RC 21059 (94164) revised November 17, 1997
Computer Science/Mathematics

IBM Research Report

Cooperative Strategies for Solving the Bicriteria Sparse Multiple Knapsack Problem

F. S. Salman
Jayant R. Kalagnanam
Sesh Murthy

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

Cooperative Strategies for Solving the Bicriteria Sparse Multiple Knapsack Problem

F. S. Salman *   *J. Kalagnanam* †   *S. Murthy* ‡

May 15, 1998

Abstract

The problem of assigning a given set of orders to the production units in the inventory arises frequently in production planning and scheduling. Typically, this problem can be formulated as a variation of the multiple knapsack problem where items (orders) are packed into knapsacks (production units). In our model we consider variations such as multiple objectives and additional assignment restrictions. We consider the objectives of both maximizing total amount of assignment and minimizing total waste due to unused knapsack capacity, as well as assignment restrictions which can be represented by sparse bipartite graphs.

The focus of this paper is on obtaining non-dominated solutions in reasonably short computational time. This paper studies real instances from the steel industry which proved very hard to solve using conventional integer programming techniques. A cooperative organization of heuristics was used to solve these instances with surprising effectiveness. Such an approach can generalized for use for other hard optimization problems.

Keywords: sparse multiple knapsack, cooperative organizations, collection of heuristic algorithms, integer programming.

*GSIA, Carnegie Mellon University, 5000 Forbes Avenue, Pittsburgh PA 15213. email: `fs2c+@andrew.cmu.edu` Supported by IBM Corporate Fellowship.

†T. J. Watson Research Center, IBM, Yorktown Heights, NY 10598. email: `jayant@watson.ibm.com`

‡T. J. Watson Research Center, IBM, Yorktown Heights, NY 10598. email: `murthy@watson.ibm.com`

1 Introduction

One of the major goals of a production facility is to utilize its inventory in the best possible way to satisfy demand. In make-to-order production systems, such as the process industry, a surplus inventory accumulates due to cancellations of orders and rejection of production units for failing to satisfy quality requirements. Clearly, it is to the advantage of the production facility to utilize this surplus inventory before planning its production acitivites. This raises the problem of assigning a list of orders to the production units in the inventory. The objectives are both maximizing the total amount of orders that are assigned, and minimizing total waste of production units. Manufacturability considerations such as the compatibility of orders and production units in terms of quality, size, etc. impose additional assignment constraints. As production operations involve more complex processes and a larger product variety, the problem becomes more constrained. The *bicriteria sparse multiple knapsack problem* that we consider in this study is motivated by this application.

In this paper we focus on the use of a team of heuristic algorithms which cooperate to generate non-dominated solutions for this problem in a short computation time. Although there exist several heuristic approaches for solving multiple knapsack problems there does not exist a single dominant algorithm. Moreover, the performance of the heuristics vary by problem instance and as a result a specific heuristic will often demonstrate poor aggregate performance over a set of problem instances. However, if the heuristics were allowed to cooperate with each other so that

- the solution generated by one heuristic can be subsequently improved by another or

- the most appropriate subset of heuristics can be used to construct solutions for a given problem instance

then the aggregate performance of a collection of cooperating heuristics over a set of problem instances may be greatly improved. For this purpose we have developed a collection of fast heuristics and incorporated them in an A-team architecture, which provides a computational framework for implementing cooperation strategies among heuristics. We present results of an experimental analysis that compares the effectiveness of these heuristics working individually, and cooperating within an A-team framework. Additionally for calibration purposes we compare these results against feasible solutions derived using integer programming formulations. Since an important consideration in a real application is the computation time required to generate solutions, we also compare the performance of such a cooperative problem solving strategy against traditional integer programming techniques.

The paper is organized as follows. We first formalize the problem in Section 2. In Section 3 we provide an overview of the related work in the literature on the two versions of the problem each with a single objective. We present our collection of heuristics and a discussion of the cooperative problem solving strategy used in Section 4. Section 5 is a summary of the results of the experimental analysis and finally we conclude in Section 6.

2 Problem Definition

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, where 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. 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.

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

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

(5) $\sum\limits_{i \in I} (c_i - \sum\limits_{j \in S_i} w_j)$ 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.

3 Related Problems in the Literature

The single objective versions of the bicriteria problem are slightly modified forms of two well-known problems in the literature.

If we consider the objective of maximizing total assigned weight alone, then the problem is a variation of the *multiple knapsack problem*, which we call the sparse multiple knapsack problem (SMK). For the objective of minimizing total waste alone, the problem reduces to a variation of the *variable-size bin packing problem*, which we refer to as the sparse bin packing problem (SBP).

In the classical multiple knapsack problem, any item can go into any knapsack, hence the bipartite graph representing the problem is complete, whereas we generalize the problem by allowing any bipartite graph. On the other hand the multiple knapsack problem has a more general objective function: there exists a positive profit p_j for assigning item j to any of the knapsacks and the objective function is to maximize the total profit of assigned items. In the application which motivated us, the profit of an assigned item can be assumed to be proportional to the weight of the item, hence we maximize total assigned weight.

The multiple knapsack problem is known to be NP-hard in the strong sense [Kar72], [MT90]. The reduction from the 3-partition problem is still valid when the objective function coefficients equal to the weights of items, so any instance of SMK with a complete bipartite graph representation is also NP-hard in the strong sense. Thus, for the objective of maximizing total assigned weight, our problem is strongly NP-hard and there exists no fully polynomial time approximation scheme for SMK unless $P = NP$.

For the multiple knapsack problem, several exact and heuristic solution methods have been developed and tested in the literature (see Martello and Toth [MT90] for a survey). Exact solution methods consist of branch and bound, and cutting planes. The branch and bound methods use bounds based on either the Lagrangean relaxation[1] (Hung and Fisk [HF78]) or the surrogate relaxation [2] (Martello and Toth [MT80, MT81a]) of the problem. The cutting plane methods use minimal cover, (1,d)-configuration and multiple cover inequalities (Ferreira, Martin and Weismantel [FMW96]). Unfortunately, these exact solution methods cannot solve large instances arising in real applications in reasonable computation time. Heuristic methods include fast greedy algorithms followed by local exchange heuristics (Martello and Toth [MT81b]), as well as non-polynomial time approaches such as solving single knapsack problems successively (Martello and Toth [MT81a]), or obtaining a feasible solution from the surrogate relaxation (Hung and Fisk [HF78]).

Considering the objective of minimizing waste alone produces an ill-posed problem with a trivial optimal solution that does not assign any items. Hence, we can consider

[1] relaxing the assignment constraints decomposes the problem into m single knapsack problems.

[2] getting a linear combination of the capacity constraints results in a single knapsack problem.

a version in which we impose the condition that all items in N or a specified subset of N must be assigned and the goal is to use knapsacks with minimum total capacity. Then, the problem is a generalization of the variable-size bin packing problem, where we allow assignment restrictions in addition.

The bin packing problem is known to be NP-hard (i.e. [GJ79]), thus the more general problem SBP is NP-hard as well. The bin-packing problem has been extensively studied in the literature and it is one of the first problems for which efficient approximation algorithms have been developed. The recent survey by Cook, Garey and Johnson [CGJ97] covers worst and average case analyses for online and offline algorithms. A previous survey by the same authors [CGJ84] considers also some variations on the problem. Exact algorithms have been developed by Martello and Toth [MT89]. The *variable-size* bin packing problem has been studied by Friesen and Langston [FL86] who provide modifications of well-known bin packing heuristics such as the next fit, first fit and best fit heuristics.

4 Cooperative 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 following paragraphs 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. We will also discuss in some detail the collection of heuristics that have been used to build the cooperative problem solving team.

4.1 The Asynchronous Teams Architecture

An asynchronous team or A-Team [TdSM93] is an architecture that facilitates multiple heuristics to work together on a common problem. Cooperation between heuristics is allowed through a shared population of candidate solutions. Figure 1 provides a schematic of this architecture. The architecture is similar to a blackboard system in that the solutions are posted onto a blackboard which is shared by all the heuristic algorithms. Each heuristic has access to the entire population of solutions and can choose an appropriate (partial) solution to work on.

The heuristics that are used in this architecture are usually classified into three categories based as follows:

- *Constructors* are heuristics which are used to create initial solutions.
- *Improvers* are heuristics which take existing partial solutions from the population and modify them to produce a new solution. The criteria used to decide whether a solution is added to the population depends on the choice of the control strategy. In purely hill climbing approaches (such as genetic algorithms) only solutions which are non-dominated would be added to the population. In variants such as simulated annealing we might allow the entry of dominated solutions into the population with the expectation that they might allow for better solutions to be created later on.
- *Destroyers* are heuristics that remove redundant solutions from the population with the intent of managing the size of the population. The determination of whether a solution is redundant is difficult and usually a destroyer is designed to retain a redundant solution with a non-zero probability. Note that destroyers are typically used with control strategies that allow for the inclusion of dominated solutions into the population.

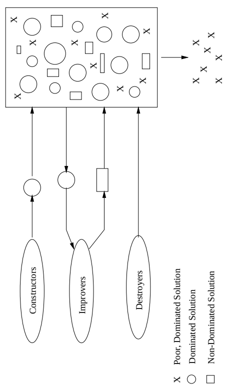


Figure 1: Schematic of an A-Team architecture

The typical approach for generating solutions using this architecture involves creating an initial population of solutions using all the constructor heuristics and subsequently evolving the population of solutions by repeated application of the improver and destroyer heuristics. At any time the set of non-dominated solutions constitute a Pareto-frontier and provide a set of non-dominated solutions to the problem. The control strategy in this solution approach prescribes the rules of collaboration between the heuristics. The control strategy specifies two rules:

- The first rule specifies how an improver heuristic picks a solution from the population to improve, and
- the second rule specifies the criteria for incorporating a new solution created by an improver heuristic into the population.

In our implementation we used a stochastic hill climbing approach for the control strategy. A stochastic hill climbing algorithm searches a space S with the aim of finding a state with optimal properties. The algorithm does this by making successive improvements to the current state $\rho \in S$. In the context of this paper, the state ρ corresponds to a population of solutions to the bicriteria multiple knapsack problem. The algorithm attempts to improve the current state ρ by making a transition to one of the neighbors τ of ρ in S. Within our A-Team implementation, this transition is made by randomly picking an improver heuristic and then randomly picking a solution from the population for the improver to work on. If the new solution generated by the improver is non-dominated then it is added to the population which corresponds to the new state τ. Since the hill climbing approach does not allow non-improving moves, we do not explicitly need to worry about managing the size of the population. As a result our implementation of A-team for the bicriteria sparse multiple knapsack problem did not require destroyers.

A-team architecture has been used to obtain good feasible solutions to various combinatorial optimization problems such as the TSP problem (Talukdar and de Souza[TdS93]), and scheduling problems arising in steel and paper manufacturing industry (Lee et al. [LMHM96], Murthy et al. [MARW97]). The reader is referred to Talukdar, de Souza and Murthy [TdSM93] for more on A-Team architecture.

4.2 Heuristics

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.

4.3 Constructor Heuristics

Simple Greedy Heuristics

These heuristics first sort the items in non-increasing order of weight and the knapsacks in non-decreasing order of capacity. There are two versions. In the first one, the next knapsack in the order, say i, is picked, and then the next item, in the order, say j, is picked. If item j is allowed to go into knapsack j, i.e. if $j \in B_i$, and w_j does not exceed the residual capacity of knapsack i, then item j is assigned to knapsack i. So, for each knapsack picked, as many items as possible are packed into it. In the second version, each item in the order is picked and assigned to the next feasible knapsack, if possible. Both of the heuristics have running time $O(n \log n + m \log m + nm)$. We call these heuristics *greedy-knapsack* and *greedy-item*.

Greedy Heuristics with Various Knapsack Selection Rules

In these heuristics the decision to pick the next knapsack depends on the assignments made upto that point. The heuristics sort the items in non-increasing order of weight, pick the next item j and then pick a knapsack $i \in A_j$ according to one of three rules. There are three versions based on picking a knapsack with 1) minimum residual capacity, 2) maximum residual capacity, or 3) minimum surplus demand, which are called *greedy-minrc, greedy-maxrc*, and *greedy-minsd*, respectively. Surplus demand of a knapsack i is the total weight of unassigned items in B_i minus the residual capacity of the knapsack. The reason for picking a knapsack with minimum surplus demand is that the smaller the surplus demand, the more likely is the item to fit the knapsack. While greedy-maxrc and greedy-minsd heuristics tend to maximize assignments without considering waste, the greedy-minrc heuristic tends to minimize waste in addition.

Successive Assignment Heuristic

This is another greedy heuristic, where rather than picking a knapsack for each item or picking items for each knapsack one by one, items are matched to knapsacks successively with the objective of maximizing total weight of items matched. At each iteration a maximum weight bipartite matching (assignment) problem is solved on a bipartite graph in which edge (i, j) exists with weight w_j only if w_j does not exceed the residual capacity of knapsack i. Initially the bipartite graph G is used. The assignments given by the maximum weight bipartite matching solution are performed. Then, the bipartite graph is updated by deleting all assigned nodes, all edges (i, j) for which w_j exceeds the residual capacity of knapsack j, and nodes with degree zero. The heuristic is repeated until the graph has no remaining edge. This heuristic is called *successive-assign*.

Randomized Heuristics

The greedy heuristics can be modified randomly in order to break the pattern of greedy choices. Suppose item j is picked in any of the greedy heuristics. The item will be considered for assignment with a probability p_j. After running through all items, the heuristic is repeated with $p_j = 1$, $\forall j$ in order to assign the remaining items. There are two versions based on the choice of p_j. In the first one p_j is proportional to the weight of item j. That is, $p_j = (w_j/\text{average weight of an item}) * C$, where C is a constant factor, so that items with larger weight are more likely to be picked. We call this heuristic *random-weight.* In the second version, called *random-degree*, $p_j = 1 - (deg(j)/\text{average degree in } G) * C$, where $deg(j)$ is the degree of node j in G. The idea is to defer the assignment of items with more choices for knapsacks and to give priority to items that may go into only a few knapsacks.

Heuristics Based on the LP Relaxation of SMK

These heuristics solve the LP relaxation of the problem for the single objective of maximizing total assigned weight and then construct a feasible solution by rounding the fractional LP solution.

The IP formulation of SMK is as follows.

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

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.

Although the LP relaxation of the multiple knapsack problem has a solution that can be constructed easily in $O(n)$ time[3], the LP relaxation solution of the sparse problem cannot be identified immediately. However, the relaxation can still be solved efficiently by a maximum flow algorithm. The continuous problem reduces to a maximum flow problem on a directed graph constructed from the bigraph G as follows. Each edge (j,i) of G is directed from the item node j to the knapsack node i and is assigned capacity w_j. A source node s is connected to each item node j via an arc (s,j) with capacity w_j. In addition a sink node t is connected to each knapsack node i via an arc (i,t) with capacity c_i. Then, the maximum flow from s to t equals the LP relaxation value and the amount of flow on arc (j,i) divided by w_j gives the value of x_{ij}. Thus, if flow on (j,i) equals w_j, i.e. $x_{ij} = 1$, then item j is assigned to knapsack i. If $0 < x_{ij} < 1$, the variable is said to be fractional (in the corresponding solution).

There are two versions of the heuristic. In the first one, a simple greedy heuristic is used to assign items corresponding to the fractional variables and all the remaining

[3]It amounts to solving the LP relaxation of a single 0-1 knapsack problem.

items. In the second version, the fractional variables are sorted in non-increasing order of their values, and for each fractional variable in the order, the assignment is done if it is feasible. Then, the remaining items are assigned greedily as in the first version. These two heuristics are called *lp-greedy* and *lp-round*.

4.4 Improver Heuristics

Local Exchange Heuristics

These local exchange heuristics aim at improving both of the objectives, and are repeated until no more improvement occurs.

1. Exchange Items Assigned to Different Knapsacks

   Consider all pairs of items assigned to different knapsacks. Swap the two items, if the swap is feasible and allows an unassigned item to be assigned to one of the two knapsacks. If the exchange of items is performed, pick the knapsack whose residual capacity has just increased and repetitively assign the item with maximum weight to it, if the assignment is feasible. This heuristic is called *exchange*.

2. Replace Assigned Items with Unassigned Items

   Replace an assigned item or a pair of assigned items, with a single unassigned item or a pair of unassigned items of larger weight. These heuristics are called *replace-single* and *replace-pair* based on whether a single item or a pair of items are replaced.

3. Rearrange

   Rearrange assigned items in different knapsacks to aggregate residual capacity into one knapsack and then use the aggregated capacity to assign new items. Swap two items assigned to two different knapsacks, if the exchange is feasible and the maximum residual capacity (over all knapsacks) increases. After all pairs of items have been considered, repeat assigning a new item with maximum weight to the knapsack with maximum residual capacity, if feasible. This heuristic is called *rearrange*.

Heuristics to Reduce Waste

1. Empty Under-utilized Knapsacks:

   This is a randomized heuristic. Two parameters are picked randomly: minimum allowable utilization and maximum allowable percentage decrease in assigned weight. Cancel assignments in all knapsacks with utilization less than minimum allowable utilization, if the decrease in weight is less than the maximum allowable percentage. This heuristic is called *empty*.

2. Empty Knapsacks Randomly and Reassign Items

   For all knapsacks that are utilized, empty the knapsack with probability (1 - utilization of the knapsack). Then, reassign items by the greedy-minrc heuristic. This heuristic is called *empty-and-reassign.*

3. Pack Again

   These are variable-sized bin packing heuristics. Cancel assignments in all knapsacks. Reassign originally assigned items by first fit decreasing or best fit decreasing heuristics (i.e. the greedy-item or greedy-minrc heuristics). These heuristics are called *pack-again-ffd*, and *pack-again-bfd*, respectively.

Heuristics to Increase Assigned Weight

Any of the constructor heuristics can be used as an improver to assign the remaining items, if feasible. In our implementations we used the simple greedy-item heuristic for this purpose and we refer to it as *assign-remaining.*

5 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.

5.1 Data

We used real data from an inventory application problem in Steel Mill Industry [KDTL97]. 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 1.

5.2 Implementation

Individual heuristics

We coded the heuristics presented in Section 4 in C++ language using the LEDA library [NU95] 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

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 1: 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.

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. These statistics and the chosen cycle lengths are given in Table 5 in Appendix.

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 general 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 in Section 4. 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 MKBP as it combines multiple knapsack and bin packing aspects. The IP formulation is as follows.

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

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 MKBP problems have the same optimal value because an optimal solution to the LP relaxation of MKBP will have zero waste. Hence, we refer to the relaxation of both problems by "the LP relaxation".

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 a general integer programming solver, $CPLEX$4.0.3 [CO94] using the default settings. 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 2. The running times given in Table 2 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 2: 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 more 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 (as seen in Table 2). In addition, the cooperative strategy has the advantage of generating many non-dominated solutions for the two objectives while IP formulations specify one objective function.

5.3 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 3. 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 4. We observe a significant difference in (assigned weight - waste) of the solutions generated by the cooperative strategy versus those generated by individual heuristics, especially 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

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 3: 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 2). Waste % is the ratio of the unused capacity to the total capacity of utilized knapsacks in percentage. Cpu time is given in seconds.

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 capable 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. Plots of the non-dominated solutions generated by the A-team implementation and the individual heuristics are given in Figures 2, 3, and 4 in the Appendix. Note that for problems d1 and d3, which are less sparse compared to the other problems, solutions with large assigned weight have small waste at the same time, so not many non-dominated solutions were generated. This can be attributed to the fact that the two objectives are almost parallel for these problems.

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 4: 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 2. Cpu time is given in seconds.

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 running time of each individual heuristic is given in Table 6 in Appendix. 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 3 and 4. 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.

Concatenation of Heuristics in the A-team Implementation

By examining the non-dominated solutions on the Pareto frontier (shown in Figures 2-4) we can identify the heuristics and the sequence in which they were applied to yield a particular solution. Naturally the question arises whether it is really necessary to randomize the sequence in which these heuristics act on each others' solutions. If it is possible to identify one or more sequences (or concatenation of heuristics) which yield the Pareto-frontier for all the problem instances, then we might abandon a stochastic control strategy for constructing these solutions. In this section we show that the concatenation of heuristics used to generate the Pareto-frontier varies significantly by problem instance. This illustrates the need to tailor the solution strategy by problem instance which is automated by the stochastic control strategy adopted in this paper.

For each non-dominated solution generated by the A-team implementation, we traced the heuristics whose output solutions were used to obtain the non-dominated solution. We can conclude that some heuristics were more effective and were repeated more. Nevertheless, we observed no regular patterns in the sequence of agents called across different problem instances. This justifies the use of an A-team approach for the concatenation of heuristics as opposed to identifying some effective patterns and using these patterns instead.

For the A-team implementation, the total number of occurrences of each heuristic which participates in the generation of non-dominated solutions (for each problem instance) is given in Table 7 in the Appendix. We see that lp-round has been the most frequently used constructor. Local exchange heuristics replace-single and replace-pair, and waste reduction heuristics empty and empty-and-reassign were the most effective improvers in generating non-dominated solutions. We also give the heuristics that output the non-dominated solutions of the individual runs in Table 8 for comparison purposes. The statistics presented in these tables clearly indicate that the patterns used for constructing the Pareto-frontier varied significantly across problem instances. Figures 5-8 in the Appendix illustrate the construction of the non-dominated solutions generated by the A-team implementation.

6 Conclusions

In this paper we introduced a bicriteria sparse multiple knapsack problem. We characterized the computational complexity of this problem experimentally by computing the gap of the LP relaxation to the best derivable bound and showed that increased sparsity makes the problem more difficult. Integer programming techniques were unable to generate solutions in a reasonable amount of time. A new cooperative problem solving technique was introduced which used a collection of fast heuristic algorithms embedded in a cooperative problem solving architecture called A-Teams. The solutions generated using this approach were shown to be superior to those derived from using these heuristics individually and to the feasible solutions derived from integer programming techniques. The cooperative problem solving strategy is shown as an alternative approach for automating the design of heuristics for hard combinatorial problems.

References

[CGJ84] E.G. Coffman, M.R. Garey, and D.S. Johnson. Approximation algorithms for bin-packing: An updated survey. In G. Ausiello, M. Lucertini, and P. Serafini, editors, *Algorithm Design for Computer System Design*, pages 49 – 106. Springer-Verlag, Wien, 1984.

[CGJ97] E.G. Coffman, M.R. Garey, and D.S. Johnson. Approximation algorithms for bin-packing: A survey. In D.S. Hochbaum, editor, *Approximation Algorithms for NP-hard Problems*, pages 46 – 93. PWS Publishing Company, Boston, 1997.

[CO94] Inc. CPLEX Optimization. *Using the $CPLEX^{TM}$ Linear Optimizer*. 1994.

[FL86] D.K. Friesen and M.A. Langston. Variable sized bin packing. *SIAM J. Computing*, 15:222 – 230, 1986.

[FMW96] C.E. Ferreira, A. Martin, and R. Weismantel. Solving multiple knapsack problems by cutting planes. *SIAM J. Optimization*, 6(3):858 – 877, 1996.

[GJ79] M.R. Garey and D.S. Johnson. *Computers and Intractibility:A Guide to the Theory of NP-Completeness.* W.H. Freeman and Co., San Francisco, 1979.

[HF78] M.S. Hung and J.C. Fisk. An algorithm for 0-1 multiple knapsack problems. *Naval Res. Logist. Quarterly*, 24:571–579, 1978.

[Kar72] R.M. Karp. Reducibility among combinatorial problems. In R.E. Miller and J.W. Thatcher, editors, *Complexity of Computer Computations*, pages 85 – 103. Plenum Press, New York, 1972.

[KDTL97] 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, Yorktown Heights, NY, 1997.

[LMHM96] H.S. Lee, S. Murthy, S.W. Haider, and D. Morse. Primary production scheduling at steel-making industries. *IBM Journal of Research and Development*, ?(?):–, 1996.

[MARW97] S. Murthy, R. Akkiraju, J. Rachlin, and F. Wu. Agent-based cooperative scheduling. Technical Report ?, IBM T. J. Watson Research Center, 1997.

[MT80] S. Martello and P. Toth. Solution of the zero-one multiple knapsack problem. *Euro. J. Oper. Res.*, 4:322–329, 1980.

[MT81a] S. Martello and P. Toth. A bound and bound algorithm for the zero-one multiple knapsack problem. *Discrete Applied Math.*, 3:275–288, 1981.

[MT81b] S. Martello and P. Toth. Heuristic algorithms for the multiple knapsack problem. *Computing*, 27:93–112, 1981.

[MT89] S. Martello and P. Toth. *Knapsack Problems.* John Wiley and Sons,Ltd., New York, 1989.

[MT90] S. Martello and P. Toth. Lower bounds and reduction procedures for the bin packing problem. *Discrete Applied Math.*, 28:59–70, 1990.

[NU95] S. Näher and C. Uhrig. Leda user manual (version r 3.2). Research Report MPI-I-95-1-002, Max-Planck-Institute für Informatik, Saarbrücken, Germany, 1995.

[TdS93] S.N. Talukdar and P. de Souza. Asynchronous organizations for multi-algorithm problems. pages 286–293. Proceeding of 8th SIGAPP Symposium on Applied Computings, Feb. 1993.

[TdSM93] S.N. Talukdar, P. de Souza, and S. Murthy. Organizations for computer-based agents. *Engineering Intelligent Systems*, 1(2):–, 1993.

Appendix

Data		100	500	1000	1500	2000	3000	Chosen
	I	1	1	2	1	1	2	
d1	II	631.35	628.03	594.25	631.35	631.35	559.03	1500
	III	631.35	628.03	627.71	631.35	631.35	631.35	
	I	6	8	10	13	10	9	
d2	II	326.55	342.74	361.44	339.51	366.75	365.34	1500
	III	345.65	397.79	396.73	396.29	396.73	396.72	
	I	1	2	1	4	4	5	
d3	II	377.06	365.10	377.52	359.99	347.04	354.39	1500
	III	377.06	377.56	377.52	377.74	377.52	377.54	
	I	9	13	19	16	26	27	
d4	II	500.49	544.09	533.71	533.81	539.57	545.01	3000
	III	564.82	582.50	584.69	593.54	585.65	597.10	
	I	4	11	13	19	18	21	
d5	II	444.55	426.11	421.73	384.63	453.75	411.19	2000
	III	482.81	493.75	495.41	493.58	495.65	495.27	
	I	4	6	6	3	5	6	
d6	II	303.52	330.57	332.93	351.21	311.54	331.84	1000
	III	365.17	369.07	375.17	375.17	375.17	375.17	

Table 5: Convergence of the A-team Implementation in terms of I) Number of Non-dominated Solutions, II) Average Assigned Weight - Waste, and III) Maximum Assigned Weight - Waste. Columns 100-3000 denote number of cycles of heuristics. The last column denotes the cycle length selected for the implementation.

	Constructors									
Data	LPR	LPG	GAR	GIR	GSD	GK	GI	SA	RD	RW
d1	1.31	1.14	0.02	0.02	0.06	0.63	0.62	4.25	0.14	0.37
d2	0.22	0.19	0.00	0.00	0.01	0.15	0.14	0.32	0.03	0.02
d3	0.85	0.72	0.01	0.01	0.04	0.34	0.33	2.59	0.10	0.30
d4	0.60	0.54	0.01	0.00	0.01	0.37	0.37	0.71	0.05	0.09
d5	0.49	0.44	0.00	0.01	0.01	0.30	0.29	0.72	0.05	0.08
d6	0.20	0.17	0.00	0.01	0.01	0.09	0.09	0.36	0.03	0.04

	Improvers							
Data	RS	RP	EM	EMR	EX	RE	PF	PB
d1	24.43	4.32	0.00	0.01	0.26	0.12	0.09	0.02
d2	0.24	0.02	0.01	0.01	0.04	0.04	0.04	0.00
d3	44.99	21.90	0.01	0.01	0.03	0.04	0.04	0.01
d4	0.45	0.01	0.00	0.01	0.07	0.07	0.07	0.01
d5	0.41	0.02	0.01	0.02	0.07	0.07	0.06	0.01
d6	0.47	0.01	0.01	0.01	0.06	0.03	0.03	0.01

Constructors	Improvers
Greedy-item : GI	Replace-single : RS
Greedy-knapsack : GK	Replace-pair : RP
Greedy-maxrc : GAR	Rearrange : RE
Greedy-minrc : GIR	Exchange : EX
Greedy-minsd : GSD	Empty : EM
Random-weight : RW	Empty-and-reassign : EMR
Random-degree : RD	Assign-remaining : AR
Successive-assign : SA	Pack-again-bfd : PB
LP-greedy : LPG	Pack-again-ffd : PF
LP-round: LPR	

Table 6: The running times of the heuristics in seconds of cpu time.

	Constructors							Improvers							
Data	LPR	GAR	GIR	GSD	GK	SA	RD	RS	RP	EM	EMR	EX	AR	PF	PB
d1	1		1					4	2						1
d2	1	1						13	8	14	3	2	1	1	
d3				1				2	3	2			1		
d4	1					1		15	5	26	5	3	1	1	
d5	1	1					1	12	7	27	7	2	2	1	
d6	1		1		1			3	5	6	1	1	1		
Total	5	2	2	1	1	1	1	49	30	75	16	8	6	3	1

Table 7: The frequency of the heuristics that yielded non-dominated solutions of the A-team implementation. Note that the heuristics that were never used do not exist in the table.

	Constructors								Improvers		
Data	LPG	GAR	GIR	GSD	GK	GI	RD	RW	RS	EM	EMR
d1							1				
d2		1	1		1	1			1	1	
d3									1		
d4				1				1	1	1	1
d5									1		
d6	1								1		
Total	1	1	1	1	1	1	1	1	5	2	1

Table 8: The heuristics that output non-dominated solutions of the individual runs. Note that for data d2, the solutions output by GI, GK and GIR heuristics are the same, and there exists a total of 4 non-dominated solutions.

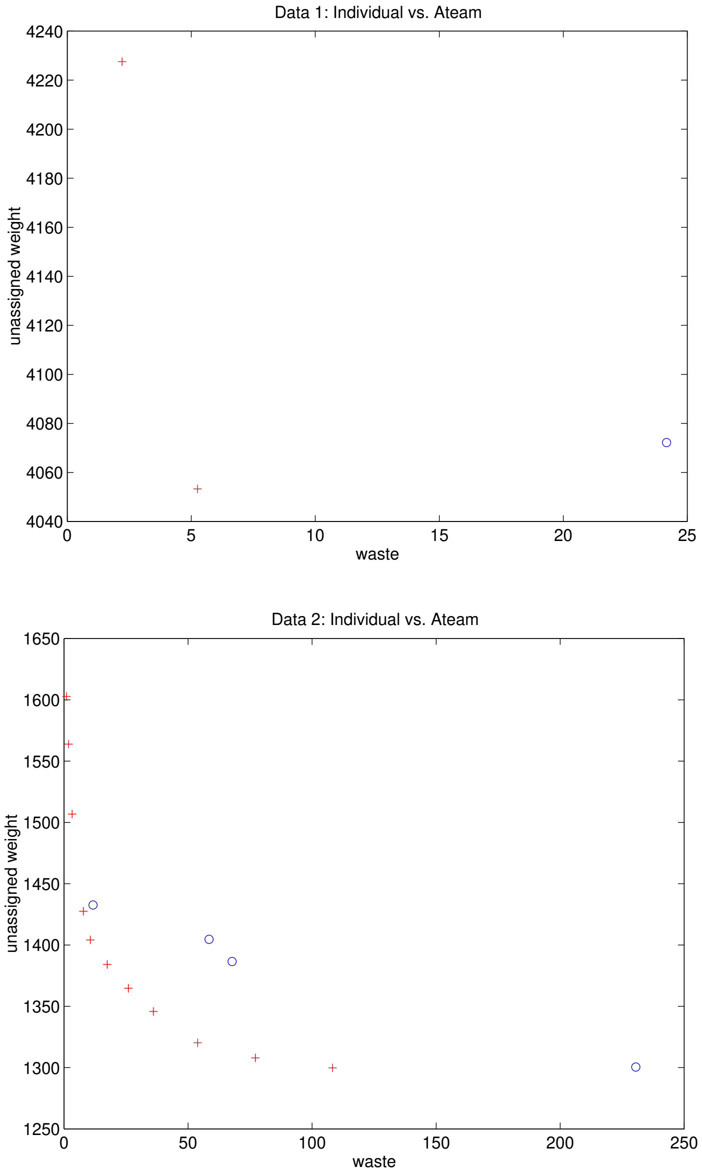


Figure 2: Non-dominated solutions generated by the A-team implementation (+) and individual runs (o) for data sets 1 and 2. The objective of maximizing assigned weight is converted to minimizing unassigned weight for presentation purposes.

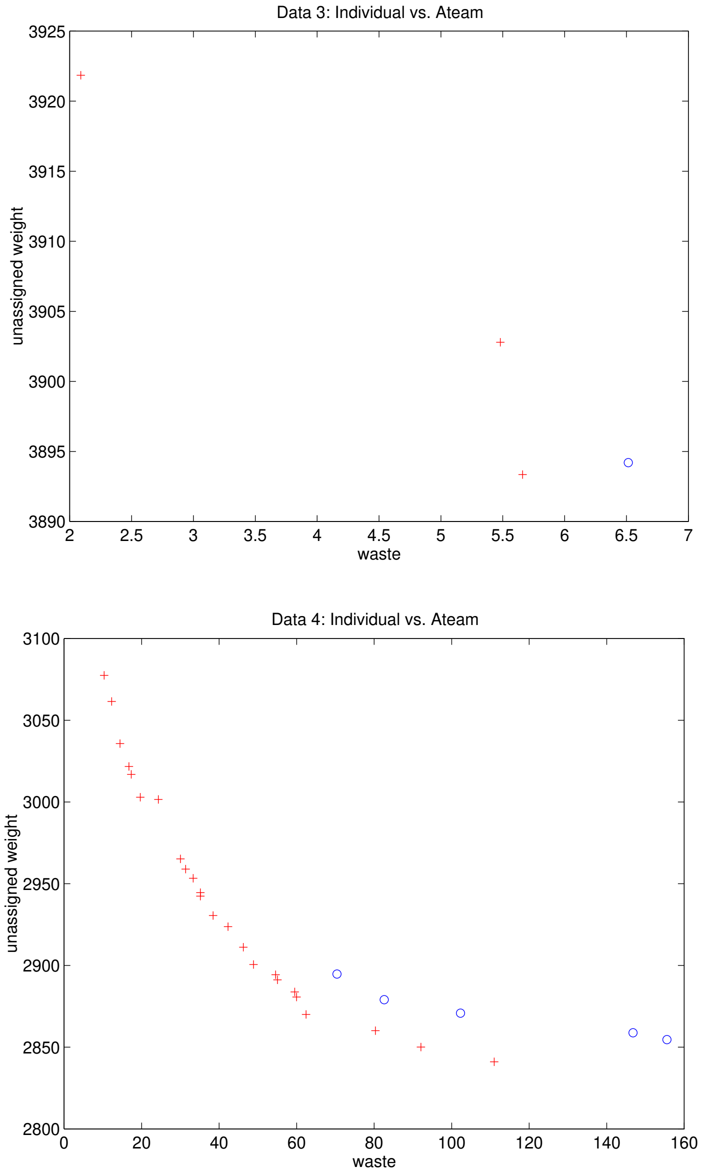
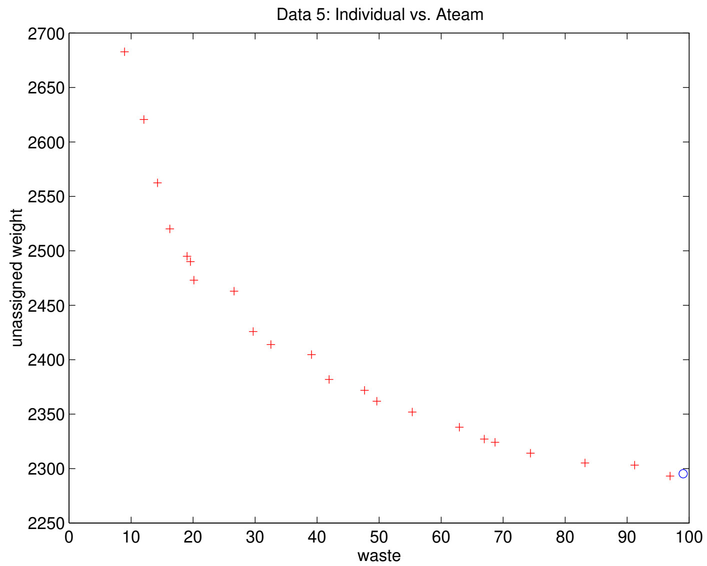


Figure 3: Non-dominated solutions generated by the A-team implementation (+) and individual runs (o) for data sets 3 and 4. The objective of maximizing assigned weight is converted to minimizing unassigned weight for presentation purposes.



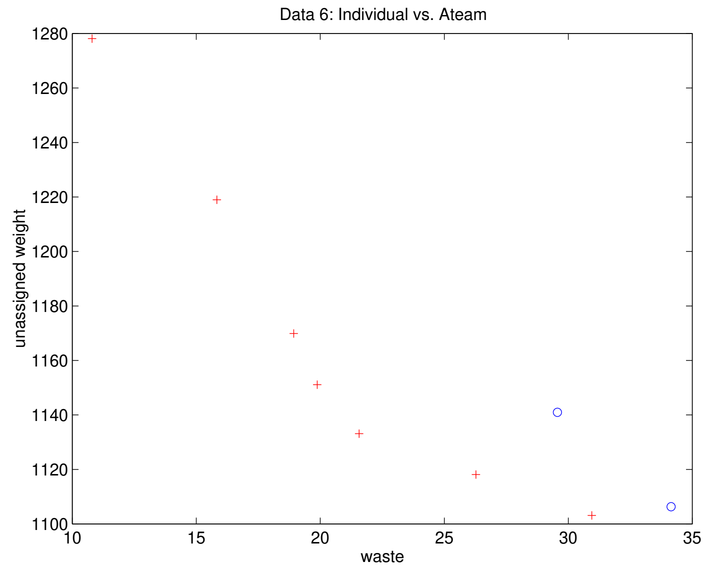


Figure 4: Non-dominated solutions generated by the A-team implementation (+) and individual runs (o) for data sets 5 and 6. The objective of maximizing assigned weight is converted to minimizing unassigned weight for presentation purposes.

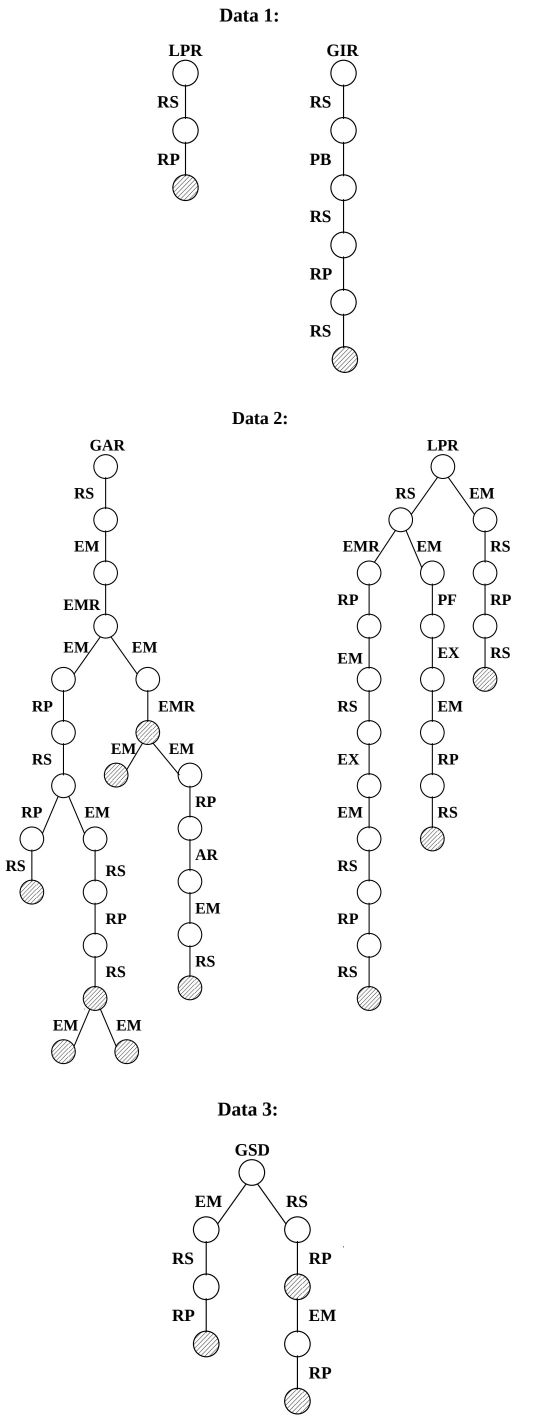


Figure 5: Trace of non-dominated solutions generated by the A-team implementation for data sets 1,2 and 3. Shaded nodes represent non-dominated solutions and empty nodes represent intermediate solutions. For each edge connecting two nodes, the code for the heuristic that output the solution represented by the following node is given.

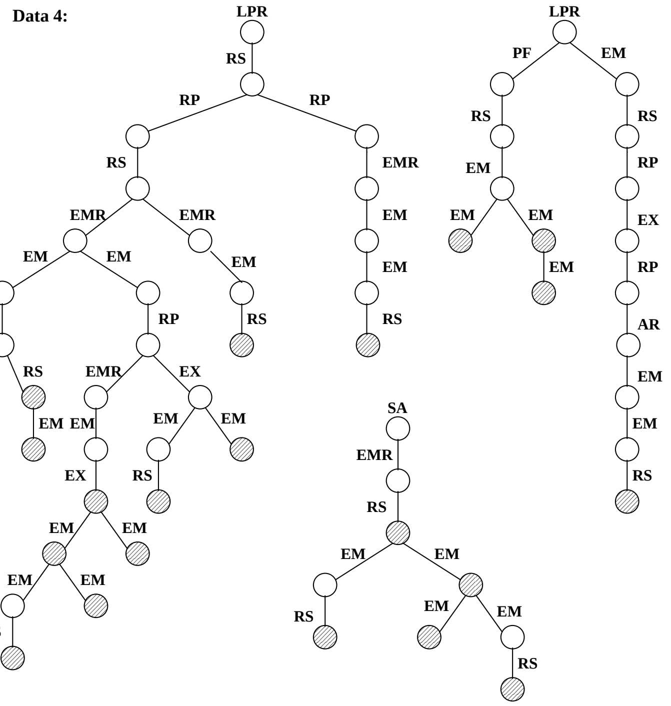


Figure 6: Trace of non-dominated solutions generated by the A-team implementation for data set 4.

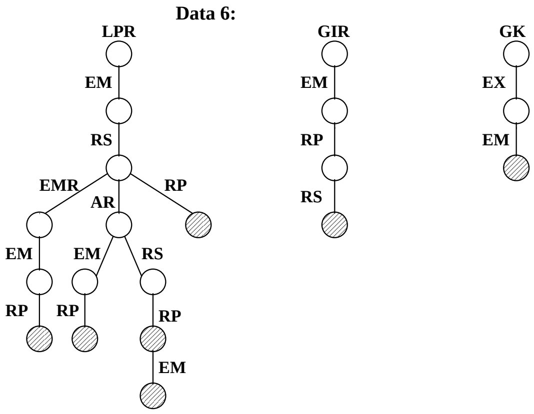


Figure 7: Trace of non-dominated solutions generated by the A-team implementation for data set 6.

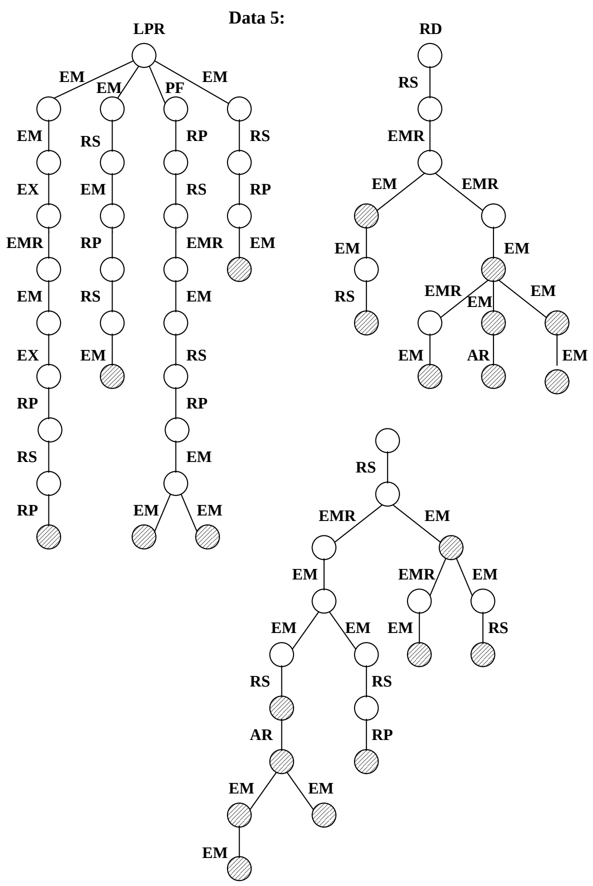


Figure 8: Trace of non-dominated solutions generated by the A-team implementation for data set 5.