Mixed-Integer Linear Programming Heuristics for the PrePack Optimization Problem

Matteo Fischetti, Michele Monaci, Domenico Salvagnin

DEI, University of Padova

Abstract

In this paper we consider a packing problem arising in inventory allocation applications, where the operational cost for packing the bins is comparable, or even higher, than the cost of the bins (and of the items) themselves. This is the case, for example, of warehouses that have to manage a large number of different customers (e.g., stores), each requiring a given set of items. For this problem, we present Mixed-Integer Linear Programming heuristics based on problem substructures that lead to easy-to-solve and meaningful subproblems, and exploit them within an overall meta-heuristic framework. The new heuristics are evaluated on a standard set of instances, and benchmarked against known heuristics from the literature. Computational experiments show that very good (often proven optimal) solutions can consistently be computed in short computing times.

Keywords: Mixed-Integer Linear Programming, Matheuristics, Packing problems, Prepack Optimization.

1. Introduction

Packing problems play an important role in practical industrial applications, and have been studied in the literature since the early 60s. In these problems, a given set of *items* has to be packed into one or more containers (*bins*) so as

Email address: {matteo.fischetti, michele.monaci, domenico.salvagnin}@unipd.it (Matteo Fischetti, Michele Monaci, Domenico Salvagnin)

to satisfy a number of constraints and to optimize some objective function.

Most of the contributions from the literature are devoted to the case where all the items have to be packed into a minimum number of bins, so as to minimize e.g. transportation costs; within these settings, only loading costs are taken into account. The resulting problem is known as the *Bin Packing Problem*, which has been widely studied in the literature, both in its one-dimensional version ([1]) and in its higher-dimensional variants ([2]).

A number of different packing problems arise in logistic applications: in [3] an optimization problem is considered in which different customers have to be served by a single depot so as to minimize the sum of the packing and of the routing cost, whereas [4] and [5] addressed the problem in which different types of bins are available, each having a given capacity and cost.

In this paper we consider a different packing problem arising in inventory allocation applications, where the operational cost for packing the bins is comparable, or even higher, than the cost of the bins (and of the items) themselves. This is the case, for example, of warehouses that have to manage a large number of different customers (e.g., stores), each requiring a given set of items. Assuming that automatic systems are available for packing, the required workforce is related to the number of different ways that are used to pack the bins to be sent to the customers. To keep this cost under control, a hard constraint can be imposed on the total number of different box configurations that are used.

Pre-packing items into box configurations has obvious benefits in terms of easier and cheaper handling, as it reduces the amount of material handled by both the warehouse and the customers. However, the approach can considerably reduce the flexibility of the supply chain, leading to situations in which the set of items that are actual shipped to each customer may slightly differ from the required one—at the expense of some cost in the objective function. Usually, overstocking is more costly than understocking, as the latter can decrease customer satisfaction. Therefore, to avoid large overstocking, an upper bound on its value is usually imposed for each store.

The resulting problem, known as *PrePack Optimization Problem* (POP),

was recently addressed in [6], where a real-world application in the fashion industry is presented, and heuristic approaches are derived using both Constraint Programming (CP) and Mixed Integer Linear Programming (MILP) techniques. POP is also the subject of recent patent applications, see [7, 8, 9, 10], where a number of heuristic approaches are proposed. Almost all these algorithms iteratively fix some part of the solution step by step. The algorithms differ on the way this is actually done: some of them solve some relaxation of a mathematical formulation of POP, whereas other algorithms fix part of the solution and try to improve it by changing the current pre-pack configuration of some items by a small amount.

Our paper is organized as follows: in Section 2 we formalize POP and recap the mathematical model given in [6]. In Section 3 we present some heuristic algorithms that are based on the mathematical formulation, and test their computational effectiveness in Section 4. Finally, Section 5 draws some conclusions and outlines future research directions.

2. Mathematical model

In this section we briefly formalize POP and review the mathematical model introduced in [6], as the heuristic algorithms to be presented in Section 3 are based on such a formulation.

As already mentioned, in POP we are given a set I of types of products and a set S of stores. Each store $s \in S$ requires an integer number r_{is} of products of type $i \in I$. Bins with different capacities are available for packing items: we denote by $K \subset Z_+$ the set of available bin capacities.

Bins must be completely filled and are available in an unlimited number for each type. A *box configuration* describes the packing of a bin, in terms of number of products of each type that are packed into it. We denote by NB the maximum number of box configurations that can be used for packing all products, and by $B = \{1, \ldots, NB\}$ the associated set.

Products' packing into boxes is described by integer variables y_{bi}: for each

product type $i \in I$ and box configuration $b \in B$, the associated variable indicates the number of products of type i that are packed into the b-th box configuration. In addition, integer variables x_{bs} are used to denote the number of bins loaded according to box configuration b that have to be shipped to store $s \in S$.

Understocking and overstocking of product i at store s are expressed by decisional variables u_{is} and o_{is}, respectively. Positive costs α and β penalize each unit of under- and over-stocking, respectively, whereas an upper bound δ_{is} on the maximum overstocking of each product at each store is also imposed.

Finally, for each box configuration $b \in B$ and capacity value $k \in K$, a binary variable t_{bk} is introduced that takes value 1 iff box configuration b corresponds to a bin of capacity k.

Additional integer variables used in the model are $q_{bis} = x_{bs}\, y_{bi}$ (number of items of type i sent to store s through boxes loaded with configuration b), hence $\sum_{b \in B} q_{bis}$ gives the total number of products of type i that are shipped to store s.

A Mixed-Integer Nonlinear Programming (MINLP) model then reads:

$$\min \sum_{s \in S} \sum_{i \in I} (\alpha u_{is} + \beta o_{is}) \tag{1}$$

$$q_{bis} = x_{bs} y_{bi} \qquad (b \in B; i \in I; s \in S) \tag{2}$$

$$\sum_{b \in B} q_{bis} - o_{is} + u_{is} = r_{is} \qquad (i \in I; s \in S) \tag{3}$$

$$\sum_{i \in I} y_{bi} = \sum_{k \in K} k\, t_{bk} \qquad (b \in B) \tag{4}$$

$$\sum_{k \in K} t_{bk} = 1 \qquad (b \in B) \tag{5}$$

$$o_{is} \leq \delta_{is} \qquad (i \in I; s \in S) \tag{6}$$

$$t_{bk} \in \{0, 1\} \qquad (b \in B; k \in K) \tag{7}$$

$$x_{bs} \geq 0 \text{ integer} \qquad (b \in B; s \in S) \tag{8}$$

$$y_{bi} \geq 0 \text{ integer} \qquad (b \in B; i \in I) \tag{9}$$

The model is of course nonlinear, as the bilinear constraints (2) involve the product of decision variables. To derive a MILP model, each x_{bs} variable is decomposed in its binary expansion using binary variables v_{bsl} $(l = 0, \dots, L)$,
85 where L is easily computed from an upper bound on x_{bs}. When these variables are multiplied by y_{bi}, the corresponding product $w_{bisl} = v_{bsl}\, y_{bi}$ are linearized with the addition of suitable constraints.

Our MILP is therefore obtained from (1)-(9) by adding

$$x_{bs} = \sum_{l=0}^{L} 2^l v_{bsl} \qquad (b \in B; s \in S) \tag{10}$$

$$v_{bsl} \in \{0, 1\} \qquad (b \in B; s \in S; l = 0, \dots, L) \tag{11}$$

and by replacing each nonlinear equation (2) with the following set of new
90 variables and constraints:

$$q_{bis} = \sum_{l=0}^{L} 2^l w_{bisl} \qquad (b \in B; i \in I; s \in S) \tag{12}$$

$$w_{bisl} \le \overline{Y} v_{bsl} \qquad (b \in B; i \in I; s \in S; l = 0, \dots, L) \tag{13}$$

$$w_{bisl} \le y_{bi} \qquad (b \in B; i \in I; s \in S; l = 0, \dots, L) \tag{14}$$

$$w_{bisl} \ge y_{bi} - \overline{Y}(1 - v_{bsl}) \qquad (b \in B; i \in I; s \in S; l = 0, \dots, L) \tag{15}$$

$$w_{bisl} \ge 0 \qquad (b \in B; i \in I; s \in S; l = 0, \dots, L) \tag{16}$$

$$\tag{17}$$

where $\overline{Y}$ denotes an upper bound on the y variables.

Note that, because of (10) and of (12), variables x_{bs} and q_{bis} could be removed from the model—in our implementation, we declare them as continuous and free variable and let the preprocessing remove them automatically.

95 In case all capacities are even, the following constraint—though redundant—plays a very important role in improving the LP bound of our MILP model:

$$\sum_{i \in I} (u_{is} + o_{is}) \ge 1 \qquad (s \in S : \sum_{i \in I} r_{is} \text{ is odd}) \tag{18}$$

Further details on the model can be found in [6].

3. Matheuristics

As also reported in [6], the MILP model of the previous section is by far too difficult to be attached by standard solvers. As a matter of fact, for real-world cases even the LP relaxation at each node turns out to be very time consuming—even by using the barrier method. So we designed ad-hoc heuristic approaches to exploit the special structure of our MILP, following a so-called *matheuristic* paradigm [11].

Each heuristic is based on the idea of iteratively solving a restricted problem obtained by fixing a subset of the variables, so as to obtain a subproblem which is (reasonably) easy to solve by a commercial MILP solver, but still able to produce improved solutions.

We have implemented two kinds of heuristics: constructive and refinement heuristics. Constructive heuristics are used to find a solution H starting from scratch. In a refinement heuristic, instead, we are given an incumbent heuristic solution $H = (x^H, y^H)$ that we would like to improve. We first fix some x and/or y variables to their value in H, thus defining a solution neighborhood $\mathcal{N}(H)$ of H. We then search $\mathcal{N}(H)$ by using a general-purpose MILP solver on the model resulting from fixing. If an improved solution is found within the given time limit, we update H and repeat; otherwise, a new neighborhood is defined in the attempt to escape the local optimum.

More elaborated schemes based on Tabu Search [12] or Variable Neighborhood Search [13] are also possible but not investigated in the present paper.

3.1. Fixing all x or y variables

A first obvious observation is that our basic MINLP model (1)-(9) reduces to a MILP if all the x (or all the y) variables are fixed, as constraints (2) trivially become linear. According to our experience, the resulting MILP (though nontrivial) is typically solved very quickly by a state-of-the-art solver, meaning that one can effectively solve a sequence of restricted MILP where x and y are fixed/unfixed, in turn, until no further improvement can be obtained.

Let $\mathcal{N}_y(H)$ and $\mathcal{N}_x(H)$ denote the solution neighborhoods of H obtained by leaving y or x free, i.e., when imposing $x = x^H$ or $y = y^H$, respectively.

A basic tool that we use in our heuristics is function $\texttt{REOPT}(S', y^H)$ that considers a store subset $S' \subseteq S$ and a starting y^H, and returns the best solution H obtained by iteratively optimizing over the neighborhoods $\mathcal{N}_x(H)$, $\mathcal{N}_y(H)$, $\mathcal{N}_x(H), \ldots$ after having removed all stores not in S', where H is updated after each optimization.

3.2. Fixing y variables for all but one configuration

Another interesting neighborhood, say $\mathcal{N}_x(H, y_\beta)$, is obtained by leaving all x variables free, and by fixing $y_{bi} = y^H_{bi}$ for all $i \in I$ and $b \in B \setminus \{\beta\}$ for a given $\beta \in B$. In other words, we allow for changing just one (out of NB) configurations in the current solution, and leave the solver the possibility to change the x variables as well.

In our implementation, we first define a random permutation $\{\beta_1, \ldots, \beta_{NB}\}$ of B. We then optimize, in a circular sequence, neighborhoods $\mathcal{N}_x(H, y_{\beta_t})$ for $t = 1, \ldots, NB, 1, \ldots$ Each time an improved solution is found, we update H and further refine it through function $\texttt{REOPT}(S, y^H)$. The procedure is stopped when there is no hope of finding an improved solution, i.e., after NB consecutive optimizations that do not improve the current H.

A substantial speedup can be obtained by heuristically imposing a tight upper bound on the x variables, so as to reduce the number $L + 1$ of binary variables v_{bsl} in the binary expansion (10). An aggressive policy (e.g., imposing $x_{bs} \leq 1$) is however rather risky as the optimal solution could be cut off, hence the artificial bounds must be relaxed if an improved solution cannot be found.

3.3. Working with a subset of stores

Our basic constructive heuristic is based on the observation that removing stores can produce a substantially easier model. A key property here is that a solution $H' = (x', y')$ with a subset of stores can easily be converted into a solution $H = (x, y)$ of the whole problem by just invoking function $\texttt{REOPT}(S, y')$.

In our implementation, we first define a store permutation $s_1, \ldots, s_{|S|}$ according to a certain criterion (to be discussed later). We then address, in sequence, the subproblem with store set $S_t = \{s_1, \ldots, s_t\}$ for $t = 0, \ldots, |S|$.

For $t = 0$, store set S_0 is empty and our MILP model just produces a y 160 solution with random (possibly repeated) configurations.

For each subsequent t, we start with the best solution $H_{t-1} = (x^{t-1}, y^{t-1})$ of the previous iteration, and convert it into a solution $H_t = (x^t, y^t)$ of the current subproblem through the refining function `REOPT`(S_t, y^{t-1}). Then we apply the refinement heuristics described in the previous subsection to H_t, reoptimizing 165 one configuration at a time in a circular vein. (To reduce computing time, this latter step can be skipped with a certain frequency—except of course in the very last step when $S_t = S$.)

Each time a solution $H_t = (x^t, y^t)$ is found, we quickly compute a solution $H = (x, y)$ of the overall problem through function `REOPT`(S, y^t) and update the 170 overall incumbent where all stores are active.

As to store sequence $s_1, \ldots, s_{|S|}$, we have implemented three different procedures. For each store pair a, b, let the dissimilarity index $dist(a, b)$ be defined as the Hamming distance between the two demand vectors $(r_{ia} : i \in I)$ and $(r_{ib} : i \in I)$.

175
- `random`: the sequence is just a random permutation of the integers $1, \ldots, |S|$;
- `most_dissimilar`: we first compute the two most dissimilar stores (a, b), i.e., such that $a < b$ and $dist(a, b)$ is a maximum, and initialize $s_1 = a$. Then, for $t = 2, \ldots, |S|$ we define $S' = \{s_1, \ldots, s_{t-1}\}$ and let

$$s_t = \text{argmax}_{a \in S \setminus S'} \{ \min\{dist(a, b) : b \in S' \} \}$$

- `most_similar`: this is just the same procedure as in the previous item, with max and min operators reverted.

The rational of the `most_dissimilar` policy is to attach first a "core problem" defined by the pairwise most dissimilar stores (those at the beginning of

the sequence). The assumption here is that our method performs better in its first iterations (small values of t) as the size of the subproblem is smaller, and we have plenty of configurations to accommodate the initial requests. The "similar stores" are therefore addressed only at a later time, in the hope that the found configurations will work well for them.

A risk with the above policy is that the core problem becomes soon too difficult for our simple refining heuristic, so the current solution in not updated after the very first iterations. In this respect, policy `most_similar` is more conservative: given for granted that we proceed by subsequently refining a previous solution with one less store, it seems reasonable not to inject too much innovation in a single iteration—as `most_dissimilar` does when it adds a new store with very different demands with respect to the previous ones.

4. Computational experiments

The heuristics described in the previous section have been implemented in C language. IBM ILOG CPLEX 12.6 [14] was used as MILP solver. Reported computing times are in CPU seconds of an Intel Xeon E3-1220 V2 quad-core PC with 16GB of RAM. For each run a timelimit of 900 seconds (15 minutes) was imposed.

Four heuristic methods have been compared: the three construction heuristics `random`, `most_dissimilar` and `most_similar` of Subsection 3.3, plus

- `fast_heu`: the fast refinement heuristic of Subsection 3.2 applied starting from a null solution $x = 0$.

All heuristics are used in a multi-start mode, e.g., after completion they are just restarted from scratch until the time limit is exceeded. At each restart, the internal random seed is not reset, hence all methods natively using a random permutation (namely, `fast_heu` and `random`) will follow a different search path at each run as the permutations will be different. As to `most_dissimilar` and `most_similar`, after each restart the sequence $s_1, \ldots, s_{|S|}$ is slightly perturbed by 5 random pair swaps. In addition, after each restart the CPLEX's random

seed is changed so as to inject diversification among each run even within the MILP solver.

Due to their heuristic nature, our methods—through deterministic—exhibit a large dependency on the initial conditions, including the random seeds used both within our code and in CPLEX. We therefore repeated several times each experiment, starting with different (internal/CPLEX) seeds at each run, and report statistical figures (median, average, ...) in the forthcoming tables.

In case all capacities are even (as it is the case in our tesbed), we compute the following trivial lower bound based on constraint (18)

$$LB := \min\{\alpha, \beta\} \times |\{s \in S : \sum_{i \in I} r_{is} \text{ is odd}\}| \tag{19}$$

and abort the execution as soon as we find a solution whose value meets the lower bound.

4.1. Testbed

Our testbed is composed by two benchmarks of instances. The first one is the benchmark considered in [6], and contains a number of subinstances of a real-world instance (named `AllColor58` in [6]) with 58 stores that require 24 (= 6×4) different items: T-shirts available in six different sizes and four different colors (black, blue, red, and green). The available box capacities are 4, 6, 8, and 10. Finally, each item has a fixed overstock limit (0 or 1) for all stores but no understock limits, and the overstock and understock penalties are $\beta = 1$ and $\alpha = 10$, respectively. From this basic instance we also derived a second benchmark by considering only a subset of the stores (see Section 4.4). All the new instances are available, on request, from the authors.

Note that our testing environment is identical to that used in [6] (assuming the PC used are substantially equivalent), so our results can fairly be benchmarked against those therein reported.

4.2. Comparison metric

To better compare the performance of our different heuristics, we also make use of an indicator recently proposed by [15, 16] and aimed at measuring the

trade-off between the computational effort required to produce a solution and the quality of the solution itself. In particular, let $\widetilde{z}_{opt}$ denote the optimal solution value and let $z(t)$ be the value of the best heuristic solution found at a time t. Then, a *primal gap function* p can be computed as

$$p(t) = \begin{cases} 1 & \text{if no incumbent found until time } t \\ \gamma(z(t)) & \text{otherwise} \end{cases} \tag{20}$$

where $\gamma(\cdot) \in [0,1]$ is the *primal gap*, defined as follows

$$\gamma(z) = \begin{cases} 0 & \text{if } |\widetilde{z}_{opt}| = |z| = 0, \\ 1 & \text{if } \widetilde{z}_{opt} \cdot z < 0, \\ \frac{z - \widetilde{z}_{opt}}{\max\{|\widetilde{z}_{opt}|, |z|\}} & \text{otherwise.} \end{cases} \tag{21}$$

Finally, the *primal integral* of a run until time $t_{\max}$ is defined as

$$P(t_{\max}) = \frac{\int_0^{t_{\max}} p(t)\,\mathrm{d}t}{t_{\max}} \tag{22}$$

and is actually used to measure the quality of primal heuristics—the smaller $P(t_{\max})$ the better the expected quality of the incumbent solution if we stopped computation at an arbitrary time before $t_{\max}$.

4.3. Results on standard instances

Our first set of experiments addresses the instances provided in [6], with the aim of benchmarking our heuristics against the methods therein proposed. Results for the easiest cases involving only $NB = 4$ box configurations (namely, instances Black58, Blue58, Red58, and Green58) are not reported as the corresponding MILP model can be solved to proven optimality in less than one seconds by our solver—thus confirming the figures given in [6].

Tables 1 and 2 report the performance of various heuristics in terms of solution value and time, and refer to a single run for each heuristic and for each instance.

Table 1 is taken from [6], where a two-phase hybrid metaheuristic was proposed. In the first phase, the approach uses a memetic algorithm to explore

Table 1: Performance of CPLEX and LNS heuristics from [6]. Single run for each instance. Times in CPU seconds (time limit of 900 sec.s).

		CPLEX		LNS	
instance	NB	value	time (s)	value	time (s)
BlackBlue10	7	66	7	21	16
BlackBlue58	7	525	43	174	74
AllColor10	14	202	49	89	293
AllColor58	14	1828	273	548	900

the solution space and builds a pool of interesting box configurations. In the second phase, a box-to-store assignment problem is solved to choose a subset of configurations from the pool—and to decide how many boxes of each configuration should be sent to each store. The box-to-store assignment problem is modeled as a (very hard in practice) MILP and heuristically solved either by a commercial solver (CPLEX) or by a sophisticated large neighborhood search (LNS) approach.

Table 2: Performance of our heuristics. Single run for each instance. Times in CPU seconds (time limit of 900 sec.s).

		`fast_heu`		`random`		`most dissimilar`		`most similar`	
instance	LB	value	time (s)	value	time (s)	value	time (s)	value	time (s)
BlackBlue10	10	10	1	10	2	10	1	10	1
BlackBlue58	58	58	4	58	3	58	2	58	4
AllColor10	6	6	29	17	82	17	734	17	379
AllColor58	42	141	71	273	722	614	105	53	66

Table 2 reports the performance of our four heuristics, as well as the lower bound value LB computed through (19)—this value turned out to coincide with the optimal value for all instances under consideration in the present subsection.

Comparing Tables 1 and 2 shows that all our four heuristics outperform those in [6]. In particular, `fast_heu` is able to find very good solutions (actually, 270 an optimal one in 3 out of 4 cases) within very short computing times. For the largest instance (AllColor58), however, `most_similar` qualifies as the best heuristic both in terms of quality and speed.

Table 3: Average performance (out of 100 runs) of our heuristics.

instance	heuristic	time (s)	time_best (s)	pint	#opt
BlackBlue10	`fast_heu`	1.08	1.08	0.34	100
	`random`	1.44	1.44	0.27	100
	`most_dissimilar`	1.26	1.26	0.25	100
	`most_similar`	1.25	1.25	0.29	100
BlackBlue58	`fast_heu`	4.61	4.61	9.88	100
	`random`	6.40	6.40	10.11	100
	`most_dissimilar`	2.76	2.76	9.13	100
	`most_similar`	5.62	5.62	15.42	100
AllColor10	`fast_heu`	71.82	71.82	3.81	100
	`random`	600.29	304.06	18.63	36
	`most_dissimilar`	704.33	241.12	19.69	27
	`most_similar`	626.15	302.40	20.54	26
AllColor58	`fast_heu`	900.00	332.43	328.20	0
	`random`	874.87	329.59	562.95	2
	`most_dissimilar`	893.48	323.93	545.47	1
	`most_similar`	859.86	287.50	404.29	1

To get more reliable information about the performance of our heuristics, we ran them 100 times for each instance, with different random seeds, and 275 took detailed statistics on each run. Table 3 reports, for each instance and for each heuristic, the average completion time (time), the average time to find its best solution (time_best), the primal integral after 900 seconds (pint, the lower the better), and the number of provably-optimal solutions found (#opt) out of the 100 runs. Note that, for all instances, a solution matching the simple lower

280 bound (19) was eventually found by at least one of our heuristics. As to the final value returned by the four heuristics, as well as the corresponding computing time, Figures 1 and 2 give box plots on the hardest instances (AllColor10 and AllColor58).

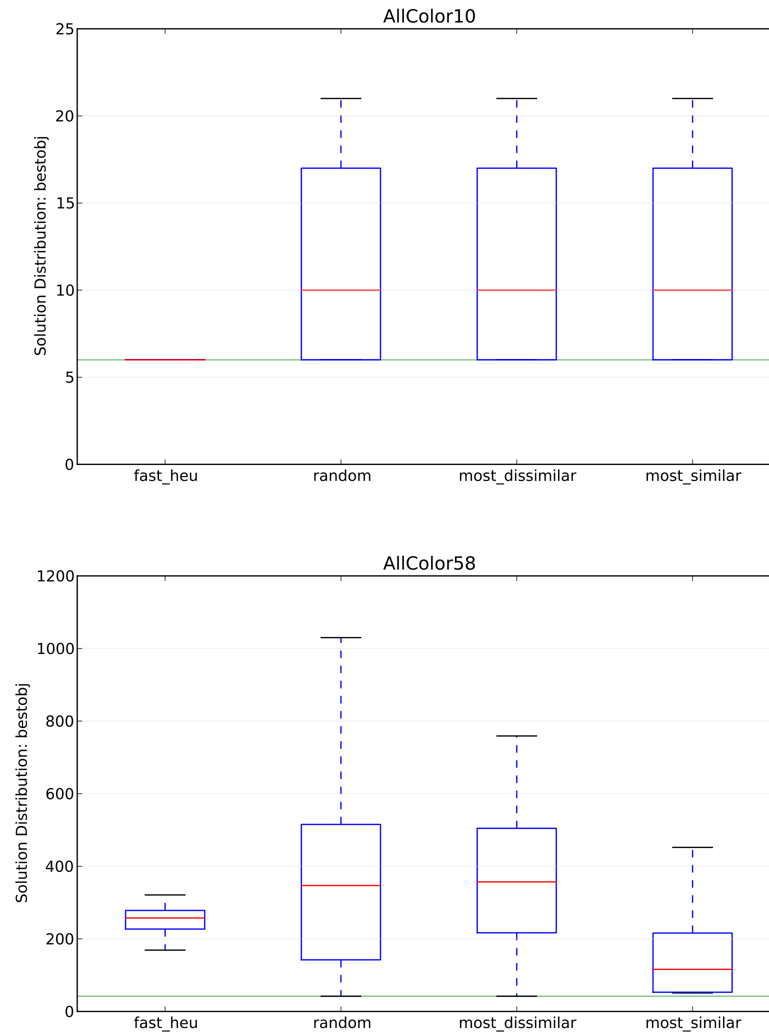

Figure 1: Box plots for objective value of best solution found by our heuristic algorithms on the hardest instances with all colors (100 runs for each method and for each instance).

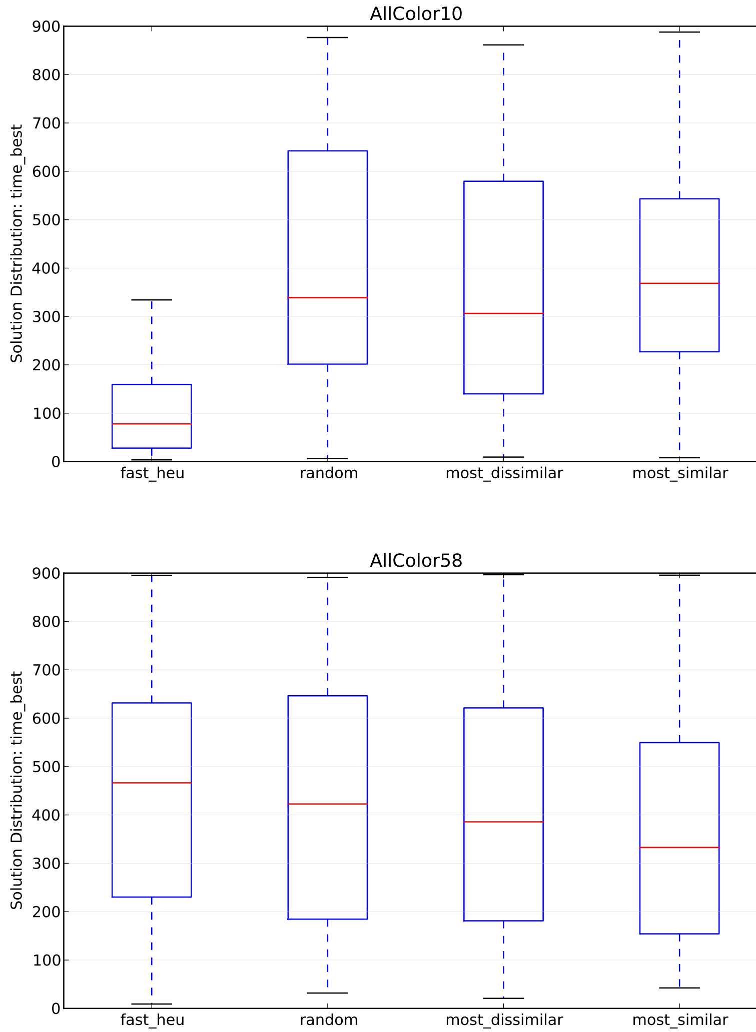

Figure 2: Box plots for time to best solution of our heuristic algorithms on the hardest instances with all colors (100 runs for each method and for each instance).

The 100-run statistics confirm that `fast_heu` is very effective in all cases, though it is outperformed by `most_similar` for the largest instance AllColor58. The results suggest that a hybrid method running `fast_heu` and `most_similar` (possibly in parallel) qualifies a robust heuristic with a very good performance

for all our instances.

4.4. Additional instances

290 To test our algorithms on a larger benchmark, we considered a number of instances randomly derived from problem AllColor58. In particular, given a parameter NS, we randomly selected NS stores and considered a sub-instance induced by the selected stores and the original 24 products (six different sizes and four different colors). We considered different values of NS, namely 10, 20, 295 30, 40 and 50; for each value of NS, 10 different instances were generated, so as to produce a benchmark with 50 new problems.

Table 4 reports the outcome of our experiments on the new instances and provides, for each algorithm, the same information given in Table 3. To make results comparable to those of Table 3, we ran each algorithm 10 times on each 300 instance, using 10 different random seeds, and grouped instances according to the value of NS. Thus, each table line refers to the 100 runs associated to a set of 10 instances (10 executions on each instance).

Computational results confirm the effectiveness of algorithms `fast_heu` and `most_similar`, the latter being the clear winner for the largest cases. As ex-305 pected, instances with a large number of stores are harder to be solved to proven optimality, and correspond to larger values for the associated primal integral.

5. Conclusions

We have presented MILP-based heuristics for the prepack optimization problem. Working with a subset of the variables (after fixing the other) we were able 310 to squeeze useful solutions out of an impossible-to-solve MILP model. This was obtained by identifying problem substructures that lead to easy-to-solve and meaningful MILP subproblems, and to exploit them within an overall metaheuristic framework—an approach known as *matheuristic*.

Our heuristics have been evaluated on a standard set of instances, and bench-315 marked against the heuristics given in [6]. Computational experiments show

Table 4: Average performance (out of 100 runs) of our heuristics on randomly generated instances.

#stores	heuristic	time (s)	time_best (s)	pint	#opt
10	`fast_heu`	13.50	12.98	2.17	99
	`random`	28.93	27.62	2.95	98
	`most_dissimilar`	28.53	27.88	2.82	99
	`most_similar`	25.08	24.75	2.88	99
20	`fast_heu`	475.92	288.52	67.62	44
	`random`	481.45	257.67	55.01	43
	`most_dissimilar`	608.64	314.64	63.90	34
	`most_similar`	360.29	232.75	49.43	50
30	`fast_heu`	607.74	335.04	172.41	30
	`random`	571.61	284.02	185.58	25
	`most_dissimilar`	654.78	251.82	184.60	22
	`most_similar`	497.63	258.20	175.77	33
40	`fast_heu`	900.00	353.69	225.49	0
	`random`	752.04	316.71	331.19	15
	`most_dissimilar`	812.51	297.55	337.96	8
	`most_similar`	690.88	280.07	238.32	26
50	`fast_heu`	900.00	293.99	309.66	0
	`random`	863.61	281.75	532.17	3
	`most_dissimilar`	894.37	227.64	465.91	1
	`most_similar`	866.94	282.08	355.46	4

that very good (often proven optimal) solutions can consistently be computed in a matter of seconds or minutes. This compares very favorably with the most effective heuristics reported in [6]; in particular, we were able to solve to proven optimality all the instances therein provided.

320 Future work can be devoted to designing alternative MILP-based schemes based on different substructures of the problem and/or on more sophisticated meta-heuristic schemes.

Acknowledgements

This research was supported by the University of Padova (Progetto di Ateneo “Exploiting randomness in Mixed Integer Linear Programming”), and by MiUR, Italy (PRIN project “Mixed-Integer Nonlinear Optimization: Approaches and Applications”). We thank Louis-Martin Rousseau for kindly providing the original instances.

References

[1] S. Martello, P. Toth, Knapsack Problems: Algorithms and Computer Implementations, John Wiley & Sons, Chichester, 1990.

[2] A. Lodi, S. Martello, M. Monaci, Two-dimensional packing problems: A survey, European Journal of Operational Research 141 (2002) 241–252.

[3] M. Iori, J.J. Salazar-Gonzales, D. Vigo, An Exact Approach for the Vehicle Routing Problem with Two-Dimensional Loading Constraints, Transportation Science 41 (2007) 253–264.

[4] M. Monaci, Algorithms for packing and scheduling problems, 4OR 1 (2004) 85–87.

[5] I. Correia, L. Gouveia, F. S. da Gama, Solving the variable size bin packing problem with discretized formulations, Computers and Operations Research 35 (2008) 2103–2113.

[6] M. Hoskins, R. Masson, G. Melanon, J. Mendoza, C. Meyer, L.-M. Rousseau, The PrePack Optimization Problem, in: H. Simonis (Ed.), Integration of AI and OR Techniques in Constraint Programming, Vol. 8451 of Lecture Notes in Computer Science, Springer Berlin / Heidelberg, 2014, pp. 136–143.

[7] R. Pratt, Computer-Implemented Systems And Methods For Pack Optimization, SAS Institute A1 (2009) US20090271241.

[8] C. Erie, J. Lee, R. Paske, J. Wilson, Dynamic bulk packing and casing, International Business Machines Corporation A1 (2010) US20100049537.

[9] A. Vakhutinsky, Retail pre-pack optimization system, Oracle International Corporation A1 (2012) US20120284071.

[10] A. Vakhutinsky, S. Subramanian, Y. Popkov, A. Kushkuley, Retail pre-pack optimizer, Oracle International Corporation A1 (2012) US20120284079.

[11] M. Boschetti, V. Maniezzo, M. Roffilli, A. R. Bolufé, Matheuristics: Optimization, Simulation and Control, in: Hybrid Metaheuristics, Springer Berlin Heidelberg, 2009, pp. 171–177.

[12] F. Glover, Tabu Search: A Tutorial, Interfaces 20 (4) (1990) 74–94.

[13] N. Mladenovic, P. Hansen, Variable neighborhood search, Computers and Operations Research 24 (11) (1997) 1097–1100.

[14] I. IBM, CPLEX 12.6 User's Manual (2014).

[15] T. Achterberg, T. Berthold, G. Hendel, Rounding and Propagation Heuristics for Mixed Integer Programming, Operations Research Proceedings 2011 (2012) 71–76.

[16] T. Berthold, Measuring the impact of primal heuristics, Operations Research Letters 41 (6) (2013) 611–614.