APPROACHES FOR SPECIAL MULTIOBJECTIVE COMBINATORIAL OPTIMIZATION PROBLEMS WITH SIDE CONSTRAINTS

A THESIS SUBMITTED TO
THE GRADUATE SCHOOL OF NATURAL AND APPLIED SCIENCES
OF
MIDDLE EAST TECHNICAL UNIVERSITY

BY

BANU AKIN

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR
THE DEGREE OF MASTER OF SCIENCE
IN
INDUSTRIAL ENGINEERING

AUGUST 2012

Approval of the thesis:

APPROACHES FOR SPECIAL MULTIOBJECTIVE COMBINATORIAL OPTIMIZATION PROBLEMS WITH SIDE CONSTRAINTS

submitted by **BANU AKIN** in partial fulfillment of the requirements for the degree of **Master of Science in Industrial Engineering Department, Middle East Technical University** by,

Prof. Dr. Canan Özgen _____
Dean, Graduate School of **Natural and Applied Sciences**

Prof. Dr. Sinan Kayalıgil _____
Head of Department, **Industrial Engineering**

Prof. Dr. Murat Köksalan _____
Supervisor, **Industrial Engineering Dept., METU**

Examining Committee Members:

Prof. Dr. Meral Azizoğlu _____
Industrial Engineering Dept., METU

Prof. Dr. Murat Köksalan _____
Industrial Engineering Dept., METU

Assoc. Prof. Dr. Yasemin Serin _____
Industrial Engineering Dept., METU

Assist. Prof. Dr. Sinan Gürel _____
Industrial Engineering Dept., METU

Assist. Prof. Dr. Banu Lokman _____
Industrial Engineering Dept., TEDU

Date:

I hereby declare that all information in this document has been obtained and presented in accordance with academic rules and ethical conduct. I also declare that, as required by these rules and conduct, I have fully cited and referenced all material and results that are not original to this work.

Name, Last Name: **Banu AKIN**

Signature:

ABSTRACT

APPROACHES FOR SPECIAL MULTIOBJECTIVE COMBINATORIAL OPTIMIZATION PROBLEMS WITH SIDE CONSTRAINTS

Akın, Banu

M.Sc., Department of Industrial Engineering

Supervisor: Prof. Dr. Murat Köksalan

August 2012, 66 pages

We propose a generic algorithm based on branch-and-bound to generate all efficient solutions of multiobjective combinatorial optimization (MOCO) problems. We present an algorithm specific to multiobjective 0-1 Knapsack Problem based on the generic algorithm. We test the performance of our algorithm on randomly generated sample problems against IBM ILOG CPLEX and we obtain better performance using a problem specific algorithm. We develop a heuristic algorithm by incorporating memory limitations at the expense of solution quality to overcome memory issues of the exact algorithm.

Keywords: Multiobjective, combinatorial optimization, knapsack problems.

ÖZ

EK KISITLARI OLAN ÖZEL ÇOK AMAÇLI KOMBİNATORYAL OPTİMİZASYON PROBLEMLERİNE YÖNELİK YAKLAŞIMLAR

Akın, Banu
Yüksek Lisans, Endüstri Mühendisliği Bölümü
Tez Yöneticisi: Prof. Dr. Murat Köksalan

Ağustos 2012, 66 sayfa

Çok amaçlı kombinatoryal problemlerin tüm etkin çözümlerini bulan genel bir dal-sınır algoritması geliştirdik. Bu genel algoritmayı baz alarak, çok amaçlı 0-1 sırt çantası problemleri için özel bir algoritma sunduk. Bu algoritmanın IBM ILOG CPLEX'le kıyasladığımız performansını ölçmek için rastgele oluşturduğumuz test problemlerini çözdük ve probleme özgü algoritma kullanarak küçük problemler için daha iyi sonuçlar aldığımızı gözlemledik. Büyük problemlerde gözlemlediğimiz hafıza sıkıntısının üstesinden gelmek için, hafıza imkanlarına bağlı olarak sonuç kalitesinden ödün verecek şekilde bir sezgisel yöntem geliştirdik.

Anahtar Kelimeler: Çok amaçlı, kombinatoryal optimizasyon, sırt çantası problemleri.

To My Family

ACKNOWLEDGEMENTS

Since I cannot find any words other than *thanks* fitting better to convey my gratitude, I would simply like to thank to Prof. Dr. Murat Köksalan, not only for giving me a second and then a third chance to accomplish a Master of Science degree but also for inspiring me to apply for this graduate program in the first place.

I know I am lucky to have an unofficial co-advisor like Banu Lokman who could listen to you, cheer you up, and explain a thesis of 88 pages in less than 30 minutes.

As for logistics support and friendship, I am grateful to Murat Ilıman and Hakan Köseoğlu. Without Hakan's support and encouragement, I am sure to run the first successful instance would take months longer.

I would like to thank my colleagues for their support throughout the thesis. I am aware to have a working environment surrounded with friends is a rare chance and I appreciate it.

Last but not least, I thank my mother and father for their upraising and for being there whenever I need.

TABLE OF CONTENTS

LIST OF TABLES

TABLES

LIST OF FIGURES

FIGURES

CHAPTER 1

INTRODUCTION

In the last decades, interest in multi-objective decision making research area has increased. As they represent the real life problems better than classical single-objective models, they are harder to solve with the increasing complexity as a consequence. With Moore's law in effect, improvements in computational capability of hardware have caused more researchers to dig into multi-objective decision making and gave courage to deal with more than 2 even 3 objectives.

Among multiple criteria decision making problems (MCDM), multiple objective combinatorial optimization (MOCO) problems are widely studied since they represent many real life cases with integer decision variables like scheduling, assignment, budgeting, etc. As, single objective cases are already NP hard problems, finding the efficient frontier for MOCO versions is a difficult task. As we discuss in the literature review chapter, studies usually focus on problem specific exact algorithms aiming to find all efficient solutions or generic heuristic algorithms aiming at approximating the efficient frontier.

We propose a generic exact branch-and-bound algorithm to find all non-dominated solutions based on Lokman and Köksalan (2012). In order to implement the algorithm, we select multi-objective 0/1 knapsack problem with single knapsack because many real life problems can be reduced to knapsack problems, such as bin-packing, subset sum, etc. We compare the algorithm with Lokman and Köksalan (2012) and within the memory limitations of branch-and-bound, we observe improvement in terms of computational time.

As the computational complexity is an important issue for the exact algorithms, we also propose a heuristic approach by modifying the exact generic algorithm with a delta approach. We also use the same type of knapsack problems for computational experiments. Although we use a constant delta for computational experiments, with varying delta, one can grasp the sections of the solution space in which the decision maker is more interested. We use the hypervolume indicator suggested by Zitzler and Thiele (1998) to compare the performance of the heuristic algorithm relative to the efficient frontier. We observe good diversity and closeness of the solutions found as long as the memory limitations are not breached.

We look at the literature in Chapter 2 to see the approaches used for multiobjective combinatorial optimization (MOCO) problems and single and multiobjective knapsack problems as a subset of MOCO. Lokman and Köksalan (2012) propose a method for finding all efficient solutions of MOCO problems which is discussed in Chapter 3. As their approach is generic, we propose a generic branch-and-bound procedure to complement their algorithm in Chapter 4 and develop a problem specific branch-and-bound procedure for knapsack problems. To overcome the limitations of the exact algorithm, we suggest a heuristic approach in Chapter 5. We present the experimental results for both of the algorithms on randomly generated knapsack problems in Chapter 7 and discuss their variations. We finally conclude our research in Chapter 8.

CHAPTER 2

LITERATURE REVIEW

2.1 MULTIPLE-OBJECTIVE COMBINATORIAL OPTIMIZATION

As the multiple-objective optimization gains popularity, variations of approaches studying MOCO increase as well. An extensive survey of MOCO is presented by Ehrgott and Gandibleux (2000). Some aim to find all efficient solutions to a given MOCO problem and prefer exact approaches despite of computational complexity and yet some others prefer approximation methods despite of loss of precision. Due to the complexity issues, exact algorithms usually are problem specific and deals with bi-objective or tri-objective problems. Ehrgott and Gandibleux (2000) state many exact approaches for MOCO are extensions of the single objective versions. However, Gavanelli (2002) and Lukasiewycz et al. (2007) develop generic exact multi-objective algorithms not based on single objective versions.

Sylva and Crema (2004) propose a generic exact algorithm to find all efficient solutions by implementing iterative method for multiple objective integer linear programs. They utilize binary variables to add new constraints according to the results obtained by previous iterations. Later, Sylva and Crema (2007) studied larger problems by modifying their algorithm and discarding the aim for finding all efficient solutions. Lokman and Köksalan (2012) propose improvements on their model by reducing the complexity and utilizing efficient methods for keeping of information. Their algorithm is discussed in more detail in section 3.2.

Among the heuristic approaches, one of the most popular one is genetic algorithms. Evolutionary approaches used for multiobjective problems are extensively studied by Deb (2001). Very promising evolutionary algorithms (EAs) like PESA (Corne et al., 2000), NSGA II (Deb et al., 2002) and SPEA2 (Zitzler et al., 2001) have gained popularity in the last decade.

Due to the complexity of MOCO, some other approaches do not aim to find all efficient solutions but focus on decision maker's (DM) preferences and tries to find a subset of efficient solutions the DM might be interested in such as Köksalan and Phelps (2007).

Introduction of heuristic methods implies the need for a performance indicator in terms of closeness and diversity. Hypervolume metric suggested by Zitzler and Thiele (1998) is widely used for this purpose. A detailed description of the method is discussed in section 7.2.2.

2.2 KNAPSACK PROBLEMS

Being one of the basic problems in combinatorial optimization, both single-objective and multi-objective versions of the knapsack problem are widely studied. Martello and Toth have published a book named Knapsack Problems (KP), Algorithms and Computer Implementations in 1990 and it is practically the most cited book in this subject. They extensively study different variations of single-objective KP, including bin-packing, subset-sum, and multiple knapsack problems. They also study exact approaches like dynamic programming and branch-and-bound algorithms for different variations in addition to the approximate approaches like greedy and probabilistic algorithms. For the single-objective 0-1 KP they propose an improved branch-and-bound algorithm based on another branch-and-bound algorithm proposed by Horowitz and Sahni (1974).

Similar to the single objective versions, multiobjective knapsack problems (MOKP) are favorite subjects for studying. Most of the studies are specifically developed for the biobjective case. Ulungu and Teghem (1994) propose two-phases methods for finding all efficient solutions and Visee et al. (1998) improves the method for

MOKP. Like many two-phases methods, they find the supported non-dominated solutions in the first phase and find the rest using information obtained from the first phase. Klamroth and Wiecek (2000) propose a dynamic programming approach while and Gandibleux and Freville (2000) utilize a tabu search approach for MOKP. In the recent years, the number of studies developed for two or more objective cases has increased. Ulungu et al. (1999) extend their research towards heuristic methods like MOSA using a simulated annealing approach. Although their method is generic for two or more objectives, they show the results for the biobjective case. Zitzler and Thiele (1999) propose an EA for MOCO. They present benchmark problems for multi-objective multiple knapsack problems with 2, 3 and 4 objectives. These problems are widely used on generic algorithms, e.g. NSGA-II, SPEA2, and PESA.

CHAPTER 3

THEORETICAL BACKGROUND

We propose a branch-and-bound based approach using the algorithm proposed by Lokman and Köksalan (2012) for multi-objective combinatorial optimization problems. So, we define the basic concepts and models on which we base our proposed algorithm.

3.1 DEFINITONS

A generic MOCO problem can be defined as:

$$\begin{aligned} &\text{"max"}\{f_1(x), f_2(x),..., f_n(x)\} \\ &s.t. \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \text{(1: MOCOP)} \\ &x \in Z^m \end{aligned}$$

where

$f_j(x)$ is the j[th] objective function of a total number of n objective functions.

x represents the decision vector and is an element of decision space Z^m.

Maximization of the objective function is denoted in quotation marks due to the fact that MOCO problems generally do not have a unique *best* solution as the objectives usually conflict with each other.

An objective function vector $(f_1(\hat{x}), f_2(\hat{x}),..., f_n(\hat{x}))$ is ***dominated*** by $(f_1(\bar{x}), f_2(\bar{x}),..., f_n(\bar{x}))$ if there exist two distinct solutions $\bar{x}$ and $\hat{x}$ satisfying $f_j(\bar{x}) \geq f_j(\hat{x}) \quad \forall j$ and $f_i(\bar{x}) > f_i(\hat{x}) \quad \forall \hat{x} \in X$ for at least one i.

In this case, $\hat{x}$ is defined as an ***inefficient*** solution.

If no such solution vector $\bar{x}$ exists, $(f_1(\hat{x}), f_2(\hat{x}),..., f_n(\hat{x}))$ is said to be ***nondominated*** and $\hat{x}$ is an ***efficient*** solution.

If there does not exist any solution $\bar{x}$ satisfying $f_i(\bar{x}) > f_i(\hat{x})$, then $\hat{x}$ is said to be ***weakly efficient***. The set of weakly efficient solutions contains all efficient solutions and possibly some special inefficient solutions.

3.2 LOKMAN AND KÖKSALAN'S (L&K) APPROACH

Lokman and Köksalan (2012) develop a new approach to find all efficient solutions to any given MOCO problem. They improve the algorithm proposed by Sylva and Crema (2004) with two algorithms. We base our proposed approach on the second algorithm they develop. Hence, we use Lokman and Köksalan's algorithm to denote Algorithm 2 they developed.

The algorithm uses a systematical approach to find all the nondominated vectors by converting the multi-objective problem (MOCOP) to a single objective problem with side constraints. The feasible solution space of (MOCOP) is divided into several subspaces according to the information gathered by previous steps of the algorithm.

So the MOCOP is modified to:

$$\begin{aligned} \max z = \quad & f_i(x) + \sum_{j \neq i} \varepsilon \, f_j(x) \\ s..t. \quad & \\ & x \in Z^m \\ & f_j(x) \geq lb_j \qquad \forall j \neq i \end{aligned} \qquad \text{(2: MOCOM)}$$

First, one of the objectives (objective i) is selected to be the major objective function in the single-objective case. Epsilon (ε) is a small enough number multiplied by the remaining objective functions to prevent the method yielding weakly efficient solutions. Then, MOCOM is solved at each step of the algorithm with different lower bounds on remaining objective functions. These lower bounds are determined according to the non-dominated objective vectors obtained in the previous steps.

In order to illustrate the approach, also assume we have already obtained nondominated objective vector set of {(150, 160, 190), (106, 210, 174), (218, 159, 142)} so far for a 3-objective MOCOP as defined in section 3.1. Also, without loss of generality assume third objective is selected to be *objective i.* In this case, we can sort the vectors in the non-decreasing order according to one of the remaining objective values. We arbitrarily choose first objective and sort the non-dominated vectors accordingly as shown in Figure 1. If there exists at least one more non-dominated vector, it should be placed in one of the locations indicated by arrows. Arrows show the potential locations for the next non-dominated vector according to its first objective value. Squares show which objective value should be used to determine the lower bound on the first objective. Similarly, dotted circles show the vectors to be considered for determination of the lower bound on the second objective.

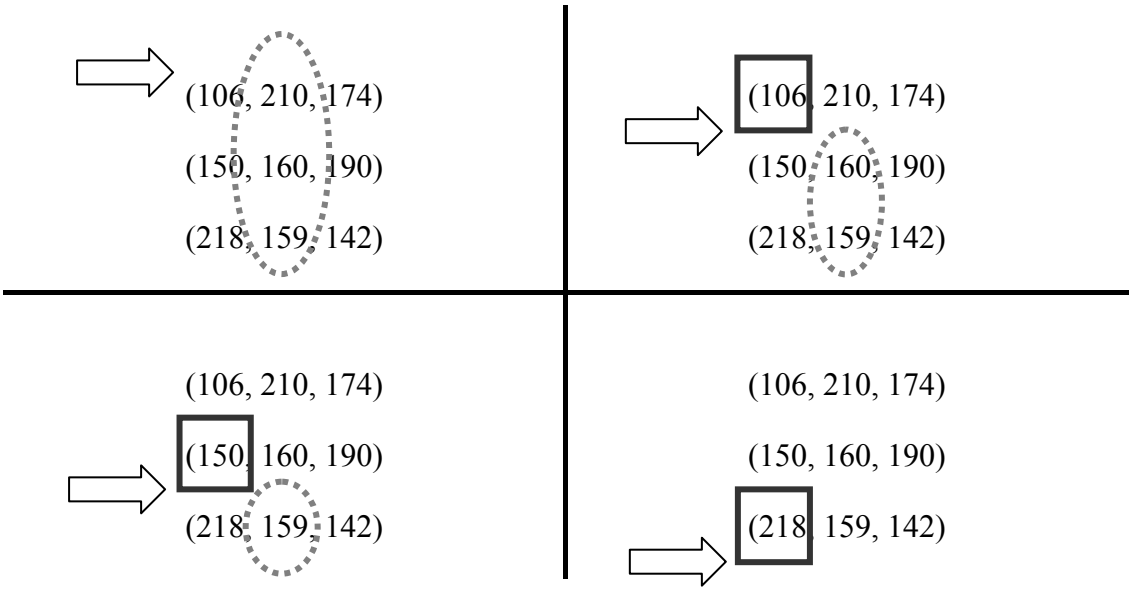

Figure 1 All alternative locations for the next non-dominated vector

As it is already sorted in non-increasing order, only the last vector above of the arrow is considered for the first objective. Since squares guarantee the candidate not to be

dominated by the ones above the arrow, only vectors below the arrow are considered for the second objective.

For each location, in order to hold the non-dominance requirements, one can determine the corresponding lower bounds in MOCOM. Since the vectors are already sorted in non-decreasing order for the first objective, first location indicates a better solution than the other vectors in terms of the second objective. If a vector in the second location exists, it should be better than the first vector in terms of the first objective and better than the rest of the list in terms of the second objective. Other cases can be analyzed similarly.

Corresponding lower bound constraints in MOCOM form of the above problem are given in Figure 2.

	$f_1(x) \geq 106+1$
$f_2(x) \geq \max(210,160,159)+1$	$f_2(x) \geq \max(160,159)+1$
$f_1(x) \geq 150+1$	$f_1(x) \geq 218+1$
$f_2(x) \geq 159+1$	

Figure 2 Lower bound constraints for the sample vector set

Although the number of cases increases with the increasing size of the list, Lokman and Köksalan (2012) show that at most two new MOCOM models are needed to be solved at each step, regardless of the list size. Their algorithm starts with no lower bounds on any of the objectives and proceeds according to the solution obtained. By populating the list one by one, Lokman and Köksalan eliminate the lower bound combinations that are already solved. Also, keeping the lower bounds resulting in infeasible solutions, the same or tighter lower bounds can be eliminated without solving new MOCOM models. In order to solve the models, they use IBM ILOG CPLEX software.

An illustrative example is studied in section 6.1 and Table 1 shows the complete list of non-dominated vectors in the order they are obtained along with their corresponding lower bound values.

This approach has been extended to four or more objectives. However, as the number of objectives increases, the complexity also increases and more than one sorted list should be kept. Details of the approach can be found in Lokman and Köksalan (2012).

CHAPTER 4

A GENERIC EXACT BRANCH-AND-BOUND ALGORITHM FOR MOCO

The main motivation behind the proposed approach is that problem specific models can improve the performance of the above mentioned Lokman and Köksalan's approach for the multi-objective combinatorial optimization problems. Being one of the most studied approaches for single objective combinatorial problems, we picked the branch-and-bound algorithm to modify for multi-objective case using the L&K approach. In addition to being widely used and studied, the structure of the optimization model solved at each step of L&K approach leads branch-and-bound based problem specific algorithms to be easily modified to represent the additional constraints of L&K approach unlike problem specific dynamic programming algorithms, network simplex algorithms, etc.

4.1 PROPOSED EXACT ALGORITHM

A branch-and bound procedure for any given single objective combinatorial optimization problem can be described as follows:

Without loss of generality assume the problem is a maximization problem over the feasible region of *S*.

$$\begin{aligned} &\max\ f(x) \\ &s..t. \\ &x \in S \\ &x_k \in \{0,1\} \end{aligned} \tag{3}$$

Any given branch-and-bound procedure divides the set Z^m into subsets or subproblems (**branching**) and computing the relevant upper bound and comparing with the best solution found so far (lower bound) to decide if each subset has a possibility to yield a better solution or not (**bounding**). After exhausting all the possible subproblems, best solution found so far is the optimal solution to the problem (3). The procedure can be represented as a *tree* and subproblems are represented as *nodes*.

In order to solve the multi-objective combinatorial optimization problem using Lokman and Köksalan's algorithm, we should solve the MOCOM model defined in section 3.2 in each step.

As the structure of the objective function remains the same as the single-objective case, the feasible set now has side constraints (actually lower bounds) on remaining objective functions. We can remove the side constraints and keep the branching phase the same (divide set of Z^m into subsets) and add those constraints as additional lower bound requirements to the bounding phase. Due to the similar structure of side constraints to the objective function, same method for computing upper bounds on the single objective can be used. By definition U_{jq}, upper bound of z_j for subset Z_q^m, is greater than or equal to the best value of z_j for subset Z_q^m. As a requirement of the problem, any feasible solution, including the optimal solution, should satisfy the side constraints, i.e. should be greater than or equal to the relevant lb_j values. Hence, any subset which does not satisfy all lower bound constraints of $U_{jq} \geq lb_j$ cannot lead to a feasible solution.

Regarding problem (4), for any subset Z_q^m and for any objective j other than objective i, let $z_{jq}^* = \max\{z_j(x) : x \in Z_q^m\}$

By definition, for all $x \in Z_q^m : U_{jq} \geq z_{jq}^*$

Also, for all $feasible\, x \in Z_q^m$: $z_{jq}^* \geq z_j \geq lb_j$ for all j. In other words, if $\bar{x} \in Z_q^m$ satisfies $z_j \geq lb_j$ for all j, $\bar{x}$ is feasible. Note that $z_j \geq lb_j$ constraints are actually the side constraints of the modified problem MOCOM.

Thus, for all feasible $x \in Z_q^m$, $U_{jq} \geq lb_j$ should also be satisfied.

In short, any branch-and-bound algorithm for a single objective combinatorial optimization problem can be adjusted to find all the non-dominated solutions of the multiobjective version of the same problem. The modified algorithm is called for each lower bound pairs determined by L&K approach. Without loss of generality, assume that the model to be solved at each step is in the form of problem MOCOM with positive coefficients. Branching (forward move), bounding (upper bound calculation) and backtracking steps are the same as the branch-and-bound algorithm to be modified. So, for each iteration in L&K approach:

Let A be the set of decision variables which are assigned to values. Let best solution so far is LB_i and lower bounds obtained from L&K algorithm are LB_j ($j \neq i$).

1. **Lokman and Köksalan's Algorithm:** Calculate LB_j according to L&K algorithm.

   If no further lower bounds are left to consider, **STOP**.

2. **Initialization:** Initially no decision variables are set: A={}

   Best solution LB_i is set to 0.

   Create first node and calculate respective upper bounds U_{jq}.

3. If $U_j \geq LB_j$ for all j,

   go to step 4.

   else

   go to **backtrack**.

4. If no more decision variables are left to decide (A^c={}),

   Update best solution as current solution and go to **backtrack**

else

Forward move: Create new node according to the original algorithm's set of rules. Update decision variable set A. Calculate upper bounds and go to step 3.

5. **Backtrack**: Backtrack according to the original algorithm's set of rules.
6. If all possible subproblems are considered

   if best solution is greater than 0,

   return objective function values of the best solution (z_j(best_solution)) and go to step 1.

   else

   return *infeasible* and go to step 1.

else

go to step 3.

The general procedure can be represented as the flow chart shown in Figure 3. Bold lines indicate modifications to the original branch-and-bound procedure. Basically the modified one calculates upper bounds for all objectives and initiates a backtracking move whenever one or more lower bound constraints are not satisfied.

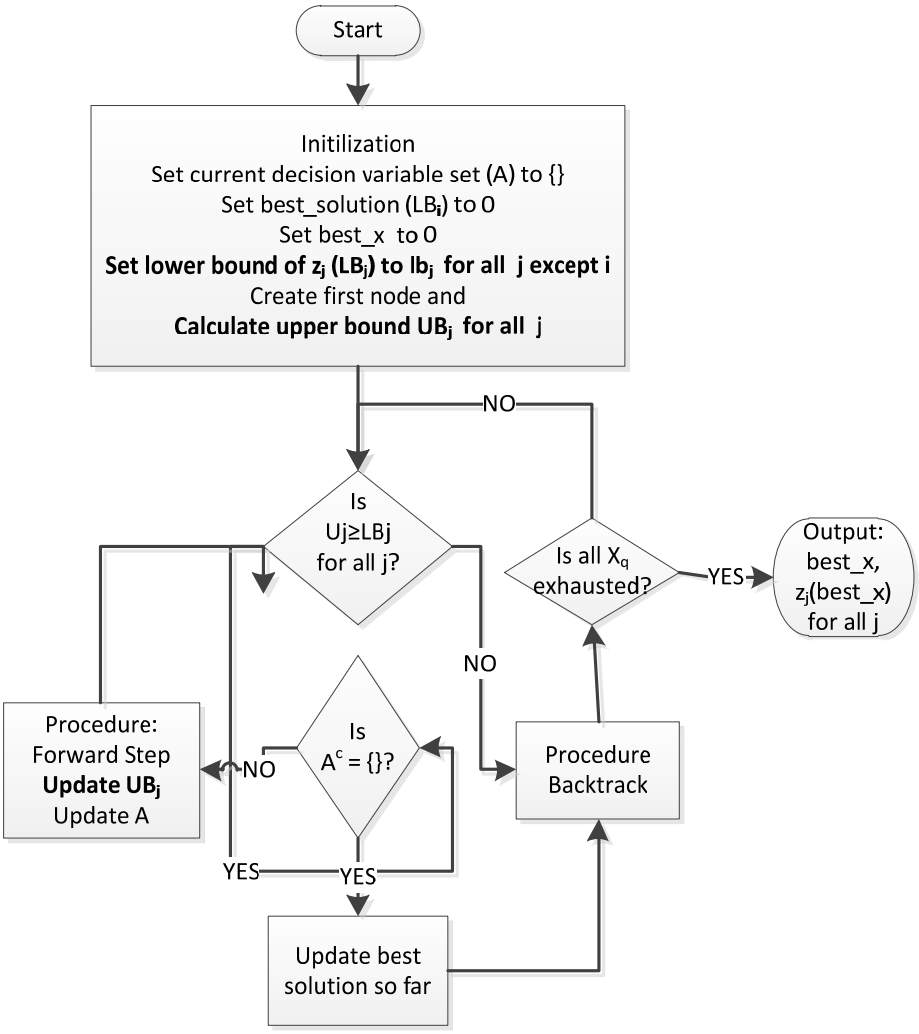

Figure 3 Modified Branch-and-Bound Procedure

4.2 IMPLEMENTATION: 0-1 KNAPSACK PROBLEM

In order to implement the model to a specific combinatorial optimization problem, we have chosen 0-1 Knapsack Problem (KP). It is a well studied problem with different variations such as subset-sum, change-making, bin-packing, etc. As indicated by Martello and Toth (1990), 0-1 KP represents many practical problems. Also, by being one of the simplest combinatorial optimization problems, it is often encountered as a subset while solving more complex problems. Due to these reasons, we studied the 0-1 KP which can be represented as:

$$\max \sum_k p_k x_k$$

s.t.

$$\sum_k w_k x_k \le C \qquad \text{(4: KP)}$$

$$x_k \in \{0,1\}$$

where p_k denotes profit of item k,

w_k denotes weight of item k,

C denotes capacity of the knapsack

x_k is 1 if the item is selected and 0 otherwise.

Without loss of generality one can assume

i. p_k, w_k, and C are greater than 0.

ii. items are ordered according to their profit value per unit weight:

$$\frac{p_1}{w_1} \ge \frac{p_2}{w_2} \ge \ldots \ge \frac{p_{K-1}}{w_{K-1}} \ge \frac{p_K}{w_K}$$

iii. no item weighs more than the capacity:

$$C \ge w_k \qquad \forall k$$

For each step in L&K algorithm the problem is transformed to the following form:

$$\max z = \sum_k p'_{ik} x_k$$

s.t.

$$\sum_k w_k x_k \le C$$

$$\sum_k p_{jk} x_k \ge lb_j \qquad \forall j \ne i \qquad \text{(5: KPM)}$$

$$x_k \in \{0,1\}$$

where

$$p'_{ik} = p_{ik} + \sum_{j \ne i} \varepsilon p_{jk} \qquad \forall k$$

In this case, p_{jk} denotes profit of item k for objective j. p'_{ik} denotes objective function coefficients of KPM modified to prevent weakly efficient solutions. Again we assume items are ordered according to their profit value (p'_{ik}) per unit weight (w_k).

4.2.1 The Horowitz-Sahni (H&S) Algorithm

Horowitz and Sahni (1974) proposed a basic algorithm to solve single-objective 0-1 knapsack problems. Considering the problem KP defined in (4.3), the Horowitz-Sahni algorithm flows as follows:

Forward Step: In this step items are inserted one by one in the ascending order as long as the capacity is not exceeded. Since the items are already sorted according to assumption (ii) in (4.3), this step represents a typical greedy move. Upper bound is compared to the best solution so far. If the upper bound indicates the possibility of a better solution, move *forward*. Else, perform *backtracking*.

Backtracking Step: Remove the last item inserted. Perform *forward* move.

When no items are left to be considered, the procedure is terminated. Best solution so far is the output and hence the solution of the problem.

Upper Bound: The upper bound used by Horowitz and Sahni is obtained by the linear programming relaxation of the integer programming problem and also known as the *Dantzig's bound*.

In this case the problem KP becomes continuous knapsack problem (KPC):

$$\begin{aligned} &\max \sum_k p_k x_k \\ &s.t. \\ &\sum_k w_k x_k \le C \\ &0 \le x_k \le 1 \end{aligned} \qquad \text{(6: KPC)}$$

The optimal solution to the continuous problem is proven to be:

$$
\begin{aligned}
&x'_k = 1 \qquad for \qquad k = 1,..., r-1 \\
&x'_k = 0 \qquad for \qquad k = r+1,..., K \\
&x'_r = \frac{C'}{w_r} \\
&where \\
&C' = C - \sum_{k=1}^{r-1} w_k \\
&r = \min\left\{ k : \sum_{j=1}^{k} w_j > C \right\}
\end{aligned}
\tag{7}
$$

The proof can be found in Martello and Toth (1990).

Since the items are already sorted according to assumption (iii) in (4.2), item *r* is the first item that does not fit into the knapsack. This item is called the critical item. Since partial inclusions are permitted with LP relaxation, remaining capacity is partially filled with the critical item to find the optimal solution.

As the LP relaxation is a natural upper bound for an IP problem, the upper bound used by Dantzig (1957) can be computed by finding the critical item with a complexity of O(n). As the profit and decision variables are integers for KP, largest integer not greater than the optimal solution value of KPC can be used as the upper bound of KP:

$$
U' = \left\lfloor \sum_{k=1}^{r-1} p_k x_k + \left(C - \sum_{k=1}^{r-1} w_k \right) \frac{p_r}{w_r} \right\rfloor \tag{8}
$$

4.2.2 The Martello-Toth (M&T) Algorithm

The algorithm proposed by Martello and Toth (1977) is an improvement on the H&S method. Basically they propose four improvements on upper bound, consecutive insertion of items, eliminating lower branches and parametric calculation of upper bounds.

1. ***Upper Bound:*** Instead of Dantzig's bound, they propose a tighter upper bound on KP using an inclusion and exclusion of the critical item.

$$U'' = \max\left(U^0, U^1\right)$$

where

$$U^0 = \left\lfloor \sum_{k=1}^{r-1} p_k + \left(C - \sum_{k=1}^{r-1} w_k \right) \frac{p_{r+1}}{w_{r+1}} \right\rfloor \tag{9}$$

$$U^1 = \left\lfloor \sum_{k=1}^{r-1} p_k + p_r - \left(w_r - \left(C - \sum_{k=1}^{r-1} w_k \right) \right) \frac{p_{r-1}}{w_{r-1}} \right\rfloor$$

Like the Dantzig's bound in (4.3.1), item *r* is the critical item. U^0 considers the case of excluding the critical item from the solution and fills the remaining capacity with the next item. On the other hand, U^1 includes the critical item and removes a portion of the previous item proportional to the excess in the capacity. The greatest of these bounds is an upper bound for KP. Similar to Dantzig's bound, complexity is O(n) for finding the critical item. The Dantzig's bound U' is dominated by U" and a detailed proof for bounds and domination can be found in Martello and Toth (1990).

2. ***Consecutive Insertion:*** Instead of inserting the items one by one during the forward move, they propose to insert consecutive items building the upper bound at once. As the H&S algorithm is a deep first branch-and-bound algorithm, by eliminating mid nodes unnecessary branching and backtracking are prevented.

3. ***Eliminating Lower Branches:*** Whenever the remaining capacity does not allow any more items to be inserted, forward move excludes all the remaining items. Thus, unnecessary branching and backtracking are again prevented.

4. ***Parametric Calculation of U'':*** Using some extra variables, the upper bound computation effort for each node is improved with the information stored.

4.2.3 Proposed Exact Algorithm

Our main problem is KPM defined in (4.2). Similar to Martello-Toth algorithm, we propose an algorithm based on Horowitz-Sahni algorithm. Due to the different properties of KPM with respect to KP, our algorithm applies some improvements of

M&T and adds some modifications to fit the special properties of KPM. Main differences to M&T procedure can be listed as:

1. ***Upper Bound:*** Dantzig's bound is modified instead of the bound used in M&T algorithm. As the problem KPM defined in (4.2) does not have integer objective function coefficients (p'_{ik}), it is not possible to round the upper bound to an integer value. However, the upper bounds for extra constraints representing the other objective functions have integer profit coefficients. Thus, Dantzig's bound is valid for objective functions represented as additional constraints.

   Calculation of bound U_j is given in (9). r_j denotes the critical item of objective j. As the profit per weight is different for each objective, sequence of the non increasing profit per weight is different for each objective. Since the item having a greater $\frac{p_{ik}}{w_k}$ value is more preferable for objective j, this sequence can be called a preference sequence. Preference sequence assumption (ii) in (4.3) only holds for the objective z of KPM. For all other objectives, the sequence should be calculated for once at the beginning and the critical r_j is calculated according to these preference sequences at each subset. Since calculating the preferences is actually sorting in non-increasing order, the complexity is O(n).

$$U_j = \begin{cases} \sum_{k=1}^{r_j-1} p'_{jk} x_k + \left(C - \sum_{k=1}^{r_j-1} w_k \right) \frac{p'_{jr_j}}{w_r} & for \;\; j = i \\ \left\lfloor \sum_{k=1}^{r_j-1} p_{jk} x_k + \left(C - \sum_{k=1}^{r_j-1} w_k \right) \frac{p_{jr_j}}{w_r} \right\rfloor & for \;\; j \neq i \end{cases} \qquad (10)$$

   Unlike M&T, for each node, not only the upper bound on objective z of KPM (U_i) is checked for violations but also the other objective functions are checked for upper bound violations using the U_j function($U_j \geq lb_j$). Even if

the best solution so far is not greater than U_i, in the case of violations of other bounds, backtracking step is applied.

2. ***Consecutive Insertion:*** Instead of using consecutive insertion method of M&T, we add nodes one by one, similar to H&S. Since the Lokman and Köksalan's algorithm searches the same solution space with different lower bound values obtained by non-dominated solutions found so far, the same branch-and-bound tree is searched for a different optimal solution in each step. Since there is only one optimal solution to KP, the consecutive insertion is an improvement for M&T procedure. However, we observed on sample problems, the nodes that are avoided to be created by consecutive insertion are eventually needed to be created in later steps.

3. ***Eliminating Lower Branches:*** As this improvement is for the capacity constraint which is the same in KP and KPM, we also eliminate the lower branches when the remaining current capacity is less than the weight of any of the remaining items.

Forward move and backtracking are the same as the M&T except for the above mentioned points. Optimal solution is found when backtracking is no more possible, i.e. no unconsidered node is left. This solution is one of the non-dominated solutions of the multi-objective KP. All objective functions (z_j) are calculated according to the optimal solution and considered as part of determining the next set of lower bounds of lb_j according to the Lokman and Köksalan's algorithm.

The general flow for the modified knapsack problem (KPM):

p_{jk}, p'_{ik} , w_k, C, lb_j are as defined in KPM in (4.2). U_j is as defined in (4.2.3). Variable l (level) indicates which item is being considered to include or exclude from the knapsack. Variable t is used for finding the level which backtrack step will jump. *best_x* is the vector to save best solution so far, *best_z* is the vector used to calculate the output of the subroutine, and *min_w* is the vector used to eliminate lower branches. It is used to keep the minimum weight among the items outside of the current solution. *cur_x* and *cur_z* are used to keep track of

the current solution. *cur_c* is the residual capacity after inclusion of the items in the current solution.

1. **Initialization I**: These steps will be performed only once.

   1.1. Find preference sequence for each j other than i ($pref_{jk}$: k^{th} position in the preference sequence of objective j).

   1.2. Find minimum weight for each level ($\min\_ w_l = \min\{w_k : k \geq l\}$). This information will be used for eliminating the lower bounds.

   1.3. Insert first node. Calculate and save U_j for all j.

2. **Lokman and Köksalan's Algorithm**: Calculate lb_j according to L&K algorithm.

   If no further lower bounds are left to consider, **STOP**.

3. **Start Subroutine**

   Initialization II: Following steps will be performed before the branch-and-bound procedure for each step of L&K algorithm, i.e. for each set of lower bounds.

   3.1. Initialize variables

   $lb_i := 0$
   $best\_x_k := 0 \quad \forall k$
   $cur\_x_k := 0 \quad \forall k$
   $cur\_z_j := 0 \quad \forall j$
   $cur\_c := C$
   $l := 1$
   $t := 0$

   3.2. If $U_j \geq lb_j$ for all j does not hold for at least one objective, go to **5.1 *(backtrack)***.

4. **Forward Move**

4.1. **while** ($cur\_c \geq w_l$ AND $U_j \geq lb_j$ $\quad for\ j \neq i$)

do

$cur\_x_l := 1$
$cur\_c := cur\_c - w_l$
$cur\_z_j := cur\_z_j + p_{jl} \quad for\ j \neq i$
$cur\_z_i := cur\_z_i + p'_{il}$
$l := l + 1$

If not already created for current solution (*cur_x*), insert new node on level *l*, calculate and save U_j for all *j*

If $U_j \geq lb_j$ for all j does not hold for at least one objective, go to **6 *(backtrack)***.

4.2. If $cur\_c < \min\_w_l$ (no more insertions are possible)

$l := K$

If not already created for current solution (*cur_x*), insert new node on level *l*, calculate and save U_j for all *j*

4.3. If current level, which is critical item, is not the last item, exclude it from the knapsack

If $l < K$

If not already created for current solution (*cur_x*), insert new node on level *l*, calculate and save U_j for all *j*

$l := l + 1$

If $U_j \geq lb_j$ for all j does not hold for at least one objective, go to **6 *(backtrack)***.

4.4. If current level (critical+1) is not the last item, repeat forward move.

If $l < K$ go to **4 (forward move).**

4.5. If $l = K$

If $cur\_c \geq w_l$ (If the last item is small enough, include in the knapsack)

$$cur\_x_l := 1$$
$$cur\_c := cur\_c - w_l$$
$$cur\_z_j := cur\_z_j + p_{jl} \qquad for\ j \neq i$$
$$cur\_z_i := cur\_z_i + p'_{il}$$

If not already created for current solution (*cur_x*), insert new node on level *l*, calculate and save U_j for all *j*

else (If the last item is not small enough, exclude from the knapsack)

If not already created for current solution (*cur_x*), insert new node on level *l*, calculate and save U_j for all *j*

5. **Update Best Solution So Far**

5.1. If $U_j \geq lb_j$ for all j

$$lb_i := cur\_z_i$$
$$best\_x_k := cur\_x_k \qquad \forall k$$

If $cur\_x_K := 1$ (If last item is included, throw it out before backtrack)

$$cur\_x_K := 0$$
$$cur\_c := cur\_c + w_K$$
$$cur\_z_j := cur\_z_j - p_{jK} \qquad for\ j \neq i$$
$$cur\_z_i := cur\_z_i - p'_{iK}$$

6. **Backtrack**

6.1. Find the item to throw out of the knapsack

Find t, $t = \max\{k < l : cur\_x_k = 1\}$

If $t > 0$

$cur\_x_t := 0$
$cur\_c := cur\_c + w_t$
$cur\_z_j := cur\_z_j - p_{jt} \quad for\ j \neq i$
$cur\_z_i := cur\_z_i - p'_{it}$
$l := t$

If not already created for current solution (cur_x), insert new node on level l, calculate and save U_j for all j

go to **4 (forward move)**.

else

go to **7 (subroutine return)**.

7. **Subroutine Return**

7.1. If $best\_x_k = 0 \quad \forall k$

No feasible solutions. **return** *infeasible*.

else

return $best\_z_j = \sum_k p_{jk} best\_x_k \quad \forall j$

go to **2**

CHAPTER 5

A GENERIC HEURISTIC BRANCH-AND-BOUND ALGORITHM FOR MOCO

As the number of objectives and number of variables increase, finding all of the efficient points gets harder. Also, decision maker might be interested in a specific region of the efficient frontier or might not be interested in solutions too close to each other. Because of these reasons, heuristic approaches are developed.

To overcome the memory drawback of branch-and-bound due to increase in size, we propose a heuristic approach by modifying the exact algorithm. The memory problems are discussed more deeply in computational results section.

5.1 PROPOSED HEURISTIC ALGORITHM

Moving away from the purpose of finding the entire efficient frontier, we propose a *delta* approach to obtain a relatively well distributed set of vectors, hopefully very close to the efficient frontier.

We modify the exact algorithm in two ways:

1. **Delta distance:** We make use of the systematical method of Lokman and Köksalan's approach for obtaining the efficient frontier. While finding the non-dominated vectors one by one, the bounds on the objectives are getting stricter, causing the feasible space for the problem MOCOM defined in section 3.2 to get smaller. Thus, the optimum solution of MOCOM found in the previous iteration can be also thought as a natural upper bound on the objective function of MOCOM for the current iteration. Using this

information, we propose to settle for the first feasible solution provided that it is at most delta units away from the previous solution. If the optimal solution to MOCOM at this iteration is even farther away than delta distance, the algorithm behaves like the original one as long as the second modification is not violated.

2. **Maximum number of nodes:** Due to the reasons to be discussed deeper in computational results section, we try to limit the memory usage of the algorithm, i.e. the number of nodes. Instead of limiting it based on the size of the problem, we propose to limit the maximum number of nodes to be created by the availability of the memory resources. Memory availability is an input and as the size of a single node is known, maximum number of nodes can be set and the algorithm can be adjusted to immediately return the best solution found up to that point. If no such solution is found, it returns infeasible. After that point, each iteration is allowed to search the tree nodes already created or create new nodes up to a number. After that limit is reached, the iteration is terminated similar to the previous condition.

These modifications are reflected to the algorithm defined in section 4.1 as follows:

Let UB be the value of the last feasible solution found by Lokman and Köksalan's algorithm in the previous iterations. It will be set to a sufficiently large number initially. An additional delta condition is added to the step 4, where the best solution found so far is updated. When the condition $UB_i - \Delta \leq LB_i$ holds, the branch-and-bound algorithm returns the current solution as the optimal solution to MOCOM.

To limit the number of nodes, let *flag* be the binary variable indicating if the predetermined limit L_1 is exceeded and let L_2 be the limit for each iteration after the limit L_1 is exceeded. In order to control those limits, we need two counters, *count_all* (for total number of nodes created) and *count* (for number of nodes created in the current iteration). Initially those variables are also 0. If the total number of nodes does not exceed L_1 yet, backtracking step is modified to return the best solution so far when this limit is reached ($count\_all \geq L_1$). *flag* is set to 1 and iterations are

limited to L_2 nodes at most from that point on. Similarly, the algorithm returns the best solution found so far when this limit is reached ($count \geq L_2$).

The complete heuristic algorithm of MOKP is given in Appendix A.

5.2 PROPOSED HEURISTIC ALGORITHM: IMPLEMENTATION

These modifications are reflected to the proposed exact algorithm for the multiobjective 0-1 Knapsack Problem defined in section 4.2.3. Only the steps considered in the previous section (update best solution and backtracking) are affected. The algorithm returns the best solution so far as the optimal solution when it is delta away from *UB*. Until the limit L_1 is reached, the backtracking step is the same as the exact algorithm. Once it is hit, the algorithm limits the number of new nodes to be created by L_2.

The complete heuristic algorithm of MOKP is given in Appendix B.

As long as the limit L_1 is not reached, the heuristic algorithm's complexity is exponential similar to the exact approach. When the limit is reached we simply prematurely terminate the branch-and-bound search. This limit is a given constant and it is independent from the number of nodes.

The heuristic algorithm is sensitive to the value of delta. In the case when we do not limit the heuristic algorithm, as delta gets smaller, the heuristic algorithm is expected to behave like the exact algorithm since it will not be able to find any feasible solutions delta units away from *UB*. When delta gets larger, the heuristic algorithm settles for poorer feasible solutions and we expect to miss a good portion of the non-dominated solutions and obtain dominated solutions far away from the efficient frontier. Delta can be thought of as a way of determining the decision maker's interest in the solution space, i.e. which distance (in objective i) between the solutions is sufficient for her to represent the solution space. With the introduction of dynamic delta altering in each iteration, different portions of the solution space can be searched more intensely than others, depending on the decision maker's preferences. In computational experiments, section 7.2.2, an approach to determine a constant delta is suggested.

At each iteration, the solution obtained might be the actual non-dominated solution of the corresponding L&K lower bounds, might be another non-dominated solution at most delta units away from the previous one, or might be a dominated solution. Thus, there is no guaranteed performance for the heuristic algorithm. Even without the limits in force, one can get delta units away in objective i from the efficient frontier. As the *UB,* objective i value of the solution obtained from the previous iteration, might be a dominated solution itself, the algorithm might actually be getting delta units away from the efficient frontier at each iteration in the worst case. Since the algorithm only checks the feasibility condition on other objectives, assessing a performance measure on those is not possible either.

CHAPTER 6

ILLUSTRATIVE EXAMPLES

6.1 AN ILLUSTRATIVE EXAMPLE FOR THE EXACT ALGORITM

In order to illustrate the model more clearly, we consider the following 3 objective 0-1 knapsack problem (P1) with 6 items:

$$
\begin{aligned}
&\text{"max"} \quad \{z_1, z_2, z_3\} \\
&s.t. \\
&21x_1 + 20x_2 + 56x_3 + 24x_4 + 51x_5 + 52x_6 \le 112 \\
&x_k \in \{0,1\}
\end{aligned}
\qquad \text{(11: P1)}
$$

$$
\begin{aligned}
&where \quad z_1 = 59x_1 + 63x_2 + 28x_3 + 15x_4 + 96x_5 + 88x_6 \\
&z_2 = 32x_1 + 83x_2 + 45x_3 + 82x_4 + 44x_5 + 17x_6 \\
&z_3 = 53x_1 + 40x_2 + 97x_3 + 37x_4 + 49x_5 + 45x_6
\end{aligned}
$$

According to Lokman and Köksalan's algorithm, we take z_3 as the primary objective to be maximized while satisfying lower bounds on z_1 and z_2. Also we take epsilon (ε) as 0.0001 to prevent the method yielding weakly efficient solutions. So the model to be solved in each step becomes:

$$\max \quad z = 53.0091x_1 + 40.0146x_2 + 97.0073x_3 + 37.0097x_4 + 49.014x_5 + 45.0105\,x_6$$

s.t.

$$21x_1 + 20x_2 + 56x_3 + 24x_4 + 51x_5 + 52\,x_6 \le 112 \qquad \text{(12: P2)}$$
$$59x_1 + 63x_2 + 28x_3 + 15x_4 + 96x_5 + 88\,x_6 \ge lb_1$$
$$32x_1 + 83x_2 + 45x_3 + 82x_4 + 44x_5 + 17x_6 \ge lb_2$$
$$x_k \in \{0,1\}$$

where the lower bounds and corresponding solutions are given at Table 1.

According to Lokman and Köksalan's algorithm, steps which are previously calculated *(3.3, 4.1, 4.2, 5.1, 5.4, 5.5, 6.1 and 6.6)* are obviously not solved again. Also, step *6.3* and *6.4* suggest a stricter lower bound on z_1 than step *6.2*. Since step *6.2* yielded an infeasible solution, stricter lower bounds cannot lead a feasible solution. Thus, they are not solved either.

Table 1 Lower Bounds and Corresponding Non-Dominated Solutions to Problem P2

Step	***lb₁***	***lb₂***	**Solution (z_1, z_2, z_3)**
1.1	0	0	(150, 160, 190)
2.1	0	161	(106, 210, 174)
2.2	151	0	(218, 159, 142)
3.1	0	211	*Infeasible*
3.2	107	161	(137, 197, 130)
3.3	151	0	*Same as step 2.2*
4.1	0	211	*Same as step 3.1*
4.2	107	161	*Same as step 3.2*
4.3	151	160	(174, 209, 126)
4.4	219	0	*Infeasible*
5.1	0	211	*Same as step 3.1*
5.2	107	198	(174, 209, 126)
5.3	138	161	(174, 209, 126)
5.4	151	160	*Same as step 4.3*

Table 1 (cont'd)

Step	lb_1	lb_2	Solution (z_1, z_2, z_3)
5.5	219	0	Same as step 4.4
6.1	0	211	*Same as step 3.1*
6.2	107	210	*Infeasible*
6.3	138	210	*Infeasible (since 6.2 is infeasible)*
6.4	151	210	*Infeasible (since 6.2 is infeasible)*
6.5	175	160	*Infeasible*
6.6	219	0	*Same as step 4.4*

Figure 4 shows the complete branch-and-bound tree used to solve problem P1 with the suggested method. P1 is converted to P2 according to Lokman and Köksalan's algorithm. Nodes are numbered according to their creation sequence. Different line styles show in which step the nodes are created: 10 new nodes in step *1.1* including the initial node, 21 new nodes in step *2.1*, 2 new nodes in *2.2* and 5 new nodes in step *3.1*. All other steps are calculated without creation of new nodes. Upper bounds on objectives z_1, z_2 and z of P2 are shown as (U_1, U_2, U_3) near the nodes. In order to save space, only the integer part of the upper bound on z is shown unless it is at the last level.

Since the procedure starts with no lower bounds on the objectives z_1 and z_2, in step *1.1* lb_1 and lb_2 are equal to 0. In order to demonstrate the procedure more clearly, we consider step *2.1* in Figure 5. Similar to Figure 4, line styles shows in which step the nodes are created and decimal part of U_3 is not shown for partial solutions. However, this time node numbers indicate the sequence the nodes are visited. If a node violates an upper bound constraint, the constraint violated is shown under the upper bound information. The model to be solved in step *2.1* is:

$$\begin{aligned}
\max \quad & z = 53.0091x_1 + 40.0146x_2 + 97.0073x_3 + 37.0097x_4 + 49.014x_5 \\
& + 45.0105\,x_6 \\
s.t. \quad & \\
& 21x_1 + 20x_2 + 56x_3 + 24x_4 + 51x_5 + 52\,x_6 \le 112 \qquad (13) \\
& 59x_1 + 63x_2 + 28x_3 + 15x_4 + 96x_5 + 88\,x_6 \ge 0 \\
& 32x_1 + 83x_2 + 45x_3 + 82x_4 + 44x_5 + 17x_6 \ge 161 \\
& x_k \in \{0,1\}
\end{aligned}$$

The lower bound on z_2 is 0 in this step and since all coefficients of our binary variables are positive, this constraint is redundant.

In this regard, nodes 4, 10, 11, 16, 20, 21, 22, 28 and 30 are eliminated due to the fact that upper bound values on z_2 are lower than the lower bound value of 161. First feasible solution to the problem P2 is found at node 7. At node 27, a better feasible solution is found and the best solution so far is updated. Node 30 represents the partial solution of $x_1=x_2=0$ (the rest is yet to be decided) and it does not meet U_2 constraint and there are no items left to exclude from the knapsack in this partial solution. Thus, the search is terminated and the best solution so far is the solution represented by node 27 (only items 2, 3 and 4 are included in the knapsack). Corresponding non-dominated objective vector is (106, 210, 174).

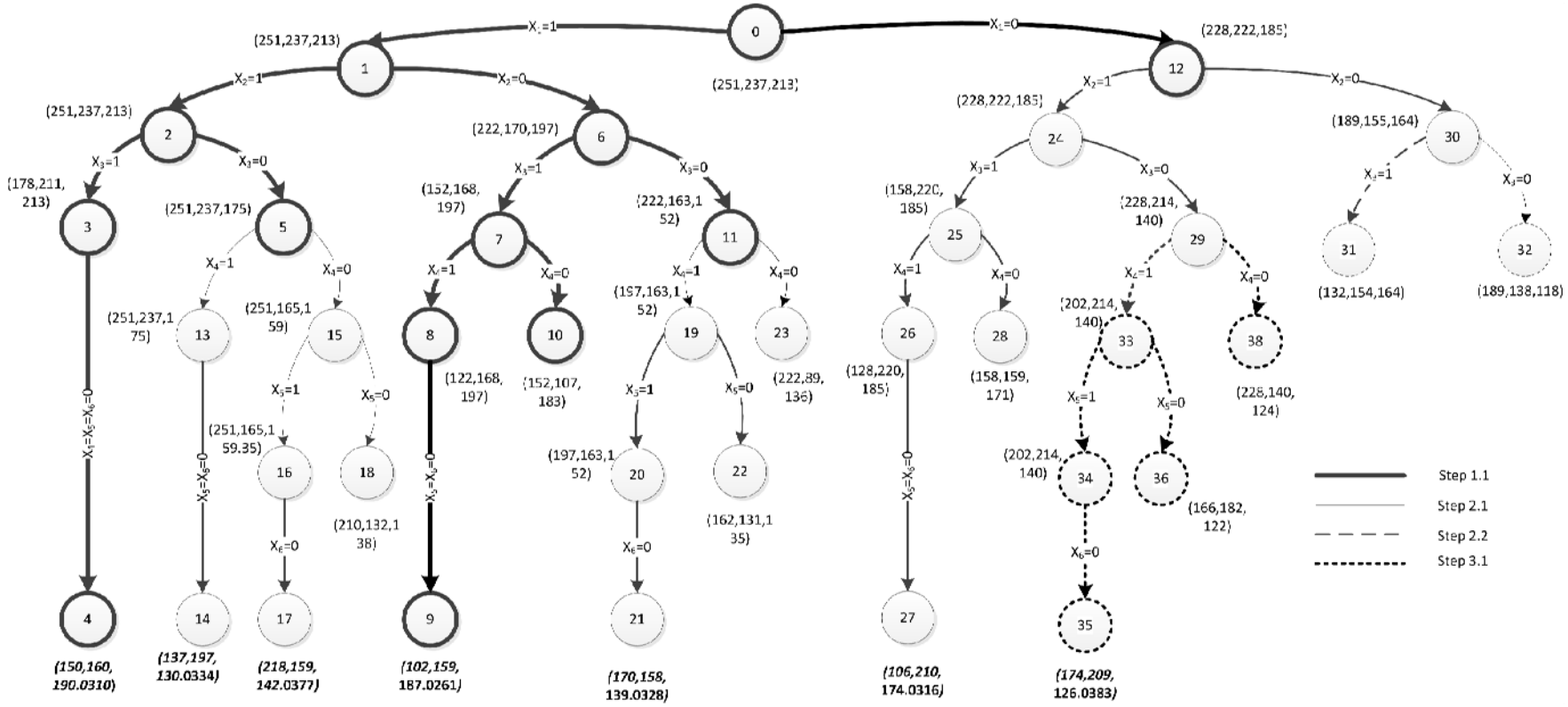

Figure 4 Branch-and-Bound Tree for problem P2 (Exact)

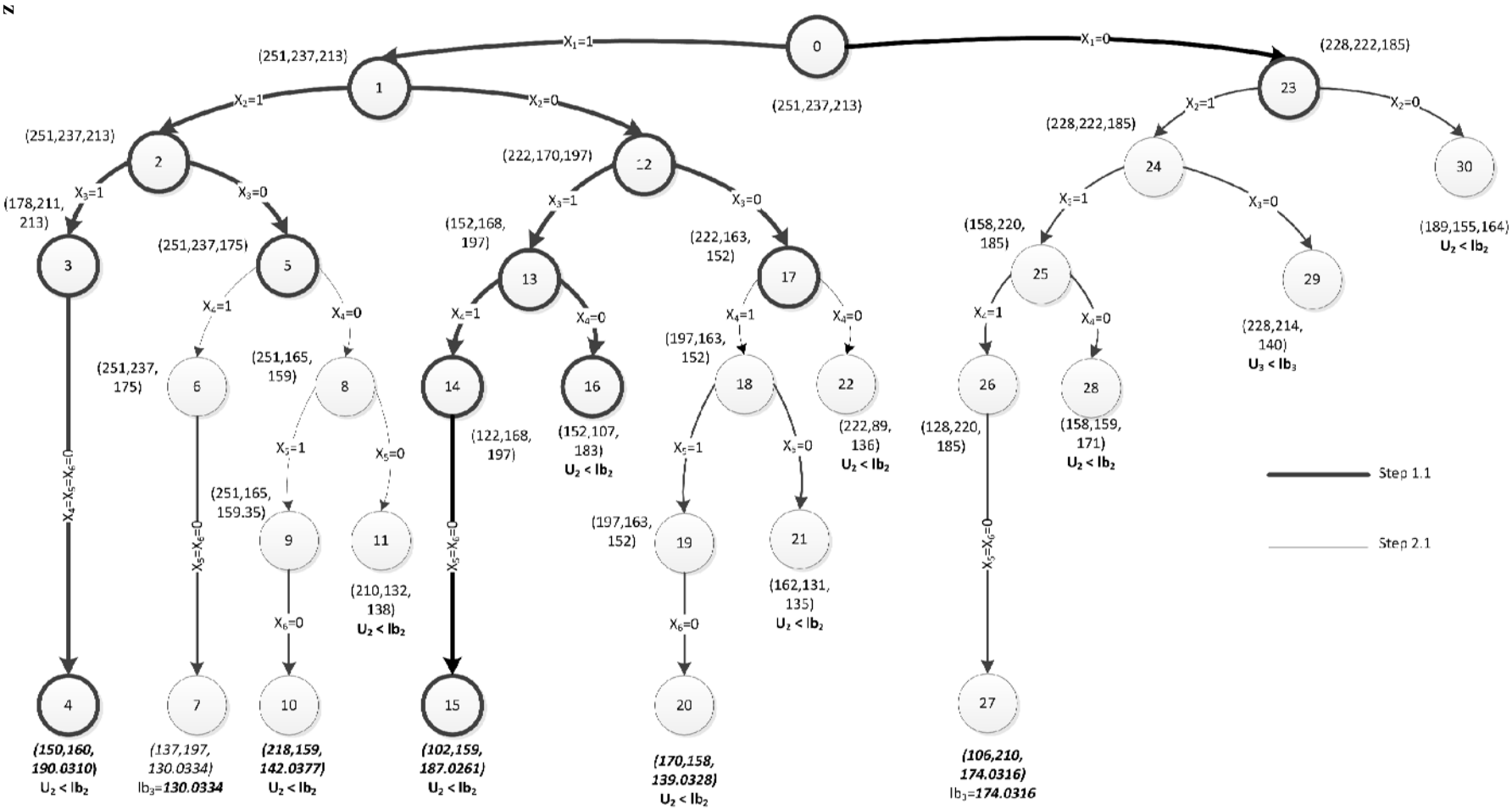

Figure 5 Partial Tree to Solve Step 2.1 (Exact)

6.2 AN ILLUSTRATIVE EXAMPLE FOR THE HEURISTIC ALGORITM

To illustrate the heuristic algorithm, problem P2 defined in section 6.1 is selected. As the heuristic algorithm is actually developed to overcome memory issues, item size 6 is relatively too small for the delta approach. For illustrative purposes, we choose delta as 65, which is an extremely large value for the size of the problem. Selection method of delta is discussed along with computational results in section 7.2. As the application of limits L_1 and L_2 is straightforward and they are not applied when the problem size is small enough to be handled, we do not illustrate it with this example.

Table 2 Lower Bounds and Corresponding Solutions to Problem P2 (Δ=65)

Step	lb_1	lb_2	Solution (z_1, z_2, z_3)
1.1	0	0	(150, 160, 190)
2.1	0	161	(137, 197, 130)
2.2	151	0	(218, 159, 142)
3.1	151	160	(174, 209, 126)
3.2	219	0	*Infeasible*
3.3	0	198	(106, 210, 174)
3.4	138	161	(174, 209, 126)
4.1	0	211	*Infeasible*
4.2	107	198	(174, 209, 126)
4.3	107	210	*Infeasible*
4.4	175	160	*Infeasible*

In Table 2, similar to Table 1, lower bounds found by Lokman and Köksalan's algorithm are shown. However, with delta being 65, same bounds do not correspond to the same solutions. As there does not exist any previous solution at the beginning of the algorithm, first run will always return the first non-dominated vector same as the exact version. So, we guarantee at least one non-dominated solution with this heuristic algorithm as long as the limits are not breached. Since the z_3 value of the first solution is 190 and delta is 65, the algorithm settles for the first feasible solution

having a z_3 value of at least 125.Thus, step 2.1 with a lower bound 161 on second objective corresponds to the vector (137, 197, 130). Now the algorithm will settle for the first solution having a z_3 value of at least 65. In step 2.2, algorithm returns the vector (218, 159, 142). If it did not find such a solution, it would behave like the exact algorithm and finds the best feasible solution. Even though the algorithm does not guarantee finding all non-dominated vectors, all of them are found with delta being 65 since the problem size is small and number of feasible solutions is limited.

Complete branch-and-bound tree and step 2.1 are shown in Figures 6 and 7. Similar to Figures 4 and 5, line styles shows in which step the nodes are created and numbers shows the creation sequence for Figure 6 and visiting sequence for Figure 7. In Figure 7, nodes created previously but not visited during this step are shown without numbering.

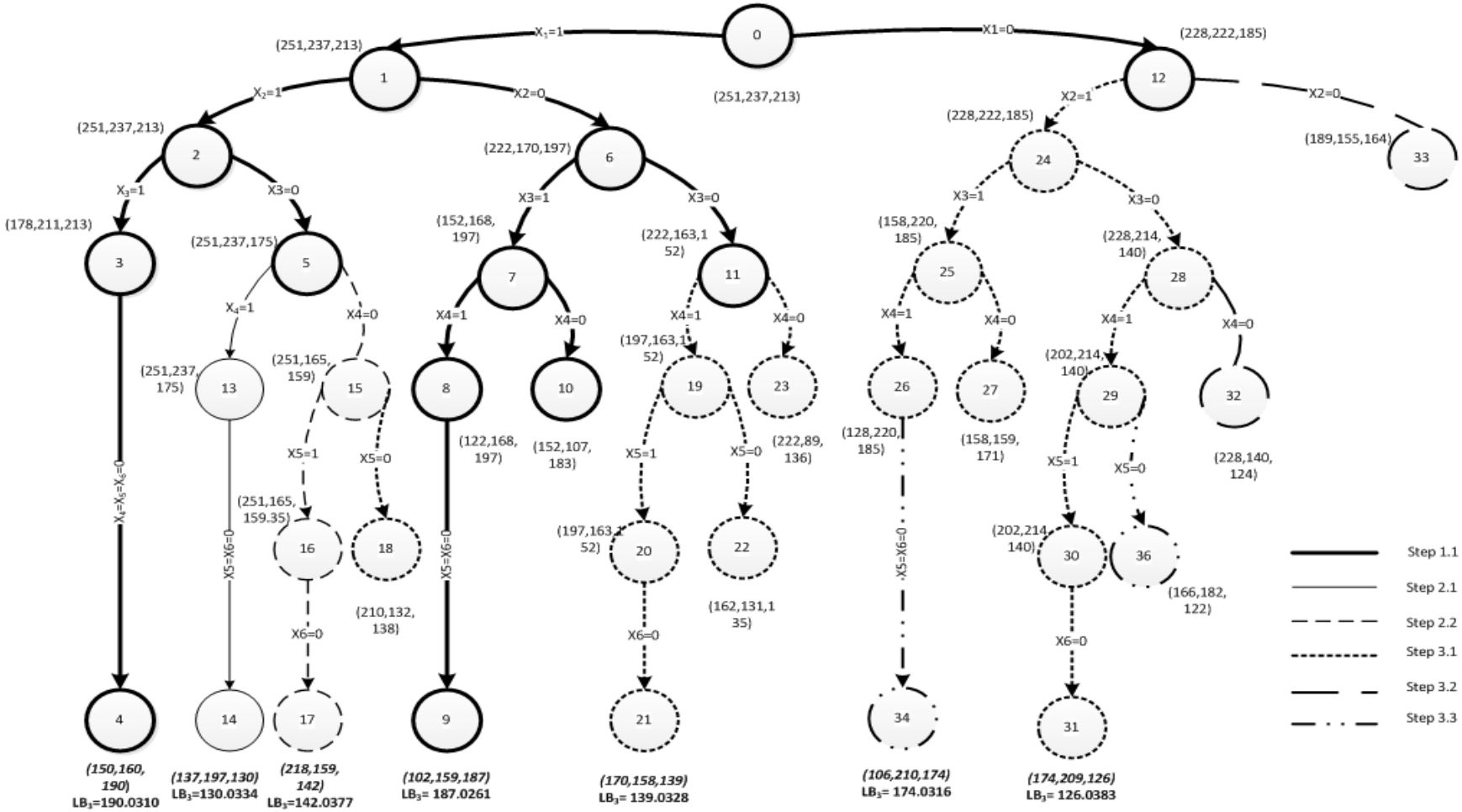

Figure 6 Branch-and-Bound Tree for problem P2 (Heuristic)

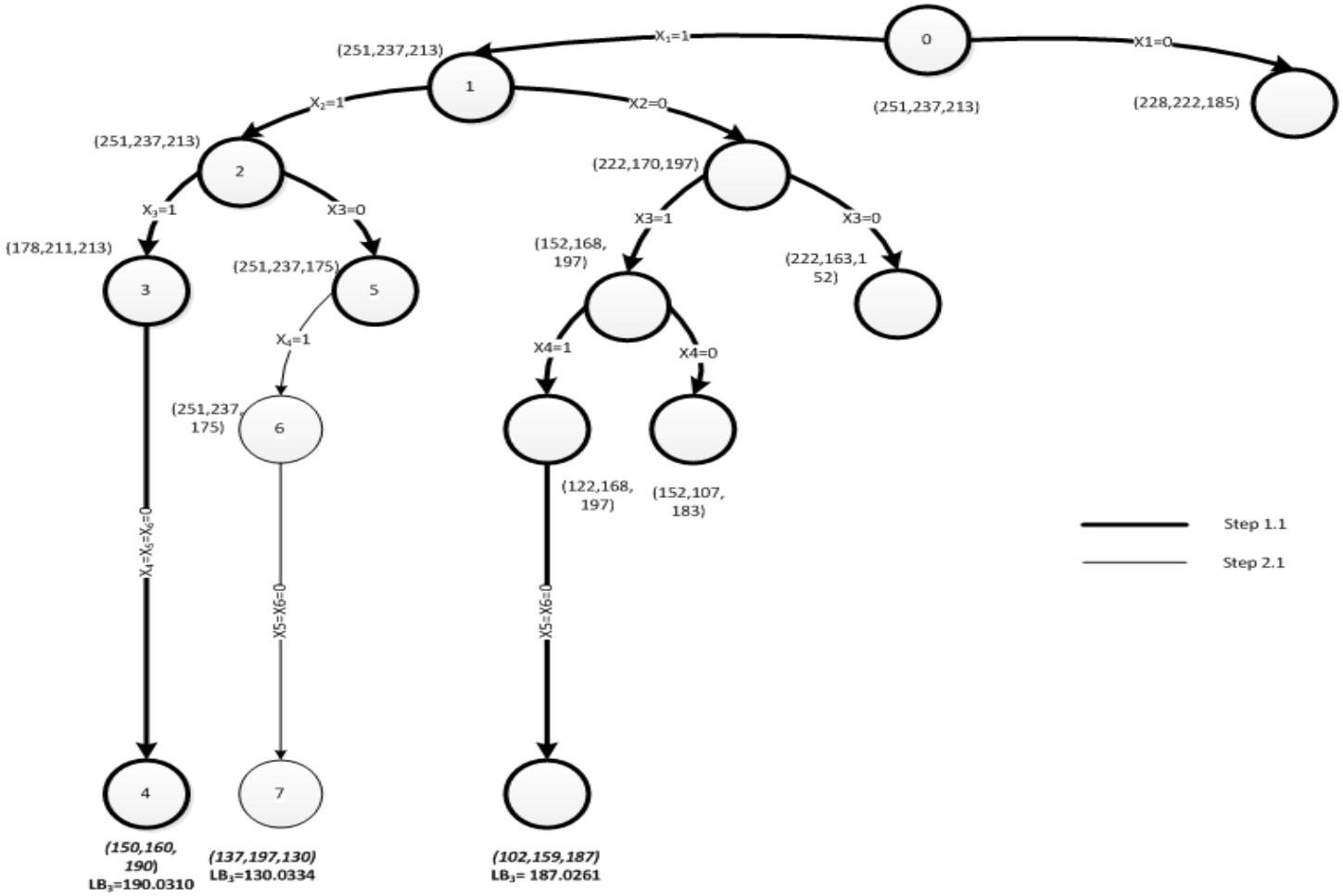

Figure 7 Partial Tree to Solve Step 2.1 (Heuristic)

CHAPTER 7

COMPUTATIONAL EXPERIMENTS

In order to compare our algorithms, we use Lokman and Köksalan's algorithm using IBM ILOG CPLEX optimization software (V12.3) on randomly generated sample problems of multi-objective 0-1 knapsack problems. To the best of our knowledge there are no other studies that present computational experiments on the type of problems we work on. Many of the studies only consider two criteria (Visee et al. (1998), Klamroth and Wiecek (2000)), some use multiple knapsack (Zitzler and Thiele (1999), Deb et al., (2000), (Corne et al., 2000). Thus, we compare the exact algorithm with L&K in terms of CPU seconds and we use hypervolume indicator for the heuristic algorithm, as we know the efficient frontiers of the sample problems.

7.1 RANDOMLY GENERATED KNAPSACK PROBLEMS

Sample problems with n objectives and m items are generated in the following form:

$$
\begin{aligned}
&\text{"max"}\left\{\sum_{k}^{m} p_{1k}x_k, \sum_{k}^{m} p_{2k}x_k, \ldots, \sum_{k}^{m} p_{nk}x_k\right\} \\
&s.t. \\
&\sum_{k}^{m} w_k x_k \le C \\
&x_k \in \{0,1\}
\end{aligned}
\tag{14}
$$

where x_k is the binary variable indicating whether the item k is included in the knapsack (1) or not (0).

p_{jk} is the profit of item k regarding objective j

w_k is the weight of item k

C is the capacity of the knapsack and $C = \frac{\sum_{k}^{m} w_k}{a}$

a is an integer to modify the size of the knapsack.

Uniformly distributed integers in the interval [10-100] are used for weight and profit coefficients. Capacity of the knapsack is a portion of total weight of the items and determined using the constant a.

All the computations are performed on a computer with 2.10GHz processor and 3.00 GB memory (RAM) of which only 1.50 GB is available for computations due to operation system's RAM requirements. As the number of variables increase, so does the number of nodes needed in the branch-and-bound tree of the algorithm. This causes an increase in the memory requirement. In order to be able to compare larger problem instances with the exact algorithm, we modify the knapsack capacity and make the comparison accordingly. For the heuristic version, no such modification is needed.

7.2 COMPUTATIONAL RESULTS

7.2.1 Proposed Exact Algorithm

We have tested our proposed exact algorithm (A&K) for several variables with number of objectives (n) being 3. As the approach is an exact method, we used CPU time in seconds as a performance measure. Also, average CPU time per models solved is examined. The comparison results can be seen in Table 3.

Table 3 Comparison of Exact Algorithm (n=3)

no. of items (m)	*Capacity modifier (a)*	*Random Problem*	*no. of models solved*	*no. of nondominated vectors*	**CPU Time Spent for Solution (sec)**	
					A&K	**L&K**
25	2	1	93	43	0.02	2.69
50	2	1	574	269	4.2	23.92
		2	659	295	2.53	30.69
		3	1050	519	9.3	74.34
		4	677	313	1.11	30.93
		5	734	348	1.48	36.07
75	4	2	1430	670	206.46	319.87
		3	1552	778	152.82	191.06
100	10	1	830	374	21.42	55.77
		2	1333	605	9.78	86.36
		3	1053	467	12.53	69.17
		4	1456	668	46.05	109.89
		5	986	450	5.16	66.33

As it is expected, as the number of variables increases, we observe an increase in the computational time to solve the problems for the same structure of capacity. Considering the randomly generated knapsack model (15), feasible space of a given problem can be reduced by setting lower capacity, i.e. increasing the capacity modifier *a*. Due to memory issues we experience, we used this property of the model to experiment on higher number of variables.

Similarly, we tested the exact algorithm for 4 objectives. Introducing a new objective leads each node keeping one more upper bound information and thus increasing the size of a node. Consequently, with a given memory size, the exact algorithm can solve smaller problems before the memory issues take effect. However, the total

number of nodes for a given number of items is a constant and having extra upper bounds on the nodes contributes to the elimination of subproblems represented by that node. This effect can also be observed in Table 4 results where n=4. Run time of the algorithm does not increase as much as the run time of L&K.

Table 4 Comparison of Exact Algorithm (n=4)

no. of items (m)	*Capacity modifier (a)*	*Random Problem*	*no. of models solved*	*no. of nondominated vectors*	**CPU Time Spent for Solution (sec)**	
					A&K	**L&K**
25	2	1	282	54	0.01	44.07
		2	2176	268	0.72	526.58
		3	1095	145	0.13	151.72
		4	1526	172	0.23	174.83
		5	3543	429	1.78	359.38
50	6	1	2090	284	0.99	306.71
		2	2138	302	1.09	277.68
		3	236	46	0.03	19.83
		4	3505	475	2.48	530.68
		5	2740	333	2.40	349.08
100	20	1	3430	447	3.67	625.56
		2	3230	460	2.41	748.21
		3	7026	789	18.47	1924.54
		4	4823	628	7.79	1063.98
		5	4690	557	6.20	1323.30

In general, proposed exact algorithm solves the problems quicker. However, it is hard to observe a pattern since we try to benefit from some special cases like lower branch elimination and some of the randomly generated problems possessing features in favor of our algorithm can be solved very quickly compared to L&K.

7.2.2 Proposed Heuristic Algorithm

Proposed heuristic algorithm is also tested using the randomly generated knapsack problems defined in 7.1.

Parametric variables defined in section 5.2 are Δ, L_1 and L_2. These variables are determined as follows:

Determining the value of Δ actually determines how distant we want our solutions to be in objective i. Since it is relatively easy to find the extreme points of the efficient frontier, we solve the model KPC with lb_j values being 0. In other words, we solve the KPC without any side constraints and we solve it selecting each objective as objective i one by one. We construct the payoff table and obtain maximum and minimum values of the original objective i. Difference between these values are divided by 100 to obtain the value of Δ. Payoff table does not guarantee obtaining the actual minimum value of objective i. On the other hand, we compared this method to using actual nadir point on sample problems and observed the difference is relatively small and the algorithm is not sensitive to changes in that scale. Instead of the payoff table, actual nadir point can also be used. Similarly, a dynamic approach can be incorporated and the value of Δ can be changed according to the non-dominated solutions found as the algorithm proceeds.

We take limit L_1 as the memory allocated divided by the size of a single node. After this limit is reached, limit L_2 is simply N. For the problems shown in Table 5 and 6, memory allocated was 0.65 GB of RAM.

We compared the algorithm to the exact method by using the hypervolume indicator suggested by Zitzler and Thiele (1998). Deb (2001) states that this indicator can be used to evaluate both closeness and diversity. Let Q be the vector set of the solutions and R be the reference point in the objective space which might be obtained by choosing the worst value for each objective function. Then, for each element of Q individual hypervolumes are calculated by taking the reference point's objective values as the diagonal corner of a hypercube. Union of these hypervolumes is the calculated hypervolume for a given set of solutions.

In order to formally give a definition, Zitzler et al. (2007) define an attainment function for an objective vector set Q. Without loss of generalization, they define a multiobjective maximization problem as:

$$\begin{aligned} &\text{"max"}\{f_1(x), f_2(x),..., f_n(x)\} \\ &s.t. \\ &x \in X \end{aligned} \tag{15}$$

and the objective space as $Z=(0,1)^n$. So, Ω is the set of all possible objective vectors and $Q \in \Omega$. Attainment function of Q (α_Q) is:

$$\begin{aligned} &\alpha_Q : [0,1]^n \to \{0,1\} \\ &\alpha_Q(z) := \begin{cases} 1 & if\ Q \succeq \{z\} \\ 0 & \text{else} \end{cases} \\ &z \in Z \end{aligned} \tag{16}$$

"$\succeq$" indicates the weakly dominance relationship. So the hypervolume of Q with respect to reference vector 0 can be defined as:

$$HV(Q) := \int_{(0,...,0)}^{(1,...,1)} \alpha_Q(z)\, dz \tag{17}$$

For example, for a biobjective problem with $Q=\{q_1,q_2,q_3\}$, hypervolume with respect to (0,0) is the union of shaded areas as shown in Figure 8.

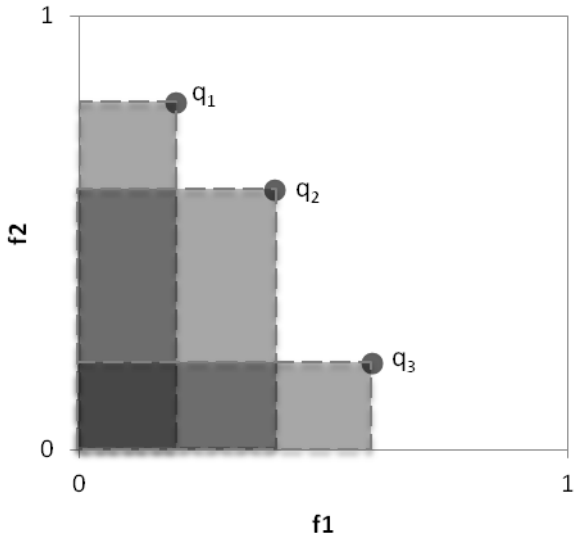

Figure 8 Illustrative Biobjective Case: Hypervolume for Q

For the case where set of pareto optimal solutions (P) is known, dividing hypervolume of obtained solutions ($HV_R(Q)$) by hypervolume of pareto optimal solutions ($HV_R(P)$) gives a coverage percentage which we used for comparing the heuristic algorithm. As this indicator is prone to scaling, all objective values are scaled beforehand. Obviously a better coverage is obtained by values closer to 1 and in the case of finding all solutions of efficient frontier, the coverage indicator will be 100%.

As it can be seen in Table 5, with 99% coverage of the volume covered by the efficient frontier, the heuristic algorithm performs very well when it does not breach memory limits for problem sizes 25 and 50 items and 3 objectives. However, there is a pay off between limiting the algorithm and solution quality as it may be observed in the case of 100 items. On average, solution time is improved by 10 times compared to the suggested exact algorithm and 100 times compared to using L&K. For 25 items, the heuristic algorithm solves the problems in less than a 100^{th} of a CPU second. Therefore, for a more meaningful comparison we chose the problems with 100 items.

Number of non-dominated vectors found is not necessarily correlated to the hypervolume covered by the algorithm. We are not only interested in maximizing the number of non-dominated vectors found but also the spread and closeness of the solutions obtained relative to the efficient frontier. Depending on the decision maker's preferences, a high number of non-dominated vectors accumulated in a restricted portion of the efficient frontier may be less desirable than a good spread of dominated vectors close to the efficient frontier. Also, if the pareto optimal solutions are denser at some areas, algorithms may tend to get stuck at those areas. One may argue a non-dominated solution found in a sparse area is better than many non-dominated solutions found in dense areas. So, we consider better hypervolume coverage as the main performance measure rather than the number of non-dominated solution vectors found. It can also be observed for the 100 items case in Table 5. In one case, only 1 of the actual non-dominated vectors is found. However, covered volume is 5% better than a similar case with 33 non-dominated vectors found.

Table 5 Comparison of Heuristic Algorithm (n=3)

no. of items (m)	*Capacity modifier (a)*	*Random Problem*	*no. of nondominated vectors*	*no. of vectors founds*	*no. of nondominated vectors found*	*% of hypervolume covered*
25	2	1	50	48	41	99%
		2	109	100	99	99%
		3	76	69	53	99%
		4	21	19	18	99%
		5	50	49	40	99%
50	2	1	269	232	183	99%
		2	295	234	204	99%
		3	519	430	307	99%
		4	170	127	121	99%
		5	405	371	326	99%
100	4	1	3837	247	9	50%
		2	2470	247	1	63%
		3	3100	308	41	55%
		4	2317	247	33	58%
		5	2004	466	36	78%

We also compared the algorithm for 4 objectives and listed the results in Table 6. Selected objective *i* is taken as the 4th objective. Similar to the exact case, to reduce the size of the feasible space we used the knapsack capacity modifier a. For 100 items, the duration for L&K ranged from 625 to 1923 CPU seconds, whereas heuristic algorithm's run time ranged from 1.28 to 17.97 CPU seconds. The heuristic algorithm runs much faster than L&K with a loss of 1% in hypervolume coverage. Distribution of non-dominated vectors found by the algorithm tends to have larger 4th objective values, i.e. as the algorithm proceeds towards stricter lower bounds of

Lokman and Köksalan's algorithm, the heuristic algorithm tends to return premature infeasible results to limit the search. One way to overcome this disadvantage may be to alter the objective *i* between the objectives and re-run the algorithm. By combining the outputs of these re-runs, a better convergence and representation of the efficient frontier is possible.

Table 6 Comparison of Heuristic Algorithm (n=4)

no. of items (m)	***Capacity modifier (a)***	***Random Problem***	***no. of nondominated vectors***	***no. of vectors founds***	***no. of nondominated vectors found***	***% of hypervolume covered***
25	2	1	429	316	297	99%
		2	172	151	139	99%
		3	145	124	112	99%
		4	268	233	218	99%
		5	54	51	43	99%
50	6	1	284	198	195	98%
		2	302	289	288	99%
		3	46	47	42	99%
		4	475	469	466	99%
		5	333	236	227	99%
100	20	1	447	355	326	99%
		2	460	442	440	99%
		3	789	759	745	99%
		4	628	409	396	97%
		5	557	459	447	98%

7.2.3 Variations and Discussion

As we develop the algorithm, both exact and heuristic, we tried different variations hoping to improve the algorithms. Algorithms defined in sections 4 and 5 are the final versions, yet we discuss some variations. We used the same randomly generated knapsack problems for these variations.

For the exact algorithm, main problem we encountered is the memory used by the branch-and-bound tree. Performances of these variations are compared using randomly generated knapsack problems with 50 items.

1. **Keeping Previous Information vs. Recalculating:** The proposed algorithm keeps all of the nodes created by previous iterations. One way to overcome this memory issue may be recreating the entire branch-and-bound tree each time without keeping any previous information. The lower bounds implied by Lokman and Köksalan's algorithm are not strict at the beginning of the algorithm. We observed this property leads creating nodes (allocating memory) and saving information costs more in CPU time compared to just calculating only the current node's upper bound information without saving. On the other hand, as the Lokman and Köksalan's algorithm progresses and challenges the model with stricter lower bounds, the search process of branch-and-bound gets longer and longer since it has to recalculate all of the information. At the end of the runs, keeping the previous nodes outperformed not keeping by 2342% on average in terms of CPU time.

2. **Keeping Level Information or Recalculating:** When the residual knapsack capacity cannot accept any of the items left, we jump to the lowest node without further branching. This causes a problem for the backtracking step because the algorithm needs to know which level it is backtracking to. We can overcome this problem with either keeping the information of level on the node itself or recalculating it using the parent-child relationship and counting the number of nodes beginning from the root node. Although the latter needs less memory, computational results showed that keeping the level information

on the node saves CPU time by 53% on average compared to recalculating this information each time the algorithm backtracks.

3. **Eliminating vs. Keeping Lower Branches:** Another obvious variation is deciding whether to keep the property of elimination of lower branches as described in section 4.2.3. If no lower branch is eliminated, keeping or recalculating the level information is not needed since the algorithm will jump one node at a time during both forward and backtracking moves. We compared the method described above to the method of creating these lower nodes. Keeping level information and eliminating the lower branches as described above outperformed keeping the lower branches by 5% on average.

4. **Deleting Nodes:** We also tried to delete the nodes up to a level when a certain limit is exceeded. Obviously, deleting is an additional operation and consumes extra CPU time. However, keeping some of the information from the deleted nodes might help the search process and save time. To test this we updated the upper bound information of a node from bottom up by taking the maximum of upper bounds of its children. Since at the lowest level, when no other variable is left to include or exclude from the knapsack, the upper bound is actually the objective value of that decision set. So, this method replaces the calculated upper bounds with stricter ones whenever the subbranches of that node reach the bottom of the tree. Unfortunately, better upper bound information did not compensate for the time spent for deleting nodes and updating information.

5. **Feeding the Lokman and Köksalan's algorithm with known non-dominated vectors:** As finding supported non-dominated vectors is relatively easy, some or all of that kind of vectors might be included to the list of non-dominated vectors kept by Lokman and Köksalan's algorithm to determine the next iteration's lower bounds. This might reduce the number of models to be solved in total. However, it requires a major change in Lokman and Köksalan's algorithm and we did not apply the change in the scope of this thesis.

For the heuristic algorithm, main motivation is handling the memory problem of the exact version while maintaining a reasonable distance from the efficient frontier. Similar to the exact algorithm, performances are tested on randomly generated knapsack problems with 3 objectives and 50 items except for the third variation.

1. **Delta Distance:** We tried the delta distance approach as a separator between the non-dominated objective vectors in terms of the i^{th} objective as defined in section 4.1. In this version, the algorithm is the same as the one defined in 5.2 except $UB_i - \Delta \geq lb_i$ is used to decide on early termination of the subroutine and no limitations on the number of nodes are used. Settling for the first solution which is at least delta far away from the previous solution obviously caused the algorithm perform worse. Moreover, the memory usage showed no significant improvement in exchange of the solution quality. We observed the main memory absorber is the case when no feasible solution exists. The algorithm cannot find any lb_i and thus, it exhausts the tree by creating new nodes until absence of a feasible solution is proved. This case is more likely to happen with stricter lower bounds determined by Lokman and Köksalan's algorithm as they limit the feasible space progressively with the introduction of new non-dominated solution vectors.

2. **Modifying lower bounds:** In order to overcome the memory issues of infeasible solutions, we tried to foresee infeasibility by looking at the difference between the upper and lower bounds of the node. If they are too close, one might argue that the possibility of that node yielding a feasible solution is dim and the node can be fathomed by risking losing a potential feasible solution. However, assessing a value to "too close" is not so easy. We used delta value and its multiples and obtained delta similar to the method discussed in section 6.2. Unfortunately, the memory problem persisted for small values and getting the difference larger just caused eliminating feasible solutions. Eventually, we chose to limit the number of nodes created rather than modifying the lower bounds.

3. **Altering objective *i*:** No matter which objective is selected as the objective *i* as defined in model (4) in section 4.1, Lokman and Köksalan's algorithm finds all non-dominated vectors for the exact case. However, for the heuristic case, experimental computations show "harder" problems tend to exceed memory limits and tend to prematurely return infeasible. A problem gets "harder" with capacity modifier *a* getting closer to 2, increasing number of items and objectives as well as stricter lower bounds imposed by Lokman and Köksalan's algorithm. This memory limitation causes gaps in the approximation of the efficient frontier when the objective *i* value gets smaller. On the other hand, selecting another objective as objective *i* means smaller values of the same objective is obtained by imposing smaller lower bounds which are imposed at the beginning of Lokman and Köksalan's algorithm. As the computational time consumed by heuristic approach is relatively much smaller than the L&K, re-running the same algorithm by altering objective *i* is less costly. We tested this variation on a randomly generated problem with 100 items, 3 objectives and a=2. Each objective's performance as well as their combined performance is shown in Table 7.

Table 7 Computational Results for Altering Objective i

Instance	*no. of nondominated vectors*	*no. of vectors founds*	*no. of nondominated vectors found*	*% of hypervolume covered*
1	2751	209	47	58%
2	2751	134	57	67%
3	2751	275	85	50%
Combined	2751	489	102	82%

Obviously solution sets found by different instances are not distinct. Both some dominated and non-dominated solutions exist in more than one instance. However, when combined, we observed not only a better coverage of the hypervolume but also a better spread of the vectors.

CHAPTER 8

CONCLUSION AND FURTHER RESEARCH

In order to find all efficient solutions of any given MOCO problem, we developed a generic branch-and-bound procedure using Lokman and Köksalan's approach (2012). We proposed an exact branch-and-bound algorithm for multi-objective 0-1 KP and computational results showed that problem specific algorithms using previous iterations' information can outperform a very reliable and fast commercial tool like IBM ILOG CPLEX. However, we also observed saving information costs memory because MOCO problems are NP-hard and as the number of variables increase, memory need for the tree also increases. In order to be able to solve larger instances benefiting previous information, some two-phase approaches can be incorporated like variation 5 we discussed in section 7.2.3. It may be promising as it may reduce the effort to generate supported solutions as well as unsupported ones. However, as discussed, this variation requires a major change in the lower bound generation method of Lokman and Köksalan's algorithm.

We also proposed a heuristic branch-and-bound algorithm for MOCO. We developed a specific heuristic algorithm for MOKP by modifying the proposed exact algorithm. We observed delta approach yields a good spread of solutions with a good coverage with respect to efficient frontier when the memory limits are not breached. However, we observed memory limitation reduces the solution quality significantly for larger problems. Among the variations for heuristic algorithm, last approach is the most promising as it combines the algorithm results for all objectives as objective i.

REFERENCES

Corne, D. W., Knowles J. D., and Oates, M. J. *The pareto envelope-based selection algorithm for multiobjective optimisation*. In M. S. et al. (Ed.), Parallel Problem Solving from Nature – PPSN VI, Berlin, pages 839–848, Springer, 2000.

Dantzig G., *Discrete variable extremum problems,* Operations Research, 5, pages 226–77, 1957.

Deb K., *Multi-Objective Optimization using Evolutionary Algorithms*, Wiley-Interscience Series in Systems and Optimization. John Wiley & Sons, Chichester, 2001.

Deb K., Agrawal S., Pratap A., Meyarivan T., *A Fast and Elitist multi-objective Genetic Algorithm: NSGA-II*, IEEE Transactions on Evolutionary Computation (IEEE-TEC), 6 (2), pages 182-197, 2002.

Ehrgott M. and Gandibleux X., *A survey and annoted bibliography of multiobjective combinatorial optimization*, OR Spektrum, 22, pages 425–460, 2000.

Gandibleux, X. and Freville, A., *Tabu search based procedure for solving the 0–1 multiobjective knapsack problem: the two objective case*, Journal of Heuristics, 6, pages 361–383, 2000.

Gavanelli M., *An algorithm for Multi-Criteria Optimization in CSPs*, In Frank van Harmelen, editor, Proc. 15thEuropean Conference on Artificial Intelligence, Lyon, France, IOS Press, 2002.

Horowitz E., Sahni S., *Computing partitions with applications to the knapsack problem*, Journal of ACM 21, pages 277-292, 1974.

Klamroth, K., and Wiecek, M., *Dynamic programming approaches to the multiple criteria knapsack problem*, Naval Research Logistics, 47, pages 57–76, 2000.

Köksalan, M. and Phelps S., *An Evolutionary Metaheuristic for Approximating Preference-Non-dominated Solutions.* INFORMS Journal on Computing 19 (2), pages 291-301, 2007.

Lokman, B. and Köksalan, M., *Finding all nondominated points of multi-objective integer programs*, Journal of Global Optimization, DOI 10.1007/s10898-012-9955-7, 2012, forthcoming

Lukasiewycz M., Glass M., Haubelt C., and Teich J., *Solving multiobjective pseudo-boolean problems*, In Proc. SAT, pages 56–69, Lisbon, Portugal, 2007.

Martello S. and Toth P., *Knapsack Problems: Algorithms and Computer Implementations,* John Wiley & Sons, Chichester–New York, 1990

Sylva J. and Crema A., *A method for finding the set of non-dominated vectors for multiple objective integer linear programs,* European Journal of Operational Research, 158(1), pages 46-55, 2004.

Sylva J. and Crema A., *A method for finding well-dispersed subsets of non-dominated vectors for multiple objective mixed integer linear programs*, European Journal of Operational Research, 180(3), pages 1011-1027, 2007

Ulungu E.L. and Teghem J., *Application of the two phases method to solve the biobjective knapsack problem,* Technical report, Faculte Polytechnique de Mons, Belgium. 1994.

Ulungu E.L., Teghem J., Fortemps P. and Tuyttens D., *MOSA method: a tool for solving multiobjective combinatorial optimization problems,* Journal of Multi-Criteria Decision Analysis, 8(4), pages 221-236, 1999.

Visee M., Teghem J., Pirlot M. and Ulungu E.L., *Two-phases method and branch and bound procedures to solve the bi-objective knapsack problem*, Journal of Global Optimization, 12, pages 139–155, 1998.

Zitzler E., Brockhoff D., and Thiele L., *The Hypervolume Indicator Revisited: On the Design of Pareto-compliant Indicators Via Weighted Integration,* In S. Obayashi et al., editors, Conference on Evolutionary Multi-Criterion Optimization (EMO 2007), volume 4403 of LNCS, pages 862–876, Springer, Berlin, 2007

Zitzler E., Laumanns M., and Thiele L., *SPEA2: Improving the Strength Pareto Evolutionary Algorithm for Multiobjective Optimization*, Technical Report in Evolutionary Methods for Design, Optimization and Control, Barcelona, 2001.

Zitzler E. and Thiele L. *Multiobjective Optimization Using Evolutionary Algorithms - A Comparative Case Study,* In Conference on Parallel Problem Solving from Nature (PPSN V), pages 292–301, Amsterdam, 1998.

Zitzler E. and Thiele L., *Multiobjective Evolutionary Algorithms: A Comparative Case Study and the Strength Pareto Evolutionary Algorithm*, IEEE Transactions on Evolutionary Computation, 3(4), pages 257-271, 1999.

APPENDIX A

GENERIC HEURISTIC ALGORITHM

In addition to the variables defined in the previous section, let *flag* be the binary variable indicating if the predetermined limit L_1 is exceeded.

1. **Lokman and Köksalan's Algorithm:** Calculate LB_j according to L&K algorithm.

   Previous solution UB is set to a sufficiently large number M.

   If no further lower bounds are left to consider, **STOP**.

2. **Initialization:** Initially no decision variables are set: A={}

   Best solution LB_i is set to 0.

   Create first node and calculate respective upper bounds U_{jq}.

3. If $U_{jq} \geq LB_j$ for all j,

   go to step 4.

   else

   go to **backtrack**.

4. If no more decision variables are left to decide (A^c={}),

   Update best solution as current solution

   If $UB - \Delta \leq LB_i$

   return objective function values of the best solution

(z_j(best_solution))

else

go to **backtrack**

else

Forward move: Create new node according to the original algorithm's set of rules. Update decision variable set A. Calculate upper bounds and go to step 3.

5. **Backtrack**: If ($count\_all \geq L_1$ **AND NOT**(flag))

$flag := 1$

if best solution is greater than 0,

return objective function values of the best solution (z_j(best_solution)) and go to step 1.

else

return *infeasible* and go to step 1.

If ($count \geq L_2$ **AND** flag)

if best solution is greater than 0,

return objective function values of the best solution (z_j(best_solution)) and go to step 1.

else

return *infeasible* and go to step 1.

else

Backtrack according to the original algorithm's set of rules.

6. If all possible subproblems are considered

if best solution is greater than 0,

return objective function values of the best solution (z_j(best_solution)) and go to step 1.

else

return *infeasible* and go to step 1.

else

go to step 3.

APPENDIX B

HEURISTIC ALGORITHM – MOKP

Similar to Appendix A, *flag* is the binary variable indicating if the predetermined limit L_1 is exceeded.

1. **Initialization I**: These steps will be performed only once.

   1.1. Find preference sequence for each j other than i ($pref_{jk}$: k^{th} position in the preference sequence of objective j).

   1.2. Find minimum weight for each level ($\min\_ w_l = \min\{w_k : k \geq l\}$). This information will be used for eliminating the lower bounds.

   1.3. Insert first node. Calculate and save U_j for all j.

   1.4. Previous solution UB is set to $\sum_k p_k$

   1.5. $count\_all := 0$

2. **Lokman and Köksalan's Algorithm**: Calculate lb_j according to L&K algorithm.

   If no further lower bounds are left to consider, **STOP**.

3. **Start Subroutine**

   Initialization II: Following steps will be performed before the branch-and-bound procedure for each step of L&K algorithm, i.e. for each set of lower bounds.

   3.1. Initialize variables

$lb_i := 0$

$best\_x_k := 0 \quad \forall k$

$cur\_x_k := 0 \quad \forall k$

$cur\_z_j := 0 \quad \forall j$

$cur\_c := C$

$l := 1$

$t := 0$

$count := 0$

3.2. If $U_j \geq lb_j$ for all j does not hold for at least one objective, go to **5.1 *(backtrack)***.

4. **Forward Move**

4.1. **while** ($cur\_c \geq w_l$ AND $U_j \geq lb_j \quad for\ j \neq i$)

do

$cur\_x_l := 1$

$cur\_c := cur\_c - w_l$

$cur\_z_j := cur\_z_j + p_{jl} \quad for\ j \neq i$

$cur\_z_i := cur\_z_i + p'_{il}$

$l := l + 1$

If not already created for current solution (*cur_x*), insert new node on level *l*, calculate and save U_j for all *j*

$count := count + 1$

$count\_all := count\_all + 1$

If $U_j \geq lb_j$ for all j does not hold for at least one objective, go to **6 *(backtrack)***.

4.2. If $cur\_c < \min\_w_l$ (no more insertions are possible)

$l := K$

If not already created for current solution (*cur_x*), insert new node on level *l*, calculate and save U_j for all *j*

$count := count + 1$

$count\_all := count\_all + 1$

4.3. If current level, which is critical item, is not the last item, exclude it from the knapsack

If $l < K$

If not already created for current solution (*cur_x*), insert new node on level *l*, calculate and save U_j for all *j*

$count := count + 1$

$count\_all := count\_all + 1$

$l := l + 1$

If $U_j \geq lb_j$ for all j does not hold for at least one objective, go to **6 *(backtrack)***.

4.4. If current level (critical+1) is not the last item, repeat forward move.

If $l < K$ go to **4 (forward move).**

4.5. If $l = K$

If $cur\_c \geq w_l$ (If the last item is small enough, include in the knapsack)

$cur\_x_l := 1$

$cur\_c := cur\_c - w_l$

$cur\_z_j := cur\_z_j + p_{jl} \qquad for\ j \neq i$

$cur\_z_i := cur\_z_i + p'_{il}$

If not already created for current solution (*cur_x*), insert new node on level *l*, calculate and save U_j for all *j*

$count := count + 1$

$count\_all := count\_all + 1$

else (If the last item is not small enough, exclude from the knapsack)

If not already created for current solution (*cur_x*), insert new node on level *l*, calculate and save U_j for all *j*

$count := count + 1$
$count\_all := count\_all + 1$

5. **Update Best Solution So Far**

5.1. If $U_j \geq lb_j$ for all j

$lb_i := cur\_z_i$
$best\_x_k := cur\_x_k \qquad \forall k$

If $UB_i - \Delta \leq lb_i$ (If this feasible solution is close enough)

go to 7 (subroutine return)

If $cur\_x_K := 1$ (If last item is included, throw it out before backtrack)

$cur\_x_K := 0$
$cur\_c := cur\_c + w_K$
$cur\_z_j := cur\_z_j - p_{jK} \qquad for\ j \neq i$
$cur\_z_i := cur\_z_i - p'_{iK}$

6. **Backtrack**

6.0 If ($count\_all \geq L$ **AND NOT**(*flag*))

$flag := 1$

go to 7 (subroutine return)

If ($count \geq N$ **AND *flag*)**

go to **7 (subroutine return)**

6.1. Find the item to throw out of the knapsack

Find *t*, $t = \max\{k < l : cur\_x_k = 1\}$

If $t > 0$

$cur\_x_t := 0$

$cur\_c := cur\_c + w_t$

$cur\_z_j := cur\_z_j - p_{jt} \qquad for\ j \neq i$

$cur\_z_i := cur\_z_i - p'_{it}$

$l := t$

If not already created for current solution (cur_x), insert new node on level *l*, calculate and save U_j for all j

$count := count + 1$

$count\_all := count\_all + 1$

go to **4 (forward move)**.

else

go to **7 (subroutine return)**.

7. **Subroutine Return**

7.1. If $best\_x_k = 0 \qquad \forall k$

No feasible solutions. **return** *infeasible*.

else

return $best\_z_j = \sum_k p_{jk} best\_x_k \qquad \forall j$

go to **2**