

Optimizing Safety Stock Placement in General Network Supply Chains

by

Ekaterina Lesnaia [O.R.]

Submitted to the Sloan School of Management
in partial fulfillment of the requirements for the degree of

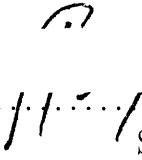
Doctor of Philosophy

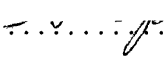
at the

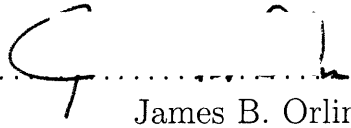
MASSACHUSETTS INSTITUTE OF TECHNOLOGY

September 2004

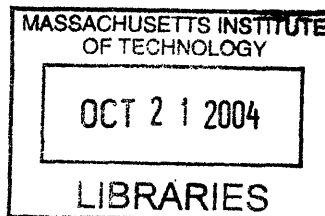
© Massachusetts Institute of Technology 2004. All rights reserved.

Author 
Sloan School of Management
August 13, 2004

Certified by 
Stephen C. Graves
Abraham J. Siegel Professor of Management Science & Engineering
Systems
Thesis Supervisor

Accepted by 
James B. Orlin
Edward Pennell Brooks Professor
Co-director, MIT Operations Research Center

ARCHIVES





Optimizing Safety Stock Placement in General Network

Supply Chains

by

Ekaterina Lesnaia

Submitted to the Sloan School of Management
on August 13, 2004, in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy

Abstract

The amount of safety stock to hold at each stage in a supply chain is an important problem for a manufacturing company that faces uncertain demand and needs to provide a high level of service to its customers. The amount of stock held should be small to minimize holding and storage costs while retaining the ability to serve customers on time and satisfy most, if not all, of the demand. This thesis analyzes this problem by utilizing the framework of deterministic service time models and provides an algorithm for safety stock placement in general-network supply chains. We first show that the general problem is NP-hard. Next, we develop several conditions that characterize an optimal solution of the general-network problem. We find that we can identify all possible candidates for the optimal service times for a stage by constructing paths from the stage to each other stage in the supply chain. We use this construct, namely these paths, as the basis for a branch and bound algorithm for the general-network problem. To generate the lower bounds, we create and solve a spanning-tree relaxation of the general-network problem. We provide a polynomial algorithm to solve these spanning tree problems. We perform a set of computational experiments to assess the performance of the general-network algorithm and to determine how to set various parameters for the algorithm. In addition to the general network case, we consider two-layer network problems. We develop a specialized branch and bound algorithm for these problems and show computationally that it is more efficient than the general-network algorithm applied to the two-layer networks.

Thesis Supervisor: Stephen C. Graves

Title: Abraham J. Siegel Professor of Management Science & Engineering Systems

Acknowledgments

The four years at MIT have been the most amazing years of my life. I would like to thank all the people who were with me and who will remain the most special in the future.

First of all, I would like to thank my advisor Stephen Graves for his guidelines, support and patience. I thank Cynthia Barnhart and Jeremie Gallien for being my thesis committee members. Also, I am grateful for Cyndy's support in my first year at MIT.

I am thankful for the financial support of the Singapore-MIT Alliance and for the opportunity to travel to Singapore.

I am grateful to all the students of the Operations Research Center. This place is unique and the students make the ORC unforgettable. I met new friends whose support helped me so much. I thank Ping, Anshul, Jose, Melvyn, David and everybody I forgot to list here.

My friends outside of the ORC believed in me and I thank them for that. I thank Dasha, Julian, Erin, Ty and Luca. My special thanks go to Iuliu, whose incredible help and support made me finish the thesis. I thank Martin for being with me.

Finally, I would like to dedicate this thesis to my family. Their love will always be with me.

Contents

1	Introduction	21
1.1	Assumptions and formulation	23
1.1.1	Assumptions	23
1.1.2	Formulation	27
1.2	Literature Review	30
1.2.1	Safety stock modeling	30
1.2.2	Complexity of concave minimization	38
1.2.3	Methods	39
1.2.4	Complexity and algorithms for the safety stock problem	45
2	Problem Characteristics	47
2.1	Complexity of the problem	47
2.1.1	General case	47
2.1.2	Restricted case: $s_i \leq T_i$	56
2.2	Optimality conditions	63
2.2.1	Restricted case: $s_i \leq T_i$	64
2.2.2	General case	69

2.2.3	Critical paths	78
3	Algorithms	105
3.1	Spanning tree algorithm	105
3.1.1	Optimality conditions	106
3.1.2	Polynomial time algorithm	110
3.2	Algorithm for general networks	121
3.2.1	Branching tree	122
3.2.2	Lower Bounds	126
3.2.3	Upper Bounds	131
3.2.4	Algorithm	132
3.3	Two-layer networks	136
3.3.1	Optimality conditions	137
3.3.2	Algorithm	140
3.3.3	Lower Bounds	142
3.3.4	Upper Bounds	142
4	Computations	153
5	Conclusions	201

List of Figures

1-1	Base-stock mechanics.	28
2-1	Graph G for the problem VC	49
2-2	Graph G' for the problem P_i	49
2-3	An example of the safety stock function.	50
2-4	Graph G'	58
2-5	An example of a network for which conditions (2.6), (2.7) and (2.8) do not hold for any optimal solution.	72
2-6	A path between nodes 1 and 6.	79
2-7	A path between nodes i and j with arc $(i, i_1) \in \mathbb{A}$ in the path.	80
2-8	A path between nodes i and j with arc $(i_1, i) \in \mathbb{A}$ in the path.	81
2-9	A path between nodes i and j with arcs $(i_{k-1}, i_k) \in \mathbb{A}$ and $(i_k, i_{k+1}) \in \mathbb{A}$ in the path.	81
2-10	A path between nodes i and j with arcs $(i_{k-1}, i_k) \in \mathbb{A}$ and $(i_k, i_{k+1}) \in \mathbb{A}$ in the path.	81
2-11	A path with arcs $(i_{k-1}, i_k) \in \mathbb{A}$ and $(i_{k+1}, i_k) \in \mathbb{A}$	81

2-12	A path between nodes i and j with arcs $(i_k, i_{k-1}) \in \mathbb{A}$ and $(i_k, i_{k+1}) \in \mathbb{A}$ in the path.	82
2-13	A path between nodes i and $i_p = j$ with arcs $(i_{p-1}, i_{p-2}) \in \mathbb{A}$ and $(i_{p-1}, i_p) \in \mathbb{A}$ in the path.	82
2-14	A path between nodes i and $i_p = j$ with arcs $(i_{p-2}, i_{p-1}) \in \mathbb{A}$ and $(i_{p-1}, i_p) \in \mathbb{A}$ in the path.	82
2-15	A path between nodes i and $i_p = j$ with arc $(i_p, i_{p-1}) \in \mathbb{A}$ in the path.	82
2-16	A path between nodes i and j with $i_k = i_r$ and arcs such that arcs $(i_k, i_{k-1}) \in \mathbb{A}$, $(i_k, i_{k+1}) \in \mathbb{A}$, $(i_{r-1}, i_r) \in \mathbb{A}$ and $(i_r, i_{r+1}) \in \mathbb{A}$	97
2-17	A path between nodes i and j with $i_k = i_r$ and arcs such that arcs $(i_k, i_{k-1}) \in \mathbb{A}$, $(i_k, i_{k+1}) \in \mathbb{A}$, $(i_r, i_{r-1}) \in \mathbb{A}$ and $(i_{r+1}, i_r) \in \mathbb{A}$	97
2-18	A path between nodes i and j with $i_k = i_r$ and arcs such that arcs $(i_k, i_{k-1}) \in \mathbb{A}$, $(i_{k+1}, i_k) \in \mathbb{A}$, $(i_r, i_{r-1}) \in \mathbb{A}$ and $(i_r, i_{r+1}) \in \mathbb{A}$	99
2-19	A path between nodes i and j with $i_k = i_r$ and arcs such that arcs $(i_k, i_{k+1}) \in \mathbb{A}$, $(i_r, i_{r-1}) \in \mathbb{A}$ and $(i_r, i_{r+1}) \in \mathbb{A}$	99
2-20	A path between nodes i and j with $i = i_k = i_0 = i_r$ and arcs $(i_1, i) \in \mathbb{A}$, $(i_r, i_{r-1}) \in \mathbb{A}$ and $(i_r, i_{r+1}) \in \mathbb{A}$, $r < p$, in the path	100
2-21	A path between nodes i and j with $i_k = i_r$ and arcs $(i_k, i_{k+1}) \in \mathbb{A}$, $(i_k, i_{k-1}) \in \mathbb{A}$, $(i_k, i_{k+1}) \in \mathbb{A}$ and $(i_k, i_{k-1}) \in \mathbb{A}$, $(i_{r-1}, i_r) \in \mathbb{A}$ and $(i_{r+1}, i_r) \in \mathbb{A}$, $r < p$, in the path	100
2-22	A path between nodes i and j with $i_k = i_r$ and arcs $(i_k, i_{k+1}) \in \mathbb{A}$, $(i_k, i_{k-1}) \in \mathbb{A}$, $(i_{r-1}, i_r) \in \mathbb{A}$ and $(i_{r+1}, i_r) \in \mathbb{A}$, $r < p$, in the path . . .	101

2-23	An instance of the safety stock problem with an optimal solution corresponding to a path with self intersection.	102
3-1	A spanning tree network for Example 5.	112
3-2	Search tree.	112
3-3	New labelling.	112
3-4	Branching tree for the general problem.	123
3-5	The network for Example 7.	129
3-6	Spanning trees and correspondent relaxed problem solutions for Example 7.	130
3-7	The objective function depends only on the outbound service times S_1, S_2, S_3	138
3-8	Branching tree for a two-layer problem with $s_i \leq T_i, i \in \mathbb{D}$	141
3-9	Two-layer network.	143
3-10	A tree chosen by the lower bound algorithm	143
3-11	The second tree chosen by the lower bound algorithm	144
3-12	Subsets	147
4-1	Average time of solving a general network problem with 20 nodes and 25 arcs using polynomial bounds.	160
4-2	Average time of solving a general network problem with 20 nodes and 25 arcs using polynomial and pseudo-polynomial bounds.	161
4-3	Average time in milliseconds for Experiment 3 to choose same, smart or random trees for the lower bounds.	164

4-4	Average number of branching points for Experiment 3 to choose same, smart or random trees for the lower bounds.	165
4-5	Average number of generated trees for Experiment 3 to choose same, smart or random trees for the lower bounds.	168
4-6	Average time in milliseconds for solving a problem with 20 nodes and 30 arcs as a function of the number of bounds per branching point, for different numbers of initial bounds in Experiment 4.	169
4-7	Average time in milliseconds for solving a problem with 20 nodes and 30 arcs as a function of the number of initial bounds for different numbers of bounds per branching point in Experiment 4.	170
4-8	Average time in milliseconds for solving a problem with 20 nodes and 20 and 23 arcs as a function of the number of initial bounds in Experiment 5.	174
4-9	Average time in milliseconds for solving a problem with 20 nodes and 41 arcs as a function of the number of initial bounds in Experiment 5.	175
4-10	The best number of initial bounds as a function of the number of arcs in Experiment 5.	176
4-11	Average time in milliseconds for solving a problem with 30 nodes and 40 arcs for different number of BBP as a function of GBBP in Experiment 6.	179
4-12	Average time in milliseconds for solving a problem with 20 nodes and 20 and 23 arcs as a function of the number of bounds per branching point in Experiment 7.	181

4-13	Average time in milliseconds for solving a problem with 20 nodes and 41 arcs as a function of the number of bounds per branching point in Experiment 7.	182
4-14	The best number of bounds per branching point as a function of the number of arcs for 20 nodes in Experiment 7.	183
4-15	Average time in milliseconds for solving a problem with 30 nodes and 42 to 51 arcs as a function of the number of bounds per branching point in Experiment 7.	184
4-16	Average time in milliseconds for solving a problem with 100 nodes and 100 to 123 arcs for 2%, 5%, 10% tolerance limits in Experiment 9.	193
4-17	Average time in milliseconds for solving a two-layer problem with 10 components and 10 demand nodes using the two-layer and general network algorithms in Experiment 11.	194

List of Tables

2.1	Intervals of the objective function value depending on SI_{N+1}	61
2.2	Configurations of nodes and adjacent arcs in the path.	80
2.3	Service times for the path on Figure 2-6.	92
2.4	Parameters of the network from Example 4.	101
2.5	Solution of the problem from Example 4.	102
3.1	Possible values of SI_j depending on the constraints on the service times.	109
3.2	Possible values of S_j depending on the constraints on the service times.	109
3.3	Lead times and guaranteed service times for Example 6	118
3.4	Initial values of $Sset[i]$ and $SIset[i]$ for Example 6.	118
3.5	Values of $Sset[i]$ and $SIset[i]$ created by procedure ComputeSandSIUp for Example 6.	119
3.6	Values of $Sset[i]$ and $SIset[i]$ created by procedure ComputeSandSI- Down for Example 6.	119
3.7	Parameters of the problem in Example 7.	129
3.8	An optimal solution for the network in Example 7.	129
3.9	Fixed solution for Example 8.	132

3.10 Parameters of the network shown on Figure 3-9	143
3.11 An optimal solution to the problem shown on Figure 3-10	144
3.12 The fixed solution for Figure 3-10 and Table 3.11	144
3.13 An optimal solution for the problem shown on Figure 3-11	145
4.1 Summary of the experiment settings.	158
4.2 Settings for Experiment 1.	158
4.3 Average time per instance in milliseconds for Experiment 1.	160
4.4 Settings for Experiment 2 to choose random, cost or layer node order.	165
4.5 Average time in milliseconds and average number of branching points for Experiment 2 to choose random, cost or layer order.	165
4.6 Settings for Experiment 3 to choose same, smart or random tree for lower bounds.	166
4.7 Average time per instance in milliseconds for Experiment 3 to choose same, smart or random trees for the lower bounds.	166
4.8 Average number of branching points for Experiment 3 to choose same, smart or random trees for the lower bounds.	167
4.9 Average total number of computed lower bounds for Experiment 3 to choose same, smart or random trees for the lower bounds.	167
4.10 Settings for Experiment 4.	171
4.11 Average time in milliseconds for Experiment 4 to solve a problem with 20 nodes and 30 arcs with 1 to 20 BBP and 10 to 200 initial bounds (IB).	172

4.12	Settings for Experiment 5.	173
4.13	Average time in milliseconds for Experiment 5 to solve a problem with 20 nodes and 20 to 44 arcs with 10 to 2000 initial bounds.	173
4.14	Settings for Experiment 6.	177
4.15	Average time in milliseconds for solving a problem with 30 nodes and 40 arcs for different number of BBP and GBBP in Experiment 6.	178
4.16	Settings for Experiment 7.	180
4.17	Average time in milliseconds for Experiment 7 to solve a problem with 20 nodes and 20 to 44 arcs with 1 GBBP and 1 to 19 BBP.	180
4.18	Average time in milliseconds for Experiment 7 to solve a problem with 30 nodes and 30 to 51 arcs with 1 GBBP and 1 to 29 BBP.	184
4.19	The number of instances with 30 nodes and 48 and 51 arcs that were not solved to optimality in 10 minutes in Experiment 7.	185
4.20	Best average time per instance in milliseconds for the two methods in Experiment 8.	186
4.21	The number of instances not solved in 10 minutes by IB and GBBP methods in Experiment 8.	187
4.22	Time distribution of solving an instance with 50 nodes and 62 arcs by the GBBP method in Experiment 8.	189
4.23	Settings for Experiment 9.	190
4.24	Average time in milliseconds for solving a problem with 100 nodes and 100 to 123 arcs for 2%, 5%, 10% tolerance limits in Experiment 9.	191

4.25	Number of instances that the algorithm failed to solve in 10 minutes for 2%, 5%, 10% tolerance limits in Experiment 9.	192
4.26	Settings for Experiment 10.	195
4.27	Average time per instance in milliseconds for solving two-layer problems in Experiment 10.	196
4.28	Number of instances not solved in 10 minutes by the two-layer algorithm in Experiment 10.	197
4.29	Settings for Experiment 11.	197
4.30	Average time in milliseconds for solving a two-layer problem with 10 components and 10 demand nodes using the two-layer and general network algorithms in Experiment 11.	198
4.31	Average time in milliseconds for solving a two-layer problem with 40 components and 40 demand nodes using the two-layer and general network algorithms in Experiment 11.	198
4.32	Number of instances that general and two-layer algorithms failed to solve in 10 minutes for problems with 40 components and 40 demand nodes in Experiment 11.	199

List of Algorithms

1	Path augmentation procedure	86
2	Labelling algorithm	111
3	Spanning tree algorithm from Graves and Willems [2000]	115
4	Service time computation	116
5	Tree Algorithm	120
6	Generate paths	125
7	Branch and bound algorithm	133

Chapter 1

Introduction

In this thesis we consider two major issues that have to be addressed in the supply chain of a manufacturing firm. On one hand, inventory across the chain has to be reduced to provide services as cheaply as possible with the least amount of assets. On the other hand, customers expect a high level of service, which includes on-time deliveries. Both issues depend on the amount of safety stocks the firm places in each location of its supply chain.

The problem being solved in the thesis deals with the amount of safety stock to hold at each stage of a supply chain. The amount must be such that a manufacturing company is able to serve its customers on time and satisfy most, if not all, of the demand. However, the amount of stock to hold must be small to minimize holding and storage costs. By solving the problem, a manufacturing company can protect itself against uncertain demand and provide a high level of service to its customers.

There are many ways of modeling safety stocks in supply chains. In the thesis, we follow the framework given in Graves and Willems [2000]. This approach falls

into the category of models that are distinguished by the assumption of guaranteed deterministic service times. In the paper, Graves and Willems [2000] formulate the safety stock problem for the general-network supply chain with a periodic review base-stock policy and without production capacity. They provide a pseudo-polynomial time algorithm for solving the problem for supply chains that can be modeled as a spanning tree.

Given the framework of Graves and Willems [2000], the primary goal of the thesis is to analyze the problem and to provide an algorithm for safety stock placement in general-network supply chains. We first show that the general problem is NP-hard. To do so, we reduce a known NP-hard problem, the vertex cover problem, to an instance of the general network safety stock problem.

Next, we characterize the optimal solution of the general-network problem. We find, that by constructing paths from a node (i.e., a stage in the supply chain) to any other node in the network, we can identify all possible candidates for the optimal service times for the nodes on the path. We show what kind of paths are valid for an optimal solution and how to construct a solution from a path.

We use the paths to develop a branch and bound algorithm for the general-network problem. By constructing all possible valid paths, we can consider all possible values for the optimal service times. Therefore, in the branching tree we enumerate the solutions according to the values that are generated by the combinations of valid paths.

To construct the lower bounds, we relax the problem to obtain a spanning tree and solve the problem restricted to the spanning tree by an algorithm similar to the

one presented in Graves and Willems [2000]. Moreover, we improve the algorithm in Graves and Willems [2000] from pseudo-polynomial to be polynomial. The spanning tree algorithm developed in the thesis is $O(N^3)$, where N is the number of nodes (stages) in the network (supply chain).

In addition to the general networks, we consider two-layer networks. We develop a specialized branch and bounds algorithm for these problems and show computationally that it runs faster than the general-network algorithm applied to the two-layer networks.

The structure of the thesis is as follows. In this chapter, we formulate the problem and discuss relevant literature. In Chapter 2, we show the complexity of the general network problem and characterize its optimal solutions. In Chapter 3 we develop a polynomial algorithm for supply chains that can be modeled as spanning trees. In the same chapter we present the branch and bound algorithms for the general and two-layer network problems. In Chapter 4, we show the results of computational experiments. In Chapter 5 we discuss the results presented in the thesis and identify opportunities for future research.

1.1 Assumptions and formulation

1.1.1 Assumptions

The following assumptions were used in the model introduced in Graves and Willems [2000]. Here, we make the same set of assumptions for the model without production

capacity.

- **Multi-stage network.** We model a supply chain as a network. Nodes and arcs of the network have natural interpretation in terms of the chain. Each node or stage in the network can be represented as a processing function as well as a location for a safety stock. We place an arc from node i to node j if the output product of stage i is needed as input for production at stage j . If a node is connected to several upstream nodes, then the node is an assembly requiring inputs from each of the upstream nodes. The nodes are potential locations for holding a safety-stock of the item processed at the node.

Due to the interpretation of the network we assume that the network does not have directed cycles. This fact says that a component once processed in a node can not return back to the node in an assembly with other components.

Let N be the number of nodes and \mathbb{A} be the set of arcs in the graph representing the chain.

- **Production lead-times.** We assume that each node j has a deterministic production lead-time T_j , where lead-time is the total time of production, including queueing, given that all necessary components are available.

Here we also introduce the maximum replenishment time for a node j :

$$M_j = T_j + \max_i \{M_i \mid i : (i, j) \in \mathbb{A}\}.$$

The maximum replenishment time is the length of the longest directed path

(with arc lengths T_j) in the network that terminates at node j , and represents the longest possible time to replenish the inventory at node j .

- **Base-stock replenishment policy.** All stages operate under a periodic-review base-stock policy with a common review period. We assume that there is no delay in ordering, therefore, all the nodes see customer demand once it occurs in the demand nodes. Based on the observed demand, each stage replenishes its inventory up to the bases stock level.
- **Demand process.** We assume that external demand occurs only in the demand nodes, which we define to be the nodes with zero out-degree. We denote the set of demand nodes as \mathbb{D} . For each node j in \mathbb{D} demand $d_j(t)$ comes from a stationary process with average demand per period μ_j .

Any other node $i \notin \mathbb{D}$ has only internal demand from its successors. We can calculate the demand in node i at time t by summing the orders placed by its immediate successors:

$$d_i(t) = \sum_{(i,j) \in \mathbf{A}} \theta_{ij} d_j(t),$$

where a scalar θ_{ij} is associated with each arc representing the number of units of upstream component i required per downstream unit j . From this relationship, we find the average demand rate for the node i to be

$$\mu_i = \sum_{(i,j) \in \mathbf{A}} \theta_{ij} \mu_j.$$

The most important assumption of the model is that demand is bounded. In

particular, for each node j there exists a function $D_j(F)$ for $F = 1, 2, \dots, M_j$, such that

1. for any period t

$$D_j(F) \geq d_j(t - F + 1) + d_j(t - F + 2) + \dots + d_j(t);$$

2. $D_j(0) = 0$;
3. the function is concave and increasing for $F = 1, \dots, M_j$;
4. $D_j(F) - F\mu_j$ is increasing in F .

The assumption of bounded demand can have several interpretations. For example, we can say that the bounded demand reflects the maximum demand that a stage will see. However, in some contexts, the demand is truly stochastic and can be hard to bound. In this case, we can say that the bound reflects the maximum demand that the company wants to satisfy from its safety stock. Under this interpretation, if the demand exceeds the bound, the company will not have enough safety stock to satisfy the demand and will have to take extraordinary measures like overtime and expediting to handle the excess demand.

- **Guaranteed outbound service times.** We assume that node j provides 100% service and promises a guaranteed service time S_j to its downstream nodes. This means that demand $d_j(t)$ that arrives at time t must be filled at $t + S_j$. Note, we assume that each non demand node j quotes the same service time to each of its downstream nodes $i : (j, i) \in \mathbb{A}$.

Also, we impose bounds on the service times for the demand nodes, i.e., $S_j \leq s_j$, $j \in \mathbb{D}$, where s_j is a given input that represents the maximum service time for the demand node j . The maximum service time is a parameter of the model known to the end customer. For example, if node i wants to serve its customers immediately, the firm has to set $s_i = 0$.

- **Guaranteed inbound service times.** Let SI_j be the inbound service time for the node j . We define inbound service time to be the time for the node j to get all of its inputs from nodes i : $(i, j) \in \mathbb{A}$ and to commence production. We require that $SI_j \geq S_i$ for all arcs $(i, j) \in \mathbb{A}$, since stage j cannot start production until all inputs have been received. We can show that, if the objective is to minimize the cost of the safety stock held in the chain, there exists an optimal solution with:

$$SI_j = \max_{(i,j) \in \mathbb{A}} S_i.$$

All the parameters described here are known except for the outbound and inbound service times. These service times are decision variables for the optimization.

1.1.2 Formulation

Suppose B_j is the base stock level for a node j and $I_j(t)$ is inventory in j at time t . At time t , stage j observes demand $d_j(t)$ and starts replenishing the demand. It places an order for the input materials to the upstream nodes and replenishes the demand at the time $t + SI_j + T_j$. However, the node guarantees to satisfy the demand at time $t + S_j$. Therefore, if $t + S_j < t + SI_j + T_j$, the stage has to always store inventory to

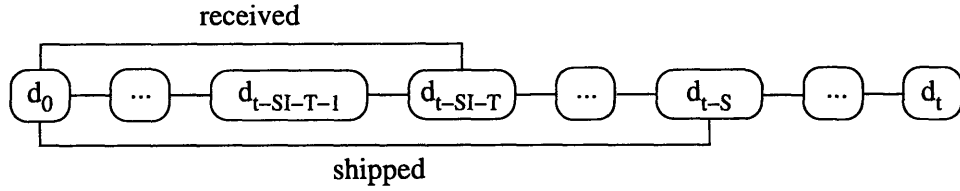


Figure 1-1: Base-stock mechanics.

cover the time interval of $SI_j + T_j - S_j$. This interval is called the net replenishment time and we will see that the inventory that covers the interval is the base-stock level.

We establish the relationship between the base stock level and the amount of the inventory. Figure 1-1 shows the timeline of the events from time 0 to time t . By the time t , all the demand that occurred prior to and including $t - S_j$ has to be satisfied, because node j guarantees 100% service within S_j period of times. At the same time, the node has to wait SI_j units of time to receive all the necessary materials to begin production and T_j units of time for its own production. Therefore, at time t , the node has replenished all of the demand that arrived before and including the time $t - SI_j - T_j$.

If $t - S_j < t - SI_j - T_j$, then node j has to store the inventory that would satisfy the demand occurred between $t - S_j$ and $t - SI_j - T_j$. Instead, it would be cheaper to delay replenishment such that the order arrives to the downstream nodes exactly when it is needed. This way, node j would not need to store unnecessary inventory. Therefore, it is always better in this case to set the outbound service time equal to the inbound service time plus the production lead time: $S_j = SI_j + T_j$.

Consequently, we only consider the case when $t - S_j \geq t - SI_j - T_j$. In this case, the node has to have the amount of inventory to cover the demand over the interval

$(t - SI_j - T_j, t - S_j]$. Therefore, the demand in the interval has to be covered from the inventory or by subtracting the demand from the base stock level. Then the finished inventory at the stage j at the end of period t is

$$I_j(t) = B_j - d_j(t - SI_j - T_j, t - S_j),$$

where $d_j(a, b)$ denotes demand at stage j over the time interval $(a, b]$.

To provide 100% service level, we require $I_j(t) \geq 0$. To satisfy this requirement, we set the base stock B_j equal to the maximum demand over an interval of length $SI_j + T_j - S_j$, namely $B_j = D_j(SI_j + T_j - S_j)$. Hence, the expected inventory at the stage j is

$$D_j(SI_j + T_j - S_j) - (SI_j + T_j - S_j)\mu_j,$$

which represents safety stock held at the stage j .

Now, we formulate the problem \mathcal{P} of finding optimal guaranteed outbound service times S_j , $j = 1, \dots, N$ and inbound service times SI_j , $j = 1, \dots, N$ in order to minimize total cost of safety stock in the chain.

$$\begin{aligned} \mathcal{P} \quad & \text{minimize} \quad \sum_{j=1}^N h_j \{D_j(SI_j + T_j - S_j) - (SI_j + T_j - S_j)\mu_j\} \\ & SI_j + T_j - S_j \geq 0, \quad j = 1, \dots, N \\ & S_i \leq SI_j, \quad (i, j) \in \mathbb{A} \\ & S_j \leq s_j, \quad j \in \mathbb{D} \\ & S_j, SI_j \geq 0, \text{ integer} \quad j = 1, \dots, N \end{aligned}$$

where h_j denotes the per-unit holding cost for inventory at stage j .

This is a problem of minimizing a concave function over a polyhedron. The objective function is the total cost of holding safety stock in the supply chain which consists of individual costs in each node. The first constraint insures that the net replenishment time is nonnegative and the objective function is well defined. The second constraint makes the inbound service time to be no greater than the outbound service time of its immediate suppliers. This is because a node can not commence production before all inputs are available. The next constraint reflects the requirement that all the outbound service times S_i of the demand nodes, which are set up by the chain, are no greater than the guaranteed service times quoted to the end customers. Finally, we require that all the service times are integers, so that the service times are expressed in the integral multiples of the underlying planning period, which might be a day or a week.

1.2 Literature Review

1.2.1 Safety stock modeling

The literature on safety stocks and inventory is very broad. Reviews of different modeling approaches can be found in Graves [1988], Graves and Willems [2003b], Minner [2000], Minner [2003]. Generally, we distinguish two types of inventory models: guaranteed service models and stochastic service models. The model in the thesis was initially stated in Graves and Willems [2000] and falls into the category of the models

with guaranteed deterministic service times. Here, we discuss relevant papers in both categories.

Deterministic service time models

Before reviewing the literature of the deterministic service models, we first provide a general overview of the approach. We discuss the assumption in section 1.1.1. In particular, the approach adopts the network representation of the supply chain and assumes that each stage can quote a service time that it can always satisfy. In contrast, stochastic service models assume that the service time between the stages will vary depending on the availability of the materials at the stage. To make the service time deterministic, the key assumption of the models is that the demand is bounded. This assumption allows the stages to be able to guarantee a service time with a finite amount of inventory. Consequently, in the guaranteed service setting, each stage has to hold sufficient inventory to satisfy the service time commitment. Therefore, the goal of models of this type is to determine the internal service times to minimize the total cost of holding inventory in supply chains. The most relevant research papers of this category are Simpson [1958], Graves and Willems [2000], Inderfurth [1991], Inderfurth and Minner [1998], Minner [1997], Magnanti et al. [2004].

The papers adopt the framework of Kimball [1988], whose paper was originally written in 1955. Kimball describes a one-stage inventory model that operates under a base-stock policy and requires that customer demand be satisfied within a specific service time. Unlike the production lead-times that are known a priori (see section 1.1.1), the service times are set according to the company's policy. In this

framework, the inventory is minimized to assume that the production system can meet "normal" demand within the service time. The notion of "normal" demand can not be set without risk which comes from the fact that customer demand is unknown. Generally, the normal demand can be described as the demand that falls into certain range most of the time. If an abnormal demand occurs, the author assumes that the system deals with it in some other way such as expediting or overtime. However, since the demand falls into the normal range most of the time, Kimball proposes to set an upper bound of the demand for each time interval. Then, the value of the bound over the replenishment time can serve as the base-stock level.

Simpson [1958] extends Kimball's principles of inventory control to accommodate a serial production system. As before, each stage of the system operates under a base-stock policy. In addition, the author describes the demand propagation along the system: "an order against the inventory for a finished item is immediately transmitted all the way back along the line to all the manufacturing operations". Again, each stage i of the system operates with a deterministic service time S_i , or the time within which the stage has to fill the order placed by the downstream stage. Inventory at stage i is designed to handle the demand up to a certain level $F_i(t)$ over the time t , where $F_i(t)$ is an upper bound on the demand at stage i over the time interval t . The base stock level is equal to the level $F_i(L_i)$ over the net replenishment time $L_i = T_i + S_{i-1} - S_i$, where T_i is the production lead time of stage i . The resulting minimization of overall inventory of the system appears to be minimization of a concave function over a polyhedron, where the service times are the decision variables. Simpson proves that the service time at node i has to be either 0 or equal to the replenishment time of

the node. This result corresponds to the "all-or-nothing" inventory stocking policy. Graves [1988] observes that the serial system can be solved as a simple dynamic program.

Inderfurth [1991] extends the Simpson model to the general divergent system model. He adopts the same framework with the deterministic service times that are quoted by the upstream nodes to the downstream. Inderfurth proves the same "all-or-nothing" safety stock policy for such systems. The author then describes a simple dynamic program of determining the safety stock.

General supply chain networks are considered in Inderfurth and Minner [1998]. The paper conveys two interesting points. First, it defines the net replenishment time for a node in a general system. Second, the authors show that the replenishment time of node i in a convergent system under two different service levels is a certain combination of the service times and lead times of the upstream nodes. Definition of the net replenishment time in node i is the following:

$$\max_{j:(j,i) \in A} \{S_j\} + T_i - S_i,$$

where A is the arc set. The definition is essentially the same as the one given in Graves [1988], where the author assumes that the service times of nodes $j : (j, i) \in A$ are the same, because "there is no value from having shorter service times for a subset of the input items". Then the safety stock has to cover the demand over the net replenishment time.

In the paper of Graves and Willems [2000] the authors state the problem of finding

the safety stock in supply chains as it is formulated in the thesis. Mathematically, the problem is a concave minimization over a polyhedron. Instead of one set of service times, two sets are introduced: inbound and outbound service times. The outbound service time S_i is the same service time as in previous papers, i.e., the time within which node i guarantees to fill the order of the downstream nodes. The outbound service time of node i quoted to the downstream nodes $j : (i, j) \in A$ are assumed the same. A discussion about maintaining different outbound service times can be found in Graves and Willems [1998]. The inbound service times are defined as

$$SI_i = \max\{S_i - T_i, \max_{(j,i) \in A} \{S_j\}\}.$$

The definition simply states that the inbound service time of stage i can not be less than the service times quoted to the stage by its direct suppliers. It also says that the net replenishment time $SI_i + T_i - S_i$ can not be negative. The authors do not discuss necessary conditions of the optimal solution for the problem; instead, they give a dynamic programming algorithm for a spanning tree network structure.

In the next four papers, the authors look at more general networks and propose algorithms of solving the problem of placing safety stock. Humair and Willems [2003] provide an algorithm for a network that consisting of clusters of commonality, where a cluster of commonality is a two-layer general network. They assume that when each cluster is replaced by a single node, the resulting network is a spanning tree. Lesnaia [2003a] describes the structure of the solution for the two-layer networks with the requirement that the nodes i that face demand quote the outbound service times to

the end customers such that $S_i \leq T_i$. In this thesis, we provide an algorithm for the networks without the restriction $S_i \leq T_i$. Unlike in Humair and Willems [2003], the algorithm we present here takes advantage of the structure of an optimal solution. In addition, we report on the computational experiments for the problems on two-layer graphs, while Humair and Willems [2003] have not presented any results from a computational experiment.

Under the same requirement $S_i \leq T_i$, Graves and Lesnaia [2004] characterize the optimal solution in the general networks. The papers Lesnaia [2003a] and Graves and Lesnaia [2004] propose branch and bound algorithms similar to the ones discussed in sections 3.3 and 3.2.

Another paper, with a solution of the general problem is Magnanti et al. [2004]. In the paper, the authors approximate the objective function with piecewise linear functions and solve the problem by a branch-and-bound algorithm using CPLEX. The authors report computational tests for randomly generated networks with up to 100 stages but with limited number of arcs. The authors report average computational time to solve a general network problem, however, the average is only taken for the instances that they were able to solve to global optimality. It is not clear, how many times the algorithm failed to find the optimal solution.

Papers by Lesnaia [2003b] and by Graves and Lesnaia [2004] are the building blocks of the thesis. Here, we want to study the safety stock problem for the general networks. In particular, we want to understand its complexity and propose a method of solving the problem.

Stochastic service time models

The main difference of the stochastic service approach from the guaranteed service approach is that the replenishment time at each stage of the supply chain network is stochastic. The models address three types of uncertainty: demand (volume and mix), process (yield, machine downtime, transportation reliability), and supply (part quality, delivery reliability). Each stage has to take into consideration the possibility that its upstream stages cause delays in the production in the stage. These delays are stochastic. Nevertheless, each stage is required to have enough inventory to meet a service level target requirement, which depends on the replenishment time.

The literature related to this approach is extensive. One of the first models of this category is the model by Clark and Scarf [1960]. In the paper, a serial system with no setup cost and periodic review is considered. The authors proved optimality of an order-up-to policy at each node of the system. This paper is a basis for the subsequent work in for the models of multistage inventory networks with centralized control.

A decentralized control model is the METRIC model of Sherbrooke [1968]. The model considers two echelons with one depot node that supplies n stocking locations of the second echelon. The model assumes a single item, stationary Poisson demand process, i.i.d. lead times and one-for-one replenishment policy. Sherbrooke develops an analytical approximate solution for characterizing the performance of stocking policies in the stages subject to a constraint on inventory investment. METRIC model and its extensions are relevant in dealing with distribution networks, especially for

spare parts inventory systems.

A general network model is studied in Lee and Billington [1993]. The one stage model assumes that the demand is normally distributed and a target service level or a target base stock levels are specified. Moving to the multi-stage system, the demand propagation is simply modeled as generation of the demand for the upstream stages by the amount needed at their downstream stages. Then, the replenishment lead time consists of the standard lead time for the transfer of materials from upstream stages, the delay incurred in this transfer, and the actual production time. The stages that face end-customer demand have target service levels. The other stages have either service or base-stock levels and are decision variables. The authors develop a search heuristic to assign the best combination of service levels at the stages of the network to insure the end-customer service levels.

The model of Lee and Billington [1993] was extended in Ettl et al. [2000]. In this model, the authors assume a one-for-one replenishment policy, normally distributed demand and formulate a problem in terms of base-stock levels. The difference from the previous model is in the description of the lead time. Ettl et al. [2000] distinguish between a nominal lead time, which is a given data, and an actual lead time, which is derived by taking into account stockout possibilities at the supply stores based on the queueing analysis. They solve a nonlinear optimization, which minimizes the total average dollar value of inventory in the network, subject to meeting the service requirements of the end customers. The problem is solved by the conjugate gradient method.

Subsequent papers in this area include Song and Yao [2002] with an analysis and

algorithms for the performance measures for the optimization problems that seek trade-off between inventory and customer service in the assemble-to-order systems. Liu et al. [2004] adopt the inventory-service optimization framework of Ettl et al. [2000], but focus on the queueing delays that occur due to the production capacity. In general, more systematic overview of the stochastic service models can be found in Zipkin [2000] and Graves and Willems [2003b].

1.2.2 Complexity of concave minimization

Problem \mathcal{P} as stated in section 1.1.1 is a minimization of a concave function over a polyhedron. Here we discuss the complexity of such problems in general and later, in section 1.2.4, will review the complexity results for the problem \mathcal{P} itself.

The general concave minimization problem is known to be NP-hard. The proof of this fact can be found in Sahni [1974] or in Vavasis [1990]. For example, in Vavasis [1990], the author shows that quadratic programming problems are NP-hard by transforming an instance of SAT (Korte and Vygen [2002]) to an instance of quadratic program:

$$\begin{aligned}
 \min \quad & \sum y_i(1 - y_i) \\
 \text{s.t.} \quad & Ay \leq b \\
 & 0 \leq y_i \leq 1
 \end{aligned} \tag{1.1}$$

The problem is a special case of the concave minimization problem. Therefore, concave minimization is NP-hard.

A special case of concave minimization problems is separable concave minimization

problems. The problems can be generally stated as

$$\begin{aligned} \min \quad & \sum_{i=1}^n f_i(x_i) \\ \text{s.t.} \quad & x \in D \cap C \subseteq \mathfrak{R}^n \end{aligned} \tag{1.2}$$

where D is a compact convex set and $C = \{x \in \mathfrak{R}^n : \alpha_i \leq x_i \leq \beta_i\}$. This kind of problems is of interest in the context of the safety stock problem, because the problem as we state in the thesis is separable concave with linear constraints. However, problem 1.1 is separable concave and, therefore, problem 1.2 is NP-hard.

Attempts to solve the concave minimization problem through piecewise linear concave approximation also lead to solving hard problems. Generally, minimizing piecewise linear concave problem on a polyhedral set is NP-hard, because the general complementarity problem, which is NP-complete (Chung [1989]), can be reduced to such problem (Mangasarian [1978]).

1.2.3 Methods

Extensive reviews of most common methods developed for continuous concave minimization problems are given in Benson [1994], Benson [1996], Horst [1984], Pardalos and Rosen [1986], which we use as references for the section. There are two classes of approaches to concave minimization: deterministic and stochastic. Here, we only consider deterministic approaches. Most popular deterministic approaches are: *enumeration*, *cutting plane*, *successive approximation* and *branch and bound*. This division of the algorithms is only theoretical. In reality, most of the algorithms are combinations

of the approaches. In addition, several techniques were suggested for the problems with specific objective functions, such as quadratic, separable, factorable, etc., or when the feasible set has some specific geometry.

Enumerative methods rely on the fact that if a continuous concave function is defined on a nonempty bounded polyhedron, then it attains minimum in one of the vertices of the polyhedron. Because the polyhedron has only a finite number of extreme points, we only need to check a finite number of possibilities. Enumerating all the extreme points and then searching gives an exact global minimum of the function.

The first enumerative approach was described in the paper of Murty [1969], who introduced the concept of extreme point ranking. The idea of the method is to rank the extreme points of polyhedron according to some linear function $l(x)$ that underestimates the objective function $z(x)$. The first ranked point x_0 is the minimum of $l(x)$ over the polyhedron. At each iteration of the algorithm, new point x_k joins the list of previously ranked points $E^{k-1} = \{x_0, \dots, x_{k-1}\}$, where $l(x_i) \leq l(x_{i+1})$. x_k is a point that minimizes $l(x)$ over the set of extreme points adjacent to E^{k-1} . At the same time, an upper bound UB is calculated. An upper bound is the minimum of $z(x)$ over the set E^k . The procedure terminates when $l(x_k) \geq UB$.

Subsequent literature addressed two issues with the described algorithm. The first issue is finding the next best extreme point. Discussion of this topic can be found in Taha [1972], Taha [1973], McKeown [1975]. The second issue is about finding a better underestimator. This question was addressed in Cabot [1974], Cabot and Francis [1970], Taha [1973], McKeown [1975], Pardalos and Rosen [1987].

A survey with comparison of the methods and computational effectiveness is pub-

lished in McKeown [1978].

Cutting plane approach was first introduced in Tuy [1964]. The method solves the problem of minimizing a concave function subject to linear constraints. The idea is to detect a local minimum and generate a cone adjacent to the vertex. Then, the algorithm cuts parts of the feasible region using Tuy cuts creating subcones. The method solves subproblems on the subcones. According to Zwart [1973], the algorithm does not guarantee finiteness. However, subsequent methods based on Tuy's idea employed other techniques like branching, estimation to guarantee the finiteness and to avoid cycling. Relevant literature is Zwart [1974] with a computationally finite algorithm, Jacobsen [1981] with a general concave minimization Tuy-type algorithm, Thoai and Tuy [1980] with branch and bound technique, Glover [1973] with an approach to the cut search.

Further developments in the area relate to using γ -cuts and facial-cuts. Cabot [1974] suggested using γ -cuts instead of Tuy cuts when an extreme point is degenerate. Horst and Tuy [1993] introduced a finite algorithm based purely on γ -cuts. A generalized version of an algorithm based on γ -cuts can be found in Benson [1999]. An advantage of using Tuy cuts and γ -cuts is that they are easy to construct. However, they become shallower as the algorithm progresses. To avoid the issue, Majthay and Whinston [1974] suggested to use facial cuts. The cuts depend only on the geometry of the feasible region and remove at least one face of the region which does not need further exploration. Facial cuts are hard to construct, however, the algorithm terminates with the a solution after a finite number of iterations.

Successive approximation methods usually construct a sequence of simpler

problems with improving solutions. The result of the algorithms is either an exact global minimum or its approximation. Most commonly used approximation techniques are outer and inner approximations of the feasible region and approximation of the objective function.

Outer approximation via inequality cuts was proposed in Horst et al. [1987] and Horst et al. [1989]. The main idea of the method is to construct a polyhedral convex sets D_k that contain the feasible region D of the problem and which vertices are easy to compute. The optimal solution of the problem over the polyhedral set gives a lower bound on the optimal solution of the original problem. Then tighter polyhedral sets are constructed $D_1 \supset D_2 \supset \dots \supset D$ with solutions of the problem over the sets forming a sequence converging to an optimal solution of the problem.

Inner approximation (or polyhedral annexation) approach uses a similar idea, but the polyhedral sets approximate the feasible region from the inside: $D_1 \subset D_2 \subset \dots \subset D$. The procedure was described in Mukhamediev [1982] and further developed in Tuy [1990]. Tuy [1990] also discussed the advantages of the polyhedral annexation versus outer approximation methods.

Finally, the third significant approximation approach is successive approximation of the objective function. One of the first algorithms was developed by Falk and Hoffman [1976]. The Falk-Hoffman algorithm uses an outer polyhedral approximation $D_1 \supset D_2 \supset \dots \supset D$ with a convex envelope of the objective function taken over D_k at step k . At each iteration, the method solves a linear program. The algorithm is finite and, unlike the outer approximation algorithms, gives a feasible solution at each iteration.

Branch and bound approach is based on dividing the constraint space and inferring relevant information about the objective function on the partitions. At each iteration of the method, previously constructed part of the feasible region is divided into more pieces. For each piece we find information about the possibility of the global minimum belonging to the piece. In branch and bound method (see Korte and Vygen [2002]), which was first introduced by Land and Doig [1960], the information is a lower and an upper bound on the objective function if defined on the piece. If the lower bound is greater than an upper bound for another piece, the considered piece can be disregarded in the further partitions.

This method depends on how well we can construct the partitioning procedure as well as on the quality of the information. If the partitioning is finite, in the worst case, the method finds an optimal solution in exponential time, because at least two pieces are produced from every previously constructed piece. In this case the method is no better than enumerating all possible solutions. To prevent the possibility, one could stop the algorithm after some fixed amount of time or set up a tolerance $\epsilon > 0$ (see Bertsekas [2000]) . However, it will not guarantee optimality of the best solution found.

The branch and bound method has been most popular in approaching concave minimization problems. The popularity is partially due to the fact that the method is applicable not only for polyhedra, but for the general convex sets. The earliest algorithms were proposed by Tuy [1964], Falk and Soland [1969] and Horst [1976]. Tuy's algorithm uses enumerative approach with cones to separate the feasible region. Falk-Soland algorithm uses hyperrectangles as the partition elements, while Horst

algorithm uses simplices.

Falk and Soland [1969] proposed a branch and bound algorithm for separable nonconvex programming problems (1.2) which we formulated in section 1.2.2. The algorithm uses convex envelopes ϕ_i of function f_i on $[\alpha_i, \beta_i]$. Then $\phi = \sum_{i=1}^n \phi_i$ is the convex envelope of $f(x)$ on C . The algorithm solves a sequence of linear or convex subproblems, which are used in the branch and bound algorithm. Falk and Soland [1969] proposed two partitioning rules that lead to convergence.

Extensions of the Falk-Soland algorithm include the algorithm of Soland [1974] that specializes for the case when D is a polyhedron. Benson [1990] used outer approximation to speed convergence of Falk-Soland method. Shectman and Sahinidis [1998] proposed a finite algorithm for the separable concave programs. In the paper by Horst [1976], the author developed an algorithm for a general nonconvex problem

$$\min \{f(x) : x \in D \subseteq \mathfrak{R}^n\},$$

where D is compact and $f(x)$ is a continuous function. Instead of rectangular partitions as in Falk and Soland [1969], general partitions in forms of simplices are used. Also, the algorithm handles general functions and does not rely on the concept of convex envelopes. It is shown in the paper that at least one accumulation point generated by the algorithm solves the problem. Finally, Benson [1982] proposed a general prototype algorithm, where the Falk-Soland and Horst algorithms are the special cases of the algorithm. Benson presented a new convergence property for the algorithms and shows that each accumulation point under the property is a global

minimum.

1.2.4 Complexity and algorithms for the safety stock problem

As we have already seen in part 1.2.2, general concave minimization problems are NP-hard. However, because the safety stock problem is defined on a particular polyhedron, in some cases we can still develop a polynomial time algorithm. For example, Graves [1988] observes that the Simpson's serial system case can be solved by a dynamic program. In Inderfurth [1991], a dynamic programming algorithm for a convergent system is described. Graves and Willems [2000] develop a dynamic programming algorithm for the spanning tree networks which runs in $O(NM^2)$ (see also Graves and Willems [2003a]), where N is the number of nodes and M is the maximum replenishment service time, which is bounded from above by $\sum_{i=1}^N T_i$.

There has been no complexity results for the problem \mathcal{P} of safety stock minimization, except for Shen [2003]. In the note, Shen [2003] proves that a similar problem with the upper bounds on the outbound service times is NP-hard. In this thesis, we prove that the general problem is also NP-hard. This development would justify the choice of the algorithms used in Graves and Lesnaia [2004] and in Magnanti et al. [2004].

Chapter 2

Problem Characteristics

2.1 Complexity of the problem

2.1.1 General case

We first determine the complexity of the problem \mathcal{P} stated in section 1.1.2. We show that the problem is NP-hard by reducing a known NP-hard problem to the safety stock problem. This will justify the choice of the algorithms we develop in the thesis. In particular, because no polynomial algorithm can be found at this moment, it is reasonable to implement branch and bound algorithms.

The idea of the proof appeared first in the unpublished note by Shen [2003]. In this note, the author reduces the Vertex Cover problem, which is known to be NP-complete, to a modified safety stock problem. The modified problem is essentially the same problem as problem \mathcal{P} , except that each node has an additional constraint on its outbound service time. The author assumes that the outbound service time for

each node is bounded from above. That means, for each node i , there exists a service time s_i , such that the outbound service time is constrained as $S_i \leq s_i$. Note, that problem \mathcal{P} also has similar service time constraints, but only for the demand nodes \mathbb{D} , while the outbound service times for the rest of the nodes are not constrained in this way. However, we found a way of reducing an instance of the minimum-size Vertex Cover problem, which is NP-hard, to an instance of problem \mathcal{P} such that a solution of problem \mathcal{P} will imply a solution of the Vertex Cover instance.

We first describe the Vertex Cover problem. A vertex cover in graph G is a subset V of vertices of G such that every edge of G is incident to at least one vertex in V (see Korte and Vygen [2002]). Then the optimization Vertex Cover problem for a graph is to find a vertex cover of minimum cardinality. This problem is NP-hard.

Now, we show how to reduce an instance VC of the Vertex Cover problem to an instance P_i of problem \mathcal{P} . Suppose, instance VC is characterized by an undirected graph G with N nodes and M edges and we want to find a minimum vertex cover. Then we perform the following steps:

1. **Make a directed graph from G .** We can arbitrarily assign directions to the edges of G . The only condition that has to be satisfied while doing so is that the directed graph has to have no directed cycles. One way of satisfying the condition is by following a simple algorithm. We first create set U with all the nodes and an empty set L . We then choose node $i \in U$ and assign direction to each arc (i, j) , $j \in U$ from node i to node j . After that we move node i to the set L and remove all the edges (i, j) . Then we pick another node from U

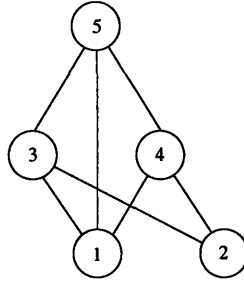


Figure 2-1: Graph G for the problem VC .

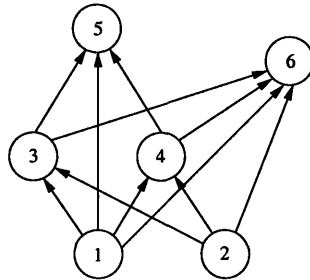


Figure 2-2: Graph G' for the problem P_i .

and repeat the procedure until the set U is empty. This simple algorithm will produce a directed graph with M directed edges in polynomial time, since at each step we set the direction of each edge such that the nodes in the set L are predecessors of the nodes in the set U .

As in the supply chain network, we will call the nodes with zero outdegree as demand nodes.

2. **Create a new node.** We create a new node $N+1$ such that every non demand node j has an edge $(j, N+1)$ directed from j to $N+1$. Let us denote the directed graph with the new node as G' . Figure 2-1 shows an example of an undirected graph G , while Figure 2-2 shows a way of transforming the graph into graph G' .

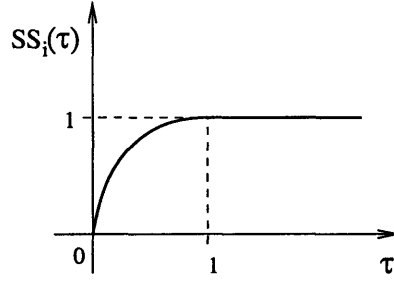


Figure 2-3: An example of the safety stock function.

3. **Assign parameters.** We define the safety stock function for each node of the graph G' . For simplicity, we call the safety stock at each node i as

$$SS_i(\tau_i) = D_i(\tau_i) - \mu_i(\tau_i),$$

where $\tau = SI_i + T_i - S_i$.

Then we require that $SS_i(\tau_i)$ is continuous, concave and satisfies

$$SS_i(\tau_i) = \begin{cases} 0, & \tau_i = 0 \\ \sqrt{\tau_i} & 0 < \tau_i < 1 \\ 1, & \tau_i \geq 1 \end{cases} \quad (2.1)$$

Without loss of generality, we can assume that the function is $\sqrt{\tau_i}$ on the interval $(0, 1)$, since τ_i takes only integer values. The most important properties of the function is that it is equal to 0 when $\tau_i = 0$ and is equal to 1 for all the other integer $\tau_i \geq 1$. Figure 2.1 shows an example of the safety stock function.

Next, we assign the per-unit cost of a safety stock:

$$h_i = \begin{cases} 1, & i = 1, \dots, N \\ N + 1, & i = N + 1 \end{cases}$$

The lead times for the nodes are:

$$T_i = \begin{cases} 1, & i = 1, \dots, N \\ 0, & i = N + 1 \end{cases}$$

Finally, the service times promised to the end customers are 1, i.e., $s_i = 1$ for all demand nodes i . As in the formulation of problem \mathcal{P} , by demand nodes we mean the nodes with zero outdegree.

The procedure described above polynomially transforms graph G to an instance P_i of the safety stock placement problem \mathcal{P} . Now, we show that an optimal solution of problem P_i determines an optimal solution of the Vertex Cover problem VC for graph G .

Lemma 1. *Suppose we have instance P_i of the safety stock problem \mathcal{P} . Then it is optimal not to hold any stock in node $N + 1$.*

Proof. We first show that the feasible region is not empty by constructing a solution with the cost N . Such solution is

- $S_{N+1} = 1, SI_{N+1} = 1$;
- $S_i = 0, SI_i = 1$ for $i \leq N$.

The solution is feasible. Indeed, nonnegativity of net replenishment time $SI_i + T_i - S_i$ is satisfied for every node. Every demand node i , including node $N + 1$, has $S_i \leq 1$. Also, for each arc (i, j) , $S_i = 0$ and $SI_j = 1$, hence, $S_i \leq SI_j$ is also true. Therefore, the solution is feasible and the feasible region is not empty. We note also, that the solution gives cost N , since each node $i \leq N$ contributes cost 1 and node $N + 1$ contributes 0 to the overall cost.

Now, we prove the statement of the lemma. Indeed, we notice that the total value of the safety stock cost in all the nodes other than $N + 1$ is at most N . This is due to the fact that the maximum of the safety stock function $SS_i(\tau)$ is 1 and holding cost $h_i = 1$ for all $i \neq N + 1$. All the unknown variables in the optimal solution take discrete values $0, 1, 2, \dots$. Therefore, if a node $i \neq N + 1$ holds non zero stock, then the holding cost in the node is 1.

On the other hand, if node $N + 1$ holds any stock, then $\tau_{N+1} = 1, 2, \dots$. Thus, in this case the value of the safety stock function $SS_{N+1}(\tau_{N+1}) = 1$ and it contributes $N + 1$ to the total holding cost, since $h_{N+1} = N + 1$.

From this we conclude that if node $N + 1$ holds stock, the total cost is at least $N + 1$. If the node does not hold any stock, then the cost of holding inventory is at most N . Therefore, we conclude that it is always better not to allow node $N + 1$ to hold any stock. □

Lemma 2. *Suppose we have instance P_i of the safety stock problem \mathcal{P} . Then in an optimal solution for each node i in the network G' we have $S_i \leq 1$.*

Proof. The statement of the lemma follows from lemma 1. We showed, that for node

$N+1$, $\tau_{N+1} = SI_{N+1} + T_{N+1} - S_{N+1} = 0$. Then, since $T_{N+1} = 0$ and $S_{N+1} \leq s_{N+1} \leq 1$, we have $SI_{N+1} \leq 1$. From the constraints of the problem P_i , $S_j \leq SI_i$ for all the arcs (i, j) in G' . Therefore, for every i connected to $N+1$, $S_i \leq 1$. Also, we know that for every demand node j , $S_j \leq 1$, which we imposed by construction of problem P_i . Therefore, we can conclude that for every node i , $S_i \leq 1$. \square

Lemma 3. *Suppose we have instance P_i of the safety stock problem \mathcal{P} . Then, in an optimal solution, for every arc (i, j) , $i, j \neq N+1$ it is impossible to have values for τ_i and τ_j such that $SS_i(\tau_i) = 0$ and $SS_j(\tau_j) = 0$.*

Proof. In lemma 2, we showed that each optimal solution of problem P_i satisfies $S_i \leq 1$. Suppose now there is an arc (i, j) , $i \neq N+1$ such that $SS_i(\tau_i) = 0$ and $SS_j(\tau_j) = 0$. That is, we suppose $\tau_i = SI_i + T_i - S_i = 0$ and $\tau_j = SI_j + T_j - S_j = 0$. Then $SI_j = S_j - T_j = S_j - 1 \leq 1 - 1 = 0$. Since $SI_j \geq 0$, we have $SI_j = 0$. Because $S_i \leq SI_j = 0$, $S_i = 0$. However, $SI_i + T_i - S_i = SI_i + 1 = 0$ by assumption and $SI_i \geq 0$. Therefore, we found a contradiction, which proves the lemma \square

From the lemma, we conclude that for every arc $(i, j) : i, j \neq N+1$, in an optimal solution, at least one node i or j holds safety stock. Therefore, the nodes with positive safety stock form a vertex cover V for the graph G . Moreover, by construction of the cost function, each node that holds safety stock contributes cost 1 to the objective function of the safety stock problem. Therefore, the objective function value is equal to the cardinality of the vertex cover V . By solving the safety stock problem, we find the minimum cost of safety stock which equals the cardinality of a vertex cover of graph G .

To prove that we can find a minimum vertex cover by solving the safety stock problem P_i , we only need to prove that the minimum of problem P_i does not depend on the orientation of the graph. Step 1 assigns the orientation to graph G arbitrarily, which determines the demand nodes and the relationship between the variables. If arc (i, j) is directed from node i to node j , then the corresponding constraint is $S_i \leq SI_j$. If, however, the orientation is reversed, the constraint is $S_j \leq SI_i$. Therefore, the problems are theoretically different and can give different solutions. Consequently, we have to show that they indeed give the same solutions independent of the orientation.

For the purposes of the next lemma, we introduce the definition of *vertex cover assignment* or *v.c. assignment* of the safety stock problem. A v.c. assignment is a distribution of the safety stock in the nodes of a graph such that for every arc (i, j) the assignment implies holding stock in i or in j or in both. We notice, that for the problem P_i , a v.c. assignment of the safety stock creates a vertex cover of graph G . This is because the cost of holding stock in a node is always one, which is equivalent to putting the node into the vertex cover set. However, in the safety stock problem setting, we refer to the v.c. assignment and in the VC problem setting - to the vertex cover.

Lemma 4. *Suppose we have a directed graph G and safety stock problem P_i on $G' = G \cup \{N + 1\}$ as described above. Then for every v.c. assignment of safety stock on G , there exists a feasible solution of the safety stock problem P_i .*

Proof. To prove the lemma we explicitly construct a solution. We consider any node i of graph G . Depending on the v.c. assignment, the node holds or does not hold

safety stock in i .

- Zero stock in i : $SI_i = 0, S_i = 1$.
- Nonzero stock in i : $SI_i = 1, S_i = 0$.

To specify a solution for all the nodes of problem P_i , we set $SI_{N+1} = 1$ and $S_{N+1} = 1$.

The solution is feasible. First, we see that for every demand node i , $S_i \leq s_i = 1$. It is also obvious that $SI_i + T_i - S_i \geq 0$ is satisfied for this solution. In the zero stock case, the solution gives $SI_i + T_i - S_i = 0 + 1 - 1 = 0$ and indeed implies zero stock. In the nonzero stock case, it satisfies $SI_i + T_i - S_i = 1 + 1 - 0 = 2$ and implies stock $SS_i(2) = 1$.

Next, we have to check that the constraint $S_i \leq SI_j$ for any arc (i, j) is satisfied. Indeed, SI_{N+1} imposes a constraint on all the outbound service times $S_i \leq 1, i \notin \mathbb{D}$. The proposed solution clearly satisfies the constraint.

Now, consider node $i \in G$. Because the solution is a v.c. assignment, if node i has zero stock, all the nodes connected to i have to have nonzero stock. That means, if $j \in G$ is downstream of i , $SI_j = 1$ and constraint $1 = S_i \leq SI_j = 1$ is satisfied. If node j is upstream of i , $S_j = 0$ and the constraint $0 = S_j \leq SI_i = 0$ is satisfied again.

If node $i \in G$ has nonzero stock, then the solution again does not violate the constraint. Indeed, as we showed before, $S_j \leq 1$ for any node in graph G , therefore, $SI_i = 1$ does not violate constraints $SI_i \geq S_j$ for all arcs (j, i) . Because $SI_j \geq 0$, $S_i = 0$ does not violate constraints $S_i \leq SI_j$ for all arcs (i, j) .

Therefore, we conclude that the solution is feasible and this proves the lemma. \square

Lemma 4 shows, that every v.c. assignment is feasible. Since every v.c. assignment of the safety stock problem P_i is equivalent to a vertex cover on graph G , we conclude that for every vertex cover of graph G we can always find a feasible solution of problem P_i .

Corollary 1. *Suppose we want to find a minimum vertex cover on an undirected graph G . Then, transforming the problem into problem P_i and solving the problem optimally, we can find a minimum vertex cover independent of the orientation assigned during the transformation.*

Proof. By lemma 3, an optimal solution of problem P_i is a v.c. assignment on G with cost K . Suppose, there exists a transformation of VC problem into problem P'_i with different orientation and with strictly smaller cost $K' < K$. But the solution of problem P'_i is a v.c. assignment on G as well. Therefore, by lemma 4 there exists a solution of problem P_i with the same cost K' , which contradicts optimality of K .

We conclude that for any orientation of graph G , problem P_i gives an optimal solution to the VC problem. □

Corollary 1 shows that by solving the safety stock problem optimally, we solve the Vertex Cover problem for the graph G . We can conclude now that problem \mathcal{P} is NP-hard.

2.1.2 Restricted case: $s_i \leq T_i$

It is common that the service times promised to the end customers are less than the lead times in the demand nodes. For example, the service times are often set to 0,

and thus requiring immediate delivery. Nevertheless, problem \mathcal{P} is stated without any constraints on the parameters of the problem. Here, we are also interested in whether the same problem with

$$s_i \leq T_i, i \in \mathbb{D} \tag{2.2}$$

is NP-hard. The proof of the section 2.1.1 can not be applied here, since node $N + 1$ is a demand node with the lead time $T_{N+1} = 0$ and guaranteed service time $s_{N+1} = 1$. Therefore, we need to prove the case separately. To answer the question we construct a similar reduction of an instance of a Vertex Cover problem to an instance of the problem \mathcal{P} with constraints (2.2).

Here we describe the construction. Suppose an instance VC of the Vertex Cover problem is represented by an undirected graph G with N nodes. We reduce VC to an instance P_i of the safety stock problem that satisfies (2.2) as follows.

1. **Make a directed graph.** The same procedure as in section 2.1.1 applies.
2. **Create two new nodes.** The first node is a node similar to the one created previously. We number the node $N + 1$ and require that every non demand node i in the graph be connected to the node $N + 1$. The second new node, $N + 2$, is only connected to the node $N + 1$ such that the direction of the arc $(N + 2, N + 1)$ is from node $N + 2$ to node $N + 1$. Figure 2-4 shows a way of transforming the graph from Figure 2-1 into the graph G' .
3. **Assign parameters.** We first specify the safety stock functions. For all the

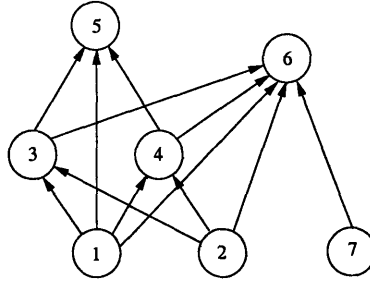


Figure 2-4: Graph G' .

nodes, including $N + 2$ and other than $N + 1$, the function is as described by equation (2.1). This time, the function for the node $N + 1$ is

$$SS_{N+1}(\tau) = (N + 1)\tau.$$

The function is linear in the net replenishment time and grows with the speed $N + 1$.

Per-unit cost of holding safety stock is

$$h_i = \begin{cases} 1, & i = 1, \dots, N, N + 1 \\ 2N + 2, & i = N + 2 \end{cases}$$

We assign lead-times T_i to be 1 for every node in the new graph. We assign guaranteed service times s_j to be 1 for all the demand nodes.

The described procedure creates instance P_i of the safety stock problem which satisfies requirement (2.2). Now we show that an optimal solution of problem P_i implies an optimal solution of problem VC .

Lemma 5. *Suppose we have instance P_i of the safety stock problem \mathcal{P} with additional requirement $s_i \leq T_i$ for the demand nodes $i \in \mathbb{D}$. Then it is optimal to have the net replenishment time τ_{N+1} equal to 1 for node $N+1$ and to hold no stock in node $N+2$.*

Proof. The feasible region of problem P_i is not empty. To show that, we construct a solution with the cost $2N + 1$. Such solution is

- $S_i = 0, SI_i = 1$ for $i \leq N$;
- $S_{N+1} = 1, SI_{N+1} = 1$;
- $S_{N+2} = 1, SI_{N+2} = 0$.

The solution is feasible. For every node of the network, $SI_i + T_i - S_i \geq 0$. For all the demand nodes i , including $N + 1$, $S_i \leq 1$. We also see, that for each arc (i, j) except $(N + 2, N + 1)$, $S_i = 0$ and $SI_j = 1$, hence, $S_i \leq SI_j$ constraint is satisfied for these arcs. For the arc $(N + 2, N + 1)$, $S_{N+2} = SI_{N+1} = 1$ and, therefore, the constraint is satisfied for all the arcs in the network. Thus, the solution is feasible.

The cost of the solution is $2N + 1$. The solution implies, that the argument of the safety stock function $\tau_i = 1$ for $i = 1, \dots, N + 1$ and $\tau_{N+2} = 0$. Therefore,

- $SS_i(\tau_i) = 1, i = 1, \dots, N$;
- $SS_{N+1}(\tau_{N+1}) = N + 1$;
- $SS_{N+2}(\tau_{N+2}) = 0$.

Then, each node $i \leq N$ contributes cost 1 and node $N + 1$ contributes $N + 1$ to the overall cost, while node $N + 2$ contributes 0 cost. Hence, the solution indeed gives the cost $2N + 1$.

Now, we argue that it is not optimal to hold stock in node $N + 2$. Suppose we allow positive stock in the node. Then, due to the shape of the safety stock function and per unit holding cost, the node contributes $2N + 2$ to the overall cost. Even if all the other nodes in the network have no safety stock, the total cost of this solution is $2N + 2$. This cost is the minimum possible cost if node $N + 2$ holds any stock.

The other possibility is that the node does not hold any stock. Since $T_{N+2} = 1$, S_{N+2} has to be at least 1. This implies $SI_{N+1} \geq 1$. Because $T_{N+1} = 1$ and $S_{N+1} \leq 1$, the argument of the safety stock function in node $N+1$ has to be at least 1. Therefore, $SS_{N+1}(\tau_{N+1}) \geq N + 1$. Again, we observe that independent of the safety stock in $N + 1$, nodes $1, \dots, N$ contribute the cost no greater than N . Therefore, cost of the stock in node $N + 1$ has to be the least possible, or $SS_{N+1}(\tau_{N+1}) = N + 1$. We see that if node $N + 2$ does not hold any stock, the overall optimal cost is no greater than $2N + 1$.

Comparing the two cases, we conclude that it is always optimal not to have stock in node $N + 2$. In this case, we also see that it is optimal to set τ_{N+1} equal to one for node $N + 1$.

Table 2.1 shows the intervals for the values of the objective function of problem P_i depending on the values for SI_{N+1} . We see again, that it is always better to have $SI_{N+1} = 1$, which allows node $N + 2$ to have zero stock and forces node $N + 1$ to have a safety stock cost equal to $N + 1$ units of cost.

□

We have already seen the following result in the proof of lemma 5. In the next

SI_{N+1}	Cost interval
0	$[2N + 2, 3N + 2]$
1	$[N + 1, 2N + 1]$
2	$[2N + 2, 3N + 2]$
\vdots	
$2 \leq k \leq N$	$[k(N + 1), k(N + 1) + N]$

Table 2.1: Intervals of the objective function value depending on SI_{N+1} .

lemma we state the result formally.

Lemma 6. *Suppose we have instance P_i of the safety stock problem \mathcal{P} with additional requirement $s_i \leq T_i$ for the demand nodes $i \in \mathbb{D}$. Then in an optimal solution, each node i in the network G' satisfies $S_i \leq 1$.*

Proof. From table 2.1 in the proof of lemma 5, we see that $SI_{N+1} = 1$ in an optimal solution. Node $N + 1$ constrains the outbound service times of all nondemand nodes from above. Since by construction of the problem $S_i \leq s_i = 1$ for all the demand nodes i , we conclude that in an optimal solution, $S_i \leq 1$ for all the nodes in the network. \square

Now, we are ready to show the result similar to the one in lemma 3.

Lemma 7. *Suppose we have instance P_i of the safety stock problem \mathcal{P} with additional requirement $s_i \leq T_i$ for the demand nodes $i \in \mathbb{D}$. Then, in an optimal solution, for every arc (i, j) , $i, j \leq N$ it is impossible to have values of the safety stock functions to be $SS_i(\tau) = 0$ and $SS_j(\tau) = 0$ at the same time.*

Proof. We have shown that in an optimal solution, all outbound service times are constrained from above by 1. Applying the same argument as in the proof of lemma 3,

we can conclude that for every arc $(i, j) : i, j = 0, \dots, N$ at least one of the nodes has to store inventory in an optimal solution. \square

Now, we can relate problem P_i and the initial vertex cover problem VC . From lemma 5, we see that the optimal cost of of problem P_i is the sum of

- the safety stock cost of $N + 1$ at node $N + 1$;
- cost $K \leq N$, where K is the number of nodes $1, \dots, N$ that hold stock in the optimal solution.

Therefore, an optimal solution is $K + N + 1$. Because the structure of the optimal solution in the nodes $1, \dots, N$ has the same properties as a vertex cover or is a v.c. assignment, by solving problem P_i and subtracting $N + 1$ from the solution we can find a solution of problem VC . We only need to show that the solution is an optimal solution of the VC problem. As in the unrestricted case, we need to show that the solution does not depend on the orientation of the graph G .

Lemma 8. *Suppose we have a directed graph G and safety stock problem P_i with condition (2.2) on $G' = G \cup \{N + 1\} \cup \{N + 2\}$ as described above. Then for every v.c. assignment of safety stock on G , there exists a feasible solution of the safety stock problem P_i with cost $N + 1 + K$. The vertex cover that corresponds to the v.c. assignment has cardinality K .*

Proof. As in the proof of lemma 4, we construct a solution. If node $i \in G$ then we consider two cases:

- Zero stock in i : $SI_i = 0, S_i = 1$.

- Nonzero stock in i : $SI_i = 1, S_i = 0$.

We also assign $SI_{N+1} = 1, S_{N+1} = 1, SI_{N+2} = 0$ and $S_{N+2} = 1$.

By similar arguments as in the proof of lemma 4, we can check that the solution is feasible. □

As a corollary from the lemma, we conclude that by solving problem P_i with constraint 2.2 optimally we obtain an optimal solution of the VC problem. Indeed, independent of the orientation of graph G , an optimal solution of problem P_i is a v.c. assignment on G . For every v.c. assignment, there exists a solution of problem P_i independent of the orientation of graph G . The cost of any v.c. assignment is equal to the cardinality of the corresponding vertex cover. Therefore, for any orientation of graph G , an optimal solution of P_i gives an optimal solution of the VC problem.

Now we can conclude that problem \mathcal{P} with requirement (2.2) is NP-hard.

2.2 Optimality conditions

In this section we discuss optimal solutions and methods of finding them. We start with the optimal solutions for the problem with $s_i \leq T_i, i \in \mathbb{D}$. For this case we identify a necessary condition for an optimal solution. We show that in an optimal solution, $S_i = s_i$ for each demand node i .

For the general case, we characterize the set of optimal solutions. Even though the problem in the general case can have multiple optimal solutions, we find a condition, such that the problem has at least one optimal solution that satisfies this condition.

Then, for all the solutions that satisfy this condition, we show how to construct paths in the network that identify these solutions. We will use the paths in Chapter 3 when we develop an algorithm for solving a general network safety stock problem. In particular, we will look at all the paths in the network and their associated solutions. Because the paths identify the solutions that satisfy the condition and there is at least one optimal solution that satisfies this condition, by enumerating all combinations of the paths, we can find an optimal solution.

To simplify notations, we call the layer of the nodes with zero indegree as components and use notation \mathbb{C} for the set of such nodes .

2.2.1 Restricted case: $s_i \leq T_i$

As in section 2.1.2, we consider a restricted case when $s_i \leq T_i$ for $i \in \mathbb{D}$. As we will see in this section this case is simpler than the general one. For this case we introduce optimality conditions, which, unlike in the general case, can hold simultaneously. We start with the condition on the outbound service times for the demand nodes.

Lemma 9. *Let j be a demand node with $s_j \leq T_j$. In an optimal solution,*

$$S_j = s_j. \tag{2.3}$$

Proof. First, we notice that the constraint polyhedron of problem \mathcal{P} is not empty. The solution for which all the service times are 0 is feasible. There is a trivial bound of 0 on the objective function, hence, the problem does not have an unbounded optimal solution. Therefore, there exists an optimal solution of the problem.

The cost function decreases when the outbound service times increase. Moreover, the only constraints on the outbound service times of a demand node j are $SI_j + T_j - S_j \geq 0$ and $0 \leq S_j \leq s_j$. The first constraint is satisfied, since $SI_j \geq 0$ and $S_j \leq s_j \leq T_j$. Therefore, in an optimal solution the outbound service time of the demand node equals to its maximum value, namely the guaranteed service time: $S_j = s_j$.

□

The next two lemmas provide characterizations of the inbound service times.

Lemma 10. *There always exists an optimal solution of the problem, such that all the inbound service times of the components are 0:*

$$SI_j = 0, j \in \mathbb{C}. \quad (2.4)$$

Proof. Suppose we have an optimal solution with cost z and service times S_i and $SI_i > 0$ for some component node i . We construct a new solution:

- $SI'_i = SI_i - \min\{SI_i, S_i\}$;
- $S'_i = S_i - \min\{SI_i, S_i\}$.

All the other service times remain the same.

First, we note that the new solution with S'_i and SI'_i , if feasible, has the same objective function value. Let us show that the solution is indeed feasible.

If $SI_i \leq S_i$, the new solution has $SI'_i = 0$, hence, $SI'_i \geq 0$ and is feasible. To check feasibility we need to check all the constraints relevant to node i . $SI'_i + T_i - S'_i =$

$0 + T_i - (S_i - SI_i) \geq 0$ because the initial solution is feasible. Also, $S'_i \leq S_i \leq SI_j$ for all $(i, j) \in \mathbb{A}$. Thus, we have constructed an optimal solution with $SI_i = 0$.

If $SI_i > S_i$, then $S'_i = 0$ and $SI'_i > 0$. However, we can decrease SI'_i to $SI''_i = 0$ and not violate any constraints, since the only constraint is $SI''_i + T_i - 0 = T_i \geq 0$. But the optimal cost of this solution is strictly less than z , because the safety stock function decreases when the inbound service time decreases. Thus, we have a contradiction of the original supposition that the initial solution is optimal.

□

Lemma 11. *There always exists an optimal solution of the problem \mathcal{P} with $s_i \leq T_i$, $i \in \mathbb{D}$, such that the inbound service time of each non component is equal to the maximal service time of its upstream nodes:*

$$SI_j = \max_i \{S_i : (i, j) \in \mathbb{A}\}, j \notin \mathbb{C}. \quad (2.5)$$

Proof. Suppose we know an optimal solution:

$$(S_1, \dots, S_N, SI_1, \dots, SI_N).$$

We first consider the non demand nodes. We assume that the statement of the problem is not true for this solution. That is, there exists node j such that

$$SI_j > \max_i \{S_i : (i, j) \in \mathbb{A}\} \text{ for } j \notin \mathbb{C}.$$

Then, we define

$$\delta_j = SI_j - \max\{S_i, (i, j) \in \mathbb{A}\}.$$

By the assumption, $\delta_j > 0$.

We define a new solution

$$SI'_j = SI_j - \delta_j,$$

$$S'_j = S_j - \min\{\delta_j, S_j\}.$$

Suppose $\delta_j \leq S_j$. Then the new solution is feasible and

$$\tau'_j = SI'_j + T_j - S'_j = SI_j - \delta_j + T_j - S_j + \delta_j = \tau_j.$$

This implies that the new solution has the same cost as the optimal solution. Therefore, we have found an optimal solution that satisfies the lemma.

Now, suppose $\delta_j > S_j$. Then the new solution is feasible. We can easily see it since in this solution $S'_j = 0$, which does not violate any constraint. We also find that $SI'_j \geq 0$ and $SI'_j \geq S_i$ for all $(i, j) \in \mathbb{A}$ by construction of δ_j . Constraint $SI'_j + T_j - S'_j = SI'_j + T_j \geq 0$ is satisfied as well. Therefore the solution is feasible.

The cost of the solution is strictly less than the cost of the original solution. Indeed,

$$\tau'_j = SI'_j + T_j - S'_j = SI_j + T_j - S_j - (\delta_j - S_j) < \tau_j.$$

Since the inventory at stage j decreases as τ_j decreases, the cost of the new solution is strictly less than the cost of the original solution. Thus, we have a contradiction as the original solution cannot be optimal.

Therefore, δ_j is always no greater than S_j , which proves the lemma for the non demand nodes.

Consider now demand node i . We have shown in lemma 9 that $S_i = s_i$ in an optimal solution. Therefore, we can not use the same argument for this node as for the non demand nodes, since we can not decrease S_i . However, since condition $SI_i + T_i - s_i \geq 0$ for all $SI_i \geq 0$ and the safety stock function increases when SI_i increases, we conclude that SI_i must be the smallest possible. Therefore, $SI_i = \max\{S_j : (j, i) \in \mathbb{A}\}$ in an optimal solution.

□

By lemmas 10 and 11, there might be multiple optimal solutions, at least one of which has the properties described in the lemmas. Moreover, from the proof of the lemma, we see that for every optimal solution that does not satisfy the property, there exists a solution that does. At the same time, lemma 9 states that all the optimal solutions satisfy (2.3).

From the proofs of the lemmas, we notice that all three conditions can hold together if the guaranteed outbound service times in the demand nodes do not exceed the lead times in the nodes: $s_i \leq T_i$. Indeed, the result of lemma 9 holds for any inbound service time SI_i , $i \in \mathbb{D}$ as discussed in the proof of the lemma. In the proof of lemma 11, we show that equation 2.5 holds for any S_i of a demand node i . Therefore, there always exists an optimal solution which satisfies all three equations (2.3), (2.4) and (2.5). As we will see in section 2.2.2, such result does not hold for the general case.

2.2.2 General case

For the general case, we prove a set of conditions that characterize some optimal solutions. The conditions are similar to the conditions stated in section 2.2.1. We start with the outbound service times for the demand nodes.

Lemma 12. *Let i be a demand node. Then there exists an optimal solution, such that*

$$S_i = s_i, i \in \mathbb{D}. \quad (2.6)$$

Proof. Since the safety stock function in each node i decreases when its outbound service time S_i increases, in an optimal solution, S_i has to be as big as allowed by the problem's constraints. The constraints on a demand node i are

- $S_i \leq SI_i + T_i$;
- $S_i \leq s_i$.

Therefore, in any optimal solution,

$$S_i = \min\{s_i, SI_i + T_i\}, i \in \mathbb{D}.$$

Suppose we have an optimal solution of the problem with cost z and service times S_i and SI_i in the demand node i , and suppose that $S_i = SI_i + T_i < s_i$. We will show that we can construct an optimal solution S'_i with $S'_i = s_i$.

Solution $S_i = SI_i + T_i < s_i$ is feasible and gives cost 0 for the safety stock function in node i . Moreover, increasing SI_i to $SI'_i = s_i - T_i$ and setting $S'_i = s_i$, with the

service times for the rest of the nodes being the same as before, gives a new solution with the same cost z and with cost 0 in demand node i . This increase in SI_i does not violate any constraints on SI'_i , as we can check that:

- $SI'_i \geq 0$;
- $SI'_i \geq S_j$ for all arcs $(i, j) \in \mathbb{A}$.

Therefore, the new solution is an optimal solution with $S'_i = s_i$.

□

We notice here, that while lemmas 9 and 12 state the same results (2.3) and (2.6), the statements of the two lemmas are different. The first one claims that all optimal solutions have property $S_i = s_i$ in the demand nodes. On the other hand, the second lemma only states that there exists a solution with this property. Intuitively, the difference in the statements can be explained as follows. If the guaranteed outbound service time of a demand node i is no greater than its lead time, $s_i \leq T_i$, the replenishment time of the node is at least as big as the guaranteed service time s_i . Therefore, to reduce the amount of inventory in the node, the node has to postpone the delivery to the end customer till the last moment. On the other hand, if the guaranteed outbound service time in node i is greater than its lead time, we do not necessarily need to push the delivery to the last moment. Since the replenishment time is smaller than the guaranteed service time s_i , we can have both a shorter outbound service time than the delivery guarantee, and zero inventory and zero cost at the demand node.

The next lemma provides the result similar to the results in lemmas 10 and 11. Because we used lemma 9 in the proof of lemma 11, we have to prove the result of the lemma again for the general case. Again, the statement of lemma 9 holds for any optimal solution, while similar result in the general case does not.

Lemma 13. *There exists an optimal solution of the problem \mathcal{P} , such that the inbound service time of each non component is equal to the maximal service time of its upstream nodes:*

$$SI_j = \max_i \{S_i : (i, j) \in \mathbb{A}\}, j \notin \mathbb{C}; \quad (2.7)$$

and the inbound service times of each component is 0:

$$SI_j = 0, j \in \mathbb{C}. \quad (2.8)$$

Proof. For the non demand nodes, the proof of the lemma is the same as in the proofs of lemmas 10 and 11. We only need to show equation 2.7 for the demand nodes. However, because in the general case, the outbound service time of a demand node can be less than its guaranteed service time s_i , we can use the same proof for the demand nodes as for the non demand nodes. Indeed, in the proof of lemma 11 we decrease both SI_j and S_j for nodes j that do not satisfy equation 2.7. Because we start with an optimal solution, the solution is feasible, i.e. $S_j \leq s_j$. The new solution $S'_j = S_j - \delta_j$ is also feasible, where δ_j is defined in lemma 11. Therefore, equation 2.7 holds for the demand nodes as well.

□

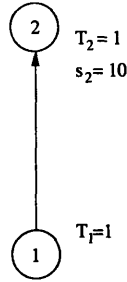


Figure 2-5: An example of a network for which conditions (2.6), (2.7) and (2.8) do not hold for any optimal solution.

Unlike in the restricted case from section 2.2.1, conditions (2.6), (2.7) and (2.8) might not all be true for any optimal solution of the problem. To show this fact, we consider a simple example where only two out of the three conditions can hold together.

Example 1. *Let us consider a graph presented on Figure 2-5. The graph has two nodes and one arc from node 1 to node 2. The lead time of each node is 1: $T_i = 1$, $i = 1, 2$. Node 2 is a demand node with the guaranteed outbound service time 10: $s_2 = 10$.*

Suppose $SI_1 = 0$ and $S_2 = 10$. Then, in order for the optimal solution to be feasible, $SI_2 + T_2 \geq S_2 = 10$. Hence, $SI_2 \geq 9$. On the other hand, for the solution to be feasible in node 1, $SI_1 + T_1 - S_1 \geq 0$, or $S_1 \leq 1$. Therefore, there are no solutions with $S_1 = SI_2$. This example shows that if conditions (2.6) and (2.8) are imposed, condition (2.7) might not necessarily hold.

If $SI_1 = 0$ and $S_1 = SI_2$, then $S_1 \leq 1$ implies $SI_2 \leq 1$. Therefore,

$$S_2 \leq SI_2 + T_2 \leq 2 < s_2 = 10.$$

From this example, we see that if conditions (2.7) and (2.8) hold together, condition (2.6) might not be feasible.

If $S_2 = 10$, then $SI_2 \geq 9$. If $SI_2 = S_1$, this implies $S_1 \geq 9$. To have a feasible solution,

$$SI_1 \geq S_1 - T_1 \geq 9 - 1 = 8.$$

Therefore, condition (2.8) need not hold if conditions (2.6) and (2.7) are imposed.

Example 1 shows that all three conditions might not hold together. However, there always is an optimal solution with two of the three conditions true. Indeed, in the proofs of the lemmas, to modify any optimal solution to satisfy conditions (2.8) and (2.7), we might need to decrease SI_i and S_i by δ_i , which does not affect other nodes. On the other hand, to satisfy (2.6) we might need to increase SI_i and S_i for the demand node i , which still keeps the solution feasible, but might violate conditions (2.7) or (2.8).

In what follows we look for the optimal solutions that satisfy conditions $SI_i = 0$, $i \in \mathbb{C}$ and $S_i = s_i$, $i \in \mathbb{D}$. Now, we proceed to describe the optimal solutions of this kind.

We observe two simple facts that follow from lemmas 12, 13 and conditions of problem \mathcal{P} .

Observation 1. Problem \mathcal{P} can be formulated in terms of outbound service times S_i only. For each solution (S_1, \dots, S_N) , the corresponding solution (SI_1, \dots, SI_N) can

be reconstructed using lemma 13:

$$\begin{aligned}
SI_i &= 0, \text{ for all components } i \in \mathbb{C}; \\
SI_i &= \max\{S_i - T_i, S_j : (j, i) \in \mathbb{A}\} \text{ for all non components } i \notin \mathbb{C}.
\end{aligned} \tag{2.9}$$

Observation 2. Problem \mathcal{P} can be formulated in terms of inbound service times SI_i only. For each solution (SI_1, \dots, SI_N) , the corresponding solution (S_1, \dots, S_N) can be reconstructed using lemmas 13 and 12:

$$\begin{aligned}
S_i &= \min\{s_i, SI_i + T_i\} \text{ for all the demand nodes } i \in \mathbb{D}; \\
S_i &= \min\{SI_i + T_i, SI_j : (i, j) \in \mathbb{A}\} \text{ for all the non demand nodes } i \notin \mathbb{D}.
\end{aligned} \tag{2.10}$$

The two observations help us characterize possible optimal solutions in terms of the input data. The next lemma states necessary conditions for the optimal solution for the problem. The objective of the lemma is to provide a more detailed description of the extreme points of the polyhedron described by the constraints of the problem \mathcal{P} . We will use lemma 14 to characterize the optimal solutions in the algorithm for the general networks in chapter 3.

We use Observation 2 in the next development of the necessary conditions. We only consider the optimal inbound service times SI_i for each node i in the network. Note, that we search for the solutions with $SI_i = 0$ for the components $i \in \mathbb{C}$. The next lemma determines all of the candidate values for SI_i in an optimal solution.

Lemma 14. *Suppose we have an instance of problem \mathcal{P} and its optimal solution. Then there always exists an optimal solution for which the inbound service time of*

node i is one of these values:

- If $i \in \mathbb{C}$, then

$$SI_i = 0;$$

- If $i \in \mathbb{D}$, then

$$SI_i = \max\{0, s_i - T_i\}; \text{ or}$$

$$SI_i = \min\{SI_j + T_j, SI_m\} \text{ for some } j : (j, i) \in \mathbb{A} \text{ and all } m \neq i : (j, m) \in$$

$$\mathbb{A};$$

- If $i \notin \mathbb{D}$ and $i \notin \mathbb{C}$, then

$$SI_i = 0; \text{ or}$$

$$SI_i = SI_k - T_i \text{ for some } k : (i, k) \in \mathbb{A}; \text{ or}$$

$$SI_i = \min\{SI_j + T_j, SI_m\} \text{ for some } j : (j, i) \in \mathbb{A} \text{ and all } m \neq i : (j, m) \in$$

$$\mathbb{A}.$$

Proof. We have already proven, that for any optimal solution we can construct another optimal solution with $SI_i = 0$ for the component nodes. In what follows, we consider the non component nodes.

Suppose we have an optimal solution of problem \mathcal{P} . We consider the inbound service time SI_i of node i . There are only three conditions imposed by the constraints of the problem that are relevant for SI_i :

$$SI_i \geq S_j, (j, i) \in \mathbb{A}; \tag{2.11}$$

$$SI_i \geq S_i - T_i \quad (2.12)$$

$$SI_i \geq 0 \quad (2.13)$$

When these constraints are satisfied, the safety stock function in node i increases if SI_i increases. Therefore, in an optimal solution SI_i has to be the smallest possible value implied by the constraints:

$$SI_i = \max\{0, S_i - T_i, S_j : (j, i) \in \mathbb{A}\}.$$

We consider the three cases for the minimum possible SI_i :

1. $SI_i = 0$;
2. $SI_i = S_i - T_i$;
3. $SI_i = S_j$ if $(j, i) \in \mathbb{A}$.

We start with the case $SI_i = 0$. This occurs when $S_i - T_i \leq 0$ and $S_j = 0$ for all $(j, i) \in \mathbb{A}$. Therefore, $SI_i = 0$ is a possibility for an optimal solution.

Suppose

$$\max\{0, S_i - T_i, S_j; (j, i) \in \mathbb{A}\} = S_i - T_i.$$

Then $SI_i = S_i - T_i$. In this case, node i does not have any inventory. We can increase both S_i and SI_i by the same amount till S_i reaches its biggest possible value and not affect the overall cost. If i is a demand node, the biggest possible value of S_i is s_i . Therefore, letting $S_i = s_i$ implies $SI_i = s_i - T_i$ in this case. Since by increasing S_i to s_i does not change the overall cost, the new solution is still optimal.

If i is not a demand node, the biggest possible value of S_i is $SI_k = \min\{SI_r, (i, r) \in \mathbb{A}\}$. Then, increasing both S_i and SI_i by the same amount till $S_i = SI_k$ preserves the overall optimal cost and gives a possible solution for the inbound service time of node i : $SI_i = SI_k - T_i$ for some $(i, k) \in \mathbb{A}$.

Suppose now,

$$\max\{0, S_i - T_i, S_j; (j, i) \in \mathbb{A}\} = S_j, (j, i) \in \mathbb{A}.$$

Because $SI_i = S_j$, we can express the sum of the safety stock functions in nodes i and j to be a function of only one variable SI_i given that we fix the rest of the variables.

That is, consider the sum function

$$SS_i(SI_i + T_i - S_i) + SS_j(SI_j + T_j - S_j)$$

with the constraint $SI_i = S_j$. Since this function is concave in SI_i , it achieves its minimum value on the boundary of the feasible set

$$\max\{0, S_i - T_i\} \leq SI_i \leq \min\{SI_j + T_j, SI_m; (j, i), (j, m) \in \mathbb{A}\},$$

where the right hand side of the condition arises from the constraint on S_j . Therefore, the inbound service time SI_i can take the value of $\min\{SI_j + T_j, SI_m; (j, i), (j, m) \in \mathbb{A}\}$.

After considering all possible optimal values of SI_i , we conclude that we can always modify the solution such that SI_i takes one of the values specified by the lemma. \square

In the next section, we extend the properties developed in lemma 14. This lemma shows how to equate the inbound service times of two nodes that are either connected by an arc or both connected to an upstream node. In the next section, we establish relationships between the service times of the nodes that form a path in the network. Lemma 14 is a building block in creating such paths.

2.2.3 Critical paths

Lemma 14 is a building block for the next development of the optimality conditions. In this section we demonstrate that all solutions that satisfy the properties listed in lemma 14 correspond to solutions that can be constructed using paths in the network. We will use this property in Chapter 3 when we search for the optimal solutions. We will look at all the possible paths, and thus at all the solutions that satisfy properties in lemma 14. Since there is an optimal solution that also satisfies the properties, we can find a solution of the safety stock problem.

For the purpose of the next lemmas we introduce some new notation. Let us consider two nodes i and j in the network and an undirected path between them:

$$P_{ij} = (i = i_0, i_1, \dots, i_p = j).$$

Even though there can be several paths between the two nodes, we only consider one path for now.

For each path P_{ij} , we define the numerical value R_{ij} , which is computed by the following procedure. Consider node c . We say that we "go upstream" if the next

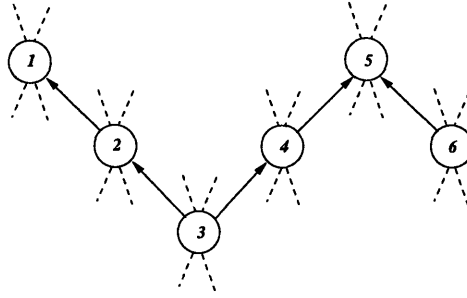


Figure 2-6: A path between nodes 1 and 6.

considered node is $q : (q, c) \in \mathbb{A}$. We also say, that we "go downstream" if the next considered node is $q : (c, q) \in \mathbb{A}$. Then we compute R_{ij} :

1. Start in node i with $R_{ij} = 0$,
2. Go along the path till node j is reached doing one of the following:
 - (a) Going upstream, add the lead time of the destination;
 - (b) Going downstream, subtract the lead time of the origin.

Let us consider an example of R_{ij} computation.

Example 2. Consider a general network and a path between nodes 1 and 6 shown on Figure 2-6. We can compute R_{16} :

$$R_{16} = T_2 + T_3 - T_3 - T_4 + T_6 = T_2 - T_4 + T_6.$$

From the definition, an important property of R_{ij} for the path P_{ij} is that for any node $k \in P_{ij}$

$$R_{ij} = R_{ik} + R_{kj}.$$

We need the definition of R_{ij} for each path in the formulation of the next lemma. We notice here that there might be multiple paths between any two nodes. However, to simplify the presentation, we will not distinguish between the different paths from i to j in this section.

We next consider the nodes and service times along the path. To make the process more clear we first specify all the possible ways we can order two or three nodes in the path. In this discussion, the start node is always node i or i_0 as the initial node of the path. The end node is node j or we also call it i_p as the p th node in the path. Any intermediate node in the path is called i_k as the k th node in the path. Table 2.2 specifies all possible combinations of the nodes and arcs which we will call configurations and number them as shown in the table. The table also refers to the figures that correspond to the configurations.

Number	Node	Adjacent arcs in path	Figure
1	$i = i_0$	$(i, i_1) \in \mathbb{A}$	2 - 7
2	$i = i_0$	$(i_1, i) \in \mathbb{A}$	2 - 8
3	i_k	$(i_{k-1}, i_k) \in \mathbb{A}, (i_k, i_{k+1}) \in \mathbb{A}$	2 - 9
4	i_k	$(i_k, i_{k-1}) \in \mathbb{A}, (i_{k+1}, i_k) \in \mathbb{A}$	2 - 10
5	i_k	$(i_{k-1}, i_k) \in \mathbb{A}, (i_{k+1}, i_k) \in \mathbb{A}$	2 - 11
6	i_k	$(i_k, i_{k-1}) \in \mathbb{A}, (i_k, i_{k+1}) \in \mathbb{A}$	2 - 12
7	$j = i_p$	$(i_{p-1}, i_p) \in \mathbb{A}$	2 - 13, 2 - 14
8	$j = i_p$	$(i_p, i_{p-1}) \in \mathbb{A}$	2 - 15

Table 2.2: Configurations of nodes and adjacent arcs in the path.



Figure 2-7: A path between nodes i and j with arc $(i, i_1) \in \mathbb{A}$ in the path.



Figure 2-8: A path between nodes i and j with arc $(i_1, i) \in \mathbb{A}$ in the path.

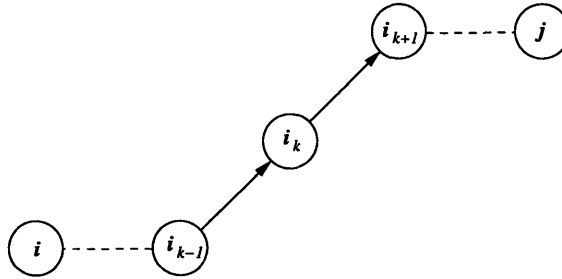


Figure 2-9: A path between nodes i and j with arcs $(i_{k-1}, i_k) \in \mathbb{A}$ and $(i_k, i_{k+1}) \in \mathbb{A}$ in the path.

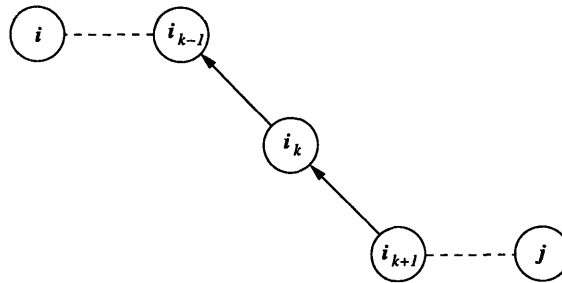


Figure 2-10: A path between nodes i and j with arcs $(i_{k-1}, i_k) \in \mathbb{A}$ and $(i_k, i_{k+1}) \in \mathbb{A}$ in the path.

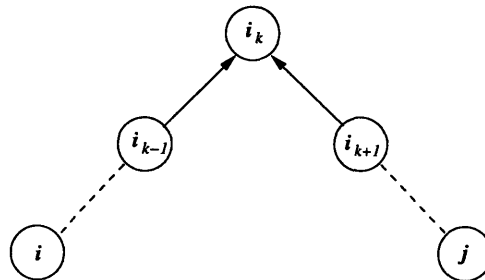


Figure 2-11: A path with arcs $(i_{k-1}, i_k) \in \mathbb{A}$ and $(i_{k+1}, i_k) \in \mathbb{A}$.

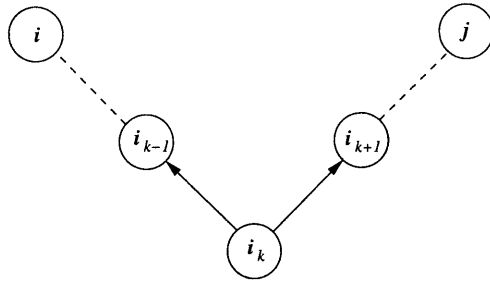


Figure 2-12: A path between nodes i and j with arcs $(i_k, i_{k-1}) \in \mathbb{A}$ and $(i_k, i_{k+1}) \in \mathbb{A}$ in the path.

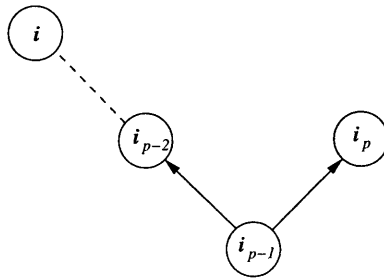


Figure 2-13: A path between nodes i and $i_p = j$ with arcs $(i_{p-1}, i_{p-2}) \in \mathbb{A}$ and $(i_{p-1}, i_p) \in \mathbb{A}$ in the path.

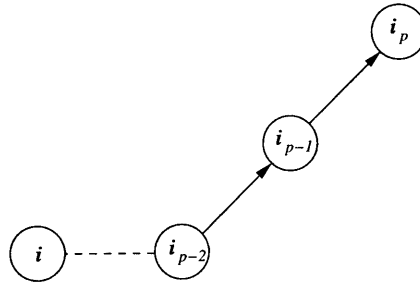


Figure 2-14: A path between nodes i and $i_p = j$ with arcs $(i_{p-2}, i_{p-1}) \in \mathbb{A}$ and $(i_{p-1}, i_p) \in \mathbb{A}$ in the path.

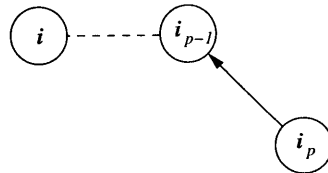


Figure 2-15: A path between nodes i and $i_p = j$ with arc $(i_p, i_{p-1}) \in \mathbb{A}$ in the path.

Lemma 15. *Suppose we have an instance of problem \mathcal{P} and its optimal solution satisfying the properties stated in lemma 14. Then for each node i , we can construct a path P_{ij} , such that*

$$SI_i = SI_j + R_{ij}$$

and such that

- $i = j$ and $P_{ij} = \{i\}$, or
- $(i, j) \in \mathbb{A}$ and $P_{ij} = \{i, j\}$, or
- $(j, i) \in \mathbb{A}$ and $P_{ij} = \{i, j\}$, or
- $(i_1, i) \in \mathbb{A}$ and $(i_1, j) \in \mathbb{A}$ for some node i_1 and $P_{ij} = \{i, i_1, j\}$.

Proof. Suppose we have an optimal solution that satisfies the properties listed in lemma 14 for each node i of the network.

We start with the possibility that $SI_i = 0$ for a non demand node i . This solution corresponds to the path of zero length P_{ii} with correspondent $R_{ii} = 0$.

If i is a demand node, then a possibility for such a node is $SI_i = \max\{0, s_i - T_i\}$, that corresponds to the zero length path P_{ii} with correspondent $R_{ii} = 0$.

Suppose now, that $SI_i = SI_k - T_i$ for some $(i, k) \in \mathbb{A}$. Then $j = k$ as shown on Figure 2-7. Therefore, $SI_i = R_{ij} + SI_j$, since $R_{ij} = -T_i$, and we have shown the lemma in case of a path with one arc and two nodes with the starting node upstream of the other node: $P_{ij} = \{i, j\}$, $(i, j) \in \mathbb{A}$.

Now consider the case when $SI_i = SI_k + T_k$, $(k, i) \in \mathbb{A}$. Then we can assign $j = k$ as in Figure 2-8 for $i_1 = j$ and equate $SI_i = SI_j + T_j = R_{ij} + SI_j$, since $R_{ij} = T_j$. In

this case the path is $P_{ij} = \{i, j\}$ with $(j, i) \in \mathbb{A}$.

Suppose $SI_i = SI_m$ for some node m such that there exists node $i_1 : (i_1, i) \in \mathbb{A}$ and $(i_1, m) \in \mathbb{A}$. Then, we assume $j = m$ (Figure 2-12 with $i_k = i_1$). In this case by lemma 14, $SI_i = S_{i_1} = SI_j$, but SI_{i_1} can not be equated with the service times of nodes i, i_1, j . However, we notice that in this case $R_{ij} = T_{i_1} - T_{i_1} = 0$ and we see that $SI_i = SI_j + R_{ij}$ and the lemma holds true. The path in this case is $P_{ij} = \{i, i_1, j\}$.

□

Corollary 2. *Suppose an arc $(i, j) \in \mathbb{A}$ is on a path defined in lemma 15. Then $S_i = SI_j$ for the solution defined by the path.*

Proof. The corollary is follows from the proofs of lemmas 14 and 15.

The goal of the path development is to have an efficient way to find the candidate values of service times for solving the safety stock problem. Lemma 15 lists three possibilities for relating SI_i to SI_j . However, more than one possible path can correspond to the same inbound service time. For example, $SI_i = SI_i$ with the path P_{ii} is always a possibility. However, such path is not useful in determining an optimal solution, and, therefore, we will not consider such paths.

Another example is that $SI_i = 0$ corresponding to the path P_{ii} . Consider now a path $P_{ij} = \{i, i_1, j\}$ such that $(i_1, i) \in \mathbb{A}$ and $(i_1, j) \in \mathbb{A}$. For this path, $SI_i = SI_j + R_{ij} = SI_j$ by definition of R_{ij} . If $SI_j = 0$, then $SI_i = 0$. Therefore, path $P_{ij} = \{i, i_1, j\}$ gives the same value to SI_i as the path $P_{ii} = \{i\}$, namely $SI_i = 0$. Thus, path P_{ij} does not provide any new information about the candidate values for the inbound service time.

In what follows, we will build longer and longer paths. We stop a path at node k if $SI_k = 0$ for a non demand node k or $SI_k = \max\{0, s_k - T_k\}$ for a demand node k . Otherwise, we always try to search for a longer path as we now describe.

We can extend the result of lemma 15 to allow longer paths. In the lemma we have shown that for some j specified in the lemma, $SI_i = R_{ij} + SI_j$. We can apply the lemma to the inbound service time of node j . In particular, by lemma 15, we can construct a path with zero, one or two arcs from node j and increase the number of nodes in the path from node i . Let us also rename the nodes such that the last node in the current path from node i is called j and the nodes of the path called i_1, i_2, \dots

We can repeat the augmentation of the path every time the path is not stopped, where we stop the path if the last $SI_j = 0$ for non demand nodes or $SI_j = \max\{0, s_j - T_j\}$ if $j \in \mathbb{D}$. Every time we augment the path, we call the last node j and all the nodes in the path starting from i as $i = i_0, i_1, \dots, i_p = j$. We state the procedure formally in Algorithm 1. We have not shown that the procedure is finite; we will establish this result later in this section. We also note that the procedure stated in Algorithm 1 identifies a path given an optimal solution that satisfies lemma 14.

So far, we have established that we can create a path that provides a value for the inbound service time SI_i . Since we do not know a priori what path corresponds to the optimal solution, we have to try all possible paths from node i to check all possible values. As we have already discussed, we stop the path if $SI_j = 0$ or $SI_j = \max\{0, s_j - T_j\}$ if j is a demand node. In what follows, we say that we stop the path if SI_j takes some particular value, independent of the path, for example 0. In fact, we can always stop the path at any node, since such a path creates a possible value

Algorithm 1 Path augmentation procedure

$P[]$ – array of the nodes in the path

AugmentPath(i)

procedure AugmentPath(k)

```
1: if  $k \in \mathbb{D}$  and  $SI_k = \max\{0, s_k - T_k\}$  then  
2:   STOP  
3: end if  
4: if  $k \notin \mathbb{D}$  and  $SI_k = 0$  then  
5:   STOP  
6: end if  
7: if  $S_k = SI_r, (k, r) \in \mathbb{A}$  then  
8:    $SI_i \leftarrow SI_i - T_k$   
9:    $P \leftarrow P \cup \{r\}$   
10:  AugmentPath(r)  
11: end if  
12: if  $SI_k = S_r, (r, k) \in \mathbb{A}$  then  
13:    $SI_i \leftarrow SI_i + T_r$   
14:    $P \leftarrow P \cup \{r\}$   
15:   if  $S_r = SI_m, (r, m) \in \mathbb{A}, m \neq k$  then  
16:      $SI_i \leftarrow SI_i - T_r$   
17:      $P \leftarrow P \cup \{m\}$   
18:      $r \leftarrow m$   
19:   end if  
20:   AugmentPath(r)  
21: end if  
end procedure
```

for the service time. We will see, that SI_j can be set according to some other path. In this case we can follow that other path, or we can just say that the path is stopped because SI_j is set to some particular value. However, now we can not conclude what is the longest path we need to consider to go through all the possible paths from node i . For example, the path can conceivably circle. To eliminate this kind of path behavior, we need to establish several characteristics of the paths.

We start with the lemma, that shows how to equate the inbound service times along the path. The lemma follows from the construction of the path and definition of R_{ij} .

Lemma 16. *Suppose we have an instance of problem \mathcal{P} and its optimal solution satisfying the properties stated in lemma 14. Suppose path P_{ij} corresponds to SI_i and is created by Algorithm 1. Then for every node $i_r, 0 \leq r \leq p$ along the path that is not connected to its neighbors as in configuration 6 of Table 2.2*

$$SI_{i_r} = R_{i_r j} + SI_j \quad (2.14)$$

Proof. We have proved the statement of the lemma in case of the paths defined in lemma 15. By the construction of the longer paths, we apply lemma 15 to the last node of the current path. Suppose, the last node is i_k and for each node $i_r, 0 \leq r \leq k$ in the path, $SI_{i_r} = R_{i_r i_k} + SI_{i_k}$. We apply lemma 15 to node i_k . If $SI_{i_k} = 0$ or $SI_{i_k} = \max\{0, s_{i_k} - T_{i_k}\}$ for $i_k \in \mathbb{D}$ by lemma 14, then i_k is the last node in the path: $k = p$. Then by assumption, equation (2.14) holds. We note that in this case, all the inbound service times along the path that do not connect to the neighbors as in

configuration 6, have to be non negative. If it is not the case, $i_k = j$ can not happen.

Otherwise, there are two possibilities for the node i_{k+1} :

1. i_{k+1} is downstream of node i_k ;
2. i_{k+1} is upstream of node i_k .

Suppose i_{k+1} is downstream of i_k . By induction, $SI_{i_r} = SI_{i_k} + R_{i_r i_k}$. Also, $SI_{i_k} = S_{i_k} - T_{i_k}$ and $S_{i_k} = SI_{i_{k+1}}$ by corollary 2. Therefore, for any node i_r , $0 \leq r \leq k$ in the path

$$SI_{i_r} = R_{i_r i_k} - T_{i_k} + SI_{i_{k+1}} = R_{i_r i_{k+1}} + SI_{i_{k+1}}$$

and we have shown that the lemma holds in this case.

If i_{k+1} is upstream of i_k , then again we can only equate $SI_{i_{k+1}}$ with other inbound service times in the path as specified in the conditions of the lemma if $SI_{i_{k+1}} + T_{i_{k+1}} \leq SI_m$ for all $(i_{k+1}, m) \in \mathbb{A}$. Then $SI_{i_k} = S_{i_{k+1}} = SI_{i_{k+1}} + T_{i_{k+1}}$ from corollary 2 and by definition of $R_{i_r i_{k+1}}$, $0 \leq r \leq k$ the lemma holds:

$$SI_{i_r} = R_{i_r i_k} + T_{i_{k+1}} + SI_{i_{k+1}} = R_{i_r i_{k+1}} + SI_{i_{k+1}}.$$

As we have discussed before, if there exists node m , such that $SI_{i_{k+1}} + T_{i_{k+1}} > SI_m$, $(i_{k+1}, m) \in \mathbb{A}$. Then we can assign $i_{k+2} = m$ (Figure 2-12). If this happens, node i_{k+1} has neighbors as in configuration 6. Therefore, we can not equate $SI_{i_{k+1}}$ with SI_{i_k} , but $SI_{i_k} = S_{i_k} = SI_{i_{k+2}}$. Since $R_{i_k i_{k+2}} = T_{i_k} - T_{i_k} = 0$,

$$SI_{i_r} = R_{i_r i_k} + SI_{i_k} = R_{i_r i_{k+2}} + SI_{i_{k+2}}.$$

This proves the lemma if $p = k + 2$.

□

The lemma shows that if SI_i is set according to some path in an optimal solution, all the inbound service times of the nodes along the path, other than nodes such as i_k presented on Figure 2-12, have to be set according to the last inbound service time as well.

We next state corollaries of the lemma. We show how to describe all the service times of the nodes in the path.

Corollary 3. *Suppose SI_i is set according to the path P_{ij} , then for any two nodes in the path i_k and i_r , $k \leq r$, that do not connect to their neighbors as in configuration 6:*

$$SI_{i_k} = R_{i_k i_r} + SI_{i_r}.$$

Proof. By lemma 16

$$SI_{i_k} = R_{i_k j} + SI_j,$$

$$SI_{i_r} = R_{i_r j} + SI_j.$$

By definition of $R_{i_k j}$ and $R_{i_r j}$

$$R_{i_k i_r} + R_{i_r j} = R_{i_k j}.$$

Therefore, $SI_{i_k} = R_{i_k i_r} + R_{i_r j} + SI_j = R_{i_k i_r} + SI_{i_r}$.

□

Corollary 3 shows that if we have found the path that corresponds to an optimal SI_i , then every inbound service time along the path can be equated with any other inbound service time in the path, except when the nodes connect to their neighbors as in configuration 6.

We next express the outbound service times along the path. This corollary is similar to corollary 2 but stated for longer paths.

Corollary 4. *Suppose in an optimal solution, node i has inbound service time SI_i that corresponds to some path $P_{i,j} = (i = i_0, i_1, \dots, i_p = j)$. Then if an arc (r, k) is in the path, $S_r = SI_k$.*

Proof. The corollary follows from the construction of the path in Algorithm 1 and corollary 2.

□

Corollary 5. *Suppose we have an instance of problem \mathcal{P} and its optimal solution satisfying the properties stated in lemma 14. Then for each node i there exists a path $P_{i,j}$ such that for every node $i_k, 0 \leq k \leq p$ along the path that is not connected to its neighbors as in configurations 2 and 5 of Table 2.2*

$$S_r = R_{k_j} + SI_j, (r, k) \in \mathbb{A} \text{ and } (r, k) \text{ is in the path .}$$

Proof. From lemma 16, $SI_k = R_{k_j} + SI_j$ for some path $P_{i,j}$. From corollary 4, for any arc (r, k) in the path, $S_r = SI_k$. Therefore, the equation stated in the corollary holds if node r has at least one arc $(r, k) \in \mathbb{A}$ in the path.

Suppose all the arcs connected to node k in the path are upstream of node k as shown on Figure 2-11. Suppose the path goes through nodes $k - 1, k, k + 1$. Then since $(k - 1, k) \in \mathbb{A}$ and $(k + 1, k) \in \mathbb{A}$, $S_{k-1} = SI_k = S_{k+1}$ by corollary 4. Therefore, since $SI_k \geq S_k + T_k$, i.e., it is not equated to the other service times in the path, S_k can not be expressed from the path. Hence, if node k is connected to its neighbors as in configurations 5, the corollary does not hold.

Similarly, if node k is the first node in the path, i.e., $k = i$, and $(i, i) \in \mathbb{A}$ is in the path, as on Figure 2-8, i has neighbors as in configuration 2. Then, $SI_i = S_{i_1}$ by lemma 4, but S_i can not be equated with any service time in the path.

□

From lemma 16 and corollary 5 we can conclude that along the path, we set at least one of the service times of any node in the path according to the inbound service time of the last node in the path. If the first arc in the path is oriented in the direction of the path (Figure 2-7), then both SI_i and S_i are known since $SI_i = SI_j + R_{ij}$ and $SI_i + T_i - S_i = 0$ as shown in the proof of lemma 14. If the first arc is oriented against the direction of the path, then only SI_i is known from the path. Indeed, we can only equate $SI_i = S_{i_1}$, but there is no node downstream of node i to set S_i .

If the path passes node i_k as shown on Figure 2-9, then $SI_{i_k} + T_{i_k} - S_{i_k} = 0$ by lemma 16. Because i_k does not connect to its neighbors as in configuration 6, we know that $SI_{i_k} = R_{i_k j} + SI_j$. Therefore, we can set both service times in the node according to the inbound service time of the last node.

Suppose the path passes node i_k as shown on Figure 2-10. Then, $S_{i_{k+1}} = SI_{i_k}$,

node	S of node	SI of node
1	unknown	$T_2 - T_4 + T_6$
2	$T_2 - T_4 + T_6$	$-T_4 + T_6$
3	$-T_4 + T_6$	unknown
4	T_6	$-T_4 + T_6$
5	unknown	T_6
6	T_6	0

Table 2.3: Service times for the path on Figure 2-6.

$SI_{i_k} + T_{i_k} - S_{i_k} = 0$ and we conclude that both SI_{i_k} and S_{i_k} can be set according to the inbound service time of the last node of the path.

Suppose the path passes node i_k as shown on Figure 2-12. Then only S_{i_k} can be set according to the last inbound service time as shown in corollary 5.

Suppose the path passes node i_k as shown on Figure 2-11. Then only SI_{i_k} can be set according to the last inbound service times as shown in lemma 16.

Let us consider an example of a path in a network and associated values of service times.

Example 3. *In this example, we again consider a general network shown on Figure 2-6. Suppose no service time is known and we want to explore all the possibilities for the inbound service time SI_1 of node 1. Then we can choose a path which starts in node 1 and ends in node 6 to consider one of the possibilities (see Figure 2-6). The values of service times are presented in table 2.3. We observe, that path 1, . . . , 6 determines what the service times are, but they may not be feasible.*

The discussion above has shown that there exists a solution, such that we can associate a path with every inbound service time. Unfortunately, the number of paths in the graph is exponentially large and we do not know a priori which path

corresponds to a particular inbound service time. Therefore, if we try to solve the problem, we need to try all the possible paths from each node of the network. In what follows we discuss all the possible paths and how we can set service times along the paths.

In the remainder of the section our goal is to show that it is enough to consider paths that are no longer than $2N$ nodes. To do that, we discuss the service times of a node that is included in a path. We show, that a path can not set a service time that has been set before. Here, we consider two possibilities of the ways the service time can be set in the first place. First, the service time is set by some other path. Second, the service time is set by the same path. Both possibilities lead us to the conclusion, that if the path tries to set already set service time, the has to stop.

We first analyze the possibility, that the node in path P_{ij} already has some service times set by some other path. Let us consider node i_k in path P_{ij} . Then, we analyze four possibilities:

1. SI_{i_k} and S_{i_k} are set before;
2. SI_{i_k} is set before, S_{i_k} is not set;
3. S_{i_k} is set before, SI_{i_k} is not set;
4. no service time is set in node i_k .

We consider each possibility separately.

SI_{i_k} and S_{i_k} are set before.

We first look at the service times of node i_k . By lemma 16 and corollary 5, there

always exists a solution, such that at least one of the service times of the nodes in the path can be set according to the last inbound service time. Therefore, we assume that SI_{i_k} is set according to some path $P_{i^1j^1}$ and S_{i_k} is set according to some path $P_{i^2j^2}$. The two paths can be the same path, but for greater generality we assume that the two paths might be different.

Suppose a path that starts in node i contains node i_k . We have already discussed, that the path P_{ij} sets at least one of the service times of each node in the path. Therefore, the path will try to set $SI_{i_k} = R_{i_kj} + SI_j$ if i_k does not connect to its neighbors as in configuration 6. Then if $R_{i_kj^1} + SI_{j^1} \neq R_{i_kj} + SI_j$, such setting is not valid. To avoid conflicts between the paths, we can assume that the parts of the paths (i_k, \dots, j) and (i_k, \dots, j^1) are the same. Since SI_{i_k} is set according to the path $P_{i^1j^1}$, we can just stop the path P_{ij} in node i_k . Therefore all the service times along the path P_{ii_k} are set: $SI_{i_r} = SI_{i_k} + R_{i_r i_k}, 0 \leq r \leq k$. In other words, we do not need to continue the path, but treat node i_k as the last node of the path.

Suppose i_k connects to its neighbors as in configuration 6, then path P_{ij} can only set S_{i_k} according to SI_j :

$$S_{i_k} = SI_{i_{k+1}} = R_{i_{k+1}} + SI_j.$$

However, if this value conflicts with the previously set value, then path P_{ij} is not valid. Therefore, we have to stop the path in node i_{k-1} in the sense that $SI_{i_{k-1}} = S_{i_k}$ and all the inbound service times $SI_{i_r} = R_{i_r i_{k-1}} + SI_{i_{k-1}}$ if the path can set SI_{i_r} .

In conclusion for this case, if SI_{i_k} and S_{i_k} are set, then any other path that goes

through node i_k can be stopped in i_k or i_{k-1} . If $(i_{k-1}, i_k) \in \mathbb{A}$ is in the path, then i_k is the last node of the path. If $(i_k, i_{k-1}) \in \mathbb{A}$, then i_{k-1} is the last node in the path.

SI_{i_k} is set, but S_{i_k} is not set.

Suppose SI_{i_k} is set by a path $P_{i_1 j^1}$. We notice here, that this case is only possible if i_k has neighbors as in configuration 5 in path P_{ij} or $i_k = i_1$ and $(i_1, i) \in \mathbb{A}$ as proved in corollary 5.

We first consider the case when both i_{k-1} and i_{k+1} are upstream of node i_k . Then i_k does not have neighbors as in configuration 6. Therefore, $SI_{i_k} = R_{i_k j} + SI_j$ according to the path P_{ij} . If (i_k, \dots, j) is not a sub path in (i_k, \dots, j^1) the value of the inbound service time set by path P_{ij} in node i_k might be in conflict with the one set by path $P_{i_1 j^1}$. Therefore, we can assume that path P_{ij} stops in $i_k = j$. In this case we set $SI_{i_r} = R_{i_r i_k} + SI_{i_k}$, $0 \leq r \leq k$ if r does not have neighbors as in configuration 6.

If both i_{k-1} and i_{k+1} are downstream of i_k , then i_k has neighbors as in configuration 6. Then P_{ij} will only try to set $S_{i_k} = SI_{i_{k-1}}$. Since S_{i_k} is not set by $P_{i_1 j^1}$, path P_{ij} can do so, as long as $SI_{i_k} + T_{i_k} - S_{i_k} \geq 0$.

S_{i_k} is set, but SI_{i_k} is not set.

We assume that S_{i_k} is set by a path $P_{i_2 j^2}$. We also notice, that this case only happens when i_k has neighbors as in configuration 6 as shown in lemma 16.

First, we assume that i_{k+1} is downstream of i_k . Then, i_k does not have neighbors as in configuration 5. Therefore, path P_{ij} will try to set S_{i_k} . However, $P_{i_2 j^2}$ has already set S_{i_k} . If i_{k-1} is upstream of node i_k , we can treat node i_k as the last node in the path: $SI_{i_k} = \max\{0, S_{i_k} - T_{i_k}\}$. Alternatively, we can say that i_{k-1} is the last node in the path: $SI_{i_{k-1}} = 0$.

Suppose i_{k-1} is downstream of node i_k , then i_k does not have neighbors as in configuration 5. Therefore, path P_{ij} will try to set S_{i_k} . Hence, the path is stopped in i_{k-1} and $SI_{i_{k-1}} = S_{i_k}$.

If both i_{k-1} and i_{k+1} are upstream of node i_k , then path P_{ij} will only try to set $SI_{i_k} = SI_j + R_{i_k j}$. Since SI_{i_k} is not set by $P_{i^2 j^2}$, path P_{ij} can set SI_{i_k} as long as $SI_{i_k} + T_{i_k} - S_{i_k} \geq 0$.

Neither SI_{i_k} nor S_{i_k} is set.

If path P_{ij} goes through the node i_k , then the path can set SI_{i_k} or S_{i_k} or both as described in lemmas 16 and 5. Since no other path has set the the service times before, P_{ij} can set them.

This concludes the discussion of a path intersecting some other path. Now we consider the cases when the path intersects itself.

We first prove that there is no need to consider path that starts and ends in the same node. Indeed, consider a path P_{ii} . Then $SI_i = R_{ii} + SI_i$. Therefore, $R_{ii} = 0$ and this path does not provide any new possibility for SI_i . Therefore, we will not consider the paths that start and end in the same node.

Now, suppose path P_{ij} includes node i_k such that it sets SI_{i_k} . These situations are presented on Figures 2-9, 2-10 and 2-11. Suppose the path intersects itself in the node: $P_{ij} = (i, \dots, i_k, \dots, i_r, \dots, j)$, $i_k = i_r$, and the path also tries to set SI_{i_r} as shown on the same figures but for i_r . Then, by corollary 3, $SI_{i_r} = SI_{i_k} + R_{i_k i_r}$. However, generally, $R_{i_k i_r}$ is not equal to zero. Therefore, such intersection is not valid. But if $R_{i_k i_r} = 0$, the cycle does not create any new value of SI_i , hence, we do not consider the intersection.

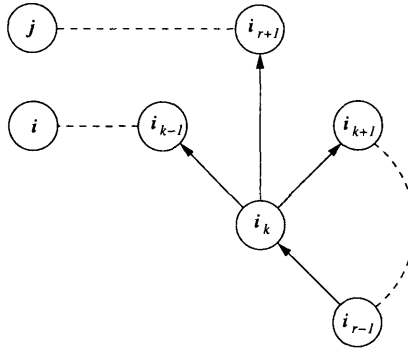


Figure 2-16: A path between nodes i and j with $i_k = i_r$ and arcs such that arcs $(i_k, i_{k-1}) \in \mathbb{A}$, $(i_k, i_{k+1}) \in \mathbb{A}$, $(i_{r-1}, i_r) \in \mathbb{A}$ and $(i_r, i_{r+1}) \in \mathbb{A}$.

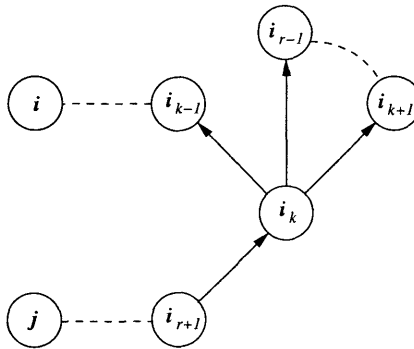


Figure 2-17: A path between nodes i and j with $i_k = i_r$ and arcs such that arcs $(i_k, i_{k-1}) \in \mathbb{A}$, $(i_k, i_{k+1}) \in \mathbb{A}$, $(i_r, i_{r-1}) \in \mathbb{A}$ and $(i_{r+1}, i_r) \in \mathbb{A}$.

Let us consider all the remaining intersection possibilities. We start with the possibility that the path sets only S_{i_k} and then both SI_{i_r} and S_{i_r} . Such possibilities are presented on Figures 2-16 and 2-17.

If $(i_{r-1}, i_r) \in \mathbb{A}$ and $(i_r, i_{r+1}) \in \mathbb{A}$ as on Figure 2-16, then $SI_{i_{k-1}} = S_{i_k} = SI_{i_{k+1}}$ and $S_{i_r} = SI_{i_{r+1}}$. Also, $SI_{i_r} = SI_{i_{k+1}} + R_{i_{k+1}i_r}$ and $SI_{i_r} + T_{i_r} - S_{i_r} = 0$. Therefore,

$$SI_{i_{k+1}} + R_{i_{k+1}i_r} + T_{i_r} - S_{i_r} = R_{i_{k+1}i_r} + T_{i_r} = 0.$$

Hence, $R_{i_{k+1}i_r} = -T_{i_r}$. However,

$$SI_{i_{k-1}} = SI_{i_{r+1}} \text{ and}$$

$$SI_{i_{k-1}} = SI_{i_{r+1}} + R_{i_{k-1}i_{r+1}} = SI_{i_{r+1}} + R_{i_{k+1}i_r} - T_{i_r} = SI_{i_{r+1}} - 2T_{i_r}.$$

Then, $T_{i_r} = 0$. Therefore, such path is not valid.

In the case of Figure 2-17, $S_{i_k} = SI_{i_{k+1}} = SI_{i_{r-1}}$. Since $SI_{i_{k+1}} = SI_{i_{r-1}} + R_{i_{k+1}i_{r-1}}$ and $R_{i_{k+1}i_{r-1}}$ generally is not zero, such intersection is not valid. Suppose $R_{i_{k+1}i_{r-1}} = 0$.

We notice, that $SI_{i_r} + T_{i_r} - S_{i_r} = 0$. Also,

$$SI_{i_{k-1}} = SI_{i_r} + R_{i_{k-1}i_r} = SI_{i_r} + T_{i_k} - T_{i_k} + T_{i_r} = SI_{i_r} + T_{i_r}.$$

Therefore, if we just let the path go from node i_{k-1} to i_k and then directly to i_{r+1} , SI_i would still be the same.

Now, let us consider the case when the path first sets both S_{i_k} and SI_{i_k} and then

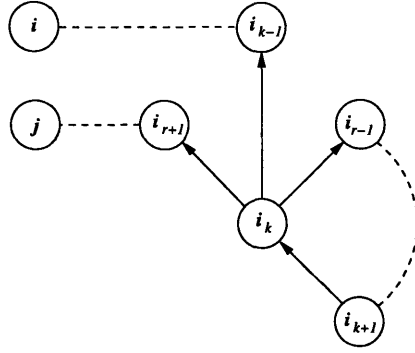


Figure 2-18: A path between nodes i and j with $i_k = i_r$ and arcs such that arcs $(i_k, i_{k-1}) \in \mathbb{A}$, $(i_{k+1}, i_k) \in \mathbb{A}$, $(i_r, i_{r-1}) \in \mathbb{A}$ and $(i_r, i_{r+1}) \in \mathbb{A}$.

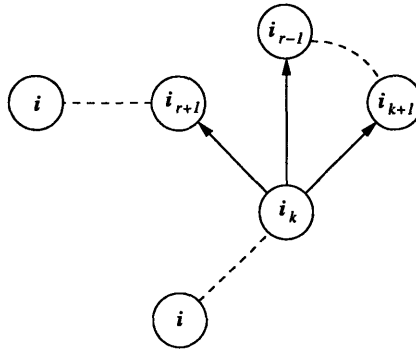


Figure 2-19: A path between nodes i and j with $i_k = i_r$ and arcs such that arcs $(i_k, i_{k+1}) \in \mathbb{A}$, $(i_r, i_{r-1}) \in \mathbb{A}$ and $(i_r, i_{r+1}) \in \mathbb{A}$.

only S_{i_r} . The cases are shown on Figures 2-18 and 2-19.

On Figure 2-18, $SI_{i_{r+1}} = S_{i_k} = SI_{i_{k-1}}$. On the other hand, $SI_{i_{k-1}} = SI_{i_{r+1}} + R_{i_{k-1}i_{r+1}}$. Generally, $R_{i_{k-1}i_{r+1}}$ is not equal to zero and such intersection is not valid. If we have $R_{i_{k-1}i_{r+1}} = 0$, we conclude, that the cycle does not add any new possibility to SI_i and the path can just go from i_{k-1} to i_k and then directly to i_{r+1} .

In the case of Figure 2-19,

$$S_{i_k} = SI_{i_{k+1}} = SI_{i_{r-1}} = SI_{i_{r+1}}.$$

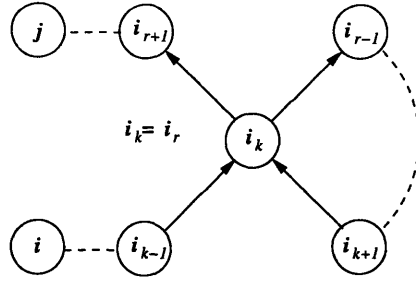


Figure 2-20: A path between nodes i and j with $i = i_k = i_0 = i_r$ and arcs $(i_1, i) \in \mathbb{A}$, $(i_r, i_{r-1}) \in \mathbb{A}$ and $(i_r, i_{r+1}) \in \mathbb{A}$, $r < p$, in the path .

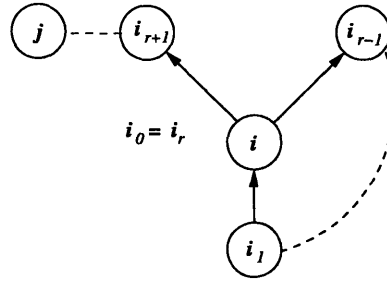


Figure 2-21: A path between nodes i and j with $i_k = i_r$ and arcs $(i_k, i_{k+1}) \in \mathbb{A}$, $(i_k, i_{k-1}) \in \mathbb{A}$, $(i_k, i_{k+1}) \in \mathbb{A}$ and $(i_k, i_{k-1}) \in \mathbb{A}$, $(i_{r-1}, i_r) \in \mathbb{A}$ and $(i_{r+1}, i_r) \in \mathbb{A}$, $r < p$, in the path .

Since $SI_{i_{k+1}} = SI_{i_{r-1}} + R_{i_{k+1}i_{r-1}}$, generally, the path is not valid as $R_{i_{k+1}i_{r-1}} \neq 0$. If $R_{i_{k+1}i_{r-1}} = 0$, the cycle does not bring any new value of SI_i and the path can go from i_k directly to i_{r+1} .

Up till now, all the cycles resulted from the path self intersection appeared to be not useful in creating new possible values of the inbound service time SI_i . There are only two self intersections left to analyze. These self intersections are useful in creating new values.

These situations are depicted on Figures 2-20, 2-21 and 2-22. Figure 2-20 shows that if initially, path P_{ij} passed node i_k such that $(i_{k+1}, i_k) \in \mathbb{A}$ and, if $k > 0$, $(i_{k-1}, i_k) \in \mathbb{A}$ as shown on Figure 2-11, then S_{i_k} is not set by the path. Then the path

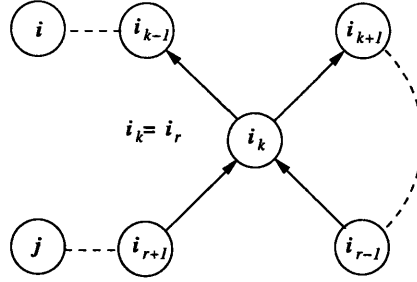


Figure 2-22: A path between nodes i and j with $i_k = i_r$ and arcs $(i_k, i_{k+1}) \in \mathbb{A}$, $(i_k, i_{k-1}) \in \mathbb{A}$, $(i_{r-1}, i_r) \in \mathbb{A}$ and $(i_{r+1}, i_r) \in \mathbb{A}$, $r < p$, in the path .

Node	1	2	3	4	5	6
T_i	7	9	4	9	10	3
s_i				3	15	
h_i	4	8	8	8	6	10

Table 2.4: Parameters of the network from Example 4.

may return to the node such that $i_r = i_k$ and arcs $(i_r, i_{r+1}) \in \mathbb{A}$ and $(i_r, i_{r-1}) \in \mathbb{A}$ and set S_{i_k} leaving SI_{i_k} unaffected. If the path is started in the node $i_k = i_0$, then the path sets $SI_i = S_1$, but does not set S_0 . Therefore, the path can return to node i as show on the figure and set $S_0 = S_{i_r}$.

Similarly, if the path first traverses node i_k such that arcs $(i_k, i_{k+1}) \in \mathbb{A}$ and $(i_k, i_{k-1}) \in \mathbb{A}$ as shown on Figure 2-22, it sets only S_{i_k} and not SI_{i_k} . Then the path can return to node $i_k = i_r$ such that $(i_{r-1}, i_r) \in \mathbb{A}$ and, if $r < p$, $(i_{r+1}, i_r) \in \mathbb{A}$, then it can set the node's SI_{i_k} without affecting the node's S_{i_k} .

To illustrate such possibilities, we give an example of a network with a path with self intersection creating an optimal solution.

Example 4. Consider an instance of the safety stock problem with the network shown on Figure 2-23. The parameters of the problem are presented in Table 2.4. An optimal

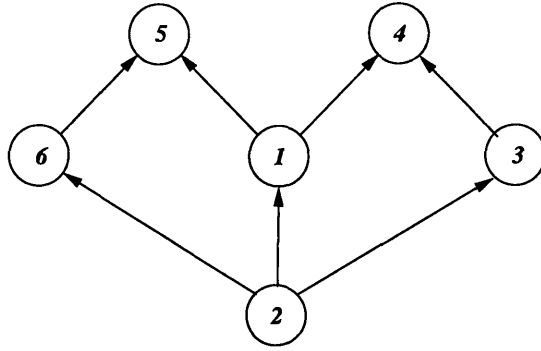


Figure 2-23: An instance of the safety stock problem with an optimal solution corresponding to a path with self intersection.

Node	1	2	3	4	5	6
SI_i	1	0	1	5	5	2
S_i	5	1	5	3	15	5

Table 2.5: Solution of the problem from Example 4.

solution of the problem corresponds to the paths:

$$P_{15} = (1, 2, 3, 4, 1, 5),$$

$$P_{65} = (6, 5).$$

The paths give an optimal solution presented in Table 2.5.

The first path assigns the inbound service time of node 1. Therefore,

$$SI_1 = 9 - 9 - 4 + 7 - 7 + 5 = 1.$$

We note that $SI_5 = s_5 - T_5 = 5$. By lemma 16, the path also assigns the inbound service times of nodes 3 and 4. That is, it sets $SI_3 = -4 + 7 - 7 + 5 = 1$ and $SI_4 = 7 - 7 + 5 = 5$.

The second path only assigns the inbound service time of node 6. Because the outbound service time of node 5 is assigned by the path P_{15} , the second path attempting to go to node 5, finds that SI_5 has already been assigned, and sets $SI_6 = SI_5 - T_6 = 2$.

After discussing all possible cases of path intersection, we make the following conclusions:

1. If a path visits node i_k such that the service times were not previously set by any other path, then P_{ij} can set the service times.
2. If a path visits node i_k and S_{i_k} is set by some other path, but SI_{i_k} is not, then the path can continue after node i_k as long as the path does not attempt to set S_{i_k} . That is, $(i_{k-1}, i_k) \in \mathbb{A}$ and $(i_{k+1}, i_k) \in \mathbb{A}$ as in Figure 2-11. Otherwise, we treat i_{k-1} as the last node of path P_{ij} : $i_{k-1} = j$.
3. If a path visits node i_k and SI_{i_k} is set by some other path, but S_{i_k} is not, then the path can continue after node i_k as long as the path does not attempt to set SI_{i_k} . That is, $(i_k, i_{k-1}) \in \mathbb{A}$ and $(i_k, i_{k+1}) \in \mathbb{A}$ as on Figure 2-12. Otherwise, we treat node i_k as the last node in the path: $i_k = j$.
4. If a path visits node i_k and both service times of node i_k are set by another path, we treat node i_k as the last node of the path: $i_k = j$.

The cases of path intersection suggest, that if we want to explore all the possibilities for the inbound service time SI_i of node i , then we do not need to construct all the paths from node i to other nodes in the network. It is enough to only construct the paths that set only those service times that have not been previously assigned.

Once the path tries to assign already known service times, we can stop the path. Again, technically, we can continue along the path that assigned the service time in the first place, but such action would not uncover any new possibility for SI_i . In fact it might even result in a cycle and an infinite path.

Now, we formulate a lemma, which is important for the algorithm that computes all the possible values of SI_i .

Lemma 17. *Suppose inbound service time is set by the path P_{ij} , then the path has at most $2N$ nodes, where N is the number of nodes in the network.*

Proof. By lemma 16 there always exists a path that determines inbound service time SI_i of any node i in the network. We have shown that if a path passes through node i_k , then it sets at least one of the service times. We have also shown, that if one of the service times is set by some other path, then P_{ij} is only allowed to set the other service time. Moreover, if P_{ij} tries to set the other service time in node i_k , then P_{ij} can be stopped in $i_k = j$ or in $i_{k-1} = j$. Since the network has $2N$ service times, the length of any path is not more than $2N$.

□

This lemma concludes the discussion of the critical paths. We will use the properties of the paths in the chapter 3 when we develop an algorithm in solving general network problems. The paths also appear to be useful for a polynomial time algorithm for the spanning tree networks that is described in Graves and Willems [2000].

Chapter 3

Algorithms

This chapter presents three algorithms for the safety stock problem. The algorithm in Section 3.1 is a polynomial time algorithm for the problems restricted to the spanning tree structure of networks. In section 3.2, we present a branch and bound algorithm for the problems on general networks. In section 3.3, we consider a special case of the general network problem – the problems on two-layer networks. We develop a special branch and bound algorithm for the special case.

3.1 Spanning tree algorithm

In this section, we show how to improve the algorithm for the spanning tree networks from Graves and Willems [2000]. The algorithm described in the paper solves the safety stock problem in pseudo-polynomial time. Namely, the running time of the algorithm is $O(NM^2)$, where M denotes the maximum service time and is bounded by $\sum_{j=1}^N T_j$. Here, we show how to solve the problem in $O(N^3)$.

3.1.1 Optimality conditions

We state optimality conditions from section 2.2.2 for the safety stock problem for a supply chain modelled as a spanning tree network. The main result of this section is that to find an optimal solution of the tree problem, we just need to consider at most $O(N)$ possible values for the service time at any node i of the network.

Let us show that we only need to consider N possible values for SI_i to find an optimal solution for SI_i . From the general case, we know that there exists an optimal solution such that SI_i can be determined by a path P_{ij} from node i to some other node j in the network. Each path from i to j generates a possible value for SI_i :

$$SI_i = R_{ij} + \begin{cases} \max\{0, s_j - T_j\}, & j \in \mathbb{D} \\ 0, & \text{otherwise.} \end{cases}$$

Since the network is a tree, there exists exactly one path between any two nodes. Therefore, there are only N possible values that can be generated using the paths. Moreover, not all the paths would give us feasible inbound service times, since R_{ij} can be negative. Hence, to find an optimal solution, we just need to search through at most N values for the inbound service times for each node in the network.

Before we describe the spanning tree algorithm, we need to discuss how we generate the possible values for the inbound service times, when there are additional constraints on the network, such as exogenous bounds on the service times. After we establish the effect of the conditions, we will then present the algorithm for the spanning tree problem.

We will use the spanning tree networks to create lower bounds on the optimal solution for the general network problem. In addition, we will compute lower bounds for the optimal solution for the general network problem given that some service times, inbound or outbound, are already known. For the bounds to be the most effective, we need to specify what values of the inbound service times we need to consider in each case.

Suppose we have a general network problem and we have fixed some of the inbound and outbound service times. A known inbound service time for SI_j creates constraints for the upstream outbound service times: $S_i \leq SI_j$ for all nodes i such that $(i, j) \in \mathbb{A}$, when SI_j is known. Similarly, a known outbound service time S_i creates constraints for its downstream inbound service times: $SI_j \geq S_i$ for all nodes j such that $(i, j) \in \mathbb{A}$ and S_i is known.

Now, suppose we remove some arcs from a general network, which is equivalent to removing the corresponding constraint $S_i \leq SI_j$ for a removed arc $(i, j) \in \mathbb{A}$. If we try to solve the relaxed problem now, we can find a lower bound on the optimal cost of the general problem. Because some arcs are removed, the solution of the relaxed problem needs not satisfy the constraints imposed by the removed arcs. Therefore, the bounds can be tightened by imposing the constraints after removing the arcs. For that we have to reconsider candidate values for the optimal service times that we calculate using paths.

We only need to reconsider the values of the last node in a path. We notice, that if we construct a path between nodes i and j to find a possible optimal value for SI_i , as shown in lemma 16, it is always true that $SI_i = R_{ij} + SI_j$. We stop the path in

node j , because SI_j can be equal to some value independent of the path, i.e. 0 or $s_j - T_j$ if $j \in \mathbb{D}$. Now, as we have additional constraints on the SI_j , we might have more possibilities for the independent values. Here, we notice, if SI_j is known, then the known value is the only possibility for SI_j . Also notice, that the end node only has to respect constraints imposed by the service times of the node, since we assume that the constraints of the type $S_i \leq SI_j, (i, j) \in \mathbb{A}$ are satisfied if we stop the path.

Suppose we have a bound on SI_j , which we denote \underline{SI}_j . Then similar to the proof of lemma 14, we list all the constraints on SI_j :

$$SI_j \geq S_k, (k, j) \in \mathbb{A}; \quad (3.1)$$

$$SI_j \geq S_j - T_j \quad (3.2)$$

$$SI_j \geq \underline{SI}_j \quad (3.3)$$

$$SI_j \geq 0 \quad (3.4)$$

Again, since the safety stock function in node j increases when SI_j increases, in an optimal solution, we can set SI_j to the smallest possible value. Therefore, \underline{SI}_j is a possible value for the end node in the path.

Suppose now S_j is known or S_j has an upper bound \bar{S}_j . Then, intuitively, node j can be treated as a demand node. In this sense, we assume here that for a demand node j , $\bar{S}_j = s_j$. Therefore, if S_j is known, then a possibility for SI_j is $\max\{\underline{SI}_j, S_j - T_j\}$, where \underline{SI}_j here might be interpreted as 0 if no bound on SI_j is imposed.

	S_j is known	\bar{S}_j is known	neither is known
SI_j is known	SI_j	SI_j	SI_j
\underline{SI}_j is known	$\max\{\underline{SI}_j, S_j - T_j\}$	$\{\underline{SI}_j, \max(S_j - T_j, \underline{SI}_j)\}$	\underline{SI}_j
neither is known	$\max\{0, S_j - T_j\}$	$\{0, \max(S_j - T_j, 0)\}$	0

Table 3.1: Possible values of SI_j depending on the constraints on the service times.

	S_j is known	\bar{S}_j is known	neither is known
SI_j is known	S_j	$\min\{\bar{S}_j, SI_j + T_j\}$	$SI_j + T_j$
\underline{SI}_j is known	S_j	$\{\bar{S}_j, \min(SI_j + T_j, \bar{S}_j)\}$	$\underline{SI}_j + T_j$
neither is known	S_j	\bar{S}_j	T_j

Table 3.2: Possible values of S_j depending on the constraints on the service times.

Consider the case when \bar{S}_j is known. Then,

$$S_j \leq SI_k, (j, k) \in \mathbb{A}; \quad (3.5)$$

$$S_j \leq SI_j + T_j \quad (3.6)$$

$$S_j \leq \bar{S}_j \quad (3.7)$$

$$S_j \geq 0 \quad (3.8)$$

Since the safety stock function in node j decreases when S_j increases, in an optimal solution, we can set S_j to be as big as possible. Then, \bar{S}_j is a possibility for S_j . In this case, $SI_j = \bar{S}_j - T_j$ is a possibility for the optimal inbound service time. We notice here, that if both \bar{S}_j and \underline{SI}_j are known, then two values for SI_j have to be considered: \underline{SI}_j and $\bar{S}_j - T_j \geq \underline{SI}_j$. The first value corresponds to the possibility that $S_j \neq \bar{S}_j$ and the second value is for the case when $S_j = \bar{S}_j$.

Tables 3.1 and 3.2 summarize the discussion of the candidate values for the in-

bound service times. The first row of the tables shows whether or not we know S_j or \bar{S}_j . The first column shows whether or not we know SI_j or \underline{SI}_j .

If we consider a path from i to j , the end node can have at most two values. Indeed, from Table 3.1 we see that no matter what we know about the last node in the path, there are at most two candidate values for SI_j . Therefore, there is at most two candidate values for the start node SI_i that can be created by the path.

Now, we return to the discussion of the spanning tree algorithm. There is only one path between any two nodes in the network. Even if we have some constraints on the service times of the network, each path produces at most two possible value for an optimal inbound or outbound service time. From this we conclude, that there are at most $2N$ values to consider for each inbound or outbound service time for each node in the tree.

The number of possible values of the service times can be reduced also due to the property that $S_i \leq M_i$ and $SI_i \leq M_i - T_i$ for every node i in the network. Therefore, the number of values that we have to consider to find an optimal solution for each service time for any node i is $\min\{2N, M_i\}$.

3.1.2 Polynomial time algorithm

For greater generality, we quote the algorithm from Graves and Willems [2000]. Then we will show that the algorithm can be modified to run in polynomial time using the optimality conditions for the tree networks discussed in section 3.1.1.

We first renumber the nodes of the tree, using the following procedure:

1. Start with all nodes in the unlabeled set, U .
2. Set $k := 1$.
3. Find a node $i \in U$ such that 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 \mathbb{A}_T defined on U .
4. Remove node i from set U and insert into the labelled set L ; label node i with index k .
5. Stop if U is empty; otherwise set $k := k + 1$ and repeat steps 3-4.

The complexity of this labelling algorithm, as quoted from Graves and Willems [2000], is $O(N^2)$. However, the same labelling can be implemented in $O(N)$. The implementation is shown in Algorithm 2 and is essentially a depth first search.

Algorithm 2 Labelling algorithm

var *currentNode* $\leftarrow 1$ – index in the new order
 ComputeNewOrder(1, \emptyset)

Procedure ComputeNewOrder(k, p)

1: **for all** nodes $j \neq p$ connected to k **do**
 2: ComputeNewOrder(j, k)
 3: **end for**
 4: *newOrder*[*currentNode*] $\leftarrow k$
 5: *currentNode* \leftarrow *currentNode* + 1

end procedure

We can visualize the algorithm as follows. Suppose we pick a node, for example, node 1. We hang the tree by node 1. Then starting from node 1, we do the depth first search on the hanged tree. We will use this visualization of the tree later in the section and call the hanged tree as a search tree.

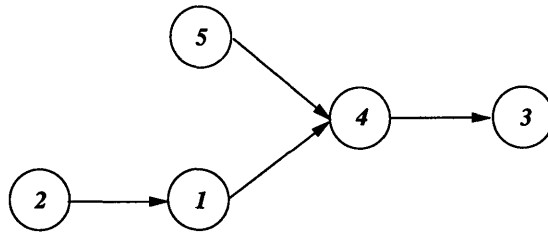


Figure 3-1: A spanning tree network for Example 5.

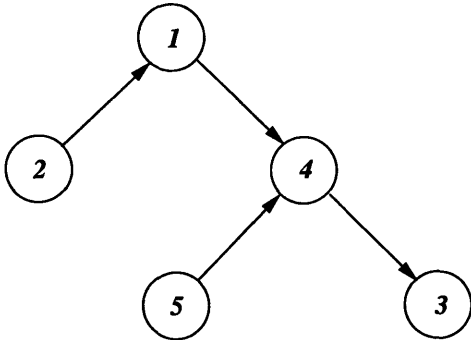


Figure 3-2: Search tree.

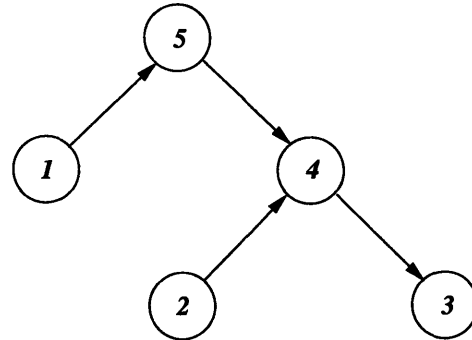


Figure 3-3: New labelling.

Let us consider an example of labelling and of a search tree.

Example 5. Consider a network presented on Figure 3-1. The search tree is shown on Figure 3-2 and a new labelling is shown on Figure 3-3.

The vector *newOrder* from Algorithm 2 is

$$\text{newOrder} = [2, 5, 3, 4, 1].$$

We will use the notation $p(k)$ for the node of the network connected to node k such that $p(k) > k$. Note that for each node $k < N$ there exists only one $p(k)$ according to the described procedure. For example, in Figure 3-3, $p(1) = 5$, $p(2) = 4$, $p(3) = 4$, $p(4) = 5$ and node 5 has no $p(5)$.

To introduce the functions of the algorithm, we take the approach provided in Graves and Willems [2000]. We first state the key relationships in the algorithm and will then explain each of the terms.

$$c_k(S, SI) = h_k \{ D_k(SI + T_k - S) - (SI + T_k - S)\mu_k \} \\ + \sum_{(i,k) \in \mathbb{A}_T, i < k} f_i(S_i^*(SI)) + \sum_{(k,j) \in \mathbb{A}_T, j < k} g_j(SI_j^*(S)),$$

where

$$f_k(S) = \min_{SI} \{ c_k(S, SI) \}$$

for $\max(0, S - T_k) \leq SI \leq M_k - T_k$ and SI integer;

$$S_i^*(SI) = \arg \min \{ f_i(S) : 0 \leq S \leq \min(SI, M_i), S \text{ integer} \};$$

$$g_k(SI) = \min_S \{ c_k(S, SI) \}$$

for $0 \leq S \leq SI + T_k$ and S integer ;

$$SI_i^*(S) = \arg \min \{ g_i(SI) : S \leq SI \leq M_i - T_i, SI \text{ integer} \}.$$

If node k is a demand node, $S \leq s_k$.

To understand the meaning of the introduced functions, we define the following subsets of nodes. For each node k we define N_k to be a subset of $\{1, \dots, k\}$ such that there exists an undirected path in the subgraph induced by N_k from each node in N_k

to k :

$$N_k = \{k\} + \bigcup_{i < k, (i,k) \in \mathbb{A}_T} N_i + \bigcup_{j < k, (k,j) \in \mathbb{A}_T} N_j.$$

We can see that the function $c_k(S, SI)$ is the holding cost of the safety stock for the subnetwork with node set N_k as a function of inbound and outbound service times at node k . The first term is the holding cost at node k . The second and the third terms correspond to the nodes that are upstream and downstream of k in N_k respectively. $f_i()$ is the minimum holding cost of the subnetwork N_i as a function of SI if i supplies node k . $g_j()$ is the minimum holding cost of subnetwork N_j as a function of S if j is downstream of k . For node kA , the algorithm evaluates $f_k(S)$ for $S = 0, \dots, M_k$ if $p(k)$ is downstream of k . It evaluates $g_k(SI)$ for $SI = 0, \dots, M_k - T_k$ if $p(k)$ is upstream of k .

For completeness, we cite the algorithm from Graves and Willems [2000]. We modify the algorithm only to accommodate the bounds on the service times that will be useful when we use the tree solutions to obtain lower bounds on the general problem. We define bounds \bar{S}_i and \underline{SI}_i as follows:

$$\bar{S}_i = \begin{cases} \infty, & \text{if all } SI_j, (i, j) \in \mathbb{A} \text{ are unknown;} \\ \min SI_j, (i, j) \in \mathbb{A} & \text{if } SI_j \text{ is known.} \end{cases}$$

$\underline{SI}_i = 0$ by default. Then, the algorithm states as follows as in Algorithm 3.

The algorithm, as stated, runs in $O(NM^3)$. We show here, how to make the algorithm run in $O(N^3)$. The main improvement in the algorithm comes from the fact that there are at most $\min\{2N, M_i\}$ values that have to be considered for each inbound

Algorithm 3 Spanning tree algorithm from Graves and Willems [2000]

```
for  $k =: 1$  to  $N - 1$  do
  if  $p(k)$  is downstream of  $k$  then
    evaluate  $f_k(S)$  for  $S = 0, \dots, \min\{M_k, \bar{S}_k\}$ 
  else if  $p(k)$  is upstream of  $k$  then
    evaluate  $g_k(SI)$  for  $SI = \max\{0, \underline{SI}_k\}, \dots, M_k - T_k$ 
  end if
end for
evaluate  $g_N(SI)$  for  $SI = \max\{0, \underline{SI}_N\}, \dots, M_N - T_N$ 
Minimize  $g_N(SI)$  for  $SI = \max\{0, \underline{SI}_N\}, \dots, M_N - T_N$  to obtain the optimal objective function value
```

service time SI_i and each outbound service time S_i as discussed in section 3.1.1.

However, the first task is to efficiently compute the candidate values. This can be done using two procedures in Algorithm 4.

The procedures in Algorithm 4 are initialized with the same k and empty p ($p = \emptyset$). We first call $\text{ComputeSandSIUp}(k, p)$ and then $\text{ComputeSandSIDown}(k, p)$ as shown in Algorithm 5. In the main part of Algorithm 5, $k = 1$, however, we can initialize the procedures with any node k in the network.

Now, we explain the meaning of the procedures. The main goal here is to compute all of the candidate values of SI_i and S_i by using the paths from node i to all possible nodes in the tree. We store the candidate inbound service time values of node k in $SIset[k]$ and the candidate outbound service time of node k in $Sset[k]$. By lemma 16, we generate a candidate value from $SI_i = SI_j + R_{ij}$ for every node i in the network where SI_j takes values from Table 3.1. We notice, that because $R_{ij} = R_{ik} + R_{kj}$ for any node k in the path, while computing value SI_i using path P_{ij} , we can simultaneously compute the value for SI_k using path P_{kj} .

Procedure $\text{ComputeSandSIUp}(k, p)$ is a depth first search starting from any node

Algorithm 4 Service time computation

procedure ComputeSandSIUp(k, p)

1: **for all** nodes $j \neq p$ connected to k **do**
2: ComputeSandSIUp(j, k)
3: **end for**
4: $SIset[k] \leftarrow \{ \text{values as defined in Table 3.1} \}$
5: $Sset[k] \leftarrow \{ \text{values as defined in Table 3.2} \}$
6: **for all** upstream nodes $j \neq p, (j, k) \in \mathbb{A}$ **do**
7: $SIset[k] \leftarrow SIset[k] \cup Sset[j]$
8: $Sset[k] \leftarrow Sset[k] \cup \{SI + T_k : SI \in SIset[k]\}$
9: **end for**
10: **for all** downstream nodes $j \neq p, (k, j) \in \mathbb{A}$ **do**
11: $Sset[k] \leftarrow Sset[k] \cup SIset[j]$
12: $SIset[k] \leftarrow SIset[k] \cup \{S - T_k : S \in Sset[k]\}$
13: **end for**

end procedure**procedure** ComputeSandSIDown(k, p)

14: **for all** upstream nodes $j \neq p$ **do**
15: $Sset[j] \leftarrow Sset[j] \cup SI[k]$
16: $SIset[j] \leftarrow SIset[j] \cup \{S - T_j : S \in Sset[j]\}$
17: ComputeSandSIDown(j, k)
18: **end for**
19: **for all** downstream nodes $j \neq p$ **do**
20: $SIset[j] \leftarrow SIset[j] \cup Sset[k]$
21: $Sset[j] \leftarrow Sset[j] \cup \{SI + T_j : SI \in SIset[j]\}$
22: ComputeSandSIDown(j, k)
23: **end for**

k , for example from node $k = 1$. We create a search tree that starts from the chosen node and constructed as shown in $\text{ComputeSandSIUp}(1, \emptyset)$. For each pair of nodes k and p , $(k, p) \in \mathbb{A}$ or $(p, k) \in \mathbb{A}$, we call node p as the parent of node k in the search tree, if $\text{ComputeSandSIUp}(k, p)$ considers node p before node k . In this case, we also call node k as a child of node p in the search tree.

The procedure works as follows. We treat each node as a possible last node in a path. Therefore, we first add the values from Tables 3.1 and 3.2 to $SIset[k]$ and $Sset[k]$ respectively for each k in the network. Then we create values for longer paths by passing all possible values of a child to its parent in the search tree. By doing so, we respect the properties of the path. In particular, if node j is upstream of its parent k in the search tree, $(j, k) \in \mathbb{A}$, then by the properties of the paths $SI_k = S_j$ is a possible value that we have to add to the set $SIset[k]$. Also, $S_k = SI_k + T_k = S_j + T_k$ is a value that we have to put into the set $Sset[k]$.

Similarly, if node j is downstream of its parent k in the search tree, $(k, j) \in \mathbb{A}$, then $S_k = SI_j$ is a possible value that we have to add to the set $Sset[k]$. Also, $SI_k = SI_j - T_k$ is a possible value that we have to put into the set $SIset[k]$ by construction of some path.

After passing all the possible values from the children to the parents in the search tree, we need to pass all the possible values from the parents to the children to account for all possible paths. Procedure $\text{ComputeSandSIDown}(k, p)$ does that in a similar way as procedure $\text{ComputeSandSIUp}(k, p)$.

We notice here, that the sets $Sset[k]$ and $SIset[k]$ can be reduced by using the constraints \underline{SI}_k and \bar{S}_k as well as any values for known SI_k and S_k . In the lines

nodes	1	2	3	4	5
T	3	4	2	2	5
s			3		

Table 3.3: Lead times and guaranteed service times for Example 6

nodes	1	2	3	4	5
$Sset$	{3}	{7}	{3}	{2}	{8}
$SIset$	{0}	{3}	{1}	{0}	{3}

Table 3.4: Initial values of $Sset[i]$ and $SIset[i]$ for Example 6.

7, 8, 15, 16, 20, 21 of Algorithm 4, we will only add those values that satisfy the constraints. In particular, we add S_k or SI_k only if $S_k \leq \bar{S}_k$, $SI_k \geq \underline{SI}_k$. Also, we do not need to add new values to set $Sset[k]$ if S_k has been assigned before. Similarly, we do not add new values to $SIset[k]$ if SI_k has been assigned before.

The complexity of Algorithm 4 is $O(N^2)$. Since the network is a spanning tree, each node has exactly one parent in the search tree. Therefore, both procedures touch each parent once. For each parent, they touch each child once as well. Therefore, the complexity of this algorithm is $O(N^2)$.

We give an example of the sets computation.

Example 6. Consider the graph shown on Figure 3-3. We assume that S_1 and SI_2 are known and that $S_1 = 3$ and $SI_2 = 3$. Therefore, $\underline{SI}_5 = 3$. The lead times and the guaranteed service times of the demand nodes are shown in Table 3.3.

Suppose we call both procedures with $k = 5$. First, in Table 3.4, we report the initial sets $Sset[i]$ and $SIset[i]$ as created by lines 4 and 5 of Algorithm 4.

Second, in Table 3.5, we report sets $Sset[i]$ and $SIset[i]$ as created by the procedure *ComputeSandSIUp* of Algorithm 4, respecting the bounds explained previously. At the

nodes	1	2	3	4	5
$Sset$	{3}	{7}	{3}	{2, 9, 1}	{8, 0, 7}
$SIset$	{0}	{3}	{1}	{0, 7}	{3}

Table 3.5: Values of $Sset[i]$ and $SIset[i]$ created by procedure ComputeSandSIUp for Example 6.

nodes	1	2	3	4	5
$Sset$	{3}	{7, 0, 8}	{3}	{2, 9, 1, 10}	{8, 0, 7}
$SIset$	{0}	{3}	{1, 2, 9, 10}	{0, 7, 8}	{3}

Table 3.6: Values of $Sset[i]$ and $SIset[i]$ created by procedure ComputeSandSIDown for Example 6.

end of this procedure, the chosen root of the search tree, node 5, has the full sets of its candidate values.

Finally, Table 3.6 shows complete sets $Sset[i]$ and $SIset[i]$ as created by the procedure ComputeSandSIDown of Algorithm 4, respecting the bounds explained previously.

After computing all possible values for the service times for all the nodes, we can solve the spanning tree problem in $O(N^3)$. The improved algorithm is in Algorithm 5.

The complexity of the spanning tree algorithm is $O(N^3)$. For each node k in the spanning tree the algorithm computes the values $c_k(S, SI)$ for all possible values of S and SI . Since the number of candidate inbound and outbound service times in node k is at most $2N$, there are $O(N^2)$ values of $c_k(S, SI)$. Computation of each value $c_k(S, SI)$ takes $O(1)$ given that we pre-compute $cS_k(S)$ and $cSI_k(SI)$. $cS_k(S)$ is the total minimum cost for all the nodes $i < k, (k, i) \in \mathbb{A}$ and $SI_i \geq S$. $cSI_k(SI)$ is the total minimum cost for all the nodes $i < k, (i, k) \in \mathbb{A}$ and $S_i \leq SI$.

Procedure precompute() computes the values of $cS_k(S)$ and $cSI_k(SI)$ for all k . The total time of computing the values for all k is $O(N^3)$. There are N nodes for

Algorithm 5 Tree Algorithm

function ComputeTreeMinCost(*tree*, *SI*, *S*, *SI*, \bar{S})

- 1: ComputeSandSIUp(1, \emptyset)
- 2: ComputeSandSIDown(1, \emptyset)
- 3: ComputeNewOrder(1, \emptyset)
- 4: **return** ComputeMinCost()

end function**function** ComputeMinCost()

- 5: **for** $k = 1$ to $N - 1$ **do**
- 6: precompute(k)
- 7: **if** $p(k)$ is upstream of k **then**
- 8: $g_k[SI] \leftarrow \min_{S \in Sset[k]} c_k(S, SI)$, **for all** $SI \in SIset[k]$
- 9: **end if**
- 10: **if** $p(k)$ is downstream of k **then**
- 11: $f_k[S] \leftarrow \min_{SI \in SIset[k]} c_k(S, SI)$, **for all** $S \in Sset[k]$
- 12: **end if**
- 13: **end for**
- 14: $g_N[SI] \leftarrow \min_{S \in Sset[N]} c_N(S, SI)$, **for all** $SI \in SIset[N]$
- 15: **return** *TreeMinCost* $\leftarrow \min_{SI \in SIset[N]} g_N[SI]$

end function**procedure** precompute(k)

- 16: **for all** $SI \in SIset[k]$ **do**
- 17: $cSI_k[SI] \leftarrow \sum_{i:(i,k) \in \mathbf{A}, i < k} \min_{S \leq SI} f_i[S]$
- 18: **end for**
- 19: **for all** $S \in Sset[k]$ **do**
- 20: $cS_k[S] \leftarrow \sum_{i:(k,i) \in \mathbf{A}, i < k} \min_{S \geq SI} g_i[SI]$
- 21: **end for**

end procedure**function** $c_k(S, SI)$

- 22: **return** $cS_k[S] + cSI_k[SI] + SS_k(S, SI)$

end function

which we pre-compute $cS_k(S)$ and $cSI_k(SI)$. For each k , there are n_k nodes $i < k$. Since the graph is a tree, $\sum n_k = N$. For each node $i < k$, we have at most $2N$ service times to select the minimum cost of the subnetwork N_i . Therefore, the total time is

$$\sum_{k=1}^N n_k * 2N * 2N = N * 2N * 2N = O(N^3).$$

So far we have presented the basic algorithm for solving the safety stock problem on a spanning tree network. Now, we note that we might have less than $2N$ possible service times to consider at each node. First, we need not consider candidate times from paths with negative R_{ij} . We also do not consider the values $SI_i > M_i - T_i$ and $S_i > M_i$. Therefore, the number of possible values of the inbound service time in node i is $\min\{2N, M_i - T_i\}$. The number of possible values of the outbound service times of node i is $\min\{2N, M_i\}$. Therefore, the complexity of the algorithm is in fact $O(N \min\{2N, M\}^2)$, where M is the maximum service time and is bounded by $\sum_{j=1}^N T_j$.

3.2 Algorithm for general networks

In this section we develop a branch and bound algorithm for general network supply chains. The algorithm searches for the solutions that satisfy properties described in lemma 14. In the next section, we construct the branching tree and suggest the bounds.

3.2.1 Branching tree

To construct a branching tree, we use the properties of an optimal solution, described in section 2.2.2. We want to find an optimal solution, that has $S_i = s_i$ for each demand node i and $SI_j = 0$ for each component j . The solution will also satisfy the properties listed in lemma 14. By Observation 2 from that section, the objective function can be expressed as a function of the inbound service times only. From section 2.2.3, we know that an optimal inbound service time for node i can be found from a finite set of candidate values that can be generated from the paths from node i to all other nodes in the network. Indeed, we showed that each path has at most $2N$ nodes, therefore, there is a finite number of paths between the nodes to consider.

Using definition of the paths, we construct a branching tree for the problem. First, we impose an order of the nodes in the graph. For now, we assume any order, but we return to the question of how to order the nodes later in this chapter.

Suppose we initiate the branching tree with node 1 in the network. We construct all the valid paths as described in section 2.2.3. From the paths P_{1j} we generate the sets of candidate inbound service times SI_1 . Notice, that there might be multiple paths between nodes 1 and j . Each path gives a different branching point. These are the first level branching points of the branching tree.

At the i th level of the branching tree, for every branching point of level $i - 1$, we will branch on candidate values for SI_i . We create one branch for each candidate value for SI_i . We compute the candidate values for SI_i from considering all of the

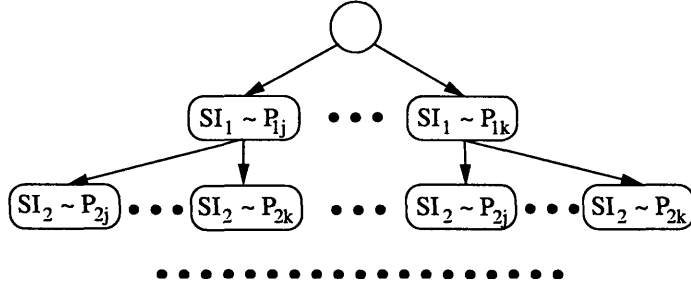


Figure 3-4: Branching tree for the general problem.

valid paths from node i to every other node j in the network:

$$SI_i = R_{ij} + \begin{cases} \max\{0, s_j - T_j\}, & j \in \mathbb{D} \\ 0, & \text{otherwise.} \end{cases} \quad (3.9)$$

These values are the branching points of level i .

Schematically, the branching tree is shown on Figure 3-4. We use notation $SI_i \sim P_{ij}$ to indicate that the value SI_j is computed using path P_{ij} by formula (3.9).

Although the number of paths from any node to any other node is large, here we indicate how to eliminate some paths and, therefore, some branches of the branching tree. First, according to the discussion in section 2.2.3, a path can be stopped if it tries to assign an already assigned service time. Therefore, the arcs that have been included in previously set paths can not enter the current path.

The next reduction in the number of paths comes from lemma 16. Suppose we have constructed a path P_{ij} . According to the lemma, for any node k in the path, that is not upstream of its neighbors in the path, $SI_k = R_{kj} + SI_j$. Thus, the path determines the value for SI_k , $k > i$. Therefore, there is only one branch at level k that is in the same branch as $SI_i \sim P_{ij}$.

Moreover, by the problem formulation, $SI_k \geq 0$. Since R_{kj} can be negative, we have to check all SI_k along the path P_{ij} . If at least one $SI_k < 0$, path P_{ij} is not valid. Consequently, we can eliminate the branch $SI_i \sim P_{ij}$. Also, we can check that all the SI_k on the path are such that $SI_k \leq M_k - T_k$.

For example, it is never valid to finish the path with an arc (k, j) if j is not a demand node. Indeed, if the situation as shown on Figure 2-14 with $i_{p-1} = k$, then the path tries to assign SI_k . But then $SI_k = R_{kj} + 0 = -T_k$. If the situation is as shown on Figure 2-13, then SI_k is not assigned, but $SI_j = SI_{i_{p-2}}$. Therefore, the path $P_{i_{p-2}}$ gives the same value of SI_i . Therefore, there is no need to consider such a path P_{ij} . We have shown, that we do not need to consider paths with the last arc going in the direction of the path.

Similarly, we can assign S_k according to the path. By corollary 5, unless k is not downstream of both of its neighbors in the path, S_k can be computed according to the path. In particular, $S_k = SI_m$, where $(k, m) \in \mathbb{A}$ and is in the path. Therefore, we can check, that $0 \leq S_k \leq M_k$. If for at least one node in the path, the condition is not satisfied, the path is not valid. Therefore, branch $SI_i \sim P_{ij}$ does not exist.

We can do even more constraint checking to eliminate some paths. Suppose path P_{ij} tries to assign SI_k , $k \in P_{ij}$. Then the path is not valid if for some $(r, k) \in \mathbb{A}$, S_r has been assigned earlier in the branching tree, and $SI_k < S_r$. That is, path P_{ij} can not violate previously assigned values of service times. Similarly, we check, that if the path assigns S_k , then for $(k, r) \in \mathbb{A}$, if SI_r has been assigned by some other path, we must have $S_k \leq SI_r$.

Finally, since we are looking for an optimal solution with $SI_i = 0$, $i \in \mathbb{C}$ and

$S_j = s_j, j \in \mathbb{D}$, we can eliminate all the paths that violate these conditions.

Algorithm 6 Generate paths

$P[]$ – array of the nodes in the generated path

GeneratePath($i, 1, true$)

procedure GeneratePath($k, ind, continue$)

```

1:  $P[ind] \leftarrow k$ 
2: process path  $P[1, \dots, ind]$ 
3: if  $continue = true$  then
4:   if  $S_k$  is not set then
5:     mark  $S_k$  to be set
6:     for all  $r, (k, r) \in \mathbb{A}$  and  $SI_r$  is not marked to be set do
7:       mark  $SI_r$  to be set
8:       if  $SI_r$  is not set then
9:         GeneratePath( $r, ind + 1, true$ )
10:      else
11:        GeneratePath( $r, ind + 1, false$ )
12:      end if
13:      unmark  $SI_r$  to be set
14:    end for
15:    unmark  $S_k$  to be set
16:  end if
17:  if  $SI_k$  is not set then
18:    mark  $SI_k$  to be set
19:    for all  $r, (r, k) \in \mathbb{A}$  and  $S_r$  is not set and  $S_r$  is not marked to be set do
20:      mark  $S_r$  to be set
21:      GeneratePath( $r, ind + 1, true$ )
22:      unmark  $S_r$  to be set
23:    end for
24:    unmark  $SI_k$  to be set
25:  end if
26: end if
end procedure

```

We call the paths, that are constructed according to the rules in section 2.2.3, as valid paths. We check all the service times, that are computed according to the valid paths, as described above. We call the qualified service times as valid service times.

Formally, an algorithm for generating valid paths is stated in Algorithm 6. The procedure, called GeneratePath($i,1,true$), will generate all the valid paths that start

in node i . For each path, line 2 will be executed. The procedure “process path” in line 2 can, for example, collect the paths into a set of paths, or compute the value of SI_i corresponding to the current path together with the values of the service times along the path. We note here that the paths can be generated as they are needed by the branch and bound algorithm avoiding storing them. In lines 4, 8, 17 and 19 we check whether S_k and SI_k have been previously set in the branching tree or equivalently by another path. In lines 5, 7, 18 and 20 we mark the service times of the nodes that we will set according to the considered path as “to be set”. In lines 13, 15, 22 and 24 we unmark the service times of the path after the path has been considered.

We now proceed to describe the bounds for the algorithm.

3.2.2 Lower Bounds

To construct a lower bound on the optimal solution, we relax some constraints of the problem \mathcal{P} . In particular, we remove a number of constraints of the form $S_i \leq SI_j, (i, j) \in \mathbb{A}$, which is the same as removing corresponding arcs (i, j) from the arc set \mathbb{A} . Our goal here is to remove enough arcs from the graph so that the resulting graph becomes a spanning tree. We solve the spanning tree problem by applying the algorithm described in section 3.1.

Since we develop a branch and bound algorithm, we wish to find lower bounds at each branch of the branching tree. For each branch, we know some service times in the network. We can construct a spanning tree and then solve the spanning tree problem for which some of the service times are set as discussed in section 3.1. To

make the lower bound as large as possible, we create an upper bound on the outbound service time of node i :

$$\bar{S}_i = \min\{M_i, SI_j : (i, j) \in \mathbb{A}, SI_j \text{ is set}\}.$$

Similarly, a lower bound on the inbound service time of node i is

$$\underline{SI}_i = \max\{0, S_j : (j, i) \in \mathbb{A}, S_j \text{ is set}\}.$$

The main question in the lower bound construction is which spanning tree to use, or equivalently, which arcs to drop from the network. One solution is to take any tree independent of what we know about the parameters of the problem, such as lead times, holding cost and so on. The tree can also be independent of the branching process, i.e., it does not benefit from the fact that we might know some service times. We call this a randomly-generated spanning tree.

Another possible spanning tree is what we call a smart tree. At any branching point, a smart tree takes into account the service times assigned earlier in the branching tree. The tree is constructed by choosing specific arcs from the general network that create spanning tree. When we choose the arcs for the smart tree, we first take arcs $(i, j) \in \mathbb{A}$ that have both S_i and SI_j unassigned. Only when we do not have this kind of arcs, we include arcs $(i, j) \in \mathbb{A}$ with assigned S_i or SI_j .

The main idea behind constructing a smart tree is that if arc $(i, j) \in \mathbb{A}$ has both S_i and SI_j assigned, the constraint $S_i \leq SI_j$ is not violated. Therefore, if we remove

this arc, the spanning tree solution does not violate the constraint on the arc. Such a tree has a better chance to give a higher lower bound than a solution of a random tree.

We will test the tree construction ideas in Chapter 4.

Another question about the lower bounds is how many lower bounds (i.e., distinct spanning trees) to consider for each branch as well as for the initial node of the branch and bound tree. It is clear, that the more spanning trees we construct each time we want to bound a solution, the more chance we have to get a tight lower bound. However, constructing spanning trees and solving the problem on the trees takes time. Therefore, we have to find an optimal balance between spending time on computing better lower bounds and spending more time on branching. We address this question in Chapter 4.

Good initial bounds can be achieved by computing more tree solutions in the beginning. However, the number of the initial bounds has to be optimized as well. We find that for sparse graphs, there is a high chance that a tree solution gives an optimal solution of the general problem. Unfortunately, there are graphs for which no tree produces an optimal solution for the general problem. Let us consider an example of such network.

Example 7. Consider the graph shown on Figure 3-5 with parameters in Table 3.7.

The safety stock function in node i is

$$SS_i(\tau_i) = \sigma_i \sqrt{\tau} + \mu_i \tau_i,$$

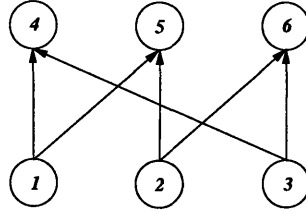


Figure 3-5: The network for Example 7.

node	1	2	3	4	5	6
T	2	9	12	19	14	19
s				15	11	1
μ	16.13	20.38	8.98	6.48	19.77	20.05
σ	17.14	18.01	18.47	10.04	16.16	5.96
h	39.04	47.7	20.74	61.87	91.13	84.73

Table 3.7: Parameters of the problem in Example 7.

where $\tau_i = SI_i + T_i - S_i$.

An optimal solution is shown in Table 3.8. The optimal cost is 10354.72.

The network is a cycle with six arcs. Therefore, we construct a spanning trees by removing one arc from the network. We can construct six different trees and solve six different relaxed problems. The trees with the solutions are shown on Figure 3-6. We see that no lower bound is equal to the optimal solution.

The example shows that computing many initial lower bounds need not solve the general network problem.

Computing initial lower bounds and branching can be considered as two competing methods for solving the safety stock problem. On one hand, we have the branching

node	1	2	3	4	5	6
S	2	9	12	15	11	1
SI	0	0	0	12	9	12

Table 3.8: An optimal solution for the network in Example 7.

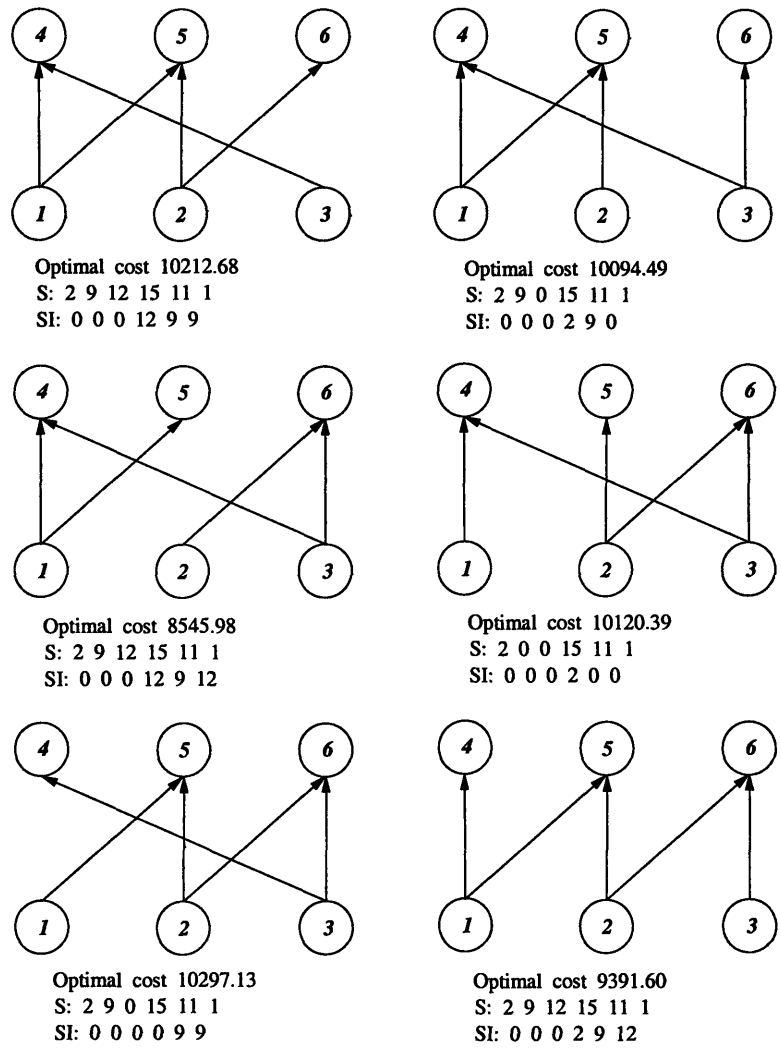


Figure 3-6: Spanning trees and correspondent relaxed problem solutions for Example 7.

tree, which can be quite big for some problems. On the other, the lower bound from the tree relaxation can find the optimal solution by chance, but there is no guarantee that there is a tree that gives the optimal solution to the general problem as shown in Example 7.

As we show in Chapter 4, on average, a good method to solve the general problem is to combine the two methods. We can compute more and more bounds on the optimal solution as we branch. For example, every time we compute the lower bound for a branch using the bounds on the service times, we can also compute an unrestricted version of the spanning tree problem to find the lower bound on the optimal solution.

3.2.3 Upper Bounds

Any feasible solution of the problem provides an upper bound to the optimal solution. For example, the lower bound solution we obtained in the previous section can be changed so that it becomes feasible as follows. The lower bound is characterized by the set of outbound service times for all of the nodes. The set of inbound service times of the nodes that correspond to the spanning tree solution is feasible for the spanning tree we choose. However, it might be infeasible if we consider the set of all arcs \mathbb{A} . In order to make the solution feasible, we can, for example, increase the inbound service time for each node until it satisfies all of the constraints implied by the outbound service times, when we consider all arcs in \mathbb{A} . The solution we get is an upper bound on the optimal solution and we will call it a fixed solution.

More precisely, if $(S_1, \dots, S_N, SI_1, \dots, SI_N)$ is a solution obtained from a spanning

node	1	2	3	4	5	6
S	2	9	12	15	11	1
SI	0	0	0	12	9	12

Table 3.9: Fixed solution for Example 8.

tree relaxation, then we generate a feasible solution to the general network problem as follows:

$$SI'_j = \max\{S_j - T_j, S_i : (i, j) \in \mathbb{A}\}.$$

Another way of defining a fixed solution is to let

$$S'_i = \min\{SI_i + T_i, SI_j : (i, j) \in \mathbb{A}\}.$$

We use one of the methods of fixing the solution in the implementation of the algorithm in Chapter 4.

Example 8. Consider the problem from Example 7. The first spanning tree relaxation on Figure 3-6 was constructed by removing arc (3,6) from the network. The solution to the relaxation is infeasible in the context of the general problem, because $S_3 = 12$, $SI_6 = 9$ and constraint $S_3 \leq SI_6$ does not hold. To fix the solution we increase SI_6 to be 12. Then the new solution is feasible and provides an upper bound on the optimal solution. The new solution is shown in Table 3.9. The cost of the new solution is 10354.72.

3.2.4 Algorithm

We conclude the development of an algorithm for general supply chain networks by

Algorithm 7 Branch and bound algorithm

function computeGenMinCost

```
1: tree  $\leftarrow$  createTree
2: lowerBound  $\leftarrow$  computeTreeMinCost(tree,  $\emptyset$ ,  $\emptyset$ ,  $\emptyset$ ,  $\emptyset$ )
3: upperBound  $\leftarrow$  fixTree(tree)
4: if lowerBound = upperBound then
5:   return lowerBound
6: end if
7: branch(1)
8: return upperBound
```

end function**procedure** *branch*(*k*)

```
9: for all valid paths  $P_{kj}$  do
10:   compute  $S_i$  and  $SI_i$  for all  $i \in P_{kj}$ 
11:   if all  $S_i$  and  $SI_i$  valid then
12:     set all  $S_i$  and  $SI_i$ 
13:     bound()
14:     unset all  $S_i$  and  $SI_i$ 
15:   end if
16: end for
```

end procedure**procedure** *bound*()

```
17: if all nodes have  $SI$  set then
18:   branchCost  $\leftarrow$  compute graph cost
19:   if branchCost < upperBound then
20:     upperBound  $\leftarrow$  branchCost
21:   end if
22: else
23:   tree  $\leftarrow$  createTree
24:   branchLowerBound  $\leftarrow$  computeTreeMinCost(tree,  $SI$ ,  $S$ ,  $\underline{SI}$ ,  $\bar{S}$ )
25:   branchUpperBound  $\leftarrow$  fixTree(tree)
26:   if branchUpperBound < upperBound then
27:     upperBound  $\leftarrow$  branchUpperBound
28:   end if
29:   if branchLowerBound < upperBound then
30:      $k \leftarrow$  next node without  $SI$  set
31:     branch( $k$ )
32:   end if
33: end if
```

end procedure

formally stating the algorithm in Algorithm 7. This is a branch and bound algorithm with the branching tree as described in section 3.2.1.

Here we comment on the notation used in Algorithm 7. In lines 1 to 6 of the algorithm, we compute the initial bounds on the optimal solution. We first create a spanning tree using procedure `createTree` (line 1). We have discussed ways to create the tree in section 3.2.2. Then we compute the optimal cost of the problem restricted to the tree network using Algorithm 5. Since at this point we do not have any service times or bounds set, we pass \emptyset as the service times and bounds to `computeTreeMinCost`. As discussed in section 3.2.3, we fix the tree solution using procedure `fixTree(tree)` to obtain a feasible solution and an upper bound. Then, if the initial upper bound is strictly greater than the lower bound, we proceed to the branching starting from node 1.

Procedure `branch(k)` computes branches for node k using paths. In line 9, we require path P_{kj} to be a valid path, where by valid we mean the conditions on the path described in section 2.2.3. The paths can be generated by Algorithm 6. Then, in line 10, we set S_i and SI_i along the path as specified in lemma 16 and corollary 5. In line 11, by valid S_i and SI_i we mean all the service times set by the path that satisfy constraints:

$$0 \leq S_i \leq \min\{M_i, \bar{S}_i\};$$

$$\underline{SI}_i \leq SI_i \leq M_i - T_i.$$

After we bound the branch, we unset SI_i and S_i along the path in line 14.

We comment on procedure `bound()`. Lines 18 to 21 correspond to the case when

all the nodes in the network have the inbound service times assigned. Then we just need to compute the cost of the branch in line 18. We compare the cost of the branch $branchCost$ and the current best solution $upperBound$ in lines 19 to 21.

If some inbound service times of the network are not assigned, we execute lines 23 to 32. We create a spanning tree as we discussed in section 3.2.2 and solve the spanning tree problem using Algorithm 5. In line 24, we pass assigned inbound and outbound service times and bounds to procedure $computeTreeMinCost$ of Algorithm 5. After computing the lower bound for the branch, we fix the solution from the spanning tree in line 25. Then we choose the best solution out of the upper bound of the branch and the current best solution. In line 29, if the lower bound of the branch is less than the current best solution, we go to the next node with the inbound service time unassigned. Because some of the inbound service times of nodes $i > k$ are set by the path, we first need to find the next node that does not have its inbound service time set. Then we branch using the inbound service times of this node (line 31).

The algorithm terminates with the optimal solution for the general network safety stock problem. The solution does not depend on which order we process the nodes in the network. However, in practice, it has been noticed that by using some orders of the nodes, on average, we can compute the optimal cost faster than using others. In Chapter 4 we show the performance of three ways of ordering the nodes. One order is random, i.e., we assign numbers $1, \dots, N$ to the nodes at random. The second order is the order of decreasing $h_i * \sigma_i$. This order tries to process the most expensive nodes first.

Finally, the third order is the layer order. We define the layer order as follows.

We first define the layers in the network. Node i belongs to layer L_k if the longest path from node i to any demand node $P_{ij} = (i, \dots, j)$ has $k + 1$ nodes. By definition, demand nodes belong to layer L_1 . Then, the layer order of the nodes in the network assigns numbers $1, 2, \dots, l_1$ to the nodes of layer L_1 , $l_1 + 1, \dots, l_2$ - to the nodes from layer L_2 , $l_{k-1} + 1, \dots, l_k$ - to the nodes from layer L_k . That is, the nodes from layer L_k have greater numbers than the nodes from layer L_i , $i < k$.

As we will show in the Chapter 4, the layer order gives the best average computation time among the three order methods described.

3.3 Two-layer networks

In this section, we consider a special case of supply chain networks that have only two subsets of nodes. The first layer is the set of components, and the second is the set of demand nodes. Arcs are only possible between the layers with each arc going from a component to a demand node. Let the number of components be m and the number of demand nodes be l .

We can give an interpretation for this kind of networks. The components are inputs to the assemblies represented by demand nodes. An arc between a component and an assembly means that the component is needed in the assembly.

Such networks were considered in Humair and Willems [2003] and called clusters of commonality. Humair and Willems [2003] study a general network that consists of several clusters of commonality, which are linked together into a spanning tree. That is, when each cluster of commonality is assumed as a single, aggregated node,

the resulting network is a spanning tree. The biggest challenge, however, is to find a solution for a general two-layer network. Therefore, here, we only concentrate on one cluster.

One approach for solving the two-layer network problems is to apply the general algorithm developed in section 3.2. As in the general algorithm, we can branch on the inbound service times of the nodes in the network. Those values for node i are computed according to the paths that start in node i and end in the other nodes of the network. This method, however, does not take advantage of the structure of the network.

Using the general method in the two-layer case unnecessarily creates too many branches in the branching tree. Indeed, the number of paths that can be generated from one node to another can be big and grows as the number of arcs increases. On the other hand, we will show that for two-layer networks the inbound service time of a demand node j is such that $SI_j \leq \max\{T_i, i \in \mathbb{C}\}$ and for each component i , $SI_i = 0$. Therefore, we should be able to eliminate unnecessary values and paths.

3.3.1 Optimality conditions

For the algorithm to be the most efficient, we need to characterize the set of optimal solutions. From the general case we know (see observation 1 from section 2.2.2) that there always exists an optimal solution such that

- $S_j = s_j, j \in \mathbb{D}$;
- $SI_i = 0, i \in \mathbb{C}$;

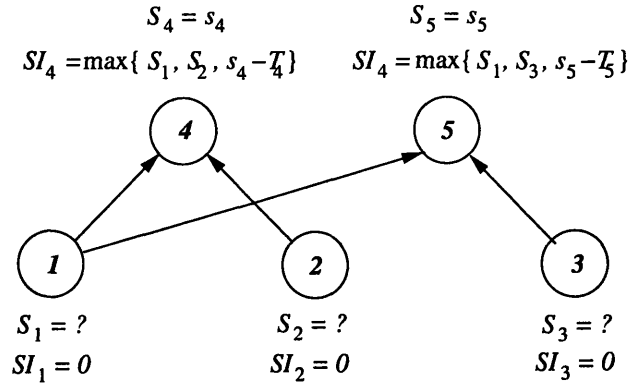


Figure 3-7: The objective function depends only on the outbound service times S_1, S_2, S_3 .

- $SI_j = \max\{s_j - T_j; S_i, \text{ for } i : (i, j) \in \mathbb{A}\}$.

Therefore, we can consider the objective function as a function of outbound service times $\{S_i, i \in \mathbb{C}\}$. This observation is illustrated in Figure 3-7, where s_4 and s_5 are known maximum service times for nodes 4 and 5. To specify the values of S_i for each node i , we prove the following lemma.

Lemma 18. *If j is a component in a two-layer network, there are four possibilities for the optimal service time S_j :*

1. $S_j = 0$;
2. $S_j = T_j$;
3. $S_j = T_k$ for $T_k < T_j$;
4. $S_j = s_i - T_i$ for $i \in \mathbb{D}$ and $0 \leq s_i - T_i \leq T_j$.

Moreover, if $S_j = T_k, T_k < T_j$, then $S_k = T_k$.

Proof. Given that we set $SI_j = 0$, the only constraints on the outbound service time of a component j are:

$$S_j \leq T_j; \quad (3.10)$$

$$S_j \leq SI_i, \text{ for } (j, i) \in \mathbb{A} \quad (3.11)$$

$$S_j \geq 0. \quad (3.12)$$

By the problem formulation, the safety stock function in node j decreases as S_j increases. Therefore, S_j has to be maximum possible allowed by the constraints of the problem, i.e.,

$$S_j = \min\{T_j, SI_i : (j, i) \in \mathbb{A}\}.$$

From the general case we know that there exists an optimal solution such that SI_i , $i \in \mathbb{D}$ is determined by some path in the network. Since the network has only two layers of nodes, any path alternates between the nodes from one layer and the other.

Consider a path P_{ik} that starts in node i , $i \in \mathbb{D}$. If node $k \in \mathbb{C}$, then $SI_k = 0$, and by the definition of R_{ik}

$$SI_i = R_{ik} + SI_k = T_{i_1} - T_{i_1} + T_{i_3} - T_{i_3} + \dots + T_{i_{p-2}} - T_{i_{p-2}} + T_k = T_k,$$

where $i, i_1, i_2, \dots, i_{p-1}, k$ are the nodes in the path. Therefore, if $T_k \leq T_j$, T_k is a possible value for S_j . We also notice that $SI_{i_{p-1}} = S_k = T_k$ by lemma 15. Therefore, $S_j = T_k$ only if $S_k = T_k$.

Suppose $k \in \mathbb{D}$. Then since we stopped the path in the demand node, $SI_k = \max\{0, s_k - T_k\}$ and

$$SI_i = R_{ik} + SI_k = T_{i_1} - T_{i_1} + T_{i_3} - T_{i_3} + \dots + T_{i_{p-1}} - T_{i_{p-1}} + SI_k = \max\{0, s_k - T_k\}.$$

Therefore, if $s_k - T_k \geq 0$ and $s_k - T_k \leq T_j$, $s_k - T_k$ is a possible value for S_j . This concludes the proof of the lemma.

□

Lemma 18 shows how to simplify the calculation of the possible values using paths. Because of the structure of the network, the possible values can be determined without computing the paths.

3.3.2 Algorithm

In this section, we describe a branch and bound algorithm for the two-layer problem. The algorithm uses lower and upper bounds developed in the later sections of this chapter.

To specify the branching tree, we use the properties of an optimal solution. We know that the objective function can be presented as a function of outbound service times for the components only. By lemma 18, the outbound service times can take only a finite number of values in an optimal solution. Therefore, the most natural branching step is to try all possible values of S_j . However, we can simplify the branching tree using lemma 18. In particular, we know that $S_j = T_k < T_j$ only if $S_k = T_k$. Therefore, we can eliminate the branch with $S_j = T_k < T_j$ if $S_k \neq T_k$.

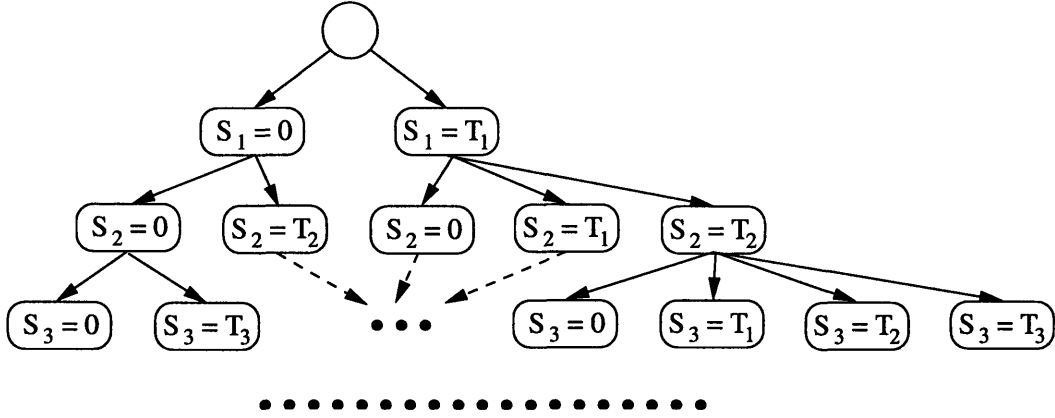


Figure 3-8: Branching tree for a two-layer problem with $s_i \leq T_i, i \in \mathbb{D}$.

Taking advantage of lemma 18, we can present the branching tree as follows. First, we order the components in the order of increasing lead-times:

$$T_1 \dots, T_m.$$

At the first level of branching we let $S_1 = 0, S_1 = T_1$ or $S_1 = s_r - T_r$ for all $r \in \mathbb{D}$ such that $T_1 \geq s_r - T_r \geq 0$. At the i th level of branching, we let $S_i = \{0, T_1, \dots, T_i\}$, making sure that $S_i = T_k < T_i$ only if $S_k = T_k$ earlier in the branching tree, and we let $S_i = s_r - T_r$ for all $r \in \mathbb{D}$ such that $0 \leq s_r - T_r \leq T_i$. Figure 3-8 shows an example of a branching tree with $s_k \leq T_k$ for all $k \in \mathbb{D}$. We can search the tree using depth first search or breadth first search.

The number of branches grows exponentially with the number of the components. Even if each outbound service time $S_i, i \in \mathbb{C}$, took only 2 possible values 0 and T_i , at each level of branching there would be twice as many branches as in the previous level. Therefore, the number of branches is at least 2^m , where m is the number

of components. In the worst case, the algorithm is no better than the complete enumeration of all possible solutions. Nevertheless, the algorithm terminates with an optimal solution (Korte and Vygen [2002]) and benefits from the effective bounds we use.

3.3.3 Lower Bounds

Two-layer network algorithm uses the same lower bounds as the general network algorithm. For each branch, we construct a spanning tree in the network and apply the tree algorithm from section 3.1. The creation of the spanning trees was discussed in section 3.2.2 and we omit it here.

3.3.4 Upper Bounds

To obtain an upper bound on the optimal solution of the problem, we can fix the tree solution, as described in section 3.2.3

The gap between the fixed solution and the optimal cost can be very large. The size of the gap can depend on the number of removed constraints that are violated by the spanning tree solution.

Example 9. *To illustrate the statement of the last paragraph, consider the network shown on Figure 3-9 with parameters specified in Table 4.26. In this and all other examples we use*

$$D(F) = k\sigma\sqrt{F} + F\mu$$

as the upper bound functions of the demand. In this example, $k = 1$.

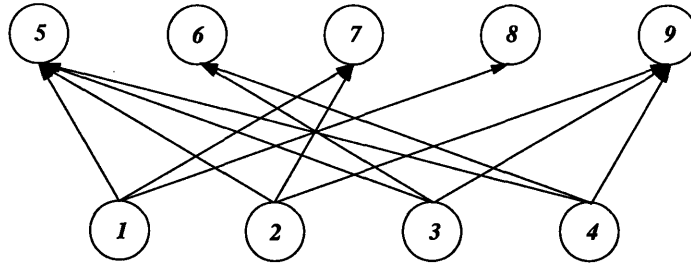


Figure 3-9: Two-layer network.

nodes	1	2	3	4	5	6	7	8	9
T	3	5	6	7	10	4	11	3	5
s					4	3	8	2	4
h	4	4	2	1	12	20	15	5	17
μ	20	25	16	30	30	17	25	16	21
σ	15	21	2	20	4	16	23	5	20

Table 3.10: Parameters of the network shown on Figure 3-9

Suppose the lower bound algorithm selects the tree presented on Figure 3-10. To construct the tree the following arcs have been removed:

$$(2, 7), (2, 9), (4, 6), (4, 9).$$

Table 3.11 shows an optimal solution to the relaxed problem. The correspondent optimal cost is 1569.3. The solution is infeasible in the original problem, since all the constraints on the removed arcs are violated.

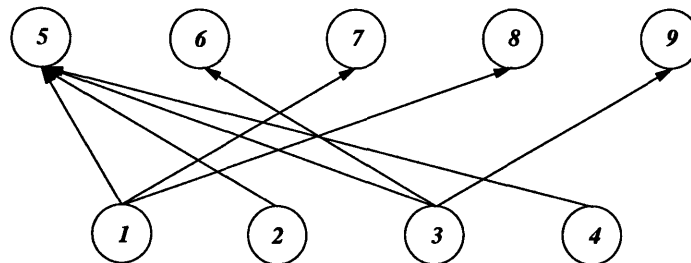


Figure 3-10: A tree chosen by the lower bound algorithm

nodes	1	2	3	4	5	6	7	8	9
<i>S</i>	0	5	0	7	4	3	8	2	4
<i>SI</i>	0	0	0	0	7	0	0	0	0

Table 3.11: An optimal solution to the problem shown on Figure 3-10

nodes	1	2	3	4	5	6	7	8	9
<i>S</i>	0	5	0	7	4	3	8	2	4
<i>SI</i>	0	0	0	0	7	7	5	0	7

Table 3.12: The fixed solution for Figure 3-10 and Table 3.11

We fix the solution by adjusting the inbound service times for the demand nodes (Table 3.12). The new solution is feasible with the cost 3154.4. The optimal cost for the problem is 1754.6, which is 60% of the cost of the fixed solution.

Suppose, however, the lower bound algorithm selects the tree shown on Figure 3-11.

The removed arcs are

$$(2, 7), (3, 9), (4, 6), (4, 9).$$

The cost of the new relaxation is 1754.6 (Table 3.13). Furthermore, the solution to the relaxed problem satisfies the removed constraints. Therefore, the cost of the relaxation equals the cost of the fixed solution, the optimal solution.

This example shows that depending on the tree, the relaxation can some times

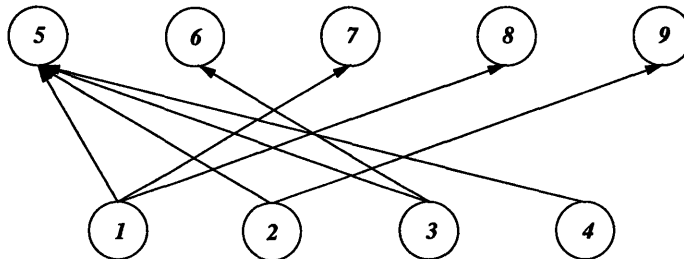


Figure 3-11: The second tree chosen by the lower bound algorithm

nodes	1	2	3	4	5	6	7	8	9
S	0	0	0	0	4	3	8	2	4
SI	0	0	0	0	0	0	0	0	0

Table 3.13: An optimal solution for the problem shown on Figure 3-11

yield very poor lower and upper bounds on the optimal solution, while other times, the relaxation gives the optimal solution.

Here, we provide two simple algorithms for tightening the upper bound given by a feasible solution to problem \mathcal{P} . First algorithm searches for the best solution among the solutions that satisfy a given ordering of the outbound service times for components. The second algorithm searches for the best solution among the solutions that satisfy a given ordering of the inbound service times for the demand nodes. We first describe the former algorithm.

Suppose we are given a priori an order for the outbound service times for the components:

$$S_1 \leq S_2 \leq \dots \leq S_m.$$

For example, we might take the order from the fixed solution. Note that we may renumber the components. We call the problem an ordered problem. In this problem we look for the best solution of the problem \mathcal{P} among the solutions that satisfy this given order of the outbound service times for the components.

As before, we use the following notation:

$$SS_i(S_i, SI_i) = h_i(D_i(SI_i + T_i - S_i) - \mu_i(SI_i + T_i - S_i));$$

We use notation $Sset[i]$ for the set of candidate values for S_i for a component i and

$$Sset[i] = \{0; T_j \leq T_i, j \in \mathbb{C}; 0 \leq s_r - T_r \leq T_i, r \in \mathbb{D}\}.$$

Then we can solve the problem optimally by the following dynamic program.

Outbound service time upper bound algorithm

I. Define subsets of demand nodes:

1. Let $L = \{1, \dots, l\}$;
2. For $i := m$ to 1

$$R_i = \{j : (i, j) \in \mathbb{A}, j \in L\};$$

$$L = L \setminus R_i.$$

II. Find the optimal solution of the ordered problem:

1. For $k := 1$ to m

2. If $k = 1$,

$$f_1(S_1) = SS_1(S_1, 0) + \sum_{i \in R_1} SS_i(s_i, \max\{S_1, s_i - T_i\}),$$

$$S_1 \in Sset[i];$$

3. If $k \neq 1$,

$$f_k(S_k) = SS_k(S_k, 0) + \sum_{i \in R_k} SS_i(s_i, \max\{S_k, s_i - T_i\}) + \min_S \{f_{k-1}(S), S \in$$

$$Sset[k-1], S \leq S_k\},$$

$$S_k \in Sset[k]$$

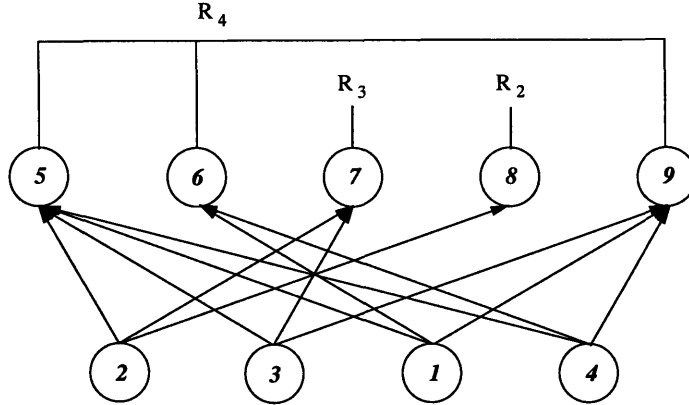


Figure 3-12: Subsets

2. Minimize $f_m(S_m)$, $S_m \in Sset[m]$ to obtain the optimal objective function value.

The first procedure defines subsets of the demand nodes R_i . For each node j in subset R_i ($i, j \in \mathbb{A}$, node i determines SI_j).

The solution obtained is optimal by the principle of optimality for dynamic programming.

As in the case of the spanning tree algorithm, functions $f_i()$ have a meaning in terms of the cost of safety stock. For each component i and outbound service time S , we say $f_i(S)$ is the minimum cost of safety stock of a subset of nodes:

$$\{i, R_i, i - 1, R_{i-1}, \dots, 1, R_1\}.$$

The algorithm gives an optimal solution to the problem \mathcal{P} if we have the right ordering of the outbound service times for the components. It is not clear how best to set the order and further research can be done in this direction. Nevertheless, for the

purpose of establishing an upper bound any order of the service times can be used.

Example 10. *In this example, we continue working with the problem started in Example 9. We solved the problem presented by the network on Figure 3-10 and obtained the optimal solution (Table 3.11). To apply the upper bound algorithm, we renumber the components in the order of increasing outbound service times. Now, node 1 has number 2, node 2 has number 3, node 3 has number 1, and node 4 remains 4.*

Using Part I of the algorithm, we divide the set of demand nodes into 4 subsets (Figure 3-12):

$$R_1 = \emptyset,$$

$$R_2 = \{8\},$$

$$R_3 = \{7\},$$

$$R_4 = \{5, 6, 9\}.$$

Executing Part II of the algorithm, we find the optimal solution given this order of the outbound service times of the components. The solution is the same as that of presented in Table 3.13. Since the solution is the same for the upper and the lower bound obtained from the tree on Figure 3-11, it is an optimal solution to the original problem.

An observation about this example is that all the outbound service times are equal in the optimal solution. Therefore, no matter what order of outbound service times we use as an input to the upper bound algorithm the outcome is always an optimal solution.

The second upper bound algorithm is similar to the outbound service time algo-

rithm, but assumes an order on the inbound service times of the demand nodes:

$$SI_1 \geq \dots \geq SI_l$$

where $1, \dots, l$ are demand nodes.

We use notation $SIset[i]$ for the set of possible inbound service times of demand node i and

$$SIset[i] = \{\max\{0, s_i - T_i\}; T_j \geq s_i - T_i, j \in \mathbb{C}; s_r - T_r \geq \max\{0, s_i - T_i\}, r \in \mathbb{D}\}.$$

Then the algorithm is as follows:

Inbound service time upper bound algorithm

I. Define subsets of components:

1. Let $L = \{1, \dots, l\}$;
2. For $i := l$ to 1
 - $R_i = \{j : (j, i) \in \mathbb{A}, j \in L\}$;
 - $L = L \setminus R_i$.

II. Find the optimal solution of the ordered problem:

1. For $k := 1$ to l
 2. If $k = 1$,
 - $g_1(SI_1) = SS_1(s_1, SI_1) + \sum_{i \in R_1} SS_i(\min\{SI_1, T_i\}, 0)$,
 - $SI_1 \in SIset[1]$;

3. If $k \neq 1$,

$$g_k(S_k) = SS_k(s_k, SI_k) + \sum_{i \in R_k} SS_i(\min\{SI_k, T_i\}, 0) + \min_{SI} \{g_{k-1}(SI), SI \in SIset[k-1], SI \geq SI_k\},$$

$$SI_k \in SIset[k]$$

2. Minimize $g_l(SI_l)$, $SI_l \in SIset[l]$ to obtain the optimal objective function value.

The solution obtained is optimal by the principle of optimality for dynamic programming.

We can alternate between the inbound service time and outbound service time algorithms to achieve a better upper bound. Suppose we first order the outbound service times of the components. Then we apply the outbound service time algorithm with this order to get a solution z_1 . The solution gives an order to the inbound service times of the demand nodes. We can apply the inbound service time algorithm for this order with the cost z_2 . Then we apply the outbound service time algorithm again. We continue, by alternating the two algorithms.

We notice, that $z_1 \geq z_2 \geq \dots$. Indeed, suppose the inbound service time algorithm computed cost z_i and then the outbound service time algorithm computed cost z_{i+1} . Since both solutions have the same order of outbound service times, $z_i \geq z_{i+1}$.

Alternating the algorithms as described is a finite procedure. Since all $z_i \geq z$, where z is the optimal cost of the problem, and the number of possible service times is finite, the procedure is finite. Therefore, the procedure terminates with the order of inbound and outbound service times that can not be improved by either upper

bound algorithm.

Chapter 4

Computations

In this chapter we present computational results of implementing the algorithms from Chapter 3. Our goals here are to demonstrate the performance of the algorithms. By doing a series of experiments we show how to choose the parameters of the algorithm to obtain the best performance.

The algorithms have been implemented in SUN Java 1.4 and the instances of the problem were solved on Pentium IV 2.8 GHz with RAM 512 MB desktop running Windows XP. We measured the CPU time in milliseconds spent on solving the instances. The time of generating instances is not included in the the results.

Now, we describe the generation of the safety stock problem instances on a network with N nodes and M arcs.

- We first present the network as a node-node adjacency matrix $A = [a_{ij}]$. Element a_{ij} of the matrix is 0 if the network does not have arc (i, j) . If $a_{ij} = 1$, then the network has arc $(i, j) \in \mathbb{A}$. We let $a_{ij} = 1$ with probability $\frac{2M}{N(N-1)}$. To

create an acyclic network, we only allow arcs (i, j) such that $i < j$. If the resulting network is disconnected, we connect the disconnected pieces by placing an arc from a node with smaller number to a node with a bigger number such that the arc connects two disconnected parts of the network. We only take the networks that have exactly M arcs.

- We generate lead time T_i as a random variable uniformly distributed between 0 and 100, unless specified otherwise. A guaranteed service time s_i for a demand node i is a random variable uniformly distributed between 0 and $2T_i$.
- As demand bound function for node i , we use

$$D_i(\tau) = k\sigma_i\sqrt{\tau} + \mu_i\tau.$$

μ_i and τ_i are defined in section 1.1.2. k is a parameter. We assume here $k = 1$.

- We use the following safety stock function in node i :

$$SS_i(\tau) = D_i(\tau) - \mu_i\tau = k\sigma_i\sqrt{\tau}.$$

- The cost of the safety stock in node i is

$$kh_i\sigma_i\sqrt{\tau} = kH_i\sqrt{\tau}.$$

We generate H_i as a random variable uniformly distributed between 0 and 100. We justify this generation of the cost function as follows. In a supply

chain, per unit holding cost h_i increases from components to demand nodes. At the same time, the standard deviation of demand σ_i typically increases from demand nodes to components. Therefore, it is reasonable to let $H_i = h_i\sigma_i$ to be uniformly distributed in some interval.

- We set the limit of 10 minutes on the running time of the algorithm for one problem instance. If necessary, we report the number of instances that exceed the time limit.

We have designed a series of experiments to find the best settings of the algorithm. Through Chapter 3, we discussed several parameters of the algorithms that have to be explored computationally. We summarize the parameters here.

1. **Node order.** In section 3.2, we discussed three types of node order which is used by the branching tree. The types are random, cost and layer. We repeat here, that the random order processes the nodes at random. The cost order numbers the nodes in the order of decreasing $H_i = \sigma_i * h_i$. The layer order processes the nodes in layers starting from the layer of the demand nodes.
2. **Tree type.** In section 3.2.2 we discussed that we compute lower bounds at each branch by a spanning-tree relaxation of the general-network problem. There are many way of creating a spanning tree for the network. We generate the trees in three ways.
 - **Same trees.** Suppose, at each branch we want to generate k lower bounds to obtain a better lower bound of the branch. The first way of generating

the spanning trees is to generate k random spanning trees and use these k trees at each branching point. We call this the same tree method.

- **Smart trees.** The second method is to generate k smart trees as discussed in section 3.2.2. We repeat here that we create a smart tree by first inserting arcs $(i, j) \in \mathbb{A}$ with unassigned S_i and SI_j into the tree. Only then, if necessary, we add the arcs with S_i or SI_j assigned. We call this the smart tree method. We note that because at each branching point different service times are assigned, the trees may be different.
- **Random trees.** The third method is to generate k random trees and we call this the random tree method. We generate k different spanning trees at each point.

3. **Number of initial bounds.** The number of initial bounds is the number of lower bounds and the number of corresponding upper bounds that are calculated by fixing the lower bound solution. The bounds were discussed in sections 3.2.2 and 3.2.3. We compute the initial bounds before the algorithm starts the branching process. These bounds are the bounds of the optimal solution of the general problem.

4. **Number of bounds per branching point (BBP).** BBP is the number of lower and corresponding upper bounds that are computed at each branching point. Each bound at a branching point is computed using its own tree, as described above (2).

5. **Number of global bounds per branching point (GBBP).** At each branching point we can compute additional bounds for the optimal solution to the general problem. We note here, that these bounds are the same as initial bounds, but computed after the branching process is started. The difference is that if the algorithm has to compute k initial bounds, it will compute all of the k bounds before starting the branching process. On the other hand, the algorithm can have less than k branches to terminate with an optimal solution. Therefore, in this case it computes less than k global bounds.

6. **Tolerance limit.** If the tolerance limit is 0, then the algorithm terminates with an optimal solution. For tolerance limit $x\% > 0$, the algorithm terminates with a solution such that the difference between the best upper bound and the best lower bound is $x\%$ of the best upper bound. We achieve this tolerance limit by bounding each branch allowing $x\%$ gap between the best lower and the best upper bound for the branch. Since an optimal solution belongs to one of the branches, the optimal solution is within $x\%$ of the best upper bound.

For each experiment, we generate a number of random instances and specify the setting of the experiment. We summarize the settings in a table similar to Table 4.1. Table 4.1, shows the parameters discussed above.

Experiment 1. Spanning tree computations

In section 3.1, we showed how to improve the performance of the algorithm from Graves and Willems [2000] for safety stock problem for a supply chain modeled as a spanning tree. The algorithm described in the paper, is a pseudo-polynomial

nodes	N
arcs	M
node order	cost, layer, random
tree type	same, smart, random
initial bounds	
bounds per branching point	
global bounds per branching point	
tolerance limit	
number of instances	

Table 4.1: Summary of the experiment settings.

nodes	20
arcs	25
node order	random
tree type	random
initial bounds	1
bounds per branching point	1
global bounds per branching point	0
error	0
number of instances	100

Table 4.2: Settings for Experiment 1.

time algorithm. The new algorithm is polynomial.

Here, we show the importance of having a polynomial time algorithm for the tree networks. We use the spanning tree networks to obtain the lower bounds for the general problem (section 3.2.2). Since the general branch and bound algorithm uses the tree algorithm repeatedly, it is important to be able to solve the problem for a spanning tree fast.

We demonstrate the difference between the polynomial and pseudo-polynomial time algorithms with the following example. We solve a general network problem by branch and bound algorithm from section 3.2, first using the pseudo-polynomial algorithm and then using the polynomial algorithm. We generate random networks with 20 nodes and 25 arcs. To see the difference, we let the lead time at each node be a random variable uniformly distributed in the interval $[0, T_{max}]$. We vary T_{max} from 10 to 360. For each value of T_{max} , we generate 100 instances of the safety stock problem and solve them first by the branch and bound algorithm with polynomially computed bounds. Then we solve the same instances by the same algorithm, but with pseudo-polynomially computed bounds.

Table 4.2 shows the parameters of the experiments.

Table 4.3 presents the results of the computations. The first column is T_{max} . The second and the third columns have the average times of solving the general network problem (Table 4.2) with the lower bounds computed using polynomial and pseudo-polynomial algorithms respectively.

Figure 4-1 shows the average time of solving a general network safety stock problem with the bounds computed by the polynomial time spanning tree algorithm. The

Max lead time	Polynomial	Pseudo-polynomial
10	88	100
60	82	670
110	83	1938
160	83	3734
210	65	6483
260	81	9475
310	81	13309
360	62	17917

Table 4.3: Average time per instance in milliseconds for Experiment 1.

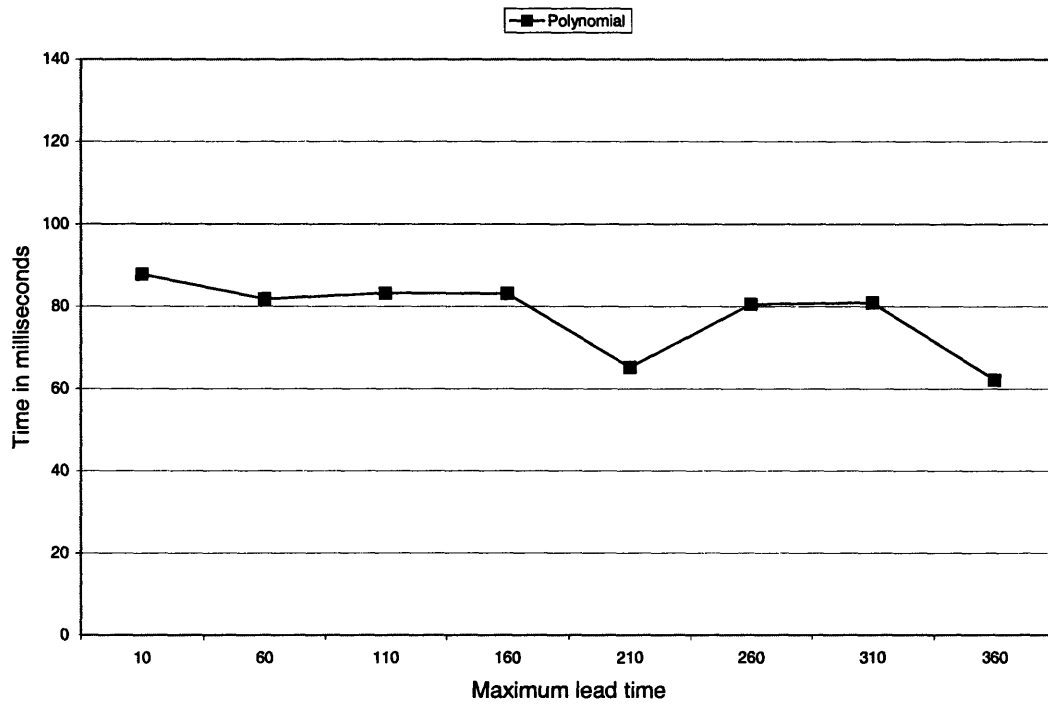


Figure 4-1: Average time of solving a general network problem with 20 nodes and 25 arcs using polynomial bounds.

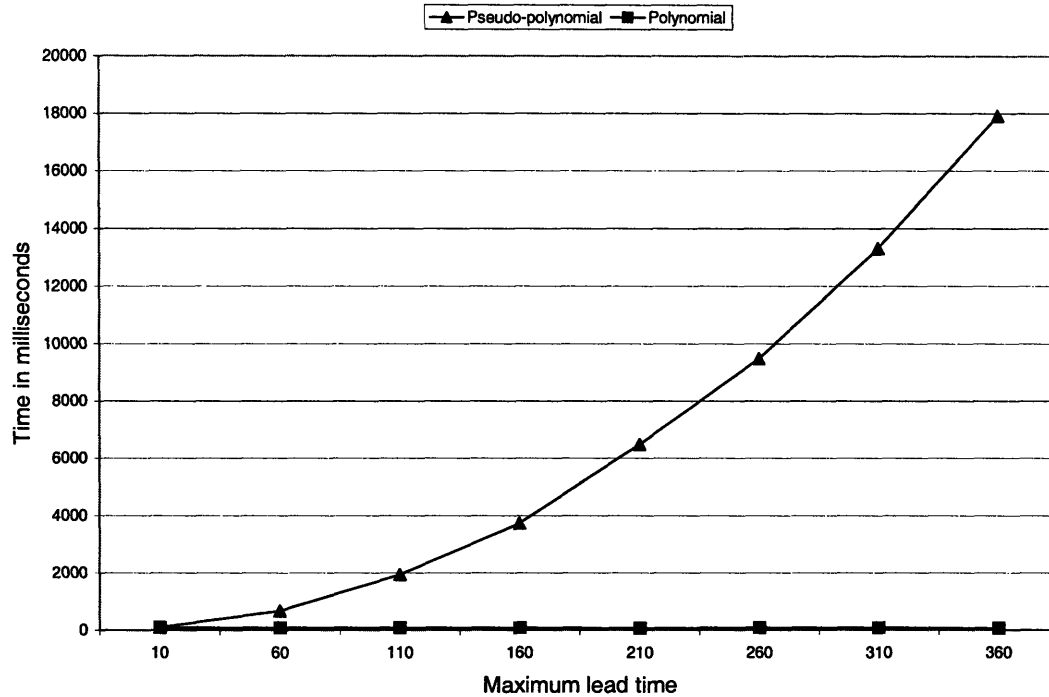


Figure 4-2: Average time of solving a general network problem with 20 nodes and 25 arcs using polynomial and pseudo-polynomial bounds.

average time is presented as a function of $Tmax$. Figure 4-2 shows the average time of solving a general network safety stock problem with the bounds computed by the pseudo-polynomial time spanning tree algorithm. For a comparison, the results of the general algorithm with polynomial bounds are depicted on the same figure.

From the computational experiments, we see that the average time of running the general algorithm with the polynomial bounds stays approximately the same for different $Tmax$. At the same time, the average time of running general time algorithm with the pseudo-polynomial bounds increases when $Tmax$ increases.

Experiment 2. Node order

In this experiment we test different orders of nodes to use in the branch and bound algorithm. We discussed three ways to order the nodes in section 3.2 – a

random order, a cost order and a layer order. Here, we test each ordering to find the best way.

We set up the experiments as follows. We randomly generate 1000 graphs with 30 nodes and 38 arcs. Then we apply the branch and bound algorithm for the general networks with the three types of node order. We choose the settings for the algorithm as shown in Table 4.14.

As we see in Table 4.5, the average time as well as the average number of branching points is smallest if we use the layer order. Therefore, in the remaining experiments, we use the layer order, as we conjecture that it is the best order among the three ordering schemes.

Experiment 3. Type of tree for the lower bounds

In this experiment, we examine what type of tree is the best to use for creating lower bounds. In section 3.2.2 we described two possible types of tree – smart tree and random tree. Here, we use both for the lower bound computation and compare the average time of solving the general safety stock problem.

The settings of the experiment are specified in Table 4.6. We generate 500 random networks with 20 nodes and 30 arcs. Each of the 500 safety stock problems on the networks is first solved with the lower and upper bounds generated by the random trees, and then with bounds generated by the smart trees.

To identify a better method between the two, we vary the number of lower bounds computed per branching point (BBP). We start with 1 bound, and consequently, with one tree per branching point. We repeat the experiment with $k = 2, \dots, 19$, where

we create up to k trees per branching point, and compute up to k lower bounds for the branch. If any of the lower bounds equals or exceeds the current best solution of the problem, the branch is not explored further.

In addition to the two methods of generating the spanning trees, we use another method. We call the tree generation in this method as same tree generation. By the same tree we mean, that for each instance of the problem, we first generate k random spanning trees, where k is the maximum number of lower bounds per branching point. The only parameters that are different at each branching point are the set service times and the bounds on the service times.

It is interesting to compare the first two methods to this method, because in this method the algorithm spends time generating the k spanning trees only once in the beginning. At the same time, the other two methods must spend time to generate new trees at each branching point.

The results of the computations are presented in Table 4.7 and Figure 4-3. The average time is presented as a function of bounds per branching point (BBP). We see, that the random tree method performs the best in terms of average running time compared to the other two methods.

Table 4.8 and Figure 4-4 show the average number of branching points generated by the algorithm when the algorithm used the three methods of tree creation. We observe that all three methods require fewer branching points as BBP increase. This seems reasonable, since k BBP generate no worse bounds than $l < k$ BBP.

From Table 4.9 and Figure 4-5 we can explain the difference in the average time between the three methods. We see, that the average total number of computed

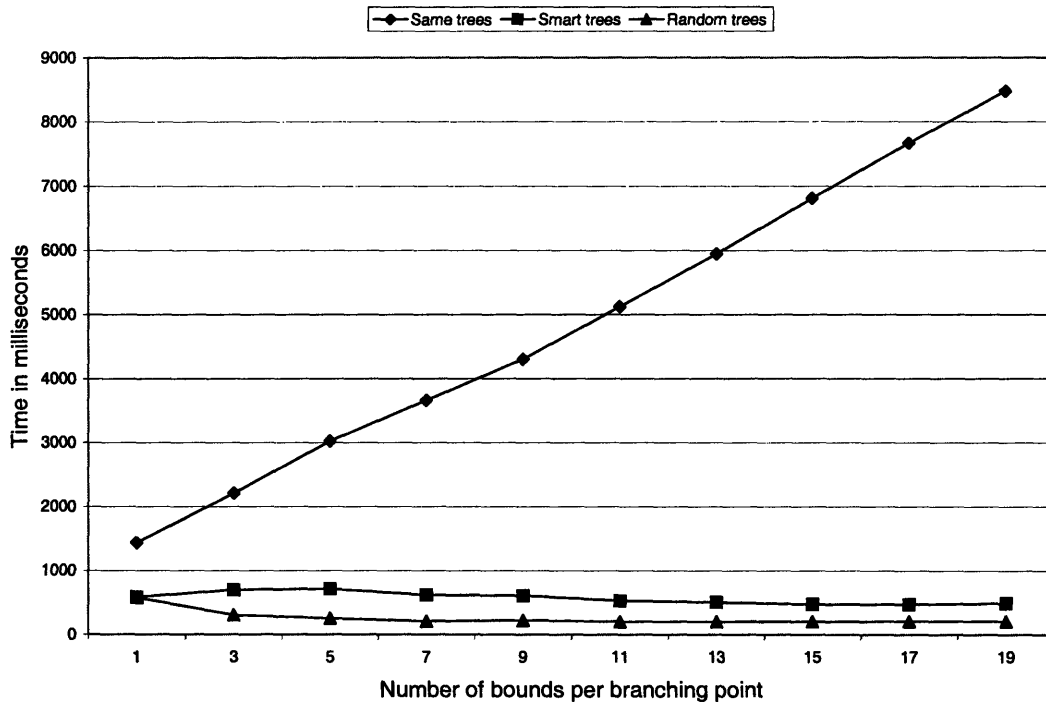


Figure 4-3: Average time in milliseconds for Experiment 3 to choose same, smart or random trees for the lower bounds.

bounds increases for the same tree method. The random tree method computes the least number of lower bounds among the three.

Intuitively, the result is expected. The same trees do not give lower bounds that differ much from branching point to branching point. Similarly, the way the smart trees are generated, does not give too much variability either. Indeed, to create a smart tree, the method first takes arcs $(i, j) \in A$ such that $S I_j$ and S_i are not set. Only then, it considers the arcs with some of the service times set. On the other hand, the random tree method creates many different trees and has a better chance of generating better bounds.

Experiment 4. Initial bounds vs. bounds per branching point

In this experiment, we solve the problem for different combinations of number of

nodes	30
arcs	38
node order	cost, layer, random
tree type	random
initial bounds	10
bounds per branching point	10
global bounds per branching point	0
tolerance limit	0
number of instances	1000

Table 4.4: Settings for Experiment 2 to choose random, cost or layer node order.

	Cost	Layer	Random
Time	2387	927	3108
Branching points	14264	6793	13873

Table 4.5: Average time in milliseconds and average number of branching points for Experiment 2 to choose random, cost or layer order.

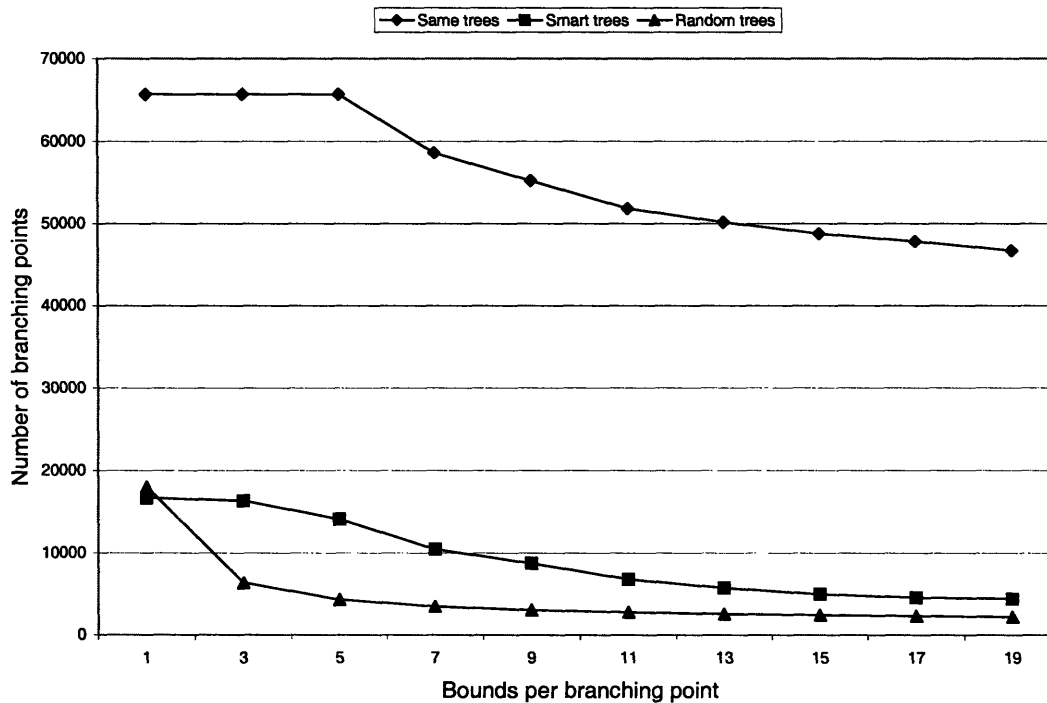


Figure 4-4: Average number of branching points for Experiment 3 to choose same, smart or random trees for the lower bounds.

nodes	20
arcs	30
node order	layer
tree type	same, smart, random
initial bounds	1
bounds per branching point	1-19
global bounds per branching point	0
tolerance limit	0
number of instances	500

Table 4.6: Settings for Experiment 3 to choose same, smart or random tree for lower bounds.

BBP	Same trees	Smart trees	Random trees
1	1438	585	577
3	2212	699	306
5	3025	716	253
7	3660	617	209
9	4300	611	223
11	5124	531	203
13	5947	508	203
15	6812	477	206
17	7668	475	209
19	8476	491	210

Table 4.7: Average time per instance in milliseconds for Experiment 3 to choose same, smart or random trees for the lower bounds.

initial bounds and bounds per branching point. We solve 100 instances of the general problem with 20 nodes and 30 arcs. We vary the number of initial bounds between 10 and 200. For each value of initial bounds, we vary the number of bounds per branching point between 1 and 20. The average running time of the algorithm is summarized in Table 4.11.

In Figure 4-6, we plot the average time as a function of bounds per branching point for different numbers of initial bounds. We notice, that the best value of bounds per branching point does not depend on the number of initial bounds. Any number of

BBP	Same trees	Smart trees	Random trees
1	65707	16696	18006
3	65707	16333	6396
5	65707	14103	4340
7	58612	10469	3501
9	55231	8749	3063
11	51840	6816	2792
13	50154	5746	2585
15	48777	4993	2435
17	47833	4570	2315
19	46703	4427	2197

Table 4.8: Average number of branching points for Experiment 3 to choose same, smart or random trees for the lower bounds.

BBP	Same trees	Smart trees	Random trees
1	65708	16697	18007
3	100089	19511	8274
5	134470	19403	6373
7	147238	16148	5477
9	163784	15029	5085
11	176218	12589	4863
13	192615	11424	4696
15	208547	10457	4587
17	225494	10140	4511
19	240199	10308	4406

Table 4.9: Average total number of computed lower bounds for Experiment 3 to choose same, smart or random trees for the lower bounds.

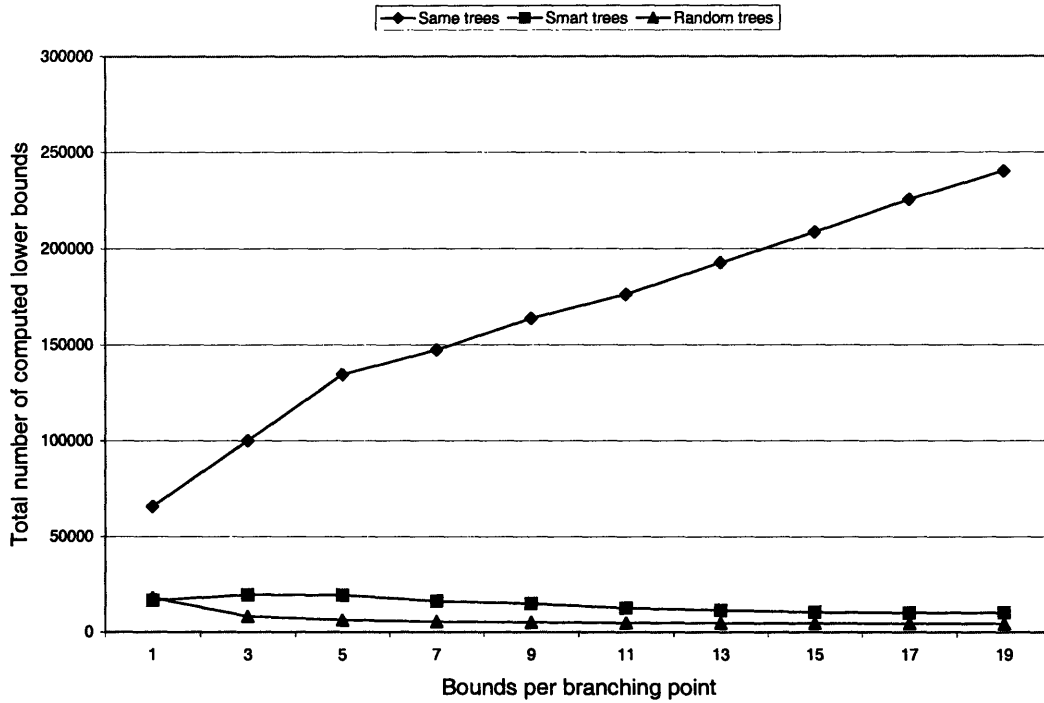


Figure 4-5: Average number of generated trees for Experiment 3 to choose same, smart or random trees for the lower bounds.

bounds per branching point between 10 and 20 on average gives the best running time.

In Figure 4-7, we plot the average time as a function of initial bounds for different numbers of bounds per branching point. We notice, that the best value of initial bounds does not depend on the number of bounds per branching point. Any number of initial bounds between 30 and 200 on average gives the best running time.

As a conclusion, we conjecture that the two parameters are independent. In the subsequent experiments, we will treat them as independent and optimize them separately.

Experiment 5. Initial bounds

In this experiment we show that best results for the number of initial bounds

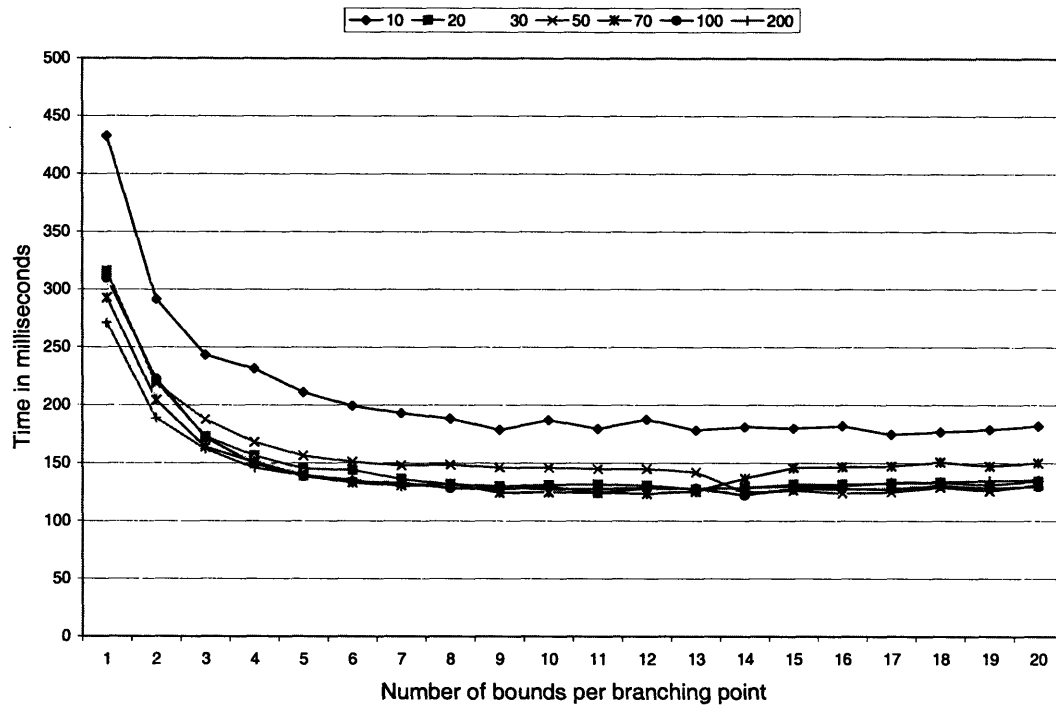


Figure 4-6: Average time in milliseconds for solving a problem with 20 nodes and 30 arcs as a function of the number of bounds per branching point, for different numbers of initial bounds in Experiment 4.

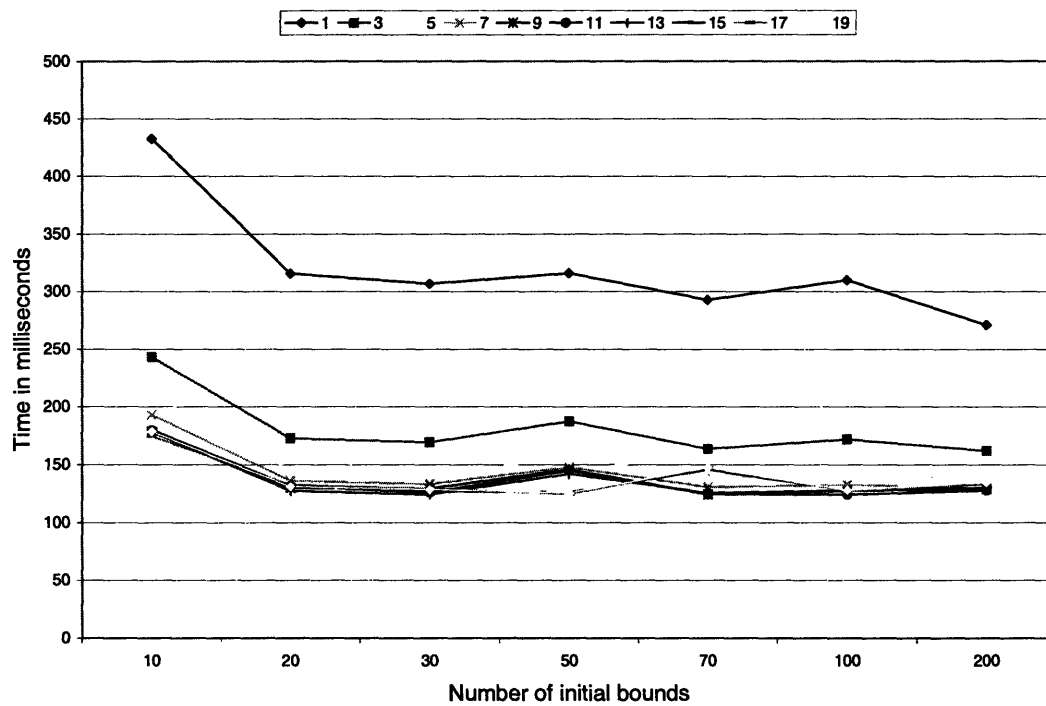


Figure 4-7: Average time in milliseconds for solving a problem with 20 nodes and 30 arcs as a function of the number of initial bounds for different numbers of bounds per branching point in Experiment 4.

nodes	20
arcs	30
node order	layer
tree type	random
initial bounds	10 to 200
bounds per branching point	1 to 20
global bounds per branching point	0
tolerance limit	0
number of instances	100

Table 4.10: Settings for Experiment 4.

varies with the size of the graph. In section 3.2.2 we saw that the lower bounds as constructed in the algorithm can give an optimal solution or a good approximation of the optimal solution. The more initial lower bounds we compute, the more chance we have to get a better bound. However, computing more initial bounds often is not necessary and the algorithm can terminate faster by branching.

In this experiment, we consider 100 instances of the general problem with 20 nodes and with number of arcs being 20, 23, 26, . . . , 44. The settings of the experiment are in Table 4.12. For each instance of the problem, we vary the number of initial bounds from 10 to 2000. We measure the average time of solving the problems.

The results of the computations are presented in Table 4.13. Figure 4-8 shows the average time of solving a problem with 20 nodes and 20 and 23 arcs as a function of the number of initial bounds. We observe, that setting the number of initial bounds to 50, gives the best time on average. If the number of initial bounds is more or less than 50, the average time to solve these problems increases.

In Figure 4-9, we see that the optimal number of initial bounds increases when the number of arcs increases. For the graphs with 20 nodes and 41 arcs, the number is

BBP	IB						
	10	20	30	50	70	100	200
1	433	316	307	316	293	310	271
2	291	220	213	219	204	223	189
3	243	173	169	187	164	172	162
4	232	156	153	168	151	149	146
5	211	145	141	156	139	139	139
6	199	144	135	151	133	134	135
7	193	136	133	148	131	133	130
8	188	132	130	148	132	128	132
9	179	130	129	146	124	127	129
10	187	131	128	146	125	130	127
11	180	131	126	145	124	124	128
12	188	131	128	145	123	128	129
13	178	127	124	142	126	128	128
14	181	129	126	124	137	122	128
15	180	131	127	126	146	128	130
16	182	132	126	124	147	128	130
17	174	132	129	125	147	127	133
18	177	133	131	129	151	130	134
19	179	131	128	126	147	128	135
20	182	135	131	131	150	130	135

Table 4.11: Average time in milliseconds for Experiment 4 to solve a problem with 20 nodes and 30 arcs with 1 to 20 BBP and 10 to 200 initial bounds (IB).

nodes	20
arcs	20 to 44
node order	layer
tree type	random
initial bounds	10 to 2000
bounds per branching point	15
global bounds per branching point	0
tolerance limit	0
number of instances	100

Table 4.12: Settings for Experiment 5.

Initial bounds	Number of arcs								
	20	23	26	29	32	35	38	41	44
10	4	16	42	195	214	2951	1657	3375	9810
20	3	15	39	191	198	3023	1583	3185	9281
50	3	15	37	186	176	2885	1464	3031	6791
100	3	15	38	189	170	2817	1244	2797	6340
200	3	17	42	189	163	2826	1201	2610	6135
500	5	24	53	186	168	1483	1085	2471	6048
1000	7	36	75	221	202	1557	1102	2411	5594
2000	12	58	121	272	276	1509	1142	2425	5524

Table 4.13: Average time in milliseconds for Experiment 5 to solve a problem with 20 nodes and 20 to 44 arcs with 10 to 2000 initial bounds.

1000. Intuitively, this result follows from the fact that as the number of arcs increases, the number of spanning trees increases. Therefore, to create good bounds we have to try more spanning trees. Figure 4-10 shows the best number of initial as a function of the number of arcs. The figure also includes an exponential regression line, showing the exponential trend.

We conclude, that to achieve the best running time of the algorithm, we need to find the best number of initial bounds as well as the number of bounds per branching point. Notice, we used 15 for the number of bounds per branching point, however, another number might give better average time. For a given number of bounds per

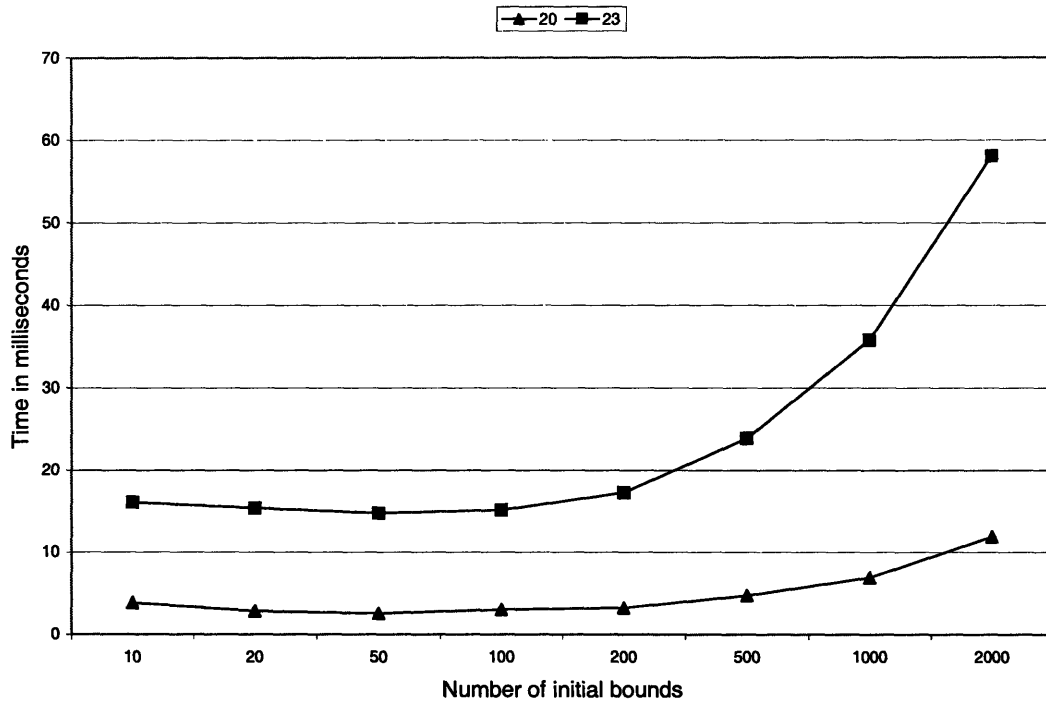


Figure 4-8: Average time in milliseconds for solving a problem with 20 nodes and 20 and 23 arcs as a function of the number of initial bounds in Experiment 5.

branching point, one can use the trend line to estimate the best number of initial bounds. For example, for 20 nodes, for our implementation of the algorithm, the trend line in Figure 4-10 is

$$\text{initial bounds} = 0.25e^{0.2 \cdot \text{number of arcs}}$$

Experiment 6. Number of global bounds per branching point (GBBP).

In Experiment 5, we concluded that we have to find the best number of initial bounds and bounds per branching point. Here, instead of finding the best number of initial bounds, we compute a number of global bounds at each branching point.

The goal of this experiment is to establish the number of global bounds per branch-

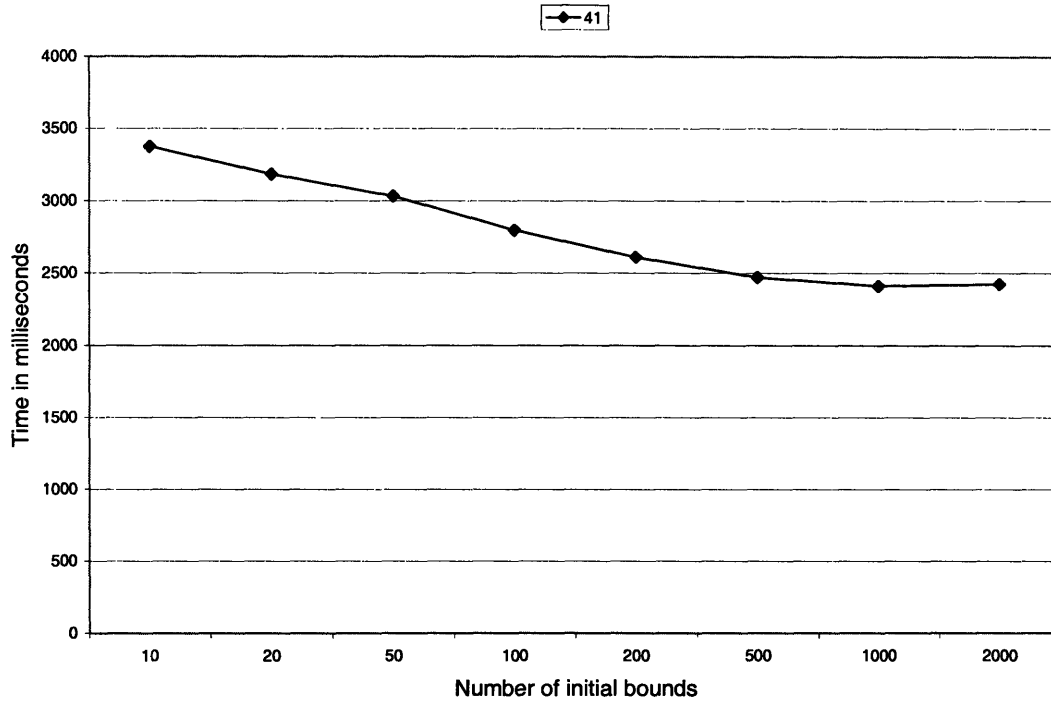


Figure 4-9: Average time in milliseconds for solving a problem with 20 nodes and 41 arcs as a function of the number of initial bounds in Experiment 5.

ing point to compute. Here, we assume that we compute at least one global bound at each branching point in addition to the branch-specific bounds that are computed in the branching point. We also want to show that the number of GBBP does not depend on the number of bounds per branching point (BBP). To do that, we vary both BBP and GBBP.

The settings of the experiments are presented in Table 6. We generate 100 random instances of the general problem with 30 nodes and 40 arcs. We solve each problem by applying the branch and bound algorithm with k bounds per branching point and l global bounds per branching point. We vary k from 1 to 29 with step 2. We vary l from 1 to $\max\{10, k\}$ with step 1.

The results of the experiments are shown in Table 4.15 and Figure 4-11. In the

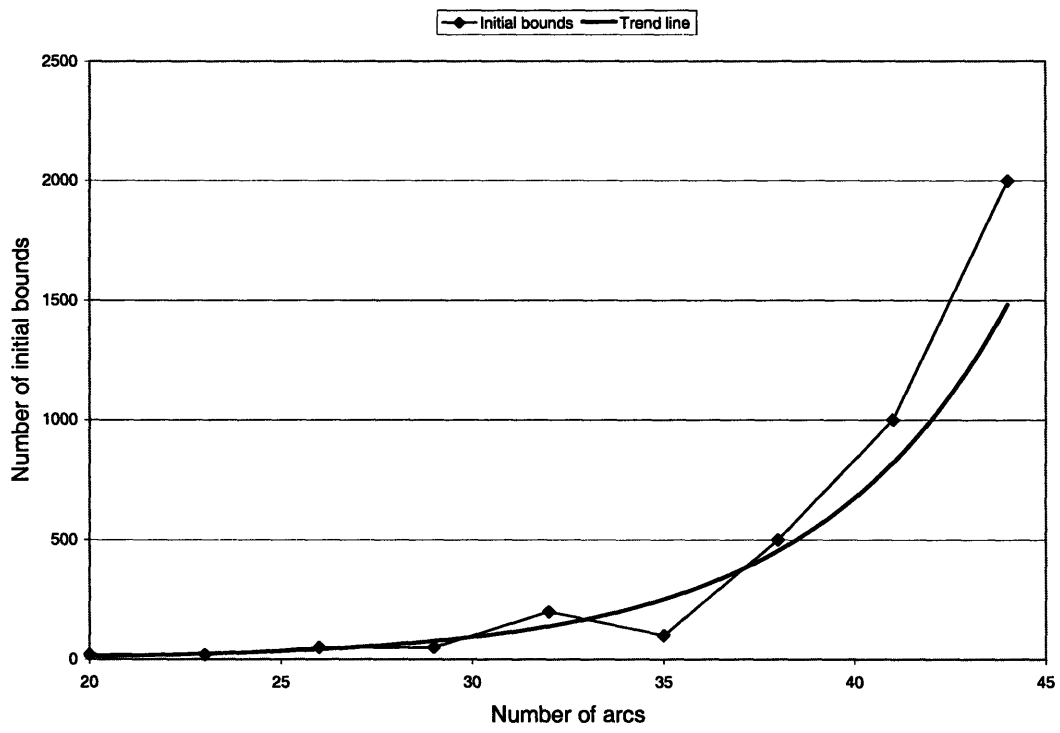


Figure 4-10: The best number of initial bounds as a function of the number of arcs in Experiment 5.

nodes	30
arcs	40
node order	layer
tree type	random
initial bounds	1
bounds per branching point	1 to 29
global bounds per branching point	1 to $\max\{10, \text{BBP}\}$
tolerance limit	0
number of instances	100

Table 4.14: Settings for Experiment 6.

table and the figure, we only show the results for a selected number of BBP. However, the results that are not stated here support the same conclusions. In particular, we see that the number of global bounds per branching point should be at most 1. For any number of bounds per branching point, the average time increases as the number of global bounds per branching point increases. From this result we conjecture, that at least for the sparse graphs, the number of global bounds computed at each branching point is at most 1 independent of the number of bounds computed to bound each branch.

We note, that the number of GBBP can be 0 or between 0 and 1. The result of Experiment 6 just states that if we compute global bounds in each branching point, then we do not need to compute more than 1 global bound per branch. We note, that the experiments with GBBP=0 are in Experiment 5.

Experiment 7. Bounds per branching point for 1 GBBP

In this experiment we compute one global bound per branching point. For different number of nodes and arcs we vary the number of bounds per branching point.

Table 4.17 shows the average time of solving a problem with 20 nodes and 20 to

	1 BBP	3 BBP	5 BBP	7 BBP	9 BBP	11 BBP	13 BBP	17 BBP	21 BBP	25 BBP	29 BBP
1 GBBP	3199	1052	744	718	706	777	669	653	852	801	738
3 GBBP	4172	1460	1080	922	872	854	790	866	807	836	836
5 GBBP	5859	1885	1300	1094	1026	990	975	937	955	949	955
7 GBBP	6923	2362	1614	1352	1240	1188	1112	1056	1026	1066	1047
9 GBBP	8360	2754	1850	1592	1392	1364	1266	1218	1180	1229	1190
11 GBBP						1472	1444	1401	1302	1296	1283
13 GBBP							1598	1513	1485	1435	1427
15 GBBP								1634	1544	1544	1542
17 GBBP								1763	1689	1625	1758
19 GBBP									1859	1873	1826
21 GBBP									1996	1949	1905
23 GBBP										2027	2028
25 GBBP										2182	2160
27 GBBP											2264
29 GBBP											2380

Table 4.15: Average time in milliseconds for solving a problem with 30 nodes and 40 arcs for different number of BBP and GBBP in Experiment 6.

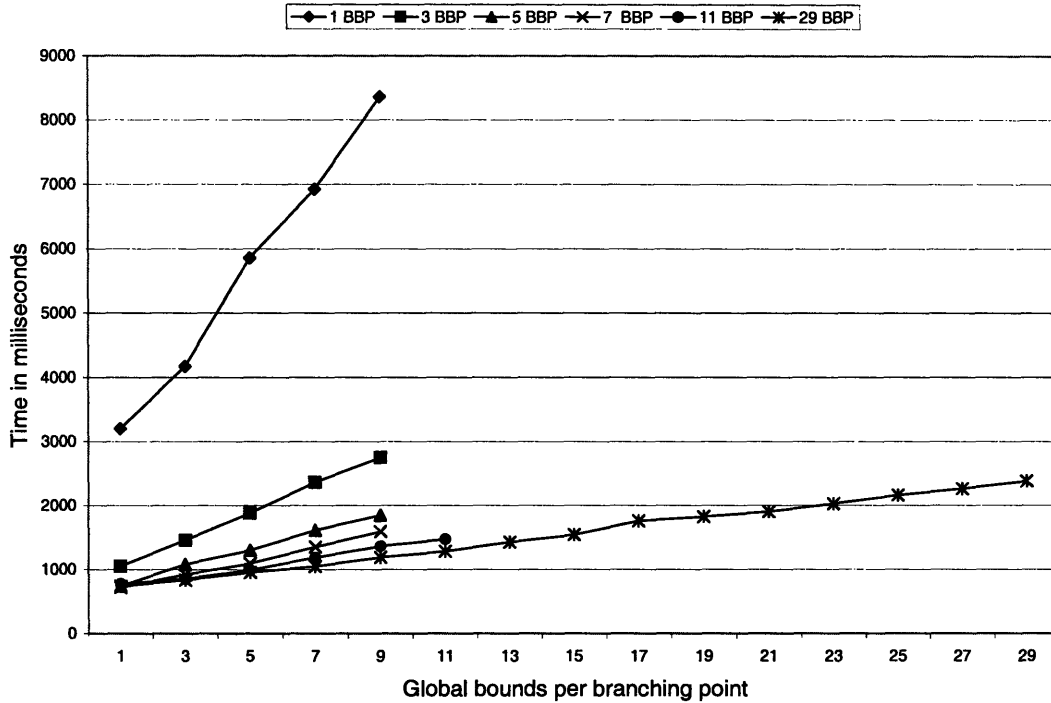


Figure 4-11: Average time in milliseconds for solving a problem with 30 nodes and 40 arcs for different number of BBP as a function of GBBP in Experiment 6.

44 arcs. Figure 4-12 presents the average time for 20 and 23 arcs as a function of bounds per branching point. We notice, that there is a value of BBP which gives the best average running time – 1 BBP for 20 arcs and 5 BBP for 23 arcs. The value increases as the number of arcs increases. For example, Figure 4-13 shows the average time for 41 arcs. The best value of BBP is 17.

If we plot the best BBP value as a function of the number of arcs (Figure 4-14), we observe that there is a linear relationship. Therefore, we suggest that an effective way to choose the best value of BBP is to use the value given by the regression line for the given number of nodes and arcs. For example, in our implementation, for 20

nodes	10 to 50
arcs	between 10 and 65
node order	layer
tree type	random
initial bounds	1
bounds per branching point	between 1 and 30
global bounds per branching point	1
tolerance limit	0
number of instances	100

Table 4.16: Settings for Experiment 7.

	Arcs								
BBP	20	23	26	29	32	35	38	41	44
1	3	28	114	362	551	4511	8392	7042	15041
3	3	16	61	241	280	2657	3429	4000	8848
5	3	14	48	218	234	1956	1971	3544	7904
7	3	14	42	225	212	1736	1585	3138	7128
9	4	15	43	213	204	1635	1464	3038	6805
11	5	16	42	207	204	1893	1360	2901	6524
13	6	16	42	201	201	1838	1331	2866	6283
15	7	19	45	211	200	1771	1325	2872	6285
17	8	21	47	183	205	1617	1303	2755	6333
19	9	22	50	182	205	1598	1304	2757	6211

Table 4.17: Average time in milliseconds for Experiment 7 to solve a problem with 20 nodes and 20 to 44 arcs with 1 GBBP and 1 to 19 BBP.

nodes the best value of BBP is estimated by

$$BBP = 1.2 * arcs - 23.$$

We present the average time of solving the problem with 30 nodes and 30 to 51 arcs as a function of BBP in Table 4.18 and Figure 4-15. We observe a drop in the average time from 1 to 10 BBP, while the higher values in the graphs do not change the time significantly, creating a flat area. For very sparse graphs, with 30 to 33 arcs

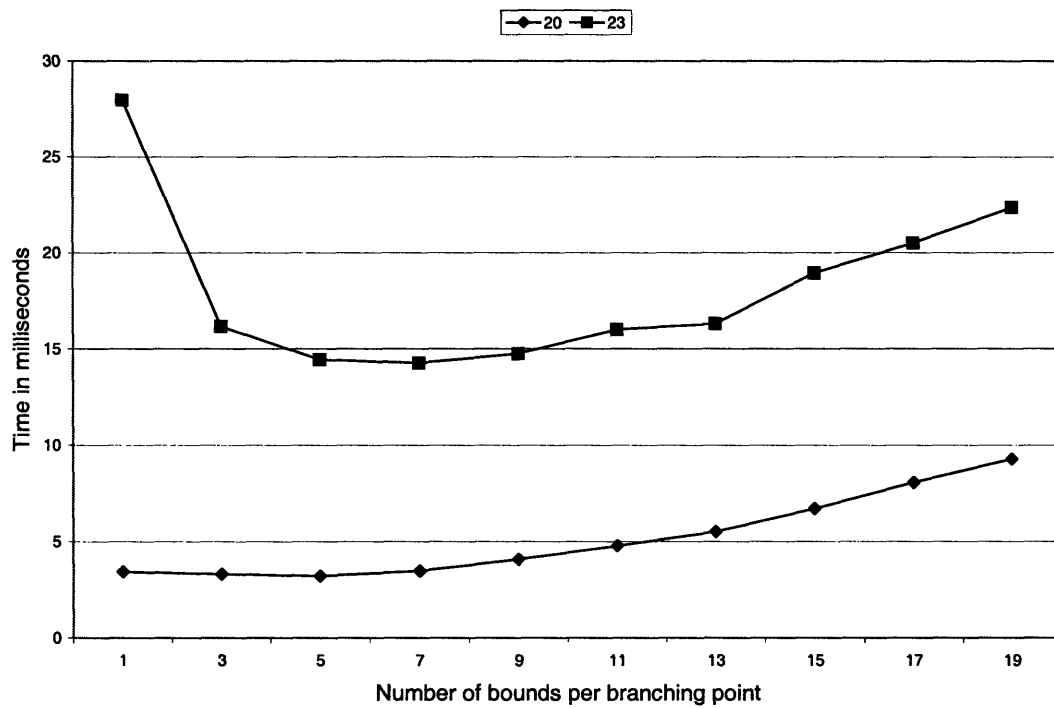


Figure 4-12: Average time in milliseconds for solving a problem with 20 nodes and 20 and 23 arcs as a function of the number of bounds per branching point in Experiment 7.

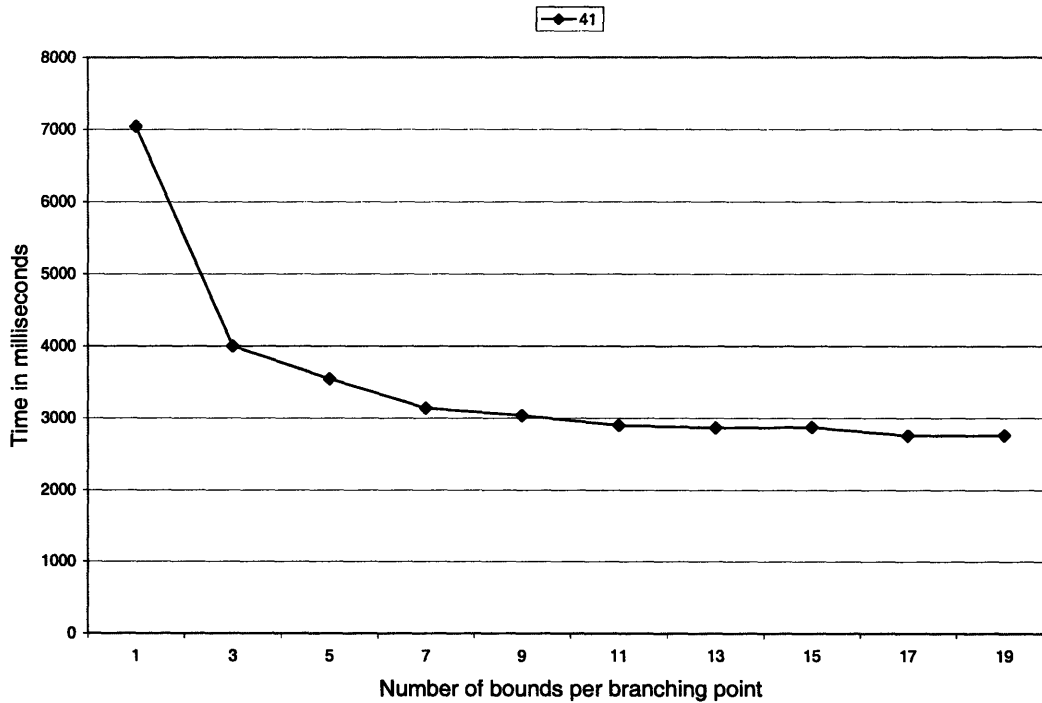


Figure 4-13: Average time in milliseconds for solving a problem with 20 nodes and 41 arcs as a function of the number of bounds per branching point in Experiment 7.

there is no significant change in time with BBP.

In Table 4.19 we report the number of instances of the problem with 30 nodes for which the algorithm failed to calculate an optimal solution within 10 minutes. For all other settings, 100% of of networks were solved under 10 minutes. We see that the number of instances depends on the number of arcs and BBP. We observed such instances only for the graphs with 48 and 51 arcs. Also, we note that there were no failed instances for 29 BBP.

We have observed the same pattern for the entire range of graph sizes that we have tested. For the problem on the graphs that we were able to solve in less than 10 minutes, 15 BBP is a good practical value. For the very sparse graphs, the best time is within milliseconds of the time with 15 BBP. For the other graphs, 15 BBP

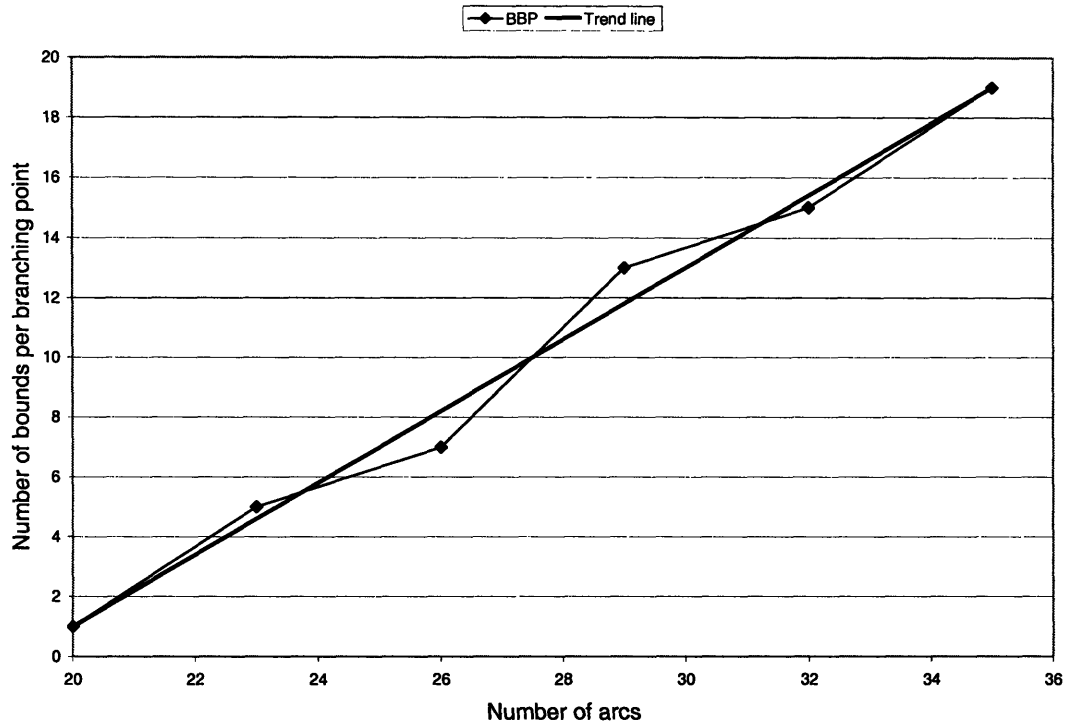


Figure 4-14: The best number of bounds per branching point as a function of the number of arcs for 20 nodes in Experiment 7.

results in near optimal times.

15 BBP can also be used as a practical value in the case when $GBBP=0$ and various numbers of initial bounds.

Experiment 8. Computing initial bounds vs. computing global bounds at every branching point

The settings of this experiment are the same as in Experiment 5 and 7 (see Tables 4.12 and 7).

In this experiment, we compare the best running times of two methods. The first method, evaluated in Experiment 5, computes all the global bounds initially. We call the method as the IB method. The second method, evaluated in Experiment 7,

BBP	Arcs							
	30	33	36	39	42	45	48	51
1	24	169	1135	3764	7336	16377	37635	73615
3	12	79	551	1628	3521	7041	26707	45360
5	14	79	462	1426	2779	5767	21105	37525
7	14	81	427	1356	2210	5031	17482	31419
9	17	83	412	1306	2030	4666	15886	29524
11	21	88	419	1292	1879	4371	15330	29397
13	22	89	419	1394	1815	4298	14943	26975
15	24	100	402	1441	1717	4150	14042	26028
17	23	94	417	1461	1725	3887	13564	25744
19	25	104	409	1296	1679	4057	13366	24770
21	29	101	407	1266	1694	3872	13259	23723
23	29	112	444	1350	1774	3775	12627	24167
25	31	117	433	1296	1733	3929	12741	23792
27	37	129	428	1332	1696	3932	12619	23372
29	38	127	433	1359	1792	3825	12945	33643

Table 4.18: Average time in milliseconds for Experiment 7 to solve a problem with 30 nodes and 30 to 51 arcs with 1 GBBP and 1 to 29 BBP.

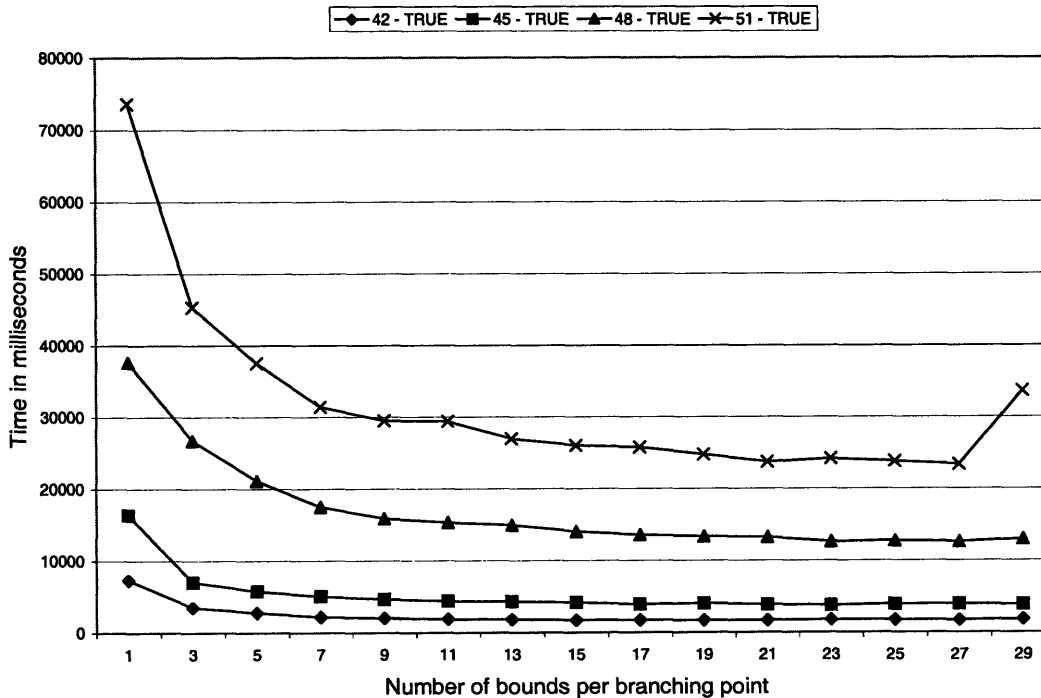


Figure 4-15: Average time in milliseconds for solving a problem with 30 nodes and 42 to 51 arcs as a function of the number of bounds per branching point in Experiment 7.

BBP	Arcs	
	48	51
1	2	3
3	0	1
5	0	1
7	0	1
9	0	1
11	0	1
13	0	1
15	0	1
17	0	1
19	0	1
21	0	1
23	0	1
25	0	1
27	0	1
29	0	0

Table 4.19: The number of instances with 30 nodes and 48 and 51 arcs that were not solved to optimality in 10 minutes in Experiment 7.

computes one global bound per branching point. We call the method as the GBBP method. We do the comparison while keeping the other relevant parameter, the number of bounds per branching point, at the practical value determined in Experiment 7, namely 15.

Table 4.20 shows that the IB method has the best average for the majority of the tested problems. Table 4.21 contains the number of instances that the methods failed to solve in 10 minutes. We notice, that there are no more than 2% of such instances among the tested problems and both methods give similar numbers. The algorithms still can solve the instances. For example, additional computations show that the instance reported in Table 4.21 with 50 nodes and 59 arcs can be solved in 30 minutes by the GBBP method.

Nodes	Arcs	GBBP best time	IB best time	Best Method
20	20	7	3	IB
20	23	19	15	IB
20	26	45	37	IB
20	29	211	186	IB
20	32	200	163	IB
20	35	1771	1483	IB
20	38	1325	1085	IB
20	41	2872	2411	IB
20	44	6285	5524	IB
25	25	19	10	IB
25	28	36	23	IB
25	31	80	69	IB
25	34	225	182	IB
25	37	384	298	IB
25	40	2823	2307	IB
25	43	2806	2592	IB
25	46	7579	6588	IB
25	49	19502	17491	IB
30	30	13	3	IB
30	33	53	33	IB
30	36	214	185	IB
30	39	766	566	IB
30	42	913	744	IB
30	45	2208	1890	IB
30	48	7469	6103	IB
30	51	13845	12265	IB
35	35	22	4	IB
35	38	227	181	IB
35	41	491	442	IB
35	44	2648	2916	GBBP
35	47	2282	1926	IB
35	50	7325	6950	IB
35	53	17990	23051	GBBP
40	40	37	20	IB
40	43	87	55	IB
40	46	1484	1083	IB
40	49	1853	1548	IB
40	52	4905	4399	IB
40	55	17545	18112	GBBP
40	58	21737	20867	IB
50	50	128	39	IB
50	53	502	152	IB
50	56	1119	1142	GBBP
50	59	2061	1750	IB
50	62	26623	18945	IB

Table 4.20: Best average time per instance in milliseconds for the two methods in Experiment 8.

Nodes	Arcs	GBBP	IB
30	51	0	1
35	53	1	1
40	52	0	1
40	58	2	2
50	59	1	1
50	62	1	1

Table 4.21: The number of instances not solved in 10 minutes by IB and GBBP methods in Experiment 8.

Obtaining the best running time for the IB method requires optimizing the number of initial bounds in addition to optimizing the number of bounds per branching point. At the same time, obtaining the best running time for the GBBP method, only requires optimizing the number of bounds per branching point as shown in Experiment 6. To optimize BBP, we find that choosing 15 works well for the problems considered here. But optimizing the number of initial bounds for the IB method is a more difficult task. In addition, the best average times of the GBBP method are close to the times obtained by the IB method. As a conclusion, the GBBP is a practical method with 1 global bound per branching point and 15 branch-specific bounds per branching point.

Comparing the average times of solving the safety stock problem using the algorithm described in the thesis to the times shown in Magnanti et al. [2004], we observe the following. For the instances that we solved, the times are comparable to those shown in Magnanti et al. [2004]. Moreover, even though the experiments were performed on computers with different speed (Pentium III 750 MHz in the case of Magnanti et al. [2004]), we expect the instances with small number of nodes to be

solved faster by the algorithm presented here. On the other hand the instances with more than 50 nodes are solved faster by the algorithm from Magnanti et al. [2004]. However, Magnanti et al. [2004] do not report experiments with the same range of arcs and nodes that are reported here. In addition, the authors only report computational times for problems that could be solved to optimality and do not report the number of instances that are or are not solved.

Finally, we show an example of the time distribution for the GBBP method. We consider 100 networks with 50 nodes and 62 arcs. According to Table 4.20, the algorithm solves a problem of this size in 27 seconds on average for the instances solved under 10 minutes. For this set, 99 out of 100 instances were solved in n under 10 minutes. In Table 4.22 we see the time distribution of solving the problems. We notice, that 87% of the instances were solved under 25 seconds. Also, the median time is under 4 seconds. This shows that the algorithm is faster than the reported average times most of the time. However, the outliers drive the average up. For example, the instance reported in Table 4.21 with 50 nodes and 62 arcs that the GBBP method failed to solve in 10 minutes was solved in 58 minutes.

Experiment 9. Tolerance limit

In this experiment, we show the performance of the algorithm for different tolerance limits. The settings of the experiment are summarized in Table 4.23. We solve the general problem with 100 nodes and 100 to 125 arcs. We generate 100 instances of each type and then solve them for 2%, 5% and 10% tolerance limits. The results are presented in Table 4.24 and in Figure 4-16. The number of instances that the

Time	Frequency
1000	13%
2000	24%
3000	11%
4000	11%
5000	7%
6000	6%
7000	1%
8000	2%
9000	1%
10000	2%
11000	1%
12000	0%
13000	0%
14000	1%
15000	0%
16000	1%
17000	2%
18000	0%
19000	0%
20000	2%
21000	0%
22000	1%
23000	1%
24000	0%
25000	0%
> 25000	13%

Table 4.22: Time distribution of solving an instance with 50 nodes and 62 arcs by the GBBP method in Experiment 8.

nodes	100
arcs	100 to 123
node order	layer
tree type	random
initial bounds	10
bounds per branching point	15
global bounds per branching point	1
tolerance limit	2%, 5%, 10%
number of instances	100

Table 4.23: Settings for Experiment 9.

algorithm failed to solve in 10 minutes are shown in Table 4.25.

From this experiment, we conclude, that the algorithm can obtain an estimate of the optimal solution fairly quickly. For the considered graphs, the 10% tolerance limit can be achieved in under 20 seconds on average.

Experiment 10. Two-layer network algorithm

In this experiment we test the performance of the two-layer network algorithm. The experiment settings are in Table 4.26. We solve two-layer network problems with 20 to 80 nodes and vary the number of arcs. We generate 100 instances of each type of network.

We use the following additional settings of the algorithm. In section 3.3.4, we described two algorithms for the upper bounds. Also, we stated a procedure that alternates between the two algorithms to obtain better bounds. In the implementation, we use the alternating procedure to obtain initial upper bounds. For each branch of the branching tree we only use the outbound service time algorithm.

The results of the computations are shown in Table 4.27. The number of instances that the algorithm failed to solve in 10 minutes are shown for each type of network

Arcs	Tolerance limit		
	2%	5%	10%
100	38	15	6
101	640	15	6
102	100	33	6
103	312	46	7
104	6658	76	15
105	591	71	17
106	2456	91	31
107	8057	97	45
108	5242	145	56
109	9870	363	58
110	9910	404	68
111	8670	413	78
112	14126	1417	137
113	31201	2256	113
114	50518	3773	141
115	45889	7888	166
116	91217	10839	269
117	107103	21611	480
118	129853	41312	1017
119	155209	46601	2615
120		78813	3085
121		118035	5527
122		148091	10780
123		164596	22728

Table 4.24: Average time in milliseconds for solving a problem with 100 nodes and 100 to 123 arcs for 2%, 5%, 10% tolerance limits in Experiment 9.

in Table 4.28. We see, that the average time of solving an instance of the two-layer problem in the tested range is under 1 minute on average if the problem is solved in under 10 minutes.

Experiment 11. Two-layer algorithm vs. general algorithm

In this experiment, we compare the performance of the general and two-layer network algorithms applied to the two-layer network problems. The settings of the experiment are in Table 4.29. We construct 100 instances of the two-layer network

Arcs	Tolerance limit		
	2%	5%	10%
100	1	0	0
101	0	0	0
102	0	0	0
103	0	0	0
104	1	0	0
105	0	0	0
106	0	0	0
107	0	0	0
108	0	0	0
109	3	0	0
110	2	0	0
111	1	0	0
112	4	1	0
113	7	1	0
114	6	1	0
115	9	0	0
116	5	2	0
117	17	2	0
118	21	1	0
119	25	2	0
120		4	0
121		9	0
122		20	0
123		23	0

Table 4.25: Number of instances that the algorithm failed to solve in 10 minutes for 2%, 5%, 10% tolerance limits in Experiment 9.

problem with 10 components and 10 demand nodes. The number of arcs is varied between 10 and 39. We solve each instance using the two-layer algorithm. Then, we solve the same instances using the general network algorithm. The average times of solving instances of the problem are in Table 4.30 and Figure 4-17. We observe that the two-layer algorithm performs better than the general algorithm on the two-layer network problems with small number of nodes.

As the number of nodes in sparse networks increases, we find that the difference

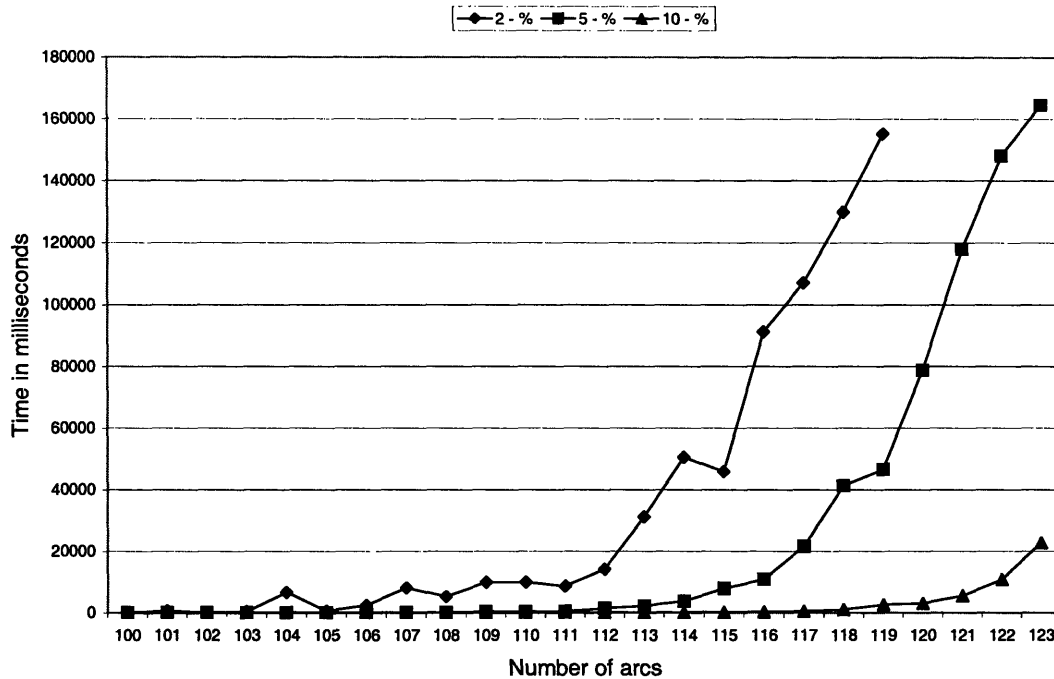


Figure 4-16: Average time in milliseconds for solving a problem with 100 nodes and 100 to 123 arcs for 2%, 5%, 10% tolerance limits in Experiment 9.

between the two algorithms becomes insignificant. We consider the networks with 40 components and 40 demand node with 80 to 90 arcs. The average time of solving the problem (for problems solved in under 10 minutes) is comparable for the two algorithms as shown in Table 4.31. In addition, the number of instances that can not be solved in 10 minutes by the two algorithms is comparable as well which is shown in Table 4.32.

As we see in the trend for the graphs with 10 components and 10 demands nodes, as the number of arcs increases, the general algorithm becomes slower. Therefore, the two-layer algorithm is preferable for the networks with larger number of arcs.

We conclude, that it is better to use the two-layer algorithm for the graphs with up to 80 nodes. For the sparse graphs with more nodes, we recommend using the

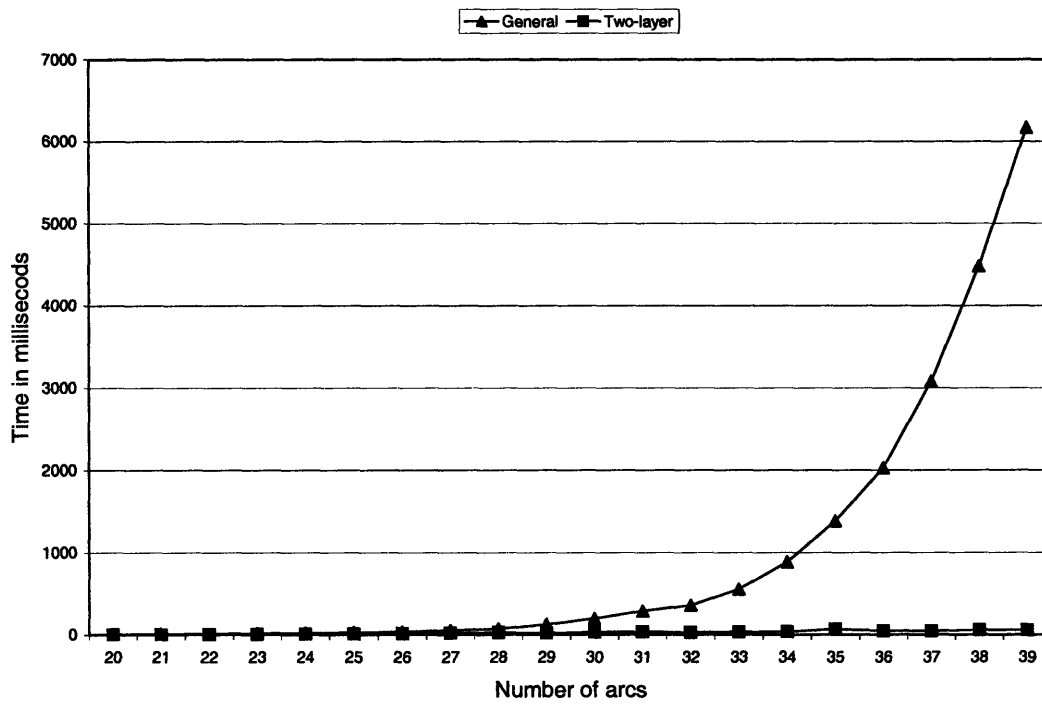


Figure 4-17: Average time in milliseconds for solving a two-layer problem with 10 components and 10 demand nodes using the two-layer and general network algorithms in Experiment 11.

components	10 to 40
demand nodes	10 to 40
arcs	10 to 91
tree type	random
initial bounds	10
bounds per branching point	15
global bounds per branching point	1
tolerance limit	0
number of instances	100

Table 4.26: Settings for Experiment 10.

general algorithm.

Arcs-Nodes	Nodes						
	20	30	40	50	60	70	80
0	2	13	8	6	3710	13	48
1	2	20	17	37	141	236	19
2	3	12	21	373	237	4556	3150
3	5	16	179	67	4553	3500	5079
4	8	22	2200	521	1695	11504	2720
5	8	33	3516	7950	11277	4359	2405
6	11	50	2761	3662	14093	7553	4622
7	19	74	1350	5946	10302	21838	11408
8	24	150	1302	2817	22654	16366	23105
9	18	170	1179	4014	23307	21582	36428
10	27	440	2586	12091	25121		34407
11	33	187	1356	10788	30342		63297
12	27	196	9618	27190			
13	29	391	6914	26291			
14	34	599	6345				
15	62	406	11246				
16	43	812					
17	46	1342					
18	53						
19	57						

Table 4.27: Average time per instance in milliseconds for solving two-layer problems in Experiment 10.

Arcs-Nodes	Nodes				
	40	50	60	70	80
1	0	1	0	1	0
2	0	0	0	1	0
3	0	0	2	2	0
4	0	0	3	2	0
5	0	2	0	10	0
6	0	3	4	6	1
7	0	1	2	13	2
8	0	3	6	8	1
9	0	3	6	13	1
10	0	4	11		7
11	0	3	13		2
12	0	7			
13	0	8			
14	3				

Table 4.28: Number of instances not solved in 10 minutes by the two-layer algorithm in Experiment 10.

components	10
demand nodes	10
arcs	10 to 39
node order	layer
tree type	random
initial bounds	1
bounds per branching point	15
global bounds per branching point	1
tolerance limit	0
number of instances	100

Table 4.29: Settings for Experiment 11.

Arcs	General	Two-layer
20	7	3
21	8	5
22	12	6
23	18	7
24	20	12
25	27	11
26	37	13
27	53	21
28	72	27
29	127	21
30	199	32
31	292	37
32	359	29
33	557	35
34	887	38
35	1385	67
36	2029	46
37	3080	47
38	4479	58
39	6168	59

Table 4.30: Average time in milliseconds for solving a two-layer problem with 10 components and 10 demand nodes using the two-layer and general network algorithms in Experiment 11.

Arcs	General	Two-layer
80	55	48
81	21	19
82	3290	3150
83	5540	5079
84	2936	2720
85	2505	2405
86	4778	4622
87	12099	11408
88	24534	23105
89	38028	36428
90	35951	34407

Table 4.31: Average time in milliseconds for solving a two-layer problem with 40 components and 40 demand nodes using the two-layer and general network algorithms in Experiment 11.

Arcs	General	Two-layer
80	0	0
81	0	0
82	0	0
83	0	0
84	0	0
85	0	1
86	1	2
87	2	1
88	1	1
89	1	7
90	7	2

Table 4.32: Number of instances that general and two-layer algorithms failed to solve in 10 minutes for problems with 40 components and 40 demand nodes in Experiment 11.

Chapter 5

Conclusions

In this thesis, we consider the problem of determining the placement of safety stock in general network supply chains. We assumed the framework of guaranteed deterministic service times and proposed an algorithm for solving the problem. In particular, we modeled the supply chain as a network with nodes of the network representing processing functions of a manufacturing company as well as the locations of the safety stock. Each stage of the supply chain operates under a periodic-review base-stock policy. Stages of the supply chain quote service times to the adjacent downstream stages and to the end customers. The most important assumption of the model is that the customer demand is bounded. Then, each stage guarantees to provide 100% of the customer demand up to the assumed bound.

We show that the general network safety stock problem is NP-hard and provide a branch and bound algorithm for solving the problem. We analyze the set of optimal solutions and find, that by constructing all the paths from one node to any other node in the network, we can enumerate all possible candidates for the optimal service

times for the node. We use the paths to construct a branching tree for the algorithm. We obtain lower bounds for the branches of the branching tree by solving a spanning tree relaxation of the problem. We develop a polynomial time algorithm to solve the spanning tree safety stock problem. The spanning tree algorithm is $O(N^3)$, where N is the number of nodes in the network. We also develop a specialized branch and bound algorithm for the two-layer network problems.

We have performed a set of computational experiments to explore the computational performance of the developed algorithms. The general network algorithm solves safety stock problems on sparse networks with up to 50 nodes in under 30 seconds on average on a Pentium IV 2.8 HGz workstation running Windows XP. The two-layer network algorithm solves the problems with the sparse graphs with up to 80 nodes in under 1 minute on average.

We have examined how to set the parameters of the algorithms to achieve the best computational performance. We suggest that the algorithm branches on the service times of the nodes starting from demand nodes and finishing with components. We observed that the best way to generate spanning trees is to generate them randomly. For the tested sparse graphs, we find that computing 15 lower and upper bounds per branch gives good results on average. We suggest using the polynomial time spanning tree algorithm to compute a number of global bounds for the optimal solution. We suggest computing global bounds every time the algorithm analyzes a branch of the branching tree.

Additional research can be done in this area. Here we list potential extensions for future research.

- **Two-layer networks.** We proposed a branch and bound algorithm for the two-layer networks. However, the complexity of the two-layer network problem remains unknown. It would be useful to know whether or not the problem is NP-hard. In addition, one could explore how to use the two-layer networks in solving problems with more general networks. Indeed, the algorithm described by Humair and Willems [2003] entails solving two-layer networks, and the method developed in this thesis might be incorporated into their algorithm. Alternatively, one might solve a general network problem by decomposing it into a series of two-layer network problems that are solved in some iterative scheme.

- **Bounds.** One could explore possible ways to obtain better lower bounds. For instance, one idea is to create lower bounds by using Lagrange relaxation. The spanning tree relaxation can be viewed as a Lagrange relaxation with Lagrange multipliers set to zero for the removed arcs. By adjusting these Lagrange multipliers, the lower bounds can be tightened. The relaxed problem with non-zero Lagrange multipliers can be solved by applying an algorithm similar to the spanning tree algorithm. However, to solve each relaxation in polynomial time, the candidate values for the service times would have to be adjusted.

For the upper bounds, it might be possible to find a condition under which the problem is solved in polynomial time. For example, we have found that in the two-layer network case, we could solve a problem with imposed order for the outbound service times of the components in polynomial time. For the general

networks, we obtain the upper bounds used here by fixing the spanning tree solution. It may be possible to find better upper bounds from the (polynomial) solution to a restricted version the problem.

- **Production capacity.** Perhaps the most interesting direction for the future research is to analyze the problem with production capacity constraints. In our work to date, we assume that the production capacity is infinite, that is, each stage of the network can process as much demand as necessary at a time. However, in reality the production capacity is often limited.

One would presumably start with a good model for a single stage system, which permits a capacity constraint. However, even with a good model for the one-stage network, there are several problems in extending the model to more complicated network structures. In particular, the biggest question is, how to model the demand propagation. One possibility is to let each node see the end customer demand and to set the safety stock accordingly. This will result in the delays in the production and additional inventory held at the nodes. Another possibility is to only allow the component nodes, the nodes with zero indegree, to see the end customer demand and to propagate the demand downstream. This will also result in some inefficiency due to the complications of modeling the timing of the arrival of the inventory to the assembly nodes.

- **Stochastic lead-times.** We assume in this thesis that the lead-times at all the stages are deterministic. However, in practice the lead-times are often stochastic. Therefore, it would be useful to incorporate the uncertain lead-times into

the model.

- **Non-stationary demand.** A more practical model can be developed by assuming non-stationary end customer demand. We assume the the demand is stationary, however, in practice the expected demand is likely to vary from period to period due to, for example, seasonal and promotional changes.

Bibliography

- H.P. Benson. On the convergence of two branch-and-bound algorithms for nonconvex programming problems. *Journal of Optimization Theory and Applications*, 36:129–134, 1982.
- H.P. Benson. Separable concave minimization via partial outer approximation and branch and bound. *Operations Research Letters*, 9:389–394, 1990.
- H.P. Benson. Concave minimization: Theory, applications and algorithms. In P.M. Pardalos and R. Horst, editors, *Handbook of Global Optimization*, volume 2 of *Nonconvex Optimization and Its Applications*, chapter 3, pages 43–148. Kluwer Academic Publishers, 1994.
- H.P. Benson. Deterministic algorithms for constrained concave minimization: a unified critical survey. *Naval Research Logistics*, 43:765–795, 1996.
- H.P. Benson. Generalized γ -valid cut procedure for concave minimization. *Journal of Optimization Theory and Applications*, 102(2):289–298, August 1999.
- D.P. Bertsekas. *Dynamic Programming and Optimal Control*, volume 1. Athena Scientific, second edition, 2000.

- A.V. Cabot. Variations on a cutting plane method for solving concave minimization problem with linear constraints. *Naval Research Logistics Quarterly*, 21:265–274, 1974.
- A.V. Cabot and R.L. Francis. Solving certain nonconvex quadratic minimization problems by ranking extreme points. *Operations Research*, 18:82–86, 1970.
- S.-J. Chung. NP-completeness of the linear complementarity problem. *Journal of Optimization Theory and Applications*, 60:393–399, 1989.
- A.J. Clark and H. Scarf. Optimal policies for a multi-echelon inventory problem. *Operations Research*, 6(4):475–490, July 1960.
- M. Ettl, G.E. Feigin, G.Y. Lin, and D.D. Yao. A supply network model with base-stock control and service requirements. *Operations Research*, 48(2):216–232, Mar. - Apr. 2000.
- J.E. Falk and K.R. Hoffman. A successive underestimation method for concave minimization problems. *Mathematics of Operations Research*, 1:251–259, 1976.
- J.E. Falk and R.M. Soland. An algorithm for separable nonconvex programming problems. *Management Science*, 15:550–569, 1969.
- F. Glover. Convexity cuts and cut search. *Operations Research*, 21:123–134, 1973.
- S. Graves and E. Lesnaia. Optimizing safety stock placement in general network supply chains. SMA Symposium, page 7 pp. SMA, January 2004. URL <https://dspace.mit.edu/handle/1721.1/3915>.

- 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. Safety stocks in manufacturing systems. *Journal of Manufacturing and Operations Management*, 1(1):67–101, 1988.
- S.C. Graves and S. Willems. Optimizing strategic safety stock placement in supply chains. URL web.mit.edu/sgraves/www/papers. long version, 1998.
- S.C. Graves and S. Willems. Erratum: Optimizing strategic safety stock placement in supply chains. *Manufacturing & Service Operations Management*, 5(2):176–177, Spring 2003a.
- S.C. Graves and S. Willems. Supply chain design: safety stock placement and supply chain configuration. In A.G. de Kok and S.C. Graves, editors, *Handbooks in OR & MS*, volume 11, pages 95–132. Elsevier, 2003b.
- R. Horst. An algorithm for nonconvex programming problems. *Mathematical Programming*, 10:312–321, 1976.
- R. Horst. On the global minimization of concave functions - introduction and survey. *OR Spektrum*, 6:195–205, 1984.
- R. Horst, N.V. Thoai, and H. Tuy. Outer approximation by polyhedral convex sets. *Operations Research Spektrum*, 9:153–159, 1987.

- R. Horst, N.V. Thoai, and H. Tuy. On an outer approximation concept in global optimization. *Optimization*, 20:255–264, 1989.
- R. Horst and H. Tuy. *Global Optimization: Deterministic Approaches*. Springer Verlag, Berlin, second revised edition, 1993.
- S. Humair and S. Willems. Optimal inventory placement in networks with clusters of commonality. Working paper, January 2003.
- 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.E. Jacobsen. Convergence of a Tuy-type algorithm for concave minimization subject to linear constraints. *Applied Mathematics and Optimization*, 7:1–9, 1981.
- G.E. Kimball. General principles of inventory control. *Journal of Manufacturing and Operations Management*, 1:119–130, 1988.
- B. Korte and J. Vygen. *Combinatorial Optimization: Theory and Algorithms*, volume 21 of *Algorithms and Combinatorics*. Springer Verlag, 2 edition, 2002.
- A.H. Land and A.G. Doig. An automatic method of solving discrete programming problems. *Econometrica*, 28:497–520, 1960.

- H.L. Lee and C. Billington. Material management in decentralized supply chains. *Operations Research*, 41(5):835–847, Sep. - Oct. 1993.
- E. Lesnaia. Optimizing safety stock placement in two-layer supply chains. SMA Symposium, page 5 pp. SMA, January 2003a.
- E. Lesnaia. Research oriented paper: Optimizing safety stock placement in two-layer supply chains. Technical report, MIT Operations Research Center, 2003b.
- L. Liu, X. Liu, and D.D. Yao. Analysis and optimization of a multistage inventory-queue system. *Management Science*, 50(3):365–380, March 2004.
- T.L. Magnanti, Z.-J. M. Shen, J. Shu, D. Simchi-Levi, and C.-P. Teo. Inventory placement in acyclic supply chain networks. 2004.
- A. Majthay and A. Whinston. Quasi-concave minimization subject to linear constraints. *Discrete Mathematics*, 9:35–59, 1974.
- O.L. Mangasarian. Characterization of linear complementarity problems as linear programs. *Mathematical Programming Study*, 7:74–87, 1978.
- P. McKeown. A vertex ranking procedure for solving the linear fixed-charge problem. *Operations Research*, 23:1182–1191, 1975.
- P.G. McKeown. Extreme point ranking algorithms: A computational survey. In W.W. White, editor, *Computers and Mathematical Programming*, National Bureau of Standards Special Publication. U.S. Government Printing Office, 1978.

- S. Minner. Dynamic programming algorithms for multi-stage safety stock optimization. *OR Spektrum*, 19:261–271, 1997.
- S. Minner. *Strategic Safety Stocks in Supply Chains*, volume 490 of *Lecture Notes in Economics and Mathematical Systems*. Springer, 2000.
- S. Minner. Multiple-supplier inventory models in supply chain management: a review. *International Journal of Production Economics*, 81-82:265–279, 2003.
- B.M. Mukhamediev. Approximate methods of solving concave programming problems. *Zhurnal Vychislitelnoj Matematiki i Matematicheskoy Fiziki*, 22:727–732, 1982. Translated: USSR Computational Mathematics and Mathematical Physics, 22(3):238-245.
- K. Murty. Solving the fixed charge problems by ranking the extreme points. *Operations Research*, 16:268–279, 1969.
- P.M. Pardalos and J.B. Rosen. Method for global concave minimization: A bibliographic survey. *SIAM Review*, 28:367–379, 1986.
- P.M. Pardalos and J.B. Rosen. *Constrained Global Optimization: Algorithms and Applications*. Springer Verlag, 1987.
- S. Sahni. Computationally related problems. *SIAM Journal on Computing*, 3:262–279, 1974.
- J.P. Shectman and N.V. Sahinidis. A finite algorithm for global minimization of separable concave programs. *Journal of Global Optimization*, 12(1):1–35, Jan 1998.

- Z.-J. Shen. Note on the safety stock placement. Unpublished, 2003.
- C. Sherbrooke. METRIC: a multi-echelon technique for recoverable item control. *Operations Research*, 16:122–141, 1968.
- K.F. Simpson. In-process inventories. *Operations Research*, 6:863–873, 1958.
- R.M. Soland. Optimal facility location with concave costs. *Operations Research*, 22: 373–382, 1974.
- J.-S. Song and D.D. Yao. Performance analysis and optimization of assemble-to-order systems with random lead times. *Operations Research*, 50(5):889–903, September–October 2002.
- H.A. Taha. On the solution of zero-one linear programs by ranking the extreme points. Technical report, University of Arkansas, Fayetteville, Arkansas, 1972.
- H.A. Taha. Concave minimization over a convex polyhedron. *Naval Research Logistics Quarterly*, 20:533–548, 1973.
- N.V. Thoai and H. Tuy. Convergent algorithms for minimizing a concave function. *Mathematics of Operations Research*, 5:556–566, 1980.
- H. Tuy. Concave programming under linear constraints. *Doklady Akademii Nauk SSSR*, 159:32–35, 1964. Translated: Soviet Mathematics Doklady, 5. pp. 1437–1440.
- H. Tuy. On polyhedral annexation method for concave minimization. In L.J. Leifman, editor, *Functional Analysis, Optimization and Mathematical Economics: a*

- collection of papers dedicated to the memory of L.V. Kantorovich*, pages 248–260.
Oxford Press, New York, 1990.
- S. Vavasis. Quadratic programming is in NP. *Information Processing Letters*, 36:
73–77, 1990.
- P. Zipkin. *Foundations of Inventory Management*. McGraw-Hill, New York, 2000.
- P.B. Zwart. Nonlinear programming: counterexamples to two global optimization
algorithms. *Operations Research*, 21:1260–1266, 1973.
- P.B. Zwart. Global maximization of a convex function with linear inequality con-
straints. *Operations Research*, 22:602–609, 1974.