OPEN-END BIN PACKING PROBLEM WITH CONFLICTS

A Thesis

by

Ece Nur Balık

Submitted to the
Graduate School of Sciences and Engineering
In Partial Fulfillment of the Requirements for
the Degree of

Master of Science

in the
Department of Industrial Engineering

Özyeğin University
May 2022

OPEN-END BIN PACKING PROBLEM WITH CONFLICTS

Approved by:

Associate Professor Ali Ekici, Advisor
Department of Industrial Engineering
Özyeğin University

Associate Professor Okan Örsan Özener
Department of Industrial Engineering
Özyeğin University

Assistant Professor Salih Tekin
Department of Industrial Engineering
TOBB University of Economics & Technology

Date Approved: 23 May 2022

To my beloved family and friends...

ABSTRACT

In this thesis study, we focus on a new variant of the famous *Bin Packing Problem* (BPP) called the *Open-End Bin Packing Problem with Conflicts* (OEBPPC) which combines the *Open-End Bin Packing Problem* (OEBPP) and the *Bin Packing Problem with Conflicts* (BPPC). In OEBPPC, the aim is to pack a set of items into the least number of bins. However, the bin capacity is allowed to be exceeded only by the last item packed into the bin, and there exist conflicts between some item pairs; they cannot be packed into the same bin. We introduce a mathematical formulation and propose lower bounding procedures for our problem. We propose a metaheuristic algorithm, namely *Variable Neighborhood Search* (VNS), to approach the optimal solution through systematic changes and improvements in the solution. We generate different sets of instances by adapting some instances from the literature to our problem. We compare the performance of our metaheuristic algorithm both against the best lower bound and other algorithms we adapted from the literature as benchmark algorithms. We observe that our proposed metaheuristic outperforms the best benchmark algorithm in 74% of the instances with varying features.

ÖZETÇE

Bu tez çalışmasında, Açık Uçlu Kutulama Problemi (AUKP) ve Çatışmalarla Kutulama Problemi (ÇKP)'ni birleştiren, ünlü Kutulama Problemi (KP)'nin yeni bir çeşidi olan Çatışmalarla Açık Uçlu Kutulama Problemi (ÇAUKP)'ne odaklanıyoruz. ÇAUKP'de amaç, bir dizi eşyayı en az sayıda kutuya paketlemektir. Ancak, kutu kapasitesinin yalnızca kutuya paketlenen son eşya tarafından aşılmasına izin verilir ve bazı eşya çiftleri arasında çelişkiler vardır; bunlar aynı kutuya paketlenemezler. Problemimiz için matematiksel bir formülasyon sunuyoruz ve alt sınır bulma yöntemleri öneriyoruz. Çözümdeki sistematik değişiklikler ve iyileştirmelerle en iyi çözüme yaklaşmak için Değişken Komşuluk Arama (DKA) adlı metasezgisel bir algoritma öneriyoruz. Literatürdeki bazı örnekleri problemimize uyarlayarak farklı örnek kümeleri oluşturuyoruz. Metasezgisel algoritmamızın performansını hem en iyi alt sınırla hem de literatürden kıyaslama algoritmaları olarak uyarladığımız diğer algoritmalarla karşılaştırıyoruz. Önerilen metasezgiselimizin, değişen özelliklere sahip örneklerin %74'ünde en iyi kıyaslama algoritmasından daha iyi performans gösterdiği gözlemlenmektedir.

ACKNOWLEDGEMENTS

I would like to express my great appreciation to my advisor, Dr. Ali Ekici, who consistently guided me in this thesis study and supported me by sharing his experiences and valuable pieces of advice throughout my Master's studies. I find myself lucky to have studied with him. I also thank my professors whom I have learned a lot from and improved myself through their lectures and more, especially Dr. Burcu Balçık, Dr. Okan Örsan Özener, Dr. Elvin Çoban and Dr. Enis Kayış.

I am grateful for my dearest mom and dad, who are always there for me. I would like to thank them for their support, love, and encouragement even after they come home spending a long day at work. I want to thank my dear brother for showing that there is always something to laugh about. Finally, I want to thank my grandmother for raising me and my aunt who is with me even from miles away.

I am genuinely thankful for my close friends for being important parts of my life, motivating me, and believing in me. Finally, I would like to thank my online officemates for their friendships and for making the zoom meetings better.

TABLE OF CONTENTS

DEDICATION . . . . . **iii**

ABSTRACT . . . . . **iv**

ÖZETÇE . . . . . **v**

ACKNOWLEDGEMENTS . . . . . **vi**

LIST OF TABLES . . . . . **viii**

LIST OF FIGURES . . . . . **ix**

I INTRODUCTION **1**

II LITERATURE REVIEW **4**

III PROBLEM DEFINITION AND FORMULATION **7**

IV LOWER BOUNDS **11**

V PROPOSED SOLUTION APPROACH **15**

5.1 Overall VNS Framework 15

5.2 Initial Solution Construction 17

5.3 Shaking 17

5.3.1 First Neighborhood Operator (N_1) 17

5.3.2 Second Neighborhood Operator (N_2) 19

5.3.3 Third Neighborhood Operator (N_3) 20

5.4 Repair Operator 21

5.5 Local Search 22

VI COMPUTATIONAL STUDY **25**

6.1 Instance generation 25

6.2 Performance of the proposed lower bounds 25

6.3 Performance of the proposed solution approach 29

VII CONCLUSION **37**

REFERENCES **38**

VITA **41**

LIST OF TABLES

1 Performance of the proposed lower bounds. 28

2 Summary of the performance of the proposed lower bounds 29

3 Comparison of VNS against the benchmark algorithms. 32

4 Summary of the results according to conflict graph density. 33

5 Summary of the results according to instance size. 33

6 Average CPU time (in seconds) of all algorithms. 35

7 Summary of the average CPU time (in seconds) of all algorithms. . 36

LIST OF FIGURES

1 A set of 8 items 8
2 Conflict matrix of 8 items 8
3 An optimal solution with 2 bins 9

CHAPTER I

INTRODUCTION

The *Bin Packing Problem* (BPP) is a well-known combinatorial problem in the literature. In a setting where there are some items to be packed into fixed-sized bins, the basic BBP aims to pack them all by using as few bins as possible. It can be achieved by finding the right combination of items to be packed in each bin without exceeding the bin capacity. The applications of BPP can be found in many different areas such as loading trucks, allocating memory in computers, and scheduling jobs to identical processors [1].

However, during the packing of items, some item pairs might be in conflict such that they cannot be packed into the same bin. This realization has emerged the variant of BPP called the *Bin Packing Problem with Conflicts* (BPPC). BPPC has many applications in real-life, not only in the distribution of some goods that cannot be delivered by the same vehicle at the same time [2] but also in the scheduling of examinations [3], balancing loads of tasks in parallel computing and assignment of processes to processors [4], storing incompatible commodities (as explosives and flammables) in warehouses [5] and resource clustering in parallel computing [6].

There is another variant of the BPP where the bin capacity can be exceeded only by the last item packed into the bin called *overflow item*. This problem is called *Open-End Bin Packing Problem* (OEBPP). In OEBPP, deciding which item to assign as the *overflow item* plays a critical role since it might change the total number of bins used. OEBPP is inspired by the fare payment system in Hong Kong's subway stations. There, a passenger buys a fixed charged ticket and the amount deducted from the ticket is determined at the end of the travel by a machine. The machine returns the ticket to the passenger if the remaining

balance is positive and keeps it otherwise. Therefore, a passenger can travel to any destination in a single trip as long as there is still a positive balance in the ticket in the beginning, so the objective is to use fewer tickets [7]. Apart from this application in transportation, Ongkunaruk states [8] that OEBPP can also be observed in manufacturing, where jobs are assigned to machines that can operate automatically once the operator loads the job. Since the objective is to minimize the total time to complete all jobs, scheduling the job with the longest processing time as the last job is wiser than other options in ordering. Therefore, the machine can be utilized better while the worker is away. In addition to the real-life application, the stacking dishes problem and bin covering problem are also some related problems to the OEBPP from the literature [7].

In this thesis study, we study the variant which is the combination of BPPC and OEBPP. The item pairs which are in conflict cannot be packed into the same bin, and the bin capacity can be exceeded only by the last item packed into the bin. We call this variant the *Open-End Bin Packing Problem with Conflicts* (OEBPPC). The objective of OEBPPC is to pack the items into the least number of bins while considering the conflicts and *overflow items*.

There can be several potential applications of this new problem in real life as the mentioned two variants above. First, it might be applied during container or truck loading in marine and ground transportation. The containers or trucks can have their upper side open in some situations, enabling them to be overloaded without forcing the rule of placing the *overflow item*. Only the last item packed into these containers/trucks can exceed the bin capacity. Among these items to be packed, there can be items of different types which might have a risk of leakage or explosion, making them unable to travel inside the same container/truck. In that case, we declare that there is a conflict between these item pairs. For example, a food box and chemicals should not be loaded into the same place together. This problem structure may also be suitable for storing several types of products in multiple warehouses where the warehouses do not have a ceiling.

To the best of our knowledge, OEBPPC has not been studied in the literature, and we are the first to study this combination of BPPC and OEBPP. We develop a metaheuristic algorithm, namely Variable Neighborhood Search. We compare its performance against several algorithms, including lower bounding procedures we develop, adaptations of approximation algorithms proposed for BPPC, and an adaptation of a metaheuristic proposed for basic BPP. Our contributions with this study can be summarized as follows:

- We introduce a new variant of BPP.
- We propose several lower bounding procedures for the problem where some of them are based on the maximal clique of the conflict graph.
- We develop a metaheuristic algorithm, namely Variable Neighborhood Search (VNS) for our problem.
- We compare the performance of the proposed metaheuristic approach against the modifications of the benchmark algorithms from the literature and detect a remarkable performance of the proposed approach.

The rest of the thesis is organized as follows. In §2, we discuss the variants of the BP and what has been studied so far in the literature related to our problem. In §3, we provide an integer programming (IP) formulation for the OEBPPC and provide an example that describes the relationship between items and the condition of the *overflow items*. In §4, we propose several lower bounding procedures, some of them are based on maximal clique of the conflict graph. In §5, we propose a VNS algorithm tailored for our problem. In §6, we present the computational analysis performed to compare our algorithm against the benchmark algorithms and the lower bounds we obtain. Finally, in §7, we conclude our study with a summary.

CHAPTER II

LITERATURE REVIEW

In this section, we review some of the articles about BPP and its variants related to our study. The basic BPP is a well-known problem that focuses on fitting a set of items into the least number of bins where the item sizes are known, and the bin capacities are the same for all bins. It is observed that the famous approximation algorithms *First-Fit* and *Best-Fit* have their worst-case ratio when items are packed in increasing order of size, *Next-Fit* performs its worst-case ratio when small, and large items are merged [1]. Therefore, pre-sorting the items according to decreasing order of sizes yields better results. It is shown that among all polynomial-time algorithms for BPP, *First-Fit Decreasing* (FFD) [9] and *Best-Fit Decreasing* (BFD) have the best possible absolute performance ratios [10]. Besides these algorithms, classic heuristic algorithms such as *Minimum Bin Slack* [11] and metaheuristic algorithms such as evolutionary algorithms - especially the *Genetic Algorithm* - ([12], [13], [14], [15], [16]), *Variable Neighborhood Search Algorithm* ([17], [18], [19]) and *Ant Colony Optimization Algorithm* [20] are some of the known algorithms developed for BPP.

In time, several variants of BPP emerged, such as *Two-Dimensional/ Three-Dimensional Bin Packing* ([21], [22]), *Open-End Bin Packing* [7], *Variable Sized Bin Packing* [23], *Bin Packing Problem with Conflicts* [24] and *Bin Packing Problem with Conflicts and Item Fragmentation* [25]. We introduce a new variant, *Open-End Bin Packing Problem with Conflicts* (OEBPPC), which is the combination of two of these known variants, *Bin Packing Problem with Conflicts* and *Open-End Bin Packing Problem.*

Leung et al. [7] introduce the *Open-End Bin Packing Problem* (OEBPP). The authors claim that in the optimal solution of OEBPP, the largest items are

packed as the last items, called *overflow items*, in each bin and the remaining items are packed into those bins using strictly less than the bin capacity. They prove that OEBPP is NP-Hard hence they propose two approximation algorithms: an online algorithm (where the knowledge of all items is not known prior to packing; therefore there is a packing order of the items since items are packed as they arrive) and an offline algorithm (the knowledge of items is known before packing so their order can be re-arranged). The offline algorithm first selects a minimum number of the largest items from the list as *overflow items* - considering the lower bound and upper bound on the optimal number of bins - and removes them from the list. Finally, it packs the remaining items into bins with less than 100% of capacity with the *Karmarkar and Karp Algorithm* [26]. It is stated that FFD or BFD algorithms might also be used instead, for this step.

Yang and Leung [27] introduce the *Ordered Open-End Bin Packing Problem* (OOEBPP). The *ordered* requirement implies that the items to be packed are sorted in an order. Considering the item sizes between (0,1], they propose three online algorithms: *Mixed-Fit* (MXF), *Next-Fit* (NF), and *Modified Best-Fit* (MBF[α]); two offline algorithms: *Greedy Look-Ahead Next-Fit* (GLANF) and *Divide-and-Pack* (DP). Later, Ceselli and Righini [28] develop a *Branch-and-Price Algorithm* for the OOEBPP.

Ongkunaruk [8] proposes the *Modified First-Fit Decreasing* (MFFD) algorithm for the OEBPP by removing k largest items in the beginning and then applying FFD to the remaining items with bins having a capacity less than 100%. Finally, these largest k items are packed into the bins in order, as *overflow items*. The author also applies FFD to the OEBPP and shows that compared to the optimal solutions, FFD and MFFD use no more than 33% and 1% bins in general.

Mohamed et al. [29] propose two heuristics, namely *Adapted First-Fit Decreasing* (AFFD) and *Adapted Minimum Bin Slack* (AMBS) for OEBPP. In the AFFD, whenever a new bin is opened, the *overflow item* of that bin is selected as largest item among already packed items but not yet assigned as an *overflow item*; call it

item j. Then the item about to be packed into the new bin is swapped with the selected item j. They perform the computational experiments with Falkenauer instances and show that AFFD performs better than MFFD in [8] on average, significantly when the problem size increases.

Gendreau et al. [24] introduce the *Bin Packing Problem with Conflicts* (BPPC), where some items are in conflict and cannot be packed together in the same bin. They represent the conflicts between items with a conflict graph. Due to the NP-hardness of the problem, they propose six different heuristics: one of them is *Modified First-Fit Decreasing* (MFFD), others are heuristics based on graph coloring and clique computations. They also propose two lower bounds, the first one ignores the conflict constraints, and the second one uses clique computation and solves a transportation problem.

Muritiba et al. [30] evaluate the upper bounds obtained from adapted versions of FFD, BFD, and *Worst-Fit Decreasing* (WFD) to BPPC. They apply a parametric approach to these algorithms where items are sorted according to their surrogate weights/sizes which is calculated as considering both item's size and its conflicts with other items. They present clique-based, surrogate relaxation, matching-based lower bounds, and a population heuristic. They also propose an exact approach that depends on the set covering formulation solved by column generation and branch-and-price algorithm.

Since the OEBPP and BPPC are both NP-hard problems, our problem OEBPPC is also NP-hard. Moreover, we observe that in the optimal solution, the *overflow items* tend to be chosen among the largest items in the item set, which we illustrate in the next section. Therefore, we propose a *Variable Neighborhood Search* (VNS) [31] algorithm for the OEBPPC. We compare its performance against several numbers: the lower bounds we obtain, solutions of the modifications of well-known heuristics proposed for BPPC and modification of a metaheuristic proposed for basic BPP.

CHAPTER III

PROBLEM DEFINITION AND FORMULATION

In this section, we introduce the formal definition of the problem and the integer programming model for our problem. It is inspired by the formulations of the *Bin Packing Problem with Conflicts* (BPPC) and the *Ordered Open End Bin Packing Problem* (OOEBPP) proposed by [24] and [28] respectively.

Let $V = \{1, 2, ..., i, ..., n\}$ be the set of items where each item has a size w_i and each bin is identical with a capacity C. Without loss of generality, we assume that $w_1 \geq w_2 \geq w_3 \geq ... \geq w_n$. If item i and item j cannot be assigned to the same bin, they are in *conflict*. Conflicts between item pairs (i, j) are represented with conflict graph $G = (V, E)$ where an existing edge $(i, j) \in E$ implies the conflict between items i and j in V. During the packing, capacity of each bin can be exceeded only by the last item packed into the bin (*overflow item*) therefore a small amount of capacity $\varepsilon = 1$ is enough to place that item. At this point, the capacity of bins C might be defined as the capacity they would have if they were all closed and none of the items were overflowing. These bins are identical; however, due to the open-end property, each might have a different overflow level hence having a different load. In other words, each bin can be filled up to a level less than the total capacity of the bin, then the *overflow item* is packed which has the size of the remaining capacity or more. Hence, it is wiser to choose the *overflow items* of each bin among the largest items in the item set, leaving more potential space to be filled inside the bins by other items. This property would yield using fewer bins to pack all the items as aimed. An illustration that explains the problem structure is presented by an example.

Example: Suppose we have 8 items of various sizes denoted as w_i's as shown in Figure 1. The conflicts between items are represented by the conflict graph as

seen in Figure 2. An edge between items (i, j) stands for an existence of conflict between the items i and j.

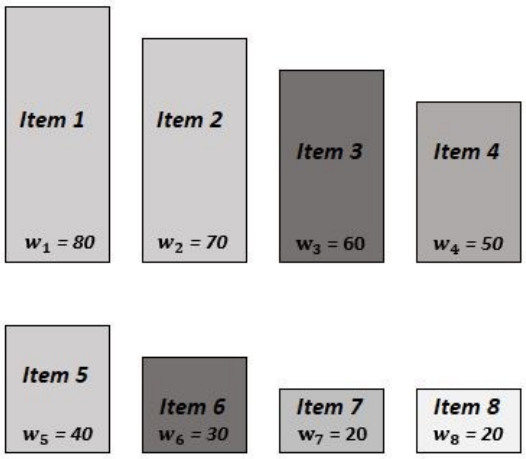

Figure 1: A set of 8 items

The solution is presented in Figure 3 where all of the items are packed using 2 bins, *Item 1* and *Item 2* being the *overflow items* of *Bin 1* and *Bin 2* respectively.

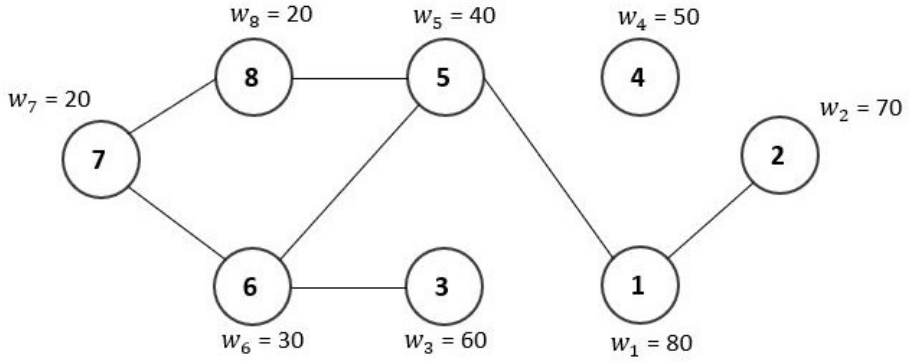

Figure 2: Conflict matrix of 8 items

The logic behind the following model is to minimize the number of *overflow items* to minimize the number of bins used. Hence it is assumed that whenever a bin is opened, an *overflow item* is assigned to that bin. This might result in bin(s) with a single item and that item is the *overflow item* of that bin, even if it is not overflowing.

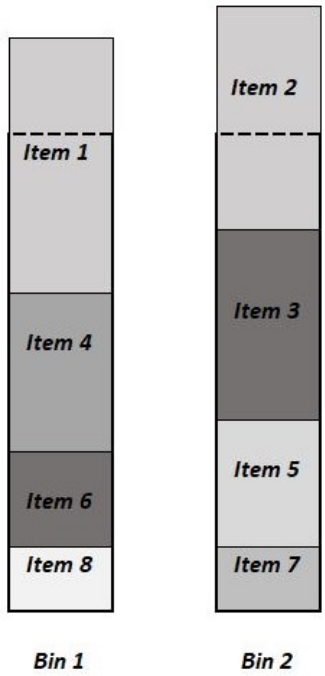

Figure 3: An optimal solution with 2 bins

The decision variables we use in this formulation are as follows:

$$x_{ij} = \begin{cases} 1, & \text{if item } i \text{ is assigned to bin in which the overflow item is item } j \\ 0, & \text{otherwise.} \end{cases}$$

$$y_i = \begin{cases} 1, & \text{if item } i \text{ is the overflow item in its bin} \\ 0, & \text{otherwise.} \end{cases}$$

The formulation can be represented as follows:

$$\text{IP: Min} \quad \sum_{i \in V} y_i \tag{1}$$

s.t.

$$y_i + \sum_{i \neq j \in V} x_{ij} = 1 \qquad \forall i \in V \tag{2}$$

$$\sum_{i \neq j \in V} w_i x_{ij} \leq (C-1) y_j \quad \forall j \in V \tag{3}$$

$$x_{ij} + y_j \leq 1 \qquad \forall (i,j) \in E,\ i \neq j \tag{4}$$

$$x_{ik} + x_{jk} \leq 1 \qquad \forall (i,j) \in E, k \in V,\ i \neq j \neq k \tag{5}$$

$$x_{ij} \in \{0,1\} \qquad \forall i,j \in V \tag{6}$$

$$y_j \in \{0,1\} \qquad \forall j \in V \tag{7}$$

In the formulation, the objective is to minimize the number of *overflow items*, therefore the number of bins used, which is represented by Equation (1). Because each *overflow item* initializes its bin. Constraints (2) ensure that each item is assigned to a bin, either as an *overflow item* of that bin or not. Constraints (3) ensure that the overall sizes of items assigned to a bin (except the *overflow item*) must leave at least one-unit capacity available for the *overflow item* of that bin. Constraints (4) prevent two conflicting items, item i and item j being in the same bin where item j is the *overflow item*. Constraints (5) prevent two conflicting items, item i and item j being in the same bin where item k is the *overflow item*. Finally, the integrality restrictions are represented by Constraints (6) and (7).

CHAPTER IV

LOWER BOUNDS

We propose three lower bounds generated by the following procedures:

- LB_1: This lower bound is motivated by the trivial continuous lower bound from the BPP [32]. Let there be bins equal to the number of items (i.e., $|V|$). We update the bin capacities by assuming each item in kth order in V is the *overflow item* of bin k and bin k can be filled up to that level. In other words, we assume that there are k types of bins, each with $C - 1 + w_k$ capacity, and can only be used once. We use B_{new} to denote these bins. We start solving the problem with these empty and updated bins (B_{new}). We calculate the sum of item sizes and place this total size in the bins in order, starting from the bin with the largest capacity while disregarding the item conflicts and integrality constraints.

- LB_2: Gendreau et al. [24] propose a maximal clique-based lower bounding procedure for BPPC which we modify for OEBPPC. First, we discover a maximal clique of the conflict graph by using Johnson's additive procedure [33]. The items in the maximal clique cannot be packed into the same bin, and we use V_c to denote these items. Since each item $i \in V_c$ should be packed into a separate bin, $|V_c|$ bins are required to pack them along with other items that are not in this maximal clique.

  In this lower bounding procedure, we consider the item conflicts while packing the items into $|V_c|$ bins where items in the maximal clique are packed. Nonetheless, we ignore the item conflicts while packing the remaining items in additional bins. For the lower bounding model, we use the bins with redefined capacities created in LB_1 (denoted as B_{new}) and assume that the bin in jth order is a bin of type j and has capacity C'_j. We define the decision

variables as follows:

$$x_{ij} = \begin{cases} 1, & \text{if bin of item } i \text{ is of type } j \\ 0, & \text{otherwise.} \end{cases} \qquad i \in V_c, j \in B_{new}$$

$$y_{ki} = \begin{cases} 1, & \text{if item } k \text{ is packed into bin of item } i \\ 0, & \text{otherwise.} \end{cases} \qquad k \in V \setminus V_c, i \in V_c$$

$$y_{k0} = \begin{cases} 1, & \text{if item } k \text{ is packed into additional bins} \\ 0, & \text{otherwise.} \end{cases} \qquad k \in V \setminus V_c$$

$$z_j = \begin{cases} 1, & \text{if bin of type } j \text{ is an additional bin} \\ 0, & \text{otherwise.} \end{cases} \qquad j \in B_{new}$$

The formulation can be represented as follows:

$$\text{IP - LB: Min} \quad |V_c| + \sum_{j \in B_{new}} z_j \tag{8}$$

s.t.

$$\sum_{k \in V \setminus V_c} y_{ki} w_k + w_i \leq \sum_{j \in B_{new}} C'_j x_{ij} \quad \forall i \in V_c \tag{9}$$

$$\sum_{j \in B_{new}} x_{ij} = 1 \quad \forall i \in V_c \tag{10}$$

$$\sum_{k \in V \setminus V_c} y_{k0} w_k \leq \sum_{j \in B_{new}} C'_j z_j \tag{11}$$

$$\sum_{i \in V_c} y_{ki} + y_{k0} = 1 \quad \forall k \in V \setminus V_c \tag{12}$$

$$\sum_{i \in V_c} x_{ij} + z_j \leq 1 \quad \forall j \in B_{new} \tag{13}$$

$$y_{ki} = 0 \quad \forall k \in V \setminus V_c, \; \forall i \in V_c, (k,i) \in E \tag{14}$$

$$y_{ki} + y_{k'i} \leq 1 \quad \forall k, k' \in V \setminus V_c, \; (k,k') \in E, \; \forall i \in V_c, k \neq k' \tag{15}$$

$$x_{ij} \in \{0,1\} \quad \forall i \in V_c, \forall j \in B_{new} \tag{16}$$

$$y_{ki} \in \{0,1\} \quad \forall k \in V \setminus V_c, \; \forall i \in V_c \tag{17}$$

$$y_{k0} \in \{0,1\} \quad \forall k \in V \setminus V_c \tag{18}$$

$$z_j \in \{0,1\} \quad \forall j \in B_{new} \tag{19}$$

Objective function (8) minimizes the number of bins used, i.e. the number of bins required to pack items in the maximal click and the total number of additional bins. Constraints (9) impose the capacity of each bin used for the items in the maximal clique and other items that are packed together with them. Constraints (10) guarantee that each bin type is used once to pack the items in the maximal clique. Constraint (11) impose the total capacity used for the items packed into additional bins. Constraints (12) ensure that each item outside the maximal clique is packed together with an item in the maximal clique or in an additional bin. Constraints (13) ensure that each bin is either a bin used to pack an item in the maximal clique or an additional bin. Item conflicts are considered by constraints (14) and (15) for bins that are used for the maximal clique items. The integrality restrictions for the decision variables are ensured by the remaining constraints.

- LB_3: Similar to the bin capacity updating procedure described in LB_1, in this lower bounding procedure, we consider item conflicts while re-defining the bin capacities. We denote the new bins with B_{new2}. Let there be $|V|$ bins. Then we assume each item in kth order in V is the *overflow item* of bin k, and for each item k:

  1. Collect the items that are not conflicting with item k, in a set called V_{nc}.
  2. Solve *Knapsack with Conflicts* with the item set V_{nc}:

$$a_i = \begin{cases} 1, & \text{if item } i \text{ is selected to be packed in the bin} \\ 0, & \text{otherwise.} \end{cases}$$

$$\text{Max} \quad \sum_{i \in V_{nc}} a_i w_i + w_k \tag{20}$$

s.t.

$$\sum_{i \in V_{nc}} a_i w_i \leq C - 1 \tag{21}$$

$$a_i + a_j \leq 1 \qquad \forall (i,j) \in E, \quad \forall (i,j) \in V_{nc}, i \neq j \tag{22}$$

$$a_i \in \{0,1\} \qquad \forall i \in V_{nc} \tag{23}$$

Objective function (20) maximizes the sum of the sizes of items that are packed in the bin of item k and the size of item k. Bin capacity is imposed by Constraints (21). Constraints (22) ensure that conflicting items cannot be packed together in the bin.

The result gives the re-defined capacity of the kth bin (type k bin) in B_{new2}. Then, we solve the IP-LB from LB_2 with B_{new2} instead of B_{new}.

CHAPTER V

PROPOSED SOLUTION APPROACH

We propose a *Variable Neighborhood Search Algorithm* (VNS) for OEBPPC to obtain solutions of high quality. VNS is a metaheuristic algorithm introduced by Mladenović and Hansen [31]. It is based on searching by systematic change of neighborhoods to escape from local minima [18]. A recent survey on VNS can be found in [34]. In the following, we propose our implementation of VNS for the OEBPPC.

5.1 Overall VNS Framework

Our VNS algorithm uses the solution obtained from *FFD for OEBPPC* as an initial solution, which will be described in details in Section 5.2. The general structure of the algorithm is presented in Algorithm 1.

It starts from an initial feasible solution S_0 that has the information of the list of the bins and items inside these bins. The best solution so far (S_{best}) is initialized as the initial solution (S_0). Then at each iteration, one of the neighborhood operators (N_k) is visited in *Shaking*, and a neighbor S' of S_{best} is selected. After visiting a neighborhood, S' is improved through *Local Search* where S'' is obtained. S_{best} is compared with S'' and updated if a better solution is found. During this comparison of S'' and S_{best}, they might have the same number of bins but the bins' load in the solutions might differ from each other. In such situations, we use the following fitness function inspired by Fleszar and Hindi [17] to compare these two solutions, along with the original objective function:

$$f(S) = \sum_{b \in B_{sol}} (l(b))^2 \tag{24}$$

where B_{sol} denotes the set of bins b used in the solution S and $l(b) = \sum_{i \in V_b} w_i$ is the load of bin b. Here V_b denotes the set of items i packed in bin b. This

function maximizes the bin loads and aims to reduce number of bins used. For two solutions with the same number of bins, higher fitness is better.

The value of $k \in \{1, 2, 3\}$ determines which neighborhood operator is going to be visited at *Shaking* in each iteration. It is initially set as $k = 1$ and increases by 1 until S_{best} value is updated or an iteration where $k = 3$ is completed. If S_{best} is updated or an iteration with $k = 3$ is completed, the value is set as $k = 1$. This indicates that the first neighborhood operator is revisited in the *Shaking* during the next iteration.

Algorithm 1 *VNS for OEBPPC*

Input: an instance for OEBPPC with V set of items, C bin capacity and conflict matrix; N_k neighborhood operators, $k \in \{1, 2, 3\}$
Output: S_{best} best solution found

S_0 is generated with *FFD for OEBPPC*
$S_{best} = S_0$
$iterations = 0$
$time\ elapsed = 0$
while stopping criteria not met **do**
  $k = 1$
  while $k \leq k\_max$ **do**
    $S' = Shaking(S_{best}, k)$
    $S'' = LocalSearch(S')$
    if S'' has less number of bins than S_{best} **then**
      $S_{best} = S''$
      $k = 1$
    else if S'' has the same number of bins as S_{best} **then**
      if $f(S'') > f(S_{best})$ **then**
        $S_{best} = S''$
        $k = 1$
      else
        $k = k + 1$
      end if
    else
      $k = k + 1$
    end if
    $iterations = iterations + 1$
  end while
end while

In the following subsections, we describe how we generate the initial solution S_0, the neighborhood operators and the improvements in detail.

5.2 Initial Solution Construction

We develop an adaptation of the FFD algorithm for our problem, which we call *FFD for OEBPPC*, in order to obtain initial feasible solutions. FFD is based on the idea that it is harder to place larger items into bins than smaller items; hence larger items should be packed first. The pseudocode of *FFD for OEBPPC* is presented in Algorithm 2. Firstly, the items are first sorted non-increasingly, then started to be packed. Initially, there is one open bin. Whenever an item is packed into a new bin (i.e., an empty bin), that item is assigned as the *overflow item* of that bin. For each item i, the open bins are evaluated starting from the first one. The item is packed into the first available bin, i.e. the bin with available capacity and items non-conflicting with item i as checked in *line 7*. If item i cannot be packed in any open bin, a new bin is opened. We assume that items packed as *overflow items* use 1 unit of capacity during packing since only 1 unit is enough to pack them.

5.3 Shaking

In the shaking step, each neighborhood operator is used to obtain different neighbor solutions of the best solution so far (S_{best}). The operators have the same logic behind them: first disrupt the solution, then repair. Hence the solution is likely to undergo a significant change where some items are removed. After this disruption in any operator, the solution is fixed by re-placing the missing items into the currently used bins or more bins. This is performed by the *Repair Operator*, an adaptation of FFD heuristic that starts from an infeasible solution. The details of the operators will be explained in the following subsections.

5.3.1 First Neighborhood Operator (N_1)

The first operator is based on fully emptying some bins randomly, and it is a familiar operator for VNS as utilized in [18] and [19]. In our setting, it randomly selects $\alpha\%$ of the bins in the best solution so far S_{best} to be emptied, then discarded

Algorithm 2 *FFD for OEBPPC*

Input: V **set of items,** C **bin capacity and conflict matrix**
Output: S_0 **initial solution**

```
sort items i ∈ V in non-increasing order of w_i
B_sol = {1}, bins used in the solution
V_b = ∅, items in bin b ∈ B_sol
for i ∈ V do
    b = 1
    while b ≤ |B_sol| do
        if i can be packed in b then
            if |V_b| = 0 then
                overflow_item_b = i
            end if
            V_b = V_b ∪ {i}
            break
        else if b = |B_sol| then
            B_sol = B_sol ∪ {b + 1}
            overflow_item_{b+1} = i
            V_{b+1} = V_{b+1} ∪ {i}
            break
        end if
        b+ = 1
    end while
end for
```

them from the solution. The pseudocode of the *First Neighborhood Operator* is presented in Algorithm 3. The items inside the selected bins to be emptied are removed one by one and become unplaced items, stored in V_u. The number of selected bins is denoted by emp_{bin}. Here, B_{sol} denotes the set of bins b used explicitly in the solution S_{best}, rather than denoting the bins used in a general solution S as in Equation 24 or Algorithm 2. The least fully bin is always among the selected bins during this selection. In other words, it is the bin with the smallest capacity used assuming that the *overflow item* consumes 1 unit of capacity from the bin capacity C. After the bins are removed from the solution, the infeasible solution S_{best} and the unplaced items V_u are sent to the *RepairOperator* to obtain a feasible solution S'. In this study, α is selected as 20.

Algorithm 3 *First Neighborhood Operator*

Input: best solution so far S_{best}**, percentage** α
Output: solution of shaking S'

1: $B' = \emptyset$, selected bins to be emptied
2: $V_u = \emptyset$, unplaced items
3: V_b, items in bin $b \in B_{sol}$
4: emp_{bin} = ceiling($\alpha\%$ * $|B_{sol}|$)
5: select the least fully bin b_{least} in B_{sol}
6: B' = a random sample of bins with size emp_{bin}-1 from B_{sol} except b_{least}
7: $B' = B' \cup \{b_{least}\}$
8: **for** $b \in B'$ **do**
9: **for** $i \in V_b$ **do**
10: $V_b = V_b - \{i\}$
11: $V_u = V_u \cup \{i\}$
12: **end for**
13: $B_{sol} = B_{sol} - \{b\}$
14: **end for**
15: $S' = RepairOperator(S_{best}, V_u)$

5.3.2 Second Neighborhood Operator (N_2)

The second operator randomly selects $\beta\%$ of the bins in the best solution so far S_{best}, then removes their *overflow items*. The pseudocode of the *Second Neighborhood Operator* is presented in Algorithm 4. The removed *overflow items* are stored in V_u. The number of selected bins is denoted by emp_{bin}. Here, B_{sol} denotes the set of bins b in the solution S_{best}, as mentioned in Section 5.3.1. Since the first item packed in a new bin is assigned as an *overflow item* in general, there can be bins with a single item that is an *overflow item*. If such a bin is selected, it will be empty after the removal. Since each bin should have an *overflow item* in general, if the selected bin b has two or more items, then a $candidate\_overflow\_item_b$ is selected as the item with the maximum size in V_b, except the $current\_overflow\_item_b$. That $candidate\_overflow\_item_b$ is assigned as $overflow\_item_b$ once the $current\_overflow\_item_b$ is removed from that bin as performed in *lines 7-10*. After this removal process is done, any bin that becomes empty is removed from the solution (*lines 16-21*). Then the infeasible solution S_{best} and the unplaced *overflow items* V_u are sent to the *RepairOperator* to obtain a feasible solution S'. In this study, β is selected as 50.

Algorithm 4 *Second Neighborhood Operator*

Input: best solution so far S_{best}**, percentage** β
Output: solution of shaking S'

$B' = \emptyset$, selected bins to get overflow items from
$V_u = \emptyset$, unplaced overflow items
emp_{bin} = ceiling($\beta\%$ * $|B_{sol}|$)
B' = a random sample of bins with size emp_{bin} from B_{sol}
for $b \in B'$ **do**
    $current\_overflow\_item_b = overflow\_item_b$
    if $|V_b| \geq 2$ **then**
        Set $i \in V_b$ with maximum size except the $current\_overflow\_item_b$ as the $candidate\_overflow\_item_b$
        $V_b = V_b - \{current\_overflow\_item_b\}$
        $overflow\_item_b = candidate\_overflow\_item_b$
    else
        $V_b = V_b - \{current\_overflow\_item_b\}$
    end if
    $V_u = V_u \cup \{current\_overflow\_item_b\}$
end for
$m = 1$
while $m \leq |B_{sol}|$ **do**
    if $m \in B_{sol}$ has capacity used = 0 **then**
        $B_{sol} = B_{sol} - m$
    end if
end while
$S' = RepairOperator(S_{best}, V_u)$

5.3.3 Third Neighborhood Operator (N_3)

The third operator randomly selects $\gamma\%$ of all items and removes them from their bins in S_{best}. The pseudocode of the *Third Neighborhood Operator* is presented in Algorithm 5. The removed items are stored in V_u. Number of selected items is denoted by emp_{item}. Here, B_{sol} denotes the set of bins b used explicitly in the solution S_{best}. Similar to the second neighborhood operator, if the selected item i is an *overflow item* of a bin b with more than 1 item, the $overflow\_item_b$ might be updated in the same way. After this removal process is done, the infeasible solution S_{best} and the unplaced items V_u are sent to the *RepairOperator* to obtain a feasible solution S'. In this study, γ is selected as 20.

Algorithm 5 *Third Neighborhood Operator*

Input: best solution so far S_{best}**, percentage** γ
Output: solution of shaking S'

1: $V' = \emptyset$, selected items to be removed
2: $V_u = \emptyset$, unplaced items
3: emp_{item} = ceiling($\gamma\%$ * $|V|$)
4: V' = a random sample of items with size emp_{item} from V
5: **for** $i \in V'$ **do**
6: **if** $i \in V_b$ is the $overflow\_item_b$ and $|V_b| > 1$ **then**
7: Set $j \in V_b$ with maximum size except i as the $candidate\_overflow\_item_b$
8: $V_b = V_b - \{i\}$
9: $overflow\_item_b = candidate\_overflow\_item_b$
10: **else**
11: $V_b = V_b - \{i\}$
12: **end if**
13: $V_u = V_u \cup \{i\}$
14: **end for**
15: $m = 1$
16: **while** $m \leq |B_{sol}|$ **do**
17: **if** $m \in B_{sol}$ has capacity used = 0 **then**
18: $B_{sol} = B_{sol} - m$
19: **end if**
20: **end while**
21: $S' = RepairOperator(S_{best}, V_u)$

5.4 Repair Operator

After the solution S_{best} has changed and the items in V_u are missing from the solution, the solution becomes infeasible. Therefore, these are sent to the *Repair Operator* to obtain a feasible solution for the *Shaking*. This operator works similar to the Algorithm 2 however, this time the algorithm starts from an infeasible solution with non-empty bins. The pseudocode of the *RepairOperator* is presented in Algorithm 6. First, the unplaced items (V_u) are sorted non-increasingly. Then each item k is tried to be packed in the first available bin considering the conflict and capacity constraints. During this repairment, if the item k to be packed is bigger than the *overflow item* of that bin b in consideration, and there is enough capacity to store the *overflow item* as a non-overflow item inside the bin, the item k becomes the new *overflow item* of that bin as performed in *lines 6-8*. If that is not possible, item k can be packed inside the bin if the bin capacity allows (*lines*

10-11). If item k cannot be packed in an open bin, a new bin (bin $b+1$) is opened and that item is placed as the *overflow item* of the new bin as in *lines 18-21*.

Algorithm 6 *Repair Operator*

Input: infeasible solution S_{best}**, unplaced items** V_u
Output: feasible solution S'

1: sort $\forall k \in V_u$, in non-increasing order of w_k
2: **for** $k \in V_u$ **do**
3: b = 1
4: **while** $b \leq |B_{sol}|$ **do**
5: **if** k is not conflicting with items in V_b **then**
6: **if** $w_k > overflow\_item_b$ **and** $overflow\_item_b$ can be packed as non-overflow item if k would be the new $overflow\ item$ **then**
7: assign $overflow\_item_b$ as non-overflow item
8: $overflow\_item_b = k$
9: **break**
10: **else if** k can be packed as non-overflow item **then**
11: $V_b = V_b \cup \{k\}$
12: **else if** $b = |B_{sol}|$ **then**
13: $B_{sol} = B_{sol} \cup \{b+1\}$
14: $overflow\_item_{b+1} = k$
15: $V_{b+1} = V_{b+1} \cup \{k\}$
16: **break**
17: **end if**
18: **else if** $b = |B_{sol}|$ **then**
19: $B = B \cup \{b+1\}$
20: $overflow\_item_b = k$
21: $V_{b+1} = V_{b+1} \cup \{k\}$
22: **break**
23: **end if**
24: $b = b + 1$
25: **end while**
26: **end for**

5.5 Local Search

At each iteration, there are two different local search procedures performed after the *Shaking*: *transfer* and *swap*. These are inspired from the local search applications in the VNS of Fleszar and Hindi [17]. In our VNS, both local search procedures have the steepest descent approach where the possible improving moves are enumerated separately for transfer and swap procedures. In each of the procedures, the move which increases the fitness function the most is performed under

capacity and conflict restrictions. Different than [17], we apply these improving moves (if any) separately for transfer and swap procedures rather than performing only one (the best improving) of them. Hence, transfer and swap operations can be performed consecutively in an iteration. According to [17], there is no need to recalculate the fitness function of a solution from scratch during the evaluation of moves, and calculating only the change in the fitness function is sufficient. Therefore, the value of transferring item i from bin a to bin b is obtained as follows:

$$\Delta f = [l(a) - w_i]^2 + [l(b) + w_i]^2 - (l(a))^2 - (l(b))^2 \tag{25}$$

Similarly, the value of swapping items i and j from their bins a and b respectively is:

$$\Delta f = [l(a) - w_i + w_j]^2 + [l(b) + w_i - w_j]^2 - (l(a))^2 - (l(b))^2 \tag{26}$$

Where $l(a)$ is the load of bin a, $l(b)$ is the load of bin b as described in Section 5.1.

If a bin's capacity is fully used, then the items inside that bin are not considered during the transfer or the swap. Even if the total loads of that bins might be increased by assigning them larger *overflow items*, it does not always improve the number of bins used in the best solution at the end of the algorithm. Also, using such boxes at each iteration significantly increases the computation time as their number in the solution increases over time. It is more difficult to find suitable items for them. Therefore, we realize that it is not worthy of this computational effort. Thus, a list V'' of all items i except those in such bins are created from the items in the solution S'. Then the items $i \in V''$ are sorted non-increasingly according to their sizes.

In the *transfer*, for each bin a with some available capacity, each item $i \in V''$ is considered starting from the end of the list. If there is no conflict and there exists a place to pack i, the value of transferring item i from bin a to bin b is calculated for each item as in Equation 25. The best transfer with highest Δf where $\Delta f > 0$ is performed. During this operator, item i might become the new $overflow\ item$

of b if it is bigger than its *overflow item* and there is enough capacity to pack that *overflow item* as non-overflow item in b. This is regardless of item i being an $overflow\,item$ or not when it is in bin a. If i is an *overflow item* in a, then a new *overflow item* is determined for that bin if there are more than one item prior to the *transfer*. After the move is performed, if the bin where item i is packed is full, item i is also removed from the list V'' and not considered in the following swap move.

The *swap* considers item pairs i and j to be swapped. Starting from the end of V'', where i is the last item in the list, all items before i in the list called j's are evaluated in order, all the way to the beginning of the list. If i is not in conflict with the items in b except j, and j is not in conflict with the items in a except i, the possible swaps are considered regarding available capacities and conditions of the *overflow items* of bins. The value of swapping item i and j are calculated for possible swaps with Equation 26. The best swap with the highest Δf where $\Delta f > 0$ is performed.

Similar to the *transfer* operator, there are several cases for the *swap*. Prior the swap, items i and j can have 4 different initial conditions:

1. Both item i and j are *overflow items* of their bins a and b respectively
2. Only item i is the *overflow item* of its bin a
3. Only item j is the *overflow item* of its bin b
4. None of these items are *overflow items*

After the move, the items can have these four similar conditions but this time in their new bins (item i in bin b and item j in bin a). All in all, items i and/or j might become the new *overflow items* of bins b and/or a. In contrast, they might be packed as a non-overflow item even though they were *overflow items* prior to the swap. If an *overflow item* leaves its bin, a new one is determined.

In this stage, the used capacities of bins are controlled before and after *swap* and whenever a bin becomes empty, it is removed from the solution.

CHAPTER VI

COMPUTATIONAL STUDY

In this section, we present the results of the computational study we perform to evaluate the performance of the proposed solution approach and the lower bounding procedures. We use Python version 3.7 and IBM ILOG CPLEX Python API (version 20.1.0) with default parameter settings for the implementation, and the computational experiments are carried out on a 64-bit PC equipped with an Intel Core i7-10510U CPU running at 1.8 GHz and 16 GB RAM.

6.1 Instance generation

To test the performance of the algorithms, we modify the instances generated by Falkenauer [12] (available at http://or.dei.unibo.it/library/bpplib). The author proposes eight classes of instances; in the first four classes, the number of items is 120, 250, 500, and 1000 respectively. The item sizes are uniformly distributed as integers in [20, 100], and the bin capacity is 150. Among 20 instances in each of these four classes, we use the first 10 instances. For each of these instances, we randomly generate conflict graphs of specific density (δ) values from the set $\{0, 0.1, 0.2, 0.3, ..., 0.9\}$. For generating the conflict graph with density δ, a random number p is uniformly generated in [0, 1) for each item pair (i and j, $i < j$) , and if $p < \delta$ an edge between these two items is added to the graph. We generate 10 different conflict graphs in a randomized approach for each base instance (4 x 10 = 40 in total), in accordance with the densities discussed above. Hence, we use 400 (= 40 x 10) instances in the computational study.

6.2 Performance of the proposed lower bounds

It is known that finding better lower bounds yields to more accurate performance evaluation of the algorithms. Therefore, we evaluate the quality of the proposed

lower bounds to find the best one. Firstly, we try to solve the OEBPPC to optimality. Since CPLEX fails to solve the IP model of the OEBPPC for some of the 500-item instances and all of the 1000-item instances in the given time (3600 s) and memory capacity, the lower bound values obtained from CPLEX are not used. Then, we develop the lower bounding procedures to obtain good lower bounds under these circumstances. As described in Chapter 4, there are 3 lower bounds. The first one (LB_1) is a trivial continuous lower bound where only covering the total size of items with the total capacity of the bins used is considered, and it can be obtained in a few seconds. Others (LB_2 and LB_3) are based on maximal clique computation and solving the IP-LB model, requiring more time. To obtain these two lower bound values, we run the IP-LB model with 3600 s time limit and report the best bound so far. For LB_3, we run the *Knapsack with Conflicts* with 1200 s time limit prior to the IP-LB model, but it returns the best solution within seconds even for the instances with the highest number of items and highest conflict graph densities.

We compare the lower bounds against the best solution found among the benchmark algorithms and our algorithm to evaluate their qualities. For each lower bound, we calculate the percentage gap as the following:

$$\text{Percentage Gap} = \frac{Best - LB}{Best} \text{ x } 100\% \tag{27}$$

where $Best$ denotes the objective function value of the best solution found, and LB denotes the lower bound. We use the best solution found to perform a fair comparison among the lower bound values. The lower the percentage gap value, the better the lower bound. The average percentage gaps for LB_1, LB_2, and LB_3 are presented in Table 1. Instance-related information is given in the first two columns as the number of items ($|V|$) and the density of the conflict graph (δ). For each $|V|$ and δ value, we generate 10 instances. Therefore, the percentage gap values are the averages of these 10 instances and the values in the bold font indicate better results compared to the values on the same row.

Since the LB_3 is the most tailored one among the lower bounding procedures,

it might be expected to perform better than the others. However, we observe that LB_3 fails to outperform other lower bounding procedures when the density of the conflict graph is lower than 0.7. Since LB_2 and LB_3 utilize the maximal cliques identified, it is an expected outcome for these lower bounding procedures not to improve the solution found in LB_1 for these instances. It is due to the smallness of the maximal clique compared to the number of items which is more significant for the large instances. This observation is obtained in [25] and it is shown that even for the smallest instances (instances of 100 items in that study) the identified maximal cliques have less than the 20% of the items when the density of the conflict graph is lower than 0.9. In addition, as instance size increases, this percentage decreases. Our findings in this study are also in line with this observation. Due to this, LB_2 and LB_3 can find suitable items to use the full capacity of the bins that contain the items in the maximal clique for such instances (knowing that each *overflow item* uses 1 unit of capacity). Therefore determining the maximal clique becomes an unnecessary task. The item conflicts considered while packing the items into the bins that include the items in the maximal clique do not create a meaningful difference for these instances. This is more significant for the instances with the 1000 items: The maximal clique includes a low percentage of items even when the conflict graph is dense. Therefore, LB_2 and LB_3 provide the same values as the trivial lower bound LB_1 for the instances with 1000 items. Hence, these values are not presented in Table 1.

When the instance size is smaller, LB_2 and LB_3 provide better results for conflict graphs with higher densities than LB_1. For instances with 120 items, LB_3 performs even better than LB_2 when density is 0.7 or higher. In detail, LB_3 provides an average gap of 16.16% (=(11.41+17.97+19.09)/3), and LB_2 provides a higher average gap of 16.63% (=(11.70+18.23+19.97)/3). In contrast, we observe that the superiority of LB_2 and LB_3 over LB_1 decreases as the instance size increases. We calculate the average values for each class of instances and each LB in Table 1 and present the average values in Table 2; we observe that these values

get closer to each other as going down the table.

Table 1: Performance of the proposed lower bounds.

$\|V\|$	δ	LB_1	LB_2	LB_3
120	0	1.92%	1.92%	1.92%
	0.1	2.25%	2.25%	2.25%
	0.2	2.89%	2.89%	2.89%
	0.3	3.21%	3.21%	3.21%
	0.4	4.42%	4.42%	4.42%
	0.5	5.63%	5.63%	5.63%
	0.6	7.63%	7.63%	7.63%
	0.7	11.70%	11.70%	**11.41%**
	0.8	20.08%	18.23%	**17.97%**
	0.9	33.56%	19.97%	**19.09%**
250	0	1.23%	1.23%	1.23%
	0.1	1.07%	1.07%	1.07%
	0.2	1.68%	1.68%	1.68%
	0.3	2.59%	2.59%	2.59%
	0.4	3.04%	3.04%	3.04%
	0.5	4.78%	4.78%	4.78%
	0.6	7.54%	7.54%	7.54%
	0.7	11.65%	11.65%	11.65%
	0.8	17.48%	17.48%	**17.35%**
	0.9	28.73%	26.83%	**26.16%**
500	0	1.24%	1.24%	1.24%
	0.1	1.69%	1.69%	1.69%
	0.2	1.85%	1.85%	1.85%
	0.3	2.52%	2.52%	2.52%
	0.4	3.42%	3.42%	3.42%
	0.5	4.51%	4.51%	4.51%
	0.6	6.82%	6.82%	6.82%
	0.7	9.79%	9.79%	9.79%
	0.8	14.87%	14.87%	14.87%
	0.9	25.84%	25.73%	**25.61%**
1000	0	2.76%	-	-
	0.1	2.70%	-	-
	0.2	2.92%	-	-
	0.3	3.30%	-	-
	0.4	3.89%	-	-
	0.5	4.94%	-	-
	0.6	6.25%	-	-
	0.7	8.27%	-	-
	0.8	12.57%	-	-
	0.9	23.55%	-	-

Table 2: Summary of the performance of the proposed lower bounds

$\|V\|$	LB_1	LB_2	LB_3
120	9.33%	7.78%	7.64%
250	7.98%	7.79%	7.71%
500	7.26%	7.24%	7.23%
1000	7.12%	-	-

6.3 Performance of the proposed solution approach

We test the performance of our algorithm against the solution of the lower bounds we obtain in Chapter 4 and several benchmark algorithms from the literature. However, since OEBPPC is a new problem, no algorithm has been developed for it. Hence, we first consider the heuristic algorithms developed by Muritiba et al. [30] known to give the best results among the heuristics developed for BPPC. The authors propose an improved version of *FFD*, *BFD*, and *WFD* for BPPC by sorting items according to the non-increasing values of *surrogate weights* instead of just using item sizes/weights. *Surrogate weight* of item i is calculated as follows:

$$w_i^s = \alpha \frac{w_i}{\bar{w}} + (1-\alpha)\frac{\delta_i}{\bar{\delta}} \tag{28}$$

where $i \in V$, α is a given parameter ($\alpha \in \{0, 0.1, 0.2, ...0.9, 1\}$), δ_i is the degree of item i on the conflict graph and $\bar{w}$ is the average size of all items and $\bar{\delta}$ is the average degree of the items on the conflict graph. First, the items are sorted in non-increasing order of their w_i^s values; then, they implement FFD, BFD, and WFD. Each algorithm is implemented with 11 different values of α, and the best result is reported for each instance. Therefore, these algorithms are denoted by FFD(α), BFD(α), and WFD(α).

We implement these three algorithms to our problem by adding OEBPP properties; whenever a new bin is opened, the first item packed in that bin is assigned as an *overflow item*. As an example, the general flow of FFD(α) is the same as with Algorithm 2 except for *line 1*.

Besides these algorithms, we adapt the metaheuristic algorithm called *Grouping Genetic Algorithm* (GGA) proposed by Falkenauer and Delchambre [35],

the concept is described in more detail in [36]. GGA is the most well-known heuristic for grouping problems such as bin packing, which uses structural information as well as the grouping character of these problems to guide the search, as mentioned in [37]. In GGA, each gene represents a group of items that are packed into a bin. This structure helps to preserve well-filled bins through generations. Using crossover, mutation, and inversion operators, they concluded that GGA performs better than FFD in though conditions especially where the residual capacity of bins is smaller.

We implement the steps described in [36] with the given fitness function (called cost function in that study), operator and parameter settings. Different than that, we keep the original objective function (minimizing the number of bins used) together with the fitness function while searching for the new best solution in each generation's population. In the mutation operator, an upper limit on the number of bins to be eliminated from the solution is not given in the study so we set the limit as at most 20% of the bins used. We also define the inversion operator as sorting the bins of the chromosome in non-increasing order of used capacities. Therefore, these bins are closer to each other to be safe against disruption. In addition, we change the stopping criteria, which will be described later.

In the implementation of our proposed solution approach VNS, we initially test using a different number of neighborhoods. We decide on using three of them by adding/removing them one by one and observing their effect on the best solution obtained. For the parameter values of α, β, and γ in the neighborhood operators, we test different values such as increasing/decreasing the current values by 5-10%, and choose the best values. We set the stopping criteria as reaching 1000 iterations and we add an additional criteria of 3600 s time limit for the 1000-item instances. For these 1000-item instances, VNS performs around 610 iterations on average.

We test the benchmark algorithms on the instances mentioned in Section 6.1. To evaluate the solution quality, we use two metrics: the average optimality gap (OG(%)) and the average execution time (CPU(s)) of the algorithms. To use the

same reference point while this evaluation, we divide the deviation of a solution from the lower bound by the lower bound as presented in [38] and [30] to calculate the optimality gap. Its formulation is as follows:

$$\text{Optimality Gap} = \frac{z_{sol} - LB}{LB} \text{ x } 100\% \tag{29}$$

where z_{sol} is the objective function value of the solution reached, and LB is the best lower bound obtained among the lower bounds for each instance. For the Class 4 instances with 1000-items, we use LB_1 as the LB in Eq.29. The average optimality gaps (OG(%)) of the solutions found for each δ value are presented in Table 3. The first column $|V|$ represents the number of items in the instance and the second column δ represents the density of the conflict graph. Under the columns of "FFD(α)", "BFD(α)", and "WFD(α)", the average optimality gaps of the solutions obtained by the parametric versions of the FFD, BFD, and WFD are presented. Under the "GGA" column, average optimality gaps of the Grouping Genetic Algorithm is provided and the average optimality gaps of the solutions from our approach are under "VNS". The results of the better-performing algorithms are denoted with bold characters in all of the tables.

As presented in Table 3, VNS performs better than all benchmark algorithms for the 120-item instances. Compared to the best performing benchmark algorithm for these instances which is GGA, VNS provides an average improvement of 6.54% over GGA when the density of the conflict graph is 0.9. For the other class of instances, VNS outperforms the benchmark algorithms when the density of the conflict graph is less than 0.6 and precisely 0.9.

For the 250-item instances, VNS performs close to GGA when graph densities are 0.6 and 0.7, with average differences of 0.17% and 0.16%. For the 0.9 graph density, VNS improves the solution of GGA by 5.60% on average. This advantage of VNS over GGA for instances with 0.9 graph density is also observed for 500-item and 1000-item instances. It shows that VNS outperforms the benchmark algorithms when the conflict graph is most dense.

Table 3: Comparison of VNS against the benchmark algorithms.

$\|V\|$	δ	FFD(α)	BFD(α)	WFD(α)	GGA	VNS
120	0	10.27%	10.27%	10.27%	4.97%	**1.98%**
	0.1	10.27%	10.27%	10.95%	4.65%	**2.32%**
	0.2	11.30%	11.27%	12.29%	5.64%	**2.99%**
	0.3	12.93%	12.94%	12.94%	6.30%	**3.31%**
	0.4	14.94%	15.24%	14.91%	6.96%	**4.65%**
	0.5	17.24%	18.23%	18.89%	7.96%	**6.31%**
	0.6	21.20%	21.57%	22.54%	12.27%	**8.28%**
	0.7	28.11%	27.11%	30.41%	17.18%	**12.91%**
	0.8	37.77%	37.13%	43.90%	28.09%	**21.95%**
	0.9	41.61%	41.61%	46.56%	30.22%	**23.68%**
250	0	9.43%	9.43%	9.43%	4.08%	**1.25%**
	0.1	9.74%	9.58%	9.58%	4.39%	**1.09%**
	0.2	10.21%	10.06%	10.06%	4.39%	**1.72%**
	0.3	11.16%	11.00%	10.84%	5.02%	**2.67%**
	0.4	12.26%	11.94%	12.72%	5.33%	**3.14%**
	0.5	13.82%	14.14%	14.77%	6.59%	**5.02%**
	0.6	16.80%	17.11%	17.74%	**8.63%**	8.80%
	0.7	22.62%	22.94%	24.66%	**13.35%**	13.51%
	0.8	31.36%	31.36%	34.65%	22.74%	**21.01%**
	0.9	49.73%	49.73%	55.02%	41.09%	**35.48%**
500	0	9.60%	9.60%	9.60%	4.49%	**1.26%**
	0.1	9.68%	9.76%	9.68%	4.25%	**1.73%**
	0.2	9.84%	10.08%	10.15%	4.64%	**1.89%**
	0.3	10.47%	10.39%	10.31%	4.72%	**2.59%**
	0.4	11.34%	11.49%	11.57%	5.43%	**3.54%**
	0.5	12.75%	12.59%	12.99%	6.14%	**4.72%**
	0.6	15.04%	14.72%	15.59%	**7.32%**	8.66%
	0.7	19.29%	19.13%	20.30%	**10.86%**	12.28%
	0.8	25.50%	26.14%	29.28%	**17.87%**	18.26%
	0.9	41.90%	41.59%	48.40%	36.48%	**34.45%**
1000	0	9.73%	9.73%	9.73%	4.53%	**3.01%**
	0.1	9.85%	9.81%	9.85%	4.53%	**2.78%**
	0.2	10.01%	9.89%	10.01%	4.64%	**3.01%**
	0.3	10.32%	10.13%	10.28%	4.84%	**3.41%**
	0.4	10.80%	10.76%	10.80%	5.12%	**4.05%**
	0.5	11.79%	11.75%	12.31%	5.72%	**5.20%**
	0.6	13.66%	13.62%	14.41%	**6.71%**	7.51%
	0.7	16.84%	16.84%	18.19%	**9.01%**	11.20%
	0.8	22.48%	22.59%	25.89%	**14.37%**	16.04%
	0.9	35.67%	35.47%	39.64%	31.14%	**30.94%**

Table 4 summarizes the results for each conflict graph density value (δ). We observe that GGA is the best performing algorithm in all of the benchmark algorithms and VNS outperforms this algorithm on average. VNS performs the most remarkable average improvement of 3.59% over GGA when the conflict graph density is 0.9. In terms of instance size, VNS shows more stable performance as the instance size increases compared to the GGA, as shown in Table 5.

Table 4: Summary of the results according to conflict graph density.

δ	FFD(α)	BFD(α)	WFD(α)	GGA	VNS
0	9.76%	9.76%	9.76%	4.52%	**1.88%**
0.1	9.88%	9.85%	10.02%	4.45%	**1.98%**
0.2	10.34%	10.32%	10.63%	4.83%	**2.40%**
0.3	11.22%	11.11%	11.10%	5.22%	**3.00%**
0.4	12.33%	12.36%	12.50%	5.71%	**3.85%**
0.5	13.90%	14.18%	14.74%	6.60%	**5.32%**
0.6	16.68%	16.76%	17.57%	8.73%	**8.31%**
0.7	21.71%	21.51%	23.39%	12.60%	**12.47%**
0.8	29.28%	29.31%	33.43%	20.77%	**19.32%**
0.9	42.23%	42.10%	47.41%	34.73%	**31.14%**

Table 5: Summary of the results according to instance size.

$\|V\|$	FFD(α)	BFD(α)	WFD(α)	GGA	VNS
120	20.57%	20.57%	22.37%	12.42%	8.84%
250	18.71%	18.73%	19.95%	11.56%	9.37%
500	16.54%	16.55%	17.79%	10.22%	8.94%
1000	15.11%	15.06%	16.11%	9.06%	8.72%

In summary, among 4 classes of instances each with 100 instances:

- For 120-item instances, VNS achieves a better solution in 84 instances and in 15 of them, it ties with the best solution found by the benchmark algorithms.
- For 250-item instances, VNS achieves a better solution in 78 instances and in 16 of them, it ties with the best solution found by the benchmark algorithms.
- For 500-item instances, VNS achieves a better solution in 72 instances and in 4 of them, it ties with the best solution found by the benchmark algorithms.

- For 1000-item instances, VNS achieves a better solution in 62 instances and in 5 of them, it ties with the best solution found by the benchmark algorithms.

Average CPU time of the algorithms for each δ value are presented in Table 6. As introduced earlier, we set the stopping criteria for VNS as performing 1000 iterations, and we add 3600 s time limit for 1000-item instances. For the GGA, we initially tried limiting the algorithm with the number of iterations; however each iteration takes a significantly large time, and the best solution is not updated rapidly. Hence we choose to focus on the time aspect for the stopping criteria. For this purpose, we first perform VNS on each class of instances and determine the maximum execution time among 100 instances of that class. Then we use that amount of time as the time limit while running GGA with that class of instances. This stopping criterion allows GGA to spend more time on an instance than VNS to compensate for its time-consuming nature of being a population-based metaheuristic. This also prevents GGA from spending many times more time than VNS, as would be the case if GGA was performed with an iteration limit as in VNS. Hence, the comparison of these two metaheuristics is fairer.

Table 6 displays the differences in CPU times (in seconds). Comparing this table to the previous one, we acknowledge that there is a trade-off between the solution time and the quality of the solution. FFD(α), BFD(α), and WFD(α) are the fastest however, they fail to perform better than other algorithms. Although more time is needed to obtain the best solutions, increased solution time does not always guarantee a better solution. In our case, it is seen that GGA spends the greatest time but is still not superior to VNS in the average case performance evaluations. When considering the instances according to the values of conflict graph density in Table 7, we observe that all algorithms tend to spend more time as the graphs become denser.

Table 6: Average CPU time (in seconds) of all algorithms.

$\|V\|$	δ	FFD(α)	BFD(α)	WFD(α)	GGA	VNS
120	0	0.00	0.00	0.00	159.41	93.67
	0.1	0.00	0.00	0.00	159.53	79.09
	0.2	0.00	0.00	0.00	161.32	82.65
	0.3	0.00	0.00	0.00	162.53	82.42
	0.4	0.00	0.00	0.00	161.47	88.93
	0.5	0.00	0.00	0.00	160.59	82.66
	0.6	0.00	0.00	0.00	160.07	95.32
	0.7	0.00	0.00	0.00	162.46	97.17
	0.8	0.00	0.00	0.00	163.87	113.89
	0.9	0.00	0.01	0.00	159.65	126.55
250	0	0.01	0.01	0.01	642.83	317.07
	0.1	0.01	0.01	0.01	635.74	321.41
	0.2	0.01	0.01	0.01	643.56	311.32
	0.3	0.01	0.01	0.01	641.95	304.69
	0.4	0.01	0.02	0.01	638.20	369.29
	0.5	0.01	0.02	0.01	646.31	468.82
	0.6	0.01	0.01	0.02	654.80	419.28
	0.7	0.01	0.02	0.02	652.98	390.47
	0.8	0.02	0.02	0.02	654.51	451.51
	0.9	0.02	0.02	0.02	667.71	459.79
500	0	0.04	0.05	0.05	3127.02	1437.82
	0.1	0.04	0.05	0.05	2984.97	1305.61
	0.2	0.04	0.05	0.05	3006.94	1449.58
	0.3	0.04	0.05	0.06	2999.46	1298.65
	0.4	0.04	0.05	0.06	3009.33	1401.52
	0.5	0.04	0.05	0.06	2982.52	1687.56
	0.6	0.04	0.05	0.06	3000.47	1936.06
	0.7	0.04	0.05	0.06	3032.40	1806.17
	0.8	0.05	0.06	0.07	3088.29	1589.20
	0.9	0.06	0.08	0.08	3046.96	1687.58
1000	0	0.16	0.20	0.20	3727.02	3329.75
	0.1	0.16	0.20	0.20	3712.50	3603.35
	0.2	0.16	0.21	0.21	3749.44	3602.45
	0.3	0.16	0.20	0.21	3748.26	3604.30
	0.4	0.16	0.20	0.21	3684.66	3602.86
	0.5	0.16	0.21	0.21	3722.02	3602.59
	0.6	0.16	0.22	0.22	3702.27	3605.27
	0.7	0.17	0.21	0.23	3752.99	3605.43
	0.8	0.18	0.24	0.24	3840.14	3604.47
	0.9	0.21	0.28	0.30	3918.96	3603.88

Table 7: Summary of the average CPU time (in seconds) of all algorithms.

δ	FFD(α)	BFD(α)	WFD(α)	GGA	VNS
0	0.05	0.07	0.07	1914.07	1294.58
0.1	0.05	0.07	0.07	1873.18	1327.36
0.2	0.05	0.07	0.07	1890.32	1361.50
0.3	0.05	0.07	0.07	1888.05	1322.51
0.4	0.05	0.07	0.07	1873.42	1365.65
0.5	0.05	0.07	0.07	1877.86	1460.41
0.6	0.05	0.07	0.07	1879.40	1513.98
0.7	0.06	0.07	0.08	1900.21	1474.81
0.8	0.06	0.08	0.08	1936.70	1439.77
0.9	0.07	0.10	0.10	1948.32	1469.45

CHAPTER VII

CONCLUSION

In this thesis, we study a new variant of the classic Bin Packing Problem called *Open-End Bin Packing Problem with Conflicts* which might be applied for container/truck loading and storing goods in the warehouses. In this problem, some item pairs have conflicts, and these items cannot be packed into the same bin. In addition, the bin capacity can be exceeded only by the last item packed into the bin, called the *overflow item*.

We first propose lower bounding procedures; some of them are based on the maximal clique of the conflict graph. Using maximal clique and determining the potential bin loads based on conflicts help us achieve better lower bounds for the instances with dense conflict graphs. Then, we develop a *Variable Neighborhood Search* (VNS) metaheuristic algorithm for our problem.

We test the performance of the proposed metaheuristic algorithm on 4 classes of instances with different instance sizes and make a comparison with the benchmark algorithms we adapt to our problem. As the summary in Table 4 indicates, VNS performs better than the best benchmark algorithm on average. In addition, it outperforms the best benchmark algorithm by a large margin when the conflict graph density is 0.9. Considering 400 instances used for the computational study, VNS provides the best solution in 296 of them. In 40 of them, it ties with the best benchmark solution.

To the best of our knowledge, OEBPPC has not been studied in the literature, and we are the first to address this problem. Further studies on our problem might involve considering OEBPPC in multi-dimensional structure, utilizing bins of variable sizes, or allowing item fragmentation.

REFERENCES

[1] E. G. Coffman, J. Csirik, G. Galambos, S. Martello, and D. Vigo, *Handbook of Combinatorial Optimization*, ch. Bin packing approximation algorithms: survey and classification, pp. 455–531. Springer, 2013.

[2] N. Christofides, A. Mingozzi, and P. Toth, *Combinatorial Optimization*, ch. Loading problems, pp. 315–338. Wiley, 1979.

[3] G. Laporte and S. Desroches, "Examination timetabling by computer," *Computers & Operations Research*, vol. 11, no. 4, pp. 351–360, 1984.

[4] K. Jansen, "An approximation scheme for bin packing with conflicts," *Journal of Combinatorial Optimization*, vol. 3, no. 4, pp. 363–377, 1999.

[5] C. Basnet and J. Wilson, "Heuristics for determining the number of warehouses for storing non-compatible products," *International Transactions in Operational Research*, vol. 12, no. 5, pp. 527–538, 2005.

[6] R. Sadykov and F. Vanderbeck, "Bin packing with conflicts: a generic branch-and-price algorithm," *INFORMS Journal on Computing*, vol. 25, no. 2, pp. 244–255, 2013.

[7] J. Leung, M. Dror, and G. Young, "Technical note: A note on an open-end bin packing problem," *Journal of Scheduling*, vol. 4, pp. 201–207, 2001.

[8] P. Ongkunaruk, *Asymptotic Worst-Case Analyses for the Open Bin Packing Problem*. Phd thesis, Virginia Polytechnic Institute and State University, 2005.

[9] E. G. Coffman, M. R. Garey, and D. S. Johnson, *Algorithm Design for Computer System Design*, ch. Approximation algorithms for bin-packing – an updated survey, pp. 49–106. Springer, 1984.

[10] D. Simchi-Levi, "New worst-case results for the bin-packing problem," *Naval Research Logistics*, vol. 41, no. 4, pp. 579–585, 1994.

[11] J. N. D. Gupta and J. C. Ho, "A new heuristic algorithm for the one-dimensional bin-packing problem," *Production Planning & Control*, vol. 10, no. 6, pp. 598–603, 1999.

[12] E. Falkenauer, "A hybrid grouping genetic algorithm for bin packing," *Journal of Heuristics*, vol. 2, no. 1, pp. 5–30, 1996.

[13] C. R. Reeves, "Hybrid genetic algorithms for bin-packing and related problems," *Annals of Operations Research*, vol. 63, no. 3, pp. 371–396, 1996.

[14] A. Stawowy, "Evolutionary based heuristic for bin packing problem," *Computers & Industrial Engineering*, vol. 55, no. 2, pp. 465–474, 2008.

[15] T. Dokeroglu and A. Cosar, "Optimization of one-dimensional bin packing problem with island parallel grouping genetic algorithms," *Computers & Industrial Engineering*, vol. 75, pp. 176–186, 2014.

[16] T. Kucukyilmaz and H. E. Kiziloz, "Cooperative parallel grouping genetic algorithm for the one-dimensional bin packing problem," *Computers & Industrial Engineering*, vol. 125, pp. 157–170, 2018.

[17] K. Fleszar and K. S. Hindi, "New heuristics for one-dimensional bin-packing," *Computers & Operations Research*, vol. 29, no. 7, pp. 821–839, 2002.

[18] V. Hemmelmayr, V. Schmid, and C. Blum, "Variable neighbourhood search for the variable sized bin packing problem," *Computers & Operations Research*, vol. 39, no. 5, pp. 1097–1108, 2012.

[19] L. M. Santos *et al.*, "A variable neighborhood search algorithm for the bin packing problem with compatible categories," *Expert Systems with Applications*, vol. 124, pp. 209–225, 2019.

[20] J. Levine and F. Ducatelle, "Ant colony optimization and local search for bin packing and cutting stock problems," *Journal of the Operational Research Society*, vol. 55, pp. 705–716, 2004.

[21] R. M. Karp, M. Luby, and A. Marchetti-Spaccamela, "A probabilistic analysis of multidimensional bin packing problems," in *Proceedings of the Sixteenth Annual ACM Symposium on Theory of Computing*, p. 289–298, Association for Computing Machinery, 1984.

[22] S. Martello, D. Pisinger, and D. Vigo, "The three-dimensional bin packing problem," *Operations Research*, vol. 48, no. 2, pp. 256–267, 2000.

[23] D. K. Friesen and M. A. Langston, "Variable sized bin packing," *SIAM Journal on Computing*, vol. 15, no. 1, pp. 222–230, 1986.

[24] M. Gendreau, G. Laporte, and F. Semet, "Heuristics and lower bounds for the bin packing problem with conflicts," *Computers & Operations Research*, vol. 31, no. 3, pp. 347–358, 2004.

[25] A. Ekici, "Bin packing problem with conflicts and item fragmentation," *Computers & Operations Research*, vol. 126, pp. 105–113, 2021.

[26] N. Karmarkar and R. M. Karp, "An efficient approximation scheme for the one-dimensional bin-packing problem," in *23rd Annual Symposium on Foundations of Computer Science*, 1982.

[27] J. Yang and J. Leung, "The ordered open-end bin packing problem," *Operations Research*, vol. 51, no. 5, pp. 759–770, 2003.

[28] A. Ceselli and G. Righini, "An optimization algorithm for the ordered open-end bin-packing problem," *Operations Research*, vol. 56, no. 2, pp. 425–436, 2008.

[29] M. Mohamed, T. Mohamed, and R. Billal, "Modeling and solving the open-end bin packing problem," *International Journal of Advanced Computer Science and Applications*, vol. 7, pp. 399–404, 2016.

[30] A. E. F. Muritiba, M. Iori, E. Malaguti, and P. Toth, "Algorithms for the bin packing problem with conflicts," *INFORMS Journal on Computing*, vol. 22, no. 3, pp. 401–415, 2010.

[31] N. Mladenović and P. Hansen, "Variable neighborhood search," *Computers & Operations Research*, vol. 24, no. 11, pp. 1097–1100, 1997.

[32] S. Martello and P. Toth, "Lower bounds and reduction procedures for the bin packing problem," *Discrete Appied Mathematics*, vol. 28, no. 1, pp. 59–70, 1990.

[33] D. Johnson, "Approximation algorithms for combinatorial problems," *Journal of Computer and System Sciences*, vol. 9, no. 3, pp. 256–278, 1974.

[34] P. Hansen, N. Mladenovic, J. Brimberg, and J. M. Pérez, *Handbook of Metaheuristics*, ch. Variable Neighborhood Search, pp. 61–86. Springer, Boston, MA, 2010.

[35] E. Falkenauer and A. Delchambre, "A genetic algorithm for bin packing and line balancing," in *Proceedings IEEE International Conference on Robotics and Automation*, pp. 1186–1192 vol.2, 1992.

[36] E. Falkenauer, "A new representation and operators for genetic algorithms applied to grouping problems," *Evolutionary Computation*, vol. 2, pp. 123–144, 1994.

[37] A. H. Kashan, A. A. Akbari, and B. Ostadi, "Grouping evolution strategies: An effective approach for grouping problems," *Applied Mathematical Modelling*, vol. 39, no. 9, pp. 2703–2720, 2015.

[38] M. Maiza, A. Labed, and M. S. Radjef, "Efficient algorithms for the offline variable sized bin-packing problem," *Journal of Global Optimization*, vol. 57, no. 3, pp. 1025–1038, 2013.

VITA

Ece Nur Balık graduated from Kadıköy Anatolian High School (KAL) in 2016. She earned her B.Sc. degree in Industrial Engineering from Özyeğin University in June 2020. She has been pursuing her M.Sc. degree in Industrial Engineering at Özyeğin University since October 2020, under the supervision of Dr. Ali Ekici. Her research interests include logistics and supply chain management.