Modeling and Algorithmic Development of a Staff Scheduling Problem

by

Sanjay Aggarwal

Thesis submitted to the Faculty of the

Virginia Polytechnic Institute and State University

in partial fulfillment of the requirements for the degree of

Master of Science

in

Industrial and Systems Engineering

APPROVED

Subhash C. Sarin, Chairman

C. Patrick Koelling

John Kobza

September, 1995

Blacksburg, Virginia

Modeling and Algorithmic Development of a Staff Scheduling Problem

by

Sanjay Aggarwal

Dr. Subhash C. Sarin, Chairman

Industrial and Systems Engineering

(ABSTRACT)

Scheduling workers in a trucking system for stripping and loading trucks is a difficult and time consuming task that involves determining the optimal number of workers. Once the number has been determined the operations manager has to assign different trucks to the workers and has to determine the schedule of the workers. In the thesis, we develop a mathematical model to solve the scheduling problem, with attention focused on minimizing the number of workers required on a particular shift and on finding the feasible assignments of workers to trucks and vice-versa. We show that this is a set partitioning problem with an additional feasibility constraint which can be solved using the column generation technique. An inherent characteristic of this problem that makes it different from other set partitioning problems is that the arrival time and scheduled departure time is different for different trucks. We illustrate the model and the proposed algorithmic approach by generating a schedule based on real data obtained from a trucking company.

Acknowledgments

I would like to thank my academic advisor Prof. S. C. Sarin for his invaluable help, encouragement and support. I am also very grateful to Prof. C. Patrick Koelling and Prof. John E. Kobza for their help and advice.

For their support, as well as understanding, I would like to thank my present and former co-workers Ric Kosiba, Ross Darrow, Richard Wade, James Hengst, and Jayendra Patel.

Most of all I thank my parents Mr. Shyam Aggarwal and Mrs. Kusum Aggarwal, and sister Pooja for their support and love. Also, without encouragement and help from my wife Preety, this work would not have been possible.

Table of Contents

Chapter 1
Introduction

1.1 Problem Description

1.11 Problem Statement

This thesis is concerned with optimally scheduling workers for stripping and loading at a terminal in a trucking system by using optimization techniques and the power of modern day computers. In effect, a scheduling tool for the above problem is presented that can be used by any trucking company where scheduling workers is a time consuming and challenging task. This terminal is like a hub where trucks from different places arrive at night. The items on these trucks are then stripped and loaded onto the trucks going to intended destinations. The stripping and loading of the trucks is done manually and the problem is to determine the optimal number of workers required to perform the desired tasks.

The trucks as they arrive are assigned to the predetermined docks at the terminal. A worker is assigned to each truck to strip it and load the items in the trucks bound for designated destinations. The trucking system is comprised of two sub processes, namely, inbound and outbound. The inbound process covers how freight moves from terminal to terminal while the outbound process covers how freight is picked up and delivered between a terminal and the customer. The staffing question pertains to both the inbound and outbound processes as both processes are maintained by the dock workers. A key information required to address this problem is the stripping and loading rates of the trucks. It is assumed that both the stripping and loading times are known with certainty.

1.12 Problem Background

This problem can be viewed as a variation of the airline crew scheduling problem. In the airline crew scheduling problem, once the flight timetable has been agreed upon, an airline is faced with manning each flight with a full complement of cabin and flight deck personnel in a manner consistent with:

1. Safety regulations
2. Union requirements
3. Company policy
4. Sound economics

The basic planning unit is the *rotation*, which is a trip flown by a crew and which is legal with respect to conditions 1, 2, and 3. A rotation, as its name implies, is usually a round

trip that takes a crew from its home location or *base* and returns it there at the end of the journey. Such a rotation may or may not contain statutory rest periods. The airline crew scheduling problem is to select a set of rotations in such a way that each flight segment or *leg* is covered at least once and the total cost is minimized. For the problem at hand, the given number of trucks have to be covered with a minimum number of workers. Since each truck is assigned to only one worker, the resulting problem is one of Boolean or (0,1) programming.

An important aspect of scheduling is to determine and define the availability of the workers. In trucking industry, for stripping and loading operations, there are two types of employees. Some of them are full time employees and the others are part time or wage employees. Depending upon the number of trucks arriving on a certain day, wage employees, if needed, are called at a relatively short notice to report to work. This undefined and flexible workforce makes the scheduling problem all the more important.

1.2 The Objective

For the problem stated above, our objective is to develop an algorithm to effectively determine the minimum number of workers required at a terminal to perform the stripping and loading operations during a shift. The trucks at a facility arrive at different times and some may still come in after the shift work has started. So it is essential that the dynamic availability of the trucks is taken into consideration at the time of algorithmic development. We will refer to this requirement as the 'feasibility requirement."

A computer implementation of the algorithm will be developed and it will be tested on the real world data provided by a trucking company.

1.3 Example of a Typical Scheduling Problem

Presented in this section is an example situation of the problem considered in this thesis. Usually, the shift starts at 12:00 midnight when the trucks begin to arrive. All the trucks have to leave by 8:00 A.M. It is assumed that no trucks arrive after the time when they can not be stripped and loaded by 8:00 A.M. Table 1.0 depicts one such arrival pattern of the trucks at a terminal. The manager at a particular hub generally has the information about the expected arrival time of each truck and its expected unloading and loading time. The operating time in Table 1.0 is the unloading and loading time for each truck. The problem then is to determine the minimum number of workers required to process these trucks.

1.4 Organization of the Thesis

In the sequel, Chapter 2 briefly reviews the work done in the area of the crew scheduling problem and various algorithms developed so far. In Chapter 3, we present the model and the algorithm developed for the staff scheduling problem in the trucking industry. Results obtained and an analysis of the results has been presented in Chapter 4. Concluding remarks and directions for future research have also been included in this chapter.

Table 1.0
An example of the truck Arrival and Operating times at a terminal

Truck #	*Expected Arrival Time*	*Operating Time*
1.	6.35	1.10
2.	5.50	0.40
3.	4.40	0.77
4.	3.15	0.40
5.	3.20	0.50
6.	2.05	0.66
7.	1.55	0.43
8.	1.20	0.90
9.	0.40	1.70
10.	1.10	1.88
11.	2.00	2.10
12.	6.40	1.00
13.	0.50	0.70
14.	1.00	1.50
15.	1.20	1.60
16.	1.30	2.00
17.	3.00	1.40
18.	3.30	0.90
19.	4.15	1.40
20.	0.55	1.00

Chapter 2
Literature Review

The scheduling of workers for stripping and loading operation in the trucking industry is a set partitioning problem where the trucks are to be divided into mutually exclusive sets to be worked on by different workers. Furthermore, each truck can be worked upon by only one worker, whereas, each worker can work on several trucks during the shift. The factor that makes this problem more complex is the variable arrival times of the trucks, which, makes it difficult to find a feasible assignment.

Let there be m trucks to be stripped during a shift. Let $\underline{a}_j$ be a vector of size m and be defined such that $a_{ij} = 1$ if truck i is assigned to worker j, and $a_{ij} = 0$, otherwise. Thus, $\underline{a}_j$ represents an assignment of some trucks to worker j. Let $y_j = 1$ if worker j is working during the shift and $y_j = 0$ otherwise. The staffing problem can thus be modeled as follows. Determine y_j so as to

$$\text{minimize} \sum_{j=1}^{N} y_j$$

$$\textit{subject to} \sum_{j=1}^{N} \underline{a}_j \, y_j = \underline{e}$$

$$y_j = 0 \textit{ or } 1 \quad \forall \quad j = 1, \ldots\ldots, N$$

Here N represents all possible assignments of a worker to perform the stripping and loading operations during a shift and $\underline{e}$ is a vector of 1's. The constraints in (1) represent the fact that each truck is served by only one worker. The trucks will be assigned in such a fashion that the total number of workers is minimized which is the objective function.

A key feature of this formulation is the use of vectors $\underline{a}_j$. The process of generating $\underline{a}_j$ is termed matrix generation. We use the knapsack algorithm to generate the columns as discussed in Chapter 3. This is the key advantage of this formulation.

The set partitioning problem has been studied in great detail since the mid sixties. Most of the work has been done for the airline crew scheduling problem, which is a set partitioning problem. Different set partitioning algorithms are reviewed to study the methodologies developed so far (see Section 2.1).

Another approach to solve the problem on hand could be the bin packing problem as discussed in section 2.2. Different methodologies used for the bin packing problem and their limitations are presented in this section.

Also, the algorithms available for knapsack problems were studied to choose the right algorithm for the column generation procedure.

Thus, the literature review can be divided and presented into the following three categories:

1. Set Partitioning problem
2. Bin-packing problem
3. Knapsack problem

2.1 Set Partitioning Problem

The set partitioning problem is a special integer linear program. This problem has a binary coefficient matrix, binary variables, and unit resources. Furthermore, all of its constraints are equations. In its standard form, the set partitioning problem is

$$\min \{ cx | Ax = e, x_j = 0 \text{ or } 1, \forall j \in N \}$$

where A is an m x n matrix of zeros and ones, c is an arbitrary n-vector, $e = (1, ..., 1)$ is an m-vector, and $N = \{1, ..., n\}$. Its name comes from the following interpretation: if the rows of A are associated with the elements of the set $M = \{1, ..., m\}$ and each column a_j of A with the subset M_j of those $i \in M$ such that $a_{ij} = 1$, then the set partitioning problem

is the problem of finding a minimum-weight family of subsets M_j, $j \in N$, which is a partition of M, each subset M_j being weighted by c_j.

In spite of its very special form, the set partitioning problem has many practical interpretations. A partial list of applications described in the literature includes: airline fleet scheduling [Lewin (1969)], truck routing [Balinski and Quandt (1964), Clarke and Wright (1964), Dantzig (1959), Garfinkel and Nemhauser (1966), Pierce (1968)], airline crew scheduling [Arabeyere et. al (1969), Kolner (1966), Spitzer (1961), Thiriez (1969)], information retrieval [Day (1965)], switching circuit design [Balinski (1965), Cobhan (1962), Paul and Unger (1959), Root (1964), stock cutting [Pierce (1970)], assembly line balancing [Salveson (1959)], capital equipment decisions [Valenta (1969)], location of offshore drilling platforms [Daly and Spierer (1969)], facilities location problems [Revelle et. al (1970)], political districting [Garfinkel (1970), Wagner (1968)], coloring problems [Busacker (1965)], symbolic logic [Cobhan et. al (1961)], and PERT-CPM [Cobhan (1961)].

The set partitioning problem has two seemingly close variations, namely, the set packing problem

$$\max \{ c'x \mid Ax \leq e, x_j = 0 \text{ or } 1, \ \forall j \in N \}$$

and the set covering problem

$$\min \{ c''x \mid Ax \geq e, x_j = 0 \text{ or } 1, \ \forall j \in N \}$$

where A, e and N are as defined in the set partitioning problem, while c' and c'' are arbitrary n-vectors.

The solution procedure for the set partitioning formulation of the airline crew scheduling problem is a three stage process: enumeration, reduction, and selection. In the enumeration stage, all feasible pairings, the columns of the constraint matrix A, are generated. This is accomplished by a straight forward enumeration procedure. Due to the combinatorial nature of the problem, however, one finds that a problem of 1,000 flight legs may generate several million feasible pairings. As a result, the reduction stage of the solution procedure is applied within the enumeration process in an attempt to reduce the problem size and the total time required to obtain a solution. The reduction procedures may be either objective or subjective. An objective reduction procedure, as in Balinski (1965) or Garfinkel and Nemhauser (1972), use both dominance and logical comparison to reduce the number of pairings. Subjective reduction may include limits on the maximum length of the pairing or elimination of all pairings with layovers at an undesirable station. These procedures can be very effective in reducing the size of the problem. Kolner (1966) reports that after application of the reduction procedures, a problem with 100-120 flight legs may generate only 2,000-3,000 pairings, as opposed to original hundred thousand.

The final stage of the set partitioning solution procedure is the selection of those pairings (columns) which satisfy all the constraints at a minimum cost. The different algorithms presented in the literature for set partitioning problem can be classified as follows:

1. Heuristic solutions
2. Implicit enumeration
3. Simplex-based cutting plane methods
4. A column generating algorithm
5. A hybrid primal cutting plane/implicit enumeration algorithm
6. A symmetric subgradient cutting plane method
7. Set partitioning via node covering
8. Formulation as a Network-Flow-Problem

Next, we briefly discuss these algorithms.

2.1.1 Heuristic solutions

Cassidy and Bennett (1975), Christofides (1974), Golden (1975), and others have demonstrated the effectiveness of heuristic techniques in solving large, complex routing and scheduling problems. Baker *et al.* (1979) demonstrated that the application of heuristic procedures to the airline crew scheduling problem can provide near-optimal solutions in a reasonable amount of computer time. The solution procedure developed requires three steps: pairing construction, pairing improvement, and composite procedures. The pairing construction procedures use the flight leg data generated by the aircraft routing as input and construct a feasible set of crew pairings. The pairing improvement procedure uses a feasible set of crew pairings as input and attempt to generate an improved set of pairings. The construction of an initial feasible solution and

the application of one or more improvement procedures is defined as a composite procedure.

Rubin (1973) presented a heuristic optimization technique to solve the airline crew scheduling problem. The proposed algorithm uses a set covering algorithm repeatedly on much smaller matrices extracted from the overall problem, generating columns as needed. This approach gave excellent results on test cases involving close to 1,000 rows. It utilizes some techniques that are more generally applicable, and some that make use of the structure of the crew scheduling problem, and that therefore are specific to it.

2.1.2. Implicit enumeration

In general, the solution space of an integer program can be assumed to possess a finite number of possible feasible points. A straightforward method for solving the integer problem is to *exhaustively* (or explicitly) enumerate all such points. In this case, the optimal solution is determined by the point(s) that yield the best (maximum or minimum) value of the objective function.

The obvious drawback of the above technique is that the number of solution points may become impractically too large, with the result that the solution can not be determined in a reasonable amount of time. The idea of *implicit* (or Partial) enumeration calls for considering only a portion (hopefully small) of all possible solution points while automatically discarding the remaining ones.

The first potentially successful work in the area of implicit enumeration was reported by Balas (1965) for solving the zero-one linear problem. Important modifications in Balas' algorithm were later given by Glover (1965c), whose work was the basis for other developments. The implicit enumeration schemes that seem to have been successful include those by Pierce (1968), Garfinkel and Nemhauser (1969), Pierce and Lasky (1973), and Marsten (1974).

In the Garfinkel and Nemhauser version, the solution space is systematically searched by generating partial solutions (assigning 0-1 values to variables taken one at a time) and exploring the logical implications of these value assignments. To start with, the matrix A is brought by row and column permutations to the staircase form as shown in figure 2.1. Within each block the columns are ordered by some heuristic criterion: according to increased costs. Then the algorithm proceeds to select one column from each block so that each row is covered at least once.

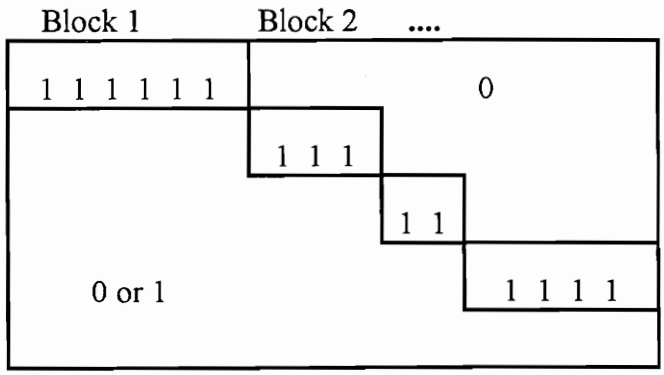

Figure 2.1

A matrix in the Staircase form

Another important improvement was suggested by Pierce and Lasky by replacing the cost vector c with the reduced cost vector of an optimal simplex tableau for the linear set-partitioning problem (LSPP). This requires of course the solution of (LSPP) prior to implicit enumeration, which for large problems is quite a high price to pay for hopefully speeding up the subsequent enumeration. Another implicit enumeration algorithm based on a different enumerative scheme was developed by Marsten. Here, the columns of A are partitioned into blocks, as in the above procedure, but the enumeration is based on assigning rows to the blocks. This enumerative scheme gives rise to a different search tree, whose number of nodes often has a smaller bound than the number of nodes in a usual search tree. Marsten also solves (LSPP) first and uses the reduced costs instead of the original ones. Marsten's code seems to perform better than the Garfinkel-Nemhauser code on low density problems, while the latter has the lead for higher densities. Unfortunately, however, it is not clear to what extent these differences can be ascribed to the different search trees used, since the two codes differ in other aspects as well.

One important fact that has emerged from Marten's experience is the difficulty of solving the linear programs associated with large set covering problems.

2.1.3. Simplex based cutting plane methods

The cutting plane methods seek a restructuring of the feasible solution space by imposing some (specially designed) constraints on the original space such that the required optimum feasible point continue to remain a proper extreme point of the modified solution space. The general idea is that these additional constraints cut off portions of the solution space such that no integer feasible points are ever excluded. The cutting plane methods we

discuss below have three characteristics in common: they are all based on the simplex method (primal and dual, or only dual), they all use the traditional cutting planes introduced by Gomory (1963a), and they are all general-purpose algorithms which take little if any advantage of problem structure. The fact that, in spite of this last feature, they perform reasonably well on set partitioning problems (SPP), shows that cutting plane approach holds great promise for this class of problems. One possible direction of improvement seems to be at hand in the form of cutting planes derived from the special structure of (SPP). These cuts attempt to substantially reduce the number of cuts needed.

The first finite cutting plane algorithm was developed by Gomory (1958), for the pure integer problem. Gomory showed how these cuts can be constructed systematically from the simplex tableau. Although the algorithm is proved to converge in a finite number of iteration, it has the drawback that the machine round-off error represents a prominent difficulty. This led to the development of a new algorithm by Gomory (1960a), which improves directly on this drawback. Later, Gomory (1960b) extended his theory to cover the mixed integer problem. Another simplex based cutting plane method that was tested on (SPP) is Gomory's (1963b) all integer algorithm. Boyd (1994) presented a technique for generating cutting planes for integer programs that is based on the ability to optimize a linear function on a polyhedron rather than explicit knowledge of the underlying polyhedral structure of the integer program.

Two possible directions of improvement seem to be open: (i) specialization of the simplex method which uses the structure of (LSPP), and (ii) use of non-simplex type methods for solving (LSPP).

2.1.4. A column generating algorithm

Next, we discuss a column generating primal simplex algorithm by Balas and Padberg (1975) for (SPP), which produces a sequence of integer solutions that converges to the optimal one. The algorithm is based on the characterization of adjacency relations among vertices of $\overline{P}_I$. The definition of $\overline{P}_I$ is as follows:

Let $\overline{P}$ be the feasible set of the linear program associated with (SPP), i.e.,

$$\overline{P} = \{x \in R^n \mid Ax = e, x \geq 0\},$$

and let

$$\overline{P}_I = conv\{x \in \overline{P} \mid x \text{ integer}\}$$

The algorithm performs non degenerate primal simplex pivots on +1 entries as long as this is possible. When this cannot be continued, degenerate pivots are performed on positive or negative entries, as long as they decrease total dual feasibility. When neither type of pivoting can be continued, a column generating procedure is used to produce a composite column defining an edge of $\overline{P}_I$ which connects the current vertex of $\overline{P}_I$ to a better one, or to establish the absence of any better vertex. The idea of allowing for degenerate pivots on -1 entries is due to Andrew, Hoffman and Krabek (1968), who implemented it and obtained remarkably good computational results in the sense of being able to get to, or close to, the integer optimum fairly often. According to Balas and Padberg (1975), it pays to go further and allow for degenerate pivots on any non zero

entry (i.e., to give up the integrality of the tableau, though not of the solution) as long as improvements can be obtained according to some reasonable criterion.

2.1.5. Hybrid primal cutting plane/implicit enumeration algorithm

Balas (1975c) proposed a hybrid algorithm, which combines a primal cutting plane approach with implicit enumeration applied to sub problems so defined as to generate an improvement at each iteration.

The algorithm starts with the linear programming relaxation of the set partitioning problem. It performs simplex pivots which leave the solution primal feasible and integer, and either reduce the objective function value, or leave the latter unchanged and reduce the absolute value of sum of negative reduced costs (this does not exclude pivots on negative entries, or pivots that make the tableau fractional, provided they occur in degenerate rows). If needed, implicit enumeration is used to generate the cutting plane.

Every iteration of this algorithm either decreases the objective function value z, or leaves z unchanged and decreases the absolute value σ of the sum of negative reduced costs. Since both z and σ are bounded from below, the procedure is finite.

2.1.6. A symmetric subgradient cutting plane method

As mentioned in the cutting plane algorithms, in solving large set partitioning problems by cutting plane techniques, the main bottleneck is handling the linear program

(LSPP). This, and the special structure of (LSPP), suggests the idea of trying to solve the latter by other methods than the (primal or dual) simplex algorithm. One such attempt by Balas and Samuelsson (1974b) uses some of the cutting planes, while solving the linear program by a subgradient of the general type used by Held and Karp (1971) for the traveling salesman problem and discussed by Held, Wolf and Crowder (1974). The subgradient approach presented differs from that of the above mentioned authors in that it works simultaneously with the primal and the dual which makes it possible to achieve faster convergence.

Among the positive features of this approach are the high degree of numerical stability, low memory requirements and an ability to use data in compactly stored form. Another main advantage of the symmetric procedure over the asymmetric subgradient method discussed in Held, Wolfe and Crowder (1974) lies in the fact that the latter requires an estimate of the optimal objective function value for a proper choice of the step length. Also, the convergence is very fast when the estimate is replaced by the actual value of the optimum. In the symmetric procedure such an estimate is not needed, since the function one minimizes is known to have 0 as its optimal value.

2.1.7. Set partitioning via node covering

The algorithm for set partitioning via node covering was presented by Balas and Samuelsson (1973), (1974a). They suggest that one way of solving set packing (SP) and set partitioning problems is to solve the associated node packing problem. The node packing problem is defined as follows:

Let a_j denote the jth column of the matrix A of (SP). The intersection graph G_A = (N, E) of A has one node for every column of A, and one edge for every pair of non orthogonal columns of A [i.e., $(i, j) \in E$ if and only if $a_i a_j \geq 1$]. Let A_G be the node-edge incidence matrix of G_A, and denote by (NP) the weighted node packing problem whose weights c_j are the same for each node as those of (SP), i.e.,

$$\text{(NP)} \qquad \max \{ cx \mid A_G^T x \leq e_q, x_j = 0 \text{ or } 1, j = 1,\ldots\ldots,n\}$$

x is a feasible (optimal) solution to (SP) if and only of it is a feasible (optimal) solution to (NP).

The weighted node packing problem is equivalent to the weighted node covering problem, since the complement of any node packing is node covering; and if the node packing is of maximum weight, its complement must certainly be of minimum weight. The authors present the procedure for the unweighted node covering problem and its extension to the weighted case.

The unweighted node covering problem can be stated as

$$\text{(NC)} \qquad \min \{ e_n x \mid A^T x \geq e_q, x_j = 0 \text{ or } 1, j = 1,\ldots\ldots,n\}$$

The given procedure terminates in a polynomially bounded number of steps, after which, if the terminating solution is not optimal, the problem is partitioned.

2.1.8 Formulation as a Network-flow problem

Whenever one regards crews as commodities flowing through a network, the formulation of the crew scheduling problem as a network-flow model seems rather straightforward. Utilizing the very efficient out-of-kilter algorithm to determine a minimal cost flow in a network (Fulkerson (1960)), several network flow models have been studied.

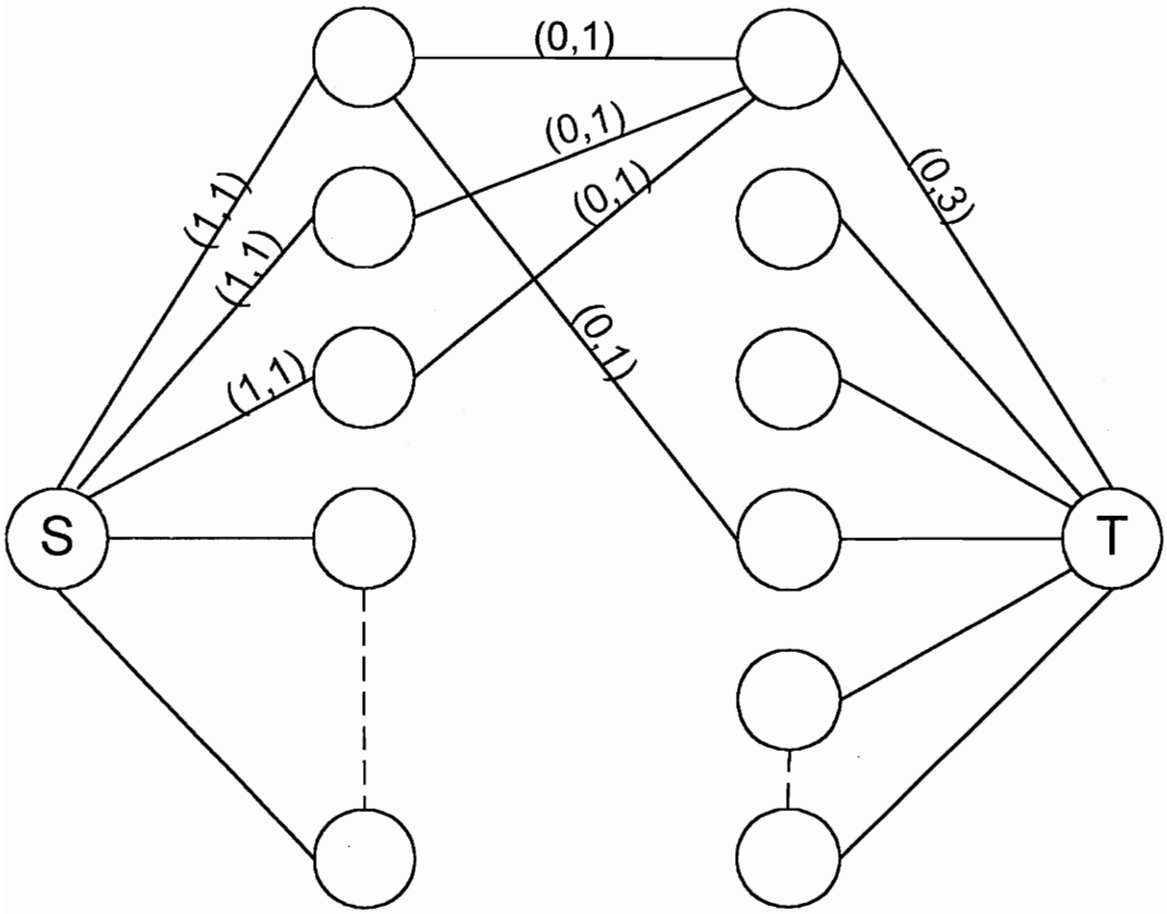

Figure 2.2
A possible network flow formulation for the crew scheduling problem

A formulation of the problem as a bipartite network with the added source S and sink T may be visualized as shown in figure above. The first column of nodes represent the smallest planning unit (normally flight legs); the second column of nodes represent daily rotations that are chain combinations of the first columns of nodes. Arcs from the source S to the first column of nodes must carry a flow of value 1; arcs leading from the first to the second column of nodes may have either z zero- or a one-value flow. However, arcs leading from the second column of nodes to the sink T must have a flow of either zero- or m-value, where m indicates the incoming number of arcs at the bottom node. This restriction of the flow value to the two extreme values prevents one from applying the out-of-kilter method, where flow of any integer value in between the two extreme may be produced.

In spite of these failures for an accurate formulation of the crew scheduling problem as a network flow model, Moreland (1966) shows how one can achieve good solutions by introducing some manual interpretive features into the model formulation. The solution may be of great interest whenever one tries to get a quick first answer to the crew scheduling problem.

2.2 Bin-packing problem

The bin packing problem can be described using the terminology of knapsack problems, as follows. Given n items and n knapsacks (or bins), with

w_j = weight of item j

c = capacity of each bin

assign each item to one bin so that the total weight of the items in each bin does not exceed c and the number of bins used is minimum. A possible mathematical formulation of the problem is

$$\text{minimize} \quad z = \sum_{i=1}^{n} y_i$$

$$\textit{subject to} \quad \sum_{j=1}^{n} w_j x_{ij} \leq c y_i , \quad i \in N = \{1,\ldots,n\},$$

where $y_i = 1$ if bin i is used; 0 otherwise,

$x_{ij} = 1$ if item j is assigned to the ith bin; 0 otherwise.

It is assumed that the weights w_j are positive integers. Hence, without loss of generality, it is also assumed that

c is a positive integer,

$w_j \leq c$ for $j \in N$

For the sake of simplicity it is also assumed that, in any feasible solution, the lowest indexed bins are used, i.e., $y_i > y_{i+1}$ for $i = 1, \ldots, n - 1$.

Almost the totality of the literature on bin packing is concerned with approximate algorithms and their performance. A brief outline of the well known approximate algorithms is presented below. Work done in the area of exact algorithms for bin-packing problems is also included in section 2.2.2

2.2.1 A brief outline of approximate algorithms

Next-Fit (NF) is the simplest approximate approach to the bin packing problem. To begin with the items are arranged in the non increasing order of their profit to weight ratios. The first item is then assigned to bin 1. Items 2,....., n are then considered by

increasing indices: each item is assigned to the current bin, if its weight is less than or equal to the remaining capacity of the bin; otherwise, it is assigned to a new bin, which becomes the current one.

First-Fit(FF) algorithm considers the items according to increasing indices and assigns each item to the lowest indexed initialized bin into which it fits; only when the current item cannot fit into any initialized bin, is a new bin introduced. This is the algorithm that best represents the problem on hand.

It has been proven in Johnson *et al.* (1974) that

$$FF(I) \leq \frac{17}{10} z(I) + 2$$

for all instances I of bin packing problem, and that there exist instances I, with $z(I)$ arbitrarily large, for which

$$FF(I) > \frac{17}{10} z(I) - 8$$

where $z(I)$ denotes the optimal solution value for any instance I

$FF(I)$ is the solution obtained through First-Fit algorithm

Another algorithm *Best-Fit* (BF), is a variation of the FF algorithm. Instead of assigning the next item to the lowest indexed bin, this algorithm assigns it to the bin with smallest residual capacity (breaking ties in favor of the lowest indexed bin). It has been proved that BF satisfies the same worst-case bounds as that of FF.

2.2.2 Exact algorithms

As already mentioned, very little can be found in the literature on the exact solution of the bin-packing problem. The computational results reported indicate that these algorithms can solve only small size instances.

Eilon and Christofides (1971) have presented a simple depth-first enumerative algorithm based on the following "best-fit decreasing" branching strategy. At any decision node, assuming that b bins have been initialized, let $(\bar{c}_{i_1},\ldots\ldots,c_{i_b})$ denote their current residual capacities sorted by increasing value, and $(\bar{c}_{i_{b+1}} \equiv c_{b+1} = c)$ the capacity of the next (not yet initialized) bin: the branching phase assigns the free item j^* of the largest weight, in turn, to bins i_s, ..., i_b, i_{b+1}, where $s = \min\{h: 1 \le h \le b+1, \bar{c}_{i_h} + w_{j^*} \le c\}$. Lower bound L_1 given in the paper is used to fathom decision nodes.

Hung and Brown (1978) have presented a branch-and-bound algorithm for a generalization of the bin packing problem to the case in which the bins are allowed to have different capacities. Their branching strategy is based on a characterization of equivalent assignments, which reduce the number of explored decision nodes. Martello and Toth (1989) have proposed an algorithm, MTP, based on a "first-fit decreasing" branching strategy. The items are initially sorted according to the decreasing weights. The algorithm indexes the bins according to the order in which they are initialized. At each decision node, the first (i.e. largest) free item is assigned, in turn, to the feasible initialized bin (by increasing index) and to a new bin. Then a *forward step* and *backtracking step* are applied recursively till the optimal is found. MTP could solve problems of size up to $n = 1000$ quite efficiently.

2.3 Knapsack Problem

In the present context, the knapsack algorithm is used to generate the columns of A matrix for the set partitioning problem. In its general form, the knapsack problem (KP) is: given a set of items and a knapsack with

p_j = profit of item j

w_j = weight of item j

M = capacity of the knapsack

select a subset of items so as to

$$\text{maximize } z = \sum_{j=1}^{m} p_j x_j$$

$$\text{subject to } W = \sum w_j x_j \leq M$$

$$\text{where } x_j = 1 \text{ if item } j \text{ is selected; } 0 \text{ otherwise}$$

It is assumed, without loss of generality, that:

(a) p_j, w_j positive integers $(j = 1, \ldots, m)$

(b) $M < \sum_{j=1}^{m} w_j$;

(c) $w_j \leq M$ $(j = 1, \ldots, m)$

The knapsack problem is one of the most intensively studied discrete programming problems. The reason for such interest derives from three facts: (a) it can be viewed as the simplest *Integer Linear Programming* problem; (b) it appears as a sub problem in many more complex problems; (c) it may represent a great many practical situations. Recently, it has been used for generating minimal cover induced constraints (Crowder, Johnson and Padberg (1983)) and in several coefficient reduction procedures for strengthening LP bounds in general integer programming (Dietrich and Escudero (1989a, 1989b)). During

the last few decades, KP has been studied through different approaches, according to the theoretical development of *Combinatorial Optimization*.

In the fifties, Bellman's (1954) *dynamic programming* theory produced the first algorithms to exactly solve the 0-1 knapsack problem. Dantzig (1957) gave an elegant and efficient method to determine the solution to the continuous relaxation of the problem, and hence an *upper bound* on z which was used in the following twenty years in almost all studies on knapsack problems (KP).

In the sixties, the dynamic programming approach to the KP and other knapsack type problems was deeply investigated by Gilmore and Gomory. Kolesar (1967) experimented with the first *branch-and-bound* algorithm for the problem. His algorithm consists of a *highest-first* binary branching scheme which: (a) at each node, selects the not-yet-fixed item j having the maximum profit per unit weight, and generates two descendent nodes by fixing x_j, respectively, to 1 and 0; (b) continues the search from the feasible node for which the value of upper bound is a maximum. The large computer memory and the time requirements of the Kolesar algorithm were greatly reduced by the Greenberg and Hegerich (1970) approach, differing in two main respects: (a) at each node, the continuous relaxation of the induced sub problem is solved and the corresponding critical item $\bar{s}$ is selected to generate the two descendent nodes (by imposing $x_{\bar{s}} = 0$ *and* $x_{\bar{s}} = 1$); (b) the search continues from the node associated with the exclusion of item $\bar{s}$ (condition $x_{\bar{s}} = 0$). When the continuous relaxation has all integer solution, the search is resumed from the last node generated by imposing $x_{\bar{s}} = 1$, i.e. the algorithm is of *depth-first* type.

In the seventies, the branch and bound approach was further developed, proving to be the only method capable of solving problems with a high number of variables. The most well known algorithm of this period is due to Horowitz and Sahni (1974). Ingargiola and Korsh (1973) presented the first *reduction procedure*, a preprocessing algorithm which significantly reduces the number of variables. Johnson (1974) gave the *first polynomial-time approximation scheme* for the subset-sum problem; the result was extended by Sahni to the 0-1 knapsack problem. The first *fully polynomial-time approximation scheme* was obtained by Ibarra and Kim (1975). Martello and Toth (1977) proposed the first upper bound dominating the value of the continuous relaxation.

The main results of the eighties concerned the solution of large-size problems, for which sorting of the variables (required by all the most effective algorithms) takes a very high percentage of the running time. Balas and Zemel (1980) presented a new approach to solve the problem by sorting, in many cases, only a small subset of the variables (the *core problem*).

Branch and bound algorithms are nowadays the most common way to effectively find the optimal value of knapsack problems. More recent techniques by Balas and Zemel (1980), Fayard and Plateau (1982), and Martello and Toth (1988) imbed the branch-and-bound process into a particular algorithmic framework to solve, with increased efficiency, large instances of the problem.

The other fundamental approach to KP is dynamic programming. This has been the first technique available for exactly solving the problem and, although its importance has

decreased in favor of branch-and-bound, it is still interesting because of the following advantages:

1. It usually beats the other methods when the instance is very hard
2. It can be successfully used in combination with branch-and-bound to produce hybrid algorithms for KP (Plateau and Elkihel, 1985) and for the other knapsack type problems (Martello and Toth (1984a))

Chapter 3
Methodology

3.1 The Model

The problem that is considered is to determine the minimum number of workers required at a truck terminal to perform the stripping and loading operations on the trucks arriving during one shift. The shift that is about 8 hours long starts late at night. While the trucks arrive at a terminal in a non-deterministic manner, based on the information already available the time of arrival of a truck can more or less be predicted in advance. Hence, for the purpose of analysis here, it will be assumed that the arrival time of a truck is known with certainty. Similarly, the departure time is assumed to be deterministic.

Let there be m trucks to be stripped during a shift. Let $\underline{a_j}$ be a vector of size m and be defined such that $a_{ij} = 1$ if truck i is assigned to worker j, and $a_{ij} = 0$, otherwise. Thus,

$\underline{a}_j$ represents an assignment of some trucks to worker j. Let $y_j = 1$ if worker j is working during the shift and $y_j = 0$ otherwise. The staffing problem can thus be modeled as follows. Determine y_j so as to

$$\text{minimize} \sum_{j=1}^{N} y_j$$

$$\textit{subject to} \sum_{j=1}^{N} \underline{a}_j \; y_j \; = \; \underline{e} \qquad (1)$$

$$y_j \; = \; 0 \; \textit{or} \; 1 \quad \forall \quad j \; = \; 1, \ldots\ldots, N$$

Here N represents all possible assignments of a worker to perform the stripping and loading operations during a shift and $\underline{e}$ is a vector of 1's. The constraints in (1) represent the fact that each truck is served by only one worker. The trucks will be assigned in such a fashion that the total number of workers is minimized, which is the objective function. This is a set partitioning problem where the trucks have to be partitioned into mutually exclusive sets so that a truck is served only by one worker while a worker could work on several trucks during the shift, albeit one-at-a-time.

A key feature of this formulation is the use of vectors $\underline{a}_j$. The process of generating $\underline{a}_j$ is termed matrix generation. This is the key advantage of this formulation. The columns $\underline{a}_j$ can be generated taking into consideration the prevailing conditions or constraints on the assignment of the workers to the trucks. For example, one important consideration is the arrival time of a truck at the terminal and its scheduled departure time. These may be

different for different trucks. Such requirements can be considered a priori during the generation of $\underline{a}_j$'s so that all assignments are feasible. Also, the employee break time can be conveniently included during this matrix generation process.

3.2 Solution Methodology

Two approaches are being used to solve this problem:

1. Solution using a bin packing heuristic procedure
2. Solution using an enumeration procedure for the underlying 0-1 integer program

3.2.1. Solution using a bin packing heuristic procedure

First-Fit (FF) algorithm has been used for obtaining a heuristic solution to the problem at hand. This algorithm best represents our situation, where we want each worker to be assigned as many trucks as he can feasibly work on in an 8-hour shift.

These are the following steps:

Step 1. Order all the trucks arriving on a given shift in the increasing order of their arrival times. Assign the first truck to worker 1.

Step 2. Consider Trucks 2, ..., *n* by increasing indices: each unassigned truck is assigned to the current worker if he can feasibly work on it within the 8-hour shift.

Step 3. Otherwise, among the remaining trucks, assign the truck with the lowest index to a new worker, which becomes the current worker. If all the trucks have been assigned, stop; otherwise, go to Step 2.

A comparable study of the solutions obtained using the FF bin packing heuristic and those obtained using the proposed algorithm is presented in the next chapter.

3.2.2. Solution using a Column Generation procedure for the underlying 0-1 integer program

The second approach is a procedure based on the column generation technique of linear programming. The following is an intuitive property of this formulation.

- if the (0-1) requirements of the variables are relaxed and the resultant problem is solved as a linear program, then at the optimum, $0 \leq y_j^* \leq 1$.

Unlike, other algorithms for the set partitioning problem which require the A matrix to be available before hand, the algorithm that we develop here uses the strategy of the *column generation* technique and generates a column to enter the basis as needed. Following are the essential steps of the algorithm:

Step1. Solve the linear relaxation of set-partitioning problem using the columns generated by the knapsack algorithm. This process has been detailed in figure 3.2.

Step2. Take the optimal basis from above and formulate it as a set covering formulation.

Step3. Use CPLEX to solve this Set-Covering problem

Step4. Delete the rows covered more than once to get a solution to the set-partitioning problem

A pictorial presentation of the above procedure is given in figure 3.1.

As mentioned above, we begin with the linear relaxation of the problem on hand.

$$\text{minimize} \sum_{j=1}^{N} y_j$$

$$\textit{subject to} \sum_{j=1}^{N} \underline{a}_j \, y_j = \underline{e}$$

$$y_j \leq 1 \quad \forall \quad j = 1,\ldots\ldots, N$$

In the above formulation the 0-1 constraint on the variables has been replaced with inequalities. The procedure for solving this relaxation is presented in figure 3.2.

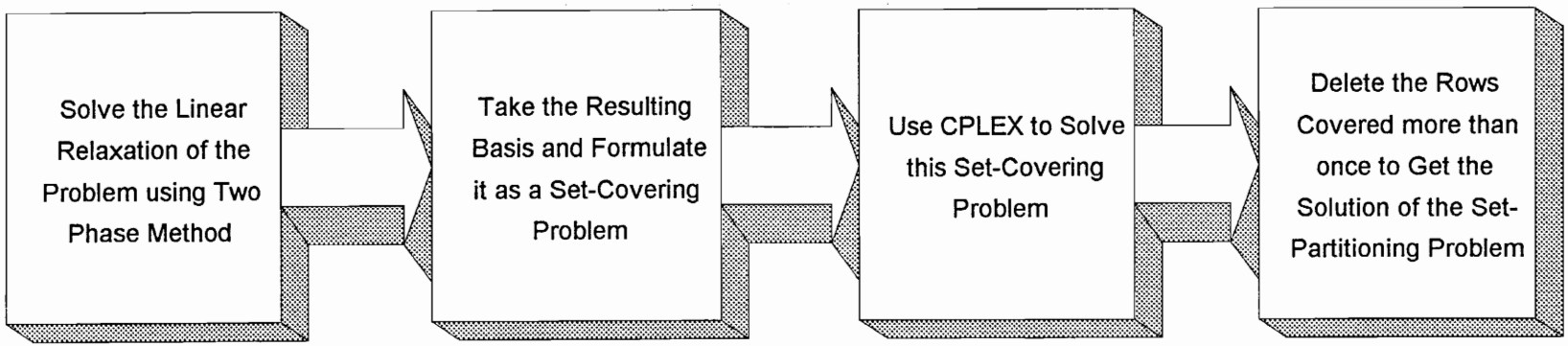

Figure 3.1
Overall Solution Procedure

The two-phase method has been used to solve this problem where we start with a basic feasible solution using artificial variables. These artificial variables are eliminated by the end of the phase I method. At the end of phase I, if there are some artificial variables still present in the basis at value 0, we pivot on them to force these out of the basis before moving to phase II. Let this initial basic feasible solution be denoted by x_B, with associated basis matrix B, and cost coefficients c_B. The simplex multipliers associated with the basis are

$$\pi = c_B B^{-1}$$

and are always made available by the simplex method. To improve the basic feasible solution we "price out" all columns corresponding to nonbasic variables by forming their relative cost coefficients.

$$\overline{c_j} = c_j - \pi \underline{a}_j$$

If

$$\min_j \bar{c}_j = \bar{c}_s < 0$$

then barring degeneracy, the current solution can be improved by introducing x_s into the basis via a pivot transformation.

As the number of columns in A is generally very large, finding min $\bar{c}_j$ by computing each $\bar{c}_j$ and comparing is very tedious, if not impossible. In general, we

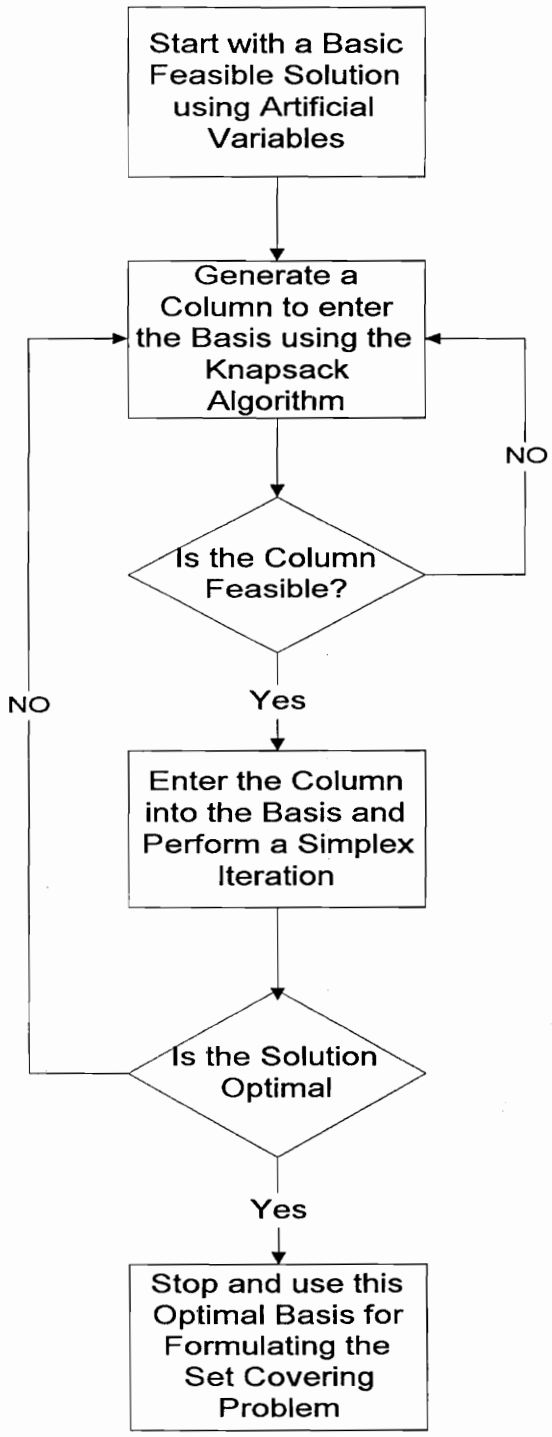

Figure 3.2
Solving the Linear Relaxation of the problem

assume that all columns, $\underline{a}_j$ are drawn from a set, S, which is a set of all m vectors that satisfy some side constraints, in our case - the feasibility constraint due to the dynamic arrival of the trucks. Here $\underline{a}_j$ are the columns of 0 and 1. The column to enter the basis is then obtained by solving the subproblem

$$maximize \quad \sum_{j=1}^{n} p_j \underline{a}_j - c_j$$

$$subject\ to \quad \sum_{j=1}^{n} w_j \underline{a}_j \leq M$$

$$\underline{a}_j = 0\ or\ 1 \quad (j = 1, \ldots, n)$$

where M = time available to a worker = *8 hrs.*

w_j = Stripping time of truck j

This approach is called column generation because, in solving the subproblem, only a small subset of columns in S are typically examined, and these are generated when needed. A knapsack algorithm discussed in the next section has been used to solve this subproblem. Thus the columns of A need not be kept in the computer storage a priori, thereby implying a definite advantage of this approach. In addition, any side constraints can be considered during the generation of the columns. Side constraints refer to the fact that the workers should be able to finish the trucks assigned to them in the stipulated time, i.e., the duration of the shift. If the assignment is infeasible, another column is generated

and tested for feasibility (refer figure 3.7). The process continues until a column with reduced cost greater than zero and satisfying the feasible assignment criterion is found.

Once the optimal solution to the relaxed problem has been found, the final step is to get the optimal integer solution to the set partitioning problem. In order to achieve this, the optimal basis of the linear program is used to form an integer program as given below. This is a set covering formulation where the equalities of the set partitioning formulation discussed earlier have been replaced by inequalities.

$$\text{minimize} \sum_{j=1}^{N} y_j$$

$$\textit{subject to} \sum_{j=1}^{N} \underline{a}_j \; y_j \; \geq \; \underline{e}$$

$$y_j \; = \; 0 \; or \; 1 \quad \forall \quad j \; = \; 1, \ldots\ldots, N$$

All the notations used above are the same as discussed before. The $\underline{a}_j$ are the columns in the final optimal basis of the linear program obtained at the end of phase II.

The set covering problem can also be defined as: Given a finite set S, a family of subsets $\{S_j \subseteq S: j \in J\}$ and costs, c_j, associated with the S_j, choose a minimum total cost collection of the subsets that includes every element of S at least once. Since, ours is an even cost problem (i.e., all cost coefficients are identity) we try to find the minimum

number of columns that will cover all the rows. Though, some of the rows will be covered more than once, they are deleted from all but one of the columns that cover it. This deletion of rows, however, would not change the best solution found earlier. The above formulation has been solved using the software *CPLEX*. *CPLEX* is a linear and integer optimizer. It solves the integer program using branch and bound procedure.

The results obtained from the bin packing heuristic approach and our algorithm are given in chapter 4.

3.3 The knapsack algorithm for the column generation

As discussed earlier, the knapsack algorithm has been used to solve the subproblem. The master problem passes down a new set of cost coefficients each time to the subproblem and receives a new column based on these cost coefficients. Though plenty of algorithms are available for solving the knapsack problem of fairly large size using branch and bound techniques, the technique presented below is different in the way that it tests for the feasibility (feasible assignment criterion) of every solution generated due to the side constraints before it can be pivoted into the simplex basis. Thus, the problem is solved as if it had another constraint for the feasible assignment criterion.

The algorithm starts by building the first current solution through the integer solution to the continuous problem given by

$$0 \leq a_j \leq 1 \qquad (j = 1, \ldots, m)$$

$$a_j = 1 \qquad (j = 1, \ldots, l)$$

$$a_j = 0 \qquad (j = l+2, \ldots, m)$$

where l = greatest index for which $\sum_{j=1}^{l} w_j \leq M$.

The solution is tested for feasibility, i.e., if the generated column is a feasible integer solution to the knapsack problem and is a feasible assignment, it is pivoted into the simplex basis. If this solution is infeasible, a depth-first branch and bound search is then performed.

A forward move is performed where an attempt is made to introduce the largest possible set of new elements into the current solution. The solution is updated by setting to 1 the values a_j of the elements found by the forward move. The solution at this stage is tested for feasibility. If infeasible, a backtracking move is performed.

A backtracking move consists of updating the current solution by setting to 0 the value of a_k, with $k = \max\{j | a_j = 1\}$. Then, if at least one of the elements following the k^{th} has a value of w_j small enough to allow its introduction into the current solution after the backtracking, the forward move is performed.

The algorithm continues with feasibility checks at each stage. As soon as a column with feasible assignment is found, it is entered into the basis. The detailed steps of the algorithm are given below. The building phase corresponds to steps 1. and 2.; and the

backtracking move to step 3. The flow diagrams for these three steps have been presented in figures 3.4, 3.5, and 3.6.

1. [*Initialize*]

Order all p_j and w_j ($j = 1, \ldots, m$) in non increasing order of p_j/w_j.

Find l = maximum index for which $IW = \sum_{j=1}^{l} w_j \leq M$.

Compute ($\text{Min}_j = \min\{w_k | j < k \leq m\}, j = 1, \ldots, m - 1$), $\text{Min}_m = M + 1$.

Set $M = M - IW$.

Set ($a_j = 1, j = 1, \ldots, l$), ($a_j = 0, j = l + 1, \ldots, m$). Check feasibility

If feasible, three possibilities exist:

(a) $l < m - 2$; set $i = l + 2$. If $M \geq \text{Min}_{i-1}$, go to 2. Otherwise, go to simplex

(b) $l = m - 2$; if $M \geq w_m$, set $i = m$, go to 2. Otherwise, go to simplex

(c) $l = m - 1$; go to simplex

Otherwise, set $i = l + 2$, go to 3.

2. [*Forward Move*]

If $w_i \leq M$, set $a_i = 1$, $M = M - IW$; if $M \geq \text{Min}_i$, set $i = i + 1$ and repeat 2, otherwise($M < \text{Min}_i$), check feasibility; if feasible go to simplex, otherwise (infeasible), set $a_i = 0$, $M = M + w_i$, $i = i + 1$. If $i > m$, go to simplex. Otherwise, repeat 2.

Otherwise($w_i > M$), set $i = i + 1$, if $i > m$, check feasibility, if feasible, go to simplex, otherwise go to 3.

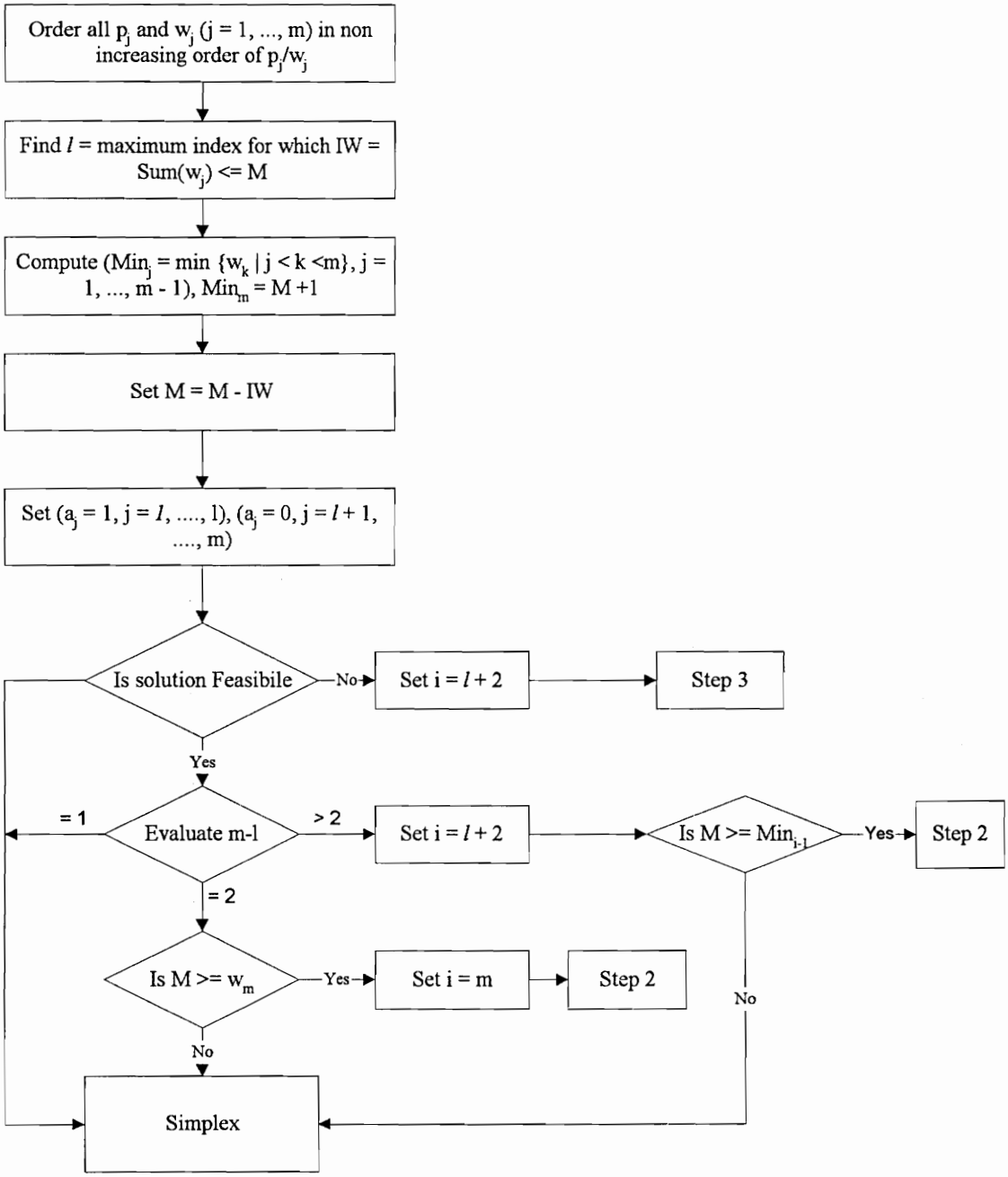

Figure 3.3

Initialization Phase for the Knapsack Algorithm

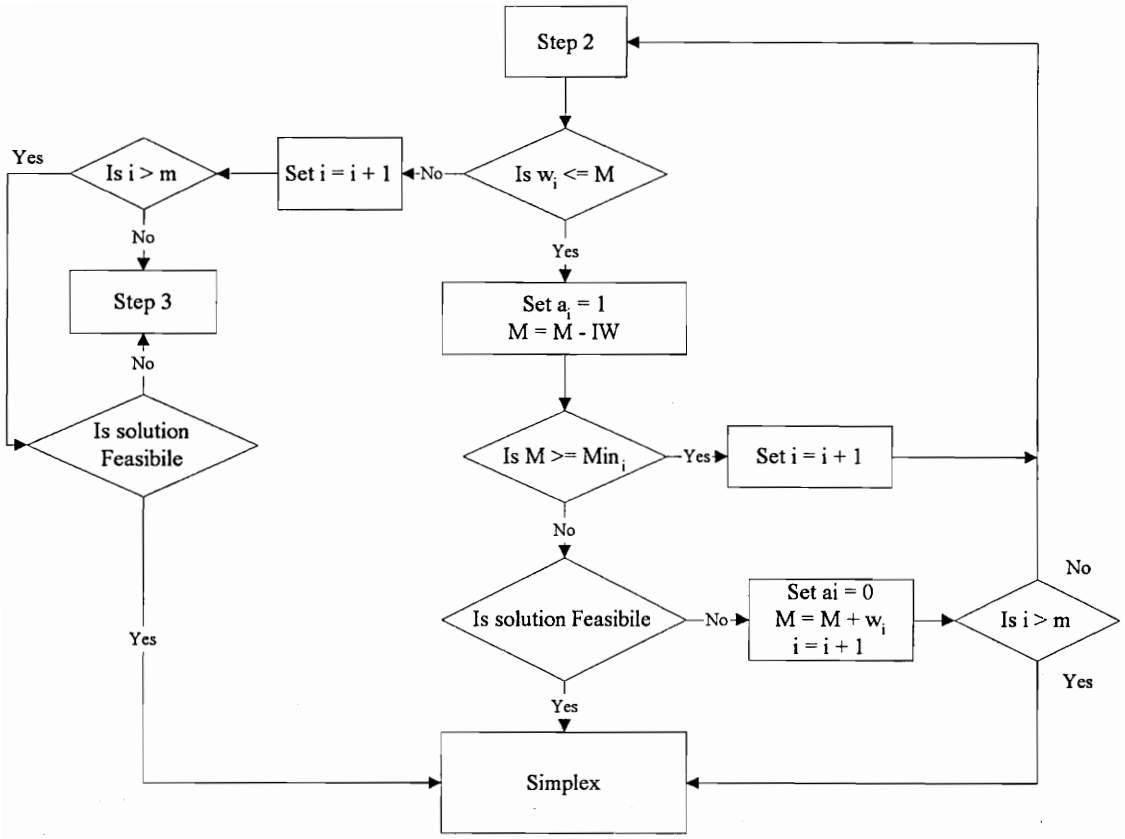

Figure 3.4

Forward move phase for the Knapsack Algorithm

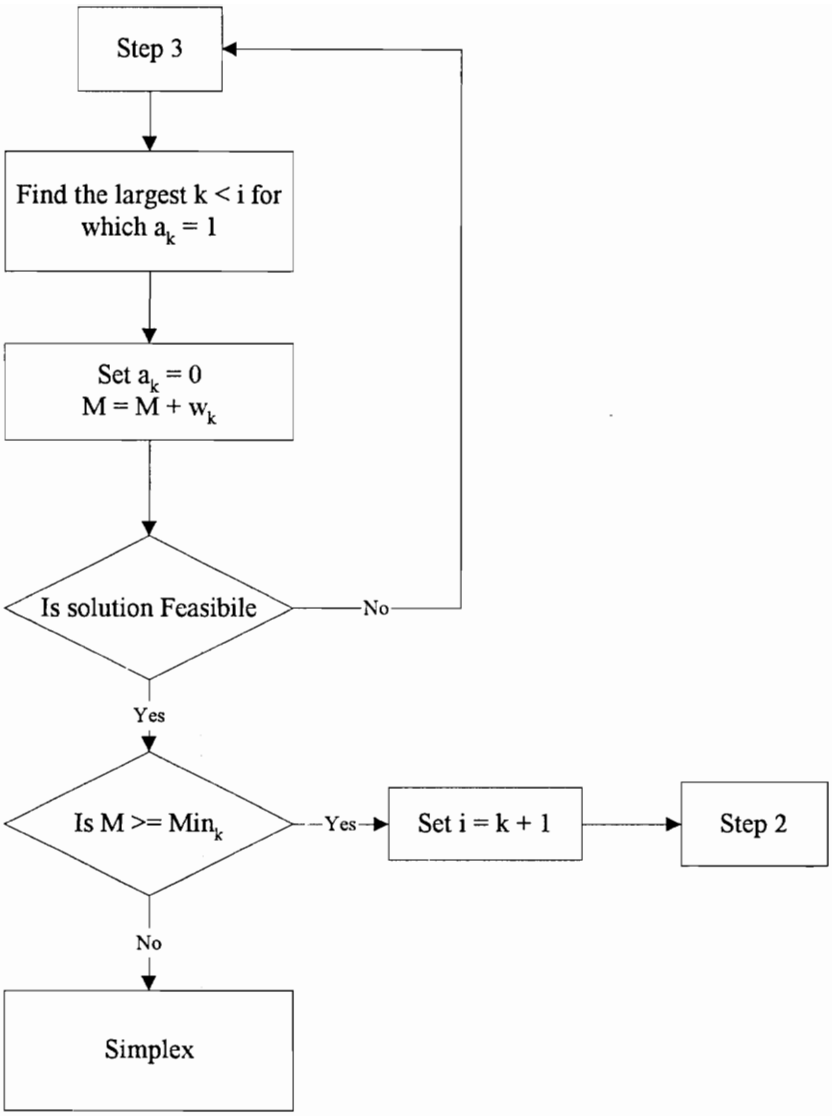

Figure 3.5

Backtrack move phase for the Knapsack Algorithm

3. [*Backtracking Move*]

Find the largest $k < i$ for which $a_k = 1$.

Set $a_k = 0$, $M = M + w_k$, check feasibility.

If feasible; if $M \geq \text{Min}_k$, set $i = k + 1$, go to 2, otherwise go to simplex

Otherwise, repeat 3.

Remarks:

(Min_j)

The vector (Min_j), whose definition is at step 1, enables one to know whether, given the current situation (i.e., the current value of M), at least one more element can be introduced into the solution: so, at steps 2 and 3, a test based on (Min_j) is useful to decide whether a forward move has to be performed or not.

(Fathoming Criterion)

If a solution corresponding to a node is an infeasible assignment, then the node is fathomed, because the set of which this solution is a subset will also give an infeasible assignment. The above statement is intuitive. If a worker can not finish work on the already assigned trucks in the stipulated time, assigning more trucks to this worker would just increase the degree of infeasibility.

3.4 Computer Implementation of the Algorithm

For computer implementation, a batch file was written which sequentially runs the algorithm code, *CPLEX* formulation code, *CPLEX*, and the post-processing subroutine. The computer code for the algorithm for the set-partitioning formulation has been written

in C. This code *algo.cpp* reads a data file, manipulates it, generates a heuristic solution and the non-integer optimal solution. The final basis of this linear optimal solution is written to an output file which is then read by *cplex.cpp* and converted to a set covering formulation readable by the *CPLEX* Optimizer. *CPLEX* then optimizes this problem and prints the integer solution to a file output.out. This solution is then read in by the post processing code *post.cpp* which manipulates the *CPLEX* output and presents the results in a form understood by managers. It includes the total stripping time of the trucks, which is just the summation of the processing time required by all the trucks in a given shift. Also included are the man-hours requirement and the truck assignments. A detailed flow diagram of the procedure is given in figure 3.6. A copy of all the three modules of the code is also included in Appendix 1.

3.4.1 Code for Set-Partitioning Algorithm

The first function *read*( ) reads the data from the input file. The next function *sort_arr*( ) counts the number of trucks arriving on that particular shift and sorts the data according to the increasing order of their arrival times. The function *heuris_sol*( ) is the bin packing heuristic solution. As discussed earlier, the First-fit approach has been used for the bin packing heuristic. This function prints out the number of workers required on a particular day and their feasible assignments.

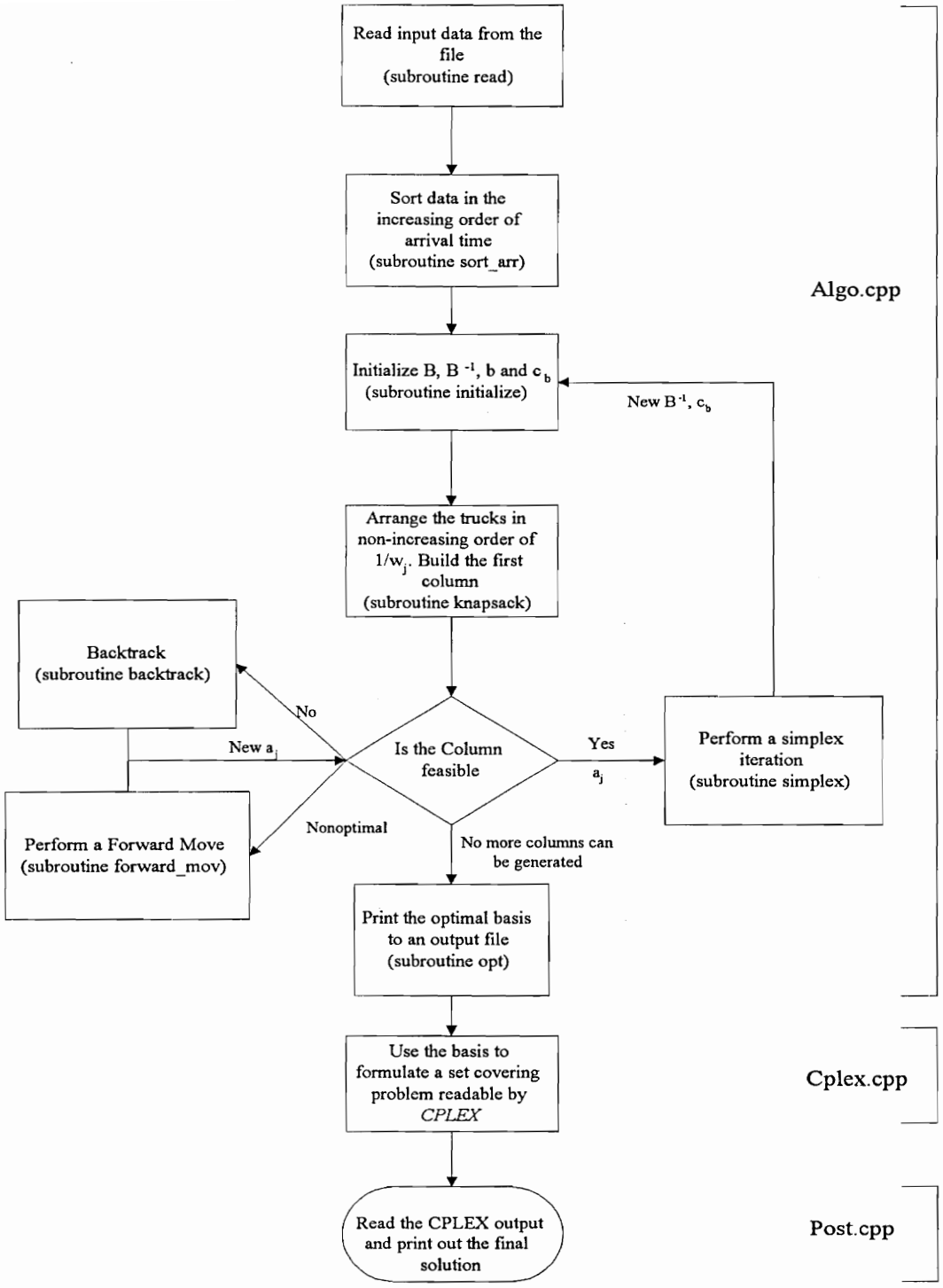

Figure 3.6

Computer implementation of the Algorithm

The next function *initialize*( ) initializes the basis B, inverse B^{-1}, the right hand side b, and the cost coefficients c for the simplex routine. The next three functions *knapsack*( ), *forward_mov*( ), and *backtrack*( ), pertain to the knapsack algorithm. The *knapsack*( ) function sorts the data in the non-increasing order of their p_j/w_j ratios. This function then builds the first solution through the integer solution to the continuous problem. This solution is tested for feasibility by the *feasibility*( ) subroutine discussed later. If feasible, the column generated is pivoted into the basis, i.e., the column is passed onto the LP subroutine *simplex*( ). Otherwise, *backtrack*( ) routine is called and a backtrack move is performed and the *feasibility*( ) subroutine is called to check for feasibility. A backtrack move is performed as long as no feasible solution is found. Once a feasible solution is found, a forward move is performed by the function *forward_mov*( ). This function attempts to introduce a set of new elements into the current solution. Again, the solution is tested for feasibility and *simplex*( ), *forward_mov( )*, or *backtrack*( ) are called depending upon whether the column is feasible or infeasible.

The *feasible*( ) subroutine begins by arranging the items according to their original order. Then, starting with the first assigned truck's arrival time, it calculates the makespan for all the trucks assigned in that particular assignment. If the makespan is less than the shift time (8 hours) and the reduced cost, i.e. $\pi \underline{a}_j - c_j$ is greater than zero, it returns a value *'GOOD'* . If the reduced cost is less than or equal to zero it returns *'NONOPTIMAL,'* and *'INFEASIBLE'* otherwise. The flowchart for feasible subroutine is given in figure 3.7.

The LP subroutine *simplex*( ) receives the column $\underline{a}_j$ from one of the knapsack functions, performs one simplex pivot, and returns the simplex multipliers π to the knapsack function for the next column to be generated. The *simplex*( ) function is a

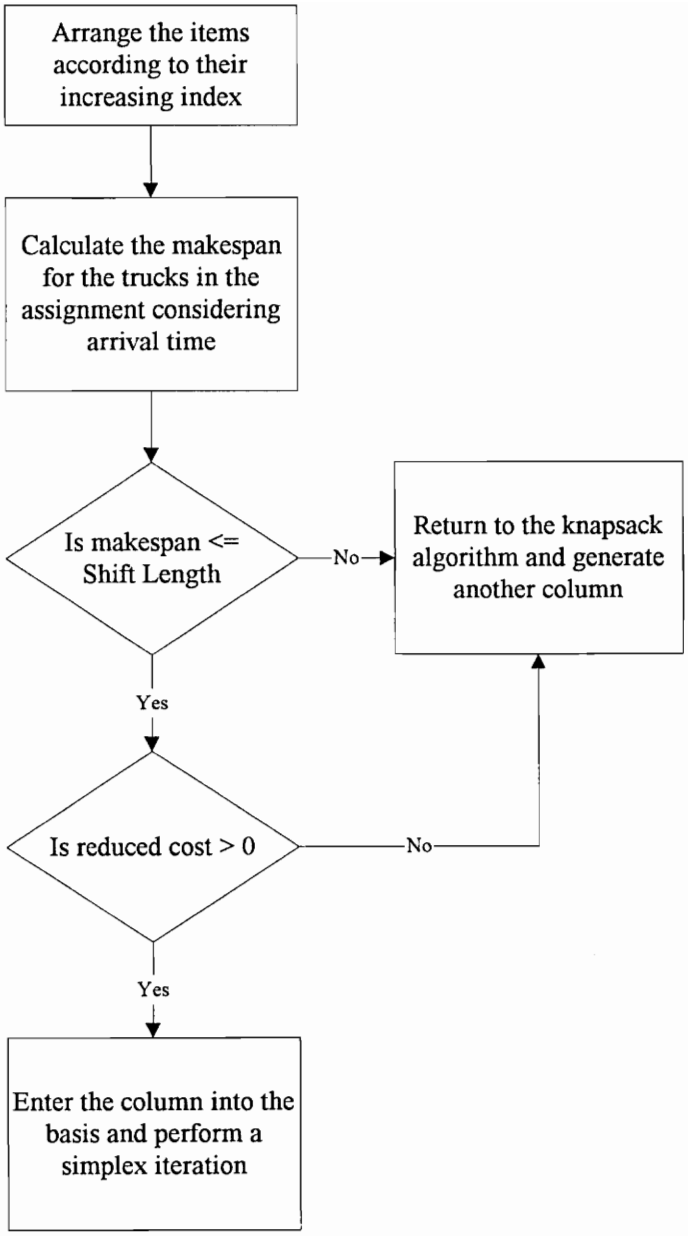

Figure 3.7

The feasibility criterion

revised simplex which makes an all artificial start and uses the two phase method to kick out the artificial variables.

The process of generating columns and pivoting them into the basis continues until no more columns with reduced cost greater than zero can be generated. At this point, the search is stopped and the final basis is printed to an output file.

3.4.2 CPLEX formulation code

This function reads the optimal basis and prints out an integer program for the set-covering problem in the *lp* form readable by *CPLEX*. *CPLEX* gives the optimal number of workers required on that shift and tells which columns in the solution cover all the rows. These columns give a feasible assignment of all the trucks to workers working on that shift and vice-versa. *CPLEX* prints out the final solution to a file.

3.4.3 Post-Processing code

This code reads the *CPLEX* output file and interprets the columns in the optimal integer basis and trucks corresponding to the numbers in those columns. It prints the final solution and gives a comparison of this solution in terms of man-hour requirement and the total stripping time required. This tells us how far we are from the actual stripping time. If all the trucks arrived at or before the beginning of each shift, the problem would be trivial and the man-hour requirement would equal the total stripping time. But, since most of the trucks arrive after the shift is started, there is a wait time for both the trucks as well as the

workers resulting in a gap between man-hour requirement generated through the algorithm and the total stripping time.

3.5 Data Storage and Input/Output Files

Once the arrival times of the trucks arriving on a given shift and their stripping time are known they are entered into a file using any text-editor. The first column of this file should have the arrival times and the second column should have the corresponding stripping times. There should be no carriage return at the end of the file as it would be read as a truck with arrival time of 0.00 and stripping time of 0.00, and eventually result in a 'floating point divide by' error.

This file is then read by the computer code to generate the heuristic and the optimal basis which is printed to the file named *basis.in*. The *CPLEX* formulation code reads basis.in and prints out the file *output.lp*. *CPLEX* reads this file and prints the optimal integer solution to the file named *output.out*. The post-processing code reads this file and prints out the final solution to *post.out*. The user does not have to remember or use these file names above. The user will just specify the input file name and then the batch file will run all the processes and at the end open the *post.out* file in the edit mode for the user.

Chapter 4
Results and Analysis

The algorithm presented in this thesis has been successfully applied to solve a scheduling problem in the trucking industry. Data were received from the Overnite Transportation Company's Roanoke, VA and Harrisburgh, PA hubs. The Roanoke hub is quite small as compared to the Harrisburgh hub. It handles about 30 trucks a day and works only one shift, which lasts 8-9 hours depending on the volume and need. The Harrisburgh hub handles on an average 200 trucks a day and works 24 hours a day divided into 3 shifts of 8 hours each.

4.1 Case Studies

4.1.1 Roanoke, VA Hub

This hub processes a maximum of 30 trucks/day. Though the average shift length is 8-9 hours, for our analysis purpose we have assumed each shift to last 8 hours. This hub

had the data on actual man-hours clocked on the days studied. A comparison of the actual vs. the requirement generated by our algorithm is presented in table 4.1. Also, the solution generated by the bin packing heuristic is presented for reference. Table 4.2 compares the solutions obtained with the IP optimal. The input data used for the Roanoke hub is appended in Appendix 2.

4.1.2 Harrisburg, PA Hub

This is one of Overnite's largest hubs. As mentioned earlier, a work day at this hub is divided into three shifts of 8 hours each. The first shift begins at 12:00 midnight. For the analysis, trucks to be processed during each shift are considered according to the following criteria:

- Trucks that arrived before the beginning of the first shift are processed in the first shift only

- Trucks that arrive during any shift and require processing time more than the time remaining in that shift are processed in the next shift. These trucks are assumed to have arrived at time 0.00 for the next shift

Approximately three weeks of data were available for this hub and the results are presented in table 4.3. Sample data for a day for this hub is also attached in Appendix 2 for reference.

Table 4.1

Roanoke, VA Hub

(Comparison between LP Lower Bound and the Solution Obtained)

DATE	TRUCKS /SHIFT	LP LOWER BOUND	SOLUTION BY PROPOSED METHOD	HEURISTIC SOLUTION	% DIFFERENCE IN LB AND NEW SOLUTION	RUN TIME MIN:SEC
10/3/94	25	5	5	6	0%	0:26
10/4/94	25	4	4	5	0%	1:27
10/5/94	15	2	2	3	0%	0:15
10/6/94	21	4	4	6	0%	0:46
10/7/94	16	4	4	5	0%	0:17
10/10/94	26	5	5	6	0%	0:25
10/12/94	16	4	4	4	0%	0:33
10/13/94	19	3	4	5	33%	0:26
10/14/94	20	5	5	7	0%	0:36
10/17/94	24	6	7	7	17%	0:21

Table 4.2

Roanoke, VA Hub

(Comparison between Optimal Solution and the Solution Obtained)

DATE	TRUCKS /SHIFT	LP LOWER BOUND	SOLUTION BY PROPOSED METHOD	IP OPTIMAL SOLUTION	% DIFFERENCE BETWEEN IP OPTIMAL AND SOLUTION OBTAINED
10/3/94	25	5	5	5	0%
10/4/94	25	4	4	4	0%
10/5/94	15	2	2	2	0%
10/6/94	21	4	4	4	0%
10/7/94	16	4	4	4	0%
10/10/94	26	5	5	5	0%
10/12/94	16	4	4	4	0%
10/13/94	19	3	4	4	0%
10/14/94	20	5	5	5	0%
10/17/94	24	6	7	6	17%

Table 4.3

Harrisburg, PA Hub

(Comparison between LP Lower Bound and the Solution Obtained)

DATE	SHIFT	TRUCK /SHIFT	LP LOWER BOUND	SOLUTION BY PROPOSED METHOD	HEURISTIC SOLUTION	% DIFFERENCE BETWEEN LB AND SOLUTION OBTAINED	RUN TIME MIN:SEC
10/1/94	a	53	18	19	26	6%	5:21
10/1/94	b	72	21	24	25	14%	9:41
10/1/94	c	37	14	15	16	7%	0:36
10/3/94	a	26	8	8	11	0%	0:32
10/3/94	b	24	9	9	10	0%	0:21
10/3/94	c	42	8	9	11	13%	2:54
10/4/94	a	67	22	23	32	5%	23:22
10/4/94	b	90	28	33	36	18%	36:58
10/4/94	c	45	13	13	13	0%	0:37
10/5/94	a	80	29	30	41	3%	27:03
10/5/94	b	86	28	31	32	11%	17:22
10/5/94	c	51	11	13	13	18%	2:28
10/6/94	a	71	23	25	29	9%	22:58

Table 4.3 (Contd.)

DATE	SHIFT	TRUCK /SHIFT	LP LOWER BOUND	SOLUTION BY PROPOSED METHOD	HEURISTIC SOLUTION	% DIFFERENCE BETWEEN LB AND SOLUTION OBTAINED	RUN TIME MIN:SEC
10/6/94	b	91	31	32	36	3%	15:23
10/6/94	c	32	8	9	10	13%	0:43
10/7/94	a	77	27	27	35	0%	21:14
10/7/94	b	70	26	29	36	12%	3:52
10/7/94	c	37	9	9	10	0%	2:30
10/8/94	a	55	18	19	24	6%	3:31
10/8/94	b	57	16	18	23	13%	1:59
10/8/94	c	10	4	4	5	0%	0:13
10/10/94	a	32	8	9	13	13%	1:19
10/10/94	b	37	10	11	13	10%	1:42
10/10/94	c	48	13	14	17	8%	3:01
10/11/94	a	74	9	11	11	22%	24:39
10/11/94	b	84	23	24	34	4%	19:29
10/11/94	c	42	24	26	26	8%	1:58
10/12/94	a	91	32	34	44	6%	33:02
10/12/94	b	70	25	28	28	12%	6:52

Table 4.3 (Contd.)

DATE	SHIFT	TRUCK /SHIFT	LP LOWER BOUND	SOLUTION BY PROPOSED METHOD	HEURISTIC SOLUTION	% DIFFERENCE BETWEEN LB AND SOLUTION OBTAINED	RUN TIME MIN:SEC
10/12/94	c	54	14	16	17	14%	7:50
10/13/94	a	78	30	31	34	3%	9:18
10/13/94	b	89	25	26	28	4%	34:08
10/13/94	c	53	12	14	17	17%	7:50
10/14/94	a	83	30	33	39	10%	18:06
10/14/94	b	92	33	34	38	3%	23:34
10/14/94	c	46	15	17	18	13%	0:50
10/15/94	a	69	29	30	37	3%	7:52
10/15/94	b	67	18	20	26	11%	4:55
10/15/94	c	26	6	7	10	17%	1:02
10/17/94	a	25	9	9	11	0%	13:26
10/17/94	b	23	6	7	8	17%	0:25
10/17/94	c	50	12	13	16	8%	7:10
10/18/94	a	61	19	20	27	5%	12:20
10/18/94	b	99	31	34	37	10%	10:00
10/18/94	c	46	9	11	11	22%	2:14

Table 4.3 (Contd.)

DATE	SHIFT	TRUCK /SHIFT	LP LOWER BOUND	SOLUTION BY PROPOSED METHOD	HEURISTIC SOLUTION	% DIFFERENCE BETWEEN LB AND SOLUTION OBTAINED	RUN TIME MIN:SEC
10/19/94	a	77	27	29	33	7%	20:08
10/19/94	b	93	29	32	34	10%	49:40
10/19/94	c	42	10	11	13	10%	1:17
10/20/94	a	75	30	32	37	7%	13:54
10/20/94	b	85	28	30	35	7%	5:56
10/20/94	c	56	12	14	17	17%	9:36
10/21/94	a	77	26	29	33	12%	13:32
10/21/94	b	76	27	28	36	4%	3:47
10/21/94	c	45	11	12	13	9%	1:17
10/22/94	a	64	26	28	32	8%	4:01
10/22/94	b	55	17	18	21	6%	3:50
10/22/94	c	28	11	12	13	9%	0:23

4.2 An Analysis of the Results

The results presented above have been found to be between 0-22% of the LP lower bound. The mean deviation from the lower bound is 8.6% with a standard deviation of 5.6%. The median is at 8.3%.

However, for the Roanoke hub, we found optimal solution to 9 out of the 10 problems solved with the presented algorithm. For the Harrisburg hub, solutions obtained are upto 30% better over the best heuristic possible.

The maximum deviation in the results has been found to occur for the '*a*' shift. This is due to the fact that most of the trucks arrive during this shift and this results in lot of slack time compared to the other shifts where most of the trucks are already available for workers to work on at the beginning of the shift.

4.3 Run Time Requirements

The run time for each problem run is presented in tables 4.1 and 4.2. These run times are on an IBM PS/277 486DX2 machine. The time requirement has been observed to greatly depend upon the data type rather than the size of the problem. The algorithm was also tried to run on an IBM 750-P90 machine with Pentium processor and the run times in all the instances tried were approximately 1/3rd of the times presented.

The following considerations have been included in the algorithm to reduce the Run time requirements:

- if even after 1000 iterations in the knapsack subroutine no feasible column is found to be enterable, further search is stopped. This was done after extensively experimenting with a number of problems which were run to the end and produced the same result as with our stopping criterion. Hence, no compromise on the quality of solutions presented has been made.

- due to the floating point calculations, the simplex multipliers are not 0 or 1 when they should be. Thus, a column is entered into the simplex basis only if the solution value will improve by more than $\in$. The $\in$ value has been set to 0.01. Again, different values of $\in$ were tried and 0.01 has been found to be the ideal value from the perspective of run time and quality of the result.

4.4 Concluding Remarks and Directions for Future Research

An algorithm producing near optimal solutions to the scheduling problem in the trucking industry has been presented in this thesis. The major advantage of the algorithm lies in the fact that the A matrix is not required a priori.

We started with an explanation of the problem and then formulated and qualified this problem as a set-partitioning problem. We then analyzed the algorithms used to solve the set-partitioning problem that have appeared in the literature so far. A column

generation technique-based solution methodology was developed and coded in the C programming language. We tested the quality of solutions generated on the actual data we received from a trucking company.

We suggest use of the presented technique to solve the airline crew scheduling problem, which is also a set-partitioning problem. In the crew scheduling problem, safety regulations and union requirements need to be considered. These conditions can be met by including them in feasibility testing of each column generated.

Also, there is a need to make this algorithm more efficient by using better memory management techniques in the C code.

The results presented could not be compared with any of the other algorithms. The reason was the absence of A matrix which is a prerequisite for all the algorithms developed and presented in the past. Work is needed to device a methodology to compare the results among different algorithms.

References

1. Andrew, G. et al., "On the Generalized Set Covering Problem," CDC, Data Centers Division, Minneapolis, 1968
2. Arabeyre, J. P. et. al. "The Airline Crew Scheduling Problem: A Survey" Transportation Science, 3, 2, 140-163, May, 1969
3. Baker, E. K. et. al, "Efficient Heuristic Solution to an Airline Crew Scheduling Problem," AIIE Transactions, 79-84, June 1979.
4. Balas, E, "Some Valid Inequalities for the Set Partitioning Problem," MSRR 368, Carnegie-Melon University, Pittsburgh, July 1975c
5. Balas, E., "An Additive Algorithm for Solving Linear Programs with 0-1 Variables," Operations Research, 13, 517-546, 1965.
6. Balas, E., and Zemel, E., "An algorithm for Large Zero-One Knapsack Problems," Operations Research, 28, 1130-1154, 1980
7. Balas, E., Padberg, M. W., "On the Set Covering Problem: II. An Algorithm for Set Partitioning," Operations Research, Vol. 23, No. 1, January-February 1975
8. Balas, E., Samuelson, H., "A Symmetric Subgradient Cutting Plane Method for Set Partitioning," W. P. 5-74-75, Carnegie-Melon Univ., Pittsburgh, August 1974b
9. Balas, E., Samuelson, H., "Finding a Minimum Node Cover in an Arbitrary Graph," MSRR 325, Carnegie-Melon University, Pittsburgh, November 1973

10. Balas, E., Samuelson, H., "Finding a Minimum Node Cover in an Arbitrary Graph II," MSRR 336, Carnegie-Melon University, Pittsburgh, April 1974a

11. Balinski, M. L. and Quandt, R., "On an Integer Program for a Delivery Problem," OperationsResearch, 12, 300-304, 1964

12. Balinski, M. L., "Integer Programming: Methods, Uses, Computation," Management Science, 12, 3, 253-313, November, 1965

13. Bellman, R., "Some Applications of the Theory of Dynamic Programming- A Review" Operations Research 2, 275-288, 1954

14. Busacker, R.G., and Saati, T. L., Finite Graphs and Nmetworks, McGraw-Hill, Newyork, 1965.

15. Cassidy, P. J. and Bennett, H. S., "TRAMPS-A Multi-Depot Vehicles Scheduling System," Operational Research Quarterly, 23, 2, 151-163, 1975.

16. Christofides, N., "The Vehicle Routing Problem," NATO Confrence on Combinatorial Optimization, Paris, Sept., 1974.

17. Clarke, G. and Wright, S. W., "Scheduling of Vehicles from a Central Depot to a Number of Delivery Points," Operations Research, 12, 4, 568-581, 1964

18. Cobham, A. et. al., "An Application of Linear Programming to the Minimization of Boolean Functions," Research Report RC-472, IBM Research Center, June, 1961.

19. Cobham, A., "A Statistcal Study of the Minimization of Boolean Functions Using Integer Programming," Research Report RC-756, IBM Research Center, June, 1962

20. Crowder, H., Johnson, E. L., and Padebrg, M., "Solving Large-Scale Zero-One Linear Programming Problems," Operations Research, 31, 5, 803-834, September-October 1983

21. Daly, W. N. and Spierer, S. J., "Private Communication," Mobile Oil Corporation, July 11, 1969

22. Dantzig, G. B. and Ramser, J. H., "The Truck Dispatching Problem," Managemment Science, 6, 1, 80-91, 1959

23. Dantzig, G. B., "Discrete Variable Extremum Problems," Operations Research, 5, 161-310, 1957

24. Day, R. H., "An Optimal Extracting From A Multiple File Data Storage System: An Application of Integer Programming" Operations Research, 13, 3, 482-494, 1965

25. Dietrich, B. L. and Escudero, L. F., "More Coeffficient Reduction for Knapsack-like Constraints in Zero-One Programs with Variable Upper Bounds" IBM T. J. Watson Research Center. RC-14389, York Town Heights(NY), 1989a

26. Dietrich, B. L. and Escudero, L. F., "New Procedures for Preprocessing Zero-One Model with Knapsack-like Constraints and Conjunctive and/or Disjunctive Variable Upper Bounds" IBM T. J. Watson Research Center. RC-14572, York Town Heights(NY), 1989b

27. Eilon, S. and Christofides, N., "The Loading Problem," Management Science, 17, 259-267, 1971

28. Fayard, D. and Plateau, G., "An Algorithm for the Solution of the Zero-One Knapsack Problem," Computing 28, 269-287, 82

29. Fulkerson, D. R., "An Out of Kilter Method for Minimal Cost Flow Problems," Rand Corporation Publication P-1825, 1960

30. Garfinkel, R. S. and Nemhauser, G. L., "Optimal Political Districting by Implicit Enumeration Techniques," Management Science, 16, 8, B495-508, 1970

31. Garfinkel, R. S. and Nemhauser, G. L., "Optimal Set Covering: A Survey," In A.Geoffrion (ed.), Perspectives on Optimization, Addison-Wesley, Reading, Mass.,1972.

32. Garfinkel, R. S. and Nemhauser, G. L., "The Set Partitioning Problem: Set Covering with Equality Constraints," Operations Research, 17, 5, 848-856, 1969.

33. Golden, Bruce L., "Vehicle Routing Problems: Formulations and Heuristic Solution Techniques," Technical Report No. 113, Operations Research Center, M.I.T., August, 1975.

34. Gomory, R. E., "All-Integer Integer Programming Algorithm," IBM Research Report RC-189, January 29, 1960

35. Gomory, R. E., "All-Integer Integer Programming Algorithm," Industrial Scheduling, J. F. Muth and G. Thompson, eds., Addison-Wesley, Reading, Mass., 1963b

36. Gomory, R. E., "An Algorithm for Integer Solutions to Linear Programs," Recent Advances in Mathematical Programming, R. L. Graves and P. Wolfe, eds., Mc-Graw Hill, New York, 1963a

37. Gomory, R. E., "An Algorithm for Intger Solutions to Linear Programs," in Graves and Wolfe (eds.) Recent Advances in Mathematical Programming, McGraw-Hill, Newyork, !963.

38. Gomory, R. E., "An Algorithm for Intger Solutions to Linear Programming," Princeton-IBM Mathematics Research Project Technical Report No. 1, November 17, 1958

39. Haorowitz, E. and Sahni, S., "Computing Partitions with Applications to the Knapsack Problem," Jpurnal of ACM 21, 277-292, 1974

40. Held, M., et al., "Validation of Subgradient Optimization," Maths Programming, 6, pp. 62-88, 1974

41. Held, M., Karp, R. M., "The Travelling Salesman Problem and Minimum Spanning Trees, Part II," Math Programming, pp. 6-25, 1971

42. Hung, M. S. and Brown, J. R., "An Algorithm for a Class of Loading Problems," Naval Research Logistics Quarterly 25, 289-297, 1978

43. Ibarra, O. H. and Kim, C. E., "Fast Approximation Algorithms for the Knapsack and Sum of Subset Problems," Journal of ACM 22, 463-468, 1975

44. Ingargiola, G. P. and Korsh, J. F., "A Reduction Algorithm for Zero-One Single Knapsack Problems," Management Science 20, 460-463, 1973

45. Johnson, D. S., "Approximation Algorithms for Combinatorial Problems," Journal of Computer and System Sciences, 9, 256-278, 1974

46. Kolesar, P. J., "A Branch and Bound Algorithm for the Knapsack Problem," Management Science 13, 723-735, 1967

47. Kolner, T. N., "Some Highlights of a Scheduling Matrix Generator System," United Airlines, Presented at the Sixth AGIFORS Symposium, Sept., 1966

48. Levin, A., "Fleet Routing and Scheduling Problem for Air Transportation System," PhD Dissertation, Massachusetts Institute of Technology, January, 1969

49. Marsten, R. E., "An Algorithm for Large Set Partitioning Problems," Management Science, 20, 5, 774-787, January, 1974.

50. Martello, S. and Toth, P., "A Mixture of Dynamic Programming and Branch-and-Bound for the Subset-Sum Problem," Management Science 30, 765-771, 1984a

51. Martello, S. and Toth, P., "A New Algorithm for the Zero-One Knapsack Problem," Management Science 34, 633-644, 1988

52. Martello, S. and Toth, P., "An Exact Algorithm for the Bin Packing Problem," Presented at EURO X, Beograd, 1989

53. Martello, S. and Toth, P., "An Upper Bound for the Zero-One Knapsack Problem and a Branch and Bound Algoriithm," European Journal of Operational Research 1, 169-175, 1977a

54. Moreland, J. A., "Scheduling of Airline Flight Crews," Master's Thesis, MIT, Department of Aeronautica and Astronotics, 1966

55. Paul, M. C. and Unger, S. H., "Minimizing the Number of States in Incompletely Specified Sequential Functions," IRE Transactions on Electronic Computers, EC-8, 356-367, 1959

56. Pierce, J. F. and Lasky, J. S., "Improved Combinatorial Programming Algorithms for a Class of All-Zero-One Integer Programming Problem," Management Science, 19, 528-543, 1973.

57. Pierce, J. F., "Application of Combinatorial Programming to a Class of All-Zero-One Integer Programming Problems," Management Science, 15, 191-209, 1968

58. Pierce, J. F., "Pattern Sequencing and Matching in Stock Cutting Operations," Tappi, 53, 4, 668-678, April, 1970

59. Plateau, G. and Elkihel, M., "A Hybrid Algorithm for the Zero-One Knapsack Problem, Methods of Operations Research 49, 277-293, 1985

60. Revelle, C. et. al., "An Analysis of Private and Public Sector Location Models," Management Science, 16, 12, 692-707, 1970

61. Root, J. C., "An Application of Symbolic Logic to a Selection Problem," Operations Research, 12, 4, 519-526, 1964

62. Rubin, J., " A Technique for the Solution of Massive Set Covering Problems, with Application to Airline Crew Scheduling," Transportation Science, 7, 1, 34-48, 1973

63. Salveson, M. E., "The Assembly Line Balancing Problem," Journal of Industrial Engineering, 6, 3, 18-25, 1955

64. Spitzer, M., "Solution to the Crew Scheduling Problem," Presented at the First AGIFORS Symposium, October, 1961

65. Thiriez, H., "Airline Crew Scheduling: A group Theoritic Approach," PhD Dissertation, Massachusetts Institute of Technology, October, 1969

66. Valenta, J. R., "Capital Equipment Decisions: A Model for Optimal Systems Interfacing, M. S. Thesis, Massachusetts Institute of Technology, June, 1969

67. Wagner, W. H., "An Application of Integer Programming to Legislative Redistricting," CROND, Inc., Presented at 34th National Meeting of ORSA, Philadelphia, Pennsylvania, November, 1968

APPENDIX - 1

'C' Codes

"ALGO.CPP"

'C' Code for Solving Linear Relaxation of the Set-Partitioning problem

```
//Code for solving the linear relaxation of Set-Partitioning problem.
//Uses two phase method for solving LP
//Columns are generated using knapsack algorithm

#include <iostream.h>
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <time.h>
#include <sys\types.h>
#include <sys\timeb.h>
#include <string.h>
#define max_size 150
#define EXIT 100
#define AGAIN 200
#define INFEASIBLE 300
#define NONOPTIMAL 400
#define GOOD 500
#define NONINTEGER 600

#define MAX_CAP 8.01
#define MIN_FLOAT 0.0001

int nn=0;

void sort_arr();
void heuris_sol();
void read();
void initialize();
int simplex();
int knapsack();
int step1();
int forward_mov();
int backtrack();
int feasible();
int opt();

char filename[20];
char tmpbuf[128], ampm[] = "AM";
// float arr_time[max_size];
// float size[max_size];
float *arr_time;
float *size;

float size1[max_size];
int number;
int index[max_size];
```

```
int sol[max_size];
int ind[max_size];
int pp = 1;
int i,j,k,l,n;
float cap;
float cb_binv[max_size];
int aj[max_size];
float yj[max_size];
float minj[max_size];
int dj[max_size];

int flag = 0;
// int b[max_size][max_size];
// float binv[max_size][max_size];

int   **b;
float **binv;

int c[max_size];
float rhs[max_size];
float org_rhs[max_size];
int x[max_size] = {0,0,0};
float z = 10;
int check;
FILE *ifp1;
FILE *ifp;
FILE *ofp1;
FILE *ofp2;
FILE *ofp3;


      float z_value = 999.9;
      int counter = 0;


void main()
{

//    char name1[30];
//    char name2[30];
//    char name3[30];

//    cout << "\nInput Data file: ";
//    cin >> name1;
//    cout << "\nOutput file (heuris soln): ";
//    cin >> name2;
//    cout << "\nOutput file (Lindo input): ";
//    cin >> name3;

     b    = (int   **) calloc (max_size, sizeof (int));
     binv = (float **) calloc (max_size, sizeof (float *));
```

```
    for (int i =0; i < max_size; i++) {
      b    [i] = (int    *) calloc (max_size, sizeof (int));
      binv [i] = (float *) calloc (max_size, sizeof (float));
    }
    size = (float *)   calloc (max_size,  sizeof (float));
    arr_time = (float *)   calloc (max_size,  sizeof (float));

//        ifp   = fopen (name1,"r");

//        ifp   = fopen ("data.in","r");
//      ofp1 = fopen ("data.out","w");

     ifp1   = fopen ("input.fil","r");
     fscanf(ifp1,"%s",filename);
     ifp = fopen(filename,"r");
     ofp1 = fopen ("trial.out","w");
     ofp2 = fopen ("basis.in","w");
     ofp3 = fopen("post.out","w");
     if (b == NULL) {
       fprintf (ofp1, "b == NULL");
       fclose (ofp1);
       exit (1);
     }
     if (binv == NULL) {
       fprintf (ofp1, "binv == NULL");
       fclose (ofp1);
       exit (1);
     }

  /* print time. */
  _strtime( tmpbuf );
  fprintf( ofp3,"\nBeginning time:\t\t\t\t%s\n", tmpbuf );


     read();
     sort_arr();
     heuris_sol();
     initialize();
     check=AGAIN;

     while(check != EXIT)
          {
          if(check == AGAIN)
          {
          check=knapsack();
          }
          if(check == NONOPTIMAL)
          {
          check=forward_mov();
          }
          if(check == INFEASIBLE)
```

```
            {
            check=backtrack();
            }
            if(check == GOOD)
            {
            check=simplex();
            }

      }
      opt();
//    lindo_form();
//   /* print time. */
 //  _strtime( tmpbuf );
 //  printf( "Ending time:\t\t\t\t%s\n", tmpbuf );
      printf("\7\7Bye!");

fclose(ifp);
fclose(ofp1);
fclose(ofp2);
fclose(ofp3);
 for ( i = 0; i <= max_size; i++) {
  free (b    [i]);
  free (binv [i]);
 }
free (b);
free (binv);
free (size);
free (arr_time);
}

void read()
{
      int j = 1;
      do
      {
            fscanf(ifp,"%f  %f",&arr_time[j], &size[j]);
            number = j;
            j++;
      }
         while (j < max_size - 1 && !feof(ifp));

}

void sort_arr()
{
          float min,temp;
          int j,k;
          for(j = 1; j <= number; j++)
          {
                    min = arr_time[j];
                    for (k = j; k <= number; k++)
```

```
                {
                        if(min >= arr_time[k])
                        {
                                min = arr_time[k];
                                temp = arr_time[j];
                                arr_time[j] = arr_time[k];
                                arr_time[k] = temp;
                                temp = size[j];
                                size[j] = size[k];
                                size[k] = temp;
                        }
                }
        }
        fprintf(ofp1, "No. of Trucks = %d\n", number);
        fprintf(ofp1, "Truck #  \tArrival Time  \tSize\n");

        for (j = 1; j <= number; j++)
        {
                fprintf(ofp1, "%4d  \t\t%f  \t%f\n",j, arr_time[j],size[j]);
       fflush (ofp1);
        }
}

void heuris_sol()
{
        int out = 0;
        int j,m;
        int k = 1;
        int first;
        float add;
        float bal_cap = MAX_CAP;
        float bal_cap1;
        int num_men = 70;
//      int men[max_size][max_size] = {0, 0, 0 };
//      int assign[max_size] = {0};
//      unsigned char *men = (unsigned char *) calloc (max_size * max_size, sizeof (unsigned char));
        unsigned char **men;
//      men = (unsigned char **) calloc (max_size * max_size, sizeof (unsigned char));
        unsigned char *assign = (unsigned char *) calloc (max_size,          sizeof (unsigned char));
 fprintf( ofp1,"\nIn heuristic Sol" );
 fflush (ofp1);
//      if (men == NULL) {
//  fprintf( ofp1,"\nmen == NULL" );
// fflush (ofp1);
//      }
      men = (unsigned char **) calloc (max_size, sizeof (unsigned char *));
      for (int i = 0;  i < max_size; i++)
         men [i] = (unsigned char *) calloc (max_size, sizeof (unsigned char ));
/*
      if (**men == NULL) {
  fprintf( ofp1,"\n**men == NULL" );
```

```
fflush (ofp1);
        }
*/

      if (men == NULL || men [i - 1] == NULL) {
 fprintf( ofp1,"\n*men == NULL" );
fflush (ofp1);
        }
        if (assign == NULL) {
 fprintf( ofp1,"\nassign == NULL" );
fflush (ofp1);
        }

            for(m = 1; m <= num_men; m++)
            {
//     fprintf( ofp1,"\nm %d", m );
 fflush (ofp1);
                        first = 0; k = 1; bal_cap = MAX_CAP, out = 0;
                        for(j = 1; j <= number; j++)
                        {
//        fprintf( ofp1,"\n\tj %d", j );
// fflush (ofp1);
                                if(assign[j] == 0)
                                {
//        fprintf( ofp1,"\n\t\t assign [%d] = %d", j, assign [j] );
 fflush (ofp1);

                                        out = 1;
                                        if(first == 0)
                                        {
                                                add = arr_time[j];
                                                first = 1;
                                                bal_cap -= add;
                                        }
                                        bal_cap1 = bal_cap - size[j];

                                        if(bal_cap1 >= 0)
                                        {
                                                bal_cap = bal_cap1;
                                                men[m][k] = j;
                                                k++;
                                                assign[j] = 1;
//        fprintf( ofp1,"\n\t\t men [%d][%d] = %d", m,k-1,men [m][k-1] );
//  fprintf (ofp1, "\nHELP?");
 fflush (ofp1);
                                        }
                                }
                        }
                        if(out == 0) break;
            }
```

```
        num_men = m - 1;

        fprintf(ofp1, "\nTotal men required: %4d",num_men);
        fprintf(ofp1, "\nMan     Truck # allotted");
fflush (ofp1);

        for(j = 1; j <= num_men; j++)
        {
                fprintf(ofp1, "\n%3d", j);
                for(int k1 = 1; k1 < number; k1++)
                {
                        if(men[j][k1] != 0)
                        {
                                fprintf(ofp1, "%5d", men[j][k1]);
                        }
                }
        }
    for ( i = 0;  i < max_size; i++)
      free (men [i]);
    free (men);
    free (assign);
}

void initialize()
{
        int j,k;

        for(j = 1; j <= number; j++)
                {
                for (k = 1; k <= number; k++)
                        {
                        if( k != j)
                                {
                                b[j][k] = 0;
                                binv[j][k] = 0.0;
                                }
                        else
                                {
                                b[j][k] = 1;
                                binv[j][k] = 1.0;
                                }
                        }
                        c[j] = 1;
                        org_rhs[j] = 1.0;
                        index[j] = j;
                        rhs[j] = 1.0;
                }
}

int knapsack()
{
```

```
        float temp;
//      float contrib[max_size];
        float *contrib = (float *) calloc (max_size, sizeof (float));
        int j,k;
        float sum = 0.0;
        int q;
        float iw;
    if (contrib == NULL) {
      fprintf (ofp1, "contrib == NULL\n");
      fclose (ofp1);
      exit (1);
    }
        for(i = 1; i <= number; i++)          // calculating Cb*binv
        {
                sum = 0.0;
                for(k = 1; k <= number; k++)
                {
                        sum = sum +  c[k] * binv[k][i];
                }
                cb_binv[i] = sum;
        }

        for(j = 1; j <= number; j++)
                {
                size1[j] = size[j];
                }

        for(k = 1; k <= number; k++)  //finding contribution pj/wj
                {
                if(fabs(cb_binv[k]) <= 0.00001)
                        contrib[k] = 0.0;
                else
                        contrib[k] = cb_binv[k] / size[k];
                }


        for(k = 1; k <= number; k++)  //Sorting items in decreasing order of pj/wj
                {
                for(j = 1; j <= number-1; j++)
                        {
                        if(contrib[j] <= contrib[j + 1])
                                {
                                temp = contrib[j];
                                contrib[j] = contrib[j+1];
                                contrib[j+1] = temp;
                                temp = size1[j];
                                size1[j] = size1[j+1];
                                size1[j+1] = temp;
                                temp = index[j];
                                index[j] = index[j+1];
                                index[j+1] = (int)temp;
```

```
                                        }
                                }
                        }

        sum = 0.0;
        cap = MAX_CAP;
        for(k =1; k <= number; k++)
        {
                if(sum <= cap)
                {
                q = k;
                sum = sum + size1[k];
                }
        }
        l = q - 1;

        iw = 0.0;
        for(j = 1; j <= l; j++)
                {
                iw = iw + size1[j];
                }

        cap = cap - iw;

        for(j = 1; j <= number; j++)    // initializing arrays to zero
                {
                dj[j] = 0;
                sol[j] = 0;
                ind[j] = 0;
                }
        pp = 1;

        for(j = 1; j <= l; j++)
                {
                dj[j] = 1;
                sol[pp] = j;
                ind[pp] = 1;
                pp++;
                }
        for(j = l+1; j <= number; j++)
                {
                dj[j] = 0;
                }

        l = l + 2;

        check = feasible();
                {
free (contrib);
                return(check);
                }
```

```
free (contrib);

fprintf(ofp1,"**********Stepping out of step 1**********");

return(1000);
}


int forward_mov()                    //Forward move
{

        for(i = 1; i <= number; i++)
                {
                if(cap + 0.0001 >= size1[i])
                        {
                        dj[i] = 1;
                        ind[pp] = 1;
                        cap = cap - size1[i];
                        check = feasible();

                if(check == EXIT)
                    return(EXIT);

                        if(check == INFEASIBLE)
                                {
                                dj[i] = 0;
                                cap = cap + size1[i];
                                sol[pp] = -i;
                                ind[pp] = 2;
                                ++pp;
                                }

                        if(check == NONOPTIMAL)
                                {
                                sol[pp] = i;
                                ind[pp] = 1;
                                ++pp;
                                }

                        if(i >= number)
                                {
                                return(INFEASIBLE);
                                }
                        if(check == GOOD)
                                {
                                return(check);
                                }
                        }

                else
```

```
                    {
                    return(INFEASIBLE);
                    }
                if(i >= number )
                    {
                    return(INFEASIBLE);
                    }

                }
fprintf(ofp1,"\nThe search is over at end of step2\n");
return(EXIT);
}

int backtrack()                 // Backtrack
{
        int kk = 1;
        int stupid;
        while(kk == 1)
                {
                kk = 0;
                for(i = pp - 1; i >=1; i--)
                        {
                        if(ind[i] == 1)
                                {
                                break;
                                }
                        }

                if(i == 0)
                        {
                        fprintf(ofp1,"\nThe search is over in step 3\n");
                        return(EXIT);
                        }
                stupid = pp;
                ind[i] = 2;
                sol[i] = - sol[i];
                pp = i + 1;
                l = abs(sol[pp-1]) + 1;
                dj[l - 1] = 0;
                cap = cap + size1[l - 1];
                check = feasible();

          if(check == EXIT)
            return(EXIT);

                if(check == INFEASIBLE)
                        {
                        kk=1;
                        continue;
                        }
                if(check == NONOPTIMAL)
```

```
                    {
                    if(stupid > number)
                            {
                            for(i = pp - 1; i >=1; i--)
                                    {
                                    if(ind[i] == 1)
                                            {
                                            break;
                                            }
                                    }
                            if(i == 0)
                                    {
                                    fprintf(ofp1,"\nThe search is over in step 3 at 2nd point\n");
                                    return(EXIT);
                                    }

                            stupid = pp;
                            ind[i] = 2;
                            sol[i] = - sol[i];
                            pp = i + 1;
                            l = abs(sol[pp-1]) + 1;
                            dj[l - 1] = 0;
                            cap = cap + size1[l - 1];

                            }
                    return(check);
                    }
            if(check == GOOD)
                    {
                    return(check);
                    }
            }
fprintf(ofp1,"**********Stepping out of step 3**********");
return(1000);
}

int feasible()
{
    int j,k;
    float temp;
    float sum,sum1;
//  int index1[max_size];
    int *index1 = (int *) calloc (max_size, sizeof (int));
    int sum2,test;
    nn++;
    if(nn >=500)                    //exiting if after 500 columns none is feasible
        {
         fprintf(ofp1,"\nI think no more colns can be generated\n");
        }
      if (index == NULL)
      {
```

```
  fprintf (ofp1, "index == NULL");
  fclose (ofp1);
  exit (1);
}


for(j = 1; j <= number; j++)
        {
        aj[j] = dj[j];
        index1[j] = index[j];
        }

for(k = 1; k <= number-1; k++)  //Sorting items in increasing
        {                       //order of index
        for(j = 1; j <= number-1; j++)
                {
                if(index1[j] > index1[j+1])
                        {
                        temp = aj[j];
                        aj[j] = aj[j+1];
                        aj[j+1] = (int)temp;
                        temp = index1[j];
                        index1[j] = index1[j+1];
                        index1[j+1] = (int)temp;
                        }
                }
        }
free (index1);
for(k = number; k >= 1; k--)  //Stop the program if search reached a point after which
        {                     // all cb_binv are zeros
        if(cb_binv >= 0)
                {
                break;
                }
        }
test = k;
sum2 = 0;
for(k = 1; k <= test; k++)
        {
        sum2 += dj[k];
        }

sum = 0.0;              // exiting the program when all dj are zeros
for(k = 1; k <= number; k++)
        {
        sum += dj[k];
        }

if(sum2== 0 || sum == 0.0||nn>=500)
        {
        fprintf(ofp1,"\n**The final basis is**\n");          // print basis
```

```
                for(j = 1; j <= number; j++)
                        {
                        for(k = 1; k <= number; k++)
                                {
                                fprintf(ofp1,"%d ",b[j][k]);
                                }
                        fprintf(ofp1,"\n");
                        }
                fprintf(ofp1,"\n");
                fprintf(ofp1,"\n**The final Solution is**\n");        // print final rhs
                for(k = 1; k <= number; k++)
                        {
                        fprintf(ofp1,"%3.2f ",rhs[k]);
                        }
                fprintf(ofp1,"\n");
                fprintf(ofp1,"\n\nThe Z value is  %3.2f\n",z);
//              printf("\nAll dj are zeros\n");
                fprintf(ofp1,"\nIs it really over\n");
//              lindo_form();

           /* print time. */
           _strtime( tmpbuf );
           fprintf( ofp1,"\nEnding time:\t\t\t\t%s\n", tmpbuf );

                return(EXIT);
                }

        sum = 0.0;
        for(j = 1; j<= number; j++)
                {
                if(aj[j] == 1)
                        {
                        if(arr_time[j] >= sum)
                                {
                                sum = arr_time[j] + size[j];
                                }
                        else
                                {
                                sum += size[j];
                                }
                        }
                }

        if(sum <= MAX_CAP)                              // Checking the feasibility
                {
                sum1 = 0.0;
                for(j = 1; j <= number; j++)  // Calculating cb*binv*aj
                        {
                        sum1 += cb_binv[j] * aj[j];
                        }
```

```
                if(flag == 0)
                        {
                        if(sum1 > 0.01)
                                {
                                return(GOOD);
                                }
                        else
                                {
                                return(NONOPTIMAL);
                                }
                        }
                if(flag == 1)           //phase 2 of simplex
                        {
                        if(sum1 > 1.01)
                                {
                                return(GOOD);
                                }
                        else
                                {
                                return(NONOPTIMAL);
                                }
                        }
                }
        else
                {
                return(INFEASIBLE);
                }
return(1000);
}

int opt()
{
        fprintf(ofp1,"\n\nThe Final Basis from opt function is\n");
        for(j = 1; j <= number; j++)
        {
                for(k = 1; k <= number; k++)
                {

                        fprintf(ofp1,"%d ",b[j][k]);
                }
                fprintf(ofp1,"\n");
        }

        for(j = 1; j <= number; j++)
        {
                for(k = 1; k <= number; k++)
                {

                        fprintf(ofp2,"%d ",b[j][k]);
                }
                fprintf(ofp2,"\n");
```

```
        }

        fprintf(ofp1,"\n\nFinal solution from opt function is is\n");
        for(j = 1; j <= number; j++)
        {
                fprintf(ofp1,"%3.2f ",rhs[j]);
        }

        fprintf(ofp1,"\n\nThe z value from opt function is   %f\n",z);
        fflush (ofp1);
//      fprintf(ofp1,"\nThe final X is\n");
//      for(j = 1; j <= number; j++)
//              fprintf(ofp1,"%d  ",x[j]);
}

int simplex()
{
        nn = 0;
        float sum, sum1,sum2,sum3, min;
//      float eta[max_size];
//      float temp[max_size][max_size];
//      float ratio[max_size];
        int elig_row=0;
        int k,j,m;
//      float binv1[max_size][max_size]={0.0,0.0,0.0};
//      int index1[max_size];
        float *eta    = (float *) calloc (max_size, sizeof (float));
        float **temp  = (float **) calloc (max_size, sizeof (float *));
        float *ratio  = (float *) calloc (max_size, sizeof (float));
        float **binv1  = (float **) calloc (max_size, sizeof (float *));
     int *index1 = (int   *) calloc (max_size, sizeof (int));


     for (int i = 0; i < max_size; i++) {
       binv1 [i] = (float *) calloc (max_size, sizeof (float));
      temp [i]  = (float *) calloc (max_size, sizeof (float));
     }
        int temp1;
    if (temp == NULL || temp [i -1] == NULL) {
       fprintf (ofp1, "temp == NULL");
       fclose (ofp1);
       exit (1);
    }
    if (binv1 == NULL || binv1 [i - 1] == NULL) {
       fprintf (ofp1, "binv1 == NULL");
       fclose (ofp1);
       exit (1);
    }


        for(j = 1; j <= number; j++)
```

```
                {
                aj[j] = dj[j];
                index1[j] = index[j];
                }
                for(k = 1; k <= number-1; k++)  //Sorting aj in increasing
                {                               //order of index
                for(j = 1; j <= number-1; j++)
                        {
                        if(index1[j] > index1[j+1])
                                {
                                temp1 = aj[j];
                                aj[j] = aj[j+1];
                                aj[j+1] = temp1;
                                temp1 = index1[j];
                                index1[j] = index1[j+1];
                                index1[j+1] = temp1;
                                }
                        }
                }

/*      fprintf(ofp1,"\nThe aj is\n");
        for(k = 1; k <= number; k++)        // printing aj
    {
    fprintf(ofp1,"%d ",aj[k]);
    }
    fprintf(ofp1,"\n");
*/

        for(k = 1; k <= number; k++)        // update entering column
                {                           // forming yj
                sum = 0;
                for(j = 1; j <= number; j++)
                        {
                        sum += binv[k][j] * aj[j];
                        }
                yj[k] = sum;
                }

        sum1 = 0.0;
        for(j = 1; j <= number; j++)  // Calculating cb*binv*aj
                {
                sum1 += c[j] * yj[j];
                }

        for(k = 1; k <= number; k++)            // Find  Ratios
                {
          if(yj[k] > MIN_FLOAT && rhs[k] > MIN_FLOAT)
                        {
                        ratio[k] = rhs[k] / yj[k];
                        }
```

```
            else if(yj[k] <= MIN_FLOAT )
                        {
                        ratio[k] = -1.0;
                        }

            else if(rhs[k] <= MIN_FLOAT)          //rhs[k] = 0
                {
                 ratio[k] = 0.0;
                }

           else{

fprintf(ofp1,"The rhs is %f and the yj is %f and ratio is %f",rhs[k],yj[k],ratio[k]);
fprintf(ofp1,"\7\7Good Bye....3,  Fix me, I am in simplex in ratios!");
exit(0);
                        }
                }

//fprintf(ofp3,".");
   min=9999.9;                                  // Initializing to large value
        for(k = 1; k <= number; k++)   // Find (minimum ratio) departing row
                {
                if(ratio[k] >= 0.0 && min > ratio[k])
                        {
                        min = ratio[k];
                        elig_row = k;
                        }
                }

        if(fabs(z) <= 0.00001 && flag == 0)// forcing the artificial variables out
                {
                for(j = 1; j <= number; j++)
                        {
                        if(x[j] == 0)
                                {
                                elig_row = j;
                                break;
                                }
                        }
                }

        for(k = 1; k <= number; k++)          //  update b
                {
                b[k][elig_row] = aj[k];
                }

/*      fprintf(ofp1,"\n**The final basis is**\n");          // print basis
          for(j = 1; j <= number; j++)
                {
                for(k = 1; k <= number; k++)
                        {
```

```
                    fprintf(ofp1,"%d ",b[j][k]);
                    }
            fprintf(ofp1,"\n");
            }
*/
    for( k = 1; k <= number; k++)                   // form eta
            {
            if(fabs(yj[k]) <= 0.00001)
                    {
                    eta[k] = 0;
                    }
            else
                    {
                    if(k != elig_row)
                            {
                            eta[k] = -1 * yj[k] / yj[elig_row];
                            }
                    else
                            {
                            eta[k] = 1 / yj[elig_row];
                            }
                    }
            }

    x[elig_row] = 1;                // record of variables entered

    for(k = 1; k <= number; k++)            //  form temp
            {                               //  initialize to 0,1
      for (j = 1; j <= number; j++)
               {
                    if(j == k)
                            {
                            temp[k][j] = 1.0;
                            }
                    else
                            {
                            temp[k][j] = 0.0;
                            }
                    }
            }

    for(k = 1; k <= number; k++)            //  form temp
            {                               // replace the column with eta
            temp[k][elig_row] = eta[k];
            }

    for(m = 1; m <= number; m++)    // multiply temp_binv[] by binv[]
            {                               // to get new 'binv'
            for(k = 1; k <= number; k++)
                    {
                    sum2 = 0.0;
```

```
                        for(j = 1; j <= number; j++)
                                {
                                sum2 += temp[m][j] * binv[j][k];
                                }
                        binv1[m][k] = sum2;
                        }
                }

        for(j = 1; j <= number; j++)
                {
                for(k = 1; k <= number; k++)
                        {
                        binv[j][k] = binv1[j][k];
                        }
                }

//      if(flag == 1 && z_value < z && z>6)

          for(j = 1; j <= number; j++)
         {
           if(rhs[j] < -0.009)

         {
//         counter++;
//         fprintf(ofp1,"\n\nThe z value increased previous one. New value is    %f\n",z);

fprintf(ofp1,"\n");

          for(j = 1; j <= number; j++)
                {
                for(k = 1; k <= number; k++)
                        {
                        fprintf(ofp1,"%d ",b[j][k]);
                        }
                        fprintf(ofp1,"\n");
           }
         fprintf(ofp1,"\n\nThe RHS is\n");
           for(j = 1; j <= number; j++)
         {
         fprintf(ofp1,"%3.2f  ",rhs[j]);
         }

         fprintf(ofp1,"\n\nThe yj is\n");
           for(j = 1; j <= number; j++)
         {
         fprintf(ofp1,"%3.2f  ",yj[j]);
         }

         fprintf(ofp1,"\n\nThe ratios are\n");
           for(j = 1; j <= number; j++)
         {
```

```
            fprintf(ofp1,"%3.2f  ",ratio[j]);
            }

            fprintf(ofp1,"\nThe eligible row is   %d",elig_row);

    /* print time. */
    _strtime( tmpbuf );
    fprintf( ofp1,"\nExiting time:\t\t\t\t%s\n", tmpbuf );

//                  exit(0);
            return(EXIT);
            }}
//          else if(flag == 1)
//            z_value = z;

          for(k = 1; k <= number; k++)        // update rhs
                  {
                  sum1 = 0.0;
                  for(j = 1; j <= number; j++)
                          {
                          sum1 += binv[k][j] * org_rhs[j];
                          }
                  rhs[k] = sum1;
                  }

          if(flag == 0)       // update c[b]
                  {
                  c[elig_row] = 0;
                  }

          sum3 = 0.0;             // Calculate Cb*binv*b (binv * b = rhs)
          for(k = 1; k <= number; k++)
                  {
                  sum3 += c[k] * rhs[k];
                  }
          z = sum3;
/*
//        if(flag == 1 && z_value < z && z>6)

             for(j = 1; j <= number; j++)
            {
              if(rhs[j] < -0.001)

            {
//            counter++;
//            fprintf(ofp1,"\n\nThe z value increased previous one. New value is    %f\n",z);

              for(j = 1; j <= number; j++)
                      {
                      for(k = 1; k <= number; k++)
                              {
```

```
                        fprintf(ofp1,"%d ",b[j][k]);
                        }
                        fprintf(ofp1,"\n");
                    }
                fprintf(ofp1,"\n\nThe RHS is\n");
                  for(j = 1; j <= number; j++)
                {
                fprintf(ofp1,"%3.2f  ",rhs[j]);
                }

               fprintf(ofp1,"\n\nThe yj is\n");
                  for(j = 1; j <= number; j++)
                {
                fprintf(ofp1,"%3.2f  ",yj[j]);
                }

               fprintf(ofp1,"\n\nThe ratios are\n");
                  for(j = 1; j <= number; j++)
                {
                fprintf(ofp1,"%3.2f  ",ratio[j]);
                }

                fprintf(ofp1,"\nThe eligigible row is   %d",elig_row);

                        exit(0);
                }}
//              else if(flag == 1)
//                z_value = z;


                  for(j = 1; j <= number; j++)
                {
                  if(rhs[j] < -0.009)
                    {
                     fprintf(ofp1,"\n The rhs turned negative, it is   %f", rhs[j]);
                    exit(0);
                    }
               }
*/
//      if(flag == 1)
//        fprintf(ofp1,"\n\nThe z value is  %f\n",z);

          if(fabs(z) <= MIN_FLOAT)          //resetting flag for phase II
                    {
                    sum = 0.0;
                    for(j = 1; j <= number; j++)
                            {
                            sum += x[j];
                            }
                    if(sum == number)
                            {
```

```
                fprintf(ofp1,"\n\n**The basis at the end of Phase I is**\n");//  print basis
                for(j = 1; j <= number; j++)
                        {
                        for(k = 1; k <= number; k++)
                                {
                                fprintf(ofp1,"%d ",b[j][k]);
                                }
                        fprintf(ofp1,"\n");
                        }
                fprintf(ofp1,"\n");

        fprintf(ofp1,"\n**The Solution at end of phase I is**\n");// print final rhs
                for(k = 1; k <= number; k++)
                        {
                        fprintf(ofp1,"%3.2f ",rhs[k]);
                        }
                fprintf(ofp1,"\n");

sum1 = 0.0;
                for(k = 1; k <= number; k++)
                        {
        sum1 += rhs[k];
                        }
                fprintf(ofp1,"\n\nThe Z value at end of Phase I is  %3.2f\n",sum1);

            /* print time. */
            _strtime( tmpbuf );
            printf( "Time at the end of phase I is:\t\t%s\n", tmpbuf );

                fprintf(ofp1,"\n**********In Phase 2***********\n\n");
                flag = 1;
                }
          }

 if(flag == 1)              //resetting cb for phase II
          {
          for(j = 1; j <= number; j++)
                {
                c[j] = 1;
                }

          sum3 = 0.0;           // Calculate z = Cb*binv*b (binv * b = rhs)
          for(k = 1; k <= number; k++)
                {
                sum3 += c[k] * rhs[k];
                }
          z = sum3;
          }

 for(j = 1; j <= number; j++)    // resetting index in seq. order
          {
```

```
                    index[j] = j;
                    }
        for ( i = 0; i < max_size; i++) {
          free (binv1 [i]);
         free (temp [i]);
        }
        free (eta);
        free (temp);
        free (ratio);
        free (binv1);
        free (index1);

            return(AGAIN);
}
```

"CPLEX.CPP"

'C' Code for formulating Set-Covering problem from the optimal basis

```
#include <iostream.h>
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
// #include <time.h>
// #include <sys\types.h>
// #include <sys\timeb.h>
#include <string.h>
#define max_size 100
int number;
void cplx_form();
int b[max_size][max_size];
FILE *ifp;
FILE *ofp;

void main()
{
//        char name1[30];
//        char name2[30];

//        cout << "\nInput Data file: ";
//        cin >> name1;
//        cout << "\n\tOutput file : ";
//        cin >> name2;

          ifp = fopen("basis.in","r");
          ofp = fopen("output.lp","w");

          cplx_form();
fclose(ifp);
fclose(ofp);

}


void cplx_form()
{
          float n;
          int i,j;
      char buf [512];

      for (j = 1; fgets (buf, sizeof (buf), ifp) != NULL; j++) {
        for (i = 1; sscanf (&buf [(i -1) << 1], " %d", &b [j][i]) == 1; i++);
      }
    number = j-1;
```

```
fprintf(ofp,"min\n ");          // printing the objective
        for(j = 1; j<= number-1; j++)
                {
                fprintf(ofp,"x%d + ",j);
                n = (float)j/10;
                if(n == (int)n)
                        {
                        fprintf(ofp,"\n");
                        }
                }
        fprintf(ofp,"x%d\n",number);

        fprintf(ofp,"st\n");            // printing Constraints
        for(j = 1; j<= number; j++)
                {
                for(i = 1; i <= number-1; i++)
                        {
                        fprintf(ofp,"%dx%d + ",b[j][i],i);
                        n = (float)i/10;
                        if(n == (int)n)
                                {
                                fprintf(ofp,"\n");
                                }
                        }
                fprintf(ofp,"%dx%d >= 1\n",b[j][number],number);
                }

        fprintf(ofp,"integers\n");    // printing variable bounds

        for(j = 1; j<= number; j++)
    {
        fprintf(ofp,"x%d ",j);    // printing variable bounds
                        n = (float)j/10;
                        if(n == (int)n)
                                {
                                fprintf(ofp,"\n");
                                }

    }
        fprintf(ofp,"\nend\n");

}
```

"POST.CPP"

'C' Code for reading the *CPLEX* output and printing the final solution

```
#include <stdlib.h>
#include <string.h>
#include <stdio.h>
#define max_size 100
#include <time.h>
#include <sys\types.h>
#include <sys\timeb.h>

#define max_cap 8.01

char tmpbuf[128], ampm[] = "AM";

//int number1 = 25;
int number;
//int b[max_size][max_size];
int **b;

FILE *ifp;
FILE *ifp1;
FILE *ifp2;
FILE *stream, *s2;
#define  MAX_LEN  100
//int man[max_size][max_size] = {0, 0, 0};
int **man;

char filename[20];
float arr_time[max_size];
float size[max_size];

void sort_arr();
void heuris_sol();
void read();
void our_sol();

int main(void)
{
     man = (int**) calloc (max_size, sizeof (int *));
    b    = (int **)  calloc (max_size, sizeof (int *));
    for (int i = 0; i <  max_size; i++) {
      b      [i] = (int *)  calloc (max_size, sizeof (int));
       man  [i] = (int*)  calloc(max_size, sizeof (int));
   }
    s2 = fopen("post.out","a");

         ifp1 = fopen ("input.fil","r");
     fscanf(ifp1,"%s",filename);
```

```
        ifp = fopen(filename,"r");
            read();
            sort_arr();
//      heuris_sol();

        stream = fopen("output.out","rb");

        our_sol();
        heuris_sol();


    /* print time. */
    _strtime( tmpbuf );
    fprintf(s2, "\n\nEnding time:\t\t\t\t%s\n", tmpbuf );


    fclose(ifp);

    if (fclose(stream))
       perror("fclose error");
    if (fclose(s2))
       perror("fclose error");


}

void our_sol()
{


        char line[MAX_LEN+2], *result;
        char x1 [6];
        int j=0;
        int n,i,k,l,p;
        int m = 0;
        float sum[max_size];
        int    n1[max_size];
        float tot_hours = 0;

//      char *str1 = "NAME";
        char *str2 = "Variable Name";
        int numresult;


/******Locating the string*******/

    while ((result = fgets(line,MAX_LEN,stream)) != NULL) {
//          fprintf(s2, "The string is %s\n", result);
        if (! strnicmp(line, str2, strlen (str2)))
          break;
```

```
  }


/******Reading the string*******/
/*
  do
   {
   j++;
   (fscanf(stream,"%s       %d",x1, &n1[j]) == 2);
//   number = j;

   }while(!feof(stream));
*/
// char tmp [MAX_LEN+2];
// fgets(tmp,MAX_LEN,stream);
// fgetc (stream);
  int t1;
  float t2;
  char c;
   memset (n1, 0, sizeof (n1));
   do
   {
   if (fscanf(stream,"%c%d  %f",&c, &t1, &t2) == 3 && c == 'x') {
     j++;
     n1 [t1] = (int) t2;
//       fprintf(s2,"\nGood - %d.  %d",t1,n1[t1]);
 } // else
//       fprintf(s2,"\nNot %c- %d.  %d",c,t1,(int) t2);

//   number = j;

   }while(!feof(stream));
// for(i = 1; i <=number;i++)
//   fprintf(s2, "The N%d = %d\n",i,n1[i]);

//     fprintf (s2, "\n%s", line);
//     fprintf (s2, "\n%s",tmp);
//      for(j = 1;j <=25;j++)
//       fprintf(s2,"\n%d.  %d",j,n1[j]);


/*****Reading the basis******/

          ifp2 = fopen("basis.in","r");
          for(j = 1; j<= number; j++)          //reading basis
                    {
                    for(i = 1; i <= number; i++)
                              {
                              fscanf(ifp2,"%d ",&b[j][i]);
                              }
```

```
                    fscanf(ifp2,"&\n");
                     }
 //for(j = 1; j <= number - 1; j++)
 //   fprintf(s2, "The N%d = %d\n", j,n1[j]);

 //    fprintf(s2, "The Number is = %d\n", number-1);


/*
          for(j = 1; j<= number; j++)          //printing  basis for testing
                    {
                    for(i = 1; i <= number; i++)
                              {
                              fprintf(s2,"%d ",b[j][i]);
                              }
                    fprintf(s2,"\n");
                     }
*/

int max_num = 0;
      for(j = 1; j <= number; j++)
        {
          if(n1[j] == 1)
            {
              m++;
              n=0;
              for(i = 1; i <= number; i++)
                {
                  if(b[i][j] == 1)
                    {
                      n++;
                      man[m][n] = i;
                    }
                 if(max_num < n)         //keeping record of max trucks
                   {                     //allotted to a worker
                     max_num = n;
                   }
                 }
            }
       }


/*        for(i = 1; i <= m; i++)
          {
                    fprintf(s2, "\n%3d", i);
                    for(j = 1; j <= max_num; j++)
                    {
                              if(man[i][j] != 0)
                              {
                                        fprintf(s2, "   %3d", man[i][j]);
                              }
```

```
                }
        }
*/

        for(i = 1; i <= m; i++)              // deleting repeated trucks
      {
      for(j = 1; j <= max_num; j++)
        {
        for(k = i+1; k <= m; k++)
          {
          for(l = 1; l <= max_num; l++)
            {
              if(man[i][j] == man[k][l])
                {
                  man[k][l] = 999;
                }
            }
          }
        }
       }

        for(i = 1; i <= m; i++)
      {
                sum[i] = 0.0;
        for(j = 1; j <= max_num; j++)
                {
                        if(man[i][j] != 0)
                        {
                        p = man[i][j];
                        if(arr_time[p] >= sum[i])
                                {
                    sum[i] = arr_time[p] + size[p];
                                }
                        else
                                {
                                sum[i] += size[p];
                                }
                        }
        }
      }
/*
        for(i = 1; i <= m; i++) //subtracting arr time of first truck from finish time of last
      {
        p = man[i][1];
         sum[i] = sum[i] - arr_time[p];
         if(sum[i] < 4.0)
           {
             sum[i] = 4.0;
           }
   }
*/
```

```
        for(i = 1; i <= m; i++)
          {
          tot_hours += sum[i];
          }

        fprintf(s2, "\nMan\tTruck # allotted        Hours required");
        for(i = 1; i <= m; i++)                          //printing final assignment
        {
         int cnt = 0;
                fprintf(s2, "\n%3d   ", i);
                for(j = 1; j <= max_num; j++)
                {
                        if ( man[i][j] != 0)
                 if ( man[i][j] != 999)
                        fprintf(s2, " %3d", man[i][j]);
                 else
                    cnt++;
               else
                        fprintf(s2, "    ");
                }
          for (j = 0;  j < cnt ; j++ )
            fprintf (s2, "    ");
          fprintf(s2,"      \t%5.2f", sum[i]);
        }

    fprintf(s2,"\n\nTotal hours required  =  %5.2f\n\n",tot_hours);
/*
        fprintf(s2, "\nMan\tTruck # allotted\t\t\t\t\tHours required");
        for(i = 1; i <= m; i++)                          //printing final assignment
        {
                fprintf(s2, "\n%3d", i);
                for(j = 1; j <= max_num; j++)
                {
                        if(man[i][j] != 0)
                        {
                                fprintf(s2, "   %.3d", man[i][j] != 0 ? man[i][j] : 0);
                        }
                }
          fprintf(s2,"\t\t\t\t\t%5.2f", sum[i]);
        }

 */
 }

void read()
{
        int j = 1;
        do
        {
                fscanf(ifp,"%f  %f",&arr_time[j], &size[j]);
                number = j;
```

```
                j++;
        }
        while (!feof(ifp));

}

void sort_arr()
{
        float min,temp;
        int j,k;
        for(j = 1; j <= number; j++)
        {
                min = arr_time[j];
                for (k = j; k <= number; k++)
                {
                        if(min >= arr_time[k])
                        {
                                min = arr_time[k];
                                temp = arr_time[j];
                                arr_time[j] = arr_time[k];
                                arr_time[k] = temp;
                                temp = size[j];
                                size[j] = size[k];
                                size[k] = temp;
                        }
                }
        }
        fprintf(s2, "No. of Trucks = %d\n", number);
        fprintf(s2, "Truck #  \tArrival Time  \tSize\n");

        for (j = 1; j <= number; j++)
        {
                fprintf(s2, "%4d  \t%5.2f  \t%5.2f\n",j, arr_time[j],size[j]);
        }
}

void heuris_sol()
{

        fprintf(s2, "\n\n*******Entered the final function----------> 1 ******");

        int out = 0;
        int j,l;
        int k = 1;
        int first;
        float add;
        float bal_cap = max_cap;
        float bal_cap1;
        int num_men = 50;
//      int men[max_size][max_size] = {0, 0, 0 };
        int assign[max_size] = {0};
```

```
float sum[max_size] = {0.0};
int p;

int **men = (int**) calloc (max_size, sizeof (int *));

for ( j = 0; j < max_size; j++) {
   men [j] = (int *) calloc (max_size, sizeof (int));
}
//      fprintf(s2, "\n\n*******Entered the final function------> 2  ******");

      for(l = 1; l <= num_men; l++)
      {
            first = 0; k = 1; bal_cap = max_cap, out = 0;
            for(j = 1; j <= number; j++)
            {
                  if(assign[j] == 0)
                  {
                        out = 1;
                        if(first == 0)
                        {
                              add = arr_time[j];
                              first = 1;
                              bal_cap -= add;
                        }
                        bal_cap1 = bal_cap - size[j];

                        if(bal_cap1 >= 0)
                        {
                              bal_cap = bal_cap1;
                              men[l][k] = j;
                              k++;
                              assign[j] = 1;
                        }
                  }
            }
            if(out == 0) break;
      }

      num_men = l - 1;


      for(int i = 1; i <= num_men; i++)
     {
            sum[i] = 0.0;
        for(j = 1; j <= number; j++)
            {
                  if(men[i][j] != 0)
                  {
                  p = men[i][j];
                  if(arr_time[p] >= sum[i])
```

```
                                {
                        sum[i] = arr_time[p] + size[p];
                                }
                            else
                                {
                                sum[i] += size[p];
                                }
                            }
                    }
                }

/*
                for(i = 1; i <=num_men; i++)  //subtracting arr time of first truck from finish time of last
                {
//                  fprintf(s2,"\nThe men[%d][1] is %d",i,men[i][1]);
                  p = men[i][1];
                  sum[i] = sum[i] - arr_time[p];
                    if(sum[i] < 4.0)
                      {
                        sum[i] = 4.0;
                       }
            }
*/
            float tot_hours = 0;
                for(i = 1; i <= num_men; i++)
                  {
                  tot_hours += sum[i];
                  }


            fprintf(s2,"\n\n\n********The Heuristic Solution is***********");
                fprintf(s2, "\nTotal men required: %4d",num_men);
                fprintf(s2, "\nMan   Truck # allotted          Hours Required");

                for(j = 1; j <= num_men; j++)
                {
                        fprintf(s2, "\n%3d", j);
                        for(int k1 = 1; k1 < number; k1++)
                        {
                                if(men[j][k1] != 0)
                                {
                                        fprintf(s2, " %3d", men[j][k1]);
                                }
                      else
                        fprintf(s2,"  ");
                        }
                  fprintf(s2,"     %5.2f",sum[j]);
                }

            fprintf(s2,"\n\nTotal Hours Required  =  %5.2f",tot_hours);
```

```
        float strip_time=0.0;
        for(j = 1; j <= number; j++)
                {
        strip_time += size[j];
        }

    fprintf(s2,"\n\n***** The total strip time is**********");
    fprintf(s2,"\n Strip time is   %5.2f\n",strip_time);
}
```

APPENDIX - 2

Data Sets

Roanoke Hub (10/03/94)

Truck #	Arrival Time	Operating Time
1.	0.00	0.75
2.	0.00	0.75
3.	0.00	1.17
4.	0.00	1.42
5.	0.00	1.75
6.	0.00	1.08
7.	0.00	1.17
8.	0.00	0.75
9.	0.00	0.83
10.	0.00	0.58
11.	0.00	1.58
12.	0.00	1.50
13.	0.00	2.08
14.	0.00	1.17
15.	0.00	0.67
16.	0.00	1.92
17.	0.00	1.75
18.	0.00	0.92
19.	2.42	1.42
20.	0.00	2.17
21.	0.00	2.00
22.	2.42	0.67
23.	0.00	3.00
24.	0.00	2.17
25.	0.00	2.25

Roanoke Hub (10/04/94)

Truck #	Arrival Time	Operating Time
1.	0.00	0.42
2.	0.00	0.75
3.	0.00	1.75
4.	0.00	0.83
5.	0.00	1.25
6.	0.00	2.83
7.	0.00	0.42
8.	0.00	1.17
9.	0.93	0.42
10.	2.43	1.17
11.	2.43	0.67
12.	3.06	1.25
13.	4.15	0.50
14.	4.42	1.33
15.	4.67	0.47
16.	4.35	0.92
17.	4.67	1.00
18.	5.75	0.67
19.	4.07	1.83
20.	5.75	1.00
21.	5.00	0.50
22.	5.50	0.17
23.	4.15	1.92
24.	4.35	1.25
25.	4.42	0.92

Roanoke Hub (10/05/94)

Truck #	Arrival Time	Operating Time
1.	0.00	0.92
2.	0.00	1.42
3.	0.00	0.58
4.	0.00	1.17
5.	0.00	0.75
6.	0.00	0.50
7.	0.00	0.92
8.	0.00	1.00
9.	0.00	1.67
10.	0.00	1.42
11.	0.00	1.17
12.	0.00	0.17
13.	0.00	0.75
14.	0.00	0.50
15.	0.00	1.58

Roanoke Hub (10/06/94)

Truck #	Arrival Time	Operating Time
1.	0.00	3.83
2.	0.00	4.00
3.	0.00	0.83
4.	0.00	0.58
5.	0.50	1.25
6.	0.50	0.67
7.	2.17	1.08
8.	2.85	0.67
9.	2.85	1.50
10.	4.08	1.08
11.	4.08	1.17
12.	4.48	1.17
13.	4.48	1.50
14.	4.75	0.58
15.	4.75	0.58
16.	4.95	0.83
17.	4.15	2.08
18.	6.00	0.50
19.	4.65	0.92
20.	4.65	1.83
21.	7.48	0.50

Roanoke Hub (10/07/94)

Truck #	Arrival Time	Operating Time
1.	0.00	0.33
2.	0.00	1.17
3.	0.00	2.75
4.	1.25	0.92
5.	0.00	3.00
6.	2.50	0.58
7.	2.57	1.17
8.	1.25	0.75
9.	4.50	1.50
10.	2.50	1.42
11.	2.50	1.83
12.	0.00	4.37
13.	2.33	2.25
14.	4.50	1.17
15.	2.57	2.83
16.	1.92	1.92

Roanoke Hub (10/10/94)

Truck #	Arrival Time	Operating Time
1.	0.00	0.50
2.	0.00	0.33
3.	0.00	0.58
4.	0.00	0.50
5.	0.00	0.92
6.	0.00	1.75
7.	0.00	1.58
8.	0.00	1.17
9.	0.00	1.25
10.	0.00	1.17
11.	0.00	1.08
12.	0.00	2.17
13.	1.25	0.42
14.	0.00	4.33
15.	0.00	2.33
16.	0.00	2.50
17.	0.55	0.83
18.	0.00	4.42
19.	0.00	0.33
20.	0.00	3.00
21.	0.00	0.17
22.	0.00	1.50
23.	0.00	1.50
24.	1.25	1.50
25.	0.00	2.67
26.	0.00	1.83

Roanoke Hub (10/12/94)

Truck #	Arrival Time	Operating Time
1.	0.00	1.75
2.	0.50	2.08
3.	2.67	0.75
4.	2.67	1.33
5.	3.33	2.00
6.	3.33	0.50
7.	3.35	0.67
8.	3.35	1.58
9.	3.57	0.75
10.	3.57	1.67
11.	3.67	0.50
12.	3.67	1.83
13.	4.23	0.83
14.	5.05	1.08
15.	5.05	0.75
16.	3.68	2.17

Roanoke Hub (10/13/94)

Truck #	Arrival Time	Operating Time
1.	0.00	1.00
2.	0.00	1.17
3.	0.25	0.58
4.	0.00	2.67
5.	0.33	1.00
6.	0.33	1.42
7.	1.50	1.67
8.	3.70	0.32
9.	3.70	1.25
10.	3.93	1.25
11.	3.93	1.37
12.	4.50	1.58
13.	4.67	0.42
14.	4.67	0.50
15.	5.50	0.67
16.	5.50	1.25
17.	2.82	2.50
18.	2.50	1.42
19.	2.50	1.50

Roanoke Hub (10/14/94)

Truck #	Arrival Time	Operating Time
1.	0.00	0.50
2.	0.00	1.50
3.	0.33	2.50
4.	0.00	2.00
5.	2.40	1.50
6.	2.33	0.50
7.	3.38	0.83
8.	3.38	2.33
9.	3.43	1.03
10.	3.43	1.08
11.	0.00	2.42
12.	3.93	3.50
13.	4.00	1.67
14.	4.00	1.92
15.	3.90	1.08
16.	5.58	1.17
17.	5.50	0.42
18.	5.50	0.75
19.	5.50	1.67
20.	3.90	1.58

Roanoke Hub (10/17/94)

Truck #	Arrival Time	Operating Time
1.	0.00	1.50
2.	0.00	0.58
3.	4.68	0.83
4.	0.00	0.75
5.	1.09	2.67
6.	0.40	1.42
7.	0.00	2.25
8.	2.10	3.25
9.	0.60	1.83
10.	0.00	2.42
11.	0.00	2.42
12.	1.20	2.58
13.	3.20	3.50
14.	0.00	1.92
15.	5.15	2.00
16.	0.00	2.50
17.	0.00	2.33
18.	1.32	3.50
19.	0.00	4.33
20.	0.00	2.75
21.	5.33	0.25
22.	0.00	0.25
23.	4.75	1.42
24.	4.75	2.50

Harrisburg Hub (10/01/94)

First Shift

Truck #	Arrival Time	Operating Time
1.	0	1.6
2.	0	1.7
3.	0	1.7
4.	0	1.5
5.	0.6	1.1
6.	0.7	1.7
7.	0.9	3.9
8.	1.1	0.5
9.	1.1	2.3
10.	1.1	2.1
11.	1.1	2.2
12.	1.2	0.8
13.	1.2	0.7
14.	1.3	3
15.	1.3	4
16.	1.3	1.4
17.	1.6	4.6
18.	1.6	0.9
19.	2.2	2
20.	3	0.5
21.	3	2.7
22.	3.6	2.2
23.	4	2.7
24.	4	1.5
25.	4.1	1.7
26.	4.4	2.3
27.	4.5	1.1
28.	4.5	1.8
29.	4.5	2.3
30.	4.5	3
31.	4.8	2.6
32.	5.2	1
33.	5.2	2
34.	5.3	1.5
35.	5.3	0.7
36.	5.3	1.8
37.	5.3	1.1

Harrisburg Hub (10/01/94)
First Shift (Contd.)

Truck #	Arrival Time	Operating Time
38.	5.3	2.3
39.	5.3	1.2
40.	5.4	1.5
41.	5.4	0.7
42.	5.4	1.8
43.	5.5	1.8
44.	5.7	1.5
45.	5.7	0.7
46.	5.8	2
47.	5.8	2
48.	5.8	2.1
49.	6.2	1.7
50.	6.4	1.2
51.	7	1
52.	7	0.6
53.	7	0.4

Harrisburg Hub (10/01/94)

Second Shift

Truck #	Arrival Time	Operating Time
1.	0	2.3
2.	0	3.3
3.	0	1.6
4.	0	1.8
5.	0	6.4
6.	0	3.8
7.	0.5	1.3
8.	0.8	1.3
9.	0.8	1.3
10.	1.7	0.6
11.	0.9	1.5
12.	0	4.5
13.	1.4	1.2
14.	0	4.4
15.	2.5	0.7
16.	0.8	2.5
17.	2.5	0.8
18.	1.7	2.3
19.	0.5	3.6
20.	0	5
21.	1.4	3
22.	1.7	3
23.	3.8	1.2
24.	2.4	3
25.	3.3	2.2
26.	4.8	0.9
27.	4.5	1.2
28.	5	1.2
29.	4.2	2.3
30.	3.4	3.2
31.	4.7	2
32.	4.7	2
33.	4.1	2.8
34.	6	1
35.	6	1
36.	0.5	6.9
37.	3.4	4.5

Harrisburg Hub (10/01/94)
Third Shift

Truck #	Arrival Time	Operating Time
1.	0	3.4
2.	0	1
3.	0	1.7
4.	0	1.3
5.	0	1
6.	0	1.4
7.	0	2
8.	0	0.6
9.	0.2	0.4
10.	0	1.4
11.	0	1.2
12.	0	1.3
13.	0	2.3
14.	0	2.3
15.	0	1.9
16.	0	1.4
17.	0	1.3
18.	0	3.7
19.	0	2.5
20.	0	2.1
21.	0.1	1
22.	0	1.2
23.	0	1.7
24.	0	3.8
25.	0	2
26.	0.1	1.3
27.	0	2.2
28.	1.1	0.6
29.	0.2	1.6
30.	0.2	1.6
31.	0	4
32.	0	2.7
33.	0	3.1
34.	1.2	0.8
35.	0	2.8
36.	0	3.2

Harrisburg Hub (10/01/94)
Third Shift (Contd.)

Truck #	Arrival Time	Operating Time
37.	0.9	1.4
38.	1	1.4
39.	0.9	1.5
40.	0	3.2
41.	0	2.9
42.	0.2	2.3
43.	1.5	1.1
44.	1.5	1.2
45.	0.2	2.6
46.	0.4	2.4
47.	1.5	1.3
48.	0	3.6
49.	1	2.1
50.	0	7
51.	0	4.1
52.	0	5.8
53.	1.5	1.8
54.	0.5	3
55.	1.2	2.4
56.	0	4.3
57.	0	4.4
58.	2	1.8
59.	2	2.4
60.	2	2.6
61.	2	2.7
62.	3	1.8
63.	1.9	3
64.	2	3
65.	2.1	3
66.	3.2	2.4
67.	4.5	1.5
68.	6	0.7
69.	3	4
70.	5.3	2
71.	5.8	1.5
72.	6	1.5

VITA

Sanjay Aggarwal was born in Delhi, India on April 21,1968. He spent most of his childhood in Ambala, India where he also attended the elementary and High School. In August 1984, his interest in studying engineering moved him to Wardha, India to attend Nagpur University. He graduated with a B. S. degree in Production Engineering in July 1988. In October 1988 he accepted a job offer in Bhopal, India with Microwin Electronics Limited as an Industrial Engineer. Sanjay quit this job in August 1992 to attend Virginia Polytechnic Institute and State University, Blacksburg, VA for a Masters' degree in Industrial and Systems Engineering. In October, 1994 Sanjay started working in the Operations Research Department at USAir as an Operations Research analyst. In September 1995 he graduated from Virginia Tech with a Master's degree. Meanwhile, he got married to pretty Preety and is now living in McLean, VA.

Sanjay Aggarwal.