Matheuristics

5

Martina Fischetti and Matteo Fischetti

Contents

M. Fischetti (✉)
Technical University of Denmark, Kongens Lyngby, Copenhagen, Denmark

Vattenfall BA Wind, Kongens Lyngby, Copenhagen, Denmark
e-mail: martina.fischetti@vattenfall.com

M. Fischetti
DEI, University of Padova, Padova, Italy
e-mail: matteo.fischetti@unipd.it

R. Martí et al. (eds.), *Handbook of Heuristics*,
https://doi.org/10.1007/978-3-319-07124-4_14

Abstract

As its name suggests, a matheuristic is the hybridization of mathematical programming with metaheuristics. The hallmark of matheuristics is the central role played by the mathematical programming model, around which the overall heuristic is built. As such, matheuristic is not a rigid paradigm but rather a concept framework for the design of mathematically sound heuristics. The aim of this chapter is to introduce the main matheuristic ideas. Three specific applications in the field of wind farm, packing, and vehicle routing optimization, respectively, are addressed and used to illustrate the main features of the method.

Keywords

Heuristics · Large scale neighborhood search · Local branching · Mathematical programming · Matheuristics

Introduction

The design of heuristics for difficult optimization problems is itself a heuristic process that often involves the following main steps.

After a clever analysis of the problem at hand and of the acceptable simplifications in its definition, one tries to set up an effective **mathematical programming** (MP) model and to solve it by a general-purpose piece of software—often a mixed-integer linear programming (MIP) solver. Due to the impressive improvement of general-purpose solvers in recent years, this approach can actually solve the instances of interest to proven optimality (or with an acceptable approximation) within a reasonable computing time, in which case of course no further effort is needed.

If this is not the case, one can insist on the MP approach and try to obtain better and better results by improving the model and/or by enhancing the solver by specialized features (cutting planes, branching, etc.). Or one can forget about MP and resort to ad hoc heuristics not based on the MP model. In this latter case, the MP model is completely disregarded or just used for illustrating the problem characteristics and/or for getting an off-line indication of the typical approximation error on a set of sample instances.

A third approach is however possible that consists in using the MP solver as a basic tool *within* the heuristic framework. This hybridization of MP with *metaheuristics* leads to the *matheuristic* approach, where the heuristic is built around the MP model. Matheuristics became popular in recent years, as witnessed by the publication of dedicated volumes and journal special issues [8, 19, 24] and by the dedicated sessions on MP and metaheuristic conferences.

Designing an effective heuristic is an art that cannot be framed into strict rules. This is particularly true when addressing a matheuristic, which is not a rigid paradigm but a concept framework for the design of mathematically sound heuristics. In this chapter, we will therefore try to illustrate some main matheuristic features with the help of different examples of application.

Section "General-Purpose MIP-Based Heuristics" describes powerful general-purpose MIP heuristics that can be used within the matheuristic framework. Interestingly, these heuristics can themselves be viewed as the first successful applications of the matheuristic idea of hybridizing MP and metaheuristics. Indeed, as noticed in [8], one of the very first illustrations of the power of the matheuristic idea is the general-purpose local branching [7] paradigm, where a black-box MIP solver is used to explore a solution neighborhood defined by invalid constraints added to the MIP model for the sake of easing its solution.

Section "Application 1: Wind Farm Layout Optimization" addresses the design of a matheuristic for wind farm optimization. This application is used to illustrate the importance of the choice of the MIP model: models that are weak in polyhedral terms can be preferred to tighter—but computationally much harder—models when heuristic (as opposed to exact) solutions are required.

Section "Application 2: Prepack Optimization" addresses a packing problem where the model is nonlinear, and the matheuristic is based on various ways to linearize it after a heuristic fixing of some variables.

Finally, section "Application 3: Vehicle Routing" is used to illustrate an advanced feature of matheuristics, namely, the solution of auxiliary MP models that describe a subproblem in the solution process. In particular, we address a vehicle routing problem and derive a matheuristic based on a set-partitioning MIP model asking for the reallocation of a subset of customer sequences subject to capacity and distance constraints.

The present chapter is based on previous published work; in particular, sections "General-Purpose MIP-Based Heuristics", "Application 1: Wind Farm Layout Optimization", "Application 2: Prepack Optimization", and "Application 3: Vehicle Routing" are based on [8, 11, 13, 15], respectively.

General-Purpose MIP-Based Heuristics

Heuristics for general-purpose MIP solvers form the basis of the matheuristic's toolkit. Their relevance for our chapter is twofold. On the one hand, they are invaluable tools for the solution of the subproblems tailored by the matheuristic when applied to a specific problem. On the other hand, they illustrate the benefits for a general-purpose MIP solver deriving from the use of metaheuristics concepts such as local search and evolutionary methods.

Modern MIP solvers exploit a rich arsenal of tools to attack hard problems. It is widely accepted that the solution of hard MIPs can take advantage from the solution of a series of auxiliary linear programs (LPs) intended to enhance the performance of the overall MIP solver. For example, auxiliary LPs may be solved to generate powerful disjunctive cuts or to implement a strong branching policy. On the other hand, it is a common experience that finding good-quality heuristic MIP solutions often requires a computing time that is just comparable to that needed to solve the LP relaxation. So, it makes sense to think of exact/heuristic MIP solvers where auxiliary MIPs (as opposed to LPs) are heuristically solved on the fly, with the aim

of bringing the MIP technology under the chest of the MIP solver itself. This leads to the idea of "translating into a MIP model" (*MIPping* in the jargon of [9]) some crucial decisions to be taken when designing a MIP-based algorithm.

We next describe the new generation of MIP heuristics that emerged in the late 1990s, which are based on the idea of systematically using a "black-box" external MIP solver to explore a solution neighborhood defined by invalid linear constraints. We address a generic MIP of the form

$$(MIP) \qquad \min c^T x \tag{1}$$

$$Ax \geq b, \tag{2}$$

$$x_j \in \{0,1\}, \ \forall j \in \mathcal{B}, \tag{3}$$

$$x_j \text{ integer}, \ \forall j \in \mathcal{G}, \tag{4}$$

$$x_j \text{ continuous}, \ \forall j \in \mathcal{C}, \tag{5}$$

where A is an $m \times n$ input matrix and b and c are input vectors of dimension m and n, respectively. Here, the variable index set $\mathcal{N} := \{1, \ldots, n\}$ is partitioned into $(\mathcal{B}, \mathcal{G}, \mathcal{C})$, where $\mathcal{B}$ is the index set of the 0-1 variables (if any), while sets $\mathcal{G}$ and $\mathcal{C}$ index the general integer and the continuous variables, respectively. Removing the integrality requirement on variables indexed by $\mathcal{I} := \mathcal{B} \cup \mathcal{G}$ leads to the so-called *LP relaxation*.

Local Branching

The *local branching* (LB) scheme of Fischetti and Lodi [7] appears to be one of the first general-purpose heuristics using a black-box MIP solver applied to subMIPs, and it can be viewed as a precursor of matheuristics. Given a *reference solution* $\bar{x}$ of a MIP with $\mathcal{B} \neq \emptyset$, one aims at finding an improved solution that is "not too far" from $\bar{x}$, in the sense that not too many binary variables need be flipped. To this end, one can define the k-opt neighborhood $\mathcal{N}(\bar{x}, k)$ of $\bar{x}$ as the set of the MIP solutions satisfying the invalid *local branching constraint*

$$\Delta(x, \bar{x}) := \sum_{j \in \mathcal{B}: \bar{x}_j = 0} x_j + \sum_{j \in \mathcal{B}: \bar{x}_j = 1} (1 - x_j) \leq k, \tag{6}$$

for a small neighborhood radius k—an integer parameter typically set to 10 or 20. The neighborhood is then explored (possibly heuristically, i.e., with some small node or time limit) by means of a black-box MIP solver. Experimental results [10] show that the introduction of the local branching constraint typically has the positive effect of driving to integrality many component of the optimal solution of the LP relaxation, improving the so-called relaxation grip and hence the capability of the MIP solver to find (almost) optimal integer solutions within short computing times.

Of course, this effect is lost if parameter k is set to a large value—a mistake that would make local branching completely ineffective.

LB is in the spirit of local search metaheuristics and, in particular, of *large-neighborhood search* (LNS) [29], with the novelty that neighborhoods are obtained through "soft fixing," i.e., through invalid cuts to be added to the original MIP model. Diversification cuts can be defined in a similar way, thus leading to a flexible toolkit for the definition of metaheuristics for general MIPs.

Relaxation-Induced Neighborhood Search

The *relaxation-induced neighborhood search* (RINS) heuristic of Danna, Rothberg, and Le Pape [4] also uses a black-box MIP solver to explore a neighborhood of a given solution $\bar{x}$ and was originally designed to be integrated in a branch-and-bound solution scheme. At specified nodes of the branch-and-bound tree, the current LP relaxation solution x^* and the incumbent $\bar{x}$ are compared, and all integer-constrained variables that agree in value are fixed. The resulting MIP is typically easy to solve, as fixing reduces its size considerably, and often provides improved solutions with respect to $\bar{x}$.

Polishing a Feasible Solution

The *polishing* algorithm of Rothberg [27] implements an evolutionary MIP heuristic which is invoked at selected nodes of a branch-and-bound tree and includes all classical ingredients of genetic computation, namely:

- *Population*: A fixed-size population of feasible solutions is maintained. Those solutions are either obtained within the branch-and-bound tree (by other heuristics) or computed by the polishing algorithm itself.
- *Combination*: Two or more solutions (the parents) are combined with the aim of creating a new member of the population (the child) with improved characteristics. The RINS scheme is adopted, i.e., all variables whose value coincides in the parents are fixed, and the reduced MIP is heuristically solved by a black-box MIP solver within a limited number of branch-and-bound nodes. This scheme is clearly much more time-consuming than a classical combination step in evolutionary algorithms, but it guarantees feasibility of the child solution.
- *Mutation*: Diversification is obtained by performing a classical mutation step that (i) randomly selects a "seed" solution in the population, (ii) randomly fixes some of its variables, and (iii) heuristically solves the resulting reduced MIP.
- *Selection*: Selection of the two parents to be combined is performed by randomly picking a solution in the population and then choosing, again at random, the second parent among those solutions with a better objective value.

Proximity Search

Proximity search [10] is a "dual version" of local branching that tries to overcome the issues related to the choice of the neighborhood radius k. Instead of hard-fixing the radius, proximity search fixes the minimum improvement of the solution value and changes the objective function to favor the search of solutions at small Hamming distance with respect to the reference one.

The approach works in stages, each aimed at producing an improved feasible solution. As in LB or RINS, at each stage a reference solution $\bar{x}$ is given, and one aims at improving it. To this end, an explicit cutoff constraint

$$c^T x \leq c^T \bar{x} - \theta \tag{7}$$

is added to the original MIP, where $\theta > 0$ is a given tolerance that specifies the minimum improvement required. The objective function of the problem can then be replaced by the proximity function $\Delta(x, \bar{x})$ defined in (6), to be minimized. One then applies the MIP solver, as a black box, to the modified problem in the hope of finding a solution better than $\bar{x}$. Computational experience confirms that this approach is quite successful (at least, on some classes of problems), due to the action of the proximity objective function that improves the "relaxation grip" of the model.

A simple variant of the above scheme, called "proximity search with incumbent," is based on the idea of providing $\bar{x}$ to the MIP solver as a staring solution. To avoid $\bar{x}$ be rejected because of the cutoff constraint (7), the latter is weakened to its "soft" version

$$c^T x \leq c^T \bar{x} - \theta(1 - \xi) \tag{8}$$

while minimizing $\Delta(x, \bar{x}) + M\xi$ instead of just $\Delta(x, \bar{x})$, where $\xi \geq 0$ is a continuous slack variable and $M \gg 0$ is a large penalty.

Application 1: Wind Farm Layout Optimization

Green energy became a topic of great interest in recent years, as environmental sustainability asks for a considerable reduction in the use of fossil fuels. The *wind farm layout optimization problem* aims at finding an allocation of turbines in a given site so as to maximize power output. This strategic problem is extremely hard in practice, both for the size of the instances in real applications and for the presence of several nonlinearities to be taken into account. A typical nonlinear feature of this problem is the interaction among turbines, also known as wake effect. The wake effect is the interference phenomenon for which, if two turbines are located one close to another, the upwind one creates a shadow on the one behind. Interference is therefore of great importance in the design of the layout as it results into a loss of power production for the turbine downstream.

We next outline the main steps in the design of a sound matheuristic scheme for wind farm layout optimization that is able to address the large-size instances arising in practical applications.

Choice of the MIP Model

Different models have been proposed in the literature to describe interference. We will consider first a simplified model from the literature [5], where the overall interference is the sum of pairwise interferences between turbine pairs. The model addresses the following constraints:

(a) a minimum and maximum number of turbines that can be built is given;
(b) there should be a minimal separation distance between two turbines to ensure that the blades do not physically clash (turbine distance constraints);
(c) if two turbines are installed, their interference will cause a loss in the power production that depends on their relative position and on wind conditions.

Let V denote the set of possible positions for a turbine, called "sites" in what follows, and let

- N_{MIN} and N_{MAX} be the minimum and maximum number of turbines that can be built, respectively;
- D_{MIN} be the minimum distance between two turbines;
- $dist(i, j)$ be the Euclidean distance between sites i and j;
- I_{ij} be the interference (loss of power) experienced by site j when a turbine is installed at site i, with $I_{jj} = 0$ for all $j \in V$;
- P_i be the power that a turbine would produce if built (alone) at site i.

In addition, let $G_I = (V, E_I)$ denote the incompatibility graph with

$$E_I = \{[i, j] \in V \times V : dist(i, j) < D_{\text{MIN}}, i < j\}$$

and let $n := |V|$ denote the total number of sites. Two sets of binary variables are defined:

$$x_i = \begin{cases} 1 & \text{if a turbine is built at site } i; \\ 0 & \text{otherwise} \end{cases} \qquad (i \in V)$$

$$z_{ij} = \begin{cases} 1 & \text{if two turbines are built at both sites } i \text{ and } j; \\ 0 & \text{otherwise} \end{cases} \qquad (i, j \in V, i < j)$$

The model then reads

$$\max \sum_{i \in V} P_i x_i - \sum_{i \in V} \sum_{j \in V, i<j} (I_{ij} + I_{ji}) z_{ij} \tag{9}$$

$$\text{s.t.} \qquad N_{\text{MIN}} \leq \sum_{i \in V} x_i \leq N_{\text{MAX}} \tag{10}$$

$$x_i + x_j \leq 1 \qquad \forall [i, j] \in E_I \tag{11}$$

$$x_i + x_j - 1 \leq z_{ij} \qquad \forall i, j \in V, i < j \tag{12}$$

$$x_i \in \{0, 1\} \qquad \forall i \in V \tag{13}$$

$$z_{ij} \in \{0, 1\} \qquad \forall i, j \in V, i < j \tag{14}$$

Objective function (9) maximizes the total power production by taking interference losses I_{ij} into account. Constraints (11) model pairwise site incompatibility. Constraints (12) force $z_{ij} = 1$ whenever $x_i = x_j = 1$; because of the objective function, this is in fact equivalent to imposing $z_{ij} = x_i x_j$.

The definition of the turbine power vector (P_i) and of interference matrix (I_{ij}) depends on the wind scenario considered, which greatly varies in time. Using statistical data, one can in fact collect a large number K of wind scenarios k, each associated with a pair (P^k, I^k) with a probability π_k, and define the average power and interference to be used in the model as:

$$P_i := \sum_{k=1}^{K} \pi_k P_i^k \qquad \forall i \in V \tag{15}$$

$$I_{ij} := \sum_{k=1}^{K} \pi_k I_{ij}^k \qquad \forall i, j \in V \tag{16}$$

While (9), (10), (11), (12), (13), and (14) turns out to be a reasonable model when just a few sites have to be considered (say $n \approx 100$), it becomes hopeless when $n \geq 1000$ because of the huge number of variables and constraints involved, which grows quadratically with n. Therefore, when facing instances with several thousand sites, an alternative (possibly weaker) model is required, where interference can be handled by a number of variables and constraints that grows just linearly with n. The model below is a compact reformulation of model (9), (10), (11), (12), (13), and (14) that follows a recipe of Glover [17] that is widely used, e.g., in the quadratic assignment problem [12, 32]. The original objective function (to be maximized), rewritten as

$$\sum_{i \in V} P_i x_i - \sum_{i \in V} \left(\sum_{j \in V} I_{ij} x_j \right) x_i \tag{17}$$

is restated as

$$\sum_{i \in V} (P_i x_i - w_i) \tag{18}$$

where

$$w_i := \left(\sum_{j \in V} I_{ij} x_j \right) x_i = \begin{cases} \sum_{j \in V} I_{ij} x_j & \text{if } x_i = 1 \\ 0 & \text{if } x_i = 0 \end{cases}$$

denotes the total interference caused by site i. Our compact model then reads

$$\max z = \sum_{i \in V} (P_i x_i - w_i) \tag{19}$$

$$\text{s.t.} \quad N_{\text{MIN}} \leq \sum_{i \in V} x_i \leq N_{\text{MAX}} \tag{20}$$

$$x_i + x_j \leq 1 \qquad \forall [i, j] \in E_I \tag{21}$$

$$\sum_{j \in V} I_{ij} x_j \leq w_i + M_i (1 - x_i) \qquad \forall i \in V \tag{22}$$

$$x_i \in \{0, 1\} \qquad \forall i \in V \tag{23}$$

$$w_i \geq 0 \qquad \forall i \in V \tag{24}$$

where the big-M term $M_i = \sum_{j \in V : [i,j] \notin E_I} I_{ij}$ is used to deactivate constraint (22) in case $x_i = 0$, in which case $w_i = 0$ because of the objective function.

Choice of Ad Hoc Heuristics

A simple 1-opt heuristic can be designed along the following lines. At each step, we have an incumbent solution, say $\tilde{x}$, that describes the best-known turbine allocation ($\tilde{x}_i = 1$ if a turbine is built at site i, 0 otherwise), and a current solution x. Let

$$z = \sum_{i \in V} P_i x_i - \sum_{i \in V} \sum_{j \in V} I_{ij} \, x_i \, x_j$$

be the profit of the current solution, $\gamma = \sum_{i \in V} x_i$ be its cardinality, and define for each $j \in V$ the extra-profit δ_j incurred when flipping x_j, namely:

$$\delta_j = \begin{cases} P_j - \sum_{i \in V : x_i = 1} (I_{ij} + I_{ji}) & \text{if } x_j = 0; \\ -P_j + \sum_{i \in V : x_i = 1} (I_{ij} + I_{ji}) & \text{if } x_j = 1 \end{cases}$$

where we assume $I_{ij} = BIG$ for all incompatible pairs $[i, j] \in E_I$, and $BIG > \sum_{i \in V} P_i$ is a large penalty value, while $I_{ii} = 0$ as usual.

We start with $x = 0$, $z = 0$, and $\gamma = 0$ and initialize $\delta_j = P_j$ for all $j \in V$. Then, we iteratively improve x by a sequence of 1-opt moves, according to the following scheme. At each iteration, we look in $O(n)$ time for the site j with maximum $\delta_j + FLIP(j)$, where function $FLIP(j)$ takes cardinality constraints into account, namely

$$FLIP(j) = \begin{cases} -HUGE & \text{if } x_j = 0 \text{ and } \gamma \geq N_{MAX} \\ -HUGE & \text{if } x_j = 1 \text{ and } \gamma \leq N_{MIN} \\ +HUGE & \text{if } x_j = 0 \text{ and } \gamma < N_{MIN} \\ +HUGE & \text{if } x_j = 1 \text{ and } \gamma > N_{MAX} \\ 0 & \text{otherwise} \end{cases}$$

with $HUGE \gg BIG$ (recall that function $\delta_j + FLIP_j$ has to be maximized).

Once the best j has been found, say j^*, if $\delta_{j^*} + FLIP(j^*) > 0$, we just flip x_{j^*}; update x, z, and γ in $O(1)$ time; update all δ_j's in $O(n)$ time (through the parametric technique described in [11]); and repeat. In this way, a sequence of improving solutions is obtained, until a local optimal solution that cannot be improved by just one flip is found. To escape local minima, a simple perturbation scheme can be implemented; see again [11] for details.

A 2-opt heuristic can similarly be implemented to allow a single turbine to move to a better site—a move that requires flipping two variables. Each 2-opt exchange requires $O(n^2)$ time as it amounts to trying n 1-opt exchanges and to apply the best one.

The Overall Matheuristic

Our final approach is a mixture of ad hoc (1- and 2-opt) and general MIP (proximity search with incumbent) heuristics and works as shown in Algorithm 1.

At Step 2, the heuristics of section "Choice of Ad Hoc Heuristics" are applied in their "initial-solution" mode where one starts with $\tilde{x} = x = 0$ and aborts execution when 1-opt is invoked 10,000 consecutive times without improving $\tilde{x}$. At Step 4, instead, a faster "cleanup" mode is applied. As we already have a hopefully good incumbent $\tilde{x}$ to refine, we initialize $x = \tilde{x}$ and repeat the procedure until we count 100 consecutive 1-opt calls with no improvement of $\tilde{x}$. As to time-consuming 2-opt exchanges, they are applied with a certain frequency and in any case just before the final $\tilde{x}$ is returned.

Two different MIP models are used to feed the proximity-search heuristic at Step 6. During the first part of the computation, we use a simplified MIP model obtained from (19), (20), (21), (22), (23), and (24) by removing all interference constraints (22), thus obtaining a much easier problem. A short time limit is imposed for each call of proximity search when this simplified model is solved. In this way we aggressively drive the solution $\tilde{x}$ to increase the number of built turbines,

Algorithm 1: The overall matheuristic framework

1: read input data and compute the overall interference matrix (I_{ij});
2: apply ad hoc heuristics (1- and 2-opt) to get a first incumbent $\tilde{x}$;
3: **while** time limit permits **do**
4: apply quick ad hoc refinement heuristics (few iterations of 1- and 2-opt) to possibly improve $\tilde{x}$;
5: if $n > 2000$, randomly remove points $i \in V$ with $\tilde{x}_i = 0$ so as to reduce the number of candidate sites to 2000;
6: build a MIP model for the resulting subproblem and apply proximity search to refine $\tilde{x}$ until the very first improved solution is found (or time limit is reached);
7: **end while**
8: **return** $\tilde{x}$

without being bothered by interference considerations and only taking pairwise incompatibility (21) into account. This approach quickly finds better and better solutions (even in terms of the true profit), until either (i) no additional turbine can be built or (ii) the addition of new turbines does in fact *reduce* the true profit associated to the new solution because of the neglected interference. In this situation we switch to the complete model (19), (20), (21), (22), (23), and (24) with all interference constraints, which is used in all next executions of Step 6. Note that the simplified model is only used at Step 6, while all other steps of the procedure always use the true objective function that takes interference into full account.

Computational Results

The following alternative solution approaches were implemented in C language, some of which using the commercial MIP-solver IBM ILOG Cplex 12.5.1 [21]; because of the big-Ms involved in the models, all Cplex's codes use zero as integrality tolerance (`CPX_PARAM_EPINT` = 0.0).

(a) `proxy`: The matheuristic outlined in the previous section, built on top of Cplex with the following aggressive parameter tuning: all cuts deactivated, `CPX_PARAM_RINSHEUR` = 1, `CPX_PARAM_POLISHAFTERTIME` = 0.0, `CPX_PARAM_INTSOLLIM` = 2;
(b) `cpx_def`: The application of IBM ILOG Cplex 12.5.1 in its default setting, starting from the same heuristic solution $\tilde{x}$ available right after the first execution of Step 2 of Algorithm 1;
(c) `cpx_heu`: Same as `cpx_def`, with the following internal tuning intended to improve Cplex's heuristic performance: all cuts deactivated, `CPX_PARAM_RINSHEUR` = 100, `CPX_PARAM_POLISHAFTERTIME` = 20% of the total time limit;

(d) `loc_sea`: A simple heuristic not based on any MIP solver, that just loops on Steps 4 of Algorithm 1 and randomly removes installed turbines from the current best solution after 10,000 iterations without improvement of the incumbent.

For each algorithm, we recorded the best solution found within a given time limit.

In our view, `loc_sea` is representative of a clever but not oversophisticated metaheuristic, as typically implemented in practice, while `cpx_def` and `cpx_heu` represent a standard way of exploiting a MIP model once a good feasible solution is known.

Our test bed refers to an offshore 3,000 × 3,000 (m) square with $D_{\text{MIN}} = 400$ (m) minimum turbine separation, with no limit on the number of turbines to be built (i.e., $N_{\text{MIN}} = 0$ and $N_{\text{MAX}} = +\infty$). Turbines are all of Siemens SWT-2.3-93 type (rotor diameter 93 m), which produces a power of 0.0 MW for wind speed up to 3 m/s, of 2.3 MW for wind speed greater than or equal to 16 m/s, and intermediate values for winds in range 3–16 m/s according to a nonlinear power curve [30]. Pairwise interference (in MW) was computed using Jensen's model [22], by averaging 250,000+ real-world wind samples. Those samples were grouped into about 500 macro-scenarios to reduce the computational time spent for the definition of the interference matrix. A pairwise average interference of 0.01 MW or less was treated as zero. The reader is referred to [6] for details.

We generated five classes of medium-to-large problems with n ranging from 1,000 to 20,000. For each class, ten instances have been considered by generating n uniformly random points in the 3,000 × 3,000 square. (Although in the offshore case turbine positions are typically sampled on a regular grid, we decided to randomly generate them to be able to compute meaningful statistics for each value of n.)

In what follows, reported computing times are in CPU sec.s of an Intel Xeon E3-1220 V2 quad-core PC with 16GB of RAM and do not take Step 1 of Algorithm 1 into account as the interference matrix is assumed to be precomputed and reused at each run.

Computational results on our instances are given in Table 1, where each entry refers to the performance of a given algorithm at a given time limit. In particular, the left part of the table reports, for each algorithm and time limit, the *number of wins*, i.e., the number of instances for which a certain algorithm produced the best-known solution at the given time limit (ties allowed).

According to the table, `proxy` outperforms all competitors by a large amount for medium-to-large instances. As expected, `cpx_heu` performs better for instances with $n = 1{,}000$ as it is allowed to explore a large number of enumeration nodes for the original model and objective function. Note that `loc_sea` has a good performance for short time limits and/or for large instances, thus confirming its effectiveness, whereas `cpx_heu` is significantly better than `loc_sea` only for small instances and large time limits.

A different performance measure is given in the right-hand side part of Table 1, where each entry gives the *average optimality ratio*, i.e., the average value of the

Table 1 Number of times each algorithm finds the best-known solution within the time limit (wins) and optimality ratio with respect to the best-known solution—the larger, the better

		Number of wins				Optimality ratio			
n	Time limit (s)	proxy	cpx_def	cpx_heu	loc_sea	proxy	cpx_def	cpx_heu	loc_sea
1,000	60	6	1	3	0	0.994	0.983	0.987	0.916
	300	4	2	4	0	0.997	0.991	0.998	0.922
	600	7	3	7	0	0.997	0.992	0.997	0.932
	900	5	2	3	0	0.998	0.993	0.996	0.935
	1,200	5	1	5	0	0.998	0.992	0.997	0.939
	1,800	5	1	4	0	0.998	0.992	0.996	0.942
	3,600	4	2	5	0	0.998	0.995	0.997	0.943
5,000	60	9	6	6	5	0.909	0.901	0.901	0.904
	300	10	0	0	0	0.992	0.908	0.908	0.925
	600	10	0	10	0	0.994	0.908	0.994	0.935
	900	10	0	0	0	0.994	0.908	0.908	0.936
	1,200	10	0	0	0	0.994	0.908	0.925	0.939
	1,800	9	0	1	0	0.996	0.908	0.971	0.946
	3,600	5	0	5	0	0.996	0.932	0.994	0.948
10,000	60	9	9	8	10	0.914	0.913	0.914	0.914
	300	10	2	2	2	0.967	0.927	0.927	0.936
	600	10	0	10	0	0.998	0.928	0.998	0.944
	900	10	0	0	0	1.000	0.928	0.928	0.948
	1,200	10	0	0	0	1.000	0.928	0.928	0.951
	1,800	10	0	0	0	1.000	0.928	0.928	0.957
	3,600	9	0	0	1	1.000	0.928	0.928	0.964
15,000	60	9	10	9	9	0.909	0.912	0.911	0.909
	300	10	8	7	8	0.943	0.937	0.935	0.937
	600	10	0	10	0	0.992	0.939	0.992	0.942
	900	10	0	0	0	1.000	0.939	0.939	0.956
	1,200	9	0	0	1	1.000	0.939	0.939	0.959
	1,800	9	0	0	1	1.000	0.939	0.939	0.965
	3,600	9	0	0	1	1.000	0.939	0.939	0.972
20,000	60	9	9	9	10	0.901	0.902	0.901	0.902
	300	10	8	10	10	0.933	0.933	0.933	0.933
	600	9	0	9	1	0.956	0.935	0.956	0.941
	900	10	0	0	0	0.978	0.935	0.935	0.945
	1,200	10	0	0	0	0.991	0.935	0.935	0.950
	1,800	10	0	0	0	0.999	0.935	0.935	0.963
	3,600	10	0	0	0	1.000	0.935	0.935	0.971
ALL	60	42	35	35	34	0.925	0.922	0.922	0.909
	300	44	20	23	20	0.966	0.939	0.940	0.930
	600	46	3	46	1	0.987	0.941	0.987	0.938
	900	45	2	3	0	0.994	0.941	0.941	0.944
	1,200	44	1	5	1	0.997	0.940	0.945	0.947
	1,800	43	1	5	1	0.999	0.940	0.954	0.955
	3,600	36	2	10	2	0.999	0.946	0.959	0.959

ratio between the solution produced by an algorithm (on a given instance at a given time limit) and the best solution known for that instance—the closer to one, the better. It should be observed that an improvement of just 1% has a very significant economical impact due to the very large profits involved in the wind farm context. The results show that `proxy` is always able to produce solutions that are quite close to the best one. As before, `loc_sea` is competitive for large instances when a very small computing time is allowed, whereas `cpx_def` and `cpx_heu` exhibit a good performance only for small instances and are dominated even by `loc_sea` for larger ones.

Application 2: Prepack Optimization

Packing problems play an important role in industrial applications. In these problems, a given set of *items* has to be packed into one or more containers (*bins*) so as 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* and has been widely studied in the literature both in its one-dimensional version [25] and in its higher-dimensional variants [23].

We will next 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 themselves. This is the case, for example, for 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 limit this cost, a hard constraint can be imposed on the total number of different box configurations that are used.

Prepacking 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. In addition, an upper bound on overstocking is usually imposed for each store.

The resulting problem, known as *prepack optimization problem* (POP), was recently addressed in [20], where a real-world application in the fashion industry is presented, and heuristic approaches are derived using both constraint programming (CP) and MIP techniques.

Mathematical Model

In this section we briefly formalize POP and review the mathematical model introduced in [20]. 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 y_{bi} 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 overstocking, 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 if 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:

$$\min \sum_{s \in S} \sum_{i \in I} (\alpha u_{is} + \beta o_{is}) \tag{25}$$

$$q_{bis} = x_{bs} y_{bi} \quad (b \in B; i \in I; s \in S) \tag{26}$$

$$\sum_{b \in B} q_{bis} - o_{is} + u_{is} = r_{is} \quad (i \in I; s \in S) \tag{27}$$

$$\sum_{i \in I} y_{bi} = \sum_{k \in K} k\, t_{bk} \quad (b \in B) \tag{28}$$

$$\sum_{k \in K} t_{bk} = 1 \quad (b \in B) \tag{29}$$

$$o_{is} \leq \delta_{is} \quad (i \in I; s \in S) \tag{30}$$

$$t_{bk} \in \{0, 1\} \quad (b \in B; k \in K) \tag{31}$$

$$x_{bs} \geq 0 \text{ integer} \quad (b \in B; s \in S) \tag{32}$$

$$y_{bi} \geq 0 \text{ integer} \quad (b \in B; i \in I) \tag{33}$$

The model is of course nonlinear, as the bilinear constraints (26) involve the product of decision variables. To derive a linear MIP model, the following standard technique can be used. Each x_{bs} variable is decomposed into its binary expansion using binary variables v_{bsl} ($l = 0, \ldots, L$), 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 final MIP is therefore obtained from (25), (26), (27), (28), (29), (30), (31), (32), and (33) by adding

$$x_{bs} = \sum_{l=0}^{L} 2^l v_{bsl} \quad (b \in B; s \in S) \tag{34}$$

$$v_{bsl} \in \{0, 1\} \quad (b \in B; s \in S; l = 0, \ldots, L) \tag{35}$$

and by replacing each nonlinear equation (26) with the following set of new variables and constraints:

$$q_{bis} = \sum_{l=0}^{L} 2^l w_{bisl} \quad (b \in B; i \in I; s \in S) \tag{36}$$

$$w_{bisl} \leq \overline{Y} v_{bsl} \quad (b \in B; i \in I; s \in S; l = 0, \ldots, L) \tag{37}$$

$$w_{bisl} \leq y_{bi} \quad (b \in B; i \in I; s \in S; l = 0, \ldots, L) \tag{38}$$

$$w_{bisl} \geq y_{bi} - \overline{Y}(1 - v_{bsl}) \quad (b \in B; i \in I; s \in S; l = 0, \ldots, L) \tag{39}$$

$$w_{bisl} \geq 0 \quad (b \in B; i \in I; s \in S; l = 0, \ldots, L) \tag{40}$$

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

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

$$\sum_{i \in I} (u_{is} + o_{is}) \geq 1 \quad \left(s \in S : \sum_{i \in I} r_{is} \text{ is odd} \right) \tag{41}$$

Matheuristics

The MIP model of the previous subsection is by far too difficult to be addressed 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. So we designed ad hoc heuristic

approaches to exploit the special structure of our MIP, following the matheuristic paradigm.

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

Two kinds of heuristics can be implemented: constructive and refinement heuristics. Constructive heuristics are used to find a solution H starting from scratch. In a refinement heuristic, instead, we are given a 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 MIP 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.

Fixing all *x* or *y* Variables

A first obvious observation is that our basic MINLP model (25), (26), (27), (28), (29), (30), (31), (32), and (33) reduces to a linear MIP if all the x (or all the y) variables are fixed, as constraints (26) trivially become linear. According to our experience, the resulting MIP (though nontrivial) is typically solved very quickly by a state-of-the-art solver, meaning that one can effectively solve a sequence of restricted MIPs where x and y are fixed, 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 $\mathcal{N}_x(H)$, $\mathcal{N}_y(H)$, $\mathcal{N}_x(H)$, etc. after having removed all stores not in S', H being updated after each optimization.

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) configuration 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 (34). 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.

Working with a Subset of Stores

Our basic constructive heuristic is based on the observation that removing stores can produce a substantially easier model. Note 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 `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 MIP model just produces a y 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 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 overall incumbent where all stores are active.

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

- `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.

Computational Experiments

The heuristics described in the previous section have been implemented in C language. IBM ILOG CPLEX 12.6 [21] was used as MIP 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 time limit of 900 s (15 m) was imposed.

Four heuristic methods have been compared: the three construction heuristics `random`, `most_dissimilar` and `most_similar` of section "Working with a Subset of Stores," plus

- `fast_heu`: The fast refinement heuristic of section "Fixing y Variables For All but One Configuration" 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 five random pair swaps. In addition, after each restart the CPLEX's random seed is changed so as to inject diversification among runs even within the MIP solver.

Due to their heuristic nature, our methods—though 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) random seeds at each run, and also report average figures.

In case all capacities are even (as it is the case in our tesbed), we compute the following trivial lower bound based on constraint (41)

$$LB := \min\{\alpha, \beta\} \cdot \left| \left\{ s \in S : \sum_{i \in I} r_{is} \text{ is odd} \right\} \right| \tag{42}$$

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

Test Bed

Our test bed coincides with the benchmark proposed in [20] and contains a number of subinstances of a real-world problem (named `AllColor58`) 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 given 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.

Note that our testing environment is identical to that used in [20] (assuming the PC used are substantially equivalent), so our results can fairly be benchmarked against those therein reported.

Comparison Metric

To better compare the performance of our different heuristics, we also make use of an indicator recently proposed by [1, 2] 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 $\tilde{z}_{opt}$ 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

$$p(t) = \begin{cases} 1 & \text{if no incumbent found until time } t \\ \gamma(z(t)) & \text{otherwise} \end{cases} \tag{43}$$

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

$$\gamma(z) = \begin{cases} 0 & \text{if } |\tilde{z}_{opt}| = |z| = 0, \\ 1 & \text{if } \tilde{z}_{opt} \cdot z < 0, \\ \frac{z - \tilde{z}_{opt}}{\max\{|\tilde{z}_{opt}|, |z|\}} & \text{otherwise.} \end{cases} \tag{44}$$

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

$$P(t_{\max}) = \frac{\int_0^{t_{\max}} p(t)\, \mathrm{d}t}{t_{\max}} \tag{45}$$

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 $t_{\max}$.

Computational Results

We addressed the instances provided in [20], with the aim of benchmarking our matheuristics 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 MIP model can be solved to proven optimality in less than one second by our solver—thus confirming the figures given in [20].

Tables 2 and 3 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 2 is taken from [20], where a two-phase hybrid metaheuristic was proposed. In the first phase, the approach uses a memetic algorithm to explore the solution space and builds a pool of interesting box configurations. In the

Table 2 Performance of CPLEX and LNS heuristics from [20]. Single run for each instance. Times in CPU seconds (time limit of 900 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

Table 3 Performance of matheuristics. Single run for each instance. Times in CPU seconds (time limit of 900 s)

						`most`		`most`	
		`fast_heu`		`random`		`dissimilar`		`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

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) MIP and heuristically solved either by a commercial solver (CPLEX) or by a sophisticated large-neighborhood search (LNS) approach.

Table 3 reports the performance of our four matheuristics, as well as the lower bound value LB computed through (42)—this latter value turned out to coincide with the optimal value for all instances under consideration in the present subsection.

Comparing Tables 2 and 3 shows that matheuristics outperform the LNS heuristics analyzed in [20]. In particular, `fast_heu` is able to find very good solutions (actually, 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.

To get more reliable information about the matheuristics' performance, we ran them 100 times for each instance, with different random seeds, and took detailed statistics on each run. Table 4 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 s (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 bound (42) was eventually found by at least one of our heuristics. 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 with respect to the #opt criterion. 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 instances.

Table 4 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

Application 3: Vehicle Routing

In this section we address the NP-hard ***distance-constrained capacitated vehicle routing problem*** (DCVRP) that can be defined as follows. We are given a central depot and a set of $n-1$ customers, which are associated with the nodes of a complete undirected graph $G = (V, E)$ where $|V| = n$, node 1 representing the depot. Each edge $[i, j] \in E$ has an associated finite cost $c_{ij} \geq 0$. Each node $j \in V$ has a request $d_j \geq 0$ ($d_1 = 0$ for depot node 1). Customers need to be served by k cycles (*routes*) passing through the depot, where k is fixed in advance. Each route must have a total duration (computed as the sum of the edge costs in the route) not exceeding a given limit D and can visit a subset S of customers whose total request $\sum_{j \in S} d_j$ does not exceed a given capacity C. The problem then consists of finding a feasible solution covering exactly once all the nodes $v \in V \setminus \{1\}$ and having a minimum overall cost; see, e.g., [3, 31].

We will next outline the refinement matheuristic for DCVRP proposed in [15].

The ASSIGN Neighborhood for TSP

Sarvanov and Doroshko (SD) investigated in [28] the so-called ASSIGN neighborhood for the pure Traveling Salesman Problem (TSP), i.e., for the problem of finding a min-cost Hamiltonian cycle in a graph. Given a certain TSP solution (viewed as node sequence $< v_1 = 1, v_2, \cdots, v_n >$), the neighborhood contains all the $\lfloor n/2 \rfloor !$ TSP solutions that can be obtained by permuting, in any possible way, the nodes in

even position in the original sequence. In other words, any solution $(\psi_1, \psi_2, \cdots, \psi_n)$ in the neighborhood is such that $\psi_i = v_i$ for all odd i. An interesting feature of the neighborhood is that it can be explored exactly in polynomial time, though it contains an exponential number of solutions. Indeed, for any given starting solution, the min-cost TSP solution in the corresponding ASSIGN neighborhood can be found efficiently by solving a min-cost assignment problem on a $\lfloor n/2 \rfloor \times \lfloor n/2 \rfloor$ matrix; see, e.g., [18]. Starting from a given solution, an improving heuristic then consists of exploring the ASSIGN neighborhood according to the following two phases:

- *node extraction*, during which the nodes in even position (w.r.t. the current solution) are removed from the tour, thus leaving an equal number of "free holes" in the sequence;
- *node reinsertion*, during which the removed nodes are reallocated to the available holes in an optimal way by solving a min-sum assignment problem.

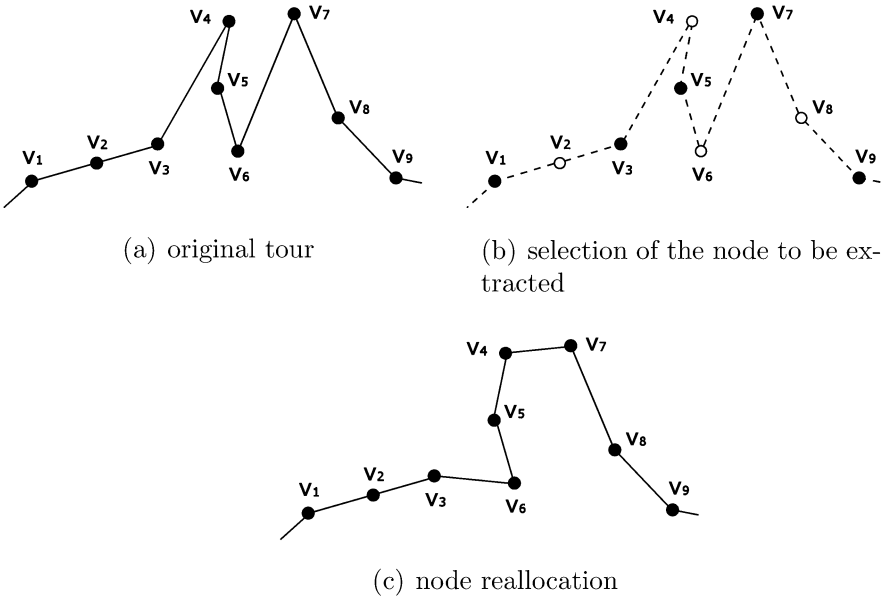
The simple example in Fig. 1 gives an illustration of the kind of "improving moves" involved in the method. The figure draws a part of a tour, corresponding to the node sequence $\langle v_1, v_2, \cdots, v_9 \rangle$. In the node extraction phase, the nodes in even position v_2, v_4, v_6 e v_8 are removed from the sequence, whereas all the other nodes retain their position. In Fig. 1b the black nodes represent the fixed ones, while the holes left by the extracted nodes are represented as white circles. If we use symbol "–" to represent a free hole, the sequence corresponding to Fig. 1b is therefore $\langle v_1, -, v_3, -, v_5, -, v_7, -, v_9 \rangle$. The second step of the procedure, i.e., the optimal node reallocation, is illustrated in Fig. 1c, where nodes v_4 and v_6 swap their position, whereas v_2 and v_8 are reallocated as in the original sequence. This produces the improved part of tour $\langle v_1, v_2, v_3, v_6, v_5, v_4, v_7, v_8, v_9 \rangle$.

In the example, the same final tour could have been constructed by a simple 2-opt move. However, for more realistic cases, the number of possible reallocation is exponential in the number of extracted nodes; hence, the possible reallocation patterns are much more complex and allow, e.g., for a controlled worsening of some parts of the solution which are compensated by large improvement in other parts.

From TSP to DCVRP

One can conjecture that the ASSIGN neighborhood would work well if applied to VRP problems. Indeed, due to the presence of several routes and of the associated route constraints, in VRP problems the node sequence is not the only issue to be considered when constructing a good solution: an equally important aspect of the optimization is to find a balanced distribution of the nodes between the routes. In this respect, heuristic refinement procedures involving complex patterns of node reallocations among the routes likely are quite effective in practice.

We can therefore extend the SD method to DCVRP so as to allow for more powerful move patterns, while generalizing its basic scheme so as to get rid of the



(a) original tour

(b) selection of the node to be extracted

(c) node reallocation

Fig. 1 A simple example of node extraction and reallocation

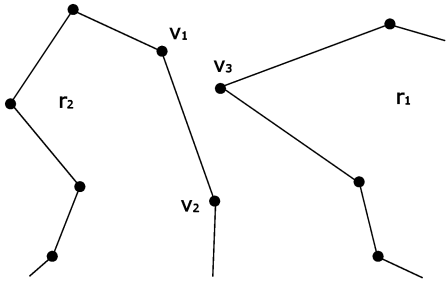


Fig. 2 The assignment of node v_3 to route r_1 is nonoptimal

too simple min-sum assignment model for node reallocation in favor of a more flexible reallocation model based on the (heuristic) solution of a more complex MIP model. The resulting matheuristic will be introduced, step by step, with the help of some illustrative examples.

Let us consider Fig. 2, where a nonoptimal part of a VRP solution is depicted. It is clear that the position of node v_3 is not very clever, in that inserting v_3 between node v_1 and v_2 is likely to produce a better solution (assuming this new solution is not infeasible because of the route constraints). Even if v_3 were an even position node, however, this move would be beyond the possibility of the pure SD method, where the extracted nodes can only be assigned to a hole left free by the removal of another node—while no hole between v_1 e v_2 exists which could accommodate v_3. The example then suggests a first extension of the basic SD method, consisting of removing the 1-1 correspondence between extracted nodes and empty holes. We

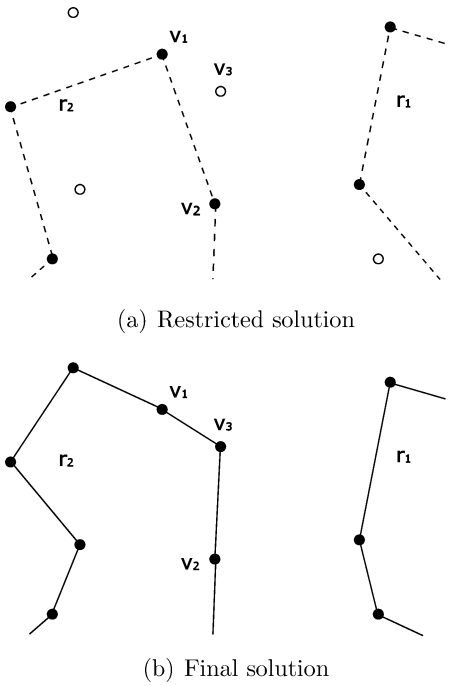


Fig. 3 Improving the solution depicted in Fig. 2

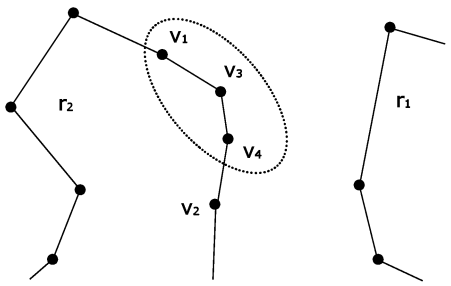


Fig. 4 Removing a sequence of nodes to better reallocate them (possibly in a different order)

therefore consider the concept of *insertion point*: after having extracted the selected nodes, we construct a *restricted solution* through the remaining nodes, obtained from the original one by short-cutting the removed nodes. All the edges in the restricted solution are then viewed as potential insertion points for the extracted nodes. In the example, removing v_3 but not v_1 and v_2 would produce the restricted solution depicted in Fig. 3a, where all dashed edges are possible insertion points for the extracted nodes—this allows the method to produce the solution in Fig. 3b.

A second important extension comes from a more flexible node extraction scheme that allows for the removal of a *sequence* of nodes; see Fig. 4 for an illustration. Once a sequence of nodes has been extracted, one can use a heuristic procedure to generate *new* sequences through the extracted nodes, to be allocated to different insertion points. To be more specific, starting from the extracted node

sequences, one can create new *derived sequences* that combine the extracted nodes in a different way and consider the possibility of assigning each derived sequence to a different insertion point. Of course, one never knows in advance which are the best sequences to be used, so all the (original and derived) sequences should be available when solving the reallocation problem.

The above considerations imply the use of a reallocation model which goes far beyond the scope of the original one, which is based on the solution of an easy min-cost assignment problem. Indeed, the new reallocation model becomes a MIP that receives as input the set of insertion points along with a (typically large) set of node sequences through the extracted nodes and provides an (almost) optimal allocation of at most one sequence to each insertion point, with the constraint that each extracted node has to belong to one of the allocated sequences, while fulfilling the additional constraints on the capacity and distance constraints on the routes. This model will be described in more detail in the next section.

The Overall Matheuristic

Here is a possible implementation of the ideas outlined in the previous section, leading to the so-called *selection, extraction, recombination, and reallocation* (SERR) matheuristic.

(i) (*Initialization*). Apply a fast heuristic method to find a first (possibly infeasible, see below) DCVRP solution.
(ii) (*Selection*). Apply one of the available criteria (to be described later) to determine the nodes to be extracted—the nodes need not be consecutive, and any node subset qualifies as a valid choice.
(iii) (*Extraction*). Extract the nodes selected in the previous step and construct the corresponding restricted DCVRP solution obtained by short-cutting the extracted nodes. All edges in the restricted solution are put in the list $\mathcal{I}$ of the available insertion points.
(iv) (*Recombination*). The node sequences extracted in the previous step (called *basic* in the sequel) are initially stored in a *sequence pool*. Thereafter, heuristic procedures (to be described later) are applied to derive new sequences through the extracted nodes, which are added to the sequence pool. During this phase, dual information derived from the LP relaxation of the reallocation model can be used to find new profitable sequences—the so-called *pricing* step. Each sequence s in the final pool is then associated with a (heuristically determined) subset I_s of the available insertion points in $\mathcal{I}$. For all basic sequences s, we assume that $\mathcal{I}_s$ contains (among others) the *pivot* insertion point associated to s in the original tour, so as to make it feasible to retrieve the original solution by just reallocating each basic sequence to the associated pivot insertion point.
(v) (*Reallocation*). A suitable MIP (to be described later in greater details) is set up and solved heuristically through a general-purpose MIP solver. This model has a binary variable x_{si} for each pair (s, i), where s is a sequence in the

pool and $i \in \mathcal{I}_s$, whose value 1 means that s has to be allocated to i. The constraints in the MIP stipulate that each extracted node has to be covered by exactly one of the selected sequences s, while each insertion point i can be associated to at most one sequence. Further constraints impose the capacity and distance constraints in each route. Once an (almost) optimal MIP solution has been found, the corresponding DCVRP solution is constructed and the current best solution is possibly updated (in which case each route in the new solution is processed by a 3-opt [26] exchange heuristic in the attempt of further improving it).

(vi) (*Termination*). If at least one improved DCVRP solution has been found in the last n iterations, we repeat from step (ii); otherwise the method terminates.

Finding a Starting Solution

Finding a DCVRP solution that is guaranteed to be feasible is an NP-hard problem; hence, we have to content ourselves with the construction of solutions that, in some hard cases, may be infeasible—typically because the total-distance-traveled constraint is violated for some routes. In this case, the overall infeasibility of the starting solution can hopefully be driven to zero by a modification of the SERR recombination model where the capacity and distance constraints are treated in a soft way through the introduction of highly penalized slack variables.

As customary in VRP problems, we assume that each node is assigned a coordinate pair (x, y) giving the geographical position of the corresponding customer/depot in a two-dimensional map.

One option for the initialization of the current solution required at step (i) of the SERR method is to apply the classical two-phase method of Fisher and Jaikumar (FJ) [14]. This method can be implemented in a very natural way in our context in that it is based on a (heuristic) solution of a MIP whose structure is close to that of the reallocation model.

According to computational experience, however, the solution provided by the FJ heuristic is sometimes "too balanced," in the sense that the routes are filled so tightly that leave not enough freedom to the subsequent steps of our SERR procedure. Better results are sometimes obtained starting from a less-optimized solution whose costs significantly exceed the optimal cost as, e.g., the one obtained by using a simplified *SWEEP* method [16].

A second possibility is instead to start from an extremely good solution provided by highly effective (and time-consuming) heuristics or metaheuristics, in the attempt of improving this solution even further.

Node selection criteria

At each execution of step (ii), one of the following selection schemes is applied.

- scheme RANDOM-ALTERNATE: This criterion is akin to the SD one and selects in some randomly selected routes all the nodes in even position, while in the remaining routes the extracted nodes are those in odd position—the position parity being determined by visiting each route in a random direction.

- scheme SCATTERED: Each node had a uniform probability of 50% of being extracted; this scheme allows for the removal of consecutive nodes, i.e., of route subsequences.
- scheme NEIGHBORHOOD: Here one concentrates on a seed node, say v^*, and removes the nodes v with a probability that is inversely proportional to the distance c_{vv^*} of v from v^*.

Schemes RANDOM-ALTERNATE and SCATTERED appear particularly suited to improve the first solutions, whereas the NEIGHBORHOOD scheme seems more appropriate to deal with the solutions available after the first iterations.

Reallocation Model

Given the sequences stored in the pool and the associated insertion points (defined through the heuristics outlined in the next subsection), our aim is to reallocate the sequences so as to find a feasible solution of improved cost (if any). To this end, we need to introduce some additional notation.

Let $\mathcal{F}$ denote the set of the extracted nodes, $\mathcal{S}$ the sequence pool, and $\mathcal{R}$ the set of routes r in the restricted solution. For any sequence $s \in \mathcal{S}$, let $c(s)$ be the sum of the costs of the edges in the sequence, and let $d(s)$ be the sum of the requests d_j associated with the *internal* nodes of s.

For each insertion point $i \in \mathcal{I}$, we define the extra-cost γ_{si} for assigning sequence s (in its best possible orientation) to the insertion point i. For each route $r \in \mathcal{R}$ in the restricted solution, let $\mathcal{I}(r)$ denote the set of the insertion points (i.e., edges) associated with r, while let $\tilde{d}(r)$ and $\tilde{c}(r)$ denote, respectively, the total request and distance computed for route r—still in the restricted tour. As already mentioned, our MIP model is based on the following decision variables:

$$x_{si} = \begin{cases} 1 \text{ if sequence } s \text{ is allocated to the insertion point } i \in \mathcal{I}_s \\ 0 \text{ otherwise} \end{cases} \tag{46}$$

The model then reads:

$$\sum_{r \in \mathcal{R}} \tilde{c}(r) + \min \sum_{s \in \mathcal{S}} \sum_{i \in \mathcal{I}_s} \gamma_{si} x_{si} \tag{47}$$

subject to:

$$\sum_{s \in v} \sum_{i \in \mathcal{I}_s} x_{si} = 1 \quad \forall v \in \mathcal{F} \tag{48}$$

$$\sum_{s \in \mathcal{S}: i \in \mathcal{I}_s} x_{si} \leq 1 \quad \forall i \in \mathcal{I} \tag{49}$$

$$\tilde{d}(r) + \sum_{s \in \mathcal{S}} \sum_{i \in \mathcal{I}_s \cap \mathcal{I}(r)} d(s) x_{si} \leq C \quad \forall r \in \mathcal{R} \tag{50}$$

$$\tilde{c}(r) + \sum_{s \in \mathcal{S}} \sum_{i \in \mathcal{I}_s \cap \mathcal{I}(r)} \gamma_{si} x_{si} \leq D \quad \forall s \in \mathcal{S},\ r \in \mathcal{R} \tag{51}$$

$$0 \leq x_{si} \leq 1 \ \text{ integer} \quad \forall s \in \mathcal{S},\ i \in \mathcal{I}_s \tag{52}$$

The objective function, to be minimized, gives the cost of the final DCVRP solution. Indeed, each objective coefficient gives the MIP cost of an inserted sequence, including the linking cost, minus the cost of the "saved" edge in the restricted solution. Constraints (48) impose that each extracted node belongs to exactly one of the selected sequences, i.e., that it is covered exactly once in the final solution. Note that, in the case of triangular costs, one could replace $=$ by $\geq$ in (48), thus obtaining a typically easier-to-solve MIP having the structure of a set-covering (instead of set-partitioning) problem with side constraints. Constraints (49) avoid a same insertion point be used to allocate two or more sequences. Finally, constraints (50) and (51) impose that each route in the final solution fulfills the capacity and distance restriction, respectively.

In order to avoid to overload the model by an excessive number of variables, a particular attention has to be paid to reduce the number of sequences and, for each sequence, the number of the associated insertion points.

Node Recombination and Construction of Derived Sequences

This is a very crucial step in the SERR method. It consists not just of generating new "good" sequences through the extracted nodes, but it also associates each sequence to a clever set of possible insertion points that can conveniently accommodate it. Therefore, one has two complementary approaches to attack this problem: (a) start from the insertion points and, for each insertion point, try to construct a reasonable number of new sequences which are likely to "fit well" or (b) start from the extracted nodes and try to construct new sequences of small cost, no matter the position of the insertion points. The following two-phase method turned out to be a good strategy in practice.

In the first phase, the sequence pool is initialized by means of the original (basic) sequences, and each of them is associated to its corresponding (pivot) insertion point. This choice guarantees that the current DCVRP solution can be reconstructed by simply selecting all the basic sequences and putting them back in their pivot insertion point. Moreover, when the NEIGHBORHOOD selection scheme is used, a further set of sequences is generated as follows. Let v^* be the extracted seed node, and let $N(v^*)$ contain v^* plus the, say, k closest extracted nodes ($k = 4$ in our implementation). A complete enumerative scheme is applied to generate all the sequences through $N(v^*)$ that are added to the pool. This choice is intended to increase the chances of locally improving the current solution, by exploiting appropriate sequences to reallocate the nodes in $N(v^*)$ in an optimal way.

The second phase is only applied for the NEIGHBORHOOD and SCATTERED selection schemes and corresponds to a pricing loop based on the dual information available after having solved the LP relaxation of the current reallocation model. The reader is again referred to [15] for details.

Examples

Extensive computational results for SERR matheuristic have been reported [15] and are omitted because of space. We only report in Figs. 5 and 6 a plot of the incumbent SERR solution for some instances from the literature. Computing time is given in CPU seconds of an old AMD Athlon XP 2400+ PC with 1 GByte RAM, using ILOG Cplex 8.0 as MIP solver. The figures show that SERR is able to significantly improve the starting solution in the early part of its computation.

Conclusion

Contamination of metaheuristics with mathematical programming leads to the concept of "matheuristics." The result is a general approach to design mathematically sound heuristics. In this chapter we presented the main ideas underlying matheuristics, and used some case studies to illustrate them. For each application, we described the specific problem at hand, the mathematical programming model

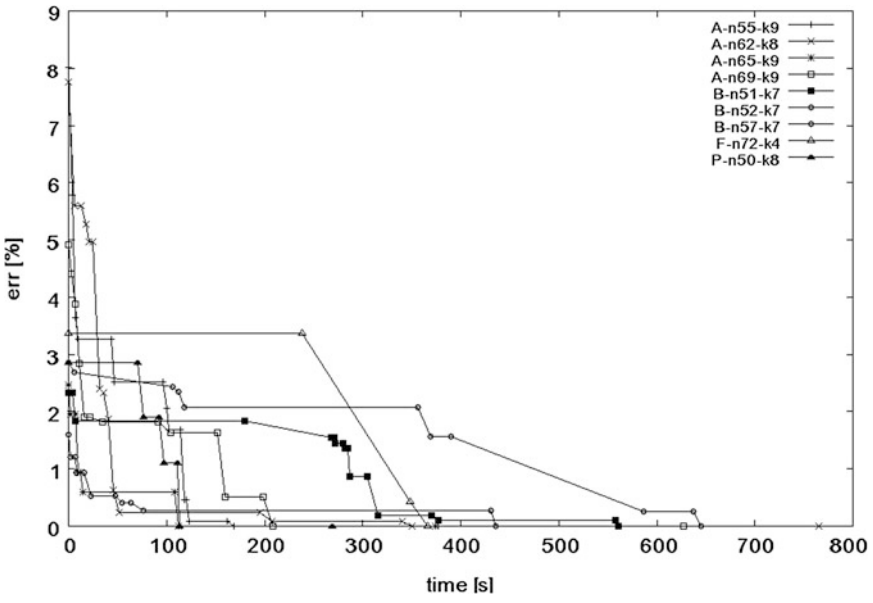


Fig. 5 Time evolution of the SERR solution for various CVRP instances, with FJ [14] initial solution

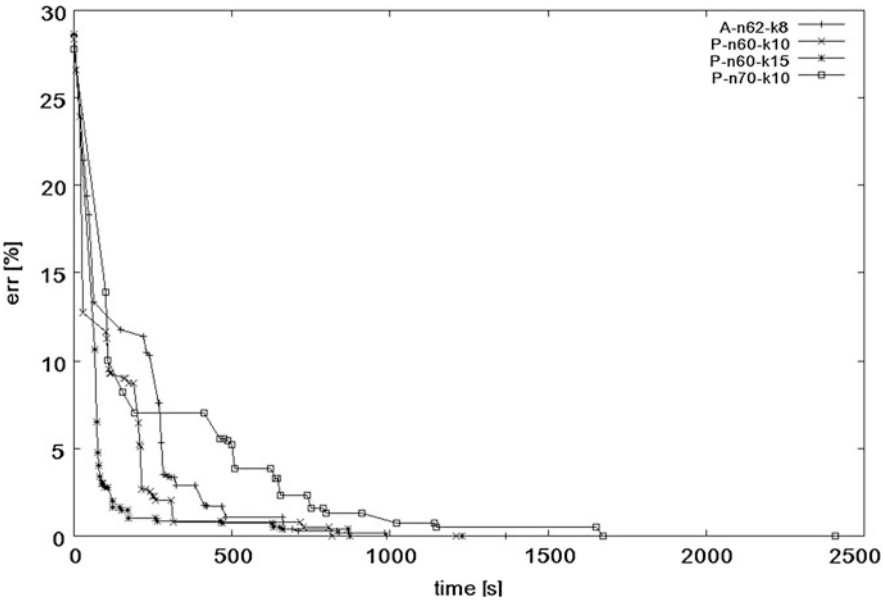


Fig. 6 Time evolution of the SERR solution for various CVRP instances, with SWEEP [16] initial solution

that formalizes it, and the way the model—or a simplification of it—can be used to produce heuristic (as opposed to exact) solutions in an effective way.

Cross-References

- ▸ Adaptive and Multilevel Metaheuristics
- ▸ Constraint-Based Local Search
- ▸ Iterated Local Search

References

1. Achterberg T, Berthold T, Hendel G (2012) Rounding and propagation heuristics for mixed integer programming. In: Operations research proceedings 2011, Zurich, pp 71–76
2. Berthold T (2013) Measuring the impact of primal heuristics. Oper Res Lett 41(6):611–614
3. Christofides N, Mingozzi A, Toth P (1979) The vehicle routing problem. Combinatorial optimization. Wiley, New York
4. Danna E, Rothberg E, Le Pape C (2005) Exploring relaxation induced neighborhoods to improve MIP solutions. Math Program 102:71–90
5. Donovan S (2005) Wind farm optimization. In: Proceedings of the 40th annual ORSNZ conference, Wellington, pp 196–205
6. Fischetti M (2014) Mixed-integer models and algorithms for wind farm layout optimization. Master's thesis, University of Padova. http://tesi.cab.unipd.it/45458/1/tesi_Fischetti.pdf

7. Fischetti M, Lodi A (2003) Local branching. Math Program 98:23–47
8. Fischetti M, Lodi A (2011) Heuristics in mixed integer programming. In: Cochran JJ (ed) Wiley encyclopedia. Volume 8 of operations research and management science. John Wiley & Sons, Hoboken, pp 738–747
9. Fischetti M, Lodi A, Salvagnin D (2010) Just MIP it! In: Maniezzo V, Stützle T, Voß S (eds) Matheuristics. Volume 10 of annals of information systems. Springer, Boston, pp 39–70
10. Fischetti M, Monaci M (2014) Proximity search for 0–1 mixed-integer convex programming. J Heuristics 6(20):709–731
11. Fischetti M, Monaci M (2016) Proximity search heuristics for wind farm optimal layout. J Heuristics 22(4):459–474
12. Fischetti M, Monaci M, Salvagnin D (2012) Three ideas for the quadratic assignment problem. Oper Res 60(4):954–964
13. Fischetti M, Monaci M, Salvagnin D (2015, to appear) Mixed-integer linear programming heuristics for the prepack optimization problem. Discret Optim
14. Fisher ML, Jaikumar R (1981) A generalized assignment heuristic for vehicle routing. Networks 11:109–124
15. De Franceschi R, Fischetti M, Toth P (2006) A new ILP-based refinement heuristic for vehicle routing problems. Math Program 105(2–3):471–499
16. Gillett BE, Miller LR (1974) A heuristic algorithm for the vehicle dispatch problem. Oper Res 22:340–349
17. Glover F (1975) Improved linear integer programming formulations of nonlinear integer problems. Manage Sci 22:455–460
18. Gutin G, Yeo A, Zverovitch A (2007) Exponential neighborhoods and domination analysis for the TSP. In: Gutin G, Punnen AP (eds) The traveling salesman problem and its variations. Volume 12 of combinatorial optimization. Springer, Boston, pp 223–256
19. Hansen P, Maniezzo V, Voß S (2009) Special issue on mathematical contributions to metaheuristics editorial. J Heuristics 15(3):197–199
20. Hoskins M, Masson R, Melanon GG, Mendoza JE, Meyer C, Rousseau L-M (2014) The prepack optimization problem. In: Simonis H (ed) Integration of AI and OR techniques in constraint programming. Volume 8451 of lecture notes in computer science. Springer, Berlin/Heidelberg, pp 136–143
21. IBM ILOG (2014) CPLEX User's Manual
22. Jensen NO (1983) A note on wind generator interaction. Technical report, Riso-M-2411(EN), Riso National Laboratory, Roskilde
23. Lodi A, Martello S, Monaci M (2002) Two-dimensional packing problems: a survey. Eur J Oper Res 141:241–252
24. Maniezzo V, Stützle T, Voß S (eds) (2010) Matheuristics – hybridizing metaheuristics and mathematical programming. Volume 10 of annals of information systems. Springer, Boston
25. Martello S, Toth P (1990) Knapsack problems: algorithms and computer implementations. John Wiley & Sons, Chichester
26. Rego C, Glover F (2007) Local search and metaheuristics. In: Gutin G, Punnen A (eds) The traveling salesman problem and its variations. Volume 12 of combinatorial optimization. Springer, Boston, pp 309–368
27. Rothberg E (2007) An evolutionary algorithm for polishing mixed integer programming solutions. INFORMS J Comput 19:534–541
28. Sarvanov VI, Doroshko NN (1981) The approximate solution of the travelling salesman problem by a local algorithm with scanning neighborhoods of factorial cardinality in cubic time (in Russian). In: Software: algorithms and programs. Mathematical Institute of the Belorussian Academy of Sciences, Minsk, pp 11–13
29. Shaw P (1998) Using constraint programming and local search methods to solve vehicle routing problems. In: Maher M, Puget J-F (eds) Principles and practice of constraint programming CP98. Volume 1520 of lecture notes in computer science. Springer, Berlin/Heidelberg, pp 417–431

30. SIEMENS AG. SWT-2.3-93 Turbine, Technical Specifications. http://www.energy.siemens.com
31. Toth P, Vigo D (2002) An overview of vehicle routing problems. In: The vehicle routing problem. SIAM monographs on discrete mathematics and applications, Philadelphia
32. Xia Y, Yuan YX (2006) A new linearization method for quadratic assignment problem. Optim Methods Softw 21:803–816