Master Thesis
Academic Year 2016-2017

How to deal correctly with Lead Time in General Supply Chains

Optimizing Safety Stock Placement under the Guaranteed Service Model approach

Maria Pilar Martín Romero
Student ID: 91655642

Supervisor:
Prof. Hiroaki Matsukawa

March 2017

Keio University
Graduate School of Science and Technology
School for Open and Environmental System

Universitat Politècnica de Catalunya (UPC)
Escola Tècnica Superior d'Enginyeria Industrial de Barcelona
Màster en Enginyeria Industrial
Màster en Enginyeria d'Organització Industrial

Abstract

In the new global economy, the inventory control has become a priority for the supply chain management. Safety stock is the sole way to fight against the demand and the supply uncertainty, so determining the amount of it that must be kept along the network to holistically minimize the risk of disruption while maximize the profit is a critical issue. For the supply side, the focus is held in the lead time variability which can significantly vary depending on the part of the supply chain or new inconvenient facts could relevantly affect the lead time. Even so, developed models have forced to assume a certain value or a distribution for the lead time, yet this is risky. Historical data is often unreliable, not available or insufficiently representative. Therefore, a new model based on the Guaranteed Service approach and combined with robust optimization techniques is proposed, working with the lead time volatility without assuming any specific distribution. Interesting features arise from the new model such as the smooth tractability of the problem, the facile computational skills required or the lack of resources needed. This approach has been formulated and tested, as well as the Guaranteed Service when a distribution is assumed for the lead time and the original model. Then, the performance of the three of them has been compared in order to find the correct way to deal with uncertain lead time. Finally, the Robust Guaranteed model benefits promise better security to companies and it also provides a powerful tool to manage the risk from the supply side.

Keywords: Lead Time, Robust analysis, Guaranteed Service approach, Inventory control, Safety stock.

Contents

List of Figures

List of Graphics

List of Tables

Chapter 1

Introduction

The supply chain world (*SC*) is a multi-factor system with different particular network agents' interests. The supply chain management comprises the management of assets (machines, buildings, patents, etc.), products (starting from the design and finishing with the process implementation), personal and several flows (information, material, money, etc.). Finding the right combination of the multiple aspects and tackle the management problem from end-to-end perspective are the key points for a company to compete in a global world.

The inherent interaction between all these factors and the interests originates a huge complexity in maximizing the supply chain benefits. The important decisions about factors such as product, personal, philosophies, assets, quantity and quality of information shared between all parts of the network and investments policy, for example, have been widely studied and they are less complicated for the companies. However, the inventory control across the supply chains, which defines the material flow, still represents a challenge for the managers' community as *CSCO [2011]* confirmed.

The inventory control constantly faces the demand and supply uncertainty. An efficient inventory management must optimize the safety stock (*SFTY*) strategy to avoid disruptions in the supply chain to control the economical uncertainty and to maximize the money-savings. Companies rely on *SFTY* to tackle the variations in order to mitigate the risk. Hence, a critical issue is to determinate the safety stock placement and the amount of stock that each stage has to hold to holistically minimize the risk of stock out and the overall supply chain cost to maximize the profit.

In order to solve the problem, the researchers initially focused their efforts to extend the Stochastic Service (*SS*) approach. The first version of the *SS* model was introduced by *Clark and Scarf [1960]* and the uncertainty of the demand and supply were defined with stochastic distributions. The other significant approach that has recently grown in attention is the Guaranteed Service (*GS*). As opposed to the *SS*, SFTY is employed to meet the demand up to certain bound and the rest of it is covered by operating flexibility. The creator of the GS is considered to be *Simpson* in 1958 and the model has been extended until *Humair and Willems [2011]*. Originally, the demands are normally and independent distributed and the lead times are deterministic. However, the basic lead time assumption is far from the real-world cases. This fact makes harder to implement the Guaranteed Service approach in companies because time and resources are only invested when the solution can be adapted to the daily essence of a company. In that way, *Humair et al. [2013]* proposed a

Random Guaranteed model which works with assuming a stochastic distribution for the lead time. Even so, how to deal with the lead time variability is a controversial topic due to the fact of the data risks - assuming an incorrect distribution and collecting unreliable, uncertain and incomplete.

The principal aim of this Master Thesis is to contribute with a new model based on the *GS* and called Robust Guaranteed Service model which proposes to deal with the uncertainty of the lead time without assuming a distribution. The performance, strengths and weaknesses of the basic *GS*, the random and the robust models are presented and compared to find the best way to deal with lead time in order to optimize the inventory. In addition, a secondary goal is to provide useful tools to the future researchers in this field so the codification of each approach is exposed in the appendixes. On the other hand, the character relationship between decision variables and the outputs, cost and safety stock, is deeply explained at the beginning of this work to fully understand the models' performance.

The thesis is structured as follows: The past literature review about the diverse approaches is presented in *Chapter 2* while in *Chapter 3* the theoretical basis for understanding the models and the numerical experiments are summarized. *Chapter 4* exposes the formulation of the original Guaranteed Service approach, adds the maximum outbound service time constraint for internal nodes and clarifies the meaning relation between parameters. In *Chapter 5*, the main innovation (the Robust Guaranteed Service approach) is formulated at first, and validated afterwards. In the next chapter, the Guaranteed Service approach under random lead times from *Humair et al. [2013]* is exposed and numerical trials comparing the three models are displayed. *Chapter 7* finishes with the conclusions and future lines.

Chapter 2

Literature Review

The state of the art regarding strategic safety stock placement in supply chains is very extensive. Starting from the origin of the more general approaches (Stochastic Service and the Guaranteed Service) placed in time as well as its most relevant extensions and successes in the next decades until nowadays. As an overview, the literature regarding the topic can be divided into approaches and its corresponding assumptions and such as, it will be detailed below using that classification.

2.1. Guaranteed Service approach (original assumptions)

This approach for modeling safety stock and inventory has been widely studied in the last 60 years. The literature shows how the knowledge on the Guaranteed Service model has been gradually extended until being able to be applied at the most complex networks.

The initial model was presented by *Simpson [1958]*. The research was only successfully conducted for serial supply chains. For the first time in this research field, the optimal solution was demonstrated to occur when the service time takes one of the extreme values of the possible domain solution.

After *Simpson*, *Inderfurth [1991]* was the next in contributing with a dynamic program to execute the Guaranteed Service model in general serial and divergent (distribution) systems. *Inderfurth* together with *Minner [1998]* extended to convergent (assembly) supply chains the dynamic procedure. They run several numerical experiments under different service measure to proof that the approach can be used in different service levels situations and the results concluded that the size and the allocation of stock depend on the service level requirements. The same author, *Minner [2000]* summarized in his book all the approaches for the safety stock placement until that moment and provided accurate definitions of the concepts involved. The book classified all the basic models in different categories depending on the assumptions used in each one: single-stage or multi-stage modeling framework, stochastic or deterministic lead times and the applicability of the model in the diverse types of networks. In addition, it extended the original model based on the material flow philosophy.

The same year that *Minner* published his book, *Graves and Willems [2000]* evolved the Guaranteed Service model multi-stage procedure to enable it to be used in more general, complex and realistic supply chains. The procedure was lately reviewed and it was correctly published by the same authors in 2003. With slight abuse of redundancy, a spanning tree algorithm was introduced for the networks that are spanning trees. The first step consists in labeling the

nodes of the supply chain using a specific method. Secondly, a dynamic program is used to solve the problem. The recursion of the program was newly limited by adding constraints bounding the decision variables (the inbound and outbound service times).

The next remarkable study about the model following the original assumptions was made by *Lesnaia [2004]*. In her Master Thesis, she reviewed all the methods appeared until that moment; by proving that the optimization of the safety stock problem in a general network is an NP-Hard problem and she demonstrated the optimality of the solution for it. Furthermore, she developed a new branch and bound algorithm based on paths.

Up to now, the sole technique to solve the optimization problem was the dynamic programming, which implementation is necessary because the objective function is non-linear. *Magnanti et al. [2006]* innovated by proposing a linear approximation technique to minimize the total cost of safety stock allocation. The main benefit of this manner is that a commercial solver can settle the optimal solution then. On the other hand, *Shu and Karimi [2009]* chose to solve the problem with heuristics. These techniques are efficient in terms of computational time but worse in terms of solution because only near-optimal solutions are reached with this method. The computational time needed is less since the heuristics are nearly independent from the size of the network.

Regarding the above-mentioned methods into account, most the literature has preferred the dynamic programming as a way of resolution. *Humair and Willems [2011]* were the ones who finally achieved the goal to solve the problem for acyclic chains (the major of real world networks are acyclic) by using a routine that includes the spanning tree algorithm already exposed and a dynamic approach. Without digging deeper on its specific method, at the beginning some links of the acyclic supply chain must be broken to achieve a spanning tree network, then the spanning tree and the dynamic program can be applied and finally, a routine tests the solution to check if all the constraints are right in all the links. In case of some broken constraints, the routine finds a near solution for the spanning tree and it tests again. This step is repeated until the routine obtains an optimal solution that fulfils all the constraints for all the links. In addition, the paper presented two extra heuristics to find near optimal solutions to the problem.

After *Humair and Willems* published his work in 2011, there was no gap for extending the research in the model with the original assumptions to different supply chain types. Therefore, lately the research topics have been focusing in modeling the assumptions of the Guaranteed Service model, in the performance of the different approaches (the Guaranteed Service against the Stochastic Service) and in mixing both approaches.

2.2. Guaranteed Service approach (modeling the assumptions)

Apart from the above-mentioned contributions, two papers went beyond the original assumptions (*Magnanti's et. al [2006]* and *Humar and Willems' [2011]* papers). The former also added the capacity constraint to the model. Also in the latter, the utility of the routine for solving the safety stock optimization problem in a supply chain under the original statements is even more relevant than what is explained before, since it allows solving any general cost objective function for general networks (in which the objective function can adopt any general cost equation, and can be non-concave, non-closed-form and/or non-continuous). Therefore, an objective function can be modeled to take into account variable lead times or non-nested review periods.

In this report, the discussion of whether it is correct to use the assumption of lead time deterministic is going to be presented. Some work has already done in this study subject: *Humair et al. [2013]* developed a safety stock expression for stochastic lead time that follows a determinate discrete or continuous variable with mean and standard deviation. Then they used the algorithm from *Humair and Willems [2011]* to get results from different networks cases.

When stochastic lead time was considered, it was either typically normally distributed or characterized by historical data: on the one hand, *Eppen and Martin [1988]* have already showed with examples that normality assumption is unwarranted. On the other hand, historical data is often not available or unreliable. That is why is needed to continuing study the case of stochastic lead time in Guaranteed Service model. More recently, *Beiran and Martín-Romero [2017]* were adopting a completely new perspective for the discussed statement. They assumed that the parameter lead time is unknown but it belongs to a range of possible values. Hence, no distribution is assumed and a new model is created as a new tool to solve the problem: the Robust Guaranteed Service model.

In Eruguz et al. [2016], an exhaustive survey about the original Guaranteed Service model and the modeling of the assumptions is made.

2.3. Stochastic Service approach

There is broad literature about the Stochastic Service model but in this section only the main research is going to be referenced because this approach is not used in this thesis.

The initial model was introduced in 1960 by *Clark and Scarf [1960]*. Then, *Diks et al. [1996]* reviewed the most successful achievements for divergent supply chains. Other remarkable literatures are *Axsäter [2003][2006]* and *Simchi-Levi and Zhao [2012].* The last-mentioned is an exhaustive survey of the different

Stochastic Service models for multi-echelon inventory systems and the performance evaluation between them.

2.4. Comparison between the Guaranteed and Stochastic Service Model

Since the creation of both approaches, controversy around which one is best has been a central topic in the research field. At least two relevant analyses have been published. The first one was conducted by *Graves and Willems [2003]* and it concluded that in general the Stochastic Service Time holds more safety stock and the total cost is higher. The second one is from *Klosterhalfen* and *Minner [2007][2010],* which detailed and compared both approaches in terms of performance, materials flow and resulting service times. They agreed with the analysis of 2003. However, it is noted that without operating flexibility measures the Stochastic Service way is better regarding total cost. In addition, it is also remarked that the Guaranteed Service model is computationally easier.

2.5. Mixed Approaches

Theoretically speaking, none approach stands out from representing the real world than the other. That may be the reason why some researchers expose models where a more realistic assumption of both approaches is done by combining to form a new model.

Rambau and Schade [2010] contemplated a Guaranteed Service model with a demand scenario sampling creating the Stochastic Guaranteed Service model. It is a model that does not upper bound the demand, in which the service level is an output of the model and that adds the cost of the extra-measures from the operating flexibility at the objective function. The difficulty in this case is to take a significant and relevant sample.

Klosterhalfen and Minner [2013] created an integrated Guaranteed- and Stochastic-Service approach for the inventory optimization in supply chains (the Hybrid-Service approach). The idea is that the model allows to each stage of the network to choose the best approach to calculate the minimum total supply chain cost.

2.6. Data

Representative data of the real-world networks is needed to run numerical examples to prove the theory. *Willems [2008]* shared data from 38 real multi-echelon supply chains that can be used in empirical studies to inform and test.

Chapter 3

Framework

An introduction to the core issues of this research is conducted in this chapter. The first subsection is focused on the main approach used in this thesis. Subsections two and five explain the different kind of networks while the third one faces the difficulties of the multi-echelon supply chains. Finally, subsection four presents the diverse stock types.

3.1. Safety Stock placement approach

There are mainly two approaches to study the safety stock placement: the Guaranteed Service model (*GS*) and the Stochastic Service model (*SS*).

The *GS* model states that every stage i in a supply chain guarantees a delivery time to its customers. Furthermore, the approach combats against the uncertainty of the data with the safety stock and the operating flexibility.

The basic assumptions for the model are mentioned below. See *Graves and Willems [2000]* for an exhaustive understanding of the statements.

- **Multi-stage Networks:** A network is represented by a graph which is simplified with nodes and arcs. Every stage in the supply chain is a node and every relation between stages is an arc. All the multi-stage systems considered in the document are acyclic networks.

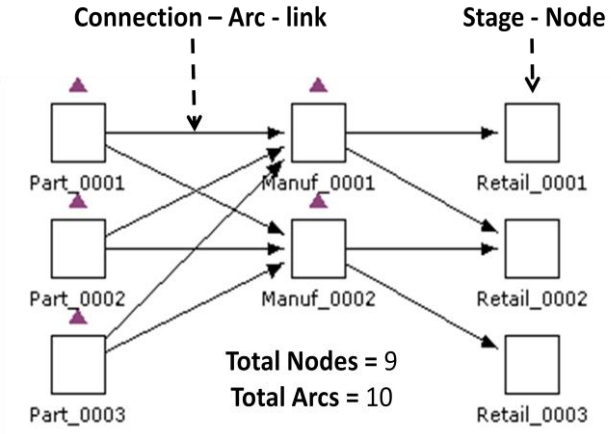


Figure 3.1: Multi-stage network example. *Willems [2008]*

Each stage performs the role of some activity of the process in a supply chain so a stage can be a supplier of raw material or component (part), an assembly and producer company (manufacturer), a transportation step between two stages (transport), a final selling point to the end-customer (retail) or a distribution centre to other stages or to the final customer (distributor).

In addition, there is no delay in ordering in the system. When the stock is placed at the warehouse, then it is ready for the immediately shipment if it is necessary.

- **Periodic-Review Base-Stock Replenishment Policy:** All stages have a Periodic-Review Base-stock Replenishment Policy with common periods.
- **Production lead times:** Lead times are known, constant and independent from the order size. Lead time is the production time for a component/product once all its parts are available. It includes the transportation time to the warehouse.
- **Capacity:** No capacity constraints referring to production volume or warehouse space in any stage.
- **Demand process:** The demand is stationary and it is only known at the stages without successors: *demand or external nodes*. The mean μ_i and the standard deviation σ_i are known for them.
  Important to note, the demand is bounded with a function $D_i(\tau)$ where τ refers to a period of time. The model contemplates how to serve the demand until the bound. When normal distribution is assumed for the demand, then:

$$D_i(\tau) = \tau \times \mu_i + z_i \times \sigma_i \times \tau \quad (1)$$

  z_i is related with the service level with the customers of stage i and it means how much the company is willing to incur into stock out.
- **Guaranteed Service times**: The outbound service time for the stage i (S_i) means that the stage i guarantees that the demand, up to a certain bound, is served to each of its downstream nodes in a period of time S_i.
  Besides that, the inbound service time for the stage i (SI_i) defines that the stage i is promised to be served and gets all its inputs in a period of time SI_i.
  Hence, every stage has two services times, the outbound and the inbound service time.

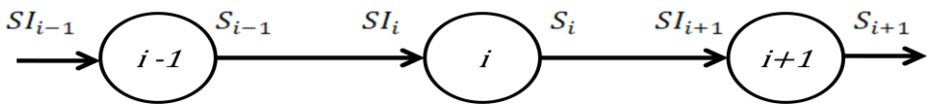


Figure 3.2: Services per stage

- **Operating flexibility:** Extra-ordinary measures such as outsourcing or working extra hours are admitted for fulfilling the demand that exceeds the limit. However, the cost of this operational flexibility is not part of the model.

As opposed to the *GS*; the Stochastic Service model considers the uncertainty and variability of the data by assuming stochastic distributions for the demand

and the lead times. Moreover, it fills all the complete demand (there is no upper bound) and it does not take into account any extra-ordinary measure. The general idea is that if there is an order and there are enough stock units as the order size, then it can be shipped to the customer. Otherwise, it has to be delayed until there is enough stock to reach the original demand order. Therefore, one of the basic differences between approaches is that there is room for delays and stock outs in the model. The other essential contrast is that the safety stock is the unique measure to battle the uncertainty of the model.

All the case studies analyzed in this report are under the Guaranteed Service model approach and discuss the assumption of lead times known and constant.

3.2. Single or multi-echelon systems

In general terms, profit is known to be the ultimate goal of a company. Therefore, every transaction between stages is possible if the end-customer buys the product and both the supply chain actors and the service to the final customer are affected for the stock policy of each stage. That is why it does not make sense to optimize the safety stock placement in a single stage and it is also the reason why it is necessary to work with a multi-echelon inventory optimization perspective even though it makes the uncertainty of the variables higher.

3.3. The difficulty behind the multi-echelon systems

There are three peculiar issues to confront: the cost propagation, the demand propagation and the mathematical methods to solve the problem.

When the data about the cost is the holding cost per stage and per unit h_i, there is no problem. However, when the data cost is the added cost c_i in each stage then some calculations must be done to find the holding cost:

$$C_i = c_i + \sum_{j \in (j,i)} c_j \tag{2}$$

C_i is the cumulative added cost per stage and per unit. The cumulative cost has to be multiple per the annual holding cost value h to find the holding cost per stage.

The theory is more complex for the demand propagation case. It is a fact that the demand is given only for external nodes j (external demand). The external demand has to be propagated to the internal nodes, i nodes.
There is one clear expression from the past literature about the topic: $\mu_i = \sum_{j \in (i,j)} \vartheta_{ij} \times \mu_j$ where ϑ_{ij} is the number of units of component produced in stage i to produced one unit in stage j. However, there are discrepancies between which expression is used to find d_i and $D_i(\tau)$ for internal nodes. The concept

that generates all the discussion and confusion is the *pooling risk*: if the inventories for different components which are needed at the same time for more than one product are joined, then the variability of the demand is reduced and the safety stock inventory decreases. Depending on the *pooling risk (p)* factor the expression for demand parameters differs (see *Graves and Willems [2000]*). This report assumes a pooling risk factor equal to 1 that means no interaction between demands coming from different stages for a simple computational reason. According to the theory in *Graves and Willems [2000]*, the expression for *p=1* used to calculate $D_i(\tau)$ at the internal demand is the same as for the external nodes, equation (1).

Finally, the discussion is focused in which mathematical method applied. The *GS* model problem has a non-linear objective function, as well as non-concave depending on the expression for the safety stock, with linear constraints. In other words, linear resolution methods cannot be used.

When the optimizing problem is only non-linear, then the dynamic programming and the linear approximation procedures are proved to find an optimal solution. The problem of both methods is that they are slow in terms of computational time. As bigger the number of stages in the problem, the bigger the recursions and the time needed. On the contrary, the heuristics methods find near-optimal solution but they required less computational time.

On the other hand, only the dynamic programming and the heuristics work when the optimizing equation is non-linear and non-concave.

3.4. Types of stock

The large majority of the stock placement optimization literature refers to a specific type of inventory: The safety stock. However, it is not as simple as just considering one type. From here on, for types of stock will appeared: base (*BS*), safety (*SFTY*), pipeline (*PS*) and early arrival stock (*ES*). Next, the document explains the main ideas and the formulation of each type. Refer to *Graves and Willems [2000]* for the demonstration of the formulas and more information about the topic.

The inventory level at stage i and period t is defined as $I_i(t) = B_i - d_i(t)$. The base stock (B_i) is the amount of stock at the inventory just before the shipment of the demand.

The *BS* should be equal or higher than the demand to not incur into a stock out. It is important to note that the *GS* avoid the stock out in its model. Adapting the argument to the *GS* model, $B_i \geq D_i(\tau)$ because the demand is limited by an upper bound and the rest of it will be served by extra-ordinary measures. The best solution in terms of cost is having the less stock units possible, so the final

expression for the *BS* is $B_i = D_i(\tau)$. τ is the net replenishment lead time $(NRLT_i)$. The expression for the $NRLT_i$ is:

$$NRLT_i = SI_i + L_i - S_i \tag{3}$$

Basically, the base stock has to cover the upper bound demand during the net replenishment time which means that there has to be enough stock for the extra time needed when the sum of the production time L_i and the inbound service time SI_i is bigger than the time the stage has to deliver the bounded demand to the customer, S_i. Therefore it can be concluded that:

$$B_i = D_i(\tau) \quad \text{where } \tau = \max(0, NRLT_i)$$

The safety stock or the expected inventory is part of the *BS*. In fact, the *SFTY* is the part of the *BS* used to cover the uncertainty of the $D_i(\tau)$ and it can be expressed by:

$$SFTY_i = z_i \times \sigma_i^d \times \sqrt{NRLT} = z_i \times \sigma_i^d \times \sqrt{SI_i + L_i - S_i} \tag{4}$$

The pipeline stock are units that have been started to be produced but that have not yet reached the warehouse since they are not finished yet (work-in-progress stock).

$$PS_i = L_i \times \mu_i \tag{5}$$

The early arrival stock (*ES*) is a new concept introduced by *Humair et al. [2011]*. The meaning of this term is going to be explained at the Random Guaranteed Service section.

3.5. Types of Supply chain

It is necessary to note that there are different supply chains depending on how the stages interact with each other because it is easier to solve the optimization problem in some chains than in others. The *SS* model is adopted normally in serial, assembly and distribution networks because the computations are easy to carry. For supply chains more complex, the number of calculations is high and the stochastic equations become hard to solve. This is the reason why lately it has been used the *GS* approach considering its less essential computational difficulty to solve the model.

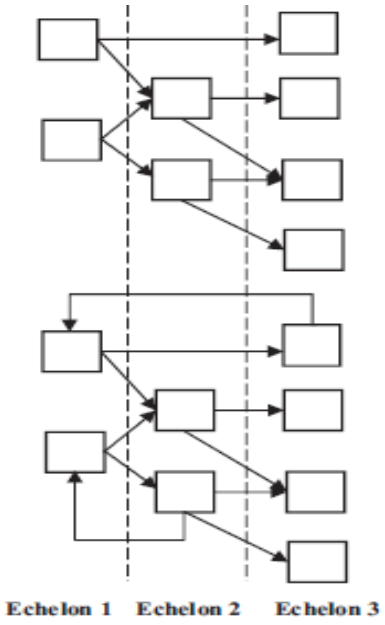


Figure 3.3: Acyclic and cyclic Networks. *Eruguz et al. [2016]*

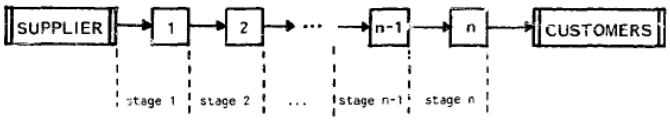


Figure 3.6: Serial production/inventory system. *Inderfuth [1991]*

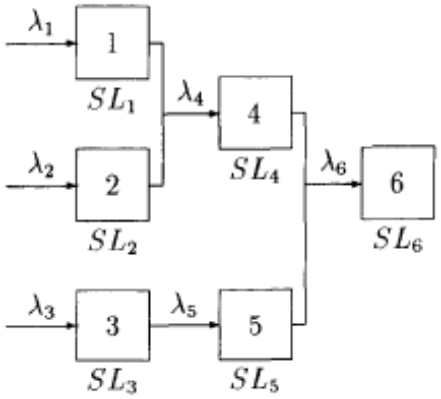


Figure 3.5: Convergent (Assembly) network. *Inderfurth and Minner [1998]*

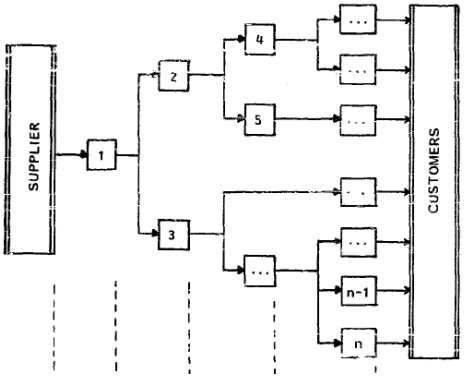


Figure 3.4: Divergent (Distribution) network. *Inderfuth [1991]*

Chapter 4

Deterministic Guaranteed Service model (GS-DET)

4.1. Formulation

One of the basic assumptions of the Guaranteed Service model is that the production lead times are deterministic, so this data is known and fixed for all the stages in the GS-DET. The following input data for all nodes is necessary to formulate the problem:

- $\boldsymbol{L_i}$**:** Production lead time for stage i.
- $\boldsymbol{h_i}$: Holding cost per unit at stage i. The data available is the added cost to the product in each stage. Therefore, the holding cost per unit is calculated with the corresponding formula noted at the *Framework*.

For the demand nodes, more information is required. They are the ones directly connected to the end-customer in the supply chain. They know the requests of the final clients and the demand of the final product produced by the whole network. The above mentioned information is as follows:

- $\boldsymbol{\mu_i^d}$**:** Mean of the demand for external nodes. This information must be transferred to the internal nodes as explained at the *Framework* section.
- $\boldsymbol{\sigma_i^d}$**:** Standard deviation of the demand for external nodes. This information must be transferred to the internal nodes as explained at the framework's section.
- $\boldsymbol{z_i}$: Safety factor related to the service level for demand stages. It means how frequently the company is willing to incur into stock out. Data for external nodes must be propagated to the internal nodes so the mean value of the demand service level is taken for the internal ones. The inverse of the cumulative standard normal function of the service level is generally the method to calculate this factor. In other scenarios, the value 1,64 is taken for all the stages.
- $\boldsymbol{E_i}$**:** Maximum service time for the external nodes. In other words, maximum outbound service time allowed for the final nodes to deliver the product to end-customer.

The program to solve the minimum cost for the placement of safety stock combines these inputs with the best decision variables to find the optimal cost for the supply chain. Hence, the outputs of the problem are:

- $\boldsymbol{S_i}$**:** Outbound service time per stage i.
- $\boldsymbol{SI_i}$: Inbound service time per stage i.

- The cost per stage according to the best service times found by the program.
- The optimal cost for the supply chain to allocate the safety stock. From now on, problem *P* refers to resolving the minimum cost for the placement of safety stock in a general supply chain.

Therefore, the mathematical formulation for the problem *P* based on the multi-stage *Willems' [2000]* model is then:

P

$$Min \sum_{i=1}^{N} h_i \times SFTY_i \tag{6}$$

s.t.

$$S_i - SI_i \leq L_i \qquad for\ i = 1,2 \dots N \tag{6a}$$
$$SI_i \geq S_j \qquad for\ (j,i) \in A \tag{6b}$$
$$S_i \leq E_i \qquad for\ all\ i \in D(G) \tag{6c}$$
$$S_i, SI_i \geq 0\ and\ integer \qquad for\ i = 1,2 \dots N \tag{6d}$$

The objective function (6) minimizes the stock. From the types of stock above mentioned in the section *Framework,* it is only the expected inventory or safety stock (*SFTY*) to be contemplated. The equation to express this expected inventory is (4). The pipeline stock only depends on the Lead time and the demand of each stage. Both parameters are constant so it can be avoid from the optimization due to the fact that the values of the decision variables not affect this type of stock.

The first constraint, 6a., defines the net replenishment time and ensures the non-negativity of the *NRLT*. If the difference between the outbound and the inbound service time is equal to the lead time, the stage is not holding safety stock. Otherwise, the stage expects inventory (more at *Humair et al. [2013]*). 6b. assures that the inbound service time meets the maximum service time of their immediate successors. The next one, 6c., guarantees that the external demand is served within the service time required. Finally, 6d. defines the nature of the decision variables.

4.2. Experimental codification

Problem *P* has already been solved using dynamic programming, linear approximation and heuristics procedures for all types of networks: serial, assembly, distribution, acyclic and cyclic. All these methods have been discussed in papers mentioned in the *Literature Review* section. However, not a single one shows the code for the program.

Focusing the attention to the dynamic procedure, one of the main contributions of this thesis is providing and explaining a code to solve the *GS-DET*. To

validate the code, it has been tested in the *Moncayo-Martinez and Ramirez [2016]* tutorial and in the supply chains from *Willems [2008]* appearing in *Humair et al. [2013]*.

Before discussing the details about the code, the theory of the dynamic programming has to be understood. Review *Graves and Willems [2000]* and its correction in 2003 for the knowledge in the algorithm for labeling the stages and the dynamic steps. Check the *Humair and Willems [2011]* paper to understand the theory behind the routine needed to solve acyclic chains.

In the first test, the six stage convergent network is a perfect spanning tree. The code program proposed in the *Appendix 1,* subsection eight, labels the nodes in a different order than the numerical case exposed in *Moncayo-Martinez and Ramirez [2016]*. The point is that the spanning tree algorithm from *Graves and Willems [2000]* allows choosing between a set of alternative in some iterations freely. Indistinctively, the results show how the theory works and the outputs match the results of the tutorial ($Zopt = 755\,um$ and *Figure 4.1* shows the optimal service times).

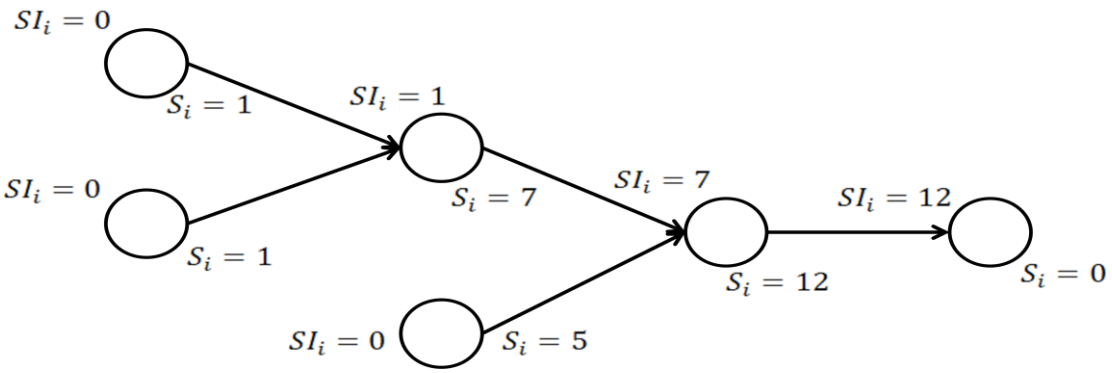


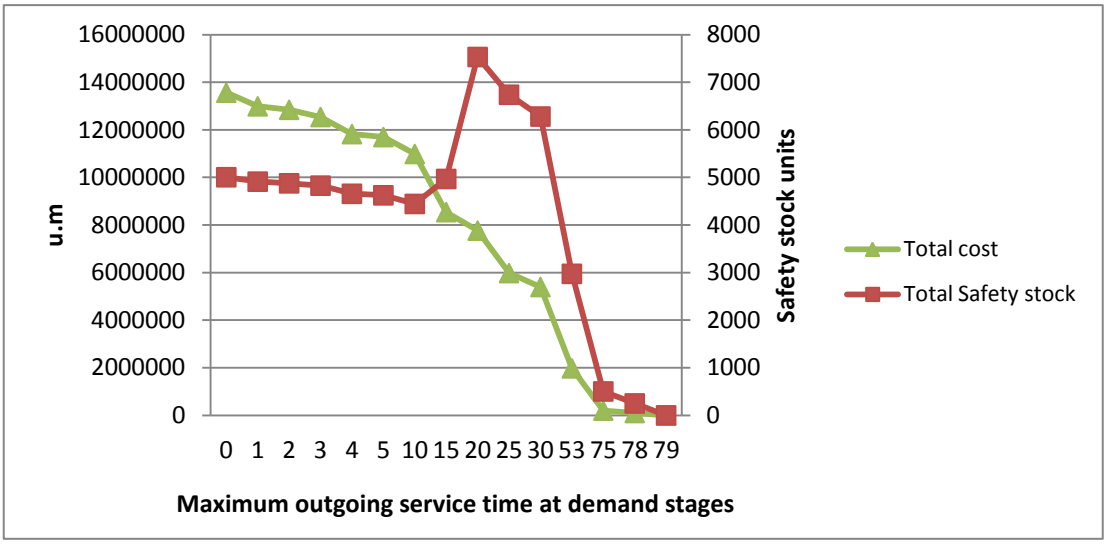
Figure 4.1: Optimal outbound and inbound service time for each stage of the tutorial's network

The network of the second case is an acyclic network which needs to break arcs to form a spanning tree. The code works for 11 out of 12 supply chains in *Humair et al. [2013]* in terms of cost and services time. The computational time increases when the number of stages per supply chain and the number of broken constraints does too.

4.3. The end-customer grade of request

In general, the conclusions in studies about the Guaranteed Service model talk about the safety stocks units, the number of stages holding it and the total cost of it. To the best of this report's knowledge, there is no analysis about the relationship between the outbound service time and the other problem's outputs. This section and the next one contribute with this new analysis. The numerical experiments were run in the supply number 03 from *Willems [2008]*.

In this first case, the main idea was to obtain different cost, safety stock and number of stages holding stock results for different end-customer's maximum service time. Afterwards, the behavior of the variables was analyzed.



Graphic 4.1: Economical effect of increasing outgoing service time in demand stages

The *Graphic 4.1* shows how the cost decreases permanently but non-lineally when the external stages allow more delivery time to its supplier. It can be also stated that the trend for the safety stock units is the same as the cost overall. For the safety stock units line, there is one discrepant point when $S_{\max\ \_demand\ stages}$ is 20 days. As it can be observed in *Table 4.1* for different outgoing service times for external nodes, the safety stock placement strategy is changing from the most expensive places to the cheapest ones when $S_{\max\ \_demand\ stages}$ increases. The strategy changes when $S_{\max\ \_demand\ stages}$ is 20 days. At this value, there are more stages holding stock and more units. However, these units are settled in cheaper places than before and in total the cost is less.

$S_{max\ \_demand\ stages}$	**Stages holding safety stock** (cost added in each stage [um])
0	**Dist1** (2 750,00) - **Dist2** (4 103,00) - **Dist3** (4 162,00) - **Dist4** (4 247,40) - **Part3** (400,00) - **Trans2** (1 503,00) - **Trans4** (2 350,00)
15	**Man1** (2 400,00) - **Trans3** (1 502,00) - **Trans4** (2 350,00) - **Part3** (400,00) - **Part5** (1 500,00)
20	**Man1** (2 400,00) - **Man2** (2 350,00) - **Part1** (1 100,00) - **Part2** (600,00) - **Part3** (400,00) - **Part4** (1 100,00) - **Part5** (1 500,00)
53	**Part1** (1 100,00) - **Part3** (400,00) - **Part5** (1 500,00)
78	**Part3** (400,00)
79	**None**

Table 4.1: Strategic safety stock placement

It is also curious that there is no safety stock and no cost when $S_{\max\ \_demand\ stages} = 79$ days. This is possible because it is the maximum replenishment time for any stage in the supply chain. Hence, mathematically speaking is possible to combine all the service times to not have stock following constraints 6a. and 6b.

To sum up, the supply chain has more reaction time against the demand when the end-customers are more flexible. If the demand stages are permitted to respond slower, then the reaction time for every stage is bigger and the cost is lower. The safety stock placement changes to allocate it in cheaper stages when $S_{\max\ \_demand\ stages}$ is higher than a certain value. The basic concepts are that slight changes of stock units in the most expensive stages mean large differences in cost because of equation (2) and that the outbound service time and the cost are antagonist variables.

At the end, this kind of analysis can help companies to perform an evaluation about the cost versus the service time and to take decisions about which variable to prioritize and until which point.

This analysis can be extended to all the networks possible with the same conclusions because the *NRLT* decreases when the outbound service time increases, so there is less safety stock in general.

4.4. Modeling the *GS-DET* with maximum outbound service times for internal nodes

In the second numerical experiments, the framework was changed by adding to the model the constraint of maximum outbound service time for all the stages. *Table 4.2* shows the data constraints imposed for different cases studied.

Real Data		**Case 1**		**Case 2**		**Case 3**		**Case 4**	
Stage	S_{max}	**Stage**	S_{max}	**Stage**	S_{max}	**Stage**	S_{max}	**Stage**	S_{max}
Dist	0	**Dist**	0	**Dist**	0	**Dist**	0	**Dist**	0
Man	-	**Man**	24	**Man**	12	**Man**	10	**Man**	0
Trans	-	**Trans**	4	**Trans**	2	**Trans**	0	**Trans**	0
Part	-	**Part**	60	**Part**	30	**Part**	20	**Part**	0

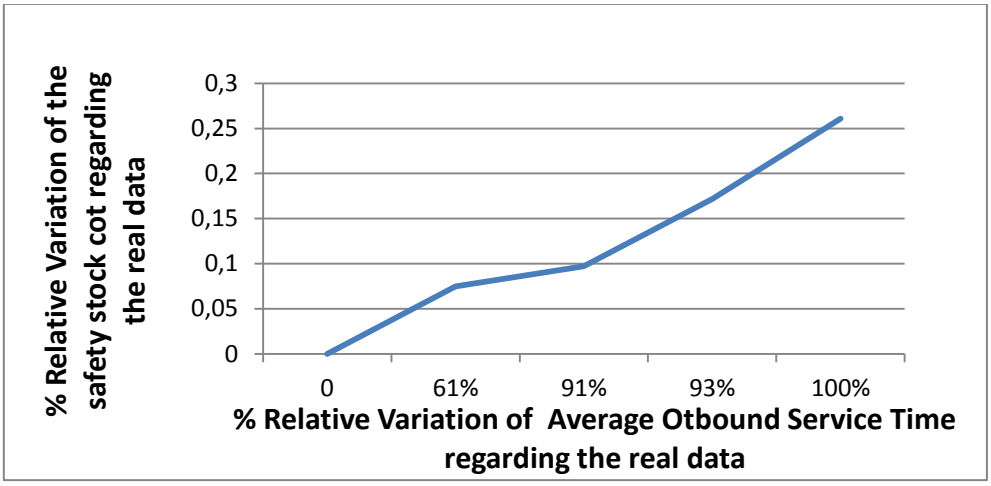
Table 4.2: Maximum outbound service time for internal nodes for the different cases tested

	Total Cost	Av. SI	Av. S	Number of stages holding SS	Total SS units	Iterations
Real	13 565 043,80	23,29	26,53	7	5 004,90	10
Case 1	14 578 648,46	7,29	10,41	8	6 395,14	5
Case 2	14 884 831,67	2,94	2,35	11	9 494,15	1
Case 3	15 894 494,00	2,35	1,76	14	10 164,60	1
Case 4	17 105 442,13	0,00	0,00	17	10 568,02	1

Table 4.3: Results for the Guaranteed Service model with maximum outbound service times for internal nodes

In *Table 4.3* the columns explain the total cost of the supply chain 03 (*Total Cost*), the average of the inbound service time for the whole network (*Av. SI*), the average for the outbound service time (*Av. S*), the total number of stages holding safety stock (Number of stages holding SS), the total units of safety stock (*Total SS units*) and the number of iterations needed in the routine for solving the problem (*Iterations*). Again, the outbound service time is antagonist to the total cost and the safety stock units. The tougher the customer is the better service he receives, but less flexible is the system and the higher the total cost. The extreme case is the number 4 where there is no freedom in terms of choosing the service times. Therefore, the response to the customers is immediately guaranteed. Big safety stock is necessary to fight the instant delivery because the production lead times are not zero.

Graphic 4.2 displays how the trend of the average relative variation of the outbound service time from the real case to 0 (case 4) is opposite from the relative variation of the cost between the real data situation and the last case. In addition, for this supply chain the complete relative variation of the outbound service time correspond to a relative cost growth of 'only' 26%.



Graphic 4.2: Safety stock versus customer service

Chapter 5

Robust Guaranteed Service model

5.1. Lead time in the Real World

The fixed, known and constant lead time assumption of the Guaranteed Service model is one of the weaknesses of this approach because it does not represent the real situation in companies, factories or distribution centers. Actually, the variability of the lead time is typically different depending on the part of the supply chain and depending on every part's function in the process. It is well-known that the raw material supplier faces longer delays and more uncertainty than the other stages. At the manufacturing parts, the knowledge of the variability of the process is higher since has been studied in many occasions so the delays are controlled and there is less uncertainty. Finally, the retail stores face only 1 or 2 days delays in general due to the fact that the end-customer's requirements are the most important constraint in the supply chain.

The lead time variability is a key issue that forms part of the uncertainty from the supplier side. Over the past two decades, several new models considering specific distributions for the lead time were introduced to fight this unreal assumption such as the Random Guaranteed Service model, which it is going to be explained in the next chapter. However, defining the right distribution for the variable in each situation is not a trivial task and the stochastic distribution selected is sometimes not reliable. Historical data of the process is often not available or not representative of the current and/or future situation. Therefore, companies are assuming a risk every time that a determinist value or a specific distribution is considered.

Recently, *Beiran and Martin-Romero [2017]* tackled this issue with a new approach called Robust Guaranteed Service model and the first experiments provide a powerful tool for the companies to at least manage the risk. This approach is the most important contribution in this thesis.

5.2. Robust Guaranteed Service approach (*GS-RO*)

The *GS-RO* deals with the lead time uncertainty without assuming any distribution avoiding risks. The unique fact known is that lead time is uncertain within a certain interval, defined by (7).

$$L_i = \bar{L}_i + ב_i \cdot \hat{L}_i \qquad for\ i = 1,2 \dots N\ and\ ב \in [-1,1] \qquad (7)$$

Inspired by the budgeted uncertain set presented in *Bental et al. [2009]* and deployed in *Berstimas and Thiele [2006]* in which the uncertainty is bounded,

$|ב_i|$ is a parameter set by the user between 0 and 1. It is assumed a fixed value and no uncertainty when $ב_i = 0$ while $ב_i = 1$ maximizes the doubt. Different levels of uncertainty can be chosen with this approach. Hence, the cost of immunization against the supply delays and the flexibility for the stages in the supply chain could be evaluated.

As opposed to all the other models used in this work, the *GS-RO* has been solved with a linear approximation method as *Manganti et al. [2006]* does in his research. For this case, the objective function (8) is separable, concave and non-decreasing. This way, the model remains a mixed-integer linear program that can be solved by commercial software such as *Gurobi* and the computational times are not long in any of the networks used (less than one minute).

The program coded in *Python* and the model formulation were first contributed by *Beiran* and the definitively version was developed by *Beiran and Martin-Romero*. The model formulation of the safety stock problem is then the next one:

P

$$min \sum_{i=1}^{N} h_i \left(\sum_{r=1}^{R} (f_r^i u_r^i + \alpha_r^i z_r^i) \right) \tag{8}$$

$$x_i = \sum_{r=1}^{R} z_r^i \qquad for\ i = 1,2 \dots N \tag{8a}$$

$$M_{r-1}^i u_r^i \leq z_r^i \leq M_r^i u_r^i \qquad for\ i = 1,2 \dots N\ and\ for\ r = 1,2 \dots R \tag{8b}$$

$$\sum_{r=1}^{R} u_r^i \leq 1 \qquad for\ i = 1,2 \dots N \tag{8c}$$

$$u_r^i \in \{0,1\} \qquad for\ i = 1,2 \dots N\ and\ for\ r = 1,2 \dots R \tag{8d}$$

$$z_r^i \geq 0 \qquad for\ i = 1,2 \dots N\ and\ for\ r = 1,2 \dots R \tag{8e}$$

$$S_i \geq E_i \qquad for\ all\ i \in D(G) \tag{8e}$$

$$SI_i \geq S_j \qquad for\ (j,i) \in A \tag{8f}$$

$$x_i \geq SI_i - S_i + L_i \qquad for\ i = 1,2 \dots N \tag{8g}$$

$$y_i \geq \Phi_i(x_i) \qquad for\ i = 1,2 \dots N \tag{8h}$$

$$SI_i, S_i, x_i, y_i \geq 0 \qquad for\ i = 1,2 \dots N \tag{8i}$$

$$y_i \in \mathbb{Z} \qquad for\ i = 1,2 \dots N \tag{8j}$$

The different terms not defined in the *GS-DET* are:

- $\boldsymbol{x_i}$**:** Is the net replenishment time.

- $\boldsymbol{y_i}$: Order point.
- $\emptyset_i(\boldsymbol{x_i})$**:** Expresses the level of safety stock in terms of the net replenishment lead time. The equation for this parameter is: $\emptyset_i(\boldsymbol{x_i}) = \boldsymbol{z_i} \cdot \boldsymbol{\sigma_i^d} \cdot \sqrt{\boldsymbol{x_i}}$
- $\bar{\boldsymbol{L}}_{\boldsymbol{i}}$**:** It is the midterm value of the interval of lead time.
- $\hat{\boldsymbol{L}}_{\boldsymbol{i}}$: It is half of the interval of lead time.
- $\boldsymbol{ב_i}$**:** The parameter to express the uncertainty of lead time.
- $\boldsymbol{R}$**:** Pieces of the linear approximation.
- $\boldsymbol{f_r^i}$**:** Independent term.
- $\boldsymbol{z_r^i}$**:** Auxiliary variable to activate the slope.
- $\boldsymbol{u_r^i}$**:** Auxiliary variable to activate the independent term.
- $\boldsymbol{\alpha_r^i}$**:** Slope.

Focusing the attention in the uncertain data, it only appears in constraint 8g. Note that there is only one uncertain parameter for each i^{th} constraint. Thus, the only possible uncertain set is a box. See *Appendix 4* for the codification of the problem *P*.

5.3. Computational results

The *GS-RO* has been validated and applied to 20 out of 38 supply chains from the data of *Willems [2008]* varying the number of stages from 8 to 159. There two types of supply chains in the database and two different analyses consequently. Some networks consider the lead time as a random variable (*r-nets*), expressing the lead time as a discrete distribution based on historical data or as a normal distribution, and some others as a deterministic value (*d-nets*). In the former, the *GS-RO* results have been compared with the results from *GS-DET* considering the mean of the lead time as the deterministic value for the variable (*Humair et al. [2013]*). If the lead time is expressed as a discrete distribution, the uncertain interval for this parameter is defined by its minimum and maximum value. Otherwise, the uncertain interval is described by the mean value plus three times its standard deviation. In the latter, the effect of the variability in the lead time has been studied by assuming a certain interval of uncertainty.

Table 5.1 and *Table 5.2* summarize the comparison results for the *r-nets* types and the meaning of the columns are:

- ***SC:*** The number of the network in the public-available database from 2008.
- ***Mid:*** Display the solution of the *GS-RO* model when the lead time has no variability. The term has to do with the fact that the lead time value taken is not the mean but the middle value from the uncertain interval for the lead time (no distribution assumed).

- ***Max:*** Corresponds to the solution the lead time has the maximum uncertainty possible.
- Δss**:** It is the percentage variation of the safety stock between the *Mid* and the *Max* result.
- S_{av}**:** It is the average of the outbound service time considering all the supply chain.
- ***Stages w stock*:** It is the number of stages holding safety stock in the solution.
- ***St stock*:** Number of stages with stochastic lead time.
- ***N*:** Total number of stages.
- Δ$LT_{(av)}$**:** Relative variation of lead time between the *Mid* and the *Max* solution.

SC	Pipeline stock		Safety stock		Δss
	Mid	Max	Mid	Max	
1	26 443	35 530	984	1 054	7%
3	51 724	56 157	5 538	5 547	0%
5	4 200 255	4 953 114	641 824	687 722	7%
6	1 461 410	1 604 567	806	905	12%
7	2059	2 447	266	217	-19%
8	805 526	849 074	34 695	40 050	15%
9	3 348 803	3 443 173	109 054	115 494	6%
10	31 698	34 473	4 112	2 340	-43%
11	2 393	3 002	402	520	30%
12	9 921 038	10 589 143	1 260 297	1 270 778	1%
14	642 693	652 932	2 927	3 469	19%
15	1 684 500	2 536 831	511 227	627 161	23%
16	1 178 854	1 404 159	78 833	92 515	17%

Table 5.1: Stock results for the *r-nets*

SC	Δss cost	S_{av}		Stages w stock		Stages stoch. leadtime	N	$\Delta LT_{(av)}$
		Mid	Max	Mid	Max			
1	4%	0,0	0,0	5	5	1	8	79%
3	5%	28,4	33,9	8	7	8	17	37%
5	14%	1,0	1,1	22	22	9	27	51%
6	23%	0,4	0,4	22	22	16	28	88%
7	20%	6,1	11,7	19	21	38	38	25%
8	4%	39,1	42,8	3	3	23	40	22%
9	5%	12,4	13,1	24	25	11	49	22%
10	36%	2,3	23,5	52	37	21	58	35%
11	32%	4,4	6,7	36	44	45	68	86%
12	3%	4,9	5,5	75	75	28	88	27%
14	40%	3,5	4,9	55	66	36	116	97%
15	21%	1,6	2,5	77	77	77	133	52%
16	31%	3,	4,3	83	84	106	145	92%

Table 5.2: Cost, outbound service time, stages and lead time results for *r-nets*

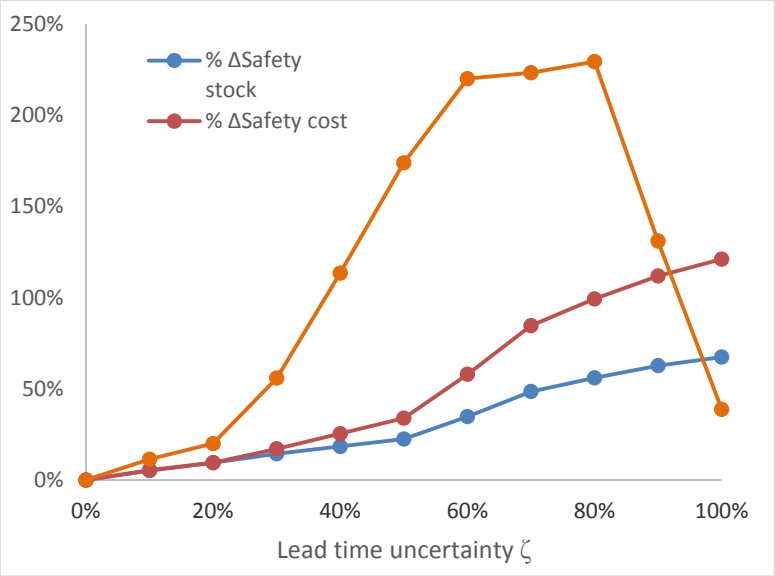
According to the *Table 5.1*, the total amount of pipeline stock increases always from the *Mid* to the *Max* case, as would have been expected intuitively. If the lead time is bigger, then the pipeline stock is too because it is function of the lead time and the demand. Regarding the total amount of safety stock, the trend is similar. It can be stated that generally the supply chains fight the uncertainty by increasing the safety stock – care must be taken because it is not a fixed rule. For instance in *SC* 07 and *SC* 10, the safety stock policy changes from more stock in stages with less holding cost to less stock in stages with more costly holding cost. That is the reason why the Δss is negative for these two systems while it is positive in the other cases. Either way, the cost rises when the supply chain faces the uncertainty from suppliers as it was expected. The numbers show that the cost related to the safety stock goes from 3% difference to a maximum of 40%. The number of stages with random lead time and the relative lead time variation should be considered to effectively compare the $\Delta ss\ cost$. For example, in *SC* 16, 106 out of 145 stages are characterized with random lead times, varying in average 92% and the increase of safety stock cost is 31%. The company should then consider if this over-cost is worth to take in order to avoid unmet demand, especially if it is taken into account that the safety stock costs represents, for this *SC*, just one fifth of the total stock cost.

The number of stages holding safety stock reacts in a similar way as the safety stock amount but with less changeability. From the results, it is concluded that *Stages w stock* does not vary (or at least not significantly) due to the lead time variability. In total, 481 stages hold safety stock for the *Mid* case and 488, seven stages more, for the worst-case scenario. Six supply chains remain the same regarding the number of stages holding safety stock, five increase the stock but two of them reduce the number of stages after applying the robust optimization. The two special cases are *SC* 03 and *SC* 10 even though the lead time considered is more than 30% bigger in average in both of situations.

Finally, the outbound service times are the other important issue to study because they are a measure of the service to the customer and a quota for the supply chain flexibility. The bigger the lead time variability so the supplier side uncertainty too, the longer the time needed for each stage of the *SC* to guarantee the same service level to its successors. A rule is impossible to make observing the numerical experiments. Anyhow, the relative variation is significantly different from one chain to another. In case that the outbound service time for an internal node does not seem realistic, feasible or acceptable for its customer, the companies can easily impose a maximum outbound service time constraint for internal nodes.

On the other hand, the *GS-RO* has been run 10 times with uncertainty going from 0 to 100% in intervals of 10% for the first six *d-nets* (*SC* 02, 04, 13, 17, 18 an 19). This work proceeded with this methodology because the robust approach cannot be directly implemented as stated before. The aim of the study

for *d-nets* was still the same as the *r-nets*, measuring how much does it affect uncertainty in terms of cost, safety stock and service time to the optimal solution when the lead time admits some random variability around its mean value. However, an aggregate analysis was less meaningful in this case, so in *Graphic 5.1* is shown the results for *SC* 04 and the possible conclusions to be extracted will be using this network as an example. The trend lines mean the variation of safety stock (blue), the variation of safety stock cost (orange) and the variation of the outbound service times in average (green).



Graphic 5.1: *SC* 04 results for the main outputs problem

For this particular network, the safety stock amount and its corresponding cost follow a similar trend. The difference between both lines began to be relevant around the 50% of the lead time uncertainty. The increase of the discrepancy is caused by the change in the safety stock policy. A few new safety stock units have been allocated in the most expensive stages. On the other side, the outbound service times experiment an uncommon tendency, having a significant growth for levels of 60% of uncertainty, then staggering and finally, dramatically decreasing. For a better understanding of this phenomenon, it should be noted that stages holding cost remains in 12 until 50% of uncertainty and then it scales to 19 stages for 60%, where it keeps constant.

Therefore, the ratification and the value of the Robust Guaranteed Service approach have been demonstrated. It is a powerful tool capable of immunizing the Guaranteed Service model against the supply uncertainty. In addition, the *GS-RO* permits to manage the risk by using a sensitive analysis that can discover how the outputs will behave.

Chapter 6

Guaranteed Service model under Random lead times (*GS-RAN*)

6.1. Model

In the previous chapter, the *GS-RO* has been proved to be a competent and powerful tool when it has been compared with the Guaranteed Service approach under deterministic lead time. In this one, the intention is also to demonstrate that makes more sense for the companies to solve the safety stock placement optimization with the *GS-RO* instead of the *GS-RAN*. For this purpose, the expressions proposed by *Humair et al. [2013]* are going to be used as the starting point of the theories developed in this chapter and the results are going to be useful for comparing both approaches (*GS-RO* versus *GS-RAN*).

The Guaranteed Service model under Random lead time incorporates an appropriate safety stock formula for the case of random lead times in a supply chain. It is a more realistic approach than the *GS-DET* but the same or even worse risks are taken by assuming a stochastic discrete or continuous distribution for the parameter. In addition to the not representative and not correct data risks, the occurrence of the early arrival stock (*ES*) phenomenon is an extra difficulty that has to be added to the model and contemplated in the objective function (more at *Humair et al. [2013]*).

The *ES* is the corresponding stock to a replenishment order arriving to the warehouse earlier than the associated customer order has been shipped. This type of stock has to be assumed by the same stage and it cannot be transferred to the downstream stages because the latter is not going to be willing to accept and deal with the associated holding cost of the extra and unnecessary units. This event can only happen when the outbound service time is bigger than the inbound service time plus the realized lead time so it is only possible when the net replenishment time is negative (more at *Humair et al. [2013]*).

6.2. Inconsistencies

The first steps for comparing *GS-RO* with *GS-RAN* were to analyze the results given by *Humair et al. [2013]* and to replicate the program proposed in the paper. Discrepancies with the authors were found in terms of results and expressions.

- **Data and results inconsistencies:** Firstly, the pipeline stock in *SC* 08 cannot change from the *GS-DET* using mean and *GS-RAN*. This is happening because the raw data for the stage *Part_01* from *SC* 08 is

incorrect. It is not possible that the lead time mean for this node is equal to 42 when 50% of the cases the lead time value is 1 and the other 50% is 45. Secondly, the total stock results are uncommon for network 09. It makes sense that the *GS-DET* using max lead time holds more total stock because it is the worst scenario possible than the *GS-RAN.* For this example, the numbers show the opposite situation so it is inconsistent.

- **Expressions inconsistencies:** The expression to minimize the safety stock under random lead time has to include the early arrival stock because every stage must find a compromise between both types of stock in this approach. Next, the equations for each stock given by *Humair et al. [2013]* are written:

$$SFTY_i = z_i \times \sqrt{Q(max\{S_i - SI_i, 0\}) \times \left(\sigma_i^d\right)^2 + \left(\mu_i^d\right)^2 \times R(max\{S_i - SI_i, 0\})} \quad (9)$$

$$EARLY_i = \mu_i^d \times (\mu_i^l - max\{S_i - SI_i, 0\} - Q(max\{S_i - SI_i, 0\})) \quad (10)$$

Where:

- ***Q(T):*** Expected value of the positive part of the NRLT when $T = max\{S_i - SI_i, 0\}$
- ***R(T):*** Variance of the positive part of the NRLT when $T = max\{S_i - SI_i, 0\}$

$$Q(T) = H_i^2(T) - T \times (1 - H_i^1(T)) \quad (11)$$

$$R(T) = T^2 \times H_i^1(T) \times \left(1 - H_i^1(T)\right) - 2 \times T \times H_i^1(T) \times H_i^2(T) + H_i^3(T) - (H_i^2(T))^2 \quad (12)$$

- $\boldsymbol{H_i^1(T)}$**:** Sum of the probabilities when the net replenishment time is negative.
- $\boldsymbol{H_i^2(T)}$**:** Expected lead time value when the net replenishment time is positive.
- $\boldsymbol{H_i^3(T)}$**:** Statistic formula for the positive net replenishment time.

The two disagreements found regarding the expressions are: the limits of *H(T)*'s functions and the behavior of the *SFTY* formula. Regarding the limits issue, the limit *T* is included in all three *Hs* formulas but this fact is not logical. *T* means a possible value for the lead time. When *T* is equal to the difference $S - SI$, then the *NRLT* is zero and the stage does not need to hold safety stock. However, when *T* is included as the lower bound for H_i^2 *and* H_i^3, it means that the previous assertion is being denied and the stage is being forced to own *SFTY*. To conclude, the expressions for H_i^2 *and* H_i^3 should be reformulated because one of the basic

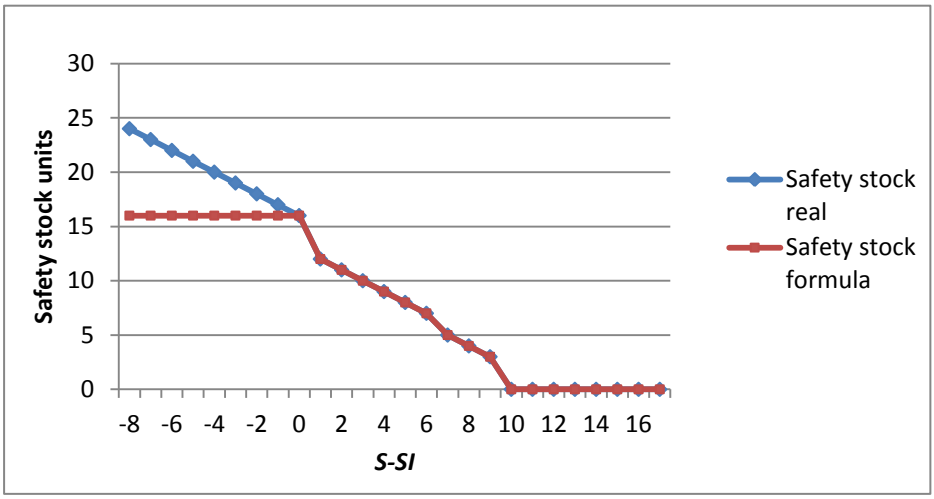
assumptions of the Guaranteed Service approach is that there is only safety stock when $NRLT > 0$. Thus, the equations would be:

$$H_i^1(T) = \sum_{l=0}^{T} P[L_i = l] \tag{13}$$

$$H_i^2(T) = \sum_{l=T+1}^{\infty} l \times P[L_i = l] \tag{14}$$

$$H_i^3(T) = \sum_{l=T+1}^{\infty} l^2 \times P[L_i = l] \tag{15}$$

Regarding the safety stock formula (9), the rising *SFTY* amount is not considered when the inbound is bigger than the outbound service time while the *NRLT* remains positive. *Graphic 6.1*, which is inspired from the figures in *Incorporating Stochastic Lead Times to the GS Model of Safety Stock Optimization*, shows how the *SFTY* and *ES* should theoretically behave depending on the service times (blue line) and displays what is happening with safety stock trend when the equation (9) is applied (red line).



Graphic 6.1: Safety on-hand stock for stochastic lead time

Thus, the *SFTY* becomes a constant when $SI > S$ and it is because the authors consider $T = max\{S_i - SI_i, 0\}$. This effect does not match with the safety stock theory due that extra safety stock is required to fight longer inbound service times when the delivery time to the customer is still the same. There is either missing information in *Humair et al. [2013]*, or the expression is not adequate for all the scenarios.

Therefore, a new expression (16) is proposed and used for the following numerical experiments to be sure that all the scenarios can be evaluated.

- The *SFTY* expression when $\sigma_i^l = 0$ is: (4).

- The equation when $SI \leq S$ is: (9).
- Finally, the formula when $SI > S$ is:

$$SFTY_i = z_i \times \sqrt{(\mu_i^l + SI_i - S_i) \times \left(\sigma_i^d\right)^2 + \left(\mu_i^d\right)^2 \times \left(\sigma_i^l\right)^2} \quad (16)$$

6.3. Numerical experiments

Taking advantage from the raw data error in supply chain 08, different values for *Part_01* have been tested to infer strengths and weaknesses of the *GS-RAN* and *GS-RO*. The varied experiments are:

- The robust approach with minimum and maximum uncertainty.
- The *GS* under random lead times with the raw data ($\mu^l_{Part_{01}} = 42, l_1 = 1\ [50\%]\ and\ l_2 = 45\ [50\%]$).
- The *GS-RAN* with ($\mu^l_{Part_{01}} = 42, l_1 = 40\ [50\%]\ and\ l_2 = 45\ [50\%]$) and ($\mu^l_{Part_{01}} = 23, l_1 = 1\ [50\%]\ and\ l_2 = 45\ [50\%]$).

	L_{Part_01}	$l_{i_{Part_{01}}}$	**Pipeline Stock**	**Early Stock**	**Safety Stock**	**Total Stock**	**Total Cost**
GS-RO ב = 0	42	[1,45]	805 525,71	-	34 694,94	840 220,65	16 780 600,60
GS-RO ב = 1	42	[1,45]	849 073,75	-	40 050,12	889 123,87	18 058 880,07
GS-RAN 1_45_42	42	[1,45]	793 786,70	42 483,14	46 030,77	882 300,61	15 035 832,90
GS-RAN 40_45_42	42	[40,45]	793 786,70	45 554,64	42 483,44	881 824,78	15 022 423,73
GS-RAN 1_45_23	23	[1,45]	764 604,50	71 662,39	46 030,77	882 297,66	15 035 832,90

Table 6.1: *GS-RO* and *GS-RAN* results for comparing both approaches

The early and the safety stock are opposites from the definition of both, the former happens when the net replenishment time is negative while the latter phenomenon occurs when the *NRLT* is positive. In general, the bigger the safety stock the smaller the early stock. This cannot be appreciated in *Table 6.1* because the data is incorrect in all *GS-RAN* cases. In any case, the *ES* is considered part of the stage stock in the *GS-RAN* model. This fact is what makes the straight comparison between the random and the robust model impossible due to the fact that the second one considers the *ES* in the pipeline stock. A better way to analyze the models is comparing the total stock and cost. Analyzing the total results, it can be stated that the robust analysis is the most conservative approach. A cheaper cost does not mean that the *GS-RAN* is the best model. In fact, *Table 6.1* shows that this approach is more conservative than the deterministic case but also the incapability of the model for detecting

the error has been proved. From the three random cases studied, the one representing the real chain seems to be *GS-RAN 40_45_42*. The other two situations differ less than 1 000 stock units so the cost is significantly similar in the three cases. Thus, the mistake made in this chain does not seem to cause a potential damage in the safety stock allocation. However, the *GS-RAN* cannot avoid or detect errors and it does not guarantee that the effect of the errors can be almost irrelevant as in this case. On the other hand, the robust approach is a way to immunize the chain against the data errors and it allows sensitive analysis so it is the most correct one to use to solve the problem with total confidence.

Chapter 7

Conclusions

The uncertainty on the supply side has been always a challenge for the inventory control and so then for the supply chain management too. The lead time variability has been one of the main concerns in this field, especially at the Guaranteed Service approach which basically assumes that the parameter is known and fixed.

A deterministic lead time was considered at the beginning of the *GS* model. However, past literature as *Humair et al. [2013]* and this thesis proved how this approach with the basic statement of fixed lead time is unrealistic; the results strongly differ from more representative results approaches and data errors cannot be detected.

Being conscious of this weakness, a program (validated using the *Moncayo-Martinez and Ramirez [2016]* tutorial and several supply chains from *Willems [2008]*) for the basic Guaranteed Service approach has been developed in this research to be able to test the reaction of the safety stock and the cost under different outbound service time situations. The study about the outbound service time can be conducted under deterministic case because the general trends of the outputs are not correlated to the way the lead time data is treated. The antagonist behavior of the outbound service time with the safety stock and the network cost is demonstrated. The shorter outbound service time in demand nodes and/or internal nodes, the tougher the customer is. The more demanding the customers are, the less flexible the supply chain is. Thus, higher amount of safety stock is needed to respond without stock outs to shorter service times and the network cost is more expensive then.

Following with the history of the *GS* approach literature, the work contributes with another codified program for the Random Guaranteed Service model which has been successfully tested in eleven supply chain cases from *Willems [2008]*. It will be a useful tool for practitioners and researchers in future work. In spite of the evidences of being a more conservative but accurate model for the safety stock allocation than the deterministic case, assuming a stochastic distribution for the lead time is not the correct way to deal with the key issue. The random approach cannot detect lead time value mistakes or incorrect distributions assumed, collecting data is still a challenge and furthermore, the data collected can be unreliable of the current or future reality so the risk is too high and it is difficult to manage.

This thesis proposes a new approach called the Robust Guaranteed Service to solve the *GS* under random lead times without assuming a determinate

stochastic distribution. Therefore, it is a model that enables to capture the variability of the lead time and the uncertainty of the supply side and at the same time, the safety stock placement is immunized against the data errors and challenges. It is the most conservative model from the three mentioned but its benefits outstands in comparison with the other two. It is protected against the errors, less investment in the collecting data challenge is required and less computational time is needed to solve the problem. In addition, the Robust Guaranteed approach is completely able to manage the risk; it allows the practitioners to decide how much risk are they willing to take and it helps them to take important inventory control decisions after analyzing the sensitive studies possible thank to uncertain lead time parameter of the model ($ב_i$).

In conclusion, the lead time should be considered variable to represent the reality but without assuming distributions to avoid data risks. The most efficient approach for this lead time treatment is the Robust Guaranteed Service approach for the arguments above-mentioned. It has been validated by testing it for twenty supply chains from *Willems [2008]* and it higher quality performance has been proved in comparison with the other two approaches.

Future work could extend the Robust Guaranteed Service model with capacities constraints and considering the cost of the operating flexibility allowed by the *GS* approach. Another possible future contribution can be testing the effectiveness of this approach in different business sectors and industries.

Bibliography

S. Axsäter. Supply chain operations: serial and distribution inventory systems. In: A.G. deKok and S.C. Graves (Eds.), *Handbook in Operations Reaseach and Science, Supply Chain Management: Design, Coordination and Operation.* Elsevier, Amsterdam, North-Holland, Chapter 11, 525-559, 2003.

S. Axsäter. Inventory Control. *Springer,* New York, 2006.

P. Beiran, M.P. Martin-Romero and H. Matsukawa. A Robust Guaranteed-Service Model to deal with Uncertain Lead Time in General Acyclic Supply Chains. *Department of Administration Engineering, Keio University*, 2017.

A. Ben-tal, L. Ghaoui and A. Nemirovski. *Robust optimization. Princeton University Press*, 2009.

D. Bertsimas and A. Thiele. A robust optimization approach to inventory theory. *Operations Research*, 54(1):150-168, 2006.

A.J. Clark and H. Scarf. Optimal policies for a multi-echelon inventory problem. *Operations Research*, 6(4):475-490, July 1960.

CSCO. Five strategies for Improving Inventory Management Across Complex Supply Chain Networks. Obtained from: *www.scdigest.com/assets/reps/exec brief network inventories.pdf*, June 2011.

E.B. Diks, A.G. deKok and A.G. Lagodimos. Multi-echelon systems: a service measure perspective. *European Journal of Operations Research*, 95(2):241-263, 1996.

G.D. Eppen and R.K. Martin. Determining safety stock in the presence of stochastic lead time and demand. *Management Science,* 34(11): 1380-1390, 1988.

S.C. Graves and S. Willems. Optimizing strategic safety stock placement in supply chains. *Manufacturing & Service Operations Management*, 1(2):68-83, Winter 2000.

S.C. Graves and S. Willems. Erratum: Optimizing strategic safety stock placement in supply chains. *Manufacturing & Service Operations Management*, 5(2):176-177, 2003.

S. Humair and S.P. Willems. Technical note. Optimizing strategic safety stock placement in general acyclic networks. Operations Research, 59(3): 781-787. 2011.

S. Humair, J.D. Ruark, B. Tomlin and S.P. Willems. Incoporating staochastic lead times into the guaranteed service model of safety stock optimization. *Interfaces*, 43(5):421-434, 2013.

K. Inderfurth. Safety stock optimization in multi-stage production systems. *International Journal of Production Economics*, 24:103-113, 1991.

K. Inderfurth and S. Minner. Safety stocks in multi-stage inventory systems under different service measures. *European Journal of Operations Research*, 106:57-73, 1998.

S.T. Klosterhalfen, D. Dittmar and S. Minner. An integrated guaranteed- and stochastic-service approach to inventory optimization in supply chains. *European Journal of Operation Research*, 231(1):109-119, 2013.

S.T. Klosterhalfen and S. Minner. Comparison of stochastic- and guaranteed-service approaches to safety stock optimization in supply chains. *Operations Research Proceedings 2006*, 485-490, 2007.

S.T. Klosterhalfen and S. Minner. Safety stock optimisation in distribution systems: a comparison of two competing approaches. *International Journal of Logistics: Research and Applications*, 13(2):99-120, 2010.

E. Lesnaia. Optimizing Safety Stock Placement in General Network Supply Chains. *Doctoral dissertation, Massachusetts Institute of Technology*, September 2004.

T.L. Maganti, Z.J.M. Shen, J.Shu, D. Simchi-Levi, C.P. Teo. Inventory placement in acyclic supply chain networks. *Operations Research Letters,* 34(2): 228-238, 2006.

S. Minner. *Strategic Safety Stocks in Supply Chains*, volume 490 of *Lecture Notes in Economics and Mathematical Systems*. Springer, 2000.

L.A. Moncayo-Martinez and A. Ramirez. A tutorial to set safety stock under guaranteed-service time by dynamic programming. *International Journal of Industrial and Systems Engineering*, October 2016.

J. Rambau and K. Schade. The stochastic guaranteed service model with recourse for multi-echelon warehouse management. *Science Direct, Electronic Notes in Discrete Mathematics*, 36:783–790, 2010.

D. Simchi-Levi and Y. Zhao. Performance evaluation of stochastic multi-echelon inventory systems: A survey. Adv. Operational Reasearch, 1-34, 2012.

K.F. Simpson. In-process inventories. *Operations Research*, 6:863-873, 1958.

S.P. Willems. Data set-real-world multiechelon supply chains used for inventory optimization. *Manufacturing Service Operations Management*, 10(1):19-23, 2008.

Acknowledges

I would like to express my sincere gratitude to my supervisor, Prof. Matsukawa, for giving me the opportunity to research at his Production & Logistics Laboratory, for his wise advices, his patience and immense knowledge.

I would also like to thank to my lab mates and friends for their support and friendship, and made my experience at Keio University impressive. I would specially acknowledge my lab mate Pablo Beiran for his continuous support, for his patience, guidance during the semester and for sharing his first research steps with me. I would like to express the honor it has been for me to work in the same team and develop the Robust Guaranteed Service approach together.

Finally, I would like to thank the support of my family: my parents and close friends. They were handling my bad temperament in my worst days, being compressive with no having enough time to share with me, giving me strength and encouragement to reach my goals and dreams, providing me with financial support and continuously supporting and motivating me.

Appendix 1

Program for solving the *Moncayo-Martinez and Ramirez [2016]* tutorial

1) The modules are imported and the data available is read from excel.

```
#modules
import xlrd
import math
##read excel
book=xlrd.open_workbook("C:/Users/Usuario/Dropbox/KEIO/MASTER
THESIS/Programing/tutorial_Willems2000/tutorial.xlsx")
arcs=book.sheet_by_name("T_LL")
data=book.sheet_by_name("T_SD")
```

2) The data variables are initiated.

```
stages={}          #name and labeling
avgD = {}          #mean demand
varD = {}          #variance of demand
stdD = {}          #deviation of demand
Cs = {}            #cost stage (not holding cost)
h=0.2              #annual holding cost
CC={}              #cumulative cost, holding cost
service = {}       #service level
z={}               #normsinv(service level)
lt_av = {}         #lead time mean
lt_std = {}        #standard deviation lead time #Lead time is
                   deterministic, lt_std=0
M={}               #replenishment time
s_out_req={}       #service time required by the last customer
Ve=[]              #demand nodes
Vs=[]              #supply nodes
Vs_sin_ord=[]      #supply node without the spanning tree labeling
orderStages=[]     #relation between excel and spanning tree labeling
```

3) Transfer data from excel to python.

```
#Total number of stages
N=len(data.col_values(0))-1
#transfer data from excel to python. (excel order data)
for row in range(N):
    stages[row+1]= [data.cell_value(row+1,0), 0]
    Cs[row+1]= data.cell_value(row+1,1)
    avgD[row+1]= data.cell_value(row+1,4)
    stdD[row+1]= data.cell_value(row+1,5)
```

```
    service[row+1]=data.cell_value(row+1,7)
    lt_av[row+1]=data.cell_value(row+1,8)
    lt_std[row+1]=data.cell_value(row+1,9)
    s_out_req[row+1]=data.cell_value(row+1,6)
    if avgD[row+1]=='':
        avgD[row+1]=0
        stdD[row+1]=0
    varD[row+1]= stdD[row+1]**2
    if lt_std[row+1]=='':
        lt_std[row+1]=0
```

4) Matrices with the relation between stages.

```
#Arcs matrix #Initial matrix dim(N*N) with all values equal zero
arc_matrix = [[0] * (len(data.col_values(0))-1) for i in range(len(data.col_values(0))-1)]
#Arc_matrix (with all the arcs, relations between stages)
for i in range(1,len(arcs.col_values(0))):
    val_extra_1=arcs.cell_value(i,0)
    val_extra_2=arcs.cell_value(i,1)
arc_matrix[stages.values().index([val_extra_1,0])][stages.values().index([val_extra_2,0])]=1

#Arc_matrix_aux1 (equal to arc_matrix)(arc_matrix_aux1 is an auxiliar matrix)
arc_matrix_aux1 = [[0] * (len(data.col_values(0))-1) for i in range(len(data.col_values(0))-1)]
for i in range(1,len(arcs.col_values(0))):
    val_extra_1=arcs.cell_value(i,0)
    val_extra_2=arcs.cell_value(i,1)

arc_matrix_aux1[stages.values().index([val_extra_1,0])][stages.values().index([val_extra_2,0])]=1
```

5) Maximum Replenishment Time algorithm.

```
#Maximum Replenishment time. (excel order)
memory_del=[]
arc_matrix_rp=[[0] * (len(data.col_values(0))-1) for i in range(len(data.col_values(0))-1)]
arc_matrix_aux=[[0] * (len(data.col_values(0))-1) for i in range(len(data.col_values(0))-1)]
for i in range(N):
    for j in range(N):
        arc_matrix_rp[i][j]=arc_matrix[i][j]
```

```
memory_del=[]

while (len(memory_del)!=N):
    for i in [x for x in xrange(N) if x not in memory_del]:
        if sum([arc_matrix_rp[j][i]==1 for j in range(N)])==0: #The algorithm is looking for columns with all zeros (no predecessors)
            M[i+1]=lt_av[i+1]+max(arc_matrix_aux[i])
            for j in range(N):
                if arc_matrix_rp[i][j]==1:
                    arc_matrix_aux[j][i]=M[i+1]      #The algorithm uses arc_matrix_aux to create the Mmax (it would be the max of the row)
                    arc_matrix_rp[i][j]=0
            memory_del=memory_del+[i]
```

6) Demand propagation algorithm. The propagation is done as simple as possible (more in *Framework* section).

```
#Demand Propagation (1 successor for all stages)
#Work in arc_matrix_demand that will be modified during the algorithm
arc_matrix_demand=[[0] * (len(data.col_values(0))-1) for i in range(len(data.col_values(0))-1)]
for i in range(N):
    for j in range(N):
        arc_matrix_demand[i][j]=arc_matrix[i][j]

memory_del=[]
while(arc_matrix_demand != [[0] * (len(arc_matrix)) for i in range(len(arc_matrix_demand))]):    #running until arc_matrix_demand becomes zero
    for row in [x for x in xrange(len(arc_matrix_demand)) if x not in memory_del]:                #It selects row with no 1 (demand nodes)
        if arc_matrix_demand[row] == ([0]*N):
            for i in range(len(arc_matrix_demand)):
                if arc_matrix_demand[i][row]==1:
                    avgD[i+1]=avgD[i+1]+avgD[row+1]
                    varD[i+1]=varD[i+1]+varD[row+1]
            memory_del=memory_del+[row]
#Values already treated are changed to zeros
    for aa in memory_del:
        for bb in range(len(arc_matrix_demand)):
            if arc_matrix_demand[bb][aa]==1:
                arc_matrix_demand[bb][aa]=0
```

7) Algorithms for other data variables.

```
#Dictionary with z=normsinv(service_level)
for k in range(N):
    z[k+1]=1.64

#List with supply nodes (no predecessors)
for i in range(N):
    if sum([arc_matrix[j][i]==1 for j in range(N)])==0: #if the column is all zeros, then it is a supply node
        Vs_sin_ord=Vs_sin_ord+[i+1]
#Cumulative cost
for i in range(N):
    if (i+1) in Vs_sin_ord:
        CC[i+1]=Cs[i+1]
    else:
        cumulative=0
        for j in range(i):
            if arc_matrix[j][i]==1:
                cumulative=cumulative+CC[j+1]
        CC[i+1]=Cs[i+1]+cumulative
```

8) Code for the Spanning Tree algorithm proposed by *Graves and Willems [2000].*

```
k=1
while(arc_matrix_aux1 != [[0] * (len(arc_matrix_aux1)) for i in range(len(arc_matrix_demand))]):
    for i in range(N):
        if sum(arc_matrix_aux1[i]) + sum([arc_matrix_aux1[j][i]==1 for j in range(len(arc_matrix_aux1))])==1:
            stages[i+1][1]=k
            k=k+1
            for j in range(len(arc_matrix_aux1)):
                arc_matrix_aux1[j][i]=0
            arc_matrix_aux1[i]=[0]*(len(arc_matrix_aux1))

for i in range(k,len(arc_matrix_aux1)+1):
    for j in range(N):
        if stages[j+1][1]==0:
            stages[j+1][1]=k
            k=k+1
```

9) Reordering the stages according the spanning tree labeling and generating the new demand and supply nodes.

```
#Arc matrix ordered and its corresponding spanning tree for the new labeling
arc_matrix_ord=[[0] * (len(data.col_values(0))-1) for i in range(len(data.col_values(0))-1)]

for i in range(1,len(arcs.col_values(0))):
    val_extra_1=arcs.cell_value(i,0)
    val_extra_2=arcs.cell_value(i,1)
    for j in range(1,N+1):
        if stages[j][0]==val_extra_1:
            break
    for k in range(1,N+1):
        if stages[k][0]==val_extra_2:
            break
    arc_matrix_ord[stages[j][1]-1][stages[k][1]-1]=1

#List with demand nodes
for k in range(N):
    if sum(arc_matrix_ord[k])==0:
        Ve=Ve+[k+1]

#List with initial nodes (no predecessors)
for i in range(N):
    if sum([arc_matrix_ord[j][i]==1 for j in range(N)])==0: #if the column is all zeros, then it is a supply node
        Vs=Vs+[i+1]
```

10) Expressions for the objective function.

```
#Safety stock
def SS(k,SI,S):
    posk=orderStages.index(k)
    if SI+lt_av[posk+1]-S>=0:
        return
z[posk+1]*math.sqrt(varD[posk+1])*math.sqrt(lt_av[posk+1]+SI-S)
    else:
        print "no possible" #There is no stock if NRLT is negative
#Cost function
def cost(k,SI,S):
    posk=orderStages.index(k)
    return h*CC[posk+1]*SS(k,SI,S)
```

11) $f(k,S,stageInfo)$ is the expression for the minimum cost of the sub-graph when the unique adjacent node with higher label is the **customer**

of k (as *Humair and Willems [2011]* wrote: let the function $f(k,S,stageInfo)$ be the optimal cost of the sub-tree connected to k after removing its unique adjacent node with higher label if k's **outgoing** service time is S). k is the stage/node, S is the given outbound service and $stageInfo$ is the current dictionary where the possible results from the nodes with less label have been stored.

```
def f(k,S,stageInfo):
    SIResults=[]
    ZResultF=[]
    posk=orderStages.index(k)

    for si in range(int(max(0,S-lt_av[posk+1])),int(M[posk+1]-
lt_av[posk+1]+1)):
        cF=cost(k,si,S)
        prevF=0.0
        nextF=0.0
```

```
for i in range(k):
    m1=[]
    if arc_matrix_ord[i][k-1]==1:
        listNF=stageInfo[i+1]
        listS=[]
        for nf in range(len(listNF)):
            listS=listS+[listNF[nf][2]]

        for nf in range(len(listNF)):
            if listNF[nf][2]<=si and listNF[nf][2]>=0:
                m1=m1+[listNF[nf][0]]
            elif si>max(listS):
                m1=m1+[float("inf")]
        if m1!=[]:
            Min1=min(m1)
            prevF=prevF+Min1
```

Evaluation of the lower label 'previous' nodes cost of k.

```
for j in range(k):
    m2=[]
    if arc_matrix_ord[k-1][j]==1:
        posj=orderStages.index(j+1)
        listNF=stageInfo[j+1]
        listSI=[]
        for nf in range(len(listNF)):
            listSI=listSI+[listS[nf][1]]
```

Evaluation of the lower label 'next' nodes cost of k.

```
            for nf in range(len(listNF)):
                if listNF[nf][1]>=S and listNF[nf][1]<=(M[posj+1]-
lt_av[posj+1]):
                    m2=m2+[listNF[nf][0]]
                elif S>max(listSI):
                    m2=m2+[float("inf")]
            if m2!=[]:
                Min2=min(m2)
                nextF=nextF+Min2

        ResultF=cF+prevF+nextF
        SIResults=SIResults+[(ResultF,si,S)]
        ZResultF=ZResultF+[ResultF]

    ResultFinalF=min(ZResultF)
    PosResultFinalF=ZResultF.index(ResultFinalF)

TupleFinalF=[(ResultFinalF,SIResults[PosResultFinalF][1],SIResults[Pos
ResultFinalF][2])]
    return TupleFinalF
```

12) g$(k, SI, stageInfo)$ is the expression for the minimum cost of the sub-graph when the unique adjacent node with higher label is the **supplier** of k (as *Humair and Willems [2011]* wrote: let the function $g(k, SI, stageInfo)$ be the optimal cost of the sub-tree connected to k after removing its unique adjacent node with higher label if k's **incoming** service time is SI). k is the stage/node, SI is the given outbound service and $stageInfo$ is the current dictionary where the possible results from the nodes with less label have been stored.

```
def g(k,SI,stageInfo):
    SResults=[]
    ZResultG=[]
    posk=orderStages.index(k)

    if k in Ve:
        smax=s_out_req[posk+1]+1
    else:
        smax=SI+lt_av[posk+1]+1

    for s in range(0,int(smax)):
        cG=cost(k,SI,s)
        prevG=0.0
        nextG=0.0
```

```
for i in range(k):
    m1=[]
    if arc_matrix_ord[i][k-1]==1:
        listNG=stageInfo[i+1]
        listS=[]
        for nf in range(len(listNG)):
            listS=listS+[listNG[nf][2]]

        for nf in range(len(listNG)):
            if listNG[nf][2]<=SI and listNG[nf][2]>=0:
                m1=m1+[listNG[nf][0]]
            elif SI>max(listS):
                m1=m1+[float("inf")]
        if m1!=[]:
            Min1=min(m1)
            prevG=prevG+Min1
```

Evaluation of the lower label ‘previous’ nodes cost of k.

```
    for j in range(k):
        m2=[]
        if arc_matrix_ord[k-1][j]==1:
            posj=orderStages.index(j+1)
            listNG=stageInfo[j+1]
            listSI=[]
            for nf in range(len(listNG)):
                listSI=listSI+[listNG[nf][1]]

          for nf in range(len(listNG)):
                if listNG[nf][1]>=s and listNG[nf][1]<=(M[posj+1]-
lt_av[posj+1]):
                    m2=m2+[listNG[nf][0]]
                elif s>max(listSI):
                    m2=m2+[float("inf")]
            if m2!=[]:
                Min2=min(m2)
                nextG=nextG+Min2
```

Evaluation of the lower label next nodes cost of k.

```
    ResultG=cG+prevG+nextG
    SResults=SResults+[(ResultG,SI,s)]
    ZResultG=ZResultG+[ResultG]

ResultFinalG=min(ZResultG)
PosResultFinalG=ZResultG.index(ResultFinalG)
```

```
TupleFinalG=[(ResultFinalG,SResults[PosResultFinalG][1],SResults[Pos
ResultFinalG][2])]
    return TupleFinalG
```

13) $model(\ )$ function is used to create all the possible stage cost for all the node.
First, it chooses between f$(k,S,stageInfo)$ or g$(k,SI,stageInfo)$ which one has to be selected for each node. Second, it evaluates the selected function for the concrete service time range of values. Third, every evaluation is cached in the $stageInfo$ dictionary.

```
def model():
  stageInfo={}
  for k in range(N):
    posk=orderStages.index(k+1)
    if (k+1)!= N:          #all stages calculation
      TupleStage=[] #the results for each node will be stored at
TupleStage
      for j in range(N):
        if k<j:
          if arc_matrix_ord[k][j]==1:
            for S in range(0,int(M[posk+1]+1)):
              TupleStage=TupleStage+f(k+1,S,stageInfo)
            stageInfo[k+1]=TupleStage

          if arc_matrix_ord[j][k]==1:
            for SI in range(0,int(M[posk+1]-lt_av[posk+1])+1):
              TupleStage=TupleStage+g(k+1,SI,stageInfo)
            stageInfo[k+1]=TupleStage
      else:
        LastTupleStage=[]
        for SI in range(0,int(M[posk+1]-lt_av[posk+1])+1):
          LastTupleStage=LastTupleStage+g(k+1,SI,stageInfo)
        stageInfo[k+1]=LastTupleStage
    return stageInfo
```

If the unique higher adjacent node of k is its customer, then f function is evaluated for the corresponding S values

If the unique higher adjacent node of k is its supplier, then g function is evaluated for the corresponding SI values

The last stage possible costs are evaluated with g function

14) $backtrack(stageInfo)$ is a function used for finding the optimal cost of the whole supply chain and the best outbound and inbound service time for each stage that fulfill all the constraints. $stageOpt$ is where the cost depending on the best services time found, the best incoming and outcoming service times are stored for each stage. $Zopt$ is the optimal cost for the supply chain. It is important to note that the main goal is to

resolve the optimization for the whole supply chain so it happens that sometimes the cost of one stage is not the optimal for itself. In conclusion, the program does not work with locals, it works with global optimization.

```
SET={}
stageOpt={}

def backtrack(stageInfo):
    for k in range (N, 0, -1):
        if k==N:
            Alternatives=stageInfo[k]
            Min3=[]

            for a in Alternatives:
                Min3=Min3+[a[0]]
            Zopt=min(Min3)
            posZopt=Min3.index(Zopt)
            SET[k]=(Alternatives[posZopt][1],Alternatives[posZopt][2])
#set=(SI,S)

        else:
            Alternatives=stageInfo[k]
            Alter=[]
            for j in range(N):
                if (k-1)<j:
                    if arc_matrix_ord[k-1][j]==1:
                        limit=SET[j+1]
                        SIlimit=limit[0]
                        Min4=[]

                        for i in range(len(Alternatives)):
                            if Alternatives[i][2]<=SIlimit:
                                Alter=Alter+[Alternatives[i]]
                            else:
                                Alter=Alter

                        for i in Alter:
                            Min4=Min4+[i[0]]
                        fog=min(Min4)
                        posfog=Min4.index(fog)
                        SET[k]=(Alter[posfog][1],Alter[posfog][2])
```

The optimal cost for the supply chain is found

When the stage k is the supplier of j, then the constraint $S_k \leq SI_j$ has to be fulfilled.

```
if arc_matrix_ord[j][k-1]==1:
  limit=SET[j+1]
  Slimit=limit[1]
  Min5=[]

  for i in range(len(Alternatives)):
     if Alternatives[i][1]>=Slimit:
          Alter=Alter+[Alternatives[i]]

     else:
         Alter=Alter

  for i in Alter:
     Min5=Min5+[i[0]]
  fog=min(Min5)
  posfog=Min5.index(fog)
  SET[k]=(Alter[posfog][1],Alter[posfog][2])
```

When the stage k is the customer of j, then the constraint $SI_k \geq S_j$ has to be fulfilled.

```
Zopt=0.0
for k in range(N):
   Zopt=Zopt+cost(k+1,SET[k+1][0],SET[k+1][1])
```

The optimal cost is calculated

```
stageOpt[k+1]=(cost(k+1,SET[k+1][0],SET[k+1][1]),SET[k+1][0],SET[k+1][1])
   print Zopt,stageOpt
```

15) To run the program and obtain the outputs.

```
stageInfo=model()
Zopt,stageOpt=backtrack(stageInfo)
```

Appendix 2

Program for optimizing the safety stock placement in General Acyclic Chains with deterministic lead time

1) The modules are imported and the data available is read from excel

```
#modules
import xlrd
import math
from scipy.sparse import csr_matrix
from scipy.sparse.csgraph import minimum_spanning_tree
from scipy.stats import norm
#read data
book=xlrd.open_workbook("C:\Users\Usuario\Dropbox\KEIO\MASTER THESIS\Programing\Data Set_org.xls")
arcs=book.sheet_by_name("12_LL") #supply chain #
data=book.sheet_by_name("12_SD") #supply chain #
```

2) The data variables and the limits are initiated.

```
stages={}                #name and labeling
avgD = {}                #mean demand
varD = {}                 #variance of demand
stdD = {}                #deviation of demand
Cs = {}                  #cost stage (not holding cost)
h=1                      #annual holding cost
CC={}                    #cumulative cost
service = {}             #service level
z={}                     #normsinv(service level)
lt_av = {}               #lead time mean without decimals
lt_av_decimal={}         ##lead time mean with decimals
lt_std = {}              #standard deviation lead time N
lt_var={}                #auxiliary
lt_max={}                #auxiliary
M={}                     #replenishment time
s_out_req={}             #service time required by the last customer
stage_lk={}              #auxiliary
prob_lk={}               #auxiliary
Ve=[]                    #demand nodes
Vs=[]                    #supply nodes
Vs_sin_ord=[]            #supply node without spanning tree labeling
orderStages=[]           #relation between excel and spanning tree labeling

#limits initialization
```

```
u={}
l={}
U={}
L={}
```

3) Transfer data from excel to python.

```
#Total number of stages
N=len(data.col_values(0))-1

#Transfer data from excel to python. (excel order data)
for row in range(N):
    stages[row+1]= [data.cell_value(row+1,0), 0]
    Cs[row+1]= data.cell_value(row+1,1)
    avgD[row+1]= data.cell_value(row+1,4)
    stdD[row+1]= data.cell_value(row+1,5)
    service[row+1]=data.cell_value(row+1,7)
    lt_av[row+1]=data.cell_value(row+1,8)
    lt_av_decimal[row+1]=data.cell_value(row+1,8)
    if (lt_av[row+1]/0.5)%2==1:
        lt_av[row+1]=lt_av[row+1]-0.01
    lt_av[row+1]=round(lt_av[row+1])
    lt_std[row+1]=data.cell_value(row+1,9)
    s_out_req[row+1]=data.cell_value(row+1,6)
    if avgD[row+1]=='':
        avgD[row+1]=0
        stdD[row+1]=0
    varD[row+1]= stdD[row+1]**2
    if lt_std[row+1]=='':
        lt_std[row+1]=0

    stage_lk[row+1]=[data.cell_value(row+1,c) for c in range (10,22,2)]
    prob_lk[row+1]=[data.cell_value(row+1,c) for c in range (11,23,2)]
    for i in range(len(stage_lk[row+1])):
        if stage_lk[row+1][i]=='':
            stage_lk[row+1][i]=0
            prob_lk[row+1][i]=0
    if stage_lk[row+1]==[0]*6 and lt_std[row+1]=='':
        stage_lk[row+1][0]=lt_av[row+1]
        prob_lk[row+1][0]=1
    if stage_lk[row+1]==[0]*6 and lt_std[row+1]!='':
        stage_lk[row+1][0]=lt_av[row+1]
        lt_var[row+1]=3*lt_std[row+1]
    for i in range(len(stage_lk[row+1])-1,0, -1):
        if stage_lk[row+1][i]==0:
```

```
            del stage_lk[row+1][i]
    lt_max[row+1]=max(stage_lk[row+1])
```

4) Matrices with the relation between stages.

```
#Arcs matrix. #Initial matrix dim(N*N) with all values equal zero
arc_matrix = [[0] * (len(data.col_values(0))-1) for i in range(len(data.col_values(0))-1)]

#Arc_matrix (with all the arcs, relations between stages)
for i in range(1,len(arcs.col_values(0))):
    val_extra_1=arcs.cell_value(i,0)
    val_extra_2=arcs.cell_value(i,1)

arc_matrix[stages.values().index([val_extra_1,0])][stages.values().index([val_extra_2,0])]=1

#Arc_matrix_aux1 (equal to arc_matrix)(arc_matrix_aux1 is an auxiliar matrix)
arc_matrix_aux1 = [[0] * (len(data.col_values(0))-1) for i in range(len(data.col_values(0))-1)]
for i in range(1,len(arcs.col_values(0))):
    val_extra_1=arcs.cell_value(i,0)
    val_extra_2=arcs.cell_value(i,1)

arc_matrix_aux1[stages.values().index([val_extra_1,0])][stages.values().index([val_extra_2,0])]=1
```

5) Maximum Replenishment Time algorithm.

```
#Maximum Replenishment time. (excel order)
memory_del=[]
arc_matrix_rp=[[0] * (len(data.col_values(0))-1) for i in range(len(data.col_values(0))-1)]
arc_matrix_aux=[[0] * (len(data.col_values(0))-1) for i in range(len(data.col_values(0))-1)]
for i in range(N):
    for j in range(N):
        arc_matrix_rp[i][j]=arc_matrix[i][j]

memory_del=[]
while (len(memory_del)!=N):
    for i in [x for x in xrange(N) if x not in memory_del]:
        if sum([arc_matrix_rp[j][i]==1 for j in range(N)])==0: #The algorithm looks for columns with all zeros (no predecessors)
```

```
        M[i+1]=lt_av[i+1]+max(arc_matrix_aux[i])
        for j in range(N):
            if arc_matrix_rp[i][j]==1:
                arc_matrix_aux[j][i]=M[i+1]    #The algorithm uses
arc_matrix_aux to create the Mmax (it would be the max of the row)
                arc_matrix_rp[i][j]=0
        memory_del=memory_del+[i]
```

6) Initial values for the limits. The service times limits are up to the programmer. In this case, the limits have been fathomed from the definition of maximum replenishment time and from the constraints 6a. and 6b. In other cases, the range of possible values can be selected using other criteria by the researcher.

```
for i in range(N):
    l[i+1]=0
    L[i+1]=0
    U[i+1]=M[i+1]-lt_av[i+1]
    if s_out_req[i+1]=='':
        u[i+1]=M[i+1]
    else:
        u[i+1]=min(s_out_req[i+1],M[i+1])
```

7) Demand propagation algorithm. The propagation is done as simple as possible (more in *Framework* section).

```
arc_matrix_demand=[[0] * (len(data.col_values(0))-1) for i in
range(len(data.col_values(0))-1)]
for i in range(N):
    for j in range(N):
        arc_matrix_demand[i][j]=arc_matrix[i][j]

memory_del=[]
while(arc_matrix_demand != [[0] * (len(arc_matrix)) for i in
range(len(arc_matrix_demand))]):    #running until arc_matrix_demand
becomes zero
    for row in [x for x in xrange(len(arc_matrix_demand)) if x not in
memory_del]:                    #It selects row with no 1 (demand
nodes)
        if arc_matrix_demand[row] == ([0]*N):
            for i in range(len(arc_matrix_demand)):
                if arc_matrix_demand[i][row]==1:
                    avgD[i+1]=avgD[i+1]+avgD[row+1]
                    varD[i+1]=varD[i+1]+varD[row+1]
            memory_del=memory_del+[row]
```

```
#The zeros already treated are changed
    for aa in memory_del:
        for bb in range(len(arc_matrix_demand)):
            if arc_matrix_demand[bb][aa]==1:
                arc_matrix_demand[bb][aa]=0
```

8) Algorithms for other data variables.

```
#Supply nodes of the excel labeling
for i in range(N):
    if sum([arc_matrix[j][i]==1 for j in range(N)])==0: #if the column is all zeros, then it is a supply node
        Vs_sin_ord=Vs_sin_ord+[i+1]

#Cumulative cost
memory_del=[]
arc_matrix_cc=[[0] * (len(data.col_values(0))-1) for i in range(len(data.col_values(0))-1)]
arc_matrix_aux=[[0] * (len(data.col_values(0))-1) for i in range(len(data.col_values(0))-1)]
for i in range(N):
    for j in range(N):
        arc_matrix_cc[i][j]=arc_matrix[i][j]

memory_del=[]
while (len(memory_del)!=N):
    for i in [x for x in xrange(N) if x not in memory_del]:
        if sum([arc_matrix_cc[j][i]==1 for j in range(N)])==0:
            CC[i+1]=Cs[i+1]+sum(arc_matrix_aux[i])
            for j in range(N):
                if arc_matrix_cc[i][j]==1:
                    arc_matrix_aux[j][i]=CC[i+1]
                arc_matrix_cc[i][j]=0
            memory_del=memory_del+[i]

#service level coefficient
#Calculation of the average service level coefficient (for those with no value assigned)
count_service=0
sum_service=0
for k in range (N):
    if service[k+1]!='':
        count_service += 1
        sum_service += service[k+1]
```

```
#The average is assigned to those stages with no value assigned
for k in range (N):
    if service[k+1]=='':
        service[k+1]=sum_service/count_service

#Dictionary with z=normsinv(service)
for k in range(N):
    z[k+1]=norm.ppf(service[k+1])
```

9) Generation and labeling of the spanning tree. An acyclic network is not a spanning tree. The arcs are ranked considering its associated cost (more at *Humair and Willems [2011]*) to choose a spanning tree from the supply chain. Once a minimum spanning tree is selected, the stages are labeled taking into account the broken links. Finally, matrices are created to store all the spanning tree information.

```
#Spanning tree
#Matrix with links and costs for choosing the spanning tree
arc_matrix_costs=[[0] * (len(data.col_values(0))-1) for i in
range(len(data.col_values(0))-1)]
for i in range(N):
    for j in range(N):
        arc_matrix_costs[i][j]=arc_matrix[i][j]

#The associate arc (i,j) in A cost= -(Ci+Cj)*stdj
for i in range(len(arc_matrix_costs)):
    for j in range(len(arc_matrix_costs)):
        if arc_matrix_costs[i][j]==1 and CC[i+1]!=0 and CC[j+1]!=0:
            arc_matrix_costs[i][j]=-
100*(CC[i+1]+CC[j+1])*(math.sqrt(varD[j+1]))

#Generation of the spanning tree
X = csr_matrix(arc_matrix_costs)
Tcsr = minimum_spanning_tree(X)
arc_matrix_costs=Tcsr.toarray().astype(int)

#Matrix with the spanning tree
arc_matrix_st=[[0] * (len(data.col_values(0))-1) for i in
range(len(data.col_values(0))-1)]
for i in range(len(arc_matrix_costs)):
    for j in  range(len(arc_matrix_costs)):
        if arc_matrix_costs[i][j]!=0:
            arc_matrix_st[i][j]=1
```

```
#Matrix storing the links that have been broken for creating the spanning tree
arc_matrix_brokenlinks=[[0] * (len(data.col_values(0))-1) for i in range(len(data.col_values(0))-1)]
for i in range(N):
    for j in  range(N):
        arc_matrix_brokenlinks[i][j]=arc_matrix[i][j]-arc_matrix_st[i][j]

#Labeling the spanning tree in stages[i][1]
#Initial value for the counter
k=1
while(arc_matrix_st != [[0] * (len(arc_matrix)) for i in range(len(arc_matrix_demand))]):
    for i in range(N):
        if sum(arc_matrix_st[i]) + sum([arc_matrix_st[j][i]==1 for j in range(len(arc_matrix_st))])==1:
            stages[i+1][1]=k
            k=k+1
            for j in range(len(arc_matrix_st)):
                arc_matrix_st[j][i]=0
            arc_matrix_st[i]=[0]*(len(arc_matrix))

for i in range(k,len(arc_matrix_st)+1):
    for j in range(N):
        if stages[j+1][1]==0:
            stages[j+1][1]=k
            k=k+1

#Spanning tree ordered cost (spanning tree)
arc_matrix_ordered_cost=[[0] * (len(data.col_values(0))-1) for i in range(len(data.col_values(0))-1)]

for i in range(1,len(arcs.col_values(0))):
    val_extra_1=arcs.cell_value(i,0)
    val_extra_2=arcs.cell_value(i,1)
    for j in range(1,N+1):
        if stages[j][0]==val_extra_1:
            break
    for k in range(1,N+1):
        if stages[k][0]==val_extra_2:
            break
    arc_matrix_ordered_cost[stages[j][1]-1][stages[k][1]-1]=arc_matrix_costs[j-1][k-1]
```

```
#The arc_matrix_st is saved again
arc_matrix_st=[[0] * (len(data.col_values(0))-1) for i in range(len(data.col_values(0))-1)]
for i in range(len(arc_matrix_costs)):
    for j in  range(len(arc_matrix_costs)):
        if arc_matrix_costs[i][j]!=0:
            arc_matrix_st[i][j]=1

#The arc matrix ordered is generated and its corresponding spanning tree
arc_matrix_ordered=[[0] * (len(data.col_values(0))-1) for i in range(len(data.col_values(0))-1)]

for i in range(1,len(arcs.col_values(0))):
    val_extra_1=arcs.cell_value(i,0)
    val_extra_2=arcs.cell_value(i,1)
    for j in range(1,N+1):
        if stages[j][0]==val_extra_1:
            break
    for k in range(1,N+1):
        if stages[k][0]==val_extra_2:
            break
    arc_matrix_ordered[stages[j][1]-1][stages[k][1]-1]=1

arc_matrix_ordered_st=[[0] * (len(data.col_values(0))-1) for i in range(len(data.col_values(0))-1)]
for i in range(N):
    for j in range(N):
        arc_matrix_ordered_st[i][j]=arc_matrix_ordered[i][j]
for i in range(N):
    for j in range(N):
        if arc_matrix_brokenlinks[i][j]==1:
            arc_matrix_ordered_st[stages[i+1][1]-1][stages[j+1][1]-1]=0

arc_matrix_ordered_brokenlinks=[[0] * (len(data.col_values(0))-1) for i in range(len(data.col_values(0))-1)]
for i in range(N):
    for j in range(N):
        arc_matrix_ordered_brokenlinks[i][j]=arc_matrix_ordered[i][j]-arc_matrix_ordered_st[i][j]
```

10) Information about the stages after the spanning tree labeling.

```
#List with demand nodes
for k in range(N):
```

```
    if sum(arc_matrix_ordered[k])==0:
        Ve=Ve+[k+1]

#List with initial nodes (no predecessors)
for i in range(N):
    if sum([arc_matrix_ordered[j][i]==1 for j in range(N)])==0: #if the
column is all zeros, then it is a supply node
        Vs=Vs+[i+1]

#Relation between stages and spanning tree labeling
for i in stages:
    orderStages=orderStages+[stages[i][1]]
```

11) Expressions for the objective function.

```
#Safety stock
def SS(k,SI,S):
    posk=orderStages.index(k)
    if SI+lt_av[posk+1]-S>=0:
        return
z[posk+1]*math.sqrt(varD[posk+1])*math.sqrt(lt_av[posk+1]+SI-S)
    else:
        print 'no stock' #representa la rallita, not possible
#Pipeline stock
def PS(k):
    posk=orderStages.index(k)
    return lt_av_decimal[posk+1]*avgD[posk+1]

#Cost function
def cost(k,SI,S):
    posk=orderStages.index(k)
    if SI+lt_av[posk+1]-S>=0:
        return CC[posk+1]*SS(k,SI,S)
    else:
        return 'no cost'
```

12) $f(k,S,stageInfo,L,U,a,Sh)$ and g$(k,SI,stageInfo,l,u,a,Sh)$ mean the same as $f(k,S,stageInfo)$ and g$(k,SI,stageInfo)$ in the first appendix. The difference remains in the limits. In this program, the limits are also an input for the functions due that depending on which iteration of the routine the functions are called, then the limits are modified to calculate the cost.

```
def f(k,S,stageInfo,L,U,a,Sh):
    SIResults=[]
```

```
    ZResultF=[]
    posk=orderStages.index(k)

    if a!=0:
        simin=int(max(Sh+1,L[posk+1],S-lt_av[posk+1]))

    else:
        simin=int(max(L[posk+1],S-lt_av[posk+1]))

    for si in range(simin,int(M[posk+1]-lt_av[posk+1]+1)):
        cF=cost(k,si,S)
        prevF=0.0
        nextF=0.0

        for i in range(k):
            m1=[]
            if arc_matrix_ordered_st[i][k-1]==1:
                listNF=stageInfo[i+1]
                listS=[]
                for nf in range(len(listNF)):
                    listS=listS+[listNF[nf][2]]

                for nf in range(len(listNF)):
                    if listNF[nf][2]<=si and listNF[nf][2]>=0:
                        m1=m1+[listNF[nf][0]]
                    elif si>max(listS):
                        m1=m1+[float("inf")]
                if m1!=[]:
                    Min1=min(m1)
                    prevF=prevF+Min1

        for j in range(k):
            m2=[]
            if arc_matrix_ordered_st[k-1][j]==1:
                posj=orderStages.index(j+1)
                listNF=stageInfo[j+1]
                listSI=[]
                for nf in range(len(listNF)):
                    listSI=listSI+[listNF[nf][1]]

                for nf in range(len(listNF)):
                    if listNF[nf][1]>=S and listNF[nf][1]<=(M[posj+1]-
lt_av[posj+1]):
                        m2=m2+[listNF[nf][0]]
```

```
                elif S>max(listSI):
                    m2=m2+[float("inf")]
            if m2!=[]:
                Min2=min(m2)
                nextF=nextF+Min2

        ResultF=cF+prevF+nextF
        SIResults=SIResults+[(ResultF,si,S)]
        ZResultF=ZResultF+[ResultF]

    ResultFinalF=min(ZResultF)
    PosResultFinalF=ZResultF.index(ResultFinalF)

TupleFinalF=[(ResultFinalF,SIResults[PosResultFinalF][1],SIResults[Pos
ResultFinalF][2])]

    return TupleFinalF

def g(k,SI,stageInfo,l,u,a,Sh):
    SResults=[]
    ZResultG=[]
    posk=orderStages.index(k)

    smax=min(SI+lt_av[posk+1]+1,M[posk+1]+1)
    if k==a and k not in Ve:
        smax=min(Sh+1,SI+lt_av[posk+1]+1)
        #smax=min(Sh+1,s_out_req[posk+1]+1,SI+lt_av[posk+1]+1)
    if k==a and k in Ve:
        smax=min(s_out_req[posk+1]+1,Sh+1,SI+lt_av[posk+1]+1)
    if k in Ve:
        smax=min(s_out_req[posk+1]+1,SI+lt_av[posk+1]+1)

  for s in range(int(l[posk+1]),int(smax)):
        cG=cost(k,SI,s)
        prevG=0.0
        nextG=0.0

        for i in range(k):
            m1=[]
            if arc_matrix_ordered_st[i][k-1]==1:
                listNG=stageInfo[i+1]
                listS=[]
                for nf in range(len(listNG)):
                    listS=listS+[listNG[nf][2]]
```

```
            for nf in range(len(listNG)):
                if listNG[nf][2]<=SI and listNG[nf][2]>=0:
                    m1=m1+[listNG[nf][0]]
                elif SI>max(listS):
                    m1=m1+[float("inf")]
            if m1!=[]:
                Min1=min(m1)
                prevG=prevG+Min1

        for j in range(k):
            m2=[]
            if arc_matrix_ordered_st[k-1][j]==1:
                posj=orderStages.index(j+1)
                listNG=stageInfo[j+1]
                listSI=[]
                for nf in range(len(listNG)):
                    listSI=listSI+[listNG[nf][1]]


                for nf in range(len(listNG)):
                    if listNG[nf][1]>=s and listNG[nf][1]<=(M[posj+1]-
lt_av[posj+1]):
                        m2=m2+[listNG[nf][0]]
                    elif s>max(listSI):
                        m2=m2+[float("inf")]
                if m2!=[]:
                    Min2=min(m2)
                    nextG=nextG+Min2

        ResultG=cG+prevG+nextG
        SResults=SResults+[(ResultG,SI,s)]
        ZResultG=ZResultG+[ResultG]

    ResultFinalG=min(ZResultG)
    PosResultFinalG=ZResultG.index(ResultFinalG)

TupleFinalG=[(ResultFinalG,SResults[PosResultFinalG][1],SResults[Pos
ResultFinalG][2])]
        return TupleFinalG
```

13) $model(l,u,L,U)$ means the same as $model(\ )$. However, in this occasion the limits for the decision variables are an input.

```
def model(l,u,L,U):
```

```
    stageInfo={}
    for k in range(N):
        posk=orderStages.index(k+1)
        if (k+1)!= N:          #all stages calculation
            TupleStage=[] #where I will store the results for each node
            for j in range(N):
                if k<j:
                    if arc_matrix_ordered_st[k][j]==1:
                        for S in range(int(l[posk+1]),int(u[posk+1]+1)):
                            TupleStage=TupleStage+f(k+1,S,stageInfo,L,U,0,0)
                        stageInfo[k+1]=TupleStage

                    if arc_matrix_ordered_st[j][k]==1:
                    for SI in range(int(L[posk+1]),int(U[posk+1]+1)):
                            TupleStage=TupleStage+g(k+1,SI,stageInfo,l,u,0,0)
                        stageInfo[k+1]=TupleStage

        else:
            LastTupleStage=[]
            for SI in range(int(L[posk+1]),int(U[posk+1]+1)):
                LastTupleStage=LastTupleStage+g(k+1,SI,stageInfo,l,u,0,0)
            stageInfo[k+1]=LastTupleStage

    return stageInfo
```

14) $upper(stageOpt, l, u, L, U, a, b, Sh)$ and $model(stageOpt, l, u, L, U, a, b)$ do the same work as $model(l, u, L, U)$. They are used instead of $model(l, u, L, U)$ when the routine calls one of the methods to solve the problem including the broken links by reducing the domain of solution changing the range of the limits.

```
def upper(stageOpt,l,u,L,U,a,b,Sh):
    stageInfoU={}
    for k in range(N):
        posk=orderStages.index(k+1)
        if (k+1)!= N:          #all stages calculation
            TupleStage=[] #where I will store the results for each node
            for j in range(N):
                if k<j:
                    if arc_matrix_ordered_st[k][j]==1:
                        if (k+1)==a:
                            Smax=min(u[posk+1]+1,Sh+1)
                        else:
                            Smax=int(u[posk+1]+1)
```

```
                for S in range(int(l[posk+1]),Smax):
                    TupleStage=TupleStage+f(k+1,S,stageInfoU,L,U,0,0)
                stageInfoU[k+1]=TupleStage

            if arc_matrix_ordered_st[j][k]==1:
                for SI in range(int(L[posk+1]),int(U[posk+1]+1)):
                    TupleStage=TupleStage+g(k+1,SI,stageInfoU,l,u,a,Sh)
                stageInfoU[k+1]=TupleStage

        else:
            LastTupleStage=[]
            for SI in range(int(L[posk+1]),int(U[posk+1]+1)):

LastTupleStage=LastTupleStage+g(k+1,SI,stageInfoU,l,u,a,Sh)
            stageInfoU[k+1]=LastTupleStage

    return stageInfoU

def lower(stageOpt,l,u,L,U,a,b,Sh):
    stageInfo={}
    sucA=[]
    for i in range(N):
        if arc_matrix_ordered[a-1][i]==1:
            sucA=sucA+[i+1]

    for k in range(N):
        posk=orderStages.index(k+1)
        if (k+1)!= N:           #all stages calculation
            TupleStage=[] #where I will store the results for each node
            for j in range(N):
                if k<j:
                    if arc_matrix_ordered_st[k][j]==1:
                        for S in range(int(l[posk+1]),int(u[posk+1]+1)):
                            if (k+1) in sucA:
                                TupleStage=TupleStage+f(k+1,S,stageInfo,L,U,a,Sh)
                            else:
                                TupleStage=TupleStage+f(k+1,S,stageInfo,L,U,0,0)
                        stageInfo[k+1]=TupleStage

                    if arc_matrix_ordered_st[j][k]==1:
                        if k+1 in sucA:
                            simin=max(Sh+1,L[posk+1])
                        else:
                            simin=int(L[posk+1])
```

```
                for SI in range(simin,int(U[posk+1])+1):
                    TupleStage=TupleStage+g(k+1,SI,stageInfo,l,u,0,0)
                stageInfo[k+1]=TupleStage

        else:
            LastTupleStage=[]
            if N==b:
                simin=max(Sh+1,L[posk+1])
            else:
                simin=int(L[posk+1])
            for SI in range(simin,int(U[posk+1])+1):
                LastTupleStage=LastTupleStage+g(k+1,SI,stageInfo,l,u,0,0)
            stageInfo[k+1]=LastTupleStage
    return stageInfo
```

15) $S\_hat(stageOpt)$ is another method called by the routine to fix the problem of including that the broken links also fulfill the constraints.

```
def S_hat(stageOpt):
    S_set1={}
    S_s1={}
    SI_si1={}

    for k in range(N):
        S_s1[k+1]=stageOpt[k+1][2]

    for i in range(N):
        SAlternatives=[0]
        for j in range(N):
            if arc_matrix_ordered[j][i]==1:
                SAlternatives=SAlternatives+[S_s1[j+1]]
            m=max(SAlternatives)
            SI_si1[i+1]=m

    costStages={}
    S_cost=0.0
    for k in range(N):
        costStages[k+1]=cost(k+1,SI_si1[k+1],S_s1[k+1])
        S_cost=S_cost+costStages[k+1]
    for k in range(N):
        S_set1[k+1]=(costStages[k+1],SI_si1[k+1],S_s1[k+1])

    return S_cost,S_set1
```

16) The function $broken(Zopt, stageOpt)$ investigates how many arcs from the broken connections to form the spanning tree do not accomplish all the constraints. The output of the function is then the number of broken restrictions and the broken connections.

```
def broken(Zopt,stageOpt):
    #number of brokenlinks
    count=0
    brokenL=[]
    for i in range(N):
        for j in range(N):
            if arc_matrix_ordered_brokenlinks[i][j]==1 and
stageOpt[i+1][2]>stageOpt[j+1][1]:
                count=count+1
                brokenL=brokenL+[(i+1,j+1)]
    return count,brokenL
```

17) Routine. The main tool of it is the dynamic process. First, it uses the model function to generate the first solution for the created spanning tree chain. Then, it checks if there is any constraint not true for any of the arcs of the whole network. If the result of the test is zero, the program finishes and displays the optimal solution. Otherwise, the next step is to call three different methods to find different solutions until there is no broken constraint. The routine always keep the best solution so far in the variable *Zbest* that should be initiated at an infinite number or big number such 10 000 000 000 000.

```
def R(Zbest):
    #step2
    stageInfo=model(l,u,L,U)
    Zopt,stageOpt=backtrack(stageInfo)

    #step3
    if Zopt>=Zbest or Zopt==float("inf"):
        return float("inf")

    #info links
    count,brokenL=broken(Zopt,stageOpt)
    iteration=1
    #step4
    while True:
        if count==0:
            Zbest=Zopt
            S_set={}
            for k in range(N):
```

```
            S_set[k+1]=stageOpt[k+1][1:]
        break

    else:
        #step5
        #info S_hat
        S_cost,S_set1=S_hat(stageOpt)

        if S_cost<Zbest:
            Zbest=S_cost

        #choose a link (j,i)
        #choose the brokenlink with the least associated cost
        #the associate arc (j,i) in A cost= -(Cj+Ci)*stdi
        cost_broken=[]
        Cost_min=0.0
        for bc in brokenL:
            posbc0=orderStages.index(bc[0])
            posbc1=orderStages.index(bc[1])
            cost_broken=cost_broken+[-
(CC[posbc0+1]+CC[posbc1+1])*math.sqrt(varD[posbc1+1])]
        Cost_min=min(cost_broken)
        posCost=cost_broken.index(Cost_min)
        a=brokenL[posCost][0] #j
        b=brokenL[posCost][1] #i
        Sh=stageOpt[b][1]+int((stageOpt[a][2]-stageOpt[b][1])/2)
        print a,b,Sh

        #upperbound
        stageInfoU=upper(stageOpt,l,u,L,U,a,b,Sh)
        SetU={}
        ZoptU,stageOptU=backtrack(stageInfoU)
        for k in range(N):
            SetU[k+1]=stageOptU[k+1]

        #lowerbound
        stageInfoL=lower(stageOpt,l,u,L,U,a,b,Sh)
        SetL={}
        ZoptL,stageOptL=backtrack(stageInfoL)
        for k in range(N):
            SetL[k+1]=stageOptL[k+1]

        Zmin=min(S_cost,ZoptU,ZoptL)
```

```
            if Zmin==S_cost:
                Zopt=S_cost
                stageOpt=S_set1
                print 'Sfirst'
            elif Zmin==ZoptU:
                Zopt=ZoptU
                stageOpt=SetU
                posa=orderStages.index(a)
                if Sh<=u[posa+1]:
                    u[posa+1]=Sh
                print 'upper'
            else:
                Zopt=ZoptL
                stageOpt=SetL
                posb=orderStages.index(b)
                print b, posb
                if Sh>=L[posb+1]:
                    L[posb+1]=Sh+1
                print 'lower'
                print L

            print Zopt,stageOpt
            #info links
            count,brokenL=broken(Zopt,stageOpt)
            iteration=iteration+1
            print count

        print iteration
        print 'final'
        return Zopt,stageOpt
```

18) $Nss(stageOpt)$ is a tool to calculate the number of stage with safety stock and the total safety stock in the network.

```
def Nss(stageOpt):
    totalSS=0.0
    Nss=0
    for i in range(N):
        print i+1, SS(i+1,stageOpt[i+1][1],stageOpt[i+1][2])
        totalSS=totalSS+SS(i+1,stageOpt[i+1][1],stageOpt[i+1][2])
        if SS(i+1,stageOpt[i+1][1],stageOpt[i+1][2])!=0:
            Nss=Nss+1
    return Nss,totals
```

19) To run the program and obtain the outputs.

```
Zopt,stageOpt=R(1111111111111111)
Nss,totalSS=Nss(stageOpt)
```

Appendix 3

Algorithms

Algorithm for Spanning tree (copied from *Graves and Willems [2000]*)

Labeling the nodes

The algorithm for labeling or re-numbering the nodes is as follows:

1. Start with all nodes in the unlabeled set, U.
2. Set k:=1.
3. Find a node $i \in U$ such that the node i is adjacent to at most one other node in U. That is, the degree of node i is 0 or 1 in the subgraph with node set U and arc set A defined on U.
4. Remove node i from set U and insert into the ñabeled set L; label node i with index k.
5. Stop if U is empty: otherwise set $k := k + 1$ and repeat steps 3 - 4.

Node N has no adjacent nodes with larger labels.

Dynamic program

1. For $k := 1$ to $N - 1$
   2. If $p(k)$ is downstream of k, evaluate $f_k(S)$ for $S = 0, 1, \dots, M_k$.
   3. If $p(k)$ is upstream of k, evaluate $g_k(SI)$ for $SI = 0, 1, \dots, M_k - L_k$.
4. For $k := N$ evaluate $g_k(SI)$ for $SI = 0, 1, \dots, M_k - L_k$.
5. Minimize $g_N(SI)$ for $SI = 0, 1, \dots, M_N - L_N$ to obtain the optimal objective function value.

From the optimal objective function, the optimal set of service times can be found by the standard backtracking procedure for a dynamic program.

The expression for the cost in each stage k is:

$$c_k(S, SI) = h_k\{D_k(SI + L_k - S) - (SI + L_k - S)\mu_k^d\} + \sum_{\substack{(i,k)\in A \\ i<k}} \min_{0 \le x \le SI} [f_k(x)]$$

$$+ \sum_{\substack{(k,j)\in A \\ j<k}} \min_{S \le y \le M_j - L_j} [g_k(y)]$$

Where:

$$f_k(S) = \min_{SI} \{c_k(S, SI)\} \; s.t. \max(0, S - L_k) \le SI \le M_k - L_k, and \; SI \; integer$$

$$g(SI) = \min_{S} \{c_k(S, SI)\}$$
$$s.t. 0 \leq S \leq SI + L_k, and\ SI\ integer$$

The General Network Algorithm (copied from *Humair and Willems [2011]*)

$\boldsymbol{T_k}$**:** Lead time production. It is change the letter to not confuse it with the limits in the constraints.

Routine *R* (arguments $\underline{z}, T, \Omega^G, \Omega$)

Step 1: Let Ω^T be the subset of constraints in Ω that correspond to T, i.e., drop all constraints from Ω that correspond to links not in T. Call P^T the problem of optimizing the cost function of P subject to Ω^T.

Step 2: Use SDP' to obtain an optimal solution to P^T. Let z^T be the optimal value of P^T, and S^T the optimal solution if one exists.

Step 3: If $z^T \geq \underline{z}$ or $z^T = \infty$, return ∞ for the cost.

Step 4: Else, if S^T satisfies the constraints in Ω^G, set $\underline{z} = z^T$ and return z^T, S^T.

Step 5: Else, S^T does not satisfy the constraints in Ω^G. Let $\hat{z}$ equal the cost of the solution $\hat{S}$, which has $\hat{S}_i = S_i^T$, and $\widehat{SI}_i = max_{(j,i)\in A}\hat{S}_j$ for all stages i. If $\hat{z} < \underline{z}$, set $\underline{z} = \hat{z}$; otherwise, leave $\underline{z}$ unchanged. Then carry out the steps below.

(a) Choose a link $(j, i) \in A$ for which $S_j^T > SI_i^T$, i.e., a constraint in Ω^G is violated. Let $\overline{S} = SI_i^T + \lfloor (S_j^T - SI_i^T)/2 \rfloor$.
(b) Carry out the *Upperbound* and *Lowerbound* steps below in order.
Upperbound: Let z_u, S_u be the solution returned by *R* for arguments $\underline{z}, T, \Omega^G$, and the updated constraint set $\Omega \cup \{S_j \leq \overline{S}\}$.
Lowerbound: Let z_l, S_l be the solution returned by *R* for arguments $\underline{z}, T, \Omega^G$, and the updated constraint set $\Omega \cup \{SI_k > \overline{S}, \forall k: (j,k) \in A\}$.

Return $min\ (\hat{z}, z_u, z_l)$ and the associated solution (breaking any tie arbitrarily).

Arguments and limits

SDP: Solving the problem P with a dynamic program.

SDP': Solving the problem P' with a dynamic program.

Ω: Set of constraint.

$\boldsymbol{S} = (S_1, SI_1, \ldots, S_n, SI_n)$: Set of inbound and outbound service times, the decision variables for P' and P.

$\underline{\boldsymbol{z}}$**:** Best solution so far.

***T*:** Spanning tree.

$L_k(S) = max\ (S - T_k.L_k)$

$\overline{U}_k = min\ (M_k - T_k.U_k)$

$u_k(SI) = min\ (SI + T_k.u_k)$

Problem ***P'*** is the same as ***P*** with two extra constraints:

$L_k \leq SI_k \leq U_k$

$l_k \leq S_k \leq u_k$

Function $f_k(S)$ and $g(SI)$ are redefined taking into account the above constraints:

If $(k, p_k) \in A$ and $k \leq n-1$, then $f_k(S) = \infty$ if $S < l_k$, $S > u_k$, or if $L_k(S) > \overline{U}_k$. Otherwise, for $l_k \leq S \leq u_k$.

$$f_k(S) = \min_{\substack{SI: \\ L_k(S) \leq SI \leq \overline{U}_k}} \{c_k(S, SI) + \sum_{\substack{(i,k)\in A \\ i<k}} \min_{x:\, x \leq SI} [f_i(x)] + \sum_{\substack{(k,j)\in A \\ j<k}} \min_{y:\, y \geq S} [g_j(y)]$$

If $(p_k, k) \in A$ and $k = n$, then $g_k(SI) = \infty$ if $SI < L_k$, $SI > U_k$, or if $l_k > U_k$. Otherwise, for $L_k \leq SI \leq U_k$.

$$g_k(SI) = \min_{\substack{S: \\ l_k \leq SI \leq u_k(SI)}} \{c_k(S, SI) + \sum_{\substack{(i,k)\in A \\ i<k}} \min_{x:\, x \leq SI} [f_i(x)] + \sum_{\substack{(k,j)\in A \\ j<k}} \min_{y:\, y \geq S} [g_j(y)]$$

Appendix 4

Program for the robust optimization

```
from gurobipy import *
import math
import xlrd
import xlwt
from scipy.stats import norm

book = xlrd.open_workbook("C:\Users\Pablo\Desktop\MSOM-06-038-R2-
modified.xls")
arcs=book.sheet_by_name("09_LL")
data=book.sheet_by_name("09_SD")
R=5000

results = xlwt.Workbook()
r_example = results.add_sheet("Example1", cell_overwrite_ok=True)
model = Model("5stage")

#Indexes

N=len(data.col_values(0))-1

arc_matrix = [[0] * (len(data.col_values(0))-1) for i in
range(len(data.col_values(0))-1)] #Initial matrix dim(N*N) with all values equal
zero
stages={}
avgD = {} #mean demand
varD = {} #variance of demand
stdD = {}
h_add = {}      #holding cost
service = {} #service level
z_lev={} #normsinv(service)
lt_av = {}  #lead time mean
lt_av_decimal = {}
lt_var = {} #var lead time
m={} #replenishment time
s_out_req={}
Ve=[]
Vs=[]
gamma = {}
stage_lk={}
prob_lk={}
```

```
lt_max={}
lt_min={}
lt_average={}

for row in range(N):
    stages[row+1]= [data.cell_value(row+1,0), 0]
    h_add[row+1]= data.cell_value(row+1,1)
    avgD[row+1]= data.cell_value(row+1,4)
    stdD[row+1]= data.cell_value(row+1,5)
    service[row+1]=data.cell_value(row+1,7)
    lt_average[row+1]=data.cell_value(row+1,8)
    lt_av[row+1]=data.cell_value(row+1,8)
    if (lt_av[row+1]/0.5)%2==1:
        lt_av[row+1]=lt_av[row+1]-0.01
    lt_av[row+1]=round(lt_av[row+1])
    lt_av_decimal[row+1]=data.cell_value(row+1,8)
    s_out_req[row+1]=data.cell_value(row+1,6)
    if avgD[row+1]=='':
        avgD[row+1]=0
        stdD[row+1]=0
    varD[row+1]= stdD[row+1]**2
    gamma[row+1]=0

#Arc_matrix (with all the arcs relation)
for i in range(1,len(arcs.col_values(0))):
    val_extra_1=arcs.cell_value(i,0)
    val_extra_2=arcs.cell_value(i,1)

arc_matrix[stages.values().index([val_extra_1,0])][stages.values().index([val_ext
ra_2,0])]=1
            #Demand propagation (average and variance)
#Arc_matrix_demand will be modified during the algorithm
arc_matrix_demand=[[0] * (len(data.col_values(0))-1) for i in
range(len(data.col_values(0))-1)]
for i in range(N):
    for j in range(N):
        arc_matrix_demand[i][j]=arc_matrix[i][j]

memory_del=[]
while(arc_matrix_demand != [[0] * (len(arc_matrix)) for i in
range(len(arc_matrix_demand))]):    #running until arc_matrix_demand
becomes zero
    for row in [x for x in xrange(len(arc_matrix_demand)) if x not in memory_del]:
#It selects row with no 1 (demand nodes)
```

```
        if arc_matrix_demand[row] == ([0]*N):
            for i in range(len(arc_matrix_demand)):
                if arc_matrix_demand[i][row]==1:
                    avgD[i+1]=avgD[i+1]+avgD[row+1]
                    varD[i+1]=varD[i+1]+varD[row+1]
                  memory_del=memory_del+[row]

    for aa in memory_del:
        for bb in range(len(arc_matrix_demand)):
            if arc_matrix_demand[bb][aa]==1:
                arc_matrix_demand[bb][aa]=0

for i in range(N):
    stdD[i+1]=math.sqrt(varD[i+1])

    #Net replenishment propagation
m={}
memory_del=[]
arc_matrix_rp=[[0] * (len(data.col_values(0))-1) for i in
range(len(data.col_values(0))-1)]
arc_matrix_aux=[[0] * (len(data.col_values(0))-1) for i in
range(len(data.col_values(0))-1)]
for i in range(N):
    for j in range(N):
        arc_matrix_rp[i][j]=arc_matrix[i][j]

memory_del=[]
while (len(memory_del)!=N):
    for i in [x for x in xrange(N) if x not in memory_del]:
        if sum([arc_matrix_rp[j][i]==1 for j in range(N)])==0: #Columns with all
zeros (no predecessors)
            m[i+1]=lt_av[i+1]+max(arc_matrix_aux[i])
            for j in range(N):
                if arc_matrix_rp[i][j]==1:
                    arc_matrix_aux[j][i]=m[i+1] #Arc_matrix_aux is used to create the
Mmax (it would be the max of the row)
                arc_matrix_rp[i][j]=0
            memory_del=memory_del+[i]

    #Holding cost calculation
h={}
memory_del=[]
arc_matrix_rp=[[0] * (len(data.col_values(0))-1) for i in
range(len(data.col_values(0))-1)]
```

```
arc_matrix_aux=[[0] * (len(data.col_values(0))-1) for i in
range(len(data.col_values(0))-1)]
for i in range(N):
    for j in range(N):
        arc_matrix_rp[i][j]=arc_matrix[i][j]

memory_del=[]
while (len(memory_del)!=N):
    for i in [x for x in xrange(N) if x not in memory_del]:
        if sum([arc_matrix_rp[j][i]==1 for j in range(N)])==0: #Columns with all
zeros (no predecessors)
            h[i+1]=h_add[i+1]+sum(arc_matrix_aux[i])
            for j in range(N):
                if arc_matrix_rp[i][j]==1:
                    arc_matrix_aux[j][i]=h[i+1] #Arc_matrix_aux is used to create the
Mmax (it would be the max of the row)
                arc_matrix_rp[i][j]=0
            memory_del=memory_del+[i]

#Service level coefficient
#Calculation of the average service level coefficient (for those with no value
assigned)
count_service=0
sum_service=0
for k in range (N):
    if service[k+1]!='':
        count_service += 1
        sum_service += service[k+1]

#The average is assigned to those stages with no value assigned
for k in range (N):
    if service[k+1]=='':
        service[k+1]=sum_service/count_service

#Dictionary with z=normsinv(service)
for k in range(N):
    z_lev[k+1]=norm.ppf(service[k+1])

#Lineal approximation
M={}
alpha={}
f={}

for i in range(N):
```

```
    for r in range (R+1):

        M[r,i+1]=0.03*r

for i in range(N):
    for r in range(R):
        alpha[r+1,i+1]=(z_lev[i+1]*stdD[i+1])*((math.sqrt(M[r+1,i+1])-
math.sqrt(M[r,i+1])))/(M[r+1,i+1]-M[r,i+1])
        f[r+1,i+1]=(z_lev[i+1]*stdD[i+1])*((M[r+1,i+1]*math.sqrt(M[r,i+1])-
M[r,i+1]*math.sqrt(M[r+1,i+1]))/(M[r+1,i+1]-M[r,i+1]))

#Decision Variables
s_in={}
s_out={}
x={}
z={}
u={}
#a & b extra are used to include the summatories in the constraints
a_extra={}
b_extra={}

for i in range(N):
    s_in[i+1] = model.addVar( lb=0.0, ub=m[i+1]-lt_av[i+1], vtype="I",
name="s_in[%s]"%((i+1)))
    s_out[i+1] = model.addVar( lb=0.0, ub=m[i+1], vtype="I",
name="s_out[%s]"%((i+1)))
    x[i+1] = model.addVar( lb=0.0, vtype="I", name="x[%s]"%((i+1)))
    a_extra[i+1] = model.addVar( lb=0.0, vtype="C", name="a[%s]"%((i+1)))
    b_extra[i+1] = model.addVar( lb=0.0, vtype="C", name="b[%s]"%((i+1)))

for i in range(N):
    for r in range(R):
        z[r+1,i+1] = model.addVar ( lb=0.0, vtype="I", name="z[%s,%s]"%(r+1,i+1))
        u[r+1,i+1] = model.addVar ( vtype="B", name="u[%s,%s]"%(r+1,i+1))

for i in range(N):
    if sum([arc_matrix[j][i]==1 for j in range(N)])==0:
        s_in[i+1]=0
        Vs=Vs+[i+1]

for k in range(N):
    if sum(arc_matrix[k])==0:
        Ve=Ve+[k+1]
model.update()
```

```
#Constraint 8a
for i in range(N):
    a_extra[i+1]=0

for i in range(N):
    for r in range(R):
        a_extra[i+1]=z[r+1,i+1]+a_extra[i+1]

for i in range(N):
    model.addConstr(x[i+1],"=", a_extra[i+1])

#Constraint 8b.
for i in range(N):
    for r in range(R):
        model.addConstr(M[r,i+1]*u[r+1,i+1],"<=", z[r+1,i+1])

for i in range(N):
    for r in range(R):
        model.addConstr(z[r+1,i+1],"<=", M[r+1,i+1]*u[r+1,i+1])

#Constraint 8c. and 8d.
for i in range(N):
    b_extra[i+1]=0

for i in range(N):
    for r in range(R):
        b_extra[i+1]=b_extra[i+1]+u[r+1,i+1]

for i in range(N):
    model.addConstr(b_extra[i+1],"<=",1)

#Constraint 8e.
for i in range(len(Ve)):
        a=Ve[i]
        model.addConstr(s_out[a],"<=",s_out_req[a])

#In supply nodes, SI=0 (by convention)
for i in range(len(Vs)):
        a=Vs[i]
        model.addConstr(s_in[a],"=",0)

#Constraint 8g.
for i in range(N):
```

```
    for j in range(N):
        if arc_matrix[i][j] ==1:
            model.addConstr(s_in[j+1],">=",s_out[i+1])

for i in range(N):
    model.addConstr(x[i+1],">=",s_in[i+1]-s_out[i+1]+lt_av[i+1])

#Constraints 8h., 8i. and 8j included in the declaration of variables

#Objective function
exp={}
cost=0
for i in range(N):
    exp[i+1]=0

for i in range(N):
    for r in range(R):
        exp[i+1]=exp[i+1]+(f[r+1,i+1]*u[r+1,i+1]+alpha[r+1,i+1]*z[r+1,i+1])

for i in range(N):
        cost=cost+h[i+1]*exp[i+1]

model.setObjective(cost,GRB.MINIMIZE)
model.update()
model.optimize()
model.printAttr('x')

safety={}
safety_new={}
cost_st={}
cost_st_stock={}
safety_total=0
pipeline_st={}
pipeline_total=0
cost_total_stock=0
cost_total=0

for i in range(N):
    safety[i+1]=math.sqrt(x[i+1].X)*stdD[i+1]*z_lev[i+1]
    pipeline_st[i+1]=lt_av_decimal[i+1]*avgD[i+1]
    cost_st[i+1]=h[i+1]*safety[i+1]
    cost_st_stock[i+1]=h[i+1]*(safety[i+1]+pipeline_st[i+1])

for i in range(N):
```

```
cost_total=cost_total+cost_st[i+1]
cost_total_stock=cost_total_stock+cost_st_stock[i+1]
safety_total=safety_total+safety[i+1]
pipeline_total=pipeline_total+pipeline_st[i+1]
```

Appendix 5

Program for optimizing the safety stock placement in General Acyclic Chains under random lead time

The necessaries changes to adapt *Appendix 2* for contemplating random lead time are exposed next.

1) Statistics of the positive shortfall ($\boldsymbol{H_i^1(T), H_i^2(T), H_i^2(T)}$).

```
def H1(n3,k): #k follows the new numeration of the nodes, spanning tree numeration
    posk=orderStages.index(k)
    stage_lk=[data.cell_value(posk+1,c) for c in range (10,22,2)]
    prob_lk=[data.cell_value(posk+1,c) for c in range (11,23,2)]
    for i in range(len(stage_lk)):
        if stage_lk[i]='':
            stage_lk[i]=0
            prob_lk[i]=0
    if stage_lk==[0]*6:
        stage_lk[0]=lt_av[posk+1]
        prob_lk[0]=1

    sum1=0
    for j in range(len(stage_lk)):
        if stage_lk[j]<=n3:
            sum1+=prob_lk[j]

    #For the case the distribution is continuous
    if lt_std[posk+1]!=0 and prob_lk[0]==1 and sum(prob_lk[1:])==0:
        sumI=integrate.quad(lambda x: (norm.pdf(x, lt_av[posk+1], lt_std[posk+1])), 0, n3)
        sum1=sumI[0]
        if sum1<0:
            sum1=0
    if sum1<0.1:
        sum1=0
    return sum1

#H2=Expected value of lead time when nrlt k(t) is non-negative [SAFETY STOCK IS NEEDED]
def H2(n4,k):
    posk=orderStages.index(k)
    stage_lk=[data.cell_value(posk+1,c) for c in range (10,22,2)]
    prob_lk=[data.cell_value(posk+1,c) for c in range (11,23,2)]
```

```
    for i in range(len(stage_lk)):
        if stage_lk[i]=='':
            stage_lk[i]=0
            prob_lk[i]=0
    if stage_lk==[0]*6:
        stage_lk[0]=lt_av[posk+1]
        prob_lk[0]=1

    new_lt=[]
    new_prob_lt=[]
    for i in range(6):
        if stage_lk[i]>n4:    #*Bigger than n4. Different value from the paper, the equal is not included. In the limits was problematic with the equal.
            new_lt=new_lt+[stage_lk[i]]
            new_prob_lt=new_prob_lt+[prob_lk[i]]
    prod2=[]
    sum2=0.0
    prod2=[a*b for a,b in zip(new_lt,new_prob_lt)]
    for i in prod2:
        sum2+=i

    if lt_std[posk+1]!=0 and prob_lk[0]==1 and sum(prob_lk[1:])==0:
        sumI=integrate.quad(lambda x: x*norm.pdf(x, lt_av[posk+1], lt_std[posk+1]), n4, np.inf)
        sum2=sumI[0]
        if sum2<0:
            sum2=0
    if sum2<0.1:
        sum2=0
    return sum2

#H3=not statistical meaning
def H3(n5,k):
    posk=orderStages.index(k)
    stage_lk=[data.cell_value(posk+1,c) for c in range (10,22,2)]
    prob_lk=[data.cell_value(posk+1,c) for c in range (11,23,2)]
    for i in range(len(stage_lk)):
        if stage_lk[i]=='':
            stage_lk[i]=0
            prob_lk[i]=0
    if stage_lk==[0]*6:
        stage_lk[0]=lt_av[posk+1]
        prob_lk[0]=1
```

```
    new_lt=[]
    new_prob_lt=[]
    for i in range(6):
        if stage_lk[i]>n5:  #*Bigger than n5. Different value from the paper, the equal is not included. In the limits was problematic with the equal.
            new_lt=new_lt+[stage_lk[i]]
            new_prob_lt=new_prob_lt+[prob_lk[i]]

    stage_lt2=new_lt
    elev2=[a*b for a,b in zip(new_lt,stage_lt2)]
    prod3=[c*d for c,d in zip(elev2,new_prob_lt)]
    sum3=0.0
    for i in prod3:
        sum3=sum3+i

    if lt_std[posk+1]!=0 and prob_lk[0]==1 and sum(prob_lk[1:])==0:
        sumI=integrate.quad(lambda x: x*x*norm.pdf(x, lt_av[posk+1], lt_std[posk+1]), n5, np.inf)
        sum3=sumI[0]
        if sum3<0:
            sum3=0
    if sum3<0.1:
        sum3=0
    return sum3
```

2) Expected value and variance of the positive part of the *NRLT* when $T = max\{S_i - SI_i, 0\}$

```
def Q(n1,k):
    q=H2(n1,k)-n1*(1-H1(n1,k))
    return q

def RR(n2,k):
    r=(n2**2)*H1(n2,k)*(1-H1(n2,k))-2*n2*H1(n2,k)*H2(n2,k)+H3(n2,k)-(H2(n2,k))**2
    return r
```

3) Expressions for the objective function.

```
def SS(k,SI,S):
    posk=orderStages.index(k)
    par=S-SI
    if lt_std[posk+1]==0:           #deterministic case
        NRLT=SI+lt_av[posk+1]-S
        if NRLT>=0:
```

```
            ss=z[posk+1]*math.sqrt(varD[posk+1])*math.sqrt(NRLT)
        else:
            ss=0
    else:
        if par<0:
            ss=z[posk+1]*math.sqrt(((lt_av[posk+1]+SI-
S)*varD[posk+1])+avgD[posk+1]*avgD[posk+1]*lt_std[posk+1]*lt_std[pos
k+1])
        else:
            ss=z[posk+1]*math.sqrt((Q(par,k))*varD[posk+1]+((avgD[posk+1]
            )**2)*RR(par,k))
    return ss

def EARLY(k,SI,S):
    posk=orderStages.index(k)
    par=max(0,S-SI)
    if par!=0:
        early=avgD[posk+1]*(Q(par,k)-lt_av[posk+1]+par)
    else:
        early=0
    return early

#cost function
def cost(k,SI,S):
    posk=orderStages.index(k)
    return CC[posk+1]*(SS(k,SI,S)+EARLY(k,SI,S))
```