Detecting Semantic Groups in MIP models

Domenico Salvagnin

IBM Italy and DEI, University of Padova, salvagni@dei.unipd.it

Abstract. Current state-of-the-art MIP technology lacks a powerful modeling language based on global constraints, a tool which has long been standard in constraint programming. In general, even basic semantic information about variables and constraints is hidden from the underlying solver. For example, in a network design model with unsplittable flows, both routing and arc capacity variables could be binary, and the solver would not be able to distinguish between the two semantically different groups of variables by looking at type alone. If available, such semantic partitioning could be used by different parts of the solver, heuristics in primis, to improve overall performance. In the present paper we will describe several heuristic procedures, all based on the concept of partition refinement, to automatically recover semantic variable (and constraint) groups from a flat MIP model. Computational experiments on a heterogeneous testbed of models, whose original higher-level partition is known a priori, show that one of the proposed methods is quite effective.

1 Introduction

Mixed-integer-programming (MIP) is a powerful paradigm to solve many combinatorial optimization problems coming from both theory and applications. Despite the admittedly limited set of constructs that are allowed in the paradigm, namely linear inequalities and integer constrained variables, it turns out that surprisingly many optimization problems of practical interest can be exactly, or approximately, formulated as MIP models [27]. At the same time, MIP solvers improved so much in the last decades that MIP is considered nowadays a mature technology. The seemingly limited language of MIP was indeed (partly) instrumental to its success: although powerful enough to model many optimization problems, it was at the same time easy enough to allow for a development of a rich and general theory, and for the definition of a standard file format (namely MPS) since the very beginning. By modeling an optimization problem as a MIP and solving it with a MIP solver, one takes advantage from decades of past (and future) developments in solver technology.

On the other hand, the limited language of MIP is actually a double-edged sword: modeling real-world problems as MIPs is far from obvious—a thing that seasoned MIP modelers tend to forget—and, more importantly, part of the global structure, which could be exploited by problem-specific approaches, is lost when translated into a flat MIP model. In general, even basic semantic information about variables and constraints is hidden from the underlying solver.

Let us consider for example the prepack optimization problem [23]. This problem arises 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. 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. Pre-packing items into box configurations has benefits in terms of easier and cheaper handling; on the other hand, it can reduce the flexibility of the supply chain, leading to situations in which the set of items that are actually shipped does not match exactly the demands—such deviations are usually penalized in the objective function. Using the notation in [15], a mixed-integer nonlinear model for the prepack optimization problem reads:

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

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

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

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

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

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

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

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

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

where I is the set of types of products, S the set of stores, $K \subset Z_+$ the set of available bin capacities, and B is the set of box configurations. Parameters r_{is} are the actual demands, while δ_{is} are upper bounds on the amount overstocking. As for variables, integer variables y_{bi} encode products' packing into boxes, while integer variables x_{bs} encode the shipping of box configurations to stores. Understocking and overstocking are expressed by decision variables u_{is} and o_{is}. Then, we have additional binary variables t_{bk} to map box configurations to bin capacities and additional integer variables $q_{bis} = x_{bs}\, y_{bi}$ used to count the number of items of type i sent to store s through boxes loaded with configuration b. Finally, the nonlinear products that define variables q_{bis} are actually formulated in a MIP framework by adding artificial binary variables (say v and w) that basically provide the binary expansion of variables x and y and the corresponding products. We refer to [15] for more details on the model. Although far from complex, model (1)-(9) is a typical example of the ingenuity needed to model a real-world problem as a MIP.

From a high-level point of view, model (1)-(9) is made of several different sets of variables that are semantically distinct: for example x variables encode shipping decisions, while y encode packing decisions. On the same line, v, and w are artificial binary variables that are needed for the sole purpose of being able to encode the constraints of the model as linear inequalities, and are also semantically different. However, this semantic grouping is completely lost and hidden from the MIP solver, that basically sees only a bunch of integer and binary variables. In other words, for the purpose of solving, model (1)-(9) gets flattened as an arbitrary general MIP model like:

$$\min\{c^T z : Az \leq b,\ z_j \in \mathbb{Z}\ \forall j \in J \subseteq \{1, \ldots, n\}\} \tag{10}$$

Semantic partitioning is not the only piece of information which is lost in the flattening process: the overall specific structure of the model (or part of it) is usually lost too, as well as the mapping between variables and elements of the sets of indices—actually, the index sets used during modeling are not even part of the model that is submitted to the solver. As a matter of fact, modern MIP solvers have a rich arsenal of algorithms that basically try to *reverse-engineer* combinatorial substructures from a flat model like (10). Unfortunately, while these procedures are usually cheap and effective, they are still heuristic in nature and can be fooled by the many transformations that are applied to a given MIP formulation in the preprocessing phase.

In this paper we are interested in general-purpose heuristic procedures to recover, or approximate, the semantic partitioning of variables (and constraints) present in the original high-level model from a flat one. The paper is organized as follows: in Section 2 we will overview existing literature on the subject and provide motivations for our study. In Section 3 we will present several different partitioning algorithms and discuss their respective strengths and weaknesses. In Section 4 we will present some computational results on a heterogenous testbed of models, showing that some methods are indeed quite successful in this reconstruction. Conclusions are finally drawn in Section 5.

2 Related Work

Current state-of-the-art MIP technology lacks a powerful modeling language based on global constraints, a tool which has long been standard in constraint programming [32]. For this reason, it has become standard practice in MIP implementations to devise algorithms that basically try to *reverse-engineer* combinatorial substructures from a flat list of linear inequalities. In [3], a procedure for detecting network structures was presented; such structure, when present, is then used to improve cutting plane separation. In [34], a procedure for detecting permutation problems, i.e., problems that optimize and arbitrary objective function over the set of all possible permutations of a given ground set, was introduced, with the purpose of devising a specialized primal heuristic for this class of problems. Similarly, MIP solvers often have simple heuristics to detect

whether the problem at hand admits a specialized solution algorithm—for example, a knapsack problem might be solved via dynamic programming—and switch to the latter according to some effort predictions.

In addition to algorithms that look for specific structures, like networks and permutations, modern MIP solvers also detect, usually during the preprocessing stage, general-purpose global structures that are used later in the process to improve the performance of the solver. Examples of global structures that are widely used include the clique table, the implication graph [36], and symmetries [28]. Those global structures are used to improve different parts of the solver, like domain propagation, cutting plane generation, and branching, see, e.g.,[1]. Recently, in [18], the clique table and the implication graph have been used to define neighborhoods for a LNS primal heuristic.

Semantic partitioning naturally belongs to the class of general-purpose global structures, like the clique table or the symmetry group of the formulation. Such piece of information, if available, could be used in many different components of a MIP solver. In particular:

- *branching*: branching rules could be biased in order to prefer branching on variables of the same class. This could help when other branching scores are flat.
- *aggregation*: if a variable from a given class can be aggregated out, chances are that all variables in the same class can be aggregated out—this happens, for example, in many time-indexed formulations for scheduling problems [8]. This would eliminate part of the guess-work in the aggregation heuristics.
- *relaxation*: a partitioning of constraints based on semantics could open the way to automatic Lagrangian relaxations, where constraints from the same class are relaxed into the objective function.
- *primal heuristics*: semantic groups can be used to devise neighborhoods in LNS approaches, for biasing fix-and-dive heuristics and to implement general-purpose metaheuristics. An example is the alternate heuristics: given two subsets of variables, it consists in alternately solving the problem with the variables of one of the subsets fixed, and this is quite effective for some classes of problems like pooling [6] and prepack optimization [15].

An alternative approach to the one studied here consists in extending the solver to accept a higher-level model in the first place. In such an enriched environment, the user is allowed to model the problem using expressions that compactly encode complex substructures, and the solver, which is fully aware of those expressions, can take advantage of specialized methods. This is the de-facto standard in constraint programming, where *global constraints* are used exactly for this purpose. A further generalization of the concept is *metaconstraints* [20,12]: a metaconstraint is syntactically specified much as a global constraint is, but it is also amended with additional annotations that specify how it is to be relaxed, how it is supposed to do constraint propagation and to direct search (via branching) in case it is violated. Metaconstraints, pioneered in the modeling system SCIL [5], are partially supported in recent versions of high-level modeling

systems, like AMPL [17], ECLiPSE [30], MiniZinc [37] and SIMPL [40]: however they are fully exploited only if the underlying solver provides some native support for them, which is currently not the case for most MIP solvers. A notable exception is the open-source solver SCIP [2], whose *constraint handlers* are basically metaconstraints implemented at the C level.

While exploiting, as opposed to reverse-engineering, higher-level knowledge has clear benefits and should be the winning approach in the long run, we have to face the fact that currently most state-of-the-art MIP solvers accept only very limited extensions w.r.t. the regular MIP language, namely indicator constraints [25] and piece-wise linear functions [19]. In addition, a modeling language based on metaconstraints is not without its share of problems: as the solver automatically translates global constraints into a MIP model, it often creates auxiliary variables. Variables introduced by different metaconstraints might actually have the same meaning, but without some form of additional annotations, like the *semantic typing* proposed in [12], the solver is unable to recognize such relationships and produce a tight model, equivalent to what a human modeler might produce by hand. The issue of modeling with metaconstraints versus reverse-engineering them from a flat model is also discussed, among others, in [31,5,21,22].

Finally, there are connections between the subject of the current paper and symmetry detection [28]: intuitively, if two variables are symmetric, they also belong to the same semantic class, but the converse is not true. As such, semantic grouping generalizes the orbit partitioning that is obtained as a side product of symmetry detection, and it applies to a much wider range of problems, albeit with a completely different usage.

3 Detection Algorithms

Recovering the high-level partition of variables (and constraints) is inherently an ill-posed problem, as MIP solvers cannot truly have a notion of semantics. As such, we need to replace the notion of belonging to the same semantic class with something that is within the reach of the solver and can be inferred from the model alone. In the present paper, we propose an overall approach based on partition refinement, a basic tool in computer science. According to [4], partition refinement is defined as follows: given a set S, an initial partition π of S into pairwise disjoint blocks (also called *cells*) $\{B_1, \ldots, B_p\}$, and a function f on S[1], the task is to find the coarsest partition of S, say $\pi' = \{E_1, \ldots, E_q\}$, such that:

1. π' is consistent with π, that is each E_i is a subset of some B_j;
2. π' is compatible with f, which means that a and b in E_i imply $f(a)$ and $f(b)$ are in some E_j.

Partition refinement can be implemented in $O(n \log n)$ time, for arbitrary π and f, where $n = |S|$.

[1] In applications, such as graph automorphism and DFA minimization, function f is extended to dependent on a more complex domain than just S.

In our context, the partition we are interested in is obviously the one of variables and constraints in the model. The final outcome will depend on two choices, namely the initial partition π and function f. As for the initial partition, we can split variables according to their type and, optionally, according to whether they appear in the objective function or not. For constraints, we can start from their initial classification. Constraint classification [2,16] is a technique used to achieve faster constraint propagation: for example, variable bounds, or set covering constraints, can be propagated way more efficiently than an arbitrary linear constraint, and MIP solvers usually implement some form of classification in order to take advantage of that. More details about the constraint classes used in this paper are given in Section 4. As for function f, it must necessarily take into account how variables and constraints are structurally connected in model: a convenient tool is to encode the connections we are interested in a graph, and then define f accordingly—in this case function f usually encodes some form of *vertex invariant* in the graph. This is the approach taken, for example, in symmetry detection codes [29,13,14], that work by implicitly constructing an auxiliary graph and computing its automorphism group. Incidentally, partition refinement is a crucial building block of all graph automorphism packages. In those algorithms, function f is the so called *connection function*: given a vertex v and a set of vertices B, $f(v, B)$ gives the number of elements in B which are connected to v. In other words, each refinement step will pick a *target cell* B_i, an *inducing cell* B_j, and it will split the vertices in B_i according to their connection count w.r.t. B_j. Note that there is no actual choice of B: each time a cell is split, its pieces will act in turn as inducing cells, and the whole process is iterated until a fixed point is reached. An example of partition refinement according to the connection function is depicted in Figure 1.

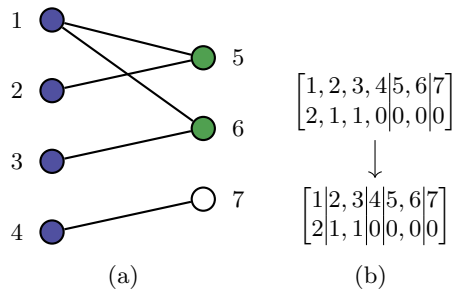

Fig. 1. Example of partition refinement according to connection function. The first row of each matrix on the right encodes the current partition, while the second gives the connection function w.r.t. cell $[5, 6]$.

In the rest of this section, we will describe several partitioning schemes, that can all be cast into the refinement framework just described.

3.1 Simple Refinements

Some simple strategies rely entirely on carefully constructing an initial partition π, and take function f as the identity. In this case, it trivially holds that $\pi' = \pi$. The two strategies that we tried in this class are:

- `type`: partition variables by type alone, and constraints according to their initial classification.
- `histogram`: partition variables and constraints according to their so called *histogram*. For variables, this amounts to counting the number of constraints in which each variable appears, and partition based on this count—analogously for constraints.

It is quite obvious that method `type` will not be powerful enough in most cases, as it is often not possible to distinguish semantically different variables by type alone—model (1)-(9) is an example, as is network design with unsplittable flows. However, it is convenient to have the method as a baseline for benchmarking.

3.2 Iterative Refinements

The strategies in this class implement the full-blown partition refinement algoritm. They all start from the same initial partitioning described at the beginning of the section, but use different connection functions. The three strategies that we tried in this class are:

- `fast`. We construct a bipartite graph $G = (V, K, E)$, where V is the set of variables, K is the set of constraints classes, and there is edge (v_i, k_j) in the graph if and only if variable v_i appears in at least one constraint of class k_j. The connection function f is the regular connection function used in graph automorphism packages.
- `recursive`. We construct a bipartite graph $H = (V, C, E)$, where V is the set of variables, C is the set of constraints, and there is edge (v_i, c_j) in the graph if and only if variable v_i appears in constraint c_j. The connection function f is a modified version of the regular connection function, that ignores actual counts when deciding how to split a cell. In other words, a target cell B_i is split only distinguishing between its elements that connect to the inducing cell B_j from those that do not, without further refinement based on the actual counts. On the example in Figure 1, refining cell $[1, 2, 3, 4]$ w.r.t. $[5, 6]$ would thus yield $[1, 2, 3|4]$ instead of $[1|2, 3|4]$.
- `auto`. We construct the same bipartite graph $H = (V, C, E)$ as in `recursive`, but use the regular connection function. Note that H is the very same bipartite graph that would be constructed to compute symmetries in a binary model [33], and `auto` then just performs the initial refinement step without doing the enumeration required to properly compute the set of generators.

It is worth noting that `fast` could have been equivalently defined as using the same connection function as `recursive`: given that the graph is bipartite,

and one set of vertices, namely K, is already partitioned into blocks of size 1, no refinement can happen on K, and the connection count of a vertex $v \in V$ with a cell of K is always at most 1. As a result, `fast` can also be equivalently implemented with a specialized algorithm that just computes for each variable the subset of constraint classes it appears in and then just splitting V according to this piece of information, with a bucket-sort like procedure. Note also that `fast` does not produce a partitioning for constraints, although such a partition can be computed a-posteriori, with a second bucket sort in which the roles of constraints and variables are reversed.

The iterative refinement used in these methods is needed to be able to distinguish variables depending of the class of variables they connect to and not just basing on the kind of constraints they participate in: for example, in the prepack model, we have two sets of binary variables, namely w and v, that encode a binary expansion of two semantically different sets of variables (x and y respectively). Without further information, a method like `fast` would not be able to distinguish w from v, as those variables appear always in constraints of the same kind. At the same time, this behaviour can be an overkill and lead to an artificial split of variables. Consider, for example, a flow formulation on a layered graph[2]: the flow variables associated to the first (and/or last) layer are usually connected to some other variables in the model, while those associated with inner layers are not. Iterative refinement will not only distinguish outer layers from inner layers, but also recursively partition flow variables by layer, ending up with a semantic class for each and every layer.

A common characteristic of all methods is that they completely ignore the actual values of the coefficients in the model, but rather distinguish only between zero and non-zero values. This is in stark contrast with the symmetry detection case: actual values are needed to compute proper symmetries, but they are completely unsuited for semantic grouping, as variable semantics are independent of numerics. Similarly, actual connection counts are ignored for all methods but `auto` and `histogram`, although the situation is not as obvious as for values: in some cases connection counts could indeed help, but they would make the partitioning process too sensitive to trivial changes. For example, fixing a variable—and getting rid of it during presolve—would create an unwanted distinction between the variables that were connected to it and those that were not, while ideally it should be a neutral chance in most circumstances. Similarly, connection counts can be misleading when zero is a legitimate value for a parameter. For example, on instance `markshare_5_0` from MIPLIB2010, which is basically a multidimensional subset sum problem, the constraint matrix is randomly generated and is *almost* fully dense: using exact counts would split the (very few) variables whose columns do not cover all rows from those who do.

Finally, all methods come in two variants: one in which the initial partition takes into account the objective function, by distinguishing whether a variable appears in it or not, and one in which no such distinction is made. It is not obvi-

[2] The same argument applies to the more common case of a general graph, but we will consider the layered case for simplicity.

ous which one is more suited for the job: in some cases, like balanced subgraph problems [24], the objective function is the only way to distinguish between vertex variables and edge variables, given the current constraint classification. On the other hand, as with the constraint matrix, zero might be a (rare but) legitimate value for some of the parameters of the model, say a cost, hence the resulting distinction would be artificial.

4 Computational Results

We implemented our code in C++, using IBM ILOG CPLEX 12.6.2 [25] as MPS reader. All tests have been performed on a PC with an Intel Core i5 CPU running at 2.66GHz, with 8GB of RAM. We collected a heterogeneous set of instances—most included in MIPLIB2010 [26]—whose high-level structure was either known or easily recoverable (by the author) from variables' and constraints' names, and used that as our testbed. As for constraint classification, we considered the following classes:

- *set covering*: inequality of the form $\sum_j x_j \geq 1$, involving binary variables only (possibly complemented).
- *set partitioning*: equality of the form $\sum_j x_j = 1$, involving binary variables only (possibly complemented).
- *set packing*: inequality of the form $\sum_j x_j \leq 1$, involving binary variables only (possibly complemented).
- *cardinality*: inequality of the form $\sum_j x_j \leq K$ or $\sum_j x_j \geq K$, involving binary variables only (possibly complemented).
- *cardinality equation*: equality of the form $\sum_j x_j = K$, involving binary variables only (possibly complemented).
- *variable bounds*: inequality of the form $ax \leq by$ or $ax \geq by$, with y binary.
- *mixed*: any inequality that does not fall in any of the classes above.
- *mixed equality*: any equality that does not fall in any of the classes above.

This classification is pretty basic, yet it can distinguish most of the constraint classes used in practice, and can be implemented (and executed) very efficiently with a single pass through the constraint matrix.

We tested all the methods described in the previous section, both in their *objective* and *no-objective* variants: detailed results about the *objective* variant are available in Table 1. In the table, we provide basic statistics about each instance (namely, number of rows m and columns n), the known number of variable semantic groups (column g), and the number of groups identified by the proposed methods.

Interpreting the results of the table is not straightforward, as a method may have incorrectly partitioned the set of variables even if it got the number of blocks right, so a deeper analysis of the outcome of the algorithms is needed. Here are some preliminary conclusions drawn from the synthetic numbers in the table plus a detailed analysis of the outcome of the algorithms on the individual instances. No method is always recovering the original semantic partitioning,

confirming that such a reverse-engineering is not a trivial task. In addition, it is pretty clear that all methods that rely on exact counts, namely `histogram` and `auto`, perform quite poorly, splitting the set of variables in way too many blocks. The phenomenon is quite clear, for example, on `seymour`, which is a pure set covering model, with all variables belonging to the same class. Surprisingly, also `recursive`, which does *not* keep exact counts, often splits the blocks too finely, in particular for scheduling models. The issue there (and on the multi-activity from [35]) is exactly the one described in the previous section concerning flow models.

Overall, `fast` qualifies as the best method: it is able to refine over the initial partitioning (`type`) when needed, and it never returns too fine a partition, achieving a reasonable approximation of the true number of blocks. The method is still not perfect though: for example, on the prepack model it cannot distinguish understocking from overstocking variables (and similarly for the multi-activity scheduling instance), and on the classification model it cannot distinguish the coefficients of the separating hyperplane from its right hand side–although whether the two are actually semantically different is debatable.

As for running times, all methods except `recursive` and `auto` always execute in a fraction of a second, while the other two can be relatively expensive, up to a few seconds. In any case, all methods required a negligible time w.r.t. the time that is needed to solve the instances, so that detection runtime is never an issue, and we are actually free to choose the method to apply based on success rather than on time.

Concerning the *no-objective* variant, the results are mixed: ignoring the objective function is not enough to fix the intrinsic weaknesses of `recursive` and `histogram`, and it only marginally affects `fast`, which is often able to infer the same partitioning with and without objective. On the one hand, ignoring the objective fixes the behaviour of `fast` on the facility location problems; on the other hand, it causes missing refinements on 5 other instances. Overall, taking the objective function into accounts seems slightly superior.

5 Conclusions

We described a family of procedures, all based on partition refinement, to heuristically recover the semantic grouping of variables from a MIP model. The problem is inherently ill-posed, given the lack of a truly semantic notion within MIP solvers, but partition refinement seems to capture decently well the concept of structurally equivalent variables, which is a proxy for semantic equivalence. Indeed, one of the proposed methods is quite successful at recovering the high-level variable partitioning of the model on an heterogeneous testbed of problems.

Still, no method is perfect, and for each example that supports a design decision, like whether to ignore the objective function or to iteratively refine the partition, there is a counterexample supporting the very opposite, confirming the fact that once semantic information is lost, it is quite difficult to recover in a robust way, if at all. As such, the experiments in the present paper are

Table 1. Number of variable groups found by *objective* variant of partition refinement methods.

	size			methods				
instance	n	m	g	`type`	`fast`	`recursive`	`histogram`	`auto`
ash608gpia-3col [26]	3,651	24,748	2	2	2	2	18	1,217
csched007 [39]	1,758	351	4	3	4	1,758	4	1,758
dfn-gwin-UUM [41]	938	158	3	3	3	8	3	26
n3700 [38]	10,000	5,150	2	2	2	2	2	2
reblock166 [9]	1,660	17,024	1	1	1	1	28	1,660
seymour [26]	1,372	4,944	1	1	1	1	126	1,155
toll-like [24]	2,883	4,408	2	2	2	2	31	2,163
triptim1 [10]	30,005	15,076	16	4	13	30,055	55	30,055
uc-case3 [26]	37,749	52,003	7	3	6	12,898	190	14,410
wachplan [26]	3,361	1,553	5	2	4	617	50	673
prepack [15]	84,376	197,154	8	4	8	12	8	12
multiactsched [35]	11,180	8,222	6	4	6	975	11	5590
classification[11]	204	100	5	3	4	4	4	4
zib54-UUE-SAN[41]	240,240	81,134	2	2	2	2	2	150
fac.location[7]	90,300	90,601	2	3	3	7	3	7

yet another piece of evidence that we should not dismiss so easily high-level knowledge about the optimization models that we want to solve.

References

1. Achterberg, T.: Constraint Integer Programming. Ph.D. thesis, Technische Universität Berlin (2007)
2. Achterberg, T.: SCIP: solving constraint integer programs. Mathematical Programming Computation 1(1), 1–41 (2009)
3. Achterberg, T., Raack, C.: The MCF-separator: detecting and exploiting multi-commodity flow structures in MIPs. Mathematical Programming Computation 2(2), 125–165 (2010)
4. Aho, A., Hopcroft, J., Ullman, J.D.: The Design and Analysis of Computer Algorithms. Addison-Wesley, Reading, MA (1974)
5. Althaus, E., Bockmayr, A., Elf, M., Jünger, M., Kasper, T., Mehlhorn, K.: SCIL - symbolic constraints in integer linear programming. In: Möhring, R.H., Raman, R. (eds.) Algorithms - ESA 2002, 10th Annual European Symposium, Rome, Italy, September 17-21, 2002, Proceedings. Lecture Notes in Computer Science, vol. 2461, pp. 75–87. Springer (2002)
6. Audet, C., Brimberg, J., Hansen, P., Digabel, S.L., Mladenovic, N.: Pooling problem: Alternate formulations and solution methods. Management Science 50(6), 761–776 (2004)
7. Avella, P., Boccia, M.: A cutting plane algorithm for the capacitated facility location problem. Computational Optimization and Applications 43(1), 39–65 (2009)
8. Baker, K.R., Trietsch, D.: Principles of Sequencing and Scheduling. Wiley (2009)

9. Bley, A., Boland, N., Fricke, C., Froyland, G.: A strengthened formulation and cutting planes for the open pit mine production scheduling problem. Computers and Operations Research 37, 1641–1647 (2010)
10. Borndörfer, R., Liebchen, C.: When Periodic Timetables are Suboptimal. In: Kalcsics, J., Nickel, S. (eds.) Operations Research Proceedings 2007. pp. 449–454. Springer (2008)
11. Brooks, J.P.: Support vector machines with the ramp loss and the hard margin loss. Operations Research 59(2), 467–479 (2011)
12. Cire, A., Hooker, J.N., Yunes, T.: Modeling with metaconstraints and semantic typing of variables. INFORMS Journal on Computing (to appear)
13. Darga, P.T., Liffiton, M.H., Sakallah, K.A., Markov, I.L.: Exploiting structure in symmetry detection for CNF. In: Proceedings of the 41th Design Automation Conference, DAC 2004, San Diego, CA, USA, June 7-11, 2004. pp. 530–534 (2004)
14. Darga, P.T., Sakallah, K.A., Markov, I.L.: Faster symmetry discovery using sparsity of symmetries. In: Proceedings of the 45th Design Automation Conference, DAC 2008, Anaheim, CA, USA, June 8-13, 2008. pp. 149–154 (2008)
15. Fischetti, M., Monaci, M., Salvagnin, D.: Mixed-integer linear programming heuristics for the prepack optimization problem. Discrete Optimization ((to appear))
16. Fischetti, M., Salvagnin, D.: Feasibility pump 2.0. Mathematical Programming Computation 1(2–3), 201–222 (2009)
17. Fourer, R., Gay, D.M., Kernighan, B.W.: AMPL: A modeling language for mathematical programming. Thomson (2003)
18. Gamrath, G., Berthold, T., Heinz, S., Winkler, M.: Structure-based primal heuristics for mixed integer programming. In: Fujisawa, K., Shinano, Y., Waki, H. (eds.) Optimization in the Real World, Mathematics for Industry, vol. 13, pp. 37–53. Springer Japan (2016)
19. GUROBI: GUROBI 6.0 User's Manual (2015)
20. Hooker, J.N.: Integrated Methods for Optimization. Springer (2006)
21. Hooker, J.N.: Logic-based modeling. In: Appa, G., Pitsoulis, M., Leonidas, S., Williams, H.P. (eds.) Handbook on Modelling for Discrete Optimization. pp. 61–102 (2006)
22. Hooker, J.N.: Hybrid modeling. In: van Hentenryck, P., Milano, M. (eds.) Hybrid Optimization: The Ten Years of CPAIOR. pp. 11–62 (2011)
23. Hoskins, M., Masson, R., Melanon, G., Mendoza, J., Meyer, C., Rousseau, L.M.: The PrePack Optimization Problem. In: Simonis, H. (ed.) Integration of AI and OR Techniques in Constraint Programming, Lecture Notes in Computer Science, vol. 8451, pp. 136–143. Springer Berlin / Heidelberg (2014)
24. Hüffner, F., Betzler, N., Niedermeier, R.: Separator-based data reduction for signed graph balancing. Journal of Combinatorial Optimization 20, 335–360 (2010)
25. IBM ILOG: CPLEX 12.6.2 User's Manual (2015)
26. Koch, T., Achterberg, T., Andersen, E., Bastert, O., Berthold, T., Bixby, R.E., Danna, E., Gamrath, G., Gleixner, A.M., Heinz, S., Lodi, A., Mittelmann, H., Ralphs, T., Salvagnin, D., Steffy, D.E., Wolter, K.: MIPLIB 2010 - Mixed Integer Programming Library version 5. Mathematical Programming Computation 3, 103–163 (2011)
27. Lodi, A.: Mixed integer programming computation. In: Jünger, M., Liebling, T.M., Naddef, D., Nemhauser, G.L., Pulleyblank, W.R., Reinelt, G., Rinaldi, G., Wolsey, L.A. (eds.) 50 Years of Integer Programming 1958-2008 - From the Early Years to the State-of-the-Art, pp. 619–645. Springer (2010)
28. Margot, F.: Symmetry in integer linear programming. In: 50 Years of Integer Programming. Springer (2009)

29. McKay, B.D.: Practical graph isomorphism (1981)
30. Milano, M.: Constraint and Integer Programming: Toward a Unified Methodology. Kluwer Academic Publishers (2003)
31. Mitra, G., Lucas, C., Moody, S., Hadjiconstantinou, E.: Tools for reformulating logical forms into zero-one mixed integer programs. European Journal of Operational Research 72(2), 262–276 (1994)
32. Rossi, F., van Beek, P., Walsh, T. (eds.): Handbook of Constraint Programming. Foundations of Artificial Intelligence, Elsevier (2006)
33. Salvagnin, D.: A dominance procedure for Integer Programming. Master's thesis, University of Padova (2005)
34. Salvagnin, D.: Detecting and exploiting permutation structures in MIPs. In: CPAIOR. pp. 29–44 (2014)
35. Salvagnin, D., Walsh, T.: A hybrid mip/cp approach for multi-activity shift scheduling. In: CP. pp. 633–646 (2012)
36. Savelsbergh, M.W.P.: Preprocessing and probing for mixed integer programming problems. ORSA Journal on Computing 6, 445–454 (1994)
37. Stuckey, P.J., Tack, G.: Minizinc with functions. In: Gomes, C.P., Sellmann, M. (eds.) Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems, 10th International Conference, CPAIOR 2013, Yorktown Heights, NY, USA, May 18-22, 2013. Proceedings. Lecture Notes in Computer Science, vol. 7874, pp. 268–283. Springer (2013)
38. Sun, M., Aronson, J.E., McKeown, P.G., Drinka, D.A.: A tabu search heuristic procedure for the fixed charge transportation problem. European Journal of Operational Research 106, 441–456 (1998)
39. Yunes, T.: CuSPLIB 1.0: A library of single-machine cumulative scheduling problems (2009), http://moya.bus.miami.edu/ tallys/cusplib/
40. Yunes, T., Aron, I.D., Hooker, J.N.: An integrated solver for optimization problems. Operations Research 58(2), 342–356 (2010)
41. SNDlib (2006), http://sndlib.zib.de