.3f when the clock is %10.3f\n",
 simlib.trnsfr[1L], simlib.trnsfr[0L], simlib.time );
exit(0);
}
return;
} /* end of function */

void /*FUNCTION*/ cancel(etype)
double etype;
{
static long int ahead, behind, item, item_, row;
static float high, low, size, value;

/* *** SEARCH THE EVENT LIST. */
if( simlib.lsize[simlib.levent - 1L] != 0L ){
row = llists.head[simlib.levent - 1L];
low = etype - 0.1;
high = etype + 0.1;
while( TRUE ){
```

```
value = llists.master[1L][row - 1L];
if( (low < value) && (high > value) )
goto L_10;

/* *** GO TO THE NEXT EVENT. */

if( row == llists.tail[simlib.levent - 1L] )
return;
row = llists.linksr[row - 1L];
}

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

/* *** CANCEL THIS EVENT. */

L_10:
if( row == llists.head[simlib.levent - 1L] ){

/* *** REMOVE THE FIRST EVENT IN THE EVENT LIST. */

removve( simlib.lfirst, simlib.levent );
}
else if( row == llists.tail[simlib.levent - 1L] ){

/* *** REMOVE THE LAST EVENT IN THE EVENT LIST. */
```

```
removve( simlib.llast, simlib.levent );
}
else{

/* *** REMOVE THIS EVENT WHICH IS SOMEWHERE IN THE MIDDLE OF
THE EVENT
 * *** LIST. */

ahead = llists.linksr[row - 1L];
behind = llists.linkpr[row - 1L];
llists.linksr[behind - 1L] = ahead;
llists.linkpr[ahead - 1L] = behind;
llists.linksr[row - 1L] = llists.nar;
llists.linkpr[row - 1L] = 0L;
llists.nar = row;
simlib.lsize[simlib.levent - 1L] = simlib.lsize[simlib.levent - 1L] -
 1L;

/* *** PLACE THE ATTRIBUTE OF THE CANCELED EVENT IN THE TRNSFR
ARRAY. */

for( item = 1L; item <= simlib.maxatr; item++ ){
item_ = item - 1;
simlib.trnsfr[item_] = llists.master[item_][row - 1L];
```

```
}

/* *** UPDATE THE AREA UNDER THE NUMBER IN LIST CURVE. */

size = simlib.lsize[simlib.levent - 1L];
timest( size, 45L );
}
}
return;
} /* end of function */


void /*FUNCTION*/ sampst(value, varibl)
double value;
long int varibl;
{
static long int ivar, ivar_, nobs[20];
static float max_[20], min_[20], sum[20];

/* *** If the variable value is improper, stop the simulation. */
if( varibl >= -20L && varibl <= 20L ){

/* *** Execute the desired option. */

if( varibl < 0L ){
```

```
/* ------------------------------------------------------------------ */

/* *** Report the results. */

ivar = -varibl;
simlib.trnsfr[0L] = 0.;
simlib.trnsfr[1L] = nobs[ivar - 1L];
simlib.trnsfr[2L] = max_[ivar - 1L];
simlib.trnsfr[3L] = min_[ivar - 1L];
if( nobs[ivar - 1L] != 0L )
simlib.trnsfr[0L] = sum[ivar - 1L]/simlib.trnsfr[1L];
}
else if( varibl > 0L ){

/* ------------------------------------------------------------------ */

/* *** Collect data. */

sum[varibl - 1L] = sum[varibl - 1L] + value;
if( value > max_[varibl - 1L] )
max_[varibl - 1L] = value;
if( value < min_[varibl - 1L] )
min_[varibl - 1L] = value;
nobs[varibl - 1L] = nobs[varibl - 1L] + 1L;
```

```
}
else{

/* -------------------------------------------------------------------- */

/* *** Initialize the routine. */

for( ivar = 1L; ivar <= 20L; ivar++ ){
ivar_ = ivar - 1;
sum[ivar_] = 0.;
max_[ivar_] = -1.e30;
min_[ivar_] = 1.e30;
nobs[ivar_] = 0L;
}
}
return;
}
else{
fprintf( stdout, " From SIMLIB:%10ld is improper value for SAMPST variable at time
%10.3f\n",
 varibl, simlib.time );
exit(0);
}
return;
} /* end of function */
```

```
void /*FUNCTION*/ timest(value, varibl)
double value;
long int varibl;
{
static long int ivar, ivar_;
static float area[45], max_[45], min_[45], preval[45], tlvc[45], treset;

/* *** If the variable value is improper, stop the simulation. */
if( varibl >= -45L && varibl <= 45L ){

/* *** Execute the desired option. */

if( varibl < 0L ){

/* ------------------------------------------------------------------ */

/* *** Report the results. */

ivar = -varibl;
area[ivar - 1L] = area[ivar - 1L] + (simlib.time - tlvc[ivar - 1L])*
 preval[ivar - 1L];
tlvc[ivar - 1L] = simlib.time;
simlib.trnsfr[0L] = area[ivar - 1L]/(simlib.time - treset);
simlib.trnsfr[1L] = max_[ivar - 1L];
```

```
simlib.trnsfr[2L] = min_[ivar - 1L];
}
else if( varibl > 0L ){

/* ---------------------------------------------------------------------- */

/* *** Collect data. */

area[varibl - 1L] = area[varibl - 1L] + (simlib.time -
 tlvc[varibl - 1L])*preval[varibl - 1L];
if( value > max_[varibl - 1L] )
max_[varibl - 1L] = value;
if( value < min_[varibl - 1L] )
min_[varibl - 1L] = value;
preval[varibl - 1L] = value;
tlvc[varibl - 1L] = simlib.time;
}
else{

/* ---------------------------------------------------------------------- */

/* *** Initialize the routine. */

for( ivar = 1L; ivar <= 45L; ivar++ ){
ivar_ = ivar - 1;
```

```
area[ivar_] = 0.;
max_[ivar_] = -1.e30;
min_[ivar_] = 1.e30;
preval[ivar_] = 0.;
tlvc[ivar_] = simlib.time;
}
treset = simlib.time;
}
return;
}
else{
fprintf( stdout, " From SIMLIB:%10ld is improper value for TIMEST variable at time
%10.3f\n",
 varibl, simlib.time );
exit(0);
}
return;
} /* end of function */


void /*FUNCTION*/ filest(list)
long int list;
{
static long int ilist;
```

```
/* *** Compute summary statistics for the list. */
ilist = -(20L + list);
timest( 0., ilist );
return;
} /* end of function */



double /*FUNCTION*/ expon(rmean)
double rmean;
{
static long int z;
static float expon_v, u;

/* *** GENERATE A U(0,1) RANDOM VARIABLE. THE FORM OF THIS
STATEMENT
 * *** DEPENDS UPON THE COMPUTER USED. */
z = 0L;
u = rannd( &z );

/* *** GENERATE AN EXPONENTIAL RANDOM VARIABLE WITH RMEAN. */

expon_v = -rmean*log( u );
return( expon_v );
} /* end of function */
```

```
double /*FUNCTION*/ unifrm(a, b)
double a, b;
{
static long int z;
static float u, unifrm_v;

/* *** GENERATE A U(0,1) RANDOM VARIABLE. THE FORM OF THIS STATEMENT
 * *** DEPENDS UPON THE COMPUTER USED. */
u = rannd( &z );

/* *** GENERATE A U(A,B) RANDOM VARIABLE. */

unifrm_v = a + (u*(b - a));
return( unifrm_v );
} /* end of function */


double /*FUNCTION*/ rannd(key)
long int *key;
{
static long int fhi, k, leftlo, xalo, xhi;
static float rannd_v;
```

```
static long a = 16807;
static long b15 = 32768;
static long b16 = 65536;
static long p = 2147483647;
static long ix = 853506241;

/* *** Prime modulus multiplicative linear congruential generator
 * *** Z(I) = ((7**5) * Z(I - 1)) (MOD(2**31 - 1))
 * *** based on Schrage's portable random number generator RAND.
 * *** (See Chapter 6 for more datail.) */
/* *** Set the default seed for the random number generator. */
/* *** Determine desired action. */
if( *key == 0L ){

/* *** Generate the next random number. */

xhi = ix/b16;
xalo = (ix - xhi*b16)*a;
leftlo = xalo/b16;
fhi = xhi*a + leftlo;
k = fhi/b15;
ix = (((xalo - leftlo*b16) - p) + (fhi - k*b15)*b16) + k;
if( ix < 0L )
ix = ix + p;
rannd_v = (float)( ix )*4.656612875e-10;
```

```
}
else if( *key > 0L ){

/* *** Set the current IX to KEY. */

ix = *key;
}
else{

/* *** Return current IX in KEY. */

*key = ix;
}
return( rannd_v );
} /* end of function */
```

# Appendix 4

Protype header file job.h

```
void arrive(long);
void cancel(double);
void depart(void);
double erlang(long,double,long);
double expon(double,long);
void file(long,long);
void filest(long);
void initlk(void);
long irandi(long,float[],long);
void prgm_(void);
double rannd(long);
void removve(long,long);
void report(void);
void sampst(double,long);
void timest(double,long);
void timing(void);
```

File jobshop.in

```
5 3 0.25 365.0
3 2 4 3 1
4 3 5
3   1   2   5
0.50 0.60 0.85 0.50
4   1   3
1.10 0.80 0.75
2   5   1   4   3
1.20 0.25 0.70 0.90 1.00
0.300 0.800 1.000
```

File job.c

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

#define TRUE 1
#define FALSE 0
typedef long LOGICAL32;
```

```
typedef char byte;
#define ADR(t,x)        ( t=(x), &t )
#define IS_ODD(j)       ((j) & 1 )

#include "JOB.h"

struct t_model {
long int jobtyp;
float length, marrvt, mservt[5][5];
long int nbusy[5], ngroup, nmachs[5], ntasks[5], ntypes;
float probd[3];
long int route[5][5], task;
}model;
struct t_simlib {
long int ldecr, levent, lfirst, lincr, llast, lrank[25], lsize[25],
 maxatr, next;
float time, trnsfr[10];
}simlib;
FILE *fp;

main( )
{
static long int i, i_, j, ntsks;
```

```
if ((fp = fopen("jobshop.in","r"))==NULL) {
   fprintf(stderr, "cannot open jobshop.in\n");
   exit(1);
}


/*    READ THE INPUT PARAMETERS */

fscanf( fp, "%ld %ld %f %f ", &model.ngroup, &model.ntypes,
 &model.marrvt, &model.length );
fscanf( fp, "" );
for( i = 1L; i <= model.ngroup; i++ ){
fscanf( fp, "%ld ", &model.nmachs[i - 1L] );
}
fscanf( fp, "" );
for( i = 1L; i <= model.ntypes; i++ ){
fscanf( fp, "%ld ", &model.ntasks[i - 1L] );
}
for( i = 1L; i <= model.ntypes; i++ ){
i_ = i - 1;
ntsks = model.ntasks[i_];
fscanf( fp, "" );
for( j = 1L; j <= ntsks; j++ ){
fscanf( fp, "%ld ", &model.route[j - 1L][i_] );
}
```

```
fscanf( fp, "" );
for( j = 1L; j <= ntsks; j++ ){
fscanf( fp, "%f ", &model.mservt[j - 1L][i_] );
}
}
fscanf( fp, "" );
for( i = 1L; i <= model.ntypes; i++ ){
fscanf( fp, "%f ", &model.probd[i - 1L] );
}


/*    Write report heading and input parameters */


fprintf( stdout, " Job-shop model\n\n Number of machine groups%20ld",
 model.ngroup );
fprintf( stdout, "\n\n Number of machines in each group       " );
for( i = 1L; i <= model.ngroup; i++ ){
fprintf( stdout, "%5ld", model.nmachs[i - 1L] );
}
fprintf( stdout, "\n" );
fprintf( stdout, "\n Number of job types%25ld", model.ntypes );
fprintf( stdout, "\n\n Number of tasks for each job type      " );
for( i = 1L; i <= model.ntypes; i++ ){
fprintf( stdout, "%5ld", model.ntasks[i - 1L] );
}
fprintf( stdout, "\n" );
```

```
fprintf( stdout, "\n Distribution function of job types  " );
for( i = 1L; i <= model.ntypes; i++ ){
fprintf( stdout, "%8.3f", model.probd[i - 1L] );
}
fprintf( stdout, "\n" );
fprintf( stdout, "\n Mean interarrival time of jobs%14.2f hours\n\n Length of the
simulation%20.1f eight-hour days\n\n\n Job Type     Machine groups on route\n",
 model.marrvt, model.length );
for( i = 1L; i <= model.ntypes; i++ ){
i_ = i - 1;
fprintf( stdout, "\n%5ld", i );
fprintf( stdout, "        " );
for( j = 1L; j <= model.ntasks[i_]; j++ ){
fprintf( stdout, "%5ld", model.route[j - 1L][i_] );
}
fprintf( stdout, "\n" );
}
fprintf( stdout, "\n\n Job Type     Mean service time (in hours) for successive tasks\n"
);
for( i = 1L; i <= model.ntypes; i++ ){
i_ = i - 1;
fprintf( stdout, "\n%5ld", i );
fprintf( stdout, "     " );
for( j = 1L; j <= model.ntasks[i_]; j++ ){
fprintf( stdout, "%8.2f", model.mservt[j - 1L][i_] );
```

```
}
fprintf( stdout, "\n" );
}

/*    Initialize all machines in all groups to the idle state */

for( i = 1L; i <= model.ngroup; i++ ){
i_ = i - 1;
model.nbusy[i_] = 0L;
}

/*    Initialize SIMLIB */

initlk();

/*    Set the maximum number of attributes per record */

simlib.maxatr = 4L;

/* *** Schedule the arrival of the first job. */

simlib.trnsfr[0L] = expon( model.marrvt, 1L );
simlib.trnsfr[1L] = 1.;
file( simlib.lincr, simlib.levent );
```

```
/*	Schedule the end of the simulation */

simlib.trnsfr[0L] = 8.*model.length;
simlib.trnsfr[1L] = 3.;
file( simlib.lincr, simlib.levent );

/*	Determine the next event routine */

while( TRUE ){
timing();

/*	CALL THE APPROPRIATE EVENT ROUTINE */

if( simlib.next == 2L ){
depart();
}
else if( simlib.next == 3L ){
goto L_10;
}
else{
arrive( 1L );
}
}
L_10:
report();
```

```
fclose(fp);

return 0;
} /* end of function */

long /*FUNCTION*/ irandi(nvalue, probd, istrm)
long int nvalue;
float probd[];
long int istrm;
{
static long int i, i_, irandi_v;
static float u;

/*    Generate a U(0,1) random varibale from stream ISTREAM */
u = rannd( istrm );

/*    Generate a random integer between 1 and NVALUE in accordance with
 *    the cumulative distribution funvtion PROBD. */

for( i = 1L; i <= (nvalue - 1L); i++ ){
i_ = i - 1;
if( u < probd[i_] )
goto L_10;
```

```
}
irandi_v = nvalue;

return( irandi_v );
L_10:
irandi_v = i;
return( irandi_v );
} /* end of function */


void /*FUNCTION*/ arrive(new)
long int new;
{
static long int group, index;
static float delay;

/* *** If this is a new arrival to the shop, generate the time of the
 * *** next arrival and determine the job type and task number of the
 * *** arriving job */
if( new == 1L ){
simlib.trnsfr[0L] = simlib.time + expon( model.marrvt, 1L );
simlib.trnsfr[1L] = 1.;
file( simlib.lincr, simlib.levent );
model.jobtyp = irandi( model.ntypes, model.probd, 2L );
model.task = 1L;
```

```
}

/*      Determine a machine group from the route matrix. */

group = model.route[model.task - 1L][model.jobtyp - 1L];

/*      Check to see whether all machines in this group are busy. */

if( model.nbusy[group - 1L] == model.nmachs[group - 1L] ){

/*      All machines in this group are busy, so place the arriving job
 *      at the end of the appropriate queue. Note that the following
 *      data are stored for each job
 *          1. Time of arrival to this machine group.
 *          2. job type.
 *          3  current task number */

simlib.trnsfr[0L] = simlib.time;
simlib.trnsfr[1L] = model.jobtyp;
simlib.trnsfr[2L] = model.task;
file( simlib.llast, group );

}
else{
```

```
/*    A machine in this group is idle, so start service on the
 *    arriving job (which has a delay of zero). */

delay = 0L;
sampst( delay, group );
index = model.ngroup + model.jobtyp;
sampst( delay, index );
model.nbusy[group - 1L] = model.nbusy[group - 1L] + 1L;
timest( (float)( model.nbusy[group - 1L] ), group );

/*    Schedule a service completion */

simlib.trnsfr[0L] = simlib.time + erlang( 2L, model.mservt[model.task -
1L][model.jobtyp - 1L],
 3L );
simlib.trnsfr[1L] = 2.0;
simlib.trnsfr[2L] = model.jobtyp;
simlib.trnsfr[3L] = model.task;
file( simlib.lincr, simlib.levent );




}
return;
} /* end of function */
```

```
void /*FUNCTION*/ depart()
{
static long int group, index, jobtq, taskq;
static float delay;

/*     Determine the machine group from which the job is departing */
model.jobtyp = simlib.trnsfr[2L];
model.task = simlib.trnsfr[3L];
group = model.route[model.task - 1L][model.jobtyp - 1L];

/*     Check to see whether the queue for this machine group is empty. */

if( simlib.lsize[group - 1L] == 0L ){

/*      The queue for this machine is empty so make a machine in
 *      this group idle. */

model.nbusy[group - 1L] = model.nbusy[group - 1L] - 1L;
timest( (float)( model.nbusy[group - 1L] ), group );

}
else{
```

```
/*   The queue is not empty, so start service on the first job in queue */

removve( 1L, group );
delay = simlib.time - simlib.trnsfr[0L];

/*   Tally this delay for this machine group. */

sampst( delay, group );

jobtq = simlib.trnsfr[1L];
taskq = simlib.trnsfr[2L];
index = model.ngroup + jobtq;
sampst( delay, index );

/*   Schedule end of service for this job at this machine group */

simlib.trnsfr[0L] = simlib.time + erlang( 2L, model.mservt[taskq - 1L][jobtq - 1L],
 3L );
simlib.trnsfr[1L] = 2.;
simlib.trnsfr[2L] = jobtq;
simlib.trnsfr[3L] = taskq;
file( simlib.lincr, simlib.levent );

}
```

```
/* *** If the current departing job has one or more tasks yet to be done,
 * *** send the job to the next machine group on its route. */

if( model.task < model.ntasks[model.jobtyp - 1L] ){
model.task = model.task + 1L;
arrive( 2L );

}
return;
} /* end of function */

void /*FUNCTION*/ report()
{
static long int i, i_, index;
static float ajdel[5], amdel[5], autil[5], avgniq[5], oajdel, sum;

/*     Compute the average total delay in queue for each jobtype and the
 *     overall average job delay. */
oajdel = 0.0;
sum = 0.0;
for( i = 1L; i <= model.ntypes; i++ ){
i_ = i - 1;
index = model.ngroup + i;
sampst( 0., -index );
ajdel[i_] = simlib.trnsfr[0L]*model.ntasks[i_];
```

```
oajdel = oajdel + (model.probd[i_] - sum)*ajdel[i_];
sum = model.probd[i_];
}

/*    Compute the average number in queue, the average utilization, and
 *    the average dealy in queue for each machine group. */

for( i = 1L; i <= model.ngroup; i++ ){
i_ = i - 1;
sampst( 0., -i );
amdel[i_] = simlib.trnsfr[0L];
filest( i );
avgniq[i_] = simlib.trnsfr[0L];
timest( 0., -i );
autil[i_] = simlib.trnsfr[0L]/model.nmachs[i_];
}
fprintf( stdout, "\n\n\n Job type     Average total delay in queue\n" );
for( i = 1L; i <= model.ntypes; i++ ){
i_ = i - 1;
fprintf( stdout, "\n%5ld%27.3f\n", i, ajdel[i_] );
}
fprintf( stdout, "\n Overall average total job delay =%10.3f\n\n\n Machine
Average number     Average       Average Delay\n group    number in queue
utilization     in queue\n",
 oajdel );
```

```
for( i = 1L; i <= model.ngroup; i++ ){
i_ = i - 1;
fprintf( stdout, "\n%5ld%17.3f%17.3f%17.3f\n", i, avgniq[i_],
 autil[i_], amdel[i_] );

}
return;
} /* end of function */




struct t_llists {
long int head[25], iout, linkpr[1000], linksr[1000];
float master[10][1000];
long int nar, tail[25];
}llists;

void /*FUNCTION*/ initlk()
{
static long int i, i_, list, list_, row, row_;

/* *** Initialize links. */
for( row = 1L; row <= 1000L; row++ ){
row_ = row - 1;
llists.linkpr[row_] = 0L;
```

```
llists.linksr[row_] = row + 1L;
}
llists.linksr[999L] = 0L;

/* *** Initialize list attributes. */

for( list = 1L; list <= 25L; list++ ){
list_ = list - 1;
llists.head[list_] = 0L;
llists.tail[list_] = 0L;
simlib.lsize[list_] = 0L;
simlib.lrank[list_] = 0L;
}

/* *** Initialize the TRANSFR */

for( i = 1L; i <= 10L; i++ ){
i_ = i - 1;
simlib.trnsfr[i_] = 0.0;
}

/* *** Initialize mnemonics for record location in lists. */

simlib.lfirst = 1L;
simlib.llast = 2L;
```

```
simlib.lincr = 3L;
simlib.ldecr = 4L;

/* *** Initialize mnemonic for event list number. */

simlib.levent = 25L;

/* *** Initialize system attributes. */

simlib.time = 0.;
llists.nar = 1L;
simlib.lrank[simlib.levent - 1L] = 1L;
simlib.maxatr = 10L;

/* *** Initialize statistical routines. */

sampst( 0., 0L );
timest( 0., 0L );

/* *** Initialize output unit number for SIMLIB error messages. */

llists.iout = 6L;
return;
} /* end of function */
```

```
void /*FUNCTION*/ file(option, list)
long int option, list;
{
static long int ahead, behind, ihead, itail, item, item_, row;
static float size;

/* *** IF THE MASTER STORAGE ARRAY IS FULL, STOP THE SIMULATION.
*/
if( llists.nar <= 0L ){
fprintf( stdout, "1        MASTER STORAGE ARRAY OVERFLOW AT TIME
%10.3e\n",
 simlib.time );

/* *** IF LIST VALUE IS IMPROPER, STOP THE SIMULATION. */

}
else if( (list >= 1L) && (list <= 25L) ){

/* *** INCREMENT THE LIST SIZE. */

simlib.lsize[list - 1L] = simlib.lsize[list - 1L] + 1L;

/* *** IF THE OPTION VALUE IS IMPROPER, STOP THE SIMULATION. */

if( (option >= 1L) && (option <= 4L) ){
```

```
/* *** FILE ACCORDING TO THE DESIRED OPTION. */

if( option != 1L ){
if( option != 2L ){

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

/* *** THE LIST IS RANKED, DETERMINE THE ITEM ON WHICH THE LIST IS
TO
 * *** BE RANKED. */

item = simlib.lrank[list - 1L];

/* *** IF AN INVALID ITEM HAS BEEN SPECIFIED, STOP THE SIMULATION.
*/

if( !((item >= 1L) && (item <= simlib.maxatr))
  ){
fprintf( stdout, "1%10ld IS AN IMPROPER VALUE FOR THE RANK OF LIST
%12ld\n",
 item, list );
exit(0);
}
else if( simlib.lsize[list - 1L] == 1L ){
```

```
goto L_40;
}
else{

/* *** SEARCH THE LIST FOR THE PROPER LOCATION. */

row = llists.head[list - 1L];
while( TRUE ){
if( option == 4L ){

/*     RANK THE LIST IN DECREASING ORDER. */

if( simlib.trnsfr[item - 1L] > llists.master[item - 1L][row - 1L] )
goto L_10;

/* *** RANK THE LIST IN INCREASING ORDER. */

}
else if( simlib.trnsfr[item - 1L] < llists.master[item - 1L][row - 1L] ){

/* *** THE CORRECT LOCATION HAS BEEN FOUND. */

goto L_10;
}
```

```
/* *** CONTINUE SEARCHING, CONSIDER THE NEXT ROW. */

behind = row;
row = llists.linksr[behind - 1L];

/* *** IF THE LAST ROW CONSIDERED WAS NOT THE TAIL OF THE LIST,
 * *** CONTINUE. */

if( llists.tail[list - 1L] == behind )
goto L_20;
}

/* *** INSERT BEFORE LAST RECORD EXAMINED. */

L_10:
if( row == llists.head[list - 1L] )
goto L_30;

/* *** INSERT IN THE PROPER LOCATION BETWEEN THE PRECEEDING
AND
 * *** SUCCEEDING RECORDS (BEHIND AND AHEAD). */

ahead = llists.linksr[behind - 1L];
row = llists.nar;
llists.nar = llists.linksr[row - 1L];
```

```
if( llists.nar > 0L )
llists.linkpr[llists.nar - 1L] = 0L;
llists.linkpr[row - 1L] = behind;
llists.linksr[behind - 1L] = row;
llists.linkpr[ahead - 1L] = row;
llists.linksr[row - 1L] = ahead;

/* *** GO TO TRANSFER THE DATA. */

goto L_50;
}
}

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

/* *** INSERT AFTER THE LAST RECORD IN THE LIST. */

L_20:
if( simlib.lsize[list - 1L] == 1L )
goto L_40;
row = llists.nar;
llists.nar = llists.linksr[row - 1L];
if( llists.nar > 0L )
llists.linkpr[llists.nar - 1L] = 0L;
itail = llists.tail[list - 1L];
```

```
llists.linkpr[row - 1L] = itail;
llists.linksr[itail - 1L] = row;
llists.linksr[row - 1L] = 0L;
llists.tail[list - 1L] = row;

/* *** GO TO TRANSFER THE DATA. */

goto L_50;
}

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

/* *** INSERT BEFORE THE FIRST RECORD IN THE LIST. */

L_30:
if( simlib.lsize[list - 1L] != 1L ){
row = llists.nar;
llists.nar = llists.linksr[row - 1L];
if( llists.nar > 0L )
llists.linkpr[llists.nar - 1L] = 0L;
ihead = llists.head[list - 1L];
llists.linkpr[ihead - 1L] = row;
llists.linksr[row - 1L] = ihead;
llists.linkpr[row - 1L] = 0L;
llists.head[list - 1L] = row;
```

```
/* *** GO TO TRANSFER THE DATA. */

goto L_50;
}

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

/* *** INSERT THE FIRST RECORD IN THE LIST. */

L_40:
row = llists.nar;
llists.nar = llists.linksr[row - 1L];
if( llists.nar > 0L )
llists.linkpr[llists.nar - 1L] = 0L;
llists.linksr[row - 1L] = 0L;
llists.head[list - 1L] = row;
llists.tail[list - 1L] = row;

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

/* *** TRANSFER THE DATA. */

L_50:
```

```
;
for( item = 1L; item <= simlib.maxatr; item++ ){
item_ = item - 1;
llists.master[item_][row - 1L] = simlib.trnsfr[item_];
}

/* *** UPDATE THE AREA UNDER THE NUMBER IN LIST CURVE. */

size = simlib.lsize[list - 1L];
timest( size, 20L + list );
return;
}
else{
fprintf( stdout, "1%10ldIS AN IMPROPER VALUE FOR FILE OPTION AT
TIME%10.3e\n",
 option, simlib.time );
}
}
else{
fprintf( stdout, "1%10ld IS AN IMPROPER VALUE FOR FILE AT TIME %10.3e\n",
 list, simlib.time );
}
exit(0);
} /* end of function */
```

```
void /*FUNCTION*/ removve(option, list)
long int option, list;
{
static long int ihead, itail, item, item_, row;
static float size;

/* *** IF THE LIST IS EMPTY, STOP THE SIMULATION. */
if( !((list >= 1L) && (list <= 25L)) ){
fprintf( stdout, "1%10ld IS AN IMPROPER VALUE FOR REMOVE LIST AT TIME
%10.3e\n",
 list, simlib.time );

/* *** IF THE LIST IS EMPTY, STOP THE SIMULATION. */


}
else if( simlib.lsize[list - 1L] > 0L ){

/* *** DECREMENT THE LIST SIZE. */

simlib.lsize[list - 1L] = simlib.lsize[list - 1L] - 1L;

/* *** IF THE OPTION VALUE IS IMPROPER, STOP THE SIMULATION. */

if( (option == 1L) || (option == 2L) ){
```

```
/* *** IF THERE IS MORE THAN ONE RECORD IN THE LIST, CONTINUE. */

if( simlib.lsize[list - 1L] == 0L ){

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

/* REMOVE THE ONLY RECORD IN THE LIST. */

row = llists.head[list - 1L];
llists.head[list - 1L] = 0L;
llists.tail[list - 1L] = 0L;

/* *** REMOVE ACCORDING TO THE DESIRED OPTION. */

}
else if( option == 2L ){

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

/* *** REMOVE THE LAST RECORD IN THE LIST. */

row = llists.tail[list - 1L];
itail = llists.linkpr[row - 1L];
llists.linksr[itail - 1L] = 0L;
llists.tail[list - 1L] = itail;
```

```
/* *** GO TO TRANSFER THE DATA. */

}
else{

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

/* *** REMOVE THE FIRST RECORD IN THE LIST. */

row = llists.head[list - 1L];
ihead = llists.linksr[row - 1L];
llists.linkpr[ihead - 1L] = 0L;
llists.head[list - 1L] = ihead;

/* *** GO TO TRANSFER THE DATA. */

}

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

/* *** TRANSFER THE DATA. */

llists.linksr[row - 1L] = llists.nar;
llists.linkpr[row - 1L] = 0L;
```

```
llists.nar = row;
for( item = 1L; item <= simlib.maxatr; item++ ){
item_ = item - 1;
simlib.trnsfr[item_] = llists.master[item_][row - 1L];
}

/* *** UPDATE THE AREA UNDER THE NUMBER IN LIST CURVE. */

size = simlib.lsize[list - 1L];
timest( size, 20L + list );
return;
}
else{
fprintf( stdout, "1%10ld IS AN IMPROPER VALUE FOR REMOVE OPTION AT
TIME %10.3e\n",
 option, simlib.time );
}
}
else{
fprintf( stdout, "1     UNDERFLOW OF LIST %2ld AT TIME %10.3e\n",
 list, simlib.time );
}
exit(0);
} /* end of function */
```

```
void /*FUNCTION*/ timing()
{

/* *** Remove the first event from the event list. */
removvve( simlib.lfirst, simlib.levent );

/* *** Check for a time reversal. */

if( simlib.trnsfr[0L] >= simlib.time ){

/* *** Advance the simulation clock. */

simlib.time = simlib.trnsfr[0L];
simlib.next = simlib.trnsfr[1L];
return;
}
else{
fprintf( stdout, " From SIMLIB: Attempt to schedule an event of type %3.0f\n at time
%10.3f when the clock is %10.3f\n",
 simlib.trnsfr[1L], simlib.trnsfr[0L], simlib.time );
exit(0);
}
return;
```

```
} /* end of function */

void /*FUNCTION*/ cancel(etype)
double etype;
{
static long int ahead, behind, item, item_, row;
static float high, low, size, value;

/* *** SEARCH THE EVENT LIST. */
if( simlib.lsize[simlib.levent - 1L] != 0L ){
row = llists.head[simlib.levent - 1L];
low = etype - 0.1;
high = etype + 0.1;
while( TRUE ){
value = llists.master[1L][row - 1L];
if( (low < value) && (high > value) )
goto L_10;

/* *** GO TO THE NEXT EVENT. */

if( row == llists.tail[simlib.levent - 1L] )
return;
row = llists.linksr[row - 1L];
}
```

```
/* ****************************************************************** */

/* *** CANCEL THIS EVENT. */

L_10:
if( row == llists.head[simlib.levent - 1L] ){

/* *** REMOVE THE FIRST EVENT IN THE EVENT LIST. */

removve( simlib.lfirst, simlib.levent );
}
else if( row == llists.tail[simlib.levent - 1L] ){

/* *** REMOVE THE LAST EVENT IN THE EVENT LIST. */

removve( simlib.llast, simlib.levent );
}
else{

/* *** REMOVE THIS EVENT WHICH IS SOMEWHERE IN THE MIDDLE OF
THE EVENT
 * *** LIST. */

ahead = llists.linksr[row - 1L];
behind = llists.linkpr[row - 1L];
```

```
llists.linksr[behind - 1L] = ahead;
llists.linkpr[ahead - 1L] = behind;
llists.linksr[row - 1L] = llists.nar;
llists.linkpr[row - 1L] = 0L;
llists.nar = row;
simlib.lsize[simlib.levent - 1L] = simlib.lsize[simlib.levent - 1L] -
 1L;

/* *** PLACE THE ATTRIBUTE OF THE CANCELED EVENT IN THE TRNSFR
ARRAY. */

for( item = 1L; item <= simlib.maxatr; item++ ){
item_ = item - 1;
simlib.trnsfr[item_] = llists.master[item_][row - 1L];
}

/* *** UPDATE THE AREA UNDER THE NUMBER IN LIST CURVE. */

size = simlib.lsize[simlib.levent - 1L];
timest( size, 45L );
}
}
return;
} /* end of function */
```

```
void /*FUNCTION*/ sampst(value, varibl)
double value;
long int varibl;
{
static long int ivar, ivar_, nobs[20];
static float max_[20], min_[20], sum[20];

/* *** If the variable value is improper, stop the simulation. */
if( varibl >= -20L && varibl <= 20L ){

/* *** Execute the desired option. */

if( varibl < 0L ){

/* ------------------------------------------------------------------- */

/* *** Report the results. */

ivar = -varibl;
simlib.trnsfr[0L] = 0.;
simlib.trnsfr[1L] = nobs[ivar - 1L];
simlib.trnsfr[2L] = max_[ivar - 1L];
simlib.trnsfr[3L] = min_[ivar - 1L];
if( nobs[ivar - 1L] != 0L )
```

```
simlib.trnsfr[0L] = sum[ivar - 1L]/simlib.trnsfr[1L];
}
else if( varibl > 0L ){

/* ------------------------------------------------------------------ */

/* *** Collect data. */

sum[varibl - 1L] = sum[varibl - 1L] + value;
if( value > max_[varibl - 1L] )
max_[varibl - 1L] = value;
if( value < min_[varibl - 1L] )
min_[varibl - 1L] = value;
nobs[varibl - 1L] = nobs[varibl - 1L] + 1L;
}
else{

/* ------------------------------------------------------------------ */

/* *** Initialize the routine. */

for( ivar = 1L; ivar <= 20L; ivar++ ){
ivar_ = ivar - 1;
sum[ivar_] = 0.;
max_[ivar_] = -1.e30;
```

```
min_[ivar_] = 1.e30;
nobs[ivar_] = 0L;
}
}
return;
}
else{
fprintf( stdout, " From SIMLIB:%10ld is improper value for SAMPST variable at time
%10.3f\n",
 varibl, simlib.time );
exit(0);
}
return;
} /* end of function */

void /*FUNCTION*/ timest(value, varibl)
double value;
long int varibl;
{
static long int ivar, ivar_;
static float area[45], max_[45], min_[45], preval[45], tlvc[45], treset;

/* *** If the variable value is improper, stop the simulation. */
if( varibl >= -45L && varibl <= 45L ){
```

```
/* *** Execute the desired option. */

if( varibl < 0L ){

/* ------------------------------------------------------------------ */

/* *** Report the results. */

ivar = -varibl;
area[ivar - 1L] = area[ivar - 1L] + (simlib.time - tlvc[ivar - 1L])*
 preval[ivar - 1L];
tlvc[ivar - 1L] = simlib.time;
simlib.trnsfr[0L] = area[ivar - 1L]/(simlib.time - treset);
simlib.trnsfr[1L] = max_[ivar - 1L];
simlib.trnsfr[2L] = min_[ivar - 1L];
}
else if( varibl > 0L ){

/* ------------------------------------------------------------------ */

/* *** Collect data. */

area[varibl - 1L] = area[varibl - 1L] + (simlib.time -
 tlvc[varibl - 1L])*preval[varibl - 1L];
if( value > max_[varibl - 1L] )
```

```
max_[varibl - 1L] = value;
if( value < min_[varibl - 1L] )
min_[varibl - 1L] = value;
preval[varibl - 1L] = value;
tlvc[varibl - 1L] = simlib.time;
}
else{

/* ------------------------------------------------------------------ */

/* *** Initialize the routine. */

for( ivar = 1L; ivar <= 45L; ivar++ ){
ivar_ = ivar - 1;
area[ivar_] = 0.;
max_[ivar_] = -1.e30;
min_[ivar_] = 1.e30;
preval[ivar_] = 0.;
tlvc[ivar_] = simlib.time;
}
treset = simlib.time;
}
return;
}
else{
```

```
fprintf( stdout, " From SIMLIB:%10ld is improper value for TIMEST variable at time
%10.3f\n",
 varibl, simlib.time );
exit(0);
}
return;
} /* end of function */


void /*FUNCTION*/ filest(list)
long int list;
{
static long int ilist;

/* *** Compute summary statistics for the list. */
ilist = -(20L + list);
timest( 0., ilist );
return;
} /* end of function */


struct t_dat {
long int ncount[100], search;
}dat;
```

```
double /*FUNCTION*/ rannd(istrm)
long int istrm;
{
static long int hi15, hi31, low15, lowprd, ovflow, zi;
static float rannd_v;
static long mult1 = 24112;
static long mult2 = 26143;
static long b2e15 = 32768;
static long b2e16 = 65536;
static long modlus = 2147483647;
static long zrng[100]={1973272912,281629770,20006270,1280689831,2096730329,
 1933576050,913566091,246780520,1363774876,604901985,1511192140,
 1259851944,824064364,150493284,242708531,75253171,1964472944,
 1202299975,233217322,1911216000,726370533,403498145,993232223,
 1103205531,762430696,1922803170,1385516923,76271663,413682397,
 726466604,336157058,1432650381,1120463904,595778810,877722890,
 1046574445,68911991,2088367019,748545416,622401386,2122378830,
 640690903,1774806513,2132545692,2079249579,78130110,852776735,
 1187867272,1351423507,1645973084,1997049139,922510944,2045512870,
 898585771,243649545,1004818771,773686062,403188473,372279877,
 1901633463,498067494,2087759558,493157915,597104727,1530940798,
 1814496276,536444882,1663153658,855503735,67784357,1432404475,
 619691088,119025595,880802310,176192644,1116780070,277854671,
 1366580350,1142483975,2026948561,1053920743,786262391,1792203830,
 1494667770,1923011392,1433700034,1244184613,1147297105,539712780,
```

```
1545929719,190641742,1645390429,264907697,620389253,1502074852,
927711160,364849192,2049576050,638580085,547070247};

/*  Prime modulus multiplicative linear congruential generator
 *  Z(I) = (630360016 * Z(I - 1)) (MOD(2**31 - 1)), based on Marse
 *  and Roberts's portable random-number generator UNIRAN. Multiple
 *  (100) streams are supported, with seeds spaced 100,000 apart.
 *  Throughout, input arguement ISTRM must be an integer giving the
 *  desired stream number. */
/*  Usage: (Three options) */
/*  1. To obtain the next U(0,1) random number from stream ISTRM,
 *     execute
 *        U = RAND(ISTRM)
 *     The real varible U will contain the next random number. */
/*  2. To set the seed for stream ISTRM to a desired value IZSET,
 *     execute
 *        CALL RANDST(IZSET,ISTRM)
 *     where IZSET*must be an integer constant or variable set to the
 *     desired seed, a number between 1 and 2147483646 (inclusive).
 *     Default seeds for all 100 streams are given in the code. */
/*  3. To get the current (most recently used intyeger in the
 *     sequence being generated for stream ISTRM into the INTEGER
 *     variable IZGET, execute
 *        IZGET = IRANDG(ISTRM) */
/*  Force saving of ZRNG between calls. */
```

```
/*      Define the constants */
/*      Set the default seeds for all 100 streams. */
/*      Generate the next random number */
zi = zrng[istrm - 1L];
hi15 = zi/b2e16;
lowprd = (zi - hi15*b2e16)*mult1;
low15 = lowprd/b2e16;
hi31 = hi15*mult1 + low15;
ovflow = hi31/b2e15;
zi = (((lowprd - low15*b2e16) - modlus) + (hi31 - ovflow*b2e15)*
 b2e16) + ovflow;
if( zi < 0L )
zi = zi + modlus;
hi15 = zi/b2e16;
lowprd = (zi - hi15*b2e16)*mult2;
low15 = lowprd/b2e16;
hi31 = hi15*mult2 + low15;
ovflow = hi31/b2e15;
zi = (((lowprd - low15*b2e16) - modlus) + (hi31 - ovflow*b2e15)*
 b2e16) + ovflow;
if( zi < 0L )
zi = zi + modlus;
zrng[istrm - 1L] = zi;
rannd_v = (2L*(zi/256L) + 1L)/16777216.0;
```

```
dat.ncount[istrm - 1L] = dat.ncount[istrm - 1L] + 1L;

return( rannd_v );
} /* end of function */


double /*FUNCTION*/ expon(rmean, istrm)
double rmean;
long int istrm;
{
static float expon_v, u;

/* *** GENERATE A U(0,1) RANDOM VARIABLE. THE FORM OF THIS
STATEMENT
 * *** DEPENDS UPON THE COMPUTER USED. */
u = rannd( istrm );

/* *** GENERATE AN EXPONENTIAL RANDOM VARIABLE WITH RMEAN. */

expon_v = -rmean*log( u );
return( expon_v );
} /* end of function */

double /*FUNCTION*/ erlang(m, rmean, istrm)
long int m;
```

```
double rmean;
long int istrm;
{
static long int i, i_;
static float erlang_v, mexp;

/*	Generate an M-Erlang random variate with mean RMEAN using stream
 *	ISTRM */
mexp = rmean/m;
erlang_v = 0.0;
for( i = 1L; i <= m; i++ ){
i_ = i - 1;
erlang_v = erlang_v + expon( mexp, istrm );

}
return( erlang_v );
} /* end of function */
```

# Appendix 5

Protype header file for Genetic Search Inventory System – seven.h

```
void
crossovr(LOGICAL32[],LOGICAL32[],LOGICAL32[],LOGICAL32[],long,long*,long*,
long*,double,double);
double decode(LOGICAL32[],long);
void decode_parms(long,long,LOGICAL32[],void*);
void demand(void);
void evalu8(void);
double expon(double,long);
void extract(LOGICAL32[],LOGICAL32[],long*,long,long);
LOGICAL32 flip(double,long);
void generation(void);
void init(void);
void initdata(void);
void initialize(void);
void initparm(void);
void initpop(void);
void initrepo(void);
```

```
long ipow(long, long);
long irandi(long,float[],long);
double map_parm(double,double,double,double);
LOGICAL32 mutation(LOGICAL32,double,long*);
double objfunc(double,double);
void ordarv(void);
void prgm_(void);
double rannd(long);
void report(void);
void reportt(void);
double rnd(double);
long select(long,double,void*);
void sim(double,double,float*);
void statistics(long,float*,float*,float*,float*,void*,float*,
float*,float*,float*);
void timing(void);
double unifrm(double,double,long);
void update(void);
void wrtchrom(LOGICAL32[],long);
```

Input file geninv.in

```
60 120 4
0.10  32.0  3.0 1.0 5.0
```

0.167 0.500 0.833 1.000

File seven.c

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

#define TRUE 1
#define FALSE 0
typedef long LOGICAL32;

#define IS_ODD(j)      ((j) & 1 )
#include "seven.h"

struct individual {
LOGICAL32 chrom[30];
float x, v, fitness;
long int parent1, parent2, xsite;
} ;
struct parmparm {
long int lparm;
float parameter, maxparm, minparm;
} ;

struct t_gene {
```

```
struct individual oldpop[100], newpop[100];
long int popsize, lchrom, gen, maxgen;
float pcross, pmutate, sumfit;
long int nmutate, ncross;
float avg, max_, min_;
struct individual population[100];
struct parmparm parms[30];
long int nparms;
float minx, minv, maxx, maxv;
}       gene;
struct t_dat {
long int ncount[100], search;
}       dat;

FILE *fpp;

main( )

{
static long int gen_, i, i_, j, j_, maxsearch, search_;
struct individual newpop[100], oldpop[100], population[100];
struct parmparm parms[30];

if((fpp = fopen("seven.dat","w"))==NULL) {
fprintf(stderr,"cannot open file seven.dat\n");
```

```
exit(1);
}

for( i = 1L; i <= 100L; i++ ){
i_ = i - 1;
dat.ncount[i_] = 0L;
}

fprintf( stderr, "%s\n", " Input the number of searches ------------> "
  );
fscanf( stdin, "%2ld", &maxsearch );

for( dat.search = 1L; dat.search <= maxsearch; dat.search++ ){
search_ = dat.search - 1;
fputc( '\n', stderr );
fputc( '\n', stderr );
fprintf( stderr, "%s\n", " The current search is ------------------> "
  );
fprintf( stderr, "%2ld\n", dat.search );
fputc( '\n', stderr );
fputc( '\n', stderr );

initialize();

for( gene.gen = 1L; gene.gen <= gene.maxgen; gene.gen++ ){
```

```
gen_ = gene.gen - 1;

fprintf( stderr, "%s\n", " The current generation is      ------------> "
  );
fprintf( stderr, "%2ld\n", gene.gen );
generation();
statistics( gene.popsize, &gene.max_, &gene.avg, &gene.min_,
 &gene.sumfit, gene.newpop, &gene.minx, &gene.minv, &gene.maxx,
 &gene.maxv );
report();

/* ADVANCE THE GENERATION */

for( j = 1L; j <= 100L; j++ ){
j_ = j - 1;

gene.oldpop[j_] = gene.newpop[j_];
}
}
}

for( i = 1L; i <= 100L; i++ ){
i_ = i - 1;
if( dat.ncount[i_] != 0L )
{
```

```
fprintf( stdout, "The Count for %ld  is %2ld\n", i, dat.ncount[i_] );
}



        fclose(fpp);}
return(0);
} /* end of function */

double /*FUNCTION*/ decode(chrom, lbits)
LOGICAL32 chrom[];
long int lbits;
{
static long int j, j_;
static float accum, decode_v, powerof2;

accum = 0L;
powerof2 = 1L;

for( j = 1L; j <= lbits; j++ ){
j_ = j - 1;
if( chrom[j_] )
accum = accum + powerof2;
powerof2 = powerof2*2L;
}
```

```
decode_v = accum;

return( decode_v );
} /* end of function */

void /*FUNCTION*/ generation()
{
static long int j, j_, jcross, mate1, mate2;
struct individual newpop[100], oldpop[100], population[100];
struct parmparm parms[30];

/*    CREATE A NEW GENERATION THROUGH SELECT, CROSSOVER, AND
MUTATION
 *   GENERATION ASSUMES AN EVEN NUMBERED POPSIZE */
for( j = 1L; j <= gene.popsize; j += 2L ){
j_ = j - 1;

mate1 = select( gene.popsize, gene.sumfit, gene.oldpop );
mate2 = select( gene.popsize, gene.sumfit, gene.oldpop );

crossovr( gene.oldpop[mate1 - 1L].chrom, gene.oldpop[mate2 - 1L].chrom,
 gene.newpop[j_].chrom, gene.newpop[j_ + 1L].chrom, gene.lchrom,
 &gene.ncross, &gene.nmutate, &jcross, gene.pcross, gene.pmutate );

/* CROSSOVER AND MUTATION - MUTATION EMBEDDED WITHIN
```

```
CROSSOVER */

/* DECODE STRING, EVALUATE FITNESS, & RECORD PARENTAGE DATA
ON BOTH
 * CHILDREN */


decode_parms( gene.nparms, gene.lchrom, gene.newpop[j_].chrom,
 gene.parms );

if (gene.parms[0L].parameter < gene.parms[1L].parameter ) {
      gene.newpop[j_].x = gene.parms[0L].parameter;
      gene.newpop[j_].v = gene.parms[1L].parameter;
   }
  else{
      gene.newpop[j_].v = gene.parms[0L].parameter;
      gene.newpop[j_].x = gene.parms[1L].parameter;
  }

gene.newpop[j_].fitness = objfunc( gene.newpop[j_].x, gene.newpop[j_].v );
gene.newpop[j_].parent1 = mate1;
gene.newpop[j_].parent2 = mate2;
gene.newpop[j_].xsite = jcross;

decode_parms( gene.nparms, gene.lchrom, gene.newpop[j_ + 1L].chrom,
```

```
 gene.parms );

 if (gene.parms[0L].parameter < gene.parms[1L].parameter ) {
       gene.newpop[j_ + 1L].x = gene.parms[0L].parameter;
       gene.newpop[j_ + 1L].v = gene.parms[1L].parameter;
    }
  else {
       gene.newpop[j_ + 1L].v = gene.parms[0L].parameter;
       gene.newpop[j_ + 1L].x = gene.parms[1L].parameter;
}

gene.newpop[j_ + 1L].fitness = objfunc( gene.newpop[j_ + 1L].x,
 gene.newpop[j_ + 1L].v );
gene.newpop[j_ + 1L].parent1 = mate1;
gene.newpop[j_ + 1L].parent2 = mate2;
gene.newpop[j_ + 1L].xsite = jcross;



}
return;
} /* end of function */

long /*FUNCTION*/ select(popsize, sumfit, vp_population)
long int popsize;
double sumfit;
```

```
void *vp_population;
{
struct individual *population = vp_population;
static long int j, j_, select_v;
static float partsum, random;

/* SELECT A SINGLE INDIVIDUAL VIA ROULETTE WHEEL SELECTION */
partsum = 0L;
j = 0L;



random = rannd( 1L )*sumfit;

/* FIND WHEEL  SLOT */

for( j = 1L; j <= popsize; j++ ){
j_ = j - 1;

partsum = partsum + population[j_].fitness;

if( (partsum >= random) || (j == popsize) )
goto L_10;

}
```

```
/* RETURN INDIVIDUAL MEMBER */

L_10:
select_v = j;

return( select_v );
} /* end of function */

LOGICAL32 /*FUNCTION*/ mutation(allelev, pmutate, nmutation)
LOGICAL32 allelev;
double pmutate;
long int *nmutation;
{
static LOGICAL32 mutate, mutation_v;

/* MUATATE AN ALLEL WITH PROBABILITY OF MUTATION, COUNT
NUMBER OF
 * MUTATIONS */
mutate = flip( pmutate, 1L );
if( mutate ){
*nmutation = *nmutation + 1L;
mutation_v = !allelev;
}
else{
mutation_v = allelev;
```

```
}
return( mutation_v );
} /* end of function */

void /*FUNCTION*/ crossovr(parnt1, parnt2, child1, child2, lchrom,
 ncross, nmutate, jcross, pcross, pmutate)
LOGICAL32 parnt1[], parnt2[], child1[], child2[];
long int lchrom, *ncross, *nmutate, *jcross;
double pcross, pmutate;
{
static long int j, j_;

if( flip( pcross, 1L ) ){
*jcross = unifrm( (float)( 1L ), (float)( lchrom - 1L ), 1L );
*ncross = *ncross + 1L;
}
else{
*jcross = lchrom;
}

/* THE FIRST EXCHANGE 1 TO 1, AND 2 TO 2 */

for( j = 1L; j <= *jcross; j++ ){
j_ = j - 1;
```

```
child1[j_] = mutation( parnt1[j_], pmutate, nmutate );
child2[j_] = mutation( parnt2[j_], pmutate, nmutate );
}

/* THE SECOND EXCHANGE 1 TO 2, AND 2 TO 1 */

if( *jcross != lchrom ){
for( j = *jcross + 1L; j <= lchrom; j++ ){
j_ = j - 1;
child1[j_] = mutation( parnt2[j_], pmutate, nmutate );
child2[j_] = mutation( parnt1[j_], pmutate, nmutate );
}
}
return;
} /* end of function */

void /*FUNCTION*/ statistics(popsize, max_, avg, min_, sumfit, vp_population,
 minx, minv, maxx, maxv)
long int popsize;
float *max_, *avg, *min_, *sumfit;
void  *vp_population;
float *minx, *minv, *maxx, *maxv;
{
struct individual *population = vp_population;
static long int j, j_;
```

```
/* INITIALIZE */
*sumfit = population[0L].fitness;
*min_ = population[0L].fitness;
*max_ = population[0L].fitness;
*maxx = population[0L].x;
*maxv = population[0L].v;
*minx = population[0L].x;
*minv = population[0L].v;

for( j = 2L; j <= popsize; j++ ){
j_ = j - 1;
*sumfit = *sumfit + population[j_].fitness;
if( population[j_].fitness > *max_ ){
*max_ = population[j_].fitness;
*maxx = population[j_].x;
*maxv = population[j_].v;
}

if( population[j_].fitness < *min_ ){

*min_ = population[j_].fitness;
*minx = population[j_].x;
*minv = population[j_].v;
}
```

```
}

*avg = *sumfit/popsize;

return;
} /* end of function */

void /*FUNCTION*/ initialize()
{
struct individual newpop[100], oldpop[100], population[100];
struct parmparm parms[30];

gene.ncross = 0L;
gene.nmutate = 0L;
/* INITIALIZATION COORDINATOR */
if( dat.search == 1L ){
initdata();
initparm();
}
initpop();
statistics( gene.popsize, &gene.max_, &gene.avg, &gene.min_, &gene.sumfit,
 gene.oldpop, &gene.minx, &gene.minv, &gene.maxx, &gene.maxv );
initrepo();
return;
} /* end of function */
```

```
struct t_param {
long int nreps;
float highval;
}       param;

void /*FUNCTION*/ initdata()
{
struct individual newpop[100], oldpop[100], population[100];
struct parmparm parms[30];

fprintf( stdout, "%s\n", " ENTER POPULATION SIZE         -----------> "
  );
fscanf( stdin, "%2ld", &gene.popsize );
fprintf( stdout, "%s\n", " ENTER CHROMOSOME LENGTH       -----------> "
  );
fscanf( stdin, "%2ld", &gene.lchrom );
fprintf( stdout, "%s\n", " ENTER MAX. GENERATIONS        -----------> "
  );
fscanf( stdin, "%2ld", &gene.maxgen );
fprintf( stdout, "%s\n", " ENTER CROSSOVER PROBABILITY  -----------> "
  );
fscanf( stdin, "%f", &gene.pcross );
```

```
fprintf( stdout, "%s\n", " ENTER MUTATION PROBABILITY    ----------> "
  );
fscanf( stdin, "%f", &gene.pmutate );
fprintf( stdout, "%s\n", " ENTER NUMBER OF REPLICATIONS ----------> "
  );
fscanf( stdin, "%2ld", &param.nreps );
return;
} /* end of function */

void /*FUNCTION*/ initpop()
{
static long int j, j1, j1_, j_;
struct individual newpop[100], oldpop[100], population[100];
struct parmparm parms[30];

/* INITIALIZE A POPULATION AT RANDOM */
for( j = 1L; j <= gene.popsize; j++ ){
j_ = j - 1;
for( j1 = 1L; j1 <= gene.lchrom; j1++ ){
j1_ = j1 - 1;

/* TOSS OF A FAIR COIN */

gene.oldpop[j_].chrom[j1_] = flip( 0.50, 1L );
```

```
}

/* NOW DECODE THE STRING */

decode_parms( gene.nparms, gene.lchrom, gene.oldpop[j_].chrom,
 gene.parms );

if (gene.parms[0L].parameter < gene.parms[1L].parameter ) {
        gene.oldpop[j_].x = gene.parms[0L].parameter;
        gene.oldpop[j_].v = gene.parms[1L].parameter;
         }
else{
        gene.oldpop[j_].v = gene.parms[0L].parameter;
        gene.oldpop[j_].x = gene.parms[1L].parameter;
         }

gene.oldpop[j_].fitness = objfunc( gene.oldpop[j_].x, gene.oldpop[j_].v );
gene.oldpop[j_].parent1 = 0L;
gene.oldpop[j_].parent2 = 0L;
gene.oldpop[j_].xsite = 0L;

}
return;
} /* end of function */
```

```
void /*FUNCTION*/ initrepo()

{

struct individual newpop[100], oldpop[100], population[100];

struct parmparm parms[30];


/* INITIAL REPORT */

fprintf( stdout, "-------------------------------------------\n" );

fprintf( stdout, "| A SIMPLE GENETIC ALGORITHM - SGA v1.0      |\n" );

fprintf( stdout, "|  (c) James M. Yunker  1991                 |\n" );

fprintf( stdout, "|       All Rights Reserved                  |\n" );

fprintf( stdout, "-------------------------------------------\n" );

fputc( '\n', stdout );

fputc( '\n', stdout );

fputc( '\n', stdout );

fputc( '\n', stdout );

fprintf( stdout, "       SGA Parameters  |\n" );

fputc( '\n', stdout );

fputc( '\n', stdout );

fputc( '\n', stdout );

fprintf( stdout, "  Population Size (popsize)          = %2ld\n",

 gene.popsize );

fprintf( stdout, "  Chromosome length (lchrom)         = %2ld\n",

 gene.lchrom );

fprintf( stdout, "  Maximum # of generations           = %2ld\n",
```

```
 gene.maxgen );
fprintf( stdout, "  Crossover probability (pmutation)  = %3.2f\n",
 gene.pcross );
fprintf( stdout, "  Mutation probability               = %4.3f\n",
 gene.pmutate );
fprintf( stdout, "  The number of replications         = %2ld\n",
 param.nreps );
fprintf( stdout, "  The highval is                     = %6.1f\n",
 param.highval );
fprintf( stdout, "  The search is                      = %2ld\n",
 dat.search );
fputc( '\n', stdout );
fputc( '\n', stdout );
fputc( '\n', stdout );
fputc( '\n', stdout );
fputc( '\n', stdout );
fputc( '\n', stdout );
fputc( '\n', stdout );
fputc( '\n', stdout );
fprintf( stdout, " Initial Generation Statistics\n" );
fputc( '\n', stdout );
fprintf( stdout, "  Initial population maximum fitness = %6.2f\n",
 gene.max_ );
fprintf( stdout, "  Initial population average fitness = %6.2f\n",
 gene.avg );
```

```
fprintf( stdout, " Initial population minimum fitness = %6.2f\n",
 gene.min_ );
fprintf( stdout, " Initial population sum of fitness  = %10.2f\n",
 gene.sumfit );
fputc( '\n', stdout );
return;
} /* end of function */



LOGICAL32 /*FUNCTION*/ flip(prob, istrm)
double prob;
long int istrm;
{
static LOGICAL32 flip_v;
static float u;

u = rannd( istrm );

flip_v = u <= prob;

return( flip_v );
} /* end of function */
```

```
double /*FUNCTION*/ objfunc(x, v)
double x, v;
{
static long int i, i_;
static float objfunc_v, sum, y, yavg;

param.highval = 300L;
sum = 0L;

for( i = 1L; i <= param.nreps; i++ ){
i_ = i - 1;
sim( x, v, &y );
sum = sum + y;
}

yavg = sum/param.nreps;

objfunc_v = param.highval - yavg;
return( objfunc_v );
} /* end of function */


void /*FUNCTION*/ wrtchrom(chrom, lchrom)
LOGICAL32 chrom[];
long int lchrom;
```

```
{
static long int i, i_;

for( i = lchrom; i >= 1L; i-- ){
i_ = i - 1;
if( chrom[i_] ){
fprintf(stdout, "%s", "1");
}
else{
fprintf(stdout, "%s", "0");
}
}
return;
} /* end of function */

void /*FUNCTION*/ report()
{
static long int j, j_, mnbigs, mnsmalls, mxbigs, mxsmalls;
static float adjmax, adjmin;
struct individual newpop[100], oldpop[100], population[100];
struct parmparm parms[30];

if( gene.gen == 1L || gene.gen == gene.maxgen ){
```

```
fprintf( stdout, "--------------------------------------------------\n" );
fputc( '\n', stdout );
fprintf( stdout, "            Population Report\n" );
fputc( '\n', stdout );
fprintf( stdout, "            Generation %2ld\n", gene.gen -
 1L );
fputc( '\n', stdout );
fprintf( stdout, "          string                    s     S    fitness\n" );

for( j = 1L; j <= gene.popsize; j++ ){
j_ = j - 1;
wrtchrom( gene.oldpop[j_].chrom, gene.lchrom );
fprintf( stdout, "   %6.2f %6.2f  %6.2f \n", gene.oldpop[j_].x,
 gene.oldpop[j_].v, gene.oldpop[j_].fitness );
}



/* New String */


fputc( '\n', stdout );
fputc( '\n', stdout );
fprintf( stdout, "            Population Report\n" );
fputc( '\n', stdout );
fprintf( stdout, "            Generation %2ld\n", gene.gen );
```

```
fputc( '\n', stdout );



fprintf( stdout, "          string                    s     S    fitness  parents  xsite\n" );
for( j = 1L; j <= gene.popsize; j++ ){
j_ = j - 1;
wrtchrom( gene.newpop[j_].chrom, gene.lchrom );
fprintf( stdout, "   %6.2f %6.2f  %6.2f    %2ld %2ld     %2ld \n", gene.newpop[j_].x,
 gene.newpop[j_].v, gene.newpop[j_].fitness, gene.newpop[j_].parent1,
 gene.newpop[j_].parent2, gene.newpop[j_].xsite );
}

fputc( '\n', stdout );
fputc( '\n', stdout );
fputc( '\n', stdout );
}
else{
fputc( '\n', stdout );
fprintf( stdout, "        ************************* \n" );
fprintf( stdout, "  The intervening chromosomes left out for space reasons\n" );
fprintf( stdout, "        ************************* \n" );
fputc( '\n', stdout );
}
```

```
/* Generation statistics and accumulated values */
fprintf( stdout, "  Note: Generation%2ld and accumulated statistics:\n",
 gene.gen );
fprintf( stdout, "  max=%6.2f min=%6.2f avg= %6.2f sum= %10.2f\n",
 gene.max_, gene.min_, gene.avg, gene.sumfit );
fprintf( stdout, "  nmutation= %2ld ncross= %2ld\n", gene.nmutate,
 gene.ncross );
fputc( '\n', stdout );
fputc( '\n', stdout );

adjmax = param.highval - gene.min_;
mxsmalls = rnd( gene.minx );
mxbigs = rnd( gene.minv );
fprintf( stdout, "  The Population maximum cost is        $ %8.2f\n",
 adjmax );
fprintf( stdout, "  The SMALLS value for the maximum cost is %3ld\n",
 mxsmalls );
fprintf( stdout, "  The BIGS value for the maximum cost is   %3ld\n",
 mxbigs );
fputc( '\n', stdout );
fputc( '\n', stdout );

adjmin = param.highval - gene.max_;
mnsmalls = rnd( gene.maxx );
mnbigs = rnd( gene.maxv );
```

```
fprintf( stdout, "  The Population minimum cost is         $ %8.2f\n",
 adjmin );
fprintf( stdout, "  The SMALLS value for the minimum cost is %3ld\n",
 mnsmalls );
fprintf( stdout, "  The BIGS value for the minimum cost is   %3ld\n",
 mnbigs );
       fprintf(fpp, "%3ld %3ld %2ld %8.2f\n", mnsmalls, mnbigs, gene.gen, adjmin);
fputc( '\n', stdout );
fputc( '\n', stdout );

return;
} /* end of function */



double /*FUNCTION*/ rnd(z)
double z;
{
static float rnd_v, z1, z2;

z1 = (long)( z );
z2 = z1 + 0.50;
if( z < z2 ){
rnd_v = z1;
}
else{
```

```
rnd_v = z1 + 1L;
}
return( rnd_v );
} /* end of function */


void /*FUNCTION*/ initparm()
{
static long int i, i_, lparm;
static float maxparm, minparm;
struct individual newpop[100], oldpop[100], population[100];
struct parmparm parms[30];

fputc( '\n', stdout );
fputc( '\n', stdout );
fprintf( stdout, "This is a s,S inventory model. There are 2 decision\n" );
fprintf( stdout, "variables s and S to input. Note the above when \n" );
fprintf( stdout, "answering the following questions :\n" );
fputc( '\n', stdout );
fputc( '\n', stdout );
fprintf( stdout, "%s\n", " Input the number of parameters -----------> "
  );
fscanf( stdin, "%ld", &gene.nparms );
fputc( '\n', stdout );
fputc( '\n', stdout );
```

```
for( i = 1L; i <= gene.nparms; i++ ){
i_ = i - 1;
fprintf( stdout, "FOR PARAMETER%2ld \n", i );
fputc( '\n', stdout );
fprintf( stdout, "%s\n", " Enter the minimum value        ------------> "
  );
fscanf( stdin, "%f", &minparm );
gene.parms[i_].minparm = minparm;
fprintf( stdout, "%s\n", " Enter the maximum value        ------------> "
  );
fscanf( stdin, "%f", &maxparm );
gene.parms[i_].maxparm = maxparm;
fprintf( stdout, "%s\n", " Enter the length               ------------> "
  );
fscanf( stdin, "%2ld", &lparm );
gene.parms[i_].lparm = lparm;
fputc( '\n', stdout );
fputc( '\n', stdout );
}
fputc( '\n', stdout );
fputc( '\n', stdout );


freopen("OUTPUT7","w",stdout);
```

```
for( i = 1L; i <= gene.nparms; i++ ){
i_ = i - 1;
fprintf( stdout, "FOR PARAMETER%2ld \n", i );
fputc( '\n', stdout );
fprintf( stdout, " the minimum value   ------------> %g \n",
 gene.parms[i_].minparm );
fputc( '\n', stdout );
fprintf( stdout, " the maximum value   ------------> %g \n",
 gene.parms[i_].maxparm );
fputc( '\n', stdout );
fprintf( stdout, " the length          ------------> %2ld \n",
 gene.parms[i_].lparm );
fputc( '\n', stdout );
fputc( '\n', stdout );
}
fputc( '\n', stdout );
fputc( '\n', stdout );

return;
} /* end of function */


void /*FUNCTION*/ extract(chromfrom, chromato, jposition, lchrom,
 lparm)
LOGICAL32 chromfrom[], chromato[];
```

```
long int *jposition, lchrom, lparm;
{
static long int j, jtarget;

j = 1L;
jtarget = *jposition + lparm - 1L;
if( jtarget > lchrom )
jtarget = lchrom;
while( *jposition <= jtarget ){
chromato[j - 1L] = chromfrom[*jposition - 1L];
*jposition = *jposition + 1L;
j = j + 1L;
}
return;
} /* end of function */

double /*FUNCTION*/ map_parm(x, maxparm, minparm, fullscale)
double x, maxparm, minparm, fullscale;
{
static float map_parm_v;

map_parm_v = minparm + (maxparm - minparm)/fullscale*x;
return( map_parm_v );
} /* end of function */
```

```
void /*FUNCTION*/ decode_parms(nparms, lchrom, chrom, vp_parms)
long int nparms, lchrom;
LOGICAL32 chrom[];
void *vp_parms;
{
struct parmparm *parms = vp_parms;
static LOGICAL32 chromtemp[30];
static long int j, jposition;
static float parameter, scale;

jposition = 1L;
j = 1L;
while( j <= nparms ){
scale = ipow(2L, parms[j - 1L].lparm) - 1L;
if( parms[j - 1L].lparm > 0L ){
extract( chrom, chromtemp, &jposition, lchrom, parms[j - 1L].lparm );
parms[j - 1L].parameter = map_parm( decode( chromtemp,
 parms[j - 1L].lparm ), parms[j - 1L].maxparm, parms[j - 1L].minparm,
 scale );
}
else{
parameter = 0L;
}
j = j + 1L;
}
```

```
return;
} /* end of function */



double /*FUNCTION*/ unifrm(a, b, istrm)
double a, b;
long int istrm;
{
static float u, unifrm_v;

/* *** GENERATE A U(0,1) RANDOM VARIABLE. THE FORM OF THIS
STATEMENT
 * *** DEPENDS UPON THE COMPUTER USED. */
u = rannd( istrm );

/* *** GENERATE A U(A,B) RANDOM VARIABLE. */

unifrm_v = a + (u*(b - a));
return( unifrm_v );
} /* end of function */



double /*FUNCTION*/ rannd(istrm)
long int istrm;
{
```

```
static long int hi15, hi31, low15, lowprd, ovflow, zi;
static float rannd_v;
static long mult1 = 24112;
static long mult2 = 26143;
static long b2e15 = 32768;
static long b2e16 = 65536;
static long modlus = 2147483647;
static long zrng[100]={1973272912,281629770,20006270,1280689831,2096730329,
 1933576050,913566091,246780520,1363774876,604901985,1511192140,
 1259851944,824064364,150493284,242708531,75253171,1964472944,
 1202299975,233217322,1911216000,726370533,403498145,993232223,
 1103205531,762430696,1922803170,1385516923,76271663,413682397,
 726466604,336157058,1432650381,1120463904,595778810,877722890,
 1046574445,68911991,2088367019,748545416,622401386,2122378830,
 640690903,1774806513,2132545692,2079249579,78130110,852776735,
 1187867272,1351423507,1645973084,1997049139,922510944,2045512870,
 898585771,243649545,1004818771,773686062,403188473,372279877,
 1901633463,498067494,2087759558,493157915,597104727,1530940798,
 1814496276,536444882,1663153658,855503735,67784357,1432404475,
 619691088,119025595,880802310,176192644,1116780070,277854671,
 1366580350,1142483975,2026948561,1053920743,786262391,1792203830,
 1494667770,1923011392,1433700034,1244184613,1147297105,539712780,
 1545929719,190641742,1645390429,264907697,620389253,1502074852,
 927711160,364849192,2049576050,638580085,547070247};
```

```
/*   Prime modulus multiplicative linear congruential generator
 *   Z(I) = (630360016 * Z(I - 1)) (MOD(2**31 - 1)), based on Marse
 *   and Roberts's portable random-number generator UNIRAN. Multiple
 *   (100) streams are supported, with seeds spaced 100,000 apart.
 *   Throughout, input arguement ISTRM must be an integer giving the
 *   desired stream number. */
/*   Usage: (Three options) */
/*   1. To obtain the next U(0,1) random number from stream ISTRM,
 *     execute
 *        U = RAND(ISTRM)
 *     The real varible U will contain the next random number. */
/*   2. To set the seed for stream ISTRM to a desired value IZSET,
 *     execute
 *        CALL RANDST(IZSET,ISTRM)
 *     where IZSET*must be an integer constant or variable set to the
 *     desired seed, a number between 1 and 2147483646 (inclusive).
 *     Default seeds for all 100 streams are given in the code. */
/*   3. To get the current (most recently used intyeger in the
 *     sequence being generated for stream ISTRM into the INTEGER
 *     variable IZGET, execute
 *        IZGET = IRANDG(ISTRM) */
/*   Force saving of ZRNG between calls. */
/*   Define the constants */
/*   Set the default seeds for all 100 streams. */
/*   Generate the next random number */
```

```
zi = zrng[istrm - 1L];
hi15 = zi/b2e16;
lowprd = (zi - hi15*b2e16)*mult1;
low15 = lowprd/b2e16;
hi31 = hi15*mult1 + low15;
ovflow = hi31/b2e15;
zi = (((lowprd - low15*b2e16) - modlus) + (hi31 - ovflow*b2e15)*
 b2e16) + ovflow;
if( zi < 0L )
zi = zi + modlus;
hi15 = zi/b2e16;
lowprd = (zi - hi15*b2e16)*mult2;
low15 = lowprd/b2e16;
hi31 = hi15*mult2 + low15;
ovflow = hi31/b2e15;
zi = (((lowprd - low15*b2e16) - modlus) + (hi31 - ovflow*b2e15)*
 b2e16) + ovflow;
if( zi < 0L )
zi = zi + modlus;
zrng[istrm - 1L] = zi;
rannd_v = (2L*(zi/256L) + 1L)/16777216.0;

dat.ncount[istrm - 1L] = dat.ncount[istrm - 1L] + 1L;

return( rannd_v );
```

```
} /* end of function */

struct t_model {
float aminus;
long int amount;
float aplus;
long int bigs;
float h, incrmc;
long int initil, invlev;
float mdemdt;
long int nevnts, next, nmnths;
float pi, setupc;
long int smalls;
float time, tlevnt, tne[5], tordc, acost;
}       model;
struct t_random {
long int nvalue;
float probd[25];
}       random;

void /*FUNCTION*/ sim(x, v, y)
double x, v;
float *y;
{
```

```
static LOGICAL32 skip;
static long int i;
FILE *fp;

model.smalls = x;
model.bigs = v;


if( !skip ){


if (( fp = fopen("geninv.in","r"))==NULL) {
  fprintf(stderr, "cannot open file geninv.in\n");
  exit(1);
}

/* *** SPECIFY THE NUMBER OF EVENT TYPES FOR THE TIMING ROUTINE
*/

model.nevnts = 5L;

/* *** READ INPUT PARAMETERS */

fscanf( fp, "%ld %ld %ld", &model.initil, &model.nmnths,
 &random.nvalue );
```

```
fscanf( fp, "%f %f %f %f %f", &model.mdemdt, &model.setupc,
 &model.incrmc, &model.h, &model.pi );
fscanf( fp, "" );
for( i = 1L; i <= random.nvalue; i++ ){
fscanf( fp, "%f", &random.probd[i - 1L] );
}


fputc( '\n', stdout );
fputc( '\n', stdout );
fprintf( stdout, "  The initial level          %3ld\n", model.initil );
fprintf( stdout, "  The number months          %3ld\n", model.nmnths );
fprintf( stdout, "  The number of demand sizes %3ld\n", random.nvalue );
fprintf( stdout, "  The mean demand time       %8.4f\n", model.mdemdt );
fprintf( stdout, "  The set up costs         $ %8.4f\n", model.setupc );
fprintf( stdout, "  The incremental costs    $ %8.4f\n", model.incrmc );
fprintf( stdout, "  The holding costs        $ %8.4f\n", model.h );
fprintf( stdout, "  The value for PI           %8.4f\n", model.pi );
fprintf( stdout, "  The probability distributions is" );
for( i = 1L; i <= random.nvalue; i++ ){
fprintf( stdout, "%8.4f", random.probd[i - 1L] );
}
fprintf( stdout, "\n" );
fputc( '\n', stdout );
fputc( '\n', stdout );
```

```
fputc( '\n', stdout );
fputc( '\n', stdout );
fputc( '\n', stdout );
fputc( '\n', stdout );

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

skip = TRUE;
fclose(fp);
}

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

/* *** RUN THE SIMULATION VARYING THE INVENTORY POLICY */

/* *** INITIALIZE THE SIMULATION */

init();

/* *** DETERMINE THE NEXT EVENT */

while( TRUE ){
```

```
timing();

/* *** CALL THE APPROPRIATE EVENT ROUTINE */

if( model.next == 2L ){
demand();
}
else if( model.next == 3L ){
goto L_10;
}
else if( model.next == 4L ){
evalu8();
}
else if( model.next == 5L ){
model.tordc = 0L;
model.aplus = 0L;
model.aminus = 0L;
model.tne[4L] = 1.e30;
}
else{
ordarv();
}
}
L_10:
reportt();
```

```
*y = model.acost;
return;
} /* end of function */

void /*FUNCTION*/ init()
{

/* *** INITIALIZE THE SIMULATION CLOCK */
model.time = 0L;

/* *** INITIALIZE THE STATE VARIABLES */

model.invlev = model.initil;
model.tlevnt = 0.;

/* *** INITIALIZE THE STATISTICAL COUNTERS */

model.tordc = 0.;
model.aplus = 0.;
model.aminus = 0.;

/* *** INITIALIZE THE EVENT LIST. SINCE NO ORDER IS OUSTANDING, THE
TIME
 * *** OF THE NEXT ORDER ARRIVAL IS SET TO INFINITY.
```

```
* *** SET WARM-UP PERIOD TO 10. */

model.tne[0L] = 1.e30;
model.tne[1L] = expon( model.mdemdt, 1L );
model.tne[2L] = model.nmnths;
model.tne[3L] = 0.;
model.tne[4L] = 10.;
return;
} /* end of function */

void /*FUNCTION*/ timing()
{
static long int i, i_;
static float rmin;

rmin = 1.e29;
model.next = 0L;

/* *** DETERMINE THE EVENT TYPE OF THE NEXT EVENT TO OCCUR */

for( i = 1L; i <= model.nevnts; i++ ){
i_ = i - 1;
if( model.tne[i_] < rmin ){
rmin = model.tne[i_];
model.next = i;
```

```
}
}

/* *** IF THE EVENT IS EMPTY (I.E., NEXT=0), STOP THE SIMULATION.
 * *** OTHERWISE, ADVANCE THE SIMULATION CLOCK */

if( model.next > 0L ){
model.time = model.tne[model.next - 1L];
return;
}
else{
fprintf( stdout, "1      EVENT LIST EMPTY\n" );
exit(0);
}
return;
} /* end of function */


void /*FUNCTION*/ ordarv()
{

/* *** UPDATE 'APLUS' AND 'AMINUS' */
update();

/* *** INCREMENT THE INVENTORY LEVEL BY THE AMOUNT ORDERED. */
```

```
model.invlev = model.invlev + model.amount;

/* *** SINCE NO ORDER IS NOW OUTSTANDING. SET THE TIME OF THE
NEXT ORDER
 * *** ARRIVAL TO INFINITY */

model.tne[0L] = 1.e30;
return;
} /* end of function */

void /*FUNCTION*/ demand()
{
static long int dsize;

/* *** UPDATE 'APLUS' AND 'AMINUS' */
update();

/* *** GENERATE THE DEMAND SIZE */

dsize = irandi( random.nvalue, random.probd, 1L );

/* *** DECREMENT THE INVENTORY LEVEL BY THE DEMAND SIZE. */

model.invlev = model.invlev - dsize;
```

```
/* *** SCHEDULE THE NEXT DEMAND */

model.tne[1L] = model.time + expon( model.mdemdt, 1L );
return;
} /* end of function */

void /*FUNCTION*/ evalu8()
{

/* *** IF THE INVENTORY LEVEL IS LESS THAN 'SMALLS,'PLACE AN ORDER
FOR
 * *** 'BIGS'-'INVLEV'ITEMS */
if( model.invlev < model.smalls ){
model.amount = model.bigs - model.invlev;
model.tordc = model.tordc + model.setupc + (model.incrmc*model.amount);

/* *** SCHEDULE THE ARRIVAL OF THE ORDER */

model.tne[0L] = model.time + unifrm( .5, 1., 1L );
}

/* *** SCHEDULE THE NEXT INVENTORY EVALUATION */

model.tne[3L] = model.time + 1.;
```

```
return;
} /* end of function */



void /*FUNCTION*/ reportt()
{
static float ahldc, aordc, ashrc;

/* *** UPDATE 'APLUS' AND 'AMINUS' */
update();

/* *** COMPUTE THE ESTIMATES OF THE DESIRED MEASURES OF
PERFORMANCE */

aordc = model.tordc/model.nmnths;
ahldc = model.h*(model.aplus/model.nmnths);
ashrc = model.pi*(model.aminus/model.nmnths);
model.acost = aordc + ahldc + ashrc;
return;
} /* end of function */

void /*FUNCTION*/ update()
{
static float tsle;
```

```
/* *** COMPUTE THE TIME SINCE THE LAST EVENT WHICH CHANGED THE
INTVENTORY
 * *** LEVEL */
tsle = model.time - model.tlevnt;
model.tlevnt = model.time;

/* *** DETERMINE WHETHER THE INVENTORY LEVEL DURING THE
PREVIOUS INTERVAL
 * *** WAS NEGATIVE, ZERO OR POSITIVE */

if( model.invlev != 0L ){
if( model.invlev > 0L ){

/* *** SINCE THE INVENTORY LEVEL DURING THE PREVIOUS INTERVAL
WAS
 * *** POSITIVE, UPDATE 'APLUS' */

model.aplus = model.aplus + (model.invlev*tsle);
}
else{

/* *** SINCE THE INVENTORY LEVEL DURING THE PREVIOUS INTERVAL
WAS
 * *** NEGATIVE, UPDATE 'AMINUS.' */
```

```
model.aminus = model.aminus + (-model.invlev*tsle);

/* *** THE INVENTORY LEVEL DURING THE PREVIOUS INTVERVAL WAS
ZERO. */

}
}
return;
} /* end of function */



double /*FUNCTION*/ expon(rmean, istrm)
double rmean;
long int istrm;
{
static float expon_v, u;

/* *** GENERATE A U(0,1) RANDOM VARIABLE. THE FORM OF THIS
STATEMENT
 * *** DEPENDS UPON THE COMPUTER USED. */
u = rannd( istrm );

/* *** GENERATE AN EXPONENTIAL RANDOM VARIABLE WITH RMEAN. */

expon_v = -rmean*log( u );
```

```
return( expon_v );
} /* end of function */



long /*FUNCTION*/ irandi(nvalue, probd, istrm)
long int nvalue;
float probd[];
long int istrm;
{
static long int i, i_, irandi_v;
static float u;

/*      Generate a U(0,1) random variable from stream ISTREAM */
u = rannd( istrm );

/*      Generate a random integer between 1 and NVALUE in accordance with
 *      the cumulative distribution funvtion PROBD. */

for( i = 1L; i <= (nvalue - 1L); i++ ){
i_ = i - 1;
if( u < probd[i_] )
goto L_10;
}
irandi_v = nvalue;
```

```
return( irandi_v );
L_10:
irandi_v = i;
return( irandi_v );
} /* end of function */

long ipow( m, n )      /* returns:  m^n */
long m;        /* base */
long n;        /* exponent */
{
long p; /* holds partial product */

if( m == 0 )
return( 0L );

if( n < 0 ){    /* test for negative exponent */
n = -n;
m = 1/m;
}

p = IS_ODD(n) ? m : 1;  /* test & set zero power */

while( n >>= 1 ){      /* now do the other powers */
m *= m;                /* sq previous power of m */
if( IS_ODD(n) ) /* if low order bit set */
```

```
p *= m;          /*     then, multiply partial product by latest power of m */
}

return( p );
}
```

# Appendix 6

Prototype header for Genetic Search Computer System – eight.h

```
void arrive(double);
void cancel(double);
void
crossovr(LOGICAL32[],LOGICAL32[],LOGICAL32[],LOGICAL32[],long,long*,long*,
long*,double,double);
double decode(LOGICAL32[],long);
void decode_parms(long,long,LOGICAL32[],void*);
void endrun(void);
double expon(double,long);
void extract(LOGICAL32[],LOGICAL32[],long*,long,long);
void file(long,long);
void filest(long);
LOGICAL32 flip(double,long);
void generation(void);
void initdata(void);
void initialize(void);
void initlk(void);
void initparm(void);
```

```
void initpop(void);
void initrepo(void);
long int ipow(long, long);
double map_parm(double,double,double,double);
LOGICAL32 mutation(LOGICAL32,double,long*);
double objfunc(double,double);
double powi (double, long int);
void prgm_(void);
double rannd(long);
void removve(long,long);
void report(void);
void reportt(void);
void sampst(double,long);
long select(long,double,void*);
void sim(double,double,float*);
void start(void);
void statistics(long,float*,float*,float*,float*,void*,float*,
float*,float*,float*);
void timest(double,long);
void timing(void);
double unifrm(long,long,long);
void wrtchrom(LOGICAL32[],long);
```

File genecomp.in

0.800

40

5000

25

File eight.c

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

#define TRUE 1
#define FALSE 0
typedef long LOGICAL32;

#define IS_ODD(j)       ((j) & 1 )

#include "./eight.h"

struct individual {
LOGICAL32 chrom[30];
float x, v, fitness;
long int parent1, parent2, xsite;
} ;
struct parmparm {
long int lparm;
float parameter, maxparm, minparm;
} ;

struct t_gene {
```

```
struct individual oldpop[100], newpop[100];
long int popsize, lchrom, gen, maxgen;
float pcross, pmutate, sumfit;
long int nmutate, ncross;
float avg, max_, min_;
struct individual population[100];
struct parmparm parms[30];
long int nparms;
float minx, minv, maxx, maxv;
}       gene;
struct t_dat {
long int ncount[100], search;
}       dat;

FILE *fpp;

main( )
{
static long int gen_, i, i_, j, j_, maxsearch, search_;
struct individual newpop[100], oldpop[100], population[100];
struct parmparm parms[30];



if ((fpp = fopen("eight.dat","w"))==NULL)  {
   fprintf(stderr,"cannot open file eight.dat\n");
```

```
  exit(1);
}



for( i = 1L; i <= 100L; i++ ){
i_ = i - 1;
dat.ncount[i_] = 0L;
}

fprintf( stderr, "%s\n", " Input the number of searches ----------------> "
  );
fscanf( stdin, "%ld", &maxsearch );

for( dat.search = 1L; dat.search <= maxsearch; dat.search++ ){
search_ = dat.search - 1;
fputc( '\n', stderr );
fputc( '\n', stderr );
fprintf( stderr, "%s\n", " The current search is ----------------------->"
  );
fprintf( stderr, "%2ld\n", dat.search );
fputc( '\n', stderr );
fputc( '\n', stderr );

initialize();
```

```
for( gene.gen = 1L; gene.gen <= gene.maxgen; gene.gen++ ){
gen_ = gene.gen - 1;

fprintf( stderr, "%s\n", " The current generation is      ------------> "
  );
fprintf( stderr, "%2ld\n", gene.gen );
generation();
statistics( gene.popsize, &gene.max_, &gene.avg, &gene.min_,
 &gene.sumfit, gene.newpop, &gene.minx, &gene.minv, &gene.maxx,
 &gene.maxv );
report();

/* ADVANCE THE GENERATION */

for( j = 1L; j <= 100L; j++ ){
j_ = j - 1;
gene.oldpop[j_] = gene.newpop[j_];
}

}
}

for( i = 1L; i <= 100L; i++ ){
i_ = i - 1;
if( dat.ncount[i_] != 0L )
```

```
{
fprintf( stdout, "The Count for %2ld  is %2ld \n", i, dat.ncount[i_] );
}

fclose(fpp);  }
return(0);
} /* end of function */

double /*FUNCTION*/ decode(chrom, lbits)
LOGICAL32 chrom[];
long int lbits;
{
static long int j, j_;
static float accum, decode_v, powerof2;

accum = 0L;
powerof2 = 1L;

for( j = 1L; j <= lbits; j++ ){
j_ = j - 1;
if( chrom[j_] )
accum = accum + powerof2;
powerof2 = powerof2*2L;
}
```

```
decode_v = accum;


return( decode_v );
} /* end of function */


void /*FUNCTION*/ generation()
{
static long int j, j_, jcross, mate1, mate2;
struct individual newpop[100], oldpop[100], population[100];
struct parmparm parms[30];


/*     CREATE A NEW GENERATION THROUGH SELECT, CROSSOVER, AND
MUTATION
 *    GENERATION ASSUMES AN EVEN NUMBERED POPSIZE */
for( j = 1L; j <= gene.popsize; j += 2L ){
j_ = j - 1;


mate1 = select( gene.popsize, gene.sumfit, gene.oldpop );
mate2 = select( gene.popsize, gene.sumfit, gene.oldpop );


crossovr( gene.oldpop[mate1 - 1L].chrom, gene.oldpop[mate2 - 1L].chrom,
 gene.newpop[j_].chrom, gene.newpop[j_ + 1L].chrom, gene.lchrom,
 &gene.ncross, &gene.nmutate, &jcross, gene.pcross, gene.pmutate );


/* CROSSOVER AND MUTATION - MUTATION EMBEDDED WITHIN
```

```
CROSSOVER */

/* DECODE STRING, EVALUATE FITNESS, & RECORD PARENTAGE DATA
ON BOTH
 * CHILDREN */


decode_parms( gene.nparms, gene.lchrom, gene.newpop[j_].chrom,
 gene.parms );
gene.newpop[j_].x = gene.parms[0L].parameter;
gene.newpop[j_].v = gene.parms[1L].parameter;
gene.newpop[j_].fitness = objfunc( gene.newpop[j_].x, gene.newpop[j_].v );
gene.newpop[j_].parent1 = mate1;
gene.newpop[j_].parent2 = mate2;
gene.newpop[j_].xsite = jcross;

decode_parms( gene.nparms, gene.lchrom, gene.newpop[j_ + 1L].chrom,
 gene.parms );
gene.newpop[j_ + 1L].x = gene.parms[0L].parameter;
gene.newpop[j_ + 1L].v = gene.parms[1L].parameter;
gene.newpop[j_ + 1L].fitness = objfunc( gene.newpop[j_ + 1L].x,
 gene.newpop[j_ + 1L].v );
gene.newpop[j_ + 1L].parent1 = mate1;
gene.newpop[j_ + 1L].parent2 = mate2;
gene.newpop[j_ + 1L].xsite = jcross;
```

```
}
return;
} /* end of function */

long /*FUNCTION*/ select(popsize, sumfit, vp_population)
long int popsize;
double sumfit;
void  *vp_population;
{
struct individual *population = vp_population;
static long int j, j_, select_v;
static float partsum, random;

/* SELECT A SINGLE INDIVIDUAL VIA ROULETTE WHEEL SELECTION */
partsum = 0L;
j = 0L;



random = rannd( 1L )*sumfit;

/* FIND WHEEL  SLOT */

for( j = 1L; j <= popsize; j++ ){
```

```
j_ = j - 1;

partsum = partsum + population[j_].fitness;

if( (partsum >= random) || (j == popsize) )
goto L_10;

}

/* RETURN INDIVIDUAL MEMBER */

L_10:
select_v = j;

return( select_v );
} /* end of function */

LOGICAL32 /*FUNCTION*/ mutation(allelev, pmutate, nmutation)
LOGICAL32 allelev;
double pmutate;
long int *nmutation;
{
static LOGICAL32 mutate, mutation_v;

/* MUATATE AN ALLEL WITH PROBABILITY OF MUTATION, COUNT
```

```
NUMBER OF
 * MUTATIONS */
mutate = flip( pmutate, 1L );
if( mutate ){
*nmutation = *nmutation + 1L;
mutation_v = !allelev;
}
else{
mutation_v = allelev;

}
return( mutation_v );
} /* end of function */

void /*FUNCTION*/ crossovr(parnt1, parnt2, child1, child2, lchrom,
 ncross, nmutate, jcross, pcross, pmutate)
LOGICAL32 parnt1[], parnt2[], child1[], child2[];
long int lchrom, *ncross, *nmutate, *jcross;
double pcross, pmutate;
{
static long int j, j_;

if( flip( pcross, 1L ) ){
*jcross = unifrm( 1L, lchrom - 1L, 1L );
*ncross = *ncross + 1L;
```

```
}
else{
*jcross = lchrom;
}

/* THE FIRST EXCHANGE 1 TO 1, AND 2 TO 2 */

for( j = 1L; j <= *jcross; j++ ){
j_ = j - 1;
child1[j_] = mutation( parnt1[j_], pmutate, nmutate );
child2[j_] = mutation( parnt2[j_], pmutate, nmutate );
}

/* THE SECOND EXCHANGE 1 TO 2, AND 2 TO 1 */

if( *jcross != lchrom ){
for( j = *jcross + 1L; j <= lchrom; j++ ){
j_ = j - 1;
child1[j_] = mutation( parnt2[j_], pmutate, nmutate );
child2[j_] = mutation( parnt1[j_], pmutate, nmutate );
}
}
return;
} /* end of function */
```

```
void /*FUNCTION*/ statistics(popsize, max_, avg, min_, sumfit, vp_population,
 minx, minv, maxx, maxv)
long int popsize;
float *max_, *avg, *min_, *sumfit;
void  *vp_population;
float *minx, *minv, *maxx, *maxv;
{
struct individual *population = vp_population;
static long int j, j_;

/* INITIALIZE */
*sumfit = population[0L].fitness;
*min_ = population[0L].fitness;
*max_ = population[0L].fitness;
*maxx = population[0L].x;
*maxv = population[0L].v;
*minx = population[0L].x;
*minv = population[0L].v;


for( j = 2L; j <= popsize; j++ ){
j_ = j - 1;
*sumfit = *sumfit + population[j_].fitness;
if( population[j_].fitness > *max_ ){
*max_ = population[j_].fitness;
```

```
*maxx = population[j_].x;
*maxv = population[j_].v;
}

if( population[j_].fitness < *min_ ){

*min_ = population[j_].fitness;
*minx = population[j_].x;
*minv = population[j_].v;
}
}

*avg = *sumfit/popsize;

return;
} /* end of function */

void /*FUNCTION*/ initialize()
{
struct individual newpop[100], oldpop[100], population[100];
struct parmparm parms[30];

gene.ncross = 0L;
gene.nmutate = 0L;
```

```
/* INITIALIZATION COORDINATOR */
if( dat.search == 1L ){
initdata();
initparm();
}
initpop();
statistics( gene.popsize, &gene.max_, &gene.avg, &gene.min_, &gene.sumfit,
 gene.oldpop, &gene.minx, &gene.minv, &gene.maxx, &gene.maxv );
initrepo();
return;
} /* end of function */


struct t_param {
long int nreps;
float highval;
}       param;

void /*FUNCTION*/ initdata()
{
struct individual newpop[100], oldpop[100], population[100];
struct parmparm parms[30];

fprintf( stdout, "%s\n", " ENTER POPULATION SIZE          -----------> "
```

```
  );
fscanf( stdin, "%ld", &gene.popsize );
fprintf( stdout, "%s\n", " ENTER CHROMOSOME LENGTH      ------------> "
  );
fscanf( stdin, "%ld", &gene.lchrom );
fprintf( stdout, "%s\n", " ENTER MAX. GENERATIONS       ------------> "
  );
fscanf( stdin, "%ld", &gene.maxgen );
fprintf( stdout, "%s\n", " ENTER CROSSOVER PROBABILITY  ------------> "
  );
fscanf( stdin, "%f", &gene.pcross );
fprintf( stdout, "%s\n", " ENTER MUTATION PROBABILITY   ------------> "
  );
fscanf( stdin, "%f", &gene.pmutate );
fprintf( stdout, "%s\n", " ENTER NO. OF REPLICATIONS    ------------> "
  );
fscanf( stdin, "%ld", &param.nreps );
return;
} /* end of function */

void /*FUNCTION*/ initpop()
{
static long int j, j1, j1_, j_;
struct individual newpop[100], oldpop[100], population[100];
struct parmparm parms[30];
```

```
/* INITIALIZE A POPULATION AT RANDOM */
for( j = 1L; j <= gene.popsize; j++ ){
j_ = j - 1;
for( j1 = 1L; j1 <= gene.lchrom; j1++ ){
j1_ = j1 - 1;

/* TOSS OF A FAIR COIN */

gene.oldpop[j_].chrom[j1_] = flip( 0.50, 1L );

}

/* NOW DECODE THE STRING */

decode_parms( gene.nparms, gene.lchrom, gene.oldpop[j_].chrom,
 gene.parms );
gene.oldpop[j_].x = gene.parms[0L].parameter;
gene.oldpop[j_].v = gene.parms[1L].parameter;
gene.oldpop[j_].fitness = objfunc( gene.oldpop[j_].x, gene.oldpop[j_].v );
gene.oldpop[j_].parent1 = 0L;
gene.oldpop[j_].parent2 = 0L;
gene.oldpop[j_].xsite = 0L;

}
```

```
return;
} /* end of function */



void /*FUNCTION*/ initrepo()
{
struct individual newpop[100], oldpop[100], population[100];
struct parmparm parms[30];

/* INITIAL REPORT */
fprintf( stdout, "-------------------------------------------\n" );
fprintf( stdout, "| A SIMPLE GENETIC ALGORITHM - SGA v1.0       |\n" );
fprintf( stdout, "|  (c) James M. Yunker  1991                  |\n" );
fprintf( stdout, "|      All Rights Reserved                    |\n" );
fprintf( stdout, "-------------------------------------------\n" );
fputc( '\n', stdout );
fputc( '\n', stdout );
fputc( '\n', stdout );
fputc( '\n', stdout );
fputc( '\n', stdout );
fputc( '\n', stdout );
fprintf( stdout, "      SGA Parameters  |\n" );
fputc( '\n', stdout );
fprintf( stdout, "  Population Size (popsize)         = %2ld\n",
 gene.popsize );
```

```
fprintf( stdout, "  Chromosome length (lchrom)        = %2ld\n",
 gene.lchrom );
fprintf( stdout, "  Maximum # of generations          = %2ld\n",
 gene.maxgen );
fprintf( stdout, "  Crossover probability (pmutation) = %3.2f\n",
 gene.pcross );
fprintf( stdout, "  Mutation probability              = %4.3f\n",
 gene.pmutate );
fprintf( stdout, "  The number of replications        = %2ld\n",
 param.nreps );
fprintf( stdout, "  The highval is                    = %6.1f\n",
 param.highval );
fprintf( stdout, "  The search is                     = %2ld\n",
 dat.search );
fputc( '\n', stdout );
fputc( '\n', stdout );
fputc( '\n', stdout );
fputc( '\n', stdout );
fputc( '\n', stdout );
fputc( '\n', stdout );
fputc( '\n', stdout );
fputc( '\n', stdout );
fprintf( stdout, " Initial Generation Statistics\n" );
fputc( '\n', stdout );
fprintf( stdout, "  Initial population maximum fitness = %6.2f\n",
```

```
gene.max_ );
fprintf( stdout, "  Initial population average fitness = %6.2f\n",
gene.avg );
fprintf( stdout, "  Initial population minimum fitness = %6.2f\n",
gene.min_ );
fprintf( stdout, "  Initial population sum of fitness  = %10.2f\n",
gene.sumfit );
fputc( '\n', stdout );
return;
} /* end of function */



LOGICAL32 /*FUNCTION*/ flip(prob, istrm)
double prob;
long int istrm;
{
static LOGICAL32 flip_v;
static float u;

u = rannd( istrm );

flip_v = u <= prob;

return( flip_v );
} /* end of function */
```

```
double /*FUNCTION*/ objfunc(x, v)
double x, v;
{
static long int i, i_;
static float objfunc_v, sum, y, yavg;

param.highval = 80000L;
sum = 0L;

for( i = 1L; i <= param.nreps; i++ ){
i_ = i - 1;
sim( x, v, &y );
sum = sum + y;
}

yavg = sum/param.nreps;

objfunc_v = param.highval - yavg;
return( objfunc_v );
} /* end of function */


void /*FUNCTION*/ wrtchrom(chrom, lchrom)
```

```
LOGICAL32 chrom[];
long int lchrom;
{
static long int i, i_;



for( i = lchrom; i >= 1L; i-- ){
i_ = i - 1;
if( chrom[i_] ){
fprintf(stdout, "%s", "1");
}
else{
fprintf(stdout, "%s", "0");
}
}
return;
} /* end of function */



void /*FUNCTION*/ report()
{


static long int j, j_;
static float adjmax, adjmin;
struct individual newpop[100], oldpop[100], population[100];
```

```
struct parmparm parms[30];

if( gene.gen == 1L || gene.gen == gene.maxgen ){
fprintf( stdout, "-------------------------------------------------\n" );
fputc( '\n', stdout );
fprintf( stdout, "            Population Report\n" );
fputc( '\n', stdout );
fprintf( stdout, "             Generation %2ld\n", gene.gen -
 1L );
fputc( '\n', stdout );
fprintf( stdout, "          string              qntm    ohd    fitness\n" );

for( j = 1L; j <= gene.popsize; j++ ){
j_ = j - 1;
wrtchrom( gene.oldpop[j_].chrom, gene.lchrom );
fprintf( stdout, "    %4.3f   %4.3f  %4.2f \n", gene.oldpop[j_].x,
 gene.oldpop[j_].v, gene.oldpop[j_].fitness );
}

/* New String */

fputc( '\n', stdout );
fputc( '\n', stdout );
fprintf( stdout, "            Population Report\n" );
fputc( '\n', stdout );
```

```
fprintf( stdout, "                Generation %2ld\n", gene.gen );
fputc( '\n', stdout );
fprintf( stdout, "            string                  qntm    ohd    fitness    parents  xsite\n" );

for( j = 1L; j <= gene.popsize; j++ ){
j_ = j - 1;
wrtchrom( gene.newpop[j_].chrom, gene.lchrom );
fprintf( stdout, "    %4.3f   %4.3f  %4.2f   %2ld %2ld      %2ld \n", gene.newpop[j_].x,
 gene.newpop[j_].v, gene.newpop[j_].fitness, gene.newpop[j_].parent1,
 gene.newpop[j_].parent2, gene.newpop[j_].xsite );
 }

fputc( '\n', stdout );
fputc( '\n', stdout );
fputc( '\n', stdout );
}
else{
fputc( '\n', stdout );
fprintf( stdout, "        ************************ \n" );
fprintf( stdout, "  The intervening chromosomes left out for space reasons\n" );
fprintf( stdout, "        ************************ \n" );
fputc( '\n', stdout );
}

fprintf( stdout, "  Note: Generation%2ld and accumulated statistics:\n",
```

```
 gene.gen );
fprintf( stdout, "  max= %6.2f min= %6.2f avg= %6.2f sum= %10.2f\n",
 gene.max_, gene.min_, gene.avg, gene.sumfit );
fprintf( stdout, "  nmutation= %2ld ncross= %2ld\n", gene.nmutate,
 gene.ncross );
fputc( '\n', stdout );
fputc( '\n', stdout );

adjmax = param.highval - gene.min_;
fprintf( stdout, "  The Population maximum cost  %9.4f\n",
 adjmax );
fprintf( stdout, "  QUANTM for the maximum cost  %9.4f\n",
 gene.minx );
fprintf( stdout, "  OVERHD for the maximum cost  %9.4f\n",
 gene.minv );
fputc( '\n', stdout );
fputc( '\n', stdout );

adjmin = param.highval - gene.max_;
fprintf( stdout, "  The Population minimum cost  %9.4f\n",
 adjmin );
fprintf( stdout, "  QUANTM for the minimum cost  %9.4f\n",
 gene.maxx );
fprintf( stdout, "  OVERHD for the minimum cost  %9.4f\n",
 gene.maxv );
```

```
fputc( '\n', stdout );
fputc( '\n', stdout );
fprintf (fpp, "%8.3f %8.4f %2ld %8.3f\n", gene.maxx, gene.maxv ,gene.gen , adjmin);

return;
} /* end of function */




void /*FUNCTION*/ initparm()
{
static long int i, i_, lparm;
static float maxparm, minparm;
struct individual newpop[100], oldpop[100], population[100];
struct parmparm parms[30];

fputc( '\n', stdout );
fputc( '\n', stdout );
fprintf( stdout, " The model is of a remote computer system.\n" );
fprintf( stdout, " It consists of two parameters.\n" );
fprintf( stdout, " The first parameter is a quantum of computer time\n" );
fprintf( stdout, " The second is system overhead\n" );
fprintf( stdout, " Keep this in mind when answering the questions\n" );
```

```
fputc( '\n', stdout );
fputc( '\n', stdout );
fprintf( stdout, "%s\n", " Input the number of parameters ------------> "
  );
fscanf( stdin, "%2ld", &gene.nparms );

fputc( '\n', stdout );
fputc( '\n', stdout );
for( i = 1L; i <= gene.nparms; i++ ){
i_ = i - 1;
fprintf( stdout, "FOR PARAMETER %2ld\n", i );
fputc( '\n', stdout );
fprintf( stdout, "%s\n", " Enter the minimum value        ------------> "
  );
fscanf( stdin, "%f", &minparm );
gene.parms[i_].minparm = minparm;
fprintf( stdout, "%s\n", " Enter the maximum value        ------------> "
  );
fscanf( stdin, "%f", &maxparm );
gene.parms[i_].maxparm = maxparm;
fprintf( stdout, "%s\n", " Enter the length               ------------> "
  );
fscanf( stdin, "%2ld", &lparm );
gene.parms[i_].lparm = lparm;
fputc( '\n', stdout );
```

```
fputc( '\n', stdout );
}

freopen("OUTPUT8","w",stdout);

for( i = 1L; i <= gene.nparms; i++ ){
i_ = i - 1;
fprintf( stdout, "FOR PARAMETER %2ld \n", i );
fputc( '\n', stdout );
fprintf( stdout, "the minimum value     ------------> %4.3f \n",
 gene.parms[i_].minparm );
fputc( '\n', stdout );
fprintf( stdout, "the maximum value     ------------> %4.3f \n",
 gene.parms[i_].maxparm );
fputc( '\n', stdout );
fprintf( stdout, "the length            ------------> %2ld \n",
 gene.parms[i_].lparm );
fputc( '\n', stdout );
fputc( '\n', stdout );
fputc( '\n', stdout );
}
fputc( '\n', stdout );
fputc( '\n', stdout );
fputc( '\n', stdout );
```

```
return;
} /* end of function */



void /*FUNCTION*/ extract(chromfrom, chromato, jposition, lchrom,
 lparm)
LOGICAL32 chromfrom[], chromato[];
long int *jposition, lchrom, lparm;
{
static long int j, jtarget;

j = 1L;
jtarget = *jposition + lparm - 1L;
if( jtarget > lchrom )
jtarget = lchrom;
while( *jposition <= jtarget ){
chromato[j - 1L] = chromfrom[*jposition - 1L];
*jposition = *jposition + 1L;
j = j + 1L;
}
return;
} /* end of function */

double /*FUNCTION*/ map_parm(x, maxparm, minparm, fullscale)
double x, maxparm, minparm, fullscale;
```

```
{
static float map_parm_v;

map_parm_v = minparm + (maxparm - minparm)/fullscale*x;
return( map_parm_v );
} /* end of function */

void /*FUNCTION*/ decode_parms(nparms, lchrom, chrom, vp_parms)
long int nparms, lchrom;
LOGICAL32 chrom[];
void  *vp_parms;
{
struct parmparm *parms = vp_parms;
static LOGICAL32 chromtemp[30];
static long int j, jposition;
static float parameter, scale;

jposition = 1L;
j = 1L;
while( j <= nparms ){
scale = ipow(2L, parms[j - 1L].lparm) - 1L;
if( parms[j - 1L].lparm > 0L ){
extract( chrom, chromtemp, &jposition, lchrom, parms[j - 1L].lparm );
parms[j - 1L].parameter = map_parm( decode( chromtemp,
 parms[j - 1L].lparm ), parms[j - 1L].maxparm, parms[j - 1L].minparm,
```

```
scale );
}
else{
parameter = 0L;
}
j = j + 1L;
}
return;
} /* end of function */



double /*FUNCTION*/ unifrm(a, b, istrm)
long int a, b, istrm;
{
static float u, unifrm_v;

/* *** GENERATE A U(0,1) RANDOM VARIABLE. THE FORM OF THIS
STATEMENT
 * *** DEPENDS UPON THE COMPUTER USED. */
u = rannd( istrm );

/* *** GENERATE A U(A,B) RANDOM VARIABLE. */

unifrm_v = a + (u*(b - a));
return( unifrm_v );
```

```
} /* end of function */


double /*FUNCTION*/ rannd(istrm)
long int istrm;
{
static long int hi15, hi31, low15, lowprd, ovflow, zi;
static float rannd_v;
static long mult1 = 24112;
static long mult2 = 26143;
static long b2e15 = 32768;
static long b2e16 = 65536;
static long modlus = 2147483647;
static long zrng[100]={1973272912,281629770,20006270,1280689831,2096730329,
 1933576050,913566091,246780520,1363774876,604901985,1511192140,
 1259851944,824064364,150493284,242708531,75253171,1964472944,
 1202299975,233217322,1911216000,726370533,403498145,993232223,
 1103205531,762430696,1922803170,1385516923,76271663,413682397,
 726466604,336157058,1432650381,1120463904,595778810,877722890,
 1046574445,68911991,2088367019,748545416,622401386,2122378830,
 640690903,1774806513,2132545692,2079249579,78130110,852776735,
 1187867272,1351423507,1645973084,1997049139,922510944,2045512870,
 898585771,243649545,1004818771,773686062,403188473,372279877,
 1901633463,498067494,2087759558,493157915,597104727,1530940798,
 1814496276,536444882,1663153658,855503735,67784357,1432404475,
```

```
619691088,119025595,880802310,176192644,1116780070,277854671,
1366580350,1142483975,2026948561,1053920743,786262391,1792203830,
1494667770,1923011392,1433700034,1244184613,1147297105,539712780,
1545929719,190641742,1645390429,264907697,620389253,1502074852,
927711160,364849192,2049576050,638580085,547070247};

/*   Prime modulus multiplicative linear congruential generator
 *   Z(I) = (630360016 * Z(I - 1)) (MOD(2**31 - 1)), based on Marse
 *   and Roberts's portable random-number generator UNIRAN. Multiple
 *   (100) streams are supported, with seeds spaced 100,000 apart.
 *   Throughout, input arguement ISTRM must be an integer giving the
 *   desired stream number. */
/*   Usage: (Three options) */
/*   1. To obtain the next U(0,1) random number from stream ISTRM,
 *     execute
 *       U = RAND(ISTRM)
 *     The real varible U will contain the next random number. */
/*   2. To set the seed for stream ISTRM to a desired value IZSET,
 *     execute
 *       CALL RANDST(IZSET,ISTRM)
 *     where IZSET*must be an integer constant or variable set to the
 *     desired seed, a number between 1 and 2147483646 (inclusive).
 *     Default seeds for all 100 streams are given in the code. */
/*   3. To get the current (most recently used intyeger in the
 *     sequence being generated for stream ISTRM into the INTEGER
```

```
*       variable IZGET, execute
*          IZGET = IRANDG(ISTRM) */
/*     Force saving of ZRNG between calls. */
/*     Define the constants */
/*     Set the default seeds for all 100 streams. */
/*     Generate the next random number */
zi = zrng[istrm - 1L];
hi15 = zi/b2e16;
lowprd = (zi - hi15*b2e16)*mult1;
low15 = lowprd/b2e16;
hi31 = hi15*mult1 + low15;
ovflow = hi31/b2e15;
zi = (((lowprd - low15*b2e16) - modlus) + (hi31 - ovflow*b2e15)*
 b2e16) + ovflow;
if( zi < 0L )
zi = zi + modlus;
hi15 = zi/b2e16;
lowprd = (zi - hi15*b2e16)*mult2;
low15 = lowprd/b2e16;
hi31 = hi15*mult2 + low15;
ovflow = hi31/b2e15;
zi = (((lowprd - low15*b2e16) - modlus) + (hi31 - ovflow*b2e15)*
 b2e16) + ovflow;
if( zi < 0L )
zi = zi + modlus;
```

```
zrng[istrm - 1L] = zi;
rannd_v = (2L*(zi/256L) + 1L)/16777216.0;

dat.ncount[istrm - 1L] = dat.ncount[istrm - 1L] + 1L;

return( rannd_v );
} /* end of function */


struct t_model {
float mserve, mthink;
long int nterml, numjob;
float overhd, quantm;
long int totjob;
float cost;
}       model;
struct t_simlib {
long int ldecr, levent, lfirst, lincr, llast, lrank[25], lsize[25],
 maxatr, next;
float time, trnsfr[10];
}       simlib;

void /*FUNCTION*/ sim(x, v, y)
double x, v;
float *y;
```

```
{
static LOGICAL32 skip;
static long int termnl, termnl_;
FILE *fp;

model.quantm = x;
model.overhd = v;

if( !skip ){

if ((fp = fopen("genecomp.in","r"))==NULL) {
   fprintf(stderr,"cannot open file genecomp.in\n");
   exit(1);
}

/* *** READ INPUT PARAMETERS. */

fscanf( fp, "%f", &model.mserve );
fscanf( fp, "%ld", &model.nterml );
fscanf( fp, "%ld", &model.totjob );
fscanf( fp, "%f", &model.mthink );

fputc( '\n', stdout );
```

```
fputc( '\n', stdout );
fprintf( stdout, "  The mean service time is    %5.2f\n", model.mserve );
fprintf( stdout, "  The number of terminals is  %5ld\n", model.nterml );
fprintf( stdout, "  The total number of jobs is %5ld\n", model.totjob );
fprintf( stdout, "  The mean think time is      %5.2f\n", model.mthink );
fputc( '\n', stdout );
fputc( '\n', stdout );

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

skip = TRUE;
fclose(fp);
}

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

/* *** RUN THE SIMULATION VARYING THE TERMINALS. */

model.numjob = 0L;

/* *** INITIALIZE SUPPORT ROUTINES. */

initlk();
simlib.maxatr = 4L;
```

```
/* *** SCHEDULE THE END OF THE WARM-UP PERIOD */

simlib.trnsfr[0L] = 50.0;
simlib.trnsfr[1L] = 3.;
file( 3L, 25L );

/* *** SCHEDULE THE FIRST ARRIVAL TO THE CPU FROM EACH
TERMINAL. */

for( termnl = 1L; termnl <= model.nterml; termnl++ ){
termnl_ = termnl - 1;
simlib.trnsfr[0L] = expon( model.mthink, 1L );
simlib.trnsfr[1L] = 1.;
simlib.trnsfr[2L] = termnl;
file( 3L, 25L );
}

/* *** DETERMINE THE NEXT EVENT. */

while( TRUE ){
timing();

/* *** CALL THE APPROPRIATE EVENT ROUTINE. */

if( simlib.next == 2L ){
```

```
endrun();
}
else if( simlib.next == 3L ){
sampst( 0., 0L );
timest( 0., 0L );
}
else if( simlib.next == 4L ){
goto L_10;
}
else{
arrive( simlib.trnsfr[2L] );
}
}
L_10:
reportt();

*y = model.cost;

return;
} /* end of function */

void /*FUNCTION*/ arrive(origin)
double origin;
{
static long int int_;
```

```
static float jobtim;

/* *** PLACE THE JOB IN THE QUEUE.
 * *** NOTE THAT THE FOLLOWING DATA ARE STORED FOR EACH JOB:
 * ***      1. TIME OF ARRIVAL TO THE QUEUE.
 * ***      2. THE TERMINAL OF ORIGIN.
 * ***      3. THE NUMBER OF QUANTUM LENGTH CPU RUNS REQUIRED.
 * ***      4. THE REMAINING AMOUNT OF SERVICE. */
jobtim = expon( model.mserve, 1L );
simlib.trnsfr[0L] = simlib.time;
simlib.trnsfr[1L] = origin;
int_ = jobtim/model.quantm;
simlib.trnsfr[2L] = int_;
simlib.trnsfr[3L] = jobtim - (int_*model.quantm);
file( 2L, 1L );

/* *** IF THE CPU IS IDLE, START A CPU RUN. */

if( simlib.lsize[1L] == 0L )
start();
return;
} /* end of function */

void /*FUNCTION*/ start()
{
```

```
static float runtim;

/* *** REMOVE JOB FROM QUEUE */
removve( 1L, 1L );

/* *** DETERMINE REQUIRED CPU TIME FOR THIS RUN. */

if( simlib.trnsfr[2L] > 0.5 ){

/* *** A FULL QUANTM IS NEEDED. */

runtim = model.quantm + model.overhd;
simlib.trnsfr[2L] = simlib.trnsfr[2L] - 1.;
}
else{

/* *** LESS THAN A FULL QUANTM IS NEEDED. */

runtim = simlib.trnsfr[3L] + model.overhd;
simlib.trnsfr[2L] = -1.;
}

/* *** PLACE JOB IN THE CPU. */

file( 1L, 2L );
```

```
/* *** SCHEDULE THE END OF THE CPU RUN. */

simlib.trnsfr[0L] = simlib.time + runtim;
simlib.trnsfr[1L] = 2.;
file( 3L, 25L );
return;
} /* end of function */

void /*FUNCTION*/ endrun()
{
static float origin, resptm;

/* *** REMOVE JOB FROM THE CPU */
removve( 1L, 2L );

/* *** IF THIS JOB IS DONE, SCHEDULE ANOTHER ARRIVAL FOR THE SAME
TERMINA */

if( simlib.trnsfr[2L] > -0.5 ){

/* *** SINCE THE JOB IS NOT FINISHED, PLACE IT AT THE END OF THE
QUEUE */

file( 2L, 1L );
```

```
start();
}
else{
resptm = simlib.time - simlib.trnsfr[0L];
sampst( resptm, 1L );
origin = simlib.trnsfr[1L];
simlib.trnsfr[0L] = simlib.time + expon( model.mthink, 1L );
simlib.trnsfr[1L] = 1.;
simlib.trnsfr[2L] = origin;
file( 3L, 25L );

/* *** INCREMENT THE NUMBER OF COMPLETED JOBS. IF ENOUGH JOBS
ARE DONE,
 * *** SCHEDULE THE END OF THE SIMULATION */

model.numjob = model.numjob + 1L;
if( model.numjob >= model.totjob ){
simlib.trnsfr[0L] = simlib.time;
simlib.trnsfr[1L] = 4.;
file( 1L, 25L );

/* *** IF THE QUEUE IS NOT EMPTY, START ANOTHER JOB. */

}
else if( simlib.lsize[0L] > 0L ){
```

```
start();
}
}
return;
} /* end of function */



void /*FUNCTION*/ reportt()
{
static float arespt, avgniq, utiliz;

sampst( 0., -1L );
arespt = simlib.trnsfr[0L];
filest( 1L );
avgniq = simlib.trnsfr[0L];
filest( 2L );
utiliz = simlib.trnsfr[0L];
model.cost = arespt*1000L + avgniq*1000L + (powi((0.005 - model.overhd)/
 0.005, 2L))*10000L + (powi((0.5 - model.quantm)/0.5, 2L))*10000L;

return;
} /* end of function */
```

```
struct t_llists {
long int head[25], linkpr[1000], linksr[1000];
float master[10][1000];
long int nar, tail[25];
}       llists;

void /*FUNCTION*/ initlk()
{
static long int i, i_, list, list_, row, row_;

/* *** Initialize links. */
for( row = 1L; row <= 1000L; row++ ){
row_ = row - 1;
llists.linkpr[row_] = 0L;
llists.linksr[row_] = row + 1L;
}
llists.linksr[999L] = 0L;

/* *** Initialize list attributes. */

for( list = 1L; list <= 25L; list++ ){
list_ = list - 1;
llists.head[list_] = 0L;
llists.tail[list_] = 0L;
simlib.lsize[list_] = 0L;
```

```
simlib.lrank[list_] = 0L;
}

/* *** Initialize the TRANSFR */

for( i = 1L; i <= 10L; i++ ){
i_ = i - 1;
simlib.trnsfr[i_] = 0.0;
}

/* *** Initialize mnemonics for record location in lists. */

simlib.lfirst = 1L;
simlib.llast = 2L;
simlib.lincr = 3L;
simlib.ldecr = 4L;

/* *** Initialize mnemonic for event list number. */

simlib.levent = 25L;

/* *** Initialize system attributes. */

simlib.time = 0.;
llists.nar = 1L;
```

```
simlib.lrank[simlib.levent - 1L] = 1L;
simlib.maxatr = 10L;

/* *** Initialize statistical routines. */

sampst( 0., 0L );
timest( 0., 0L );

/* *** Initialize output unit number for SIMLIB error messages. */

return;
} /* end of function */

void /*FUNCTION*/ file(option, list)
long int option, list;
{
static long int ahead, behind, ihead, itail, item, item_, row;
static float size;

/* *** IF THE MASTER STORAGE ARRAY IS FULL, STOP THE SIMULATION.
*/
if( llists.nar <= 0L ){
fprintf( stdout, "          MASTER STORAGE ARRAY OVERFLOW AT TIME
%10.3e\n",
 simlib.time );
```

```
/* *** IF LIST VALUE IS IMPROPER, STOP THE SIMULATION. */

}
else if( (list >= 1L) && (list <= 25L) ){

/* *** INCREMENT THE LIST SIZE. */

simlib.lsize[list - 1L] = simlib.lsize[list - 1L] + 1L;

/* *** IF THE OPTION VALUE IS IMPROPER, STOP THE SIMULATION. */

if( (option >= 1L) && (option <= 4L) ){

/* *** FILE ACCORDING TO THE DESIRED OPTION. */

if( option != 1L ){
if( option != 2L ){

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

/* *** THE LIST IS RANKED, DETERMINE THE ITEM ON WHICH THE LIST IS
TO
 * *** BE RANKED. */
```

```
item = simlib.lrank[list - 1L];

/* *** IF AN INVALID ITEM HAS BEEN SPECIFIED, STOP THE SIMULATION.
*/

if( !((item >= 1L) && (item <= simlib.maxatr))
  ){
fprintf( stdout, "%10ld IS AN IMPROPER VALUE FOR THE RANK OF LIST
%12ld\n",
 item, list );
exit(0);
}
else if( simlib.lsize[list - 1L] == 1L ){
goto L_40;
}
else{

/* *** SEARCH THE LIST FOR THE PROPER LOCATION. */

row = llists.head[list - 1L];
while( TRUE ){
if( option == 4L ){

/*     RANK THE LIST IN DECREASING ORDER. */
```

```
if( simlib.trnsfr[item - 1L] > llists.master[item - 1L][row - 1L] )
goto L_10;

/* *** RANK THE LIST IN INCREASING ORDER. */

}
else if( simlib.trnsfr[item - 1L] < llists.master[item - 1L][row - 1L] ){

/* *** THE CORRECT LOCATION HAS BEEN FOUND. */

goto L_10;
}

/* *** CONTINUE SEARCHING, CONSIDER THE NEXT ROW. */

behind = row;
row = llists.linksr[behind - 1L];

/* *** IF THE LAST ROW CONSIDERED WAS NOT THE TAIL OF THE LIST,
 * *** CONTINUE. */

if( llists.tail[list - 1L] == behind )
goto L_20;
}
```

```
/* *** INSERT BEFORE LAST RECORD EXAMINED. */

L_10:
if( row == llists.head[list - 1L] )
goto L_30;

/* *** INSERT IN THE PROPER LOCATION BETWEEN THE PRECEEDING
AND
 * *** SUCCEEDING RECORDS (BEHIND AND AHEAD). */

ahead = llists.linksr[behind - 1L];
row = llists.nar;
llists.nar = llists.linksr[row - 1L];
if( llists.nar > 0L )
llists.linkpr[llists.nar - 1L] = 0L;
llists.linkpr[row - 1L] = behind;
llists.linksr[behind - 1L] = row;
llists.linkpr[ahead - 1L] = row;
llists.linksr[row - 1L] = ahead;

/* *** GO TO TRANSFER THE DATA. */

goto L_50;
}
}
```

```
/* ******************************************************************** */

/* *** INSERT AFTER THE LAST RECORD IN THE LIST. */

L_20:
if( simlib.lsize[list - 1L] == 1L )
goto L_40;
row = llists.nar;
llists.nar = llists.linksr[row - 1L];
if( llists.nar > 0L )
llists.linkpr[llists.nar - 1L] = 0L;
itail = llists.tail[list - 1L];
llists.linkpr[row - 1L] = itail;
llists.linksr[itail - 1L] = row;
llists.linksr[row - 1L] = 0L;
llists.tail[list - 1L] = row;

/* *** GO TO TRANSFER THE DATA. */

goto L_50;
}

/* ******************************************************************** */
```

```
/* *** INSERT BEFORE THE FIRST RECORD IN THE LIST. */

L_30:
if( simlib.lsize[list - 1L] != 1L ){
row = llists.nar;
llists.nar = llists.linksr[row - 1L];
if( llists.nar > 0L )
llists.linkpr[llists.nar - 1L] = 0L;
ihead = llists.head[list - 1L];
llists.linkpr[ihead - 1L] = row;
llists.linksr[row - 1L] = ihead;
llists.linkpr[row - 1L] = 0L;
llists.head[list - 1L] = row;

/* *** GO TO TRANSFER THE DATA. */

goto L_50;
}

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

/* *** INSERT THE FIRST RECORD IN THE LIST. */

L_40:
```

```
row = llists.nar;
llists.nar = llists.linksr[row - 1L];
if( llists.nar > 0L )
llists.linkpr[llists.nar - 1L] = 0L;
llists.linksr[row - 1L] = 0L;
llists.head[list - 1L] = row;
llists.tail[list - 1L] = row;

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

/* *** TRANSFER THE DATA. */

L_50:
;
for( item = 1L; item <= simlib.maxatr; item++ ){
item_ = item - 1;
llists.master[item_][row - 1L] = simlib.trnsfr[item_];
}

/* *** UPDATE THE AREA UNDER THE NUMBER IN LIST CURVE. */

size = simlib.lsize[list - 1L];
timest( size, 20L + list );
return;
}
```

```
else{
fprintf( stdout, "%10ldIS AN IMPROPER VALUE FOR FILE OPTION AT
TIME%10.3e\n",
 option, simlib.time );
}
}
else{
fprintf( stdout, "%10ld IS AN IMPROPER VALUE FOR FILE AT TIME %10.3e\n",
 list, simlib.time );
}
exit(0);
} /* end of function */

void /*FUNCTION*/ removve(option, list)
long int option, list;
{
static long int ihead, itail, item, item_, row;
static float size;

/* *** IF THE LIST IS EMPTY, STOP THE SIMULATION. */
if( !((list >= 1L) && (list <= 25L)) ){
fprintf( stdout, "%10ld IS AN IMPROPER VALUE FOR REMOVE LIST AT TIME
%10.3e\n",
 list, simlib.time );
```

```
/* *** IF THE LIST IS EMPTY, STOP THE SIMULATION. */

}
else if( simlib.lsize[list - 1L] > 0L ){

/* *** DECREMENT THE LIST SIZE. */

simlib.lsize[list - 1L] = simlib.lsize[list - 1L] - 1L;

/* *** IF THE OPTION VALUE IS IMPROPER, STOP THE SIMULATION. */

if( (option == 1L) || (option == 2L) ){

/* *** IF THERE IS MORE THAN ONE RECORD IN THE LIST, CONTINUE. */

if( simlib.lsize[list - 1L] == 0L ){

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

/* REMOVE THE ONLY RECORD IN THE LIST. */

row = llists.head[list - 1L];
llists.head[list - 1L] = 0L;
llists.tail[list - 1L] = 0L;
```

```
/* *** REMOVE ACCORDING TO THE DESIRED OPTION. */


}
else if( option == 2L ){


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


/* *** REMOVE THE LAST RECORD IN THE LIST. */


row = llists.tail[list - 1L];
itail = llists.linkpr[row - 1L];
llists.linksr[itail - 1L] = 0L;
llists.tail[list - 1L] = itail;


/* *** GO TO TRANSFER THE DATA. */


}
else{


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


/* *** REMOVE THE FIRST RECORD IN THE LIST. */


row = llists.head[list - 1L];
ihead = llists.linksr[row - 1L];
```

```
llists.linkpr[ihead - 1L] = 0L;
llists.head[list - 1L] = ihead;

/* *** GO TO TRANSFER THE DATA. */

}

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

/* *** TRANSFER THE DATA. */

llists.linksr[row - 1L] = llists.nar;
llists.linkpr[row - 1L] = 0L;
llists.nar = row;
for( item = 1L; item <= simlib.maxatr; item++ ){
item_ = item - 1;
simlib.trnsfr[item_] = llists.master[item_][row - 1L];
}

/* *** UPDATE THE AREA UNDER THE NUMBER IN LIST CURVE. */

size = simlib.lsize[list - 1L];
timest( size, 20L + list );
return;
}
```

```
else{
fprintf( stdout, "%10ld IS AN IMPROPER VALUE FOR REMOVE OPTION AT
TIME %10.3e\n",
 option, simlib.time );
}
}
else{
fprintf( stdout, "     UNDERFLOW OF LIST %2ld AT TIME %10.3e\n",
 list, simlib.time );
}
exit(0);
} /* end of function */




void /*FUNCTION*/ timing()
{

/* *** Remove the first event from the event list. */
removve( simlib.lfirst, simlib.levent );

/* *** Check for a time reversal. */

if( simlib.trnsfr[0L] >= simlib.time ){
```

```
/* *** Advance the simulation clock. */

simlib.time = simlib.trnsfr[0L];
simlib.next = simlib.trnsfr[1L];
return;
}
else{
fprintf( stdout, " From SIMLIB: Attempt to schedule an event of type %3.0f\n at time
%10.3f when the clock is %10.3f\n",
 simlib.trnsfr[1L], simlib.trnsfr[0L], simlib.time );
exit(0);
}
return;
} /* end of function */

void /*FUNCTION*/ cancel(etype)
double etype;
{
static long int ahead, behind, item, item_, row;
static float high, low, size, value;

/* *** SEARCH THE EVENT LIST. */
if( simlib.lsize[simlib.levent - 1L] != 0L ){
row = llists.head[simlib.levent - 1L];
low = etype - 0.1;
```

```
high = etype + 0.1;
while( TRUE ){
value = llists.master[1L][row - 1L];
if( (low < value) && (high > value) )
goto L_10;

/* *** GO TO THE NEXT EVENT. */

if( row == llists.tail[simlib.levent - 1L] )
return;
row = llists.linksr[row - 1L];
}

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

/* *** CANCEL THIS EVENT. */

L_10:
if( row == llists.head[simlib.levent - 1L] ){

/* *** REMOVE THE FIRST EVENT IN THE EVENT LIST. */

removve( simlib.lfirst, simlib.levent );
}
else if( row == llists.tail[simlib.levent - 1L] ){
```

```
/* *** REMOVE THE LAST EVENT IN THE EVENT LIST. */

removve( simlib.llast, simlib.levent );
}
else{

/* *** REMOVE THIS EVENT WHICH IS SOMEWHERE IN THE MIDDLE OF
THE EVENT
 * *** LIST. */

ahead = llists.linksr[row - 1L];
behind = llists.linkpr[row - 1L];
llists.linksr[behind - 1L] = ahead;
llists.linkpr[ahead - 1L] = behind;
llists.linksr[row - 1L] = llists.nar;
llists.linkpr[row - 1L] = 0L;
llists.nar = row;
simlib.lsize[simlib.levent - 1L] = simlib.lsize[simlib.levent - 1L] -
 1L;

/* *** PLACE THE ATTRIBUTE OF THE CANCELED EVENT IN THE TRNSFR
ARRAY. */

for( item = 1L; item <= simlib.maxatr; item++ ){
```

```
item_ = item - 1;
simlib.trnsfr[item_] = llists.master[item_][row - 1L];
}

/* *** UPDATE THE AREA UNDER THE NUMBER IN LIST CURVE. */

size = simlib.lsize[simlib.levent - 1L];
timest( size, 45L );
}
}
return;
} /* end of function */

void /*FUNCTION*/ sampst(value, varibl)
double value;
long int varibl;
{
static long int ivar, ivar_, nobs[20];
static float max_[20], min_[20], sum[20];

/* *** If the variable value is improper, stop the simulation. */
if( varibl >= -20L && varibl <= 20L ){

/* *** Execute the desired option. */
```

```
if( varibl < 0L ){

/* ------------------------------------------------------------------ */

/* *** Report the results. */

ivar = -varibl;
simlib.trnsfr[0L] = 0.;
simlib.trnsfr[1L] = nobs[ivar - 1L];
simlib.trnsfr[2L] = max_[ivar - 1L];
simlib.trnsfr[3L] = min_[ivar - 1L];
if( nobs[ivar - 1L] != 0L )
simlib.trnsfr[0L] = sum[ivar - 1L]/simlib.trnsfr[1L];
}
else if( varibl > 0L ){

/* ------------------------------------------------------------------ */

/* *** Collect data. */

sum[varibl - 1L] = sum[varibl - 1L] + value;
if( value > max_[varibl - 1L] )
max_[varibl - 1L] = value;
if( value < min_[varibl - 1L] )
```

```
min_[varibl - 1L] = value;
nobs[varibl - 1L] = nobs[varibl - 1L] + 1L;
}
else{

/* ------------------------------------------------------------------ */

/* *** Initialize the routine. */

for( ivar = 1L; ivar <= 20L; ivar++ ){
ivar_ = ivar - 1;
sum[ivar_] = 0.;
max_[ivar_] = -1.e30;
min_[ivar_] = 1.e30;
nobs[ivar_] = 0L;
}
}
return;
}
else{
fprintf( stdout, " From SIMLIB:%10ld is improper value for SAMPST variable at time
%10.3f\n",
 varibl, simlib.time );
exit(0);
}
```

```
return;
} /* end of function */

void /*FUNCTION*/ timest(value, varibl)
double value;
long int varibl;
{
static long int ivar, ivar_;
static float area[45], max_[45], min_[45], preval[45], tlvc[45], treset;

/* *** If the variable value is improper, stop the simulation. */
if( varibl >= -45L && varibl <= 45L ){

/* *** Execute the desired option. */

if( varibl < 0L ){

/* ---------------------------------------------------------------------- */

/* *** Report the results. */

ivar = -varibl;
area[ivar - 1L] = area[ivar - 1L] + (simlib.time - tlvc[ivar - 1L])*
 preval[ivar - 1L];
tlvc[ivar - 1L] = simlib.time;
```

```
simlib.trnsfr[0L] = area[ivar - 1L]/(simlib.time - treset);
simlib.trnsfr[1L] = max_[ivar - 1L];
simlib.trnsfr[2L] = min_[ivar - 1L];
}
else if( varibl > 0L ){

/* ------------------------------------------------------------------- */

/* *** Collect data. */

area[varibl - 1L] = area[varibl - 1L] + (simlib.time -
 tlvc[varibl - 1L])*preval[varibl - 1L];
if( value > max_[varibl - 1L] )
max_[varibl - 1L] = value;
if( value < min_[varibl - 1L] )
min_[varibl - 1L] = value;
preval[varibl - 1L] = value;
tlvc[varibl - 1L] = simlib.time;
}
else{

/* ------------------------------------------------------------------- */

/* *** Initialize the routine. */
```

```
for( ivar = 1L; ivar <= 45L; ivar++ ){
ivar_ = ivar - 1;
area[ivar_] = 0.;
max_[ivar_] = -1.e30;
min_[ivar_] = 1.e30;
preval[ivar_] = 0.;
tlvc[ivar_] = simlib.time;
}
treset = simlib.time;
}
return;
}
else{
fprintf( stdout, " From SIMLIB:%10ld is improper value for TIMEST variable at time
%10.3f\n",
 varibl, simlib.time );
exit(0);
}
return;
} /* end of function */


void /*FUNCTION*/ filest(list)
long int list;
{
```

```
static long int ilist;

/* *** Compute summary statistics for the list. */
ilist = -(20L + list);
timest( 0., ilist );
return;
} /* end of function */




double /*FUNCTION*/ expon(rmean, istrm)
double rmean;
long int istrm;
{
static float expon_v, u;

/* *** GENERATE A U(0,1) RANDOM VARIABLE. THE FORM OF THIS
STATEMENT
 * *** DEPENDS UPON THE COMPUTER USED. */
u = rannd( istrm );

/* *** GENERATE AN EXPONENTIAL RANDOM VARIABLE WITH RMEAN. */

expon_v = -rmean*log( u );
return( expon_v );
```

```
} /* end of function */



/* powi.c - calc x^n, where n is an integer! */

/* Very slightly modified version of power() from Computer Language, Sept. 86,
pg 87, by Jon Snader (who extracted the binary algorithm from Donald Knuth,
"The Art of Computer Programming", vol 2, 1969).
powi() will only be called when an exponentiation with an integer
exponent is performed, thus tests & code for fractional exponents were
removed.        -- cw lightfoot, COBALT BLUE
  COBALT BLUE has NO copyright on the functions in this file.  The source may
  be considered public domain.
 */


double powi( x, n )    /* returns:  x^n */
double x;        /* base */
long int n;              /* exponent */
{
double p;        /* holds partial product */

if( x == 0 )
return( 0. );
```

```
if( n < 0 ){    /* test for negative exponent */
n = -n;
x = 1/x;
}

p = IS_ODD(n) ? x : 1;  /* test & set zero power */

while( n >>= 1 ){       /* now do the other powers */
x *= x;                 /* sq previous power of x */
if( IS_ODD(n) ) /* if low order bit set */
p *= x;         /*      then, multiply partial product by latest power of x */
}

return( p );
}

/*          *          *          *          */

long ipow( m, n )       /* returns:  m^n */
long m;         /* base */
long n;         /* exponent */
{
long p; /* holds partial product */
```

```
if( m == 0 )
return( 0L );

if( n < 0 ){    /* test for negative exponent */
n = -n;
m = 1/m;
}

p = IS_ODD(n) ? m : 1;  /* test & set zero power */

while( n >>= 1 ){       /* now do the other powers */
m *= m;                 /* sq previous power of m */
if( IS_ODD(n) ) /* if low order bit set */
p *= m;         /*      then, multiply partial product by latest power of m */
}

return( p );
}

/*          *          *          *          */

#ifdef __STDC__
short int sipow( short m, short n )
#else
short int sipow( m, n ) /* returns:  m^n */
```

```
short int m;          /* base */
short int n;          /* exponent */
#endif
{
int ip; /* integer version of product */
long p; /* holds partial product */

if( m == 0 )
return( 0 );

if( n < 0 ){    /* test for negative exponent */
n = -n;
m = 1/m;
}

p = IS_ODD(n) ? m : 1;  /* test & set zero power */

while( n >>= 1 ){       /* now do the other powers */
m *= m;                 /* sq previous power of m */
if( IS_ODD(n) ) /* if low order bit set */
p *= m;         /*      then, multiply partial product by latest power of m */
}

ip = (short int)p;
if( ip != p  )
```

```
puts( "WARNING: integer truncation in sipow()\n" );

return( ip );
}
```

# Appendix 7

Protype Header File Genetic Search Jobshop – nine.h

```
void arrive(long);
void cancel(double);
void
crossovr(LOGICAL32[],LOGICAL32[],LOGICAL32[],LOGICAL32[],long,long*,long*,
long*,double,double);
double decode(LOGICAL32[],long);
void decode_parms(long,long,LOGICAL32[],void*);
void depart(void);
double erlang(long,double,long);
double expon(double,long);
void extract(LOGICAL32[],LOGICAL32[],long*,long,long);
void file(long,long);
void filest(long);
LOGICAL32 flip(double,long);
void generation(void);
void initdata(void);
void initialize(void);
void initlk(void);
```

```
void initparm(void);
void initpop(void);
void initrepo(void);
long int ipow(long, long);
long irandi(long,float[],long);
LOGICAL32 mutation(LOGICAL32,double,long*);
double objfunc(double,double,double,double,double);
void prgm_(void);
double rannd(long);
void removve(long,long);
void report(void);
void reportt(void);
void sampst(double,long);
long select(long,double,void*);
void sim(double,double,double,double,double,float*);
void statistics(long,float*,float*,float*,float*,void*,long*,
long*,long*,long*,long*,long*,long*,long*,
long*,long*);
void timest(double,long);
void timing(void);
double unifrm(long,long,long);
void wrtchrom(LOGICAL32[],long);
```

Input file genejob.in

5 3 0.125 365.0

4 3 5

3 1 2 5

0.50 0.60 0.85 0.50

4 1 3

1.10 0.80 0.75

2 5 1 4 3

1.20 0.25 0.70 0.90 1.00

0.300 0.800 1.000

File – nine.c

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

#define TRUE 1
#define FALSE 0
typedef long LOGICAL32;

#define IS_ODD(j)       ((j) & 1 )

#include "./nine.h"

struct individual {
LOGICAL32 chrom[30];
float t, u, x, v, z, fitness;
long int parent1, parent2, xsite;
} ;
struct parmparm {
long int lparm;
float parameter;
} ;

struct t_gene {
```

```
struct individual oldpop[100], newpop[100];
long int popsize, lchrom, gen, maxgen;
float pcross, pmutate, sumfit;
long int nmutate, ncross;
float avg, max_, min_;
struct individual population[100];
struct parmparm parms[30];
long int nparms, mint, minu, minx, minv, minz, maxt, maxu, maxx,
 maxv, maxz;
}       gene;
struct t_dat {
long int ncount[100], search;
}       dat;

FILE *fpp;

main( )
{
static long int gen_, i, i_, j, j_, maxsearch, search_;
struct individual newpop[100], oldpop[100], population[100];
struct parmparm parms[30];

if ((fpp = fopen("nine.dat","w"))==NULL)  {
  fprintf(stderr,"cannot open file nine.dat\n");
 exit(1);
```

```
}

for( i = 1L; i <= 100L; i++ ){
i_ = i - 1;
dat.ncount[i_] = 0L;
}



fprintf( stdout, "%s\n", " Input the number of searches ------------> "
  );
fscanf( stdin, "%2ld", &maxsearch );

for( dat.search = 1L; dat.search <= maxsearch; dat.search++ ){
search_ = dat.search - 1;
fputc( '\n', stderr );
fputc( '\n', stderr );
fprintf( stderr, "%s\n", " The current search is ------------------> "
  );
fprintf( stderr, "%2ld\n", dat.search );
fputc( '\n', stderr );
fputc( '\n', stderr );

initialize();

for( gene.gen = 1L; gene.gen <= gene.maxgen; gene.gen++ ){
```

```
gen_ = gene.gen - 1;

fprintf( stderr, "%s\n", " The current generation is     ------------> "
  );
fprintf( stderr, "%2ld\n", gene.gen );
generation();
statistics( gene.popsize, &gene.max_, &gene.avg, &gene.min_,
 &gene.sumfit, gene.newpop, &gene.mint, &gene.minu, &gene.minx,
 &gene.minv, &gene.minz, &gene.maxt, &gene.maxu, &gene.maxx,
 &gene.maxv, &gene.maxz );
report();

/* ADVANCE THE GENERATION */

for( j = 1L; j <= 100L; j++ ){
j_ = j - 1;
gene.oldpop[j_] = gene.newpop[j_];
}

}
}

for( i = 1L; i <= 100L; i++ ){
i_ = i - 1;
if( dat.ncount[i_] != 0L )
```

```
{
fprintf( stdout, "The Count for %2ld  is %2ld \n", i, dat.ncount[i_] );
}


  fclose(fpp);}
return(0);
} /* end of function */


double /*FUNCTION*/ decode(chrom, lbits)
LOGICAL32 chrom[];
long int lbits;
{
static long int j, j_;
static float accum, decode_v, powerof2;


accum = 0L;
powerof2 = 1L;


for( j = 1L; j <= lbits; j++ ){
j_ = j - 1;
if( chrom[j_] )
accum = accum + powerof2;
powerof2 = powerof2*2L;
}
```

```
decode_v = accum;
if( decode_v == 0L )
decode_v = 1L;

return( decode_v );
} /* end of function */

void /*FUNCTION*/ generation()
{
static long int j, j_, jcross, mate1, mate2;
struct individual newpop[100], oldpop[100], population[100];
struct parmparm parms[30];

/*     CREATE A NEW GENERATION THROUGH SELECT, CROSSOVER, AND
MUTATION
 *    GENERATION ASSUMES AN EVEN NUMBERED POPSIZE */
for( j = 1L; j <= gene.popsize; j += 2L ){
j_ = j - 1;

mate1 = select( gene.popsize, gene.sumfit, gene.oldpop );
mate2 = select( gene.popsize, gene.sumfit, gene.oldpop );

crossovr( gene.oldpop[mate1 - 1L].chrom, gene.oldpop[mate2 - 1L].chrom,
 gene.newpop[j_].chrom, gene.newpop[j_ + 1L].chrom, gene.lchrom,
 &gene.ncross, &gene.nmutate, &jcross, gene.pcross, gene.pmutate );
```

```
/* CROSSOVER AND MUTATION - MUTATION EMBEDDED WITHIN
CROSSOVER */

/* DECODE STRING, EVALUATE FITNESS, & RECORD PARENTAGE DATA
ON BOTH
 * CHILDREN */


decode_parms( gene.nparms, gene.lchrom, gene.newpop[j_].chrom,
 gene.parms );
gene.newpop[j_].t = gene.parms[0L].parameter;
gene.newpop[j_].u = gene.parms[1L].parameter;
gene.newpop[j_].x = gene.parms[2L].parameter;
gene.newpop[j_].v = gene.parms[3L].parameter;
gene.newpop[j_].z = gene.parms[4L].parameter;
gene.newpop[j_].fitness = objfunc( gene.newpop[j_].t, gene.newpop[j_].u,
 gene.newpop[j_].x, gene.newpop[j_].v, gene.newpop[j_].z );
gene.newpop[j_].parent1 = mate1;
gene.newpop[j_].parent2 = mate2;
gene.newpop[j_].xsite = jcross;

decode_parms( gene.nparms, gene.lchrom, gene.newpop[j_ + 1L].chrom,
 gene.parms );
gene.newpop[j_ + 1L].t = gene.parms[0L].parameter;
```

```
gene.newpop[j_ + 1L].u = gene.parms[1L].parameter;
gene.newpop[j_ + 1L].x = gene.parms[2L].parameter;
gene.newpop[j_ + 1L].v = gene.parms[3L].parameter;
gene.newpop[j_ + 1L].z = gene.parms[4L].parameter;
gene.newpop[j_ + 1L].fitness = objfunc( gene.newpop[j_ + 1L].t,
 gene.newpop[j_ + 1L].u, gene.newpop[j_ + 1L].x, gene.newpop[j_ + 1L].v,
 gene.newpop[j_ + 1L].z );
gene.newpop[j_ + 1L].parent1 = mate1;
gene.newpop[j_ + 1L].parent2 = mate2;
gene.newpop[j_ + 1L].xsite = jcross;


}
return;
} /* end of function */

long /*FUNCTION*/ select(popsize, sumfit, vp_population)
long int popsize;
double sumfit;
void *vp_population;
{
struct individual *population = vp_population;
static long int j, j_, select_v;
static float partsum, random;
```

```
/* SELECT A SINGLE INDIVIDUAL VIA ROULETTE WHEEL SELECTION */
partsum = 0L;
j = 0L;



random = rannd( 1L )*sumfit;

/* FIND WHEEL  SLOT */

for( j = 1L; j <= popsize; j++ ){
j_ = j - 1;

partsum = partsum + population[j_].fitness;

if( (partsum >= random) || (j == popsize) )
goto L_10;

}

/* RETURN INDIVIDUAL MEMBER */

L_10:
select_v = j;

return( select_v );
```

```
} /* end of function */

LOGICAL32 /*FUNCTION*/ mutation(allelev, pmutate, nmutation)
LOGICAL32 allelev;
double pmutate;
long int *nmutation;
{
static LOGICAL32 mutate, mutation_v;

/* MUATATE AN ALLEL WITH PROBABILITY OF MUTATION, COUNT
NUMBER OF
 * MUTATIONS */
mutate = flip( pmutate, 1L );
if( mutate ){
*nmutation = *nmutation + 1L;
mutation_v = !allelev;
}
else{
mutation_v = allelev;

}
return( mutation_v );
} /* end of function */

void /*FUNCTION*/ crossovr(parnt1, parnt2, child1, child2, lchrom,
```

```
ncross, nmutate, jcross, pcross, pmutate)
LOGICAL32 parnt1[], parnt2[], child1[], child2[];
long int lchrom, *ncross, *nmutate, *jcross;
double pcross, pmutate;
{
static long int j, j_;

if( flip( pcross, 1L ) ){
*jcross = unifrm( 1L, lchrom - 1L, 1L );
*ncross = *ncross + 1L;
}
else{
*jcross = lchrom;
}

/* THE FIRST EXCHANGE 1 TO 1, AND 2 TO 2 */

for( j = 1L; j <= *jcross; j++ ){
j_ = j - 1;
child1[j_] = mutation( parnt1[j_], pmutate, nmutate );
child2[j_] = mutation( parnt2[j_], pmutate, nmutate );
}

/* THE SECOND EXCHANGE 1 TO 2, AND 2 TO 1 */
```

```
if( *jcross != lchrom ){
for( j = *jcross + 1L; j <= lchrom; j++ ){
j_ = j - 1;
child1[j_] = mutation( parnt2[j_], pmutate, nmutate );
child2[j_] = mutation( parnt1[j_], pmutate, nmutate );
}
}
return;
} /* end of function */


void /*FUNCTION*/ statistics(popsize, max_, avg, min_, sumfit, vp_population,
 mint, minu, minx, minv, minz, maxt, maxu, maxx, maxv, maxz)
long int popsize;
float *max_, *avg, *min_, *sumfit;
void  *vp_population;
long int *mint, *minu, *minx, *minv, *minz, *maxt, *maxu, *maxx, *maxv,
 *maxz;
{
struct individual *population = vp_population;
static long int j, j_;


/* INITIALIZE */
*sumfit = population[0L].fitness;
*min_ = population[0L].fitness;
*max_ = population[0L].fitness;
```

```
*maxt = population[0L].t;
*maxu = population[0L].u;
*maxx = population[0L].x;
*maxv = population[0L].v;
*maxz = population[0L].z;
*mint = population[0L].t;
*minu = population[0L].u;
*minx = population[0L].x;
*minv = population[0L].v;
*minz = population[0L].z;

for( j = 2L; j <= popsize; j++ ){
j_ = j - 1;
*sumfit = *sumfit + population[j_].fitness;
if( population[j_].fitness > *max_ ){
*max_ = population[j_].fitness;
*maxt = population[j_].t;
*maxu = population[j_].u;
*maxx = population[j_].x;
*maxv = population[j_].v;
*maxz = population[j_].z;
}

if( population[j_].fitness < *min_ ){
```

```
*min_ = population[j_].fitness;
*mint = population[j_].t;
*minu = population[j_].u;
*minx = population[j_].x;
*minv = population[j_].v;
*minz = population[j_].z;
}
}

*avg = *sumfit/popsize;

return;
} /* end of function */

void /*FUNCTION*/ initialize()
{
struct individual newpop[100], oldpop[100], population[100];
struct parmparm parms[30];
gene.ncross = 0L;
gene.nmutate = 0L;

/* INITIALIZATION COORDINATOR */
if( dat.search == 1L ){
initdata();
initparm();
```

```
}
initpop();
statistics( gene.popsize, &gene.max_, &gene.avg, &gene.min_, &gene.sumfit,
 gene.oldpop, &gene.mint, &gene.minu, &gene.minx, &gene.minv,
 &gene.minz, &gene.maxt, &gene.maxu, &gene.maxx, &gene.maxv, &gene.maxz );
initrepo();
return;
} /* end of function */




struct t_param {
long int nreps;
float highval;
}       param;


void /*FUNCTION*/ initdata()
{
struct individual newpop[100], oldpop[100], population[100];
struct parmparm parms[30];


fprintf( stdout, "%s\n", " ENTER POPULATION SIZE        ------------> "
  );
fscanf( stdin, "%2ld", &gene.popsize );
fprintf( stdout, "%s\n", " ENTER CHROMOSOME LENGTH       -----------> "
```

```
  );
fscanf( stdin, "%2ld", &gene.lchrom );
fprintf( stdout, "%s\n", " ENTER MAX. GENERATIONS      ------------> "
  );
fscanf( stdin, "%2ld", &gene.maxgen );
fprintf( stdout, "%s\n", " ENTER CROSSOVER PROBABILITY  ------------> "
  );
fscanf( stdin, "%f", &gene.pcross );
fprintf( stdout, "%s\n", " ENTER MUTATION PROBABILITY   ------------> "
  );
fscanf( stdin, "%f", &gene.pmutate );
fprintf( stdout, "%s\n", " ENTER NO. OF REPLICATIONS    ------------> "
  );
fscanf( stdin, "%2ld", &param.nreps );



return;
} /* end of function */

void /*FUNCTION*/ initpop()
{
static long int j, j1, j1_, j_;
struct individual newpop[100], oldpop[100], population[100];
struct parmparm parms[30];
```

```
/* INITIALIZE A POPULATION AT RANDOM */
for( j = 1L; j <= gene.popsize; j++ ){
j_ = j - 1;
for( j1 = 1L; j1 <= gene.lchrom; j1++ ){
j1_ = j1 - 1;

/* TOSS OF A FAIR COIN */

gene.oldpop[j_].chrom[j1_] = flip( 0.50, 1L );

}

/* NOW DECODE THE STRING */

decode_parms( gene.nparms, gene.lchrom, gene.oldpop[j_].chrom,
 gene.parms );
gene.oldpop[j_].t = gene.parms[0L].parameter;
gene.oldpop[j_].u = gene.parms[1L].parameter;
gene.oldpop[j_].x = gene.parms[2L].parameter;
gene.oldpop[j_].v = gene.parms[3L].parameter;
gene.oldpop[j_].z = gene.parms[4L].parameter;
gene.oldpop[j_].fitness = objfunc( gene.oldpop[j_].t, gene.oldpop[j_].u,
 gene.oldpop[j_].x, gene.oldpop[j_].v, gene.oldpop[j_].z );
gene.oldpop[j_].parent1 = 0L;
gene.oldpop[j_].parent2 = 0L;
```

```
gene.oldpop[j_].xsite = 0L;

}
return;
} /* end of function */


void /*FUNCTION*/ initrepo()
{
struct individual newpop[100], oldpop[100], population[100];
struct parmparm parms[30];

/* INITIAL REPORT */
fprintf( stdout, " ---------------------------------------------\n" );
fprintf( stdout, " | A SIMPLE GENETIC ALGORITHM - SGA v1.0       |\n" );
fprintf( stdout, " | (c) James M. Yunker 1991                    |\n" );
fprintf( stdout, " |      All Rights Reserved                    |\n" );
fprintf( stdout, " ---------------------------------------------\n" );
fputc( '\n', stdout );
fputc( '\n', stdout );
fputc( '\n', stdout );
fputc( '\n', stdout );
fputc( '\n', stdout );
fputc( '\n', stdout );
fprintf( stdout, "         SGA Parameters  |\n" );
```

```
fputc( '\n', stdout );
fprintf( stdout, "   Population size (popsize)          = %2ld\n",
 gene.popsize );
fprintf( stdout, "   Chromosome length (lchrom)          = %2ld\n",
 gene.lchrom );
fprintf( stdout, "   Maximum # of generations            = %2ld\n",
 gene.maxgen );
fprintf( stdout, "   Crossover probability (pmutation)   = %3.2f\n",
 gene.pcross );
fprintf( stdout, "   Mutation probability                = %4.3f\n",
 gene.pmutate );
fprintf( stdout, "   Number of replications              = %2ld\n",
 param.nreps );
fprintf( stdout, "   The highval is                      = %6.1f\n",
 param.highval );
fprintf( stdout, "   The search is                       = %2ld\n",
 dat.search );
fputc( '\n', stdout );
fputc( '\n', stdout );
fputc( '\n', stdout );
fputc( '\n', stdout );
fputc( '\n', stdout );
fputc( '\n', stdout );
fputc( '\n', stdout );
fputc( '\n', stdout );
```

```
fputc( '\n', stdout );
fputc( '\n', stdout );
fprintf( stdout, "  Initial population maximum fitness  = %10.2f\n",
 gene.max_ );
fprintf( stdout, "  Initial population average fitness  = %10.2f\n",
 gene.avg );
fprintf( stdout, "  Initial population minimum fitness  = %10.2f\n",
 gene.min_ );
fprintf( stdout, "  Initial population sum of fitnesses = %10.2f\n",
 gene.sumfit );
fputc( '\n', stdout );
return;
} /* end of function */



LOGICAL32 /*FUNCTION*/ flip(prob, istrm)
double prob;
long int istrm;
{
static LOGICAL32 flip_v;
static float u;

u = rannd( istrm );
```

```
flip_v = u <= prob;

return( flip_v );
} /* end of function */



double /*FUNCTION*/ objfunc(t, u, x, v, z)
double t, u, x, v, z;
{
static long int i, i_;
static float objfunc_v, sum, y, yavg;

param.highval = 5000000L;
sum = 0L;

for( i = 1L; i <= param.nreps; i++ ){
i_ = i - 1;

sim( t, u, x, v, z, &y );

sum = sum + y;

}

yavg = sum/param.nreps;
```

```
objfunc_v = param.highval - yavg;

return( objfunc_v );
} /* end of function */


void /*FUNCTION*/ wrtchrom(chrom, lchrom)
LOGICAL32 chrom[];
long int lchrom;
{
static long int i, i_;


for( i = lchrom; i >= 1L; i-- ){
i_ = i - 1;
if( chrom[i_] ){
fprintf(stdout, "%s", "1");
}
else{
fprintf(stdout, "%s", "0");
}
}
return;
} /* end of function */
```

```
void /*FUNCTION*/ report()
{
static long int j, j_, m1, m10, m2, m3, m4, m5, m6, m7, m8, m9;
static float adjmax, adjmin;
struct individual newpop[100], oldpop[100], population[100];
struct parmparm parms[30];

if( gene.gen == 1L || gene.gen == gene.maxgen ){
fprintf( stdout, " ---------------------------------------------------\n" );
fputc( '\n', stdout );
fprintf( stdout, "            Population Report\n" );
fputc( '\n', stdout );
fprintf( stdout, "            Generation %2ld\n", gene.gen -
 1L );
fputc( '\n', stdout );
fprintf( stdout, "         string          1  2  3  4  5  fitness\n" );


for( j = 1L; j <= gene.popsize; j++ ){
j_ = j - 1;
wrtchrom( gene.oldpop[j_].chrom, gene.lchrom );
m1 = gene.oldpop[j_].t;
```

```
m2 = gene.oldpop[j_].u;
m3 = gene.oldpop[j_].x;
m4 = gene.oldpop[j_].v;
m5 = gene.oldpop[j_].z;
fprintf( stdout, "  %2ld  %2ld  %2ld  %2ld  %2ld   %10.2f \n", m1,
 m2, m3, m4, m5, gene.oldpop[j_].fitness );
}

/* New String */

fputc( '\n', stdout );
fputc( '\n', stdout );
fprintf( stdout, "                          Population Report\n" );
fputc( '\n', stdout );
fprintf( stdout, "                          Generation %2ld\n",
 gene.gen );
fputc( '\n', stdout );
fprintf( stdout, "         string          1  2  3  4  5  fitness   parents site\n" );

for( j = 1L; j <= gene.popsize; j++ ){
j_ = j - 1;
wrtchrom( gene.newpop[j_].chrom, gene.lchrom );
m6 = gene.newpop[j_].t;
m7 = gene.newpop[j_].u;
m8 = gene.newpop[j_].x;
```

```
m9 = gene.newpop[j_].v;
m10 = gene.newpop[j_].z;
fprintf( stdout, "  %2ld  %2ld  %2ld  %2ld  %2ld   %10.2f  %2ld %2ld   %2ld \n",
 m6, m7, m8, m9, m10, gene.newpop[j_].fitness, gene.newpop[j_].parent1,
 gene.newpop[j_].parent2, gene.newpop[j_].xsite );
}

fputc( '\n', stdout );
fputc( '\n', stdout );
fputc( '\n', stdout );
}
else{
fputc( '\n', stdout );
fprintf( stdout, "        ************************ \n" );
fprintf( stdout, "  The intervening chromosomes left out for space reasons\n" );
fprintf( stdout, "        ************************ \n" );
fputc( '\n', stdout );
}

/* Generation statistics and accumulated values */
fprintf( stdout, "  Note: Generation %2ld and accumulated statistics:\n",
 gene.gen );
fprintf( stdout, "  max= %10.2f min= %10.2f avg= %10.2f sum= %10.2f\n",
 gene.max_, gene.min_, gene.avg, gene.sumfit );
fprintf( stdout, "  nmutation= %2ld ncross= %2ld\n", gene.nmutate,
```

```
gene.ncross );
fputc( '\n', stdout );
fputc( '\n', stdout );


adjmax = param.highval - gene.min_;
fprintf( stdout, "  The maximum overall average cost is %10.2f\n",
 adjmax );
fprintf( stdout, "  The machines at 1 for maximum cost is %2ld\n",
 gene.mint );
fprintf( stdout, "  The machines at 2 for maximum cost is %2ld\n",
 gene.minu );
fprintf( stdout, "  The machines at 3 for maximum cost is %2ld\n",
 gene.minx );
fprintf( stdout, "  The machines at 4 for maximum cost is %2ld\n",
 gene.minv );
fprintf( stdout, "  The machines at 5 for maximum cost is %2ld\n",
 gene.minz );
fputc( '\n', stdout );
fputc( '\n', stdout );


adjmin = param.highval - gene.max_;
fprintf( stdout, "  The minimum overall average cost is %10.2f\n",
 adjmin );
```

```
fprintf( stdout, " The machines at 1 for minimum cost is %2ld\n",
 gene.maxt );
fprintf( stdout, " The machines at 2 for minimum cost is %2ld\n",
 gene.maxu );
fprintf( stdout, " The machines at 3 for minimum cost is %2ld\n",
 gene.maxx );
fprintf( stdout, " The machines at 4 for minimum cost is %2ld\n",
 gene.maxv );
fprintf( stdout, " The machines at 5 for minimum cost is %2ld\n",
 gene.maxz );
fprintf(fpp, "%2ld %2ld %2ld %2ld %2ld %2ld %10.2f\n", gene.maxt, gene.maxu,
gene.maxx, gene.maxv, gene.maxz, gene.gen, adjmin);
fputc( '\n', stdout );
fputc( '\n', stdout );


return;
} /* end of function */




void /*FUNCTION*/ initparm()
{
static long int i, i_, lparm;
struct individual newpop[100], oldpop[100], population[100];
```

```
struct parmparm parms[30];

fputc( '\n', stdout );
fputc( '\n', stdout );
fprintf( stdout, "The following is a model  is for a job-shop that  \n" );
fprintf( stdout, "has 5 stations each using the same type of machine, \n" );
fprintf( stdout, "the five parameters are the number of machines at \n" );
fprintf( stdout, "each of the five stations. Keep this in mind when \n" );
fprintf( stdout, "answering the following questions\n" );
fputc( '\n', stdout );
fputc( '\n', stdout );
fprintf( stdout, "%s\n", " Input the number of parameters ------------> "
  );
fscanf( stdin, "%2ld", &gene.nparms );

fputc( '\n', stdout );
fputc( '\n', stdout );
for( i = 1L; i <= gene.nparms; i++ ){
i_ = i - 1;
fprintf( stdout, "FOR PARAMETER %2ld \n", i );
fputc( '\n', stdout );
fprintf( stdout, "%s\n", " Enter the length              ------------> "
  );
fscanf( stdin, "%2ld", &lparm );
gene.parms[i_].lparm = lparm;
```

```
fputc( '\n', stdout );
fputc( '\n', stdout );
}

 freopen("OUTPUT9","w",stdout);
 freopen("OUT9","w",stderr);


for( i = 1L; i <= gene.nparms; i++ ){
i_ = i - 1;
fprintf( stdout, "FOR PARAMETER %2ld \n", i );
fputc( '\n', stdout );
fprintf( stdout, " the length ----------------> %2ld \n",
 gene.parms[i_].lparm );
fputc( '\n', stdout );
fputc( '\n', stdout );
}
fputc( '\n', stdout );
return;
} /* end of function */


void /*FUNCTION*/ extract(chromfrom, chromato, jposition, lchrom,
 lparm)
LOGICAL32 chromfrom[], chromato[];
```

```
long int *jposition, lchrom, lparm;
{
static long int j, jtarget;

j = 1L;
jtarget = *jposition + lparm - 1L;
if( jtarget > lchrom )
jtarget = lchrom;
while( *jposition <= jtarget ){
chromato[j - 1L] = chromfrom[*jposition - 1L];
*jposition = *jposition + 1L;
j = j + 1L;
}
return;
} /* end of function */


void /*FUNCTION*/ decode_parms(nparms, lchrom, chrom, vp_parms)
long int nparms, lchrom;
LOGICAL32 chrom[];
void  *vp_parms;
{
struct parmparm *parms = vp_parms;
static LOGICAL32 chromtemp[30];
static long int j, jposition;
```

```
static float parameter, scale;

jposition = 1L;
j = 1L;
while( j <= nparms ){
scale = ipow(2L, parms[j - 1L].lparm) - 1L;
if( parms[j - 1L].lparm > 0L ){
extract( chrom, chromtemp, &jposition, lchrom, parms[j - 1L].lparm );
parms[j - 1L].parameter = decode( chromtemp, parms[j - 1L].lparm );
}
else{
parameter = 0L;
}
j = j + 1L;
}
return;
} /* end of function */


double /*FUNCTION*/ unifrm(a, b, istrm)
long int a, b, istrm;
{
static float u, unifrm_v;

/* *** GENERATE A U(0,1) RANDOM VARIABLE. THE FORM OF THIS
```

```
STATEMENT
 * *** DEPENDS UPON THE COMPUTER USED. */
u = rannd( istrm );

/* *** GENERATE A U(A,B) RANDOM VARIABLE. */

unifrm_v = a + (u*(b - a));
return( unifrm_v );
} /* end of function */


struct t_model {
long int jobtyp;
float length, marrvt, mservt[5][5];
long int nbusy[5], ngroup, nmachs[5], ntasks[5], ntypes;
float probd[3];
long int route[5][5], task;
float cost;
}       model;
struct t_simlib {
long int ldecr, levent, lfirst, lincr, llast, lrank[25], lsize[25],
 maxatr, next;
float time, trnsfr[10];
}       simlib;
```

```
void /*FUNCTION*/ sim(t, u, x, v, z, y)
double t, u, x, v, z;
float *y;
{
static LOGICAL32 skip;
static long int i, i_, j, ntsks;
FILE *fp;

/*     MAIN SIMULATION PROGRAM */
model.nmachs[0L] = t;
model.nmachs[1L] = u;
model.nmachs[2L] = x;
model.nmachs[3L] = v;
model.nmachs[4L] = z;

if( !skip ){

/*     Open input and output files. */



if ((fp = fopen("genejob.in","r"))==NULL) {
  fprintf(stderr, "cannot open file genejob.in\n");
  exit(1);
}
```

```
/*    READ THE INPUT PARAMETERS */

fscanf( fp, "%ld %ld %f %f", &model.ngroup, &model.ntypes,
 &model.marrvt, &model.length );
fscanf( fp, "" );
for( i = 1L; i <= model.ntypes; i++ ){
fscanf( fp, "%ld", &model.ntasks[i - 1L] );
}
for( i = 1L; i <= model.ntypes; i++ ){
i_ = i - 1;
ntsks = model.ntasks[i_];
fscanf( fp, "" );
for( j = 1L; j <= ntsks; j++ ){
fscanf( fp, "%ld", &model.route[j - 1L][i_] );
}
fscanf( fp, "" );
for( j = 1L; j <= ntsks; j++ ){
fscanf( fp, "%f", &model.mservt[j - 1L][i_] );
}
}
fscanf( fp, "" );
for( i = 1L; i <= model.ntypes; i++ ){
fscanf( fp, "%f", &model.probd[i - 1L] );
}
```

```
/*	Write report heading and input parameters */

fprintf( stdout, "\n Number of job types%25ld", model.ntypes );
fprintf( stdout, "\n\n Number of tasks for each job type      " );
for( i = 1L; i <= model.ntypes; i++ ){
fprintf( stdout, "%5ld", model.ntasks[i - 1L] );
}
fprintf( stdout, "\n" );
fprintf( stdout, "\n Distribution function of job types  " );
for( i = 1L; i <= model.ntypes; i++ ){
fprintf( stdout, "%8.3f", model.probd[i - 1L] );
}
fprintf( stdout, "\n" );
fprintf( stdout, "\n Mean interarrival time of jobs%14.3f hours\n\n Length of the
simulation%20.1f eight-hour days\n\n\n Job Type     Machine groups on route\n",
 model.marrvt, model.length );
for( i = 1L; i <= model.ntypes; i++ ){
i_ = i - 1;
fprintf( stdout, "\n%5ld", i );
fprintf( stdout, "        " );
for( j = 1L; j <= model.ntasks[i_]; j++ ){
fprintf( stdout, "%5ld", model.route[j - 1L][i_] );
}
fprintf( stdout, "\n" );
}
```

```
fprintf( stdout, "\n\n Job Type     Mean service time (in hours) for successive tasks\n"
);
for( i = 1L; i <= model.ntypes; i++ ){
i_ = i - 1;
fprintf( stdout, "\n%5ld", i );
fprintf( stdout, "     " );
for( j = 1L; j <= model.ntasks[i_]; j++ ){
fprintf( stdout, "%8.2f", model.mservt[j - 1L][i_] );
}
fprintf( stdout, "\n" );
}
fputc( '\n', stdout );
fputc( '\n', stdout );
fputc( '\n', stdout );
fputc( '\n', stdout );
skip = TRUE;
fclose(fp);
}

/*    Initialize all machines in all groups to the idle state */

for( i = 1L; i <= model.ngroup; i++ ){
i_ = i - 1;
model.nbusy[i_] = 0L;
}
```

```
/*     Initialize SIMLIB */

initlk();

/*     Set the maximum number of attributes per record */

simlib.maxatr = 4L;

/* *** Schedule the arrival of the first job. */

simlib.trnsfr[0L] = expon( model.marrvt, 1L );
simlib.trnsfr[1L] = 1.;
file( simlib.lincr, simlib.levent );

/* *** Schedule the end of the warm-up period. */

simlib.trnsfr[0L] = 8.*5L;
simlib.trnsfr[1L] = 3.;
file( simlib.lincr, simlib.levent );

/*     Schedule the end of the simulation */

simlib.trnsfr[0L] = 8.*model.length;
simlib.trnsfr[1L] = 4.;
```

```
file( simlib.lincr, simlib.levent );

/*      Determine the next event routine */

while( TRUE ){
timing();

/*      CALL THE APPROPRIATE EVENT ROUTINE */

if( simlib.next == 2L ){
depart();
}
else if( simlib.next == 3L ){
sampst( 0., 0L );
timest( 0., 0L );
}
else if( simlib.next == 4L ){
goto L_10;
}
else{
arrive( 1L );
}
}
L_10:
reportt();
```

```
*y = model.cost;
return;
} /* end of function */

long /*FUNCTION*/ irandi(nvalue, probd, istrm)
long int nvalue;
float probd[];
long int istrm;
{
static long int i, i_, irandi_v;
static float u;

/*     Generate a U(0,1) random variable from stream ISTREAM */
u = rannd( istrm );

/*     Generate a random integer between 1 and NVALUE in accordance with
 *     the cumulative distribution funvtion PROBD. */

for( i = 1L; i <= (nvalue - 1L); i++ ){
i_ = i - 1;
if( u < probd[i_] )
goto L_10;
}
irandi_v = nvalue;
```

```
return( irandi_v );
L_10:
irandi_v = i;
return( irandi_v );
} /* end of function */

void /*FUNCTION*/ arrive(new)
long int new;
{
static long int group, index;
static float delay;

/* *** If this is a new arrival to the shop, generate the time of the
 * *** next arrival and determine the job type and task number of the
 * *** arriving job */
if( new == 1L ){
simlib.trnsfr[0L] = simlib.time + expon( model.marrvt, 1L );
simlib.trnsfr[1L] = 1.;
file( simlib.lincr, simlib.levent );
model.jobtyp = irandi( model.ntypes, model.probd, 1L );
model.task = 1L;
}

/*    Determine a machine group from the route matrix. */
```

```
group = model.route[model.task - 1L][model.jobtyp - 1L];

/*    Check to see whether all machines in this group are busy. */

if( model.nbusy[group - 1L] == model.nmachs[group - 1L] ){

/*    All machines in this group are busy, so place the arriving job
 *    at the end of the appropriate queue. Note that the following
 *    data are stored for each job
 *        1. Time of arrival to this machine group.
 *        2. job type.
 *        3  current task number */

simlib.trnsfr[0L] = simlib.time;
simlib.trnsfr[1L] = model.jobtyp;
simlib.trnsfr[2L] = model.task;
file( simlib.llast, group );

}
else{

/*    A machine in this group is idle, so start service on the
 *    arriving job (which has a delay of zero). */
```

```
delay = 0L;
sampst( delay, group );
index = model.ngroup + model.jobtyp;
sampst( delay, index );
model.nbusy[group - 1L] = model.nbusy[group - 1L] + 1L;
timest( (float)( model.nbusy[group - 1L] ), group );

/*     Schedule a service completion */

simlib.trnsfr[0L]   =   simlib.time   +   erlang(   2L,   model.mservt[model.task   -
1L][model.jobtyp - 1L],
 1L );
simlib.trnsfr[1L] = 2.0;
simlib.trnsfr[2L] = model.jobtyp;
simlib.trnsfr[3L] = model.task;
file( simlib.lincr, simlib.levent );

}
return;
} /* end of function */

void /*FUNCTION*/ depart()
{
static long int group, index, jobtq, taskq;
static float delay;
```

```
/*     Determine the machine group from which the job is departing */
model.jobtyp = simlib.trnsfr[2L];
model.task = simlib.trnsfr[3L];
group = model.route[model.task - 1L][model.jobtyp - 1L];

/*     Check to see whether the queue for this machine group is empty. */

if( simlib.lsize[group - 1L] == 0L ){

/*      The queue for this machine is empty so make a machine in
 *      this group idle. */

model.nbusy[group - 1L] = model.nbusy[group - 1L] - 1L;
timest( (float)( model.nbusy[group - 1L] ), group );

}
else{

/*     The queue is not empty, so start service on the first job in queue */

removve( 1L, group );
delay = simlib.time - simlib.trnsfr[0L];

/*     Tally this delay for this machine group. */
```

```
sampst( delay, group );

jobtq = simlib.trnsfr[1L];
taskq = simlib.trnsfr[2L];
index = model.ngroup + jobtq;
sampst( delay, index );

/*     Schedule end of service for this job at this machine group */

simlib.trnsfr[0L] = simlib.time + erlang( 2L, model.mservt[taskq - 1L][jobtq - 1L],
 1L );
simlib.trnsfr[1L] = 2.;
simlib.trnsfr[2L] = jobtq;
simlib.trnsfr[3L] = taskq;
file( simlib.lincr, simlib.levent );

}

/* *** If the current departing job has one or more tasks yet to be done,
 * *** send the job to the next machine group on its route. */

if( model.task < model.ntasks[model.jobtyp - 1L] ){
model.task = model.task + 1L;
arrive( 2L );
```

```
}
return;
} /* end of function */



struct t_llists {
long int head[25], iout, linkpr[100000], linksr[100000];
float master[10][100000];
long int nar, tail[25];
}       llists;

void /*FUNCTION*/ initlk()
{
static long int i, i_, list, list_, row, row_;

/* *** Initialize links. */
for( row = 1L; row <= 100000L; row++ ){
row_ = row - 1;
llists.linkpr[row_] = 0L;
llists.linksr[row_] = row + 1L;
}
llists.linksr[99999L] = 0L;

/* *** Initialize list attributes. */
```

```
for( list = 1L; list <= 25L; list++ ){
list_ = list - 1;
llists.head[list_] = 0L;
llists.tail[list_] = 0L;
simlib.lsize[list_] = 0L;
simlib.lrank[list_] = 0L;
}

/* *** Initialize the TRNSFR */

for( i = 1L; i <= 10L; i++ ){
i_ = i - 1;
simlib.trnsfr[i_] = 0.0;
}

/* *** Initialize mnemonics for record location in lists. */

simlib.lfirst = 1L;
simlib.llast = 2L;
simlib.lincr = 3L;
simlib.ldecr = 4L;

/* *** Initialize mnemonic for event list number. */
```

```
simlib.levent = 25L;

/* *** Initialize system attributes. */

simlib.time = 0.;
llists.nar = 1L;
simlib.lrank[simlib.levent - 1L] = 1L;
simlib.maxatr = 10L;

/* *** Initialize statistical routines. */

sampst( 0., 0L );
timest( 0., 0L );

/* *** Initialize output unit number for SIMLIB error messages. */

llists.iout = 6L;
return;
} /* end of function */

void /*FUNCTION*/ file(option, list)
long int option, list;
{
static long int ahead, behind, ihead, itail, item, item_, row;
static float size;
```

```
/* *** IF THE MASTER STORAGE ARRAY IS FULL, STOP THE SIMULATION.
*/
if( llists.nar <= 0L ){
fprintf( stdout, "         MASTER STORAGE ARRAY OVERFLOW AT TIME
%10.3e\n",
 simlib.time );

/* *** IF LIST VALUE IS IMPROPER, STOP THE SIMULATION. */

}
else if( (list >= 1L) && (list <= 25L) ){

/* *** INCREMENT THE LIST SIZE. */

simlib.lsize[list - 1L] = simlib.lsize[list - 1L] + 1L;

/* *** IF THE OPTION VALUE IS IMPROPER, STOP THE SIMULATION. */

if( (option >= 1L) && (option <= 4L) ){

/* *** FILE ACCORDING TO THE DESIRED OPTION. */

if( option != 1L ){
if( option != 2L ){
```

```
/* ****************************************************************** */

/* *** THE LIST IS RANKED, DETERMINE THE ITEM ON WHICH THE LIST IS
TO
 * *** BE RANKED. */

item = simlib.lrank[list - 1L];

/* *** IF AN INVALID ITEM HAS BEEN SPECIFIED, STOP THE SIMULATION.
*/

if( !((item >= 1L) && (item <= simlib.maxatr))
  ){
fprintf( stdout, "%10ld IS AN IMPROPER VALUE FOR THE RANK OF LIST
%12ld\n",
 item, list );
exit(0);
}
else if( simlib.lsize[list - 1L] == 1L ){
goto L_40;
}
else{

/* *** SEARCH THE LIST FOR THE PROPER LOCATION. */
```

```
row = llists.head[list - 1L];
while( TRUE ){
if( option == 4L ){

/*     RANK THE LIST IN DECREASING ORDER. */

if( simlib.trnsfr[item - 1L] > llists.master[item - 1L][row - 1L] )
goto L_10;

/* *** RANK THE LIST IN INCREASING ORDER. */

}
else if( simlib.trnsfr[item - 1L] < llists.master[item - 1L][row - 1L] ){

/* *** THE CORRECT LOCATION HAS BEEN FOUND. */

goto L_10;
}

/* *** CONTINUE SEARCHING, CONSIDER THE NEXT ROW. */

behind = row;
row = llists.linksr[behind - 1L];
```

```
/* *** IF THE LAST ROW CONSIDERED WAS NOT THE TAIL OF THE LIST,
 * *** CONTINUE. */

if( llists.tail[list - 1L] == behind )
goto L_20;
}

/* *** INSERT BEFORE LAST RECORD EXAMINED. */

L_10:
if( row == llists.head[list - 1L] )
goto L_30;

/* *** INSERT IN THE PROPER LOCATION BETWEEN THE PRECEEDING
AND
 * *** SUCCEEDING RECORDS (BEHIND AND AHEAD). */

ahead = llists.linksr[behind - 1L];
row = llists.nar;
llists.nar = llists.linksr[row - 1L];
if( llists.nar > 0L )
llists.linkpr[llists.nar - 1L] = 0L;
llists.linkpr[row - 1L] = behind;
llists.linksr[behind - 1L] = row;
llists.linkpr[ahead - 1L] = row;
```

```
llists.linksr[row - 1L] = ahead;

/* *** GO TO TRANSFER THE DATA. */

goto L_50;
}
}

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

/* *** INSERT AFTER THE LAST RECORD IN THE LIST. */

L_20:
if( simlib.lsize[list - 1L] == 1L )
goto L_40;
row = llists.nar;
llists.nar = llists.linksr[row - 1L];
if( llists.nar > 0L )
llists.linkpr[llists.nar - 1L] = 0L;
itail = llists.tail[list - 1L];
llists.linkpr[row - 1L] = itail;
llists.linksr[itail - 1L] = row;
llists.linksr[row - 1L] = 0L;
llists.tail[list - 1L] = row;
```

```
/* *** GO TO TRANSFER THE DATA. */

goto L_50;
}

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

/* *** INSERT BEFORE THE FIRST RECORD IN THE LIST. */

L_30:
if( simlib.lsize[list - 1L] != 1L ){
row = llists.nar;
llists.nar = llists.linksr[row - 1L];
if( llists.nar > 0L )
llists.linkpr[llists.nar - 1L] = 0L;
ihead = llists.head[list - 1L];
llists.linkpr[ihead - 1L] = row;
llists.linksr[row - 1L] = ihead;
llists.linkpr[row - 1L] = 0L;
llists.head[list - 1L] = row;

/* *** GO TO TRANSFER THE DATA. */

goto L_50;
}
```

```
/* ******************************************************************** */

/* *** INSERT THE FIRST RECORD IN THE LIST. */

L_40:
row = llists.nar;
llists.nar = llists.linksr[row - 1L];
if( llists.nar > 0L )
llists.linkpr[llists.nar - 1L] = 0L;
llists.linksr[row - 1L] = 0L;
llists.head[list - 1L] = row;
llists.tail[list - 1L] = row;

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

/* *** TRANSFER THE DATA. */

L_50:
;
for( item = 1L; item <= simlib.maxatr; item++ ){
item_ = item - 1;
llists.master[item_][row - 1L] = simlib.trnsfr[item_];
}
```

```
/* *** UPDATE THE AREA UNDER THE NUMBER IN LIST CURVE. */

size = simlib.lsize[list - 1L];
timest( size, 20L + list );
return;
}
else{
fprintf( stdout, "%10ldIS AN IMPROPER VALUE FOR FILE OPTION AT
TIME%10.3e\n",
 option, simlib.time );
}
}
else{
fprintf( stdout, "%10ld IS AN IMPROPER VALUE FOR FILE AT TIME %10.3e\n",
 list, simlib.time );
}
exit(0);
} /* end of function */

void /*FUNCTION*/ timing()
{

/* *** Remove the first event from the event list. */
removve( simlib.lfirst, simlib.levent );
```

```
/* *** Check for a time reversal. */

if( simlib.trnsfr[0L] >= simlib.time ){

/* *** Advance the simulation clock. */

simlib.time = simlib.trnsfr[0L];
simlib.next = simlib.trnsfr[1L];
return;
}
else{
fprintf( stdout, " From SIMLIB: Attempt to schedule an event of type %3.0f\n at time
%10.3f when the clock is %10.3f\n",
 simlib.trnsfr[1L], simlib.trnsfr[0L], simlib.time );
exit(0);
}
return;
} /* end of function */

void /*FUNCTION*/ cancel(etype)
double etype;
{
static long int ahead, behind, item, item_, row;
static float high, low, size, value;
```

```
/* *** SEARCH THE EVENT LIST. */
if( simlib.lsize[simlib.levent - 1L] != 0L ){
row = llists.head[simlib.levent - 1L];
low = etype - 0.1;
high = etype + 0.1;
while( TRUE ){
value = llists.master[1L][row - 1L];
if( (low < value) && (high > value) )
goto L_10;

/* *** GO TO THE NEXT EVENT. */

if( row == llists.tail[simlib.levent - 1L] )
return;
row = llists.linksr[row - 1L];
}

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

/* *** CANCEL THIS EVENT. */

L_10:
if( row == llists.head[simlib.levent - 1L] ){
```

```
/* *** REMOVE THE FIRST EVENT IN THE EVENT LIST. */

removve( simlib.lfirst, simlib.levent );
}
else if( row == llists.tail[simlib.levent - 1L] ){

/* *** REMOVE THE LAST EVENT IN THE EVENT LIST. */

removve( simlib.llast, simlib.levent );
}
else{

/* *** REMOVE THIS EVENT WHICH IS SOMEWHERE IN THE MIDDLE OF
THE EVENT
 * *** LIST. */

ahead = llists.linksr[row - 1L];
behind = llists.linkpr[row - 1L];
llists.linksr[behind - 1L] = ahead;
llists.linkpr[ahead - 1L] = behind;
llists.linksr[row - 1L] = llists.nar;
llists.linkpr[row - 1L] = 0L;
llists.nar = row;
simlib.lsize[simlib.levent - 1L] = simlib.lsize[simlib.levent - 1L] -
 1L;
```

```
/* *** PLACE THE ATTRIBUTE OF THE CANCELED EVENT IN THE TRNSFR
ARRAY. */

for( item = 1L; item <= simlib.maxatr; item++ ){
item_ = item - 1;
simlib.trnsfr[item_] = llists.master[item_][row - 1L];
}

/* *** UPDATE THE AREA UNDER THE NUMBER IN LIST CURVE. */

size = simlib.lsize[simlib.levent - 1L];
timest( size, 45L );
}
}
return;
} /* end of function */

void /*FUNCTION*/ sampst(value, varibl)
double value;
long int varibl;
{
static long int ivar, ivar_, nobs[20];
static float max_[20], min_[20], sum[20];
```

```
/* *** If the variable value is improper, stop the simulation. */
if( varibl >= -20L && varibl <= 20L ){

/* *** Execute the desired option. */

if( varibl < 0L ){

/* ------------------------------------------------------------------ */

/* *** Report the results. */

ivar = -varibl;
simlib.trnsfr[0L] = 0.;
simlib.trnsfr[1L] = nobs[ivar - 1L];
simlib.trnsfr[2L] = max_[ivar - 1L];
simlib.trnsfr[3L] = min_[ivar - 1L];
if( nobs[ivar - 1L] != 0L )
simlib.trnsfr[0L] = sum[ivar - 1L]/simlib.trnsfr[1L];
}
else if( varibl > 0L ){

/* ------------------------------------------------------------------ */

/* *** Collect data. */
```

```
sum[varibl - 1L] = sum[varibl - 1L] + value;
if( value > max_[varibl - 1L] )
max_[varibl - 1L] = value;
if( value < min_[varibl - 1L] )
min_[varibl - 1L] = value;
nobs[varibl - 1L] = nobs[varibl - 1L] + 1L;
}
else{

/* ------------------------------------------------------------------- */

/* *** Initialize the routine. */

for( ivar = 1L; ivar <= 20L; ivar++ ){
ivar_ = ivar - 1;
sum[ivar_] = 0.;
max_[ivar_] = -1.e30;
min_[ivar_] = 1.e30;
nobs[ivar_] = 0L;
}
}
return;
}
else{
fprintf( stdout, " From SIMLIB:%10ld is improper value for SAMPST variable at time
```

```
%10.3f\n",
 varibl, simlib.time );
exit(0);
}
return;
} /* end of function */

void /*FUNCTION*/ timest(value, varibl)
double value;
long int varibl;
{
static long int ivar, ivar_;
static float area[45], max_[45], min_[45], preval[45], tlvc[45], treset;

/* *** If the variable value is improper, stop the simulation. */
if( varibl >= -45L && varibl <= 45L ){

/* *** Execute the desired option. */

if( varibl < 0L ){

/* ------------------------------------------------------------------ */

/* *** Report the results. */
```

```
ivar = -varibl;
area[ivar - 1L] = area[ivar - 1L] + (simlib.time - tlvc[ivar - 1L])*
 preval[ivar - 1L];
tlvc[ivar - 1L] = simlib.time;
simlib.trnsfr[0L] = area[ivar - 1L]/(simlib.time - treset);
simlib.trnsfr[1L] = max_[ivar - 1L];
simlib.trnsfr[2L] = min_[ivar - 1L];
}
else if( varibl > 0L ){

/* ----------------------------------------------------------------- */

/* *** Collect data. */

area[varibl - 1L] = area[varibl - 1L] + (simlib.time -
 tlvc[varibl - 1L])*preval[varibl - 1L];
if( value > max_[varibl - 1L] )
max_[varibl - 1L] = value;
if( value < min_[varibl - 1L] )
min_[varibl - 1L] = value;
preval[varibl - 1L] = value;
tlvc[varibl - 1L] = simlib.time;
}
else{
```

```
/* ------------------------------------------------------------------ */

/* *** Initialize the routine. */

for( ivar = 1L; ivar <= 45L; ivar++ ){
ivar_ = ivar - 1;
area[ivar_] = 0.;
max_[ivar_] = -1.e30;
min_[ivar_] = 1.e30;
preval[ivar_] = 0.;
tlvc[ivar_] = simlib.time;
}
treset = simlib.time;
}
return;
}
else{
fprintf( stdout, " From SIMLIB:%10ld is improper value for TIMEST variable at time
%10.3f\n",
 varibl, simlib.time );
exit(0);
}
return;
} /* end of function */
```

```
void /*FUNCTION*/ filest(list)
long int list;
{
static long int ilist;

/* *** Compute summary statistics for the list. */
ilist = -(20L + list);
timest( 0., ilist );
return;
} /* end of function */



double /*FUNCTION*/ expon(rmean, istrm)
double rmean;
long int istrm;
{
static float expon_v, u;

/* *** GENERATE A U(0,1) RANDOM VARIABLE. THE FORM OF THIS
STATEMENT
 * *** DEPENDS UPON THE COMPUTER USED. */
u = rannd( istrm );

/* *** GENERATE AN EXPONENTIAL RANDOM VARIABLE WITH RMEAN. */
```

```
expon_v = -rmean*log( u );
return( expon_v );
} /* end of function */



double /*FUNCTION*/ erlang(m, rmean, istrm)
long int m;
double rmean;
long int istrm;
{
static long int i, i_;
static float erlang_v, mexp;

/*      Generate an M-Erlang random variate with mean RMEAN using stream
 *     ISTRM */
mexp = rmean/m;
erlang_v = 0.0;
for( i = 1L; i <= m; i++ ){
i_ = i - 1;
erlang_v = erlang_v + expon( mexp, istrm );

}
return( erlang_v );
} /* end of function */
```

```
double /*FUNCTION*/ rannd(istrm)
long int istrm;
{
static long int hi15, hi31, low15, lowprd, ovflow, zi;
static float rannd_v;
static long mult1 = 24112;
static long mult2 = 26143;
static long b2e15 = 32768;
static long b2e16 = 65536;
static long modlus = 2147483647;
static long zrng[100]={1973272912,281629770,20006270,1280689831,2096730329,
 1933576050,913566091,246780520,1363774876,604901985,1511192140,
 1259851944,824064364,150493284,242708531,75253171,1964472944,
 1202299975,233217322,1911216000,726370533,403498145,993232223,
 1103205531,762430696,1922803170,1385516923,76271663,413682397,
 726466604,336157058,1432650381,1120463904,595778810,877722890,
 1046574445,68911991,2088367019,748545416,622401386,2122378830,
 640690903,1774806513,2132545692,2079249579,78130110,852776735,
 1187867272,1351423507,1645973084,1997049139,922510944,2045512870,
 898585771,243649545,1004818771,773686062,403188473,372279877,
 1901633463,498067494,2087759558,493157915,597104727,1530940798,
 1814496276,536444882,1663153658,855503735,67784357,1432404475,
```

```
619691088,119025595,880802310,176192644,1116780070,277854671,
1366580350,1142483975,2026948561,1053920743,786262391,1792203830,
1494667770,1923011392,1433700034,1244184613,1147297105,539712780,
1545929719,190641742,1645390429,264907697,620389253,1502074852,
927711160,364849192,2049576050,638580085,547070247};

/*   Prime modulus multiplicative linear congruential generator
 *   Z(I) = (630360016 * Z(I - 1)) (MOD(2**31 - 1)), based on Marse
 *   and Roberts's portable random-number generator UNIRAN. Multiple
 *   (100) streams are supported, with seeds spaced 100,000 apart.
 *   Throughout, input arguement ISTRM must be an integer giving the
 *   desired stream number. */
/*   Usage: (Three options) */
/*   1. To obtain the next U(0,1) random number from stream ISTRM,
 *      execute
 *         U = RAND(ISTRM)
 *      The real varible U will contain the next random number. */
/*   2. To set the seed for stream ISTRM to a desired value IZSET,
 *      execute
 *         CALL RANDST(IZSET,ISTRM)
 *      where IZSET*must be an integer constant or variable set to the
 *      desired seed, a number between 1 and 2147483646 (inclusive).
 *      Default seeds for all 100 streams are given in the code. */
/*   3. To get the current (most recently used intyeger in the
 *      sequence being generated for stream ISTRM into the INTEGER
```

```
*       variable IZGET, execute
*          IZGET = IRANDG(ISTRM) */
/*    Force saving of ZRNG between calls. */
/*    Define the constants */
/*    Set the default seeds for all 100 streams. */
/*    Generate the next random number */
zi = zrng[istrm - 1L];
hi15 = zi/b2e16;
lowprd = (zi - hi15*b2e16)*mult1;
low15 = lowprd/b2e16;
hi31 = hi15*mult1 + low15;
ovflow = hi31/b2e15;
zi = (((lowprd - low15*b2e16) - modlus) + (hi31 - ovflow*b2e15)*
 b2e16) + ovflow;
if( zi < 0L )
zi = zi + modlus;
hi15 = zi/b2e16;
lowprd = (zi - hi15*b2e16)*mult2;
low15 = lowprd/b2e16;
hi31 = hi15*mult2 + low15;
ovflow = hi31/b2e15;
zi = (((lowprd - low15*b2e16) - modlus) + (hi31 - ovflow*b2e15)*
 b2e16) + ovflow;
if( zi < 0L )
zi = zi + modlus;
```

```
zrng[istrm - 1L] = zi;
rannd_v = (2L*(zi/256L) + 1L)/16777216.0;

dat.ncount[istrm - 1L] = dat.ncount[istrm - 1L] + 1L;

return( rannd_v );
} /* end of function */



void /*FUNCTION*/ removve(option, list)
long int option, list;
{
static long int ihead, itail, item, item_, row;

/* *** If the list value is improper, stop the simulation. */
if( !(list >= 1L && list <= 25L) ){
fprintf( stdout, " From SIMLIB:%10ld is an improper value for REMOVE LIST at time
%10.3f\n",
 list, simlib.time ):

/* *** If the list is empty, stop the simulation. */

}
else if( simlib.lsize[list - 1L] > 0L ){
```

```
/* *** Decrement the list size. */

simlib.lsize[list - 1L] = simlib.lsize[list - 1L] - 1L;

/* *** If the option value is improper, stop the simulation. */

if( option == 1L || option == 2L ){

/* *** If there is more than one record in the list, continue. */

if( simlib.lsize[list - 1L] == 0L ){

/* ------------------------------------------------------------------- */

/* *** Remove the only record in the list. */

row = llists.head[list - 1L];
llists.head[list - 1L] = 0L;
llists.tail[list - 1L] = 0L;

/* *** Remove according to the desired option. */

}
else if( option == 2L ){
```

```
/* ------------------------------------------------------------------ */

/* *** Remove the last record in the list. */

row = llists.tail[list - 1L];
itail = llists.linkpr[row - 1L];
llists.linksr[itail - 1L] = 0L;
llists.tail[list - 1L] = itail;

/* *** Go to transfer the data. */

}
else{

/* ------------------------------------------------------------------ */

/* *** Remove the first record in the list. */

row = llists.head[list - 1L];
ihead = llists.linksr[row - 1L];
llists.linkpr[ihead - 1L] = 0L;
llists.head[list - 1L] = ihead;

/* *** Go to transfer the data. */
```

```
}

/* ------------------------------------------------------------------- */

/* *** Transfer the data. */

llists.linksr[row - 1L] = llists.nar;
llists.linkpr[row - 1L] = 0L;
llists.nar = row;
for( item = 1L; item <= simlib.maxatr; item++ ){
item_ = item - 1;
simlib.trnsfr[item_] = llists.master[item_][row - 1L];
}

/* *** Update the area under the number in list curve. */

timest( (float)( simlib.lsize[list - 1L] ), 20L + list );
return;
}
else{
fprintf( stdout, " From SIMLIB:%10ld is improper value for REMOVE OPTION at
time %10.3f\n",
 option, simlib.time );
}
```

```
}
else{
fprintf( stdout, " From SIMLIB: underflow of list %2ld at time %10.3f\n",
 list, simlib.time );
}
exit(0);
} /* end of function */

long ipow( m, n )       /* returns:  m^n */
long m;         /* base */
long n;         /* exponent */
{
long p; /* holds partial product */

if( m == 0 )
return( 0L );

if( n < 0 ){    /* test for negative exponent */
n = -n;
m = 1/m;
}

p = IS_ODD(n) ? m : 1;  /* test & set zero power */

while( n >>= 1 ){       /* now do the other powers */
```

```
m *= m;                /* sq previous power of m */
if( IS_ODD(n) ) /* if low order bit set */
p *= m;        /*     then, multiply partial product by latest power of m */
}

return( p );
}

void /*FUNCTION*/ reportt()
{
static long int i, i_, index, kount;
static float ajdel[5], amdel[5], autil[5], avgniq[5], oajdel, sum;

/*    Compute the average total delay in queue for each jobtype and the
 *    overall average job delay. */
model.cost = 0.0;
 oajdel = 0.0;
 kount = 0;
sum = 0.0;
for( i = 1L; i <= model.ntypes; i++ ){
i_ = i - 1;
index = model.ngroup + i;
sampst( 0., -index );
ajdel[i_] = simlib.trnsfr[0L]*model.ntasks[i_];
oajdel = oajdel + (model.probd[i_] - sum)*ajdel[i_];
```

```
sum = model.probd[i_];
}

/*    Compute the average number in queue, the average utilization, and
 *    the average dealy in queue for each machine group. */

for( i = 1L; i <= model.ngroup; i++ ){
i_ = i - 1;
sampst( 0., -i );
amdel[i_] = simlib.trnsfr[0L];
filest( i );
avgniq[i_] = simlib.trnsfr[0L];
timest( 0., -i );
autil[i_] = simlib.trnsfr[0L]/model.nmachs[i_];
}
for( i = 1L; i <= model.ngroup; i++ ){
i_ = i - 1;
kount = kount + model.nmachs[i_];
}
model.cost = oajdel*1000L + kount*1000L;
return;
} /* end of function */

p
```

# Vita

James M. Yunker was born on May 3, 1951, in Owensboro, Kentucky. He received a BA and BS in Engineering from Vanderbilt University in 1973 and 1974, respectively and an MS in chemical engineering from the University of California at Berkeley in 1977. He received an MBA and MS from The University of Michigan in 1983 and 1986, respectively.

He worked for Ashland Chemical Company 1977 to 1978 as a research chemical engineer. He also worked at Ashland Oil as a process design engineer from 1978 to 1981.