RC 21171 (94615) May 4, 1998
Computer Science/Mathematics

IBM Research Report

Inventory Matching Problems in the Steel Industry

compiled by:

Jayant R. Kalagnanam
Milind W. Dawande
Mark Trumbo
Ho Soo Lee

IBM Research Division
T.J. Watson Research Center
Yorktown Heights, New York 10598

IBM Research Division
Almaden · Austin · T.J. Watson · Beijing · Haifa · Tokyo · Zurich

Abstract

This report is an effort to consolidate two different flavors of a surplus inventory matching problem encountered in the steel industry:

- Surplus Unfinished Inventory Matching, and
- Surplus Finished Inventory Matching.

This report is compiled from two other reports that describe the heuristic solution approaches developed for a real world application of these problems [1, 2]. The focus here is to motivate these problems from an operations planning point of view.

The initial sections of this report provide an overview of the production planning process in the steel industry and situate the surplus inventory problem within this context. Subsequently the report is split into distinct parts:

Part I: Surplus Unfinished Inventory Matching

Part II: Surplus Finished Inventory Matching

The problem formulations and solution approaches in these sections are based on the following technical reports respectively.

1. Kalagnanam,J.R., M. Dawande, M. Trumbo, H.S. Lee (1998) "The Surplus Inventory Matching Problem", IBM Research Report 21071 (94285), IBM Research Division, Yorktown Hts, NY.
2. Salman, F.S., J.R. Kalagnanam, S. Murthy (1997) "Cooperative Strategies for Solving the Bicriteria Sparse Multiple Knapsack Problem", IBM Research Report 21059 (94164), IBM Research Division, Yorktown Hts, NY.

Key Words: Matching; surplus inventory; integer programming; sparse bicriteria multiple knapsack problem; multiple knapsack with colors; bipartite graphs.

1 Introduction

We introduce two problems that arise from operations planning applications in the process industry. These problems involves matching an order book against surplus inventory as a preprocessing step to production planning and were encountered in the operation of a large steel plant. In this paper we describe these problems and the heuristic solutions that were developed to solve it. The problem formulations generalize beyond the steel application to other process industries such as paper and metal processing industries. The solutions described here have been deployed in a steel plant and are now used in daily mill operations. We also present a rigorous analysis comparing the performance of the heuristic solutions against optimal solutions using integer programming for small instances. However, the large instances could not be solved to optimality.

Operations planning in a process industry typically begins with a order book which contains a list of orders that need to be satisfied. The initial two steps in an operations planning exercise involves (1) trying to satisfy orders from the order book using leftover stock from the surplus inventory and (2) subsequently designing productions units for manufacture from the remaining orders. This process is presented schematically in Figure 1. In this paper, we discuss the first of the two steps from a optimization perspective.

An important characteristic of a process industry is that production is on a made-to-order basis. As a consequence, the surplus inventory is produced for one of the following reasons:

- Orders are cancelled after a set of items are produced to satisfy it,

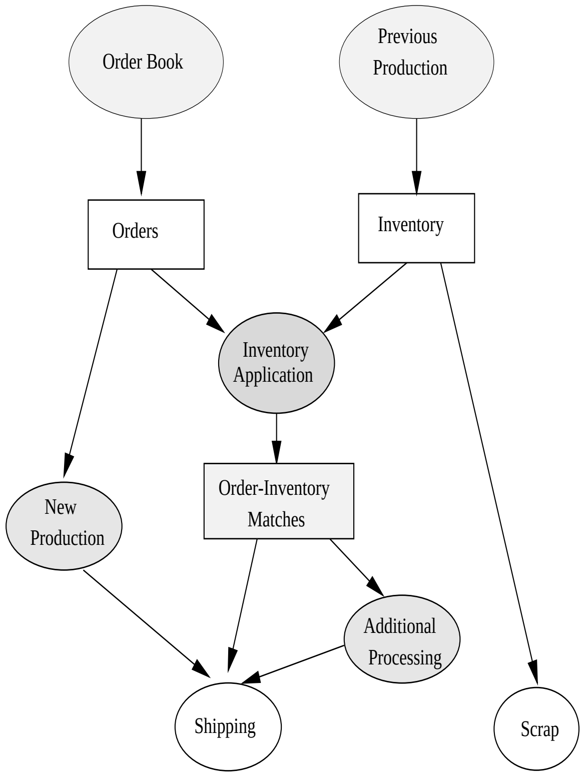

Figure 1: Conceptual flow of operations planning in the process industry

- Items are below the quality level required for the orders that they were being produced for, and
- Surplus items are requested during production or scheduling to satisfy restrictions on machines or operations.

The production process in a steel plant is shown in Figure 2. As shown in the figure a slab cast from the furnace is rolled through a hot and cold strip mill into a coil after which it is finished according to order specifications. However, a slab tagged as being a surplus item before the hot strip mill for any of the reasons mentioned above is *unfinished* and is removed to inventory. Such *unfinished inventory* provides great flexibility in terms of the number of different orders types that can be potentially fulfilled from the same slab by different processing routes through the finishing mill. It is indeed this flexibility that provides an opportunity for optimizing the application of the order book against the surplus inventory. Production units which are tagged as surplus after the finishing mill for quality reasons are classified as *finished inventory* and can only be applied against other orders for which the quality requirements can be met.

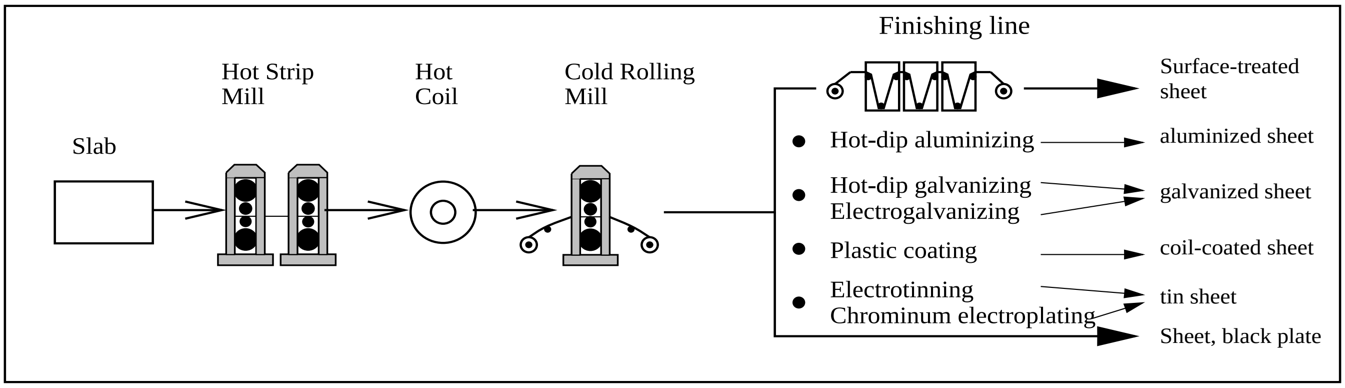

Figure 2: Schematic of a Steel Mill

The problem of applying orders against an existing surplus inventory has a strong flavor of a matching problem - we call this the **Surplus Inventory Matching Problem**. The goal of Inventory Matching is to maximize the total weight of the order book that is applied against the surplus inventory. An additional goal is to minimize the unused weight of those slabs that are matched against orders. Hence surplus inventory matching problem is a bicriteria problem.

Additionally there are two classes of constraints that need to be considered when matching orders against surplus inventory.

- **Assignment Restrictions:** When matching an order against an inventory item it is necessary to consider geometric and quality attributes which restrict the number of potential available matches.

- **Processing Constraints:** While packing multiple orders on an inventory item additional processing constraints that restrict the set of feasible orders that can be packed together need to be considered.

The surplus inventory matching problem can be modeled as variations of the multiple knapsack problem. A bipartite graph is constructed with two sets of nodes, one for the orders and the other for the inventory items (such as slabs). Refer to Figure 3. The edges in this graph represent the potential matches between orders and slabs. In the general case when all matches can be applied against all inventory items the bipartite graph is complete. However the assignment restrictions render many of the edges in the bigraph infeasible and this (often) results in a sparse bigraph. A model based on this representation is referred to as a *sparse bicriteria multiple knapsack problem.* We will show that this model is useful for modeling the surplus inventory matching problem when we retrict ourselves to finished inventory.

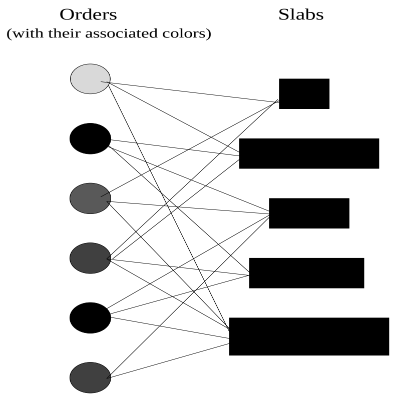

Figure 3: The surplus inventory matching problem

Processing constraints restrict the set of orders that can be packed together. Orders have associated with them a unique route required to finish to specification. If orders with different routes are packed to the same inventory item then these orders need to be separated before the finishing mill. This requires cutting operations which are expensive and hence are restricted to no more than one cut per slab. This translates to having no more than two different routes packed onto the same slab. From a modeling perspective it is convenient to represent each route with a unique *color* and associate a unique color with each order. Now we can describe the processing constraint as a color constraint which restrict the total number of colors per slab to be no more than two. Now the multiple knapsack problem can be extended to have color constraints so that no more than two colors are packed to the same inventory item. This is referred to as the *multiple knapsack problem with color* and this is used to model unfinished surplus inventory matching.

As indicated above, two different flavors of the surplus inventory matching problem arise in the steel industry.

- Matching an order book against the unfinished inventory is modeled using the multiple knapsack with color constraints. Orders are matched against unfinished inventory and need to be finished according to order specifications. Both assignment and processing constraints are considered.

- Matching an order book against the finished inventory can be modeled using the bicriteria sparse multiple knapsack problem. Since only finished inventoy is being used in the matching no further processing is necessary to finish orders. Only assignment restrictions are considered for matching.

The rest of the paper is organized as follows. Section 2 provides a detailed description of the order book and the surplus inventory. Both the assignment and the processing constraints are discussed in detail. Subsequently the report is divided into two distinct parts. Part I deals with the unfinished inventory matching problem and Part II provides an overview of the finished inventory problem.

2 Characterization of the Surplus Inventory Matching Problem

We introduced the surplus inventory problem in terms of an order book which contains a list of orders and their specifications. We also referred to surplus items in the inventory. In this section we provide a detailed description of the order book and the surplus inventory. The specification of orders and the use of surplus inventory (or slabs) in the process industry has some unique attributes which are important in understanding the integer formulations that arise while modeling these problems. Additionally we also discuss the assignment and process constraints for matching.

2.1 The Order Book and Surplus Inventory

The order book contains a list of orders from various customers. Each order has a target weight (O_t) that needs to be delivered. However, there are allowances with respect to this target weight which specify the minimum (O_{min}) and maximum weight (O_{max}) that are accepted at delivery. Over and above the total weight (per order) that needs to be delivered there are additional restrictions regarding the size and number of units into which this order can be factorized at delivery. For example, with each order is associated a range for the weight of the production units which are delivered. Let us assume that the minimum weight for the production unit is PU_{min} and the maximum is PU_{max}. Then for each order we need to deliver an integral number of production units (PU_{number}) of size in the interval [PU_{min}, PU_{max}] so that the total order weight delivered is in the range [O_{min}, O_{max}]. In order to fulfill an order, we need to choose a size for the production unit (PU_{size}) and the number of production units (PU_{number}) to be produced such that

$$O_{min} \leq PU_{size} \times PU_{number} \leq O_{max} \tag{1}$$

$$PU_{min} \leq PU_{size} \leq PU_{max}$$

$$PU_{number} \in \{0, 1, 2, ...\}$$

Notice that the PU_{number} is a general integer variable. Additionally, the constraint represented by Equation 1 is a bilinear constraint.

In addition to the weight requirements each order has three other classes of attributes: (1) The first set pertains to the quality requirements such as grade, surface and internal properties of the material to be delivered. (2) The second set are physical attributes such as the width and thickness of the product delivered. (3) The third set of attributes refer to the finishing process that needs to be applied to the production units. For example, car manufacturers often require the steel sheets to be galvanized.

As discussed in the introduction, production in the process industry is usually on a made-to-order basis and inventory is produced due to manufacturing considerations. Associated with each item[1] in the surplus inventory are three sets of attributes: (1) The first attribute pertains to quality requirements exactly as in the case of orders, (2) the second case pertains to the physical dimensions of the slab such as the width,

[1]For simplicity, the surplus items in the inventory will be referred to as **slabs** in the rest of the paper

thickness and the weight of the slab, and finally (3) associated which each item is a tag that indicates whether it is an unfinished or finished inventory item.

2.2 Constraints for the Matching Problem

The surplus inventory matching problem requires that we maximize the total weight of applied orders while minimizing the unused portion of the applied slabs, subject to certain constraints that arise out of manufacturing considerations. In this subsection we explicate these constraints.

For a given order book, we first assign for each order a set applicable slabs from the surplus inventory. In the general case when we have no restrictions, all slabs can be applied against any order. Let us represent this assignment using nodes for orders and slabs, and arcs to indicate orders and slabs which can be applied against each other. This leads to an undirected[2] bipartite graph from orders to slabs (in the surplus inventory) which is complete since all orders and slabs are applicable against each other. Furthermore, if we decompose each order into constituent production units, then we can replace each order node by a corresponding set of production units (by assuming an appropriate size for the production unit). Each production unit would have the same set of arcs as the parent order. The complete bipartite graph can now be solved as a multiple knapsack problem to maximize the total applied order weight. Each arc represents a potential decision variable which determines the corresponding slab (knapsack) into which the production unit is to be packed. Note however, the multiple knapsack solution does not minimize the unused portions of the applied slabs. In addition this formulation has assumed the size of the production unit rather than

[2]The graph is undirected because a slab j, applicable to order i implies that order i is applicable against slab j.

optimizing for this. However, it is useful to carry around a bipartite graph representation (shown in Figure 3) of the problem since it provides a useful structure for describing some of the manufacturing constraints.

2.2.1 Assignment Constraints

Two sets of constraints arise as a set of assignment restrictions in terms of the applicable slabs for each order. These assignment restrictions are based on quality and physical dimension considerations.

1. The first restriction is that for a given order only slabs which are of the same quality or higher quality can be applied. If we were to list the orders and slabs in terms of non-decreasing value of quality, then the quality restriction would lead to a staircase structure. Consider the zero/one row for each order: the quality of slabs improves as we move from left to right. Therefore once we find a slab of good enough quality, all subsequent slabs are applicable to this order. Therefore quality restrictions might actually reduce the total number of applicable slabs for any order.

2. The second set of restrictions arise from considerations of width and thickness of the orders and the slabs. Usually, the thickness and width of a slab can be altered using rolling, however there is a corresponding range which identifies the limits based on machining or rolling considerations. For example, a slab of width S_w and thickness S_t can be rolled to a slab of width in the range $[S_w^{min}, S_w^{max}]$ and thickness in the range $[S_t^{min}, S_t^{max}]$. As long as the order width and thickness requirements fall into this range, the order can be applied against the slab. These consideration further restrict the number of slabs that can be assigned against an order. Notice

that this additional restriction does not affect the staircase structure but only makes it sparse.

After incorporating these two restrictions, the order applicability matrix becomes quite sparse, usually about 5% of the entries are non-zero. As a result these restrictions lead to a more generalized version of the multiple knapsack problem where the assignment restrictions can be specified as a bipartite graph. The conventional multiple knapsack is an instance of this general problem with a complete bipartite graph. The problem that we have outlined so far with the two restrictions above presents another instance with a sparse staircase structure.

2.2.2 Color Constraints for Packing

The final set of constraints pertain to packing multiple orders on a slab. The assignment restrictions specify a list of orders that can be applied against any slab in the surplus inventory. However, not all orders assignable to a slab can be packed together on the slab. Such packing constraints emerge out of process considerations in the hot and cold mill and the finishing line in the case of a steel mill. Consider a schematic diagram of the route of a slab through a steel mill (Figure 2). A slab is sent through a hot strip mill and a cold mill (if required) and subsequently to the finishing line. After the hot/cold mill the slab is in form of a sheet or a coil. Before the coils are sent to the finishing line, they are cut according to different order specifications. Since orders with different requirements for the finishing line are cut from the slab before processing it is possible to pack such orders on the same slab. However, cutting coils is time consuming and cumbersome and most important, the cutting machine is often the bottleneck in the process flow. Hence often strong constraints are posed in terms of the number of cuts per slab that are allowed based on the current state of the cutting machine.

The simplest representation of this constraint is to specify limit on the number of cuts or the number of different order types (i.e. orders that need to be separated before the finishing line). In order to represent this constraint more formally we introduce a color attribute for each order which describes the set of finishing operations that are required. Orders which require the same set of finishing operations are considered to be of the same type (and hence the same color) and they do not need to be separated before the finishing line. Orders that require different operations in the finishing line are of different type (and hence of different color) have to be separated before the finishing line. Associating a color with each order based on the finishing operations, we can specify a constraint in terms of a limit on the number of different colored orders that are allowed on the same slab. We refer to these process based constraints as the *color constraints.*

Adding the color constraints and decomposing the orders into their constituent production units provides an interesting variation to the multiple knapsack problem (shown in Figure 3). Notice however that this is still an incomplete version of the surplus inventory application problem where we have assumed a production unit size to decompose the order into production units.

Part I

The Unfinished Surplus Inventory Matching Problem

3 The Unfinished Surplus Inventory Matching Problem

We first formulate the unfinished surplus inventory matching problem. We then provide a simplified formulation which removes the non-linearities in the orginal model. A multi-assignment based heuristic is then discussed briefly. Computational results from solving a set of real instances to optimality and comparisons with the heuristic are provided.

4 Problem Formulation (P^*)

Problem Constraints

$$
\begin{array}{rlll}
 & \sum_{i \in N_j} s_j^i z_j^i & \leq W_j l_j & 1 \leq j \leq M \\
 & \sum_{c \in C_j} y_j^c & \leq 2 & 1 \leq j \leq M \\
 & z_j^i & \leq \dfrac{O_{max}^i}{PU_{min}^i} y_j^{c(i)} & 1 \leq i \leq N,\ 1 \leq j \leq M \\
PU_{min}^i \leq & s_j^i & \leq PU_{max}^i & 1 \leq i \leq N,\ 1 \leq j \leq M \\
 & y_j^c & \in \{0,1\} & \forall c \in C_j,\ 1 \leq j \leq M \\
 & l_j & \in \{0,1\} & 1 \leq j \leq M \\
 & z_j^i & \in Z^1 & 1 \leq j \leq M
\end{array}
\tag{2}
$$

Problem Objectives

- Maximize applied weight.

$$\sum_{i=i}^{N} \sum_{j \in N_i} s_j^i z_j^i$$

N	$=$	Total number of orders.
M	$=$	Total number of slabs.
N_i	$=$	Set of slabs incident to order i.
N_j	$=$	Set of orders incident to slab j.
s_j^i	$=$	Production unit size of order i obtained from slab j.
z_j^i	$=$	Number of production units for order i from slab j.
C_j	$=$	Set of colors incident on slab j.
y_j^c	$=$	1 if an order(s) of color c obtains material from slab j; 0 otherwise.
W_j	$=$	Weight of slab j.
PU_{min}, PU_{max}	$=$	Minimum and maximum production unit sizes, respectively, for order i.
O_{min}^i, O_{max}^i	$=$	Minimum and maximum order weight, respectively, for order i.
l_j	$=$	1 if slab j is used to supply some order(s); 0 otherwise.

Table 1: List of notations

- Minimize surplus weight: A surplus is accounted for slab j only if we use the slab.

$$\sum_{j=1}^{M}(W_j l_j - \sum_{i \in N_j} s_j^i z_j^i)$$

If slab j is used, the first set of constraints indicate that the total production material from slab j cannot exceed the weight of slab j. The second set of constraints bounds the number of distinct colors on slab j. The third and the fourth set of constraints set the bounds for the number of production units and the production unit size, respectively, for order i from slab j. Note that the expression for the total applied weight for order i, $\sum_{j \in N_i} s_j^i z_j^i$ is nonlinear. As such the total applied weight, $\sum_{i=i}^{N} \sum_{j \in N_i} s_j^i z_j^i$, which is one of the objectives for the problem, and the constraint set (2) are nonlinear. Thus, the above formulation is a nonlinear integer program. In the absence of efficient, general purpose solution algorithms for this class of problems, we attempt to remove the nonlinearity in the above formulation by assuming that for a given order, the production unit size is constant.

4.1 Simplified Problem Formulation (P)

In the original formulation, the production unit size for order i was allowed to vary between PU^i_{min} and PU^i_{max}. To get a linear formulation, we assume that the production unit size for a given order is constant. For a simplified formulation, we split the max order weight, O^i_{max}, into several production units of equal size, O^i, where $PU^i_{min} \leq O^i \leq PU^i_{max}$. For example, consider an order i with $O^i_{max} = 30$ tons, $PU^i_{min} = 4$ tons and $PU^i_{max} = 6$ tons. We split order i into six production units each having weight 5 tons. Thus, instead of one node (in Figure 2) corresponding to order i, we now have 5 identical nodes. Figure 3 illustrates this splitting. From an

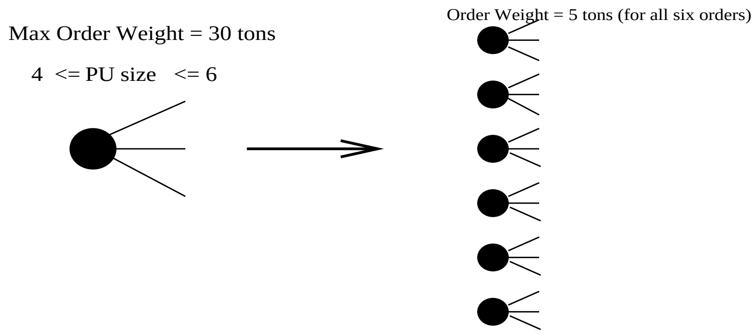

Figure 4: Splitting of an order to simplify the problem

optimization point of view, note that such a splitting prevents us from exploiting the fact that the production unit size of a given order can be any value between PU^i_{min} and PU^i_{max}. However, such a simplification results in an integer *linear* program for which several well-known solution methods exist. Note that if O^i_{max} can be split into an integer number of production unit sizes between PU^i_{min} and PU^i_{max} then such a splitting is always possible. If O^i_{max} is in the "block-out" region (i.e. it cannot be split into an integer number of production unit sizes between PU^i_{min} and PU^i_{max}), we follow a conservative strategy to make $\lfloor \frac{O^i_{max}}{PU^i_{max}} \rfloor$ production units each of size PU^i_{max}.

<u>Problem Constraints</u>

$$\begin{array}{lll}
\sum_{i\in N_j} O^i x_j^i & \leq W_j l_j & 1 \leq j \leq M \\
\sum_{j\in N_i} x_j^i & \leq 1 & 1 \leq i \leq N \\
\sum_{c\in C_j} y_j^c & \leq 2 & 1 \leq j \leq M \\
x_j^i & \leq y_j^{c(i)} & 1 \leq i \leq N,\ 1 \leq j \leq M \\
x_j^i & \in \{0,1\} & 1 \leq i \leq N,\ 1 \leq j \leq M \\
y_j^c & \in \{0,1\} & \forall c \in C_j,\ 1 \leq j \leq M
\end{array}$$

<u>Problem Objectives</u>

N	=	Total number of orders.
M	=	Total number of slabs.
N_i	=	Set of slabs incident to order i.
N_j	=	Set of orders incident to slab j.
x_j^i	=	1 if PU i is assigned to some slab; 0 otherwise.
C_j	=	Set of colors incident on slab j.
y_j^c	=	1 if an order(s) of color c obtains material from slab j; 0 otherwise.
O^i	=	Weight of order i.
W_j	=	Weight of slab j.
l_j	=	1 if slab j is used to supply some order(s); 0 otherwise.

Table 2: List of notations

- Maximize applied weight.

$$\sum_{i=i}^{N} \sum_{j\in N_i} O^i x_j^i$$

- Minimize surplus weight: A surplus is accounted for slab j only if we use the slab.

$$\sum_{j=1}^{M} (W_j l_j - \sum_{i\in N_j} O^i x_j^i)$$

4.2 Problem Complexity

Lemma 4.1 *The Unfinished Surplus Inventory Matching Problem, P, is NP-Complete.*

Proof: If there are no assignment restrictions (i.e. an order can be assigned to any slab), no color restrictions (i.e. all orders have the same color) and a single objective, namely, to maximize the applied order weight; the unfinished surplus inventory matching problem specializes to the multiple knapsack problem [7] which is known to be NP-Complete [5].

□

5 A Multi-Assignment Based Heuristic

One of the key considerations for developing a solution for this inventory application problem was the requirement that the run time for generating the solution be less than a couple of minutes. Therefore the design of heuristic was motivated by the desire for near-optimal solutions that can be generated within a couple minutes. We will show in the following sections that large instances of the inventory matching problem are hard to solve optimally using integer programming techniques and these techniques do not return solutions even after a few hours. For real applications such brittle behavior is unacceptable and it more important to provide near a optimal solution quickly rather than fail completely. It is with this motivation that we have developed a fast heuristic algorithm based on building blocks from network flow algorithms.

Note: Unfortunately, due to confidentiality restrictions, we cannot provide a detailed description of the heuristic we used for solving the problem. However, below we describe, in brief, the main idea behind the heuristic.

The fast algorithm described in this section is a heuristic search algorithm. The search is conducted in a space of matches (or edges of the bipartite graph). The algorithm generates feasible solutions rapidly by assigning multiple edges in each iteration. These feasible solutions are subsequently refined by undoing multiple poor matches to jump to nodes on the search tree which represent partial solutions. These partial solutions then become the root node for the subsequent search. Since both the forward step and backtracking step are fast and create multiple assignments and unassignments this algorithm returns near-optimal feasible solutions quickly.

We instantiate several matches at once and thereby *diving* down the search tree along a particular branch. We identify several matches at once using the assignment problem which provides a lower bound for the solution. Moreover a single application of the weighted bipartite matching algorithm (assignment problem) provides only a partial solution (i.e. we are only halfway down the search tree) and we need to apply this iteratively until we generate a complete solution. After each application of the assignment problem we use a constraint checker to prune out the inapplicable matches in the bipartite graph. Once a complete solution is generated, it is evaluated for multiple objectives such as applied quantity and partial surplus using a *multi-key filter* and based on this evaluation a subset of the matches are marked as undesirable. The search then *backjumps* to a point in the search tree where the state of the solution is such that the undesirable matches have not been applied. This entire process of *diving down* the search tree and then *backjumping* to different point (and maybe branch) of the search constitutes one iteration in the search and is illustrated in Figure 5. This search is continued until several non-dominating solutions are found or the algorithm times out.

The multi-assignment backjumping algorithm consists of three main steps:

Diving: Create a feasible solution by applying Iterative Bipartite Matching on a given initial solution. If feasible solution is near-optimal, store solution in a non domi-

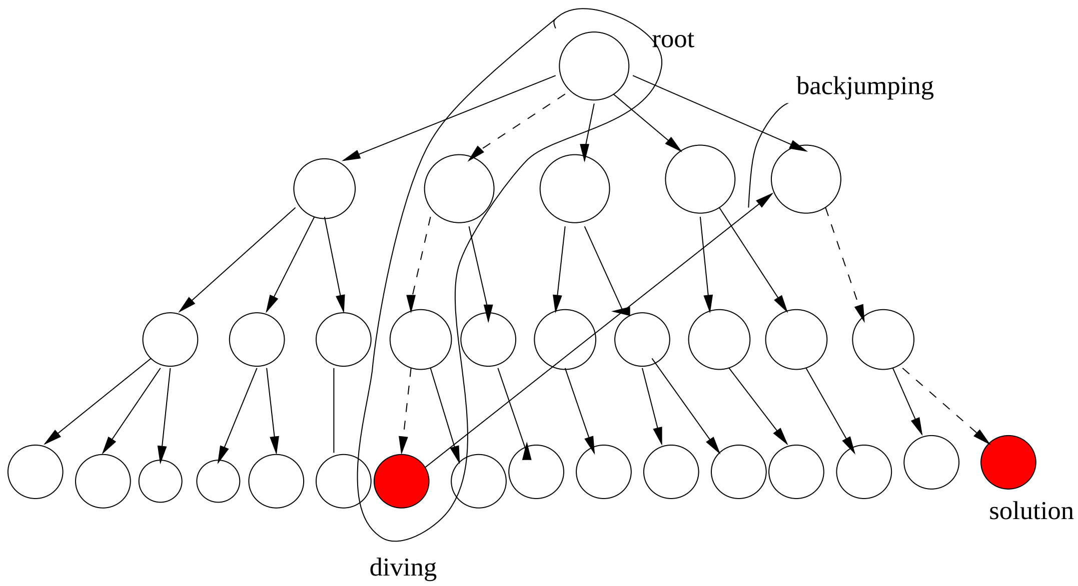

Figure 5: Schematic of the multi-assignment based backjumping algorithm

nated set,

Multi-Key Filter: Use a multi-key sort to identify undesirable matches in a given feasible solution,

BackJumping: Remove undesirable matches from the feasible solution to create a partial solution, and backjump to the position of the partial solution in the search tree. Go back to the first step.

6 Computational Results

In this section we compare the performance of the heuristic, briefly discussed in the previous section, against optimal solutions generated using integer programming. This comparison is performed on a set of real instances encountered in the steel plant. Note that for large instances integer programming techniques could not solve the problem to optimality in well over six hours. However, the smaller instances that were solved to optimality indicate that the heuristic solution provides results within 3% of the optimal. The runtime of the heuristic for all the instances was within a few seconds. A detailed comparison of the results and the runtimes for the heuristic and the integer programming is shown in Tables 3-6.

6.1 Test Problems

In this section, we report our experience with solving problem P on two real world problem instances, g_1 and g_2. These problem instances are based on data from the operations of a large steel plant. As explained above, we first decompose the input graph connected components. Problem g_1 has 38 connected components while problem g_2 has 29 connected components. The characteristics of these components are

Problem name	Number of orders	Number of slabs	Number of colors	Number of edges	Edge density
1	7217	50	1436	36381	10%
2	1524	24	347	9578	26%
3	6	12	2	34	47%
4	413	13	74	1209	22%
5	81	9	24	590	80%
6	200	15	30	383	12%
7	4	3	1	12	100%
8	10	2	4	15	75%
9	5	4	2	19	95%
10	55	6	19	162	49%
11	50	12	16	168	28%
12	3	6	1	18	100%
13	51	3	5	153	100%
14	199	11	59	753	34%
15	50	7	10	80	22%
16	46	59	20	1111	40%
17	39	3	23	57	48%
18	1	2	1	2	100%
19	365	18	110	1097	16%
20	197	2	127	299	75%

Table 3: Problem Characteristics: Problem g1

shown in Tables 3 and 4 respectively. In Tables 3 and 4, the second, third and fourth columns indicate the number of orders, number of slabs and number of distinct colors, respectively, for these connected components. The edge density, calculated as $\frac{\text{Number of edges}}{\text{Number of Slabs} \times \text{Number of Orders}}$ is given in column 6.

Problem name	Number of orders	Number of slabs	Number of colors	Number of edges	Edge density
1	5119	20	552	14750	14%
2	638	35	88	3208	13%
3	19	4	6	43	56%
4	2	3	1	6	100%
5	60	7	12	163	38%
6	407	4	24	1097	67%
7	11	1	2	11	100%
8	111	6	21	445	66%
9	17	1	15	17	100%
10	44	1	15	44	100%
11	4	1	1	4	100%
12	183	8	33	396	27%
13	49	10	10	161	32%
14	39	4	6	73	46%
15	3	4	1	12	100%
16	12	1	2	12	100%
17	23	4	5	81	88%
18	6	1	2	6	100%

Table 4: Problem Characteristics: Problem g2

6.2 Decomposing the problems

If the input graph (Figure 1) has several connected components, solving problem P for the whole graph (instead of solving each connected component separately) can become computationally expensive for methods which use the linear programming relaxation of P since, in general, the dual linear program becomes highly degenerate. Hence, we consider each connected component separately. We use a simple *depth-first search* [6] procedure to find these connected components. If a connected component has a balanced sparse cut (A cut C of G is a set of edges which when removed decomposes the graph into two or more connected components. A sparse cut is a cut with a small number of edges. A balanced sparse cut is a sparse cut which decomposes the graph into two components of roughly the same size), then it is possible to further reduce the size of the problems to be considered by branching on the edges of the cut. Unfortunately, for our problems, the balanced cuts were of about size 100 which was not small enough to exploit divide and conquer strategies.

6.3 Computational Experience

Since the problem we consider is a bi-criteria problem, in general, no single objective function models the optimization of both the criteria exactly. Here, we consider a well known variation where a "budget" constraint on the partial surplus is added and then the applied weight is maximized. For getting a value for this budget, we use the multi-assignment based heuristic given in the previous section. The results for this heuristic are given in Tables 5 and 6. We use the partial surplus value returned by the heuristic (say, ps^*) to add the constraint $\textit{Partial Surplus} \leq ps^*$ to the problem formulation in Section 3.2 and then maximize the applied weight. The results of solving this variation to optimality (using CPLEX 5.0) are also presented in Tables 5 and 6. We also use these results to compare the performance of the heuristic. The idea is to bound the partial surplus by the same value as that returned by the heuristic and then compare the applied weights for the optimum and heuristic solutions. For Tables 5 and 6, the time refers to seconds on an IBM RS/6000 workstation with 64 MB of memory.

For problem g_1, CPLEX was unable to solve 4 components to optimality while for problem g_2, 10 components could not be solved to optimality. For such components, we compare the heuristic with the best integer feasible solution found within a time limit of 3 hours. Note that since these components could not be solved to optimality, the solution obtained by the heuristic might be better than the best integer feasible solution found. For example, components 16 and 19 of problem g_1. The heuristic requires 2 seconds for all the components of problem g_2 and 3 seconds for all the components of problem g_1. The average relative error in the applied weight, where the relative error for each component is calculated as $\frac{\text{optimum value - heuristic solution}}{\text{optimum value}} \times 100$, is 2.49% and 3.05% for problems g_1 and g_2 respectively. Since there is a large disparity between the sizes of the different components, a more reasonable measure is the *weighted* relative

Problem	Heuristic Solution			Optimal Solut ion		
name	Appl. Wt.	Part. Surplus	Time	Appl. Wt.	Part. Surplus	Time
1	408.79	10.69		***	***	10800.00*
2	601.00	247.00		612.75	216.69	10800.00*
3	104.78	44.321		105.63	43.47	0.65
4	2.80	24.58		2.80	24.58	0.01
5	97.29	73.90		101.03	70.17	1.34
6	53.00	2.95		54.90	1.48	10800.00*
7	12.00	0.97		12.00	0.97	0.05
8	118.26	3.60	Total	119.75	3.04	10800.00*
9	20.00	1.00	Time	21.00	1.00	1.05
10	18.00	1.37	= 2 Sec.	19.37	0.00	0.01
11	10.00	1.96		10.00	1.96	0.01
12	151.91	14.94		157.78	9.08	10.06
13	79.80	21.48		80.06	18.58	1.57
14	79.23	6.79		80.21	5.80	0.92
15	26.00	13.47		26.00	13.47	0.01
16	11.59	2.23		13.79	0.03	0.04
17	71.00	9.73		76.69	4.03	3.23
18	15.00	1.25		15.00	0.35	0.03

Table 5: Comparing the heuristic and optimal solution: Problem g2

* Time limit (10800 Seconds) exceeded.

*** No feasible solution found within the time limit.

error. Here, we weight the relative error for each component by the number of edges in that component. The average weighted relative error in the applied weight, where the weighted relative error for each component is calculated as

$$\text{number of edges } \times \frac{\text{optimum value - heuristic solution}}{\text{optimum value}} \times 100$$

is 2.89% and 2.33% for problems g_1 and g_2 respectively.

Problem	Heuristic Solution			Optimal Solution		
name	Appl. Wt.	Part. Surplus	Time	Appl. Wt.	Part. Surplus	Time
1	1035.96	72.58	Total Time = 3 Sec.	***	***	10800.00*
2	627.38	11.78		***	***	10800.00*
3	13.99	16.54		14.25	5.79	0.03
4	299.53	20.53		301.62	19.06	10800.00*
5	162.72	15.16		174.93	2.95	10800.00*
6	266.81	61.26		271.11	57.98	10800.00*
7	17.50	15.54		17.50	5.93	0.02
8	34.86	8.95		35.37	2.16	0.89
9	23.95	83.93		25.54	82.34	0.02
10	124.04	9.28		127.97	5.35	10800.00*
11	173.43	31.28		175.00	30.13	0.85
12	27.00	8.06		28.00	7.56	0.05
13	32.00	8.80		32.04	8.76	0.48
14	262.00	6.01		264.33	4.18	10800.00*
15	128.31	20.07		132.40	17.24	7090.47
16	473.39	165.66		443.40	148.37	10800.00*
17	65.20	32.14		67.00	0.96	3.88
18	40.00	0.56		40.00	0.56	73.56
19	417.18	28.27		410.04	27.02	10800.00*
20	39.44	3.76		42.60	0.76	5.65

Table 6: Comparing the heuristic and optimal solution: Problem g1

* Time limit (10800 Seconds) exceeded.

*** No feasible solution found within the time limit.

Part II

The Finished Surplus Inventory Matching Problem

7 The Finished Surplus Inventory Matching Problem

In this part we model the finished surplus inventory matching problem as a bicriteria sparse multiple knapsack problem. An integer programming formulation of this model is provided and computational experiments on a set of hard real world instances are conducted. We also present some classes of general heuristics for multiple knapsack problems that were used to solve the bicriteria problem.

7.1 The Bicriteria Sparse Multiple Knapsack Problem

We are given a set of items $N = \{1, \ldots, n\}$ and a set of knapsacks $M = \{1, \ldots, m\}$. Each item $j \in N$ has a weight w_j and each knapsack $i \in M$ has a capacity c_i associated with it. All w_j and c_i are positive real numbers. In addition, for each item $j \in N$ a set $A_j \subseteq M$ of knapsacks that can hold item j is specified. Although A_j's suffice to represent the assignment restrictions, for convenience we also specify for each knapsack $i \in M$, the set $B_i \subseteq N$ of items that can be assigned to the knapsack.

The goal is to find an assignment of items to the knapsacks. That is, for each knapsack $i \in M$, we need to choose a subset S_i of items in N to be assigned to knapsack i, such that:

(1) All S_i's are disjoint. (Each item is assigned to at most one knapsack.)

(2) Each S_i is a subset of B_i, for $i = 1, \ldots, m$. (Assignment restrictions are satisfied.)

(3) $\sum_{j \in S_i} w_j \leq c_i$, for $i = 1, \ldots, m$. (Total weight of items assigned to a knapsack does not exceed the capacity of the knapsack.)

(4) $\sum_{i \in M} \sum_{j \in S_i} w_j$ is maximized. (Total weight of items assigned is maximized.)

(5) $\sum_{i \in I}(c_i - \sum_{j \in S_i} w_j)$ is minimized, where $I \subseteq M$ denotes the set of indices of non_empty S_i. (Total waste due to the unused portion of each utilized knapsack is minimized.)

We refer to this problem as the bicriteria sparse multiple knapsack problem (BSMK).

Without loss of generality we assume that $w_j \leq c_i, \forall j \in B_i$, otherwise j can be removed from B_i. The problem becomes trivial if all A_j's are disjoint, or if $\sum_{j \in B_i} w_j \leq c_i, \ \forall i \in M$. In the case that all B_i's are disjoint, the problem decomposes into m single 0-1 knapsack problems. Thus, we exclude these cases from consideration.

Note that the assignment restrictions can also be represented by a bipartite graph, where the two disjoint node sets of the graph correspond to the sets N and M. Let $G = (V, E)$ be the corresponding bipartite graph with $V = N \cup M$. Then, there exist an edge $(i, j) \in E$ between nodes i and j if and only if $j \in B_i$. With this representation the sparsity of the problem refers to the edge sparsity of the bipartite graph G. The *bicriteria* problem is more relevant for sparser problems because for more constrained problems, a solution with maximum assigned weight does not necessarily have small waste.

7.2 Problem Formulation

In this section we provide two formulations for the finished inventory matching problem. First we formulate the problem for a single objective of maximizing total assigned weight - we refer to this as the sparse multiple knapsack problem (SMK). Later we extend this formulation to incorporate the second objective of minimizing waste called the bicriteria sparse multiple knapsack problem (BSMK).

The IP formulation of SMK is as follows.

$$
\begin{array}{ll}
\max & \sum_{i \in M} \sum_{j \in B_i} w_j\, x_{ij} \\
\text{st} & \\
& \sum_{j \in B_i} w_j\, x_{ij} \le c_i, \;\; i \in M \\
& \sum_{i \in A_j} x_{ij} \le 1, \;\; j \in N \\
& x_{ij} \in \{0,1\}, \;\; i \in A_j, \;\; j \in N
\end{array}
$$

where the 0-1 variable x_{ij} denotes whether item j is assigned to knapsack i. The LP relaxation corresponds to relaxing integrality of these variables.

In the second problem considered, the objective function is the sum of the two objectives, that is we maximize assigned weight minus waste. We call this problem BSMK as it combines multiple knapsack and bin packing aspects. The IP formulation is as follows.

$$
\begin{array}{ll}
\max & \sum_{i \in M} \sum_{j \in B_i} w_j\, x_{ij} - \big(\sum_{i \in M} c_i\, z_i - \sum_{i \in M} \sum_{j \in B_i} w_j\, x_{ij} \big) \\
\text{st} & \\
& \sum_{j \in B_i} w_j\, x_{ij} \le c_i\, z_i, \;\; i \in M \\
& \sum_{i \in A_j} x_{ij} \le 1, \;\; j \in N \\
& x_{ij} \in \{0,1\}, \;\; i \in A_j, \;\; j \in N \\
& z_i \in \{0,1\}, \;\; i \in M
\end{array}
$$

where we introduce the 0-1 variable z_i to denote whether any item is assigned to knapsack i. The objective function equals $2 \sum_{i \in M} \sum_{j \in B_i} w_j\, x_{ij} - \sum_{i \in M} c_i\, z_i$, and the LP relaxation corresponds to relaxing the integrality of all variables. The LP relaxation of both SMK and BSMK problems have the same optimal value because an optimal solution to the LP relaxation of BSMK will have zero waste. Hence, we refer to the relaxation of both problems by “the LP relaxation”.

7.3 Problem Complexity

The SMK problen can be modeled as a generalized assignment problem (GAP) with missing edges weighted with zero profit. It is well known that GAP is NP-complete [2].

8 Heuristics for the Multiple Knapsack Problem

With the aim of generating non-dominated solutions for BSMK, we have developed a collection of constructor and improver heuristics. Most of these heuristics are simple greedy heuristics which are adapted from the heuristics in the literature used for multiple knapsack and variable-size bin packing problems. In addition, we have a few randomized heuristics.

The construction heuristics are mainly greedy heuristics with various item and knapsack selection rules, in addition to a couple of heuristics that round the LP relaxation solution of SMK. Most of these constructors aim at maximizing total assigned weight.

The improver heuristics are either local exchange heuristics which aim to improve both of the objectives, or heuristics which rearrange assigned items among knapsacks and unassign some items for the purpose of minimizing total waste. In the next two subsections we give a description of each heuristic.

Constructors	Improvers
Simple Greedy Heuristics	Local Exchange Heuristics
Greedy Heuristics with Knapsack Selection	Heuristics to Increase Assigned Weight
Succesive Assignment	Heuristics to Eliminate Waste
LP Relaxation based Heuristics	

Table 7:

8.1 Cooperative Strategies for Problem Solving

Given an NP-hard optimization problem, it is difficult to design heuristic algorithms which exhibit uniformly superior performance over all problem instances. An alternate approach to tackle difficult problems is to organize a collection of heuristic algorithms so that they can cooperate with each other and uniformly exhibit superior performance which might not have been possible if they were used separately. Such an approach is especially attractive when the collection of heuristic algorithms vary in their performance over problem instances in an unpredictable way. Another ingredient required for cooperative problem solving is an architecture that facilitates cooperation between the heuristic algorithms and a control strategy that defines the rules of collaboration among heuristics.

In the original report we discuss in more detail the organization (i.e. the architecture and the control strategy) that we have used to build a cooperative problem solving team of heuristics for the multiple knapsack problem.

9 Computational Experiments

In this section we examine the performance of the cooperative problem solving strategy and compare its behavior against traditional integer programming based techniques for BSMK problem using computational experiments. We also compare the performance of the cooperative strategy against individual heuristics in an effort to quantify the improvements gained by cooperation. Finally we analyze the non-dominated solutions to identify concatenations of heuristics that generate good solutions.

9.1 Data

We used real data from an inventory application problem in Steel Mill Industry [1]. For the instances available to us, the number of items vary between 111 and 439, while the number of knapsacks is between 18-43. The sparsity of the problems are in the range 10% - 28%. Size and sparsity of these instances are summarized in Table 8.

Data	n	m	sparsity %	Tot. Cap.	Tot. Weight
d1	439	24	27.7	641.85	4689.91
d2	111	35	12.8	1009.32	1770.81
d3	393	18	26.7	388.84	4276.53
d4	209	43	10.6	889.21	3528.04
d5	191	35	14.2	730.81	2885.49
d6	155	18	18.6	446.32	1509.22

Table 8: Information on real-life data. Sparsity denotes the edge density of the bipartite graph representation in percentage of the number of edges of a complete bipartite graph. The last two columns denote the total capacity of knapsacks and the total weight of items.

9.2 Implementation

Individual heuristics

We coded the heuristics presented earlier in C++ language using the LEDA library and performed the tests on an IBM RS4000 machine operating under AIX. First, we collected solutions output by each constructor and improver heuristic and then found the non-dominated solutions among all collected solutions. When running the improvers, we used the solution provided by the greedy-knapsack constructor as input. The randomized heuristics were run 10 times and the best solution among all runs was output.

A-team

We incorporated the individual heuristics into an A-team architecture. We set the parameters such as the stopping time and the probabilities for picking constructors and improvers by a few initial tests. The probabilities for some of the improvers, (such as replace-single, replace-pair, empty, and empty-and-reassign) which were more effective during initial runs were increased. We examined the convergence of the solution population by running the A-team code for a cycle of 100, 500, 1000, 1500, 2000 and 3000 heuristics. By observing the number of non-dominated solutions output, the maximum and average value of assigned weight minus waste of these non-dominated solutions, we decided on the cycle length of the A-team run for each data set.

IP based approaches

In order to assess the performance of the heuristics and the computational difficulty of the problem, we tried to solve two IP's each with a single objective by a branch-and-cut method. The first problem considered to compare our results is the sparse multiple knapsack problem, SMK, where the objective is to maximize assigned weight only. The IP formulation of SMK was given earlier. In the second problem considered, the objective function is the sum of the two objectives, that is we maximize assigned weight minus waste (BSMK problem).

In order to obtain the best bounds possible in a reasonable computation time, we added the best lower bound obtained from our heuristics to the IP formulation of SMK and MKBP, and solved them by $CPLEX$. After 1 hour of CPU time, we added the upper bound output by CPLEX to the formulation and ran CPLEX again (we waited longer for larger instances such as d1 and d3). We repeated this procedure until no better bounds were found in more than 2 hours. A comparison of the LP relaxation values and the upper bounds generated by a branch and bound method for data sets d1 to d6 are given in Table 9. The running times given in Table 9 also give a crude idea on the computational difficulty of solving the problems by exact methods. It took 2.4 hours to get the optimal solution to SMK for data set d2, and we could not obtain the optimal solution for any other problem.

		SMK			MKBP		
Data	LP bound	UB	%Gap	Time (hrs)	UB	%Gap	Time (hrs)
d1	641.85	639.70	0.34	9.2	639.70	0.34	2.6
d2	612.64	472.14	29.76	2.4	403.83	51.70	4.4
d3	385.37	384.99	0.10	6.2	381.14	1.11	8.0
d4	785.67	687.63	14.05	4.0	606.86	29.46	6.9
d5	659.39	598.38	9.92	16.1	510.79	29.10	6.4
d6	444.01	424.20	4.67	8.2	414.37	7.15	5.0

Table 9: A comparison of the LP relaxation value and the best upper bound (UB) obtained by a branch and bound method for the SMK and MKBP problems. The column (% Gap) denotes the deviation in % of the LP bound from the best available bound. Time denotes the cpu time to obtain the given bounds.

Here, we note that as the problem gets sparser, it gets harder in the sense that a solution that maximizes assigned weight does not necessarily have small waste, hence the choice of the knapsack to which an item is assigned becomes more critical. As the problem gets more relaxed (that is, the bipartite graph representation gets closer to a complete graph), the problem gets closer to the multiple knapsack problem and usually, maximizing assigned weight suffices to minimize waste at the same time.

The sparsity of the problem plays a role also in determining the strength of the LP relaxation. As sparsity increases, the gap between the LP relaxation value and the optimal value gets larger for both problems.

We also collected feasible solutions output by CPLEX and recorded the cpu times to obtain the solutions. In these runs we did not provide any bounds to the objective function initially, but reran CPLEX with previously obtained bounds whenever we had to stop due to memory problems. We stopped this procedure when no improvement could be obtained till the computer ran out of memory.

One may also consider using the best lower bound output by the individual heuristics in the IP formulation, as opposed to a cooperative strategy. However, using the heuristics in this way does not improve upon the performance of the branch-and-cut method of CPLEX significantly. Even using the better bounds output by the cooperative strategy does not yield the optimal solution in reasonable computation time.

9.3 Comparative Evaluation of the Cooperative Strategy

Quality of Solutions

The quality of the solutions generated by using a cooperative strategy are significantly better than the ones generated by individual runs, especially in the waste objective. A comparison of the solutions with maximum assigned weight generated by A-team implementation and individual runs is provided in Table 10. The waste of these solutions are also given in the table. We see that the cooperation of the heuristics have been useful to decrease the waste of the solutions that have the maximum assigned weight. We also note that we could not obtain any solutions with a better assigned weight by the exact solution method (using $CPLEX$) for any of the problems except for d2.

Solutions with maximum value of (assigned weight - waste), that are generated by the A-team implementation, individual heuristics, and CPLEX are given in Table 11. We observe a significant difference in (assigned weight - waste) of the solutions generated by the cooperative strategy versus those generated by individual heuristics, especially

Data		AW	Ratio	Waste	Waste %	Cpu Time
d1	I	636.60	0.9952	5.25	0.82	4399.06
	II	617.69	0.9656	24.16	3.76	37.81
	III	601.18	0.9398	40.67	6.24	29622.41
d2	I	470.98	0.9975	108.24	18.69	70.01
	II	470.32	0.9961	230.50	32.89	1.48
	III	472.14	1.0000	110.04	18.90	8236.05
d3	I	383.20	0.9954	5.64	1.45	9834.42
	II	382.33	0.9931	6.52	1.68	72.32
	III	366.10	0.9509	22.74	5.85	46009.14
d4	I	686.99	0.9990	110.97	13.91	318.17
	II	673.39	0.9793	155.54	18.76	3.44
	III	686.99	0.9990	140.68	17.00	39923.20
d5	I	592.33	0.9899	96.92	14.06	183.70
	II	590.23	0.9864	99.02	14.37	3.06
	III	591.33	0.9882	97.92	14.21	39360.16
d6	I	406.12	0.9574	30.95	7.08	76.46
	II	402.92	0.9498	34.15	7.81	1.63
	III	402.97	0.9500	34.10	7.80	46570.32

Table 10: A comparison of the solution with *maximum assigned weight* obtained by I) the A-team implementation, II) individual runs of all heuristics, and III) branch-and-cut. AW is assigned weight. Ratio is the ratio of AW to the best available bound for the assigned weight objective (from Table 9). Waste % is the ratio of the unused capacity to the total capacity of utilized knapsacks in percentage. Cpu time is given in seconds.

for the harder instances such as d2 and d4. Slightly better solutions could be obtained by $CPLEX$ for only d2 and d4 in 4.4 and 6.9 hours, respectively. For larger instances such as d1 and d3, the feasible solutions output by CPLEX are significantly inferior to those output by the cooperative strategy with a difference of 10% and 7% of the best upper bound available, respectively.

The collection of heuristics are able to generate significantly more non-dominated solutions when they cooperate in an A-team implementation. Few heuristics are able to effectively improve both the objectives at the same time. As a result, while the individual heuristics are good enough to maximize assigned weight, they are not ca-

Data		AW-Waste	Ratio	AW	Waste	Waste %	Cpu Time
d1	I	631.35	0.9869	636.60	5.25	0.82	4399.06
	II	593.53	0.9278	617.69	24.16	3.76	37.81
	III	565.62	0.8842	603.74	38.11	5.94	44543.33
d2	I	396.72	0.9835	450.59	53.87	10.68	70.01
	II	326.48	0.8094	338.14	11.66	3.33	1.48
	III	398.86	0.9888	471.31	72.45	13.32	15681.26
d3	I	377.56	0.9906	383.20	5.64	1.45	9834.41
	II	375.81	0.9860	382.33	6.52	1.68	72.32
	III	354.36	0.9297	371.60	17.24	4.43	28723.18
d4	I	595.55	0.9814	657.99	62.45	8.67	318.17
	II	566.41	0.9333	649.01	82.61	11.29	3.44
	III	605.34	0.9975	667.99	62.66	8.58	24942.31
d5	I	497.14	0.9733	580.33	83.19	12.54	183.70
	II	491.21	0.9617	590.23	99.02	14.37	3.06
	III	465.90	0.9121	564.62	99.08	14.93	23117.35
d6	I	375.17	0.9054	406.12	30.95	7.08	76.46
	II	368.70	0.8898	402.92	34.15	7.81	1.63
	III	365.83	0.8829	x 401.45	35.62	8.15	18203.67

Table 11: A comparison of the solution with *maximum (assigned weight - waste)* obtained by I) the A-team implementation, II) individual runs of all heuristics, and III) branch-and-cut. The ratio is obtained using the best available upper bound for maximizing assigned weight minus waste obtained from Table 9. Cpu time is given in seconds.

pable of minimizing waste at the same time. On the other hand, using a cooperative organization, the heuristics which favor maximizing assigned weight and those which favor minimizing waste have the chance to take the output of one another as input. Therefore, they generate solutions with better values in both objectives.

Run Time

Clearly, combining the heuristics by the cooperative strategy increases running time as a cycle of 1500 - 3000 heuristics are run. However, still the run times are in a reasonable range and are significantly smaller compared to that of the branch-and-cut method.

A single run of each individual heuristic takes between 0.01 - 45 seconds, depending on the size of the problem and the heuristic used. The most time-consuming heuristic is the replace-single heuristic, which takes 45 seconds for d3 and only 0.24 seconds for d2. The constructor heuristics take very little time. While greedy heuristics run in less than a second, the lp-round and lp-greedy heuristics take slightly more time (0.17 - 1.31 seconds). The most time-consuming constructor successive-assign takes 0.32 to 4.25 seconds of cpu time.

The A-team implementation takes a time of approximately 1 minute - 3 hours, depending on the size of the problem. Still, the run times are significantly small compared to that of the branch-and-cut approach of CPLEX, which is in the order of 3 - 13 hours.

All the run times are given in Tables 10 and 11. In these tables, the run time for individual heuristics is the total time over all heuristics, as the best solution was picked after running all the heuristics.

References

[1] J. Kalagnanam, M. Dawande, M. Trumbo, and H. S. Lee. "The surplus inventory matching problem in the process industry". Technical Report RC21071, IBM T.J. Watson Research Center, 1998.

[2] S. Martello and P. Toth. "Lower bounds and reduction procedures for the bin packing problem." *Discrete Applied Math.*, 28:59–70, 1990.

[3] Ahuja, R.K., Magnanti, T.L. and Orlin, J.B. (1993), *Network Flows,* Prentice Hall, New Jersey.

[4] Ferreira, C. E., Martin, A. and Weismantel, R. (1996), *Solving multiple knapsack problems by cutting planes,* SIAM J. Optimization 6, 858-877.

[5] Garey, M. R. and Johnson, D. S. (1979), *Computers and Intractibility: A Guide to the Theory of NP-Completeness,* W. H. Freeman and Co., San Francisco.

[6] Horowitz, E. and Sahni, S. (1978), *Fundamentals of Data Structures,* Computer Science Press, Inc.

[7] Martello, S. and P. Toth (1990) *Knapsack Problems: Algorith ms and Computer Implementations,* Wiley, Chichester, U.K.

[8] Nemhauser, G.L., Savelsbergh, M.W.P and Sigismondi, G.S. (1994), *MINTO, a Mixed INTeger Optimizer,* Oper. Res. Letters 15, 47-58.

[9] Savelsbergh, M.W.P and Nemhauser, G.L. (1996), *Functional description of MINTO, a Mixed INTeger Optimizer (Version 2.3),* Report COC-91-03D, Georgia Institute of Technology, Atlanta, GA.

[10] *Steel Manual* (1992), Verein Deutscher Wisenhuttenleute, Germany.