GENETIC ALGORITHM APPROACH FOR REORDER CYCLE TIME DETERMINATION IN MULTI-STAGE SYSTEMS

by

Heidi Lízabeth Romero Encarnación

A thesis submitted in partial fulfillment
of the requirements for a degree of

MASTER IN SCIENCE
in
Industrial Engineering

UNIVERSITY OF PUERTO RICO
MAYAGÜEZ CAMPUS
2003

Approved by:

_____	_____
Sonia M. Bartolomei Suárez, Ph.D. Member, Graduate Committee	Date
_____	_____
William Hernández Rivera, Ph.D. Member, Graduate Committee	Date
_____	_____
Viviana I. Cesaní Vázquez, Ph.D. President, Graduate Committee	Date
_____	_____
Miguel A. Seguí Figueroa, L.L.M. Representative of Graduate Studies	Date
_____	_____
Agustín Rullán Toro, Ph.D. Chairperson of the Department	Date

ABSTRACT

The objective of this research is to provide a genetic algorithm to determine the reorder cycle time for multi-stage serial and assembly systems. Demand for end item is assumed to occur at a constant and continuous rate. Production is instantaneous and no backorders are allowed. Both setup and echelon holding costs are charged at each stage. The attention is on nested and stationary policies. Furthermore, the reorder interval for each echelon is restricted to be not only integer, but also a power of two times a base planning period. The performance of the genetic algorithm is evaluated in comparison with an optimal approach proposed by *Maxwell and Muckstadt (1985)*, obtaining solutions from zero to five percent from the optimum for small problems. Experimentation is conducted to determine the genetic algorithm parameters and in addition to evaluate the robustness of the proposed methodology.

RESUMEN

El objetivo de esta investigación es proveer un algoritmo que calcule el tiempo de reordenar productos en un sistema de múltiples etapas con estructura serial y de ensamblaje. La demanda del producto final es constante y continua. La producción es instantánea y no se permiten órdenes atrasadas. Los costos de ordenar y de mantener en inventario se cargan a cada etapa. El enfoque es en políticas jerárquicas y estáticas. Además, el tiempo de reordenar tiene la restricción de ser entero y la potencia de dos de un periodo base de planificación. El algoritmo genético es evaluado en comparación con la solución óptima desarrollada por *Maxwell y Muckstadt (1985)*, obteniendo soluciones de cero a cinco por ciento por encima de la solución óptima para problemas pequeños. Se realizaron experimentos para determinar los valores óptimos de los parámetros del algoritmo genético y en adición para evaluar cuan robusta es la metodología propuesta.

DEDICATORY

This thesis is dedicated to my parents Marítza and José, to my sister and brother Annie-Belle and José Francisco, and my dear nephew José Javier. A special dedication to my fiancé Anthony

ACKNOWLEDGEMENTS

I am greatly indebted to my advisor professor Viviana Cesaní for her guidance and constant support during the investigation. I wish to thank the members of my committee Professors Sonia Bartolomei and William Hernández for valuable discussions. I am also indebted to professor M. Fatih Tasgetiren who contributed with the idea of the investigation focus and chromosome representation. Thanks to Zuriel Correa for his disinterested support in learning me C++ programming language.

I appreciated the support offered by the University of Puerto Rico for the opportunity to obtain the master degree and the Industrial Engineering Department for their financial support.

Thanks God for all the blessings received. Several people have been very helpful to me during the last three years. In particular I wish to thank my new family in Agüadilla, family Rodríguez Michel: Rosa Luz, Francisco, Rosita, Francisco Alberto, Tito, Sairi and Victoria Isabel, thanks for been tremendously supportive, for Rivera's family, and all their friends who always treat me as part of their family. An special thank to Dr. Andrés Calderón and his wife Rosa.

My profound thanks for my friends: Darwin, Betty, Carolina, Atilio, Amelia, Omar, Julián, Alejandro, Marlene, Jeannette, Jamell, Martha, Dennys, Zuriel, Giannina, Juan Guillermo, Catherine, Alexis, Yaleidi, Janet, Carlos Andrés, Karina, André, Geovannie, Paola, Maria, Jesús, and all who share special moments during these years.

TABLE OF CONTENTS

TABLE OF CONTENTS VI
LIST OF TABLES VIII
LIST OF FIGURES IX
LIST OF APPENDIXES XI
LIST OF APPENDIXES XI

CHAPTER I 1

INTRODUCTION 1
1. 1. Justification 1
1.2. Purpose of the study 4
1.3 Scope 5

CHAPTER II 8

LITERATURE REVIEW 8
2.1. Introduction 8
2.2. Lot Sizing Problems 8
2.3. Reorder Cycle Time Problems 16
2.4. Genetic Algorithm Models 23
2.5. Conclusions 25

CHAPTER III 27

METHODOLOGY 27
3.1. Introduction 27
3.2. Problem Definition 28
3.3. Maxwell and Muckstadt Approach 28
3.4. Genetic Algorithm Approach 32
3.5. Experimental Design 41

3.7. Conclusions *44*

CHAPTER IV **46**

EXPERIMENTAL ANALYSIS 46

4.1. GA parameters results *46*

4.2. GA robustness experiment results *55*

4.3. Conclusions *58*

CHAPTER V **60**

CONCLUSIONS AND DISCUSSIONS 60

5.1. Conclusions *60*

REFERENCES 63

LIST OF TABLES

Table 4.1 Summary GA results for parameters set up 47

Table 4.2 Multiple regression analysis for cost 48

Table 4.3 Analysis of variance for cost 48

Table 4.4 Multiple regression analysis for cost with significant factors 50

Table 4.5 Analysis of variance for cost with significant factors..... 50

Table 4.6 Multiple regression analysis for time..... 51

Table 4.7 Analysis of variance for time..... 51

Table 4.8 Multiple regression analysis for time with significant factors..... 52

Table 4.9 Analysis of variance for time with significant factors..... 53

Table 4.10 Optimization spreadsheet parameters 54

Table 4.11 Summary comparison of ga and maxwell and muckstadt 54

Table 4.12 Summary ga results for robustness design..... 56

LIST OF FIGURES

Figure 1.1 Multi-stage serial and assembly structures 5

Figure 2.1 Cycle inventory level over time 9

Figure 3.1 General ga procedure 33

Figure 3.2 Chromosome representation 34

Figure 3.3 One point mutation 36

Figure 3.4. Two point crossover 36

Figure 3.5 Example problem structure 38

Figure 3.6 First generation of problem example using ga 39

Figure 3.7 Solution to problem example 40

Figure 3.8 Example problem solution using ga 41

Figure 4.1 Main effects plots for cost 57

Figure 4.2 Normal probability plot of the residuals for robustness experiment 57

Figure C.1 Residuals plot for cost versus the order of the data 93

Figure C.2 Mean and 95 percent intervals plot for cost versus generations 94

Figure C.3 Mean and 95 percent intervals plot for cost versus problem size 94

Figure C.4 Residuals plot for time versus the order of the data 95

Figure C.5 Mean and 95 percent intervals plot for time versus crossover 95

Figure C.6 Mean and 95 percent intervals plot for time versus generations 96

Figure C.7 Mean and 95 percent intervals plot for cost versus mutation 96

Figure C.8 Mean and 95 percent intervals plot for time versus the problem size 97

LIST OF APPENDIXES

ALGORITHMS CODES 67

A.1 Maxwell and Muckstadt (1985) C ++ code *67*

A.2 Genetic algorithm C++ code *67*

PROBLEM INSTANCES 89

B.1.Problem data generator code (Matlab 6.5) *89*

B.2 Example problem *89*

RESIDUALS ANALYSIS 92

C.1 Residuals analysis for cost response in the GA parameters experiment *92*

C.2 Residuals analysis for time response in the GA parameters experiment *92*

CHAPTER I

INTRODUCTION

1. 1. Justification

An accelerating trend toward globalization marked the latter half of the twentieth century and the beginning of the present one. It is common to see a company design, produce and distribute products through a global network to provide the best customer service at the lowest price. Coordination throughout the entire logistical system must be planned and managed, because of the impact in costs that it represents to the companies and their opportunity to compete in today's global market. The supply chain management is defined by the Council of Logistic Management as:

"The process of planning, implementing and controlling the efficient, cost effective flow and storage of materials, in-process inventory, finished good, and related information from point-of-origin to point-of-consumption for the purpose of conforming to customers requirements".

A central issue in the supply chain performance is the inventory management. Inventories are present at every stage of the supply chain as raw materials to finished goods. The inventory acts as a buffer against any uncertainty, but holding inventory is costly and runs the risk of product deterioration and obsolescence. The focus of inventory problems traditionally has been on lot size determination. Supply occurs in discrete batches or lots and items proceeds through a sequence of stages. The issue of the lot sizing is to determine how large these lots should be trying to find the best

balance between fixed costs and inventory holding costs. Ford Harris in 1915 introduced the classic Economic Lot Size Model which serves as reference for many other research studies.

Therefore, the lot sizing problem can be formulated as the problem of determining the reorder interval time, because of a functional relationship between the lot size and the manufacturing cycle time. Due to the fact that this problem is continuous and that the reorder optimal interval can take any positive real value, is often impractical to implement it. This is referred to a discrete problem imposing the restriction that the reorder interval can take only positive integer values.

Maxwell and Muckstadt (1985) explain the advantages of formulating the problem in terms of reorder intervals rather than in terms of lot sizes. They establish three principal reasons for this: (1) the experience that production planning is more naturally centered around the frequency of production because it dictates the numbers of set-ups, the requests for tooling and fixtures, and the demands on the material handling system, (2) the mathematical representation of the model is simplified, and (3) from a scheduling point of view it is often practical to keep reorder intervals constant in the face of minor changes to demand forecasts and to adjust lot sizes accordingly.

A special case is given by considering the discrete problem with the power-of-two restrictions in which the reorder interval is constraint to be not only integer, but also a power of two. The power-of-two policy was developed by *Roundy (1985).* It considers the problem of determining the reorder interval instead of the reorder

quantity and has the advantage of an easy implementation, even if the system is very complex and it is known that the cost of the optimal solution for the discrete problem using the power-of-two solution of a continuous problem is within about 6% of the cost of the optimal solution of the continuous problem without those restrictions. Implementing power of two policies makes production scheduling easier, and ensures that production cycles regenerate as frequently as possible, so that inventory imbalances that in practice can be easily corrected.

Although considerable research has been devoted to traditional methods of search, optimization using such methods is not that efficient, particularly in finding a solution for very complex search space. Furthermore, significant less attention has been paid to stochastic search and optimization techniques like genetic algorithms. *Khouja, Michalewicz and Wilmot (1998)* presented a genetic algorithm for solving the Economic Lot Size Scheduling Problem finding better solutions than the iterative dynamic programming approach. Genetic algorithms have been employed to solve optimization problems across all disciplines and interests, obtaining global optimal or near optimal solutions in complex search spaces. Their simplicity permits their use to solve difficult problems, showing an important reduction in the computational time. It would thus be of interest to learn how genetic algorithms work for the reorder cycle time problem with the power of two restrictions.

The aim of this research is to present a genetic algorithm to find a solution to the problem of determining the reorder cycle time that minimizes the total cost in multi-stage serial and assembly systems. It provides a solution in costs at least eight

percent above optimum for small problems with a computational time of no more than four seconds, for large problems (2000 nodes) the solution obtained is from 6 to 29 percent above optimum, tested in different problem instances. The cost function is composed by the fixed ordering cost and the holding cost. It is assumed that the cycle length satisfy the power of two restrictions to take the advantages of the policy already explained and the computational advantages of the genetic algorithms.

1.2. Purpose of the study

The principal objective of this research is to find a solution to the problem of determining the reorder cycle time in multi-stage serial and assembly systems using a genetic algorithm approach, satisfying the power of two restrictions. Some other secondary objectives are:

- To determine the effectiveness of the GA approach, comparing the results obtained using the proposed algorithm with the *Maxwell and Muckstadt (1985)* methodology. The effectiveness of a policy is 100% times the ratio of the minimum of the average cost over all policies to the average cost of the policy in question.
- To determine the efficiency of the genetic algorithm approach, considering that one of the advantages from using this approach is to reduce the computational time while obtaining a good near optimal solution.

- To determine the robustness of the methodology using design of experiments. The factors considered included: problem size, setup or ordering costs, and holding costs.

1.3 Scope

The genetic algorithm developed is based on the assumption of a multi-stage serial and assembly systems. A stage might consist of an operation such as procurement of raw materials, fabrication of parts or assembly. The serial structure is the simplest type of multi-stage structures in which materials enter the first (1) stage and progressively pass through a sequence of stages until final product exits at the last (5) stage (Figure 1.a). In the assembly structure each operation has a unique successor, but may have several predecessors stages (Figure 1.b). The serial structure is considered as a special case of the assembly, having each stage just one predecessor.

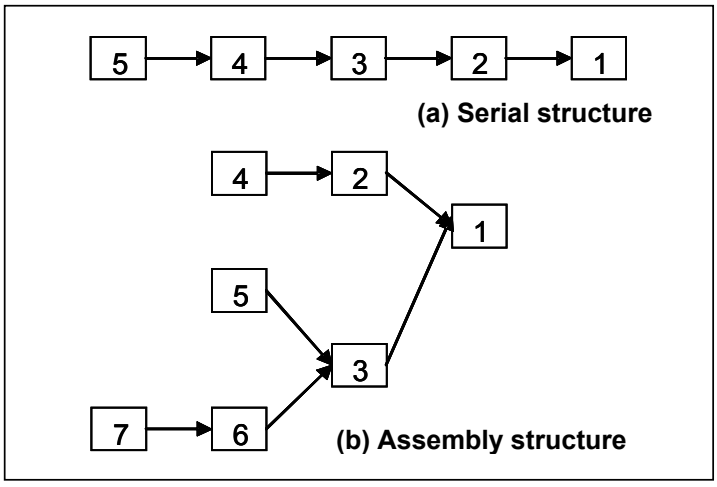


Figure 1.1 Multi-Stage Serial and Assembly Structures

The structure of the systems is limited to those that could be represented by an acyclic directed graph. Each node in the graph represents manufacturing, assembly or distribution operations, and the arcs indicate the flow of materials, components, subassemblies, assemblies, or finished product from one stage or operation to the next. This research do not intend to show the interaction between the reorder cycle time and the principal factors that could affect it at each stage. This could be considered as one of the proposed future research studies in this area.

Demand for each end item is assumed to occur at a constant and continuous rate, and is given for a planning horizon of *n* periods. Production is instantaneous and no backorders are allowed and unconstraint capacity at each node is assumed. The cost function is composed by the fixed setup cost and the holding cost. Fixed setup costs and echelon holding costs are changed at each stage.

It is assumed that the cycle length should satisfy the power of two restrictions, which applies zero inventory ordering and stationary-nested policies. The zero inventory ordering occurs when each facility orders only when its inventory is zero. A stationary policy is one in which each facility uses a fixed order quantity and a fixed interval time between successive orders. In a nested policy each facility orders every time any of its suppliers orders.

The organization of the document is as follows. Chapter II describes a review of the most important contributions in lot sizing problems for single and multi-stage models, for reorder cycle time models, including some approaches with the power of two restrictions, and the application of genetic algorithms in lot sizing problems. In

Chapter III, the two-phase algorithm proposed by the *Maxwell and Muckstadt (1985)* is presented and the genetic algorithm approach is described in detail. At the end of this chapter, two experiments are proposed, one for the determination of the optimal parameters for the genetic algorithm and one to measure the robustness of the methodology proposed. The experimental analysis is shown in Chapter IV. Finally, Chapter V briefly summarizes the conclusions and some recommendations for future works.

CHAPTER II

LITERATURE REVIEW

2.1. Introduction

Inventory problems have been studied for many years. This review describes some of the most important contributions in this field. It includes methods used to solve single and multi-stage lot sizing problems. For multi-stage systems, some models are shown that deal with special cases like capacity constraint and joined setup costs. Finally, it is presented some genetic algorithms applications that can be considered as previous work in lot sizing problems.

2.2. Lot Sizing Problems

2.1.1. Single Stage Models

For many years the main focus of the inventory theory has been in the lot size determination. Many authors try to solve the single stage problem. The classic Economic Lot Size Model, introduced by Ford Harris in 1915, is a very basic model that considered a warehouse facing constant demand for a single item. It assumes constant fixed cost, instantaneous batch delivery following a deterministic lead time, all replenishment orders are for the same quantity and no shortages are allowed. The total cost per time TC (Q), is composed by ordering cost, product purchase cost and inventory holding cost.

$$TC(Q) = ordering\cos t + purchased\cos t + inventoryholding\cos t$$

$$TC(Q) = \frac{AD}{Q} + CD + \frac{hQ}{2} \qquad \text{(Equation 2.1)}$$

Based on the cycle inventory level over time, shown in Figure2.1, the inventory level decreased constantly from the order quantity size (Q) to zero each cycle, and averages Q/2. The process repeats each time Q units are sold (every T=Q/D), integrating over this cycle length it can be found the average inventory, Ī.

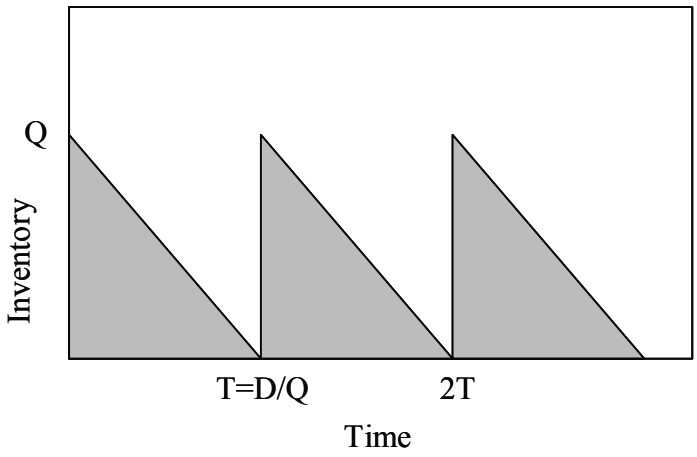


Figure 2.1 Cycle inventory level over time

$$\bar{I} = \frac{1}{Q/D}\int_0^{Q/D}(Q - tD)dt = \frac{D}{Q}\left(Qt - \frac{Dt^2}{2}\Big|_0^{Q/D}\right) = \frac{D}{Q}\left(\frac{Q^2}{D} - \frac{Q^2}{2D}\right) = \frac{Q}{2} \qquad \text{(Equation 2.2)}$$

To find the optimal order quantity it is necessary to differentiate Equation 2.1 with respect to Q and set the results to zero. These yields:

$$\frac{dTC(Q)}{dQ} = \frac{-AD}{Q^2} + \frac{h}{2} = 0 \text{ (Equation 2.3)}$$

$$Q^* = \sqrt{\frac{2AD}{h}} \quad \text{(Equation 2.4)}$$

Another important issue in the EOQ model is the definition of the total cost for the optimal quantity (Q*). In this case, ordering and holding costs are equal, so:

$$\frac{AD}{Q^*} = \frac{AD}{\sqrt{\frac{2AD}{h}}} = \sqrt{\frac{ADh}{2}} \quad \text{(Equation 2.5)}$$

Inventory holding cost per period is

$$\frac{hQ^*}{2} = \frac{h\sqrt{\frac{2AD}{h}}}{2} = \sqrt{\frac{ADh}{2}} \text{ (Equation 2.6)}$$

The total cost using the optimum lot size quantity is determined in Equation 2.8:

$$TC(Q^*) = \sqrt{\frac{ADh}{2}} + CD + \sqrt{\frac{ADh}{2}} = 2\sqrt{\frac{ADh}{2}} + CD = \sqrt{2ADh} + CD \text{ (Equation 2.7)}$$

$$TC(Q^*) = \sqrt{2ADh} + CD \qquad \text{(Equation 2.8)}$$

All this description has been provided to describe the relationship between the order quantity and the reorder cycle time both assumes to be constant. The Economic Order Quantity is used as reference point in a lot of methods proposed later.

Veinott (1967) showed that a broad class of problems (including deterministic single and multi-facility economic lot size) can be formulated as minimizing a concave

function over the solution set of Leontief substitution system. To understand what does this means it is necessary to introduce some concepts. A matrix A is called Leontief if it has exactly one positive element in each column and there is a nonnegative (column) vector x for which Ax is positive. The linear program for finding a (column) vector $x = (x_j)$, called optimal, is given by:

Objective function:

$$\text{Minimizes } cx$$

Subject to: $Ax = b, \quad x \geq 0,$ (Equation 2.9)

If A is Leontief, and $b \geq 0$, Equation 2.4 is a Leontief substitution system and has $X(b) \cap S$ as it solution set. S is the set of programs x for which $x_i x_j = 0$ for all pairs (i,j) in a specified set. In applications it is often appropriate to impose additional restrictions of the form $x_i x_j = 0$, for example in production problems if it is possible to produce only one product in each period.

Leontief substitution systems seem to provide a natural setting for studying inventory models with concave costs. Their applications are on single and multi-facility lot size problem, lot-size-smoothing and warehousing models. Their algorithms required a computational effort that increases algebraically with the size of the problem instead of exponentially.

2.1.2. Multi-stage Models

Multi-echelon inventory systems can be used to optimize the deployment of inventory in a supply chain. Multi-stage manufacturing situations (raw materials, components, subassemblies, assemblies) are conceptually very similar to multi-echelon

inventory systems. Multi-echelon models examine the entire system, searching better solutions for the entire chain, not each stage independently. This coordination has the advantage of given better global solutions. In the multi-stage systems there have been a lot of contributions in serial, assembly, distribution, general and some special structures.

The serial and assembly structures were explain in section 1.3. In the distribution structure each production stage has at most one predecessor stage but may have several successors. The tree structure combines the features of an assembly and distribution structure. Finally, the general structure shows a different relation between stages and is very common when multiple products share some of the same components. Each stage can have multiple successors and predecessors.

Clark and Scarf (1960) introduced the echelon stock concept which permits some very convenient mathematical simplifications. They define the echelon stock of echelon j (in general multi-echelon system) as the number of units in the system that are at, or have passed through, echelon j but have as yet not been specifically committed to outside customers. They considered the problem of determining optimal purchasing quantities in a multi-stage serial and distribution models. Echelon j stock may often be considered to be the facility j value-added inventory. The Clark-Scarf model allows stochastic demand and convex holding costs, but setup costs are assumed to be associated with no more than two facilities.

Crowston, Wagner and Henshaw (1972) made a comparison of exact and heuristics routines for lot size determination in multi-stage assembly systems. They concluded that economic lot sizes in multi-stage assembly systems can be determined by

dynamic programming for problems of moderate size, while heuristic search routines appear to be promising for large problems. Using these results *Crowston, Wagner and Williams (1973)* present a model for multi-stage assembly systems to compute a set of optimal lot sizes so that the lot size at each facility is a positive integer multiple of the lot size at its successor facility. It is important to mention that they considered the serial system as a special case of the assembly system. Their model assumes constant continuous final demand, instantaneous production at each stage and infinite planning horizon.

A few years later, *Williams (1982)* proved that the well known theorem by Crowston, Wagner and Williams (1973) shows to be defective. The theorem establishes that an optimal solution to the batch size determination problem for multi-echelon production/inventory assembly structures is characterized by a set of lot sizes, such that the lot size at each stage must be an integer multiple of the lot size at its successor stage. The theorem proved to be defective at the point that results were extended from two level systems to more general assembly systems.

Schwarz (1973) deals with a one-warehouse n-retailer deterministic inventory system with known demands. As a conclusion, he shows that the form of the optimal policy can be very complex for more than four retailers and he argues for restricting attention to a simpler class of strategies (where each location's order quantity does not change with time) and develops an effective heuristic for finding good solutions.

Schwarz and Schrage (1975) make use of the myopic strategy. Myopic policies optimize a given objective function with respect to any two stages and ignore multi-stage

interaction effects. Optimal and near optimal policies were proposed for multi-echelon production/inventory assembly systems under continuous review with constant demand over and infinite planning horizon. Schwarz and Schrage model was widely used as a standard among the multi-stage production/inventory models.

Szendrovits (1981) presented a comment on the optimality in Schwarz and Schrage model, considering that their restrictions could be helpful to facilitate analytical tractability, but do not necessarily lead to optimal inventory policies as claimed by the authors. Szendrovits showed that a lower cost solution could be obtained in sample problems when the integrality constraint was violated. The example provided a lower cost solution by permitting two lots at a given stage to provide the total input for the three lots at its successor stages. Hence, it is not well established in the literature that the theorem does not characterize optimality for the case of finite production rate.

Later, *Blackburn and Millen (1985)* proposed simple cost modifications to improve the global optimality of the Schwarz and Schrage procedure. The effectiveness of these alternative modifications was tested through a series of simulation experiments.

A new formulation of the lot sizing problem in multi-stage assembly systems which leads to an effective optimization algorithm was proposed by *Afentakis, Gavish and Karmarkar (1984).* The problem was reformulated in terms of echelon stock which simplifies it decomposition by a Lagrangean relaxation method. A Branch and Bound algorithm which uses the bounds obtained by the relaxation was developed and tested.

A significant amount of work in this area has focus on evaluating the performance of the proposed techniques. *Blackburn and Millen (1985)* examined seven different

heuristic algorithms, six combination of methods and four cost modification procedures. A series of simulation experiments was conducted and it was concluded that the combination methods when used with some of the cost modifications result in enhanced performance in comparison to other sequential approaches. *Axsäter (1986)* analyzed the applicability in practice of some standard lot sizing problems and the way in which some adjustments can be considered. Assumptions in lot sizing models and the extent to which these assumptions are valid in practical situations are discussed.

A branch-and-bound based algorithm for optimal lot sizing of products with a complex product structure was proposed by *Afentakis and Gavish (1986)*. It assumed unconstraint production facilities and suggested that the formulation of the lot sizing problem in terms of its echelon stock, and the use of Lagrangean relaxation, seems to yield efficient algorithms. *Afentakis (1987)* developed an improved heuristic method for the dynamic lot-sizing problem in multi-stage production systems. This is a generalization of the single stage Wagner-Within algorithm, and attempts to optimize over all stages simultaneously, while building the production plans in a forward manner.

Billington, Blackburn, Maes, Millen, and Wassenhove (1994) examined the performance of heuristics found effective for the capacitated multiple-product, single stage problem in multi-stage settings. This study is one of the most comprehensive in terms of the number of methods examined and the conditions under which they were examined. The single-stage heuristics included in the study are: *Dixon/Silver (1981), Lambrecht and Vanderveken (1979)*, the *Dogramaci, Panayiotopoulos and Adam (1981)*, and different versions of the ABC heuristics of *Maes and Van Wassenhove (1986)*. These

heuristics are altered in two ways: (1) they allow the inclusion of the cost modification procedures developed by Blackburn and Millen, and (2) the feasibility routines have been modified to work in multi-stage environments. Both modifications attempt to coordinate decisions made across stages concerning lot sizes.

2.3. Reorder Cycle Time Problems

Based on the traditional Economic Order Quantity model showed before, the time between two consecutive orders, called reorder interval, is constant and proportional to the order quantity. The lot sizing problem can be formulated as the determination of the reorder cycle interval. Based on Equation 2.4 and T=Q/D, and ignoring the production cost (because it won't affect for the comparison), the optimum reorder interval can be derived as

$$T^* = \frac{Q^*}{D} = \frac{\sqrt{\frac{2AD}{h}}}{D} = \sqrt{\frac{2AD}{hD^2}} = \sqrt{\frac{2A}{hD}} \quad \text{(Equation 2.10)}$$

$$T^* = \sqrt{\frac{2A}{hD}} \quad \text{(Equation 2.11)}$$

The total cost based on the lead time can be derived based on Equation 2.1

$$TC(T) = \frac{AD}{TD} + \frac{hTD}{2} = \frac{A}{T} + \frac{hTD}{2} \quad \text{(Equation 2.12)}$$

$$TC(T) = \frac{A}{T} + \frac{hTD}{2} \quad \text{(Equation 2.13)}$$

The total cost for the optimum reorder interval is given by

$$TC(T)^* = \frac{A}{T^*} + \frac{hT^* D}{2} = \frac{A}{\sqrt{\frac{2A}{hD}}} + \frac{hD}{2}\sqrt{\frac{2A}{hD}} = \sqrt{\frac{hDA}{2}} + \sqrt{\frac{hDA}{2}} = 2\sqrt{\frac{hDA}{2}}$$

(Equation 2.14)

$$TC(T)^* = \sqrt{2hDA} \qquad \text{(Equation 2.15)}$$

The multi-stage lot sizing problem can be formulated as follow

$$Minimize \sum_{j=1}^{i} \frac{A_i}{T_i} + g_i T_i$$

Subject to:

$T_i \geq T_{i-1}$ Nested ness constraints (Equation 2.16)

As mentioned before, this problem is continuous and the reorder optimal interval can take any positive real value. However, their solution presents some difficulties. This is the reason to solve it as a discrete problem, imposing the restriction that the reorder interval can take only positive integer values.

There are several reasons to formulate the lot sizing model in terms of reorder intervals as described in the justification part (section 1.1). A lot of authors have been developing new techniques to solve the problem in terms of this point of view; some of the most important are mentioned here.

Elmaghraby (1978) analyzed the economic lot scheduling problem (ELSP), which arises from the desire to accommodate the cyclical production pattern when several products are made on a single facility. This work reviews the contributions to the problem, and extends the analysis in four directions: (1) offers an improved analytical approach based on dynamic programming. It tries to guarantee feasibility at the outset, by imposing some constraints on the cycle times, then to optimize the individual cycle duration subject to the imposed constraint. The solution obtained in this manner is feasible and optimal over its set of solutions; (2) a test of feasibility of a given set of parameters, through an integer linear programming formulation; (3) a systematic procedure for escape from infeasibility, when the set of parameters were judge infeasible; and (4) a procedure for the determination of a basic period for a given set of multipliers to achieve a feasible schedule.

Szendrovits (1975) presented the functional relationship between the production lot size, the manufacturing cycle time and the average process inventory in a production system, and illustrated the resulting effect on the conventional Economic Lot Quantity model. He treats the manufacturing cycle time as a function of the lot size in a multi-stage production system. This model was called the economic production quantity (EPQ). This study challenges the widely accepted doctrine of the efficiency of long production runs.

Roundy (1985) introduced two simple policies called q-optimal integer-ratio and optimal power-of-two, which are proved to be 94% and 98% effective. The effectiveness of a policy is 100% times the ratio of the minimum of the average cost over all policies to

the average cost of the policy in question. Both policies are very efficient and their most important advantage is the flexibility it allows in choosing the order intervals to correspond to easily-implemented time periods.

The power-of-two policy is a special case of the discrete problem for determining the reorder cycle time, in which the reorder interval is constraint to be not only integer, but also a multiple of two. It allows us to obtain an extremely efficient algorithm which produces a policy having an average cost within 2% of the minimum possible. *Mitchell (1987)* extended Roundy's results for the backlogging problem, obtaining a 98% effective policy for the backlogging problem in O (N log N) time.

Maxwell and Muckstadt (1985) presented an algorithm that can be used to find consistent and realistic reorder intervals for each item in large-scale production-distribution systems. Attention was restricted to policies that are nested, stationary, and a power-of-two multiple of a base planning period. The model that results from the assumptions is an integer nonlinear programming problem. It was showed that the solution to this problem is similar to that of the economic lot size problem with a modified echelon holding cost for an operation, to reflect the precedence constraints of the production-distribution system.

Roundy (1986) studied a multi-product multi-stage production inventory system in continuous time. In process and finished goods were referred to as products and inventories of a single item held at different locations were treated as different products. External demand can occur for any or all of the products at a constant, product-dependent rate. In the new policy defined by Roundy each product uses a stationary interval of time

between successive orders, and the ratio of the order intervals of any two products is an integer power of two. The effectiveness of an optimal power-of-two policy is at least 98%. The algorithm is efficient for very large systems.

Askin and Goldberg (2002) shows the demonstration of the statement that says that the total cost using the power-of-two policy cannot be increased by more than 6% above the optimum. They suppose that a non optimal cycle time T = α T* is used. Showing the ratio of the true optimal objective value to the objective value under the non optimal cycle time:

$$\frac{TC(T)}{TC(T^*)} = \frac{\frac{A}{\alpha T^*} + \frac{h\alpha T^* D}{2}}{\sqrt{2AhD}} = \frac{\alpha^{-1} + \alpha}{2} \text{ (Equation 2.17)}$$

Restricting the cycle length of a product to $T_i = 2^{K_i} T_L$ for some non-negative integer k_i=0,1,2,3,…, considering that T_L is defined as a convenient cycle length that may be a day, week or some other natural period, it is necessary to ensure that $T^* \geq T_L$. Because the cost function TC(T) is convex in T, it was chosen the smallest k satisfying $TC(2^{K+1} T_L) \geq TC(2^{K_i} T_L)$ as the optimal power-of-two policy. Combining these results with Equation 10, then

$$\left(\alpha_1 + \alpha_1^{-1}\right) \geq \left(\alpha_2 + \alpha_2^{-1}\right), \text{ (Equation 2.18)}$$

$$\alpha_1 = \frac{2^{K+1} T_L}{T^*} \text{ and } \alpha_2 = \frac{2^K T_L}{T^*} \text{ (Equation 2.17 and 2.19)}$$

Substituting and rearranging terms,

$$\left(\frac{2^{K+1}T_L}{T^*}+\frac{T^*}{2^{K+1}T_L}\right)\geq\left(\frac{2^{K}T_L}{T^*}+\frac{T^*}{2^{K}T_L}\right) \text{(Equation 2.20)}$$

$$\frac{\left(2^{k+1}-2^{K}\right)T_L}{T^*}\geq\frac{T^*}{2^{K+1}T_L} \text{ (Equation 2.21)}$$

$$2^{k}T_L\geq\frac{T^*}{\sqrt{2}} \quad \text{Lower bound (Equation 2.22)}$$

Using the same procedure

$$2^{k}T_L\leq\sqrt{2}T^* \quad \text{Upper bound (Equation 2.23)}$$

Combining equations 2.22 and 2.23, $\frac{T^*}{\sqrt{2}}\leq 2^{k}T_L\leq\sqrt{2}T^*$ for the optimal power-of-two choice of k. Finally, the $\frac{TC(T)}{TC(T^*)}$ relation tells us that

$$\frac{TC(\sqrt{2}T^*)}{TC(T^*)}=\frac{TC(\frac{T^*}{\sqrt{2}})}{TC(T^*)}=1.06 \quad \text{(Equation 2.24)}$$

Then it is proved that using the power-of-two restrictions ensures at most 6% from the optimal cost solution.

Jackson, Maxwell and Muckstadt (1988) had reviewed the Maxwell and Muckstadt (1985) model, proving a useful invariance property of the optimal partition of such systems, and used these results as the basis for algorithms to solve a capacitated version of the Maxwell-Muckstadt model. They suggest that the algorithm perform well in cases characterized by many operations per work center, however this reasoning was

based on limited argument and experience with practical examples. This approach can be effectively used to establish reorder intervals in many industrial environments.

The power-of-two policy has been extended to solve more complex problems, showing that it maintains it effectiveness. One of the major complications in managing multi-item inventory systems stems from the fact that various components, in particular, setup costs, are often jointly incurred between several distinct items. It is presented two cases with joint setup costs were power-of-two policy were applied successfully.

Jackson, Maxwell and Muckstadt (1985) presented an efficient procedure for the joint replenishment problem under the restriction that the reorder intervals must be power of two times a based period length. To solve the joint replenishment problem requires answering two questions: (1) what is the optimal time between major setups? , and (2) what is the optimal reorder interval for each item. They demonstrate by analytic means rather than experimentation that the worst case performance is within 6% of optimality. The performance bound is more than adequate given the typical errors in estimates of the setup costs, the holding costs, and the demand rate.

Federgruen and Zheng (1992) extended the results obtained by Roundy (1985) to a general joint setup cost structure. The joint cost structure often reflects economies of scale which invoke the need for careful coordination of the items replenishment strategies, and the joint replenishment problem is the most multi-item inventory model with joint setup costs. They derived two efficient algorithms to compute an optimal power-of-two policy. The problem of determining the optimal power-of-two policy can be formulated as a nonlinear mixed integer program.

Federgruen, Zheng and Queyranne (1992) generalized Roundy's results. They considered a production-distribution network represented by a general directed acyclic network showing that the power-of-two policies are close to optimal in a general class of production-distribution networks with general joint setup costs.

2.4. Genetic Algorithm Models

Traditional methods of search and optimization are not that efficient in finding a solution for very complex search space. Genetic algorithms are stochastic search techniques based on the mechanism of natural selection and natural genetics, which requires little information to search effectively in a large or poorly understood search space.

Some of the principal advantages of the genetic algorithms are versatility, flexibility, simplicity and efficiency. They have been employed to solve optimization problems across all disciplines and interests and their simplicity permits to solve difficult problems as NP-hard problems, for machine learning and also for evolving simple programs, and the efficiency can be seen in an important reduction in the computational time. Genetic algorithms explore the solution space based on random search methods. They can find the global optimal solution or near optimal in complex search spaces. In particular a genetic search, progress through a population of points in contrast to the single point of focus like most search algorithms.

Because of the already mentioned advantages of using genetic algorithms, this technique has been widely used to solve a variety of problems in different fields of study.

Applications of genetic algorithms in production planning and inventory management include assembly line balancing, buffer size optimization, production scheduling and manufacturing cell design.

Hernández and Süer (1999) presented an application of genetic algorithms to obtain the order quantity for an uncapacitated, no shortages allowed, single-item, single-level situation, lot sizing problem. Each chromosome consists of *n* genes. Each gene refers to a period. The gene *i* of a chromosome indicates if an order has been placed in period *i* or not. Genes might have 0 or 1 value; a value of 1 indicates that an order has been placed in that period, and 0 otherwise. Experimentation was conducted to evaluate how different aspects of genetic algorithm affect the results. The aspects analyzed were: selection strategies, scaling (it forces higher reproduction probabilities to those chromosomes that represent better solutions), order and carrying costs, and net requirements. It was observed how scaling has the biggest impact.

Among the limited applications of GA to inventory problems, focusing on the reorder cycle time, the work of *Khouja, Michalewicz and Wilmot (1998)* is worth examining. They proposed the use of genetic algorithms to solve the Economic Lot Size Scheduling Problem (ELSP). The ELSP is an NP hard inventory problem which tries to schedule the production of several different items in the same facility on repetitive basis. They used the problem proposed by *Bomberger (1966)* where the facility is such that only one item can be produced at a time, there is a setup cost and a setup time associated with producing each item. The demand rate for each item is known and constant over an infinite horizon, and no shortages are allowed. Bomberger developed a dynamic

programming solution. In *Khouja, Michalewicz and Wilmot (1998)* they proposed a genetic algorithm approach. In this approach the chromosome represent floating point fundamental cycle (T) and integer multipliers (k_i's) of the basic period for each product.

Because of the advantages already explained of GA, in this research it is shown the use of a genetic algorithm to find the reorder cycle time in multi-stage serial and assembly systems, which minimize the cost function.

The power-of-two policy has been used in industry for many years, and extensive research studies on the efficiency of this restriction have been done. Based on that, the present approach includes the power of two restrictions. The new approach is compared with the methodology for the implementation of the power-of-two policy, presented *Maxwell and Muckstadt (1985)* as a nonlinear integer problem. One of the main contributions of this research is a methodology that could be easily implemented particularly in industrial applications, and that could be used to develop future studies including additional restrictions as capacity constraints.

2.5. Conclusions

It is shown that several research studies have been done for many years focusing in lot size determination for single stage systems like the classic Economic Lot Size Model by Harry Ford, and multi-stage inventory systems as *Clark and Scarf (1960)*, *Afentakis and Gavish (1986)* and *Schwarz (1973)*. Some applications for multi-stage models make use of a myopic strategy were the objective function is optimized based on any two stages, as done by Schwarz and Schrage (1957).

After the formulation of the lot sizing problem as the problem of determining the reorder cycle time, a lot of authors have been developed new techniques, like Elmaghraby (1978) who proposed an analytical approach based on dynamic programming. Moreover, Roundy (1985) introduced two policies called q-optimal integer-ratio and optimal power-of-two, which are proved to be 94% and 98% effective. The power-of-two policy is a special case of a discrete problem for determining the reorder cycle time, in which the reorder interval is constraint to be not only integer, but also a power of two. Consequently, *Maxwell and Muckstadt (1985)*, *Roundy (1986)* and Federgruen and Zheng (1992), proved the advantages of this policy applying it to problems with additional restrictions.

All the research studies previously mentioned used traditional search methods, which are proven to be not very efficient in finding a solution for complex search spaces. Less attention has been paid to stochastic search and optimization techniques like genetic algorithms. *Hernández and Süer (1999)* apply genetic algorithm for lot sizing problem in a single stage situation. In addition, *Khouja, Michalewicz and Wilmot (1998)* presented genetic algorithm approaches to inventory problems focusing on the reorder cycle time. However, literature have not been address about genetic algorithms applications using the power-of-two restrictions, taking the advantages already explained of this policy.

The next chapter describes the first genetic algorithm developed to solve a problem of determining the reorder cycle time determination in multi-stage serial and assembly systems, considering the power-of-two restrictions.

CHAPTER III

METHODOLOGY

3.1. Introduction

The methodology used in the present work includes three major tasks: (1) development of a genetic algorithm to solve the proposed problem, (2) measurement of the effectiveness of the genetic algorithm, and (3) identification of its robustness using experimental design.

To satisfy the objectives of this research, it is necessary to follow some steps described in detail next. The first part of this section is the problem definition, trying to clearly establish the restrictions considered. Next, two ways to solve the proposed problem are presented: (1) the optimal power-of-two policy formulated as a nonlinear integer-programming problem, proposed by *Maxwell and Muckstadt (1985)*; and (2) the genetic algorithm approach using the power-of-two restrictions. These models are programmed using the computer programming language Borland C++, and codes are available in Appendix A.

The GA approach and the optimal power-of-two methodology, *Maxwell and Muckstadt (1985)*, modeled as nonlinear integer programming problems are compared to define the effectiveness of the genetic algorithm. The effectiveness is described as 100% times the ratio of the average cost over the traditional approach to the average cost of the GA approach. However, based on a single observation of a particular case it is not possible to reach conclusions about the effectiveness of the genetic approach. For that

purpose an experiment is conducted to explore this issue further, and is described in detail it section 3.5.

3.2. Problem Definition

An algorithm to determine the reorder cycle time in multi-stage serial and assembly systems (Figures 1.1.a, 1.1.b) is developed. The structure of the systems is limited to those that can be represented by an acyclic directed graph. Each node in the graph represents manufacturing, assembly or distribution operations, and the arcs indicate the flow of materials, components, subassemblies, assemblies, or finished product from one stage or operation to the next.

Demand for each end item is assumed to occur at a constant and continuous rate. Production is instantaneous and no backorders are allowed. Fixed setup costs and echelon holding costs are changed at each stage. The capacity at each node is unconstraint.

3.3. Maxwell and Muckstadt Approach

A power-of-two policy, as described by *Roundy (1986)*, is a sequence $T = (T_n: n \in N)$ of positive numbers with the following three properties. First, orders for product n are placed once every $T_n > 0$ units of time beginning at time zero. Second, $T_n = 2^{Kn} \beta$ for all products n and for some $1 \leq \beta < 2$, where K_n is an integer. Finally, the Zero-Inventory Property holds that an order is placed for a product only when the inventory of that product is zero.

Maxwell and Muckstadt (1985) presented a method for computing power-of two policy, based on the assumptions presented previously in the problem definition. Let G represents the acyclic directed graph corresponding to the production and distribution system. Let N (G) represents the node set and A (G) the arc set corresponding to G. The costs considered in the model are fixed setup costs A_i, for i Є N (G), and the echelon holding costs, h_i, for i Є N (G).

Let T_i for i Є N (G), represent the reorder interval at operation i and let T_L be the base planning period, measured in unit time (minutes, days, weeks, months, year, etc.). The reorder interval for each operation is expressed as a multiple of T_L. Let M_i for i Є N(G) represent the multiple of the base planning period per reorder interval for operation i. Also, for all i Є N (G), let $g_i = h_i\lambda_i/2$, the average echelon holding cost per unit time (the same unit time used to determine the demand) for operation i when $T_i = 1$. The model can be stated as:

$$\text{Minimize } \sum_{i \in N(G)} \left[\frac{A_i}{T_i} + g_i T_i \right]$$

Subject to:

$$T_i = M_i T_L, \qquad i \in N(G),$$

$$M_i \geq M_j, \qquad (i, j) \in A(G),$$

$$M_i = 2^{ki}, \qquad k=0,1,2,3,\ldots$$

This formulation is called Problem P. Problem P is a large-scale, nonlinear integer programming problem. In practical situations, the sets N (G) and A (G) could contain many thousands of elements. To solve Problem P they used a two step procedure. In the

first step they solved the relaxed version of this problem to establish what group of operations must have identical reorder intervals. The mathematical formulation of the relaxed problem, which is called Problem RP, replaces for each i Є N (G) the integrality constraint on M_i with the constraint $M_i \geq 1$, and replaces the requirement that $T_i \geq T_L$ with $T_i \geq 0$.

Problem RP (relaxed problem) is

$$\text{Minimize } \sum_{i \in N(G)} \left[\frac{A_i}{T_i} + g_i T_i \right]$$

Subject to:

$$T_i \geq T_j \geq 0 \qquad (i,j) \in A(G),$$

Jackson, Maxwell and Muckstadt (1988) showed the characterization of the optimal solution. They established the correspondence between the solutions of problem RP and ordered partitions of the graph G. Define a sub graph G' of the graph G to consist of a subset N (G') of the node set N (G) together with the associated arc set A (G') where (i,j) Є A (G') if and only if i Є N(G'), j Є N(G'), and (i,j) Є A (G). An ordered collection of sub graphs (G_1, G_2,…, Gn) of G is said to be ordered by precedence if for any $1 \leq p < q \leq$ N there does not exist a node j Є N (G_p) and a node j Є N (G_q) such that (i,j) Є A (G). That is, no node in N (G_p) precedes any node in N (G_q) if q > p. The collection of sub graph (G_1, G_2,…, Gn) forms an ordered partition of the graph G if

(a) the node subsets N (G_1), N (G_2),…, N (Gn) form a partition of the node set N (G), and

(b) the collection is ordered by precedence.

A directed cut of a sub graph G' is simply an ordered (binary) partition (G'^{-}, G'^{+}) of the sub graph G'. Suppose that the reorder intervals share a common value: $T_i = T$ for all i Є N (G'). Then the optimal value of T is given by:

$$T = \left[\frac{\sum_{i \in N(G)} A_i}{\sum_{i \in N(G)} g_i} \right]^{1/2}$$

Letting

$$A(G') = \sum_{i \in N(G)} A_i \quad \text{and}$$

$$g(G') = \sum_{i \in N(G)} g_i$$

Then T = (A (G')/ g (G')) $^{1/2}$.

The optimal solution of problem P can be found if the solution to problem RP is known. The optimal value ok M_i for i Є N (G) can be found by calculating

$$k = \left| -\log_2 T_L - \log_2 \sqrt{2} + \log_2 \{K(G')/g(G')\} \right| \qquad \text{(Equation 3.1)}$$

where $\lceil x \rceil$ is the smallest integer greater than or equal to x. Using this k_i the optimum M_i*'s* are obtained substituting k on $M_i = 2^{ki}$. More details are provided in *Maxwell and Muckstadt (1985)*.

3.4. Genetic Algorithm Approach

The genetic algorithm is a general method for solving "search for solutions" problems *Mitchell (1998)*. The idea is to efficiently find a solution to a problem in large space of candidate solutions.

The algorithm started with a set of solutions called population and each candidate solution is represented by a chromosome. An outline of the basic genetic algorithm is shown in Figure 3.2 and described in detail next:

1. *Start.* Generate a random population of *n* chromosomes.
2. *Fitness.* Evaluate the fitness of each chromosome in the population.
3. *New population.* Create a new population by repeating the following steps until the new population is complete.
    a. Select two parent chromosomes from a population according to their fitness (the better fitness, the higher chance to be selected).
    b. Crossover. With a crossover probability cross over the parents to form two new offspring. The idea is that the children should be a combination of their parents.
    c. Mutation. Alter the offspring at each locus (position in chromosome), based on the mutation probability and prevent falling into a local optimum.
    d. Place the resulting chromosomes in the new population.
4. *Selection*. Based on their fitness, select the *n* chromosomes to form the current population.

5. *Test*. If the end condition is satisfied, stop, and return the best solution in current population.

6. *Loop*. Go to step 2.

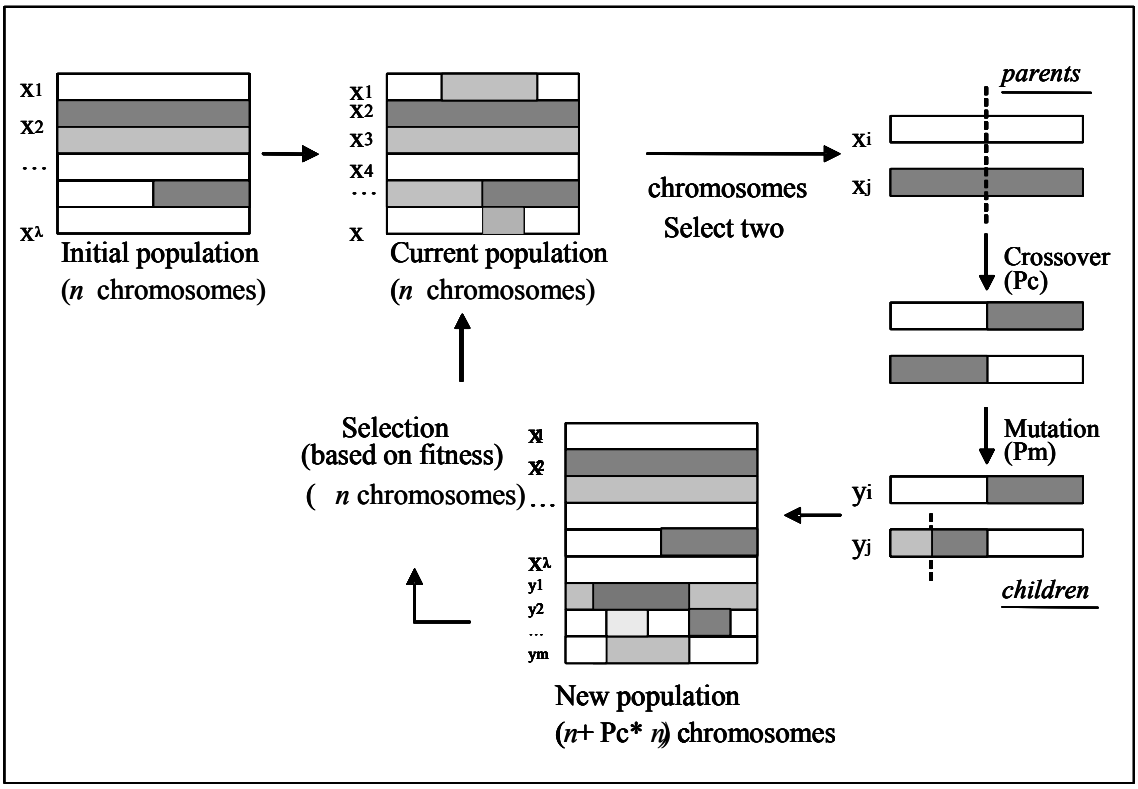


Figure 3.1 General GA procedure

The simple procedure just described is the basis for most applications of genetic algorithms. In the proposed GA, chromosomes represent (k_i's, $1 \leq i \leq n$) as shown in Figure 3.2.

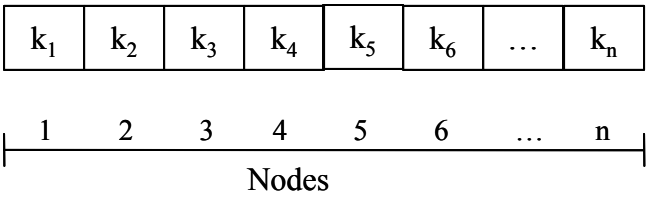


Figure 3.2 Chromosome representation

These variables are the exponents or power of two that function as multipliers of the basic period length to define the reorder cycle time for each period (i). It is necessary to define lower and upper bounds to these variables. The lower bound is defined (satisfying the restriction of $T_i \geq T_{i-1}$), as

$$k_i^{LB} = \max imum\{0, k_j\} \quad \ldots \text{ j successor(i)}$$

To define the upper bound, consider $k_i^{UB} = k_i^{LB} + 5$. This upper bound is used in the generation of the initial population used by GA for the given problem. After having a genetic representation of potential solutions it is needed to define a way to create an initial population of solutions.

3.4.1. Initial population

The initial population of individuals is generated randomly formed by *n* number of chromosomes, were *n* is the population size. The number of individuals in the population is considered one of the factors evaluated using experimentation.

The lower and upper bounds used to generate the *k's* that formed each individual in the population are described before as k_i^{LB} and k_i^{UB}. The initial population generated for this problem is composed: half by feasible solutions and the other half infeasible. This infeasibility consist of a violation of the nested ness constraint which establish $T_i > T_j$, for all j that is successor of i. For each chromosome defined a fitness value is assigned.

3.4.2. Fitness function

To evaluate the fitness of each individual in the population it is necessary to first convert the k_i in reorder cycle time using $T_i = 2^{ki} T_L$. This application of GA is function optimization, where the goal is to find a set of parameter values that minimize the objective function. The fitness function f (m) for every member of the population is defined as the inverse of the objective function.

$$f(m) = \frac{1}{\sum_{i \in N(G)} \left[\frac{A_i}{T_i} + g_i T_i \right]} \quad \text{(Equation 3.6)}$$

3.4.3. Genetic operators

There are two types of operators involved in the genetic algorithm proposed: mutation and crossover.

The mutation used is one point mutation, which selects randomly one point in the chromosome and changes the k value with another between zero and two, as shown in Figure 3.3.

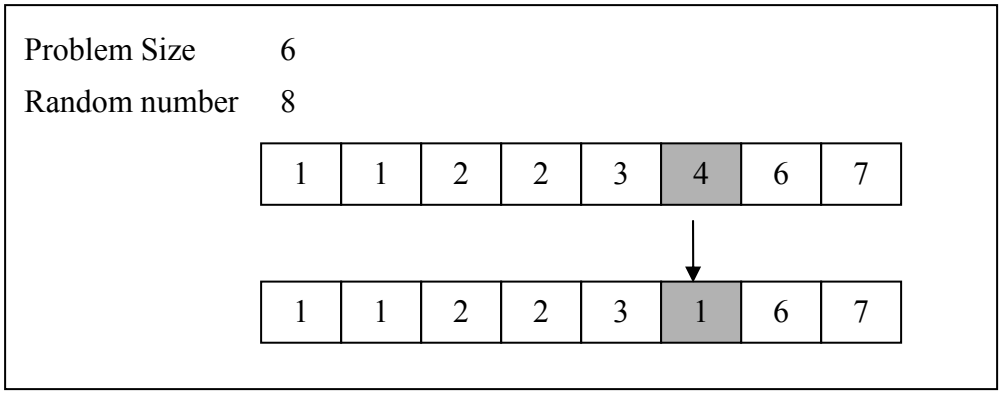


Figure 3.3 One point mutation

The crossover selected is the two-point crossover. Two chromosomes are selected randomly from a range zero to population size, to determine the position of the crossover points. This type of crossover combines the features of the two parents to form two offsprings, as illustrated in Figure 3.4.

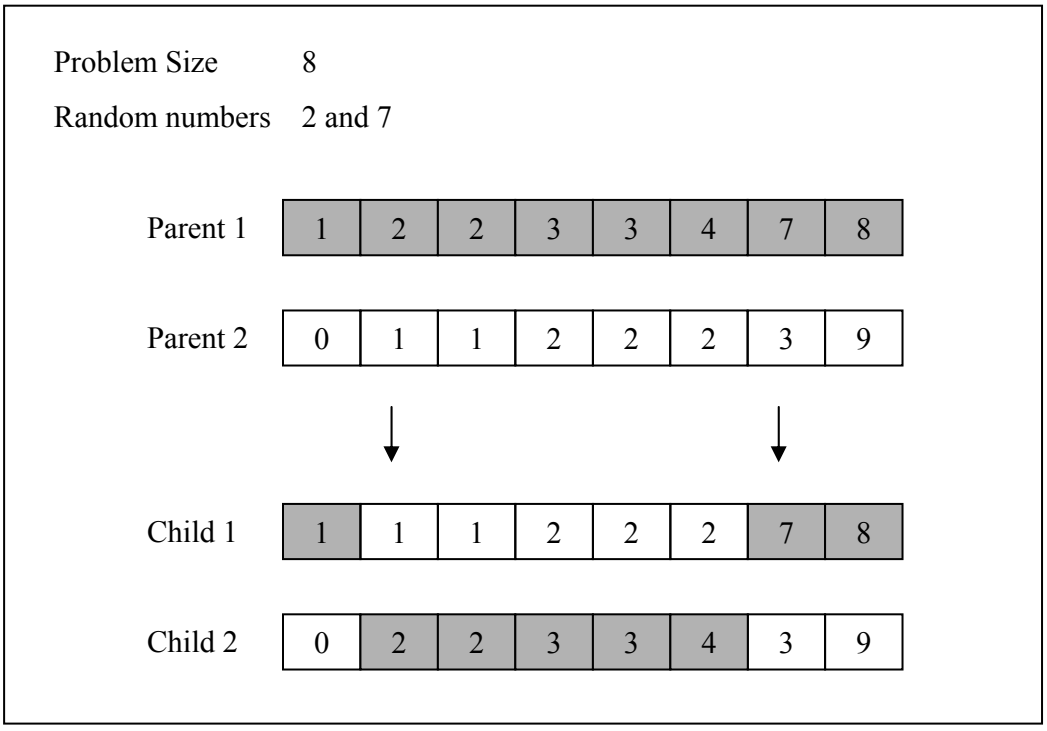


Figure 3.4. Two point crossover

3.4.4. Selection

Selection in general is a consequence of competition between individuals in a population. It is referred as the way that the individuals are selected to form the new population after each generation. The selection method used in the genetic algorithm developed is the tournament selection. Two chromosomes are selected at random and the best is kept and included in the new population.

Because crossover and mutation operations could generate infeasibility on the modified chromosome, a repairing technique is applied to guarantee that the nested ness constraints are not violated. This operation consists of an evaluation of each chromosome after selection to detect any violation to the previously defined restrictions. If it is required, changes k for a randomly generated number from the k value of the direct successor of the node where the violation occurs.

3.3.2. Example Problem

To illustrate the previously defined genetic algorithm, a simple example is presented. The example intends to show not only the step by step sequence of operations but also the application of the results in practice. For simplicity, only the most important details are provided. It is assume that the problem satisfy all the assumptions considered for the algorithm development.

The problem consists of a candy production line, which requires six principal processes. These processes consist of: (1) packaging; (2) mixing and coxing; (3) ordering the candies packages; (4) the ordering of imported sugar; (5) water purification and (6)

ordering artificial flavors and colorants. The problem data is included in Table 3.1, and its assembly structure is shown in Figure 3.5.

Table 3.1 Example problem data

Stage	$Demand_i$ (units/week)	Setup/ ordering $cost_i$	Echelon holding $cost_i$	g_i
1	10	20	1	5
2	20	20	2	20
3	10	20	1	5
4	40	20	1.5	30
5	20	20	1	10
6	10	45	1	5

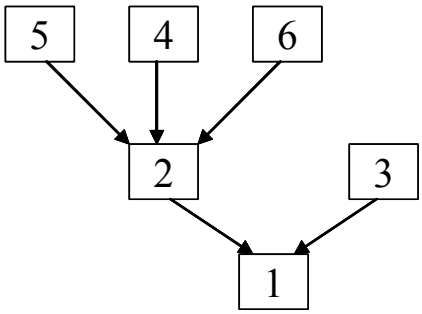


Figure 3.5 Example problem structure

a) G.A. parameters:

- Population Size = 4 chromosomes
- Generations = 1
- Probability of crossover = 50 %
- Probability of Mutation = 25%

b) Generation #1:

The genetic algorithm proposed is shown in Figure 3.6. using the parameters previously defined and the data in Table 3.2. In Figure 3.6, Rand represent the random numbers generated during the sequence. The initial population is formed by four chromosomes. Each one has a fitness value calculated using equation 3.6, and represents the inverse of the total cost function based on the objective of minimization. To apply the crossover operation, a random number is generated and compared with the probability of crossover; if the probability of crossover is bigger than the random number, then two chromosomes are selected to do the crossover, else continue to mutation.

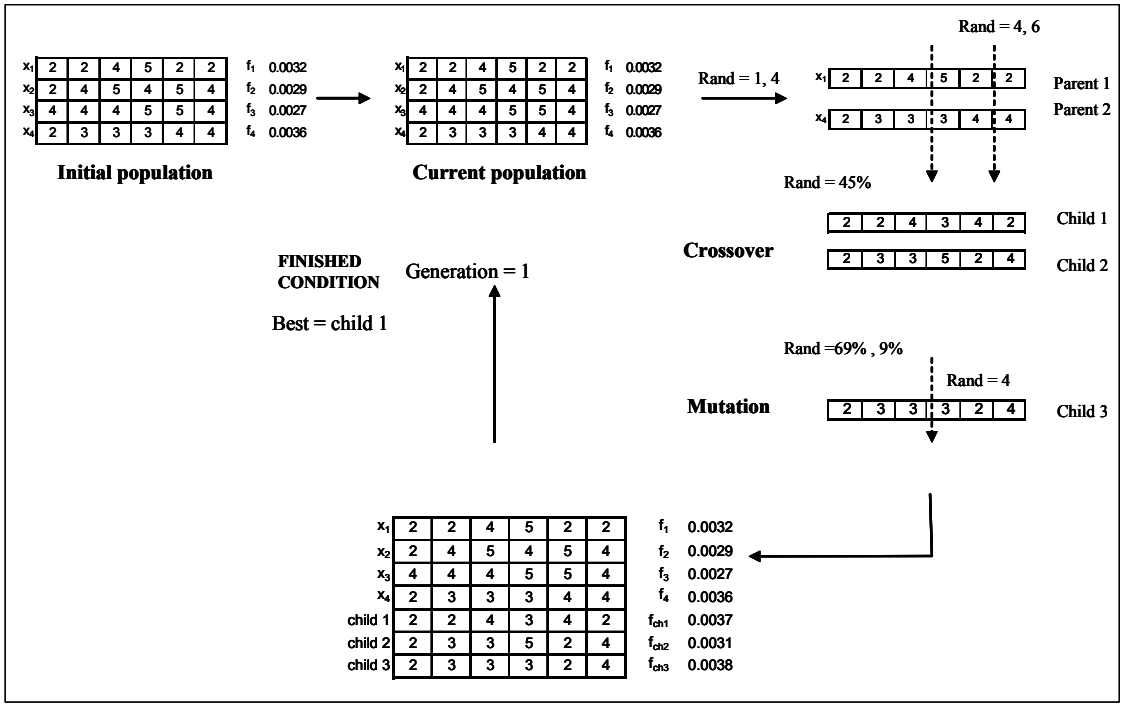


Figure 3.6 First generation of problem example using GA

In the present example, chromosomes 1 and 4 are selected randomly from the population and cross by chromosomes 4 and 6. After the crossover is completed,

mutation is executed as described in Section 3.4.3. Finally, the new population is formed by all chromosomes of the current population plus the children generated. This example defined as a terminal condition, the completion of one generation, and thus making the chromosome with the best fitness generated as our solution. The best solution is represented by the chromosome shown in Figure 3.7.

2	2	4	3	4	2

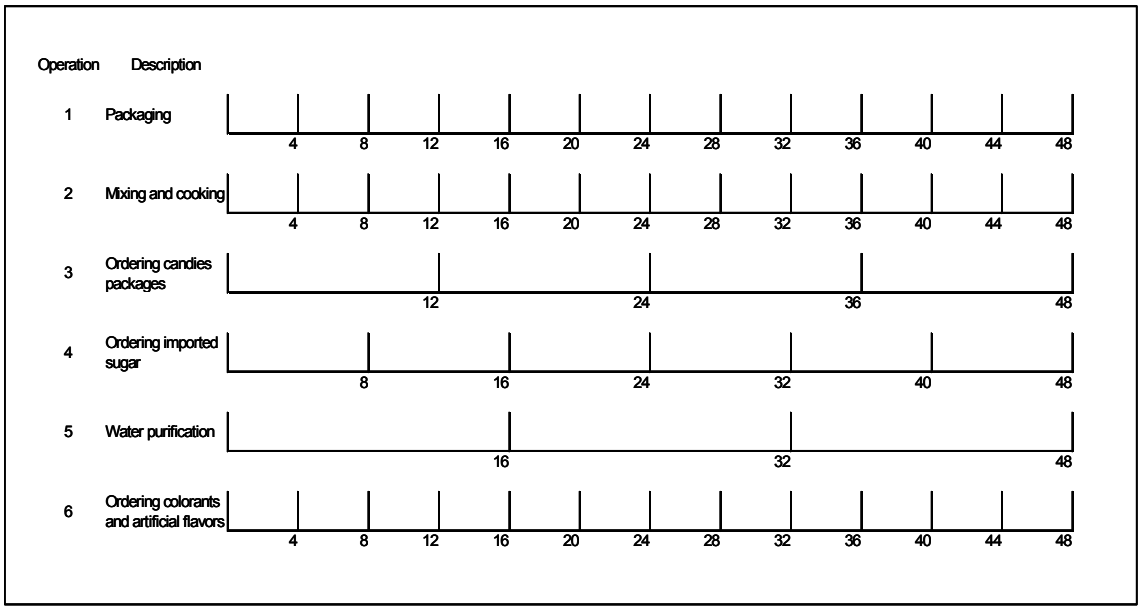
Total cost = US$ 269.17

Fitness = 0.0037

Figure 3.7 Solution to problem example

Once this solution is obtained, it can be applied as shown in Figure 3.8, using a timescale to help visualize the nested ness and stationary policies. First, transform the previous results as reorder cycle time (T_i) using $T_i = 2^{k_i}$, were k_i's are the output of G.A. For this problem the T_i's are: $T_1 = 4$, $T_2 = 4$, $T_3 = 16$, $T_4 = 8$, $T_5 = 16$, and $T_6 = 4$. In this example T_L is one week. For each operation the order size is equal to the demand, because the objective is to satisfy the demand requirements.

Every time one order is placed or a setup operation is done, it also has to be placed by all its predecessors. This is the practical meaning of the nested ness policy which established that the reorder cycle time for an operation have to be at least as large as it successor. The stationary policy is also satisfy because the order is placed in the same interval time and the quantity is always the same.



Note: TL is one week, which means that all time units are express in weeks

Figure 3.8 Example problem solution using GA

The previous example intends to demonstrate the real application of the methodology proposed. In the next section, the experimentation to compare both methodologies, Maxwell and Muckstadt and the genetic algorithm, is defined.

3.5. Experimental Design

Experimental design is a critically important tool in engineering world for improving the performance of manufacturing processes. Experimentation should be viewed as part of the scientific process and as one of the ways to learn how the systems

work. An experiment is conducted to: (1) to determine the best GA parameters settings; and (2) to determine the robustness of the proposed algorithm.

3.6.1. GA parameters in the experimental design

Response surface methodology is a collection of mathematical and statistical techniques that are useful for the modeling and analysis of problems in which a response of interest is influenced by several variables and the objective is to optimize this response. Central composite design (CCD) is a response surface method that allows one to keep the size and complexity of the design low and simultaneously obtain some protection against curvature, as described by *Montgomery (2001)*.

One important decision to make when implementing a genetic algorithm is how to set the parameters values. In order to satisfy this condition a central composite design is selected which consists of a 2^{5-1} design augmented with two center points in each block, ten axial and one center axial point, to obtain an indication of curvature and fit a second-order model if it is required.

Axial points have all of the factors set to the midpoint, except one factor, which has the value +/- alpha. The value for alpha is calculated in each design for both rotatability and orthogonality of blocks. In this design alpha is set to one, because some factor cannot assume values bigger than their upper bound, this is commonly referred to as a face-centered central composite design. This design only requires three levels for each factor. Center points, as implied by the name, are points with all levels set to the

midpoint of each factor range. Center points are usually repeated four to six times to get a good estimate of experimental error (pure error).

The parameters to be tested and their respective levels are: (1) Problem size, 10, 505 and 1000 nodes; (2) Probability of crossover, 0.5, 0.75 and 1.0; (3) Probability of mutation, 0.01, 0.255 and 0.50; (4) Population size, 30, 515 and 1000 chromosomes and (5) Number of generations, 50, 475 and 1000 generations. The demand, setup costs and holding costs were generated using a uniform distribution with the following upper and lower parameters 1-200, 5-500, 0.1-2, respectively (See Appendix B1). The levels of the factors evaluated are considered based on the typical values used in previous work done in similar applications and trying to include an extensive region of experimentation.

From this experiment is expected to obtain two responses: cost (measure in $/unit time) and time (measure in unit time). To optimize both responses is used an approach presented by *Artiles (1996)*, using standardize loss functions integrated with specification limits define for each factor. This method is easily implemented using a spreadsheet.

3.6.2. Robustness experimental design

After having the GA parameters that provides the best results, a 2^3 experiment is conducted to determine the robustness of the methodology proposed. The importance of the robustness of a methodology is that measures its ability to give the expected results in a variety of problems that can be implemented. In this research, the problem instances are changed by: (1) Setup or ordering costs, uniformly distributed (5-500) (low) and

uniformly distributed (500-1000) (high); (2) Holding costs, uniformly distributed (0.5-2) (low) and uniformly distributed (2-5) (high); and (3) Problem size, 10 and 2000.

As it was described in the GA parameters experiment, the responses were cost and time, and the demand generated is uniformly distributed between 1 and 200.

3.7. Conclusions

This chapter describes the characteristics of the problem solved and the two approaches compared: Maxwell and Muckstadt (1985) and the genetic algorithm developed.

Maxwell and Muckstadt define the problem as a large-scale nonlinear integer programming, and define a two step procedure to solve a relaxed version of the problem which identifies the operations that share the reorder interval. Having the solution to the relaxed problem, is evaluated the value of the reorder interval for each operation using Equation 3.1 (See Appendix A.1).

The genetic algorithm is a general procedure that could be applied to a variety of problems, combining some strategies to find the best solution in very complex search spaces. The genetic algorithm developed starts from a population were half of the solutions are infeasible trying to enrich the search space. The infeasibility consists of a violation of a nested ness constraint. The operators used are: a two-point crossover, one-point mutation and a repairing procedure to make bring an infeasible solution to the feasible region. The last technique shows to work very well in problems with similar

restrictions. The selection apply is the tournament selection which compares between two chromosomes.

Two experiments are designed. The first is a central composite deign to find the settings of the probability of crossover, probability of mutation, number of generations and population that optimize the solutions found with the genetic algorithm. The output of this experiment are cost and time express in percentages of the difference between the genetic algorithm proposed and Maxwell and Muckstadt. This optimization is done using the loss function combined with the specification limits proposed by *Artiles (1996)*.

Once the settings that optimize the genetic algorithm are found, a factorial experiment is used to evaluate how changes in the setup cost, holding cost and problem size change the genetic algorithm output. This is defined as the robustness of the methodology.

The next chapter presents the experiments and identify the more important characteristic of the solutions obtained.

CHAPTER IV

EXPERIMENTAL ANALYSIS

4.1. GA parameters results

To evaluate the GA parameters three different problem instances of size 10, 505 and 1000 nodes are generated, using the code shown in Appendix B.1. Setup costs, holding costs and demand are uniformly distributed between 5 and 500, 0.1 and 2, and 1 to 200, respectively, changing the problem size.

The summary results for the GA parameters determination experiment are shown in Table 4.1. The first two columns of Table 4.1 presents the standard and run order of the central composite design experiment conducted, while the third column shows at which block correspond each run. The next five columns identify the level of each factor considered in the experimentation part. The columns named cost and time show the results obtained during the experiment.

Cost and time are considered for determining the parameters of GA that give the best solution, considering important that GA gives a solution not only effective (near optimal) but also efficient (in less computational time).

Table 4.1 Summary GA results for parameters set up

Std. Order	*Run Order*	*Blocks*	*A: Problem Size*	*B: Pcross*	*C: Pmut*	*D: PopSize*	*E: Generations*	*Cost (US$)*	*Time (secs.)*
9	1	1	10	0.5	0.01	1000	50	2,927.39	0.37
22	2	1	505	0.75	0.255	515	475	187,296.91	109.31
3	3	1	10	1	0.01	30	50	3,067.84	0.03
2	4	1	1000	0.5	0.01	30	50	414,409.64	0.91
16	5	1	1000	1	0.5	1000	1000	382,567.75	1,134.22
8	6	1	1000	1	0.5	30	50	413,437.20	1.54
14	7	1	1000	0.5	0.5	1000	50	413,437.20	1.56
10	8	1	1000	0.5	0.01	1000	1000	407,315.30	660.58
1	9	1	10	0.5	0.01	30	1000	3,022.10	0.67
13	10	1	10	0.5	0.5	1000	1000	2786.50	8.56
17	11	1	505	0.75	0.255	515	475	187296.91	109.43
6	12	1	1000	0.5	0.5	30	1000	394700.86	24.01
21	13	1	505	0.75	0.255	515	475	187296.91	109.48
5	14	1	10	0.5	0.5	30	50	2925.15	0.03
15	15	1	10	1	0.5	1000	50	2792.50	0.60
20	16	1	505	0.75	0.255	515	475	187296.91	109.38
18	17	1	505	0.75	0.255	515	475	187296.91	109.35
11	18	1	10	1	0.01	1000	1000	2867.33	9.95
4	19	1	1000	1	0.01	30	1000	415136.76	28.36
19	20	1	505	0.75	0.255	515	475	187296.91	109.43
12	21	1	1000	1	0.01	1000	50	407365.63	46.09
7	22	1	10	1	0.5	30	1000	2786.50	0.76
32	23	2	505	0.75	0.255	515	1000	181568.90	229.71
26	24	2	505	1	0.255	515	475	188367.08	127.87
31	25	2	505	0.75	0.255	515	50	192915.93	11.41
24	26	2	1000	0.75	0.255	515	475	401511.99	217.32
27	27	2	505	0.75	0.01	515	475	193831.75	99.00
28	28	2	505	0.75	0.5	515	475	182171.38	118.17
25	29	2	505	0.5	0.255	515	475	187076.06	90.22
30	30	2	505	0.75	0.255	1000	475	193334.11	212.13
33	31	2	505	0.75	0.255	515	475	187296.91	108.91
29	32	2	505	0.75	0.255	30	475	191473.62	6.31
23	33	2	10	0.75	0.255	515	475	2649.00	2.25

In order to fit the data to a regression model, an independent analysis for each response, cost and computational time, is required. The first step is to identify significant factors in the model, for that purpose a multiple regression analysis and an analysis of

variance (ANOVA) are developed for both responses. The analysis of variance is a partition of the total variability into its component parts.

Table 4.2 Multiple regression analysis for cost

Dependent variable: Cost

Parameter	Estimate	Standard Error	T Statistic	P-Value
CONSTANT	6747.05	28813.0	0.234167	0.8170
Crossover	7644.11	86028.1	0.088856	0.9300
Generations	-4.54615	17.1844	-0.264551	0.7938
Mutation	-10737.6	30874.0	-0.347789	0.7313
Population	-21.6226	15.9019	-1.35975	0.1877
Problem	349.255	14.9878	23.3027	0.0000
Crossover^2	-6608.9	57243.4	-0.115453	0.9091
Generations^2	-0.00241235	0.0160467	-0.150333	0.8819
Mutation^2	-2216.77	59603.7	-0.0371918	0.9707
Population^2	0.0181496	0.0152097	1.19329	0.2455
Problem^2	0.0569161	0.0146014	3.89799	0.0008

Table 4.3 Analysis of variance for cost

Source	Sum of Squares	Df	Mean Square	F-Ratio	P-Value
Model	7.32221E11	10	7.32221E10	2322.95	0.0000
Residual	6.93465E8	22	3.15211E7		
Total (Corr.)	7.32914E11	32			

R-squared = 99.9054 percent
R-squared (adjusted for d.f.) = 99.8624 percent
Standard Error of Est. = 5614.37
Mean absolute error = 3264.4
Durbin-Watson statistic = 2.14129 (P=0.2962)
Lag 1 residual autocorrelation = -0.0976557

The output shows the results of fitting a model to describe the relationship between cost and the independent variables. The equation of the fitted model is as follows:

$$\begin{aligned}\text{Cost} = {} & 6747.05 + 7644.11 * \text{Pcross} - 4.55 * \text{Gens} - 10737.6 * \text{Pmut} - 21.62 * \text{Population} \\ & + 349.26 * \text{ProblemSize} - 6608.9 * \text{Pcross}\text{\^{}}2 - 0.002412 * \text{Gens}\text{\^{}}2 - 2216.77 * \text{Pmut}\text{\^{}}2 \\ & + 0.018 * \text{PopSize}\text{\^{}}2 + 0.0569 * \text{ProblemSize}\text{\^{}}2\end{aligned}$$

Since the P-value in table 4.3 is less than 0.01, there is a statistically significant relationship between the variables at the 99% confidence level. The R-Squared statistic indicates that the model as fitted explains 99.91% of the variability in cost. The adjusted R-squared statistic, which is suitable for comparing models with different numbers of independent variables, is 99.86%. The standard error of the estimates shows the standard deviation of the residuals to be 5,614.37. This value can be used to construct limits for new observations.

The mean absolute error (MAE) of 3,264.4 is the average value of the residuals. The Durbin-Watson (DW) statistic tests the residuals to determine if there is any significant correlation based on the order in which they occur. Since the P-value is greater than 0.05 there is no indication of significant autocorrelation in the residuals.

In determining whether the model can be simplified, notice that the highest P-value on the independent variables is 0.9707, belonging to mutation^2. Since the P-value is greater or equal to 0.10, that term is not statistically significant at the 90% or higher confidence level. Consequently, some factors are removed from the model. The final model is stated as:

Regression model for Cost:

$$\text{Cost} = 6149.02 - 7.03322 * \text{Gens} - 11868.2 * \text{Pmut} + 339.343 * \text{ProblemSize} + 0.0667298 * \text{ProblemSize}^2$$

(Equation 4.1)

To define the fit of the current equation including only the terms considered important, a multiple regression analysis and ANOVA are presented in Tables 4.4 and 4.5. A residuals analysis is included in Appendix C1.

Table 4.4 Multiple regression analysis for cost with significant factors

```
------------------------------------------------------------------------------
Dependent variable: Cost
------------------------------------------------------------------------------
                                     Standard          T
Parameter               Estimate       Error       Statistic        P-Value
------------------------------------------------------------------------------
CONSTANT                 6149.02        2626.27        2.34135         0.0266
Generations             -7.03322        2.63007       -2.67416         0.0124
Mutation                -11868.2        5104.63       -2.32499         0.0275
Problem                  339.343        8.06333        42.0848         0.0000
Problem^2              0.0667298     0.00758146        8.80171         0.0000
------------------------------------------------------------------------------
```

Table 4.5 Analysis of variance for cost with significant factors

```
------------------------------------------------------------------------------
Source              Sum of Squares     Df  Mean Square    F-Ratio      P-Value
------------------------------------------------------------------------------
Model                 7.32126E11      4   1.83031E11    6501.19       0.0000
Residual               7.88298E8     28    2.81535E7
------------------------------------------------------------------------------
Total (Corr.)         7.32914E11     32

R-squared = 99.8924 percent
R-squared (adjusted for d.f.) = 99.8771 percent
Standard Error of Est. = 5305.99
Mean absolute error = 3433.97
Durbin-Watson statistic = 2.40324 (P=0.0949)
Lag 1 residual autocorrelation = -0.225803
```

Tables 4.4 and 4.5, show the statistical significance of each variable as added to the model. For the response variable time, the same analysis is done and is presented in Tables 4.6 and 4.7.

Table 4.6 Multiple regression analysis for time

Dependent variable: Time

Parameter	Estimate	Standard Error	T Statistic	P-Value
CONSTANT	598.642	664.366	0.901073	0.3839
Crossover	-694.539	1900.73	-0.365407	0.7207
Generations	-0.479544	0.438935	-1.09252	0.2944
Mutation	-1551.0	806.882	-1.92222	0.0768
Population	-0.267428	0.407918	-0.655592	0.5235
Problem	-0.2662	0.388665	-0.684908	0.5054
Crossover*Mutatio	1715.37	501.581	3.41992	0.0046
Crossover^2	-4.50166	1252.94	-0.00359286	0.9972
Mutation^2	-12.3507	1304.61	-0.00946703	0.9926
Problem^2	0.00000187184	0.000319596	0.00585688	0.9954
Population^2	-4.52136E-7	0.000332911	-0.00135813	0.9989
Generations^2	-0.00000297021	0.000351231	-0.00845658	0.9934
Problem*Crossover	0.263354	0.248257	1.06081	0.3081
Crossover*Populat	0.26532	0.253376	1.04714	0.3141
Crossover*Generat	0.229557	0.258551	0.88786	0.3907
Mutation*Generati	0.276153	0.263828	1.04672	0.3143
Mutation*Populati	0.227004	0.258547	0.877999	0.3959
Mutation*Problem	0.219779	0.253324	0.867583	0.4014
Generations*Probl	0.000472566	0.000130581	3.61894	0.0031
Generations*Popul	0.000464772	0.000133274	3.48735	0.0040

Table 4.7 Analysis of variance for time

Source	Sum of Squares	Df	Mean Square	F-Ratio	P-Value
Model	1.37018E6	19	72114.9	4.78	0.0031
Residual	196317.0	13	15101.3		
Total (Corr.)	1.5665E6	32			

R-squared = 87.4678 percent
R-squared (adjusted for d.f.) = 69.1515 percent
Standard Error of Est. = 122.887
Mean absolute error = 55.392
Durbin-Watson statistic = 2.19784 (P=0.2396)
Lag 1 residual autocorrelation = -0.13205

For the multiple regression analysis of time response, originally all interactions are included as a result of previous experimentation that shows an R-Squared not bigger than 66% when the model only considered linear and quadratic terms. Including interactions terms the R-Squared statistic indicates that the model as fitted explains 87.46% of the variability in the response time.

The standard error of the estimate shows the standard deviation of the residual to be 122.887. The absolute value of the residuals is equal to the absolute value (55.39). There is no indication of serial autocorrelation in the residuals. The model requires a simplification, notice in the highest P-values belonging to some independent parameters. The simplified model is presented as follows with its statistical justification. A residuals analysis is included in Appendix C2.

Regression model for Time:

$$\text{Time} = 204.311 - 312.416 * \text{Pcross} - 0.2279 * \text{Gens} - 1118.96 * \text{Pmut} + 1715.37 * \text{Pcross} * \text{Pmut} + 0.0004607 * \text{Gens} * \text{Pr oblemSize} + 0.00045 * \text{Gens} * \text{PopSize}$$

(Equation 4.2)

Table 4.8 Multiple regression analysis for time with significant factors

Dependent variable: Time

Parameter	Estimate	Standard Error	T Statistic	P-Value
CONSTANT	204.311	116.933	1.74726	0.0924
Crossover	-312.416	146.198	-2.13695	0.0422
Generations	-0.227938	0.0736832	-3.09349	0.0047
Mutation	-1185.96	334.052	-3.55023	0.0015
Crossover*Mutatio	1715.37	424.913	4.03698	0.0004
Generations*Probl	0.000460701	0.0000722582	6.37575	0.0000
Generations*Popul	0.000452612	0.0000737481	6.13727	0.0000

Table 4.9 Analysis of variance for time with significant factors

Source	Sum of Squares	Df	Mean Square	F-Ratio	P-Value
Model	1.28472E6	6	214120.0	19.76	0.0000
Residual	281777.0	26	10837.6		
Total (Corr.)	1.5665E6	32			

R-squared = 82.0123 percent
R-squared (adjusted for d.f.) = 77.8613 percent
Standard Error of Est. = 104.104
Mean absolute error = 60.0485
Durbin-Watson statistic = 2.86107 (P=0.0048)
Lag 1 residual autocorrelation = -0.436087

The numerical optimization technique used was proposed by *Artiles (1996)*, using standardized loss functions integrated with specification limits defined for each factor. The results obtained are presented in Table 4.10.

In Table 4.10 the columns “min” and “max” represents the lower and upper specifications limits for each factor. The column called “Optimal Values” shows the optimal settings proposed by this optimization technique for multiple responses. Factors represented the factor included in the model. In this table factor A represent Problem Size, B is the probability of crossover, C is the probability of mutation, D the population size and D the number of generations. In columns Cost and Time, for each factor is necessary to define the coefficient calculated by the regression. As a result, the loss function is evaluated in the cell under “Objective function”. The objective is to minimize the loss function.

Table 4.10 Optimization spreadsheet parameters

Min	Values	Max	Factors	Cost	Time	
10	**10**	1000	**ProblemSize**	339.34	0	
0.5	**1**	1	**Pcross**	0	-312.42	
0.01	**0.5**	0.5	**Pmut**	-11868.2	-1185.96	
30	**30**	1000	**PopSize**	0	0	
50	**425.135**	1000	**Generations**	-7.03	-0.23	
	100		**ProblemSize^2**	0.07	0	
	0.5		**Pcross*Pmut**	0	1715.37	
	4251.35		**Gen*ProblemSize**	0	0.00046	
	12754.05		**Gen*PopSize**	0	0.00045	
	1		**constante**	6149.02	204.31	
			Value	626.621	66.55019	
			LSL	615	75	
			Target	625	80	
			USL	635	85	**Objective Function**
			Loss	0.026276	7.235898	**7.262173278**

The optimal settings for the GA parameters values obtained from the optimization technique applied are: 1.0 for the crossover probability, 0.50 for mutation probability, 30 chromosomes in the population size, and 425 generations. These parameters are used as settings for the genetic algorithm to be analyzed further. With those parameters the GA is run and their results compared with the ones offered by *Maxwell and Muckstadt (1985)* as shown in Table 4.11.

Table 4.11 Summary comparison of GA and Maxwell and Muckstadt

	M&M		GA		Comparison	
Problem Size	***Cost***	***Time***	***Cost***	***Time***	***% Cost***	***% Time***
10	2,667.77	0.09	2,786.50	0.46	4.45	412.22
505	163,381.47	14.32	190,979.58	6.61	16.89	(53.84)
1000	333,607.70	26.99	407,474.88	13.06	22.14	(51.62)

The comparison columns show the difference percentage wise between GA and Maxwell and Muckstadt results. This relationship is defined as shown in equations 4.3 and 4.4.

$$\%\,\text{cost} = \frac{\left(GA_{cost} - M\,\&\,M_{cost}\right)}{M\,\&\,M_{cost}} \quad \text{(Equation 4.3)}$$

$$\%\,\text{time} = \frac{\left(GA_{time} - M\,\&\,M_{time}\right)}{M\,\&\,M_{time}} \quad \text{(Equation 4.4)}$$

Results show that GA offered solutions 4.45 percent above optimum in a computational time of 0.46 seconds. The difference in percentages from GA and Maxwell and Muckstadt results in cost increase as the problem size, but this relationship is not necessarily linear. For large problems, these differences are on average 22 percent. The negative values in the time column indicate that GA work faster than the Maxwell and Muckstadt algorithm for medium (505 nodes) and large (1000 nodes) problems.

4.2. GA robustness experiment results

To evaluate the robustness of the results obtained from the GA algorithm a 2^3 factorial experiment is conducted and the summary results are shown in Table 4.8. This experiment intends to define the distance in cost of GA from the optimal solution changing some aspects of the original environment under the algorithm was tested.

Table 4.12 Summary GA results for robustness design

Blocks	*Setup cost*	*Holding Cost*	*Problem Size*	*GA cost*	*M&M cost*	*% difference*
1	-1	1	1	1,117,191.94	782,943.15	29.92
1	-1	-1	1	430,565.28	314,350.41	26.99
1	1	-1	1	646,809.34	500,868.38	22.56
1	-1	-1	-1	2,528.28	2,440.14	3.49
1	-1	1	-1	3,609.23	3,609.23	0.00
1	1	1	1	1,292,768.45	927,083.72	28.29
1	1	-1	1	649,632.37	503,724.26	22.46
1	-1	-1	1	451,589.18	312,267.80	30.85
1	1	1	1	1,328,031.11	931,729.15	29.84
1	1	1	-1	6,528.01	6,528.01	0.00
1	-1	1	1	1,082,920.22	744,381.48	31.26
1	1	-1	-1	5,330.89	5,330.89	0.00
1	-1	-1	-1	2,447.68	2,447.68	0.00
1	-1	1	-1	4,903.39	4,645.94	5.25
1	1	1	-1	7,531.10	7,531.10	0.00
1	1	-1	-1	3,940.45	3,940.45	0.00

From the previous table can be derived some conclusions about the consistent difference in percentages in cost between both methodologies, which appears to be greater for larger problem size (around 22 and 31 percent), while the optimum solution is achieve in the 75% of the times for small problems.

The main effects plots which define the relation between cost and, setup cost, holding cost and the problem size are shown in Figure 4.1. The problem size is the factor that most affect cost response as shown graphically in Figure 4.1.

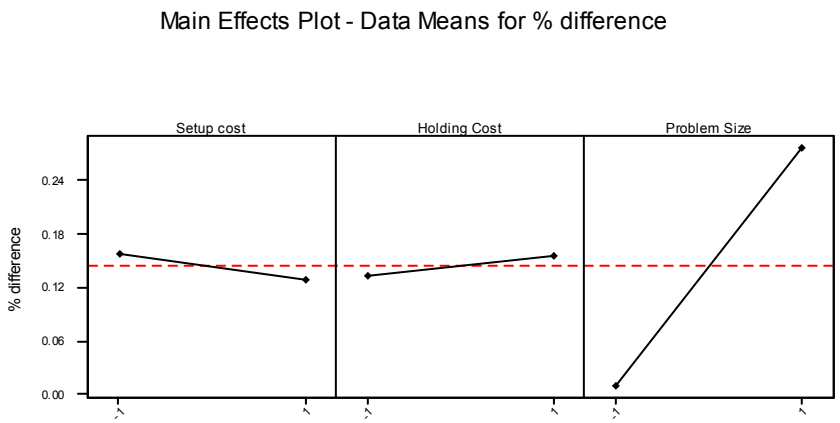


Figure 4.1 Main effects plots for cost

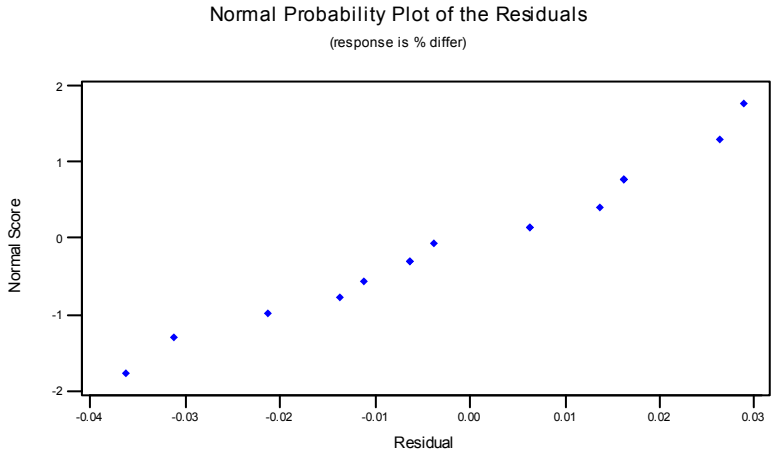


Figure 4.2 Normal probability plot of the residuals for robustness experiment

4.3. Conclusions

The first experiment conducted tried to set the values of some of the principal factor traditionally studied in GA that affects directly the results obtained. These factors are: probability of crossover, probability of mutation, population size and number of generations. The idea of this experiment is to define the parameters that offered the best cost or nearest optimal cost in little computational time. For that purpose, a multiple regression analysis and analysis of variance are developed to fit the data to an equation, considering only the factors that affect directly each response. These analyses are done independently for each response: cost and computational time and the regression models obtained are represented in Equations 4.1 and 4.2.

To optimize both responses an approach proposed by *Artiles (1996)* is used, as explained in section 3.6.1., and the results are:

- Probability of crossover: 100%
- Probability of mutation: 50 %
- Population size: 30 chromosomes
- Number of generations: 425 generations

To compare the outputs from GA and Maxwell and Muckastadt, the GA is evaluated for each problem size (small: 10 nodes and large: 1000 nodes), using the settings previously obtained. For the cost response, in the small problem size solutions four percent greater than the optimum are obtained, 17 percent for medium and 22 percent for large problems. For the computational time response, GA gives solutions more than 50% faster than Maxwell and Muckstadt for medium and large problems. For

small problems, Maxwell and Muckstadt gives solutions faster than GA, but this percentage is insignificant, only a difference of 0.28 seconds.

The second experiment identifies the robustness of the methodology proposed, based on its consistency in giving solutions with the same percentage differences from the optimal in difference problem instances. The parameters tested are setup and holding cost for each problem size. The experiment gives from zero to five percent near optimal solutions for small problems, and from 22 to 31% near optimal solutions for large problems. The optimal solutions in small problems are obtained 75% of the time.

CHAPTER V

CONCLUSIONS AND DISCUSSIONS

5.1. Conclusions

The main contribution of this research is the development of the first genetic algorithm approach to determine the reorder cycle time in multi-stage serial and assembly systems, considering the power of two restrictions. A genetic algorithm has not been addressed before in the literature to solve this problem.

Once the work is done, several questions have to be answered about the genetic algorithms which were formulated as secondary objectives of this research, and are related to its effectiveness, efficiency and robustness. The genetic algorithm developed shows to be effective, because it provides solutions in costs from zero to five percent above optimum for small problems and from 22 to 31 percent for large problems (1000 nodes). Optimal solutions are obtained 75 percent of the time for small problems.

The genetic algorithm is an efficient tool to find a near optimal solution in little time. The algorithm programmed in C++ and using a Pentium 4, provides solutions in less than one second for small problems, and less than 40 seconds for larger problems (1000 nodes).

An experiment is conducted to measure the ability of the genetic algorithm to provide solutions of the same quality in different problems instances. The experiment shows that GA gives very accurate solutions, comparing the results obtained in the first experiment after the optimization part with the ones in the robustness experiment. Using the optimal GA parameters, the result for small problems is four percent above optimal,

and from zero to five percent near optimal. For large problems, the results are 22.14 percent near optimal solution, while the robustness experiment shows an interval of 22 to 31 percent above optimal. It can be seen that for small problems the methodology is more accurate.

The flexibility is a very important characteristic of the genetic algorithm, making this methodology attractive to find a solution when the problem at hand does not satisfy all the assumptions stated in *Maxwell and Muckstadt (1985)*. It can even be extended to solve a different application with similar restrictions, like the one of finding the reorder cycle time for multi-stage serial and assembly systems without considering the power of two restrictions.

The GA developed can be easily extended to applications like the previously defined and for project planning and scheduling problems, because of the similarities in the chain structure and the precedence restrictions. The changes required will only include considering that each gene of the chromosome structure represent the reorder cycle time (T_i) of the corresponding period instead of the power of two, and eliminate the required transformation of the exponents in reorder cycle time. This is an advantage that cannot be obtained using the Maxwell and Muckstadt approach.

Another modification that can be done in the genetic algorithm is the inclusion of precedence restrictions to extend the methodology to solve problems with more general structures. The only modification required is to allow each stage to have more than one successor, including the successor's length in the structure defined as an structure called individual in the program.

The current problem can be extended considering capacity restrictions or joint cost. Some mathematical models have been developed considering these restrictions, facilitating a comparison between the genetic algorithm and their performance. A comparison between GA and another heuristics developed in this area, such as tabu search or simulated annealing, can be done to compare their performance.

REFERENCES

- Afentakis, Panayotis. "A Parallel Heuristic Algorithm for Lot-Sizing in Multi-stage Production Systems". *IIE Transactions*, 19 (1), 34-42. March 1987.

- Afentakis, Panayotis and Gavish, Bezalel. "Optimal Lot-Sizing Algorithms for Complex Product Structures". *Operations Research*, 34 (2), March-April 1986.

- Afentakis, Panayotis; Gavish, Bezalel and Karmarkar, Uday. "Computationally Efficient Optimal Solutions to the Lot-Sizing Problem in Multi-stage Assembly Systems". *Management Science*, 30 (2), 222-239. February 1984.

- Artiles Leon, Noel. "A Pragmatic Approach to Multi-Response Problems Using Loss Functions". Quality Engineering, 9(2), 213-220. 1996.

- Askin, Ronald G. and Goldberg, Jeffrey B. *Design and Analysis of Lean Production Systems*. John Wiley & Sons, Inc. New York, 2002.

- Axsäter, Sven. "Evaluation of Lot-Sizing Techniques". *International Journal Of Production Research*, 24 (1), 51-57. 1986

- Billington, Peter; Blackburn, Joseph; Maes, Johan; Millen, Robert and Wassenhove, Luk N. Van. "Multi-Item Lotsizing in Capacitated Multi-stage Serial Systems". *IIE Transactions*, 26 (2), 12-17. March 1994.

- Blackburn, Joseph D. and Millen, Robert A. "An Evaluation of Heuristic Performance in Multi-stage Lot-Sizing Systems". *International Journal Of Production Research*, 23 (5), 857-866. 1985.

- Blackburn, Joseph D. and Millen, Robert A. "Improved Heuristics for Multi-stage Requirements Planning Systems". *Management Science*, 28 (1), 44-56. January 1982.

- Bomberger, Earl E. "A Dynamic Programming Approach to a Lot Size Scheduling Problem". *Management Science*, 12 (11), 778-784, July 1966.

- Clark, Andrew J. and Scarf, Herbert. "Optimal Policies for a Multi-Echelon Inventory Problem". *Management Science*, 23, 475-490. 1960.

- Crowston, Wallace B.; Wagner, Michael and Williams, Jack F. "Economic Lot Size Determination in Multi-stage Assembly Systems". *Management Science,* 19 (5), 517-527. January 1973.

- Dixon, P. S. and Silver, E. A. "A Heuristic Solution Procedure for the Multi-item Single-level, Limited Capacity, Lot Sizing Problem". *Journal of Operations Management*, 2 (1), 23-29. October 1981.

- Dogramaci, A. Panayiotopoulos, J. E. and Adam N. R. "The Dynamic Lotsizing Problem for Multiple items under Limited Capacity". *AIIE Transactions*, 13 (4), 294-303. December 1981.

- Elmaghraby, Salah E. "The Economic Lot Scheduling Problem (ELSP): Review and Extensions". *Management Science*, 24 (6), 587-598. February 1978.

- Federgruen, Awi; Wagner, M. H. and Henshaw, A. "A Comparison of Exact and Heuristic Routines for Lot-Size Determination in Multi-stage Assembly Systems". *AIIE Transactions*, 4, 313-317. December 1972.

- Federgruen, Awi and Zheng, Yu-Sheng. "The Joint Replenishment Problem with General Joint Cost Structures". *Operations Research*, 40 (2), 384-403. March-April 1992.

- Federgruen, Awi; Queyranne, M. and Zheng, Yu-Sheng. "Simple Power-of-two Policies are Close to Optimal in a General Class of Production/ Distribution Networks with General Joint Setup Costs". *Mathematics of Operations Research,* 17 (4), 951-963. November 1992.

- Gen, Mitsuo and Cheng, Runwei. *Genetic Algorithms & Engineering Design.* Wiley Interscience Publication. Canada, 1997.

- Hernández, W. and Süer, Gürsel A. "Genetic Algorithms in Lot Sizing Decisions". Proceeding of the Congress on Evolutionary Computation. Washington, D.C. 1999.

- Jackson, Peter L.; Maxwell, William L. and Muckstadt, John A. "Determining Optimal Reorder Intervals in Capacitated Production/ Distribution Systems". *Management Science*, 34 (8), 938-958. August 1988.

- Jackson, Peter L.; Maxwell, William L. and Muckstadt, John A. "The Joint Replenishment Problem with Power-of-two Restrictions". *AIIE Transactions*, 17 (1), 25-32. March 1985.

- Khouja, Moutaz; Michalewicz, Zbigniew and Wilmot, Michael. "The Use of Genetic Algorithms to Solve the Economic Lot Size Scheduling Problem". *European Journal Of Operational Research*, 110 (1), 509-524, 1998.

- Lambrecht, M.R. and Vanderveken, H. "Heuristic Procedure for the Single Operation Multi-item Loading Problem". *AIIE Transactions*, 11 (4), 319-326. DECEMBER 1979.

- Maes, J. and Van Wassenhove, L.N. "A Simple Heuristic for the Multi-item Single-level Capacitated Lotsizing Problem". *Operations Research Letter*, 4 (6), 265-274. April 1986.

- Maxwell, William L. and Muckstadt, John A. "Establishing Consistent and Realistic reorder Intervals in Production/ Distribution Systems". *Operations Research,* 33 (6), 1316-1341. November-December 1985.

- Mitchell, Joseph S. B. "A 98% Effective Lot-Sizing for One-Warehouse, Multi-retailer Inventory Systems with Backlogging". *Operations Research*, 35 (3), 399-404. May-June 1987.

- Mitchell, Melanie. *An Introduction to Genetic Algorithms*. The MIT Press. Massachusetts, 1998.

- Montgomery, Douglas C. *Design and Analysis of Experiments*. Fifth Edition. John Wiley & Sons, Inc. New York, 2001.

- Roundy, Robin. "A 98% Effective Lot Sizing Rule for a Multi-product, Multi-stage Production/ Inventory System". *Mathematics of Operations Research*, 11 (4), 699-727. November 1986.

- Roundy, Robin. "A 98% Effective Integer-Ratio Lot Sizing for One-Warehouse Multi-Retailer Systems". *Management Science*, 31 (11), 1416-1430. November 1985.

- Schönsleben, Paul. *Integral Logistics Management. Planning ε Control of Comprehensive Business Processes*. The St. Lucie Press. Florida, 2000.

- Schwarz, Leroy B. "A Simple Continuous Review Deterministic One-Warehouse N-Retailer Inventory Problem". *Management Science*, 19 (5), 555-565. January, 1973.

- Schwarz, Leroy B. and Schrage, Linus. "Optimal and System Myopic Policies for Multi-Echelon Production/ Inventory Assembly Systems". *Management Science*, 21 (11), 1285-1294. July 1975.

- Silver, Edward A.; Pyke, David F. and Peterson, Rein. *Inventory Management and Production Planning and Scheduling.* John Wiley & Sons. New York, 1998, Third Edition.

- Simchi-Levi, David; Kaminsky, Philip and Simchi-Levi Edith. *Designing and Managing the Supply Chain. Concepts, Strategies, and Case Studies.* Irwin McGraw-Hill. United States of America, 2000.

- `Szendrovits, Andrew Z. "Comments on the Optimality in Optimal and System Myopic Policies for Multi-Echelon Production/ Inventory Assembly Systems". *Management Science*, 27 (9), 1081-1087. September 1981.

- Szendrovits, Andrew Z. "Manufacturing Cycle Time Determination for a Multi-stage Economic Production Quantity Model". *Management Science*, 22 (3), 298-308. November 1975.

- Veinott, Arthur F. Jr. "Minimum Concave-Cost Solution of Leontief Substitution Models of Multi-Facility Inventory Systems". *Operations Research,* 17 (2), 262-291. 1967.

- Williams, Jack F. "On the Optimality of Integer Lot Size Ratios in "Economic Lot Size Determination in Multi-stage Assembly Systems". *Management Science,* 28 (11), 1341-1349. November 1982.

- Zipkin, Paul H. *Foundations of Inventory Management*. McGraw-Hill. United States of America, 2000.

APPENDIX A

ALGORITHMS CODES

A.1 Maxwell and Muckstadt (1985) C ++ code

A.2 Genetic algorithm C++ code

A.1. Maxwell and Muckstadt (1985) C ++ code

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

#define InputFile "Problem_1005.txt"
#define OUTPUTFILE "Output_1005.xls"
# define ArrayBuffer 1005  //nodes ArrayBuffer = n

int TL = 1;

typedef struct CArray
{
  int Elements[ArrayBuffer];
  //each element is a Node... it is an array of nodes or a group of nodes
  int Length;
  int CannotCut;
}               CuttedArray;

CuttedArray aG[ArrayBuffer]; //contains x CuttedArray elements

typedef struct NArray
{
  int DirectPredecessors[ArrayBuffer];
  int DirectPredLength;
  int Predecessors[ArrayBuffer];
  int PredLength;

}               NodeArray;

NodeArray Nodes[ArrayBuffer]; //contains x CuttedArray elements

  //function prototyping
void ReadData();
double getColumnSum(int Col,CuttedArray aGroup);
int isDone();
void VG(CuttedArray *aVG);
float log2(double x);

//end function prototyping

float Log2TL = log2(TL);
float Log2Sqrt2 = log2(sqrt(2));

double Total_g=0;
int Length_aG=0;

CuttedArray OutputArray[2];
int Length_OutputArray;

float Matrix[ArrayBuffer][ArrayBuffer+4];
int n;
int minvalue;
int change;
int reference;
```

```
float TimeElapsed, PrintDelay, OperationTime;



void main()
{
  printf("\nPrograma corriendo...");
  clock_t TimeStart1, TimeEnd1, TimeStart2, TimeEnd2;
  float PrintDelay = 0;
  float OperationTime = 0;
  TimeStart1 = clock();

  CuttedArray nullArray;
  int i,j,kk,h;
  CuttedArray TempArray[ArrayBuffer];
  int Length_TempArray = 0;

  for(i=0;i<=ArrayBuffer;i++){nullArray.Elements[i] = 0;}
  //initialize nullArray
  nullArray.CannotCut = 0;
  nullArray.Length = 0;

  TimeEnd1 = clock(); //computing this time
  OperationTime += (float(TimeEnd1)-float(TimeStart1));

  //-------------------------Data--------------------------------
  //read the data and put it in Matrix[][]
        ReadData();

  //we already calculate g - it was on ReadData function


  //---------------------Partition-----------------------------
   TimeStart2 = clock();

   for(i=0;i<=n;i++)
   {
        Nodes[i].DirectPredLength = 0;
      Nodes[i].PredLength = 1;
   }

   for(i=0;i<=n;i++)
   {
      aG[0].Elements[i] = n-i;
      //set the first element of the aG array to be : n, n-1,n-2...1

      for (j=0;j<=n;j++)
      {
        if (Matrix[i][j+4] > 0)
         {
                Nodes[j].DirectPredLength += 1;
            Nodes[j].DirectPredecessors[Nodes[j].DirectPredLength - 1] = i;
            break;
         }
      }
   }

   aG[0].Length    = n+1;
   aG[0].CannotCut = 0;
```

```
   Length_aG = 1; //lenghts are on base 1... 1 means one element

   for (i=0;i<=n;i++)
   {
        // in this part it is included the reference node in its predecessors
      Nodes[i].Predecessors[0] = i;


       if ( Nodes[i].DirectPredLength > 0)
      {
        for (j = 1; j <= Nodes[i].DirectPredLength; j++)
                {
                Nodes[i].PredLength += 1;
           Nodes[i].Predecessors[j] = Nodes[i].DirectPredecessors[j-1];
         }
      }
      j = 1;

      while ( Nodes[i].Predecessors[j] > 0 )
      {
        for (kk = 0; kk < Nodes[Nodes[i].Predecessors[j]].DirectPredLength;
kk++)
         {
                Nodes[i].PredLength += 1;
           Nodes[i].Predecessors[Nodes[i].PredLength - 1] =
Nodes[Nodes[i].Predecessors[j]].DirectPredecessors[kk];
         }
         j++;

      }


   }//end for i

        //iterate to partitionate the array aG

   while(isDone() == 0)
   {
        //clean up the TempArray vector
      for(i=0;i<Length_TempArray;i++){TempArray[i] = nullArray;}

      Length_TempArray = 0;
      //this means the number of elements in the array-1

      for(i=0;i<Length_aG;i++)
      {
        if(aG[i].CannotCut == 0)
         {
                VG(&aG[i]); //gives an array with 1 or 2 elements in
OutputArray

            for(j=0;j<Length_OutputArray;j++)
            {
                TempArray[Length_TempArray] = OutputArray[j];
                                Length_TempArray += 1;
            }

                //clean up OutputArray
            for(h=0;h<Length_OutputArray;h++){OutputArray[h] = nullArray;}
```

```
         }else{

         //we cannot cut the array so we can pass it directly to the temp array
                TempArray[Length_TempArray] = aG[i];
                Length_TempArray += 1;
         }//end if


        } //end for i

       //assigning the temp array to our aG vector
       for(i=0;i<Length_TempArray;i++){aG[i] = TempArray[i];}
       Length_aG = Length_TempArray;

    }//end while

    double TotalSetup[ArrayBuffer],Totalg[ArrayBuffer];
    float T[ArrayBuffer], MiTL, k[ArrayBuffer], M[ArrayBuffer];
    double Cost[ArrayBuffer], TotalCost = 0;

    for(i=0;i<Length_aG;i++) //for each partition
    {
         TotalSetup[i] = 0;
       Totalg[i] = 0;

       for(j=0;j<aG[i].Length;j++) //for each node in partition
       {
         TotalSetup[i] += Matrix[aG[i].Elements[j]][1];
          Totalg[i] += Matrix[aG[i].Elements[j]][3];
       }

       k[i] = ceil(log2(TotalSetup[i] / Totalg[i]) - Log2Sqrt2 - Log2TL);

       if ( k[i] < 0 )
       {
         k[i] = 0;
       }

       M[i] = pow(2,k[i]);
       //calculate the Ti
       MiTL = M[i]*TL;
       for(j=0;j<aG[i].Length;j++)
       {
          T[aG[i].Elements[j]] = MiTL;
          Cost[aG[i].Elements[j]] = (Matrix[aG[i].Elements[j]][1] / MiTL) +
(Matrix[aG[i].Elements[j]][3] * MiTL);
          TotalCost += Cost[aG[i].Elements[j]];
       }
    }

    TimeEnd2 = clock();

    OperationTime += (float(TimeEnd2)-float(TimeStart2));

    //print the results
    FILE *Fout;
    Fout = fopen(OUTPUTFILE,"w+");//open the file for output
    fprintf(Fout,"<html><head><title>Results</title><style>td{text-
align:center;}</style></head><body>\n");
```

```
   //print the time
   TimeElapsed = (OperationTime - PrintDelay)/ CLOCKS_PER_SEC;
   fprintf(Fout,"<p>Time for completion:%.3f<br>TotalCost:%.2f<br>\n",
   TimeElapsed,double(TotalCost));
   fprintf(Fout,"<table border=1 cellpadding=2 cellspacing=0>\n");
   fprintf(Fout,"<tr><td colspan=6 style='font-weight:bold;text-
align:center;background:#C0C0C0;'>Results</td>\n");
   fprintf(Fout,"<tr style='text-align:center;'><td><b> </b></td>");
   fprintf(Fout,"<td><b>k<sub>i</sub></b></td>");
   fprintf(Fout,"<td><b>T<sub>i</sub></b></td>");
   fprintf(Fout,"<td><b>Setup Cost<sub>i</sub></b></td>");
   fprintf(Fout,"<td><b>g<sub>i</sub></b></td>");
   fprintf(Fout,"<td><b>Cost<sub>i</sub></b></td>");
   fprintf(Fout,"\n</tr>\n");
   for(i=0;i<=n;i++)
{               i         ki          Ti         Ai             gi         Costi

fprintf(Fout,"<tr><td>%d</td><td>%.0f</td><td>%.0f</td><td>%.2f</td><td>%.2f</t
d><td>%.2f</td></tr>\n",i+1,(log(T[i])/log(2)),T[i],float(Matrix[i][1]),float(M
atrix[i][3]),float(Cost[i]));

   fprintf(Fout,"<tr><td colspan=5 style'text-align:right;'><p
align=right><b>Total Cost:</b></td><td>%f</td></tr>\n",TotalCost);
   fprintf(Fout,"<tr><td colspan=5 style'text-align:right;'><p
align=right><b>Total Time:</b></td><td>%f</td></tr>\n",TimeElapsed);
   fprintf(Fout,"</table>\n");
   fprintf(Fout,"</body>");

   fclose(Fout); //close the file

        printf("\nFin del programa...");
}

/*********************************end main*******************************/

double getColumnSum(int Col, CuttedArray aGroup)
{
        double TotalSum=0;
        for(int i=0;i<aGroup.Length;i++)
   {
        TotalSum += Matrix[aGroup.Elements[i]][Col];
   }
        return TotalSum;
}

void ReadData()
{
        int i=0;
   int j;
   FILE *fin;
   clock_t TimeStart3, TimeEnd3;

   if ((fin=fopen(InputFile,"r"))==NULL)
   {
        printf("Warning! %s No data available..\n",fin);
      //getch();
      exit(1);
   }
   Total_g = 0;
```

```
   while(! feof(fin))
   {
        fscanf(fin,"%f\t%f\t%f",&Matrix[i][0], &Matrix[i][1], &Matrix[i][2]);

      TimeStart3 = clock();
      Matrix[i][3]=Matrix[i][0] * Matrix[i][2] / 2; //reserver the space for g
      Total_g += Matrix[i][3];

      TimeEnd3 = clock();
      OperationTime += (float(TimeEnd3)-float(TimeStart3));


      for(j=1;j<=ArrayBuffer;j++)
      {
        fscanf(fin,"%f",&Matrix[i][j+3]);
      }
      i++;
   }
   n=i-1;

   fclose(fin);
}

int isDone()
{
        for(int i=0;i<Length_aG;i++)
        { //if we cannot cut any of the arrays we already finish ,so the loop
is done.
    //if we find one that can be cut then we are not done so we return isDone =
false which is = 0
                if(aG[i].CannotCut == 0){return 0;}
        }

  //if we are here we didnt find an array that can be cutted, so we are done...
return isDone = true which is = 1
  return 1;
}

void VG(CuttedArray *aVG)
{
        double TotalSum;
        int Cutpoint, wasfound;
        float DG = getColumnSum(1,*aVG) /
             getColumnSum(3,*aVG);
        int i,j,i1;
        float MaxVG = -10000;
        int minvalue = n+1;
        int reference;
        int change;
        clock_t TimeStart4, TimeEnd4, TimeStart5, TimeEnd5, TimeStart6,
TimeEnd6;

        /******************start debugging******************/
        TimeStart4 = clock();
   printf("\nEntering in VG. Elements:\n");
        for(i=0;i<aVG->Length;i++){printf(" # %d # ",aVG->Elements[i]);}
   TimeEnd4 = clock();

        /*********************end debugging******************/
        PrintDelay += (float(TimeEnd4) - float(TimeStart4));
```

```
        for(i=0;i<aVG->Length;i++)
        {
        if(minvalue > aVG->Elements[i])
      {
        minvalue = aVG->Elements[i];
      }
        }

        //iterate to get the MaxVG... if MaxVG is positive we can cut...else we
can't



        Length_OutputArray = 2;

   for(i=0;i<aVG->Length;i++)
   {
        OutputArray[0].Length = 0;
      OutputArray[1].Length = 0;

      change = 1;
      reference = 0;

      for(i1=0;i1<Nodes[aVG->Elements[i]].PredLength;i1++)
      {
        if(Nodes[aVG->Elements[i]].Predecessors[i1] == minvalue )
         {
                change = 0;
           reference = 1;
           break;
         }
      }

      for(j=0; j<aVG->Length;j++)
      {
        for(i1=0; i1 < Nodes[aVG->Elements[i]].PredLength;i1++)
         {
             if(Nodes[aVG->Elements[i]].Predecessors[i1] == aVG->Elements[j])
           {
            OutputArray[change].Elements[OutputArray[change].Length] = aVG-
            >Elements[j];
            OutputArray[change].Length++;
           }
         }
      }

      for(i1=0;i1<aVG->Length;i1++)
      {
        wasfound = 0;
         for(j=0;j<OutputArray[change].Length;j++)
         {
                if(aVG->Elements[i1]==OutputArray[change].Elements[j])
           {
               wasfound = 1;
              break;
           }
         }
```

```
         //if the element wasn't found on the aVG array then we writeit on
OutputArray[1]
         if(wasfound==0)
         {
              OutputArray[reference].Elements[OutputArray[reference].Length]=
              aVG->Elements[i1] ;
              OutputArray[reference].Length++;
         }
        }

      TotalSum = 0;

      for(j=0;j<OutputArray[1].Length;j++)
      {
             TotalSum += (Matrix[OutputArray[1].Elements[j]][1] / DG) -
             Matrix[OutputArray[1].Elements[j]][3];
      }

      if(TotalSum > MaxVG)
      {
        MaxVG = TotalSum;
                Cutpoint = aVG->Elements[i];
      }

      TimeStart5 = clock();
      printf("\nCurrent MaxVg=%f",MaxVG);
      TimeEnd5 = clock();
      PrintDelay += (float(TimeEnd5) - float(TimeStart5));
   }

        if(MaxVG > 0)
        {
        //cut the array
      Length_OutputArray = 2; //because we cut it in 2
      OutputArray[0].Length = 0;
      OutputArray[1].Length = 0;

      change = 1;
      reference = 0;

      //cutpoint is the mark of where im going to cut the array, that is where
MaxVG occured
      for(i1=0;i1<Nodes[Cutpoint].PredLength;i1++)
      {
        if(Nodes[Cutpoint].Predecessors[i1] == minvalue )
         {
                change = 0;
           reference = 1;
           break;
         }
      }

      for(j=0; j<aVG->Length; j++)
      {
        for(i1=0; i1 < Nodes[Cutpoint].PredLength;i1++)
        {
             if(Nodes[Cutpoint].Predecessors[i1] == aVG->Elements[j])
             {
```

```
              OutputArray[change].Elements[OutputArray[change].Length] = aVG-
              >Elements[j];
              OutputArray[change].Length++;
              }
          }
        }

          OutputArray[0].CannotCut = 0;

          for(i=0;i<aVG->Length;i++)
          {
               wasfound = 0;
               for(j=0;j<OutputArray[change].Length;j++)
               {
                  if(aVG->Elements[i]==OutputArray[change].Elements[j])
                  {
                      wasfound = 1;
                      break;
                  }
          }

          if(wasfound==0) //if the element wasnt found on the aVG array then we
writeit on OutputArray[1]
          {

               OutputArray[reference].Elements[OutputArray[reference].Length]=
               aVG->Elements[i] ;
               OutputArray[reference].Length++;
          }
       }

          TimeStart6 = clock();
          printf("\nHubo corte!");
          TimeEnd6 = clock();

          PrintDelay = (float(TimeEnd6)-float(TimeStart6));

          }else{
       //no cut needed
          Length_OutputArray = 1;
       OutputArray[0].Length = 0;

       for(i=0;i<aVG->Length;i++)
       {
          OutputArray[0].Elements[OutputArray[0].Length] = aVG->Elements[i];
           OutputArray[0].Length++;
       }
       OutputArray[0].CannotCut = 1;
       printf("\nNO Hubo corte!");

    } // end if

// we return OutputArray and Length_OutputArray but they were included in this
function...
// as pointer parameters(variables by reference)
}

float log2(double x)
{
```

```
        return (log10(x) / log10(2));
}
```

A.2 Genetic algorithm C++ code

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

#define DATAINPUTFILE "Problem_10.txt" // to read on ReadData() function
#define INSTANCESINPUTFILE "Instances_33.txt" //to read the on Readinstances()
#define OUTPUTFILE "Output_33.xls" //format: Output_GA_InstanceNumber
#define NUMBEROFINSTANCES 1
#define NUMBEROFREPLICATIONS 1 // Number of replications
#define MAXPOP 515
#define MAXNODE 10
#define MAXGENERATIONS 1050
#define INFINITY 1000000000000.0

//typedef int chromosome[MAXNODE];
typedef struct ind{
int chrom[MAXNODE];
double fitness;
double totalcost;
float ProbSelection;
        }individual;

typedef struct node{
int DirectSuccesor;  //succesors[MAXNODE];
    }nodes;

nodes Node[MAXNODE];

float urand(float low, float upper);
void readinstdata();
void readdata();
void CreateInitialPopulation();
void PrintPopulation(individual Population[], int popStart, int popEnd);
void crossover_default(individual *child1, individual *child2);

int TL = 1;
void mutation_fixed(individual *child3);
void Repairing(individual *child);
int NumOfNodes;
int PopSize;
int NumberOfChildren;
int ReplicationCounter;
int InstanceCounter;

void computefitness(individual *ActualIndividual);
individual tournamentselect();

float  pcrossover;
float  pmutation;

float MaxiPop[NUMBEROFINSTANCES+1];
```

```
float MaxiNode[NUMBEROFINSTANCES+1];
float Probcrossover[NUMBEROFINSTANCES+1];
float Probmutation[NUMBEROFINSTANCES+1];
int LastChange_GenNumber;
int CurrentGeneration;

float setupcost[MAXNODE];
float demand[MAXNODE];
float holdingcost[MAXNODE];
float MATRIX[MAXNODE][MAXNODE];
float avecheloncost[MAXNODE];
float pcycletime;
float rcycletime[MAXNODE];
float OperationTime;
float PrintDelay;
float InitialTime;
float TimeElapsed;

FILE *fin, *finstances, *fout;

individual oldpop[MAXPOP+MAXPOP+MAXPOP]; //initial population (mu)
individual newpop[MAXPOP]; //population after selection (lambda)
individual Best, Best_Feasible; //

int main()
{
   clock_t TimeStart1, TimeEnd1, TimeStart2, TimeEnd2;

   InitialTime = 0;

   int i,i1;
   double besttotalcost;
   double bestfeasibletotalcost;
   float randomvalue;
   int iCurrentRow; //used to format the excel output
   individual child, child1, child2, child3;

   //randomize();
   //srand(21637913); //introduce a seed to replicate the same results

   readinstdata(); // for each instance read MaxiPop, MaxiNode, Probcrossover,
Probmutation,

   for(InstanceCounter = 1; InstanceCounter <= NUMBEROFINSTANCES;
InstanceCounter++)
   {
      PopSize    = MaxiPop[InstanceCounter];
      NumOfNodes = MaxiNode[InstanceCounter];
      pcrossover = Probcrossover[InstanceCounter];
      pmutation  = Probmutation[InstanceCounter];

      // open the output file and print instance information
      fout = fopen(OUTPUTFILE,"a");

      //row just for formatting
      if(InstanceCounter == 1)
      {
        fprintf(fout,"<table><tr><td style='background:#c0c0c0;font-
size:medium;'>Output file : GA</td></tr></table>\n");
      }
```

```
      //table that holds the whole page
      fprintf(fout,"<table border=2 cellpadding=0 cellspacing=1
style='border:2px outset black;'><tr><td>");

      fprintf(fout,"<table border=1 cellpadding=0 cellspacing=0>");
      fprintf(fout,"<tr><td style='text-align:center;font-
size:medium;background:#c0c0c0;;'>");
      fprintf(fout,"Instance <b>%d</b> of
<b>%d</b>",InstanceCounter,NUMBEROFINSTANCES);
      fprintf(fout,"     Pcross=%3f",pcrossover);
      fprintf(fout,"     Pmutation=%3f",pmutation);
      fprintf(fout,"     Max Gen=%d",MAXGENERATIONS);
      fprintf(fout,"</td></tr>");

      // 1) Get costs and relationship data
      readdata(); //demand, setupcost, holdingcost, MATRIX[i][j] showing
relationship between nodes

      for(ReplicationCounter = 1; ReplicationCounter <= NUMBEROFREPLICATIONS;
ReplicationCounter++)
      {

            TimeStart1 = clock();
         OperationTime = 0;
         PrintDelay = 0;
         TimeElapsed = 0;
         CurrentGeneration = 0;
         LastChange_GenNumber = 0;

         Best.totalcost = 0;
         Best_Feasible.totalcost = 0;
         Best.fitness = 0;
         Best_Feasible.fitness = 0;

        //set default of Best
        for(i=0;i<NumOfNodes;i++)
        {
         Best.chrom[i] = 10;
         Best_Feasible.chrom[i] = 10;

         besttotalcost = double(double(setupcost[i]/(TL*pow(2,Best.chrom[i])))
                       + double(avecheloncost[i]*pow(2,Best.chrom[i])));
         bestfeasibletotalcost =
double(double(setupcost[i]/(TL*pow(2,Best_Feasible.chrom[i])))
                              +
double(avecheloncost[i]*pow(2,Best_Feasible.chrom[i])));
                        Best.totalcost += besttotalcost;
         Best_Feasible.totalcost += bestfeasibletotalcost;

         Best.fitness += 1/besttotalcost;
         Best_Feasible.fitness += 1/bestfeasibletotalcost;

        }

        // 2) Create the initial population. Evaluation is included.
        CreateInitialPopulation();

        // Syntax: PrintPopulation(individual Population[], int popStart, int
popEnd). Debugging.
```

```
        TimeStart2 = clock();

        //PrintPopulation(oldpop, 0, PopSize);

        TimeEnd2 = clock();
        PrintDelay = (float(TimeEnd2)-float(TimeStart2));

        //Iterate until terminal condition(s) is(are) reached.

        while(CurrentGeneration < MAXGENERATIONS)
        {
                    NumberOfChildren = 0;

            // 3) Crossover
            for(i1=0; i1 < PopSize; i1++)
            {
               randomvalue = urand(0,1);
             if(randomvalue <= pcrossover)
               {
                  //syntax: void crossover_default(individual &child1,
individual &child2)
                  crossover_default(&child1, &child2);
                  NumberOfChildren = NumberOfChildren + 1;
                  oldpop[PopSize+NumberOfChildren]=child1;
                  Repairing(&oldpop[PopSize+NumberOfChildren]);
                  computefitness(&oldpop[PopSize+NumberOfChildren]);

                  NumberOfChildren = NumberOfChildren + 1;
                  oldpop[PopSize+NumberOfChildren]=child2;
                  Repairing(&oldpop[PopSize+NumberOfChildren]);
                  computefitness(&oldpop[PopSize+NumberOfChildren]);
               }

               // 4) Mutation
               randomvalue = urand(0,1);
               if(randomvalue <= pmutation)
               {
                  //syntax: mutation_default(individual &child3)
                  mutation_fixed(&child3);
                  NumberOfChildren = NumberOfChildren + 1;
                  oldpop[PopSize+NumberOfChildren]=child3;
                  Repairing(&oldpop[PopSize+NumberOfChildren]);
                  computefitness(&oldpop[PopSize+NumberOfChildren]);
               }
            }//end for i1

            // 5) Select individuals (new population) Tourselect();

            newpop[0] = Best_Feasible;

            for(i=1;i<PopSize;i++)
            {
                newpop[i] = tournamentselect();
            }

            for(i=0;i<PopSize;i++)
            {
                oldpop[i] = newpop[i];
            }
```

```
            // 6) Display current status on screen

            TimeStart2 = clock();
            clrscr();
            printf("\nInstance %d of %d",InstanceCounter, NUMBEROFINSTANCES);
            printf("\nGen %d of %d", CurrentGeneration, MAXGENERATIONS);
            printf("\nRep %d of %d", ReplicationCounter, NUMBEROFREPLICATIONS);
            printf("\nPopSize=%d  PCross=%f  PMut=%f",PopSize, pcrossover,
pmutation);
            printf("\nLast Gen changed: %d",LastChange_GenNumber);
            printf("\nBest_Feasible cost= %f", Best_Feasible.totalcost);
            //printf("\nBest cost= %f", Best.totalcost);

            TimeEnd2 = clock();
            PrintDelay = (float(TimeEnd2)-float(TimeStart2));

            //add generation
            CurrentGeneration++;

        }//end while

        // 7) Print results to file

        TimeEnd1 = clock();
        OperationTime +=(float(TimeEnd1) - float(TimeStart1));

        //TimeElapsed = (OperationTime + InitialTime - PrintDelay)/
CLOCKS_PER_SEC;
        TimeElapsed = (InitialTime + OperationTime - PrintDelay) / CLK_TCK;

        fprintf(fout,"<tr><td>");
        fprintf(fout,"Replication %d of %d", ReplicationCounter,
NUMBEROFREPLICATIONS);
        fprintf(fout,"     Gen=%d",CurrentGeneration);
        fprintf(fout,"    Last Gen changed=%d",LastChange_GenNumber);
        fprintf(fout,"<Table border=0 cellpadding=2 cellspacing=0>");
        fprintf(fout,"<tr><td><b>Total Cost=</b></td><td><b>%2f</b></td>",
Best_Feasible.totalcost);
        fprintf(fout,"<td rowspan=2>Chrom = ");

      for(i1=0;i1<NumOfNodes;i1++)
      {
        fprintf(fout,"%d   ",Best_Feasible.chrom[i1]);
      }

      fprintf(fout,"</td></tr>");
      fprintf(fout,"<tr><td><b>Total
Time=</b></td><td><b>%6f</b></td></tr></table>", TimeElapsed);
      fprintf(fout,"</td></tr>");

   } //end for ReplicationCounter

   //close the table of the replications
   fprintf(fout,"</table>");

   //write the summary
   fprintf(fout,"</td><td><table border=1 cellpadding=0 cellspacing=0
style='border-style:2px outset #c0c0c0;'>");
   fprintf(fout,"<tr><td colspan=2 style='text-align:center;font-
size:medium;background:#c0c0c0;'>Data</td></tr>");
```

```
   iCurrentRow = (InstanceCounter-1) * (4 + 3*(NUMBEROFREPLICATIONS-1)) + 4;
   fprintf(fout,"<tr style='font-
weigth:bold;background:#c0c0c0'><td>Cost</td><td>Time</td></tr>");

   for(i1=1; i1 <= NUMBEROFREPLICATIONS;i1++)
   {
       fprintf(fout,"<tr><td>=B%d</td><td>=B%d</td></tr>", iCurrentRow,
iCurrentRow+1);
      iCurrentRow = iCurrentRow + 3;
   }

   fprintf(fout,"</table>");//closes the data table

   //summary table
   fprintf(fout,"</td><td><table border=2 cellpadding=0 cellspacing=0
style='border-style:2px outset #c0c0c0;'>");
   fprintf(fout,"<tr><td colspan=5 style='text-align:center;font-
size:medium;background:#c0c0c0;'>Summary for Instance
%d</td></tr>",InstanceCounter);
   fprintf(fout,"<tr
style='background:#c0c0c0;'><td> </td><td>Mean</td><td>Std</td><td>Max</td
><td>Min</td></tr>");

   iCurrentRow = (InstanceCounter - 1) * (4 + 3*(NUMBEROFREPLICATIONS-1)) + 4;
   fprintf(fout,"<tr><td style='background:#c0c0c0;'><b>Total
Cost</td><td>=average(D%d:D%d)</td>", iCurrentRow,  iCurrentRow +
NUMBEROFREPLICATIONS - 1);
   fprintf(fout,"<td>=stdev(D%d:D%d)</td>",  iCurrentRow,  iCurrentRow +
NUMBEROFREPLICATIONS - 1);
   fprintf(fout,"<td>=max(D%d:D%d)</td>",  iCurrentRow,  iCurrentRow +
NUMBEROFREPLICATIONS - 1);
   fprintf(fout,"<td>=Min(D%d:D%d)</td></tr>",  iCurrentRow,  iCurrentRow +
NUMBEROFREPLICATIONS - 1);

   fprintf(fout,"<tr><td style='background:#c0c0c0;'><b>Total
Time</td><td>=average(E%d:E%d)</td>",  iCurrentRow,  iCurrentRow +
NUMBEROFREPLICATIONS - 1);
   fprintf(fout,"<td>=stdev(E%d:E%d)</td>",  iCurrentRow,  iCurrentRow +
NUMBEROFREPLICATIONS - 1);
   fprintf(fout,"<td>=max(E%d:E%d)</td>",  iCurrentRow,  iCurrentRow +
NUMBEROFREPLICATIONS - 1);
   fprintf(fout,"<td>=Min(E%d:E%d)</td></tr>",  iCurrentRow,  iCurrentRow +
NUMBEROFREPLICATIONS - 1);
   fprintf(fout,"</table>");//closes the summary table

   //close the table that holds the page
   fprintf(fout,"</td></tr></table>");

   //close the output file
   fclose(fout);

   } //end for InstanceCounter

   printf("\n\nDone :D");
   getch();
} //end main function

/***** FINISH *****************/

void readinstdata()
```

```
{
   int i=1;
   if((finstances=fopen(INSTANCESINPUTFILE,"r"))==NULL)
   {
      printf("Warning! %s No instances data available..\n",finstances);
      getch();
      exit(1);
   }

   while(! feof(finstances))
   {
      fscanf(finstances,"%f\t%f\t%f\t%f",&MaxiPop[i], &MaxiNode[i],
&Probcrossover[i],&Probmutation[i]);
      i++;
   }

   fclose(finstances);
}


void readdata()
{
   int i=0;
   int j;
   clock_t TimeStart2, TimeEnd2;

   if ((fin=fopen(DATAINPUTFILE,"r"))==NULL)
   {
      printf("Warning! %s No input data available..\n",fin);
      getch();
      exit(1);
   }

   while(! feof(fin))
   {
      fscanf(fin,"%f\t%f\t%f",&demand[i], &setupcost[i], &holdingcost[i]);

      TimeStart2 = clock();

      avecheloncost[i] = 0.5 * demand[i] * holdingcost[i];

      TimeEnd2 = clock();

      InitialTime += (float(TimeEnd2)-float(TimeStart2));

      for(j=0;j<NumOfNodes;j++)
      {
        fscanf(fin,"%f",&MATRIX[i][j]);
      }
      i++;
   }

   fclose(fin);

   //predecesors calculation wa

   TimeStart2= clock();

   for(i=0;i<NumOfNodes;i++)
   {
```

```
        for (j=0;j<NumOfNodes;j++)
       {
        if (MATRIX[i][j]>0)
         {
            Node[i].DirectSuccesor = j;
            break;
         }
       }
   }

   Node[0].DirectSuccesor = 0;

   TimeEnd2 = clock();
   InitialTime += (float(TimeEnd2)-float(TimeStart2));
}


void CreateInitialPopulation()
{
   int i,j,end;
   float Lowk;

   end = (int)(PopSize /2);

   for(j=0;j<end;j++)
   {
      for(i=0;i<NumOfNodes;i++)
      {
        if (i == 0)
         {
             oldpop[j].chrom[i] = urand(0, 5);
         }else{
             Lowk = oldpop[j].chrom[Node[i].DirectSuccesor];
            oldpop[j].chrom[i] = (int)(urand(Lowk ,Lowk + 5));
         }
      }

      computefitness(&oldpop[j]);

   }

       for (j=end;j<PopSize;j++)
   {
        for(i=0;i<NumOfNodes;i++)
        {
                oldpop[j].chrom[i] = urand(0 , 5);
        }
        computefitness(&oldpop[j]);
   }
}

void PrintPopulation(individual Population[], int popStart, int popEnd)
{
   int i,j;
   FILE *fout_pop;

   fout_pop=fopen("initpop.html","a");
   fprintf(fout_pop,"\n<hr><hr><hr>\n"); //write a separation line

   for(i=popStart;i<popEnd;i++)
```

```
   {
      fprintf(fout_pop,"<br>%d) Chrom = ",i);
      for(j=0;j<NumOfNodes;j++)
      {
        fprintf(fout_pop,"%d %s",Population[i].chrom[j],",");
      }
      fprintf(fout_pop,"Fitness = %f \n",Population[i].fitness);
   }
   fclose(fout_pop);
}

void computefitness(individual *ActualIndividual)
{
   int i;
   float po2;
   double totalcost = 0;
   double currentcost;
   double fitness = 0;

   for(i=0;i<NumOfNodes;i++)
   {
      po2 = pow(2,ActualIndividual->chrom[i]);
      pcycletime = TL*po2;
      currentcost = double(double(setupcost[i]) / pcycletime
                           + double(avecheloncost[i]) * pcycletime);
      totalcost += currentcost;
      fitness += 1 / currentcost;
   }

   ActualIndividual->totalcost = totalcost;
   ActualIndividual->fitness = fitness;

   //Best_Feasible
   if(totalcost < Best_Feasible.totalcost)
   {
      for(i=0;i<NumOfNodes;i++)
      {
        Best_Feasible.chrom[i] = ActualIndividual->chrom[i];
      }
      Best_Feasible.totalcost = totalcost;
      Best_Feasible.fitness = fitness;

   //store when was last change of the feasible Best
   LastChange_GenNumber = CurrentGeneration;
   }
}

void crossover_default(individual *child1, individual *child2)
{
   int i, sd, jcross1;
   individual parent1,parent2;

   sd=(int)urand(0,PopSize);
   parent1 = oldpop[sd];
   parent2 = tournamentselect();

   jcross1=(int)urand(1,(int)NumOfNodes);

   for (i=0;i<NumOfNodes;i++)
   {
```

```
      if(i>=0 && i<jcross1)
      {
        child1->chrom[i]=parent2.chrom[i];
        child2->chrom[i]=parent1.chrom[i];
      }else{
        child1->chrom[i]=parent1.chrom[i];
        child2->chrom[i]=parent2.chrom[i];
      }
   }
}

void mutation_fixed(individual *child3)
{
   int i;
   int Ksubs, Lowsubs;
   int RandomIndividual = (int)urand(0,PopSize);
   int cutpoint;

   cutpoint = (int)urand(0,NumOfNodes);

      if(cutpoint == 0)
      {
        Ksubs = urand(0 , 2);
      }else{
        Lowsubs =
oldpop[RandomIndividual].chrom[Node[cutpoint].DirectSuccesor];
        Ksubs = (int)urand(Lowsubs, Lowsubs + 2);
      }

      oldpop[RandomIndividual].chrom[cutpoint] = Ksubs;

   for(i=0;i<NumOfNodes;i++)
   {
      child3->chrom[i]= oldpop[RandomIndividual].chrom[i];
   }
}

void Repairing(individual *child)
{
   int i , Low;

   for(i=1;i<NumOfNodes;i++)
   {
       if(child->chrom[i]<child->chrom[Node[i].DirectSuccesor])
      {
         Low = child->chrom[Node[i].DirectSuccesor];
       child->chrom[i] = urand(Low,Low + 5);
      }
   }
}

individual tournamentselect()
{
   int i,j;
   individual selected;
   selected.fitness = INFINITY;

   for(i=0;i<2;i++)
   {
      j = (int) urand(0,PopSize);
```

```
      if(oldpop[j].fitness < selected.fitness)
      {
        selected = oldpop[j];
      }
   }
   return selected;
}

float urand(float lower, float upper)
{
   float u;
   u=(float)(lower+(upper-lower)*( (float) rand()/(float) RAND_MAX) );
   if (u==lower || u==upper)
   {
        u=urand(lower,upper);
   }
   return u;
}
```

APPENDIX B

PROBLEM INSTANCES

B.1.Problem data generator code (Matlab 6.5)

B.2 Example problem

B.1 Problem Data Generator code (Matlab 6.5)

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%% DATA INPUTS %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%Problem Size
%ProblemSize = input('Problem Size: ');
ProblemSize = 5000;

%Demand parameters
%min_demand = input('Minimum demand parameter: ');
min_demand = 1;

%max_demand = input('Maximum demand parameter: ');
max_demand = 200;

%Setup cost parameters
%min_setup = input('Minimum Setup costs parameter: ');
min_setup = 5;

%max_setup = input('Maximum Setup costs parameter: ');
max_setup = 500;

%Holding cost parameters
%min_holding = input('Minimum Holding costs parameter: ');
min_holding = 1;

%max_holding = input('Maximum Holding costs parameter: ');
max_holding = 2;

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%% CODE %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

filename = ['Problem_' num2str(ProblemSize) '.txt'];

demand = min_demand + rand(ProblemSize,1)*(max_demand - min_demand);
SetupCost = min_setup + rand(ProblemSize,1)*(max_setup - min_setup);
HoldingCost = min_holding + rand(ProblemSize,1)*(max_holding - min_holding);

% Relationship Matrix
Matrix = zeros(ProblemSize);

% i represent the row number and j the column number
p = rand(ProblemSize,1);
for i=1:ProblemSize
     Probability = 1/(i-1);
    for j=1:ProblemSize
        if j < i
            if  p(i,1) < Probability*j %&& sum(Matrix(i,:),1) == 0
                Matrix(i,j) = 1;
                break; %sale del for j
            end %if
        end %if
    end %for j
    r=i
end %for i


%%%%%%%%%%%%%%%%%%%%%%%%%%   write to a file %%%%%%%%%%%%%%%%%%%%%%%%%%%%%
output = fopen(filename,'w');

for i=1:ProblemSize
```

```
    fprintf (output, '%.0f\t %.0f\t %.2f\t', demand(i,1), SetupCost(i,1),
HoldingCost(i,1));

    for j=1:ProblemSize
        if j < ProblemSize
            fprintf (output, '%.0f\t' , Matrix(i,j));
        elseif i == ProblemSize
            fprintf (output, '%.0f' , Matrix(i,j));
        else
            fprintf (output, '%.0f\n' , Matrix(i,j));
        end
    end
end

fclose (output);
disp('Fin del proceso')
```

B.2. Example Problem

```
Problem Size: 10

Demand        Setup cost    Holding cost

176           360           1.13
168           411           1.15
133           285           1.78
46            233           1.13
118           420           1.23
110           224           1.18
62            249           1.91
136           432           1.51
2             36            1.50
9             89            1.75


Relationship Matrix

0      0      0      0      0      0      0      0      0      0
1      0      0      0      0      0      0      0      0      0
0      1      0      0      0      0      0      0      0      0
0      0      1      0      0      0      0      0      0      0
0      0      0      1      0      0      0      0      0      0
1      0      0      0      0      0      0      0      0      0
0      1      0      0      0      0      0      0      0      0
0      0      0      1      0      0      0      0      0      0
1      0      0      0      0      0      0      0      0      0
0      0      0      0      0      0      1      0      0      0
```

It is not possible to show the data of problem examples with 1005 and 2000 nodes, because of the limitations on space, but they can be generated using the problem data generator code shown in Appendix B1.

APPENDIX C

RESIDUALS ANALYSIS

C.1 Residuals analysis for cost response in the GA parameters experiment

C.2 Residuals analysis for time response in the GA parameters experiment

C.1 Residuals analysis for cost response in the GA parameters experiment

To verify randomness is presented a plot of residuals versus the order of the data in Figures C.1. It does not show a tendency in the data, considering that this condition is satisfied.

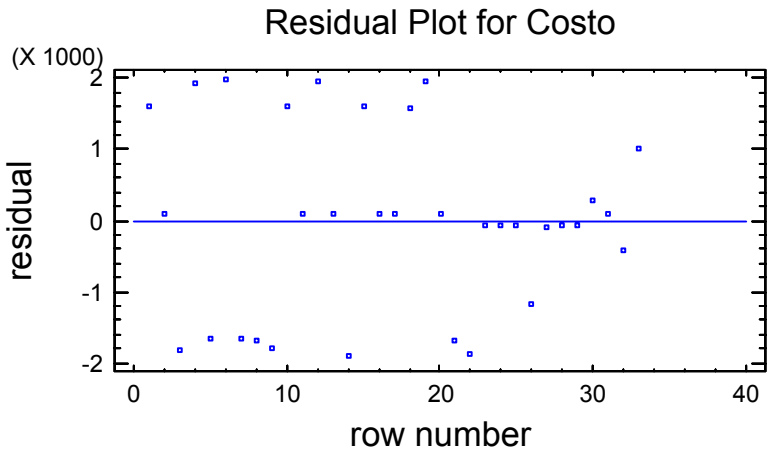


Figure C.1 Residuals plot for cost versus the order of the data

Another assumption is the equal variance. Only the most important factors considered in the regression model are included.

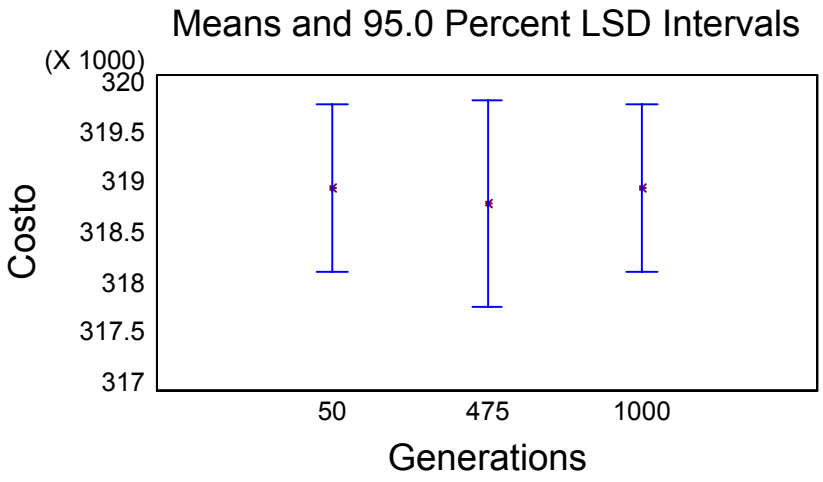


Figure C.2 Mean and 95 percent intervals plot for cost versus generations

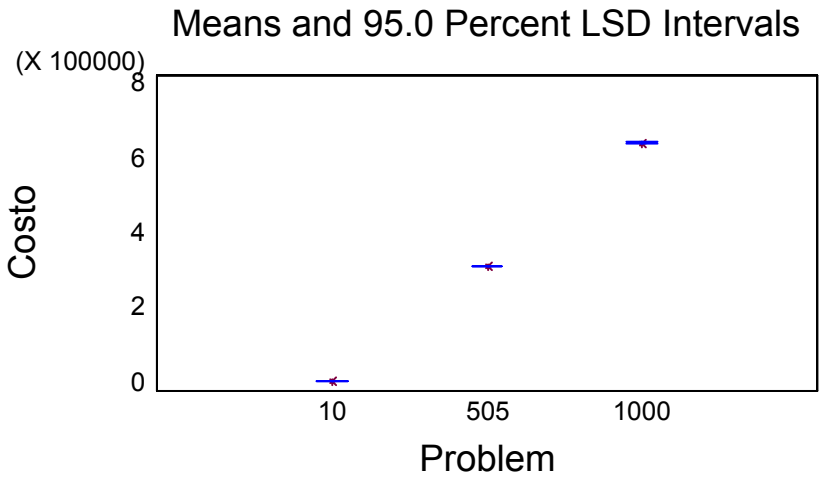


Figure C.3 Mean and 95 percent intervals plot for cost versus problem size

C.2 Residuals analysis for time response in the GA parameters experiment

The same analysis done for cost response is shown for time response.

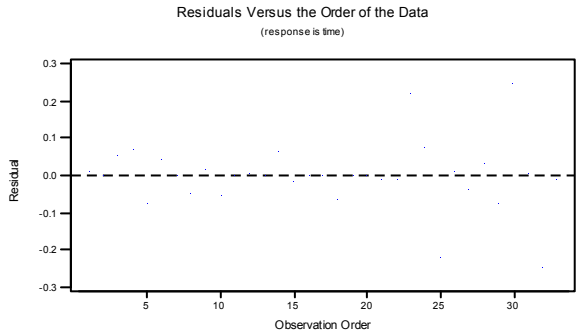


Figure C.4 Residuals plot for time versus the order of the data

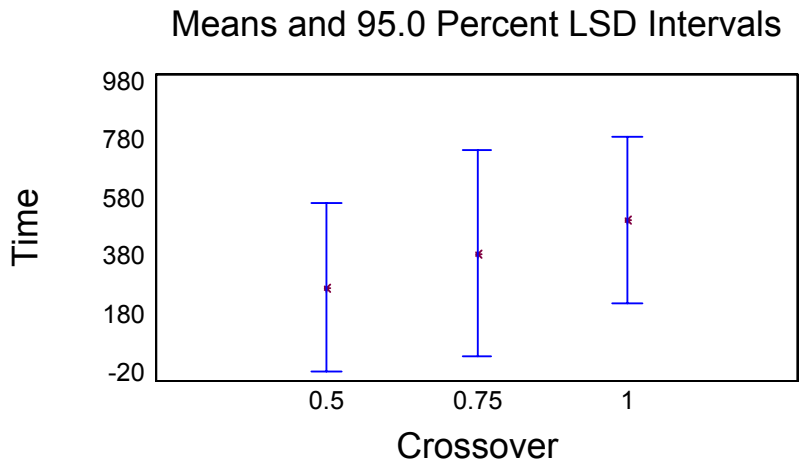


Figure C.5 Mean and 95 percent intervals plot for time versus crossover

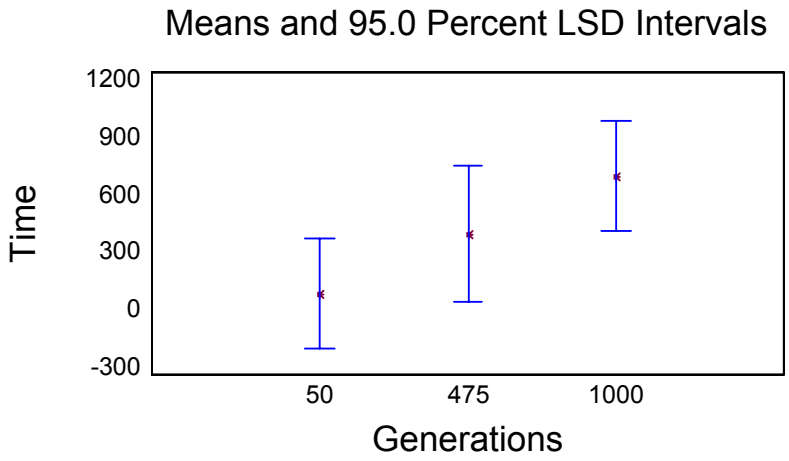


Figure C.6 Mean and 95 percent intervals plot for time versus generations

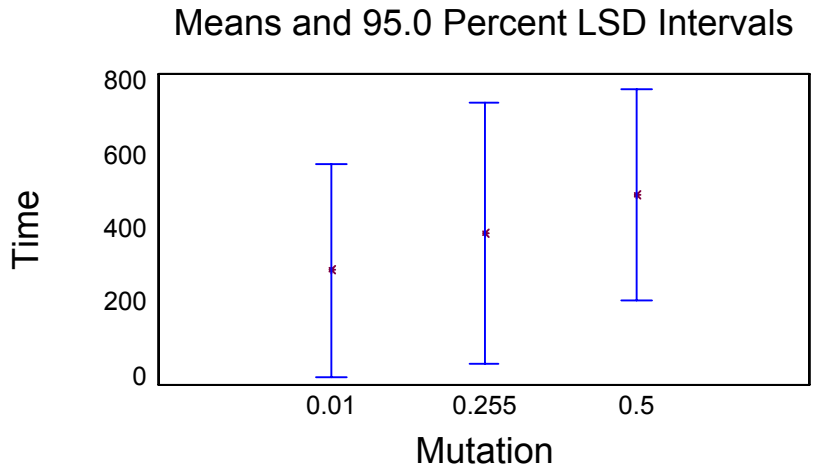


Figure C.7 Mean and 95 percent intervals plot for cost versus mutation

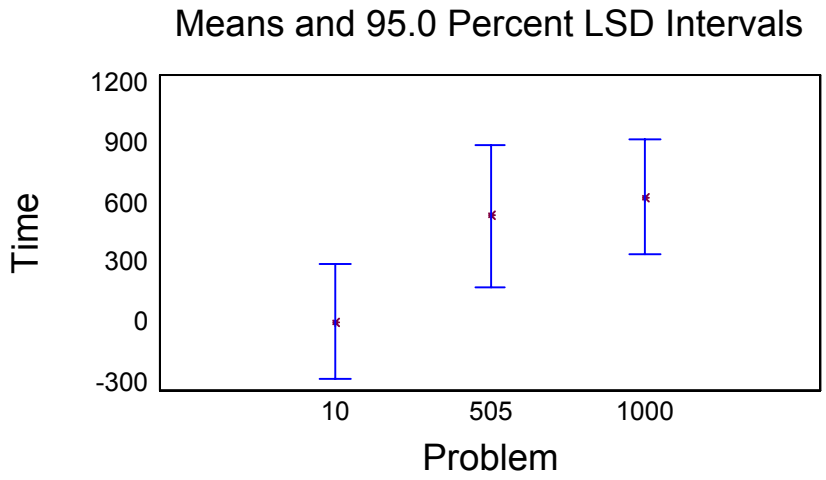


Figure C.8 Mean and 95 percent intervals plot for time versus the problem size