IEEE Access
Multidisciplinary : Rapid Review : Open Access Journal

Received July 9, 2021, accepted August 30, 2021, date of publication September 3, 2021, date of current version September 10, 2021.

Digital Object Identifier 10.1109/ACCESS.2021.3109976

A Heuristic Approach to the Design of Optimal Cross-Docking Boxes

ROBERT NIEUWENHUIS, ALBERT OLIVERAS, AND ENRIC RODRÍGUEZ-CARBONELL
Computer Science Department, Technical University of Catalonia, 08034 Barcelona, Spain

Corresponding author: Albert Oliveras (oliveras@cs.upc.edu)

This work was supported in part by the Spanish Ministerio de Ciencia e Innovación (MICINN) Project under Grant RTI2018-094403-B-C33.

ABSTRACT Multinational companies frequently work with manufacturers that receive large orders for different products (or product varieties: size, shape, color, texture, material), to serve thousands of different final destinations (e.g., shops) requesting a combination of different quantities of each product. It is not the manufacturers' task to create the individual shipments for each final destination. But manufacturers can deliver part of their production in so-called *cross-docking* boxes (or other containers) of a few (say, three) types, each type containing a given assortment, i.e., different quantities of different products. At a logistics center, the shipments for each destination are then made of cross-docking boxes plus additional "picking" units. The expensive part is the picking, since cross-docking boxes require little or no manipulation. The problem we solve in this paper is, given a large set of orders for each destination, to design the cross-docking box types in order to minimize picking. We formally define a variant of this problem and develop a heuristic method to solve it. Finally, we present extensive experimental results on a large set of real-world benchmarks proving that our approach gives high-quality solutions (optimal or near optimal) in a very limited amount of time.

INDEX TERMS Artificial intelligence, combinatorial optimization, heuristic algorithms, logistics.

I. INTRODUCTION

It is widely accepted that the logistics industry plays a major role in today's economy [1]. Both in emerging markets and in advanced economies well-designed logistic processes may give important competitive advantages to companies or regions [2]. Cross-docking is one of these important processes in logistics that should be particularly well-designed [3].

Products rarely travel directly from the manufacturer to their final destination. Logistic centers are intermediate destinations where products are stored and combined in order to be later distributed. However, the ideal situation is one where pallets or boxes in which incoming products have been packed do not have to be stored or opened, but instead can be directly sent to their final destinations. In order to maximize the number of products that can be dispatched in this quick and inexpensive way, the key ingredient is the design of appropriate compositions of boxes where products are to be packed. An alternative would be to design a huge number of different boxes, but unfortunately, most manufacturers only accept to pack products in a very limited number of different compositions.

A promising approach to finding a convenient set of cross-docking boxes is to use Combinatorial Optimization methods [4]. Combinatorial Optimization problems consist in finding an optimal object from a finite set of objects. Approaches to Combinatorial Optimal could be divided into complete and incomplete methods. Given enough time, complete methods always end up computing the optimal solution. Hence, they somehow perform an exhaustive search. However, sometimes the search space is too large to be fully explored and one has to resort to incomplete methods, that do not always compute the optimal solution, but are expected to give good solutions in a reasonable amount of time.

In order to obtain a complete method, one can design an ad hoc algorithm for a concrete Combinatorial Optimization problem. This has the drawback that all implementation work has to be done from scratch. In order to obtain a competitive method, it will be necessary to spend a huge amount of time improving the implementation and, if the problem varies a little bit, it is very likely that most of this

The associate editor coordinating the review of this manuscript and approving it for publication was Vivek Kumar Sehgal.

This work is licensed under a Creative Commons Attribution 4.0 License. For more information, see https://creativecommons.org/licenses/by/4.0/

work on low-level implementation details will have to be repeated again. An alternative to this scenario is to translate our problem into a different one for which very efficient implementations already exist. We want to highlight three remarkable options with huge success stories behind them:

- **Integer Linear Programming (ILP) [5]:** given a set of linear constraints and an objective function (a linear expression), we search for an integer assignment to the variables that satisfies all linear constraints and minimizes the objective function.
- **SAT [6]:** given a Boolean formula, constructed over a set of Boolean variables and connectives *not* ($\neg$), *and* ($\wedge$), *or* ($\vee$), we search for a Boolean assignment to the variables that satisfies the formula.
  Even though this is a feasibility problem, one can solve, e.g., minimization problems by constructing a series of successive SAT problems F_k, where F_k expresses that we look for a solution with cost at most k.
- **Satisfiability Modulo Theories (SMT) [7]:** we are given a (quantifier-free) first-order formula and we have to determine its satisfiability modulo a background theory $\mathcal{T}$. In this paper, $\mathcal{T}$ will be the theory of the integers and formulas will be built from (non-)linear constraints that will be combined via the Boolean connectives $\neg$, $\wedge$, $\vee$.

Regarding incomplete methods, a prominent example is the one of metaheuristics [8]. As opposed to tailored local-search approaches for a concrete problem, metaheuristics are high-level independent algorithmic frameworks that can be applied to a large variety of problems. Some metaheuristic examples are: GRASP [9], Tabu Search [10], Simulated Annealing [11], Variable Neighborhood Search [12] or Ant Colony Optimization [13].

The goal of this paper is to solve the problem of designing optimal cross-docking boxes using some of the methods we have mentioned before. More precisely, its main contributions are:

- We precisely define the problem of designing optimal cross-docking boxes.
- We introduce a non-linear integer programming formulation of the problem.
- We present, at a sufficient level of detail, a Variable-Neighborhood-Search-like metaheuristic for this problem.
- We report experimental results on real-world benchmarks. We study the impact of some decision designs and evaluate the quality of our approach by comparing it with an SMT-based complete approach.

The rest of the paper is organized as follows: in Section II we study related work. In Section III we give formal and informal presentations of the problem, and prove its NP-hardness. After that, in Section IV we introduce an incomplete metaheuristic-based approach for solving it. Section V reports on exhaustive experimental results on a very large set of real-world benchmarks and we conclude in Section VI.

II. RELATED WORK

Existing research aimed at improving the effectiveness of cross-docking in industry is both large and diverse. Since providing an exhaustive list is out of the scope of this paper, we will only mention some representative works in order to get a grasp of the diversity of the problems studied and their corresponding solving methods. Existing efforts on improving cross-docking can be classified according to three decision levels [14]. In the *strategic level*, long-term decisions are taken, such as determining the locations and the amount of cross-docking nodes. The seminal work of [15] explored a mixed-integer linear programming model for determining the location and the number of cross-docks in a load-driven system. In [16], again a mixed-integer programming model is proposed in order to decide which warehouses and cross-docks are opened. In this particular case, the suggested solving method is based on Simulated Annealing. The use of heuristic methods for these problems is not rare, and techniques such as Tabu Search or Particle Swarm Optimization are used in [17] and [18], for example.

The *tactical level* addresses problems with an impact on the mid-term horizon. Most work on this level is devoted to the design of the best layout in the cross-dock. This includes defining the shape of the facility, the number of doors, the material flow or the temporary storage area location. One interesting work is [19], where a genetic algorithm is used to minimize the labor workload and the lead time in a manufacturing industry context. A different modeling paradigm is used in [20], where determining the storage locations that minimize the forklift trucks travel distances is modeled as a minimum-cost-flow problem.

Finally, the *operational level* covers decisions in the short-term planning horizon (days or weeks). Research in this level can be divided into five areas [14]: scheduling, transshipment, dock door assignment, product allocation and vehicle routing. Scheduling is a very broad and prolific area, whose aim is to define the sequence of inbound and outbound trucks at a given set of dock doors. The case with one inbound and one outbound truck is studied in [21], where a polynomial approximation algorithm is given, as well as a branch-and-bound algorithm that is suitable for middle-sized problems. Transshipment, as defined by [22], answers four questions: how much to ship, between which locations, at what times and on which routes. The first study on dock door assignment was [23], where a door assignment is sought that minimizes forklifts travel distance. The solving technology they propose is a microcomputer-based tool based on bilinear programming. Some other works, such as [24], also model the problem as a non-linear program, but a heuristic method based on a genetic algorithm is used. Product allocation is the least studied of all five areas. We would like to mention [25], where the problem is to determine which products or which percentage of a certain product should go through by cross-docking. Finally, there are very few works about vehicle routing in the context of cross-docking, probably due to the vast amount of existing literature for general vehicle routing. One example

is [26], where a tabu-search heuristic is presented to minimize travel distance while respecting time-window constraints.

All in all, it is clear that cross-docking has been studied from multiple perspectives. The derived problems are countless, as well as the solving methods being used. However, to the best of our knowledge, the problem we will introduce, which should be placed in the operational level, has not been studied elsewhere.

III. PROBLEM DEFINITION

In this section, we first give an informal presentation of the problem, with some illustrative examples. Then, we formally define it by giving a non-linear integer programming model. Finally, we prove the NP-hardness of the problem.

A. INFORMAL PRESENTATION

Let us consider a company that manufactures a certain finite set of products. Periodically, customers place orders, indicating how many units of each product they request. In order to dispatch the orders we are given a set of box templates, which specify the minimum and maximum number of product units and the mandatory products in each box.

For each template we need to construct a box: determine the exact number of units of each product it will contain, taking into account the minimum and maximum, and the mandatory products. Each order should be exactly obtained by choosing a set of boxes plus some additional products that will be added manually, the so-called picking. Among all possible boxes, we are generally interested in the ones that allows us to minimize the picking.

Example 1: Let us consider the set of products {A,B,C,D} and the following three orders:

	A	B	C	D
Order $O1$	1	2	3	1
Order $O2$	1	1	1	1
Order $O3$	3	4	7	2

If we have two box templates, both with a minimum of 4 units and a maximum of 20, and no mandatory product in them, the boxes that minimize picking are

	A	B	C	D
Box $B1$	1	2	3	1
Box $B2$	1	1	1	1

Let us first compute the picking we can obtain with these boxes. We can serve order $O1$ with one box $B1$, and no picking is required. Similarly, $O2$ can be served with one box $B2$, again with no picking. Finally, order $O3$ can be served with two boxes $B1$, which gives a picking of 2 units: one unit of product A and one unit of C. This is indeed the optimal solution. □

However, in order to further reduce picking, we are allowed to increase the number of units of products in the orders. More concretely, for each order we know in how many units each product can be incremented, and also the maximum amount of increments across all products in this order. Globally, there is an additional maximum amount of increments for each product to be used among all orders.

Example 2: Let us consider our previous example and now allow 1 unit of increment in all orders and products, at most 1 unit in each order and at most 2 units for each product globally.

This allows us to increase 1 unit of C in $O2$ and 1 unit of D in $O3$:

	A	B	C	D
Order $O1$	1	2	3	1
Order $O2$	1	1	2	1
Order $O3$	3	4	7	3

In this situation, the optimal boxes are

	A	B	C	D
Box $B1$	1	2	3	1
Box $B2$	1	1	2	1

since $O1$ can be served with $B1$, order $O2$ is served with $B2$ and $O3$ can be obtained with 1 box $B1$ and 2 boxes $B2$, hence incurring in no picking at all. □

B. MATHEMATICAL FORMULATION

In order to give a precise mathematical formulation of our problem, let us consider the following sets:

- $\mathcal{P}$: set of products
- $\mathcal{O}$: set of orders
- $\mathcal{T}$: set of box templates

and the following input data, with their type indicated below:

- K: ($\mathbb{R}$) relative cost of one cross-docking box w.r.t. one unit of picking
- $unitsOrder_{o,p}$: ($\mathbb{N}$) number of units of product p in order o, for each $o \in \mathcal{O}, p \in \mathcal{P}$
- $minUnits_t$: ($\mathbb{N}$) minimum number of product units in template t, for each $t \in \mathcal{T}$
- $maxUnits_t$: ($\mathbb{N}$) maximum number of product units in template t, for each $t \in \mathcal{T}$
- $required_t$: (set of ($\mathcal{P}$)) set of products that should be present in template t, for each $t \in \mathcal{T}$
- $maxInc_{o,p}$: ($\mathbb{N}$) maximum allowed increment for product p in order o, for each $o \in \mathcal{O}, p \in \mathcal{P}$
- $maxInc_o$: ($\mathbb{N}$) maximum allowed increment in order o, for each $o \in \mathcal{O}$
- $totalMaxInc_p$: ($\mathbb{N}$) maximum allowed increment in product p, for each $p \in \mathcal{P}$

We will now introduce the following integer variables in order to formally model our problem. As mentioned, for each box template $t \in \mathcal{T}$, we will have to specify a box that will be referred to as a *box of type* t.

- $\nu(o,t)$: ($\mathbb{N}$) number of boxes of type t assigned to order o, for each $o \in \mathcal{O}, t \in \mathcal{T}$
- $\beta(t,p)$: ($\mathbb{N}$) number of units of product p in box of type t, for each $t \in \mathcal{T}, p \in \mathcal{P}$

$\pi(o,p)$: picking resulting from product p in order o,
($\mathbb{N}$) for each $o \in \mathcal{O}, p \in \mathcal{P}$
$\delta(o,p)$: increment of product p in order o,
($\mathbb{N}$) for each $o \in \mathcal{O}, p \in \mathcal{P}$

Note that all variables are non-negative integers.

We can now model our problem as the following non-linear program:

$$min \sum_{\substack{o\in\mathcal{O}\\ p\in\mathcal{P}}} \pi(o,p) + K \cdot \sum_{\substack{o\in\mathcal{O}\\ t\in\mathcal{T}}} \nu(o,t)$$

$$\text{subject to} \sum_{t\in\mathcal{T}} (\nu(o,t)\cdot\beta(t,p)) + \pi(o,p) = unitsOrder_{o,p} + \delta(o,p) \quad \forall o \in \mathcal{O}, p \in \mathcal{P} \tag{1}$$

$$minUnits_t \le \sum_{p\in\mathcal{P}} \beta(t,p) \le maxUnits_t \quad \forall t \in \mathcal{T} \tag{2}$$

$$\beta(t,p) > 0 \quad \forall t \in \mathcal{T}, p \in required_t \tag{3}$$

$$\delta(o,p) \le maxInc_{o,p} \quad \forall o \in \mathcal{O}, p \in \mathcal{P} \tag{4}$$

$$\sum_{p\in\mathcal{P}} \delta(o,p) \le maxInc_o \quad \forall o \in \mathcal{O} \tag{5}$$

$$\sum_{o\in\mathcal{O}} \delta(o,p) \le totalMaxInc_p \quad \forall p \in \mathcal{P} \tag{6}$$

The objective function considers the total picking and the number of boxes assigned to orders. The constant K allows one to express which is the relative cost of one cross-docking box w.r.t. the cost of one unit of picking. If $K = 0$ then we would model the cost of Example 1, where only picking was considered. Otherwise, its semantics is that K units of picking have the same cost as one cross-docking box.

Equation 1 expresses that units in an order are split into cross-docking boxes, increments and picking. Note that this is the only non-linear component of the model. Equations 2- 3 express that boxes should satisfy the requirements of the templates w.r.t. minimum/maximum amount of units and mandatory products. Equations 4-6 impose that the three limitations on increments are fulfilled.

Note that, in general, non-linear integer programming is an undecidable problem [27]. However, we can see that in our case there is only one non-linear constraint, and the only non-linear multiplications are of the form $\nu(o,t) \cdot \beta(t,p)$. Fortunately, variables $\beta(t,p)$ are all upper and lower bounded and this makes the problem decidable: we can build a finite set of linear programs by instantiating these variables in all possible ways and take the best solution among all programs. Although this does not seem to be a practical approach, we will see in Section V that it is the basis of a complete method for this type of problems.

C. HARDNESS OF THE PROBLEM

Let us consider *Unbounded Subset Sum Problem* (USSP) [28]: given n different positive integers $\{a_1, a_2, \cdots, a_n\}$ and a positive integer b, we want to determine whether there exist integers $x_i \ge 0$ such that $\sum_{i=1}^{n} a_i x_i = b$. This problem is known to be NP-hard [28].

We will prove that the decision version of our problem, where we ask for a zero-cost solution, is NP-hard by reducing USSP to it. Given an instance of USSP, we will build the following instance of our problem:

- $\mathcal{P} : \{p\}$ (one product)
- $\mathcal{O} : \{o\}$ (one order)
- $\mathcal{T} : \{t_1, t_2, \cdots, t_n\}$
- $K = 0$
- $unitsOrder_{o,p} = b$
- $minUnits_{t_i} = a_i$, for all $1 \le i \le n$
- $maxUnits_{t_i} = a_i$, for all $1 \le i \le n$
- $required_{t_i} = \emptyset$, for all $1 \le i \le n$
- $maxInc_{o,p} = 0$
- $maxInc_o = 0$
- $totalMaxInc_p = 0$

Intuitively, since there are no increments available, and no picking is allowed (because we look for a zero-cost solution), we have to serve the order only with the boxes. Since we only have one product, this amounts to obtain the integer t via a non-negative linear combination of the a_i's.

Algorithm 1 - Function `VNS`

```
1: Returns: <Boxes bestBoxes, Solution bestSol>
2: Identify identical orders and build set of orders O with multi-
   plicity
3: Order O according to some criterion
4: bestSol.cost ← ∞
5: while not timeLimit do
6:     B ← generateBoxes()
7:     Solution localSol ← computeSol(B)
8:     n1 ← 1; n2 ← 1; large ← false
9:     while not timeLimit do
10:        <tmpBoxes,tmpSol> ← bestNeighbor(n1,n2,B)
11:        if tmpSol.cost < localSol.cost then
12:            localSol ← tmpSol; B ← tmpBoxes
13:            large ← false;
14:            n1← 1; n2← 1;
15:        else if not large then
16:            large ← true
17:            n1← 5; n2← 5;
18:        else break
19:    if localSol.cost < bestSol.cost then
20:        bestSol ← localSol; bestBoxes ← B
21: return <bestBoxes,bestSol>
```

IV. SOLVING METHOD

When solving hard combinatorial problems, it is important to know how much time can we afford to solve them. In problems such as designing the next-month schedule of the employees of a company, the available solving time might be of up to several hours. In other cases, thousands of problems need to be solved every day and hence we can only spend some seconds in each of them.

The real-world application that inspired this problem requires a very short solving time. This, together with the fact that the problem is NP-hard, led us to consider an incomplete

Algorithm 2 - Function `generateBoxes`

```
 1: Returns: Boxes boxes
 2: for t ∈ 𝒯 do
 3:     Box b;
 4:     for p ∈ t.required do b[p] ← 1
 5:     unitsInBox ← |t.required|
 6:     toPlace ← rand(t.min,t.max) - unitsInBox
 7:     while toPlace > 0 do
 8:         p ← randomly select one product
 9:         b[p] ← b[p] + 1
10:         toPlace ← toPlace - 1
11:     boxes ← boxes ∪ {b}
12: return boxes
```

method, in which we may sacrifice quality solution in order to obtain a quick response.

Among all incomplete optimization methods we have chosen a metaheuristic, namely a variant of Variable Neighborhood Search (VNS). Our overall approach can be explained as a two-step process. In the first step, for each template $t \in \mathcal{T}$ we build a box, thus giving a set of boxes B. In the second step, an exhaustive search component is in charge of determining how to serve all orders using boxes B. The VNS determines how these two steps are interleaved.

In Algorithms 1-3 a high-level view of the method is described. The main function is `VNS` in Algorithm 1, which starts exploiting that it is very common that different customers place the same orders. All these identical orders are identified, and a new set of orders with their corresponding multiplicities is created. After that, these new orders are ordered according to some criterion. Details about which criteria are useful will be discussed in Section V.

After this preprocessing, the search for a solution starts. Apart from the selected boxes, a solution will contain the cost and for each order: (i) the number of boxes of each type assigned to the order, and (ii) the increments in the demand of each product. The solution search starts by creating a random set of boxes. Then, in Line 7, the exhaustive search component is called, which returns a solution using these boxes. After this initial step, better solutions are searched by considering boxes in the neighborhood and the solutions one can obtain with them. The parameters $n1$ and $n2$ specify how large the neighborhood is. In Algorithm 1 the Boolean variable *large* expresses whether to search in a small or a large neighborhood, being the large one explored only if no improvement is possible within the small one. This is done until a certain time limit is exceeded.

In Algorithm 3 a detailed description of the function exploring the neighborhood of a solution is given. The basic idea is to improve the solution by slightly modifying the boxes configuration. Three possibilities are considered:

- Select two boxes $b1$ and $b2$, two products $p1$ and $p2$ and then (i) decrement a certain amount of units of $p1$ in $b1$, and (ii) increment the same amount of units of $p2$ in $b2$. The amount of units moved is upper bounded by $max1$. This is done for all boxes and products. Note that $b1$ could be equal to $b2$ and hence we allow moving units within the same box. Of course, the resulting boxes are explored only if they meet the constraints of their corresponding box templates.
- Select one box b and one product p and decrement some units of these products. The amount of units decremented is upper bounded by $max2$.
- Select one box b and one product p and increment some units of these products. The amount of units decremented is upper bounded by $max2$.

Algorithm 3 - Function `bestNeighbor`

```
 1: Input: int max1, int max2, Boxes B
 2: Returns: <Boxes bestBoxes, Solution bestSol>
 3: bestSol ← computeSol(B)
 4: bestBoxes ← B
 5: for b1 ∈ B, p1 ∈ 𝒫 do                    ▷ First type of neighbor
 6:     top ← min(max1,b1[p1])
 7:     for chg ∈ 1...top do
 8:         for b2 ∈ B, p2 ∈ 𝒫 do
 9:             b1[p1] ← b1[p1] - chg
10:             b2[p2] ← b2[p2] + chg
11:             if correct(b1) ∧ correct(b2) then
12:                 sol ← computeSol(B)
13:                 if sol.cost < bestSol.cost then
14:                     bestSol ← sol
15:                     bestBoxes ← B;
16:             b1[p1] ← b1[p1] + chg
17:             b2[p2] ← b2[p2] - chg
18: for b ∈ B, p ∈ 𝒫 do                      ▷ Second type of neighbor
19:     top ← min(max2,b[p])
20:     for chg ∈ 1...top do
21:         b[p] ← b[p] - chg
22:         if correct(b) then
23:             sol ← computeSol(B)
24:             if sol.cost < bestSol.cost then
25:                 bestSol ← sol
26:                 bestBoxes ← B;
27:         b[p] ← b[p] + chg
28: for b ∈ B, p ∈ 𝒫 do                      ▷ Third type of neighbor
29:     for chg ∈ 1...max2 do
30:         b[p] ← b[p] + chg
31:         if correct(b) then
32:             sol ← computeSol(B)
33:             if sol.cost < bestSol.cost then
34:                 bestSol ← sol
35:                 bestBoxes ← B;
36:         b[p] ← b[p] - chg
37: return <bestBoxes,bestSol>
```

As we can see from Algorithm 3 all possibilities are explored. Observe that by simply changing the parameters $max1$ and $max2$ one can explore different neighborhoods. This would allow us change the core loop of Algorithm 1 in order to explore not only two neighborhoods, as it is done in this presentation, but rather an increasingly large set of neighborhoods. Also, note that Algorithm 3 works in a *best-improvement* manner. That is, it explores the whole neighborhood and returns the best solution in it. We could easily change it to work in a *first-improvement* way, stopping

Algorithm 4 - Function `computeSol`

```
1:  Input: Boxes B
2:  Returns: Solution sol
3:  sol.cost ← 0                                         ▷ Total cost
4:  for o ∈ Orders do
5:      maxIncO ← min(∑_{p∈P} maxInc[o][p], maxInc[o])
6:      bestC ← ∞
7:      optOrder(B,1,o,maxIncO,maxInc[o],assign,incP);
8:      assiInc ← bestA
9:      incrs ← bestI
10:     cInc ← bestC
11:     nInc ← o.multiplicity
12:     for p ∈ P do
13:         if nInc · bestI[p] > totMaxInc[p] then
14:             nInc ← ⌊ totMaxInc[p]/bestI[p] ⌋
15:     for p ∈ P do
16:         totMaxInc[p] = totMaxInc[p] - nInc·bestI[p];
17:     nNoInc ← o.multiplicity - nInc;
18:     if nNoInc ≠ 0 then
19:         bestC ← ∞
20:         optOrder(B,1,o,0,maxInc[o],assign,incP)
21:         assiNoInc ← bestA
22:         cNoInc ← bestC
23:     sol[o] ← construct(nInc,assiInc,assiNoInc,incrs)
24:     sol.cost ← sol.cost + cInc·nInc + cNoInc·nNoInc
25: return sol
```

as soon as it finds a better solution. This possibility will be evaluated in Section V.

Let us now focus on the most complex part of the algorithm: the exhaustive part component described in Algorithms 4 and 5. As we have done so far, we will present the algorithms in a top-down way. Function `computeSol` receives a set of boxes B and returns a solution using these boxes. The first important remark is that this function might not return the optimal way to use these boxes. However, as we will see in the experiments, the solution quality is extremely good.

The algorithm processes the orders one by one. For each order, we compute *maxIncO* (the maximum number of unit increments we can assign to this order) and use function `optOrder` to make two computations. First of all, in line 7, we compute the best way to serve this order using boxes B and considering the increment limits *maxIncO* and *maxInc*[o], which defines the maximum increment units for each product in order o. The meaning of the additional parameters 1, *assign* and *incP* will be explained later. As a result of this call, the best way to serve this order allowing increments will be stored in global variables *bestA* (best assignment), *bestI* (best increments) and *bestC* (best cost).

Note that this first computation ignores *totalMaxInc*, which defines a global limit on the number of increments we can assign to each product. To address this fact, we compute the maximum number of orders of this type than can be served with the computed increments (remember that the order has a certain multiplicity). This is done in lines 12-14.

The remaining orders, as many as nNoInc (line 17), will be assigned with no increments. This computation is done in line 20. Finally, in line 23 these two ways to serve the order are combined and stored in the solution. That is, the first *nInc* orders will be served with the solution that considers increments and the remaining ones with the solution with no increments.

It is easy to see that this function might not compute the best way to serve all orders using boxes B, since orders are processed one by one, and the first orders will have more possibilities to be served with increments than the last ones. Obviously, this does not always correspond to the optimal solution.

Let us finish the presentation of the algorithm with function `optOrder`. Given an order o, the maximum number of unit increments available in total (*maxIncO*) and per product (*maxIncP*), it computes the optimal way of serving order o using boxes B. The algorithm performs an exhaustive search starting with the first box type (this is why parameter *bId* is 1 in the two calls to this function in Algorithm 4). All options for serving part of the order with a certain number of boxes of this type are considered, including the possibility of incrementing the order. For each such option, the part of the order which has not yet been served is dispatched with the rest of the box types we have not used yet. Once all options have been explored, the best possibility is stored. Although non-recursive implementations of this idea are possible, we present a recursive pseudo-code that is as close as possible to the code we have used for the experimental evaluation in Section V.

This overall idea is presented in detail in Algorithm 5. At any call to function `optOrder`, a partial solution considering boxes $B[1 \ldots bId - 1]$ is stored in *assign* (which determines the number of boxes of each type that are used) and in *incP* (which determines how many increments for each product we have used in the partial solution we are extending). Lines 7-24 consider the case in which *bId* does not refer to the last box, whereas lines 26-46 consider the last-box treatment. In the first case, lines 9-12 assign from 0 up to M boxes of type $B[bId]$ and select the best of these possibilities. After that, increments are considered in the loop in lines 13-24. Note that when we exit the previous loop, exactly $M+1$ boxes have been assigned, and hence there are some products for which the order is now negative and should be incremented. Line 15 checks whether incrementing these products would violate some increment limits. If this is the case, we stop considering increments (line 24). Otherwise, lines 16-23 compute the cost of adding one additional box and using the necessary increments. After that, one additional box is assigned and we loop again.

The treatment of the last box follows along the same lines. In lines 26-31, the situation in which no increments are used is computed. Note that, since there are no more boxes left, we only consider assigning the maximum number of possible boxes of this type. We compute picking (line 27) and the number of boxes used (line 28) in order to evaluate the cost of this solution. If this improves upon the best solution found so far, which is stored in global variables *bestC* (cost), *bestA* (assignment) and *bestI* (increments), we update the

Algorithm 5 - Function `optOrder`

```
1: Input: Boxes B, int bId, Order O, int maxIncO,
2:        int maxIncP[p ∈ 𝒫], int assign[1...nBoxes]
3:        int incP[p ∈ 𝒫]
4: Returns: int cost
5: Global: int bestC, int bestA[1...nBoxes], int bestI[p∈𝒫]
6: M ← num of boxes B[bId] that fit in order O
7: if bId < nBoxes then                                ▷ Not last box
8:     assign[bId] ← 0; cost ← inf
9:     for k in 0...M do
10:        cost ← min(cost,optOrder(B,bId+1,O,...))
11:        for (p ∈ 𝒫) O[p] ← O[p] - B[bId][p]
12:        assign[bId] ← assign[bId] + 1
13:    while true do
14:        sumNegs ← − Σ_{p∈𝒫 with O[p]<0} O[p]
15:        if ⋀_{p∈𝒫 with O[p]<0} (-O[p] ≤ maxIncP[p]) ∧ sumNegs ≤ maxIncO
   then
16:            maxIncO ← maxIncO − sumNegs
17:            for p ∈ 𝒫 with O[p] < 0 do
18:                incP[p] ← incP[p] − O[p]
19:                maxIncP[p] ← maxIncP[p] + O[p]
20:                O[p] ← 0
21:            cost ← min(cost,optOrder(B,bId+1,O,...))
22:            for p ∈ 𝒫 do O[p] ← O[p] − B[bId][p]
23:            assign[bId] ← assign[bId] + 1
24:        else   break
25: else                                              ▷ Treat last box
26:    assign[bId] ← M
27:    pick ← Σ_{p∈𝒫}(O[p] - M · B[bId][p])
28:    boxes ← Σ_{1≤i≤nBoxes} assign[i]
29:    cost ← pick + K·boxes
30:    if cost < bestC then
31:        bestC ← cost; bestA ← assign; bestI ← incP
32:    repeat ← (maxIncO > 0); extra ← 1;
33:    while repeat do
34:        for p ∈ 𝒫 do
35:         locIncP[p] ← max(0,(M+extra)·B[bId][p]-O[p])
36:        inc ← Σ_{p∈𝒫} locIncP[p]
37:        if ⋀_{p∈𝒫} (locIncP[p] ≤ maxIncP[p]) ∧ inc ≤ maxIncO) then
38:            assign[bId] ← M + extra
39:            pick ← Σ_{p∈𝒫}(O[p]+locIncP[p]-(M+extra)·B[bId][p]))
40:            cost ← min(cost,pick + K·Σ_{1≤i≤nBoxes} assign[i])
41:            for p ∈ 𝒫 do fIncP[p] ← incP[p] + locIncP[p]
42:            if cost < bestC then
43:                bestC ← cost; bestA ← assign; bestI ← fIncP
44:            repeat ← (inc < maxIncO)
45:        else  repeat ← false
46:        extra ← extra + 1
47: return cost;
```

information, as done in line 31. After that, we start exploring the possibility of using increments in order to use extra boxes. While this is possible, we compute the units to be incremented (lines 35-36). If this does not exceed the maximum available increments and for no product we exceed its increment limit (checked in line 37) we assign one additional box, compute the cost of this solution and update the necessary data structures.

V. EXPERIMENTAL EVALUATION

We will start our experimental evaluation presenting some data about the benchmarks we will use for the analysis. Then, we will show the impact of different design decisions and evaluate the quality of the solutions given by our approach.

A. BENCHMARKS

We have done our experimental evaluation on a set of 11521 benchmarks that correspond to the problems generated by a logistics company over a certain period of time. We consider that dealing with real-world benchmarks as opposed to randomly generated ones is highly beneficial because certain characteristics, such as the number of box templates or the number of products definitely have an impact on the performance of different solving methods.

Let us start examining some data about the benchmarks. First of all, let us focus on the number of box templates: 5858 benchmarks only had 1 box template, 4899 had 3 templates, whereas 764 had 4 templates. Hence, we can see that on the problems generated by this concrete company, the number of box templates is relatively low.

Secondly, another important measure is the number of orders. In 6077 benchmarks there are less than 50 orders. The number of orders for the remaining benchmarks varies a lot. The information is displayed in Figure 1, on the left histogram. We can observe that, despite there are more benchmarks with around 100 orders, benchmarks with up to 1200 orders are not rare, and they can even go up to 1700.

Finally, we want to evaluate the number of products in each benchmark. Numbers range from 1 to 8 and details can be seen on the right histogram in Figure 1. We can see that almost half of the benchmarks contain 4 different products.

B. RESULTS

In this section we will study the impact of some variants of the algorithm presented in the previous section and evaluate the quality of the solutions computed by our approach.

1) SORTING THE SET OF ORDERS

As we mentioned in Section IV, in our heuristic method identical orders are identified and a set of orders with multiplicity is created. Then, this set of orders is sorted according to some criteria. In the following we evaluate the impact of changing the sorting criterion.

The first criterion we considered was to compute, for each order, the product $number_units * multiplicity$, and sort the orders decreasingly w.r.t. this measure. The idea is that we want to use the available increments for orders that account for a large number of product units and hence these should be treated first. To start with, let us show that using such an ordering is clearly superior to randomly sorting the orders. We executed all benchmarks with a time limit of 10 seconds. Results are reported in Figure 2, in the two top-most plots. The left histogram shows that, in many benchmarks, sorting the orders decreasingly w.r.t. the aforementioned product

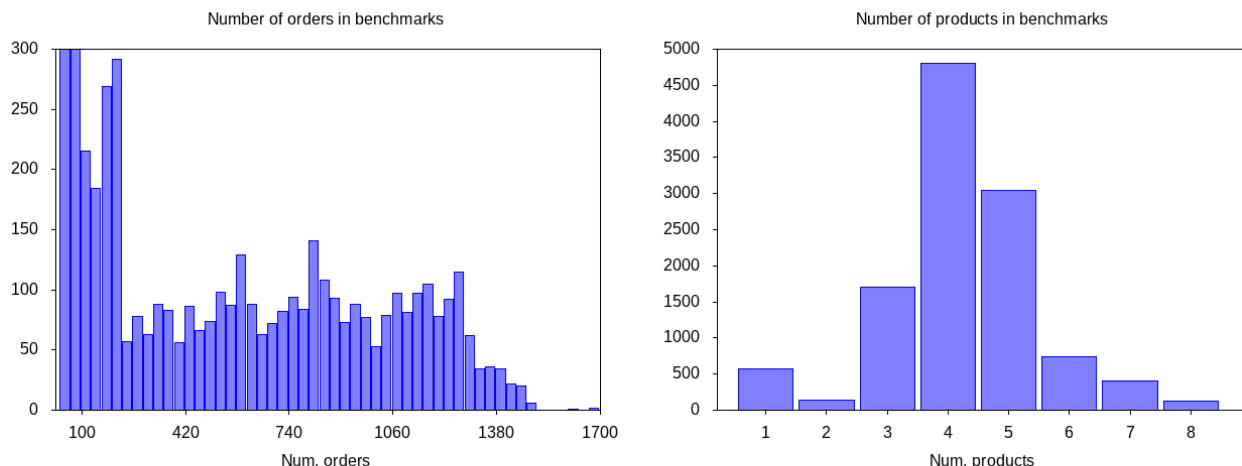

FIGURE 1. On the left, histogram representing the number of benchmarks with a certain number of orders. On the right, the histogram represents the number of benchmarks with a certain number of products.

allows us to obtain better-cost solutions within the time limit of 10 seconds. For example, in 22 benchmarks an improvement between 1 and 2% was obtained. The opposite case never takes place, this is why we see no red column in the histogram. Note that benchmarks for which the two methods give same-quality solutions do not appear in the histogram. They all appear in the right scatter plot, in which we show the time taken by the two methods to obtain that solution. A point (x, y) in the plot indicates that the decreasing product approach took x seconds to compute the solution, whereas it took y seconds when orders are sorted randomly. Again we can see that sorting the orders decreasingly w.r.t. the product is superior to doing it randomly.

The second criterion we tried was to sort the orders increasingly w.r.t. the same product. Results comparing this with a decreasing ordering can be seen in the following two plots in Figure 2. Conclusions are again obvious: sorting the orders decreasingly is better both in solution quality and in time.

Finally, we tried to sort the orders w.r.t. the ratio $number_units/multiplicity$, and do it increasingly and decreasingly. The last four plots in Figure 2 show the results. Clearly, sorting them decreasingly w.r.t. this ratio (i.e. orders with large number of units and small multiplicity first) is not a good idea. This is not a big surprise because orders with large multiplicities exist and it seems a good idea to treat them first. What works very well in practice is to sort the orders increasingly w.r.t. this ratio. This yields very similar results to sorting them decreasingly w.r.t. the product. Note, for example, that only in 12 of the 11521 different benchmarks one method could give a better solution than the other within the 10 seconds time limit.

2) BEST IMPROVEMENT VS. FIRST IMPROVEMENT

In this section we want to consider the possibility of turning `bestNeighbor`, which was presented as a best-improvement approach, into a first-improvement approach, where one stops exploring the neighborhood as soon as she finds a neighbor that improves the starting point. Again, there are two dimensions that we want to explore: the quality of the solution and the time needed to produce such a solution. Note that the following experiments are done using the decreasing-product ordering and that the experiments in the previous section were done using a best-improvement approach.

Again, we executed all benchmarks with a time limit of 10 seconds per instance, collecting the time needed to output the best solution. The left histogram in Figure 3 represents the number of benchmarks for which the quality of the solution computed by each method improves upon the other one for a certain percentage. For example, the columns labeled with 2 indicate that in 148 benchmarks best-improvement obtains a solution which is between 2 and 4% better than the one obtained by first-improvement. Similarly, first-improvement is better with this percentage range in 15 benchmarks. Overall, the histogram clearly shows that best-improvement is superior in obtaining better solutions within the 10 seconds time limit.

In order to evaluate how fast the two methods are in obtaining solutions, we collected all benchmarks for which at least one of the two methods took more than one second (i.e. non-trivial problems), and for which they both gave the same solution. The right histogram in Figure 3 represents the number of benchmarks for which each method improves the other one, in time, for a certain percentage. For example, the columns labeled with 70 indicates that in 34 benchmarks, best-improvement was faster by a percentage between 70 and 75%, whereas first-improvement was faster by the same percentage in 19 benchmarks. Again, best-improvement is the winner in this comparison. Also, we want to remark that, very surprising for us, in a very large number of benchmarks (137) the improvement obtained by best-improvement is more than 95%. We want to point out that we did not use a scatter plot in this occasion because the plot did not give an accurate enough picture of the situation. In this case, the histogram was much clearer.

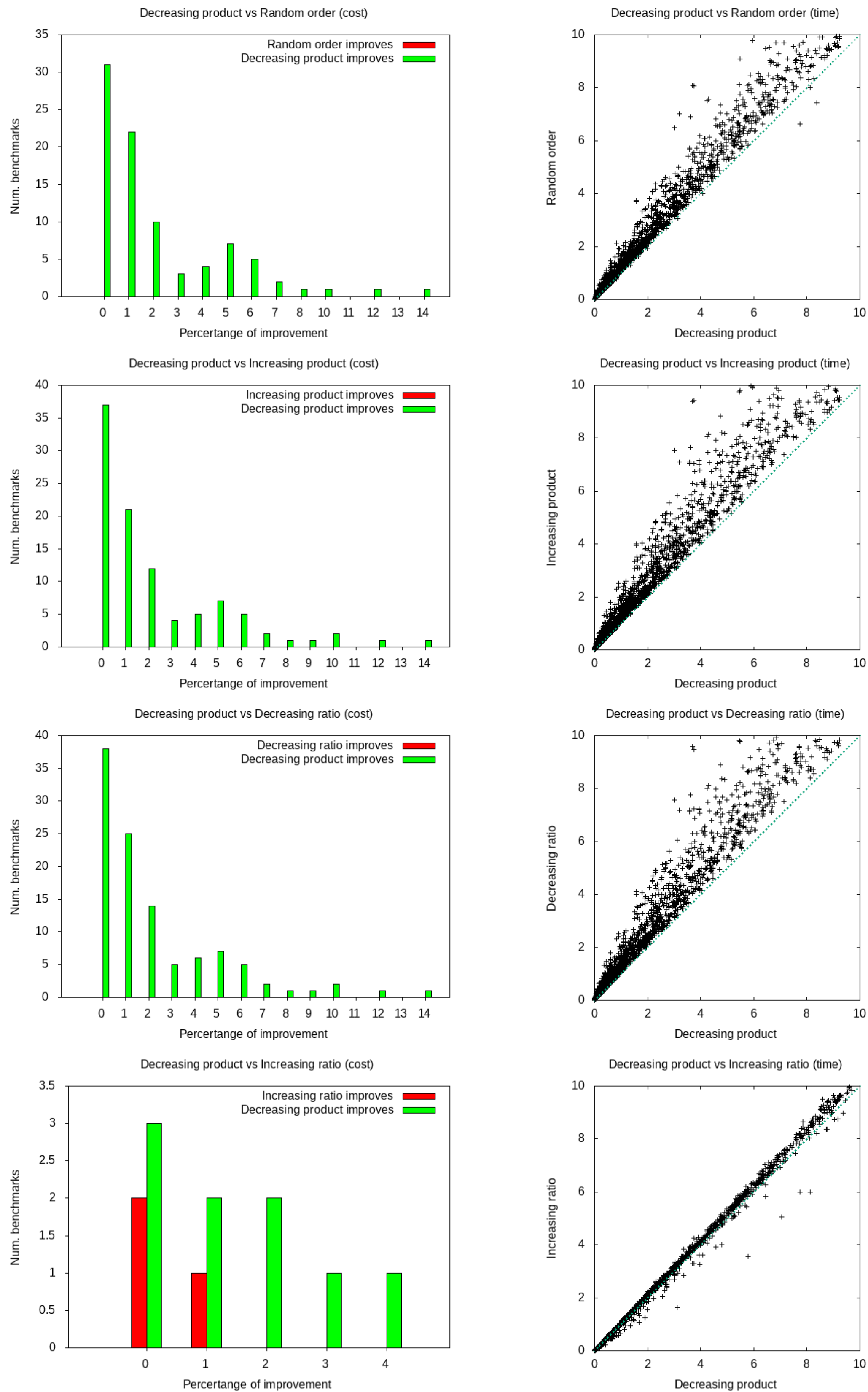

FIGURE 2. Plots comparing the behavior of sorting the orders using different criteria. On the left, histograms show the number of benchmarks for which one ordering produces solutions that are better, in some percentage, than the other ordering. On the right, scatter plots compare the time between two sorting criteria for benchmarks in which both gave solutions with the same cost. We plot the time it takes them to compute such a solution.

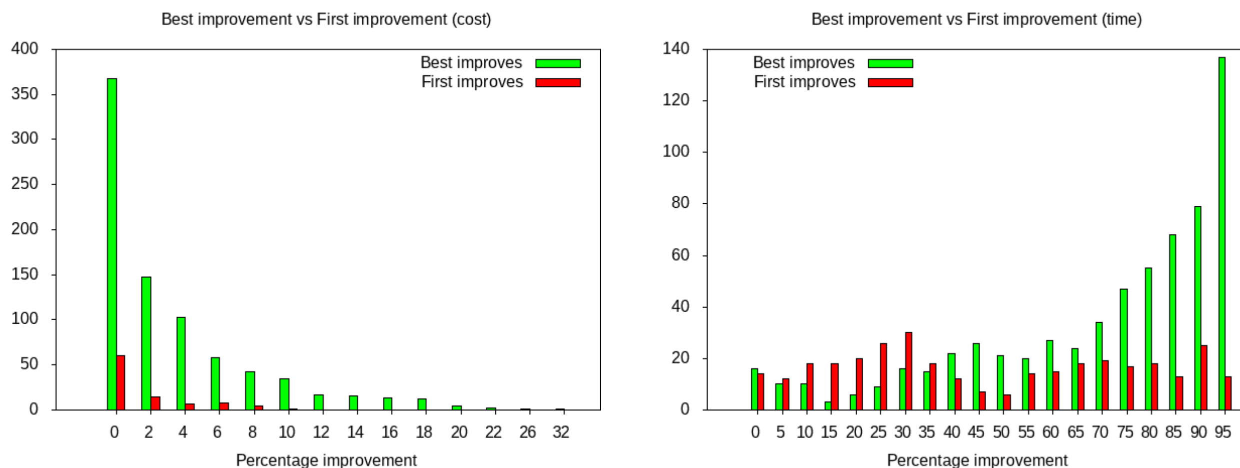

FIGURE 3. On the left, histogram representing the number of benchmarks for which best-improvement or first-improvement produce solutions that are better, in some percentage, than the other alternative. On the right, a similar histogram focuses on problems for which the two alternatives produce the same solution, but one of them is quicker, in some percentage, than the other.

3) SOLUTION QUALITY

One of the main drawbacks of metaheuristics is that they do not give any guarantee about how far the solutions computed are from the optimal solution. In order to evaluate how good the solutions computed by the metaheuristic within the 10 seconds time limit are, we decided to use a complete method, with the hope that it could compute optimal solutions for at least a subset of the benchmarks.

As a complete method we wrote the mathematical formulation of Section III-B for all benchmarks and ran our Barcelogic SMT solver [29] on them. On non-linear integer problems where integer variables appearing in non-linearities are bounded, this SMT solver turns out to be a complete method [30]. To achieve completeness, the formula is linearized by considering all possible values that variables appearing in non-linearities can take. However, instead of doing that right from the beginning, a variety of techniques are applied in order to do it incrementally. Other SMT solvers like MathSAT [31], CVC [32] or Yices [33] can also deal with this type of formulas but had worse performance on this particular family of benchmarks.

Since on non-linear problems the Barcelogic SMT solver can only check for feasibility, we collected the cost k of the best solution that our metaheuristic approach could compute and generate two formulas: one where solutions with cost at most k are searched, and another one where only solutions with cost at most $k-1$ are admitted. If the first formula turns out to be satisfiable, and the second one unsatisfiable we can state that our metaheuristic computed the optimal solution.

We executed all benchmarks with a time limit of 30 seconds in each of the two formulas, and we could certify that on 5627 our metaheuristic computed the optimal solution. On the remaining benchmarks, the SMT solver always exhausted the time limit on the two formulas without giving an answer. We repeated the experiment on 50 randomly chosen benchmarks among these difficult ones, with a time limit of 30 minutes, and the SMT solver could never determine that the second formula was satisfiable. This means that, even with a 30 minute time limit, the SMT solver could never find a better solution than the one given by our heuristic approach. In 2 of the 50 benchmarks it could certify that the solution given by our method was optimal.

All in all, we can conclude that the quality of the solutions given by our approach is very good, in particular if we compare it with a complete approach using an SMT solver for non-linear integer arithmetic.

VI. CONCLUSION AND FUTURE WORK

We introduced the problem of designing optimal cross-docking boxes. After giving an informal presentation, we gave a precise mathematical definition of the problem via a non-linear program. We presented a heuristic approach, based on Variable Neighborhood Search, for solving it. On a large set of real-world benchmarks we showed that our solving method provides high-quality solutions in few seconds.

As future work, we plan to deal with further variants of this problem and to study to which extent complete methods can be of any use, either by finding solutions or computing lower bounds.

REFERENCES

[1] A. Gani, "The logistics performance effect in international trade," *Asian J. Shipping Logistics*, vol. 33, no. 4, pp. 279–288, Dec. 2017.
[2] M. Hirschinger, A. Spickermann, E. Hartmann, H. von der Gracht, and I.-L. Darkow, "The future of logistics in emerging markets-fuzzy clustering scenarios grounded in institutional and factor-market rivalry theory," *J. Supply Chain Manage.*, vol. 51, no. 4, pp. 73–93, Oct. 2015.
[3] J. J. Vogt, "The successful cross-dock based supply chain," *J. Bus. Logistics*, vol. 31, no. 1, pp. 99–119, Mar. 2010.
[4] D.-Z. Du and P. M. Pardalos, Eds., *Handbook of Combinatorial Optimization*. New York, NY, USA: Springer, 1999.
[5] A. Schrijver, *Theory of Linear and Integer Programming*. Hoboken, NJ, USA: Wiley, 1987.
[6] A. Biere, M. J. H. Heule, H. van Maaren, and T. Walsh, Eds., *Handbook of Satisfiability* (Frontiers in Artificial Intelligence and Applications), vol. 185. Amsterdam, The Netherlands: IOS Press, Feb. 2009.

[7] W. C. Barrett, R. Sebastiani, A. S. Seshia, and C. Tinelli, "Satisfiability modulo theories," in *Handbook of Satisfiability*. Amsterdam, The Netherlands: IOS Press, 2009, pp. 825–885.
[8] F. W. Glover and G. A. Kochenberger, Eds., *Handbook of Metaheuristics* (International Series in Operations Research & Management Science), vol. 57. Norwell, MA, USA: Kluwer, 2003.
[9] P. Festa and M. G. C. Resende, "GRASP," in *Handbook of Heuristics*, R. Martí, P. M. Pardalos, and M. G. C. Resende, Eds. New York, NY, USA: Springer, 2018, pp. 465–488.
[10] M. Laguna, "Tabu search," in *Handbook of Heuristics*, R. Martí, P. M. Pardalos, and M. G. C. Resende, Eds. New York, NY, USA: Springer, 2018, pp. 741–758.
[11] S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi, "Optimization by simulated annealing," *Science*, vol. 220, no. 4598, pp. 80–671, Jul. 1983.
[12] P. Hansen and N. Mladenovic, "Variable neighborhood search," in *Handbook of Heuristics*, R. Martí, P. M. Pardalos, and M. G. C. Resende, Eds. New York, NY, USA: Springer, 2018, pp. 759–787.
[13] M. López-Ibáñez, T. Stützle, and M. Dorigo, "Ant colony optimization: A component-wise overview," in *Handbook of Heuristics*, R. Martí, P. M. Pardalos, and M. G. C. Resende, Eds. New York, NY, USA: Springer, 2018, pp. 371–407.
[14] D. Agustina, C. K. M. Lee, and R. Piplani, "A review: Mathematical modles for cross docking planning," *Int. J. Eng. Bus. Manage.*, vol. 2, p. 13, Sep. 2010.
[15] H. D. Ratliff, J. V. Vate, and M. Zhang, "Network design for load-driven cross-docking systems," Georgia Inst. Technol., Atlanta, GA, USA, Tech. Rep., 1998.
[16] V. Jayaraman and A. Ross, "A simulated annealing methodology to distribution network design and management," *Eur. J. Oper. Res.*, vol. 144, no. 3, pp. 629–645, Feb. 2003.
[17] C. S. Sung and S. H. Song, "Integrated service network design for a cross-docking supply chain network," *J. Oper. Res. Soc.*, vol. 54, no. 12, pp. 1283–1295, Dec. 2003.
[18] M. Bachlaus, M. K. Pandey, C. Mahajan, R. Shankar, and M. K. Tiwari, "Designing an integrated multi-echelon agile supply chain network: A hybrid taguchi-particle swarm optimization approach," *J. Intell. Manuf.*, vol. 19, no. 6, pp. 747–761, Dec. 2008.
[19] K. Hauser and C. Chung, "Optimization of a crossdocking layout using genetic algorithms," in *Proc. Annu. Meeting Decis. Sci. Inst.*, Jan. 2003, pp. 1435–1440.
[20] I. F. A. Vis and K. J. Roodbergen, "Positioning of goods in a cross-docking environment," *Comput. Ind. Eng.*, vol. 54, no. 3, pp. 677–689, Apr. 2008.
[21] F. Chen and C.-Y. Lee, "Minimizing the makespan in a two-machine cross-docking flow shop problem," *Eur. J. Oper. Res.*, vol. 193, no. 1, pp. 59–72, Feb. 2009.
[22] A. Lim, Z. Miao, B. Rodrigues, and Z. Xu, "Transshipment through crossdocks with inventory and time windows," *Nav. Res. Logistics*, vol. 52, no. 8, pp. 724–733, Dec. 2005.
[23] L. Y. Tsui and C.-H. Chang, "A microcomputer based decision support tool for assigning dock doors in freight yards," *Comput. Ind. Eng.*, vol. 19, nos. 1–4, pp. 309–312, 1990.
[24] Y. Oh, H. Hwang, C. N. Cha, and S. Lee, "A dock-door assignment problem for the Korean mail distribution center," *Comput. Ind. Eng.*, vol. 51, no. 2, pp. 288–296, Oct. 2006.
[25] Z. Li, C. Sim, M. Y. H. Low, and Y. Lim, "Optimal product allocation for crossdocking and warehousing operations in FMCG supply chain," in *Proc. IEEE Int. Conf. Service Oper. Logistics, Inform.*, vol. 2, Nov. 2008, pp. 2963–2968.
[26] M. Wen, J. Larsen, J. Clausen, J.-F. Cordeau, and G. Laporte, "Vehicle routing with cross-docking," *J. Oper. Res. Soc.*, vol. 60, pp. 1708–1718, Dec. 2009.
[27] V. Y. Matiyasevich, *Hilbert's Tenth Problem*. Cambridge, MA, USA: MIT Press, 1993.
[28] H. Kellerer, U. Pferschy, and D. Pisinger, *Knapsack Problems*. New York, NY, USA: Springer, Jan. 2004.
[29] M. Bofill, R. Nieuwenhuis, A. Oliveras, E. Rodríguez-Carbonell, and A. Rubio, "The barcelogic SMT solver," in *Computer Aided Verification* (Lecture Notes in Computer Science), vol. 5123, 2008, pp. 294–298.
[30] C. Borralleras, D. Larraz, E. Rodríguez-Carbonell, A. Oliveras, and A. Rubio, "Incomplete SMT techniques for solving non-linear formulas over the integers," *ACM Trans. Comput. Log.*, vol. 20, no. 4, pp. 25:1–25:36, 2019.
[31] A. Cimatti, A. Griggio, B. Schaafsma, and R. Sebastiani, "The MathSAT5 SMT solver," in *Proc. TACAS*, in Lecture Notes in Computer Science, vol. 7795, N. Piterman and S. Smolka, Eds. Berlin, Germany: Springer, 2013.
[32] C. W. Barrett, C. L. Conway, M. Deters, L. Hadarean, D. Jovanovic, T. King, A. Reynolds, and C. Tinelli, "CVC4," in *Computer Aided Verification* (Lecture Notes in Computer Science), Snowbird, UT, USA, vol. 6806, G. Gopalakrishnan and S. Qadeer, Eds. New York, NY, USA: Springer, Jul. 2011, pp. 171–177.
[33] B. Dutertre, "Yices 2.2," in *Computer-Aided Verification* (Lecture Notes in Computer Science), vol. 8559, A. Biere and R. Bloem, Eds. New York, NY, USA: Springer, Jul. 2014, pp. 737–744.

ROBERT NIEUWENHUIS received the B.S. and Ph.D. degrees in computer science from the Technical University of Catalonia (UPC), Barcelona, Spain, in 1987 and 1990, respectively. Since 2003, he has been a Professor of computer science with UPC. He is well known for his research at UPC and abroad, such as the Max-Planck Institute, on automated reasoning, constraints, SAT, SMT, or ILP, with highly-cited publications and recognition as an invited speaker, a program committee chair, and an editorial board membership in main conferences and journals. He is also known from the creation of the Barcelogic software tools for SAT and SMT, as well the company of the same name specialized in hard combinatorial optimization problems for scheduling and timetabling.

ALBERT OLIVERAS received the B.S. degree in mathematics and the Ph.D. degree in computer science from the Technical University of Catalonia, Barcelona, Spain, in 2002 and 2006, respectively.

From 2006 to 2012, he was an Assistant Professor with the Department of Computer Science, Technical University of Catalonia. Since 2012, he has been an Associate Professor with the Technical University of Catalonia. He is the author of several highly-cited research articles, and the developer of several tools for SAT, SMT, and ILP. His research interests include logics in computer, with special interest in SAT, SMT, and variants of those problems.

ENRIC RODRÍGUEZ-CARBONELL received the B.S. degree in mathematics and the Ph.D. degree in computer science from the Technical University of Catalonia (UPC), Barcelona, Spain, in 2002 and 2006, respectively. Since 2012, he has been an Associate Professor with the Department of Computer Science, Technical University of Catalonia. His main research interests include applications of logics to computer science, in particular to program verification and analysis and to combinatorial problem solving.