2003

A comparison of constraint programming and integer programming for an industrial planning problem

Shelley M. Heist
Lehigh University

Heist, Shelley M.

A Comparison of Constraint Programming and Integer Programming for an Industrial...

May 2003

A Comparison of Constraint Programming and Integer Programming for an Industrial Planning Problem

by

Shelley M. Heist

A Thesis

Presented to the Graduate and Research Committee

of Lehigh University

in Candidacy for the Degree of

Master of Science

in

Industrial and Systems Engineering

Lehigh University

May 2003

This thesis is accepted and approved in partial fulfillment of requirements for the Master of Science.

April 25, 2003

Date

Thesis Advisor

Co-Advisor

Chairperson of Department

Acknowledgements

I am sincerely grateful to Air Products and Chemicals for the internship that led to this research. Dr. Ted Ralphs, Tom Brinker, Andy Bringhurst, Jim Hutton, and Eve Pellecchia provided valuable feedback and ideas during the development and revision of this thesis. I especially want to thank my parents, fiancé, family, and friends for their moral support during my time at Lehigh and always.

Table of Contents

List of Tables

List of Figures

Abstract

In this paper, we explore modeling alternatives and solution approaches for a production and distribution planning problem at Air Products and Chemicals, Inc. The problem is currently modeled as a linear program and solved using a custom built application based on an interior point algorithm. Using ILOG's OPL Studio as a modeling and solution framework, we present both a constraint programming formulation and an integer programming formulation of the planning problem. We compare solution techniques for these two formulations and identify strengths and weaknesses of the alternatives.

1. Introduction

Air Products and Chemicals, Inc. (APCI) is a multinational producer and supplier of atmospheric gases, process and specialty gases, performance materials, and chemical intermediates for customers in the technology, energy, and healthcare markets. The firm uses operations research (OR) techniques to model and solve problems in several business areas. This thesis deals with the formulation and solution of a production and distribution planning problem for one of those business areas.

The problem is to determine a minimum cost production and distribution plan for a product subject to resource constraints on the plants, delivery fleets, and drivers. The problem is not easily categorized since it combines elements of several different traditional operations research models, including the lot sizing and transportation problems, a network flow problem, and a supply chain optimization problem with one tier.

Problems such as those listed above can be modeled and solved using a variety of operations research techniques. Researchers in the mathematical programming community usually take a two-step approach by first formulating a mathematical programming model of the system and then solving the model using one of a variety of techniques. The model defines the problem, variables, and constraints but does not dictate how to find a solution. While the mathematical programming community has had much success in separating problem formulation from solution method, there is another school of thought that encourages the two steps to be combined as one. Constraint programming (CP) is a methodology that encourages the model and the desired solution technique to be declared in a single "formulation."

The production and distribution planning problem at APCI is currently modeled as a linear program (LP), following the traditional mathematical programming paradigm, and solved using an interior point algorithm. While the existing methodology has worked well, there is an interest in discovering how either constraint programming or integer programming (IP) might be used to solve this planning problem. Much of this interest is motivated by APCI's implementation of the enterprise resource planning tool known as Systems, Applications & Products in Data Processing (SAP). SAP is an information-technology based software system that provides companies with a comprehensive solution for managing data and solutions related to financials, human resources, operations, and corporate services. SAP touts the ability of its software to formulate and solve various planning and decision models using the Advanced Planning and Optimization module, which relies on a constraint programming engine called ILOG Scheduler. With the widespread use of SAP in a variety of companies, it is useful to investigate the effectiveness of constraint programming for industrial applications as compared to more traditional mathematical programming techniques. With this in mind, we model and solve APCI's production planning and distribution problem using both a constraint programming approach and an integer programming approach. We will show the formulation of the model as a constraint program, evaluate the success of domain reduction and constraint propagation for the model, and discuss constraint programming as a solution method for other industrial applications. For comparison, we also formulate an integer programming model of the same problem and solve it using traditional techniques such as branch and bound.

To be precise, we introduce some terminology that we will use throughout the paper. The term *problem* refers the process of determining the desired operating state of a system under a given set of conditions. A *mathematical model* is a simplified representation of the problem that captures the essential mathematical relationships between various components of the system. We will use the terms *formulation* and *model* interchangeably to refer to a mathematical model. A model is made up of *variables* and *constraints*; the variables indicate the state of the system, and they can assume values from a specified set called the *domain*. A constraint consists of a *function* expressing the relationship between some variables. When evaluated for a specific set of variables, the function's values must be in a specified domain. A set of values for the model values that satisfies the relationship established is said to *satisfy* the constraint. Finally, a *feasible solution* is a set of values for the model variables that satisfies all the constraints. A mathematical model has an *objective function* involving some or all of variables, which is either maximized or minimized over the set of all feasible solutions.

2. Tree Search Overview

Since tree search is a basic solution technique employed in both constraint and integer programming, we begin with a generic overview of *tree search* as a technique for finding feasible solutions to mathematical models. The essence of tree search is a "divide and conquer" strategy that is applied when the full mathematical model is too complex to analyze explicitly. We "divide" the original feasible set into manageable *subproblems* and "conquer" by attempting to find feasible solutions to the subproblems and then putting the results together to obtain a feasible solution to the full model.

The strategy that we describe here is termed tree search because its graphical representation resembles an upside-down tree (see Figure 1). The *search tree* consists of *nodes* arranged in a hierarchical structure, as in the familiar structure of a "family tree." Associated with each node is a subproblem, defined as a subset of the original feasible set. Each node N in the tree is connected by *branches* to a unique *parent,* whose associated subproblem contains that of N and (optionally) to one or more children, whose subproblems are further divisions of that of N. The children of a node are said to *descend* from the parent. The root node is at the top of the hierarchy and is the sole node without a parent. The subproblem associated with the root node is the original feasible set. A node from which no others descend is called a *leaf.* Figure 1 on the next page shows a graphical representation of a tree.

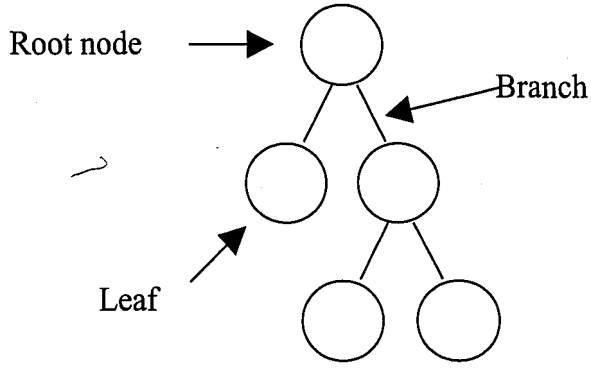

Figure 1: Graphical Representation of Tree Search

Every tree search algorithm is defined by four elements: a *node- processing* procedure, *pruning rules, branching rules*, and *search strategy.* The processing step is applied at every node of the search tree beginning with the root node, and usually involves some attempt to further reduce the feasible set associated with the node by applying logical rules. The pruning rules provide conditions that establish when no further division of S^i is necessary, e.g., when S^i can be shown to be empty. The branching rule is used to decide how a given subproblem S should be broken down into further subproblems S^i such that $\bigcup_{i=1}^{k} S^i = S$. Branching is typically accomplished by partitioning a variable's domain. The variable whose domain gets divided is chosen according to a selection strategy, which is specific to the application. Finally, the search strategy dictates the order in which we processing the subproblems. Two common examples of search strategy include depth-first search and best-first search. Depth-first search proceeds down a branch of the search tree, always selecting a child of the current subproblem to process until the current node can be pruned. If an infeasible node is found during pruning, the algorithm backtracks to most recent node that has been created

but not explored. Best-first search chooses to process the node with the most "potential" according to some criteria, for example, the node with the best value of the objective function.

Overall a tree search algorithm works as follows. A list of nodes that are candidates to be processed is maintained throughout the algorithm. This list initially contains only the root node. At each step in the algorithm, a node is chosen and processed. If the processing results in pruning of the node by one of the pruning rules, then the node is discarded. Otherwise, the node is further divided, resulting in two or more children, which are then added to the list of candidates. This procedure is iterated until no nodes remain on the candidate list. Note that a strong pruning rule is essential to reducing the size of the search tree, since without a pruning rule, exploration of the search tree would amount to complete enumeration of all feasible solutions.

3. Constraint Programming Overview

3.1 Literature Search

Constraint programming, a relatively new technique for solving optimization problems, has its roots in artificial intelligence and computer science. Lustig and Puget [9] trace the history of CP from initial work on constraint satisfaction problems in the 1970s. Research on arc-consistency techniques formed the foundation of CP literature and the development of computer languages specifically for combinatorial problems followed. In the 1980s, this led to the addition of constraint satisfaction algorithms to the Prolog language and then the addition of constraint programming features to general purpose programming languages in the 1990s. Since its inception, constraint programming has seen success in solving optimization problems of a combinatorial nature, most notably in sequencing and scheduling applications. Other applications of constraint programming include the solution of cutting-stock problems [5], crew assignment [3], and network flow problems [2].

The creation of the Optimization Programming Language (OPL) by Van Hentenryck [17] was a major step toward integrating constraint programming with traditional mathematical programming techniques. Since the late 1990s, the idea of integrating constraint programming and mathematical programming has taken off, resulting in a number of research papers comparing constraint programming with traditional mathematical programming. Smith, et al [15] compares CP and IP methods to solve a boat-scheduling problem called the progressive party problem. Their results showed that CP was able to quickly solve several instances that could not be solved using IP

techniques. In [4], the authors use CP and IP to model an allocation of tasks to machines in a manufacturing problem.

Other researchers went one step further and developed hybrid methods that integrate both CP and IP solution methodologies into a single framework. For example, [13] proposes just such a framework for unifying CP and IP models and solution techniques. Using this technique, Kim and Hooker [8] model a fixed-charge network flow problem using logical constraints and 0-1 variables. In the search tree, the logical constraints trigger domain propagation and a projection is used to solve a relaxed problem and obtain a bound on the objective. In another example, Timpe [16] solves a planning and scheduling problem in the chemical industry using a mixed integer program as a master process that solves a small constraint program every few nodes. The result is an efficient solution technique that can prune infeasible subproblems early in the search process.

For additional background material, refer to Hooker [6,7], or [11] and [19] for a comprehensive treatment of constraint programming. In addition, Bartak [1] provides an excellent review of the algorithms that drive constraint programming systems.

3.2 Formulation Techniques

We next discuss some of the underlying concepts of the constraint programming approach, first concentrating on modeling and then solution techniques. To begin, we point out that the word "programming" in the constraint programming literature has a different meaning than the word "programming" as used by the mathematical programming community. In mathematical programming, the word "programming" came about as a term used to define operations and activities "planning" in the U.S. military. In contrast, since CP evolved in the discipline of computer science, the word

"programming" refers to a procedural problem declaration and specification of solution method, as in "computer programming." As such, the constraint program is made up of two pieces: the constraints that comprise the formulation and a search strategy for solving the problem. This is in contrast to a mathematical program, which describes a model but does not specify how to solve it.

Traditionally, a constraint programming formulation consists only of a list of decision variables, domains, a set of constraints to be satisfied, and a search procedure. This traditional definition refers to a *constraint satisfaction problem* (CSP), since a solution is any assignment of values to variables such that all constraints are satisfied. Over time, the CSP has been extended to include an objective function, creating the CP optimization model that we refer to from here forward. Optimization is easily incorporated into constraint programming by adding a constraint to reflect that each successive solution must be better than the last.

In theory, a CP formulation's decision variable domains can consist of integer values, real values, set elements, or even subsets of sets. In practice, however, constraint programming solvers typically require the variables to have *discrete* domains, meaning that they are finite, because of the solution algorithms involved. For example, ILOG's OPL Studio [17] requires the use of constraints that are constructed from discrete data and discrete variables. Lastly, constraint programming formulations achieve significant gains in modeling flexibility due to the fact that variable domains can be non-convex. In other words, where traditional mathematical programming restricts variable domains to discrete or continuous intervals, constraint programming allows the domains to be the union of several intervals, i.e., to have "holes."

Within a CP, the constraints can take many forms and are not required to be linear. A common theme in constraint programming is the use of *global constraints* to represent relationships between variables. Global constraints are constraints used to enforce complex relationships among a number of variables. For example, the most well known global constraint, *all-different*, requires that a set of variables all take different values. The *cumulative* global constraint is used to ensure that a series of jobs are scheduled so that they do not overlap. Global constraints such as these are analogous to function calls within a computer program; the global constraint expresses the relationship between variables and specifies that a specific *domain reduction* technique should be used in the search strategy applied to that constraint. Modern high-level constraint programming software incorporates keywords that represent the global constraints and invoke specialized solution algorithms when they are detected.

Another useful CP technique, called *variable subscripting*, allows expressions of the form *cost[s, producer[s]]* where the variable *producer[s]* is used to index the matrix of costs. Other common CP constructs include logical constraints constructed using statements such as *if-then* or logical, relational, or binary operatives. Lastly, *meta constraints*, or constraints involving other constraints, can also be used in constraint programming. One example of a meta constraint is $\sum_j (x_j = i)$, which is used to count the number of x variables equal to i.

3.3 Node Processing

Given a CP formulation as described above, constraint programming systems use a search tree to find optimal solutions. The CP system iteratively applies the node

processing techniques known as *constraint propagation* and *domain reduction* to attempt to reduce the feasible set that defines the subproblem as much as possible. Constraint propagation and domain reduction work together to reduce the variable domains as much as possible. First, some constraint is selected from the formulation and a *consistency technique* associated with the constraint is applied to the relevant variables and their domains. The consistency technique operates on a *constraint graph* where the nodes correspond to variables and the edges are labeled by constraints. Which particular consistency technique is used depends on the type of constraint being examined. For example, if the constraint is *all-different*, an edge-finder algorithm is used in domain reduction; flow algorithms are used to reduce the domains of when variable subscripting is used. The algorithm operates on the graph to identify values that cannot occur in any partial solution and removes them from the variable domains. See [1] for a more thorough explanation of consistency techniques and algorithms. The smaller domains are passed to other constraints where they are further reduced, thus implementing a form of constraint propagation. Note that each node represents a partial solution over some subset of the model variables. For an example of how node processing works, see Section 6.2.

3.4 Branching

In CP terminology, the branching rule is called a *search strategy*. Thus, the search strategy constructs the search tree by specifying an ordering of variables to branch on and an associated ordering of the values to try. For example, if we have the variables x and y and their associated domains D_x and D_y, a generic search strategy might specify "assign values to the x variable first, and try the values of D_x in increasing order." Most

constraint programming systems provide a default search strategy that can be used in the absence of a search tailored to the particular problem. One common strategy is to identify the variable with the smallest domain as a starting point and branch by partitioning that variable's domain into two or more pieces. This "first-fail" principle operates on the idea that it is best to begin with the variable that is the most restricted and find infeasibilities early to limit the size of the search tree. As an example of CP branching, consider a variable x whose domain has been reduced to $D_x = \{1,2,3,4\}$. Subproblems are created by setting $D_x = \{1,2,\}$ in one child and $D_x = \{3,4\}$ in another child.

3.5 Pruning

A node can be pruned at any point of the search tree if one of the following situations is encountered. First, a node can be pruned if the domains of all the variables are reduced to a singleton, at which point a full solution is found. Second, if any variable's domain is reduced to the empty set, then an infeasible partial solution has been identified and the node can be pruned. Third, a node can be pruned if domain reduction has implied that the optimal objective function value of the associated subproblem is no better than the best objective value found so far.

3.6 Search Procedure

The constraint programming search tree is explored using a *search procedure*, which details the order in which nodes should be considered for processing (see Section 6.2 for an example). Depth-first search is commonly used as the default search procedure in constraint programming systems.

3.7 Overall Algorithm

The techniques of node processing, branching (search strategy), pruning, and search procedure all work together in an iterative algorithm to limit the size of the search tree and find solutions in an efficient manner (see Section 2). The algorithm begins with the root node and can be summarized as follows:

1. Apply constraint propagation and domain reduction to the current node to eliminate as many values from the variable domains as possible.
2. Evaluate the node according to the pruning rules. If the node is feasible, proceed to step 3. If the node is infeasible proceed to step 4.
3. Use the search strategy to choose a variable and branch by splitting the variable's domain.
4. Choose a node for processing according to the search procedure and return to step 1.

Many constraint programming systems describe the search tree in terms of *choice points*, *failures*, and *solutions*. Choice points correspond to non-terminal nodes of the tree from which some branching occurs. Failures are leaves of the tree that represent infeasible partial solutions, and solutions occur at leaves of the tree where every variable's domain has been reduced to a single value.

4. Integer Programming Background

4.1 Historical Review

Mathematical programming originated with the seminal work in linear programming by George Dantzig for the United States Department of Defense in 1947. Since then, mathematical programming and optimization techniques have come to guide decision-making in nearly every industry and every large company worldwide. In an attempt to model and solve more complex problems, operations researchers have added nonlinear, dynamic, integer, convex, and semi-definite programming, among others, to the repertoire of mathematical programming models with well-developed solution techniques. One particularly well-developed solution technique, called *branch and bound*, was introduced by Land and Doig [10]. Branch and bound is frequently used to find solutions to integer programming problems; it is a tree search procedure in which a bound on the optimal value to each subproblem is obtained by solving a relaxation. In this thesis we assume the use of a linear programming relaxation in the branch and bound tree.

Since we will be comparing CP techniques to integer programming modeling and solution techniques, we provide a brief review of the most fundamental concepts in IP. For a complete treatment, we refer the reader to the text of Nemhauser and Wolsey [12].

4.2 Formulation Techniques

A pure integer program consists of a linear objective function, $c \in R^n$, subject to linear constraints, $[A,b] \in R^{m \times n+1}$, as follows:

$$\min \quad cx$$

$$s.t. \quad Ax \geq b$$

$$x \in Z^n$$

The pure IP is a special case of the Mixed Integer Program (MIP), which can contain both integer- and real-valued variables. Yet another variation is the Binary Integer Program (BIP), in which integer variables are restricted to the values of {0,1}.

At first glance, integer programs seem to place more restrictions on the model than do constraint programs. Despite the apparent restrictions on the form that constraints and variables can take, we emphasize that there is a rich potential for modeling real world problems using IP, which we illustrate with a few examples. Suppose we want to include a variable in our model whose actual domain is a set of cities, such as {Allentown, Bethlehem, Easton}. An IP model could accommodate this using a mapping of integers to represent the set, i.e. {1, 2, 3} where 1 = Allentown, 2 = Bethlehem, and 3 = Easton. As a second example of the flexibility of IP, consider how the global *all-different* constraint used in constraint programming can be integrated into an integer program using the binary variable $y_{ij} \in \{0,1\}$, which takes the value 1 if $x_i = j$ and 0 otherwise. While it is not as succinct as the CP version, an IP example of *all-different*(x_1, x_2, x_3) is modeled on the following page.

$$x_i = \sum_{j=1}^{3} j\, y_{ij} \quad \forall\, i \in 1..3$$

$$\sum_{i=1}^{3} y_{ij} = 1 \quad \forall\, j \in 1..3$$

$$\sum_{j=1}^{3} y_{ij} = 1 \quad \forall\, i \in 1..3$$

$$x_i \in \{1, 2, 3\},\ y_{ij} \in \{0,1\}$$

Even if-then constraints can be modeled with the addition of 0-1 variables to an IP model. However, the number of extra variables that must be added to achieve this flexibility can make the IP become difficult or nearly impossible to solve.

4.3 Node Processing

The bounding phase of branch and bound is a node processing procedure that occurs at every node beginning with the root. We obtain a bound on the objective value by solving some relaxation of the integer program. Usually, this is a linear programming relaxation that is created by simply dropping the integrality constraints on the variables and solving the relaxation using an LP algorithm. Solving the linear relaxation gives a full solution over all the variables and objective value Z_{LP}, which is retained as a lower bound on the optimal integer objective value if we are minimizing, or an upper bound if we are maximizing. This bound is used both in the pruning procedure and in guiding the search procedure.

4.4 Branching

Branching occurs in the branch and bound tree by examining the solution to the LP relaxation at the current node. If the solution to the LP relaxation contains all integer-

valued variables, a feasible solution is found. Otherwise, we choose a variable with a fractional value to branch on according to a given selection criteria and branch by rounding the variable up in one branch and rounding it down in the other. For example, if the relaxation produces a solution containing $x_i = 2.5$, we create two new subproblems by adding the constraint $x_i \leq 2$ to the new node in the left branch, and $x_i \geq 3$ to the node in the right branch. Generally speaking, branching occurs by imposing additional linear constraints that serve to partition the feasible region of a subproblem into smaller pieces.

4.5 Pruning

Nodes can be pruned in the branch and bound tree if one of three situations occurs. First, we can prune a node if it's objective function value is worse than the best incumbent integer solution retained so far, because we know that the objective function will only become worse as we proceed down a branch of the tree. Second, a node can be pruned if it solves to an integer solution during node processing. Lastly, a node can be pruned if it is determined to be infeasible during the node-processing step.

4.6 Search Procedure

The last piece of a branch and bound algorithm is a search strategy specifying the order in which to explore nodes of the tree. Depth-first search can be used to save memory in a branch and bound tree, since we must only remember the difference in solution between two successive nodes of the search tree. The advantage of depth-first search is that it tends to produce feasible solutions early in the search process. Alternatively, best-first search, e.g., always choosing to process the node with the best bound, can be used when there exists a good bound on the optimal objective function

value. This approach tends to minimize the size of the search tree. In practice, a combination of depth-first and best-first search is implemented by diving down the search tree to find a feasible solution and then choosing to process the node with the best bound.

4.7 Overall Algorithm

As detailed in the previous sections, branch and bound relies on a divide and conquer strategy to explore the finite space of feasible integer solutions. At every node in the branch and bound tree, beginning with the root node, we perform the following steps:

1. Perform node processing by solving a relaxation of the current problem.
2. Evaluate the node according to the pruning rules. If the node is feasible, proceed to step 3. If the node is infeasible proceed to step 4.
3. Choose a fractional variable to branch on.
4. Choose a node for processing according to the search procedure and return to step 1

The basic branch and bound algorithm described above can be augmented to specify more unique search strategies. One idea is to create more complex branching rules by branching on sets of variables. Another idea, implemented in *branch and cut* algorithms, is the addition of *valid inequalities* or *cutting planes* to each node of the search tree. The cutting planes tighten the relaxation by eliminating fractional solutions, leading to a closer approximation to the convex hull of integer solutions.

5. Problem Description

The production and distribution planning problem at Air Products and Chemicals, Inc. (APCI) is to determine a minimum cost production plan and distribution framework for a product subject to resource constraints on the plants, delivery fleets, and drivers. The production and distribution network under consideration is composed of production plants, fleets of vehicles, driver pools, and customer zones. There is an existing set of plants across the United States and Canada that can be used for production, and APCI also has its own fleet of delivery vehicles, some of which are located at production plant sites. Each fleet is co-located with a pool of drivers that can be used in the delivery process. The customer demand is grouped into a few large geographical regions, and within a region the demand is grouped into clusters called customer zones. The model does not accommodate individual customer demands because it would simply be too cumbersome and is only marginally beneficial.

Finally, the travel links that traverse the production and distribution network are based on a pre-specified set of "movements" that dictate allowable combinations of plant, fleet, and customer zone. Movements are generated only for customer zones that incur demand in a particular period. They are useful in eliminating undesirable pickup and delivery combinations. There are three different types of movements that can occur, and they are illustrated in Figure 2.

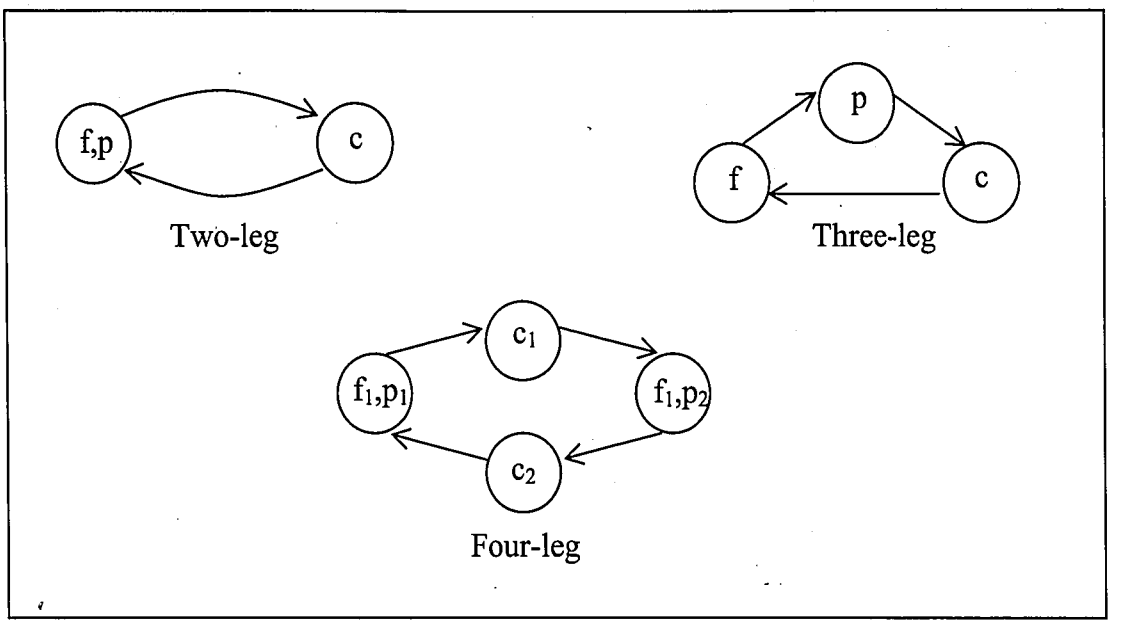

Figure 2: Types of Movements

A two-leg movement can only occur when a delivery uses a co-located plant and fleet to satisfy demand in a customer zone. The fleet picks up product at its base, travels to the customer, and then returns to where it started. Three-leg movements occur when a plant and fleet are not co-located. The fleet begins at its base, travels to a plant to pick up product, and then delivers to the customer zone before returning home. Finally, four-leg movements work as follows: co-located plant and fleet are used to satisfy a customer zone, but before returning to its base the same fleet picks up at a different plant and delivers to a second customer zone.

APCI uses a hierarchy of three different planning models to determine the production, fleet assignment, and driver utilization for liquid and bulk product. The hierarchy consists of a long term model, with a time horizon of several years to guide the company's strategic plan; a mid-term planning model solved over a period of months for budgeting; and an operational planning model solved on a monthly basis to guide the

actual production and distribution. In this paper we are concerned with operational planning and therefore our model deals with a planning horizon of one month.

Currently, APCI models the problem described above as a linear program, and solves the linear programming relaxation using a custom built application called Linear Programming System (LiPS). LiPS interfaces with SQL databases to filter and obtain data, thus creating a case or a specific data set. Once the case is created, LiPS sends the data to Dash Optimization's XPRESS-MP software, which creates the model code, solves the problem using a barrier method, and sends the solution back to LiPS for reporting. The solution is examined and the variables relevant to number of trucks, number of drivers, etc. are rounded to be integer. The actual mathematical model used in the process is proprietary, but we provide a simplified conceptual model to help characterize the problem. The conceptual model does not include all of the decision variables and constraints that are addressed in the full implementation at Air Products.

To sum up, each month the firm has several production decisions to make, namely: which plants to use, how much product to produce at each plant, and how much product to purchase from competitors. On the distribution side, we must determine the volume of product to ship between plants and customer zones, as well as which fleets to use for each delivery. Note that the model determines the customer zone demand that a particular fleet and driver pool will satisfy, but not the routing sequence that will take place for the fleet over the course of the planning period. The detailed distribution scheduling decisions are carried out by a scheduler using a separate modeling system.

The next few tables describe the notation used in the linear program. First, we define the sets used throughout the problem:

S = The set of all plants.
F = The set of all fleets.
D = The set of all driver pools.
C = The set of customer zones.

The decision variables used are as follows:

Production Variables	X_i = Volume of product to produce at plant I
	Y_i = Binary operating variable for plant I
	C_i = Volume of product to purchase from competitors for plant i
Distribution Variables	Z_{icjf} = Volume delivered from plant i to customer zone c returning to plant j using fleet f
	W_{ijf} = Empty return trip from plant j to plant i using fleet f.
	M_f = Total distance traveled by fleet f
	H_f = Total delivery time for fleet f
	L_d = Total loading and unloading time by driver pool d

The costs associated with the decision variables are:

Production Costs	p_i = Variable production cost at plant i
	q_i = Fixed production cost at plant i
	o_i = Outsourcing cost for plant i
Distribution Costs	m_d = Operating cost of driver d (\$/mile)
	v_f = Operating cost of vehicle in fleet f (\$/mile)
	l_d = Loading and unloading costs for driver pool d (\$/hr)
	e_f = Loading and unloading costs for vehicle in fleet f (\$/hr)

Finally, we have some additional problem parameters:

l_i = minimum desired production for plant i
u_i = maximum production capacity for plant i
d_c = demand at customer zone c
c_f = capacity of vehicles in fleet f
t_d = driver hours available in driver pool d (driver hrs/day * # of drivers * # of days)
t_f = fleet hours available in fleet f (vehicle hrs/day * # of vehicles * # of days)
δ_{ij} = one way distance between locations i and j
t_{ij} = one way driving time between locations i and j
p_f = average loading time for fleet f

Using the notation described on the previous page, we arrive at the following mixed integer program.

$$\min \sum_{i \in S} p_i X_i + q_i Y_i + o_i C_i + \sum_{d \in D} \sum_{f \in F} (m_d + v_f) M_f + (2 l_d + e_f) L_d$$

$$X_i \le u_i Y_i \qquad \forall i \in S \qquad (1)$$

$$X_i \ge l_i Y_i \qquad \forall i \in S \qquad (2)$$

$$X_i + C_i - \sum_{c \in C} \sum_{j \in S} \sum_{f \in F} Z_{icjf} \ge 0 \qquad \forall i \in S \qquad (3)$$

$$\sum_{i \in S} \sum_{j \in S} \sum_{f \in F} Z_{icjf} - d_c \ge 0 \qquad \forall c \in C \qquad (4)$$

$$M_f = \sum_{i \in S} \sum_{c \in C} \sum_{j \in S} (\delta_{ic} + \delta_{jc})(Z_{icjf} / c_f) + \delta_{ij} W_{ijf} \qquad \forall f \in F \qquad (5)$$

$$H_f = \sum_{i \in S} \sum_{c \in C} \sum_{j \in S} (t_{ic} + t_{jc})(Z_{icjf} / c_f) + t_{ij} W_{ijf} \qquad \forall f \in F \qquad (6)$$

$$L_f = \sum_{i \in S} \sum_{c \in C} \sum_{j \in S} p_f (Z_{icjf} / c_f) \qquad \forall f \in F \qquad (7)$$

$$H_f + 2L_f \le t_d \qquad \forall d \in D, f = d \qquad (8)$$

$$H_f + L_f \le t_f \qquad \forall f \in F, d = f \qquad (9)$$

$$\sum_{c \in C} \sum_{j \in S} \sum_{f \in F} Z_{icjf} + W_{ijf} - Z_{jcif} - W_{jif} = 0 \qquad \forall i \in S \qquad (10)$$

(11)

$$X_i, C_i \ge 0 \qquad \forall i \in S$$

$$Y_i \in 0..1 \qquad \forall i \in S$$

$$Z_{icjf} \ge 0 \qquad \forall i, j \in S, \forall c \in C, \forall f \in F$$

$$M_f, H_f \ge 0 \qquad \forall f \in F$$

$$L_d \ge 0 \qquad \forall d \in D$$

The objective of the problem is to minimize fixed and variable production costs, outsourcing costs, mileage costs, loading and unloading costs. Constraint 1 prevents production from occurring at a plant that is not operating and ensures that production at plant *i* does not exceed capacity. Similarly, constraint 2 ensures that some minimum level of production is met if plant *i* is in operation. The third constraint enforces that the amount of product shipped to customer zones must be less than or equal to the total product produced or purchased at a plant. Constraint 4 provides that enough product is shipped into a zone to meet the customer zone demand. Constraints 5 and 6 calculate the total miles and driving hours that a fleet is used, respectively, and constraint 7 calculates the total loading and unloading time for fleet *f*. Constraint 8 states that the total driving time and loading/unloading time cannot exceed the driver hours available in driver pool *d*. Similarly, constraint 9 ensures that driving, loading, and unloading time does not exceed the total time available for fleet *f*. Constraint 10 is a flow balance constraint for three-leg movements. Finally, constraints 11 are the non-negativity and binary constraints on the decision variables.

As stated before, the model presented in this paper is a simplified version of the actual model. We have chosen to omit several realities that would make the mathematical model more complicated, for example: start-up costs for dormant production sites, the trade-off between inexpensive and expensive production resources, and outsourcing drivers at an additional cost. We do not consider the option of purchasing new vehicles or hiring new drivers. Finally, inventory was not considered because the product in question is usually only produced to satisfy contractual demand and is not stored over time.

6. Constraint Programming Approach

6.1 Formulation

The constraint programming formulation presented in this paper was created using expert knowledge of APCI's planning problem to deduce patterns of problem solution and implementation. The biggest observation guiding the structure of the formulation is that, with few exceptions, solutions to the existing linear programming problem involve each customer zone being sourced by one single production facility. Past solutions show that only those customer zones with particularly high demand were supplied by more than one production facility, and all other customer zones had their full demand supplied by a single production facility. This solution structure brings to mind the generalized assignment problem, which has historically been solved effectively using the tools of constraint programming. To leverage this, we created a constraint programming formulation that assigns one plant as a supplier and one fleet as a shipper to each customer zone so that no demand splitting occurs.

Based on historical data, we also notice that the set of open production facilities is very stable, meaning that the set of plants that are open for production does not change much over time and decisions to open and close production sites are made infrequently. With this in mind and since the focus of this paper is on operational planning, we have eliminated the binary Y_i variables from the constraint program.

A third observation gathered from the data is that the fleets and driver pools are geographically co-located. Thus, we can do away with the set D of drivers, and interpret each fleet in F as having both vehicles and drivers, each with their own resource constraints. This eliminates one set of subscripts and makes for a more readable

program. When a fleet is assigned to distribute product to a customer zone, use of both the vehicles and the drivers is implied.

Having described some of the modeling considerations, we now cover the notation used in the constraint program. The sets are defined as follows:

P = The set of all plants.
F = The set of all fleets and driver pools.
C = The set of customer zones.

As mentioned before, several variables presented in this formulation take on values by assignment. We use variables of the form

$$Supplier[c] = p \text{ and } Shipper[c] = f$$

in which one plant p is assigned to supply the full demand of customer zone c, and one fleet f is assigned to deliver the full demand of customer zone c. Each variable has a specific domain from which it is assigned a value. The variables and their domains are:

Variable	**Domain**
$Supplier_c$ = plant assigned to supply customer zone c demand	P
$Shipper_c$ = fleet assigned to ship customer zone c demand	F
$Miles_f$ = total miles traveled by fleet f	Integers in 0..maxMiles
$DriveMinutes_f$ = total driving minutes for fleet f	Integers in 0..maxMinutes
$LoadMinutes_f$ = total loading/unloading minutes for fleet f	Integers in 0..maxLoadMinutes

In the domains above, the upper bounds maxMiles, maxMinutes, and maxLoadMinutes are determined heuristically using the problem data.

The cost notation and other parameters are the same as presented in the last section and will not be repeated here. The only additional parameter we need is *trips[c]*, which is the number of trips needed to satisfy the demand at customer zone c. This

parameter is calculated by dividing the demand of customer zone *c* by the vehicle capacity and rounding up to the nearest integer.

Following Lustig [9] and others, we present the constraint program as it would be written in OPL, Optimization Programming Language. The constraints are numbered for ease of discussion, but these numbers do not appear in the actual program.

```
enum plants = ...;
enum fleets = ...;
enum custZones = ...;
int+ pVariableCost[plants] = ...;
int+ prodmin[plants] = ...;
int+ prodmax[plants] = ...;
int+ fVariableCost[fleets] = ...;
int+ fCapacity = ...;
int+ fMinutes = ...;
int+ vehicles[fleets] = ...;
int+ dVariableCost[fleets] = ...;
int+ dLoadCost[fleets] = ...;
int+ drivers[fleets] = ...;
int+ dMinutes = ...;
int+ loadTime = ...;
int+ demand[custZones] = ...;
int+ pfdistance[plants, fleets] = ...;
int+ fcdistance[fleets, custZones] = ...;
int+ pcdistance[plants, custZones] = ...;
int+ pfdrivingTime[plants, fleets] = ...;
int+ fcdrivingTime[fleets, custZones] = ...;
int+ pcdrivingTime[plants, custZones] = ...;
int+ trips[custZones] = ...;
int days = ...;
```

Figure 3: OPL Representation of the Constraint Program Parameters

```
var plants Supplier[custZones];
var fleets Shipper[custZones];
var int+ ProdArray[plants,custZones] in 0..1;
var int+ ShipArray[fleets,custZones] in 0..1;
var int+ VarProdCost[plants] in 0..maxProdCost;
var int+ VarFleetCost[fleets] in 0.. maxFleetCost;
var int+ VarDriverCost[fleets] in 0.. maxDriverCost;
var int+ LoadUnloadCost[fleets] in 0.. maxLoadCost;

minimize  sum(p in plants) VarProdCost[p] + sum(f in fleets) VarFleetCost[f] +
VarDriverCost[f]  + LoadUnloadCost[f]

subject to{
1. forall(p in plants)
   VarProdCost[p] = sum(c in custZones) VariableCost[p]*ProdArray[p,c]*demand[c];

2. forall(f in fleets)
   VarFleetCost[f] = sum(c in custZones) ShipArray[f,c] * (pfdistance[Supplier[c],f] +
   pcdistance[Supplier[c],c] + fcdistance[f,c]) * fVariableCost[f] * trips[c];

3. forall(f in fleets)
   VarDriverCost[f] = sum(c in custZones) ShipArray[f,c] * (pfdistance[Supplier[c],f] +
   pcdistance[Supplier[c],c] + fcdistance[f,c]) * dVariableCost[f] * trips[c];

4. forall(f in fleets)
   LoadUnloadCost[f] = sum(c in custZones) ShipArray[f,c] * trips[c] * loadTime * 2 *
   dLoadCost[f];

5. forall(p in plants, c in custZones) (Supplier[c]=p) => (ProdArray[p,c]=1);

6. forall(f in fleets, c in custZones) (Shipper[c]=f) => (ShipArray[f,c]=1);

7. forall(p in plants) sum(c in custZones) ProdArray[p,c]*demand[c] <= prodmax[p]*days;

8. forall(p in plants) sum(c in custZones) ProdArray[p,c] >= prodmin[p]*days;

9. forall(f in fleets)
  sum(c in custZones) ShipArray[f,c]*trips[c]*(pfdrivingTime[Supplier[c],f] +
  pcdrivingTime[Supplier[c],c] +fcdrivingTime[f,c]+loadTime*2 )<= drivers[f]*dMinutes;

10. forall(f in fleets)
    sum(c in custZones) ShipArray[f,c]*trips[c]*(pfdrivingTime[Supplier[c],f] +
    pcdrivingTime[Supplier[c],c] + fcdrivingTime[f,c] + loadTime*2 ) <=
    vehicles[f]*fMinutes*days;
};

search{ generate(Shipper); generate(Supplier); };
```

Figure 4: OPL Representation of the Constraint Program

Constraint 1 calculates the production cost at each plant *p*. Constraints 2 and 3 calculate the per-mile cost for each fleet *f* and for each driver in fleet *f*, respectively. These last two constraints use a CP technique known as variable subscripting, described in Section 4.2. Constraint 4, also a calculation, computes the total time spent loading and unloading by a fleet *f*. Note that the structure imposed by constraints 3 and 4 allows for the modeling of two and three-leg movements but not four-leg movements. Constraints 5 and 6, two more nonlinear constraints, use logical if-then modeling to enforce relationships between variables. Constraint 5 states that "if plant *p* is the supplier of customer zone *c*, then assign a 1 to row *p*, column *c* of the variable *ProdArray*." Similarly, constraint 6 ties together the shipper assigned to customer zone *c* and the variable *ShipArray*. The purpose of these two constraints is to create a record in memory of the customer zones serviced by each plant and fleet, to be used with capacity constraints later on. The problem can be modeled without these extra *ProdArray* and *ShipArray* variables using logical constraints, but their presence aids constraint propagation and increases solution speed. Constraint 7 ensures that plant p can satisfy the demand assigned to it based on its capacity over the planning period, and constraint 8 is a minimum production requirement for each plant. Constraints 9 and 10 are more nonlinear constraints that use the variable subscripting technique. They constrain the variable assignments according to the number of driver hours and fleet hours that can be used in the planning period.

We have made use of several constraint programming modeling constructs in the constraint program presented here, including the use of variable subscripting and the modeling of logical conditions. In addition, we purposely created several cost variables

instead of calculating the total cost in the objective function. Separating out the costs allows for better constraint propagation throughout the program.

The search procedure for the constraint program uses OPL's *generate* function. The *generate(variable)* statement instructs the solver to create the search tree according to the first-fail principle we described in Section 3.4. The solver chooses the variable with the smallest domain size and assigns to that variable the smallest value in its domain, since this is a minimization problem. In the computational results section, we experimented with other search procedures for this problem, including the use of a "first-feasible" search procedure and a maximal regret heuristic.

6.2 Search Tree Example

We use an example problem with 3 plants, 2 fleets, and 3 customers to illustrate the construction and exploration of the CP search tree. Suppose we have data for the problem as shown in Figure 5.

```
plants = {PlantA, PlantB, PlantS};
fleets = {FleetM, FleetC};
custZones = {NewMexico,Texas, Alabama};

pVariableCost = [39, 43, 10];
prodmin = [2, 2, 3];
prodmax = [3, 3, 3];

fVariableCost = [7,7];
fCapacity = 3;
fMinutes = 9;
vehicles = [5, 7];

dVariableCost = [49, 47];
dLoadCost = [32, 31];
drivers = [6, 10];
dMinutes = 78;
loadTime = 1;

demand = [20, 30, 24];

pfdistance = [ [14,  7], [14,  7], [5 ,  6] ];
fcdistance = [ [21, 17, 11], [11, 6, 5] ];
pcdistance = [ [12, 5, 4], [12, 5, 4], [16, 12, 8]];

pfdrivingTime = [ [16,  8], [16,  8], [5 ,  7]];
fcdrivingTime = [ [46, 37, 24], [23, 14, 13] ];
pcdrivingTime = [ [27, 12, 9], [27, 12, 9], [36, 27, 18] ];

trips = [7, 10, 8];

days = 31;
```

Figure 5: Data for the CP Search Tree Example

The search tree produced in ILOG's Solver is shown in Figure 6, below. The nodes are numbered to reflect the order in which they are explored. Black nodes indicate choice points, light grey nodes are feasible solutions to the problem, and dark grey nodes indicate failures. Note that the failures represent both infeasible solutions and nodes that have been pruned due to sub-optimality.

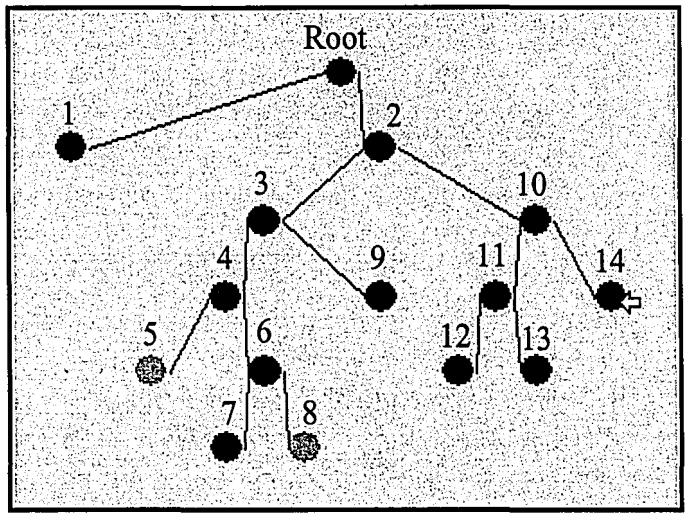

Figure 6: CP Search Tree Example

The effects of constraint propagation and domain reduction can be seen in the root node of the search tree. Constraint 9 of the CP, the constraint on driver hours, is used to eliminate Fleet M from the domain of Shipper[Texas], since the minimum total delivery time over all possible suppliers is 10 * (16 +12 + 37 + 1 * 2) = 670, but Fleet M is constrained to 468 driver hours per period. Using this information, the domain of Shipper[Texas] is reduced to Fleet C, and the propagation of this information leads to the assignment ShipArray[Fleet C, Texas]=1. The partial assignment is used to re-calculate the fleet cost lower bounds using logical implication. The new delivery cost domains are:

Fleet	VarFleetCost	VarDriverCost	LoadUnloadCost
M	[0..4795]	[0..33565]	[0..960]
C	[1050..6489]	[7050..43569]	[620..1550]

From the root node, the search procedure branches on the domains of Shipper[New Mexico] and Shipper[Alabama] as dictated by the search strategy *generate*, since the Shipper variables have the smallest domains. Due to the branching, in node 1 we have Shipper[New Mexico] = Fleet M and Shipper[Alabama] = Fleet C; these assignments were made according to the *generate* strategy since they are most likely to fail. The reduced domains are communicated to the constraints and the partial assignment is found to be infeasible, since Fleet C only has 110 driver minutes left per period and the assignment of Fleet C to Alabama will require at least 240 minutes. The search tree next explores node 2, where branching has dictated that Shipper[NewMexico] = Fleet C and Shipper [Alabama] = Fleet M. Consequently, as a result of constraint propagation and domain reduction, the ShipArray variables have all been fixed at 0 or 1, corresponding to the Shipper assignments. The assignments are propagated to the constraints that contain Shipper and ShipArray variables, and the information is used to calculate the new delivery costs below.

Fleet	VarFleetCost	VarDriverCost	LoadUnloadCost
M	[1120..2296]	[7840..16072]	512
C	[2030..4529]	[13630..30409]	1054

All the Shipper and ShipArray variables have been assigned, and so we branch on Supplier domains from node 2 as specified by the search strategy. In node 3 the domain Supplier[New Mexico] has been reduced to {Plant A} and the domain Supplier[Texas] has been reduced to {Plant A, Plant B}. Node 4 is created by further branching on the domain of Supplier[Texas], which is reduced to {Plant A}, and at node 5 the branching has fixed Supplier[Alabama] = Plant A. At this point all of the variable domains have

been reduced to singletons, and we have the first full solution with an objective value of $Z_{CP} = 38504$. This objective value is retained as an incumbent solution and is used as an upper bound to prune search nodes. For example, node 6 is pruned because its objective value $Z_{CP} = 47560$ is greater than the incumbent solution. Branching, domain reduction, and constraint propagation continue as described above. To summarize, node 8 produces the second feasible solution with the new objective bound $Z_{CP} = 35568$. The rest of the search tree is explored and nodes are pruned due to sub-optimality. The search has exhausted all of the options and the last best feasible solution, from node 8, is the optimal solution to the problem.

7. Integer Programming Approach

7.1 Formulation

For comparison purposes, we present a binary integer program of the production planning and distribution problem. The input data and parameters are identical to those presented in Figure 2 of Section 6. For ease of comparison with the CP model, the BIP is presented in the OPL language below.

```
var int Delivery[plants,fleets,custZones] in 0..1;

minimize
  sum(p in plants, f in fleets, c in custZones)
  (demand[c] * pVariableCost[p] * Delivery[p,f,c] +
  (pfdistance[p,f] + pcdistance[p,c] + fcdistance[f,c]) * trips[c] * fVariableCost[f] *
Delivery[p,f,c] +
  (pfdistance[p,f] + pcdistance[p,c] + fcdistance[f,c]) * trips[c] * dVariableCost[f] *
Delivery[p,f,c] +
  dLoadCost[f] * 2 * loadTime * trips[c] * Delivery[p,f,c]  )

subject to{

1. forall(c in custZones)
  sum(p in plants, f in fleets) Delivery[p,f,c] = 1;

2. forall(p in plants)
  sum(f in fleets, c in custZones) Delivery[p,f,c] * demand[c] <= prodmax[p] * days;

3. forall(f in fleets)
  sum(p in plants, c in custZones)
      ((2 * loadTime + pfdrivingTime[p,f] + pcdrivingTime[p,c] + fcdrivingTime[f,c]) *
       Delivery[p,f,c] * trips[c] ) <= drivers[f] * dMinutes;

4. forall(f in fleets)
  sum(p in plants, c in custZones)
       ((2 * loadTime + pfdrivingTime[p,f] + pcdrivingTime[p.c] + fcdrivingTime[f,c]) *
       Delivery[p,f,c] * trips[c]  ) <= vehicles[f] * fMinutes * days;
```

Figure 7: OPL representation of the Binary Integer Program

The decision variable Delivery[plants,fleets,custZones] is assigned the value 1 if the full demand of customer zone *c* is supplied by plant *p* using fleet *f*, and 0 otherwise.

The objective is to minimize total production, fleet, and driver costs subject to several constraints. Constraint 1 enforces the "no demand splitting rule," by allowing only one combination of plant and fleet to equal 1 for each customer in the Delivery[p,f,c] matrix. Constraint 2 is a maximum production capacity constraint for each plant. Constraint 3 is a restriction on the total loading, unloading, and driving time per driver pool. Similarly, constraint 4 restricts the total loading, unloading, and driving time per fleet.

7.2 Search Tree Example

We use an example problem with 3 plants, 3 fleets, and 6 customers to illustrate the construction and exploration of the BIP search tree. CPLEX is able to solve this problem in the root node of the search tree, so we turned off the node presolve for this example. This essentially prevented CPLEX from using its own domain reduction techniques at each node.

Suppose we have a set of data and parameters relevant to the following enumerated sets:

plants = {Plant A, Plant P, Plant S}

fleets = {Fleet D, Fleet G, Fleet N}

custZones = {Florida, Pennsylvania, Ohio, Alabama, New York, Texas }

The search tree that would be generated to solve the BIP is shown in Figure 8 on the next page. The nodes are numbered to reflect the order in which they are explored.

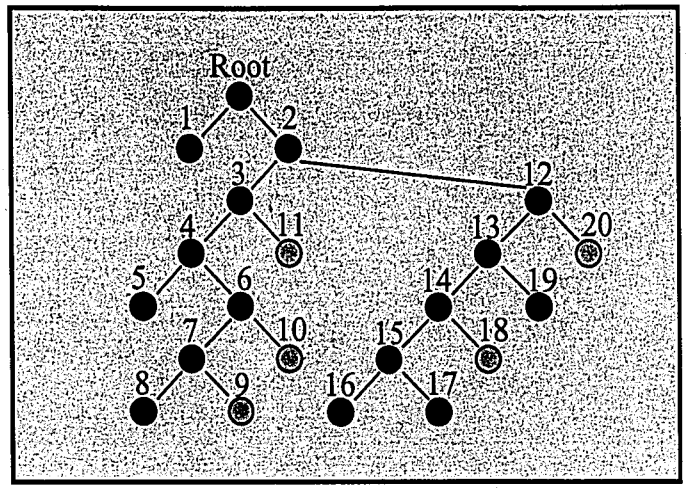

Figure 8: BIP Search Tree Example

As in Section 6.2, we will describe a few steps of the search tree and then summarize the rest. At the root node, an LP relaxation of the BIP is solved to obtain the objective value $Z_{LP} = 583.2$. There are two variables in the root node that have fractional values: Delivery[Plant S, Fleet D, Florida] = 0.77 and Delivery[Plant S, Fleet G, Florida] = 0.23. In this problem, fixing a variable to 1 causes eight other variables to be fixed at 0, so we continually select branching variables that are closest to 1 and set them to 1. Therefore, from the root node, we choose to branch by setting Delivery[Plant S, Fleet D, Florida] = 1 in node 1, and Delivery[Plant S, Fleet D, Florida] = 0 in node 2. With the added constraint, the LP relaxation at node 1 proves to be infeasible, so node 1 is pruned and we move on to node 2. Solving the LP relaxation at node 2 gives the fractional values Delivery[Plant S, Fleet D, Ohio] = 0.41 and Delivery[Plant S, Fleet G, Ohio] = 0.59. We branch on the fractional solution by creating node 3 and adding the constraint Delivery[Plant S, Fleet G, Ohio] = 1. The search proceeds in this same manner, i.e. solving the LP relaxation and branching on fractional variables for nodes 4, 6, and 7, until a feasible integer solution is found at node 9. Before reaching node 9, nodes 5 and 8

were pruned by infeasibility. The objective value at node 9, $Z_{IP} = 611$, is now used as an upper bound to prune nodes that are suboptimal. At node 10, another feasible integer solution is found and the bound is updated to $Z_{IP} = 590$. The bound is updated to $Z_{IP} = 588$ at node 11 since another feasible integer solution has been found. Bounding and branching occur at nodes 12 through 15 and node 16 is pruned due to infeasibility. Node 17 is pruned by suboptimality since its objective value, $Z_{LP} = 599.8$, is greater than the current best objective. At node 18 the bound is updated to $Z_{IP} = 587$, and then node 19 is pruned because its objective value, $Z_{LP} = 588.7$ is greater than Z_{IP}. Finally, at node 20, the feasible integer solution $Z_{IP} = 586$ is found, and this is the optimal solution since there are no more nodes to explore.

8. Computational Testing and Results

Ten test cases were created using actual data from Air Products and Chemicals. Each test case represents a 12-facility network, consisting of 3 plants, 3 fleets, and 6 customer zones. The constraint programs were solved using ILOG Solver and the binary integer programs were solved using CPLEX MIP. We use the test cases to compare the solution times and search tree size for the constraint program and the binary integer program.

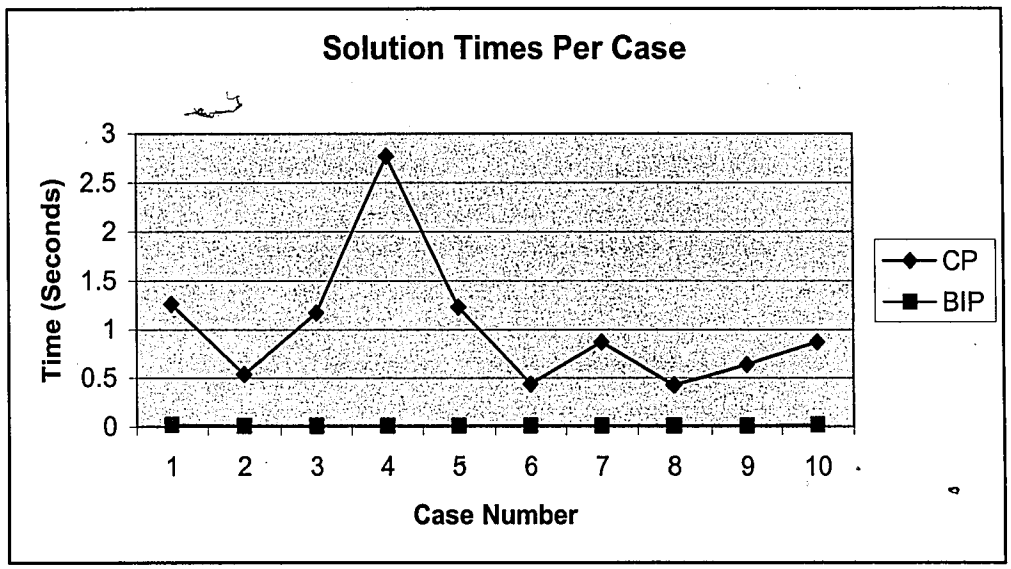

Figure 9: Solution Times for the CP and BIP Models

Figure 9 graphs the solution times per case for both the CP and BIP models. The CP solution times ranged from 0.43 seconds to 2.77 seconds, while the BIP consistently solved in 0.01 or 0.02 seconds for every case. We compare the size of the search trees in Table 1 on the next page.

Table 1: Search Tree Size Per Case

	CP Model				BIP Model	
Case Number	Choice Points	Failures	Feasible Solutions	Total Nodes	Nodes	Iterations
1	1514	1489	26	3029	0	6
2	549	515	35	1099	0	0
3	980	943	38	1961	0	0
4	6942	6930	13	13885	0	0
5	591	505	87	1183	0	0
6	352	346	7	705	0	0
7	1019	996	24	2039	0	0
8	359	339	21	719	0	0
9	374	326	49	749	0	0
10	1675	1659	17	3351	0	0

The size of the CP search tree varies widely over the ten cases, while CPLEX solved the BIP in the root node of the search tree for every instance. Figure 10 shows that the size of the CP search tree and the solution time are directly related.

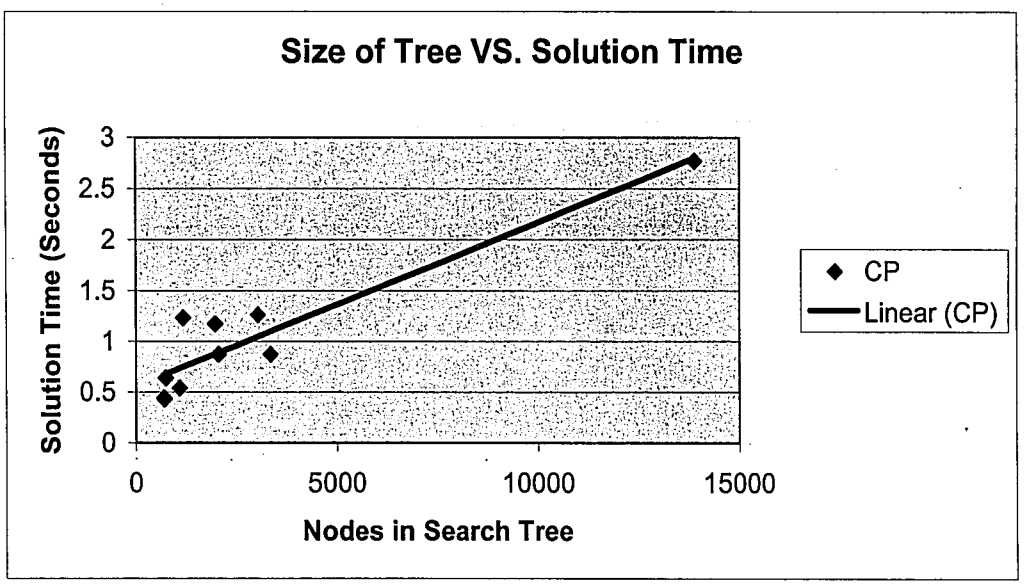

Figure 10: Size of Tree Vs. Solution Time for the CP Model

We examined the data sets that incurred large search trees and high solution times and determined that those cases were loosely constrained in terms of resources. Since the amount of customer demand was low compared to the production, driver, and fleet capacities, there were many possible combinations to explore in the search tree.

More extensive testing was conducted using various branching rules and search procedures on the 10 test cases. For the constraint program, we explored two additional search procedures in addition to the "first-fail" procedure outlined in Section 6.1. The first additional search procedure, which we call "first-feasible," appears in OPL as follows:

```
search{
forall(c in custZones ordered by decreasing demand[c])
    tryall(f in fleets ordered by decreasing drivers[f])
      Shipper[c] = f;
};
```

The "first feasible" search identifies the customer zone *c* with the highest demand, and attempts to assign fleet *f* with the most drivers to Shipper[c]. The idea behind this procedure is that since customer zones with the highest demand will take up the most resources, incur the most trips and possibly the most cost, they should be taken care of first. By ordering the fleets in decreasing order of the number of drivers, we have the best chance of finding a feasible assignment of fleet to customer zone.

The second search procedure is based on a maximal regret heuristic as follows:

```
search {
  forall(f in fleets ordered by decreasing regretdmin(VarFleetCost[f]))
    tryall(c in custZones ordered by increasing fcdrivingTime[f,c])
      Shipper[c] = f;
};
```

The maximal regret heuristic identifies the fleet with the maximal regret, meaning the fleet with the largest difference between the two lowest values of its VarFleetCost domain. Once this fleet is identified we rank the customers in order of the nearest driving time to the longest driving time with respect to the fleet, and pair up the fleet and customer zone that are closest. In essence, we attempt to find the least cost solutions more quickly by pinpointing the fleet with the most to lose if it doesn't get its cheapest assignment, and pairing up fleets and customers that will incur low cost because they are nearby.

In addition to the two new search procedures, we tried a best-first search strategy for exploring the constraint programming search tree. The following figures show the results of using depth first search and best first search for all three search procedures.

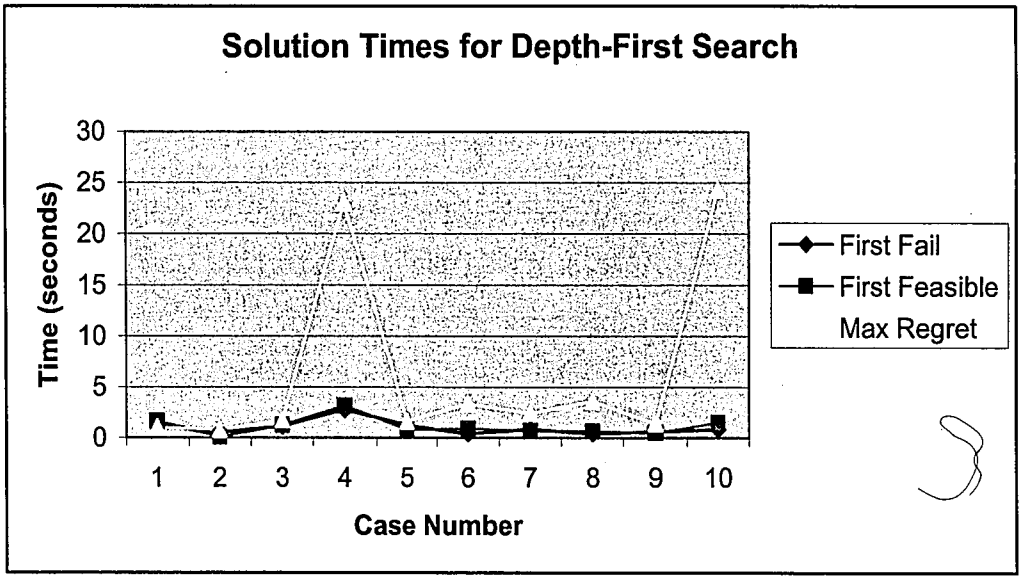

Figure 11: Solution Times Per Case for the Depth First Search

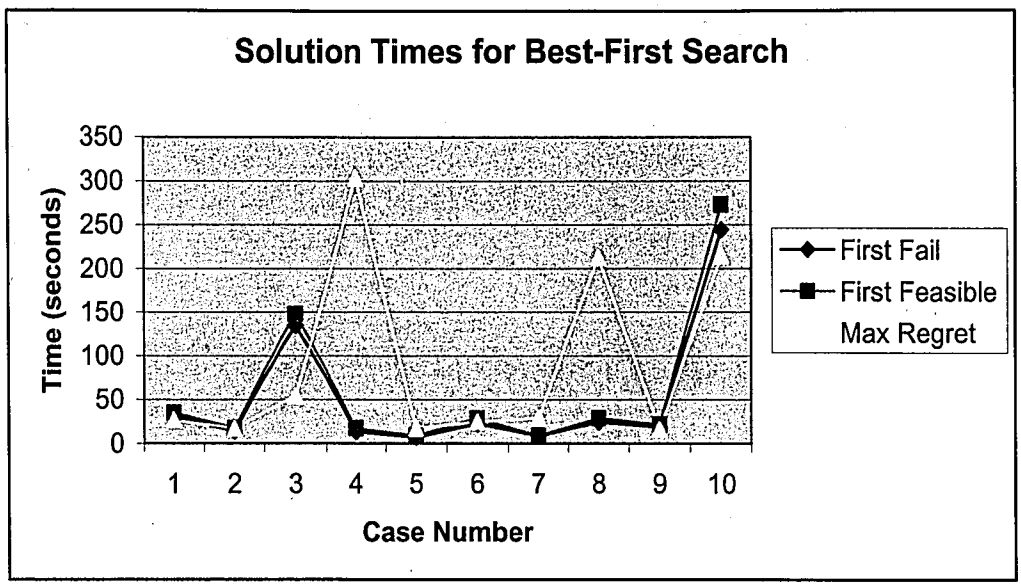

Figure 12: Solution Times Per Case for the Best First Search

Figures 11 and 12 clearly show that depth-first search is the better search strategy for the constraint program. It is not surprising that the best-first search performed poorly, since a good upper bound may not be available at the nodes of the search tree.

Since depth-first proved to be the better strategy, we display the solution times for each case and each search procedure in Table 2 below. The fastest solution time for each case is displayed in bold print.

Table 2: Solution Times for Best-First Search

Case Number	First Fail	First Feasible	Maximal Regret
1	1.26	1.64	**1.1**
2	0.54	**0.06**	0.75
3	**1.17**	1.26	1.75
4	**2.77**	3.13	23
5	1.23	**0.78**	1.61
6	**0.44**	0.92	3.38
7	0.87	**0.68**	2.42
8	**0.43**	0.66	3.59
9	0.64	**0.49**	1.43
10	**0.87**	1.52	24.31

Table 2 shows that the first fail search strategy produced the fastest solutions in 5/10 cases, while the first feasible strategy was also successful by providing the fastest solutions in 4/10 cases. The maximal regret heuristic does not appear to be a good search strategy for the production and distribution planning problem.

We were unable to perform additional branching and search strategy experimentation for the Binary Integer Program, since it solved all of the test cases in the root node. If the problems had been more difficult to solve, one could experiment with several strategies. For example there are several options for node selection, including depth-first search and best-bound search. If depth-first search is used, the branching direction can be specified as either "down branch first" indicating a preference for processing the node corresponding to the rounded down fractional value or "up branch first" to process the node with the fractional value rounded up. For this problem, we would expect "up branch first" to produce faster solutions, because fixing a variable to 1 implies that eight other variables are set to 0, while fixing a variable to 0 implies nothing. However, we would in general expect the best-bound search to outperform depth-first search for the BIP, since it is a "smarter" search procedure that makes use of bounding to determine which nodes are the best candidates.

9. Conclusions

Several conclusions were reached through this research. First, we conclude that the binary integer program presented in this paper is a more promising and reliable alternative than the constraint program for the production and distribution planning problem at Air Products and Chemicals. While the constraint program allows for a more natural statement of the model, the binary integer program is more compact and has faster solution times in every case. In addition, the BIP is a better option from a usability perspective since we do not have to compute variable upper bounds. This contrasts with the constraint program, where a heuristic bounding procedure must be run to determine variable upper bounds every time the data is changed or a new data set is introduced.

The binary integer program presented in this thesis performs well because its linear relaxation aids the branch and bound tree in several aspects. First, the bounds guide the search procedure, which leads to an earlier detection of feasible solutions. Second, information obtained from the LP relaxation allows variable domains to be reduced. The bounds also allow pruning to occur earlier in the search tree. Lastly, the LP relaxation can lead to the discovery of feasible solutions without the domain being reduced to a single point, as in CP.

We also arrive at some conclusions for comparing CP and IP in general. While best-first search is known to be efficient for integer programming, it does not perform well on constraint programming problems due to the lack of a bound at the search tree nodes. Additionally, we find that it is possible to create tailor-made search strategies that are competitive with the well-known "first-fail" strategy.

Future research on this problem should incorporate both constraint programming techniques and integer programming techniques in a hybrid model. The model should integrate the logical constructs of constraint programming as "switches" that enforce constraints from the integer programming model. For example, using the notation from Section 7.1, a constraint in the hybrid model would be:

(Delivery[p,f,c] = 1) => (Driver Resource Constraint[p,f,c])

Constraints of this form provide a natural representation of the relationships, but they also can be relaxed and used in a search tree to obtain bounds, thus leveraging the strengths of both constraint programming and integer programming.

Bibliography

[1] Bartak, R. (1999) Constraint programming: In pursuit of the holy grail. In Proceedings of the Week of Doctoral Students (WDS), Prague, Czech Republic.

[2] Bockmayr, A., Pisaruk, N., and Aggoun, A. (2001) Network flow problems in constraint programming. *Principles and Practice of Constraint Programming – CP 2001*, Springer LNCS **2239**, 196 - 210.

[3] Christodoulou, G., Stamatopoulos, P. Crew Assignment by Constraint Logic Programming. Proceedings of the 2nd Hellenic Conference on Artificial Intelligence SETN-2002 (Companion Volume), pp. 117-127, Thessaloniki, 2002

[4] Darby-Dowman, K., Little, L., Mitra, G., and Zaffalon, M. (1997) Constraint Logic Programming and Integer Programming Approaches and Their Collaboration in Solving an Assignment Scheduling Problem. *Constraints*, **1**(3), 245-265.

[5] Dincbas, M., Simonis, H., and van Hentenryck, P. Solving a Cutting-Stock problem in constraint logic programming. In R. Kowalski and K. Brown, editors, *Logic Programming*, pages 42-58. 1988.

[6] Hooker, J.N. (2000) *Logic-Based Methods for Optimization: Combining Optimization and Constraint Satisfaction.* John Wiley & Sons, New York.

[7] Hooker, J.N. (2001) Logic, Optimization, and Constraint Programming.

[8] Kim, H.-J. and Hooker, J. N. (2002) Solving fixed-charge network flow problems with a hybrid optimization and constraint programming approach. *Annals of Operations Research,* **115,** 95-124.

[9] Lustig, I.J. and Puget, J.F. (2001) Program Does Not Equal Program: Constraint Programming and Its Relationship to Mathematical Programming. *Interfaces,* **31**(6), 29-53.

[10] Land, A., Doig, A. (1960) An automatic method for solving discrete programming problems. *Econometrica,* **28**(3), 497-520.

[11] Marriot, K., Stuckey, P.J. (1998) *Programming With Constraints.* MIT Press.

[12] Nemhauser, G.I., Wolsey, L.A. (1988) *Integer and Combinatorial Optimization.* John Wiley & Sons, New York.

[13] Ottosson, G., Thorsteinsson, E.S., and Hooker, J.N. (2002) Mixed Global Constraints and Inference in Hybrid CLP-IP Solvers. *Annals of Mathematics and Artificial Intelligence*, **34**.

[14] Puget, J.F. and Lustig, I.J. (2001). "Constraint programming and maths programming," *The Knowledge Engineering Review,* **16**(1), 5-23.

[15] Smith, B.M., Brailsford, S.C., Hubbard, P.M., and Williams, H.P. (1996) The Progressive Party Problem: Integer Linear Programming and Constraint Programming Compared. *Constraints,* **1,** 119-138.

[16] Timpe, C. (2002) Solving planning and scheduling problems with combined integer and constraint programming. *OR Spectrum*, **24**(4), 431-448.

[17] Van Hentenryck, P. (1999) The OPL Optimization Programming Language. *MIT Press*, Cambridge, Massachusetts.

[18] Van Hentenryck, P., Perron, L., and Puget J.-F. (2000) Search and Strategies in OPL. *ACM Transactions on Computational Logic*, **1**(2), 285-320.

[19] Wallace, M. (1998) Constraint Programming. In: Liebowitz, J. (ed.), The Handbook of Applied Expert Systems, CRC Press, 1998.

Vita

Shelley Heist, daughter of Margaret and John Heist, was born and raised in the Lehigh Valley of Pennsylvania. She attended Northampton Area High School and went on to the Pennsylvania State University, graduating in May 2000 with a Bachelor of Science degree in Industrial Engineering. Shelley was then accepted at Lehigh University to pursue graduate studies. She will use her Masters degree in Industrial and Systems Engineering in her career as an Analyst for marketRx, Inc. of Bridgewater, NJ beginning in June 2003. Shelley is engaged to marry Mark Sherman on August 9 of this year.

END OF TITLE