Supply Chain Optimization: Formulations and Algorithms

by

Carl E. Wike
B.S., Applied Mathematics, North Carolina State University

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

Master of Science in Operations Research

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

February 1999

Author ....
Sloan School of Management
August 31, 1998

Certified by ....
Dimitris J. Bertsimas
Boeing Professor of Operations Research
Thesis Supervisor

Accepted by ....
James B. Orlin
E. Pennell Brooks Professor of Operations Research
Codirector, Operations Research Center

Supply Chain Optimization: Formulations and Algorithms

by

Carl E. Wike

B.S., Applied Mathematics, North Carolina State University

Submitted to the Sloan School of Management
on August 31, 1998, in partial fulfillment of the
requirements for the degree of
Master of Science in Operations Research

Abstract

In this thesis, we develop practical solution methods for a supply chain optimization problem: a multi-echelon, uncapacitated, time-expanded network of distribution centers and stores, for which we seek the shipping schedule that minimizes total inventory, backlogging, and shipping costs, assuming deterministic, time-varying demand over a fixed time horizon for a single product. Because of fixed ordering and shipping costs, this concave cost network flow problem is in a class of NP-hard network design problems. We develop mathematical programming formulations, heuristic algorithms, and enhanced algorithms using approximate dynamic programming (ADP). We achieve a strong mixed integer programming (MIP) formulation, and fast, reliable algorithms, which can be extended to problems with multiple products.

Beginning with a lot-size based formulation, we strengthen the formulation in steps to develop one which is a variation of a node-arc formulation for the network design problem. In addition, we present a path-flow formulation for the single product case and an enhanced network design formulation for the multiple product case.

The basic algorithm we develop uses a dynamic lot-size model with backlogging together with a greedy procedure that emulates inventory pull systems. Four related algorithms perform local searches of the basic algorithm's solution or explore alternative solutions using pricing schemes, including a Lagrangian-based heuristic.

We show how approximate dynamic programming can be used to solve this supply chain optimization problem as a dynamic control problem using any of the five algorithms. In addition to improving all the algorithms, the ADP enhancement turns the simplest algorithm into one comparable to the more complex ones.

Our computational results illustrate that our enhanced network design formulation almost always produces integral solutions and can be used to solve problems of moderate size (3 distribution centers, 30 stores, 30 periods). Our heuristic methods, particularly those enhanced by ADP methods, produce near optimal solutions for truly large scale problems.

Thesis Supervisor: Dimitris J. Bertsimas
Title: Boeing Professor of Operations Research

Acknowledgments

I would like to thank everybody at the MIT Operations Research Center for a memorable experience over these past two years.

I am deeply grateful to my thesis advisor, Professor Dimitris Bertsimas, for the many hours he has spent with me on this research effort and for his patience and guidance through it all. This has been a most enjoyable learning experience. His commitment to excellence in academics is a trait I am very familiar with, and so...

I take this opportunity to thank my parents, Mr. and Mrs. C. E. Wike, Sr. I am just one of thousands of North Carolinians who learned so much from them.

I owe my wife Ellen a huge debt of gratitude. None of this would have happened without her encouragement, support, and sense of humor.

To our daughter Tara (MIT '97): thanks for being my best friend on campus. We are so proud of you.

Finally, to our son Sean, who lives every day to the fullest and is an inspiration to us all: this is for you.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

As businesses worldwide face intense competition in an age of instant communication, there has been increasing emphasis on reducing costs in their supply chains. The costs involved are huge. In the United States alone, inventories cost over $1 trillion per year; logistics account for 30% of the cost of goods sold; and the third-party logistics outsourcing business has grown to $50 billion per year. This has recently generated a considerable effort in research activity in operations research and related fields, combining new methodologies with the established theories of inventory, distribution, and production planning.

This thesis focuses on a common type of supply chain optimization problem encountered in managing a distribution network, providing a theoretical context, practical algorithms, computational examples, and a discussion of implementation issues. Although there are many interesting theoretical aspects to this computationally complex problem, the emphasis throughout will be on near optimal, efficient algorithms that are practical to implement.

The remainder of this chapter gives a general introduction to the type of supply chain optimization problem we are interested in along with a review of the relevant literature.

1.1 Supply Chain Optimization

This section describes supply chain optimization in general and introduces the specific problem we are interested in.

The General Problem

The total supply chain for an enterprise consists of all aspects of the supply, production, and distribution of products from raw material acquisition to manufacturing to the delivery of finished goods to the retail customers.

The objective in supply chain optimization is to minimize the total supply chain costs while meeting customer demand within a target level of customer service.

The major cost areas affected by supply chain decisions are:

1. Inventory Holding Costs.
2. Inventory Shortage Costs (backorders, lost sales).
3. Transportation Costs (shipping or freight).

Supply chain optimization models are concerned with two major decision categories: design and control.

Design decisions involve the structure of the supply chain:

1. Whether to have a distribution center (DC).
2. Facility location of a DC or warehouse.
3. Choice of transportation modes.
4. Sourcing of suppliers.

Control decisions attempt to develop an optimal policy for moving goods over time through a supply chain network:

1. Inventory policy: when to order and how much to order.

2. What transportation mode to use.
3. Vehicle routing.
4. Inventory routing.

The trends of the last decade have been to reduce costs in portions of the supply chain by pushing inventory management back on the suppliers, employing lean manufacturing and just-in-time (JIT) inventory practices, and contracting out to third-party logistics providers while seeking to maintain high levels of customer service. These simultaneous goals are often in conflict. Achieving minimal inventory may require more frequent shipments. Reducing freight or improving demand fill rates may increase inventory levels.

Many businesses, large retailers in particular, have converted from the traditional "Push" system (building to inventory to meet forecasted demand) to a "Pull" system, where sales generate replenishment orders. In this new method, customer demand at the retail end pulls products through the supply chain all the way back from the supply source.

Technology has played a major role in the development of inventory pull systems and other just-in-time techniques. Corporate information systems are now able to generate replenishment orders at any point in the supply chain instantly, using real-time inventory and shipping information plus purchasing information which is constantly updated by customer transactions. We will use the idea of a Pull system to construct an algorithm in Chapter 3.

Although attempts are often made to optimize portions of the supply chain, possibly in conflict with global optimization goals, it would probably be infeasible to attempt a global solution for any large-scale enterprise. A more realistic approach would be to find the right sub-division of the problem, perhaps into areas of supply, production, and distribution.

In this study we will focus on a distribution problem: the shipments and positioning of inventory in a multi-echelon (or multi-level) distribution network over time that will meet demand while minimizing the total distribution costs that are affected by those decisions.

The Specific Problem

Suppose we have a multi-echelon distribution network where goods move from factories to distribution centers or warehouses and from there to stores. If we had a perfect forecast of demand over a finite time horizon, we would like to find the optimal movement of products through the network that would minimize total system costs. When would we ship, and when would we carry inventory?

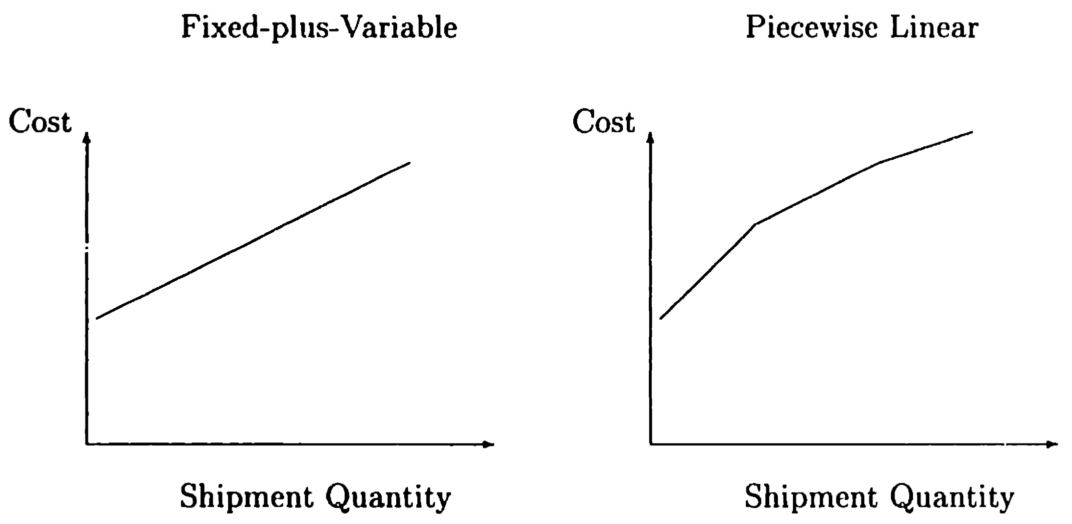
For our problem, we will assume deterministic, time-varying demand. There are some limitations and benefits to modeling demand this way. A stochastic demand model would be more realistic. However, these models often assume stationarity, which is not applicable because many products have a relatively short life cycle, follow irregular seasonal demand patterns, or are influenced by promotions.

The benefit of the deterministic, time-varying approach is in budgeting and planning. If we can solve this type of problem quickly, then we could simulate various scenarios and analyze the results. We could also analyze a company's supply chain cost performance by finding what the optimal or near-optimal solution would be given the historical demand.

The distribution network will have a general, layered structure with the layers being factories, distribution centers, warehouses, and stores. At times, we will restrict it to have two echelons or levels, where the first echelon is from DC's to stores, and the second echelon is from the factory to the DC's. We will also at times restrict the network structure to be arborescent - a node can have at most one source - which is realistic for many but not all distribution problems. The arborescent networks will allow us to separate larger problems into sub-problems.

Our models will seek to minimize total system costs, which are composed of inventory holding costs, backorder penalty costs, and the costs of shipping. The holding and penalty costs can be modeled as linear functions of excess inventory or shortages. If shipping costs were linear, the optimization problem would be easy to solve. Un-

Figure 1-1: Shipping Cost Functions



fortunately, they are not, which results in the most important complication we have with this problem.

Shipping costs are concave because there is a fixed cost (for ordering, setup, etc.) in addition to variable costs. LTL (less-than-truckload) shipping models formulate the concave shipping costs as a concave, piecewise linear function that is based on the typical volume discounts offered by freight companies. For our purposes, we will use a fixed-plus-variable approximation for these costs (see Figure 1-1).

There are several benefits of using a fixed-plus-variable freight cost.

- Since the total variable costs are unaffected by a solution (ultimately, all the demand has to be shipped at some time), we can treat these costs as constant and ignore them (Zangwill [41]).

- This approach is much simpler than the LTL models with the piecewise linear costs while still providing a good approximation of the shipping costs.

- A fixed cost is useful if the model is applied to factory production (in which case the fixed cost is the setup cost for production).

- A more subtle benefit is that for a model with daily time increments, having a fixed cost component can avoid solutions where there are shipments every day, no matter how small. This allows us to use small time increments and forces the model to determine when to ship.

Following the literature, we will formulate and solve single product models. Multiple product models provide more complexity than benefit because of the usual high correlation of demand among products (Lee and Nahmias [20]). Some of the formulations and algorithms we develop may be extended to the multiple product case.

In modeling this distribution problem, we can choose whether to assume uncapacitated inventory storage and shipping or to apply limits. The uncapacitated case is a common assumption in network design and concave cost network flow models.

Another common research practice has been to assume fixed lead times or even to ignore lead times. Models and algorithms are usually developed with zero lead times and put into practice by a simple translation of the demand periods. Of course, an extension of the model to include choices of shipping modes would require explicit incorporation of lead times.

To summarize, we will try to find the optimal daily or weekly shipment schedule of a single product over a multi-echelon distribution network, for a fixed time horizon, assuming unlimited capacity, fixed shipping costs, linear inventory and shortage costs, zero lead times, and deterministic, time-varying demand.

1.2 Literature Review

The research literature that is relevant to our particular supply chain optimization problem can be found in the emerging supply chain literature and in the extensive body of literature in distribution-inventory theory, production planning, network design, and related theory. The following discussion is organized into these general categories:

1. Literature that is specific to supply chain management and optimization.

2. Multi-echelon distribution and inventory theory.

3. Dynamic lot-size models that are used for production planning.

4. Concave cost network flow models.

5. Network design models.

6. Neuro-dynamic programming.

Supply Chain Specific Literature

Insight into the general business problems of supply chain management can be found in many recent articles and books in the business literature (Poirier and Reiter [27]; Davis [9]; Lee and Billington [18]).

The survey by Thomas and Griffin [34] categorizes the relevant OR literature into strategic planning models and operational planning models. They separate strategic planning models into mixed integer program (MIP) and non-MIP model groups, and they group operational planning models into three major components of the supply chain:

1. Buyer-vendor coordination.

2. Production-distribution coordination.

3. Inventory-distribution coordination.

An influential paper by Geoffrion and Graves [16] for the models and case studies that use MIP's employs Bender's decomposition in a model that includes DC selection (using binary forcing variables) in a case study for Hunt-Wesson. Other case studies have used this MIP and decomposition approach because these models tend to have embedded networks. (Also see the Ault Foods case [28], and the Libbey-Owens-Ford case [22].)

The best example of using a MIP for a supply chain design problem is a case study for DEC (Arntzen et al [2]). In this model, the objective function is a combination of costs and total time (length of supply chain). This model and others like it follow the Geoffrion and Graves model using indicator variables that are forcing constraints for what plants and DCs to open or close. This model also includes bills-of-materials, tariffs, and duties.

Some interesting non-MIP strategic planning models that use stochastic demand can be found in Cohen and Lee [8]. This is an integrated production and distribution system, where the authors decompose the problem into material control, production, inventory, and distribution. Another example is the Lee and Billington [19] article on the Hewlett-Packard supply chain, one of many articles on HP in the literature.

Thomas and Griffin [34] conclude that the most popular supply chain models are MIP's with underlying network structures that lead to decomposition. They point out that many models use linear transportation cost functions which are not realistic; and that the literature is sparse in models at the operational level, which are mostly stochastic models that have the previously mentioned problems with assumptions about stationarity.

Aside from the afore-mentioned, there is not much to be found in the operations research literature that is specific to supply chains. To probe deeper into the issues and problems of specific aspects of supply chain optimization, we can refer to the vast literature in inventory and distribution theory, production planning, and logistics.

Multi-echelon Distribution

As discussed in Federgruen [12], the only known exact solution to the problem of an optimal multi-echelon distribution policy is by Clark and Scarf [7]. This result is limited to serial networks where the setup costs are zero except for outside orders.

Clark and Scarf provide a method for decomposing a serial distribution network's optimal policy problem into single location problems. They prove that local optimal policies are together globally optimal. This model has these restrictions: the network must be in series, and the only fixed ordering or shipment costs occur with shipments to the first facility in the series. An instance where this model is useful is with military supply depots. The solution decomposition method involves a recursion where the s-type policy is found for the lowest echelon, the s-type policy for the next higher echelon is determined, based on the additional penalty costs for inadequate shipments to the next lower echelon (given its s-type policy), and so on until the highest echelon is reached, which can have either an s-type policy or (s, S)-type policy if there is a fixed ordering cost. (In an s-type policy, if the inventory x at the end of a period is less than a threshold s, we order the quantity, $s - x$, to bring inventory up to s. In an (s, S)-type policy, where $s < S$, if the ending inventory x is less than s, we order $S - x$. (s, S) policies are used when there is a fixed ordering cost, which is justified only when inventory falls below s.)

For a discussion of s-type policies and (s, S) policies, their use in the context of general inventory theory under the category of periodic review single location stochastic demand models, and how they are computed, refer to Lee and Nahmias [20], Nahmias [23], Bertsekas [4]. Scarf [30] proved the optimality of (s, S) policies.

Clark and Scarf discuss the problem with extending their result to networks that are not serial. This is because the penalty function for having inadequate inventory at a supply facility which ships to more than one location cannot be determined if there is an inventory imbalance at those locations.

Research in the area of multi-echelon distribution policies under uncertainty related to Clark and Scarf has been extended by Federgruen and Zipkin [13], and others.

For the deterministic distribution problem we are interested in, if we have a series network, we can use the Clark and Scarf result if we can find the s and (s, S) policies. For deterministic demand, (s, S) policies can be found using a dynamic lot-size model with backlogging, which we will discuss next.

Dynamic Lot-Size Models

The single location inventory replenishment problem for deterministic time-varying demand is equivalent to the dynamic lot-size problem used in production planning. For a general discussion of these models, see Ahuja et al [1] and Lee and Nahmias [20].

Wagner-Whitin [38] formulated the dynamic lot-size problem for fixed setup (ordering) costs and deterministic, time-varying demand, without backlogging, and solved it with dynamic programming. Because of the computational efforts required to solve large models, the Silver-Meal heuristic was developed (Lee and Nahmias [20]), although the Wagner-Whitin algorithm poses no computational difficulty today.

Wagner and Whitin showed that these lot-size models have a feature that is known in the literature as "exact requirements." This means that any optimal solution must have these characteristics: in any time period, if we ship, we only ship an integral sum of demand requirements for a contiguous sequence of periods. We only order if inventory is zero. Because of fixed setup costs, it is sub-optimal to pay unnecessary holding or penalty costs by shipping fractional orders. As a result, at any point in time, the inventory level and the shipments in process are either zero or a sum of demands. They incorporated this important feature into a dynamic programming algorithm.

Zangwill [39] extended the dynamic lot-size problem to include the case of backlogging, solving it with a DP algorithm. With backlogging, in a generalization of the Wagner-Whitin model, we only order if inventory is less than or equal to zero. The author develops a proof that a concave cost function for this model is minimum at an extreme point of the feasible region.

Zangwill [40],[41] extended the concave cost, backlogging model to a multi-echelon (single source, multiple location) network and provided a dynamic network representation of the single location problem, giving a more intuitive explanation of the exact requirements result. This paper is often referenced for the proof that concave cost functions on a network are minimum at extreme points (also see Ahuja et al [1]). Zangwill provided more DP algorithms for multiple echelons with special structures such as parallel and serial networks,

Veinott [37] formulated the Wagner-Whitin and Zangwill models as Leontief substitution models and provided additional DP algorithms for this class of problems.

Concave Cost Network Flows

With its characterization of optimal solutions at extreme points, Zangwill's paper was the beginning of the minimum concave cost network flow literature. Algorithms that search the extreme points for optimal solutions are thus either exponential in time or are polynomial only for special cases. For example, Zangwill's DP algorithm for a single source network with multiple stores is exponential in the number of stores.

Soland [32] developed a branch and bound algorithm using linear approximations as lower bounds of the concave cost function in a facility location model. Gallo and Sodini [15] developed a vertex-following heuristic that finds a local minimum. Florian and Robillard [14] developed an enumerative search technique for extreme point solutions.

Further research in the Wagner-Whitin and Zangwill models was done by Erickson et al [10], who proposed a send-and-split DP algorithm which solves minimum concave cost network problems in polynomial time for networks which have a special structure.

Balakrishnan and Graves [3] developed a Lagrangian-based heuristic that uses dual ascent and sub-gradient optimization to solve a piecewise linear concave cost network flow problem. This model is applicable to the LTL shipping problem among others.

Chan et al [6] applied the Balakrishnan and Graves method for an LTL problem using a time-extended network with shipping and inventory links. Other research in LTL models has been done by Powell and Sheffi [29] who solve it as a network design problem.

Although the distribution problem we are interested in is a minimum concave cost network flow, we are more particularly interested in a special sub-class of these problems: fixed charge network design models which we will discuss next.

Network Design

Magnanti and Wong [21] provide a comprehensive survey of the research in network design. Ahuja et al [1] give an introduction to this problem, which is that of solving a multi-commodity flow problem, while at the same time specifying the arcs to use in a network in which some of the arcs have a fixed cost.

Using the dynamic network representation described in Chapter 2, the multi-echelon distribution network can be represented as a network design problem, where the commodities are the demands for each store and time period and the fixed cost arcs are the shipping arcs. In this context, the exact requirements feature is more obvious. It would be sub-optimal to split the path of a commodity's flow from the factory source to the final node at a particular store and time period. Each demand commodity is satisfied by a single path over the shipping links and inventory links.

This class of problems has been shown to be NP-hard (Johnson, Lenstra, Rinnooy Kan [17]). Van Roy and Wolsey [36] have developed a framework for formulating valid inequalities for the fixed charge network problem. Pochet and Wolsey [26] apply these results to develop strong LP formulations for the dynamic lot-size problem, which is also a network design problem. Pochet and Wolsey provide a tighter alternative formulation using a facility location formulation that is also given in Nemhauser and Wolsey [25].

Research in algorithms and in the polyhedral structure of these models has resulted in the expected: strong formulations require an exponential number of constraints, DP algorithms perform in exponential time in the worst case, and special cases which can be solved quickly by DP can also be tightly formulated.

Neuro-dynamic Programming

Some motivation for the use of approximate dynamic programming in Chapter 4 was provided by the case study by Van Roy et al [35] using neuro-dynamic programming (see Bertsekas and Tsitsiklis [5]), which compares the results of using NDP to a two echelon s-type heuristic from Nahmias and Smith [24].

1.3 Contributions of the Thesis

For this thesis, we developed:

1. Mathematical programming formulations.

2. Heuristic algorithms.

3. Approximate dynamic programming algorithms.

The results that contribute to advancing the state-of-the-art in this field are:

1. Stronger formulations.
2. Fast, reliable algorithms for single product models.
3. Extensions of the formulations and algorithms for models that have multiple products.

1.4 Structure of the Thesis

The remainder of this thesis is structured as follows. Chapter 2 presents a mathematical programming approach to this problem. It starts with an initial formulation that is standard in the research literature and develops tighter but more complex formulations. Chapter 3 describes a family of algorithms that are motivated by the idea of Inventory Pull systems. Chapter 4 shows how we can use approximate dynamic programming (ADP) to obtain better solutions for the algorithms in Chapter 3. Chapter 5 compares the computational results of each approach: mathematical programming, heuristic algorithms, and ADP algorithms. Chapter 6 summarizes the conclusions, insights, and implementation issues for these formulations and algorithms. The C program code for the algorithms and the model generators is included in the Appendix.

Chapter 2

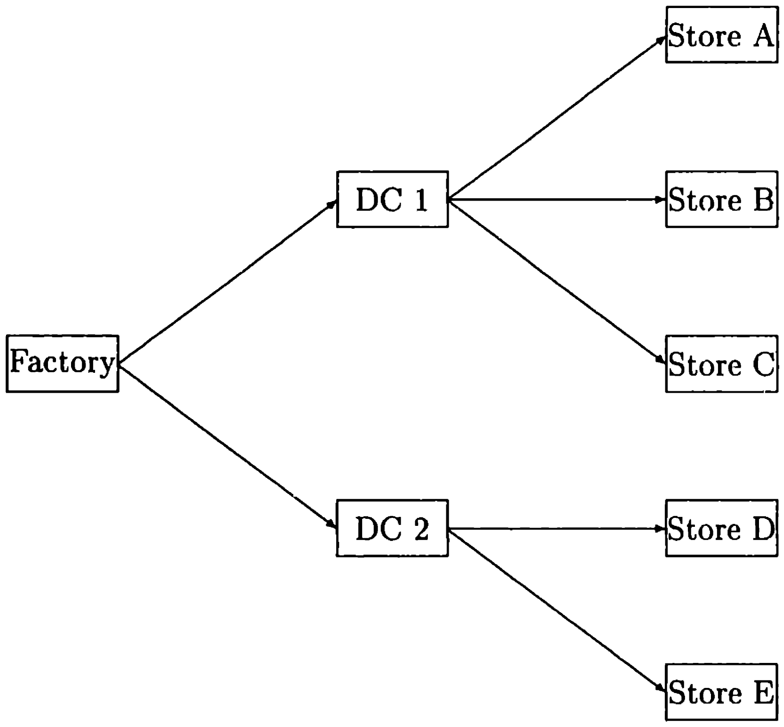
Mathematical Programming Formulations

One approach to solving the type of supply chain optimization problem described in Chapter 1 is to use mathematical programming. In this chapter, we will present formulations of the problem as a mixed integer program, starting with an initial formulation of a type that is commonly described in the literature. We will then develop more complex but tighter formulations. We will formulate the problem for a general, layered network. For the algorithms in Chapter 3 and the computations in Chapter 4, we will assume the network topology to be arborescent, meaning that every node has at most one predecessor node. See Figures 2-1 and 2-2, respectively, for examples of arborescent and non-arborescent networks. We will describe each formulation in a separate section after we first describe the problem in more detail.

2.1 Problem definition

In this section, we will define the terminology, the timing aspects, and details of the costs involved, and we will explain how this problem can be represented by a dynamic network.

Figure 2-1: Arborescent Network



Terminology

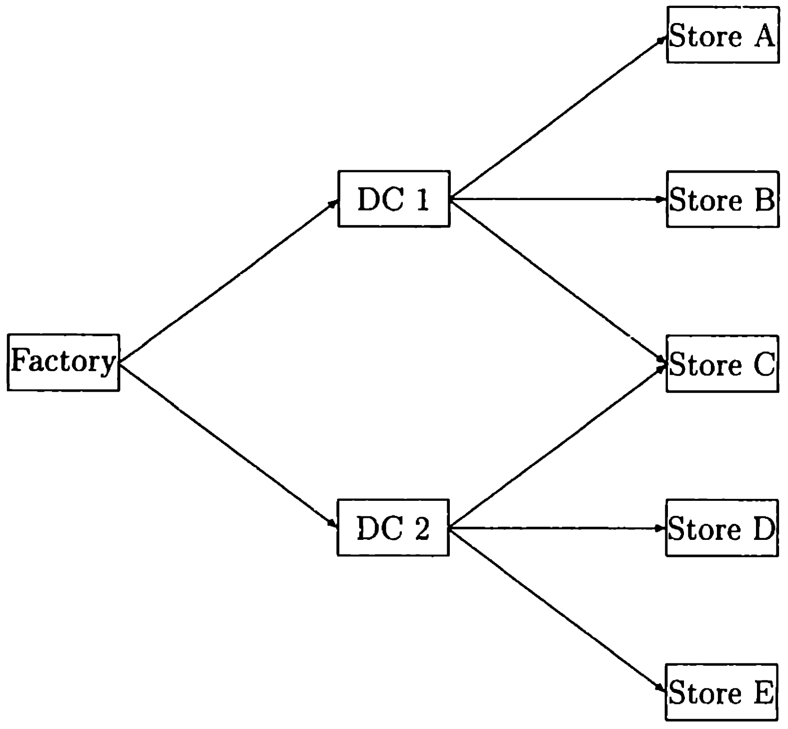
There are three types of location nodes in the distribution network - factories, distribution centers (DC's) (or warehouses), and stores.

Factories ship but do not receive; stores receive but do not ship; DC's receive from factories and ship to stores. Factories have no inventory but are infinite sources for the DC's and stores.

Inventory at the stores can be positive or negative. If inventory is positive, this is excess inventory, for which there is a holding cost. If inventory is negative, this is a shortage of inventory requiring a backorder to meet demand, for which there is a backorder or penalty cost.

An echelon refers to the level of shipments in the network. In a two-echelon network, the first echelon is from the DC to the stores involving what we will call the "DC-to-stores" sub-problem, and the second echelon is from the factory to the DC's involving the "Factory-to-DC" sub-problem.

Figure 2-2: Non-arborescent Network



We will sometimes use the term "lots" when discussing the demand at a store for a particular time period.

Timing of shipments and inventory

The standard practice in inventory theory literature is to use beginning inventory in formulations instead of ending inventory, although holding and penalty costs are functions of ending inventory.

At any non-factory location, in time period t, this is what happens:

1. Add the shipment received to the beginning inventory.

2. Subtract the demand if this is a store.

3. Subtract the shipments to the stores if this is a DC.

4. The remaining ending inventory is the beginning inventory for the next period $(t+1)$.

The time horizon for inventory-related costs is from $t = 1$ to $t = T$, whereas the time horizon for inventory movements is from $t = 0$ to $t = T - 1$. The initial inventory is at time 0, and the final inventory is at time T.

Costs

Inventory holding costs are applied to positive ending inventory. These are typically inventory carrying costs such as capital cost of inventory and warehousing.

Shortage or backorder penalty costs are applied to negative ending inventory. This can be the cost of backordering or a penalty for lost sales.

A fixed shipping cost is incurred whenever there is a shipment. As previously mentioned, we can ignore variable shipping costs. In our model, we can allow any of these costs to vary by time or by location.

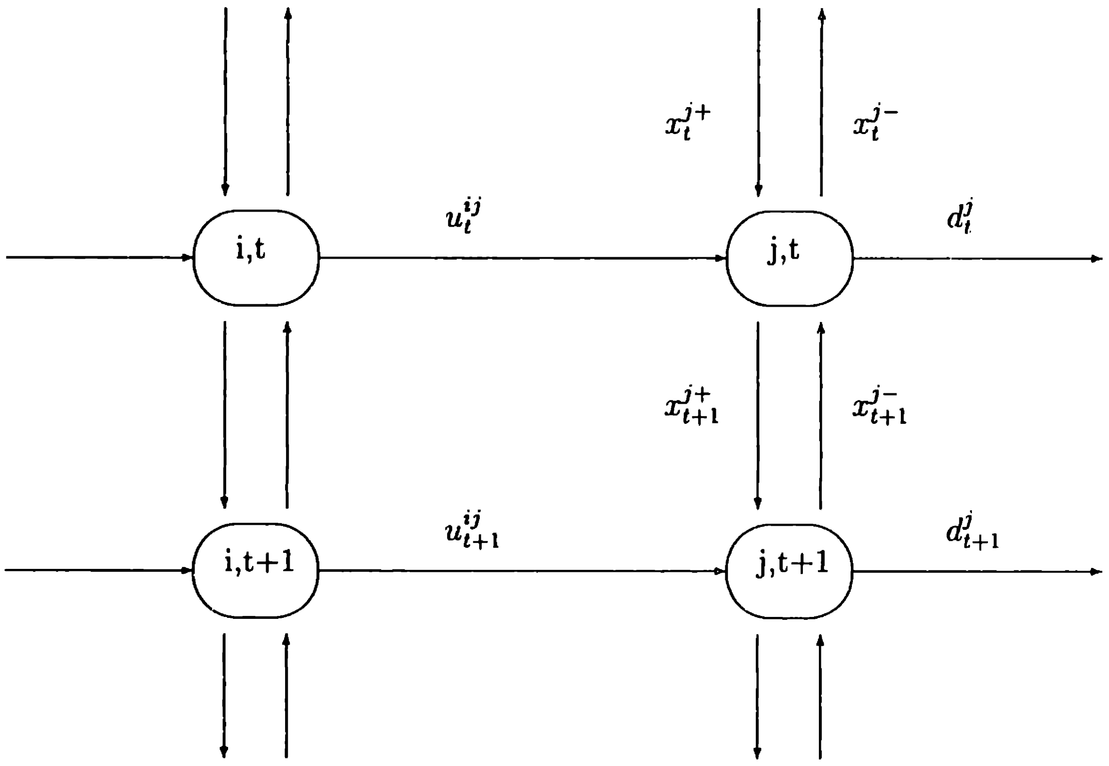
In general, the holding costs are less than the penalty costs, and the fixed shipping costs are substantial enough to force the model occasionally to incur holding or penalty costs. Also, we can typically expect inventory holding costs at stores to exceed those at distribution centers, and the fixed shipping costs to the DC's to exceed those to the stores.

Dynamic Network

The flow of goods from factories to distribution centers to stores can be represented by a network, which we will call the physical network. To represent the flow of goods from location to location over time, we add the dimension of time to the network to create a dynamic network or time-expanded network. Each node in the dynamic network then references a location and a point in time (day, week, etc.) and can be denoted by (i, t).

There are two types of arcs in this dynamic network - shipping links and inventory links. Shipments from one location-time to another flow over a shipping link. Inventory from one location-time to the next flow over inventory links. There are two types of these - positive and negative - depending on whether inventory is carried over to the next period or is backordered from the next period. Shipping arcs may

Figure 2-3: Dynamic Network



be denoted as (i, j, t) for shipments from location i to location j, leaving i at time t. Figure 2-3 displays a section of a dynamic network. The shipping and inventory arcs are labeled with the variables that are used in the formulation we will describe next.

2.2 Formulation 1

Our initial formulation for a general network is a combination of a network flow model and a commonly used mixed integer program formulation for a dynamic lot-size model (see Shapiro [31]). This formulation uses decision variables that control shipping and inventory by specifying how much to ship and how much to keep in inventory at each location and each point in time.

We are given a time-expanded dynamic network (N, A) over a fixed time horizon $t = 0, \ldots, T$, where:

1. N_f is the factory node set.

2. N_d is the DC node set.

3. N_s is the retail store node set.

4. $(i, j, t) \in A$ leaves node i at time t to go to node j.

5. d_t^j is the demand at retail node j at time t.

6. h_t^j is the inventory holding cost per unit at node j at time t.

7. p_t^j is the backorder penalty cost per unit at node j at time t.

8. f_{ijt} is the fixed cost for ordering or shipping between nodes i and j at time t.

Furthermore, M_j is the total downstream demand at node j over all time periods:

$$M_j = \sum_{k \in D_j} \sum_{t=0}^{T-1} d_t^k \quad \text{where} \quad D_j = \text{descendants of } j.$$

The decision variables are as follows:

1. x_t^{j+} is the excess inventory at node j at time t.

2. x_t^{j-} is the inventory shortage (that is backlogged) at node j at time t.

3. x_t^j is the net beginning inventory $(= x_t^{j+} - x_t^{j-})$.

4. u_{ijt} is the quantity shipped along shipping arc (i, j, t).

5. z_{ijt} is a binary indicator variable that decides if we ship from i to j at time t.

The formulation is as follows:

$$\text{minimize} \quad \sum_{t=1}^{T} \sum_{j \notin N_f} (h_t^j x_t^{j+} + p_t^j x_t^{j-}) + \sum_{t=0}^{T-1} \sum_{(i,j,t) \in A} f_{ijt} z_{ijt} \tag{2.1}$$

$$\text{subject to} \quad x_{t+1}^j = x_t^j + \sum_{(i,j,t) \in A} u_{ijt} - d_t^j, \qquad \forall j \in N_s, \quad \forall t = 0, \ldots, T-1, \tag{2.2}$$

$$x_{t+1}^j = x_t^j + \sum_{(i,j,t)\in A} u_{ijt} - \sum_{(j,k,t)\in A} u_{jkt}, \qquad \forall j \in N_d, \quad \forall t = 0,\ldots,T-1, \tag{2.3}$$

$$u_{ijt} \le M_j z_{ijt}, \qquad \forall (i,j,t) \in A, \quad \forall t = 0,\ldots,T-1, \tag{2.4}$$

$$x_0^j = x_T^j = 0, \qquad \forall j \notin N_f, \tag{2.5}$$

$$x_t^j = x_t^{j+} - x_t^{j-}, \qquad \forall j \in N, \quad \forall t = 0,\ldots,T, \tag{2.6}$$

$$x_t^{j+}, x_t^{j-} \ge 0, \qquad \forall j \in N_s \cup N_d, \quad \forall t = 0,\ldots,T,$$

$$u_{ijt} \ge 0, \qquad \forall (i,j,t) \in A, \quad \forall t = 0,\ldots,T-1,$$

$$z_{ijt} \in \{0,1\}, \qquad \forall (i,j,t) \in A, \quad \forall t = 0,\ldots,T-1.$$

The three components of the objective function (2.1) are the inventory holding cost, the backorder penalty cost, and the fixed ordering or shipment costs.

The constraints (2.2) and (2.3) control the balance of flow at the stores in meeting demand, and at the distribution centers in meeting the stores' replenishment needs. Constraint (2.4) is a forcing constraint - a shipment can only happen if the corresponding binary shipping variable is 1. This forces the model to incur the fixed cost of shipping. Constraint (2.5) causes the model to start and end with zero inventories. Constraint (2.6) decomposes x_t^j into positive and negative components. For n nodes in an arborescent network, we have $4(n-1)T$ variables and $2(n-1)T$ constraints.

If we load this formulation and the data for a given problem directly into an optimization package, we will discover that it has a very weak LP relaxation. This is because of the weak bound in the forcing constraint (2.4) which causes a large gap between the integer solution and the LP lower bound. As a result, a branch-and-bound procedure is impractical for all but the smallest problems. As an example, a problem with one factory, one DC, 10 stores, and 10 time periods had an optimal MIP solution of 4885 but an LP solution of 600. This large gap rendered it impractical to solve with this formulation. This motivated the search for a tighter formulation.

2.3 Formulation 2

Formulation 1 uses decision variables for each time period for the shipments between locations and the inventory at each location. The weak forcing constraint bound, however, causes a large gap between the LP relaxation and the optimal solution. The single location lot-size formulation, which our initial formulation is based on, can be improved by a variation of a facility location formulation (Pochet and Wolsey([26] and Nemhauser and Wolsey [25]) with the result that the LP relaxation is integral. The formulation we present in this section is an extension of this method. For this formulation, we will assume there are two echelons.

In Formulation 2, we have the same demands and cost parameters as in Formulation 1, but the new decision variables are as follows:

1. q^t_{jks} is the quantity shipped from DC j to store k at time s for the purpose of meeting that store's demand in period t.

2. q^t_{ijs} is the quantity shipped from the factory i to DC j at time s to meet that DC's demand in period t - i.e., the total demand requirement imposed by the stores on that DC in period t.

3. z_{ijt} is the binary indicator variable for whether we ship from location i to location j at time t.

In addition, we will let $P(i)$ be the set of immediate predecessors of node i - more specifically, $P(i) = \{j \in N : (j, i, t) \in A, \forall t\}$. Similarly, we will let $F(i)$ be the set of immediate followers of node i - i.e., $F(i) = \{j \in N : (i, j, t) \in A, \forall t\}$.

Formulation 2 is as follows:

$$\text{minimize} \sum_{k\in N_d\cup N_s}\sum_{j\in P(k)}\left\{\sum_{t=0}^{T-1} f_{jkt}z_{jkt}+\sum_{t=1}^{T-1}\sum_{s=0}^{t-1}(h_s^k+\cdots+h_{t-1}^k)q_{jks}^t+\right.$$
$$\left.\sum_{t=0}^{T-2}\sum_{s=t+1}^{T-1}(p_{t+1}^k+\cdots+p_s^k)q_{jks}^t\right\} \qquad (2.7)$$

$$\text{subject to} \quad \sum_{j\in P(k)}\sum_{s=0}^{T-1} q_{jks}^t = d_t^k, \qquad \forall k\in N_s, \forall t=0,\ldots,T-1. \qquad (2.8)$$

$$\sum_{i\in P(j)}\sum_{r=0}^{T-1} q_{ijr}^s = \sum_{k\in F(j)}\sum_{t=0}^{T-1} q_{jks}^t, \qquad \forall j\in N_d, \forall s=0,\ldots,T-1, \qquad (2.9)$$

$$q_{jks}^t \le d_t^k z_{jks}, \qquad \forall k\in N_s, \forall j\in P(k), \quad \forall s,t=0,\ldots,T-1, \qquad (2.10)$$

$$q_{ijs}^t \le \Big(\sum_{k\in F(j)}\sum_{r=0}^{T-1} d_r^k\Big) z_{ijs}, \qquad \forall j\in N_d, \forall i\in P(j), \quad \forall s,t=0,\ldots,T-1, \qquad (2.11)$$

$$q_{ijs}^t \ge 0, \qquad \forall t=0,\ldots,T-1, \forall (i,j,s)\in A,$$

$$z_{ijt}\in\{0,1\}, \qquad \forall (i,j,t)\in A.$$

The objective function (2.7) has a similar fixed shipping cost as before, but the inventory holding costs and the backorder penalty costs are applied to the q_{jks}^t variables according to whether $s < t$ (holding) or $s > t$ (backorder penalty). If $s = t$, then there is no holding or penalty cost.

The new formulation gives us balance of flow and the forcing constraints required for the fixed shipping costs:

1. Constraint (2.8) ensures that the shipments to the stores meet the store demands.

2. Constraint (2.9) maintains flow balance at each DC.

3. Constraint (2.10) is a forcing constraint whereby q_{jks}^t is positive only if $z_{jks} = 1$ in which case we incur the fixed shipment cost to the store.

4. Constraint (2.11) is a forcing constraint for shipments from the factory to each DC.

By adding another time dimension to the model, we make a significant increase in its size. If there are m arcs and n nodes in the physical network, Formulation 2 has $mT^2 + mT$ variables and $mT^2 + (n-1)T$ constraints. However, we achieve a much tighter bound in (2.10) because the q^t_{jks} variables enable us to use d^k_t as a coefficient instead of the total demand, M_k. This gives us a tighter LP relaxation, which enables us to use Formulation 2 to solve larger problems than Formulation 1. In the example given for Formulation 1, where the LP solution was 600 and the MIP solution was 4885, Formulation 2 has an LP solution of 4023. We benefit from the fact that the q^t_{jks} variables carry more information for the model.

However, in constraint (2.11), we have to use the DC's total downstream demand as the coefficient in the forcing constraints. This gives us a similar problem as before, which causes performance limitations as we increase the problem size. This prompts us to consider improving this formulation further, which we will discuss next.

2.4 Formulation 3

In Formulation 2, the decision variables q^t_{jks} control shipments from node j to node k at time s that will meet demand at node k at time t. This enables us to have tight forcing bounds from the DC's to the stores, but not from the factory to the DC's. In this section, we will enhance this formulation further by using decision variables to control how much is shipped along every arc (i, j, s) that is destined to satisfy the demand at store k at time t. If we do this, we also have to balance the flow of the store demand-specific inventory at each DC.

For this new Formulation 3, we have a general network with two or more echelons and an unrestricted number of DC's and warehouses that move goods from the factory to the stores. The decision variables are now:

1. q^{kt}_{ijs} is the quantity shipped along arc (i, j, s) to meet the demand at store k at time t.

2. z_{ijs} is the forcing variable for arc (i, j, s).

3. x^{kt}_{js} is the inventory at node j (DC or warehouse) at time s that will go to store k to satisfy demand at time t.

Formulation 3 is as follows:

$$\text{minimize} \quad \sum_{(i,j,s)\in A} f_{ijs} z_{ijs} + \sum_{k\in N_s} \sum_{j\in N_d} \sum_{s=0}^{T-1} \sum_{t=0}^{T-1} h^j_s x^{kt}_{js} +$$

$$\sum_{k\in N_s} \sum_{i\in P(k)} \left\{ \sum_{t=1}^{T-1} \sum_{s=0}^{t-1} (h^k_s + \cdots + h^k_{t-1}) q^{kt}_{iks} + \sum_{t=0}^{T-2} \sum_{s=t+1}^{T-1} (p^k_{t+1} + \cdots + p^k_s) q^{kt}_{iks} \right\} \qquad (2.12)$$

$$\text{subject to} \qquad \sum_{i\in P(k)} \sum_{s=0}^{T-1} q^{kt}_{iks} = d^k_t, \qquad \forall k \in N_s, \quad \forall t = 0, \ldots, T-1, \qquad (2.13)$$

$$\sum_{j\in F(i)} \sum_{s=0}^{T-1} q^{kt}_{ijs} = d^k_t, \qquad \forall k \in N_s, \quad \forall i \in N_f, \quad \forall t = 0, \ldots, T-1, \qquad (2.14)$$

$$q^{kt}_{ijs} \le d^k_t z_{ijs}, \qquad \forall t = 0, \ldots, T-1, \quad \forall k \in N_s, \quad \forall (i, j, s) \in A, \qquad (2.15)$$

$$x^{kt}_{js} + \sum_{i\in P(j)} q^{kt}_{ijs} = x^{kt}_{j,s+1} + \sum_{i\in F(j)} q^{kt}_{jis}, \qquad \forall s, t = 0, \ldots, T-1, \quad \forall k \in N_s, \quad \forall j \in N_d, \qquad (2.16)$$

$$x^{kt}_{j0} = x^{kt}_{jT} = 0, \qquad \forall t = 0, \ldots, T-1, \quad \forall k \in N_s, \quad \forall j \in N_d, \qquad (2.17)$$

$$q^{kt}_{ijs}, x^{kt}_{js} \ge 0, \qquad \forall k \in N_s, \quad \forall (i, j, s) \in A,$$

$$\forall t = 0, \dots, T-1,$$

$$z_{ijt} \in \{0, 1\}, \qquad \forall (i, j, t) \in A.$$

The constraints are similar in purpose to those in Formulation 2:

1. Constraints (2.13) and (2.14) ensure that the shipments meet the demands.
2. Constraint (2.15) is the forcing constraint for each arc.
3. Constraint (2.16) maintains flow balance at each DC or warehouse.
4. Constraint (2.17) sets the initial and final inventories to zero.

As the complexity of the network increases, we increase the number of variables and constraints by incremental orders of T^2, depending on the number of echelons and shipping arcs. For example, even if our network is arborescent, we will have $(2E-1)KT^2$ variables and constraints, where E is the number of echelons and K is the number of stores. To reduce the size of large problems, we could limit the number of time periods for backlogging and for ordering in advance.

As expected, this formulation is very tight. In the example we discussed for Formulations 1 and 2, the LP relaxation for Formulation 3 had an integral optimal solution of 4885. In the test results that will be presented in Chapter 5, the LP relaxation of this formulation yielded integral solutions in 5 out of 6 test problems.

2.5 Formulation 4

Formulation 3 is similar to a network design arc-flow formulation that uses multi-commodity flows (where a commodity in our case is a "lot" or the demand for a store for a time period). It uses a disaggregated forcing constraint, which is tighter than an aggregated forcing constraint (Magnanti and Wong [21]). Following a common practice used with multi-commodity flow problems, we could convert from a node-arc formulation to a path-flow formulation as a way to solve large problems. In this section, we will describe how this could be done, although this formulation was not implemented as part of this research.

Each lot (k,t) - that is, the demand at store k at time t - is shipped from the factory to the store over a path p of shipping arcs and inventory arcs in the dynamic network. We will define P_{kt} to be the set of possible paths for a lot (k,t).

In place of the shipping variables q_{ijs}^{kt} and the inventory variables x_{js}^{kt}, we will use instead a path flow variable y_p^{kt} for the flow over path p that is destined to meet the demand at store k at time t. We will use the same binary forcing variable z_{ijt}. We will assume we have the same cost parameters as before and that we have a way of computing the cost of a path, c_p^k, based on the inventory arcs in the path.

Formulation 4 is as follows:

$$\begin{array}{llll}
\text{minimize} & \displaystyle\sum_{k\in N_s}\sum_{t=0}^{T-1}\sum_{p\in P_{kt}} c_p^{kt} y_p^{kt} + \sum_{(i,j,t)\in A} f_{ijt} z_{ijt} & & (2.18)\\
\text{subject to} & \displaystyle\sum_{p\in P_{kt}} y_p^{kt} = d_t^k, & \forall k \in N_s, & (2.19)\\
 & & \forall t = 0,\dots,T-1, & \\
 & \displaystyle\sum_{p\in P_{kt}:(i,j,s)\in p} y_p^{kt} \le d_t^k z_{ijs}, & \forall k \in N_s, & (2.20)\\
 & & \forall (i,j,s) \in A, & \\
 & & \forall t = 0,\dots,T-1, & \\
 & y_p^{kt} \ge 0, & \forall k \in N_s, & \\
 & & \forall p \in P_{kt}, & \\
 & & \forall t = 0,\dots,T-1, & \\
 & z_{ijt} \in \{0,1\}, & \forall (i,j,t) \in A. &
\end{array}$$

The objective function (2.18) has the inventory costs and the fixed shipping costs. Constraint (2.19) ensures that the demands are met, and constraint (2.20) is the forcing constraint for each shipping arc. A flow balance constraint is not required when we use path-flow variables.

Our new formulation has much fewer constraints but a huge number of variables, now that we have added one for each possible path in the network. We could consider using column generation as a solution approach. To do this, we could start with an initial set of paths that we could obtain from a heuristic algorithm, and then use a shortest path algorithm to "price out" paths in P_{kt} in the sub-network for each store k and each time period t.

2.6 Formulation 5

In this section, we will extend Formulation 3 to the case where there are multiple products. In this situation, we are interested in controlling the shipments and inventories of more than one product. We will assume the products are shipped together and that a shipment of one or more products incurs only one fixed cost. Otherwise we would have separate single-product sub-problems. Except for the shipping variables, z_{ijt}, we will add a super-script for product v to the data and the decision variables which were used in Formulation 3. We will assume there are V products.

The cost parameters and data in the multiple problem are:

1. f_{ijt} is the fixed cost for shipping on arc (i, j, t).

2. h_t^{vj} is the holding cost for product v at location j at time t.

3. p_t^{vj} is the penalty cost for product v at location j at time t.

4. d_t^{vj} is the demand for product v at location j at time t.

The decision variables are as follows:

1. q_{ijs}^{vkt} is the quantity shipped along arc (i, j, s) to meet the demand for product v at store k at time t.

2. z_{ijs} is the forcing variable for arc (i, j, s).

3. x_{js}^{vkt} is the inventory of product v at node j (DC or warehouse) at time s that will go to store k to satisfy demand at time t.

Formulation 5 is as follows:

$$\text{minimize} \quad \sum_{(i,j,s)\in A} f_{ijs} z_{ijs} + \sum_{v=1}^{V} \sum_{k\in N_s} \sum_{j\in N_d} \sum_{s=0}^{T-1} \sum_{t=0}^{T-1} h_s^{vj} x_{js}^{vkt} +$$

$$\sum_{v=1}^{V} \sum_{k\in N_s} \sum_{i\in P(k)} \left\{ \sum_{t=1}^{T-1} \sum_{s=0}^{t-1} (h_s^{vk} + \cdots + h_{t-1}^{vk}) q_{iks}^{vkt} + \sum_{t=0}^{T-2} \sum_{s=t+1}^{T-1} (p_{t+1}^{vk} + \cdots + p_s^{vk}) q_{iks}^{vkt} \right\} \tag{2.21}$$

subject to

$$\sum_{i\in P(k)} \sum_{s=0}^{T-1} q_{iks}^{vkt} = d_t^{vk}, \quad \forall k \in N_s, \ \forall v = 1,\ldots,V, \ \forall t = 0,\ldots,T-1, \tag{2.22}$$

$$\sum_{j\in F(i)} \sum_{s=0}^{T-1} q_{ijs}^{vkt} = d_t^{vk}, \quad \forall k \in N_s, \ \forall v = 1,\ldots,V, \ \forall i \in N_f, \ \forall t = 0,\ldots,T-1, \tag{2.23}$$

$$q_{ijs}^{vkt} \le d_t^{vk} z_{ijs}, \quad \forall t = 0,\ldots,T-1, \ \forall k \in N_s, \ \forall v = 1,\ldots,V, \ \forall (i,j,s) \in A, \tag{2.24}$$

$$x_{js}^{vkt} + \sum_{i\in P(j)} q_{ijs}^{vkt} = x_{j,s+1}^{vkt} + \sum_{i\in F(j)} q_{jis}^{vkt}, \quad \forall s,t = 0,\ldots,T-1, \ \forall k \in N_s, \ \forall v = 1,\ldots,V, \ \forall j \in N_d, \tag{2.25}$$

$$x_{j0}^{vkt} = x_{jT}^{vkt} = 0, \quad \forall t = 0,\ldots,T-1, \ \forall k \in N_s, \ \forall v = 1,\ldots,V, \ \forall j \in N_d, \tag{2.26}$$

$$
\begin{aligned}
q_{ijs}^{vkt}, x_{js}^{vkt} \geq 0, \qquad & \forall k \in N_s, \\
& \forall v = 1, \ldots, V, \\
& \forall (i,j,s) \in A, \\
& \forall t = 0, \ldots, T-1, \\
z_{ijt} \in \{0,1\}, \qquad & \forall (i,j,t) \in A.
\end{aligned}
$$

The constraints are the same as those in Formulation 3 except for the added product dimension. In this new formulation, we multiply the number of variables and constraints in Formulation 3 by a factor of V, the number of products.

In summary, the mathematical programming approach requires tighter formulations which greatly increase the numbers of variables and constraints in the model. To solve medium-to-large scale problems, an alternative is to develop heuristic algorithms, which is the subject of the next chapter.

Chapter 3

Algorithms

As we saw in Chapter 2, if we use mathematical programming, our distribution problem requires many variables and constraints, which results in computer memory and execution time problems. Another approach is to develop heuristic algorithms which will provide near optimal solutions quickly without severe memory requirements.

In this chapter, we will present a family of algorithms that are motivated by the Inventory Pull system that was mentioned in Chapter 1. We will start by giving a dynamic lot-size algorithm that is used for all these algorithms. We will then describe the Pull algorithm and four variations that use this algorithm as an initial solution. The computational test results for these algorithms are presented in Chapter 5.

3.1 Single Location Dynamic Lot-Size Algorithm

The family of algorithms that are described in this chapter use a single location dynamic lot-size algorithm, which we present in this section, that is based on Zangwill [39] and Veinott [37]. We will allow retail stores to make the decision to backlog unfilled demand. To avoid complications, we will not allow DC's and warehouses to make a decision to backlog. The reason for doing this is to avoid situations where a store chooses to order in advance to avoid the backorder penalty, but would still have to backorder if the DC chooses to backorder the order from the store.

We assume, similar to a production planning model, that a facility orders and receives shipments u_i from a supplier to meet demand d_i in time periods $i = 1, \ldots, T$ (which we will use in this section instead of $i = 0, \ldots, T-1$). Any unsatisfied demand at a store is backlogged. As before, we assume zero initial and final inventories, zero lead times, shipments have a fixed cost (plus a variable cost which we can ignore because the total variable cost is independent of the shipping schedule), excess inventory has a holding cost, and a shortage incurs a penalty cost.

The single location dynamic lot-size algorithm is based on a key result by Zangwill [39], that is also described by Lee and Nahmias [20], which we state without proof in the following theorem.

Theorem 1 *An optimal production (shipping) schedule $\{u_i\}, i = 1, \ldots, T$ for the dynamic lot-size model with backlogging satisfies the "Exact Requirements" property, which means there exists a sequence of integers, $\{S_i\}, i = 0, \ldots, T$ where*

$$0 = S_0 \le S_1 \le \cdots \le S_T = T.$$

such that:

$$u_i = 0 \quad \text{if} \quad S_{i-1} = S_i,$$

otherwise

$$u_i = \sum_{j=1+S_{i-1}}^{S_i} d_j, \quad \forall i = 1, \ldots, T.$$

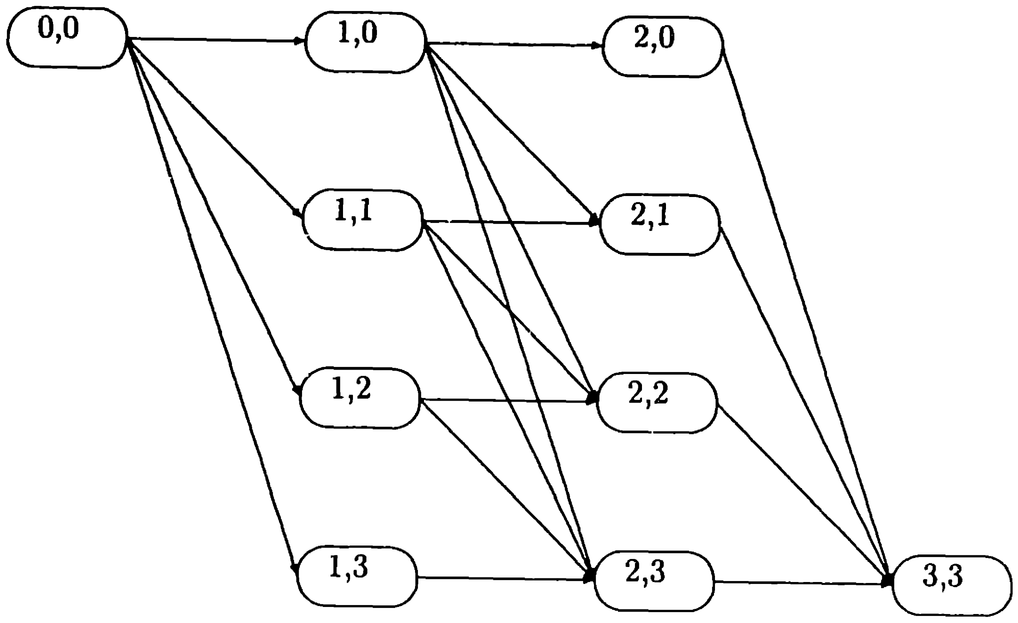
S_i is the number of periods for which the total demand requirement has been received as of period i. If $S_i > i$, we are ordering in advance. If $S_i < i$, we are backordering.

Viewed this way, the optimization problem becomes that of finding the minimum cost monotonic, non-decreasing integer sequence $\{S_i\}$ from 0 to T, where the cost of each sequence can be calculated by (a) accumulating the costs of the inventory positions at each time period i, and (b) accumulating fixed costs for shipments whenever

$S_i > S_{i-1}$.

To accomplish this, we can represent the S_i's in a network (see Figure 3-1) and use dynamic programming, for example, to find the optimal sequence as the shortest path. Using the terminology of dynamic programming, the stage of the system is i and the state is S_i. The nodes in the network representation of this system are labeled (i, S_i). We have nodes (i, S_i) for all $i = 1, \ldots, T-1$ and all $S_i = 0, \ldots, T$, in addition to the initial nodes $(0, 0)$ and the terminal node (T, T). For cases where backlogging is not permitted, we can only have nodes where $i \leq S_i$. Arcs connect node (i, S_i) to node $(i+1, S_k)$ for all $S_i \leq S_k$.

Figure 3-1: (i, S_i) Network for $T = 3$



For every (i, S_i), we define the inventory position, $I(i, S_i)$ as

$$I(i,S_i) = \begin{cases} \sum_{k=1}^{S_i} d_k - \sum_{k=1}^{i} d_k, & \text{if } S_i > 0, \\ -\sum_{k=1}^{i} d_k, & \text{if } S_i = 0. \end{cases}$$

The inventory position is thus the net inventory as calculated by the cumulative demand received minus the cumulative demand at that time.

If h is the inventory holding cost and p is the backorder penalty cost, then the cost of the inventory position $C(i, S_i)$ is

$$C(i,S_i) = h(I(i,S_i))^+ - p(I(i,S_i))^-.$$

If f is the fixed shipment cost, then we define the arc cost from (i, S_i) to $(i+1, S_k)$ in the following way:

$$\text{Cost from} \quad (i,S_i) \quad \text{to} \quad (i+1,S_k) = \begin{cases} C(i+1,S_k), & \text{if } S_k = S_i, \\ f + C(i+1,S_k), & \text{if } S_k > S_i. \end{cases}$$

The optimal $\{S_i\}$ sequence is therefore the shortest path from the starting node $(0,0)$ to the final destination, (T,T) using these arc costs. The shortest path was solved by dynamic programming for this implementation. Some comments about the computer program (see the DP function in Appendix B):

1. The inventory positions and costs for the T^2 nodes are calculated before the execution of the shortest path algorithm.

2. All inventory positions and costs can be calculated in a time of $O(T^2)$ using an outside loop of T time periods and an inside loop of T time periods.

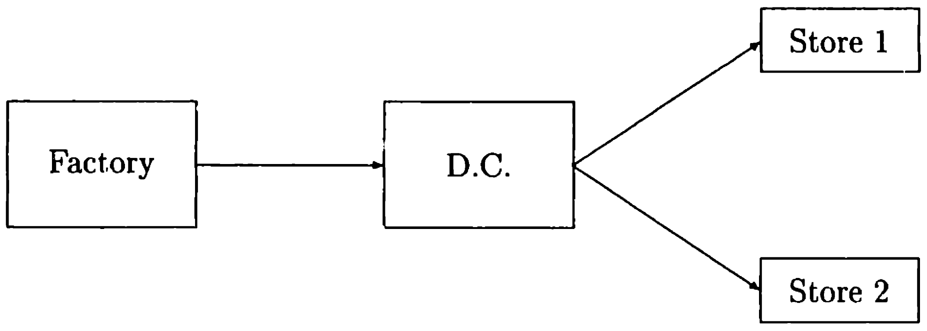
3. The arc costs are calculated on an as-needed basis, which implies that the arc costs do not need to be stored.

3.2 The Basic Pull Algorithm

In this section, we will introduce the Basic Pull algorithm, which will serve as the foundation for all the other algorithms to be described in this chapter and the next. We will explain the motivation for this algorithm, the steps involved, the limitations of the algorithm, and how it works using a small example.

If our network only consists of one store which receives shipments from one factory, then we can solve this problem by using the dynamic lot-size algorithm with backlogging. We can also solve for networks with one factory, no DC's, and multiple stores by solving separate lot-size sub-problems. The first level of complication arises when we have a factory shipping to a DC which ships to one or more stores (see Figure 3-2 below).

Figure 3-2: Two echelon model



If we extend the idea of decomposing the problem into individual lot-size problems, we get the following "Pull" algorithm, which we will describe for the two-echelon, one DC case.

1. For each store k, obtain the optimal shipment schedule from the DC to the store, $\{u_t^k\}_{t=0}^{T-1}$, by solving that store's dynamic lot-size model with backlogging that meets the demand, $\{d_t^k\}_{t=0}^{T-1}$.

2. Accumulate the DC-to-stores shipments by time period to get the replenishment

requirements at the DC:

$$r_t = \sum_{k=1}^{K} u_t^k, \quad t = 0, \ldots, T-1.$$

3. Use $\{r_t\}_{t=0}^{T-1}$ as the DC's demand from the factory and solve the factory-to-DC dynamic lot-size model without backlogging to get the shipment schedule from the factory to the DC.

This algorithm follows the Just-In-Time method where the stores "pull" the demand through the network. It is "greedy" from the stores' perspective. The algorithm can be extended to more than two echelons by continuing the process of pulling from the next echelon upstream in the same way. We can use the Pull algorithm for non-arborescent networks by having each location pull only from one preferred supply location. This method would have obvious limitations, but it could provide a good starting point for an algorithm.

The Pull algorithm is sub-optimal when the DC-to-store replenishment requirement vector $\boldsymbol{r} = (r_0, \ldots, r_{T-1})$ requires shipments from the factory to the DC which make the total system costs sub-optimal. The solution is optimal for each store but there are excessive costs at the second echelon (from the factory to the DC). If the Pull algorithm could find the right $\boldsymbol{r}$, we would get the optimal solution.

As mentioned in the previous section and also in Chapter 1, optimal solutions for problems of this type have the Exact Requirements property whereby shipments are always either zero or a sum of downstream demands. We could start with the vector $\boldsymbol{r}$ that the Pull algorithm generates and try to modify it, while maintaining the Exact Requirements feature, in order to improve the solution. This is the motivation for the next four algorithms we will discuss. The first two look for ways to shift the DC-to-stores shipment schedule in such a way that the resulting $\boldsymbol{r}$ gives a better solution. The last two try to influence the DC-to-stores schedules by applying penalty prices. All four algorithms start with the Pull algorithm.

Example

We will illustrate the algorithms with a simple example of a two-echelon network with one factory, one DC, and two stores. We will use a holding cost of \$2 per unit and a backorder penalty cost of \$5 per unit at each location. The fixed cost of shipping is \$150 from the factory to the DC and \$50 from the DC to the stores. The demand requirement is (15,15,10,10,5) for Store 1 and (5,10,15,20,25) for Store 2 for time periods 0,1,2,3,4. The quantities and dollars in this example were chosen for the purpose of illustrating the different features of the algorithms.

Table 3.1 displays the result of the Pull algorithm for this example. The row and column entries are self-explanatory. The costs are displayed next to the quantities involved - e.g., for store 2, at time period 1 an inventory of -5 is a shortage that is backordered, costing \$25; at time period 2, an inventory of 15 has a holding cost of \$30; and at time period 3, the shipment of 20 from the DC has a fixed cost of \$50. As an example of the transition from one time period to the next, at the DC we start with a beginning inventory of 0, receive 60 from the factory, ship a total of 30 to the two stores, resulting in a beginning inventory of 30 at time period 1. The row, "Shipments to Stores" is the previously mentioned $\boldsymbol{r}$ vector.

Since the Pull algorithm optimizes each store's shipment schedule, then \$170 and \$205 are the minimum achievable costs for the stores. However the DC is forced to incur extra costs to satisfy the accumulated shipments to the stores of (30,30,25,20,25) because the stores' shipment schedules of (30,0,25,0,0) and (0,30,0,20,25) are not synchronized. The optimal solution is given in Table 3.2. The store costs in the optimal solution are a total of \$175 higher than in the Pull algorithm solution, but the timing of the shipments cost the DC \$350 less for a net decrease of \$175. The algorithms which will be described next will attempt to balance the shipments in the solution to the Pull algorithm in a way that will cost the stores more but result in significant savings for the DC.

Table 3.1: Basic Pull Algorithm

	0	1	2	3	4	Total
DC						
Beginning Inventory/$	0/0	30/60	0/0	45/90	25/50	200
Ship: Factory to DC/$	60/150	0/0	70/150	0/0	0/0	300
Shipments to Stores	30	30	25	20	25	
DC Total $	150	60	150	90	50	500
Store 1						
Beginning Inventory/$	0/0	15/30	0/0	15/30	5/10	70
Ship: DC to Store/$	30/50	0/0	25/50	0/0	0/0	100
Demand	15	15	10	10	5	
Store 1 Total $	50	30	50	30	10	170
Store 2						
Beginning Inventory/$	0/0	-5/25	15/30	0/0	0/0	55
Ship: DC to Store/$	0/0	30/50	0/0	20/50	25/50	150
Demand	5	10	15	20	25	
Store 2 Total $	0	75	30	50	50	205
Grand Total $	200	165	230	170	110	875

Table 3.2: Optimal Solution

	0	1	2	3	4	Total
DC						
Beginning Inventory/$	0/0	0/0	0/0	0/0	0/0	0
Ship: Factory to DC/$	0/0	130/150	0/0	0/0	0/0	150
Shipments to Stores	0	130	0	0	0	
DC Total $	0	150	0	0	0	150
Store 1						
Beginning Inventory/$	0/0	-15/75	25/50	15/30	5/10	165
Ship: DC to Store/$	0/0	55/50	0/0	0/0	0/0	50
Demand	15	15	10	10	5	
Store 1 Total $	0	125	50	30	10	215
Store 2						
Beginning Inventory/$	0/0	-5/25	60/120	45/90	25/50	285
Ship: DC to Store/$	0/0	75/50	0/0	0/0	0/0	50
Demand	5	10	15	20	25	
Store 2 Total $	0	75	120	90	50	335
Grand Total $	0	350	170	120	60	700

3.3 Single Shift Algorithm

The algorithm we will discuss in this section makes lot-by-lot adjustments to the DC-to-store shipment schedules (recalling that a lot is the demand at a store at a given time period). These are the steps in the Single Shift algorithm:

1. Perform the Pull algorithm.
2. Do the following for each store and each time period:
   (a) Find the earliest time period and the latest time period that this lot can be shipped as an alternative to its current scheduled shipment time.
   (b) For each time period from the earliest to the latest alternative shipping periods, calculate a trial solution by assuming the shipment for this lot has been shifted to this period, recalculating the costs for this store, and executing the factory-to-DC portion of the Pull algorithm.
   (c) Keep the lower bound for these trial solutions.
   (d) If the lower bound is less than the current solution, replace the current solution with it.

See Table 3.3 to see how this algorithm improves upon the Basic Pull algorithm. The schedule for Store 1 is unchanged, but the lots for Store 2 have been shifted: 15 from period 1 to 2 and 25 from period 4 to 3. This increases Store 2's total cost from \$205 to \$225, but the new replenishment requirement for the DC of (30,15,40,45,0) allows the DC's total cost to drop from \$500 to \$420 for a net decrease of \$60.

3.4 Multiple Shift Algorithm

A limitation with the Single Shift algorithm is that extra fixed costs would be required for many alternative schedules for which we only look at shipping individual lots earlier or later. For some cases, we may want to improve the solution by shifting

Table 3.3: Single Shift Algorithm

	0	1	2	3	4	Total
DC						
Beginning Inventory/$	0/0	15/30	0/0	45/90	0/0	120
Ship: Factory to DC/$	45/150	0/0	85/150	0/0	0/0	300
Shipments to Stores	30	15	40	45	0	
DC Total $	150	30	150	90	0	420
Store 1						
Beginning Inventory/$	0/0	15/30	0/0	15/30	5/10	70
Ship: DC to Store/$	30/50	0/0	25/50	0/0	0/0	100
Demand	15	15	10	10	5	
Store 1 Total $	50	30	50	30	10	170
Store 2						
Beginning Inventory/$	0/0	-5/25	0/0	0/0	25/50	75
Ship: DC to Store/$	0/0	15/50	15/50	45/50	0/0	150
Demand	5	10	15	20	25	
Store 2 Total $	0	75	50	50	50	225
Grand Total $	200	135	250	170	60	815

entire shipments which may consist of multiple lots, which is the method we describe in this section.

The steps in the Multiple Shift algorithm are as follows:

1. Perform the Pull algorithm.

2. For each store, do the following steps for each time period for which there is a scheduled shipment:

   (a) Recalculate the store's cost and execute the factory-to-DC portion of the Pull algorithm, assuming the entire shipment to the store has been shifted either one period earlier or one period later.

   (b) Keep the lower bound for these trial solutions.

   (c) If the lower bound is less than the current solution, replace the current solution with it.

Table 3.4 is a display of the results of this algorithm for our two store example. Store 2 stays unchanged from the the Pull solution. For Store 1, the entire shipment

Table 3.4: Multiple Shift Algorithm

	0	1	2	3	4	Total
DC						
Beginning Inventory/$	0/0	30/60	0/0	0/0	25/50	110
Ship: Factory to DC/$	60/150	0/0	0/0	70/150	0/0	300
Shipments to Stores	30	30	0	45	25	
DC Total $	150	60	0	150	50	410
Store 1						
Beginning Inventory/$	0/0	15/30	0/0	-10/50	5/10	90
Ship: DC to Store/$	30/50	0/0	0/0	25/50	0/0	100
Demand	15	15	10	10	5	
Store 1 Total $	50	30	0	100	10	190
Store 2						
Beginning Inventory/$	0/0	-5/25	15/30	0/0	0/0	55
Ship: DC to Store/$	0/0	30/50	0/0	20/50	25/50	150
Demand	5	10	15	20	25	
Store 2 Total $	0	75	30	50	50	205
Grand Total $	200	165	30	300	110	805

of 25 gets shifted from time period 2 to 3. As a result, Store 1's total cost goes from \$170 to \$190, but the new replenishment requirement of (30,30,0,45,25) at the DC lowers the DC's costs from \$500 to \$410 for a net savings of \$70.

3.5 Pricing Algorithm

The two preceding algorithms look for ways to adjust the DC's replenishment requirements vector $\boldsymbol{r}$ from the DC-to-store end by shifting either lots or entire shipments. The algorithm which we discuss in this section looks at potentially beneficial shifts at the Factory-to-DC end and uses that information in a simple pricing mechanism to influence the shipment schedules from the DC to the stores.

The steps in the Pricing algorithm are as follows:

1. Perform the Pull algorithm.

2. For each time period in which there is a shipment from the factory to the DC, do the following:

(a) Execute the factory-to-DC sub-problem, assuming the entire shipment to the DC has been shifted either one period earlier or one period later.

(b) Keep the lower bound for these trial sub-problem solutions.

(c) If the lower bound is less than the current solution for the factory-to-DC sub-problem, then do the following:

   i. Calculate the potential savings per unit at the DC and use this as a price premium for shipments to the stores in the time period that had the lower bound.

   ii. Execute the Pull algorithm, using this price premium.

   iii. Recalculate the cost of this solution without the prices.

   iv. If the new solution is less than the current solution, replace the current solution with it.

In this algorithm, the stores propose a schedule to the DC. The DC sees if a shift in this schedule would be less costly from its perspective; then it uses premium pricing to try to influence the stores to change their schedules. Another alternative would be to use discounts or a combination of premium and discounts. In the next section, we will present a mathematical programming approach to pricing, but first we will discuss the results for our two store example.

For the example, after the initial Pull solution (iteration 1), this pricing algorithm executed two more iterations before terminating. The results of these iterations are displayed in Table 3.5 and Table 3.6. The initial Pull solution resulted in a replenishment vector of (30,30,25,20,25). This algorithm found that from the DC's perspective, it would be better to shift 25 from time period 2 to 3 - i.e., the DC would prefer (30,30,0,45,25) and so it placed a premium price on time period 2. As a result, neither store ordered and received shipments in period 2.

After iteration 2, the replenishment vector was (15,55,0,35,25). The DC preferred a shift of the 55 units from period 1 to period 0 and thus priced period 1 at a premium. The result is in Table 3.6. Since no other improving results could be found, this was the final solution.

Table 3.5: Pricing Algorithm - Second Iteration

	0	1	2	3	4	Total
DC						
Beginning Inventory/$	0/0	55/110	0/0	0/0	25/50	160
Ship: Factory to DC/$	70/150	0/0	0/0	60/150	0/0	300
Shipments to Stores	15	55	0	35	25	
DC Total $	150	110	0	150	50	460
Store 1						
Beginning Inventory/$	0/0	0/0	10/20	0/0	5/10	30
Ship: DC to Store/$	15/50	25/50	0/0	15/50	0/0	150
Demand	15	15	10	10	5	
Store 1 Total $	50	50	20	50	10	180
Store 2						
Beginning Inventory/$	0/0	-5/25	15/30	0/0	0/0	55
Ship: DC to Store/$	0/0	30/50	0/0	20/50	25/50	150
Demand	5	10	15	20	25	
Store 2 Total $	0	75	30	50	50	205
Grand Total $	200	235	50	250	110	845

Table 3.6: Pricing Algorithm - Final Iteration

	0	1	2	3	4	Total
DC						
Beginning Inventory/$	0/0	0/0	0/0	0/0	25/50	50
Ship: Factory to DC/$	70/150	0/0	0/0	60/150	0/0	300
Shipments to Stores	70	0	0	35	25	
DC Total $	150	0	0	150	50	350
Store 1						
Beginning Inventory/$	0/0	25/50	10/20	0/0	5/10	80
Ship: DC to Store/$	40/50	0/0	0/0	15/50	0/0	100
Demand	15	15	10	10	5	
Store 1 Total $	50	50	20	50	10	180
Store 2						
Beginning Inventory/$	0/0	25/50	15/30	0/0	0/0	80
Ship: DC to Store/$	30/50	0/0	0/0	20/50	25/50	150
Demand	5	10	15	20	25	
Store 2 Total $	50	50	30	50	50	230
Grand Total $	250	100	50	250	110	760

3.6 Lagrangian-based Pull Algorithm

In this section, we will discuss another pricing-oriented algorithm that starts with the Basic Pull solution. Since we will describe this algorithm for the case of an arborescent, two-echelon model, we will begin by restating Formulation 1. In doing so, we can use a simpler indexing scheme by replacing arc indices with node indices. To avoid confusion, we will try to use i for a DC, j for a store (or for a node in general), and k for the factory. As defined in the previous chapter, $F(i)$ is the set of immediate follower nodes of node i. And, $i = \text{pred}(j)$ means that node i is the immediate predecessor of node j.

$$
\begin{array}{llll}
\text{minimize} & \sum_{t=1}^{T} \sum_{j \notin N_f} (h_t^j x_t^{j+} + p_t^j x_t^{j-}) + \sum_{t=0}^{T-1} \sum_{j \notin N_f} f_t^j z_t^j & & (3.1) \\
\text{subject to} & x_{t+1}^j = x_t^j + u_t^{ij} - d_t^j, & \forall j \in N_s, & (3.2) \\
& & i = \text{pred}(j), & \\
& & \forall t = 0, \dots, T-1, & \\
& x_{t+1}^i = x_t^i + u_t^{ki} - r_t^i, & \forall i \in N_d, & (3.3) \\
& & k = \text{pred}(i), & \\
& & \forall t = 0, \dots, T-1, & \\
& \sum_{t=0}^{T-1} r_t^i = \sum_{j \in F(i)} \sum_{t=0}^{T-1} d_t^j, & (= M_i) \quad \forall i \in N_d, & (3.4) \\
& r_t^i = \sum_{j \in F(i)} u_t^{ij}, & \forall i \in N_d, & (3.5) \\
& & \forall t = 0, \dots, T-1, & \\
& u_t^{ij} \le M_j z_t^j, & \forall (i,j,t) \in A, & (3.6) \\
& & \forall t = 0, \dots, T-1, & \\
& x_0^j = x_T^j = 0, & \forall j \notin N_f, & (3.7) \\
& x_t^j = x_t^{j+} - x_t^{j-}, & \forall j \in N, & (3.8) \\
& & \forall t = 0, \dots, T, & \\
& x_t^{j+}, x_t^{j-} \ge 0, & \forall j \in N_s \cup N_d, &
\end{array}
$$

$$\begin{aligned} & && \forall t = 0, \ldots, T, \\ & r_t^i \geq 0, && \forall i \in N_d, \\ & && \forall t = 0, \ldots, T-1, \\ & u_t^{ij} \geq 0, && \forall (i,j,t) \in A, \\ & && \forall t = 0, \ldots, T-1, \\ & z_t^j \in \{0,1\}, && \forall (i,j,t) \in A, \\ & && \forall t = 0, \ldots, T-1. \end{aligned}$$

The major difference between this formulation and Formulation 1 is the introduction of a decision variable r_t^i and the linking constraints (3.5).

If we relax the linking constraint (3.5), using λ_t^i as the dual multipliers for DC i at time t, we add the following terms to the objective function:

$$\lambda_t^i(\sum_i u_t^{ij} - r_t^i) \quad \text{where} \quad \lambda_t^i \quad \text{is sign free.}$$

In so doing, this problem splits into these sub-problems:

1. Individual DC-to-store sub-problems for each store.

2. Factory-to-DC sub-problem.

The individual DC-to-store sub-problem for store j is:

$$\begin{aligned} \text{minimize} \quad & \sum_{t=1}^{T}(h_t^j x_t^{j+} + p_t^j x_t^{j-}) + \sum_{t=0}^{T-1}(f_t^j z_t^j + \lambda_t^i u_t^{ij}) && i = \text{pred}(j) \\ \text{subject to} \quad & x_{t+1}^j = x_t^j + u_t^{ij} - d_t^j, && i = \text{pred}(j), \forall t = 0, \ldots, T-1, \\ & u_t^{ij} \leq M_j z_t^j, && i = \text{pred}(j), \forall t = 0, \ldots, T-1, \\ & x_0^j = x_T^j = 0, && \\ & x_t^j = x_t^{j+} - x_t^{j-}, && \forall t = 0, \ldots, T, \\ & x_t^{j+}, x_t^{j-} \geq 0, && \forall t = 0, \ldots, T, \\ & u_t^{ij} \geq 0, && i = \text{pred}(j), \forall t = 0, \ldots, T-1, \\ & z_t^j \in \{0,1\}, && \forall t = 0, \ldots, T-1. \end{aligned}$$

As mentioned before, this sub-problem can be solved by the algorithm for the dynamic lot-size model with backlogging.

The Factory-to-DC sub-problem for DC i is:

$$
\begin{array}{lll}
\text{minimize} & \sum_{t=1}^{T}(h_t^i x_t^{i+} + p_t^i x_t^{i-}) + \sum_{t=0}^{T-1} f_t^i z_t^i - \sum_{t=0}^{T-1} \lambda_t^i r_t^i & \\
\text{subject to} & x_{t+1}^i = x_t^i + u_t^{ki} - r_t^i, & k = \text{pred}(i), \forall t = 0, \ldots, T-1, \\
& \sum_{t=0}^{T-1} r_t^i = \sum_{j \in F(i)} \sum_{t=0}^{T-1} d_t^j & (= M_i), \\
& r_t^i \geq 0, & \forall t = 0, \ldots, T-1, \\
& u_t^{ki} \leq M_i z_t^i, & k = \text{pred}(i), \forall t = 0, \ldots, T-1, \\
& x_0^i = x_T^i = 0, & \\
& x_t^i = x_t^{i+} - x_t^{i-}, & \forall t = 0, \ldots, T, \\
& x_t^{i+}, x_t^{i-} \geq 0, & \forall t = 0, \ldots, T, \\
& u_t^{ki} \geq 0, & k = \text{pred}(i), \forall t = 0, \ldots, T-1, \\
& z_t^i \in \{0, 1\}, & \forall t = 0, \ldots, T-1.
\end{array}
$$

This sub-problem has a trivial solution where we ship all of M_i from the factory to the DC in the time period which has the maximum λ_t^i. By doing so, we only incur the fixed shipping cost once, the inventory costs are zero, and the λ - based costs are minimized.

The Lagrangian relaxation can be solved by using the method by Everett [11] which is described in Appendix A. This method is folded into the solution method for the Lagrangian-based heuristic, which we describe next, in which the Lagrangian multipliers used in solving the relaxation are used as additional prices in the algorithm.

These are the steps in the algorithm:

1. Start with an initial set of λ - multipliers.

2. Perform the following steps either in a loop with a counter or until the multipliers converge within a tolerance limit.

Table 3.7: Lagrangian Pull Algorithm

	0	1	2	3	4	Total
DC						
Beginning Inventory/$	0/0	0/0	0/0	0/0	0/0	0
Ship: Factory to DC/$	45/150	0/0	85/150	0/0	0/0	300
Shipments to Stores	45	0	85	0	0	
DC Total $	150	0	150	0	0	300
Store 1						
Beginning Inventory/$	0/0	15/30	0/0	15/30	5/10	70
Ship: DC to Store/$	30/50	0/0	25/50	0/0	0/0	100
Demand	15	15	10	10	5	
Store 1 Total $	50	30	50	30	10	170
Store 2						
Beginning Inventory/$	0/0	10/20	0/0	45/90	25/50	160
Ship: DC to Store/$	15/50	0/0	60/50	0/0	0/0	100
Demand	5	10	15	20	25	
Store 2 Total $	50	20	50	90	50	260
Grand Total $	250	50	250	120	60	730

(a) Execute the Pull algorithm with the shipment quantities multiplied by the λ prices.

(b) Perform each store's individual sub-problem as described above.

(c) Perform the DC's sub-problem.

(d) The total obtained for the stores plus the DC is then the solution value for the Lagrangian relaxation with these λs.

(e) Save the lower bound for the Pull algorithm and the upper bound for the relaxation.

(f) Update the λs using the Everett method and repeat at step 3.

The results of applying the Lagrangian Pull algorithm to the two store example are given in Table 3.7. The algorithm achieved the best result - $730 as compared to the optimal $700 - of all the algorithms in this chapter. When compared to the initial Pull solution, it can be seen that Store 1 is unchanged, but the shipments to Store 2 have been shifted, at Store 2's expense, to line up with the shipments to Store 1, which greatly benefits the DC.

Chapter 4

An Approximate Dynamic Programming Approach

In this chapter, we will represent the supply chain optimization problem as a dynamic optimization problem which can be solved by dynamic programming. We will then explain how approximate dynamic programming (ADP), using any heuristic algorithm, gives a practical method for finding solutions that also enhances the heuristic. Finally, we will describe how this method was implemented for the algorithms that were discussed in Chapter 3, using the Pull algorithm as a specific example.

4.1 Dynamic Programming Approach

In this section, we will define our supply chain distribution problem as a discrete-time dynamic system that can be solved as an optimal control problem using dynamic programming. For consistency, we will use the same notation that we used for Formulation 1 in Chapter 2. An abbreviated review follows:

We are given a time-expanded dynamic network (N, A), over a fixed time horizon $t = 0, \ldots, T$, where:

1. N_f is the factory node set.

2. N_d is the DC node set.

3. N_s is the retail store node set.

4. $(i,j,t) \in A$ leaves node i at time t to go to node j.

5. d_t^j is the demand at retail node j at time t.

6. h_t^j is the inventory holding cost per unit at node j at time t.

7. p_t^j is the backorder penalty cost per unit at node j at time t.

8. f_{ijt} is the fixed cost for ordering or shipping between nodes i and j at time t.

At any time t, the state of the system is represented by the state vector, $\boldsymbol{x}_t = (x_t^j)$, where x_t^j is the net beginning inventory at location j at time t, which could be negative if there is a backorder. The initial system state is $\boldsymbol{x}_o = (0,\ldots,0)$.

The decision (or control) vector at time t is $\boldsymbol{u}_t = (u_{ijt})$, where u_{ijt} is the non-negative quantity shipped along shipping arc (i,j,t).

Related to $\boldsymbol{u}_t$ is the vector, $\boldsymbol{z}_t = (z_{ijt})$, where z_{ijt} is a binary indicator variable for shipping from i to j at time t. More specifically,

$$z_{ijt} = \begin{cases} 1, & \text{if} \quad u_{ijt} > 0, \\ 0, & \text{if} \quad u_{ijt} = 0. \end{cases}$$

The system state $\boldsymbol{x}_t$ evolves from time t to time $t+1$ according to the following state transition function:

$$x_{t+1}^j = \begin{cases} x_t^j + \sum\limits_{(i,j,t)\in A} u_{ijt} - d_t^j, & \text{if} \quad j \in N_s, \\ x_t^j + \sum\limits_{(i,j,t)\in A} u_{ijt} - \sum\limits_{(j,k,t)\in A} u_{jkt}, & \text{if} \quad j \in N_d. \end{cases} \tag{4.1}$$

We will denote Eq. (4.1) by $\boldsymbol{x}_{t+1} = f_t(\boldsymbol{x}_t, \boldsymbol{u}_t)$.

For a given state $\boldsymbol{x}_t$, the cost incurred by the control, $\boldsymbol{u}_t$, is:

$$g_t(\boldsymbol{x}_t, \boldsymbol{u}_t) = \sum_{j \notin N_f} (h_t^j x_t^{j+} + p_t^j x_t^{j-}) + \sum_{(i,j,t)\in A} f_{ijt} z_{ijt},$$

where $x_t^{j+} = \max(0, x_t^j)$ and $x_t^{j-} = -\min(0, x_t^j)$.

Comparing this function $g_t(\cdot,\cdot)$ to the objective function for Formulation 1, we can see that the optimization problem is to find the optimal value:

$$J^*(\boldsymbol{x}_o) = \min \sum_{t=0}^{T} g_t(\boldsymbol{x}_t, \boldsymbol{u}_t).$$

The dynamic programming algorithm (see e.g., Bertsekas [4]) gives the optimal solution, $J^*(\boldsymbol{x}_o) = J_0(\boldsymbol{x}_o)$ where J_0 is the last step in the following recursion from $T-1,\ldots,0$:

$$J_t(\boldsymbol{x}_t) = \min_{\boldsymbol{u}_t \geq 0} [g_t(\boldsymbol{x}_t, \boldsymbol{u}_t) + J_{t+1}(f_t(\boldsymbol{x}_t, \boldsymbol{u}_t))],$$

where $J_T(\boldsymbol{x}_T) = 0$.

Using the cost-to-go function values $J_t(\boldsymbol{x}_t)$, from this recursion, the optimal sequence of control decisions, $\{\boldsymbol{u}_t\}, t = 0,\ldots,T-1$, starting with $\boldsymbol{x}_o = (0,\ldots,0)$, can be obtained by performing the following iteration from $t = 0,\ldots,T-1$:

$$\boldsymbol{u}_t = \arg\min_{(\boldsymbol{u}_t \geq 0)} [g_t(\boldsymbol{x}_t, \boldsymbol{u}_t) + J_{t+1}(f_t(\boldsymbol{x}_t, \boldsymbol{u}_t))]. \tag{4.2}$$

If we simplify this expression by defining

$$Q_t(\boldsymbol{x}_t, \boldsymbol{u}_t) = g_t(\boldsymbol{x}_t, \boldsymbol{u}_t) + J_{t+1}(f_t(\boldsymbol{x}_t, \boldsymbol{u}_t)),$$

then the iteration (4.2) becomes:

$$\boldsymbol{u}_t = \arg\min_{(\boldsymbol{u}_t \geq 0)} [Q_t(\boldsymbol{x}_t, \boldsymbol{u}_t)], \tag{4.3}$$

which we will use in the next section.

4.2 Approximate Dynamic Programming

Executing the dynamic programming algorithm described in the previous section would of course be impractical given the dimensionality of the state space. In this section, we will show how we can substitute a heuristic solution in the iteration (4.3) to create an approximate dynamic programming (ADP) algorithm. The motivation for this substitution is that even though it may be impossible to determine the exact cost-to-go function values for $J_t(\boldsymbol{x}_t)$ in Eq. (4.2), we can approximate the values of $Q_t(\boldsymbol{x}_t, \boldsymbol{u}_t)$ in Eq. (4.3) with solutions which can be quickly obtained from a heuristic algorithm.

To enable this substitution in the iteration (4.3), we will define $Q_t^H(\boldsymbol{x}_t, \boldsymbol{u}_t)$ to be the value of heuristic algorithm H given system state $\boldsymbol{x}_t$ and control $\boldsymbol{u}_t$.

The ADP algorithm then becomes the following iteration from $t = 0, \ldots, T-1$:

$$\boldsymbol{u}_t = \arg\min_{(\boldsymbol{u}_t \geq 0)} [Q_t^H(\boldsymbol{x}_t, \boldsymbol{u}_t)]. \tag{4.4}$$

Although this alleviates the complication of evaluating the cost-to-go function, we are still faced with executing this approximation algorithm over all $\boldsymbol{u}_t \geq 0$. To limit the search of possible values of $\boldsymbol{u}_t$, one method we used is to vary the binary vector $\boldsymbol{z}_t$ that controls when and where shipping occurs, in a pre-specified way. If we have a physical network with m arcs, then at each stage t we would need to run the heuristic algorithm 2^m times to find the best $\boldsymbol{z}_t$. For the purpose of efficiency, the general ADP algorithm that was implemented simplified the search for $\boldsymbol{z}_t$ in the following way:

1. Start with all $z_{ijt} = 0$ for all (i, j, t).

2. For every time period $t = 0, \ldots, T-1$, and for every shipping arc (i, j, t), do the following:

   (a) Execute the heuristic algorithm H with $z_{ijt} = 1$, thus forcing a shipment to occur along this arc. Let Q_1^H be this solution value.

   (b) Execute the heuristic algorithm H with $z_{ijt} = 0$, thus restricting any shipment from occurring along this arc. Let Q_0^H be this solution value.

   (c) Set $z_{ijt} = \arg\min(Q_0^H, Q_1^H)$.

The above is an example of various ways to implement an ADP algorithm. Another way to think about this procedure is as an enhancement of the heuristic algorithm that is employed in the approximation. In the next section, we will discuss how the algorithms we introduced in Chapter 3 were enhanced by the ADP method.

4.3 ADP for the Pull algorithm family

In this section, we will show how the ADP version of the Basic Pull algorithm was implemented, how it performs, and how the ADP enhancement was implemented for the other algorithms. These are the steps involved for the Basic Pull algorithm:

1. Start with a current solution in which nothing has been pre-set.

2. For each time period and for each store, do the following:

   (a) Force a shipment to occur from the DC to the store at this time, execute the Pull algorithm using the current solution, and save this trial result.

   (b) Prevent a shipment from the DC to the store at this time, execute the Pull algorithm using the current solution, and save this trial result.

   (c) Select the option - forcing (1) or preventing (0) - that results in the minimum of these two trial solutions to be pre-set in the current solution.

The results of applying the ADP version of the Pull algorithm to the simple example we used in Chapter 3 are displayed in the three tables below. For store 1 at time 0, this algorithm started with the same result as the Pull solution, which is in Table 4.1. In Table 4.2, after forcing the shipment from the DC to Store 2 at time 0, Store 2's total cost goes from \$205 to \$210, but this enables the DC's cost to decrease from \$500 to \$400. The total cost of \$780 stays as the minimum for several more iterations until the algorithm achieves its lowest bound by preventing a shipment to Store 2 at time period 4. It turns out that the final solution, which is displayed in Table 4.3, is the same as the solution given by the Lagrangian Pull algorithm.

In this example, we executed the ADP Pull algorithm for all time periods and store shipments. We could also have controlled the shipments from the factory to the DC. Because this algorithm is monotonically decreasing, then for large problems we

Table 4.1: ADP Pull - First Iteration

	0	1	2	3	4	Total
DC						
Beginning Inventory/$	0/0	30/60	0/0	45/90	25/50	200
Ship: Factory to DC/$	60/150	0/0	70/150	0/0	0/0	300
Shipments to Stores	30	30	25	20	25	
DC Total $	150	60	150	90	50	500
Store 1						
Beginning Inventory/$	0/0	15/30	0/0	15/30	5/10	70
Ship: DC to Store/$	30/50	0/0	25/50	0/0	0/0	100
Demand	15	15	10	10	5	
Store 1 Total $	50	30	50	30	10	170
Store 2						
Beginning Inventory/$	0/0	-5/25	15/30	0/0	0/0	55
Ship: DC to Store/$	0/0	30/50	0/0	20/50	25/50	150
Demand	5	10	15	20	25	
Store 2 Total $	0	75	30	50	50	205
Grand Total $	200	165	230	170	110	875

Table 4.2: ADP Pull - Second Iteration

	0	1	2	3	4	Total
DC						
Beginning Inventory/$	0/0	0/0	0/0	25/50	25/50	100
Ship: Factory to DC/$	45/150	0/0	85/150	0/0	0/0	300
Shipments to Stores	45	0	60	0	25	
DC Total $	150	0	150	50	50	400
Store 1						
Beginning Inventory/$	0/0	15/30	0/0	15/30	5/10	70
Ship: DC to Store/$	30/50	0/0	25/50	0/0	0/0	100
Demand	15	15	10	10	5	
Store 1 Total $	50	30	50	30	10	170
Store 2						
Beginning Inventory/$	0/0	10/20	0/0	20/40	0/0	60
Ship: DC to Store/$	15/50	0/0	35/50	0/0	25/50	150
Demand	5	10	15	20	25	
Store 2 Total $	50	20	50	40	50	210
Grand Total $	250	50	250	120	110	780

Table 4.3: ADP Pull - Final Iteration

	0	1	2	3	4	Total
DC						
Beginning Inventory/$	0/0	0/0	0/0	0/0	0/0	0
Ship: Factory to DC/$	45/150	0/0	85/150	0/0	0/0	300
Shipments to Stores	45	0	85	0	0	
DC Total $	150	0	150	0	0	300
Store 1						
Beginning Inventory/$	0/0	15/30	0/0	15/30	5/10	70
Ship: DC to Store/$	30/50	0/0	25/50	0/0	0/0	100
Demand	15	15	10	10	5	
Store 1 Total $	50	30	50	30	10	170
Store 2						
Beginning Inventory/$	0/0	10/20	0/0	45/90	25/50	160
Ship: DC to Store/$	15/50	0/0	60/50	0/0	0/0	100
Demand	5	10	15	20	25	
Store 2 Total $	50	20	50	90	50	260
Grand Total $	250	50	250	120	60	730

may want to save computer time by stopping the program and using the last solution it has found.

The ADP versions of the other four algorithms - Single Shift, Multiple Shift, Pricing, and Lagrangian Pull - use the same generic structure as described above for the basic Pull algorithm. The algorithms that shift lots or multiple-lot shipments are straight-forward to implement this way because they are monotonically decreasing. However, because of the pricing schemes used by the latter two algorithms, we have to save the lowest bound solution during the execution of the ADP version.

The dramatic improvement that we saw in the simple example where a non-ADP Pull solution of $870 went down to $730 in the ADP version ($700 is the optimal), continued to occur in other tests. As we will see in the next chapter on computational results, the ADP version improved the results of all the algorithms and enhanced the Basic Pull algorithm significantly.

Chapter 5

Computational Results

In this chapter, we will describe how the formulations, algorithms, and ADP versions of the algorithms have been implemented and how well they perform. We will give an overview of the computer environment, the program architecture, and the test data, and we will tabulate the results of each algorithm. Although the focus was more on results than on speed, we found that other than the size constraints imposed by the tighter MIP formulations, problems on the scale of three DC's, 10-30 stores, and 10-30 time periods did not require an exceptional amount of computer power.

5.1 Computer Environment and Program Architecture

In this section, we describe how the formulations and algorithms were implemented on the computer.

The mixed integer programs were run on a Sun SPARCstation 20 model 60, using the C-plex optimization package, version 5.0. The algorithms were run on the same Sun workstation and also on a Dell Pentium 90 MHz computer where they were developed. As mentioned above, it was not necessary to use more power than was available on these two modest platforms. The algorithms were written in C, using C++-style comments. They were compiled by Visual C++ and the gnu C++ compiler.

There are eight C programs, all of which read the problem data from a common data file. This data file has the number of nodes, arcs, stores, time periods, the cost parameters, and the demands over time for all the stores. All eight programs use a common function to read the problem data and prepare the data structures in memory that describe the network, costs, and demand schedule. See Appendix B for a partial listing of the source code and a sample data file.

There are three programs which generate mixed integer programming models for Formulations 1, 2, and 3, respectively. These programs generate LP formatted files that are input to C-plex. These files contain algebraic descriptions of the MIPs which were useful for testing the formulations. For larger problems, it would be necessary to load a problem's data - the A matrix, the cost coefficients, the right hand side, etc. - directly into memory to be solved by C-plex.

There is a program for each of the five algorithms:

1. The Basic Pull algorithm.
2. The Single Shift algorithm.
3. The Multiple Shift algorithm.
4. The Pricing algorithm.
5. The Lagrangian Pull algorithm.

Each of these programs has an option to run the ADP version as described in the previous chapter. As an example of the general structure of these five programs, this is how the Basic Pull program works:

1. Load the problem data.
2. If the ADP option has been selected, control the program by iterating by time period and by store, setting the binary variables accordingly, and doing the following:
3. Run the single location DP algorithm for each store.

4. Accumulate the replenishment demands on the DC's.

5. Run the same DP algorithm, but without backlogging, for each DC.

6. Accumulate the total inventory, penalty, and shipping costs, and display the total cost.

7. If we are running the ADP version, the program continues to the next store or time period.

5.2 Computational Results

In this section, we will describe the test data used and the results of the computations for each of the algorithms. The optimal solutions were achieved by executing the mixed integer program using Formulation 3. As mentioned in Chapter 2, the LP relaxation of Formulation 3 resulted in an integral optimal solution for all but one of these test problems. Also, as previously mentioned, because of the large gap between the integer solution and the LP relaxation, Formulations 1 and 2 result in branch-and-bound searches that are practical only for small problems.

We used 6 test data sets for an arborescent network with two echelons, multiple (up to 3) DC's, as many as 10 stores per DC, and 10-30 time periods. The largest had 3 DC's, 30 stores, and 30 time periods.

The demands for each store were generated randomly from a uniform distribution or were given a varying demand pattern where stores could have increasing, decreasing, constant, or alternating demands. The costs applied were such that inventory costs were less than penalty costs, DC inventory costs were less than store inventory costs, and fixed shipping costs to the DC's were less than the costs to the stores. The fixed shipping costs were large enough to avoid situations where it would be economical to ship every time.

If we have an arborescent network, which means every node has at most one predecessor node, then because we are assuming unlimited capacity at the factory, we can sub-divide a problem with several DC's into separate single DC networks (consisting of the factory, the DC, and the stores that receive shipments from that DC). We did not need to do this for the algorithms but did so for the MIP Formulation 3 because of the size of the model. Using the Sun SPARCstation, the largest MIP that could be loaded into memory and solved was for one DC, 10 stores, and 30 time periods, although the solution time was usually less than 3 minutes. A larger workstation could load and solve a larger problem, but eventually we would run into size limitations.

At the core of the algorithms is the single location lot-size algorithm. Since the execution time for this DP algorithm is $O(T^2)$, an initial concern was to find out how big a problem the simplest program (Basic Pull) could handle. A test of a one DC, 5 store problem with 300 time periods ran in about one minute. Since we can specify a very detailed demand pattern with this many time periods, we could easily apply the Basic Pull algorithm to large problems.

As mentioned above, instead of analyzing the computation time, the main interest in the computational experiments was to compare the results of the algorithms. Table 5.1 contains the results of the algorithms and the optimal solutions using Formulation 3. Most of these test runs for the algorithms ran in less than one or two minutes. The only exceptions were the ADP versions of the Single Shift, Multiple Shift, and Lagrangian Pull algorithms, which took two hours or longer to run to completion for all but the first data set. For these three programs, execution was halted after six hours for test problems 4, 5, and 6.

For test problem 6, the solution of the LP relaxation for Formulation 3 was 37,659, which is very close to the integral optimal solution of 37,669. For the other five test problems, the LP relaxation for Formulation 3 was integral. As expected, the local search algorithms - Single Shift and Multiple Shift - make some improvements to the Basic Pull solution. The other two algorithms, which use a pricing mechanism, achieve even better results. These latter two, the Pricing algorithm and the Lagrangian Pull algorithm, are able to explore a wider region of solutions, although the Lagrangian Pull algorithm was unable to improve upon the Basic Pull solution for the last two test problems. With a couple of exceptions for the pricing-oriented heuristics, the ADP versions improve upon the results for each algorithm. Interestingly, the most dramatic improvement is with the Basic Pull algorithm. This makes the ADP version of the the Basic Pull algorithm an attractive choice for implementation because it achieves good results, it is simple, and it runs much faster than the ADP versions of the other algorithms.

To summarize, our results illustrate:

1. Formulation 3 is very strong. It almost always produces integral solutions and can be used to solve problems of a moderate size (3 DC's, 30 stores, 30 time periods).

2. All the heuristic algorithms produce near optimal solutions for large scale problems.

3. The Lagrangian Pull heuristic performed the best in most cases, followed by the Pricing algorithm, showing that pricing mechanisms yield better solutions than local search techniques.

4. The ADP enhancement enables a simple algorithm like the Basic Pull to achieve solutions that are comparable to the best heuristic.

Table 5.1: Computational Results

Test Problem	1	2	3	4	5	6
DC's	1	1	3	3	3	3
Stores	10	10	30	30	30	30
Time Periods	10	20	20	30	30	30
Formulation 1						
- Variables	462	902	2706	4026	4026	4026
- Constraints	264	484	1452	2112	2112	2112
Formulation 2						
- Variables	1210	4620	13,860	30,690	30,690	30,690
- Constraints	1210	4620	13,860	30,690	30,690	30,690
Formulation 3						
- Variables	3110	12,220	36,660	81,990	81,990	81,990
- Constraints	3200	12,400	37,200	82,800	82,800	82,800
Optimal	4,550	8,960	26,880	40,098	37,616	37,669
Basic Pull	4,885	9,744	29,232	43,413	39,216	39,344
- with ADP	4,646	9,530	28,815	42,870	38,529	38,753
Single Shift	4,814	9,673	29,019	43,398	39,131	39,230
- with ADP	4,596	9,306	28,747	43,308	39,100	39,167
Multiple Shift	4,831	9,674	29,022	43,245	39,040	39,101
- with ADP	4,596	9,386	28,803	43,065	38,982	39,010
Pricing	4,716	9,564	29,052	42,128	37,951	37,929
- with ADP	4,716	9,461	28,407	42,045	37,951	37,929
Lagrangian Pull	4,615	9,339	28,281	43,242	39,216	39,344
- with ADP	4,615	9,250	28,281	43,161	38,997	39,255

Chapter 6

Concluding Remarks

In this chapter, we summarize the findings of this research effort, discuss implementation issues, and point out areas for further study. To develop practical solution methods for a challenging supply chain optimization problem, we have used mathematical programming, dynamic programming, and an approximate dynamic programming (ADP) technique for enhancing algorithms. The main results that we have achieved are:

1. A strong mixed integer programming formulation.
2. Good, practical algorithms, especially those that use a pricing mechanism.
3. The ADP enhancement can greatly improve a simple algorithm.
4. The formulations and algorithms can be extended to problems with multiple products.

Formulating this problem as a mixed integer program requires very many variables and constraints in order to solve a problem of any significant size. Applying Formulation 3 for medium-to-large scale problems would necessitate limiting the time frame for backorders and advance orders. As computer power increases, this type of formulation will become more viable. Formulation 4 could be explored further as a solution method.

The dynamic programming-based algorithms run efficiently and in tests yielded near-optimal solutions. The local search algorithms, which shift individual lots or multiple lot shipments, provide better solutions than the Basic Pull algorithm, but the best performing algorithms were those that used a pricing scheme. Further enhancements are certainly possible in this area.

The ADP enhancement, which can be applied to any heuristic algorithm, had the greatest impact on the Basic Pull algorithm. This method enables us to convert a simple algorithm into one that performs nearly as well as a much more complex algorithm.

In keeping with the original purpose regarding implementation, the algorithms are reasonably efficient and would not be difficult to integrate with existing data bases and programs. Applying these algorithms would involve some modifications to account for lead times and non-zero initial and final inventories.

There are several issues to be addressed if we extend the formulations and algorithms to handle multiple products. Formulation 5 for multiple products was discussed at the end of Chapter 2. A small test of this formulation indicates that it performs as well as Formulation 3, which it is based on. To develop multiple product algorithms would require modifying the single location, dynamic lot-size algorithm to accommodate multiple products. This could be done by extending the state space definition of section 3.1, but we would then face the problem of increased dimensionality.

We have assumed throughout that demand is deterministic, thereby focusing on the complexities of a general distribution network with fixed shipping costs. Future research to solve this problem for the case of stochastic demand would obviously be of great value.

Appendix A

Everett's Method

The Lagrangian relaxation used in the Lagrangian-based algorithm described in Chapter 3 employs a method by Everett [11] for updating the Lagrangian multipliers. To implement the Everett method, a slight modification was made to the method which is described in detail by Stock-Patterson [33], and is given in an abbreviated form here. The modification was to apply the method to equality constraints instead of inequality constraints.

Let λ_j^k be the value of the Lagrangian multiplier for the jth constraint: $a_j' x^k = b_j$, where x^k is the solution at the kth iteration.

$$\text{If } a_j' x^k > b_j \text{ then } \lambda_j^{k+1} = \lambda_j^k + \delta_j^k,$$
$$\text{If } a_j' x^k < b_j \text{ then } \lambda_j^{k+1} = \lambda_j^k - \delta_j^k,$$

where the parameters δ_j^k are updated by:

$$\begin{aligned}
\text{If } (a_j' x^k - b_j)(a_j' x^{k-1} - b_j) > 0, \text{ then } \delta_j^{k+1} &= \epsilon_1 \delta_j^k, \\
\text{If } (a_j' x^k - b_j)(a_j' x^{k-1} - b_j) < 0, \text{ then } \delta_j^{k+1} &= \epsilon_2 \delta_j^k, \\
\text{If } (a_j' x^k - b_j)(a_j' x^{k-1} - b_j) = 0, \text{ then } \delta_j^{k+1} &= \delta_j^k.
\end{aligned}$$

In general, $\epsilon_1 > 1$ and $\epsilon_2 < 1$; the actual values that were used were 1.5 and 0.3, respectively.

Appendix B

Program Source Code

This appendix contains the C program source code (with C++ style comments) to perform the Basic Pull algorithm with the ADP enhancement option and to generate the Cplex .LP file for Formulation 3. It also contains a description of the input data file that the programs use.

The eight programs in their entirety are too lengthy for their codes to be printed here. These programs are:

1. Formulation 1 generator.
2. Formulation 2 generator.
3. Formulation 3 generator.
4. Basic Pull algorithm.
5. Single Shift algorithm.
6. Multiple Shift algorithm.
7. Pricing algorithm.
8. Lagrangian Pull algorithm.

B.1 Basic Pull Algorithm

This section contains the program source code for the Basic Pull algorithm. The programs for the other four algorithms use the same functions that are in this program but have the added features that are described in Chapter 3. The ADP function is the same for all five algorithms.

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

// pull.cpp     Basic Pull Algorithm

// To run: pull filename(.dat assumed) switch(-a or -h)

#include <stdio.h>
#include <stdlib.h>
#include <string.h>

void adp(void);
void driver(void);
void loadup(FILE *fp);
void dp(bool backlog,double fixcost,long node);
void pullstore(void);
void pulldc(void);
void cleardp(void);
void showpull(void);

#define MAXNODES 50
#define MAXARCS 100
#define MAXTIME 110
#define MAXLINE 80

long NODES, ARCS, STORES, TIMES;
```

```
struct Node { long   NId;
              double NHolding;
              double NPenalty;
              long   NBegInvy;
              long   NFinInvy;
              long   NType;
              long   NSource;  // source node id or zero if factory
              double NPullMinCost;
              long   NPullShipQty[MAXTIME];
              long   NPullDemand[MAXTIME];
              long   NPullShipments;
              int    ZSet[MAXTIME];
              int    ZPolicy[MAXTIME];
              long   NTotDemand;
              long   NDemand[MAXTIME]; };

struct Arc  { long   AId;
              long   NFrom;
              long   NTo;
              long   ADelay;
              double AFixCost;
              double AVarCost; };

struct StateNode {long SInventory;
                  double SInvyCost;
                  double SCost2Go;
                  long SState2Go;};

struct Node NodeTable[MAXNODES];
struct Arc  ArcTable[MAXARCS];
struct StateNode SS[MAXTIME][MAXTIME];
```

```
bool backlog = true;

double totpullmin;       /* global */

// ********************   MAIN PROGRAM   *******************

int main(int argc,char *argv[])
{
FILE * fp;
char * pchar;
char probname[MAXLINE];
char filename[MAXLINE];
int namelen = 0;

if (argc<3)
  {
   printf("Usage: PULL filename switch(-a,-h)\n");
   exit(1);
  }

pchar = argv[1];

for (int ctr=0;*pchar != '\0';ctr++,pchar++,namelen++)
    probname[ctr] = *pchar;

probname[namelen] = '\0';

printf("Problem: %s\n",probname);

strcpy(filename,probname);
strcat(filename,".dat");
```

```
if ((fp=fopen(filename,"r"))==NULL)
  {
   printf("Can't open problem data file\n");
   exit(2);
  }
loadup(fp);

if (strcmp(argv[2],"-a")==0)
adp();
else
driver();

fclose(fp);

return 0;
}

//  ********************    ADP FUNCTION    ***************

void adp(void)
{
long node,t;
double solution_0,solution_1;

for (t=0;t<TIMES;t++)
{
            for (node=0;node<NODES;node++)
            {
                if (NodeTable[node].NType != 3) continue;  // stores

                NodeTable[node].ZSet[t] = 1;
                NodeTable[node].ZPolicy[t] = 0;    // Try 0 path
                driver();
                solution_0 = totpullmin;
                NodeTable[node].ZPolicy[t] = 1;    // Try 1 path
                driver();
                solution_1 = totpullmin;
                NodeTable[node].ZPolicy[t] =
                   (solution_0 < solution_1) ? 0 : 1;
                printf("node %d time %d Sol_0 %f Sol_1 %f\n",
                        node,t,solution_0,solution_1);
            }
}
}

// ********************   DRIVER FUNCTION   **************

void driver(void)
{
cleardp();
pullstore();
pulldc();
// showpull();
}

// *****************   CLEARDP  FUNCTION   ****************

void cleardp(void)
{
long k,t;

for (k=0;k<NODES;k++)
  {
        NodeTable[k].NPullMinCost = 0.0;
        NodeTable[k].NPullShipments = 0;

        for (t=0;t<TIMES;t++)
          {
            NodeTable[k].NPullShipQty[t] = 0;
            NodeTable[k].NPullDemand[t] = 0;
          }
      }
}

//  ***************   PULLSTORE FUNCTION   ****************

void pullstore(void)
{
double fixcost,varcost,invycost;
long stage,state,rnode,invyin,invyout,shipqty,source;
long nextstate,shipments,beginvy,inventory;

//  For each retail node, execute DP to get shipment schedule

for (rnode=0;rnode<NODES;rnode++)
{
if (NodeTable[rnode].NType != 3) continue;   // retail=3

int rnodeid = NodeTable[rnode].NId;
source = NodeTable[rnode].NSource;

for (int jarc=0;jarc<ARCS;jarc++)
{
if (ArcTable[jarc].NTo==rnodeid)
{
```

```
fixcost = ArcTable[jarc].AFixCost;
varcost = ArcTable[jarc].AVarCost;
}
}

//  set up inventory costs at each stage and state

invyout = 0;

for (stage=1;stage<TIMES;stage++)
{
invyout += NodeTable[rnode].NDemand[stage-1];
invyin = 0;

for (state=0;state<=TIMES;state++)
{
                    if (state > 0)
                       invyin += NodeTable[rnode].NDemand[state-1];
                    beginvy = NodeTable[rnode].NBegInvy;
                    inventory = beginvy + invyin - invyout;
                    invycost = (inventory >= 0)?
                            (inventory * NodeTable[rnode].NHolding) :
                            (-inventory * NodeTable[rnode].NPenalty);
                    SS[stage][state].SInventory = inventory;
                    SS[stage][state].SInvyCost = invycost;
}
}
dp(backlog,fixcost,rnode);       // execute DP with backlog

//  At the conclusion, cost to go at (0,0) is optimal

NodeTable[rnode].NPullMinCost = SS[0][0].SCost2Go;
```

```
// Save the resulting shipping schedule for this store.

/* S-Sequence code - used in Single and Multiple shift algs. */

/* S(0)=0,...,S(TIMES)=TIMES   */

long node2go = 0;

for (stage=0;stage<=TIMES-1;stage++)
{
NodeTable[rnode].NSequence[stage] =
node2go;
node2go =
SS[stage][node2go].SState2Go;
}
NodeTable[rnode].NSequence[TIMES] = TIMES;

/* END of S-Sequence code */

shipments = 0;
stage = 0;
state = 0;

while(state<=TIMES-1)
{
nextstate = SS[stage][state].SState2Go;
shipqty = 0;
if (nextstate != state)
{
                   for (int s=state;s<nextstate;s++)
                   {
                    shipqty += NodeTable[rnode].NDemand[s];
/* In shift algs.*/     NodeTable[rnode].NSchedRec[s] = stage;
                }
                shipments++;
                NodeTable[rnode].NPullShipQty[stage]=shipqty;
// accumulate shipqty as pull demand at the source DC
                if (source)
                    NodeTable[source-1].NPullDemand[stage]
                              += shipqty;
                state = nextstate;
}
stage++;
}
NodeTable[rnode].NPullShipments = shipments;
}
return;
}

//  *****************   PULLDC FUNCTION   *****************

void pulldc(void)
{
double fixcost,varcost,invycost;
long stage,state,dcnode,invyin,invyout,shipqty;
long nextstate,shipments,beginvy,inventory;

totpullmin=0.0;    // this is global

//  For each DC node, execute DP to get shipment schedule
//  but do not allow backlogging.

for (dcnode=0;dcnode<NODES;dcnode++)
```

```
{
    if (NodeTable[dcnode].NType != 2) continue;   // DC=2

    int dcnodeid = NodeTable[dcnode].NId;

    for (int jarc=0;jarc<ARCS;jarc++)
    {
        if (ArcTable[jarc].NTo==dcnodeid)
        {
           fixcost = ArcTable[jarc].AFixCost;
           varcost = ArcTable[jarc].AVarCost;
        }
    }

    for (int t=0; t<TIMES; t++)
        NodeTable[dcnode].NPullShipQty[t] = 0;

//  set up inventory costs at each stage and state

invyout = 0;

for (stage=1;stage<TIMES;stage++)
{
            invyout += NodeTable[dcnode].NPullDemand[stage-1];
            invyin = 0;

            for (state=0;state<=TIMES;state++)
            {
                if (state > 0)
                   invyin += NodeTable[dcnode].NPullDemand[state-1];
                beginvy = NodeTable[dcnode].NBegInvy;
                inventory = beginvy + invyin - invyout;
                invycost = (inventory >= 0)?
                    (inventory * NodeTable[dcnode].NHolding) :
                    (-inventory * NodeTable[dcnode].NPenalty) ;
                SS[stage][state].SInventory = inventory;
                SS[stage][state].SInvyCost = invycost;
            }
}

        dp(!backlog,fixcost,dcnode);  // execute DP without backlog

//  At the conclusion, cost to go at (0,0) is optimal

NodeTable[dcnode].NPullMinCost = SS[0][0].SCost2Go;

//  Save the resulting shipping schedule for this DC

shipments = 0;
stage = 0;
state = 0;

while(state<=TIMES-1)
{
             nextstate = SS[stage][state].SState2Go;
             shipqty = 0;
             if (nextstate != state)
             {
                for (int s=state;s<nextstate;s++)
                    shipqty += NodeTable[dcnode].NPullDemand[s];

                shipments++;
                NodeTable[dcnode].NPullShipQty[stage]=shipqty;
                state = nextstate;
            }
            stage++;
}
NodeTable[dcnode].NPullShipments = shipments;
}
for (int k=0;k<NODES;k++)
totpullmin += NodeTable[k].NPullMinCost;

        printf("Pull - Total Min Cost = %f\n",totpullmin);

return;
}

//   ********************   DP FUNCTION   ******************

void dp(bool backlog,double fixcost,long node)
{
double mincost,arccost,cost;
long timelimit,stage,state,nextstate,node2go,startstate,begstate;

  SS[0][0].SCost2Go = 0;
  SS[0][0].SState2Go = 0;
  SS[TIMES][TIMES].SInventory = 0;
  SS[TIMES][TIMES].SInvyCost = 0;
  SS[TIMES][TIMES].SCost2Go = 0;
  SS[TIMES][TIMES].SState2Go = 0;

//  Backward DP algorithm, where node = (stage,state)
//  The initial node is (0,0)  Final node=(TIMES,TIMES)

  for (stage=TIMES-1;stage>=0;stage--)
  {
```

```
if (stage==0)
   timelimit = 0;
else
   timelimit = TIMES;

if (backlog)
   begstate = 0;
else
   begstate = stage;

for (state=begstate;state<=timelimit;state++)
{
    mincost = 99999999;
    if (stage==TIMES-1)
       startstate = TIMES;
    else
    if (!backlog && state==begstate)
       startstate = state + 1;
    else
       startstate = state;
    for (nextstate=startstate;nextstate<=TIMES;nextstate++)
    {
        if (nextstate==state)
           arccost = SS[stage+1][nextstate].SInvyCost;
        else
           arccost = fixcost + SS[stage+1][nextstate].SInvyCost;
        cost = arccost + SS[stage+1][nextstate].SCost2Go;

        if (NodeTable[node].ZSet[stage])
        {
           if ((NodeTable[node].ZPolicy[stage]==1) &&
               (nextstate==state))    // must ship
```

```
                cost = 999999999;
                if ((NodeTable[node].ZPolicy[stage]==0) &&
                    (nextstate!=state))    // can not ship
                cost = 999999999;
             }

             if (cost < mincost)
             {
                mincost = cost;
                node2go = nextstate;
             }
          }
          SS[stage][state].SCost2Go = mincost;
          SS[stage][state].SState2Go = node2go;
      }
  }
//  At the conclusion, cost to go at (0,0) is optimal

return;
}

// *******************   SHOWPULL  FUNCTION  **************

void showpull(void)
{
long k,t;

for (k=0;k<NODES;k++)
{
    printf("NodeId %d Pull Min Cost %f Shipments %d\n",
            NodeTable[k].NId,
            NodeTable[k].NPullMinCost,
            NodeTable[k].NPullShipments);


    for (t=0;t<TIMES;t++)
        printf("t %d ShipQty %d\n",t,NodeTable[k].NPullShipQty[t]);
}
}


//  *******************   LOADUP FUNCTION   ***************


void loadup(FILE *fp)
{
long j,k,nodeid,beginvy,fininvy,fromnode,tonode;
long delay,t,nfrom,nto,source,holding,penalty,fixcost;
long begndx,endndx,tdemand,varcost;
long d[10];
char line[MAXLINE];


//  Read numbers of nodes, arcs, stores, and times


fscanf(fp,"%80s",line);


if (fscanf(fp,"%d %d %d %d",&NODES,&ARCS,&STORES,&TIMES)!=4)
  {
   printf("Expecting numbers of nodes,arcs,stores,
           times on first record\n");
   exit (3);
  }


//  Read in node info
//  assume nodes are labeled 1,2,...,n
//  and will be stored in    0,1,...,n-1
```

```
fscanf(fp,"%80s",line);

for (k=0; k<NODES; k++)
  {
    if (fscanf(fp,"%d %d %d %d %d",&nodeid,&holding,&penalty,
                  &beginvy,&fininvy) != 5)
        {   printf("Expecting node,holding,penalty,
                   beginvy,fininvy\n");
            exit (4);
        }

    if ((nodeid < 1) || (nodeid > NODES))
      {    printf("Node is out of bounds\n");
           exit (5);
      }
    NodeTable[k].NId = nodeid;
    NodeTable[k].NHolding = holding/100.0;
    NodeTable[k].NPenalty = penalty/100.0;
    NodeTable[k].NBegInvy = beginvy;
    NodeTable[k].NFinInvy = fininvy;
    NodeTable[k].NPullMinCost = 0;
    NodeTable[k].NPullShipments = 0;
    NodeTable[k].NSource = 0;

    for (j=0;j<TIMES;j++)
{
NodeTable[k].NDemand[j] = 0;
NodeTable[k].NPullShipQty[j] = 0;
NodeTable[k].ZSet[j] = 0;
NodeTable[k].ZPolicy[j] = 0;
}
  }
```

```
// Read in arc info

  fscanf(fp,"%80s",line);

  for (k=0;k<ARCS;k++)
    {
      if (fscanf(fp,"%d %d %d %d %d",&fromnode,&tonode,
                 &delay,&fixcost,&varcost)!=5)
        {
         printf("Expecting from,to,delay,fixed,variable\n");
         exit(6);
        }
         ArcTable[k].AId = k+1;
         ArcTable[k].NFrom = fromnode;
         ArcTable[k].NTo = tonode;
         ArcTable[k].ADelay = delay;
         ArcTable[k].AFixCost = fixcost/100.0;
         ArcTable[k].AVarCost = varcost/100.0;
         ArcTable[k].pNFrom = NULL;
         ArcTable[k].pNTo = NULL;
         NodeTable[tonode-1].NSource = fromnode;
    }

// Read in store demand in groups of 10

  fscanf(fp,"%80s",line);

  for (k=0;k<STORES;k++)
    { for (t=0;t<TIMES;t+=10)
        {
          if (fscanf(fp,"%d %d %d %d %d %d %d %d %d %d %d %d %d",
              &nodeid,&begndx,&endndx,&d[0],&d[1],&d[2],&d[3],
              &d[4],&d[5],&d[6],&d[7],&d[8],&d[9]) != 13)
            {printf("Expecting node,time,demand\n");
             exit(7);
            }
          for (j=begndx;j<=endndx;j++)
            {
              NodeTable[nodeid-1].NDemand[j] = d[j-t];
 // accumulate up the tree
              source = NodeTable[nodeid-1].NSource;
              while (source)
              {
                    NodeTable[source-1].NDemand[j] += d[j-t];
                    source = NodeTable[source-1].NSource;
              }
            }
        }
    }

// calculate total demand

   for (k=0;k<NODES;k++)
   {
   tdemand = 0;
   for (t=0;t<TIMES;t++)
   tdemand += NodeTable[k].NDemand[t];
   NodeTable[k].NTotDemand = tdemand;
   }

/*  Figure out the types of each node:

1 = factory
```

```
2 = DC or warehouse
3 = retail store
*/
  for (k=0;k<NODES;k++)
    {
      nfrom =0;
      nto = 0;
      for (j=0;j<ARCS;j++)
        {
          if (ArcTable[j].NFrom == NodeTable[k].NId) nfrom++;
          if (ArcTable[j].NTo == NodeTable[k].NId) nto++;
        }
      if ((nfrom == 0) && (nto == 0))
        { printf("Nfrom and Nto equal 0\n");
          exit(9);
        }
      if ((nfrom ==0) && (nto > 0)) NodeTable[k].NType = 3;
      if ((nfrom > 0) && (nto > 0)) NodeTable[k].NType = 2;
      if ((nfrom > 0) && (nto ==0)) NodeTable[k].NType = 1;
    }
  return;
}
```

B.2 Formulation 3 Generator

This section contains the program source code that is unique to the Formulation 3 Generator. The Loadup function is the same one used in the Basic Pull program listed above.

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

```
//  form3.cpp      Formulation 3 Generator

//  To run: form3 filename(.dat assumed)
//  Creates a .LP file for Cplex.

#include <stdio.h>
#include <stdlib.h>
#include <string.h>

void loadup(FILE *fp);
void writelp(FILE *fplp);

#define MAXNODES 50
#define MAXARCS 100
#define MAXTIME 60
#define MAXLINE 80

long NODES, ARCS, STORES, TIMES;

struct Node { long   NId;
              double NHolding;
              double NPenalty;
              long   NBegInvy;
              long   NFinInvy;
              long   NType;
              long   NSource;
              double NPullMinCost;
              long   NPullShipQty[MAXTIME];
              long   NPullDemand[MAXTIME];
              long   NPullShipments;
              long   NTotDemand;
              long   NDemand[MAXTIME]; };

struct Arc   { long   AId;
               long   NFrom;
               long   NTo;
               long   ADelay;
               double AFixCost;
               double AVarCost; };

struct Node NodeTable[MAXNODES];
struct Arc  ArcTable[MAXARCS];

// ******************   MAIN PROGRAM   ********************

int main(int argc,char *argv[])
{
FILE * fp,fplp,fpord;
char * pchar;
char probname[MAXLINE];
char filename[MAXLINE];
int namelen = 0;

if (argc<2)
  {
   printf("Usage: %s FILE\n","form3");
   exit(1);
  }
pchar = argv[1];

for (int ctr=0;*pchar != '\0';ctr++,pchar++,namelen++)
    probname[ctr] = *pchar;
```

```
probname[namelen] = '\0';
printf("Problem: %s\n",probname);
strcpy(filename,probname);
strcat(filename,".dat");

if ((fp=fopen(filename,"r"))==NULL)
  {
   printf("Can't open problem data file\n");
   exit(2);
  }
strcpy(filename,probname);
strcat(filename,".lp");

if ((fplp=fopen(filename,"w"))==NULL)
{
printf("Can't open LP file\n");
exit(2);
}
loadup(fp);
writelp(fplp);
fclose(fp);
fclose(fplp);
return 0;
}

// ******************   WRITELP FUNCTION   ***************

void writelp(FILE *fplp)
{
long t,i,j,k,nodeid,fromnode,tonode,demand;
double holding,penalty,holdcost,pencost,afixcost,avarcost;
```

```
//  Objective function

  fprintf(fplp,"Minimize\n");

  for (k=0;k<ARCS;k++)
  {
          fromnode = ArcTable[k].NFrom;
          tonode = ArcTable[k].NTo;
          afixcost = ArcTable[k].AFixCost;
          avarcost = ArcTable[k].AVarCost;

          for (t=0;t<TIMES;t++)
          {
                fprintf(fplp,"+%0.2fz%02d_%02d_%02d",
                        afixcost,fromnode,tonode,t);
                if (((t%5)==4) || (t==TIMES-1)) fprintf(fplp,"\n");
          }
  }

/*  now for the q variables     */

  for (k=0;k<NODES;k++)
  {
      if (NodeTable[k].NType != 3) continue;

      nodeid = NodeTable[k].NId;
      holdcost = NodeTable[k].NHolding;
      pencost  = NodeTable[k].NPenalty;

      for (t=0;t<TIMES;t++)
      {
          if (t > 0)
```

```
         {
            for (i=0;i<=t;i++)
            {
                holding = (t-i)*holdcost;
                fprintf(fplp,"+%0.2fq%02d_%02d_%02d",
                              holding,nodeid,i,t);
                if (((i%5)==4) || (i==TIMES-1)) fprintf(fplp,"\n");
            }
         }
         if (t < TIMES-1)
         {
            for (i=t+1;i<TIMES;i++)
            {
                penalty = (i-t)*pencost;
                fprintf(fplp,"+%0.2fq%02d_%02d_%02d",
                              penalty,nodeid,i,t);
                if (((i%5)==4) || (i==TIMES-1)) fprintf(fplp,"\n");
            }
         }
      }
      for (t=0;t<TIMES;t++)
      {
         for (i=0;i<TIMES;i++)
         {
             fprintf(fplp,"+%0.2fx%02d_%02d_%02d",
                           holdcost,nodeid,i,t);
             if (((i%5)==4)||(i==TIMES-1)) fprintf(fplp,"\n");
         }
      }
  }

// Constraints -- meet store demand - 1st echelon.
```

```
  fprintf(fplp,"Subject to\n");

  for (k=0;k<NODES;k++)
  {
          if (NodeTable[k].NType != 3) continue;

          nodeid = NodeTable[k].NId;

          for (t=0;t<TIMES;t++)
          {
              demand = NodeTable[k].NDemand[t];

              fprintf(fplp,"d%02d_%02d: ",nodeid,t);

              for (i=0;i<TIMES;i++)
              {
                  fprintf(fplp,"+q%02d_%02d_%02d",nodeid,i,t);
                  if (((i%5)==4) || (i==TIMES-1)) fprintf(fplp,"\n");
              }
              fprintf(fplp,"=%d\n",demand);
          }
  }

//  Constraints -- meet store demand - 2nd echelon

  for (k=0;k<NODES;k++)
  {
          if (NodeTable[k].NType != 3) continue;

          nodeid = NodeTable[k].NId;
```

```
            for (t=0;t<TIMES;t++)
            {
                demand = NodeTable[k].NDemand[t];

                fprintf(fplp,"d2%02d_%02d: ",nodeid,t);

                for (i=0;i<TIMES;i++)
                {
                    fprintf(fplp,"+y%02d_%02d_%02d",nodeid,i,t);
                    if (((i%5)==4) || (i==TIMES-1)) fprintf(fplp,"\n");
                }
                    fprintf(fplp,"=%d\n",demand);
            }
    }

//  Constraints - fixed charge forcing - stores

    for (j=0;j<ARCS;j++)
    {
        fromnode = ArcTable[j].NFrom;
        tonode = ArcTable[j].NTo;
        if (NodeTable[tonode-1].NType!=3) continue;

        for (t=0;t<TIMES;t++)
        {
            demand = NodeTable[tonode-1].NDemand[t];

            for (i=0;i<TIMES;i++)
            {
                fprintf(fplp,"f%02d%02d%02d: %dz%02d_%02d_%02d-
                              q%02d_%02d_%02d>=0\n",tonode,i,t,
                              demand,fromnode,tonode,i,tonode,i,t);
                if (((i%5)==4) || (i==TIMES-1)) fprintf(fplp,"\n");
            }
        }
    }

//  Constraints - fixed charge forcing - 2nd echelon

    for (j=0;j<ARCS;j++)
    {
        fromnode = ArcTable[j].NFrom;
        tonode = ArcTable[j].NTo;
        if (NodeTable[tonode-1].NType!=3) continue;

        for (t=0;t<TIMES;t++)
        {
            demand = NodeTable[tonode-1].NDemand[t];

            for (i=0;i<TIMES;i++)
            {
                fprintf(fplp,"f2%02d%02d%02d: %dz%02d_%02d_%02d-
                             y%02d_%02d_%02d>=0\n",tonode,i,t,
                             demand,1,fromnode,i,tonode,i,t);
                if (((i%5)==4) || (i==TIMES-1)) fprintf(fplp,"\n");
            }
        }
    }

//  Balance of flow at DC inventory nodes

     for (k=0; k<NODES; k++)
     {
         if (NodeTable[k].NType != 3) continue;
        nodeid = NodeTable[k].NId;

        for (t=0;t<TIMES;t++)
        {
            for (i=0;i<TIMES;i++)
            {
                fprintf(fplp,"db%02d%02d%02d: ",k,i,t);

                if (i==0)
                {
                   fprintf(fplp,"+x%02d_%02d_%02d",nodeid,i+1,t);
                   fprintf(fplp,"-y%02d_%02d_%02d+q%02d_%02d_%02d",
                                 nodeid,i,t,nodeid,i,t);
                }
                if ((i>0)&&(i<TIMES-1))
                {
                   fprintf(fplp,"+x%02d_%02d_%02d-x%02d_%02d_%02d",
                                 nodeid,i+1,t,nodeid,i,t);
                   fprintf(fplp,"-y%02d_%02d_%02d+q%02d_%02d_%02d",
                                 nodeid,i,t,nodeid,i,t);
                }
                if (i==TIMES-1)
                {
                   fprintf(fplp,"-x%02d_%02d_%02d",nodeid,i,t);
                   fprintf(fplp,"-y%02d_%02d_%02d+q%02d_%02d_%02d",
                                 nodeid,i,t,nodeid,i,t);
                }
                   fprintf(fplp," = 0\n");
            }
        }
    }
```

```
// Binaries - formerly Bounds and Integer variables

  fprintf(fplp,"Binaries\n");

  for (j=0;j<ARCS;j++)
  {
      fromnode = ArcTable[j].NFrom;
      tonode = ArcTable[j].NTo;

      for (t=0;t<TIMES;t++)
      {
          fprintf(fplp,"z%02d_%02d_%02d ",fromnode,tonode,t);
          if (((t%5)==4) || (t==TIMES-1)) fprintf(fplp,"\n");
      }
  }
  fprintf(fplp,"End\n");
}
```

B.3 Input Data File

The input data file follows this format:

1. A line of text is required as a header record that separates each group of data records.

2. The data records consist of numbers separated by blanks.

3. Nodes are number sequentially starting at 1.

4. The first data record has the number of nodes, arcs, stores, and time periods.

5. The second group of data records has for each node, the holding cost, the penalty cost, and the initial and final inventory (which are usually set to zero). The

cost fields assume a two-digit decimal.

6. The third group of data records has for each arc, the from-node, the to-node, the lead time (which we set to zero), the fixed shipping cost, and the variable shipping cost (ignored).

7. The fourth group of data records has the demand data in sets of ten time periods: the store node number, the starting time period, and the ending time period, followed by ten demand numbers.

The following is the input data file for test problem 1:

```
Nodes-Arcs-Stores-Periods
12 11 10 10
Node-Holding-Cost-Penalty-cost-Initial-invy-Final-invy
1 000 000 0 0
2 200 1000 0 0
3 300 500 0 0
4 300 500 0 0
5 300 500 0 0
6 300 500 0 0
7 300 500 0 0
8 300 500 0 0
9 300 500 0 0
10 300 500 0 0
11 300 500 0 0
12 300 500 0 0
Node-from-Node-To-Time-delay-Fixed-cost-Variable-cost
1 2 0 10000 000
2 3 0 5000 000
2 4 0 5000 000
2 5 0 5000 000
2 6 0 5000 000
```

```
2 7 0 5000 000

2 8 0 5000 000

2 9 0 5000 000

2 10 0 5000 000

2 11 0 5000 000

2 12 0 5000 000

Store-node-start-time-end-time-10-demands-zero-fill

3   0 10 10 10 10 10 10 10 10 10 10 10

4   0 10  6  7  8  9 10 11 12 13 14 15

5   0 10 15 14 13 12 11 10  9  8  7  6

6   0 10  5 15  5 15  5 15  5 15  5 15

7   0 10  8 10 12  8 10 12  8 12 12 10

8   0 10  5 10 15  5 10 15  5 10 15 10

9   0 10  8  9 10 11 12  8  9 10 11 12

10  0 10 12 11 10  9  8 12 11 10  9  8

11  0 10  8  9 10 11 12 12 11 10  9  8

12  0 10  5 10 15 10  5 10 15 10  5 10
```

Bibliography

[1] R.K. Ahuja, T.L. Magnanti, and J.B. Orlin. *Network Flows: Theory, Algorithms, and Applications.* Prentice Hall, Englewood Cliffs, NJ, 1993.

[2] B.C. Arntzen, G.G. Brown, T.P. Harrison, and L.L. Trafton. Global supply chain management at Digital Equipment Corportation. *Interfaces*, 25(1):69–93, 1995.

[3] A. Balakrishnan and S.C. Graves. A composite algorithm for a concave-cost network flow algorithm. *Networks*, 19:175–202, 1989.

[4] D.P. Bertsekas. *Dynamic Programming and Optimal Control.* Athena Scientific, Belmont, MA, 1995.

[5] D.P. Bertsekas and J.N. Tsitsiklis. *Neuro-Dynamic Programming.* Athena Scientific, Belmont, MA, 1996.

[6] L.M.A. Chan, A. Muriel, and D. Simchi-Levi. Supply-chain management: integrating inventory and transportation. August 1997.

[7] A.J. Clark and H. Scarf. Optimal policies for a multi-echelon inventory problem. *Management Science*, 6:475–490, 1960.

[8] M.A. Cohen and H.L. Lee. Strategic analysis of integrated production-distribution systems: Models and methods. *Operations Research*, 36(2):216–228, 1988.

[9] T. Davis. Effective supply chain management. *Sloan Management Review*, pages 35–46, Summer 1993.

[10] R.E. Erickson, C.L. Monma, and A.F. Veinott. Send-and-split method for minimum concave-cost network flows. *Math. Oper. Res.*, 12:634–664, 1987.

[11] H. Everett. Generalized Lagrange multiplier method for solving problems of optimum allocation of resources. *Operations Research*, 3:399–417, 1963.

[12] A. Federgruen. Centralized planning models for multi-echelon inventory systems under uncertainty. In S.C.Graves et al, editor, *Handbooks in OR & MS, Vol.4*. Elsevier Science Publishers, B.V., 1993.

[13] A.G. Federgruen and P. Zipkin. Computational issues in an infinite-horizon, multi-echelon inventory model. *Operations Research*, 32:818–836, 1984.

[14] M. Florian and P. Robillard. An implicit enumeration algorithm for the concave cost network flow problem. *Management Science*, 18(3):184–193, November 1971.

[15] G. Gallo and C. Sodini. Concave cost minimization on networks. *European Journal of Operations Research*, 3:239–249, 1979.

[16] A.M. Geoffrion and G.W. Graves. Multicommodity distribution system design by Bender's decomposition. *Management Science*, 20(5):822–844, 1974.

[17] D.S. Johnson, J.K. Lenstra, and A.H.G. Rinnooy Kan. The complexity of the network design problem. *Networks*, 8:279–285, 1978.

[18] H.L. Lee and C. Billington. Managing supply chain inventory: pitfalls and opportunities. *Sloan Management Review*, pages 65–73, Spring 1992.

[19] H.L. Lee and C. Billington. Material management in decentralized supply chains. *Operations Research*, 41(5):835–847, 1993.

[20] H.L. Lee and S. Nahmias. Single-product, single-location models. In S.C.Graves et al, editor, *Handbooks in OR & MS, Vol.4*. Elsevier Science Publishers, B.V., 1993.

[21] T.L. Magnanti and R.T. Wong. Network design and transportation planning: Models and algorithms. *Transportation Science*, 18:1–55, 1984.

[22] C.H. Martin, D.C. Dent, and J.C. Eckhart. Integrated production, distribution, and inventory planning at Libbey-Owens-Ford. *Interfaces*, 23(3):68–78, 1993.

[23] S. Nahmias. *Production and Operations Analysis.* Richard D. Irwin, Homewood, IL, 1989.

[24] S. Nahmias and S.A. Smith. Optimizing inventory levels in a two-echelon retailer system with partial lost sales. *Management Science*, 40(5):582–596, 1994.

[25] G.L. Nemhauser and L.A. Wolsey. *Integer and Combinatorial Optimization.* Wiley, New York, NY, 1996.

[26] Y. Pochet and L.A. Wolsey. Lot-size models with backlogging: strong reformulations and cutting planes. *Mathematical Programming*, 40:317–335, 1988.

[27] C.C. Poirier and S.E. Reiter. *Supply Chain Optimization.* Barrett-Koehler, San Francisco,CA, 1996.

[28] J. Pooley. Integrated production and distibution facility planning at Ault Foods. *Interfaces*, 24(4):113-121, 1994.

[29] W.B. Powell and Y. Sheffi. The load planning problem of LTL motor carriers: Problem description and a proposed solution approach. *Trans. Res.*, 17A:471–480, 1983.

[30] H. Scarf. The optimality of (s,S) policies in the dynamic inventory problem. In P.Suppes K.Arrow, S.Karlin, editor, *Mathematical Methods in the Social Sciences.* Stanford University, 1960.

[31] J.F. Shapiro. Mathematical programming models and methods for production planning and scheduling. In S.C.Graves et al, editor, *Handbooks in OR & MS, Vol.4.* Elsevier Science Publishers, B.V., 1993.

[32] R.M. Soland. Optimal facility location with concave costs. *Operations Research*, pages 373–382, 1974.

[33] S. Stock-Patterson. *Dynamic Flow Management Problems in Air Transportation.* PhD thesis, M.I.T., 1997.

[34] D.J. Thomas and P.M. Griffin. Coordinated supply chain management. *European Journal of Operations Research*, 94:1–15, 1996.

[35] B. Van Roy, D.P. Bertsekas, Y. Lee, and J.N. Tsitsiklis. A neuro-dynamic programming approach to retailer inventory management. 1997.

[36] T.J. Van Roy and L.A. Wolsey. Valid inequalities and separation for uncapacitated fixed cost networks. *Operations Research Letters*, 4(3):105–112, 1985.

[37] A.F. Veinott. Minimum concave-cost solution of Leontief substitution models of multi-facility inventory systems. *Operations Research*, 17:262–291, 1969.

[38] H.M. Wagner and T.M. Whitin. Dynamic version of the economic lot size model. *Management Science*, 5:89–96, 1958.

[39] W.I. Zangwill. A deterministic multi-period production scheduling model with backlogging. *Management Science*, 13:105–119, 1966.

[40] W.I. Zangwill. Minimum concave cost flows in certain networks. *Management Science*, 14:429–450, 1968.

[41] W.I. Zangwill. A backlogging model and a multi-echelon model of a dynamic economic lot size production system - a network approach. *Management Science*, 15(9):506–527, May 1969.