Using Functional Programming to Recognize Named Structure in an Optimization Problem: Application to Pooling

Francesco Ceccon, Georgia Kouyialis, and Ruth Misener
Dept. of Computing, Imperial College London, South Kensington SW7 2AZ, UK

DOI 10.1002/aic.15308
Published online May 24, 2016 in Wiley Online Library (wileyonlinelibrary.com)

Branch-and-cut optimization solvers typically apply generic algorithms, e.g., cutting planes or primal heuristics, to expedite performance for many mathematical optimization problems. But solver software receives an input optimization problem as vectors of equations and constraints containing no structural information. This article proposes automatically detecting named special structure using the pattern matching features of functional programming. Specifically, we deduce the industrially-relevant nonconvex nonlinear Pooling Problem within a mixed-integer nonlinear optimization problem and show that we can uncover pooling structure in optimization problems which are not pooling problems. Previous work has shown that preprocessing heuristics can find network structures; we show that we can additionally detect nonlinear pooling patterns. Finding named structures allows us to apply, to generic optimization problems, cutting planes or primal heuristics developed for the named structure. To demonstrate the recognition algorithm, we use the recognized structure to apply primal heuristics to a test set of standard pooling problems. © 2016 The Authors AIChE Journal published by Wiley Periodicals, Inc. on behalf of American Institute of Chemical Engineers *AIChE J*, 62: 3085–3095, 2016
Keywords: optimization, automatic problem recognition, pooling problem, process networks optimization, functional programming

Introduction and Literature Review

The pooling problem is an industrially-relevant nonconvex nonlinear optimization problem minimizing cost on a feed-forward network of input nodes, intermediate storage or *pool* nodes, and output nodes.[1] Variants of the pooling problem have applications including[2]: crude oil scheduling,[3–6] water networks,[7] natural gas production,[8,9] fixed-charge transportation with product blending,[10] hybrid energy systems,[11] and multi-period blend scheduling.[12] The difficulty of the pooling problem arises from the nonconvex bilinear terms representing linear blending in the pools. These bilinear terms, which are necessary for tracking quality across the network, typically multiply flow rates and concentrations or flow fractions. It is possible to integrate additional complexity into the pooling problem, e.g., allowing mutable topological decisions[13,14] or nonlinear blending rules.[15] A wide variety of pooling variants with generic process networks applications can be found in MINLPLib.[16]

In solving process networks optimization problems there is a common theme: it is much easier to solve large-scale instantiations of the standard, archetypal pooling problem than it is to solve variants including mutable topology, nonlinear blending, or temporal aspects. For example, a recent primal heuristic performs consistently well on the order of 10k variables and constraints,[17] but the approach exploits the standard pooling network structure and does not apply to pooling variants. Other recent advances specially designed for the standard pooling problem include: developing cutting planes,[18,19] discretizing the problem,[20,21] and analyzing computational complexity.[22,23] All of these approaches require knowledge of global problem structure including network topology and node identity.

Previous work in mixed-integer optimization (MIP) has found that using network structure can significantly help generate strong cutting planes.[24] Automatically identifying these embedded networks in large-scale optimization models is NP-hard,[25] but there exist several polynomial time approximation algorithms to find good networks.[25–27] State-of-the-art MIP solver software, including MINTO[28] and CPLEX use preprocessing heuristics to automatically find these network patterns. More recent work has considered detecting more complex structures such as multi-commodity flow[29] and recovering variable meaning from flat MIP structures.[30]

Due to nonlinearities, deterministic global optimization solvers do not currently infer pooling network structure and therefore will solve the pooling problem and its variants via branch-and-cut on the McCormick[31] bilinear convex hull.[32–38] Recent advances in solving pooling problems are therefore unavailable to solver technology; without automatically deducing global network structure, the solvers cannot incorporate primal heuristics or cutting planes based on process network structure. But, for generic process networks problems,

Additional Supporting Information may be found in the online version of this article.

Correspondence concerning this article should be addressed to R. Misener at r.misener@imperial.ac.uk.

The copyright line in this article was changed on 16 June 2016 after original online publication.

Table 1. Notation Used in Standard Pooling Problem Formulations

Type	Name	Description
Indices	$i \in \{1, \ldots, I\}$	Input streams (raw materials or feed stocks)
	$l \in \{1, \ldots, L\}$	Pools (blending facilities)
	$j \in \{1, \ldots, J\}$	Output streams (end products)
	$k \in \{1, \ldots, K\}$	Attributes (qualities monitored)
Variables	ξl	Flow from input i to pool l
	$q_{i,l}$	Flow from input i to pool l, as a fraction of total flow into l
	$v_{i,l,j}$	Flow from input i to output j through pool l ($v_{i,l,j}=q_{i,l}\,y_{l,j}$)
	$y_{l,j}$	Flow from pool l to output j
	$z_{i,j}$	Bypass flow from input i to output j
	$p_{l,k}$	Level of quality attribute k in pool l
Parameters	c_i	Unit cost of raw material feed stock i
	d_j	Unit revenue of product j
	$A_i^L - A_i^U$	Availability bounds of input i
	S_l	Capacity of pool l
	$D_j^L - D_j^U$	Demand bounds for product j
	$C_{i,k}$	Level of quality k in raw material feed stock i
	$P_{j,k}^L - P_{j,k}^U$	Acceptable composition range of quality k in product j

even locally-relevant, single equation reformulation toward a pooling problem may expedite deterministic global optimization solvers.[39–41] Solver ANTIGONE[37,38] will, in preprocessing, eliminate variables, disaggregate bilinear terms, and add Reformulation-Linearization Technique equations. Although these local transformations never consider the whole optimization problem, global solvers ANTIGONE, BARON, COUENNE, and LINDO can solve an additional 10% of the MINLPLib2 process networks problems after applying the ANTIGONE preprocessing.[41] Other best practices for solving process networks problems with off-the-shelf software includes asserting mass balances with the Reformulation-Linearization Technique,[42–45] disaggregating bilinear terms,[46] and developing cuts from disaggregated bilinear terms.[47,48]

This manuscript proposes automatically recognizing pooling structure within a mixed integer nonlinear optimization problem (MINLP). The pooling structure inside of a generic process optimization problem is a subset of the entire problem, so specialized, pooling-specific, cutting planes will also be valid bounds for the entire process networks problem. The Results section of this article shows that primal heuristic solutions to the standard pooling problem would be a good starting point for primal heuristics for the entire optimization problem. The wider goal may be stated as *detecting named structures within an MINLP optimization problem*; Liberti[49] posed this broader challenge to researchers after seeing this work presented at the Oberwolfach MINLP workshop.

Identifying pooling problem structure hinges on pattern matching. Patterns are defined by which variables and coefficients are expected in a constraint and by constraint bounds. The implementation is in F#, a strongly typed functional programming language targeting the .NET runtime environment. Pattern matching, one of the most distinctive F# features, has many uses, from decomposing data to control flow. The core concept is defining how data is expected to look and acting accordingly.

This manuscript begins by defining the standard pooling problem and introduces a canonical PQ-formulation. We propose the canonical formulation to avoid developing algorithms specialized to each known formulation since there are many similar formulations.[1,43,44,50–52] The recognition algorithm simultaneously converts the MINLP to canonical pooling form and recognizes network structure. The article describes the algorithm implementation; the online supplementary material justifies why the algorithm is implemented in F#. After presenting the results, the article concludes by generalizing the work to other optimization problems.

Statement of the Archetypal Standard Pooling Problem

This section defines the standard pooling problem using the Table 1 notation.[2] The first formulation, due to Haverly,[1] is the P-formulation:

$$\min_{\boldsymbol{x},\boldsymbol{y},\boldsymbol{z},\boldsymbol{p}} \sum_{i\in I,\, l\in L} c_i\, x_{i,l} - \sum_{l\in L,\, j\in J} d_j\, y_{l,j} - \sum_{i\in I,\, j\in J} (d_j - c_i)\, z_{i,j}$$

such that

$$\text{Material Capacities} \begin{cases} A_i^L \le \sum_{l\in L} x_{i,l} + \sum_{j\in J} z_{i,j} \le A_i^U & \forall i \in I \\ \sum_{j\in J} y_{l,j} \le S_l & \forall l \in L \\ D_j^L \le \sum_{l\in L} y_{l,j} + \sum_{i\in I} z_{i,j} \le D_j^U & \forall j \in J \end{cases}$$

$$\text{Material Balances} \begin{cases} \sum_{i\in I} x_{i,l} - \sum_{j\in J} y_{l,j} = 0 & \forall l \in L \\ \sum_{i\in I} C_{i,k}\, x_{i,l} - p_{l,k} \sum_{j\in J} y_{l,j} = 0 & \forall l \in L,\, k \in K \end{cases}$$

$$\text{Product Quality} \begin{cases} \sum_{l\in L} p_{l,k}\, y_{l,j} + \sum_{i\in I} C_{i,k}\, z_{i,j} \ge P_{j,k}^L \left(\sum_{l\in L} y_{l,j} + \sum_{i\in I} z_{i,j}\right) & \forall j \in J,\, k \in K \\ \sum_{l\in L} p_{l,k}\, y_{l,j} + \sum_{i\in I} C_{i,k}\, z_{i,j} \le P_{j,k}^U \left(\sum_{l\in l} y_{l,j} + \sum_{i\in I} z_{i,j}\right) & \forall j \in J,\, k \in K \end{cases} \tag{P}$$

Ben-Tal et al.[50] develop a mathematically equivalent Q-formulation that introduces *fractional* flow rates $q_{i,l}=x_{i,l}/\sum_{\hat{i}\in I} x_{\hat{i},l}$ for arcs between an input i and pool l:

$$\min_{\boldsymbol{y},\boldsymbol{z},\boldsymbol{q}} \sum_{i\in I,\, l\in L,\, j\in J} c_i\, q_{i,l}\, y_{l,j} - \sum_{l\in L,\, j\in J} d_j\, y_{l,j} - \sum_{i\in I,\, j\in J} (d_j - c_i)\, z_{i,j}$$

such that

$$\text{Material Capacities} \begin{cases} A_i^L \le \sum_{l,j} q_{i,l}\, y_{l,j} + \sum_{j\in J} z_{i,j} \le A_i^U & \forall i \in I \\ \sum_{j\in J} y_{l,j} \le S_l & \forall l \in L \\ D_j^L \le \sum_{l\in L} y_{l,j} + \sum_{i\in I} z_{i,j} \le D_j^U & \forall j \in J \end{cases}$$

$$\text{Product Quality} \begin{cases} \sum_{i,l} C_{i,k}\, q_{i,l}\, y_{l,j} + \sum_{i\in I} C_{i,k}\, z_{i,j} \ge P_{j,k}^L \left(\sum_{l\in L} y_{l,j} + \sum_{i\in I} z_{i,j}\right) & \forall j \in J,\, k \in K \\ \sum_{i,l} C_{i,k}\, q_{i,l}\, y_{l,j} + \sum_{i\in I} C_{i,k}\, z_{i,j} \le P_{j,k}^U \left(\sum_{l\in L} y_{l,j} + \sum_{i\in I} z_{i,j}\right) & \forall j \in J,\, k \in K \end{cases}$$

$$\text{Simplex} \quad \left\{ \sum_{i\in I} q_{i,l} = 1 \quad \forall l \in L \right. \tag{Q}$$

The P- and Q formulations have variable bounds:

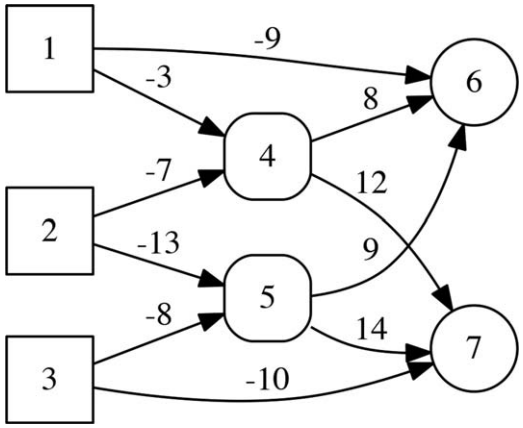


Figure 1. Example of a pooling problem.

The canonical PQ-formulation of this example is illustrated in Figure 11.

$$
\begin{array}{llll}
0 & \le q_{i,l} & \le 1 & \forall i \in I,\ l \in L \\
0 & \le x_{i,l} & \le \min\{A_i^U,\ S_l,\ \sum_{j\in J} D_j^U\} & \forall i \in I,\ l \in L \\
0 & \le y_{l,j} & \le \min\{S_l,\ D_j^U,\ \sum_{i\in I} A_i^U\} & \forall l \in L,\ j \in J \\
0 & \le z_{i,j} & \le \min\{A_i^U,\ D_j^U\} & \forall i \in I,\ j \in J \\
\min_i C_{i,k} & \le p_{l,k} & \le \max_i C_{i,k} & \forall l \in L,\ k \in K
\end{array}
$$

Combining the P- and Q-formulations, Quesada and Grossmann[43] develop the PQ Cut and Tawarmalani and Sahinidis[44] justify its utility. The PQ-formulation is equivalent to the Q-formulation with the addition of the PQ Cut:

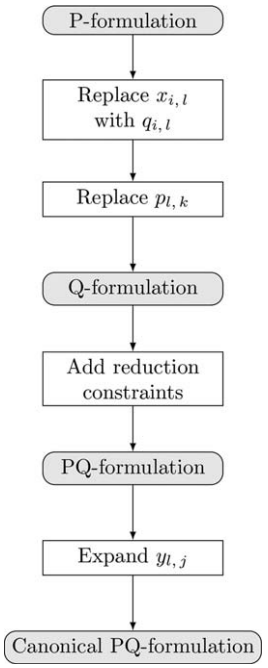


Figure 2. Steps required to create a canonical PQ-formulation when the initial problem is expressed as a P- or Q-formulation; formulations detected via pattern matching are transferred to canonical form.

$$
\sum_{i\in I} q_{i,l}\, y_{l,j} = y_{l,j} \quad \forall l \in L,\ j \in J \qquad \text{(PQ Cut)}
$$

Canonical Representation for the PQ-Formulation

Beyond the P-, Q-, and PQ-formulations, there are several other mathematically equivalent pooling problem formulations.[51,52] But we want to automatically translate every pooling problem into just one formulation (here, the proposed canonical PQ-formulation) and save implementation time by specializing methodologies such as cutting planes to just one formulation.

The proposed canonical PQ-formulation is equivalent to a PQ-formulation that introduces auxiliary variables $v_{i,l,j}$ and expands all product terms $q_{i,l}\, y_{l,j}$ into $v_{i,l,j}$.[17] The formulation also uses the PQ Cut to replace each flow $y_{l,j}$ with the relative sum of flows $v_{i,l,j}$ in every equation except for the PQ Cut and the definition $v_{i,l,j} = q_{i,l}\, y_{l,j}$. After rearranging terms, the canonical PQ-formulation becomes:

$$
\min_{\boldsymbol{y},\boldsymbol{z},\boldsymbol{q},\boldsymbol{v}} \sum_{i\in I,\, l\in L,\, j\in J} (c_i - d_j)\, v_{i,l,j} + \sum_{i\in I,\, j\in J} (c_i - d_j)\, z_{i,j}
$$

such that

$$
\begin{array}{ll}
\text{Path Definition} & \left\{ v_{i,l,j} = q_{i,l}\, y_{l,j} \quad \forall i \in I,\ j \in J,\ l \in L \right. \\
\text{Simplex} & \left\{ \sum_{i\in I} q_{i,l} = 1 \quad \forall l \in L \right. \\
\text{Reduction} & \left\{ \sum_{i\in I} v_{i,l,j} = y_{l,j} \quad \forall l \in L,\ j \in J \right. \\
\text{Input Capacities} & \left\{ A_i^L \le \sum_{l\in L,\, j\in J} v_{i,l,j} + \sum_{j\in J} z_{i,j} \le A_i^U \quad \forall i \in I \right. \\
\text{Pool Capacities} & \left\{ \sum_{i\in I,\, j\in J} v_{i,l,j} \le S_l \quad \forall l \in L \right. \\
\text{Output Capacities} & \left\{ D_j^L \le \sum_{i\in I,\, l\in L} v_{i,l,j} + \sum_{i\in I} z_{i,j} \le D_j^U \quad \forall j \in J \right. \\
\text{Product Quality} & \left\{ \begin{array}{l} \sum_{i,l} (C_{i,k} - P_{j,k}^L)\, v_{i,l,j} + \sum_{i\in I} (C_{i,k} - P_{j,k}^L)\, z_{i,j} \ge 0 \quad \forall j \in J,\ k \in K \\ \sum_{i,l} (C_{i,k} - P_{j,k}^U)\, v_{i,l,j} + \sum_{i\in I} (C_{i,k} - P_{j,k}^U)\, z_{i,j} \le 0 \quad \forall j \in J,\ k \in K \end{array} \right.
\end{array}
\qquad \text{(PQ)}
$$

The canonical PQ-formulation of the Figure 1 example pooling problem is visualized in Figure 11. Rows represent constraints and columns denote variables. The canonical PQ-formulation enables the pattern matching algorithm by allowing the algorithm to focus on linear terms; bilinear terms only appear in the PQ *Path Definition* constraints. Reformulating products $q_{i,l}\, y_{l,j}$ using $v_{i,l,j}$ implies that identifying pooling constraints and building the network can be completed by primarily analyzing linear equations. Expressing the material capacity and product quality constraints with respect to only variables $\boldsymbol{v}$ and $\boldsymbol{z}$ simplifies the network building step and allows us to deduce parameter values for $C_{i,k}$, $P_{j,k}^L$, $P_{j,k}^U$.

The canonical PQ-formulation would perform badly in a branch-and-bound framework due to possibly weak relaxations and poor interval arithmetic.[34,37,46] But this article aims to uncover structure in preprocessing, so it is sensible to

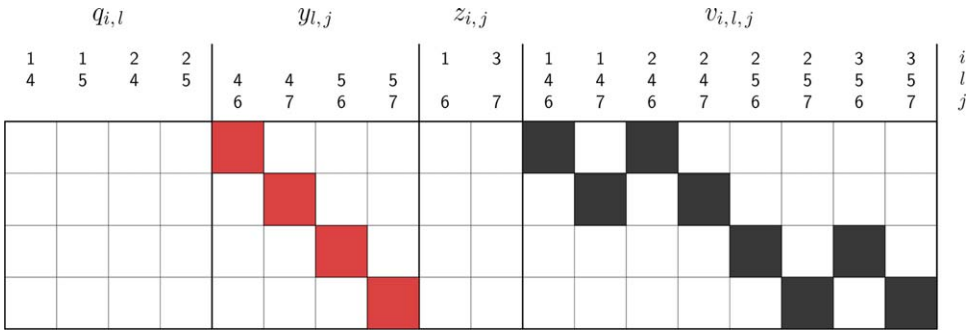


Figure 3. Coefficients in reduction constraints.

[Color figure can be viewed in the online issue, which is available at wileyonlinelibrary.com.]

reformulate to a representation allowing easy network topology discovery. Once the algorithm deduces underlying structure, it may reformulate toward representations designed for whatever algorithm will best solve the problem.

Developing a Pattern Matching Method for Uncovering Pooling Problem Structure

This section describes how to: (1) reformulate a pooling problem to the canonical PQ-formulation, (2) identify the pooling problem constraints in an optimization problem, and (3) build a pooling problem network. We introduce the three steps with respect to *finding patterns*; and subsequently describe implementing the pattern matching algorithm using functional programming. Note that the pattern matching algorithm creates canonical pooling problems at the same time as it identifies pooling problem constraints, so the manuscript sections are coupled.

We take, as input, optimization problems with *flat* structure, that is, optimization problems where the variables and constraints are represented only by positions in a vector and have no associated identifier. The following discussion develops as if *every* constraint and/or variable will fit into some pooling meaning or another. But, under the hood, the implementation splits the MINLP into two parts, a pooling problem and a remaining optimization problem. The algorithm first assumes that every constraint and variable does not belong to the pooling problem and then, if it proves that the constraint or variable fits a known pattern, moves the variable or constraint to the pooling problem.

Converting to the canonical PQ-formulation

Figure 2 outlines steps transforming an optimization problem toward the canonical PQ-formulation. The original problem formulation is not known *a priori*, so when the algorithm finds equations consistent with either a P or Q-formulation, it transforms toward an equivalent PQ-formulation. One difference between the P- and PQ-formulations is that the former uses flow rate variables ξl whereas the latter introduces *fractional* flow rate variables $q_{i,l}$ for arcs between an input i and pool l such that ξl and $q_{i,l}$ are related:

$$x_{i,l} = q_{i,l} \sum_{j \in J} y_{l,j} \quad \forall i \in I,\ l \in L \tag{1}$$

To replace flow rates ξl with $q_{i,l}$, the algorithm first identifies which variables are ξl; it does this by looking for equality constraints with constant coefficient 0 and all linear coefficients equal to either $+1$ or -1. Using the material balance constraints, i.e., $\sum_{i \in I} x_{i,l} - \sum_{j \in J} y_{l,j} = 0\ \forall l \in L$, it labels variables with $+1$ and -1 coefficients as ξl and $y_{l,j}$, respectively. Then it introduces new variables $q_{i,l}$ for each variable ξl and replaces each ξl using Eq. 1.

The algorithm subsequently eliminates variables $p_{l,k}$ by noticing that, if it replaces ξl in the Eq. 2 quality balance constraints, it can map each $p_{l,k}$ to the equivalent $\sum_{i \in I} C_{i,k}\, q_{i,l}$:

$$\begin{aligned} p_{l,k} \sum_{j \in J} y_{l,j} = \quad & \sum_{i \in I} C_{i,k}\, x_{i,l} \\ & \sum_{i \in I} C_{i,k} \left(q_{i,l} \sum_{j \in J} y_{l,j} \right) \forall l \in L, k \in K \\ & \left(\sum_{i \in I} C_{i,k}\, q_{i,l} \right) \sum_{j \in J} y_{l,j} \end{aligned} \tag{2}$$

The Q-formulation is attained by replacing variables $p_{l,k}$ with the relevant $\sum_{i \in I} C_{i,k}\, q_{i,l}$. From the Q-formulation, we obtain the PQ-formulation by taking all bilinear terms $q_{i,l}\, y_{l,j}$ and, for each $y_{l,j}$, adding the PQ Cut; this step is similar to existing algorithms.[37,39,47,48] The last step in transforming to a canonical PQ-formulation is replacing, using the PQ Cut, each occurrence of $y_{l,j}$ in the material capacity and product quality constraints with the sum $\sum_{i \in I} v_{i,l,j}$.

Identifying pooling problem constraints

Coupled to transforming the optimization problem into the canonical PQ-formulation is the challenge of understanding each constraint and variable with respect to the pooling problem. This identification step enables the network extraction. Similar to the previous section, constraint and variable identification focuses on pattern matching. We show how to use the constraints to gain more problem information.

Identifying Reduction and Path Definition Constraints. . Path Definitions, $v_{i,l,j} = q_{i,l}\, y_{l,j}$, are easy to find. The algorithm uses the Path Definitions to find which $y_{l,j}$ is assigned to each $v_{i,l,j}$ and then identify Reduction Constraints:

$$\left(\sum_{i \in I} v_{i,l,j} \right) - y_{l,j} = 0 \quad \forall l \in L,\ j \in J \tag{3}$$

Specifically, we find Reduction Constraints by first grouping $v_{i,l,j}$ using Path Definitions, i.e., aggregate $v_{i,l,j}$ with the corresponding $y_{l,j}$, and then look for constraints where path flows $v_{i,l,j}$ have coefficients $+1$ and $y_{l,j}$ has coefficient -1. Finding these Reduction Constraints helps identify variable subscripts since every $v_{i,l,j}$ has the same (l, j) indices as the $y_{l,j}$ variable. Figure 3 visualizes the Eq. 3 reduction constraints for the Figure 1 example problem. Recall that constraints are identified

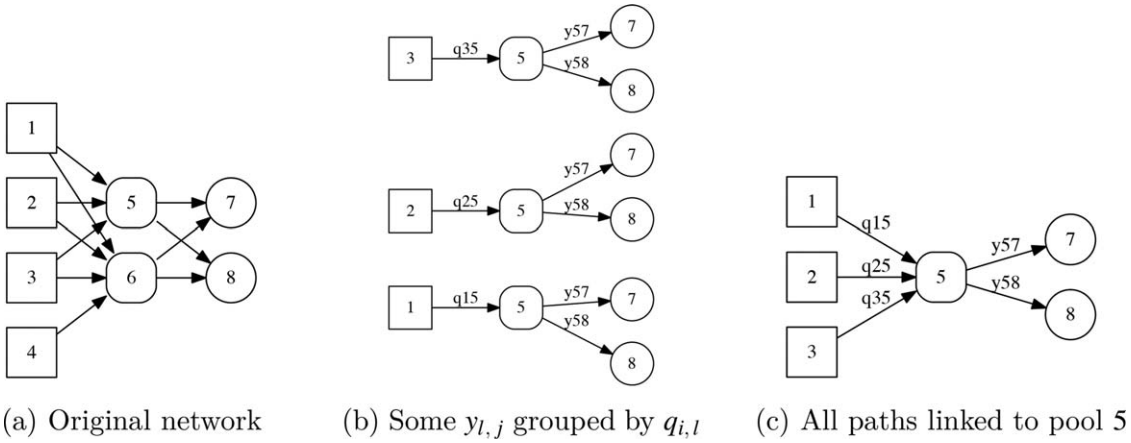


Figure 4. Visualization for grouping path flow variables into set $v(l)$ where all $v_{i,l,j}$ have the same l.

We do not know the original network diagrammed in (a) but we can use, (b), the Path Definition constraints to group each flow $y_{l,j}$ with corresponding $q_{i,l}$. All the sub-networks connected to pool l, (b), have identical $y_{l,j}$ network connections, so we obtain a sub network around each pool, (c).

in tandem with transforming to the PQ-formulation. Variable $v_{i,l,j}$ replaces any bilinear terms $q_{i,l}\,y_{l,j}$ uncovered in this anaysis.

Identifying Flows Through Pools. Identifying the reduction and path definition constraints (Eq. 3) allows the algorithm to find what pool l corresponds to each variable and parameter. We define a set $v(\hat{l})=\{v_{i,l,j}|l=\hat{l}\}$ where all the $v_{i,l,j}$ are grouped by l and follow the process visualized in Figure 4. Observe that path definitions $v_{i,l,j}=q_{i,l}\,y_{l,j}$ identify term pairs $q_{i,l}$ and $y_{l,j}$ which must have the same pool index l. For each flow $q_{i,l}$, we collect all flows $y_{l,j}$ appearing jointly in path definitions and obtain a set of $y_{l,j}$ for each $q_{i,l}$ (Figure 4b). The algorithm subsequently aggregates the $q_{i,l}$ with matching flows $y_{l,j}$ (Figure 4c). After finding $v(l)\,\forall\, l \in L$, we tag Pool Capacity constraints by recognizing Eq. 4 as a sum of $v_{i,l,j}$ variables over a set $v(l)$:

$$\sum_{i\in I, j\in J} v_{i,l,j} \le S_l \quad \forall l \in L \tag{4}$$

Identifying Input and Output Capacities. Next, the algorithm identifies what input i or output j indices correspond to each variable by differentiating the Input Capacity (Eq. 5) from the Output Capacity (Eq. 6) constraints. Recall that the algorithm has already identified the flow variables $v_{i,l,j}$, Reduction Constraints (Eq. 3), and Pool Capacity constraints (Eq. 4), so the remaining $v_{i,l,j}$ sums with all coefficients $+1$ must be either Input or Output Capacity constraints:

$$A_i^L \le \sum_{l\in L, j\in J} v_{i,l,j} + \sum_{j\in J} z_{i,j} \le A_i^U \quad \forall i \in I \tag{5}$$

$$D_j^L \le \sum_{i\in I, l\in L} v_{i,l,j} + \sum_{i\in I} z_{i,j} \le D_j^U \quad \forall j \in J \tag{6}$$

To identify Input Capacity constraints we define set $v(\hat{l},\hat{j})=\{v_{i,l,j}|l=\hat{l}, j=\hat{j}\}$ where all the variables $v_{i,l,j}$ are grouped by (l, j) pairs. The algorithm easily constructs sets $v(l, j)$ by using the Path Definitions to aggregate $v_{i,l,j}$ corresponding to the same $y_{l,j}$. To find the Input Capacity constraints, observe that all $v_{i,l,j}$ variables in Eq. 5 are sums over (l, j) and therefore each $v_{i,l,j}$ in the sum will correspond to a different (l, j) pair. So the algorithm tags input capacity constraints as those where every variable $v_{i,l,j}$ belongs to a different set $v(l, j)$.

Identifying Output Capacity constraints (Eq. 6) is analogous to tagging the Input Capacity constraints; we define set $v(\hat{i},\hat{l})$

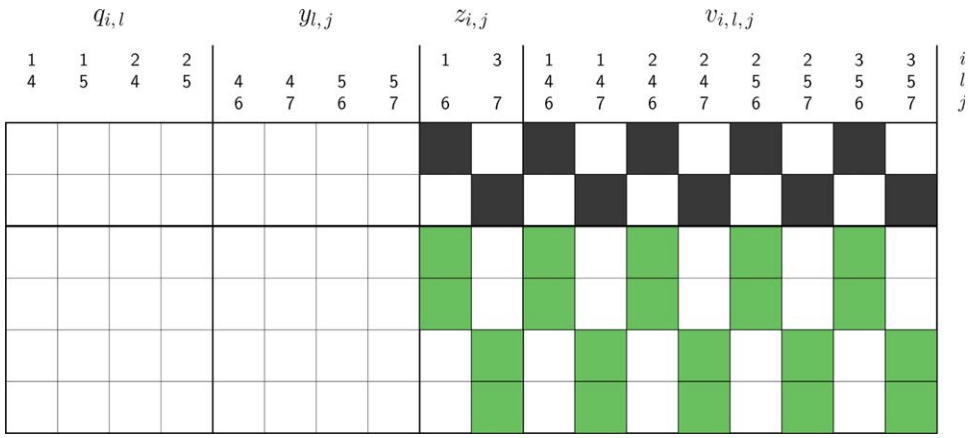


Figure 5. The Output Capacity (black) and Product Quality (green) constraints use the same variables but have different coefficients.

In this example, the first two Product Quality constraints match the first Output Capacity equation and the last two Product Quality Constraints match the second Output Capacity constraint. [Color figure can be viewed in the online issue, which is available at wileyonlinelibrary.com.]

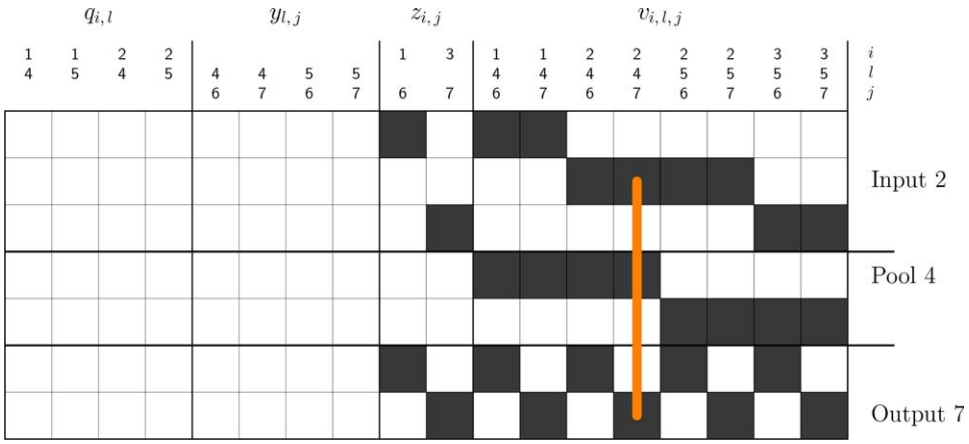


Figure 6. Path from input 2 to output 7 through pool 4.

[Color figure can be viewed in the online issue, which is available at wileyonlinelibrary.com.]

$=\{v_{i,l,j}|i=\hat{i},\ l=\hat{l}\}$ and construct all the sets $v(i,\ l)$ by using the Path Definitions to aggregate $v_{i,l,j}$ corresponding to the same $q_{i,l}$. Output Capacity constraints are sums over $v_{i,l,j}$ where every variable $v_{i,l,j}$ belongs to a different set $v(i,\ l)$.

After finding the Input and Output Capacity constraints, the algorithm labels the extra variables appearing in Eqs. 5 and 6 as $z_{i,j}$. If there are input or output nodes which are not connected to any pool, there may be unidentified inputs i or outputs j; the extra $z_{i,j}$ variables identify these input i or output j nodes which transmit or receive, respectively, bypass flows.

Identifying Quality Constraints. The only remaining equations to identify in the canonical PQ-formulation are the Product Quality constraints. Product Quality equations have the same variables as the Output Capacity constraints (Eq. 6), but the Output Capacities always have coefficient $+1$, while the Product Quality constraints each have coefficient $C_{i,k}-P^L_{j,k}$ or $C_{i,k}-P^U_{j,k}$.

The algorithm tags the k quality constraints associated with output j by looking for Product Quality constraints where the same variables appear with different coefficients, an example is illustrated in Figure 5. The $|K|\times|J|$ product quality constraints are used to deduce input and output qualities, $C_{i,k}$, $P^L_{j,k}$, $P^U_{j,k}$; Section describes this procedure. We assume that each of the $|K|$ qualities are given in the same order for every one of the Product Quality constraints.

Building the pooling problem network

Once all constraints and variables have been labeled, building the pooling network is almost trivial. We tag each network node with respect to an input i, pool l or output j; we also associate each node with (possibly infinite) capacity bounds. The algorithm also determines network flow arcs using path variables $v_{i,l,j}$ together with Eqs. 4–6 to find all inter-node connections. Figure 6 shows a path from input 2 to output 7 to pool 4 from the Figure 1 example.

Finding Input Cost and Output Revenue. The objective of a pooling problem is to minimize cost. After transformation to the canonical PQ-formulation, the objective has the form:

$$\sum_{i\in I,\,l\in L,\,j\in J}(c_i-d_j)\,v_{i,l,j}+\sum_{i\in I,\,j\in J}(c_i-d_j)\,z_{i,j}=\sum_{i\in I,\,l\in L,\,j\in J}\gamma_{i,j}\,v_{i,l,j}+\sum_{i\in I,\,j\in J}\gamma_{i,j}\,z_{i,j} \tag{7}$$

where $\gamma_{i,j}=c_i-d_j$. We already know the variables $v_{i,l,j}$ and $z_{i,j}$ and their associated node indices, Linear Program (8) decomposes the $\gamma_{i,j}$ parameters into input costs c_i and output revenue d_j. The solution to LP (8) may produce different values for c_i and d_j than the original network, but the objective value will the be same.

$$\begin{aligned}\min_{c,\,d}\quad & \sum_{i\in I}c_i+\sum_{j\in J}d_j\\ \text{s.t. } & c_i-d_j=\gamma_{i,j}\quad \forall i\in I, j\in J\\ & c_i\geq 0\quad \forall i\in I\\ & d_j\geq 0\quad \forall j\in J\end{aligned} \tag{8}$$

Finding Input and Output Qualities. Just as parameters c_i, d_j are not directly in Eq. 7, pooling problem parameters $C_{i,k}$, $P^L_{j,k}$, $P^U_{j,k}$ are not directly present in the canonical PQ-formulation. To find these input and output quality parameters, define $\Delta^L_{i,j,k}=C_{i,k}-P^L_{j,k}$ and $\Delta^U_{i,j,k}=C_{i,k}-P^U_{j,k}$ and re-write the Product Quality constraints:

$$\sum_{i,l}\Delta^L_{i,j,k}\,v_{i,l,j}+\sum_{i\in I}\Delta^L_{i,j,k}\,z_{i,j}\geq 0\quad \forall j\in J,\ k\in K \tag{9}$$

$$\sum_{i,l}\Delta^U_{i,j,k}\,v_{i,l,j}+\sum_{i\in I}\Delta^U_{i,j,k}\,z_{i,j}\leq 0\quad \forall j\in J,\ k\in K \tag{10}$$

Linear Program (11) deduces equivalent numerical values for $C_{i,k}$, $P^L_{j,k}$, and $P^U_{j,k}$. Here we assume that the indices of the input nodes i, output nodes j, and tracked qualities k are known. We also assume that a Product Quality constraint for

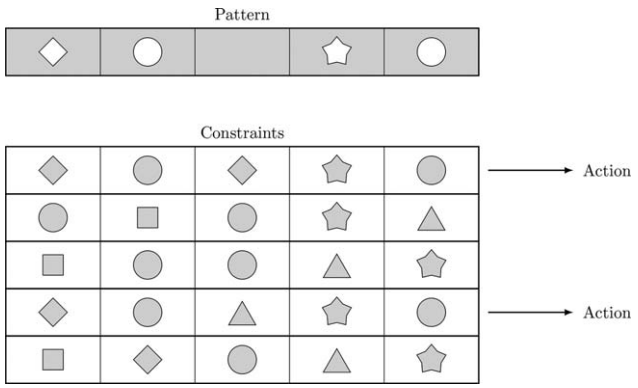


Figure 7. If a constraint matches a pattern described in Sections &, execute an action to transform or identify a constraint.

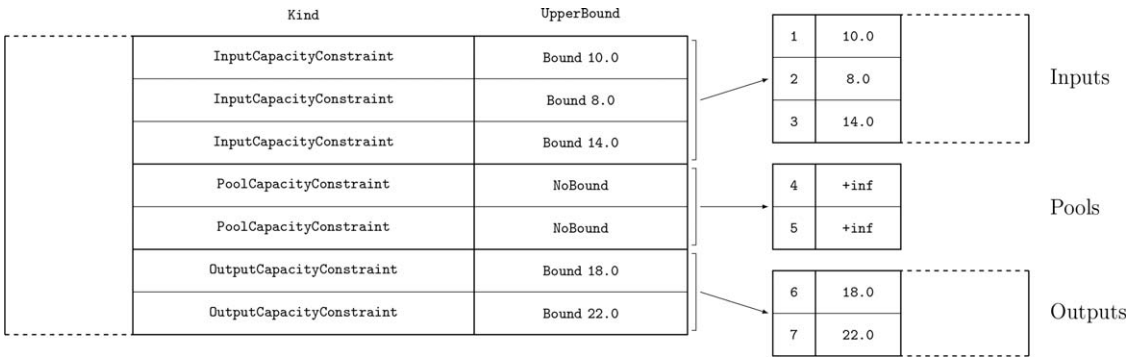


Figure 8. Building nodes from capacity constraints.

(i, j, k) only exists if input i and output j are connected indirectly or via a pool.

$$\begin{aligned} &\underset{C,P^U,P^L}{\text{minimize}} \sum_{i\in I,k\in K} C_{i,k} + \sum_{j\in J,k\in K} P^L_{j,k} + \sum_{j\in J,k\in K} P^U_{j,k} \\ &\text{subject to } C_{i,k} - P^L_{j,k} = \Delta^L_{ijk} \quad \forall i \in I, j \in J, k \in K \\ &\qquad C_{i,k} - P^U_{j,k} = \Delta^U_{ijk} \quad \forall i \in I, j \in J, k \in K \end{aligned} \tag{11}$$

Solving LP (11) may produce different values for $C_{i,k}$, $P^L_{j,k}$, and $P^U_{j,k}$ than in the original problem, but the feasible space defined by the constraints is identical.

Implementation Sketch

This section sketches implementing the algorithm; the online supplementary material gives more details. We describe data structures, discuss IO facilities, show how to implement the algorithms.

Data structures

A pooling network is characterized by its inputs, pools, outputs, and arcs. Mathematical optimization problems are defined by a set of variables, an objective function, and a set of constraints; MINLP may also have an associated pooling network. The online supplementary material defines several modules and types which generically represent MINLP optimization problems. The most general module, `Problem`, has a record field to contain a `PoolingNetwork`. When the algorithm initiates, the `PoolingNetwork` field is `None`. After the implementation terminates, `PoolingNetwork` contains the standard pooling network and the `constraints` field holds additional equations irrelevant to a standard pooling framework.

Reading and writing problems

The implementation reads input problems in OSiL format; OSiL is a widely supported optimization format and is easy to parse since it is based on XML.[53] We provide two distinct output formats: the first writes the pooling network to a Graphviz file and the second writes the pooling network to a Dey and Gupte[17] formatted AMPL data file. The Graphviz file output, which may be subsequently converted to an image, generated the results shown in this article.

Transforming and identifying constraints

The implementation concurrently: (1) reformulates a pooling problem to the canonical PQ-formulation and (2) identifies the pooling problem constraints in an optimization problem. All functions transforming and identifying constraints follow a common pattern: scan every constraint and, if it matches a given pattern, execute an action. Figure 7 illustrates the procedure and the online supplement gives more specific details. Actions can be as simple as updating the constraint type or as complex as replacing a variable in the whole problem.

Building the network

The final implementation phase is building the network. We begin by building inputs, pools, and outputs from the constraints. As shown in Figure 8, node capacities are either the constraint upper bound or otherwise infinite. Finally, we need to find the price and quality specifications for both the inputs

Table 2. Results on Standard and Extended Pooling Problems[54]

Problem	Network found?	Same as original?	Qualities	Notes
Adhya 1	Yes	Yes	Yes	
Adhya 2	Yes	Yes	Yes	
Adhya 3	Yes	Yes	Yes	
Bental 4	Yes	No	No	Missing one input not connected to pool
Bental 5	Yes	No	No	Duplicate inputs with infinite capacity
Foulds 2	Yes	No	No	Missing inputs not connected to pool
Foulds 3	Yes	No	Yes	Duplicate infinite capacity inputs; Different input cost and quality
Foulds 4	Yes	No	Yes	”
Foulds 5	Yes	No	Yes	”
Haverly 1	Yes	No	No	Missing input not connected to pool
Haverly 2	Yes	No	No	”
Haverly 3	Yes	No	No	”
RT 2	Yes	No	No	Duplicate outputs
EPA Small	Yes	Yes	Yes	Original problem modified to include output capacities
EPA Midsize	Yes	Yes	Yes	”
EPA Large	Yes	Yes	Yes	”

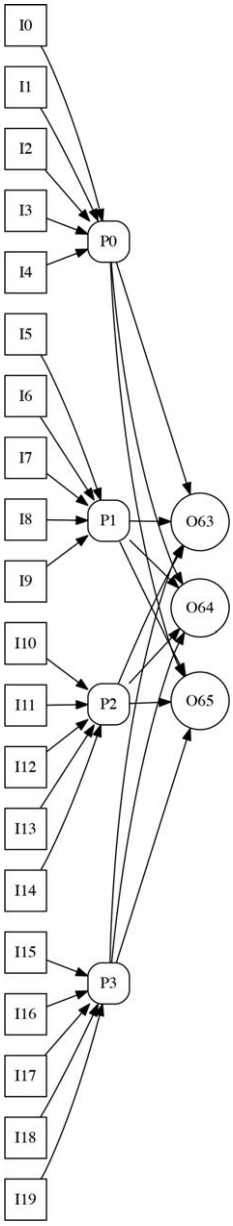


Figure 9. Reconstructed network for MINLPLib2 test case Lee1.

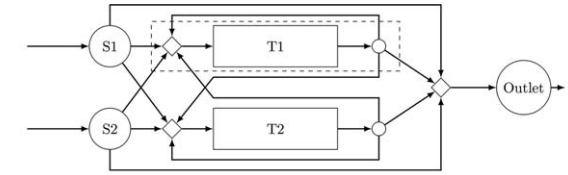
This network is mathematically equivalent to the original Lee1 network which has 5 rather than 20 input nodes. The original network has five input nodes with infinite capacities; the implementation cannot distinguish this from five input nodes for each of the pools (all of which still have infinite capacities).

and outputs; we find these parameters by solving LPs (8) and (11).

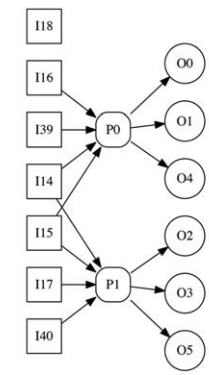
Results and Discussion

We tested the implementation on three sets of input OSiL[53] files: the 70 large-scale standard pooling Dey and Gupte[17] examples, the 16 standard and extended Misener et al.[54] examples, and the 1342 MINLPLib2* tests cases. For each test set, we read in a flat optimization problem and try to produce a pooling network.

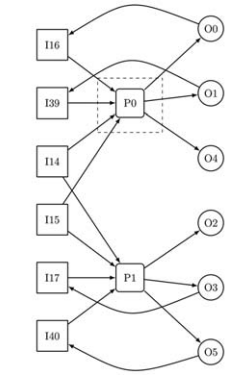
*http://www.gamsworld.org/minlp/minlplib2/html/; Accessed July 2015.



(a) Example waste treatment problem. Small circles represent splitters, diamonds are mixers.



(b) The pooling problem found in the water treatment problem.



(c) Joining the extra inputs/outputs yields a new network with loops, similar to the original problem.

Figure 10. Pooling problem found in the `wastewater02m2`.

The implementation successfully deduces the original network structure for all 70 Dey and Gupte[17] examples, i.e., large-scale, standard pooling problems with up to 40 input, 30 pool, and 50 output nodes. The associated optimization problems have up to 11,442 variables and 12,883 constraints. These are the largest pooling problems solved to date and we therefore have effectively no size limitation for the detection methodology.

Table 2 completely documents the results on the standard and extended pooling problems from Misener et al.[54] The

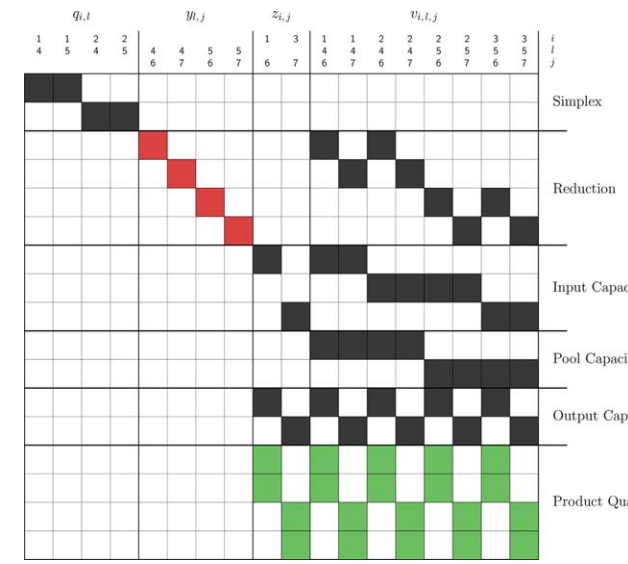


Figure 11. Visualization of the canonical PQ-formulation for the Figure 1 example pooling problem.

Black boxes represent a coefficient equal to 1, red boxes a coefficient of −1 and green boxes all other nonzero coefficients. [Color figure can be viewed in the online issue, which is available at wileyonlinelibrary.com.]

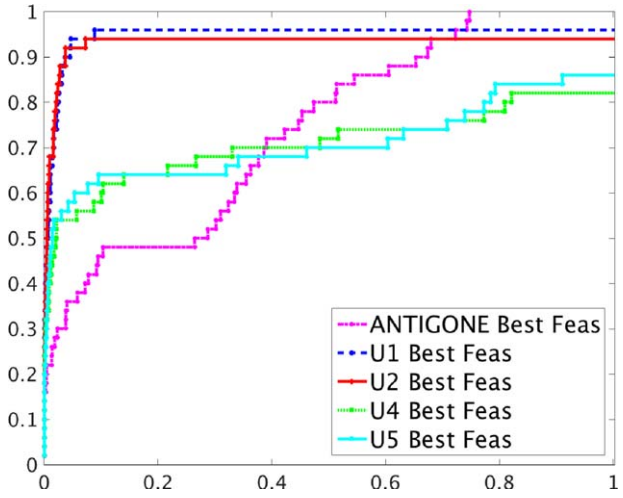


Figure 12. Performance profile showing the advantage of using the pattern-finding algorithm for the Dey and Gupte[17] standard pooling test set.

This performance profile shows the improvements possible in using the U1, U2, U4, and U5 approximations proposed by Dey and Gupte[17] with respect to the built-in ANTIGONE heuristic after our code found the pooling problem in the flat MINLP. [Color figure can be viewed in the online issue, which is available at wileyonlinelibrary.com.]

implementation has no problem excluding the nonlinear Environmental Protection Agency[15] blending rules from consideration. The pooling problems in this test set are less stylized and not consistently defined as the Dey and Gupte[17] test set, so observe that, although we get a pooling network for all the problems, it is not necessarily the original pooling network.

Running our implementation on the 1342 test cases in MINLPLib2, three produce complete pooling problems and 78 more produce a pooling-like network. The first two problems are from Lee and Grossmann.[55] The original optimization problems are generalized pooling problems with mutable network topology, but the implementation finds an equivalent standard pooling problem. Figure 9 shows the network produced for `Lee1`; this is mathematically equivalent to the original `Lee1` network because the input capacities are infinite. The third completely reconstructed problem is `Haverly`. The `Haverly` test case did not do as well in the Misener et al.[54] test suite; this result reminds us that our hunt for pooling problems within a larger optimization problem is ultimately a heuristic.

As an example of the 78 additional problems where the implementation finds a pooling-like network, we discuss the `wastewater02m2` wastewater network problem.[56] As illustrated in Figure 10a, the wastewater problem no longer has a feed-forward network because flows exiting the treatment units (`T1` and `T2`) may be sent through the treatment units again. The pooling problem we find in Figure 10b is exciting because it has *flattened* the wastewater treatment problem into a pooling problem. The two extra inputs and outputs illustrated in Figure 10b could be made equivalent to the original topology by using some of the additional constraints in the auxiliary problem, see Figure 10c. Note in Figure 10c that the number of pipes entering/exiting pools is the same as the number of pipes entering/exiting the treatment units in Figure 10a.

The 16 standard and extended Misener et al.[54] examples and the 78 MINLPLib2 test cases with network structure can be addressed with off-the-shelf solver software,[37] but the 70 large-scale standard pooling Dey and Gupte[17] examples are currently out of reach for solver software. To test the potential of using the Dey and Gupte[17] heuristic in tandem with our pattern-finding approach, we compare using performance profiles[57]:

$$t_{p,s} \equiv \text{Performance metric, } i.e., \text{ best feasible solution, for problem } p \text{ by solver } s \in S$$

$$r_{p,s} \equiv \frac{t_{p,s} - \min_{p \in P}\left\{t_{p,s'} \,:\, s' \in S\right\}}{\min_{p \in P}\left\{t_{p,s'} \,:\, s' \in S\right\}}; \quad s \in S$$

$$\rho_s(\tau) = \frac{1}{n_p}\,\text{size}\left\{p \in P \,:\, r_{p,s} \le \tau\right\}$$

After finding the structure in the Dey and Gupte[17] test set, we initialized ANTIGONE, and several of the Dey and Gupte[17] approximation algorithms (U1, U2, U4, U5) and ran each for 30 min. Figure 12 shows the advantage of using the new heuristic in comparison to the built-in ANTIGONE heuristics for this problem class.

Although our pattern-matching algorithm coupled with the Dey and Gupte[17] heuristic is much more powerful than the heuristics currently in ANTIGONE, we have not added the new pattern matching code to ANTIGONE because, at this point, the pattern-matching time significantly slows down ANTIGONE performance on non-pooling problems. But we are still very interested in these new results because we know that there are several cutting plane approaches for the pooling problem under development.[18,19] Once both cutting plane and heuristic approaches are well-developed for the pooling problem, finding large pooling problems will be even more advantageous. Furthermore, this article proves that optimization researchers can easily continue studying the more stylized standard pooling problem since we can find pooling structure within larger, flat MINLP.

Conclusions

This article has explored, for the first time, the possibility of finding special named structure within an MINLP optimization problem; this significantly extends work by the MIP community that has shown how to find network structure for better primal heuristics and cutting planes. We are specifically interested in the range of novel techniques which have been developed for the standard pooling problem and wanted to see how far we could get in recognizing pooling structure within a general MINLP. We have shown that we can detect all standard and extended pooling problems in the literature and that we can find pooling networks in 6% of MINLPLib. We have also shown that, after detecting the pooling problem in the flat MINLP, we can apply a good heuristic approach to get a good approximation solution.

Acknowledgments

This work is supported by EPSRC DTP funding to G.K., a Royal Academy of Engineering Research Fellowship to R.M., and EPSRC Grant EP/M028240/1.

Literature Cited:

1. Haverly CA. Studies of the behavior of recursion for the pooling problem. *ACM SIGMAP Bull.* 1978;25:19–28.

2. Misener R, Floudas CA. Advances for the pooling problem: modeling, global optimization, and computational studies. *Appl Comput Math*. 2009;8(1):3–22.
3. Lee H, Pinto JM, Grossmann IE, Park S. Mixed-integer linear programming model for refinery short-term scheduling of crude oil unloading with inventory management. *Ind Eng Chem Res*. 1996; 35(5):1630–1641.
4. Li J, Li A, Karimi IA, Srinivasan R. Improving the robustness and efficiency of crude scheduling algorithms. *AIChE J*. 2007;53(10): 2659–2680.
5. Li J, Misener R, Floudas CA. Continuous-time modeling and global optimization approach for scheduling of crude oil operations. *AIChE J*. 2012a;58(1):205–226.
6. Li J, Misener R, Floudas CA. Scheduling of crude oil operations under demand uncertainty: a robust optimization framework coupled with global optimization. *AIChE J*. 2012b;58(8):2373–2396.
7. Galan B, Grossmann IE. Optimal design of distributed wastewater treatment networks. *Ind Eng Chem Res*. 1998;37(10):4036–4048.
8. Selot A, Kuok LK, Robinson M, Mason TL, Barton PI. A short-term operational planning model for natural gas production systems. *AIChE J*. 2008;54(2):495–515.
9. Li X, Armagan E, Tomasgard A, Barton PI. Stochastic pooling problem for natural gas production network design and operation under uncertainty. *AIChE J*. 2011;57(8):2120–2135.
10. Papageorgiou DI, Toriello A, Nemhauser GL, Savelsbergh MWP. Fixed-charge transportation with product blending. *Transport Sci*. 2012;46(2): 281–295.
11. Baliban RC, Elia JA, Misener R, Floudas CA. Global optimization of a MINLP process synthesis model for thermochemical based conversion of hybrid coal, biomass, and natural gas to liquid fuels. *Comput Chem Eng*. 2012;42:64–86.
12. Kolodziej SP, Grossmann IE, Furman KC, Sawaya NW. A discretization-based approach for the optimization of the multiperiod blend scheduling problem. *Comput Chem Eng*. 2013;53:122–142.
13. Meyer CA, Floudas CA. Global optimization of a combinatorially complex generalized pooling problem. *AIChE J*. 2006;52(3):1027–1037.
14. Misener R, Floudas CA. Global optimization of large-scale pooling problems: quadratically constrained MINLP models. *Ind Eng Chem Res*. 2010;49(11):5424–5438.
15. Misener R, Gounaris CE, Floudas CA. Mathematical modeling and global optimization of large-scale extended pooling problems with the (EPA) complex emissions constraints. *Comput Chem Eng*. 2010; 34(9):1432–1456.
16. Bussieck MR, Drud AS, Meeraus A. MINLPLib—a collection of test models for mixed-integer nonlinear programming. *INFORMS J Comput*. 2003;15(1):114–119.
17. Dey SS, Gupte A. Analysis of MILP techniques for the pooling problem. *Oper Res*. 2015;63(2):412–427.
18. D'Ambrosio C, Linderoth J, Luedtke J. *Integer Programming and Combinatoral Optimization: 15th International Conference, IPCO 2011, New York, NY, USA, June 15-17, 2011*. Proceedings, chapter Valid Inequalities for the Pooling Problem with Binary Variables. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011:117–129.
19. D'Ambrosio C, Linderoth J, Luedtke J, Miller A. Strong convex nonlinear relaxations of the pooling problem. http://minlp.cheme.cmu.edu/2014/papers/linderoth.pdf, 2014.
20. Gounaris CE, Misener R, Floudas CA. Computational comparison of piecewise-linear relaxations for pooling problems. *Ind Eng Chem Res*. 2009;48(12):5742–5766.
21. Castro PM, Teles JP. Comparison of global optimization algorithms for the design of water-using networks. *Comput Chem Eng*. 2013;52: 249–261.
22. Boland N, Kalinowski T, Rigterink F. A polynomially solvable case of the pooling problem. *arXiv.org*, 2015.
23. Haugland D. The computational complexity of the pooling problem. *J Glob Optim*. 2015:1–17.
24. Van Roy TJ, Wolsey LA. Solving mixed integer programming problems using automatic reformulation. *Oper Res*. 1987;35(1): 45–57.
25. Brown GG, Wright WG. Automatic identification of embedded network rows in large-scale optimization models. *Math Program*. 1984; 29(1):41–56.
26. Bixby RE, Fourer R. Finding embedded network rows in linear programs I. Extraction heuristics. *Manag Sci*. 1988;34(3):342–376.
27. Gülpinar N, Gutin G, Mitra G, Zverovitch A. Extracting pure network submatrices in linear programs using signed graphs. *Discrete Appl Math*. 2004;137(3):359–372.
28. Nemhauser GL, Savelsbergh MWP, Sigismondi GC. MINTO, a mixed INTeger optimizer. *Oper Res Lett*. 1994;15(1):47–58, 1994.
29. Achterberg T, Raack C. The MCF-separator: detecting and exploiting multi-commodity flow structures in MIPs. *Math Program Comput*. 2010;2(2):125–165.
30. Salvagnin D. Detecting semantic groups in MIP models. In: *Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems (CPAIOR)*, Lecture Notes in Computer Science, 2016.
31. McCormick GP. Computability of global solutions to factorable nonconvex programs: part 1-convex underestimating problems. *Math Program*. 1976;10(1):147–175.
32. Sahinidis NV. BARON: a general purpose global optimization software package. *J Glob Optim*. 1996;8(2):201–205.
33. Achterberg T. SCIP: solving constraint integer programs. *Math Program Comput*. 2009;1(1):1–41.
34. Belotti P, Lee J, Liberti L, Margot F, Wächter A. Branching and bounds tightening techniques for non-convex MINLP. *Optim Method Softw*. 2009;24(4–5):597–634.
35. Lin Y, Schrage L. The global solver in the LINDO API. *Optim Method Softw*. 2009;24(4–5):657–668.
36. Berthold T, Gleixner AM, Heinz S, Vigerske S. Analyzing the computational impact of MIQCP solver components. *Num Algebra Control Optim*. 2012;2(4):739–748.
37. Misener R, Floudas CA. GloMIQO: global mixed-integer quadratic optimizer. *J Glob Optim*. 2013;57(1):3–50.
38. Misener R, Floudas CA. ANTIGONE: algorithms for coNTinuous integer global optimization of nonlinear equations. *J Glob Optim*. 2014;59(2–3):503–526.
39. Liberti L, Pantelides CC. An exact reformulation algorithm for large nonconvex NLPs involving bilinear terms. *J Glob Optim*. 2006; 36(2):161–189.
40. Liberti L, Cafieri S, Tarissan F. Reformulations in mathematical programming: a computational approach. In: Abraham A, et al., editors, *Foundations of Computational Intelligence Volume 3, volume 203 of Studies in Computational Intelligence*. Berlin: Springer, 2009:153–234.
41. Misener R, Smadbeck JB, Floudas CA. Dynamically-generated cutting planes for mixed-integer quadratically-constrained quadratic programs and their incorporation into GloMIQO 2.0. *Optim Method Softw*. 2014;30(1):215–249.
42. Sherali HD, Adams WP. *A Reformulation-Linearization Technique for Solving Discrete and Continuous Nonconvex Problems*. Nonconvex Optimization and Its Applications. Dordrecht, Netherlands: Kluwer Academic Publishers, 1999.
43. Quesada I, Grossmann IE. Global optimization of bilinear process networks with multicomponent flows. *Comput Chem Eng*. 1995; 19(12):1219–1242.
44. Tawarmalani M, Sahinidis NV. *Convexification and Global Optimization in Continuous and Mixed-Integer Nonlinear Programming: Theory, Applications, Software, and Applications. Nonconvex Optimization and Its Applications*. Norwell, MA: Kluwer Academic Publishers, 2002.
45. Karuppiah R, Grossmann IE. Global optimization for the synthesis of integrated water systems in chemical processes. *Comput Chem Eng*. 2006;30:650–673.
46. Tawarmalani M, Ahmed S, Sahinidis NV. *Product disaggregation in global optimization and relaxations of rational programs. Optim Eng*. 2002;3:281–303.
47. Zorn K, Sahinidis NV. Computational experience with applications of bilinear cutting planes. *Ind Eng Chem Res*. 2013;52(22):7514–7525.
48. Zorn K, Sahinidis NV. Global optimization of general non-convex problems with intermediate bilinear substructures. *Optim Method Softw*. 2014;29(3):442–462.
49. Liberti L. https://www.mfo.de/document/1543/preliminary_OWR_2015_46.pdf. Open Problems Session at the Oberwolfach MINLP Workshop, 2015.
50. Ben-Tal A, Eiger G, Gershovitz V. Global minimization by reducing the duality gap. *Math Program*. 1994;63:193–212.
51. Audet C, Brimberg J, Hansen P, Le Digabel S, Mladenovic N. Pooling problem: alternate formulations and solution methods. *Manage Sci*. 2004;50(6):761–776.

52. Alfaki M, Haugland D. A multi-commodity flow formulation for the generalized pooling problem. *J Glob Optim*. 2013;56(3):917–937.
53. Fourer R, Ma J, Martin K. OSiL: an instance language for optimization. *Comput Optim Appl*. 2010;45(1):181–203.
54. Misener R, Thompson JP, Floudas CA. APOGEE: global optimization of standard, generalized, and extended pooling problems via linear and logarithmic partitioning schemes. *Comput Chem Eng*. 2011; 35(5):876–892.
55. Lee S, Grossmann IE. Global optimization of nonlinear generalized disjunctive programming with bilinear equality constraints: applications to process networks. *Comput Chem Eng*. 2003;27(11):1557–1575.
56. Castro PM, Matos HA, Novais AQ. An efficient heuristic procedure for the optimal design of wastewater treatment systems. *Resour Conservat Recycling* 2007;50(2):158–185.
57. Dolan ED, Moré JJ. Benchmarking optimization software with performance profiles. *Math Program*. 2002;91:201–213.

Manuscript received Dec. 8, 2015, and revision received May 1, 2016.