A BRANCH-AND-PRICE ALGORITHM FOR BIN PACKING PROBLEM

MASOUD ATAEI

A THESIS SUBMITTED TO THE FACULTY OF GRADUATE STUDIES

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS

FOR THE DEGREE OF

MASTER OF SCIENCE

GRADUATE PROGRAM IN

APPLIED AND INDUSTRIAL MATHEMATICS

YORK UNIVERSITY

TORONTO, ONTARIO

August 2015

Abstract

Bin Packing Problem examines the minimum number of identical bins needed to pack a set of items of various sizes. Employing branch-and-bound and column generation usually requires designation of the problem-specific branching rules compatible with the nature of the pricing sub-problem of column generation, or alternatively it requires determination of the k-best solutions of knapsack problem at level k^{th} of the tree. Instead, we present a new approach to deal with the pricing sub-problem of column generation which handles two-dimensional knapsack problems. Furthermore, a set of new upper bounds for Bin Packing Problem is introduced in this work which employs solutions of the continuous relaxation of the set-covering formulation of Bin Packing Problem. These high quality upper bounds are computed inexpensively and dominate the ones generated by state-of-the-art methods.

Keywords: Bin Packing Problem, Branch-and-Bound, Column Generation.

To my parents

Acknowledgments

I would like to express my deep sense of appreciation to my supervisor, Dr. Michael Chen. Had it not been for his invaluable guidance and tremendous support, I would not be able to surmount all obstacles laid in the path of research. His flexibility and dedication allowed me to explore different areas of Operational Research that caught my interest while keeping in sight my road map and final goals. Not only had I the opportunity to pursue my interests in mathematics during the time of working with Dr. Chen, but also I learned how to follow my heart when making vital decisions in life and free myself from those pressures which mechanize the mind and make for routine thinking as founding president of York University, Murray G. Ross once envisaged.

Furthermore, I would like to thank Dr. Murat Kristal for taking time, to be part of the supervisory committee, and for advising me on my research.

I am also grateful to Professor Jeff Edmonds for agreeing to be on my examining committee.

Throughout my study at York University, I benefited a lot from professors and friends who were always helpful. I especially want to thank Dr. Vladimir Vinogradov for his unconditional generosity and help. My sincere thanks also to Mourad Amara who applied his editorial talents to my thesis and made it look better. I am also thankful to Primrose Miranda for her administrative support during the whole period of my study.

Last but not least, I want to thank my parents from whom I learned the alphabet of love and to whom I am indebted for all the achievements I have in life. Even though the words are not enough when it comes to expressing appreciations for love, I want to give my special thanks to my mother who made lots of sacrifices to be with me in Canada and support me thoroughly during the time I was working on my thesis. Also, my deepest gratitude to my father both as a person I always admired in my life and as a university professor who unfailingly helped and encouraged me to realize the true nature of doing research. My thanks finally go to my beloved sister, Minoo, for her whole-hearted love and encouragement during the years of my schooling.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Overview

In a *Bin Packing Problem* (BPP), given an unlimited number of identical bins with a capacity $c > 0$ and a set of items $I = \{i_1, i_2, \dots, i_n\}$ each of which has a weight w_i ($0 < w_i \leq c$); the goal is to pack all the items in a minimum number of bins without exceeding bins capacities.

The importance of BPP stems from its numerous applications in industry and logistics such as loading trucks, queuing television commercials into given time slots, assigning tasks to machines, and placing data into memory blocks of fixed-size, to name a few. Furthermore, BPP often arises as a frequently called sub-problem in some practical cases in the areas of transportation and supply chain management, and it needs to be solved in a shortest amount of time. BPP has been studied by many researchers during the last few decades, and a large number of heuristics, meta-heuristics, and approximations, have been proposed to tackle this problem.

Although there are a considerable number of methods to find solutions to BPP in a reasonable time, very few of them guarantee the optimality of the solutions which are

referred as the exact methods in the literature. It is worth mentioning that majority of the so-called exact methods are computationally expensive, and the need for a fast and comprehensive exact method is felt in the area. To this end, we develop a framework based on *branch-and-price* method to obtain the optimal solutions to the BPP. This new exact method is a *branch-and-bound tree* which employs *column generation* as well as a set of newly proposed strong upper bounds to the BPP.

Employing column generation in a branch-and-bound tree requires development of rigorous methods to solve the *pricing sub-problem* of the column generation. More specifically, while proceeding into the depth of the branch-and-bound tree, feasible loading patterns known as *forbidden patterns* come about and should systematically be excluded from the search domain of the *one-dimensional knapsack problem*. To address the issue of having *polluted pricing sub-problem* in column generation, two main approaches have been introduced according to the literature.

The first approach determines k-best solutions of the one-dimensional knapsack problem at the $(k-1)^{\text{th}}$ level of the tree to ensure that enough feasible loading patterns exist to be injected into the *master problem* of the column generation in case all the forbidden patterns are met during the solving procedure. Although being a simple and straightforward idea, determination of the k-best solutions of a knapsack problem demands expensive computational resources.

In the second approach, a set of branching rules is designed to avoid emergence of the forbidden patterns in the pricing sub-problem of the column generation during the branch-and-bound procedure. In other words; an implicit direction in dealing with the forbidden patterns is preferred in this approach. Even though the methods developed in this category benefited from capability of solving wide range of BPP instances, the exorbitant computational costs remain a concern. For, in the present methods, branching occurs on the pair of items whereas the dimension of the problem after exclusion of these items remains close to the dimension of the parent problem, and solving the residual problem is as time-consuming as solving the parent one. Therefore, without generalizing such branching rules to branch on more items rather than on a pair of them, *branch-and-price procedure* cannot be performed with the desired efficiency.

In this research, we propose a new approach for solving the polluted pricing sub-problem of column generation. In a nutshell, a constraint called *decrement constraint* is added to the one-dimensional knapsack problem whenever one of the forbidden patterns is met during the column generation procedure. This extra constraint compels the pricing sub-problem to generate the next feasible solution of the one-dimensional knapsack problem, and then the generated column is passed to the master problem of column generation. Immature termination of the column generation, which might occur due to deployment of this method, and its consequent impacts on the branch-and-price procedure is investigated and resolved in this thesis.

Our further contribution is to propose a set of new upper bounds for BPP. These upper bounds are constructed and developed by using the solutions of the *continuous relaxation of the set-covering formulation* of BPP, and brings about more justifications to perform a time-consuming procedure such as column generation. For, shortly after the lower bound of the BPP is provided by solving the continuous relaxation of the set-covering formulation of the problem by column generation, a strong upper bound is derived for the BPP by utilizing the information of the last iteration of the column generation. In other words, affording additional operational costs to obtain the upper bound of the BPP by employing the well-known, yet not well-performed, *heuristics* is bypassed by resorting to said bounds.

1.2 Thesis outline

Chapter 2: In this chapter, we will review the different aspects of BPP and the methods developed in the literature to solve this problem.

Chapter 3: This chapter will present solving procedure of the master problem of set-covering formulation of BPP by column generation. A new method of warm starting the column generation to reduce the computational time of the process will also be presented.

Chapter 4: In this chapter we will investigate the structure of the solutions provided by solving the continuous relaxation of set-covering formulation of BPP to propose an upper bounding procedure for BPP by using these solutions.

Chapter 5: This chapter will be an account of the improvements on the upper bounding procedure developed in chapter 4.

Chapter 6: The focus of this chapter will be employing the information provided by dual version of BPP to increase the efficiency and performance of the methods developed in chapter 5. The general scheme for deriving an upper bound for BPP to be used in branch-and-price procedure will also be presented in this chapter.

Chapter 7: In the final chapter of the thesis, a branch-and-price procedure is proposed to solve the BPP instances to optimality or prove the optimality of the obtained upper bound solutions. This branch-and-price procedure employs a generic branching strategy developed based upon the results obtained from the previous chapters as well as a new search strategy called *batch diving*.

Chapter 2

Literature Review

2.1 Definitions

In this section, we define the most important terms that will be used throughout the thesis. The notations and definitions are borrowed from [1], and more details are available in the same reference.

Definition 2.1 (Linear program)

Let $n, m \in \mathbb{N}$, $\mathbf{C} \in \mathbb{R}^n$, $\mathbf{A} \in \mathbb{R}^{m \times n}$, and $\mathbf{b} \in \mathbb{R}^m$. The following optimization problem is called a *linear program* (LP).

$$\begin{aligned} &\max \mathbf{CX} \\ &\mathbf{AX} \leq \mathbf{b} \\ &\mathbf{X} \in \mathbb{R}_+^n \end{aligned} \tag{2.1}$$

where $\mathbb{R}_+^n$ denotes non-negative real numbers.

Definition 2.2 (Integer program)

Let $n, m \in \mathbb{N}$, $\mathbf{C} \in \mathbb{R}^n$, $\mathbf{A} \in \mathbb{R}^{m \times n}$, and $\mathbf{b} \in \mathbb{R}^m$. The following optimization problem is called an *integer program* (IP).

$$\begin{aligned} &\max \mathbf{CX} \\ &\mathbf{AX} \leq \mathbf{b} \\ &\mathbf{X} \in \mathbb{Z}_+^n \end{aligned} \tag{2.2}$$

where $\mathbb{Z}_+^n$ denotes non-negative integer numbers.

Definition 2.3 (Convex combination)

Given a set $S \subseteq \mathbb{R}^n$, the point $\mathbf{X} \subseteq \mathbb{R}^n$ is a *convex combination* of points of S if there exists a finite set of t points of $\mathbf{X}$ in S and a $\mathbf{\Lambda} \in \mathbb{R}_+^t$ with $\sum_{i=1}^t \lambda_i = 1$ and $\mathbf{X} = \sum_{i=1}^t \lambda_i \, x_i$.

Definition 2.4 (Convex hull)

Given a set $S \subseteq \mathbb{R}^n$, the set of all points that are convex combinations of S is called *convex hull* of S and is denoted by $conv(S)$.

2.2 Problem formulation

2.2.1 Compact formulation of BPP

The BPP can be formulated as the following integer programming model, formulated as a set of equations that read:

$$\begin{aligned} &\min \sum_{k=1}^{K} \lambda_k \\ &\textstyle\sum_{k=1}^{K} x_{ik} = 1 && i = 1, \dots, n \\ &\textstyle\sum_{i=1}^{n} w_i x_{ik} \le c\lambda_k && k = 1, \dots, K \\ &\lambda_k \in \{0,1\} && k = 1, \dots, K \\ &x_{ik} \in \{0,1\} && i = 1, \dots, n\ ,\ k = 1, \dots, K. \end{aligned} \tag{2.3}$$

where λ is load of a pattern and K denotes the maximum number of loading patterns. Also, $x_{ik} = 1$ if item $i \in I$ is accommodated into pattern k, otherwise it is 0. The first constraint ensures that all the items are assigned to the bins, and the second constraint requires that summation of the weights of the items in each bin does not exceed the bin capacity.

Dropping the assumption of integrality, we get the following linearly relaxed model:

$$\begin{aligned} &\min \sum_{k=1}^{K} \lambda_k \\ &\textstyle\sum_{k=1}^{K} x_{ik} = 1 && i = 1, \dots, n \\ &\textstyle\sum_{i=1}^{n} w_i x_{ik} \le c\lambda_k && k = 1, \dots, K \\ &\lambda_k \in [0,1] && k = 1, \dots, K \\ &x_{ik} \in \{0,1\} && i = 1, \dots, n\ ,\ k = 1, \dots, K. \end{aligned} \tag{2.4}$$

Model (2.4) has a trivial LP bound of:

$$\begin{aligned} &LP = \frac{\sum_{i=1}^{n} w_i}{c} \\ &\lambda_k = \frac{LP}{K} && k = 1, \dots, K \\ &x_{ik} = \frac{1}{K} && i = 1, \dots, n\ ,\ k = 1, \dots, K. \end{aligned}$$

Not only is this straightforward derivation of the LP bound poorly conceived and its use in branch-and-bound procedure results in lowering the performance of the latter, but also the structure of the compact formulation is highly symmetrical and this is an additional reason why this formulation of BPP does not suit the nature of the branch-and-bound trees [2,3].

2.2.2 Set-covering formulation of BPP

As a remediation to the effects of the weak LP bound provided by compact formulation of BPP, Dantzig-Wolfe decomposition, presented in [4], could be applied to partition the constraints of the model (2.3) into a set of master constraints and a set of sub-problem constraints [2]. These partitions break the symmetric structure of the compact formulation and provide the means for the model to be solved by column generation.

Column generation is an approach to solve linear programming models where the number of variables compared to the number of constraints is relatively large and enumerating all of the patterns to be used in simplex methods is not cost-effective. Even though the expression "column generation" never appears in the paper by Ford and Fulkerson in [5], where the non-basic patterns are generated repeatedly by solving a sub-problem and then added to the master problem until the optimal solution is found, one could safely associate the origin of this method to these authors, according to Nemhauser in [2]. The details of column generation will be elaborated later in this chapter, but first we shall address the deficiencies of the compact formulation and the way to overcome them by the model introduced by Vance in [6].

In reference [6], Vance utilizes Dantzig-Wolfe decomposition and replaces the second constraint of the model (2.3), also known as knapsack constraint, by a convex combination of the extreme points $(x_{1k}^j, x_{2k}^j, \dots, x_{nk}^j)^T\ j = 1, \dots, P$ of the set:

$$conv\{x_k : \sum_{i=1}^{n} w_i x_{ik} \leq c, x_{ik} \in \{0,1\} \text{ for } i = 1, \dots, n\}.$$

The new formulation is expressed as the following equations that read:

$$\begin{aligned}
&\min \sum_{k=1}^{K} \sum_{j=1}^{P} \lambda_k^j \\
&\textstyle\sum_{k=1}^{K} \sum_{j=1}^{P} x_{ik}^j \lambda_k^j = 1 && i = 1, \dots, n \\
&\textstyle\sum_{j=1}^{P} \lambda_k^j \leq 1 && k = 1, \dots, K \\
&\textstyle\sum_{j=1}^{P} x_{ik}^j \lambda_k^j \in \{0,1\} && i = 1, \dots, n, k = 1, \dots, K \\
&\lambda_k^j \geq 0 && j = 1, \dots, P, k = 1, \dots, K.
\end{aligned} \tag{2.5}$$

Here, the first set of constraints ensures, once more, the assignment of all the items to patterns and the convexity constraints guarantee that the patterns are in the convex hull of the binary solutions of the knapsack problem. Furthermore, the integrality constraint demands that convex combination of the extreme points in the convex hull should result in a binary pattern.

Vance simplifies the above formulation by replacing the extreme points of the convex hull by a subset of feasible solutions of the knapsack constraints. Assuming patterns $(x_{1k}^j, x_{2k}^j, \dots, x_{nk}^j)^T\ j = 1, \dots, P'$ as representing the feasible loading patterns to:

$$\sum_{i=1}^{n} w_i x_{ik} \leq c,\ x_{ik} \in \{0,1\}\ \ i = 1, \dots, n,$$

then, it becomes possible to drop the convexity and integrality constraints requiring that $\lambda_k^j \in \{0,1\}$. The resulting simplified model is equivalent to the model, first introduced by Gilmore and Gomory in [7], and is expressed by the following set of equations:

$$\begin{aligned} &\min \sum_{k=1}^{P'} \lambda_k \\ &\textstyle\sum_{k=1}^{P'} x_{ik}\lambda_k = 1 && i = 1, \dots, n \\ &\lambda_k \in \{0,1\} && j = 1, \dots, P'. \end{aligned} \tag{2.6}$$

It is worth to note that P' is an upper bound to the number of feasible loading patterns. The model of Gilmore and Gomory has a strong LP bound and is considered a cornerstone in most of the advances in the field of integer programming.

An alternate set-covering formulation is derived by replacing the equality constraint of (2.6) by an inequality one. This second model is equivalent to its original one, but it facilitates implementation challenges of finding *basic feasible solution* (BFS) when the model is solved by simplex-based approaches. Initializing column generation by a high quality BFS has direct impacts on the total number of columns generated as prescribed by Belov and Scheithauer in [8], where they use a FFD-like sub-diagonal matrix to generate the BFS.

In the following, the alternate model is presented as:

$$\begin{aligned} &\min \sum_{k=1}^{P'} \lambda_k \\ &\textstyle\sum_{k=1}^{P'} x_{ik}\lambda_k \geq 1 && i = 1, \dots, n \\ &\lambda_k \in \{0,1\} && j = 1, \dots, P'. \end{aligned} \tag{2.7}$$

We indicate here, that the model derived above, has a same formulation as that of, the *one-dimensional Cutting Stock Problem* (CSP). In CSP, given a set of items, each of which has a specific weight and a demand associated to the items, one is asked to find a minimum number of cutting patterns so that the demands are satisfied and summation of the weights of the items in each pattern does not exceed the capacity. These two problems, CSP and BPP, share the same nature and the only difference between them is the perspective from which the problem is investigated. More precisely, in BPP, items are labeled; have we had several items with the same weights, BPP distinguishes among them and appends a label to each item, while CSP associates all the items of the same size a required demand. It obviously follows that in CSP non-repeated items have demands equal to 1. Nevertheless, either formulation leads to the same optimal solution and the minimum number of the patterns (bins) is found.

2.3 Column generation

Column generation is a technique to solve huge linear and integer programming problems as mentioned earlier. Nemhauser presents a complete history of column generation in his paper [2]. Generally speaking, all column generation techniques involve establishing relations between linear programming models called master problems and their corresponding pricing sub-problems. Regardless of the structure of the master problem and its corresponding pricing sub-problem, the master problem always relies on its pricing sub-problem for the generation of new patterns, and the pricing sub-problem is called until a specific termination condition is filled.

In BPP, the master problem is the same one we presented in model (2.7) and its pricing sub-problem is a one-dimensional knapsack problem. Column generation starts by defining a problem called restricted master problem which includes a basic feasible solution to the model. Then, the restricted master problem is solved by a revised simplex method to optimality, and thereafter its dual variables or shadow prices are assumed as being the cost function of the one-dimensional knapsack problem. The constraint of the pricing sub-problem ensures that the generated column meets the capacity constraint of the problem. The generated column will then be added to the restricted master problem and the information of the revised simplex method will be updated. The structure of the pricing sub-problem for BPP reads:

$$\begin{aligned} &\max \sum_{i=1}^{n} \pi_i \, x_i \\ &\textstyle\sum_{i=1}^{n} w_i x_i \leq c \\ &x_i \in \{0,1\} \qquad\qquad i = 1, \dots, n \end{aligned} \tag{2.8}$$

where $\mathbf{\Pi} = (\pi_1, \pi_2, \dots, \pi_n)$ is a vector of shadow prices and $\mathbf{X} = (x_1, x_2, \dots, x_n)$ solution to the knapsack problem.

A column generated by solving the knapsack problem (2.8) will terminate column generation procedure in case its reduced cost $(1 - \mathbf{\Pi X})$ turns negative. Were it to be the case, termination of column generation dictates that all the non-basic patterns with positive reduced costs are exhausted. It is also worthwhile to mention, that patterns can either be

generated or added to the master problem sequentially, or else, a pool of patterns with positive reduced costs could be generated and then added to the master problem. Either approach used, the method for solving pricing sub-problem should be capable of finding the exact solution responsible for terminating column generation; otherwise, column generation might be terminated prematurely, in which case, its objective value would not represent a lower bound of the problem.

An alternative method for generating patterns might be using a two-phase method as presented in [9]. In the first phase, knapsack problem is solved by a fast heuristic or an approximation algorithm until the reduced cost of the generated column gets negative. At this point, the second phase is invoked in which an exact method is used to prove the optimality of the column generation. This approach is still subjected to scrutiny of researchers in the area. On the one hand, it is likely that the use of heuristics and approximation algorithms will increase the number of generated patterns; on the other hand, heuristics and approximation algorithms run faster compared to exact methods, and generating more patterns does not necessarily imply increasing the total computational time of the column generation procedure. To the best of our knowledge, there is no general rule of thumb governing all the instances of BPP to set a threshold to control the tradeoff between the number of columns generated and the overall computational time.

The other challenging part when implementing the revised simplex method and column generation has to do with the update of the basis performed repeatedly. Updating the basis requires adding the generated column to the basis and then finding the inverse of the basis for subsequent use. However, it is not affordable to find the inverse of the basis at each step of column generation. For, success in finding the inverse of the large matrices relies heavily on the condition number of the matrix. Condition number of a matrix A is denoted by $\kappa(A)$, and it is a measure of how a small change in elements of the matrix A will affect the outputs. More precisely, having a matrix A with condition number $\kappa(A) = 10^k$, we might loose up to k digits of accuracy in the output of the inverse operation. The accumulated error when finding the inverse of the basis at each step of column generation directly affects the final results of the procedure. Even for well-conditioned matrices where the condition number is relatively small, the procedure of finding the inverse matrix is quite time-consuming and it

should be avoided in column generation where the basis is updated for thousands of times even for relatively small BPPs with 200 items.

To this end, instead of inverting the basis at each step of column generation, the inverse matrix is computed at the first step when the restricted master problem is defined, and then as the new columns are presented to the master problem, the inverse of the basis is updated. Furthermore, most of the times, it is beneficial to find the LU factorization of the basis at the first step of column generation, and then update the LU factorization of the basis when new columns are presented. Elble and Sahinidis in [10] reviewed the most recent techniques for updating the basis for simplex methods. Employing LU factorization of basis reduces computational time of the column generation since triangular matrices derived from LU factorization can be saved as sparse matrices which in turn reduces the memory in use and affects the overall computational time of the procedure.

2.4 Round up properties of set-covering formulation of BPP

Set-covering formulation of BPP provides a strong lower bound for the number of optimal bins used as mentioned before. Not only is this bound strong, it also holds interesting properties which are additional reasons for the vast use of set-covering formulation of BPP in the literature. These properties are discussed in the following sections.

2.4.1 Round-up property (RUP)

An instance I is said to hold a round-up property (RUP) if its optimal value, $OPT(I)$, equates with that of the LP relaxation, $z_c(I)$. In other words:

$$OPT(I) = z_c(I).$$

2.4.2 Integer round up property (IRUP)

Baum and Trotter in [11] introduced the concept of the integer round-up property (IRUP), and Marcotte in [12] using decomposition properties of certain knapsack polyhedral proved that integer round-up property (IRUP) holds true for certain classes of CSP. An instance I is said to have integer round up property (IRUP) if the difference of its optimal value, $OPT(I)$, and its LP relaxation, $z_c(I)$, is less than one. In mathematical terms this is expressed by:

$$OPT(I) = \lceil z_c(I) \rceil.$$

2.4.3 Modified integer round up property (MIRUP)

Marcotte in [13] encountered an instance of CSP for which the integer round up property (IRUP) does not hold true. Initiated by Marcotte, Scheithauer and Terno [14] introduced the concept of the modified integer round up property (MIRUP), and they proposed a conjecture in which all the instances of CSP feature modified integer round up property (MIRUP). An instance I is said to have modified integer round up property (MIRUP) if its optimal value, $OPT(I)$, is not greater than its LP relaxation, $z_c(I)$, plus one. Put explicitly:

$$OPT(I) \leq \lceil z_c(I) \rceil + 1.$$

They also presented a new approach to solve CSP which is predicated on this conjecture. Hitherto, this conjecture holds true for all the instances of BPP and CSP found in the literature.

2.5 Round up solutions

Another interesting aspect of the continuous relaxation of set-covering formulation is that basic solutions are deemed near-optimal (optimal), and by this token they are useful to derive the optimal solution of BPP. References [15,16] are accounts of the rounding basic solutions procedure of LP relaxation and how the consequent residual problems are dealt with, using a heuristic algorithm called *sequential value correction* (SVC). However, these methods fail to establish the optimal solution for BPP for a considerable number of instances. With respect to the fact that basic solutions of LP relaxation of BPP are near-optimal, Bansal et *al*., in [17], proposed an approximation method based on a combination of the randomized rounding of the near-optimal solutions. But, their approximation method has not been tested on the benchmark instances of BPP in order to compare its results with state-of-the-art methods.

2.6 Branch-and-price

The term branch-and-price points to a category of branch-and-bound procedures in which the use of column generation is adapted. Branch-and-price starts by solving the continuous relaxation of the set-covering formulation of BPP at the root node by column generation, and then a branch-and-bound procedure is invoked in search of the optimal solution. At all of the other nodes of the tree, column generation is again used to find the lower bound of the nodes. The performance of branch-and-price procedure inherently depends on the quality of the upper and lower bounds found at each node, and more importantly its efficiency relies on the branching as well as the search strategies.

2.6.1 Branching Strategies

A standard branching scheme might branch on basic patterns of the LP relaxation of the set-covering formulation of BBP. It is trivial to solve the left branch (also known as branch of Ones) since it simply requires, items of a particular pattern to be included in the solution. Therefore, these items could be eliminated from the original landscape of the problem, and then the remaining problem is solved to optimality. However, dealing with the right branch (also known as branch of Zeros) is a daunting task. The reason for this is the fact that this branch contains the pattern that should be excluded from the feasible search space of the knapsack problem.

To manage the polluted search space of the knapsack problem, different directions could be taken. These directions could be classified in the categories listed below.

2.6.1.1 Finding the k-best solutions of knapsack problem

The first approach to tackle the polluted knapsack problem comes from the very nature of the branch-and-bound trees. Simply put, at the $(k\text{-}1)^{th}$ level of the tree, one needs to generate k patterns. This guarantees that enough patterns are available in case all of the forbidden patterns are met while proceeding with column generation. But, this simplistic approach, nonetheless, is not practical to implement, considering the scarcity of reliable methods of finding the k-best solutions of knapsack problems, in the literature.

To make a case, Leão et *al*., in [18], propose quite an effective branch-and-bound procedure to find the k-best solutions for both bounded and unbounded knapsack problems. Their method is based on the long- and short-backtracking techniques first introduced by Gilmore and Gomory in [19], and it is capable of solving large-scale knapsack problems in a reasonable time. Though, this method is effective, so long that the aim is to solve the integer knapsack problems rather than the binary ones. In the case of binary knapsack problems, the powerful long-backtracking tool is not utilized, rather the authors resort to the short-backtracking ones. Solving then, reduces the branch-and-bound procedure to a complete enumeration of the tree. Thus, their proposed method cannot be employed cost-effectively in the branch-and-price method for BPP.

Beside the attempts to find the k-best solutions of the knapsack problem by branch-and-bound- and dynamic programming-based methods, Sarin et *al*., in [20], propose a method stemming from schedule algebra [21]. Although introducing this new perspective in solving knapsack problems itself, is valuable enough, the authors in [20], never compared the efficiency and correctness of their method with competing methods from the literature, and as far as we are concerned, there exists no guarantee that their method is sufficient to determine the k-best solutions of knapsack problem [18].

2.6.1.2 Implicit methods in exclusion of the forbidden patterns from search domain

This category refers to a set of branching rules which are compatible with the nature of the polluted pricing sub-problem of column generation. The idea was pioneered by Ryan and Foster in [22], and it was elaborated by Vance et *al*., in [30] and Barnhart et *al*., in [23]. Basically, these methods attempt to find the appropriate pairs of items for branching which will avoid adding extra constraints to the pricing sub-problem. More specifically, for each fractional basic solution, there exists a fractional counterpart in the basis which complements its value and this leads to a pair of branching constraints. To comprehend the quintessence of this approach, let us assume that each column of the matrix **M** representing a basic solution of the LP relaxation of the set-covering formulation of BPP is expressed as:

$$\mathbf{M} = \begin{matrix} 1 \\ r \\ \\ \\ \\ \\ \\ n \end{matrix} \begin{bmatrix} . & . & . & . \\ . & . & . & . \\ . & . & . & . \\ . & y_{rk'} = 1 & . & . \\ . & . & . & . \\ . & . & . & . \\ . & . & . & . \\ . & . & . & . \end{bmatrix}$$

where the value for pattern $\lambda_{k'}$ is fractional and the rows correspond to the labels of the items.

Because in pattern $\lambda_{k'}$, the item r is present, then there must be another fractional pattern, namely $\lambda_{k''}$, for which the row r is 1. This is shown in the following matrix $\mathbf{M}'$:

$$\mathbf{M}' = \begin{matrix} 1 \\ r \\ \\ \\ \\ \\ \\ n \end{matrix} \begin{bmatrix} . & . & . & . & . \\ . & . & . & . & . \\ . & . & . & . & . \\ . & y_{rk'} = 1 & . & y_{rk''} = 1 & . \\ . & . & . & . & . \\ . & . & . & . & . \\ . & . & . & . & . \\ . & . & . & . & . \end{bmatrix}.$$

Since the basis is free of the duplicated columns, Barnhart et *al*., in [23] concluded that there must be a row *s* such that either $y_{sk'} = 1$ or $y_{sk''} = 1$ but not both. Different possibilities of such a situation can be viewed in the following matrices:

$$\begin{matrix} 1 \\ \\ r \\ s \\ \\ n \end{matrix} \begin{bmatrix} . & . & . & . & . \\ . & . & . & . & . \\ . & . & . & . & . \\ . & y_{rk'} = 1 & . & y_{rk''} = 1 & . \\ . & . & . & . & . \\ . & y_{sk'} = 1 & . & y_{sk''} = 0 & . \\ . & . & . & . & . \\ . & . & . & . & . \end{bmatrix}$$

and,

$$\begin{matrix} 1 \\ \\ r \\ s \\ \\ n \end{matrix} \begin{bmatrix} . & . & . & . & . \\ . & . & . & . & . \\ . & . & . & . & . \\ . & y_{rk'} = 1 & . & y_{rk''} = 1 & . \\ . & . & . & . & . \\ . & y_{sk'} = 0 & . & y_{sk''} = 1 & . \\ . & . & . & . & . \\ . & . & . & . & . \end{bmatrix}.$$

The submatrix shown in figure (2.1) summarizes our discussion.

	k'	k''
r	1	1
s	0	1

Figure 2.1 Submatrix representing branching pairs [30].

The pair *(r,s)* will then be assumed as the pair of branching constraints, having a left branch as $\sum_{k:y_{rk}=1,y_{sk}=1} \lambda_k = 1$ and a right branch as $\sum_{k:y_{rk}=1,y_{sk}=1} \lambda_k = 0$.

Even though one might benefit from elimination of the need to add an extra constraint to the pricing sub-problem by following this approach, the reduction in the dimension of the problem after branching is not significant, since branching occurs on a pair of items and the consequent nodes will have dimensions close to their parent ones. The generalization of this method to branch on larger subsets of items is yet to be done and it is not clear how extension of the method to larger subsets could be solved efficiently [3].

2.6.2 Search strategies

Another important factor in successful and efficient implementation of branch-and-price methods is the order the nodes of the tree are searched. In a *breadth-first* search strategy, one explores the nodes located on the same level and then moves to the other levels and exploration continues until a termination condition is met.

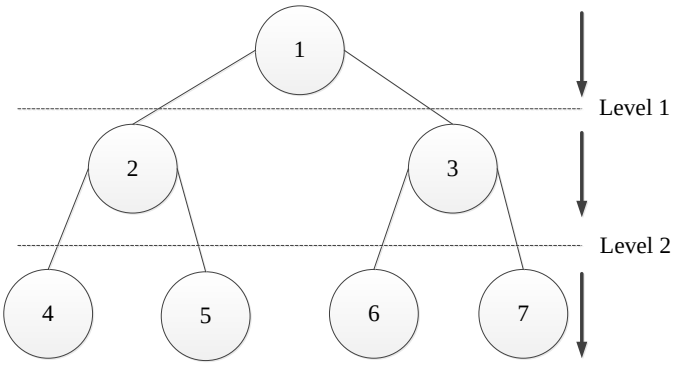


Figure 2.1 Demonstration of the breadth-first search strategy in branch-and-bound tree.

The advantage of the breadth-first node search strategy lays in the fact that the global lower bound of the problem gets improved as one proceeds in the depth of the tree, since the lower bound at each level is equivalent to the minimum of the lower bounds of the nodes included in that level. For instance, lower bound at level 1 of figure (2.1) equals to:

$$\min\{LB(2), LB(3)\}$$

where LB(.) denotes the lower bound of a specific node. To give another example, lower bound at level 2 of figure (2.1) can be derived as:

$$\min\{LB(4), LB(5), LB(6), LB(7)\}$$

and the same premise holds for the rest of the levels.

It is noteworthy that breadth-first search strategy will play a key role in establishing the efficiency of the branch-and-price method if the lower bound derived at the root node is not close to the optimal value of the problem. In case of set-covering formulation of BPP, considering that the LP bound at the root node is strong and it will not get improved significantly as one proceeds deep into the tree, the other search strategies might be used to explore the nodes.

Another well-known and widely-used search strategy is *depth-first* search strategy. Unlike the breadth-first strategy where the focus is put on improving the global lower bound, the goal of the depth-first strategy is to improve the quality of the global upper bound of the problem. One special case of the depth-first strategy is called *diving* according to work described in [24], where the left branches (branches of Ones) are always chosen until reaching to an integer solution. Baladi et *al.*, in [25], reported a high quality of the upper bound derived by diving in their branch-and-price method for generalized bin packing problems. Further comparison of the different search strategies implemented in integer programming software systems could be found in [24] where Atamtürk and Savelsbergh thoroughly investigated the impacts of using depth-first, breadth-first, *best-bound*, and *worst-bound* search strategies on the efficiency of the branch-and-bound procedures.

2.6.3. Termination conditions

Regardless of the chosen search strategy for exploring the nodes, exploration of all of the exponential number of nodes is not affordable and a set of termination conditions must be set to terminate the branch-and-bound procedure. There are different choices for the termination condition in the literature among which *Maximum Nodes Explored, Maximum Time Spent,* and *Relative Gap between Upper and Lower Bounds* as well as *Absolute Gap between Upper and Lower Bounds,* have received more attention. Depending on the structure of the problem, combination of such criteria could be applied to terminate the branch-and-bound procedure.

2.6.4 Primal approximations, heuristics, and meta-heuristics

Primal *approximations, heuristics*, and *meta-heuristics* are attributed to algorithms which are able to find feasible solutions to integer programming problems in a short amount of time. These algorithms are used at each node of the branch-and-price tree (or depending on the problem only at the root node) to strengthen (or derive) the upper bound. That is, primal techniques attempt to find high quality solutions early in the search tree to reduce the number of nodes explored. For instance, if a tight upper bound is found at the root node, then the branch-and-price will be used only to prove the optimality of the obtained solution, and the procedure will be terminated if a termination condition like Maximum Nodes Explored is met. The use of such techniques along with a branch-and-bound procedure is highly recommended to reduce computational time of the procedure [24].

We would like to elaborate more on the threefold name of the methods capable of finding an upper bound for BPP. Approximations are those methods for which the proof of goodness of the approximate exists. In other words, one is able to prove how close the solutions of the approximation methods are to the optimal solution. Different design and analysis schemes for approximation algorithms are treated in [26].

Heuristics, however, are attributed to the methods which lack the proof for goodness of the approximate. That is, there exists no prior knowledge on the closeness of the solutions

provided by heuristics to the optimal solution, and empirical investigation of the designed heuristics on benchmark instances is the only way to gain knowledge about their performance.

Lastly, meta-heuristics are mainly higher-level heuristic methods. Although there is no vivid boundary between heuristics and meta-heuristics, what distinguishes meta-heuristic methods is the vast use of stochastic processes they rely upon. Usually inspired by nature, meta-heuristics attempt to search the solution domain of the problem for an optimal solution by using innovative operators to escape the local optimal points and move toward the global one. Similarly to heuristics, the goodness of these methods could only be investigated empirically.

In the following sections, we will shortly describe the most renowned of primal methods in the literature.

2.6.4.1 Approximations

One of the widely accepted analysis schemes for goodness of the approximation algorithms is assessing their performance by assigning them a positive real number called *worst-case performance ratio.* Considering U_{BPP} to be the universal set of all BPP instances, $OPT(I)$ to be the optimal value of an instance I for a minimization problem, and $A(I)$ to be the value of approximation algorithm for instance I, then *worst-case performance ratio* of approximation algorithm A, denoted by $\mathcal{R}_A$, is the minimum real number greater than 1 such that:

$$\frac{A(I)}{OPT(I)} \leq \mathcal{R}_A, \quad \forall I \in U_{BPP}.$$

Listing all the approximation algorithms proposed for BPP and investigating the different aspects of the worst-case performance ratio such as its asymptotic behavior, are beyond the scope of this research. For of this work, it only suffices to mention few approximation methods which might provide an insight into the upper bounding techniques, we will develop in the next chapters. As far as we know, Coffman et *al.*, in [28] reviewed the most recent advances in approximation algorithms, and more information can be found in their review paper.

2.6.4.1.1 First fit Algorithm (FF)

First fit algorithm (FF) packs each item in the first partially filled bin found, that has sufficient residual capacity to accommodate the item; otherwise the algorithm opens a new bin and assigns the item to it. For FF, worst-case performance ratio is derived in [29] as:

$$\mathcal{R}_{FF} = \frac{17}{10}.$$

2.6.4.1.2 First fit decreasing Algorithm (FFD)

First fit decreasing algorithm (FFD) is a first fit algorithm where the items are sorted in decreasing order of the weights in advance. For FFD, worst-case performance ratio is derived in [31] as being:

$$\mathcal{R}_{FFD} = \frac{3}{2}.$$

2.6.4.1.3 Best fit algorithm (BF)

Best fit algorithm (BF) keeps track of the residual capacity of the open bins and assigns each item to a bin having the maximum fill. If the item is not packable in any of the open bins, a new bin will be opened to pack it. For BF, *worst-case performance ratio* is derived in [30] as:

$$\mathcal{R}_{BF} = \frac{17}{10}.$$

2.6.4.1.4 Best fit decreasing algorithm (BFD)

Best fit decreasing algorithm is a best fit (BF) algorithm with the difference that items are sorted in descending order of weights in advance. For BFD, *worst-case performance ratio* is derived in [31] as being:

$$\mathcal{R}_{BFD} = \frac{3}{2}.$$

2.6.4.2 Heuristics and meta-heuristics

2.6.4.2.1 First fit algorithm with n pre-allocated-items (FF-n)

This recently proposed algorithm first opens a separate bin for each of the items having weights greater than half of the capacity, and then packs the rest of the items by applying first fit algorithm on random permutation of the remaining set of items [27].

2.6.4.2.2 BISON

Proposed by Scholl et *al*., in [32], the fast hybrid procedure for exactly solving BPP (BISON) employs tabu search and a branch-and-bound procedure.

2.6.4.2.3 HI_BP

The hybrid improvement heuristic (HI_BP) proposed by Alvim et *al*., in [33] is inspired by the BISON method of Scholl et *al*., in [32], and the authors bring the reduction technique of Martello and Toth [34] into their method to improve the quality of the solutions obtained. Basically, the reduction technique of Martello and Toth proposed in [34], includes a dominance criterion and a sufficient condition to investigate the dominance relation among bins, and it suits well the relatively easy BPP instances where the distribution of the weights is uniform and the problem contains as much as large items as the small ones.

2.6.4.2.4 Perturbation-SAWMBS

Fleszar and Charalambous in [35], proposed an average-weight-controlled bin oriented heuristic for BPP (Perturbation-SAWMBS) in which an operator is employed to avoid early packing of the small items which is the tendency of the majority of the heuristics for BPP and usually causes undesired impacts on the outcome of the procedures.

2.6.4.2.5 Hybrid grouping genetic algorithm (HGGA)

The evolutionary hybrid grouping genetic algorithm for BPP (HGGA) is one of the first successful implementation of the population-based algorithms to explore the solution space of BPP. Falkenauer in [36], used problem-oriented crossover, mutation, and selection operators along with the dominance criterion of Martello and Toth reported in [34], to derive an upper bound for BPP.

2.6.4.2.6 Grouping genetic algorithm with controlled gene transmission (GGA-CGT)

Proposed by Quiroz-Castellanos et *al.*, in [27], grouping genetic algorithm with controlled gene transmission (GGA-CGT) advances the operators of its mother-type algorithm, HGGA, to improve the quality of the solutions found for BPP. The method GGA-CGT, currently outperforms all the other heuristic and meta-heuristic methods from the literature. Yet, it fails to find optimal solutions for some benchmark instances.

2.6.4.2.7 Weight annealing heuristic (WA)

Weight annealing heuristic (WA), for solving BPP was proposed by Loh et *al.*, [37]. Basically, in the weight annealing approach, weights of the items packed in bins are inflated to allow more movements between the items of different bins to perform local searches with the aim of obtaining better solutions for the BPP. Subsequently, the inflation rate is decreased through execution of further iterations of the algorithm, and this resides at the core of the annealing process. At the final iteration of the weight annealing, the inflation rate becomes zero which corresponds to the original landscape of the problem.

2.6.5 Algorithms for solving pricing sub-problem (one-dimensional knapsack problem)

The literature of the methods of solving one-dimensional knapsack problems is rich and this is due to its vast use in operational research as a constitutive model that is widely encountered. Different methods and reduction schemes exist to exactly solve one-dimensional knapsack problems in most of which, dynamic programming and branch-and-bound procedures are extensively used [1]. These simple dynamic programming and branch-and-bound procedures could solve *weakly correlated* and *uncorrelated* instances to optimality, even for large numbers of items [38]. Weakly correlated instances are those for which the correlation between profit and weight of each item is loose. Also, for uncorrelated instances, there exist no correlation between profit and weight of each item.

However, for *strongly correlated* instances, more advanced methods should be employed to solve the problem to optimality. An extreme case of strongly correlated instances could be seen in *subset-sum knapsack problems* where profit and weight of each item are equal. Were it to be the case, branch-and-bound procedure would be completely ineffective [39]. The reason for this is that the majority of the upper bounding techniques for one-dimensional knapsack problems rely on the associated efficiencies. An efficiency of a specific item i in model (2.8) is defined as:

$$e_i = \frac{\pi_i}{w_i}.$$

Since in subset-sum knapsack problems, all of the efficiencies of items take the value 1, all of the mentioned upper bounds give the trivial value of capacity c. Therefore, catastrophic behavior of the branch-and-bound procedure is not far-fetched, because with a lack of an upper bound, the procedure will turn into a complete enumeration of the tree.

To this end, more advanced methods should be invoked to solve strongly correlated and subset-sum knapsack problems. Among such methods, the branch-and-bound scheme of Martello and Toth reported in [40], is notable where the authors use Lagrangian relaxations of the cardinality constraints generated from extended covers, to derive strong upper bounds for their scheme. Also, Pisinger in [41], initializes his proposed dynamic programming scheme by a core problem, and then the core is expanded until the optimal solution is obtained. The method proposed by Pisinger is able to handle strongly correlated knapsack

problems even for numbers of items as large as 10000. Moreover, Martello et *al.*, in [38], take advantage of the concept of the core problem introduced in [41], and initialize their dynamic programming scheme with a more sophisticated core problem, compared to the one proposed in [41]. Also, the authors of [38], utilize surrogate relaxation of the cardinality constraints to derive strong upper bounds in each state of the dynamic programming. All in all, satisfying results are obtained by using the method of Martello et *al.*, proposed in [38], for solving one-dimensional knapsack problems where all kinds of instances with different correlations could be solved in less than a second, including the case of large-scale problems with 10000 items.

2.7 Benchmark instances

During the past twenty years, different classes of instances have been suggested to evaluate the performance of the methods proposed for solving BPP. Among these classes, the following ones have gained more attention due to the difficulties they entail.

***1) Uniform* class**: This class of instances was proposed by Martello and Toth in [34], where the weights of the items are uniformly random between [20,100], and items are to be packed in bins of capacity $c = 150$. Later, Falkenauer in [36], used 120, 250, 500, and 1000 items, each of which having a weight drawn from the uniform distribution mentioned above, to test the performance of HGGA. These uniform classes, each of them, containing 20 instances, are known as u120, u250, u500, and u1000. It is worth mentioning that all the instances of the uniform class hold IRUP.

***2) Triplet* class**: In this class proposed by Falkenauer in [36], weights of items are drawn uniformly from the interval (0.25,0.50), and the capacity of the bins are set to $c = 1$. Different classes of triplets are available with 60, 120, 249, and 501 items, and are named t60, t120, t249, and t501, respectively. The reason these instances are called triplets is that the optimal solution always contains three items; one big item and two small ones where the big item is defined to have a weight greater than the third of the bin capacity and the small item is defined to have a weight less than the third. It should also be noted that all the instances belonging to this class hold RUP.

3) Hard28* class*: This class is a collection of the 28 most difficult instances for BPP which was gathered by Belov et *al.*, in [8]. This class is composed of items having RUP, IRUP, and MIRUP. For this reason, it is not easy to propose a generalized method to solve all the instances of this class, and this is observable in performance of state-of-the-art methods where none of them have been able to find the optimal solution for all the 28 instances.

The other notable benchmark instances are listed in table (2.1):

Table 2.1 Classes of benchmark instances for BPP

Class	Authors	Reference
Data Set 1	Scholl et *al.*	[32]
***Data Set* 2**	Scholl et *al.*	[32]
Data Set 3	Scholl et *al.*	[32]
Was 1	Schwerin and Wäscher	[47]
Was 2	Schwerin and Wäscher	[47]
Gau 1	Wäscher and Gau	[48]

Chapter 3

Solving the master problem by column generation

Recalling that the set-covering formulation of BPP (2.7) developed by Gilmore and Gomory in [7], which will serve as the master problem of our column generation procedure is defined as:

$$\min \sum_{k=1}^{P'} \lambda_k$$

$$\sum_{k=1}^{P'} x_{ik}\lambda_k \geq 1 \qquad i = 1, \dots, n$$

$$\lambda_k \in \{0,1\} \qquad j = 1, \dots, P'.$$

Where λ_k is a feasible loading pattern, $x_{ik} = 1$ if item i is assigned to pattern k, and 0 otherwise, and where P' denotes an upper bound for the number of the feasible loading patterns.

3.1 Restricted master problem (RMP)

Column generation procedure starts with defining the restricted master problem (RMP). Singleton bins where each item is accommodated in a separate bin can be used to define the RMP. This solution is an obvious and, at the same time, the most expensive upper bound of BPP. To reduce the number of the generated patterns during the procedure of column generation, it is beneficial to employ FFD-based algorithms to define the RMP.

FFD establishes the solutions by first sorting the items in the descending manner of their weights, and then a bin is opened and the first item is assigned to this bin. For the rest of the items, if they fit into the residual capacity of the open bins, they are accommodated, otherwise a new bin will be opened and this process will be continued until all the items are packed.

To illustrate how FFD works, the example given by Martello in [34] is treated.

Example 3.1

Assume the weights $\mathbf{W} = [99,94,79,64,50,46,43,37,32,19,18,7,6,3]$ are given for a set of $n = 14$ items, and the goal is to pack these items in the minimum number of the identical bins of capacity $c = 100$.

Following the steps of FFD, the following packing configuration of items into the bins is obtained:

$B_1 = \{1\}, B_2 = \{2,13\}, B_3 = \{3,10\}, B_4 = \{4,9,14\}, B_5 = \{5,6\}, B_6 = \{7,8,11\}, B_7 = \{12\}.$

To make use of these solutions to define RMP, solutions should be converted into a matrix form where the columns of the matrix account for each pattern and the rows are representatives of the items. The matrix illustrated below, incarnates a BFS for the RMP of our example.

$$\mathbf{BFS} = \begin{array}{c} 1 \\ 2 \\ 3 \\ 4 \\ 5 \\ 6 \\ 7 \\ 8 \\ 9 \\ 10 \\ 11 \\ 12 \\ 13 \\ 14 \end{array} \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 1 \end{bmatrix}$$

As it is notable, this matrix is not a square one whereas the matrix used as the basis in revised simplex method should be a square matrix. At this point, heuristics might come in handy to add columns to the matrix and make it square. Here, the simplest technique would be to add singleton bins starting from the first item. The result of the augmentation could be seen in the following matrix:

$$\mathbf{BFS} = \begin{array}{c} 1 \\ 2 \\ 3 \\ 4 \\ 5 \\ 6 \\ 7 \\ 8 \\ 9 \\ 10 \\ 11 \\ 12 \\ 13 \\ 14 \end{array} \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}.$$

However, when one pays more attention to this square matrix, its rank is less than the dimension of the matrix since there are duplicated columns present in the matrix. That is, the determinant of the matrix is 0, and consequently it is not invertible to be used as the basis for RMP. To this end, further manipulations should be carried out to make it invertible. Instead of performing more operations on the matrix depicted above, we propose a simple heuristic based on FFD to warm start the column generation. This process starts by defining a lower triangular matrix where all the elements on the diagonal are unity, and then filling the lower triangular elements based on a FFD algorithm. Having a diagonal of ones ensures

the independency of the columns as well as the independency of the rows and consequently existence of the inverse matrix. The formal description of this algorithm is presented in table (3.1).

Table 3.1 Pseudo-code for FFD-based BFS algorithm

algorithm FFD-based BFS

Input: number of items (n), weights of the items ($\mathbf{W}_{1\times n}$), bin capacity (c)

Output: square matrix **BFS**

Step 1. Define a lower triangular square matrix of dimension n called **BFS** with diagonal all ones.

Step 2. **For** j=1 to n-1

For i=j+1 to n

If $(\mathbf{W})(\mathbf{BFS}^{j}) + \mathbf{W}_{i} \leq c$

$\mathbf{BFS}_{i,j} = 1$

Else

$\mathbf{BFS}_{i,j} = 0$

End of **i** loop

End of **j** loop

Step 3. Output is **BFS.**

Applying FFD-based BFS algorithm to items of example (3.1), we get the following BFS

$$\mathbf{BFS} = \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \\ 6 \\ 7 \\ 8 \\ 9 \\ 10 \\ 11 \\ 12 \\ 13 \\ 14 \end{array} \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 1 & 1 & 1 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \end{bmatrix}$$

Although FFD-based BFS algorithm provides relatively a good starting point for the revised simplex method compared to singleton bins, it might be possible for basic solutions of the revised simplex method to take on negative values when such a BFS is used to define the master problem. Values of the basic solutions could be denoted by $\mathbf{X}_B$. However, naïve implementation of the revised simplex method relies upon the positive values of $\mathbf{X}_B$ for determination of the leaving pattern of basis. We will discuss the details of the revised simplex method later in this chapter, and for the moment let us see how $\mathbf{X}_B$ is updated during column generation procedure. Considering the right hand side of the constraints to be $\mathbf{b}=\mathbf{1}_{n\times 1}$ in model (2.7), $\mathbf{X}_B$ is derived as:

$$(\mathbf{BFS})(\mathbf{X}_B) = (\mathbf{b}) \tag{3.1}$$

or, equivalently we have:

$$(\mathbf{X}_B) = (\mathbf{BFS})^{-1}(\mathbf{b}) \tag{3.2}$$

where $(\mathbf{BFS})^{-1}$ denotes the inverse of **BFS**.

Back to example (3.1), by applying Gaussian elimination to solve (3.1) we get the following

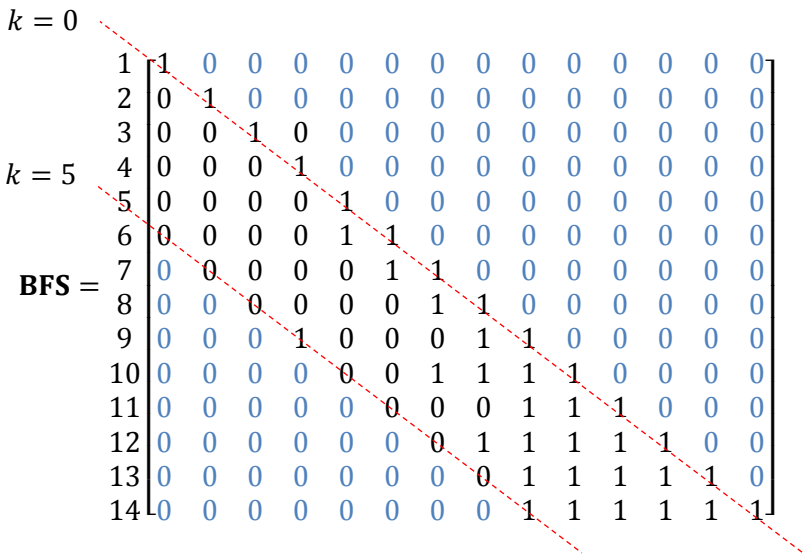
$$\mathbf{X}_B = [1 \quad 1 \quad 1 \quad 1 \quad 1 \quad 0 \quad 1 \quad 0 \quad 0 \quad -1 \quad 2 \quad 0 \quad -1 \quad -1]^T.$$

Vector $\mathbf{X}_B$ derived for this example contains negative elements and special treatments should be applied since negative values of $\mathbf{X}_B$reflect the infeasibility of the basic solutions. Typically, primal-dual simplex methods are invoked in such cases to solve the problem to optimality, but inasmuch as our interest is to implement the standard revised simplex method, a refinement step will be performed on $\mathbf{X}_B$ to ensure the positive values of the elements of $\mathbf{X}_B$.

Definition 3.1 Function TriangularUp(BFS, k)

Function TriangularUp(**BFS**, k) returns all the elements on and above the kth diagonal of the **BFS**.

For instance, TriangularUp(**BFS**, 0) will return the same matrix as **BFS**, and TriangularUp(**BFS**, 5) will return the following matrix where all the elements below the 5th diagonal are zero.



Obviously, TriangularUp($\mathbf{BFS}, 0$) will have a positive $\mathbf{X}_B$, but we are interested in including as many items as possible into the bins while preserving the positive sign of the $\mathbf{X}_B$. The straightforward implementation of refinement process would then be starting from the main diagonal, increasing k and computing the $\mathbf{X}_B$, and continuing the process until a negative $\mathbf{X}_B$ is met. However, having more items packed in the bins suggests a better warm start for column generation. Thus, the refinement process needs to be performed backwards. This means, refinement starts from the last diagonal and k is decreased until a desired **BFS** (**BFS** that leads to a positive $\mathbf{X}_B$) is encountered.

One might argue that the latter approach for refinement has an expensive computational time for large matrices since in the worst case n diagonals should be evaluated, and for each diagonal Gaussian elimination should be employed to solve the system of linear equations. This is generally true, but what makes the use of such an approach an affordable one, is the fact that generating a good **BFS** as the starting point of column generation has direct impacts on the computational time of the procedure. Warm starting the column generation usually reduces total number of the patterns generated during the procedure by a factor of several thousand even for BPPs with relatively small dimensions. It should also be noted that running time of Gaussian elimination is negligible compared to generating new patterns by solving one-dimensional knapsack problems.

The pseudo-code for FFD-based BFS with refinement is presented in Table (3.2).

Table 3.2 Pseudo-code for FFD-based BFS with refinement algorithm

algorithm FFD-based BFS with refinement

Input: number of items (n), weights of the items ($\mathbf{W}_{1\times n}$), bin capacity (c), right hand side of the constraints ($\mathbf{b}_{n\times 1}$)

Output: square matrix BFS

Step 1. Define a lower triangular square matrix of dimension n called **BFS** with diagonal all ones.

Step 2. **For** j=1 to n-1

For i=j+1 to n

If $(\mathbf{W})(\mathbf{BFS}^{j}) + \mathbf{W}_{i} \leq c$

$\mathbf{BFS}_{i,j} = 1$

Else

$\mathbf{BFS}_{i,j} = 0$

End of **i** loop

End of **j** loop

Step 3. **For** k= n to 0

BFS ← TriangularUp(**BFS**, k)

$\mathbf{X}_{B} = (\mathbf{BFS})^{-1}(\mathbf{b})$

If all elements of $\mathbf{X}_{B}$ are non-negative

Break

Else

Continue

End of **k** loop

Step 4. Output is **BFS.**

For the instance from example (3.1), the desired **BFS** is found on the 6^{th} diagonal by applying FFD-based BFS with refinement algorithm, and the matrix is indicated in the following as being:

$$k = 6 \qquad \mathbf{BFS} = \begin{matrix} 1 \\ 2 \\ 3 \\ 4 \\ 5 \\ 6 \\ 7 \\ 8 \\ 9 \\ 10 \\ 11 \\ 12 \\ 13 \\ 14 \end{matrix} \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 1 & 1 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \end{bmatrix}.$$

Note should be made that number of the items lost from the bins due to applying refinement process can vary from one problem to another. For the current example, refinement process eliminates only %11 of the items packed in the bins.

3.2 Revised simplex method

Having defined the RMP by the BFS we derived in the last section, the components of revised simplex method including values of the basic solutions, shadow prices, and inverse of the basis will be computed. Afterwards, a new entering pattern will be generated by solving the one-dimensional knapsack problem, and it will replace the leaving pattern of the basis. Ratio test is the tool to determine the leaving pattern. Then, the inverse of the basis will be updated by using information obtained from entering pattern, and this process will be continued until a termination condition is met. In minimization linear models, revised simplex method ends whenever all the non-basic patterns with positive reduced costs are exhausted. From a practical point of view, considering a tolerance instead of an absolute zero to determine the positive attribute of the reduced costs is essential to terminate the column generation. The details of the revised simplex method are presented in the Table (3.3).

Table 3.3 Pseudo-code for revised simplex algorithm

Algorithm revised simplex

Input: number of items (n), weights of the items ($\mathbf{W}_{1\times n}$), bin capacity (c), right hand side of the constraints ($\mathbf{b} = \mathbf{1}_{n\times 1}$), coefficients of the patterns in objective function ($\mathbf{C} = \mathbf{1}_{n\times 1}$), **BFS**, TolCG $= 10^{-10}$

Output: Final basis of revised simplex method ($\mathbf{B}$), values of the basic solutions ($\mathbf{X_B}$), optimal value of the problem (OPTCG)

Step 1. $\mathbf{B} \leftarrow$ use FFD-based BFS with refinement algorithm to generate **BFS**

Step 2. $\mathbf{B}^{-1} \leftarrow$ Find inverse of matrix $\mathbf{B}$

Step 3. While true

$\mathbf{X}_B = (\mathbf{B}^{-1})(\mathbf{b})$

$\mathbf{\Pi} = (\mathbf{C}^T)(\mathbf{B}^{-1})$

OPT $= (\mathbf{\Pi})(\mathbf{b})$

// Finding the entering column

Solve the knapsack problem $\{\max(\mathbf{\Pi})(\mathbf{X}_e)$ subject to $(\mathbf{W})(\mathbf{X}_e) \leq c, \mathbf{X}_e \in \{0,1\}\}$

// Computing the reduced cost

$r = \mathbf{\Pi X}_e - 1$

If $r \leq$ TolCG

Optimality is proved

Break

Else

$\bar{\mathbf{X}}_e = (\mathbf{B}^{-1})(\mathbf{X}_e)$

If $\bar{\mathbf{X}}_{e_i} \leq 0 \quad \forall i = 1,2,\dots,n$

Problem is unbounded

Break

Else

// Finding the leaving row

$l = \text{argmin}_i \left\{ \frac{\mathbf{X}_{B_i}}{\bar{\mathbf{X}}_{e_i}} \right\} \ \bar{\mathbf{X}}_{e_i} > 0 \ \text{ for } i = 1,2,\dots,n$

$\text{pivot} = \bar{\mathbf{X}}_{e_l}$

// Updating the inverse matrix

Define the identity matrix $\mathbf{A}$

$\mathbf{A}_i^l = -\frac{\bar{\mathbf{X}}_{e_i}}{\text{pivot}} \ \text{ for } i = 1,2,\dots,n, i \neq l$

$\mathbf{A}_i^l = \frac{1}{\text{pivot}} \ \text{for } i = l$

$\mathbf{B}^{-1} \leftarrow (\mathbf{A})(\mathbf{B}^{-1})$

// Updating the basis

$\mathbf{B}^l = \mathbf{X}_e$

End of **while** loop

Step 4. Output is $\mathbf{B}$, $\mathbf{X}_B$, OPTCG.

Following example illustrates iterations of the revised simplex algorithm.

Example 3.2

Given $n = 4$, $\mathbf{W} = [79,64,32,19]$, and $c = 100$, our aim is to pack all the items in the minimum number of the bins used.

First, **BFS** is established using FFD-based BFS with refinement algorithm and it is called matrix **B**.

$$\mathbf{B} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 1 & 1 & 0 \\ 1 & 0 & 1 & 1 \end{bmatrix}$$

$$\mathbf{B}^{-1} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & -1 & 1 & 0 \\ -1 & 1 & -1 & 1 \end{bmatrix}$$

Iteration 1:

$$\mathbf{X}_B = (\mathbf{B}^{-1})(\mathbf{b}) = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & -1 & 1 & 0 \\ -1 & 1 & -1 & 1 \end{bmatrix} \begin{bmatrix} 1 \\ 1 \\ 1 \\ 1 \end{bmatrix} = \begin{bmatrix} 1 \\ 1 \\ 0 \\ 0 \end{bmatrix}$$

$$\mathbf{\Pi} = (\mathbf{C}^T)(\mathbf{B}^{-1}) = \begin{bmatrix} 1 & 1 & 1 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & -1 & 1 & 0 \\ -1 & 1 & -1 & 1 \end{bmatrix} = \begin{bmatrix} 0 & 1 & 0 & 1 \end{bmatrix}$$

$$\text{OPTCG} = (\mathbf{\Pi})(\mathbf{b}) = \begin{bmatrix} 0 & 1 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 \\ 1 \\ 1 \\ 1 \end{bmatrix} = 2$$

$$\mathbf{X}_e = \begin{bmatrix} 0 \\ 1 \\ 0 \\ 1 \end{bmatrix}$$

$$r = \mathbf{\Pi X}_e - 1 = \begin{bmatrix} 0 & 1 & 0 & 1 \end{bmatrix} \begin{bmatrix} 0 \\ 1 \\ 0 \\ 1 \end{bmatrix} - 1 = 1$$

$$\overline{\mathbf{X}}_e = (\mathbf{B}^{-1})(\mathbf{X}_e) = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & -1 & 1 & 0 \\ -1 & 1 & -1 & 1 \end{bmatrix} \begin{bmatrix} 0 \\ 1 \\ 0 \\ 1 \end{bmatrix} = \begin{bmatrix} 0 \\ 1 \\ -1 \\ 2 \end{bmatrix}$$

$$l = \text{argmin}_i \left\{ \frac{\mathbf{X}_{B_i}}{\overline{\mathbf{X}}_{e_i}} \right\} \ \bar{X}_{e_i} > 0 \ \ for \ \ i = 1,2,\dots,4 \Rightarrow l = 4$$

$$pivot = \overline{\mathbf{X}}_{e_l} = \overline{\mathbf{X}}_{e_4} = 2$$

$$\mathbf{A} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & -0.5 \\ 0 & 0 & 1 & 0.5 \\ 0 & 0 & 0 & 0.5 \end{bmatrix}$$

$$\mathbf{B}^{-1} \leftarrow (\mathbf{A})(\mathbf{B}^{-1}) = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & -0.5 \\ 0 & 0 & 1 & 0.5 \\ 0 & 0 & 0 & 0.5 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & -1 & 1 & 0 \\ -1 & 1 & -1 & 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0.5 & 0.5 & 0.5 & -0.5 \\ -0.5 & -0.5 & 0.5 & 0.5 \\ -0.5 & 0.5 & -0.5 & 0.5 \end{bmatrix}$$

$$\mathbf{B}^l = \mathbf{X}_e \Rightarrow \mathbf{B}^4 = \mathbf{X}_e \Rightarrow \mathbf{B} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 1 \\ 0 & 1 & 1 & 0 \\ 1 & 0 & 1 & 1 \end{bmatrix}$$

Iteration 2:

$$\mathbf{X}_B = (\mathbf{B}^{-1})(\mathbf{b}) = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0.5 & 0.5 & 0.5 & -0.5 \\ -0.5 & -0.5 & 0.5 & 0.5 \\ -0.5 & 0.5 & -0.5 & 0.5 \end{bmatrix} \begin{bmatrix} 1 \\ 1 \\ 1 \\ 1 \end{bmatrix} = \begin{bmatrix} 1 \\ 1 \\ 0 \\ 0 \end{bmatrix}$$

$$\mathbf{\Pi} = (\mathbf{C}^T)(\mathbf{B}^{-1}) = [1 \quad 1 \quad 1 \quad 1] \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0.5 & 0.5 & 0.5 & -0.5 \\ -0.5 & -0.5 & 0.5 & 0.5 \\ -0.5 & 0.5 & -0.5 & 0.5 \end{bmatrix} = [0.5 \quad 0.5 \quad 0.5 \quad 0.5]$$

$$\text{OPTCG} = (\mathbf{\Pi})(\mathbf{b}) = [0.5 \quad 0.5 \quad 0.5 \quad 0.5] \begin{bmatrix} 1 \\ 1 \\ 1 \\ 1 \end{bmatrix} = 2$$

$$\mathbf{X}_e = \begin{bmatrix} 0 \\ 0 \\ 1 \\ 1 \end{bmatrix}$$

$$r = \mathbf{\Pi X}_e - 1 = [0.5 \quad 0.5 \quad 0.5 \quad 0.5] \begin{bmatrix} 0 \\ 0 \\ 1 \\ 1 \end{bmatrix} - 1 = 0 \leq TolCG$$

Algorithm is terminated, and outputs are:

$$\mathbf{B} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 1 \\ 0 & 1 & 1 & 0 \\ 1 & 0 & 1 & 1 \end{bmatrix}, \mathbf{X}_B = \begin{bmatrix} 1 \\ 1 \\ 0 \\ 0 \end{bmatrix}, \text{OPTCG} = 2.$$

3.3 Computational results

In this section we depict the results of warm starting the column generation versus starting by singleton bins in figures (3.1) to (3.5). MATLAB 2015b was used as the main hub of programming for the implementation purposes. Also, to solve the one-dimensional knapsack problem, the dynamic programming scheme of Martello et *al.*, presented in [38], was employed. The C program code of this method is available online [49]. The C program code was downloaded and modified to be called from MATLAB 2015b.

The x-axis in figures (3.1) to (3.5) denotes each instance of a particular class and the y-axis shows total number of the columns generated during column generation procedure. Experiments to investigate effects of warm starting column generation were performed on u120, u250, t60, t120, and Hard28 class of benchmark instances.

As observed from the figures (3.1) to (3.5), FFD-based BFS with refinement algorithm reduces number of columns generated for the majority of the instances. However, this algorithm does not have a stable behavior, and for some instances starting from singleton bins results in a less number of columns generated. What accounts for this behavior of FFD-based BFS with refinement algorithm, is the fact that first fit decreasing algorithm fails to provide a strong upper bound for BPP for instances of triplets. Therefore, the starting point produced by FFD-based BFS with refinement algorithm might compel column generation to start off searching from a more distant point from the optimal one compared to the starting point provided by singleton bins.

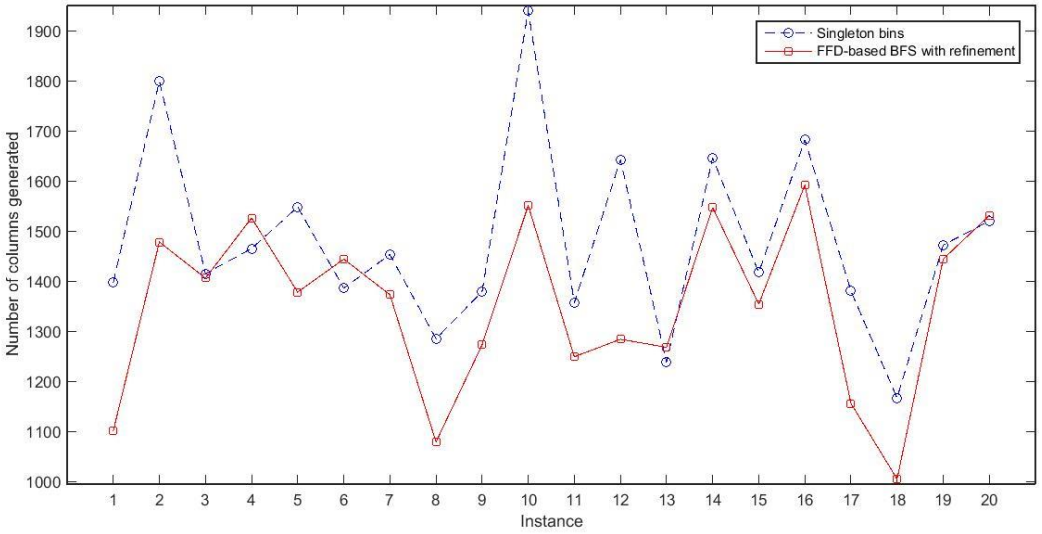


Figure 3.1 Comparison of number of columns generated during column generation procedure for class u120.

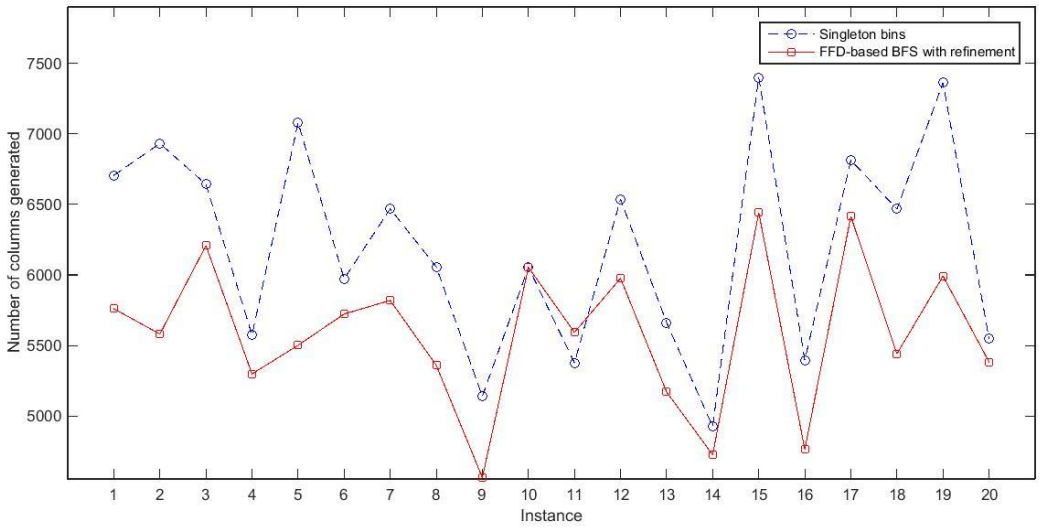


Figure 3.2 Comparison of number of columns generated during column generation procedure for class u250.

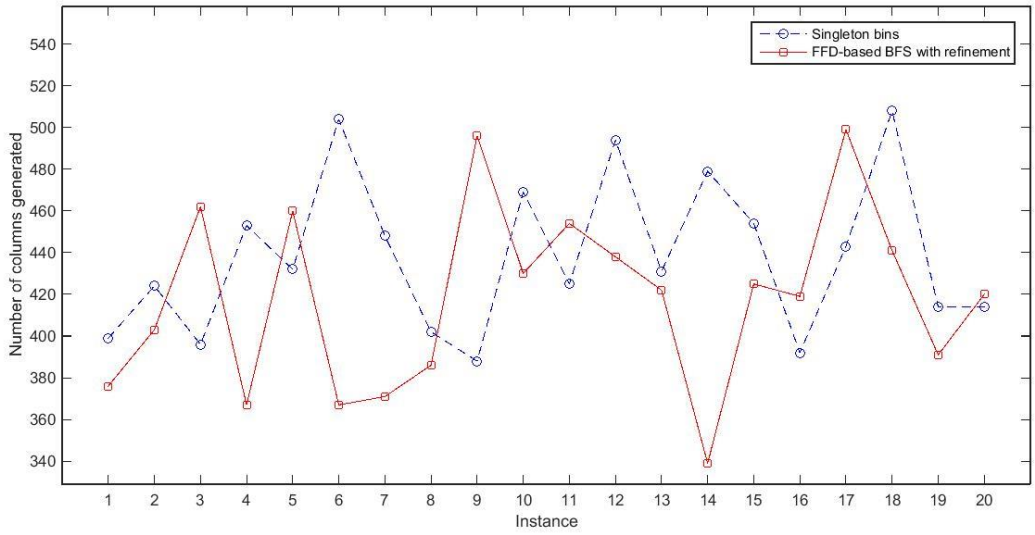


Figure 3.3 Comparison of number of columns generated during column generation procedure for class t60.

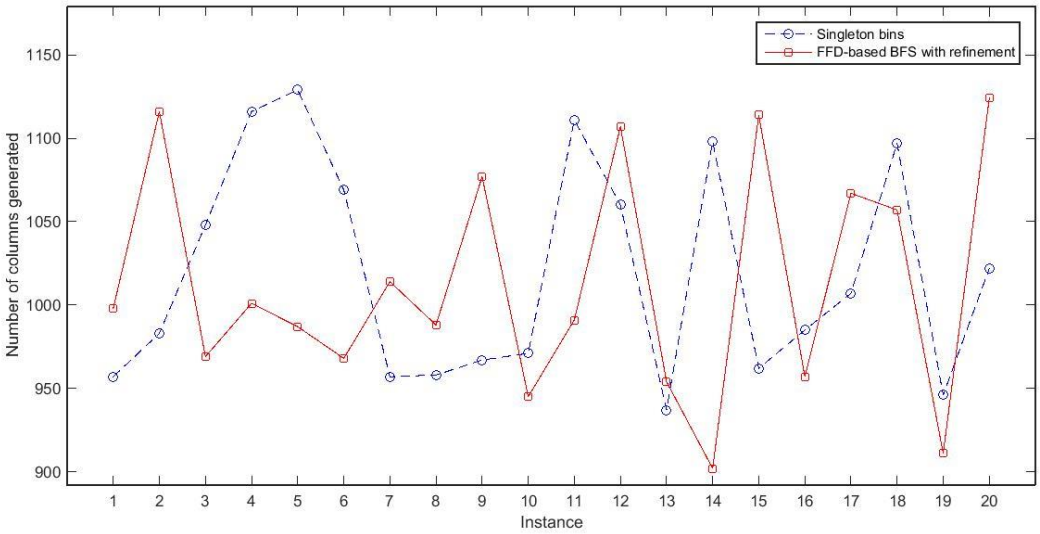


Figure 3.4 Comparison of number of columns generated during column generation procedure for class t120.

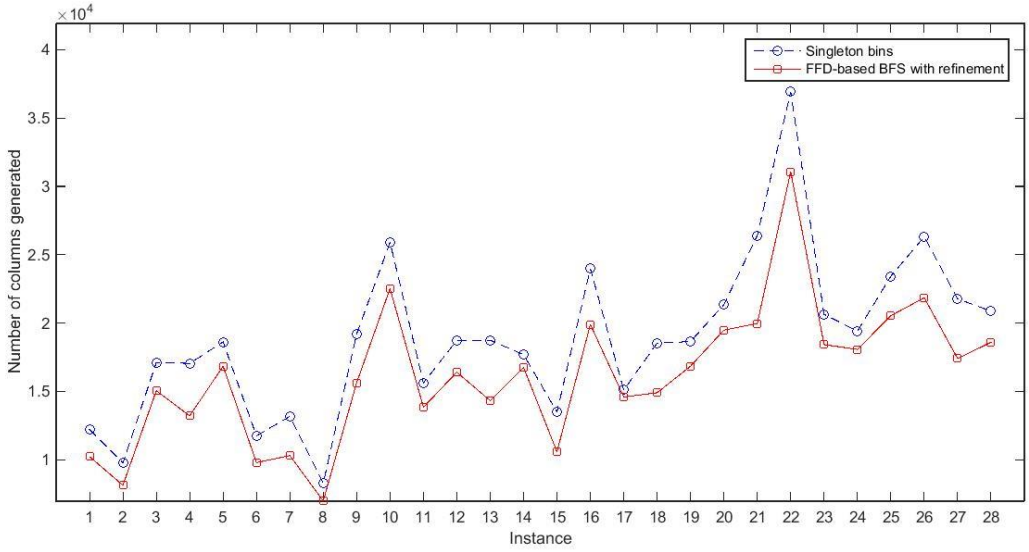


Figure 3.5 Comparison of number of columns generated during column generation procedure for class Hard28.

Chapter 4

An upper bound for BPP using solutions of the continuous relaxation of set-covering formulation of BPP (SCCUB)

4.1. SCCUB procedure

In this chapter we introduce a new upper bounding procedure for BPP which uses solutions of the continuous relaxation of set-covering formulation presented in model (2.7). This bound is called SCCUB (abbreviation for *Set-Covering Continuous Upper Bound*).

The rationale for the development of SCCUB is that LP solutions of BPP are near-optimal (optimal), and it is possible to derive a high quality bound by merely using these solutions. In fact, the notion that column generation procedure has decided to choose these patterns as the basic solutions of LP out of the exponential number of the patterns in the search space, asserts the near optimality of these solutions.

The simple, yet powerful SCCUB procedure is developed based on the following few premises:

Premise 1: The process of solving LP relaxation of BPP by column generation is a time-consuming process, and not only should the LP bound be used in further procedures like branch-and-price method, but also the LP solutions should be utilized to construct an upper bound for BPP to make the benefits of performing the column generation twofold: 1- Finding a strong lower bound for BPP. 2- Deriving a strong upper bound for BPP in a short amount of time.

Premise 2: Any procedure involving column generation to establish the upper bound for BPP by using LP solutions should reduce the dimension of the problem significantly to ensure that the upper bound will be derived shortly after the lower bound is obtained.

Once the basic patterns are available following the termination of the column generation, the procedure of constructing the upper bound using these patterns will be started. To articulate the premises, however, one should first investigate the structure of the LP patterns. The following definition will be the starting point of our investigation.

Definition 4.1 Counterpart patterns

The pattern(s) of basis complementing the value of a certain basic pattern λ_k, is (are) called counterpart(s) of λ_k.

At the beginning of the upper bounding procedure, all basic patterns are placed in a candidate pool of upper bound solutions. Motivated by reduction of the dimension of the problem significantly, one might attempt to include all the members of the candidate pool into the upper bound solution. However, Definition (4.1) implies that inclusion of a certain pattern in the upper bound solution means exclusion of its counterpart(s) from the candidate pool. Hence, not all of the candidate patterns could be used in the upper bound solution and a criterion needs to be formulated to guide selection of the most desired basic patterns. There might be different criteria for selecting some of the basic patterns and locating them into the upper bound solution. The filling factor of the basic patterns, dominance relations among the basic patterns, values that basic patterns take on, and number of the counterparts for each candidate pattern are some of the possible criteria, to name a few. In this research,

we are motivated to know how effective the criterion that involves considering the values of the basic patterns is.

We mentioned before that solutions of LP relaxation are valuable from the point of view that column generation process agrees on their presence in the basis. Indeed, what distinguishes column generation from other lower bounding techniques for BPP is its systematic approach towards solving LP relaxation of the problem. Therefore, the values column generation assigns to the basic patterns are as important as its capability to choose basic patterns among exponential number of the feasible loading patterns. Roughly speaking, the higher the value of a basic pattern is, the more importance has column generation put on this pattern. In summary, values of the basic solutions can provide a good criterion to include basic patterns in the upper bound solution.

Proposition 4.1 Selecting combination of the patterns having values greater than 0.5 has the minimum operational cost for selection of the combination of the patterns from candidate pool.

Proof Summation of the value of a specific basic pattern with the value(s) of its counterpart(s) is always 1. Thus, it is impossible for a pattern of value greater than 0.5 to have a counterpart of value greater than 0.5, and we can fix any combination of the basic patterns of value greater than 0.5 as the partial upper bound solutions without paying any operational cost for removing their counterparts from the candidate pool.

Since our goal is to reduce the dimension of the problem as much as possible, the selection scheme could be described as selecting all the patterns having values greater than 0.5. We would like to point out that even though this selection scheme takes the least computational effort, selecting all the patterns having values greater than 0.5 as the partial solutions of the upper bound might not lead into the maximal reduction in the dimension of the problem. Put differently, there might be a combination of patterns from patterns of values greater than 0.5 and patterns of values not greater than 0.5 that leads into the maximum elimination of the items of the problem. Nevertheless, our computational experiments reveal that on average %76 of the dimension of the problem is reduced by following this scheme which is remarkable in a sense that the residual problem can be solved cost-effectively thereafter.

In short, SCCUB procedure can be described as follows. Once the relaxation of the set-covering formulation of BPP, presented in (2.7) is solved, all the basic patterns having values greater than 0.5 are fixed as the partial upper bound solutions. Then, the residual problem will again be solved by column generation and the same selection scheme will be applied to fix more patterns of the upper bound solution, and this process will be continued until all the items are fixed. SCCUB procedure is presented in Table (4.1). Also, it is noted that diving, where one always chooses the basic pattern with the highest value in order to get an upper bound for the problem, could be seen as a special case of SCCUB procedure.

Table 4.1 Pseudo-code for procedure SCCUB

Procedure SCCUB

Input: number of items (n), weights of the items ($\mathbf{W}_{1\times n}$), bin capacity (c)

Output: Upper bound solution (SCCUB) and optimal number of patterns (OPT(SCCUB))

Step1. Define SCCUB.

Step2. **While** true

Solve continuous relaxation of set-covering formulation of BPP by **algorithm** revised simplex. Return **B** and $\mathbf{X}_B$.

Find basic patterns having values greater than 0.5.

If no basic patterns is found with value greater than 0.5

Add the pattern with the largest value to SCCUB.

Else

Add all the patterns having values greater than 0.5 to SCCUB.

Eliminate items of SCCUB from **W** and update n.

If n=0

Break

End of **while** loop

Step 3. Outputs are SCCUB and OPT(SCCUB).

4.2 Computational results

In this section, we solve LP relaxation of instances of u120, u250, u500, t60, t120, t249, t501, and Hard28, and we run SCCUB procedure presented in table (4.1) for each instance to derive the upper bound. The results are displayed in tables (4.3) to (4.10), and definition of the symbols used in this section and the following ones could be found in Table (4.2). Also, figures (4.1) to (4.8) represent a comparison of the quality of the upper bounds derived by using SCCUB procedure to those derived by using FF and FF-n algorithms. Furthermore, the known IPs for the instances are depicted in figures (4.1) to (4.8) in order to show how close SCCUB can approximate the known IP. Note should be made that for triplets, the outcome of FF-n algorithm does not differ from that of FF algorithm since in triplets, all the items have weights less than half of the bin capacity. Furthermore, the results shown for FF and FF-n are averaged over 100 replications.

Table 4.2 Definition of the symbols

Symbol	Definition
IP	Integer programming value
LP	Linear programming relaxation value
col_0	Number of the columns generated columns at the root node, initializing **BFS** by singleton bins
col_{0W}	Number of the columns generated at the root node, initializing **BFS** by FFD-based BFS with refinement algorithm (warm start)
nod	Average number of the nodes explored
red_0	Average number of the columns reduced at the root node
$red_{0\%}$	Average percentage of the columns reduced at root node $(\frac{red_0}{SCCUB})$
t_0	Average computational time at root node (seconds)
$t_{overall}$	Average total computational time of the procedure (seconds)
$t_{relative}$	$\frac{t_{overall}}{t_0}$

As observed from the tables (4.3) to (4.10), SCCUB procedure provides a good upper bound for BPP. For all the uniform instances, SCCUB procedure is capable of finding the optimal solution, and for the rest of the instances it provides an upper bound with an only one extra bin from the optimal solution in case its value is not equal to the optimal solution.

Relative time (t_{relative}) introduced in the tables (4.3) to (4.10), measures the computational justifiability of SCCUB procedure. More precisely, this parameter shows how much extra time should be spent to find the SCCUB upper bound after the column generation procedure is terminated at the root node. On average, SCCUB procedure runs in %7 of the computational time of the column generation procedure at the root node.

Another parameter represented in these tables is the reduction in the dimension of the problem by using SCCUB procedure ($\text{red}_{0\%}$). It is worth to note that this criterion does not measure reduction in the dimension of the problem item-wise; rather it only provides a measure of the percentage of the patterns added to the upper bound solution at the first iteration of SCCUB procedure. Nevertheless, on average, %76 of the patterns of the upper bound solutions are obtained right at the first iteration of the SCCUB procedure.

Another important criterion which might bring insight into the performance of the SCCUB procedure is number of the nodes explored during SCCUB procedure which is on average 4.9 for the instances of these three classes.

Furthermore, as depicted in figures (4.1) to (4.8), SCCUB upper bounds dominate the ones obtained by applying FF algorithm, for all the instances of u120, u250, u500, t60, t120, t249, t501, and Hard28 classes. Also, SCCUB upper bounds dominate the FF-n ones for all the classes except instances of the class Hard28 where both methods manifest the same behavior.

All in all, SCCUB procedure has a stable behavior, and it can be used as an upper bounding technique for BPP to impose a strong upper bound on the problem.

Table 4.3 Computational results of SCCUB for instances of U120

Instance	IP	LP	col_0	col_{0W}	nod	red_0	$red_{0\%}$	t_0	$t_{overall}$	$t_{relative}$	SCCUB	FF	FF-n
U120_00	48	47.2660	1398	1102	5	37	77	0.695	0.785	1.129	48	51.29	50.57
U120_01	49	48.0486	1800	1479	5	42	86	0.918	0.988	1.075	49	51.67	50.93
U120_02	46	45.2933	1416	1407	3	37	80	0.862	0.934	1.083	46	48.77	48.16
U120_03	49	48.6260	1466	1527	4	43	88	1.006	1.058	1.051	49	52.67	52.04
U120_04	50	49.0850	1549	1379	6	40	80	0.829	0.918	1.108	50	52.98	52.16
U120_05	48	47.4898	1387	1445	3	38	80	0.860	0.911	1.059	48	51.34	50.16
U120_06	48	47.5800	1454	1374	5	40	83	0.845	0.918	1.085	48	51.32	50.91
U120_07	49	48.6599	1286	1080	4	38	78	0.680	0.746	1.096	49	52.66	51.97
U120_08	50	49.9116	1380	1274	4	43	86	0.745	0.790	1.060	50	53.85	53.49
U120_09	46	45.8000	1941	1552	4	39	85	0.946	0.990	1.045	46	49.23	49.01
U120_10	52	51.2824	1358	1250	6	41	79	0.743	0.865	1.164	52	55.72	54.50
U120_11	49	48.3929	1643	1285	4	43	88	0.766	0.832	1.085	49	52.31	51.21
U120_12	48	47.8667	1239	1269	4	38	80	0.763	0.841	1.101	48	51.61	51.00
U120_13	49	48.0133	1647	1548	4	36	73	0.939	1.030	1.096	49	51.85	50.91
U120_14	50	49.1701	1418	1355	4	40	80	0.854	0.937	1.096	50	53.03	52.24
U120_15	48	47.3841	1683	1593	5	41	85	0.960	1.013	1.055	48	51.19	50.26
U120_16	52	51.3333	1381	1157	4	47	90	0.705	0.747	1.059	52	55.80	54.59
U120_17	52	51.5000	1168	1006	3	46	88	0.708	0.739	1.044	52	56.01	54.60
U120_18	49	48.3815	1473	1445	5	41	84	0.890	0.934	1.049	49	52.26	51.02
U120_19	49	48.8639	1521	1532	3	43	88	0.918	0.959	1.043	49	52.77	51.86
Minimum	46	45.2933	1168	1006	3	36	73	0.680	0.739	1.043	46	48.77	48.16
Average	49.05	48.4974	1480	1352	4.2	40.6	82.9	0.831	0.896	1.079	49.05	52.41	51.57
Maximum	52	51.5000	1941	1593	6	47	90	1.006	1.058	1.164	52	56.01	54.60

Table 4.4 Computational results of SCCUB for instances of U250

Instance	IP	LP	col_0	col_{0W}	nod	red_0	$red_{0\%}$	t_0	$t_{overall}$	$t_{relative}$	SCCUB	FF	FF-n
U250_00	99	98.5533	6706	5760	6	78	79	9.335	9.504	1.018	99	105.02	103.47
U250_01	100	99.0267	6932	5581	5	85	85	9.159	9.266	1.011	100	105.70	104.08
U250_02	102	101.4218	6643	6208	5	82	80	10.262	10.485	1.021	102	108.17	106.35
U250_03	100	99.4267	5574	5298	6	78	78	8.928	9.099	1.019	100	105.77	104.72
U250_04	101	100.6133	7081	5501	4	82	81	9.267	9.380	1.012	101	107.19	105.99
U250_05	101	100.8267	5973	5723	3	80	79	9.627	9.766	1.014	101	107.61	106.39
U250_06	102	101.0267	6469	5819	6	87	85	9.438	9.579	1.014	102	107.55	106.05
U250_07	103	102.8852	6057	5359	3	86	84	8.887	9.033	1.016	103	110.01	108.53
U250_08	105	104.9184	5139	4563	3	87	83	7.740	7.842	1.013	105	112.34	110.42
U250_09	101	100.2014	6055	6054	4	85	84	10.333	10.471	1.013	101	107.02	105.94
U250_10	105	104.3946	5377	5593	5	87	83	9.562	9.682	1.012	105	111.44	110.15
U250_11	101	100.7133	6537	5976	4	84	83	9.970	10.069	1.010	101	107.62	106.38
U250_12	105	104.9772	5662	5170	3	91	87	8.614	8.691	1.008	105	112.23	111.22
U250_13	103	102.0407	4924	4724	6	86	84	7.469	7.600	1.017	103	109.15	107.01
U250_14	100	99.1667	7401	6443	5	81	81	10.384	10.543	1.015	100	105.6	104.47
U250_15	105	104.8611	5396	4763	3	91	87	8.146	8.212	1.008	105	112.1	110.08
U250_16	97	96.5133	6814	6416	5	72	74	11.257	11.468	1.018	97	102.83	102
U250_17	100	99.1667	6468	5440	6	82	82	9.196	9.365	1.018	100	105.59	104.25
U250_18	100	99.7000	7363	5992	4	83	83	10.030	10.126	1.009	100	105.98	104.91
U250_19	102	101.3600	5547	5383	6	84	82	8.858	8.975	1.013	102	108.21	107.4
Minimum	97	96.5133	4924	4563	3	72	74	7.469	7.600	1.008	97	102.83	102
Average	101.6	101.0897	6206	5588	4.6	83.5	82.2	9.323	9.458	1.013	101.6	107.85	106.49
Maximum	105	104.9772	7401	6443	6	91	87	11.257	11.468	1.021	105	112.34	111.22

Table 4.5 Computational results of SCCUB for instances of U500

Instance	IP	LP	col_0	col_{0W}	nod	red_0	$red_{0\%}$	t_0	$t_{overall}$	$t_{relative}$	SCCUB	FF	FF-n
U500_00	198	197.5800	-	28231	5	160	81	210.056	210.596	1.002	198	209.38	207.15
U500_01	201	200.8467	-	24837	5	167	83	200.846	208.834	1.039	201	212.20	209.99
U500_02	202	201.4400	-	25785	7	159	79	196.818	197.652	1.004	202	213.20	210.36
U500_03	204	203.8133	-	28321	5	175	86	210.582	210.911	1.001	204	215.68	213.46
U500_04	206	205.1133	-	24228	5	172	84	175.703	176.158	1.002	206	217.32	214.41
U500_05	206	205.0867	-	26012	6	173	84	203.465	203.907	1.002	206	217.45	215.11
U500_06	207	206.9058	-	21242	4	184	89	149.977	150.192	1.001	207	219.89	216.37
U500_07	204	203.9800	-	27470	4	177	87	217.750	218.033	1.001	204	215.83	213.88
U500_08	196	195.6800	-	27734	7	158	81	201.133	201.654	1.002	196	206.89	204.96
U500_09	202	201.0600	-	25437	6	170	84	201.060	215.984	1.074	202	212.76	210.99
U500_10	200	199.0667	-	27786	6	157	79	200.400	201.018	1.003	200	210.74	207.33
U500_11	200	199.4267	-	27806	6	161	81	223.483	224.369	1.004	200	211.19	209
U500_12	199	198.6200	-	28893	7	164	82	228.878	229.263	1.001	199	209.73	207.80
U500_13	196	195.5867	-	29987	6	153	78	223.679	224.288	1.002	196	206.41	204.86
U500_14	204	203.0267	-	29498	6	174	85	239.334	239.788	1.001	204	214.42	212.49
U500_15	201	200.1333	-	25527	6	159	79	242.047	242.694	1.002	201	211.81	209.97
U500_16	202	201.0067	-	25860	8	152	75	231.250	232.780	1.006	202	212.23	210.05
U500_17	198	197.4267	-	31398	6	158	80	304.924	305.825	1.003	198	208.62	207
U500_18	202	201.2933	-	26035	5	173	86	211.808	212.095	1.001	202	213.12	211.10
U500_19	196	195.6333	-	29348	7	164	84	283.622	284.212	1.002	196	207.03	205.30
Minimum	196	195.5867	-	21242	4	152	75	149.977	150.192	1.001	196	206.41	204.86
Average	201.2	200.6363	-	27071	5.8	165.5	82.4	217.840	219.512	1.007	201.2	212.29	210.07
Maximum	207	206.9058	-	31398	8	184	89	304.924	305.825	1.074	207	219.89	216.37

Table 4.6 Computational results of SCCUB for instances of t60

Instance	IP	LP	col_0	col_{0W}	nod	red_0	$red_{0\%}$	t_0	$t_{overall}$	$t_{relative}$	SCCUB	FF	FF-n
t60_00	20	20.0000	399	376	1	20	100	0.199	0.215	1.080	20	22.72	-
t60_01	20	20.0000	424	403	3	5	25	0.205	0.348	1.699	20	22.75	-
t60_02	20	20.0000	396	462	3	16	80	0.241	0.276	1.147	20	22.97	-
t60_03	20	20.0000	453	367	4	7	33	0.201	0.294	1.461	**21**	22.73	-
t60_04	20	20.0000	432	460	4	18	86	0.226	0.282	1.247	**21**	22.74	-
t60_05	20	20.0000	504	367	1	20	100	0.202	0.218	1.079	20	22.78	-
t60_06	20	20.0000	448	371	5	16	76	0.204	0.266	1.303	**21**	22.68	-
t60_07	20	20.0000	402	386	5	15	71	0.208	0.270	1.297	**21**	22.72	-
t60_08	20	20.0000	388	496	5	8	40	0.251	0.381	1.520	20	22.62	-
t60_09	20	20.0000	469	430	1	20	100	0.227	0.243	1.072	20	22.69	-
t60_10	20	20.0000	425	454	5	14	67	0.227	0.296	1.305	**21**	22.77	-
t60_11	20	20.0000	494	438	1	20	100	0.221	0.237	1.073	20	22.85	-
t60_12	20	20.0000	431	422	5	11	52	0.218	0.297	1.362	21	22.76	-
t60_13	20	20.0000	479	339	3	7	35	0.197	0.307	1.554	20	22.84	-
t60_14	20	20.0000	454	425	1	20	100	0.224	0.241	1.077	20	22.84	-
t60_15	20	20.0000	392	419	2	14	70	0.212	0.252	1.187	20	22.72	-
t60_16	20	20.0000	443	499	1	20	100	0.252	0.268	1.061	20	22.95	-
t60_17	20	20.0000	508	441	1	20	100	0.219	0.236	1.077	20	22.86	-
t60_18	20	20.0000	414	391	1	20	100	0.204	0.221	1.081	20	22.78	-
t60_19	20	20.0000	414	420	1	20	100	0.218	0.234	1.072	20	22.84	-
Minimum	20	20.0000	388	339	1	5	25	0.197	0.215	1.061	20	22.62	-
Average	20	20.0000	438	418	2.6	15.5	76.8	0.217	0.269	1.237	20.3	22.78	-
Maximum	20	20.0000	508	499	5	20	100	0.252	0.381	1.699	21	22.97	-

Table 4.7 Computational results of SCCUB for instances of t120

Instance	IP	LP	col_0	col_{0W}	nod	red_0	$red_{0\%}$	t_0	$t_{overall}$	$t_{relative}$	SCCUB	FF	FF-n
t120_00	40	40.0000	957	998	5	29	71	0.631	0.718	1.138	**41**	45.06	-
t120_01	40	40.0000	983	1116	4	32	78	0.692	0.768	1.110	**41**	44.88	-
t120_02	40	40.0000	1048	969	5	28	68	0.629	0.714	1.135	**41**	45.12	-
t120_03	40	40.0000	1116	1001	5	23	56	0.638	0.780	1.221	**41**	45.07	-
t120_04	40	40.0000	1129	987	5	25	61	0.618	0.746	1.206	**41**	45.39	-
t120_05	40	40.0000	1069	968	5	29	71	0.619	0.705	1.138	**41**	45.31	-
t120_06	40	40.0000	957	1014	4	26	63	0.675	0.784	1.162	**41**	44.98	-
t120_07	40	40.0000	958	988	6	30	73	0.723	0.813	1.125	**41**	45.15	-
t120_08	40	40.0000	967	1077	6	27	66	0.727	0.834	1.145	**41**	45	-
t120_09	40	40.0000	971	945	6	25	61	0.607	0.731	1.203	**41**	45.07	-
t120_10	40	40.0000	1111	991	6	27	66	0.639	0.734	1.149	**41**	45.05	-
t120_11	40	40.0000	1060	1107	5	28	68	0.734	0.821	1.119	**41**	45.26	-
t120_12	40	40.0000	937	954	5	29	71	0.620	0.711	1.146	**41**	45.28	-
t120_13	40	40.0000	1098	902	5	28	68	0.612	0.707	1.155	**41**	45.04	-
t120_14	40	40.0000	962	1114	6	32	78	0.722	0.800	1.107	**41**	45.20	-
t120_15	40	40.0000	985	957	5	25	61	0.649	0.758	1.168	**41**	45.06	-
t120_16	40	40.0000	1007	1067	5	31	76	0.703	0.780	1.108	**41**	45.08	-
t120_17	40	40.0000	1097	1057	5	30	73	0.664	0.745	1.121	**41**	45.30	-
t120_18	40	40.0000	946	911	5	23	56	0.602	0.732	1.216	**41**	45.29	-
t120_19	40	40.0000	1022	1124	5	32	78	0.734	0.814	1.108	**41**	45.26	-
Minimum	40	40.0000	937	902	4	23	56	0.602	0.705	1.107	41	44.88	-
Average	40	40.0000	1019	1012	5.1	27.9	68.1	0.661	0.759	1.149	41	45.1425	-
Maximum	40	40.0000	1129	1124	6	32	78	0.734	0.834	1.221	41	45.39	-

Table 4.8 Computational results of SCCUB for instances of t249

Instance	IP	LP	col_0	col_{0W}	nod	red_0	$red_{0\%}$	t_0	$t_{overall}$	$t_{relative}$	SCCUB	FF	FF-n
t249_00	83	83.0000	-	2567	5	60	71	4.690	4.900	1.044	**84**	93.54	-
t249_01	83	83.0000	-	2810	5	63	75	4.583	4.777	1.042	**84**	93.16	-
t249_02	83	83.0000	-	2639	7	59	70	4.574	4.804	1.050	**84**	92.74	-
t249_03	83	83.0000	-	2666	6	58	69	4.558	4.806	1.054	**84**	93.28	-
t249_04	83	83.0000	-	2702	5	57	68	4.987	5.260	1.054	**84**	93.36	-
t249_05	83	83.0000	-	2753	6	55	65	4.585	4.880	1.064	**84**	93.05	-
t249_06	83	83.0000	-	2872	6	58	69	5.424	5.703	1.051	**84**	93.23	-
t249_07	83	83.0000	-	2701	6	49	58	4.873	5.381	1.104	**84**	93.30	-
t249_08	83	83.0000	-	2681	5	54	64	5.027	5.334	1.060	**84**	93.30	-
t249_09	83	83.0000	-	2653	5	57	68	4.654	4.943	1.062	**84**	93.11	-
t249_10	83	83.0000	-	2461	6	56	67	4.420	4.746	1.073	**84**	93.55	-
t249_11	83	83.0000	-	2999	6	56	67	5.167	5.470	1.058	**84**	93.19	-
t249_12	83	83.0000	-	2740	6	57	68	5.065	5.358	1.057	**84**	92.64	-
t249_13	83	83.0000	-	2838	6	57	68	5.265	5.551	1.054	**84**	93.44	-
t249_14	83	83.0000	-	2918	6	56	67	5.015	5.339	1.064	**84**	92.71	-
t249_15	83	83.0000	-	2863	5	56	67	5.198	5.550	1.067	**84**	93.11	-
t249_16	83	83.0000	-	2648	6	56	67	4.868	5.252	1.078	**84**	93.11	-
t249_17	83	83.0000	-	2503	6	58	69	4.628	4.905	1.059	**84**	92.77	-
t249_18	83	83.0000	-	2611	5	57	68	4.801	5.076	1.057	**84**	93.21	-
t249_19	83	83.0000	-	2821	7	58	69	4.842	5.117	1.056	**84**	93.51	-
Minimum	83	83.0000	-	2461	5	49	58	4.42	4.746	1.042	84	92.64	-
Average	83	83.0000	-	2722	5.7	56.8	67.7	4.861	5.157	1.060	84	93.16	-
Maximum	83	83.0000	-	2999	7	63	75	5.424	5.703	1.104	84	93.55	-

Table 4.9 Computational results of SCCUB for instances of t501

Instance	IP	LP	col_0	col_{0W}	nod	red_0	$red_{0\%}$	t_0	$t_{overall}$	$t_{relative}$	SCCUB	FF	FF-n
t501_00	167	167.0000	-	7727	7	102	61	66.437	69.126	1.040	**168**	187.41	-
t501_01	167	167.0000	-	8002	6	109	65	72.544	74.507	1.027	**168**	187.40	-
t501_02	167	167.0000	-	8057	8	111	66	70.569	72.484	1.027	**168**	186.32	-
t501_03	167	167.0000	-	7790	5	113	67	76.333	78.034	1.022	**168**	186.40	-
t501_04	167	167.0000	-	7336	7	107	64	64.056	66.019	1.030	**168**	186.70	-
t501_05	167	167.0000	-	7725	5	106	63	66.357	68.324	1.029	**168**	187.11	-
t501_06	167	167.0000	-	7543	6	111	66	62.456	64.085	1.026	**168**	186.58	-
t501_07	167	167.0000	-	6955	6	109	65	63.195	65.338	1.033	**168**	186.62	-
t501_08	167	167.0000	-	7339	6	106	63	65.143	67.660	1.038	**168**	186.45	-
t501_09	167	167.0000	-	7902	6	107	64	61.868	63.783	1.030	**168**	186.84	-
t501_10	167	167.0000	-	7731	6	111	66	61.457	63.041	1.025	**168**	186.86	-
t501_11	167	167.0000	-	8127	5	102	61	63.171	65.124	1.030	**168**	186.34	-
t501_12	167	167.0000	-	7362	6	118	70	56.929	58.090	1.020	**168**	187.12	-
t501_13	167	167.0000	-	8059	6	112	67	64.075	65.424	1.021	**168**	186.74	-
t501_14	167	167.0000	-	7656	7	109	65	63.349	64.982	1.025	**168**	187.02	-
t501_15	167	167.0000	-	7493	7	104	62	56.923	58.949	1.035	**168**	186.64	-
t501_16	167	167.0000	-	7673	6	107	64	59.282	61.070	1.030	**168**	186.81	-
t501_17	167	167.0000	-	7328	6	108	64	56.973	58.785	1.031	**168**	186.88	-
t501_18	167	167.0000	-	7857	7	105	63	61.315	63.418	1.034	**168**	186.89	-
t501_19	167	167.0000	-	7923	6	111	66	61.907	63.480	1.025	**168**	187.61	-
Minimum	167	167.0000	-	6955	5	102	61	56.923	58.090	1.02	168	186.32	-
Average	167	167.0000	-	7679	6.2	108.4	64.6	63.716	65.586	1.02	168	186.83	-
Maximum	167	167.0000	-	8127	8	118	70	76.333	78.034	1.04	168	187.61	-

Table 4.10 Computational results of SCCUB for instances of Hard28

Instance	IP	LP	col_0	col_{0W}	nod	red_0	$red_{0\%}$	t_0	$t_{overall}$	$t_{relative}$	SCCUB	FF	FF-n
BPP14	62	60.9980	12232	10232	4	50	81	7.882	7.990	1.013	62	65.86	62
BPP832	60	59.9975	9766	8141	4	49	80	6.535	6.604	1.010	**61**	63.75	61
BPP40	59	**59.0091**	17126	15058	6	43	72	11.602	11.806	1.017	**60**	63.01	60
BPP360	62	62.0000	17051	13209	5	54	87	9.629	9.726	1.010	62	67.40	63.12
BPP645	58	57.9990	18621	16853	5	47	80	13.005	13.106	1.007	**59**	61.70	59
BPP742	64	64.0000	11753	9793	4	59	91	7.314	7.365	1.007	**65**	69.070	65
BPP766	62	61.9990	13178	10310	4	55	87	8.133	8.201	1.008	**63**	66.24	63
BPP60	63	62.9979	8279	6982	5	57	89	5.754	5.832	1.013	**64**	67.44	64.01
BPP13	67	66.9997	19195	15637	5	51	75	13.962	14.173	1.015	**68**	72.38	68
BPP195	64	63.9960	25907	22528	5	47	72	21.037	21.296	1.012	**65**	67.74	65
BPP709	67	67.0000	15620	13845	4	53	78	12.803	12.953	1.011	**68**	71	68
BPP785	68	67.9943	18733	16410	5	56	81	15.155	15.289	1.008	**69**	72.80	69
BPP47	71	71.0000	18745	14327	5	62	86	12.219	12.317	1.008	**72**	76.87	72.02
BPP181	72	71.9985	17697	16756	4	61	84	15.119	15.218	1.006	**73**	77.14	73
BPP359	76	74.9983	13497	10582	5	63	83	9.753	9.841	1.009	76	80.74	76
BPP485	71	70.9973	23993	19901	4	65	90	17.491	17.554	1.003	**72**	76.28	72
BPP640	74	74.0000	15101	14594	5	63	84	12.376	12.475	1.008	**75**	80.37	75.01
BPP716	76	75.0000	18542	14907	5	62	82	12.608	12.714	1.008	76	81.77	76.06
BPP119	77	76.0000	18649	16849	5	55	71	18.902	19.153	1.013	77	80.87	77.37
BPP144	73	73.0000	21362	19489	7	53	72	21.817	22.151	1.015	**74**	77.43	74
BPP561	72	71.9960	26410	19972	5	55	75	21.996	22.205	1.009	**73**	75.81	73.27
BPP781	71	70.9990	36921	31074	7	59	82	35.770	35.936	1.004	**72**	75.97	72
BPP900	75	74.9961	20643	18426	5	59	78	19.607	19.775	1.008	**76**	80.26	76.10

Table 4.10 continued

Instance	IP	LP	col_0	col_{0W}	nod	red_0	$red_{0\%}$	t_0	$t_{overall}$	$t_{relative}$	SCCUB	FF	FF-n
BPP175	84	83.0000	19415	18074	5	74	88	19.224	19.317	1.004	84	90.26	84.13
BPP178	80	79.9953	23428	20543	4	73	90	22.412	22.465	1.002	**81**	85.74	81
BPP419	80	79.999	26310	21851	4	66	81	22.618	22.770	1.006	**81**	85.86	81
BPP531	83	83.0000	21768	17403	5	73	87	16.432	16.521	1.005	**84**	90.25	84.38
BPP814	81	81.0000	20879	18602	5	70	85	18.645	18.775	1.006	**82**	87.28	82.21
Minimum	58	57.9990	8279	6982	4	43	71	5.754	5.832	1.003	59	61.7	59
Average	67.3	67.0992	16752	14320	4.8	55.2	81.2	12.654	12.788	1.010	68.05	71.99	68.13
Maximum	77	76.0000	25907	22528	7	65	91	21.817	22.151	1.017	77	81.77	77.37

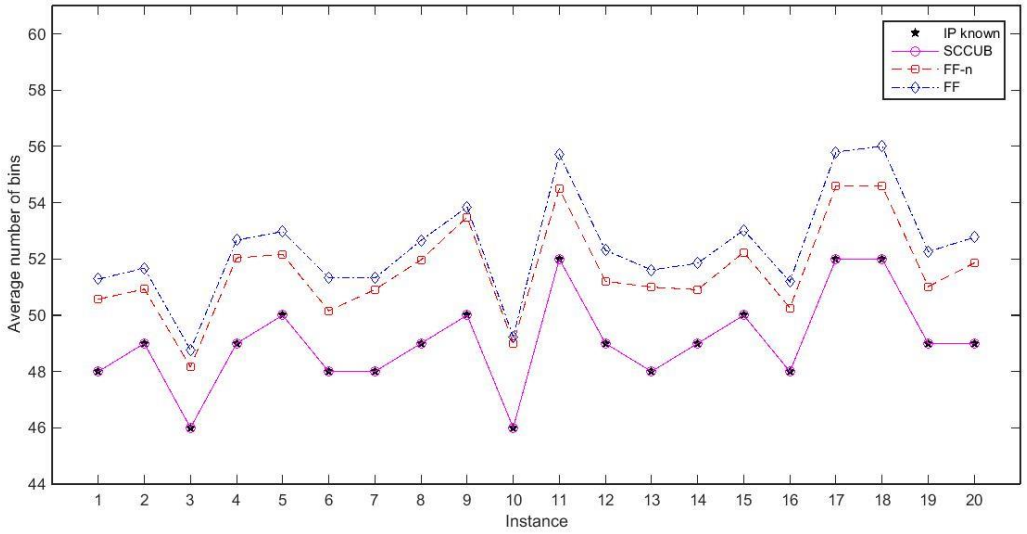


Figure 4.1 Comparison of the upper bounds generated by SCCUB, FF, and FF-n with known IPs for instances of class u120.

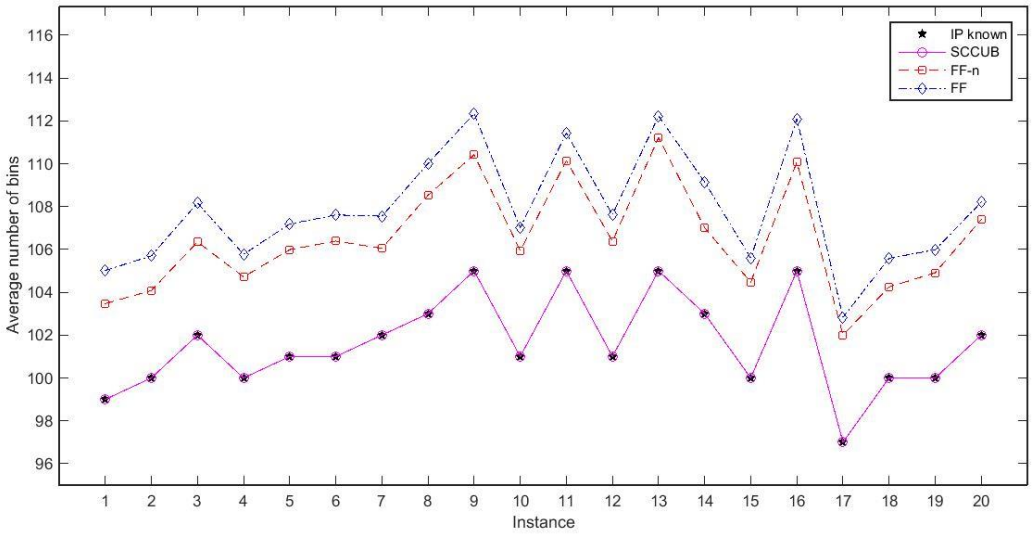


Figure 4.2 Comparison of the upper bounds generated by SCCUB, FF, and FF-n with known IPs for instances of class u250.

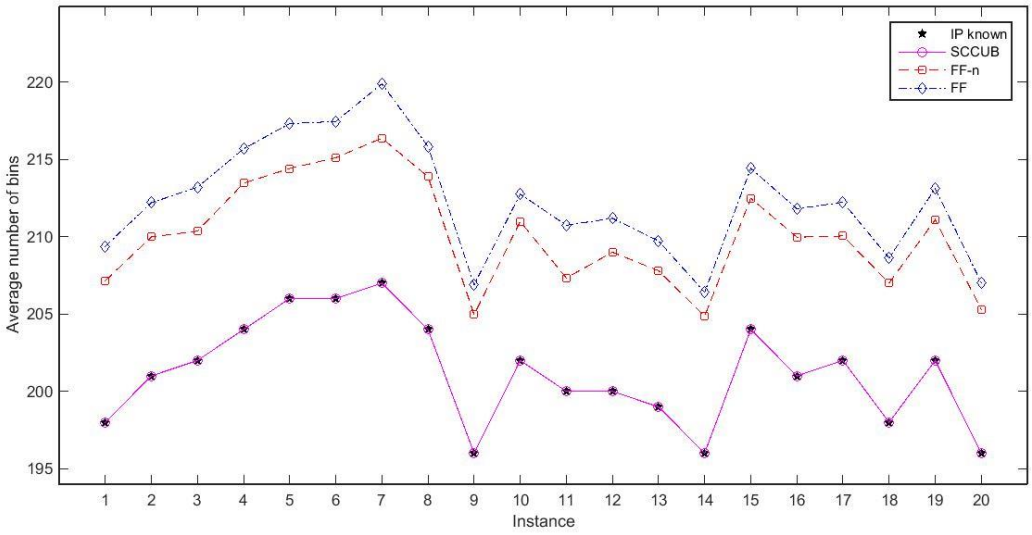


Figure 4.3 Comparison of the upper bounds generated by SCCUB, FF, and FF-n with known IPs for instances of class u500.

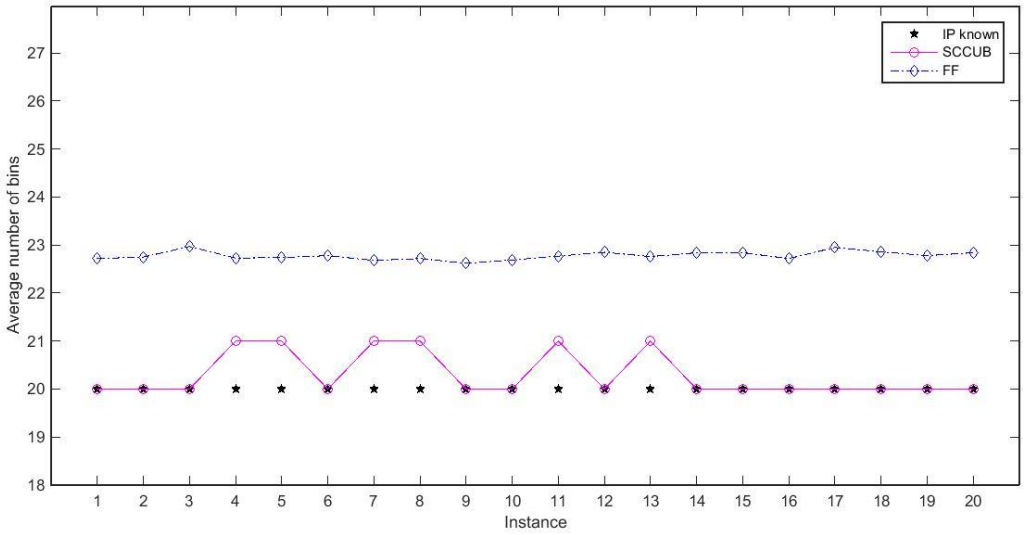


Figure 4.4 Comparison of the upper bounds generated by SCCUB, FF, and FF-n with known IPs for instances of class t60.

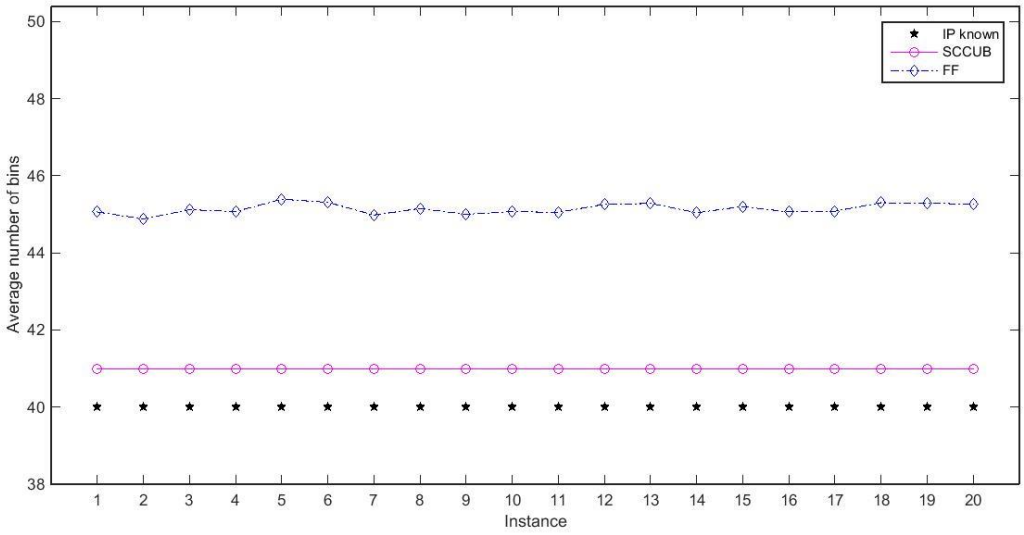


Figure 4.5 Comparison of the upper bounds generated by SCCUB, FF, and FF-n with known IPs for instances of class t120.

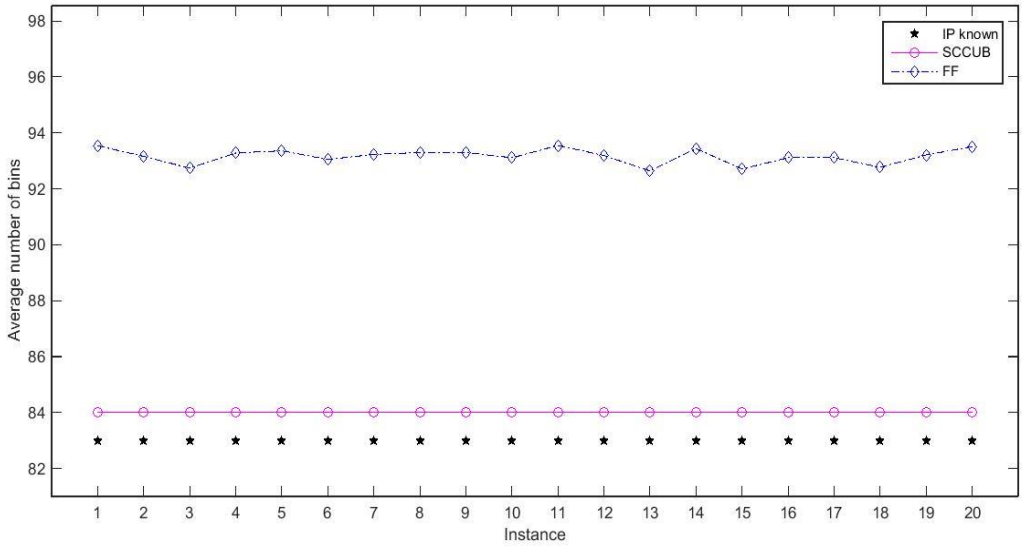


Figure 4.6 Comparison of the upper bounds generated by SCCUB, FF, and FF-n with known IPs for instances of class t249.

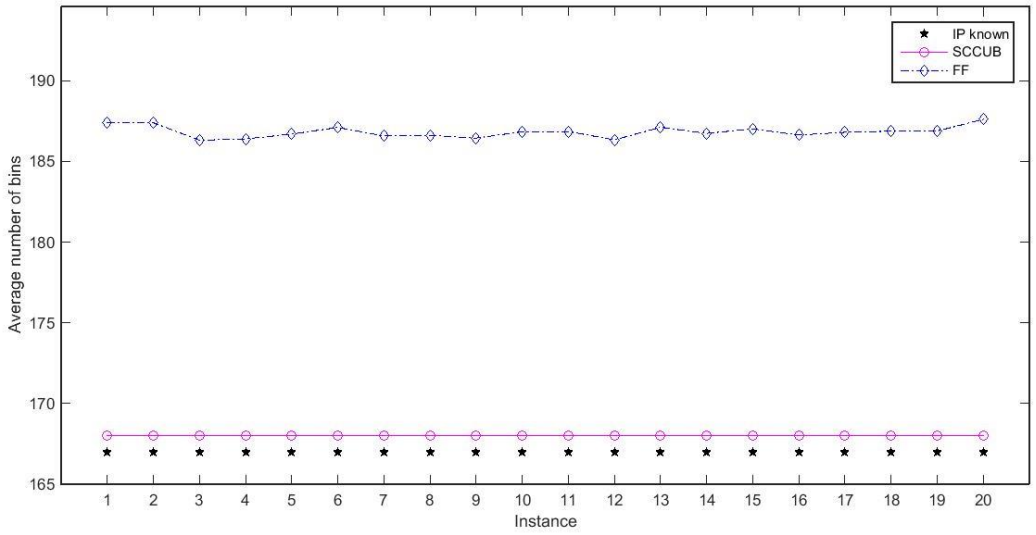


Figure 4.7 Comparison of the upper bounds generated by SCCUB, FF, and FF-n with known IPs for instances of class t501.

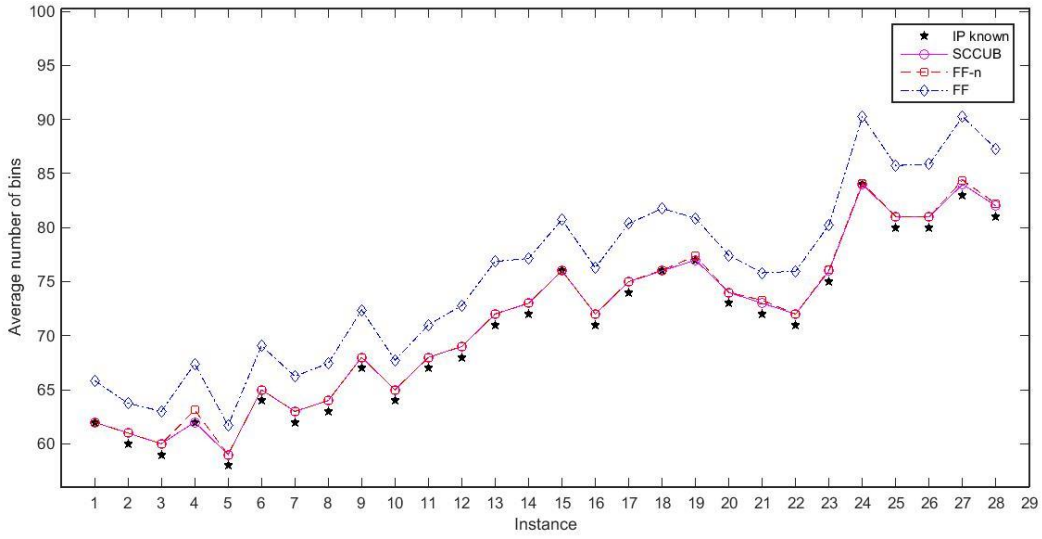


Figure 4.8 Comparison of the upper bounds generated by SCCUB, FF, and FF-n with known IPs for instances of class Hard28.

Chapter 5

Upper bounds for BPP using solutions of the continuous relaxation of set-covering formulation with an elimination operator -p (SCCUB-p)

5.1. Procedure SCCUB-p

In this chapter we improve the quality of the upper bounds obtained for BPP through application of SSCUB procedure, by introducing an elimination operator -p. This operator is simply a selection scheme for selecting a combination of the patterns from the candidate pool of SCCUB procedure at each of its steps.

Observation 5.1 Excluding few patterns from the list of patterns having values greater than 0.5 at each step of SCCUB procedure might lead to a better upper bound solution for BPP.

In other words, although solutions of the continuous relaxation of set-covering formulation of BPP are near optimal, it might be the case that combination of all the basic patterns having values greater than 0.5 at each step of SCCUB procedure is not the best choice of constructing the partial upper bound solutions. To this end, some (all) of the patterns of that

step should not be included in the partial upper bound solution in order to get a strong upper bound at the end of the upper bounding procedure.

We believe that it is possible to find a rule for exclusion of such undesired patterns based on information of the counterparts of the basic patterns and some dominance relations among them. However, our efforts for finding a general rule as the selection scheme governing all the instances of BPP were not successful, and it seems that combinations of different criteria should be employed to provide a rigorous selection scheme capable of selecting the right patterns having values greater than 0.5 at each step of SCCUB procedure. It is also emphasized that for a few instances of BPP selecting patterns having values greater than 0.5 is not sufficient to construct a high quality partial upper bound solution. Rather, a good selection scheme should also consider basic patterns whose values are not greater than 0.5., as it should, as well find their best combination to be fixed as the partial upper bound solution.

Nevertheless, in the current research, we prefer to deal with such problematic instances by using our branch-and-price method instead of extending the selection scheme to consider all the basic patterns. This is to keep the computational costs of the selection scheme in the minimum level.

In this perspective, an elimination operator -p is introduced which eliminates %p of the patterns of each step of SCCUB procedure randomly. That is, at each step of SCCUB-p procedure, %p of the patterns are excluded from the list of the patterns having values greater than 0.5, and the remaining patterns of values greater than 0.5 will be considered as the partial solutions of the upper bound solution. It is worth mentioning that our traditional SSCUB procedure is a special case of SCCUB-p procedure where $p = 0$ and no pattern is eliminated from the list of patterns having values greater than 0.5.

The details and pseudo-code of SCCUB-p procedure is found in table (5.1).

Table 5.1 Pseudo-code for procedure SCCUB-p

Procedure SCCUB-p

Input: number of items (n), weights of the items ($\mathbf{W}_{1\times n}$), bin capacity (c), percentage of patterns to be eliminated from list of patterns having values greater than 0.5 (p).

Output: Upper bound solution (SCCUB-p) and number of optimal patterns (OPT(SCCUB-p)).

Step1. Define SCCUB-p.

Step2. **While** true

Solve continuous relaxation of set-covering formulation of BPP by **algorithm** revised simplex. Return **B** and $\mathbf{X}_B$.

Find basic patterns having values greater than 0.5.

m ← number of patterns having values greater than 0.5

If no basic patterns is found with value greater than 0.5

Add the pattern with the largest value to SCCUB-p.

Else

$$e = \left\lfloor \frac{p}{100} \times m \right\rfloor$$

Randomly choose e patterns from list of the patterns with values greater than 0.5 and eliminate them from the list.

Add all the remaining patterns from list of the patterns with values greater than 0.5 to SCCUB-p.

Eliminate items of SCCUB-p from **W** and update n.

If n=0

Break

End of **while** loop

Step 3. Outputs are SCCUB-p and OPT(SCCUB-p).

5.2 Computational results

In this section, experiments are carried out for the instances of triplets and Hard28 to show how SCCUB procedure could be improved by applying the elimination operator. Our previous experimental results from chapter 4, testify that optimal solution for uniform class of instances is found by using the simple SCCUB procedure, and employing SCCUB-p procedure will only increase the computational time of the procedure while providing the same tight upper bound.

Designation of the proper value for the elimination operator –p requires further investigations on the structure of the BPP. In the present work, we only conduct our experiments with values of $p = 0.25, 0.5$, and 0.75.

Tables (5.2) to (5.6) show the computed upper bounds by using SCCUB-p procedure for the instances of triplets and Hard28 with their corresponding relative times. Results for SCCUB-25, SCCUB50, and SCCUB75 are averaged over100 replications.

As observed from the tables, SCCUB50 and SCCUB75 find the optimal solution for majority of the instances of triplets and Hard28. Specially, SCCUB75 solves some of the unsolved instances of Hard28 class. More exactly, SCCUB75 finds the optimal solution to 24 instances of Hard28 class whereas the alleged state-of-the-art method only solves 16 of them.

Furthermore, figures (5.1) to (5.10) compare the quality of the solutions found using SCCUB-p procedure for different values of p, and compare the average relative times of the procedure obtained for different values of p. These figures indicate a dramatic growth in the relative time of SCCUB75 procedure. On the one hand, quality of the solutions derived by SCCUB75 is high. On the other hand, on average as much as computational time spent to solve the root node by column generation, time needs to be spent on running each iteration of the SCCUB75 procedure. Therefore, an intelligent combination of the SCCUB-p upper bounding procedures should be designed to make the computational costs of our upper bounding technique affordable. Such a design will be presented in the next chapter.

Table 5.2 Computational results of SCCUB-p for instances of t60

Instance	IP	SCCUB-0	SCCUB-25	SCCUB-50	SCCUB-75	$t^{\text{SCCUB-0}}_{\text{relative}}$	$t^{\text{SCCUB-25}}_{\text{relative}}$	$t^{\text{SCCUB-50}}_{\text{relative}}$	$t^{\text{SCCUB-75}}_{\text{relative}}$
t60_00	20	20	20	20	20	1.080	1.080	1.080	1.080
t60_01	20	20	20.79	20.63	20.82	1.699	1.565	1.877	2.218
t60_02	20	20	20	20	20	1.147	1.098	1.235	1.553
t60_03	20	**21**	20.87	20.67	20.37	1.461	1.337	1.501	2.138
t60_04	20	**21**	20.95	20.60	20.18	1.247	1.069	1.213	1.444
t60_05	20	20	20	20	20	1.079	1.079	1.079	1.079
t60_06	20	**21**	**21**	**21**	20.85	1.303	1.117	1.312	1.865
t60_07	20	**21**	20.96	20.83	20.27	1.297	1.110	1.292	1.523
t60_08	20	20	20	20	20	1.520	1.301	1.391	1.589
t60_09	20	20	20	20	20	1.072	1.072	1.072	1.072
t60_10	20	**21**	**21**	**21**	20.86	1.305	1.130	1.310	1.830
t60_11	20	20	20	20	20	1.073	1.073	1.073	1.073
t60_12	20	**21**	20.86	20.55	20.42	1.362	1.189	1.371	1.867
t60_13	20	20	20.22	20.35	20.18	1.554	1.547	1.704	2.120
t60_14	20	20	20	20	20	1.077	1.077	1.077	1.077
t60_15	20	20	20	20	20.03	1.187	1.120	1.205	1.470
t60_16	20	20	20	20	20	1.061	1.061	1.061	1.061
t60_17	20	20	20	20	20	1.077	1.077	1.077	1.077
t60_18	20	20	20	20	20	1.081	1.081	1.081	1.081
t60_19	20	20	20	20	20	1.072	1.072	1.072	1.072
Minimum	20	20	20	20	20	1.061	1.061	1.061	1.061
Average	20	20.30	20.33	20.28	20.19	1.237	1.162	1.254	1.464
Maximum	20	21	21	21	20.86	1.699	1.565	1.877	2.218

Table 5.3 Computational results of SCCUB-p for instances of t120

Instance	IP	SCCUB-0	SCCUB-25	SCCUB-50	SCCUB-75	$t^{\text{SCCUB-0}}_{\text{relative}}$	$t^{\text{SCCUB-25}}_{\text{relative}}$	$t^{\text{SCCUB-50}}_{\text{relative}}$	$t^{\text{SCCUB-75}}_{\text{relative}}$
t120_00	40	**41**	40.79	40.88	40.62	1.138	1.187	1.466	2.306
t120_01	40	**41**	40.99	40.9	40.66	1.110	1.146	1.497	2.421
t120_02	40	**41**	40.89	40.82	40.66	1.135	1.235	1.628	2.488
t120_03	40	**41**	40.96	40.75	40.6	1.221	1.345	1.792	2.872
t120_04	40	**41**	40.91	40.8	40.66	1.206	1.302	1.677	2.778
t120_05	40	**41**	40.95	40.84	40.64	1.138	1.195	1.613	2.626
t120_06	40	**41**	40.79	40.85	40.66	1.162	1.260	1.62	2.573
t120_07	40	**41**	40.98	40.84	40.71	1.125	1.191	1.578	2.426
t120_08	40	**41**	40.97	40.83	40.63	1.145	1.182	1.549	2.531
t120_09	40	**41**	40.89	40.89	40.7	1.203	1.320	1.673	2.711
t120_10	40	**41**	**41**	40.9	40.76	1.149	1.229	1.612	2.736
t120_11	40	**41**	40.96	40.88	40.64	1.119	1.190	1.554	2.508
t120_12	40	**41**	40.88	40.83	40.54	1.146	1.223	1.590	2.445
t120_13	40	**41**	40.85	40.79	40.57	1.155	1.259	1.657	2.741
t120_14	40	**41**	40.91	40.89	40.72	1.107	1.145	1.456	2.364
t120_15	40	**41**	40.76	40.82	40.59	1.168	1.297	1.654	2.646
t120_16	40	**41**	40.94	40.85	40.6	1.108	1.152	1.442	2.459
t120_17	40	**41**	40.98	40.87	40.66	1.121	1.174	1.576	2.538
t120_18	40	**41**	40.92	40.81	40.63	1.216	1.381	1.781	2.925
t120_19	40	**41**	40.99	40.8	40.71	1.108	1.138	1.424	2.410
Minimum	40	41	40.76	40.75	40.54	1.107	1.138	1.424	2.306
Average	40	41	40.9155	40.842	40.648	1.149	1.22755	1.59195	2.5752
Maximum	40	41	41	40.9	40.76	1.221	1.381	1.792	2.925

Table 5.4 Computational results of SCCUB-p for instances of t249

Instance	IP	SCCUB-0	SCCUB-25	SCCUB-50	SCCUB-75	$t_{relative}^{SCCUB-0}$	$t_{relative}^{SCCUB-25}$	$t_{relative}^{SCCUB-50}$	$t_{relative}^{SCCUB-75}$
t249_00	83	**84**	83.87	83.79	83.59	1.044	1.134	1.392	2.260
t249_01	83	**84**	83.95	83.85	83.61	1.042	1.112	1.367	2.249
t249_02	83	**84**	83.88	83.79	83.64	1.050	1.137	1.404	2.244
t249_03	83	**84**	83.93	83.85	83.57	1.054	1.155	1.441	2.350
t249_04	83	**84**	83.93	83.8	83.53	1.054	1.153	1.410	2.250
t249_05	83	**84**	83.92	83.78	83.66	1.064	1.151	1.434	2.225
t249_06	83	**84**	83.89	83.83	83.64	1.051	1.136	1.393	2.133
t249_07	83	**84**	83.87	83.75	83.64	1.104	1.233	1.519	2.290
t249_08	83	**84**	83.92	83.81	83.64	1.060	1.175	1.475	2.332
t249_09	83	**84**	83.88	83.80	83.60	1.062	1.157	1.427	2.256
t249_10	83	**84**	83.88	83.80	83.60	1.073	1.177	1.465	2.393
t249_11	83	**84**	83.85	83.74	83.58	1.058	1.157	1.412	2.218
t249_12	83	**84**	83.92	83.77	83.53	1.057	1.153	1.399	2.163
t249_13	83	**84**	83.9	83.84	83.60	1.054	1.139	1.383	2.072
t249_14	83	**84**	83.96	83.82	83.65	1.064	1.160	1.435	2.194
t249_15	83	**84**	83.91	83.86	83.60	1.067	1.160	1.440	2.236
t249_16	83	**84**	83.91	83.79	83.61	1.078	1.176	1.480	2.393
t249_17	83	**84**	83.87	83.81	83.60	1.059	1.147	1.425	2.222
t249_18	83	**84**	83.90	83.84	83.64	1.057	1.154	1.408	2.235
t249_19	83	**84**	83.86	83.73	83.45	1.056	1.141	1.397	2.170
Minimum	83	84	83.85	83.73	83.45	1.042	1.112	1.367	2.072
Average	83	84	83.90	83.80	83.59	1.060	1.155	1.425	2.244
Maximum	83	84	83.96	83.86	83.66	1.104	1.233	1.519	2.393

Table 5.5 Computational results of SCCUB-p for instances of t501

Instance	IP	SCCUB-0	SCCUB-25	SCCUB-50	SCCUB-75	$t^{\text{SCCUB-0}}_{\text{relative}}$	$t^{\text{SCCUB-25}}_{\text{relative}}$	$t^{\text{SCCUB-50}}_{\text{relative}}$	$t^{\text{SCCUB-75}}_{\text{relative}}$
t501_00	167	**168**	167.89	167.80	167.60	1.040	1.128	1.369	2.180
t501_01	167	**168**	167.89	167.73	167.50	1.027	1.106	1.323	2.064
t501_02	167	**168**	167.89	167.73	167.66	1.027	1.106	1.323	2.053
t501_03	167	**168**	167.91	167.75	167.58	1.022	1.094	1.291	2.003
t501_04	167	**168**	167.94	167.80	167.70	1.030	1.130	1.376	2.230
t501_05	167	**168**	167.91	167.75	167.66	1.029	1.126	1.363	2.175
t501_06	167	**168**	167.93	167.75	167.75	1.026	1.084	1.298	2.037
t501_07	167	**168**	167.93	167.82	167.64	1.033	1.123	1.374	2.195
t501_08	167	**168**	167.88	167.82	167.66	1.038	1.135	1.383	2.231
t501_09	167	**168**	167.91	167.82	167.63	1.030	1.114	1.340	2.143
t501_10	167	**168**	167.92	167.85	167.61	1.025	1.106	1.330	2.091
t501_11	167	**168**	167.87	167.66	167.69	1.030	1.132	1.374	2.133
t501_12	167	**168**	167.84	167.82	167.66	1.020	1.090	1.317	2.087
t501_13	167	**168**	167.89	167.83	167.64	1.021	1.101	1.315	2.079
t501_14	167	**168**	167.86	167.78	167.57	1.025	1.109	1.324	2.078
t501_15	167	**168**	167.88	167.69	167.59	1.035	1.134	1.394	2.251
t501_16	167	**168**	167.87	167.82	167.51	1.030	1.118	1.358	2.187
t501_17	167	**168**	167.91	167.80	167.51	1.031	1.125	1.374	2.1872
t501_18	167	**168**	167.88	167.80	167.65	1.034	1.120	1.374	2.0899
t501_19	167	**168**	167.86	167.83	167.56	1.025	1.106	1.334	2.1432
Minimum	167	168	167.84	167.66	167.50	1.020	1.084	1.291	2.003
Average	167	168	167.89	167.78	167.61	1.028	1.114	1.346	2.131
Maximum	167	168	167.94	167.85	167.75	1.040	1.135	1.394	2.251

Table 5.6 Computational results of SCCUB-p for instances of Hard28

Instance	IP	SCCUB-0	SCCUB-25	SCCUB-50	SCCUB-75	$t_{relative}^{SCCUB-0}$	$t_{relative}^{SCCUB-25}$	$t_{relative}^{SCCUB-50}$	$t_{relative}^{SCCUB-75}$
BPP14	62	62	62	62	62	1.013	1.022	1.180	1.792
BPP832	60	**61**	60.94	60.99	60.88	1.010	1.016	1.257	1.644
BPP40	59	**60**	**60**	**60**	**60**	1.017	1.105	1.122	1.666
BPP360	62	62	62.99	62.92	62.6	1.010	1.030	1.150	1.746
BPP645	58	**59**	**59**	**59**	58.93	1.007	1.027	1.186	1.708
BPP742	64	**65**	**65**	64.98	64.94	1.007	1.016	1.130	1.863
BPP766	62	**63**	**63**	**63**	62.98	1.008	1.021	1.124	1.648
BPP60	63	**64**	**64**	**64**	**64**	1.013	1.027	1.133	1.682
BPP13	67	**68**	**68**	**68**	67.97	1.015	1.053	1.223	9.409
BPP195	64	**65**	**65**	64.89	64.71	1.012	1.042	1.194	1.742
BPP709	67	**68**	**68**	**68**	67.97	1.011	1.038	1.191	1.664
BPP785	68	**69**	68.97	68.91	68.64	1.008	1.037	1.168	1.726
BPP47	71	**72**	71.8	71.63	71.35	1.008	1.032	1.170	1.654
BPP181	72	**73**	**73**	**73**	72.98	1.006	1.028	1.148	1.727
BPP359	76	76	76	76	76	1.009	1.031	1.175	1.837
BPP485	71	**72**	**72**	71.97	71.8	1.003	1.013	1.110	1.062
BPP640	74	**75**	74.95	74.81	74.6	1.008	1.028	2.555	1.058
BPP716	76	76	76	76	76	1.008	1.046	1.207	1.735
BPP119	77	77	77	77	77	1.013	1.048	1.201	1.732
BPP144	73	**74**	**75**	**74**	**74**	1.015	1.059	1.203	1.707
BPP561	72	**73**	72.9	72.7	72.61	1.009	1.038	1.190	1.707
BPP781	71	**72**	71.97	71.78	71.7	1.004	1.020	1.144	1.640
BPP900	75	**76**	75.99	75.9	75.85	1.008	1.037	1.175	1.729

Table 5.6 continued

Instance	IP	SCCUB-0	SCCUB-25	SCCUB-50	SCCUB-75	$t^{\text{SCCUB-0}}_{\text{relative}}$	$t^{\text{SCCUB-25}}_{\text{relative}}$	$t^{\text{SCCUB-50}}_{\text{relative}}$	$t^{\text{SCCUB-75}}_{\text{relative}}$
BPP175	84	84	84	84	84	1.004	1.019	1.134	1.580
BPP178	80	**81**	80.99	80.83	80.63	1.002	1.015	1.115	2.254
BPP419	80	**81**	**81**	**81**	**81**	1.006	1.029	1.167	1.712
BPP531	83	**84**	83.80	83.56	83.25	1.005	1.028	1.519	1.640
BPP814	81	**82**	81.71	81.41	81.30	1.006	1.029	1.157	1.542
Minimum	58	59	59	59	58.93	1.002	1.013	1.11	1.058
Average	70.42	71.21	71.25	71.15	71.06	1.008	1.033	1.229	1.950
Maximum	84	84	84	84	84	1.017	1.105	2.555	9.409

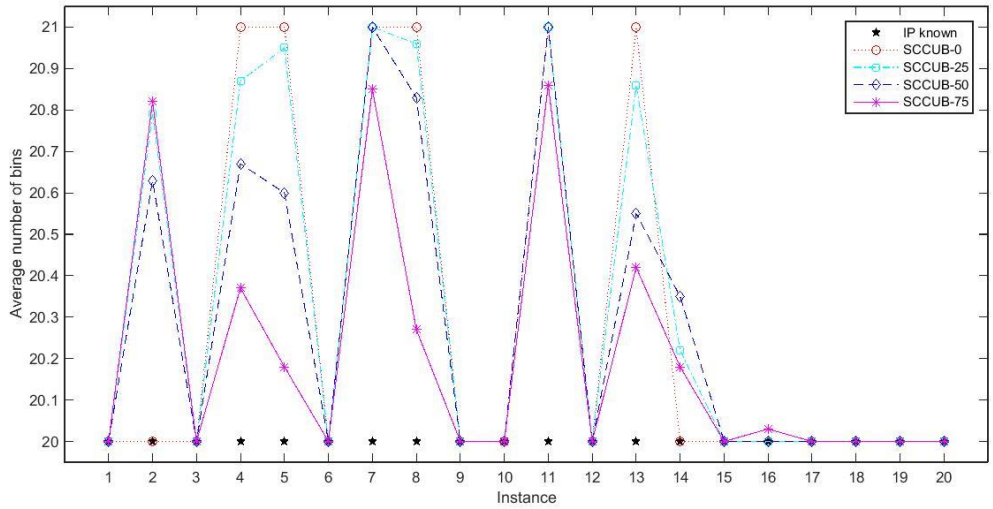


Figure 5.1 Comparison of the upper bounds generated by SCCUB-0, SCCUB-25, SCCUB-50, SCCUB-75 with known IPs for instances of class t60.

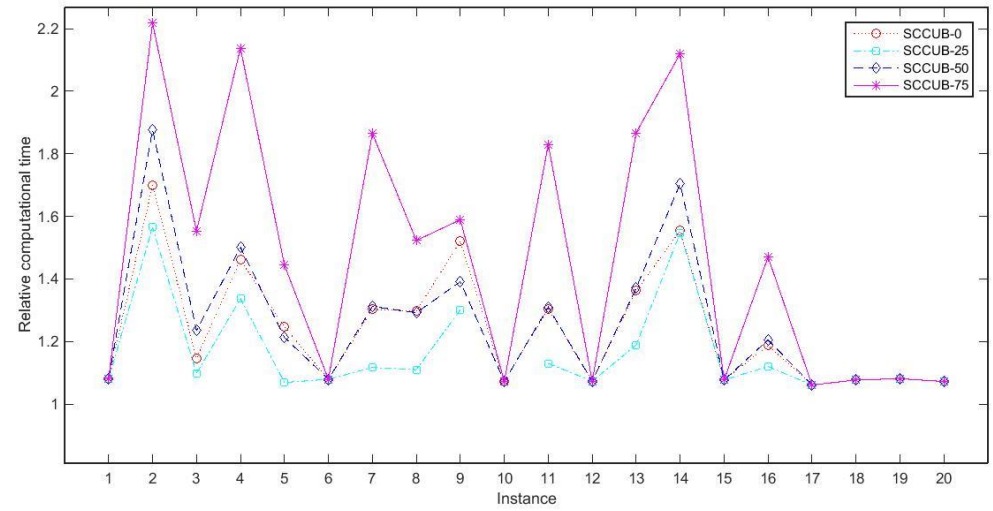


Figure 5.2 Comparison of the relative time of SCCUB-0, SCCUB-25, SCCUB-50, and SCCUB-75 for instances of class t60.

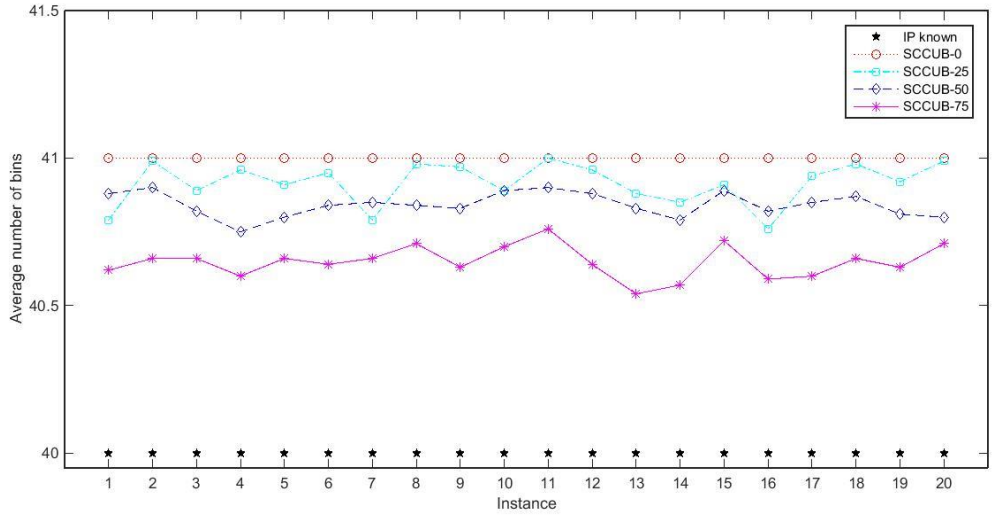


Figure 5.3 Comparison of the upper bounds generated by SCCUB-0, SCCUB-25, SCCUB-50, SCCUB-75 with known IPs for instances of class t120.

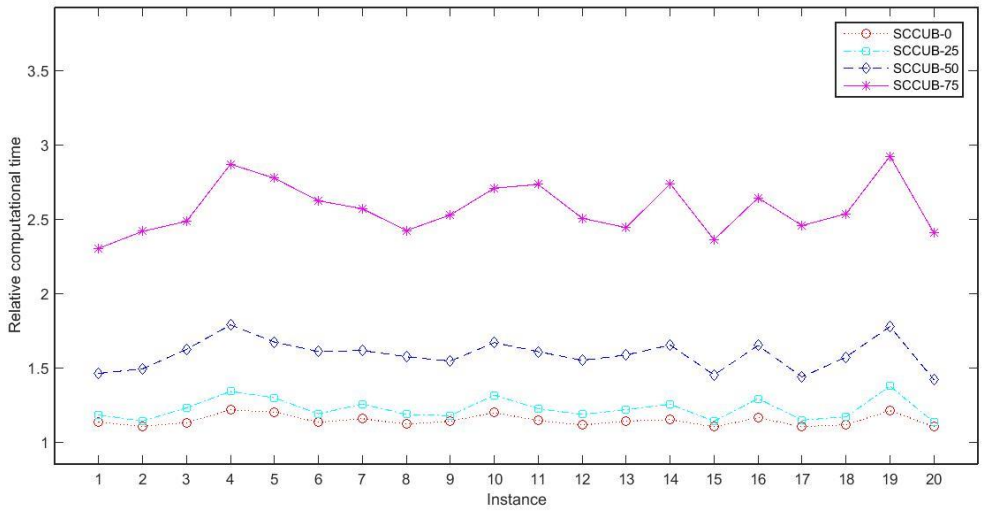


Figure 5.4 Comparison of the relative time of SCCUB-0, SCCUB-25, SCCUB-50, and SCCUB-75 for instances of class t120.

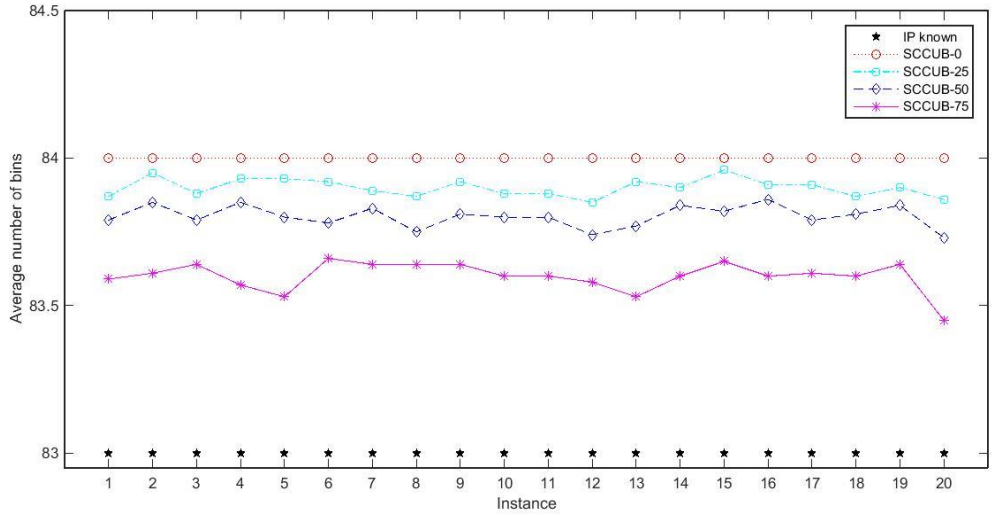


Figure 5.5 Comparison of the upper bounds generated by SCCUB-0, SCCUB-25, SCCUB-50, SCCUB-75 with known IPs for instances of class t249.

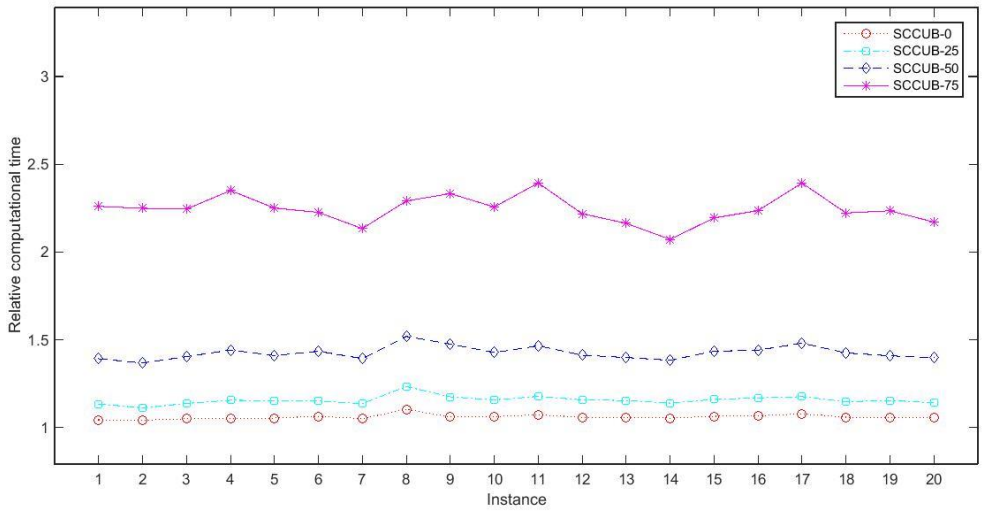


Figure 5.6 Comparison of the relative time of SCCUB-0, SCCUB-25, SCCUB-50, and SCCUB-75 for instances of class t249.

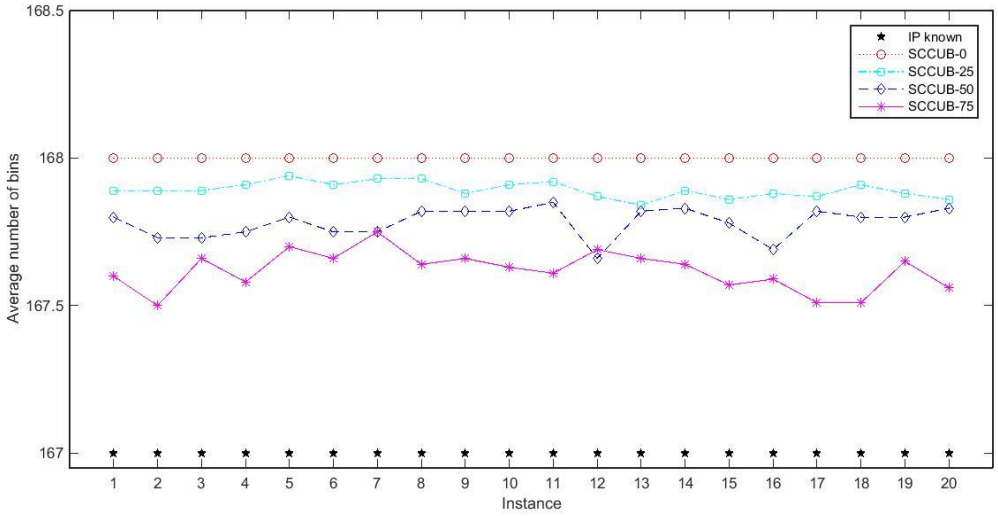


Figure 5.7 Comparison of the upper bounds generated by SCCUB-0, SCCUB-25, SCCUB-50, SCCUB-75 with known IPs for instances of class t501.

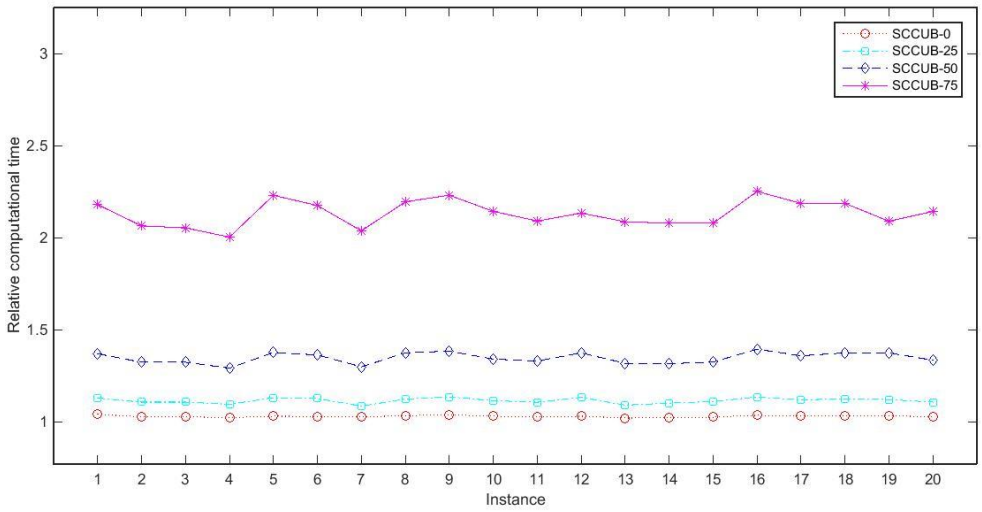


Figure 5.8 Comparison of the relative time of SCCUB-0, SCCUB-25, SCCUB-50, and SCCUB-75 for instances of class t501.

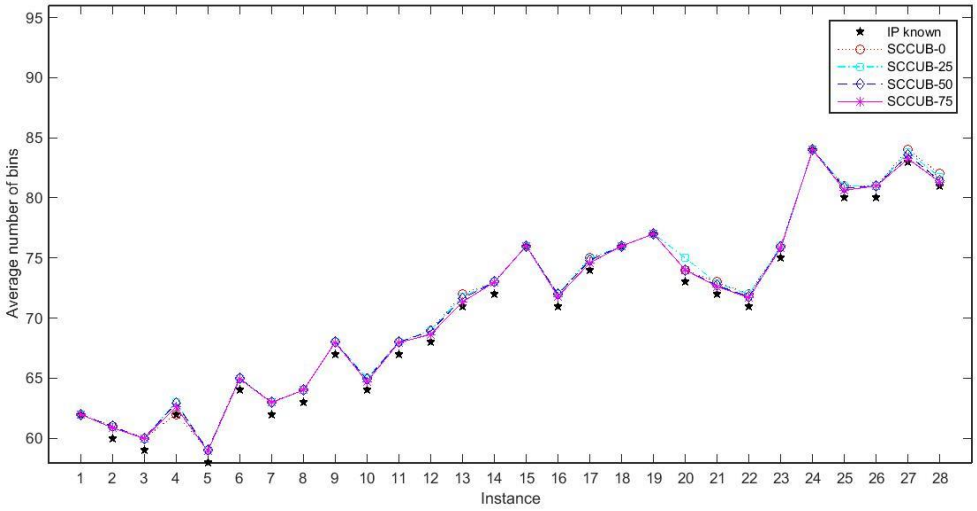


Figure 5.9 Comparison of the upper bounds generated by SCCUB-0, SCCUB-25, SCCUB-50, SCCUB-75 with known IPs for instances of class Hard28.

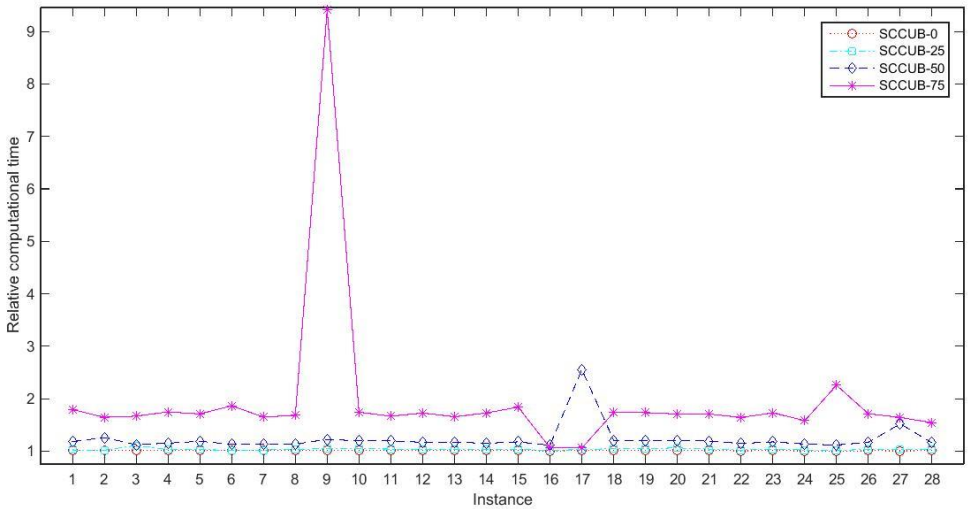


Figure 5.10 Comparison of the relative time of SCCUB-0, SCCUB-25, SCCUB-50, and SCCUB-75 for instances of class Hard28.

Chapter 6

Upper bounding technique for BPP using solutions of the continuous relaxation of set-covering formulation with an elimination operator- p and a dual BPP approach (SCCUB-p-d)

In this chapter we will design an efficient upper bounding technique for BPP based on the SCCUB-p procedure introduced previously. Bringing intelligent algorithms into a procedure like SCCUB-p, however, requires prior knowledge of the *Maximum Cardinality Bin Packing Problem* (also known as the *dual bin packing problem*). To this end, details of the dual BPP are presented and the way the latter is combined to the SCCUB-p procedure in order to construct a rigorous upper bounding technique for BPP will be described.

6.1 Maximum cardinality bin packing problem

In the following, we investigate the relation between the bin packing problem and its dual problem and show how information provided by a dual bin packing problem could be useful in reducing the computational time of the SCCUB-p procedure.

Maximum cardinality bin packing problem (CBP) accounts for the case where; m bins having each of them a capacity $c > 0$ are identical and where a set of items $I = \{i_1, i_2, \dots, i_n\}$ each one of them with a weight $0 < w_i \leq c$ are to be packed optimally into the said bins.

This problem which is sometimes known as the dual version of the bin packing problem could be formulated using the following set of expressions:

$$\begin{aligned} &\max \sum_{i=1}^{n} \sum_{k=1}^{m} x_{ik} \\ &\textstyle\sum_{i=1}^{n} w_i x_{ik} \leq c && k = 1,2, \dots, m \\ &\textstyle\sum_{k=1}^{m} x_{ik} \leq 1 && i = 1,2, \dots, n \qquad (6.1) \\ &x_{ik} \in \{0,1\} && i = 1, \dots, n\,,\ k = 1, \dots, m. \end{aligned}$$

Labbé et *al*., in [50], first sorted the items in the ascending order of weights, and then proposed the following upper bounds for MKP.

Upper bound $\bar{U}_0$

The first upper bound on MKP is given by:

$$\bar{U}_0 = \max_{1 \leq k \leq n} \{k : \textstyle\sum_{i=1}^{k} w_i \leq mc\}.$$

This is a valid upper bound on MKP since it packs the first $\bar{U}_0$ smallest items into m bins.

Upper bound $\bar{U}_1$

To derive the second upper bound on MKP, an entity $Q(j)$ is introduced as being:

$$Q(j) = \max\left\{k : j \leq k \leq n, \textstyle\sum_{i=1}^{k} w_i \leq jc\right\},\ j = 1, \dots, m.$$

In which case, $U_1(j)$ could be considered as an upper bound of the MKP and reads:

$$U_1(j) = Q(j) + \left\lfloor \frac{Q(j)}{j} \right\rfloor (m - j)$$

In fact, given a specific value to j, this upper bound packs as many items as possible into j bins after choosing the ones with smallest weights among all the items. Then, the rest of the bins $(m - j)$ are filled, each one by at most $\left\lfloor \frac{Q(j)}{j} \right\rfloor$ items. Because, the items are sorted in the ascending order of the weights, and this fill, is as if the larger items are treated as the packed (smaller) ones.

To derive a stronger upper bound on MKP, $\overline{U}_1$ should then be computed using:

$$\overline{U}_1 = \min_{j=1,\dots,m} U_1(j).$$

Upper bound $\overline{U}_3$

A valid upper bound for $i = 1, \dots, n$ could be obtained from the following expression:

$$U_2(i) = (i - 1) + m \left\lfloor \frac{c}{w_i} \right\rfloor.$$

This upper bound is derived again based on the fact that the items are sorted well in advance. For a certain item i, $\left\lfloor \frac{c}{w_i} \right\rfloor$ is an upper bound on the number of the items having weights larger than item i and that are potentially packable into a bin, and the term $(i - 1)$ implies that all the items having weights less than item i are packed into $(i - 1)$ bins.

Then, a valid upper bound on MKP denoted $\overline{U}$ is expressed as being:

$$\overline{U} = \min\{ \overline{U}_1, \overline{U}_2, \overline{U}_3 \}.$$

6.2 Procedure SCCUB-p-d

In this section, we will describe how it is possible to reduce computational time of the SCCUB-p procedure by utilizing information of the upper bound of MKP obtained a-priory.

If one supposes at the first step of the SCCUB-p procedure that m bins (patterns) are chosen to be included in the SCCUB-p solution as the partial upper bound solutions, and that the chosen bins in total contain n itmes, then, the sufficient condition for m bins to be the optimal number of bins to accommodate these n items is that the upper bound of MKP with n items and $(m-1)$ bins should be less than n. In other words, were it to be the case that the upper bound placed on the number of the items to be packed in $(m-1)$ is not less than n, the configuration of the n items into m bins derived at the first step of SCCUB-p procedure might not represent the optimal configuration of bins eligible to harbor the n items. The same premise holds for the rest of the steps of the SCCUB-p procedure.

Proposition 6.1 Given the configuration of n items packed into m bins, the sufficient condition for m to be the optimal number of bins to pack n items is that the upper bound placed on the number of the items packed into $(m-1)$ should be less than n. In mathematical terms, we have:

$$\overline{U}_{m-1}^{n} < n \Rightarrow (m,n) \text{ is optimal} \tag{6.2}$$

where $\overline{U}_{m-1}^{n}$ shows an upper bound to the number of items packed into $(m-1)$ bins, and the pair of integers (m,n) is the packing configuration of n items into m bins.

Proof Suppose $\overline{U}_{m-1}^{n} < n$ but (m,n) is not optimal. Then, the optimal number of bins to pack n items should be less than m. In other words, the optimal configuration of items into bins, is (m',n) where $m' < m$. But, we know that not all of the items could be packed into m' bins since:

$$\overline{U}_{m'}^{n} \leq \overline{U}_{m-1}^{n} < n.$$

This is a contradiction since we assumed that all of the items are packable into m' bins. Therefore, the couple (m,n) is the optimal configuration of items into bins.

Proposition 6.2 Given a configuration of n items packed into m bins, the necessary condition for m not to be the optimal number of bins for packing n items is that the upper bound of the number of the items packed into $(m-1)$ should not be less than n. In mathematical terms, we have:

$$(m,n) \text{ is not optimal} \Rightarrow \ \overline{U}_{m-1}^{n} \geq n \tag{6.3}$$

Proof This deduction is immediate from relation (6.2).

Let us again consider the first step of the SCCUB-p procedure where m bins are produced to accommodate n items. Having considered the proposition (6.1), if the upper bound computed for the n items destined to be packed into $(m-1)$ is not less than n, then the SCCUB-p procedure could be terminated since the tight upper bound is not likely to be attained at the end of the procedure. The same premise holds true for all of the other steps of the SCCUB-p procedure. It is essential to point out that the above-mentioned premise heavily relies on the quality of the upper bound provided by $\overline{U}$. Meaning that, the following expression is not generally true:

$$\overline{U}_{m-1}^{n} \geq n \Rightarrow (m,n) \text{ is not optimal.} \tag{6.4}$$

Our observations on the performance of $\overline{U}$ on BPP instances show the high quality of this bound. Therefore, this bound could be useful in reducing the computational resources needed by the SCCUB procedure.

We would like to elaborate more on the mentioned premise by putting it into decomposition terminology where each step of the SCCUB-p procedure might be viewed as a block of an optimization problem.

Definition 6.1 Blocks of the SCCUB-p procedure

A block of the SCCUB-p procedure at iteration k is defined as being the partial upper bound solution fixed until the k^{th} iteration.

For instance, if n' items are packed into m' bins at the first iteration of the SCCUB-p, then the first block of the optimization problem could be regarded as the optimization problem of having n' items and m' as the optimal number of bins to pack the items.

Let us now move on to the second iteration of the SCCUB-p where n'' items are packed into m'' bins. The second block can be viewed as an optimization problem where the $(n' + n'')$ items are packed into $(m' + m'')$ bins, and so on. It is obvious that definition (6.1) implies that the very last block of the procedure corresponds to our original BPP where $n = n' + n'' + \cdots n^{end}$ items are packed into $m = m' + m'' + \cdots m^{end}$ bins.

Now, for each block of the SCCUB-p procedure, if the sufficient condition given in proposition (6.1) does not hold, we might terminate the procedure knowing that the optimal solution is not likely to be obtained by executing the remaining steps. However, it is vital to point out that if each block of the SCCUB-p procedure holds the sufficient condition of proposition (6.1), this does not guarantee the achievement of the optimal solution at the end of the procedure. For, each block could be viewed as an *on-line Bin Packing Problem* where some of the items are yet to arrive in future steps. Therefore, optimality of each block is not a guaranty for optimality of the upper bound solution found at the end of the procedure, except for the very last iteration where all the items are present and no more items will be expected to arrive in the future.

Table (6.1) presents the pseudo-code for the SCCUB-p procedure with dual BPP approach (SCCUB-p-d). As noticed, the valuable root node basic solutions are saved after the revised simplex algorithm is terminated at the first iteration of SCCUB-p-d. This will reduce the total computational time of the procedure since the need for solving the LP relaxation from the scratch for the future iterations is eliminated.

Following the termination of revised simplex algorithm, SCCUB-0 is invoked to find an upper bound solution for the problem. If the gap between upper and lower bounds is less than 1, then SCCUB-p-d will be terminated knowing that optimality of the upper bound solution obtained is proved. Otherwise, SCCUB-50, for which our previous experiments testify its high performance, will be run in aim of finding a tight upper bound solution. If the latter procedure fails to close the gap and prove the optimality of the upper bound solutions, after 30 iterations, our most computationally expensive procedure, SSCUB-75, will be executed for 20 iterations. Finally, the SCCUB-p-d procedure is terminated by reaching the maximum iterations (50 iterations).

Table 6.1 Pseudo-code for procedure SCCUB-p-d

Procedure SCCUB-p-d

Input: number of items (n), weights of the items ($\mathbf{W}_{1\times n}$), bin capacity (c), maxit = 50

Output: Upper bound solution (SCCUB-p-d), maximum GAP between upper and lower bound (GAP)

Step 1. Define SCCUB-p-d.

Step 2. Solve continuous relaxation of set-covering formulation of BPP by **algorithm** revised simplex method. Return **B**, $\mathbf{X}_B$, and OPTCG.

Step 3. Run SCCUB-0. Return SCCUB-0 and OPT(SCCUB-0).

Step 4. GAP ⟵ OPT(SCCUB-0) – ⌈OPTCG⌉

Step 5. n′ ← n

$\mathbf{W}' \leftarrow \mathbf{W}$

Step 6. **If** GAP < 1

SCCUB-0 is the optimal solution for BPP.

Terminate **Procedure** SCCUB-p-d.

Else

Saved$\mathbf{B} \leftarrow \mathbf{B}$

Saved$\mathbf{X}_B \leftarrow \mathbf{X}_B$

Define SCCUB-p-d-2

Flag = 0

it = 1

While true

If it > maxit

break

If it ≤ 30

p = 50

Else

p = 75

If Flag ≠ 0

Solve continuous relaxation of set-covering formulation of BPP by **algorithm** revised simplex method. Return **B** and $\mathbf{X}_B$ (**Inputs**: n′, **W**′, c).

Find basic patterns having values greater than 0.5.

m ← number of patterns having values greater than 0.5

If no basic pattern is found with value greater than 0.5

Add the pattern with the largest value to SCCUB-p-d-2.

Else

$e = \left\lfloor \frac{p}{100} \times m \right\rfloor$

Randomly choose e patterns from list of the patterns with values greater than 0.5 and eliminate them from the list.

Add all the remaining patterns from list of the patterns with values greater than 0.5 to SCCUB-p-d-2.

$\bar{U} \leftarrow$ Find the upper bound on number of the items of SCCUB-p-d-2 to be packed in OPT(SCCUB-p-d-2) – 1 bins.

If $\bar{U} \geq$ number of the items of SCCUB-p-d-2

$n' \leftarrow n$

$\mathbf{W}' \leftarrow \mathbf{W}$

$\mathbf{B} \leftarrow$ Saved$\mathbf{B}$

$\mathbf{X}_B \leftarrow$ Saved$\mathbf{X}_B$

$it = it + 1$

$Flag = 0$

Eliminate items of SCCUB-p-d-2 from $\mathbf{W}'$ and update n'.

If $n' = 0$

If OPT(SCCUB-p-d-2) < OPT(SCCUB-p-d)

SCCUB-p-d $\leftarrow$ SCCUB-p-d-2

If $it = 30$ **or** it=50

GAP $\longleftarrow$ OPT(SCCUB-p-d) – ⌈OPTCG⌉

If GAP < 1

SCCUB-p-d is the optimal solution for BPP.

Break

$n' \leftarrow n$

$\mathbf{W}' \leftarrow \mathbf{W}$

$\mathbf{B} \leftarrow$ Saved$\mathbf{B}$

$\mathbf{X}_B \leftarrow$ Saved$\mathbf{X}_B$

$it = it + 1$

$Flag = 0$

$Flag = 1$

End of **while** loop

Step 7. Outputs are SCCUB-p-d and GAP.

6.3 Computational results

In this section, the results of running SCCUB-p-d procedure for u120, u250, u500, t60, t120, t249, t501, and Hard28 class of instances are presented. We compare our results with results retrieved from [27,33,35,36,37].

Table (6.2) displays comparison of the upper bounds obtained by applying SCCUB-p-d procedure with those derived by using state-of-the-art methods. In this table, total number of the instances in each class, and number of the instances in each class for which the methods fail to provide the optimal solution are reported. As observed, SCCUB-p-d dominates all the other methods for the given instances. More specifically, SCCUB-p-d procedure reduces number of the unsolved instances for class Hard28 from 12 to 4, which testifies its high performance.

Another superiority of this upper bounding technique over the other methods is that it provides the gap between upper and lower bound of the problem. Maximum of the gap between upper and lower bounds of the instances in each class is also reported in table (6.2) and is denoted by GAP. Entities of the column GAP in table (6.2) come with two associated numbers. For instance, X(Y) means that for Y instances of a specific class, maximum gap of X is derived.

As observed from this table, optimality of the upper bounds derived by using SCCUB-p-d procedure is proved for all the instances of u120, u250, u500, t60, t120, t249, and t501. Also, for 19 instances of Hard28 class, optimality of the solutions is proved, and for the other 9 instances, maximum gap of 1 between upper and lower bound of each instance is obtained. It is worth mentioning that 5 instances out of the ones for which the optimality is not proved, hold MIRUP. That is, the gap between the upper and lower bounds of these instances is larger than 1. Therefore, the only way to prove optimality of these instances is to perform a branch-and-price procedure with a termination condition of Maximum Nodes Explored. The typical termination condition of branch-and-price algorithms which relies on improvement of the gap between upper and lower bound of the problem is completely ineffective for these instances since it is not possible to close the gap anymore.

However, for the remaining unsolved instances (4 instances) of Hard28 class, an appropriately designed branch-and-price could be invoked to find the optimal solutions. Developing such a branch-and-price procedure is concern of the next chapter.

Besides, average computational time (in seconds) of the SCCUB-p-d procedure is compared with other methods in table (6.3). This table also contains information about the processors by which each of the methods was tested for the benchmark instances.

Table 6.2 Comparison of performance of SCCUB-p-d with state-of-the art methods

Class	Instance	WA	HGGA	Pert-SAWMBS	HI_BP	GGA-CGT	SCCUB-p-d	GAP
	Total	Opt not found	Opt not found	Opt not found	Opt not found	Opt not found	Opt not found	
U120	20	2	2	0	0	0	0	0(20)
U250	20	4	3	1	0	0	0	0(20)
U500	20	2	0	0	0	0	0	0(20)
t60	20	20	2	0	0	0	0	0(20)
t120	20	20	0	0	0	0	0	0(20)
t249	20	20	0	0	0	0	0	0(20)
t501	20	20	0	0	0	0	0	0(20)
Hard28	28	-	-	23	23	12	4	1(9), 0(19)

Table 6.3 Comparison of computational time of SCCUB-p-d with state-of-the art methods

Class	Instance	WA	HGGA	Pert-SAWMBS	HI_BP	GGA-CGT	SCCUB-p-d
	Total	t_{avg}	t_{avg}	t_{avg}	t_{avg}	t_{avg}	t_{avg}
U120	20	0.01	381	0.00	0.00	0.00	0.89
U250	20	0.05	1337	0.01	0.12	0.30	9.45
U500	20	0.21	1015	0.00	0.00	0.16	219.51
t60	20	0.01	47	0.00	0.37	0.05	1.23
t120	20	0.02	79	0.00	0.85	0.15	33.11
t249	20	0.11	728	0.00	0.22	0.39	186.66
t501	20	0.48	1663	0.01	2.49	1.30	2153.60
Hard28	28	-	-	0.24	-	2.40	156.97
Processor		Core2 2.33GHz	R4000 50GHz	Core2 2.33GHz	P4 1.7GHz	Core2 1.86GHz	Core3 2.10GHz

Chapter 7

A branch-and-price procedure for BPP with a generic branching strategy

This chapter presents a branch-and-price procedure for solving BPP where a *decrement constraint* is used in its pricing sub-problem to address the issue of having forbidden patterns in the search space of the one-dimensional knapsack problem. Also, a generic branching strategy, inspired by results from previous chapters is presented for this branch-and-price procedure.

7.1 Pricing sub-problem of column generation

As mentioned earlier, the main concern in developing branch-and-price procedures is the difficulty in the way, the issue of having forbidden patterns in the search space of the pricing sub-problem is resolved. Our approach in solving the polluted pricing sub-problem of column generation is simple and straightforward. Whenever one of the forbidden patterns is encountered while proceeding with column generation, one has to solve a two-dimensional

knapsack problem to derive the *next feasible solution* to the pricing sub-problem of column generation.

Recall that the one-dimensional knapsack problem presented in model (2.8) has the following structure:

$$\begin{aligned} &\max \sum_{i=1}^{n} \pi_i \, x_i \\ &\textstyle\sum_{i=1}^{n} w_i x_i \leq c \\ &x_i \in \{0,1\} \qquad\qquad i = 1, \dots, n \end{aligned}$$

where the vector $\mathbf{\Pi} = (\pi_1, \pi_2, \dots, \pi_n)$ represents shadow prices and $\mathbf{X} = (x_1, x_2, \dots, x_n)$, the solution to the one-dimensional knapsack problem.

Now, let us assume that at the k^{th} iteration of column generation, a pattern is generated by solving the above model and that happens to be one of the forbidden patterns. Also, let us name the objective value of the generated pattern, z_k, where z_k is derived as follows:

$$z_k = (\mathbf{\Pi})(\mathbf{X}).$$

Then, the next feasible solution to the one-dimensional knapsack problem could be obtained by solving the following model which reads:

$$\begin{aligned} &\max \sum_{i=1}^{n} \pi_i \, x_i \\ &\textstyle\sum_{i=1}^{n} w_i x_i \leq c \\ &\textstyle\sum_{i=1}^{n} \pi_i \, x_i \leq \; z_k - D \\ &x_i \in \{0,1\} \qquad\qquad i = 1, \dots, n \end{aligned} \tag{7.1}$$

where *D* denotes the value of the decrement. This value could be considered as being the minimum value of shadow prices. The second constraint in model (7.1) ensures that the generated pattern will have an objective value less than z_k. In other words, this approach guarantees that the generated pattern will have a different composition of items than the ones found in the forbidden pattern. It is noted that one could continue using the same approach in case; the newly generated pattern is still one of the forbidden patterns.

An important aspect of model (7.1) that affects the efficiency of the column generation process in a direct way is the value which is assigned to the decrement. For large values of

D, there might be a possibility of skipping some of the high-quality solutions of one-dimensional knapsack problem. Additionally, the worst case scenario is that where the pattern generated by this model is responsible for termination of column generation process. In such a case, since model (7.1) does not guarantee achievement of the *second best solution* of the knapsack problem, column generation might get terminated prematurely. That is, there exist some patterns with positive reduced costs that have been omitted from being added to the master problem, though the column generation process is still potent. To alleviate this problem, we could make use of a *minimal decrement*, *MD*, in the second constraint of mode (7.1). The new model is derived as follows:

$$\begin{aligned} &\max \sum_{i=1}^{n} \pi_i x_i \\ &\textstyle\sum_{i=1}^{n} w_i x_i \leq c \\ &\textstyle\sum_{i=1}^{n} \pi_i x_i \leq z_k - MD \\ &x_i \in \{0,1\} \qquad\qquad i = 1, \dots, n \end{aligned} \tag{7.2}$$

Even though model (7.2) generates higher quality patterns compared to model (7.1), it still does not guarantee achievement of the second best solution for the one-dimensional knapsack problem, and the immature termination of column generation still remains a concern. At this point, if the pattern responsible for terminating column generation is the very same pattern generated by model (7.2), the corresponding node of the branch-and-bound tree should not be pruned by bounds. For, the value derived at the end of the column generation process as, being the lower bound of the node, is not representing the true lower bound for it. In fact, we could not even call such a value a lower bound on the node since the term "bound" is only allowed to be employed if the set-covering formulation of BPP for a specific node is solved to optimality, which obviously is not the case in immature termination of column generation.

We mentioned earlier that model (7.2) generates better patterns in quality compared to the ones generated by model (7.1), however, determination of the minimal decrement value itself is a daunting procedure. This is because, different combinations of shadow prices should be considered to find the true value of *MD*. To keep the computational efforts at a minimum level, we could employ model (7.1) in our branch-and-bound procedure, with the

difference of a small numeric being used as D instead of considering it to be the minimum value of shadow prices. Our experiments reveal that the model (7.1), with $D = 10^{-7}$ generates relatively high-quality patterns to be passed on, to the master problem.

7.2 Branching and search strategies

As to the branching and search strategies, we are inspired by the results obtained in the previous chapters by employing SCCUB-p procedure. On the one hand, SCCUB-0, where all the patterns having values greater than 0.5 are included in the solution, reduces the dimension of the problem significantly and an integer solution to the problem is found quickly. On the other hand, optimal solution is obtained for the majority of the BPP instances by employing more time-consuming procedures like SCCUB-75, where only %25 of the patterns having values greater than 0.5 are included in the solution. The mentioned random processes are arguably amenable to deterministic versions of them, which subsequently, we propose as novel branching and search strategies.

Following the termination of column generation at the root node of the branch-and-bound tree, an initial *core* of patterns is created and that contains the most promising scenarios of the combination of the basic patterns that could lead to the optimal solution of BPP. The first element of the initial core is constituted of %25 of the best patterns having values greater than 0.5. The second scenario added to the initial core, would be that where one excludes the worst pattern out of the elite group (%25 of the best patterns having values greater than 0.5) and considers the remaining patterns as being the second element of the initial core. This elimination process continues until the pattern with the highest value is added as a single pattern to the initial core. At this point, to ensure the capability of the branch-and-bound procedure in exploring the whole search space of the problem, a scenario will be added to the core in which, the pattern with the highest value is considered as a forbidden pattern.

To understand how the initial core is constructed, let us assume that the following basic patterns having values greater than 0.5, are obtained through, solving the continuous relaxation of the set-covering formulation of BPP at the root node.

Pattern	A	B	C	D	E	F	G	H	I	J
Value	1	0.9	0.85	0.85	0.7	0.65	0.65	0.6	0.55	0.55

Firstly, the number of the patterns to be eliminated is computed as:

$$\mathrm{e} = \left\lfloor \frac{75}{100} \times 10 \right\rfloor = 7$$

So that patterns A, B, and C are left after the elimination process.

The first element of the initial core will then be constructed as:

$$Core^{I} = \{ \begin{matrix} MUP_1 = \{A, B, C\} \\ FP_1 = \emptyset \end{matrix} \}$$

where MUP denotes the *Must Used Patterns*, FP the forbidden ones, and $Core^{I}$ the initial core.

The second scenario will be eliminating the last pattern of the MUP_1 and considering the rest of the patterns as MUP_2 and adding it to the initial core. The core will then become:

$$Core^{I} = \{ \begin{Bmatrix} MUP_1 = \{A, B, C\} \\ FP_1 = \emptyset \end{Bmatrix}, \begin{Bmatrix} MUP_2 = \{A, B\} \\ FP_2 = \emptyset \end{Bmatrix} \}.$$

Following the same method, the core will be expanded as:

$$Core^{I} = \{ \begin{Bmatrix} MUP_1 = \{A, B, C\} \\ FP_1 = \emptyset \end{Bmatrix}, \begin{Bmatrix} MUP_2 = \{A, B\} \\ FP_2 = \emptyset \end{Bmatrix}, \begin{Bmatrix} MUP_3 = \{A\} \\ FP_3 = \emptyset \end{Bmatrix} \}.$$

Since the last pattern added to the core is a single one, its forbidden version should also be considered, and this will complete the process of building the initial core, and the resulting initial core will be:

$$Core^{I} = \{ \begin{Bmatrix} MUP_1 = \{A, B, C\} \\ FP_1 = \emptyset \end{Bmatrix}, \begin{Bmatrix} MUP_2 = \{A, B\} \\ FP_2 = \emptyset \end{Bmatrix}, \begin{Bmatrix} MUP_3 = \{A\} \\ FP_3 = \emptyset \end{Bmatrix}, \begin{Bmatrix} MUP_4 = \emptyset \\ FP_4 = \{A\} \end{Bmatrix} \}.$$

Following the completion of the initial core, each element of the core starting from the first one will be considered as the branching constraints, and in all of the subsequent nodes, the same process will be employed to expand the core until all the nodes are explored and

pruned by either integrality or infeasibility or bound. We call this strategy of searching the solution space, a *batch diving* strategy.

However, for more difficult instances of BPP, it might be possible that the initial core constructed this way, is not the best combination of patterns for finding the optimal solution in the shortest amount of time.

To this end, if the number of the nodes explored by using the initial core, exceeds a specific threshold, a different core will be fed into the branch-and-bound procedure.

For the new core, all the patterns having values greater than 0.5, will be considered. But, only every other pattern starting from the pattern with the highest value will be used to build the core.

The core established by using the method described hereby, for our example, will be derived as being:

$$Core' = \{\begin{Bmatrix} MUP_1 = \{A,C,E,G,I\} \\ FP_1 = \emptyset \end{Bmatrix}, \begin{Bmatrix} MUP_2 = \{A,C,E,G\} \\ FP_2 = \emptyset \end{Bmatrix}, \begin{Bmatrix} MUP_3 = \{A,C,E\} \\ FP_3 = \emptyset \end{Bmatrix}, \begin{Bmatrix} MUP_4 = \{A,C\} \\ FP_4 = \emptyset \end{Bmatrix}\}.$$

Again, branch-and-bound procedure will start searching for the optimal solution, but this time, special focus is geared toward branching on patterns of the $Core'$. If the branch-and-bound procedure fails to find the optimal solution after exhausting a limit number of nodes, another core will be inputted.

This new core is built by considering all the patterns having values greater than 0.5, but only every other third pattern will be selected for inclusion into the core. For the current example, $Core''$ could be viewed as being:

$$Core'' = \{\begin{Bmatrix} MUP_1 = \{A,D,G,J\} \\ FP_1 = \emptyset \end{Bmatrix}, \begin{Bmatrix} MUP_2 = \{A,D,G\} \\ FP_2 = \emptyset \end{Bmatrix}, \begin{Bmatrix} MUP_3 = \{A,D\} \\ FP_3 = \emptyset \end{Bmatrix}\}.$$

Similarly, $Core'''$ and $Core''''$ will be constructed by considering every other fourth and every other fifth patterns, respectively.

$$Core''' = \{\begin{Bmatrix} MUP_1 = \{A,E,I\} \\ FP_1 = \emptyset \end{Bmatrix}, \begin{Bmatrix} MUP_2 = \{A,E\} \\ FP_2 = \emptyset \end{Bmatrix}\}.$$

$$Core'''' = \{\begin{Bmatrix} MUP_1 = \{A,F\} \\ FP_1 = \emptyset \end{Bmatrix}\}.$$

Such choices for the cores of the patterns in the branch-and-bound procedure, allow for the exploration of various and at the same time, promising regions of the solution space.

7.3 Pruning the nodes by bound

There is a common trend among researchers when it comes to pruning the nodes by bounds in the branch-and-bound procedures, and the common approach has been widely presented in the operational research text books. Simply put, a specific node is pruned by bound if the lower bound derived for that very node, exceeds the global upper bound of BPP. Now, any of the nodes in the branch-and-bound tree will be pruned by bound if the following condition holds true:

$$LB(.) - GlobUB > 0$$

where $LB(.)$ denotes the lower bound derived for a specific node, and $GlobUB$ is, the global upper bound of BPP.

However, our experiments on BPP instances revealed that, the pruning by bound rule is effective inasmuch as the obtained global upper bound is tight. In other words, even when a relatively good upper bound with only one extra bin compared to the optimal solution is employed, this leads the branch-and-price method to a catastrophic behavior. Because, for only a few of the nodes, the lower bounds of the nodes, do exceed the global upper bound mentioned above.

Therefore, in order to increase the efficiency of the procedure, a tight upper bound should be employed. Practically, this could be carried out through use of the information provided by the dual version of the BPP we presented in the previous chapter. More precisely, let us assume that given n items, the upper bound obtained on the objective value of the BPP is m bins. Then, an upper bound on the number of items to be packed in $(m-1)$ bins is computed. If this upper bound is less than n, we would know that $(m-1)$ could not be a candidate for being an optimal number of bins, and the branch-and-price procedure will use m bins as the global upper bound to prune the nodes. However, if the upper bound on the number of items destined to be packed in $(m-1)$ bins was not less than n, then $(m-1)$

bins will be regarded as the global upper bound of the procedure to be used in pruning the nodes.

The details of such a pruning scheme and the generic branching and search strategies proposed could be found in table (7.1) where we present the pseudo-code for branch-and-price procedure.

Table 7.1 Pseudo-code for procedure branch-and-price

Procedure branch-and-price

Input: number of items (n), weights of the items ($\mathbf{W}_{1\times n}$), bin capacity (c), maxnodes = 10000, maxtime = 7200 (s)

Output: GlobUB and GlobUBSol

Step 1. Find the upper bound for BPP by performing **procedure** SCCUB. Return SCCUB, OPT(SCCUB), $\mathbf{B}$, $\mathbf{X}_B$, and OPTCG.

Step 2. GlobUB=OPT(SCCUB)

GlobUBSol=SCCUB

Step 3. Diff = GlobUBSol – OPTCG

Step 5. If Diff < 1

GlobUBSol is optimal. Return GlobUBSol and GlobUB.

Terminate **procedure** branch-and-price.

Step 6. Compute $\overline{U}^{n}_{GlobUBSol-1}$

Step 7. If $\overline{U}^{n}_{GlobUBSol-1} \geq n$

CandidateOPT $\leftarrow$ $\overline{U}^{n}_{GlobUBSol-1}$

Else

CandidateOPT $\leftarrow$ GlobUB

Step 8. Find $\mathbf{X}^{i}_{B} > 0.5$ for i=1,...,n and sort $\{\mathbf{X}^{i}_{B} : \mathbf{X}^{i}_{B} > 0.5$ for i=1,...,n$\}$ in descending order.

Step 9. Sort $\{\mathbf{B}^{i} : \mathbf{X}^{i}_{B} > 0.5$ for i=1,...,n$\}$ according to $\{\mathbf{X}^{i}_{B} : \mathbf{X}^{i}_{B} > 0.5$ for i=1,...,n$\}$.

Step 10. $\mathbf{B} \leftarrow \{\mathbf{B}^{i} : \mathbf{X}^{i}_{B} > 0.5$ for i=1,...,n$\}$

Step 11. Define Core, Core^{I}, MUP, FP

Step 12. For i = $\lceil 0.25 \times n(\mathbf{B}) \rceil$ to 1

MUP = $\mathbf{B}^{1 \text{ to } i}$

FP = $\emptyset$

Add (MUP,FP) to Core^{I}

MUP = $\emptyset$

FP = $\mathbf{B}^{1}$

Add (MUP,FP) to $Core^I$

Step 13. Counter = 0

Step 14. While true

If Counter = 0

Add $Core^I$ to the beginning of Core

ElseIf Counter = 500

Add $Core^1$ to the beginning of Core

ElseIf Counter = 1000

Add $Core^2$ to the beginning of Core

ElseIf Counter = 1500

Add $Core^3$ to the beginning of Core

ElseIf Counter = 2000

Add $Core^4$ to the beginning of Core

If Counter ≥ maxnodes or time ≥ maxtime or n(Core) = 0

Break

Parent ← Choose the first element of Core

Perform **Algorithm** revised simplex method for Parent. Return **B**, $\mathbf{X}_B$, and OPTCG.

If $\mathbf{X}_B^i$ integer ∀i=1,…,n

If OPTCG = CandidateOPT

GlobUB = OPTCG

GlobUBSol = **B**

Break

Eliminate the first element of the core

Continue

If OPTCG > CandidateOPT and the pattern responsible for terminating the column generation is not generated by two-dimensional knapsack problem

Eliminate the first element of the core

Continue

Eliminate the first element of the core

/ Batch diving

Find $\mathbf{X}_B^i > 0.5$ for i=1,…,n and sort $\{\mathbf{X}_B^i : \mathbf{X}_B^i > 0.5$ for i=1,…,n$\}$ in descending order.

Sort $\{\mathbf{B}^i : \mathbf{X}_B^i > 0.5$ for i=1,…,n$\}$ according to $\{\mathbf{X}_B^i : \mathbf{X}_B^i > 0.5$ for i=1,…,n$\}$.

$\mathbf{B} \leftarrow \{\mathbf{B}^i : \mathbf{X}_B^i > 0.5 \text{ for i=1,...,n}\}$

For i = ⌈0.25 × n(**B**) ⌉ to 1

MUP = $\mathbf{B}^{1 \text{ to i}}$

FP = ∅

Add (MUP,FP) to Core

If Counter = 0

Define Core^1

For i = n(**B**) to 1 with decrement = 2

MUP = $\mathbf{B}^{1 \text{ to i with increment=2}}$

FP = ∅

Add (MUP,FP) to Core^1

Define Core^2

For i = n(**B**) to 1 with decrement = 3

MUP = $\mathbf{B}^{1 \text{ to i with increment=3}}$

FP = ∅

Add (MUP,FP) to Core^2

Define Core^3

For i = n(**B**) to 1 with decrement = 4

MUP = $\mathbf{B}^{1 \text{ to i with increment=4}}$

FP = ∅

Add (MUP,FP) to Core^3

Define Core^4

For i = n(**B**) to 1 with decrement = 5

MUP = $\mathbf{B}^{1 \text{ to i with increment=5}}$

FP = ∅

Add (MUP,FP) to Core^4

Counter = Counter +1

End of **while** loop

Step 15. Outputs are GlobUB and GlobUBSol.

7.4 Computational results

In this section, computational results of performing branch-and-price procedure for instances of u120, u250, u500, t60, t120, t249, t501, and Hard 28 are presented. To solve the two-dimensional knapsack problem presented in model (7.1), the method of Martello and Toth, developed in reference [51], was adapted.

As observed from tables (7.2) to (7.9), optimal solution for all of the instances of BPP is obtained by using branch-and-price procedure except two instances of the Hard28 class. The reason for failing to derive the optimal solution to BPP40 is that, our implementation of revised simplex method is not capable of solving the continuous relaxation of the set-covering formulation to optimality. More precisely, the known LP for this instance is 58.999, as derived in [15], whereas the LP provided by our implementation of revised simplex method provides the value 59.009. Therefore, the branch-and-price procedure gets terminated at the root node and no branching occurs to search for the optimal solution.

Another instance which remains unsolved by using our developed branch-and-price scheme is BPP60. The most plausible explanation for failure of the procedure to find the optimal solution to this instance is that, the different choices of cores of patterns, fed into the procedure were not sufficient to lead to obtaining the optimal solution for this instance.

Furthermore, the class of Hard28 contains 5 instances, BPP14, BPP359, BPP716, BPP119, and BPP175 which hold MIRUP. That is, the optimal solution is found at the root node whereas the procedure does not get terminated by bound since the difference of optimal value and LP relaxation of the problem is greater than one. Hence, one of the conditions like maximum of nodes explored or maximum time spent should be met to prove the optimality of the upper bound solution. For all of the MIRUP instances, the procedure got terminated by filling the latter termination condition which is maximum time spent.

All in all, satisfactory results are obtained by employing the proposed branch-and-price procedure to solve instances of BPP, and this procedure could be a better alternative to the state-of-the-art methods.

Table 7.2 Computational results of procedure branch-and-price for instances of u120

Instance	**IP**	**OPT**	**nod**	t_{opt}
U120_00	48	48	5	0.785
U120_01	49	49	5	0.988
U120_02	46	46	3	0.934
U120_03	49	49	4	1.058
U120_04	50	50	6	0.918
U120_05	48	48	3	0.911
U120_06	48	48	5	0.918
U120_07	49	49	4	0.746
U120_08	50	50	4	0.790
U120_09	46	46	4	0.990
U120_10	52	52	6	0.865
U120_11	49	49	4	0.832
U120_12	48	48	4	0.841
U120_13	49	49	4	1.030
U120_14	50	50	4	0.937
U120_15	48	48	5	1.013
U120_16	52	52	4	0.747
U120_17	52	52	3	0.739
U120_18	49	49	5	0.934
U120_19	49	49	3	0.959
Total	981	981	85	17.935
Average	49.05	49.05	4.25	0.896

Table 7.3 Computational results of procedure branch-and-price for instances of u250

Instance	**IP**	**OPT**	**nod**	t_{opt}
U250_00	99	99	6	9.504
U250_01	100	100	5	9.266
U250_02	102	102	5	10.485
U250_03	100	100	6	9.099
U250_04	101	101	4	9.380
U250_05	101	101	3	9.766
U250_06	102	102	6	9.579
U250_07	103	103	3	9.033
U250_08	105	105	3	7.842
U250_09	101	101	4	10.471
U250_10	105	105	5	9.682
U250_11	101	101	4	10.069
U250_12	105	105	3	8.691
U250_13	103	103	6	7.600
U250_14	100	100	5	10.543
U250_15	105	105	3	8.212
U250_16	97	97	5	11.468
U250_17	100	100	6	9.365
U250_18	100	100	4	10.126
U250_19	102	102	6	8.975
Total	2032	2032	92	189.156
Average	101.60	101.60	4.6	9.457

Table 7.4 Computational results of procedure branch-and-price for instances of u500

Instance	IP	OPT	nod	t_{opt}
U500_00	198	198	5	210.596
U500_01	201	201	5	208.834
U500_02	202	202	7	197.652
U500_03	204	204	5	210.911
U500_04	206	206	5	176.158
U500_05	206	206	6	203.907
U500_06	207	207	4	150.192
U500_07	204	204	4	218.033
U500_08	196	196	7	201.654
U500_09	202	202	6	215.984
U500_10	200	200	6	201.018
U500_11	200	200	6	224.369
U500_12	199	199	7	229.263
U500_13	196	196	6	224.288
U500_14	204	204	6	239.788
U500_15	201	201	6	242.694
U500_16	202	202	8	232.780
U500_17	198	198	6	305.825
U500_18	202	202	5	212.095
U500_19	196	196	7	284.212
Total	4024	4024	117	4390.253
Average	201.2	201.2	5.85	219.512

Table 7.5 Computational results of procedure branch-and-price for instances of t60

Instance	IP	OPT	nod	t_{opt}
t60_00	20	20	1	0.215
t60_01	20	20	3	0.348
t60_02	20	20	3	0.276
t60_03	20	20	7	0.547
t60_04	20	20	9	0.451
t60_05	20	20	1	0.218
t60_06	20	20	17	0.805
t60_07	20	20	12	0.636
t60_08	20	20	5	0.381
t60_09	20	20	1	0.243
t60_10	20	20	38	1.891
t60_11	20	20	1	0.237
t60_12	20	20	5	0.297
t60_13	20	20	3	0.307
t60_14	20	20	1	0.241
t60_15	20	20	2	0.252
t60_16	20	20	1	0.268
t60_17	20	20	1	0.236
t60_18	20	20	1	0.221
t60_19	20	20	1	0.234
Total	400	400	113	8.304
Average	20	20	5.6	0.415

Table 7.6 Computational results of procedure branch-and-price for instances of t120

Instance	IP	OPT	nod	t_{opt}
t120_00	40	40	9	1.182
t120_01	40	40	14	1.494
t120_02	40	40	217	9.049
t120_03	40	40	8	1.074
t120_04	40	40	12	1.183
t120_05	40	40	6	0.881
t120_06	40	40	208	11.409
t120_07	40	40	7	0.921
t120_08	40	40	7	1.865
t120_09	40	40	15	1.551
t120_10	40	40	14	1.904
t120_11	40	40	95	6.146
t120_12	40	40	7	0.973
t120_13	40	40	6	0.964
t120_14	40	40	65	5.900
t120_15	40	40	77	3.974
t120_16	40	40	9	1.395
t120_17	40	40	61	3.847
t120_18	40	40	69	4.22
t120_19	40	40	17	1.590
Total	800	800	923	61.522
Average	40	40	46.1	3.076

Table 7.7 Computational results of procedure branch-and-price for instances of t249

Instance	**IP**	**OPT**	**nod**	t_{opt}
t249_00	83	83	91	9.108
t249_01	83	83	16	6.662
t249_02	83	83	117	10.676
t249_03	83	83	8	6.565
t249_04	83	83	48	8.511
t249_05	83	83	12	6.988
t249_06	83	83	10	7.270
t249_07	83	83	73	9.921
t249_08	83	83	11	6.868
t249_09	83	83	17	6.927
t249_10	83	83	14	7
t249_11	83	83	15	7.299
t249_12	83	83	81	8.627
t249_13	83	83	41	8.802
t249_14	83	83	11	7.395
t249_15	83	83	13	7.202
t249_16	83	83	11	7.050
t249_17	83	83	31	7.040
t249_18	83	83	12	6.972
t249_19	83	83	65	9.502
Total	1660	1660	697	156.385
Average	83	83	34.8	7.819

Table 7.8 Computational results of procedure branch-and-price for instances of t501

Instance	IP	OPT	nod	t_{opt}
t501_00	167	167	14	83.360
t501_01	167	167	26	84.004
t501_02	167	167	1304	138.896
t501_03	167	167	38	84.530
t501_04	167	167	15	75.016
t501_05	167	167	94	81.515
t501_06	167	167	37	80.585
t501_07	167	167	17	79.714
t501_08	167	167	731	124.688
t501_09	167	167	17	99.924
t501_10	167	167	101	102.307
t501_11	167	167	11	103.661
t501_12	167	167	15	91.061
t501_13	167	167	11	102.221
t501_14	167	167	166	111.395
t501_15	167	167	18	97.506
t501_16	167	167	13	98.645
t501_17	167	167	61	97.624
t501_18	167	167	63	106.466
t501_19	167	167	16	96.478
Total	3340	3340	2768	1939.596
Average	167	167	138.4	96.979

Table 7.9 Computational results of procedure branch-and-price for instances of Hard28

Instance	IP	OPT	nod	t_{opt}
BPP14	62	62	3519	7200
BPP832	60	60	171	66.970
BPP40	59	**60**	7	11.806
BPP360	62	62	1	9.823
BPP645	58	58	571	411.029
BPP742	64	64	124	50.031
BPP766	62	62	590	374.303
BPP60	63	**64**	2696	7200
BPP13	67	67	312	409.105
BPP195	64	64	12	52.879
BPP709	67	67	43	71.221
BPP785	68	68	15	43.116
BPP47	71	71	22	36.759
BPP181	72	72	4693	3982.410
BPP359	76	76	3993	7200
BPP485	71	71	49	52.510
BPP640	74	74	17	41.755
BPP716	76	76	2630	7200
BPP119	77	77	7734	7200
BPP144	73	73	531	370.963
BPP561	72	72	14	58.668
BPP781	71	71	16	116.238
BPP900	75	75	117	70.810
BPP175	84	84	3371	7200
BPP178	80	80	12	72.437
BPP419	80	80	362	286.721
BPP531	83	83	17	52.880
BPP814	81	81	15	51.458
Total	1972	1974	31654	49893.890
Average	70.42	70.50	1130.5	1781.925

Conclusion and future research

In this research, a branch-and-price procedure was proposed to solve instances of BPP to optimality. The proposed procedure was developed based upon our observations and investigations on the set-covering formulation of BPP and its different aspects. A few stochastic schemes were designed to investigate the structure of the basic solutions of continuous relaxation of the mentioned formulation. The results of performing these stochastic schemes on the benchmark instances of BPP confirmed that such solutions could be employed to derive tight upper bound solutions on BPP instances. Inspired by the results of stochastic schemes, a generic branching rule and a new search strategy named batch diving were proposed to be used in the branch-and-price procedure that enabled the procedure to solve majority of the benchmark instances of BPP to optimality.

Furthermore, a new approach in dealing with the forbidden patterns of the branch-and-bound tree was proposed which is different from the schemes developed in the literature and handles two-dimensional knapsack problems. Moreover, we showed, in this work, the relation between BPP and the maximum cardinality BPP, and that how the information provided by the latter could be used in solving procedure of the BPP.

Further research could be done to thoroughly investigate the relation between BPP and its dual version problem, in order to increase the efficiency of the branch-and-price procedure to solve BPP instances. Moreover, generalizations of the proposed branching and search strategies require more research in order to find the optimal solution to the instances that remained unsolved in this thesis.

Bibliography

1- Nemhauser G.L. and L.A. Wolsey, Integer and Combinatorial Optimization, Wiley, 1998.

2- Nemhauser G.L., Column generation for linear and integer programming, *Documenta Mathematica – Extra Volume ISMP* 65-73, 2012.

3- Vance P.H., C. Barnhart, E.L. Johnson and G.L. Nemhauser, Solving binary cutting stock problems by column generation and branch-and- bound, *Computational Optimization and Applications* **3(2)**:111-130, 1994.

4- Dantzig G.B. and P. Wolfe, Decomposition principle for linear programs, *Operations Research* **8(1)**:101-111, 1960.

5- Ford L.R. and D.R. Fulkerson, A suggested computation for maximal multicommodity network flows, *Management Science* **5**:97-101, 1958.

6- Vance P.H., Branch-and-price algorithms for the one-dimensional cutting stock problem, *Computational Optimization and Applications* **9**:211-228, 1998.

7- Gilmore P.C. and R.E. Gomory, A linear programming approach to the cutting stock problem, *Operations Research* **9(6)**:849-859, 1961.

8- Belov G. and G. Scheithauer, A branch-and-cut-and-price algorithm for one-dimensional stock cutting and two-dimensional two-stage cutting, *European Journal of Operational Research* **171(1)**: 85–406, 2006.

9- Savelsbergh, M.W.P., A branch-and-price algorithm for the generalized assignment problem, *Operations Research* **45**:831-841, 1997.

10- Elble J.M. and N.V. Sahinidis, A review of the LU update in the simplex algorithm, *International Journal of Mathematics in Operational Research* **4(4)**, 366–399, 2012.

11- Baum S. and L.E. Trotter, Jr. Integer rounding for polymatroid and branching optimization problems, *SIAM Journal on Algebric and Discrete Methods* **2(4)**:416-425, 1981.

12- Marcotte O., The cutting stock problem and integer rounding, *Mathematical Programming* **33**:82-92, 1985.

13- Marcotte O., An instance of the cutting stock problem for which the rounding property does not hold, *Operations Research Letters* **4(5)**:239-243, 1986.

14- Scheithauer G. and J. Terno, The modified integer round-up property of the one-dimensional cutting stock problem, *European Journal of Operational Research* **84(3)**:562–571, 1995.

15- Belov G. and G. Scheithauer, A cutting plane algorithm for the one-dimensional cutting stock problem with multiple stock lengths, *European Journal of Operational Research* **141**:274–294, 2002.

16- Belov G. and G. Scheithauer, Setup and open stacks minimization in one-dimensional stock cutting, Technical Report, Dresden University, 2003.

17- Bansal N., A. Caprara, and M. Sviridenko, A new approximation method for set covering problems, with applications to multidimensional bin packing, *SIAM Journal on Computing* **39(4)**:1256–1278, 2009.

18- Leão A.A.S., L.H. Cherri and M.N. Arenales, Determining the k-best solutions of knapsack problem, *Computers and Operations Research* **49**:71-82, 2014.

19- Gilmore, P.C. and R.E Gomory, A linear-programming approach to the cutting stock problem-part II, *Operations Research* **11(6)**:863:888, 1963.

20- Sarin S.C., Y. Wang and D.B. Chang, A schedule algebra based approach to determine the K-best solutions of a knapsack problem with a single constraint, *Lecture Notes in Computer Science* **3521**:440-449, 2005.

21- Giffler B. Schedule algebra: A progress report, *Naval Research Logistics Quarterly* **15(2)**:255-280, 1968.

22- Ryan D.M. and B.A. Foster, An integer programming approach to scheduling, In: Wren, A. (ed.) Computer Scheduling of Public Transport Urban Passenger Vehicle and Crew Scheduling, North Holland, Amsterdam, pages 269–280, 1981.

23- Barnhart C., E.L. Johnson, G.L. Nemhauser, M.W.P. Savelsbergh and P.H. Vance, Branch-and-price: Column generation for solving huge integer programs, *Operations Research* **46(3)**:316-329, 1998.

24- Atamtürk A. and M.W.P. Savelsbergh, Integer-programming software systems, *Annals of Operations Research* **140**:67-124, 2005.

25- Baldi M.M., T.G. Crainic, G. Perboli and R. Tade, The generalized bin packing problem, *Transportation Research* **48(6)**:1205-1220, 2012.

26- Williamson D.P. and D.B. Shmoys, The Design of Approximation Algorithms, Cambridge University Press, 2011.

27- Quiroz-Castellanos M., L. Cruz-Reyes, J. Torres-Jimenez. C. Gmez, H.J.F. Huacuja and A.C.F. Alvim, A grouping genetic algorithm with controlled gene transmission for the bin packing problem, *Computers and Operations Research* **55**:52-64, 2015.

28- Coffman E.G. Jr., J. Csirik, G. Galambos, S. Martello and D. Vigo, Bin packing approximation algorithms: Survey and classification, In: P.M. Pardalos, D.Z. Du, R.L. Graham, Handbook of Combinatorial Optimization, Springer, New York, 2013.

29- Dósa G. and J. Sgall, First fit bin packing: A tight analysis, In *Leibniz International Proceedings in Informatics* **20**:538-549, 2013.

30- Dósa G. and J. Sgall, Optimal analysis of best fit bin packing, Proc. of the 31st Int. Colloquium on Automata, Languages, and Programming (ICALP), *Lecture Notes in Computer Science* **8572**: 429-441, Springer, 2014.

31- Simchi-Levi D., New worst-case results for the bin-packing problem, *Naval Research Logistics* **41(4)**:579-585, 1994.

32- Scholl A., R. Klein and C. Jürgens, Bison: a fast hybrid procedure for exactly solving the one-dimensional bin packing problem, *Computers and Operations Research* **24(7)**:627-645, 1997.

33- Alvim A.C.F., C.C. Ribeiro, F. Glover and D.J. Aloise, A hybrid improvement heuristic for the one-dimensional bin packing problem, *Journal of Heuristics*, **10(2)**:205-229, 2004.

34- Martello S., and P. Toth, Lower bounds and reduction procedures for the bin packing problem, *Discrete Applied Mathematics* **28(1)**:59-70, 1990.

35- Fleszar K., and C. Charalambous, Average-weight-controlled bin-oriented heuristics for the one-dimensional bin-packing problem, *European Journal of Operational Research* **210(2)**:176-184, 2011.

36- Falkenauer E., A hybrid grouping genetic algorithm for bin packing, *Journal of Heuristics* **2(1)**:5-30, 1996.

37- Loh K.H., B. Golden and E. Wasil, Solving the one-dimensional bin packing problem with a weight annealing heuristic, *Computers and Operations Research* **35**:2283-2291, 2008.

38- Martello S., D. Pisinger and T. Toth, Dynamic programming and strong bounds for the 0-1 knapsack problem, *Management Science* **45(3)**:414-424, 1999.

39- Martello S. and P. Toth, Knapsack Problems: Algorithms and Computer Implementations, John Wiley & Sons, Chichester, 1990.

40- Martello S. and P. Toth, Upper bounds and algorithms for hard 0-1 knapsack problems, *Operations Research* **45**:768-778, 1997.

41- Pisinger D., A minimal algorithm for the 0-1 knapsack problem, *Operations Research*, **45**:758-767, 1997.

47- Schwerin P. and G. Wäscher, The bin-packing problem: A problem generator and numerical experiments with FFD packing and MTP, *International transactions in operational research* **4 (5-6)**:377-389, 1997.

48- Wäscher G. and T. Gau, Heuristics for the integer one-dimensional cutting stock problem: A computational study, *Operations-Research-Spektrum* **18(3)**:131–144, 1996.

49- Pisinger D., C program code for dynamic programming and strong bounds for the 0-1 knapsack problem, retrieved from http://www.diku.dk/~pisinger/codes.html.

50- Labbé M., G. Laporte and S. Martello, Upper bounds and algorithms for the maximum cardinality bin packing problem, *European Journal of Operational Research* **143(3)**:490-498, 2003.

51- Martello S., P. Toth, An exact algorithm for the two-constraint 0-1 knapsack problem, *Operations Research* **51**:826-835, 2003.