Satisfiability Modulo Theories for Process Systems Engineering

Miten Mistry, Andrea Callia D'Iddio, Michael Huth, Ruth Misener[*]

Department of Computing; Imperial College London; South Kensington SW7 2AZ; UK

Abstract

Process systems engineers have long recognized the importance of both logic and optimization for automated decision-making. But modern challenges in process systems engineering could strongly benefit from methodological contributions in computer science. In particular, we propose *satisfiability modulo theories* (SMT) for process systems engineering applications. We motivate SMT using a series of test beds and show the applicability of SMT algorithms and implementations on (i) two-dimensional bin packing, (ii) model explainers, and (iii) MINLP solvers.

Keywords: satisfiability modulo theories, mixed-integer optimization, generalized disjunctive programming, mixed logical-linear programming

1. Introduction

Process systems engineers have long recognized the importance of both logic and optimization for automated decision-making (Jain and Grossmann 2001, Hooker and Ottoson 2003, Maravelias and Sung 2009, Trespalacios and Grossmann 2014). Early work on disjunctive programming is motivated by (i) the practical need to naturally model logical conditions such as dichotomies and implications and (ii) the theoretical insights gained from novel structural characterizations (Balas 1979). Contributions highlighting the importance of both logic and optimization have diverse applications, e.g. spatial layout (Sawaya and Grossmann 2005), modeling contracts in supply chain optimization (Park et al. 2006, Rodriguez and Vecchietti 2009), and manufacturing systems (Fattahi et al. 2014).

While process systems engineers have been developing methods at the interface of logic and optimization, the computer science community has also been developing hybrid logic/optimization approaches. Typical computer science applications requiring both logic and optimization are operating system scheduling and motion planning in robotics (Aminof et al. 2011, Raman et al.

[*]`r.misener@imperial.ac.uk`; Tel: +44 (0) 20759 48315

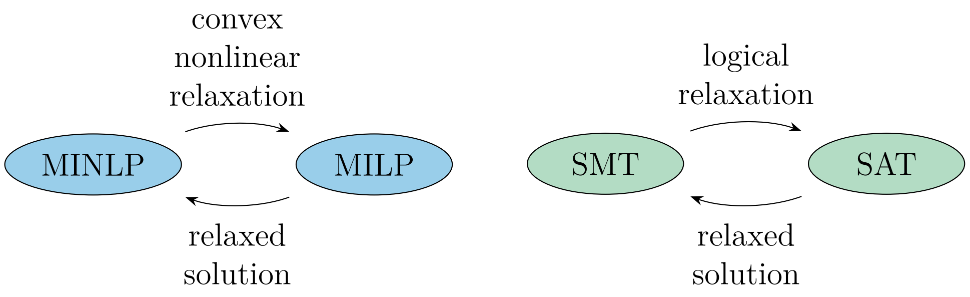


Figure 1: Mixed-integer nonlinear optimization problems (MINLP) may be solved as a series of mixed integer linear optimization problems (MILP). Satisfiability modulo theories problems (SMT) may be solved as a series of propositional satisfiability problems (SAT).

2013, Beaumont et al. 2015). The differences in application domains between process systems engineering and computer science have unfortunately led to a divergence in mathematical developments.

But modern challenges in process systems engineering could strongly benefit from methodological contributions in computer science. In particular, we propose *satisfiability modulo theories* (SMT) for process systems engineering applications. We motivate SMT using a series of test beds and show the applicability of SMT algorithms and implementations.

Section 2 reviews background in both optimization and logic. Section 3 discusses existing optimization/logic hybrids. Section 4 describes three domains where SMT is highly applicable to PSE: (§4.1) two-dimensional bin packing, (§4.2) model explainers, and (§4.3) SMT-based MINLP solvers. Table 1 summarizes our position that SMT has complementary strengths and weaknesses with respect to mixed-integer nonlinear optimization (MINLP). We propose SMT as a methodology to address several challenges in process systems engineering. We also provide a *needs analysis* identifying the SMT development required by the process systems engineering community.

Parts of this paper have been previously published, e.g. Callia D'Iddio and Huth (2017) describe the Section 4.3 `ManyOpt` tool in detail. The purpose and novelty of this paper is to show the broad applicability of SMT to PSE.

2. Definitions & Background

Section 2.1 fixes variable notation. Sections 2.2 – 2.4 review mixed-integer nonlinear optimization (MINLP), propositional satisfiability (SAT), and satisfiability modulo theories (SMT), respectively. Figure 1 diagrams how solving MINLP problems as a series of mixed integer linear optimization problems is analogous to solving SMT problems as a series of SAT problems.

Table 1: Complementary strengths in SMT/MINLP inspire applying SMT to PSE

	SMT	**MINLP**	
Traditional Community	Computer science	Engineering	Division in developments stemming from divergent applications
Deductive Reasoning	**Strong**	**Limited**	GDP less flexible than SMT
Nonlinear Functions	**Limited**	**Strong**	Transcendental functions in MINLP
Optimizing Objective	**Weak**	**Strong**	Tightly integrated in MINLP
Propositional Satisfiability	**Strong**	**Weak**	Logical propositions not typical in MINLP
Warm Starting	**Strong**	**Weak**	Would strongly benefit MINLP
Scalability	**Limited**	**Limited**	SMT : limited for nonlinear functions MINLP: limited for large problems[†]
Typical Applications	**SMT :** Software verification; Scheduling **MINLP:** Energy systems design; Biomedical engineering		

[†] For some problems, MINLP can reliably address 10^3 variables/constraints; but 10^2 variables/constraints are more typical for general problem classes

2.1. Notation for Logic & Optimization

Logic and optimization requires three variable types: (i) continuous variables $\boldsymbol{x} \in \mathbb{R}^{n_C}$, (ii) integer variables $\boldsymbol{y} \in \mathbb{Z}^{n_I}$, and (iii) Boolean (propositional) variables $\boldsymbol{Y} \in \{\mathit{True}, \mathit{False}\}^{n_B}$. Propositional connectives are: $\wedge$ (and), $\vee$ (or), $\neg$ (not), $\rightarrow$ (if…then) and $\leftrightarrow$ (if and only if). The shorthand $\underset{i \in I}{\veebar} Y_i$ represents that exactly one of the propositional variables Y_i is true.

2.2. Mixed-Integer Nonlinear Optimization

Traditional mathematical optimization formulations only incorporate continuous and integer variables. MINLP is defined as (Boukouvala et al. 2016):

$$\begin{aligned} \min_{\boldsymbol{x}} \quad & f_0(\boldsymbol{x}) \\ \text{s.t.} \quad & b_i^{\text{LO}} \leq f_i(\boldsymbol{x}) \leq b_i^{\text{UP}} \quad \forall\, i \in \mathcal{M} := \{1, \ldots, M\} \\ & x_j^{\text{LO}} \leq \quad x_j \leq x_j^{\text{UP}} \quad \forall\, j \in \mathcal{N} := \{1, \ldots, N\} \\ & x_j \in \mathbb{Z} \qquad\qquad\quad \forall\, j \in \mathcal{I} \subseteq \mathcal{N} \end{aligned} \tag{MINLP}$$

where $\mathcal{M}$, $\mathcal{N}$, and $\mathcal{I}$ represent sets of constraints, variables, and discrete variables, respectively. The objective and constraints are functions $f_i : \mathbb{R}^N \mapsto \mathbb{R} \ \forall \ i \in \{0, \ldots, M\}$. Parameters $b_i^{\mathrm{LO}} \in \mathbb{R} \cup \{-\infty\}$ and $b_i^{\mathrm{UP}} \in \mathbb{R} \cup \{+\infty\}$ bound the set of constraints $\mathcal{M}$; parameters $x_j^{\mathrm{LO}} \in \mathbb{R} \cup \{-\infty\}$ and $x_j^{\mathrm{UP}} \in \mathbb{R} \cup \{+\infty\}$ bound the set of variables $\mathcal{N}$. We assume that it is possible to infer finite bounds on the variables $\boldsymbol{x}$ and that the image of f_i is finite on $\boldsymbol{x}$.

2.3. Propositional Satisfiability

Traditional propositional satisfiability only incorporates Boolean variables. SAT is defined:

> Given a propositional formula φ built from the variables Y_i, is there a truth assignment that satisfies φ?

There exist efficient, satisfiability-preserving transformations from any propositional formula φ to conjunctive normal form (CNF), so SAT solvers typically assume that φ is written in CNF. In CNF: literals, i.e. propositional variables Y_i or their negation $\neg Y_i$, form clauses P_j, i.e. disjunctions ($\vee$) of literals. The final propositional formula $\varphi = \bigwedge\limits_{j=1}^{m} P_j$ is a conjunction of clauses.

SAT applications include: planning (Kautz and Selman 1992), model checking (Biere et al. 1999) and scheduling (Zhang 2002). Although SAT is $\mathcal{NP}$-complete (Cook 1971) and the worst-case complexity is exponential, modern SAT solvers can handle problems with hundreds of thousands of variables (Malik and Zhang 2009).

Most SAT solvers use the Davis-Putnam-Logemann-Loveland (DPLL) search algorithm (Davis and Putnam 1960, Davis et al. 1962). DPLL fixes variable Y_i assignments, i.e. truth assignments, using a tree-based branching approach. DPLL propagates truth assignments to all clauses P_j. Propagating truth values may allow DPLL to assign further variables a truth value. If DPLL finds that a partial assignment is *unsatisfiable*, i.e. cannot satisfy φ, then the algorithm backtracks and assigns a different value to one of the variables. DPLL continues until it either: (i) finds a combination of truth values for Y_i satisfying φ or (ii) proves the formula φ is unsatisfiable. DPLL also has functionality supporting warm starts.

SAT solving techniques include (Biere et al. 2009): (i) Boolean constraint propagation, where the current fixed variable set implies variable assignments, (ii) resolution, where sets of clauses derive additional clauses, (iii) and conflict driven clause learning, where an unsatisfiable result derives extra clauses pruning the search tree (Davis and Putnam 1960, Davis et al. 1962, Silva and Sakallah 1996). SAT solving methods are highly applicable to optimization (Hooker and Osorio 1999, Achterberg 2007a).

2.4. Satisfiability Modulo Theories

SMT incorporates continuous, integer, and Boolean variables to assess constraint set satisfiability by separating truth value assignment from the correctness reasoning with respect to a *theory*. SMT consists of: (i) a SAT solver and (ii) a theory solver for a theory of our choice (de Moura and Bjørner 2008a). The SMT approach to constraint satisfaction uses powerful SAT solving to derive a, potentially smaller, set of constraints to assess theory satisfiability. A background theory is a set of axioms and symbols, e.g. the theory of arithmetic. An SMT solver consists of a SAT solver and a theory solver. The idea is to leverage the strength and robustness of modern SAT solvers to search for a feasible solution. The modeling framework exposed by SMT allows for Boolean variables to be used with background theory variables, e.g $Y \rightarrow (x \geq 0)$ where x is continuous and Y is Boolean, so SMT is a natural choice when logical decisions form a part of the modeled system.

SMT research dates back to the 1970s with early work on decision procedures (Nelson and Oppen 1979, 1980, Shostak 1979, 1982). Available SMT theories include: Equality with Uninterpreted functions ($\mathcal{EUF}$), linear arithmetic $\mathcal{LA}$, and arrays $\mathcal{AR}$ (Biere et al. 1999). Most SMT solvers, e.g. `Z3`, can also handle nonlinear arithmetic, i.e. polynomial functions. DPLL(T) generalizes DPLL (Ganzinger et al. 2004). SMT is primarily applied in program verification and formal methods, but it also has scheduling and planning applications (Bjørner and De Moura 2011). SMT provides a (provable) guarantee of feasibility/infeasibility. This is the different from the idea of a *feasibility pump* that uses heuristics to (hopefully) generate a feasible solution, e.g. in D'Ambrosio et al. (2012).

SMT assesses the satisfiability of a model and, if the model is satisfiable, the SMT solver returns a witness. If the model is unsatisfiable the SMT solver can return an unsatisfiable core, a mutually unsatisfiable subset of model constraints. An unsatisfiable core is a useful tool when addressing why a model does not behave how we expect or to understand why a model fails. Commonly used SMT solvers include Z3 (de Moura and Bjørner 2008a) and MathSAT (Cimatti et al. 2013).

Example 1. Suppose that we wish to satisfy Eq. (1). Equation (1) combines SAT and the theory of real arithmetic.

$$(x_1 \leq 1) \wedge (x_2 \leq 2) \wedge ((x_1 \geq 5) \vee (x_3 \leq 3)) \wedge (x_1 + x_2 + x_3 \geq 10) \tag{1}$$

For Eq. (1), SMT leverages a SAT solver by replacing each inequality with auxiliary propositional variables, e.g. $Y_1 = (x_1 \leq 1)$, and assessing propositional satisfiability of the resulting formula:

$$Y_1 \wedge Y_2 \wedge (Y_3 \vee Y_4) \wedge Y_5. \tag{2}$$

The SAT solver returns an assignment satisfying Eq. (2), e.g. $Y_i = True,\ \forall\, i$. Then, the real arithmetic theory solver checks the propositional variable meaning. Here, the theory solver deduces that the assignment is incorrect because we cannot have both $Y_1 = (x_1 \leq 1) = True$ and $Y_3 = (x_1 \geq 5) = True$. The theory solver encodes additional propositional clauses, e.g. $(\neg Y_1 \vee \neg Y_3)$, augments Eq. (2), and passes Eq. (3) to the SAT solver:

$$Y_1 \wedge Y_2 \wedge (Y_3 \vee Y_4) \wedge Y_5 \wedge (\neg Y_1 \vee \neg Y_3). \tag{3}$$

SMT iterates between the SAT and theory solvers until the algorithm terminates, in this case with the point $x_1 = 5, x_2 = 2, x_3 = 3$. □

Example 1 suggests that the SAT and theory solvers are disjoint, but the most efficient and stable SMT tools integrate the two components (Sebastiani 2007). Interaction between the theory solver and partial SAT solutions allow the theory solver to identify unsatisfiability in a partial assignment.

The efficacy of an SMT solver depends on the quality of the theory solver since a propositional encoding has to be created for the SAT solver. If the encoding is weak and the theory solver cannot strengthen it effectively, the SMT solver will, in worst case, enumerate all propositional solutions.

3. Logic/Optimization Hybrids

This section reviews mathematical modeling approaches combining optimization and logic: disjunctive programming (§3.1), generalized disjunctive programming (§3.2), and mixed logical-linear programming (§3.3). We also discuss prior work on optimization methods using SMT (§3.4) and logic-based Benders decomposition, a commonly-used solution protocol (§3.5).

3.1. Disjunctive Programming

Balas developed disjunctive programming in the 1970s (Balas 1974, 1975, 1977, 1979). A disjunctive program is given by a linear objective and the disjunction of systems of linear constraints:

$$\begin{aligned}
\min_{\boldsymbol{x}} \quad & \boldsymbol{c}^\top \boldsymbol{x} \\
\text{s.t.} \quad & \bigvee_{i=1}^{k} \boldsymbol{A}_i \boldsymbol{x} \le \boldsymbol{b}_i \\
& \boldsymbol{x} \in \mathbb{R}^n,
\end{aligned} \tag{4}$$

with parameters $\boldsymbol{c} \in \mathbb{R}^n$, $\boldsymbol{A}_i \in \mathbb{R}^{m\times n}$ and $\boldsymbol{b}_i \in \mathbb{R}^m \ \forall i = 1, \ldots, k$. The disjunctive program in Eq. (4) may be written equivalently as a mixed-integer linear program (MILP) by introducing binary variables $\boldsymbol{y}$ and auxiliary continuous variables $\boldsymbol{x}_i$ (Hooker 2002):

$$\begin{aligned}
\min_{\boldsymbol{x},\boldsymbol{x}_i,\boldsymbol{y}} \quad & \boldsymbol{c}^\top \boldsymbol{x} \\
\text{s.t.} \quad & \boldsymbol{A}_i \boldsymbol{x_i} \le \boldsymbol{b}_i y_i, && \forall i = 1, \ldots, k \\
& \boldsymbol{x} = \boldsymbol{x}_1 + \ldots + \boldsymbol{x}_k \\
& \boldsymbol{x} \in \mathbb{R}^n \\
& \boldsymbol{x}_i \in \mathbb{R}^n && \forall i = 1, \ldots, k \\
& \boldsymbol{y} \in \{0, 1\}^k .
\end{aligned} \tag{5}$$

Disjunctive programming allows model developers to write certain problems more concisely and/or more meaningfully. For example, selecting one element i out of a set $i \in \{1, \ldots, n\}$, i.e. set partitioning, is a common constraint in process systems engineering:

$$\sum_{i=1}^{n} y_i = 1 \text{ where } y_i \in \{0, 1\} . \tag{6}$$

As a disjunctive constraint, Eq. (6) is easily identified as a selection constraint:

$$\bigvee_{i=1}^{n} \left(y_i = 1 \wedge \bigwedge_{i \ne j} y_j = 0 \right) .$$

Most approaches for solving disjunctive programs replace propositional variables with binary variables. In the resulting model, $y_i = 1$ implies a set of active constraints and $y_i = 0$ implies associated inactive constraints.

3.2. Generalized Disjunctive Programming

Raman and Grossmann (1994) developed generalized disjunctive programming (GDP), an extension of disjunctive programming incorporating nonlinear functions. GDP offers a natural, intuitive framework to model applications with logical dependencies, e.g. (i) job shop scheduling and (ii) process network superstructure design (Türkay and Grossmann 1996, Lee and Grossmann 2000). A GDP is formulated (Grossmann and Ruiz 2012):

$$
\begin{aligned}
\min_{\boldsymbol{x},\boldsymbol{Y},\boldsymbol{c}} \quad & Z = f(\boldsymbol{x}) + \sum_{k\in K} c_k \\
\text{s.t.} \quad & g(\boldsymbol{x}) \le \boldsymbol{0} \\
& \bigvee_{i\in D_k} \begin{bmatrix} Y_{ik} \\ r_{ik}(\boldsymbol{x}) \le \boldsymbol{0} \\ c_k = Y_{ik} \end{bmatrix} \qquad k \in K \\
& \Omega(\boldsymbol{Y}) = \mathit{True} \\
& \boldsymbol{x_l} \le \boldsymbol{x} \le \boldsymbol{x_u} \\
& \boldsymbol{x} \in \mathbb{R}^n, c_k \in \mathbb{R},\ Y_{ik} \in \{\mathit{True},\ \mathit{False}\},
\end{aligned}
\tag{7}
$$

where $f : \mathbb{R}^n \to \mathbb{R}$, $g : \mathbb{R}^n \to \mathbb{R}^m$, and $r_{ik} : \mathbb{R}^n \to \mathbb{R}^p$ may be nonlinear nonconvex functions. Each disjunction $k \in K$ is composed of conjunctions $i \in D_k$, i.e. the Eq. (7) square brackets. Each conjunction has a Boolean variable Y_{ik}, inequality $r_{ik}(\boldsymbol{x}) \le \boldsymbol{0}$, and cost variable c_k. If $Y_{ik} = \mathit{True}$ then the formulation enforces both $r_{ik}(\boldsymbol{x}) \le \boldsymbol{0}$ and $c_k = Y_{ik}$. Otherwise they are ignored. Propositional formula $\Omega(\boldsymbol{Y}) = \mathit{True}$ typically contains an exclusive disjunction assumption $\underset{i\in D_k}{\veebar} Y_{ik}$ for each $k \in K$. The exclusive disjunction assumption ensures correctness by restricting c_k to a single Y_{ik}.

To leverage existing MINLP solvers and relaxation techniques, GDP models are often reformulated as MINLPs (Ruiz et al. 2012). A GDP model may be reformulated as using either the big-M (Nemhauser and Wolsey 1988) or hull relaxation (Lee and Grossmann 2000) reformulation. Both reformulations replace propositional variables Y_{ik} with binary variables y_{ik} where $Y_{ik} = \mathit{True} \leftrightarrow y_{ik} = 1$. The reformulations also substitute:

$$\sum_{i\in D_k} y_{ik} = 1, \qquad \forall k \in K \tag{8}$$

$$\boldsymbol{Ay} \le \boldsymbol{a}, \tag{9}$$

for $\Omega(\boldsymbol{Y}) = \mathit{True}$, where Eq. (8) corresponds to $\underset{i\in D_k}{\veebar} Y_{ik}$ for each $k \in K$. The big-M reformulation replaces the Eq. (7) disjunction with:

$$r_{ik}(\boldsymbol{x}) \le \boldsymbol{M_{ik}}(1 - y_{ik}), \qquad \forall k \in K,\ i \in D_k,$$

whereas the hull reformulation uses the constraints:

$$\begin{aligned}
y_{ik} r_{ik}(\boldsymbol{\nu_{ik}}/y_{ik}) &\leq \mathbf{0}, && \forall k \in K, i \in D_k \\
\mathbf{0} \leq \boldsymbol{\nu_{ik}} &\leq y_{ik}\boldsymbol{x_u}, && \forall k \in K, i \in D_k \\
\boldsymbol{\nu_{ik}} &\in \mathbb{R}^n.
\end{aligned}$$

The hull relaxation is at least as tight as big-M but incorporates additional continuous variables ($\boldsymbol{\nu_{ik}}$). There has been significant research into appropriate algorithms for interesting classes of GDP's (Türkay and Grossmann 1996, Lee and Grossmann 2000, Vecchietti et al. 2003, Ruiz et al. 2012, Trespalacios and Grossmann 2016).

3.3. Mixed Logical-Linear Programming

Mixed logical-linear programming (MLLP) is formulated (Hooker and Osorio 1999):

$$\begin{aligned}
&\min \;\; \boldsymbol{c}^\top \boldsymbol{x} \\
&\text{s.t.} \quad p_j(\boldsymbol{Y}, \boldsymbol{y}) \rightarrow (\boldsymbol{A_j x} \geq \boldsymbol{a_j}),\ j \in \mathcal{J} \mid q_i(\boldsymbol{Y}, \boldsymbol{y}),\ i \in \mathcal{I}\ .
\end{aligned} \tag{10}$$

Eq. (10) splits the constraints into continuous and logical parts, on the left and right of the bar, respectively. The logical part consists of formulas $q_i(\boldsymbol{Y}, \boldsymbol{y})$ where $\boldsymbol{Y} \in \{\mathit{True}, \mathit{False}\}^{n_B}$ and $\boldsymbol{y} \in \mathbb{Z}^{n_I}$. The continuous part is formulated as logical implications such that if $p_j(\boldsymbol{Y}, \boldsymbol{y})$ is true then the constraint $\boldsymbol{A_j x} \leq \boldsymbol{a_j}$ is imposed.

MLLP models are solved by branching on the propositional variables $\boldsymbol{Y}$ and discrete variables $\boldsymbol{y}$. As branching takes place, MLLP progressively strengthens the relaxation by enforcing constraints $\boldsymbol{A_j x} \geq \boldsymbol{a_j}$ if the corresponding antecedent p_j is true. Since the logical part is separated from the continuous part, MLLP enables propositional satisfiability algorithms to derive further logical constraints and prune the search space.

An MLLP model may look different from the equivalent MILP. For many cases, e.g. where MILP binary variables model existence or assignment, MLLP may result in an easier-to-comprehend model with fewer variables. MLLP may be extended to models with nonlinear constraints, i.e. to mixed logical-nonlinear programming (MLNLP) (Türkay and Grossmann 1996, Bollapragada et al. 2001, Bemporad and Giorgetti 2004, 2006, Carbonneau et al. 2011, 2012).

3.4. Optimization Methods based on Satisfiability Modulo Theories

Even advances such as GDP (Grossmann and Ruiz 2012) cannot compete with the expressiveness of constraints written in SMT solvers. SMT solvers support logical theories and dependencies, do precise arithmetic, and enable incremental solving. But SMT solvers may have performance issues with division, reasoning over integers, and only limited support for transcendental functions (de Moura and Passmore 2013). MINLP tools support the transcendental functions and scale well for mixed integer reasoning (Carvajal et al. 2014), but MINLP solvers cannot solve incrementally and have limited support for logical constraints and are sensitive to rounding errors. Table 1 summarizes distinctions between SMT and MINLP.

Two of the most prominent SMT-based optimization methods are optimization modulo theories and integer linear programming modulo theories.

Optimization modulo theories integrates optimization and SMT with respect to the theory of linear arithmetic over the rationals ($\mathcal{LA}(\mathbb{Q})$) (Sebastiani and Tomasi 2015). The models are equivalent to MILP problems. Sebastiani and Tomasi (2015) consider different approaches to solve the MILP problems, e.g. offline and inline schemas with linear, binary or adaptive search. Sebastiani and Tomasi (2015) compare optimization modulo theories versus linear GDP using both a convex hull and a big-M relaxation. The comparisons, based on strip packing and job shop scheduling case studies (Sawaya and Grossmann 2005), show that an SMT solver may be used for optimization.

Integer linear programming modulo theories is an optimization framework where MILP, rather than SAT, is leveraged as the efficient solver (Manolios and Papavasileiou 2013). Integer linear programming modulo theories is an optimization framework in which difference logic is used to communicate with the solver. Manolios and Papavasileiou (2013) implement their framework as a constraint handler for the MILP solver SCIP (Achterberg 2007b, 2009). A weakness of an MILP-based approach is that floating point calculations may lead to wrong answers. Errors based on floating point do not happen in SMT because all formulae evaluate to true or false only.

3.5. Logic-Based Benders Decomposition

Hybrid optimization/logic approaches have been developed combining mixed-integer linear programming (MILP) and constraint programming (CP), e.g. Jain and Grossmann (2001), Maravelias and Grossmann (2004), Li and Womer (2008), Sitek (2014), or multiple levels of MILP, e.g. Maravelias

(2006). The hybrid formulations usually use logic-based Benders decomposition (LBBD) (Hooker and Ottoson 2003), a generalization of Benders decomposition (Benders 1962). The principles of Benders decomposition remain: we have a master problem and a subproblem which generates cuts if the solution from the master problem is infeasible. The difference is that LBBD requires a logic proof deriving an objective bound. Other hybrid algorithms use branch-and-check (Thorsteinsson 2001) or Lagrangian decomposition (Papageorgiou and Trespalacios 2016).

Hybrid MILP/CP methods are typically applied to scheduling and its variants (Sitek 2014). This is reasonable: CP is very good at assessing scheduling feasibility. The problem with hybrid MILP/CP is that, if the application does not have a suitable CP constraint, a hybrid method may be poor since bespoke CP constraints take full advantage of very specific mathematical structures. This manuscript evaluates satisfiability modulo theories as an alternative to CP in the hybrid scheme.

4. Satisfiability Modulo Theories for Process Systems Engineering

This section shows the applicability of SMT algorithms and implementations with respect to: (i) two-dimensional bin packing, (ii) model explainers, and (iii) MINLP solvers.

4.1. Two-Dimensional Bin Packing

Two-dimensional bin packing (2BP) is the problem:

> Given a set of rectangular items $\mathcal{I}$. What is the minimum number of rectangular bins with width W and height H needed to pack all items $\mathcal{I}$ without overlapping or rotating the items?

This problem has typology class **2BP|O|F** (Lodi et al. 1999), i.e. oriented and free cutting. See Table 2 for descriptions of the sets, parameters and variables mentioned in the formulations.

4.1.1. Logical Model

In 2BP a core constraint is: *two different items in the same bin should not overlap.* A logical formulation naturally captures this if-then relationship as the following model shows.

$$\min \sum_{b=1}^{N} z_b \tag{11a}$$

Table 2: Model symbols for the two-dimensional bin packing (2BP) problem.

Name	Description
Sets	
$i, j \in \mathcal{I} = \{1, \ldots, N\}$	Items
Parameters	
W, H	Width and height of the bins respectively
W_i, H_i	Width and height of item i respectively
Variables	
ζ_b	Activity of bin b, Boolean
ζ_{ib}	Item i assigned to bin b, Boolean
z_b	Activity of bin b, Binary
z_{ib}	Item i assigned to bin b, Binary
$z_{ij}^{(k)}$	Activity of disjunct i, j, k, Binary
x_i, y_i	Lower left coordinate of item i
v	Largest active bin index
m_i	Bin containing item i
l_{ij}	Item i left of item j
b_{ij}	Item i below item j
p_{ij}	Item i in lower index bin than item j

$$\text{s.t.} \bigvee_{b=1}^{N} \left(\zeta_{ib} \wedge \bigwedge_{b' \neq b} \neg \zeta_{ib'} \right) \qquad \forall i \in \mathcal{I} \tag{11b}$$

$$(\zeta_{ib} \wedge \zeta_{jb}) \rightarrow \begin{cases} (x_i + W_i \leq x_j) \vee (x_j + W_j \leq x_i) \\ \vee (y_i + H_i \leq y_j) \vee (y_j + H_j \leq y_i) \end{cases} \qquad \forall b \in \mathcal{B}, i, j \in \mathcal{I}, i < j \tag{11c}$$

$$\zeta_{ib} \rightarrow \zeta_b \qquad \forall b \in \mathcal{B}, i \in \mathcal{I} \tag{11d}$$

$$\zeta_b \rightarrow (z_b = 1) \qquad \forall b \in \mathcal{B} \tag{11e}$$

$$\neg \zeta_b \rightarrow (z_b = 0) \qquad \forall b \in \mathcal{B} \tag{11f}$$

$$0 \leq x_i \leq W - W_i, 0 \leq y_i \leq H - H_i \qquad \forall i \in \mathcal{I} \tag{11g}$$

$$\zeta_b, \zeta_{ib} \in \{\text{True}, \text{False}\}, z_b \in \{0, 1\} \qquad \forall b \in \mathcal{B}, i \in \mathcal{I} \tag{11h}$$

Equation (11a) minimizes the number of active bins. Equation (11b) fixes each item into a single bin. Equation (11c) ensures that any two items in the same bin do not overlap. Equation (11d) states that a bin must be active if it contains items. Equations (11e) and (11f) transform Boolean variables to binary variables to form the Eq. (11a) objective.

4.1.2. MILP Models

The MILP formulation (Pisinger and Sigurd 2007) is:

$$
\begin{aligned}
&\min v && && \text{(12a)}\\
&\text{s.t. } l_{ij} + l_{ji} + b_{ij} + b_{ji} + p_{ij} + p_{ji} \geq 1 && \forall i,j \in \mathcal{I}, i < j && \text{(12b)}\\
&x_i - x_j + W l_{ij} \leq W - W_i && \forall i,j \in \mathcal{I} && \text{(12c)}\\
&y_i - y_j + H b_{ij} \leq H - H_i && \forall i,j \in \mathcal{I} && \text{(12d)}\\
&m_i - m_j + N p_{ij} \leq n - 1 && \forall i,j \in \mathcal{I} && \text{(12e)}\\
&1 \leq m_i \leq v && \forall i \in \mathcal{I} && \text{(12f)}\\
&m_i \leq i && \forall i \in \mathcal{I} && \text{(12g)}\\
&0 \leq x_i \leq W - W_i, 0 \leq y_i \leq H - H_i && \forall i \in \mathcal{I} && \text{(12h)}\\
&l_{ij}, b_{ij}, p_{ij} \in \{0,1\} && \forall i,j \in \mathcal{I} && \text{(12i)}\\
&m_i, v \in \mathbb{Z} && \forall i \in \mathcal{I} && \text{(12j)}
\end{aligned}
$$

Equation (12a) minimizes the number of active bins. Equation (12b) states that items are in different bins or that they do not overlap. Equations (12c) and (12d) characterize non-overlapping items. Equation (12e) characterizes items being placed in different bins. Equation (12f) assigns the maximal active bin to v. Equation (12g) is a symmetry breaking constraint.

4.1.3. Optimizing with SMT

SMT assesses feasibility of a constraint set C. When C is feasible, an SMT solver provides a feasible solution, otherwise SMT derives an unsatisfiable core. A simple iterative approach addresses the Eq. (11) *optimization* problem: Begin by removing the Eq. (11a) optimization objective which results in the Eqs. (11b) to (11h) feasibility problem, let this feasibility problem be F_0. After solving F_0 with SMT, calculate U_1, the objective function evaluated at the F_0 feasible solution. Define feasibility problem F_1 by extending F_0 with additional constraint:

$$\sum_{b=1}^{N} z_b < U_1,$$

i.e. bound the objective function. Repeat this process until the SMT solver proves some F_i, $i > 0$, is infeasible, and thereby conclude that the F_{i-1} solution is optimal. This iterative approach solves successive feasibility problems to derive a sequence of decreasing objective values $\{U_i\}$. The difference between feasibility problems F_0 and F_i is that F_i has additional constraints:

$$\sum_{b=1}^{N} z_b < U_{i'}, \quad \forall i' \in \{1, \dots, i\}. \tag{13}$$

The Eq. (11) optimization problem may alternatively be addressed via SMT with (i) one of the Section 3.4 optimization frameworks or (ii) a black-box SMT-based optimization solver (Bjørner et al. 2015, Sebastiani and Trentin 2015, Callia D'Iddio and Huth 2017). The following discussion uses SMT as a feasibility solver to leverage unsatisfiable cores for cut derivation and branching. Our methods are similar to using a logic-based Benders decomposition where the master problem is embedded into the algorithm.

4.1.4. Symmetry in Two-Dimensional Bin Packing

This section develops methods where an SMT solver assesses two decision problems. The first is the two-dimensional orthogonal packing problem (Baker et al. 1980):

$$\text{OPP}(\mathcal{I}', W, H) \tag{14}$$

that questions whether a single bin of width W and height H packs all items in $\mathcal{I}'$. The constraints of feasibility problem OPP are the consequent of Eq. (11c) and Eq. (11g). As shorthand, we use $\text{OPP}^{(\mathcal{I})}$ since all bins are equivalent. The second decision problem is the 2BP decision problem:

$$\text{D2BP}(\mathcal{I}, k, W, H) \tag{15}$$

that questions whether k or fewer bins each of width W and height H can pack all items in $\mathcal{I}$. To form the D2BP feasibility model, we remove the Eq. (11a) objective from the Eq. (11) optimization model and set:

$$\zeta_b = \text{False}, \quad \forall b \in \{k+1, \dots, N\}. \tag{16}$$

Propagating the Eq. (16) assignments reduces D2BP to variables and constraints that only involve bins $b \in \{1, \dots, k\}$. As shorthand, we use $\text{D2BP}^{(\text{k})}$ since our algorithms are only concerned with the number of available bins.

Two-dimensional bin packing exhibits symmetry, e.g. permuting bin indices immediately results in an identical packing. For an optimal 2BP solution with objective k, the same solution occurs $\binom{N}{k}k!$ times. One approach to break this symmetry is by adding additional constraints, e.g. Eq. (12g). But additional constraints only handle symmetries at a global level and further symmetries arise at a local level when considering OPP.

Descending Strategy. Consider an algorithm that iteratively bounds the objective with Eq. (13) constraints. Assume that, in the current iteration, the algorithm checks for $(k-1)$ bins or fewer. Constraint (13) is problematic because it symmetrically allows any subset of $(k-1)$ active bins, i.e. the SMT solver checks D2BP$^{\text{(k-1)}}$ once for each subset of $(k-1)$ bins. We break the Eq. (13) symmetry by adding a constraint to deactivate bins, i.e. Eq. (16) sets $\neg\zeta_b$. Iteratively introducing Eq. (16) is the *descending* algorithm. Since bin deactivation becomes part of the algorithm, we remove variables z_b (and associated constraints) from the Eq. (11) formulation. This algorithm aids satisfiability searches because the SMT conflict graph from any previous iteration is always valid in later iterations.

An alternative method is equivalent to binary search. Initialize the algorithm by setting an lower and upper bound on the problem (1 and N). At each iteration, activate a number of bins that is halfway between the bounds. Update the lower/upper bound depending on whether the halfway problem is unsatisfiable/satisfiable. Terminate when the bounds are equal. We do not use the binary search algorithm because it assesses unsatisfiability, a costly operation for D2BP and therefore heuristically poor for this particular application. Section 4.3.8 describes how MINLP solver `ManyOpt` enables binary search in a more generic context.

Ascending Strategy. While the descending algorithm progressively builds a conflict graph, it can struggle to efficiently prove optimality as symmetry occurs when addressing infeasibilities. For example, if OPP$^{(\mathcal{I}')}$ is unsatisfiable for some subset of items $\mathcal{I}'$, then having items $\mathcal{I}'$ in any bin for a D2BP$^{\text{(k)}}$ instance is a symmetry in unsatisfiability checks. The lower bounding *ascending algorithm* reduces symmetry in optimality proofs. This algorithm naturally extends to a branch-and-bound algorithm described in Section 4.1.5.

At a high level, the ascending and descending algorithms are opposites. The descending algorithm initially activates all bins and iteratively deactivates bins until the first unsatisfiable result (optimality proof). The ascending algorithm initially deactivates all but one bin and iteratively activates one additional bin until the first satisfiable (optimal) result, i.e. it assesses D2BP$^{\text{(k)}}$ for $k = 1, \ldots, N$ (in order) and terminates at the first satisfiable k.

Each unsatisfiable ascending algorithm iteration proves that we need at least one more bin. Also, each unsatisfiable iteration generates an unsatisfiable core. In the 2BP case, any Eq. (11c) constraints in the unsatisfiable core are a conflicting subset of items. Assuming that the k^{th} iteration has k active bins and the items corresponding to Eq. (11c) constraints are $\{i_1, \dots, i_t\}$, the unsatisfiable core has the interpretation:

items $\{i_1, \dots, i_t\}$ cannot be packed into k bins.

In iteration $(k+1)$, the ascending algorithm derives cuts relating to this set of t items and thereby prevents symmetric unsatisfiability assessments. The symmetric property is: any set of items that cannot be packed in the same bin cannot be packed in any bin. Between ascending algorithm iterations, we run intermediate OPP checks on unsatisfiable core subsets and add:

$$\bigvee_{i \in \mathcal{I}'} \neg \zeta_{ib}, \quad \forall b \in \mathcal{B}, \tag{17}$$

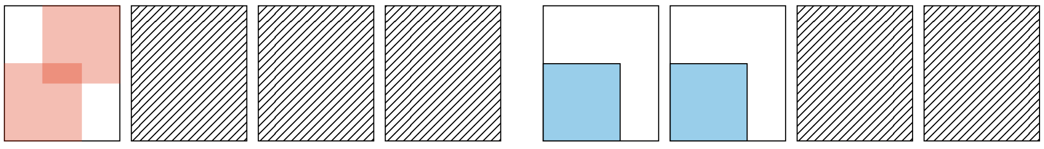
for any unsatisfiable result of $\text{OPP}^{(\mathcal{I}')}$. The MILP equivalent of Eq. (17) is:

$$\sum_{\substack{i,j \in \mathcal{I}' \\ i \neq j}} p_{ij} + p_{ji} \geq 1. \tag{18}$$

Eq. (17) and (18) have the same feasible space, but the Eq. (18) cut can lose its logical meaning in an MILP solving strategy with relaxed fractional values. Eq. (17) will not combine the $\mathcal{I}'$ items because Eq. (17) is a propositional clause that is not relaxed in the SAT subsolve of an SMT solver.

When deriving the Eq. (17) cuts, deciding OPP is expensive for each unsatisfiable core subset since there are exponentially many such subsets. But the many OPP checks may be unnecessary since any unsatisfiable set of items dominates any superset for OPP. We leverage this dominance property by checking all subsets in ascending size order and filtering any dominated supersets. Filtering all subsets of a given size terminates the OPP checks and the ascending algorithm continues by assessing the next iteration (additional bin). Effectively, we're building a dictionary of previously checked subsets to prevent repeat checks.

The Eq. (17) cuts aid SMT unsatisfiability proofs by reducing the number of symmetric OPP checks required to assess D2BP. But, if k active bins are sub-optimal, the SMT solver will still have to derive an unsatisfiable core. Building this new core may contain redundant checks that the ascending algorithm has not yet investigated. We aim to (partially) break this symmetry by fixing items and thereby construct a partial optimal solution as the algorithm progresses, as Fig. 2 shows. The unsatisfiable core associated with $\text{D2BP}^{(\text{k})}$ implies that we require at least $(k+1)$ bins to pack the items.


(a) Iteration 1: unsatisfiable core. (b) Iteration 1: fixed items.

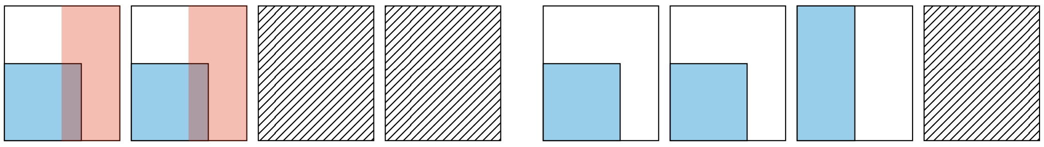

(c) Iteration 2: unsatisfiable core. (d) Iteration 2: fixed items.

Figure 2: How the algorithms fix initial items on iterations 1 and 2. Running the algorithm with one bin may return an unsatisfiable core with two items only (a), this means that these two items must be in their own bins as shown by (b). If later unsatisfiable cores only contain one unfixed item, e.g., (c), then the unfixed item is also placed in a separate bin as shown by (d).

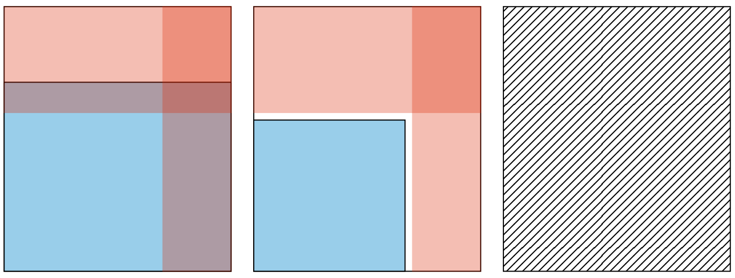


Figure 3: A scenario where we cannot definitively choose an item to fix in the next bin. The blue items are assigned to their respective bins (the position is not fixed within the bin) and the red (translucent) items are unfixed. Here one of the two red items must be placed in its own bin but we do not know which choice definitely leads to an optimal solution.

If the unsatisfiable core in first iteration (one bin) contains only two items, Fig. 2a, then we pack these items separately, Fig. 2b. On any later iteration, if the corresponding unsatisfiable core reduces to one item after filtering fixed items, then we know that this single unfixed item can be placed in the next bin, Figs. 2c and 2d show this for iteration 2. We stop fixing items if, after filtering, the corresponding unsatisfiable core has more than one item.

The description above concludes the ascending algorithm definition. this paragraph motivates the branch-and-bound extension described in the next section. The Fig. 2 process simply derives a set of items that must be placed separately in any feasible solution. When we stop fixing items, the corresponding unsatisfiable core, after filtering, contains more than one item. Here the interpretation is: one of the unfixed items must be placed in its own bin, see e.g. Fig. 3. As Fig. 3 shows, to continue fixing items requires assessing alternatives, motivating a branch-and-bound strategy.

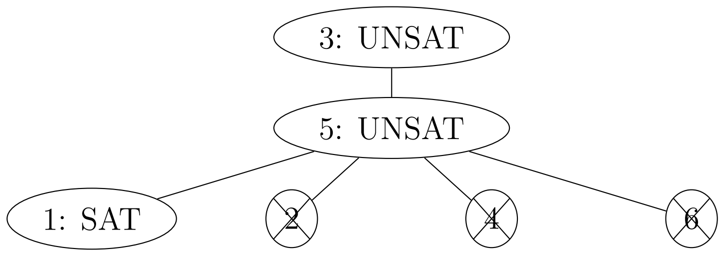


Figure 4: An example branching tree for an instance with optimal objective 3. Each node contains an item i and a node's depth, d, corresponds to fixing item i in bin $d+1$. Crossed nodes are pruned. In this instance the first iteration fixed items 3 and 5 in bins 1 and 2. The second iteration derives unsatisfiable core of unfixed items $\{1, 2, 4, 6\}$ (the alternative choices for bin 3) and fixes item 1 in bin 3. The third iteration finds fixing items 3, 5 and 1 separately is feasible. The remaining nodes are pruned since a feasible solution equal to their depth has already been found. There are no further branches after pruning, therefore the bottom left branch gives an optimal solution.

4.1.5. SMT Based Branch-and-Bound for 2BP

The branch-and-bound algorithm extends the ascending algorithm of Section 4.1.4 by branching on alternative choices from an unsatisfiable core of unfixed items. Figure 4 provides an overview of the structure of the branch-and-bound tree. Taking the root node as a special case where we pop an item from the first unsatisfiable core and fix it in the first bin, each node branches on the elements of the unfixed items in its corresponding unsatisfiable core. The branch-and-bound algorithm matches the ascending algorithm up to a chain of unary branches from the root node of the tree. If the branch-and-bound algorithm requires a non-trivial decision, e.g. Fig. 3, it forms a set of branches, each of which corresponds to an unfixed item in the current unsatisfiable core. These branches form nodes at some depth d and fix their item into bin $(d + 1)$. An SMT assessment at a given node addresses a slightly stricter version of D2BP since there is an item fixed in each bin, i.e. assessing whether we satisfy 2BP in exactly $(d + 1)$ bins with the corresponding items fixes. Since the depth of a node corresponds to how many bins are active, the branch-and-bound algorithm does not search beyond the depth of any feasible node that it has found. The branch-and-bound algorithm terminates by exhausting each alternative path.

Since the branch-and-bound algorithm extends the ascending algorithm, it still derives Eq. (17) cuts in a local setting. Assuming that $\mathrm{OPP}^{(\mathcal{I}')}$ is unsatisfiable with $j \in \mathcal{I}'$ and j fixed in bin b, the associated local cut is:

$$\bigvee_{\substack{i \in \mathcal{I}' \\ i \neq j}} \neg \zeta_{ib}. \tag{19}$$

Furthermore, any local cuts, i.e. cuts depending on a fixed item, are promoted to global cuts by including their corresponding fixed item when the branch-and-bound algorithm investigates sibling or ancestor branches. So all Eq. (17) cuts can become global regardless of where in the tree they are derived.

Fixing items in the branch-and-bound tree reduces the number of symmetric checks on a given root to leaf path. Adding/promoting Eq. (17) cuts aids unsatisfiability proofs across the entire unexplored tree. But assessing alternative branches of a given unsatisfiable core retains the symmetry of bin permutations, hence sibling branches contain identical solutions. The main problem is that, if we have explored a particular branch having fixed item i_1, then in a sibling branch, having the freedom to select any bin for i_1 includes paths explored by the first branch due to bin permutations. Lemma 1 derives *push back* cuts that remove this symmetry by forcing explored branch items to be pushed back into earlier bins.

Lemma 1. *Assume that the branch-and-bound algorithm is assessing alternatives at some node of depth $b-1$, i.e. each of the first b bins have one fixed item, and that this node is unsatisfiable. Let this node have unsatisfiable core of unfixed items $\mathcal{I}' = \{i_1, \dots, i_m\}$, $m > 1$, i.e. we have to assess each of these items being placed in bin $(b+1)$. When branching on element i_k, $k > 1$, if we add the cuts*

$$\bigvee_{b'=1}^{b} \zeta_{ib'}, \quad \forall i \in \mathcal{I}(k) = \{i_{k'} | k' < k\}, \tag{20}$$

then among all branches the best objective will match that of assessing these alternative branches without adding the Eq. (20) cuts.

Proof. Define OPT_k, $k \in \mathcal{K} = \{1, \dots, m\}$ as the optimal objective of branch i_k with its corresponding Eq. (20) cuts. Assume, for a contradiction, that there exists item $i_{k'}$, $1 < k' \leq m$ such that without adding its Eq. (20) cuts, gives optimal objective OPT' that satisfies:

$$OPT' < \min_{k \in \mathcal{K}} \{OPT_k\}. \tag{21}$$

Let f_1 map items to bins for the OPT' solution. We define the set S (items that violate Eq. (20)) as:

$$S = \{k \,|\, f_1(i_k) \geq b+1,\, i_k \in \mathcal{I}(k')\}.$$

Clearly S is non-empty, otherwise $OPT' = OPT_{k'} \geq \min_{k \in \mathcal{K}}\{OPT_k\}$. Let $k_l = \min S$. We permute the bins such that item k_l is in bin $b+1$ with corresponding item to bin map:

$$f_2(i) = \begin{cases} b+1, & \text{if } f_1(i) = f_1(i_{k_l}) \\ f_1(i_{k_l}), & \text{if } f_1(i) = b+1 \\ f_1(i), & \text{otherwise.} \end{cases}$$

But f_2 is a feasible solution for branch i_{k_l} with its associated Eq. (20) cuts as $k_l = \min S$ (note that branch i_1 does not add any cuts). Since f_2 has objective OPT':

$$\min_{k \in \mathcal{K}} \{OPT_k\} \leq OPT_{k_l} \leq OPT',$$

contradicting the Eq. (21) assumption. □

Repeatedly applying Lemma 1 in the branch-and-bound algorithm adds a level of independence between alternative search paths. Eq. (20) constraints aid unsatisfiability proofs when branching on later elements of $\mathcal{I}'$, since there are a larger number of pushed back items. Furthermore, we automatically remove pushed back items from new branch sets even though they are unfixed, thus reducing the number of branching decisions. Lemma 2 proves that, when pushing back item i, we also can push back all items i' that are identical to i.

Lemma 2. *Assume that the branch-and-bound algorithm is assessing alternatives at some node of depth $(b-1)$, i.e. each of the first b bins have one fixed item, and that this node is unsatisfiable. Let this node have unsatisfiable core of unfixed items $\mathcal{I}' = \{i_1, \ldots, i_m\}$, $m > 1$, i.e. we have to assess each of these items being placed in bin $(b+1)$. Then when we push back item $i \in \mathcal{I}'$ (according to Eq. (20)) we can also push back all items $i' \in \mathcal{I}$ that are identical to i.*

Proof. Let item i' be identical to i such that i' is not fixed or pushed back (the lemma holds trivially for these cases). With item i pushed back according to its Eq. (20) constraints, pick any feasible solution with i' placed in bin $b' \geq b+1$ and let f_1 be the associated item to bin map. Then the following map:

$$f_2(j) = \begin{cases} b+1, & \text{if } f_1(j) = b' \\ b', & \text{if } f_1(j) = b+1 \\ f_1(j), & \text{otherwise} \end{cases}$$

permutes bins $(b+1)$ and b'. But f_2 is a feasible solution for the item i branch if we swap identical items i and i'. Since branch i contains any feasible solution with item i' in bin $b' \geq b+1$, we can also push back i' as well as i. $\square$

Finally, since we only fix a single item per bin, we eliminate mirror and rotational symmetries of fixed items by limiting their center to the lower left quadrant of their bins with the constraints:

$$x_i \leq \frac{W - W_i}{2}, \quad y_i \leq \frac{H - H_i}{2}. \tag{22}$$

Equation (22) cuts involve the continuous variables, so they correspond to the arithmetic theory solver aspect of an SMT solver. Since SAT solver checks relax the arithmetic theories, these cuts may be less effective in SMT.

4.1.6. Numerical Results

We solve the MILP and SMT models using CPLEX 12.7 and Gurobi 6.0.3, and Z3 4.5.1 (de Moura and Bjørner 2008a), respectively. The MILP models are in Pyomo (Hart et al. 2011, 2012), and the Z3 implementation is in Python. All test cases were run on a HP EliteDesk 800 G1 TWR with 16GB RAM and an Intel® Core$^{\text{TM}}$ i7-4770 @ 3.40Ghz running Ubuntu 16.04.1 LTS. The test set contains 500 instances grouped into 10 classes. Each class has 10 instances with 20, 40, 60, 80 and 100 items. These instances were originally generated by Berkey and Wang (1987). Martello and Vigo (1998) and Lodi et al. (1999) describe the differences between classes.

Figure 5 is a performance profile comparing time-to-convergence (Dolan and Moré 2002). We compare the descending, ascending and branch-and-bound algorithm to CPLEX and Gurobi. We also add a *can solve* line for the Pisinger and Sigurd (2007) column generation and constaint programming results. We use SMT to assess OPP, Pisinger and Sigurd (2007) use a specialized algorithm for the underlying OPP decision problem, the 'P&S (2007) solved' *can solve* line in Fig. 5 gives an indication of the kind of performance improvement achieved by using more bespoke methods. We compare against the mixed-integer solvers in the subsequent analysis. The branch-and-bound algorithm performs well and solves most tractable problems within two orders of magnitude of the fastest solve time. The descending algorithm and the mixed-integer solvers perform similarly. The performance difference between the ascending and branch-and-bound algorithms, where the branch-and-bound algorithm solves twice as many instances in the hour, quantifies the effect of 2BP symmetry. Overall, the methods can solve about half of the instances within the hour. For the branch-and-bound algorithm,

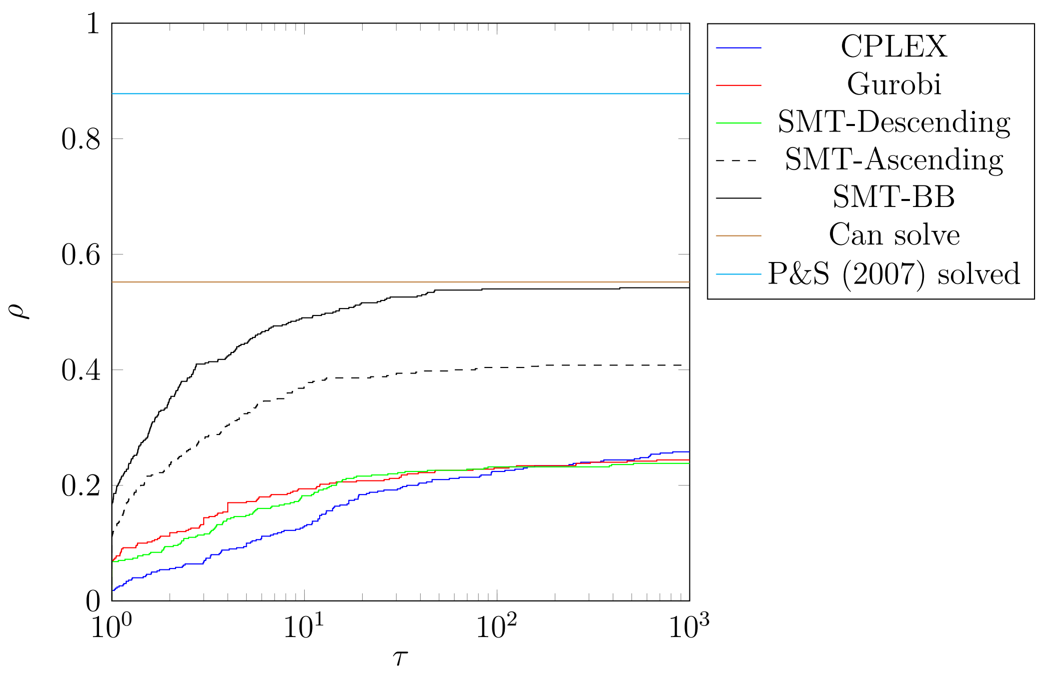


Figure 5: Performance profile for 500 bin packing instances. Each solver has a timelimit of one hour. All solvers are limited to one thread. P&S (2007) is the number of problems that Pisinger and Sigurd (2007) solve to optimality.

this limitation is due to larger instances requiring deeper searches within the tree. Deeper searches require larger unsatisfiable cores when proving unsatisfiability and, for the initial part of the search, there are fewer Eq. (17) cuts present to aid such proofs. However, assuming that there has been sufficient exploration, i.e. we have generated push back constraints and symmetry breaking Eq. (17) cuts, SMT BB appears to speed up in its later assessments.

Table 3 displays the number of instances optimally solved in each class. The branch-and-bound algorithm generally outperforms the other solvers, but there is a fair amount of differences among classes. Generally, the classes where the branch-and-bound algorithm solves more instances contain larger items, i.e. problems with easy-to-generate unsatisfiability proofs. For classes 2, 4 and 6, where instances contain many small items, all methods exhibit inferior performance. While the optimal objective is smaller for these instances, for SMT the unsatisfiable cores are relatively large and therefore take longer to generate. Our algorithms use SMT as a black box for unsatisfiable core generation, so the branch-and-bound algorithm may not have sufficient time to explore the tree. The relative quality of the branch-and-bound algorithm shows the usefulness of an unsatisfiable core in algorithm design.

Figure 6 shows a performance profile comparing heuristic performance. The descending algorithm is a relatively good heuristic. The descending al-

Table 3: The number of instances (Berkey and Wang 1987, Martello and Vigo 1998) solved to optimality by each solver. Each class has 50 instances.

Class	CPLEX	Gurobi	SMT-Descending	SMT-Ascending	SMT-BB
1	10	7	11	22	**36**
2	10	10	**11**	**11**	**11**
3	11	12	17	33	**45**
4	11	11	**12**	11	11
5	13	13	13	32	**43**
6	11	13	**14**	**14**	**14**
7	7	4	10	11	**15**
8	5	3	10	10	**16**
9	47	**49**	1	36	**49**
10	12	8	21	26	**33**

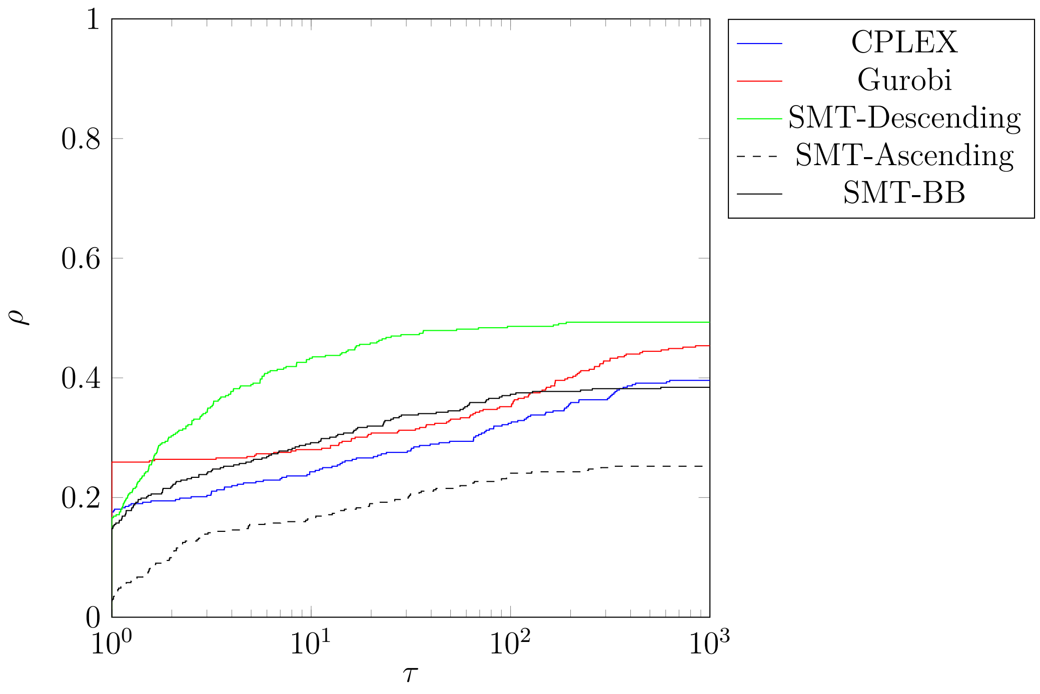


Figure 6: Performance profile for 500 bin packing instances comparing heuristic times to optimality. Each solver has a timelimit of one hour. All solvers are limited to one thread.

gorithm progressively reduces the number of available bins, to reduce symmetry, while building a globally applicable conflict graph, that helps in finding heuristic solutions. The ascending algorithm performs less well as its first 'heuristic' solution is an optimal solution, i.e. it has to prove optimality. The branch-and-bound algorithm's heuristic performance is also quite good, this suggests that the tree depth should not be too far from the optimal depth, i.e. the branch-and-bound algorithm does not have to search too far.

4.2. Unsatisfiable Core for Cut Generation and Model Explainers

SMT solvers can derive an *unsatisfiable core* of a constraint set, i.e. a constraint subset that is unsatisfiable. This section discusses cut generation and model explainers as two applications of unsatisfiable cores.

4.2.1. Cut Generation

Logic-based Benders decomposition (LBBD) is a framework for solving problems with master-subproblem structure, e.g. a high level assignment and subproblem assessments resulting from the assignments. In LBBD, the subproblem generates cutting planes to add to the master problem. These cutting planes can be *infeasibility cuts* which exclude infeasible solutions or *lower bounding cuts* which enforce an objective bound.

In each major iteration, a subproblem can generate a cut incorporating all of its corresponding assignments, but the resulting cut is often weak because fewer assignments may give an equivalent cut, i.e. a smaller unsatisfiable core. Hooker (2007) develops an approach generating stronger cuts by repeating local subproblem assessments and removing subsets that do not change the underlying result. Hooker (2007) thereby derives a minimal unsatisfiable core, an unsatisfiable core where removing any constraint makes it satisfiable. SMT does not necessarily calculate minimal unsatisfiable cores, but the reported cores may be smaller than the entire assignment set. Smaller cores imply fewer of the expensive local iterations needed to reduce to a minimal unsatisfiable core.

4.2.2. Model Explainers

Practical applications of optimization models and algorithms involve using abstract modeling software. Consider a scenario where a mathematical model is known to be correct and an instance is known to be feasible. If a solver or algorithm states infeasibility, then an implementation needs to be corrected. Here SMT is a useful tool to aid development.

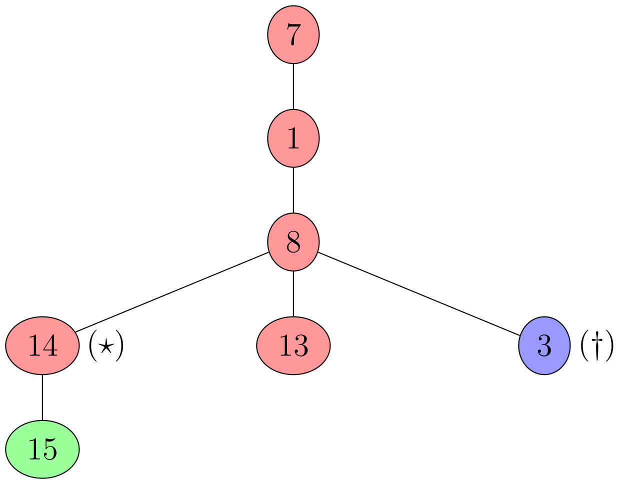


Figure 7: An example of a Section 4.1.5 branch-and-bound tree where an incorrect implementation requires correction. Red nodes are unsatisfiable, green nodes are satisfiable and blue nodes are reported unsatisfiable nodes that we know a feasible solution for, i.e. an incorrect conclusion. The error assumed here is that we generate a local cut that item 15 is not in bin 4 at (⋆), however the incorrect implementation fails to remove/promote the local cut before switching to branch (†) giving the wrong conclusion.

A feasible but incorrect model may be over- or under-constrained. Over-constrained models may report an sub-optimal objective or infeasible constraints. Reasoning the incorrectness of a model implementation may be difficult because of various properties, e.g. the instance size or incorrect application of transformation techniques. Analyzing a concrete instance can be a time consuming task since it may have a large number of constraints. But an incorrect model still contains a constraint subset explaining what is wrong, i.e. an unsatisfiable core.

When an incorrect model is over-constrained, it is sufficient to limit our discussion to the infeasible case since, if the model reports a sub-optimal objective, we are questioning why a better objective is infeasible. The unsatisfiable core corresponding to the infeasible model may have many constraints, but we can filter, i.e. eliminate, correct constraints such as variable bounds. After filtering, the constraint subset may be much smaller than the entire instance and analysis becomes easier. If we assume further that we know a feasible solution, we can add constraints fixing variables to this solution. Fixing variables may produce an unsatisfiable core in terms of the feasible solution that we fed to the model. Fixing a solution is more useful when the incorrect model reports the wrong optimal objective (here we would need to know a feasible solution that is better than the reported optimal objective).

Another, more likely case for elusive bugs is when we have an algorithm that adds and removes constraints, i.e. an issue may only occur for larger instances. Compulsory computational considerations may be a reason for having such a case, e.g. a given node of a branch-and-bound algorithm can add and remove constraints to an existing instance, or build a fresh instance. However, we may prefer the former when considering the computational cost of building a fresh instance. Figure 7 provides a concrete example in the context of the branch-and-bound algorithm of Section 4.1.5. The incorrect implementation assumed here is that local cuts are not removed when switching to branches where they are no longer valid. As Fig. 7 shows, we get a reported optimal objective of 5 when we should get 4. After filtering constraints at (†) in Fig. 7, e.g. variable bounds and inactive bins, an unsatisifable core consists of only 7 constraints among which are constraints generated by the algorithm. In particular, we have the constraint *item 15 is not in bin 4*, the significance of this constraint is that it corresponds to a local cut generated at ($\star$) therefore it is no longer valid telling us that there is an issue with handling of local cuts. The fact that we can limit our view to just a few constraints makes this error easier to find. Given just the result 'infeasible', finding such an error may require multiple re-solves with minor source code changes or log file parses, both of these tasks are time consuming (more so if the only failing instances are large).

4.3. Optimization Solvers based on Satisfiability Modulo Theories

SMT-based optimization solvers typically extend SMT solvers, e.g. Barcelogic (Bofill et al. 2008), Yices (Dutertre and de Moura 2006) and Z3 (de Moura and Bjørner 2008a). This section discusses solvers: νZ (§4.3.1), OptiMathSAT (§4.3.2) Symba (§4.3.3), and ManyOpt (§4.3.4). We focus on ManyOpt, the first SMT-based MINLP solver.

4.3.1. νZ

The νZ extension to the SMT solver Z3 adds optimization functionality (Bjørner and Phan 2014, Bjørner et al. 2015). νZ addresses weighted Max-SAT/SMT and linear arithmetic problems. νZ also supports multi-objective optimization via: lexicographic, Pareto fronts and box objectives. The νZ algorithms resemble the Nieuwenhuis and Oliveras (2006) approach, but, instead of implementing additional rules, νZ updates a variable value as the search progresses. The solver also has a resolution-based approach that reasons based on the unsatisfiable core (Narodytska and Bacchus 2014).

Bjørner and Phan (2014) solve the MILP subproblems by using SMT to find a feasible solution and invoking the simplex algorithm to minimize under the given integral assignment, the integral assignment is then rejected and a bounding constraint is placed on the objective.

4.3.2. OptiMathSAT

OptiMathSAT (Sebastiani and Trentin 2015) extends the SMT solver MATSAT5 (Cimatti et al. 2013). The optimization capabilities are similar to νZ, but the optimization constraints are formulated differently. Sebastiani and Tomasi (2015) compare OptiMathSAT to a GDP framework and show that the solver is competitive with the state-of-the-art.

4.3.3. Symba

Symba (Li et al. 2014) is built on top of Z3, but its implementation differs from νZ. Symba optimizes for linear *real* arithmetic (not limited to rational) however it does not support strict inequalities. The optimization process maintains a triple $\langle M, U, O\rangle$ where M is a set of models, U is an underapproximation and O is an overapproximation. The inference rules correspond to initialization, checking for unboundedness and tightening the under and over approximations. Symba only uses Z3 as a black box.

4.3.4. ManyOpt

`ManyOpt` (Callia D'Iddio and Huth 2017) is a deterministic ϵ-global optimization MINLP solver that extends SMT solving. ManySAT (Hamadi et al. 2009) inspires the `ManyOpt` design. The hypothesis is that no single method wins on a varied class of optimization problems. Therefore, `ManyOpt` parallelizes different approaches on a problem input and speeds up solving time by returning the first to "win" on that input. The `ManyOpt` solver is presented elsewhere, e.g. Callia D'Iddio and Huth (2017). The discussion here describes the relationship to the PSE community.

4.3.5. ManyOpt *Tool Architecture*

Figure 8 diagrams the `ManyOpt` structure. As input, `ManyOpt` takes (i) a relative optimality tolerance for the computed global optimum and (ii) an OSiL model (Fourer et al. 2010) of an MINLP problem. Since `ManyOpt` relies on exisiting SMT solvers, all nonlinearities must be polynomial. Then, `ManyOpt` executes *in parallel* a set of dataflows on the OSiL input. As shown in Fig. 8, one orange block is effectively a single feature and a feature vector is a selection of at least one feature from each layer, i.e. a feature vector realizes configurable tool layers. A feature is a particular method and a feature vector represents combining features to create an SMT-based optimizer. The tool layers are: pre-processing, integrality management, continuous relaxation optimization, and feasibility checking. Each layer contains an extensible feature list. For each feature vector, the data flow illustrated in Fig. 8 is:

- **SMTLIB 2.0 representation** This layer transforms the OSiL input into several SMTLIB 2.0 representations. These representations reflect the different feature vectors that will subsequently compete in parallel.

- **Pre-processing** SMT solving is rooted in SAT solving, so Boolean variables $\boldsymbol{Y} \in \{\mathit{True}, \mathit{False}\}^{n_B}$ are most natural to SMT. Recognizing that binary variables approximate Boolean variables better than general integer variables, this phase may transform the SMTLIB 2.0 representation from an MINLP formulation allowing general integer variables to a formulation where all discrete variables are binary. Section 4.3.6 discusses the *binarization* and *binarized flattening* techniques.

- **Integrality management** This layer, the first of three in the main optimization process, develops logical equivalents of branch-and-bound. Section 4.3.7 discusses the *one-by-one* and *all-in-one* approaches. This layer produces a continuous relaxation on integer and binary variables only. The propositional formulas are not relaxed. The layer subsequently relies on the *continuous relaxation optimization* layer to solve them. SMT solvers typically accept nonlinear arithmetic coupled with Boolean constraints, so this layer transforms the MINLP into problem(s) the SMT solvers can handle.

- **Continuous relaxation optimization** This layer takes a nonlinear optimization problem with propositional formulas and reduces it to a series of feasibility checks, i.e. SMT solves. Section 4.3.8 describes the *naive*, *unbounded binary search*, and *hybrid* features.

- **Feasibility checking** An SMT solver, using an SMTLIB 2.0 representation, verifies the feasibility of an NLP problem. `ManyOpt` supports the solver `Z3` (de Moura and Bjørner 2008b) directly. `ManyOpt` also supports SMT solvers `MathSAT` (Bruttomesso et al. 2008), `CVC4` (Barrett et al. 2011) and `YICES` (Dutertre 2014) through the `PySMT` library (Gario and Micheli 2015).

Many of the `ManyOpt` features mimick techniques familiar from MINLP optimization, but the `ManyOpt` approach tends to delegate algorithmic aspects to the SMT solver and focus on declarative aspects. In other words,

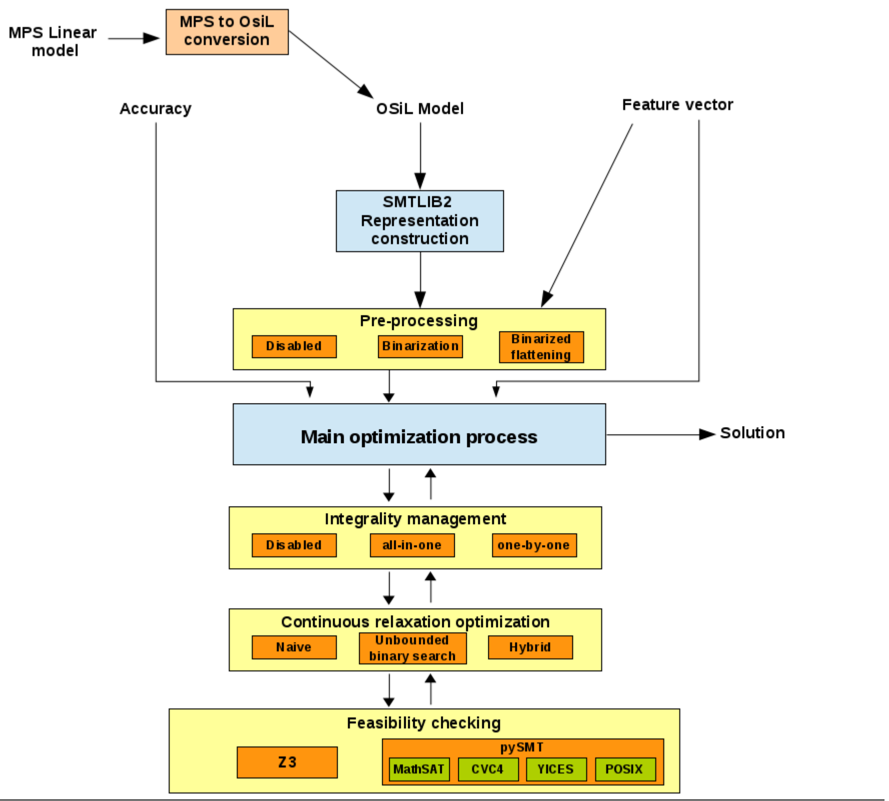


Figure 8: The architecture, approach, and dataflow of `ManyOpt` (Callia D'Iddio and Huth 2017). Section 4.3.5 describes how each layer (yellow block) contains a number of possible features (orange blocks). A feature vector is a specific choice of a feature (or features) from each layer. `ManyOpt` competes many feature vectors in parallel.

`ManyOpt` solves MINLP by solving a series of SMT problems rather than directly developing the algorithmic machinery present in most MINLP solvers.

4.3.6. ManyOpt *Pre-Processing Features*

Disabled. The MINLP can be directly sent to the main optimization process.

Binarization. Binarization transforms integer variables into binary $\{0,1\}$ variables, e.g. as described by Floudas (1995). An integer variable $x \in \mathbb{Z}$ bounded by $x^{\mathrm{LO}} \leq x \leq x^{\mathrm{UP}}$ may be replaced by q many variables $y_1, \ldots, y_q$ with $q = 1 + \left\lceil \log_2(x^{\mathrm{UP}} - x^{\mathrm{LO}}) \right\rceil$, equality constraint $x = x^{\mathrm{LO}} + y_1 + \cdots + 2^{q-1} y_q$, and $y_i \in \{0,1\}$ for all $i = 1, \ldots, q$.

Binarized flattening. An alternative to (algorithmic) branch-and-bound declaratively *flattens* the integer variables during the pre-processing stage, thus moving algorithmic complexity to the SMT solvers via the logical OR ($\vee$) operator. In other words, binarized flattening reformulates all integer variables $x \in \mathbb{Z}$ with $x^{\mathrm{LO}} \leq x \leq x^{\mathrm{UP}}$ into a continuous variable $x \in \mathbb{R}$ and Eq. (23) introduces one OR operator for each integer in the range $[x^{\mathrm{LO}}, x^{\mathrm{UP}}]$.

$$\left(x = x^{\mathrm{LO}}\right) \vee \left(x = x^{\mathrm{LO}} + 1\right) \vee \cdots \vee \left(x = x^{\mathrm{UP}} - 1\right) \vee \left(x = x^{\mathrm{UP}}\right). \qquad (23)$$

Binarization and binarized flattening. We can combine *binarization* and *binarized flattening* using $1 + \left\lceil \log_2(x^{\mathrm{UP}} - x^{\mathrm{LO}}) \right\rceil$ OR operators.

4.3.7. ManyOpt *Integrality Management Features*

If the pre-processing layer uses the *binarized flattening* feature or if there are no discrete variables in the optimization problem, then this layer is irrelevant. Otherwise, this layer performs the same function as a standard branch-and-bound approach by developing a new constraint or set of constraints that enforce any integrality violated in the *feasibility checking* layer.

Disabled. Disabling integrality management delegates enforcing integrality to the *feasibility checking* layer, where SMT solvers can attempt to manage this – for example with *integer* type variables.

One-by-one. If a relaxed solution violates at least one integrality constraint, choose one variable x whose value v violates an integrality constraints and add the assertion $x \leq \lfloor v \rfloor \vee x \geq \lceil v \rceil$ to the optimization problem. Since disjunctions are supported in SMT as assertions, this avoids splitting into two optimization problems by delegating combinatorial complexity to the SAT engine of the SMT solver.

All-in-one. The one-by-one approach only adds one assertion per iteration, but SMT solvers permit adding more than one constraint at a time. The *all-in-one* approach collects *all* variables whose values violate integrality constraints in a feasible solution and adds the above disjunction as an assertion for *all* such variables simultaneously.

4.3.8. ManyOpt *Continuous relaxation optimization features*

This layer provides algorithms to solve nonlinear optimization problems with propositional formulas. `ManyOpt` supports: the *naive* method (Eén and Sörensson 2006), the *unbounded binary search* method (Beaumont et al. 2015, Callia D'Iddio and Huth 2017), and a *hybrid* combination of these two methods (Callia D'Iddio and Huth 2017).

Naive. The naive approach is equivalent to the Section 4.1.4 *descending strategy.* It consists of a loop in which the feasibility checker (i) finds a value for the objective function and (ii) attempts to find a lower value by adding an assertion that the objective function must be smaller than that value, e.g. Eén and Sörensson (2006). When the problem becomes infeasible, the last found objective value, if there is one, is the optimal solution.

Unbounded binary search. This method has two main phases:

1. The *bounds search* which establishes initial lower and upper bounds on the objective value. The optimal value is between these bounds.
2. The *bisection phase*, in which the interval between the lower and upper objective bound is split in two equal parts, until the optimal value of the objective is found, relative to the specified accuracy.

Hybrid method. This modifies unbounded binary search so that in each phase a *naive* step decides if the current value is optimal and the method can stop.

4.3.9. Manyopt *experimental results*

`ManyOpt` is not directly comparable to other MINLP solvers in the process systems engineering literature, e.g. ANTIGONE (Misener and Floudas 2014), because it has a different set of capabilities. `ManyOpt` relies on exact, rational arithmetic in an SMT solver, so it is robust to floating point rounding errors. In the Neumaier (2004) classification, `ManyOpt` is a *rigorous* method. Meanwhile, ANTIGONE (Misener and Floudas 2014) relies on floating point arithemtic in a linear programming solver and is therefore a *complete* method (Neumaier 2004), i.e. ANTIGONE assumes exact computations. At this writing, most SMT solvers only support nonlinearity in the form of polynomials. Since ManyOpt relies on the SMT solvers, it does not currently support transcendental functions.

Warm-starting enables what-if scenarios. ManyOpt calls an SMT solver, so it directly inherits the warm-start capabilities of SMT. To see the advantage of warm-starting an MINLP solver, consider prob03 from MINLPLib2:

$$\begin{aligned} \min_{x_1,x_2} \quad & 3 \cdot x_1 + 2 \cdot x_2 \\ & x_1 \cdot x_2 \geq 3.5 \\ & x_1, x_2 \in \{1,2,3,4,5\}. \end{aligned}$$

Solving prob03, ManyOpt accurately gives an objective value of 10 ($x_1 = x_2 = 2$). But what if, after inspecting the solution, we want to also add the constraint $x_1 \leq x_2 - 1$? In ManyOpt, we can immediately add the constraint and, after warm-starting, get the new optimal objective value of 11 ($x_1 = 1, x_2 = 4$). Of course, an alternative (and the only option in most MINLP solvers) is to cold-start a new MINLP problem.

In the case of prob03, ManyOpt uses 37 calls to SMT solver Z3 for the initial solve and 19 additional Z3 calls for the warm-start. For a cold-start of the new problem, ManyOpt requires 39 Z3 calls. So, for prob03, using the warm-starting utility in ManyOpt saves 20 Z3 calls. Of course, SMT solvers use exponential algorithms, so there is no guarantee that warm-starting ManyOpt will improve performance. But the core ManyOpt strategy is to parallelize different approaches to a problem input, so the warm-starting utility offers a possibility of returning more quickly by giving more feature vectors. ManyOpt has more opportunities to benefit from warm-starting if the problem has many integer variables.

Warm-starting enables model exploration. When solving the classical pooling problem haverly in MINLPLib2 with a relative optimality tolerance of 10^{-6}, ManyOpt returns an objective value of -399.9997. The known optimal objective value is -400 and ManyOpt is correct with respect to the tolerance. Investigating a solution, ManyOpt also returns some small values for flows within the network, e.g. $x_3 = 125/33554432, x_6 = x_8 = 1/4096$. As engineers, we may wish to know: (i) are those flow effectively 0? or (ii) will setting those flows to 0 somehow damage the best feasible solution? In ManyOpt, we can set those three variables to 0, i.e. $x_3 = x_6 = x_8 = 0$, and warm-start the solver. The returned solution, with objective value -399.9999, shows that those flows are effectively 0.

Other experimental results. Appendix A gives some experimental results for MINLPLib (Bussieck et al. 2003) benchmarks solved by `ManyOpt`. The results show that `ManyOpt` is an effective MINLP solver and that `ManyOpt` can be used on a wide range of test instances. `ManyOpt` is frequently slower than state-of-the-art MINLP solvers, but it offers the trade-off of a completely different method to address the problem.

5. Conclusion

This manuscript proposes applying satisfiability modulo theories to process systems engineering. We motivate our position using three test beds: (i) two-dimensional bin packing, (ii) model explainers, and (iii) MINLP solvers.

Acknowledgments

The following support is gratefully acknowledged: the EPSRC Centre for Doctoral Training in High Performance Embedded and Distributed Systems to M.M. (EP/L016796/1), EPSRC with a Doctoral Training Fees Award to A.C.D., EPSRC projects EP/N020030/1 and EP/N023242/1 to M.R.H., an EPSRC Research Fellowship to R.M. (EP/P016871/1), and EPSRC project EP/P008739/1 to R.M.

References

Achterberg, T. 2007a. Conflict analysis in mixed integer programming. *Discrete Optim.* **4** 4–20.

Achterberg, T. 2007b. Constraint integer programming. Ph.D. thesis, Technische Universität Berlin.

Achterberg, T. 2009. SCIP: solving constraint integer programs. *Math. Program. Comput.* **1** 1–41.

Aminof, B., O. Kupferman, R. Lampert. 2011. Formal analysis of online algorithms. *Automated Technology for Verification and Analysis, 9th International Symposium, ATVA 2011, Taipei, Taiwan. Proceedings*. 213–227.

Baker, B. S., E. G. Coffman, Jr., R. L. Rivest. 1980. Orthogonal packings in two dimensions. *SIAM J. Comput.* **9** 846–855.

Balas, E. 1974. Intersection cuts from disjunctive constraints. Tech. Rep. Management Sciences Research Report No. 330, Graduate School of Industrial Administration, Carnegie Mellon University.

Balas, E. 1975. Disjunctive programming: Cutting planes from logical conditions. *Nonlinear Programming 2*. Elsevier, 279–312.

Balas, E. 1977. A note on duality in disjunctive programming. *J. Optim. Theory Appl.* **21** 523–528.

Balas, E. 1979. Disjunctive programming. *Ann. Discrete Math.* **5** 3–51.

Barrett, C., C. L. Conway, M. Deters, L. Hadarean, D. Jovanovic, T. King, A. Reynolds, C. Tinelli. 2011. CVC4. *Computer Aided Verification - 23rd International Conference, CAV 2011, Snowbird, UT, USA*. 171–177.

Beaumont, P., N. Evans, M. Huth, T. Plant. 2015. Confidence analysis for nuclear arms control: SMT abstractions of Bayesian belief networks. *Computer Security - ESORICS 2015 - 20th European Symposium on Research in Computer Security, Vienna, Austria.* 521–540.

Bemporad, A., N. Giorgetti. 2004. *SAT-Based Branch & Bound and Optimal Control of Hybrid Dynamical Systems.* Springer Berlin Heidelberg, Berlin, Heidelberg, 96–111.

Bemporad, A., N. Giorgetti. 2006. Logic-based solution methods for optimal control of hybrid systems. *IEEE Transactions on Automatic Control* **51** 963–976.

Benders, J. F. 1962. Partitioning procedures for solving mixed-variables programming problems. *Numerische Mathematik* **4** 238–252.

Berkey, J. O., P. Y. Wang. 1987. Two-dimensional finite bin-packing algorithms. *Journal of the Operational Research Society* **38** 423–429.

Biere, A., A. Cimatti, E. Clarke, Y. Zhu. 1999. *Tools and Algorithms for the Construction and Analysis of Systems: 5th International Conference, TACAS'99*, chap. Symbolic Model Checking without BDDs. Springer Berlin Heidelberg, Berlin, Heidelberg, 193–207.

Biere, A., M. Heule, H. van Maaren, T. Walsh. 2009. *Handbook of Satisfiability: Volume 185 Frontiers in Artificial Intelligence and Applications.* IOS Press, Amsterdam, The Netherlands.

Bjørner, N., L. De Moura. 2011. Satisfiability modulo theories: Introduction and applications. *Commun. ACM* 69–77.

Bjørner, N., A.-D. Phan. 2014. νZ - maximal satisfaction with Z3. *Proceedings of the 6th International Symposium on Symbolic Computation in Software Science, SCSS*, vol. 30. 1–9.

Bjørner, N., A.-D. Phan, L. Fleckenstein. 2015. *Tools and Algorithms for the Construction and Analysis of Systems: 21st International Conference, TACAS 2015*, chap. νZ - An Optimizing SMT Solver. Springer Berlin Heidelberg, 194–199.

Bofill, M., R. Nieuwenhuis, A. Oliveras, E. Rodríguez-Carbonell, A. Rubio. 2008. *Computer Aided Verification: 20th International Conference, CAV 2008*, chap. The Barcelogic SMT Solver. Springer Berlin Heidelberg, 294–298.

Bollapragada, S., O. Ghattas, J. N. Hooker. 2001. Optimal design of truss structures by logic-based branch and cut. *Oper. Res.* **49** 42–51.

Boukouvala, F., R. Misener, C. A. Floudas. 2016. Global optimization advances in mixed-integer nonlinear programming, MINLP, and constrained derivative-free optimization, CDFO. *Eur. J. Oper. Res.* **252** 701–727.

Bruttomesso, R., A. Cimatti, A. Franzén, A. Griggio, R. Sebastiani. 2008. The MathSAT 4SMT solver. *Computer Aided Verification, 20th International Conference, CAV 2008, Princeton, NJ, USA.* 299–303.

Bussieck, M. R., A. S. Drud, A. Meeraus. 2003. MINLPLib—A Collection of Test Models for Mixed-Integer Nonlinear Programming. *INFORMS J. Comput.* **15** 114–119.

Callia D'Iddio, A., M. Huth. 2017. Manyopt: An extensible tool for mixed, non-linear optimization through SMT solving. *CoRR* **abs/1702.01332**.

Carbonneau, R. A., G. Caporossi, P. Hansen. 2011. Globally optimal clusterwise regression by mixed logical-quadratic programming. *Eur. J. Oper. Res.* **212** 213 – 222.

Carbonneau, R. A., G. Caporossi, P. Hansen. 2012. Extensions to the repetitive branch and bound algorithm for globally optimal clusterwise regression. *Comput. Oper. Res.* **39** 2748 – 2762.

Carvajal, R., S. Ahmed, G. Nemhauser, K. Furman, V. Goel, Y. Shao. 2014. Using diversification, communication and parallelism to solve mixed-integer linear programs. *Oper. Res. Lett.* **42** 186–189.

Cimatti, A., A. Griggio, B. J. Schaafsma, R. Sebastiani. 2013. *Tools and Algorithms for the Construction and Analysis of Systems: 19th International Conference, TACAS 2013*, chap. The MathSAT5 SMT Solver. Springer Berlin Heidelberg, 93–107.

Cook, Stephen A. 1971. The complexity of theorem-proving procedures. *Proceedings of the third annual ACM symposium on Theory of computing - STOC '71*. ACM Press, New York, New York, USA, 151–158.

D'Ambrosio, C., A. Frangioni, L. Liberti, A. Lodi. 2012. A storm of feasibility pumps for nonconvex MINLP. *Math. Program.* **136** 375–402.

Davis, M., G. Logemann, D. Loveland. 1962. A machine program for theorem-proving. *Commun. ACM* **5** 394–397.

Davis, M., H. Putnam. 1960. A computing procedure for quantification theory. *J. ACM* **7** 201–215.

de Moura, L., N. Bjørner. 2008a. *Tools and Algorithms for the Construction and Analysis of Systems: 14th International Conference, TACAS 2008*, chap. Z3: An Efficient SMT Solver. Springer Berlin Heidelberg, 337–340.

de Moura, L., N. Bjørner. 2008b. Z3: an efficient SMT solver. *Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS 2008, Budapest, Hungary*. 337–340.

de Moura, L., G. O. Passmore. 2013. Computation in real closed infinitesimal and transcendental extensions of the rationals. M. P. Bonacina, ed., *Automated Deduction - CADE-24*, *Lecture Notes in Computer Science*, vol. 7898. Springer Berlin Heidelberg, 178–192.

Dolan, E. D., J. J. Moré. 2002. Benchmarking optimization software with performance profiles. *Math. Program.* **91** 201–213.

Dutertre, B. 2014. Yices 2.2. *Computer Aided Verification - 26th International Conference, CAV 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, AU*. 737–744.

Dutertre, B., L. de Moura. 2006. *Computer Aided Verification: 18th International Conference, CAV 2006*, chap. A Fast Linear-Arithmetic Solver for DPLL(T). Springer Berlin Heidelberg, 81–94.

Eén, N., N. Sörensson. 2006. Translating pseudo-Boolean constraints into SAT. *JSAT* **2** 1–26.

Fattahi, A., S. Elaoud, E. S. Azer, M. Turkay. 2014. A novel integer programming formulation with logic cuts for the U-shaped assembly line balancing problem. *International J. Production Res.* **52** 1318–1333.

Floudas, C.A. 1995. *Nonlinear and mixed-integer optimization: fundamentals and applications*. Oxford University Press, New York, NY.

Fourer, R., J. Ma, R. K. Martin. 2010. OSiL: An instance language for optimization. *Comp. Opt. Appl.* **45** 181–203.

Ganzinger, H., G. Hagen, R. Nieuwenhuis, A. Oliveras, C. Tinelli. 2004. *Computer Aided Verification: 16th International Conference, CAV 2004*, chap. DPLL(T): Fast Decision Procedures. Springer Berlin Heidelberg, 175–188.

Gario, M., A. Micheli. 2015. PySMT: a solver-agnostic library for fast prototyping of SMT-based algorithms. *In Proc. of the 13th International Workshop on Satisfiability Modulo Theories (SMT)*. 373–384.

Grossmann, I. E., J. P. Ruiz. 2012. Generalized disjunctive programming: A framework for formulation and alternative algorithms for MINLP optimization. J. Lee, S. Leyffer, eds., *Mixed Integer Nonlinear Programming*. Springer New York, New York, NY, 93–115.

Hamadi, Y., S. Jabbour, L. Sais. 2009. ManySAT: A parallel SAT solver. *JSAT* **6**. IOS Press.

Hart, W. E., J.-P. Watson, D. L. Woodruff. 2011. Pyomo: modeling and solving mathematical programs in Python. *Math. Program. Comput.* **3** 219–260.

Hart, William E, Carl Laird, Jean-Paul Watson, David L Woodruff. 2012. *Pyomo–optimization modeling in Python*, vol. 67. Springer Science & Business Media.

Hooker, J. N. 2002. Logic, optimization, and constraint programming. *INFORMS J. Comput.* **14** 295–321.

Hooker, J. N. 2007. Planning and scheduling by logic-based Benders decomposition. *Oper. Res.* **55** 588–602.

Hooker, J. N., M. A. Osorio. 1999. Mixed logical-linear programming. *Discrete Appl. Math.* **96-97** 395–442.

Hooker, J. N., G. Ottoson. 2003. Logic-based Benders decomposition. *Math. Program.* **96** 33–60.

Jain, V., I. E. Grossmann. 2001. Algorithms for Hybrid MILP/CP Models for a Class of Optimization Problems. *INFORMS J. Comput.* **13** 258–276.

Kautz, H., B. Selman. 1992. Planning as satisfiability. *European Conference on Artificial Intelligence*. ECAI '92, John Wiley & Sons, Inc., New York, NY, USA, 359–363.

Lee, S., I. E. Grossmann. 2000. New algorithms for nonlinear generalized disjunctive programming. *Comput. Chem. Eng.* **24** 2125–2141.

Li, H., K. Womer. 2008. Scheduling projects with multi-skilled personnel by a hybrid MILP/CP Benders decomposition algorithm. *J. Sched.* **12** 281–298.

Li, Y., A. Albarghouthi, Z. Kincaid, A. Gurfinkel, M. Chechik. 2014. Symbolic optimization with SMT solvers. *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages - POPL '14*. ACM Press, New York, New York, USA, 607–618.

Lodi, A., S. Martello, D. Vigo. 1999. Heuristic and metaheuristic approaches for a class of two-dimensional bin packing problems. *INFORMS J. Comput.* **11** 345–357.

Malik, S., L. Zhang. 2009. Boolean satisfiability from theoretical hardness to practical success. *Commun. ACM* **52** 76.

Manolios, P., V. Papavasileiou. 2013. *Computer Aided Verification: 25th International Conference, CAV 2013*, chap. ILP Modulo Theories. Springer Berlin Heidelberg, 662–677.

Maravelias, C. T. 2006. A decomposition framework for the scheduling of single- and multi-stage processes. *Comput. Chem. Eng.* **30** 407–420.

Maravelias, C. T., I. E. Grossmann. 2004. A hybrid MILP/CP decomposition approach for the continuous time scheduling of multipurpose batch plants. *Comput. Chem. Eng.* **28** 1921–1949.

Maravelias, C. T., C. Sung. 2009. Integration of production planning and scheduling: Overview, challenges and opportunities. *Comput. Chem. Eng.* **33** 1919–1930.

Martello, S., D. Vigo. 1998. Exact solution of the two-dimensional finite bin packing problem. *Management Sci.* **44** 388–399.

Misener, R., C. A. Floudas. 2014. ANTIGONE: Algorithms for coNTinuous Integer Global Optimization of Nonlinear Equations. *J. Glob. Optim.* **59** 503–526.

Narodytska, N., F. Bacchus. 2014. Maximum satisfiability using core-guided MaxSAT resolution. *AAAI Conference on Artificial Intelligence*. 2717–2723.

Nelson, G., D. C. Oppen. 1979. Simplification by cooperating decision procedures. *ACM T. Program. Lang. Syst.* **1** 245–257.

Nelson, G., D. C. Oppen. 1980. Fast decision procedures based on congruence closure. *J. ACM* **27** 356–364.

Nemhauser, G. L., L. A. Wolsey. 1988. *Integer and Combinatorial Optimization*. Wiley-Interscience, New York, NY, USA.

Neumaier, A. 2004. Complete search in continuous global optimization and constraint satisfaction. *Acta Numerica* **13** 271–369.

Nieuwenhuis, R., A. Oliveras. 2006. *Theory and Applications of Satisfiability Testing - SAT 2006*, chap. On SAT Modulo Theories and Optimization Problems. Springer Berlin Heidelberg, 156–169.

Papageorgiou, D. J., F. Trespalacios. 2016. Pseudo basic steps: Bound improvement guarantees from Lagrangian decomposition in convex disjunctive programming URL `http://www.optimization-online.org/DB_FILE/2016/09/5627.pdf`.

Park, M., S. Park, F. D. Mele, I. E. Grossmann. 2006. Modeling of purchase and sales contracts in supply chain optimization. *2006 SICE-ICASE International Joint Conference*. 5727–5732.

Pisinger, D., M. Sigurd. 2007. Using decomposition techniques and constraint programming for solving the two-dimensional bin-packing problem. *INFORMS J. Comput.* **19** 36–51.

Raman, R., I. E. Grossmann. 1994. Modelling and computational techniques for logic based integer programming. *Comput. Chem. Eng.* **18** 563–578.

Raman, V., N. Piterman, H. Kress-Gazit. 2013. Provably correct continuous control for high-level robot behaviors with actions of arbitrary execution durations. *2013 IEEE International Conference on Robotics and Automation, Karlsruhe, Germany*. 4075–4081.

Rodriguez, M. A., A. Vecchietti. 2009. Logical and generalized disjunctive programming for supplier and contract selection under provision uncertainty. *Ind. Eng. Chem. Res.* **48** 5506–5521.

Ruiz, J. P., J.-H. Jagla, I. E. Grossmann, A. Meeraus, A. Vecchietti. 2012. *Algebraic Modeling Systems: Modeling and Solving Real World Optimization Problems*, chap. Generalized Disjunctive Programming: Solution Strategies. Springer Berlin Heidelberg, Berlin, Heidelberg, 57–75.

Sawaya, N. W., I. E. Grossmann. 2005. A cutting plane method for solving linear generalized disjunctive programming problems. *Comput. Chem. Eng.* **29** 1891–1913.

Sebastiani, R. 2007. Lazy Satisfiability Modulo Theories. *Journal on Satisfiability, Boolean Modeling and Computation* **3** 141–224.

Sebastiani, R., S. Tomasi. 2015. Optimization modulo theories with linear rational costs. *ACM T. Comput. Log. (TOCL)* **16** 1–43.

Sebastiani, R., P. Trentin. 2015. *Computer Aided Verification: 27th International Conference, CAV 2015*, chap. OptiMathSAT: A Tool for Optimization Modulo Theories. Springer International Publishing, 447–454.

Shostak, R. E. 1979. A practical decision procedure for arithmetic with function symbols. *J. ACM* **26** 351–360.

Shostak, R. E. 1982. *6th Conference on Automated Deduction: New York, USA*, chap. Deciding combinations of theories. Springer Berlin Heidelberg, Berlin, Heidelberg, 209–222.

Silva, J. P. M., K. A. Sakallah. 1996. GRASP: a new search algorithm for satisfiability. *IEEE/ACM International Conference on Computer-aided Design*. ICCAD '96, IEEE Computer Society, Washington, DC, USA, 220–227.

Sitek, P. 2014. A hybrid CP/MP approach to supply chain modelling, optimization and analysis. *Computer Science and Information Systems (FedCSIS)*. 1345–1352.

Thorsteinsson, E. S. 2001. Branch-and-check: A hybrid framework integrating mixed integer programming and constraint logic programming. *Principles and Practice of Constraint Programming - CP 2001* **2239** 16–30.

Trespalacios, F., I. E. Grossmann. 2014. Review of mixed-integer nonlinear and generalized disjunctive programming methods. *Chemie Ingenieur Technik* **86**.

Trespalacios, F., I. E. Grossmann. 2016. Cutting plane algorithm for convex generalized disjunctive programs. *INFORMS J. Comput.* **28** 209–222.

Türkay, M., I. E. Grossmann. 1996. Logic-based MINLP algorithms for the optimal synthesis of process networks. *Comput. Chem. Eng.* **20** 959–978.

Vecchietti, A., S. Lee, I. E. Grossmann. 2003. Modeling of discrete/continuous optimization problems: Characterization and formulation of disjunctions and their relaxations. *Comput. Chem. Eng.* **27** 433–448.

Zhang, H. 2002. Generating college conference basketball schedules by a SAT solver. *Proceedings of the Fifth International Symposium on the Theory and Applications of Satisfiability Testing*. 281–291.

Appendix A. `ManyOpt`Statistics for Solved Benchmarks

Statistics for solved benchmarks

Benchmark	Type	#Var	#Con	Found objective	Solved by	#Iter	Solving time	Total time
gear	INLP	5	1	0.0007	allinone.ubs	48	0.19	0.28
nvs04	INLP	3	1	0.72	allinone.ubs	55	0.32	0.38
nvs06	INLP	3	1	1.7703	onebyone.nve	28	0.40	0.47
nvs07	INLP	4	3	4.0	allinone.hyb	37	0.23	0.29
nvs16	INLP	3	1	0.7031	allinone.nve	36	0.24	0.28
prob02	IQCP	7	9	112235.0	bin_flat.hyb	36	17.71	17.76
prob03	IQCP	3	2	10.0	bin_flat.ubs	5	0.00	0.03
nvs03	IQCQP	3	3	16.0	bin_flat.nve	4	0.02	0.06
nvs10	IQCQP	3	3	-310.8	bin_flat.hyb	12	0.13	0.19
nvs11	IQCQP	4	4	-431.0	bin_flat.nve	9	2.71	2.72
nvs12	IQCQP	5	5	-481.2	bin_flat.nve	12	992.20	999.18
nvs15	IQP	4	2	1.0	bin_flat.ubs	4	0.02	0.03
st_miqp2	IQP	5	4	2.0	allinone.hyb	43	0.31	0.38
st_miqp3	IQP	3	2	-6.0	nobb.ubs	14	0.02	0.06
st_test1	IQP	6	2	-32.0059	bin_flat.ubs	20	0.11	0.14
st_test2	IQP	7	3	-9.25	bin_flat.nve	3	26.04	26.07
st_test3	IQP	14	11	-7.0	bin_flat.ubs	6	41.77	41.93
st_test4	IQP	7	6	-7.0	nobb.ubs	14	0.06	0.08
st_testph4	IQP	4	11	-80.5	bin_flat.nve	6	0.03	0.06
ex1221	MBNLP	6	6	7.6671	bin_flat.hyb	4	0.13	0.14
ex1226	MBNLP	6	6	-17.0	bin_flat.hyb	2	0.02	0.03
gear2	MBNLP	29	5	0.0003	allinone.nve	10	1.35	1.45
st_e15	MBNLP	6	6	7.6671	bin_flat.hyb	4	0.12	0.15
autocorr_bern20-03	MBQCP	22	2	-72.0	bin_flat.nve	74	0.35	0.41
autocorr_bern25-03	MBQCP	27	2	-92.0	bin_flat.nve	94	3.65	3.78
sporttournament06	MBQCP	17	2	12.0	bin_flat.hyb	12	0.05	0.07
st_e13	MBQCP	3	3	2.0	bin_flat.nve	2	0.01	0.03
alan	MBQP	9	8	2.9252	bin_flat.nve	7	0.06	0.08
gbd	MBQP	5	5	2.2	allinone.ubs	12	0.00	0.03
st_e27	MBQP	5	7	2.0	bin_flat.hyb	4	0.14	0.16
gear3	MINLP	9	5	0.0003	allinone.nve	50	0.29	0.37
gear4	MINLP	7	2	1.6434	bin_flat.hyb	52	18.97	19.37
nvs01	MINLP	4	4	12.4696	allinone.hyb	92	1.73	1.87
nvs21	MINLP	4	3	-5.6844	onebyone.ubs	157	1.78	1.96
st_e38	MINLP	5	4	7197.7277	allinone.ubs	56	2.37	2.48

Benchmark	Type	#Var	#Con	Found objective	Solved by	#Iter	Solving time	Total time
st_e40	MINLP	5	9	30.4142	bin_flat.nve	5	0.62	0.67
tln2	MIQCP	9	13	5.3	bin_flat.hyb	8	0.06	0.08
chance	NLP	5	4	29.8946	nobb.ubs	14	0.24	0.26
ex14_1_1	NLP	4	5	0.0009	nobb.ubs	13	1.47	1.49
ex14_1_5	NLP	7	7	0.0	nobb.ubs	6	0.02	0.04
ex4_1_1	NLP	2	1	-7.4867	nobb.hyb	4	0.02	0.04
ex4_1_3	NLP	2	1	-443.6709	nobb.nve	12	0.08	0.12
ex4_1_4	NLP	2	1	0.0	nobb.nve	6	0.02	0.05
ex4_1_5	NLP	3	1	0.0	nobb.ubs	31	0.15	0.18
ex4_1_6	NLP	2	1	7.0	nobb.nve	3	0.00	0.03
ex4_1_7	NLP	2	1	-7.5	nobb.nve	5	0.02	0.04
ex4_1_8	NLP	3	2	-16.7388	nobb.nve	5	0.03	0.05
ex4_1_9	NLP	3	3	-5.5078	nobb.hyb	16	0.06	0.11
ex7_2_2	NLP	7	6	-0.3885	nobb.nve	12	123.07	123.11
ex7_3_1	NLP	5	8	0.3427	nobb.nve	60	5.57	5.67
ex7_3_2	NLP	5	8	1.0903	nobb.ubs	15	0.11	0.14
ex8_1_3	NLP	3	1	3.0	nobb.hyb	10	0.28	0.30
ex8_1_4	NLP	3	1	0.0	nobb.hyb	2	0.00	0.03
ex8_1_5	NLP	3	1	-1.0312	nobb.ubs	12	0.18	0.21
ex8_1_6	NLP	3	1	-10.086	nobb.nve	5	0.08	0.11
mathopt1	NLP	3	3	0.0	nobb.nve	18	0.09	0.12
mathopt2	NLP	3	5	0.0	nobb.hyb	2	0.01	0.02
mathopt5_4	NLP	2	1	0.0	nobb.hyb	4	0.02	0.04
mathopt5_7	NLP	2	1	-4.4365	nobb.nve	11	0.08	0.10
mathopt5_8	NLP	2	1	-0.6857	nobb.nve	6	0.03	0.05
prob09	NLP	4	2	0.0	nobb.nve	4	0.02	0.04
rbrock	NLP	3	1	0.0	nobb.ubs	42	0.19	0.24
st_e06	NLP	4	4	0.0	nobb.hyb	2	0.01	0.02
st_e11	NLP	4	3	189.3116	nobb.nve	9	1.69	1.72
st_e12	NLP	5	4	-4.5139	nobb.ubs	16	1.86	1.88
st_e17	NLP	3	2	376.2924	nobb.ubs	18	0.07	0.10
st_e19	NLP	3	3	-118.7046	nobb.ubs	22	0.18	0.20
circle	QCP	4	11	4.5751	nobb.ubs	13	3.45	3.48
ex3_1_1	QCP	9	7	7049.2485	nobb.ubs	25	0.31	0.33
ex3_1_4	QCP	4	4	-3.9997	nobb.ubs	14	0.16	0.19
ex5_2_2_case1	QCP	10	7	-399.9995	nobb.ubs	28	0.48	0.50

Benchmark	Type	#Var	#Con	Found objective	Solved by	#Iter	Solving time	Total time
ex5_2_2_case2	QCP	10	7	-599.9997	nobb.ubs	30	0.42	0.43
ex5_2_2_case3	QCP	10	7	-749.9996	nobb.ubs	30	0.47	0.52
ex5_3_2	QCP	23	17	1.8647	nobb.ubs	12	52.57	52.60
ex5_4_2	QCP	9	7	7512.2304	nobb.ubs	20	0.41	0.45
ex9_1_1	QCP	14	13	-13.0	nobb.nve	9	0.00	0.03
ex9_1_2	QCP	11	10	-16.0	nobb.nve	4	0.00	0.03
ex9_1_4	QCP	11	10	-37.0	nobb.nve	3	0.02	0.04
ex9_1_5	QCP	14	13	-1.0	nobb.hyb	4	0.00	0.02
ex9_1_8	QCP	15	13	-3.25	nobb.nve	4	0.01	0.02
ex9_2_3	QCP	17	16	0.0002	nobb.ubs	18	0.36	0.38
haverly	QCP	13	10	-399.9997	nobb.ubs	28	0.58	0.62
house	QCP	9	9	-4499.9999	nobb.ubs	32	0.98	1.03
pointpack02	QCP	6	4	1.9999	nobb.ubs	14	0.17	0.18
st_e01	QCP	3	2	-6.666	nobb.ubs	16	253.42	253.46
st_e02	QCP	4	4	201.1593	nobb.nve	2	0.04	0.06
st_e05	QCP	6	4	7049.2495	nobb.ubs	34	0.83	0.85
st_e07	QCP	11	8	-399.9997	nobb.hyb	53	205.10	205.15
st_e08	QCP	3	3	0.7419	nobb.ubs	12	0.06	0.09
st_e18	QCP	3	5	-2.8283	nobb.ubs	12	0.06	0.08
st_e30	QCP	15	16	-1.581	nobb.ubs	14	23.73	23.75
st_e33	QCP	10	7	-399.9997	nobb.ubs	28	193.08	193.14
st_e34	QCP	7	5	0.0156	nobb.nve	6	0.02	0.03
wastewater02m1	QCP	20	15	130.7026	nobb.nve	97	309.32	309.38
dispatch	QCQP	5	3	3155.2883	nobb.nve	69	38.64	38.77
ex5_2_4	QCQP	8	7	-449.9992	nobb.ubs	28	0.76	0.79
ex9_2_4	QCQP	9	8	0.5	nobb.nve	3	0.02	0.04
ex9_2_6	QCQP	17	13	-0.9992	nobb.ubs	12	0.31	0.34
ex9_2_8	QCQP	7	6	1.5	nobb.ubs	3	0.01	0.03
st_e09	QCQP	3	2	-0.499	nobb.ubs	10	0.04	0.06
ex2_1_3	QP	14	10	-14.999	nobb.hyb	33	19.71	19.77
ex2_1_4	QP	7	6	-11.0	nobb.nve	3	0.01	0.03
immun	QP	22	8	0.0	nobb.ubs	76	24.21	24.26
nemhaus	QP	6	6	31.0	nobb.hyb	2	0.01	0.02
sambal	QP	18	11	3.9686	nobb.ubs	30	219.76	219.78
st_bpaf1b	QP	11	11	-42.9625	nobb.ubs	22	0.35	0.39
st_bpk1	QP	5	7	-12.9994	nobb.ubs	18	0.13	0.14

Benchmark	Type	#Var	#Con	Found objective	Solved by	#Iter	Solving time	Total time
st_bpv1	QP	5	5	10.0	nobb.ubs	25	0.00	0.04
st_bpv2	QP	5	6	-7.9994	nobb.ubs	18	0.06	0.08
st_bsj2	QP	4	6	1.0005	nobb.hyb	16	0.40	0.41
st_bsj3	QP	7	2	-86768.5498	nobb.ubs	44	5.15	5.22
st_cqpf	QP	5	7	-2.75	nobb.hyb	4	0.01	0.03
st_cqpjk2	QP	4	2	-12.499	nobb.ubs	13	0.06	0.08
st_e22	QP	3	6	-84.9998	nobb.ubs	20	0.11	0.14
st_e25	QP	5	9	0.8908	nobb.nve	75	35.62	35.69
st_e26	QP	3	5	-185.7792	nobb.hyb	18	0.11	0.12
st_glmp_fp1	QP	5	9	10.0	nobb.ubs	18	453.69	453.75
st_glmp_fp3	QP	5	9	-12.0	nobb.hyb	4	0.01	0.02
st_glmp_kk90	QP	6	8	3.0001	nobb.ubs	16	405.05	405.09
st_glmp_kk92	QP	5	9	-12.0	nobb.hyb	6	0.04	0.05
st_glmp_kky	QP	8	14	-2.4998	nobb.nve	46	827.40	827.46
st_glmp_ss1	QP	6	12	-24.5705	nobb.nve	115	901.45	901.52
st_ht	QP	3	4	-1.5997	nobb.ubs	14	0.08	0.10
st_pan1	QP	4	5	-5.2831	nobb.ubs	16	0.44	0.48
st_ph11	QP	4	5	-11.2807	nobb.ubs	18	0.15	0.19
st_ph12	QP	4	5	-22.6246	nobb.ubs	20	0.19	0.21
st_ph13	QP	4	11	-11.2807	nobb.ubs	18	0.21	0.23
st_ph14	QP	4	11	-229.7221	nobb.ubs	26	0.56	0.60
st_ph3	QP	7	6	-420.2347	nobb.hyb	52	423.17	423.25
st_phex	QP	3	6	-84.9996	nobb.ubs	17	0.09	0.11
st_qpk1	QP	3	5	-2.9995	nobb.ubs	16	0.07	0.12
st_z	QP	4	6	0.0	nobb.nve	3	0.03	0.06
graphpart_2pm-0044-0044	BQP	49	17	-13.0	bin_flat.nve	12	178.03	178.18
fac1	MBNLP	23	19	160912612.351	bin_flat.ubs	34	9.36	9.40
elf	MBQCP	55	39	0.1916	bin_flat.hyb	20	64.50	64.54
sporttournament08	MBQCP	30	2	24.0	bin_flat.hyb	19	1035.19	1035.30
ex1263a	MIQCP	25	36	19.6	bin_flat.hyb	8	2.20	2.27
ex1264a	MIQCP	25	36	8.6	bin_flat.hyb	7	1.37	1.42
ex1265a	MIQCP	36	45	10.3	bin_flat.nve	9	1.23	1.28
tln4	MIQCP	25	25	8.3	bin_flat.hyb	6	27.37	27.43
tln5	MIQCP	36	31	10.3	bin_flat.nve	12	1532.82	1539.89
ex14_2_5	NLP	5	6	0.0002	nobb.ubs	13	7.74	7.80
ex4_1_2	NLP	2	1	-663.5	nobb.ubs	18	3.10	3.13

Benchmark	Type	#Var	#Con	Found objective	Solved by	#Iter	Solving time	Total time
linear	NLP	25	21	89.0006	nobb.nve	595	4.28	4.68
kall_congruentcircles_c41	QCP	13	25	0.8584	nobb.hyb	2	0.32	0.35
prolog	QCQP	21	23	0.0	nobb.ubs	27	1028.42	1028.49
st_qpc-m3c	QP	11	11	0.0	nobb.hyb	4	0.54	0.57
st_qpc-m4	QP	11	11	0.0	nobb.nve	3	0.62	0.64
hmittelman	BNLP	17	8	13.0	bin_flat.hyb	2	0.04	0.06
graphpart_2g-0044-1601	BQP	49	17	-954077.0	bin_flat.nve	28	773.26	773.45
ex1263	MBQCP	93	56	19.6	bin_flat.nve	12	1.67	1.69
ex1264	MBQCP	89	56	8.6	bin_flat.nve	8	0.41	0.45
nous1	MBQCQP	51	44	None	allinone.nve	1	0.01	0.03
nous2	MBQCQP	51	44	None	allinone.nve	1	0.01	0.02
hybriddynamic_fixed	MBQP	72	80	1.4737	bin_flat.nve	2	0.01	0.04
ex1266a	MIQCP	49	54	16.3	bin_flat.hyb	16	19.50	19.55
tloss	MIQCP	49	54	16.3	bin_flat.nve	19	20.15	20.41
tltr	MIQCP	49	55	48.0666	bin_flat.nve	9	8.82	8.88
fac2	MBNLP	67	34	331837498.177	bin_flat.ubs	48	1022.48	1022.65
ex1265	MBQCP	131	75	10.3	bin_flat.ubs	16	0.72	0.75
ex1266	MBQCP	181	96	16.3	bin_flat.hyb	2	3.05	3.27
autocorr_bern20-05	MBNLP	22	2	-416.0	bin_flat.nve	418	193.18	193.63
fac3	MBQP	67	34	31982309.8485	bin_flat.ubs	46	41.29	41.40
portfol_roundlot	MINLP	18	12	None	allinone.ubs	1	0.02	0.04
edgecross10-010	MBQCP	92	482	1.0	bin_flat.nve	13	1.66	1.71
edgecross10-020	MBQCP	92	482	11.0	bin_flat.ubs	24	1734.18	1734.31