Synthesis of Constraints for Mathematical Programming With One-Class Genetic Programming

Tomasz P. Pawlak and Krzysztof Krawiec

***Abstract*—Mathematical programming (MP) models are common in optimization of real-world processes. Models are usually built by optimization experts in an iterative manner: an imperfect model is continuously improved until it approximates the reality well-enough and meets all technical requirements (e.g., linearity). To facilitate this task, we propose a genetic one-class constraint synthesis method (GOCCS). Given a set of exemplary states of normal operation of a business process, GOCCS synthesizes constraints in linear programming or nonlinear programming form. The synthesized constraints can be then paired with an arbitrary objective function and supplied to an off-the-shelf solver to find optimal parameters of the process. We assess GOCCS on three families of MP benchmarks and conclude promising results. We also apply it to a real-world process of wine production and optimize that process.**

***Index Terms*—Business process, constraint acquisition, linear programming (LP), model induction, wine quality.**

I. Introduction

EACH nontrivial noncombinatorial optimization problem involves a model of the process of interest that comprises three key elements: 1) process variables; 2) constraints on their values; and 3) an objective function. Though there is a multitude ways in which such models can be expressed, linear programming (LP) and nonlinear programming (NLP) formalisms are *de facto* standards of representation, both in academia and industry.

A typical practice is to build models manually, which may be, however, error-prone, laborious and time-consuming. Some characteristics of the process may not be known to the expert, and hence mistakenly omitted in the model. The characteristics of the process may require advanced (e.g., nonlinear) modeling techniques that are typically more difficult to use. Also, few experts combine practical domain knowledge with competencies in modeling techniques. Last but not least, expert's subjective preferences may lead to models that are biased or even flawed.

Manuscript received March 29, 2017; revised September 13, 2017 and January 20, 2018; accepted May 9, 2018. Date of publication May 11, 2018; date of current version January 28, 2019. This work of T. P. Pawlak was supported in part by the Poznań University of Technology, Poland, under Grant 09/91/DSMK/0634, and in part by the Foundation for Polish Science. The work of K. Krawiec was supported by the National Science Centre, Poland, under Grant 2014/15/B/ST6/05205. *(Corresponding author: Tomasz P. Pawlak.)*

The authors are with the Institute of Computing Science, Poznań University of Technology, 60-965 Poznań, Poland (e-mail: tpawlak@cs.put.poznan.pl; krawiec@cs.put.poznan.pl).

Color versions of one or more of the figures in this paper are available online at http://ieeexplore.ieee.org.

Digital Object Identifier 10.1109/TEVC.2018.2835565

An alternative to manual modeling is *synthesis* of a model from observations, where an observation is a snapshot of the state of the process (as captured by the variables) accompanied with the value of the objective function. Such a *regression problem* can be solved with a range of methods in statistics and/or machine learning (ML). No wonder automatic or computer-assisted construction of objective functions is routinely used in business in order to, e.g., decrease manufacturing costs or improve product quality.

In contrast to this, except for a handful of works which we review in Section III, little has been done to automate or support the acquisition of *constraints* from data. This is striking, because manual modeling of constraints faces similar challenges as those for objective functions. More than that: one may argue that modeling of constraints is even more challenging and laborious, e.g., when there are many of them. It is not unusual for constraints to be more complex than the objective function, e.g., when they involve auxiliary variables. Also, constraints are essential, not least because good solutions often dwell in proximity to constraints in the feasible region.

To fill this niche, we proposed GenetiCS [1], a method that employs genetic programming (GP) to synthesize constraints from examples of feasible and infeasible states of a process. GP allows controlling the syntax of evolved expressions so that they are acceptable by mathematical programming (MP) solvers, while the models used in ML do not fit that purpose. GenetiCS proved capable of synthesizing readable and accurate constraints for problems featuring low-to-medium number of variables, requiring just one or multiple constraints, and for the number of examples ranging from dozens to hundreds.

In this paper, we address the main limitation of GenetiCS, namely the requirement for examples representing both feasible and infeasible states (two *decision classes* in ML terms). In many real-world settings, the latter can be observed only occasionally, e.g., when a production plant fails or a process is disrupted by some rare event. As a consequence, infeasible examples may be few and far between, if not entirely absent, which significantly limits the practicality of GenetiCS.

The main contribution of this paper is thus genetic one-class constraint synthesis (GOCCS), a GP-based method that synthesizes constraints for LP and NLP models *from feasible examples only*, i.e., handles it as a *one-class classification problem* (Section IV). When evaluated on a range of benchmarks, GOCCS consistently produces high-quality constraints (Section V). We also apply it to a real-world problem

of modeling wine composition (Section VI), obtaining constraints that confirm domain knowledge and lead to interesting insights.

II. One-Class Constraint Synthesis Problem

Let $x_1, x_2, \ldots, x_n \in \mathbb{R}$ be variables, $s = (x_1, x_2, \ldots, x_n)$ denotes a state of the process of interest, and S be a set of examples of feasible states. A constraint $c(s)$ is an expression $p(s) \leq q(s)$, where $p : \mathbb{R}^n \to \mathbb{R}$ and $q : \mathbb{R}^n \to \mathbb{R}$ are functions of a certain class H (e.g., linear and polynomial). A *constraint synthesis problem* is a tuple (S, H), and solving it requires finding a set of constraints of class H that hold for all elements of S. Formally, any set of constraints C such that $\forall_{s \in S} \forall_{c \in C}\, c(s)$ holds is a solution to (S, H).

Under this formulation, every constraint synthesis problem has multiple solutions, including trivial ones. For instance, the set of linear constraints $C = \{x_1 < x_1 + 1\}$ is formally a solution to *all* constraint synthesis problems. However, this relaxed formalization is intended and will suffice for the discourse that follows. Which solutions should be favored to others depends on application and user's preferences, which we will come back to in Section IV-C.

The above formulation is agnostic about the objective function, which typically forms a part of an MP model. For this reason, in the following we use the terms "set of constraints" and "model" interchangeably.

III. Related Work

Most of the related studies pose constraint synthesis problem in a two-class way, where feasible and infeasible examples of states of a process are required.

The problem of two-class constraint synthesis for LP and NLP was defined in [2], formulated as a mixed-integer LP (MILP) problem, and solved using an off-the-shelf solver. This method yields promising results on synthetic benchmarks and a real-world problem of concrete mixture modeling. However, the MILP problem is NP-hard, and for large sets of examples the solver often depletes the computational budget, resulting in a suboptimal model.

As already stated in Section I, the precursor for GOCCS is GenetiCS [1]. It is faster than the method in [2], however, it does not guarantee finding the optimal model. The key difference with respect to GOCCS is that GenetiCS requires examples of both feasible and infeasible states, which is critical in practice, as the latter are often few and far between. Among other differences, GenetiCS uses discrete-valued fitness function calculated for each example separately and Lexicase selection [3] to pick parents based on these separate fitness assessments while, as it becomes clear in Section IV-C, GOCCS uses two fitness functions and NSGA-II selection [4].

Regarding two-class constraint synthesis in the form typical for constraint programming (CP), the Conacq system [5] synthesizes constraints expressed as a conjunctive normal form of terms, each comparing a single variable to a constant. An interactive variant of Conacq [6] repetitively queries an expert to classify artificial examples designed in a way that makes them highly informative for synthesis. The experiment shows that, of several proposed strategies of creating examples, the best one requires less than 100 queries to build a correct model. A follow-up approach by the same authors, QuAcq [7], allows for missing values of some variables, and so requires less knowledge from an expert. For constraints involving only $=$ or $\neq$ comparisons, QuAcq is proven to be asymptotically optimal in the number of expert's queries needed to converge.

Several studies tackled one-class constraint synthesis problems. Concerning LP constraints, a method that builds a convex hull and clusters its facets using k-means algorithm is proposed in [8]. Since the cost of building a convex hull is exponential with respect to the number of variables, the method becomes technically infeasible for over a dozen variables.

Regarding synthesis of constraints in CP form, model seeker [9] uses a handcrafted library of abstract constraints with associated metadata to synthesize concrete constraints as Prolog clauses. The algorithm yielded promising results on several real-world problems. Another method [10] learns first-order logic clauses using inductive logic programming and estimates their weights using preference learning. Performance of this method is human-competitive when assessed on a few synthetic benchmarks.

There is also ongoing research on synthesis of models in forms different than those normally considered in MP. For instance, learning modulo theories (LMT) [11] is a framework for building first-order logic theories from examples and background knowledge constraints, and predictive entropy search with constraints [12] is a tool to learn Bayesian models of constraints and objective function from examples.

The constraint synthesis problem as posed in Section II may be considered as a one-class classification problem. Many ML models for classification can be transformed into MP models, however, the resulting MP models may have undesirable properties. For instance, support vector data description (SVDD) [13] is a one-class equivalent of support vector machine [14] that wraps examples using a multidimensional sphere which is transformable into an arbitrary shape using kernel functions. SVDD constructs thus a single nonlinear constraint and assumes examples to be located in a single connected region in the decision space, while in real-world problems they may be arbitrarily distributed. The POSC4.5 algorithm [15] samples unlabeled examples prior to building a decision tree, and then uses the feasible and unlabeled examples as separate classes to be learned by C4.5 [16]. A decision tree is a disjunction of conjunctions of linear constraints that involve one variable each, and thus a decision tree implements a disjunction of hypercubes. Disjunction of constraints is not directly representable by MP models and requires auxiliary binary variables and extra constraints. The latter makes models difficult to interpret, while the former causes model solving to be NP-hard.

GP approaches have been previously proposed for one-class settings [17], [18], however, in application to conventional classification problems rather than constraint synthesis. One-class problems have been also approached with deep learning:

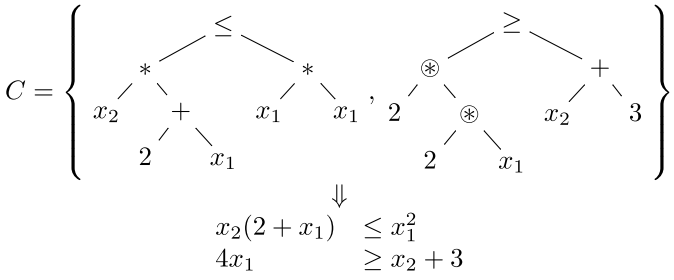

Fig. 1. GP individual comprising two trees and the corresponding model with two constraints.

autoencoders [19] are neural networks trained under the autoassociative regime, where a learner is required to *reproduce* the input pattern at its output. This must be achieved with hidden layers that are substantially smaller than input dimensionality, so that the network is forced to learn a compressed latent representation of the training set. However, the (often very complex and nonlinear) models implemented by neural networks are of no use in constraint synthesis problems, where constraints have to be represented in a compact symbolic form that is acceptable for solvers [20].

GOCCS fills the gap in the above spectrum of approaches by synthesizing constraints from feasible examples only (one-class synthesis), and presenting them in the form compatible with LP and NLP solvers (unlike Conaq [5] and neural networks [20]). Also, GOCCS is designed to synthesize a low number of easy-to-interpret, short, linear, or nonlinear constraints (unlike SVDD [13]), does not rely on background knowledge (unlike LMT [11]) nor handcrafted databases (unlike model seeker [9]), and controls the size of the synthesized constraints (unlike POSC4.5 [15]).

IV. Genetic Programming Algorithm

GOCCS uses GP to synthesize constraints in the form defined in the constraint synthesis problem (Section II). The following sections detail the components of our GP system.

A. Constraints Representation

A GOCCS individual (candidate solution) C is a variable-size set of expressions, each represented as a strongly typed tree [21]. Each tree $c \in C$ encodes a single constraint of the form $p(s) \leq q(s)$ or $p(s) \geq q(s)$, where the root node is either $\leq$ or $\geq$, and its two children subtrees encode $p(s)$ and $q(s)$ using one of two sets of instructions.

1) $L = \{+, -, \circledast, x_1, x_2, \ldots, x_n, \text{ERC}\}$ for synthesis of linear constraints, where $\circledast$ is a strongly typed multiplication accepting a constant as the left argument and any other instruction as the right argument, and ERC is an ephemeral random constant [22] initially drawn from $\mathcal{N}(0, 1)$,
2) $P = L \cup \{*\} \setminus \{\circledast\}$ for synthesis of polynomial constraints, where $*$ is multiplication accepting any arguments.

Fig. 1 shows an exemplary GP individual comprising two trees and the corresponding model with two constraints.

Algorithm 1 Full and Grow Initializations. *MinConstr* and *MaxConstr* Are Minimum and Maximum Numbers of Constraints, Respectively, *MaxDepth* Is Maximum Depth of Trees, $\mathcal{U}(a, b)$ Draws Uniformly an Integer From Range $[a, b]$

```
1: function FULL( )
2:     C ← ∅
3:     while |C| < MaxConstr do
4:         C ← C ∪ {FULLCONSTR(1)}
5:     return C
6: function FULLCONSTR(d)
7:     if d < MaxDepth then
8:         r ←PICKNONTERMINAL( )
9:         for i = 1..ARITY(r) do
10:            r_i ← FULLCONSTR(d + 1)
11:    else
12:        r ←PICKTERMINAL( )
13:    return r
14: function GROW( )
15:    c ← U(MinConstr, MaxConstr)
16:    C ← ∅
17:    while |C| < c do
18:        C ← C ∪ {GROWCONSTR(1)}
19:    return C
20: function GROWCONSTR(d)
21:    if d < MaxDepth then
22:        r ←PICKINSTRUCTION( )
23:        for i = 1..ARITY(r) do
24:            r_i ← GROWCONSTR(d + 1)
25:    else
26:        r ←PICKTERMINAL( )
27:    return r
```

B. Genetic Programming Operators

We adapt initialization and search operators of strongly typed GP [21] to evolve sets of constraints. For population initialization, we use the Full and Grow initialization operators from Algorithm 1. Full creates *MaxConstr* constraints in a top-down manner. For each constraint, beginning from the root node, it draws a nonterminal instruction for the current node if the current depth is less than *MaxDepth*, and a terminal otherwise. Then, it recursively repeats these steps for the children. In contrast, Grow draws the number of constraints to be created from the range [*MinConstr*, *MaxConstr*] and picks the instructions from the entire instruction set until *MaxDepth* is reached, when it picks from the terminals only. When creating children for a given node, Full and Grow use only instructions satisfying type requirements, i.e., comparisons $\{\leq, \geq\}$ for the root node, constants for the left argument of $\circledast$, and all instructions except comparisons for the remaining nodes. A population is initialized using the Ramped Half-and-Half (Rhh) scheme [22], by repeatedly calling Full or Grow with 50% probability each until the desired size is reached.

After initialization (and after each application of search operators described below) we eliminate syntactic duplicates within each individual. We define syntactic equivalence of trees recursively: two trees are considered equivalent if they use the same operator in their root nodes and all their corresponding children subtrees are equivalent. For commutative instructions $(+, *, \circledast)$ the order of children is ignored.

Note that initialization, as well as all operators described below, cares only about the syntactic validity

Algorithm 2 CTX and CTM. POP(X) Draws Uniformly and Removes an Item From X, TREESWAPPINGXOVER Is the Strongly Typed Tree Swapping Crossover [21]

function CTX(C_1, C_2)
  $C'_1, C'_2 \leftarrow \emptyset$
  while $C_1 \neq \emptyset \wedge C_2 \neq \emptyset$ **do**
    $c_1 \leftarrow$ POP(C_1)
    $c_2 \leftarrow$ POP(C_2)
    $\{c'_1, c'_2\} \leftarrow$ TREESWAPPINGXOVER(c_1, c_2)
    $C'_1 \leftarrow C'_1 \cup \{c'_1\}$
    $C'_2 \leftarrow C'_2 \cup \{c'_2\}$
  $C'_1 \leftarrow C'_1 \cup C_1$
  $C'_2 \leftarrow C'_2 \cup C_2$
  return $\{C'_1, C'_2\}$
function CTM(C)
  $C_r \leftarrow$ RHH( )
  $\{C', C'_r\} \leftarrow$ CTX(C, C_r)
  return C'

Algorithm 3 CSX and CSM. $\mathcal{U}(a, b)$ Draws Uniformly an Integer From the Range $[a, b]$

function CSX(C_1, C_2)
  $C'_1, C'_2 \leftarrow \emptyset$
  for $c \in C_1 \cup C_2$ **do** ▷ Iterate over all constraints in C_1 and C_2
    $i \leftarrow \mathcal{U}(1, 2)$
    $C'_i \leftarrow C'_i \cup \{c\}$
  return $\{C'_1, C'_2\}$
function CSM(C)
  $C_r \leftarrow$ RHH( )
  $\{C', C'_r\} \leftarrow$ CSX(C, C_r)
  return C'

of constraints: the models may contain redundant or inconsistent constraints (having no solution satisfying all of them). We allow such models in the population as they may give rise to more useful ones when processed by search operators (below).

New models are created with several search operators. Constraint tree crossover (CTX) and constraint tree mutation (CTM) shown in Algorithm 2 modify individual constraints. Given parents C_1 and C_2, CTX draws without repetition constraints c_1 and c_2, respectively, from C_1 and C_2, crosses them over using the strongly typed tree swapping crossover [21], and adds the resulting constraints c'_1 and c'_2 to the respective offspring constraint sets C'_1 and C'_2 (initially empty). These steps are repeated until all constraints from one parent have been used; the remaining constraints in the other parent are copied to the respective offspring. CTM applied to a parent C initializes a random model C_r using RHH, then calls CTX(C, C_r) and returns the first of the obtained offspring. Note that CTX and CTM preserve the number of constraints in the manipulated models.

We also introduce constraint swapping crossover (CSX) and constraint swapping mutation (CSM) (Algorithm 3) that move constraints between models but, in contrast to CTX and CTM, do not modify them. Given two parents C_1 and C_2, CSX randomly distributes their constraints to two offspring C'_1 and C'_2. CSM applied to a parent C initializes a random model C_r using RHH, then calls CSX(C, C_r) and returns the first of the obtained offspring. The expected number of constraints in the offspring produced by CSX equals to the mean number of constraints in the parents, i.e., $E(|C'_1|) = E(|C'_2|) = (|C_1| + |C_2|)/2$; however, in an extreme case $C'_1 = \emptyset$ and $C'_2 = C_1 \cup C_2$ (or reversely). We allow for models that host no constraints as they can reduce the number of constraints in models they crossover with.

Algorithm 4 GCM. $\mathcal{N}(\mu, \sigma)$ Draws a Value From Normal Distribution With μ Mean and σ^2 Variance, REPLACE(c, a, a') Replaces a With a' in c

function GCM(C)
  $C' \leftarrow \emptyset$
  for $c \in C$ **do**
    $a \leftarrow$PICKCONSTANT(c)
    $a' \leftarrow \mathcal{N}(a, 1)$
    $c' \leftarrow$REPLACE(c, a, a')
    $C' \leftarrow C' \cup \{c'\}$
  return C'

To tune numerical constants in constraints, we apply Gaussian constant mutation (GCM, Algorithm 4). Given a parent C, GCM draws a constant a from each constraint $c \in C$ and replaces it with a constant a' drawn from the normal distribution $\mathcal{N}(a, 1)$. Constraints without constants remain unaffected.

The models produced by the above operators may contain trivially incorrect constraints. To overcome this, in every generation after breeding, we remove from the offspring the constraints $p(s) \lesseqgtr q(s)$, where $p(s) = \text{const}$ and $q(s) = \text{const}$, as they are either met or violated for all s. In addition to simplifying the offspring (and thus serving as a means of bloat control), this "reactivates" the constraint sets that would be otherwise violated for all $s \in S$.

C. Fitness Evaluation

As signaled in Section II, the constraint synthesis problem is underconstrained: typically there is an infinite number of constraint sets C that perfectly delineate a given set of examples of feasible states. Therefore, a fitness function needs to express user's preference concerning the desirable characteristics of C. We take into account two characteristics that seem to be universally desirable: the "tightness" of the constraints with respect to the region of feasible examples sampled by S, and (implicitly) the simplicity of constraints.

We draw a sample of *unlabeled artificial examples* in $\mathbb{R}^n$ that are intended to simulate the infeasible states, assuming that the states in close proximity of the feasible examples are likely to be feasible too. We estimate the extent of that proximity for each $s \in S$ individually, by calculating the distance threshold $t(s) = \min_{s' \in S, s \neq s'} d(s, s')$, where d is the Canberra distance between s and s'

$$d(s, s') = \sum_{i=1}^{n} \frac{|x_i - x'_i|}{|x_i| + |x'_i|},\ x_i \in s, x'_i \in s'.$$

We define the set S° of all states that are not further away from any $s \in S$ than the corresponding threshold $t(s)$, i.e., $S^\circ = \{s' : \exists_{s \in S} d(s, s') \leq t(s)\}$. Then, for each variable x_i we estimate its domain $\mathbb{D}_i$ by extending the range observed in S with the

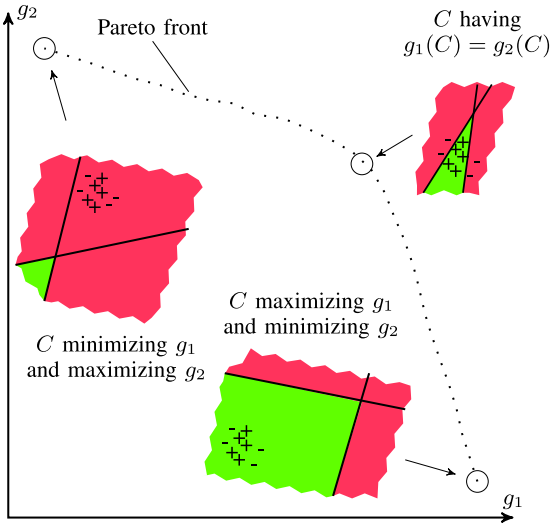

Fig. 2. Pareto front of g_1 and g_2 and exemplary MP models from different front parts; green and red colors refer to feasible and infeasible regions, resp.; pluses and minuses refer to feasible and unlabeled examples, resp.

thresholds of extreme examples

$$\mathbb{D}_i = \left[\min_{x_i \in s \in S}(x_i) - t_i(s_{\min}),\ \max_{x_i \in s \in S}(x_i) + t_i(s_{\max})\right]$$
$$s_{\min} = \operatorname*{argmin}_{s \in S} x_i,\ x_i \in s, \qquad s_{\max} = \operatorname*{argmax}_{s \in S} x_i,\ x_i \in s$$

where t_i is equivalent to t restricted to dimension i

$$t_i(s) = \min_{s' \in S, s' \neq s} \frac{|x_i - x_i'|}{|x_i| + |x_i'|},\ x_i \in s, x_i' \in s'.$$

Next, we calculate $\overline{S^\circ} = \Pi_{i=1}^n \mathbb{D}_i \setminus S^\circ$, the complement of S° with respect to the Cartesian product of variables' domains. Finally, we sample $\overline{S^\circ}$ for a set of unlabeled artificial examples U, such that $|U| = |S|$.[1]

We adjust the thresholds $t(s)$ locally because the examples of feasible states may be distributed nonuniformly. In this way, we surround each feasible example in S with a "t-margin." When merged for all examples, those margins form connected regions in S°. This is important, as only connected regions can be expressed by conjunctions of constraints in C. We use Canberra distance to bring the variables of different magnitude to the same range and avoid the low-magnitude variables to become negligible.

The resulting sample U is a nonparametric approximation of the unknown distribution of infeasible states. It is prepared once, before the run of GOCCS, and together with S forms the basis for fitness evaluation. To evaluate a set of constraints C, we calculate the numbers of true positive and true negative decisions made by the constraints in C

$$g_1(C) = |\{s \in S : \forall_{c \in C}\, c(s)\}|$$
$$g_2(C) = |\{s \in U : \exists_{c \in C}\, \neg c(s)\}|.$$

Both g_1 and g_2 are to be maximized, however, they favor models of different characteristics. Fig. 2 illustrates the trade-off between them for linear constraints. Any model with constraints delineating a region that is a superset of S maximizes g_1. This includes the trivial empty model $C = \emptyset$ (because it does not constrain solution space at all) and an infinite number of "excessively relaxed" models. In turn, g_2 is maximized by any model having at least one constraint and an empty intersection of the delineated region with U. This includes all inherently incoherent models (i.e., such that $\{s : \forall_{c \in C} c(s)\} = \emptyset$), and an infinite number of overly restrictive models. In general, the more constraints in a model, the more likely it scores low on g_1 and high on g_2. A model achieving high scores on both g_1 and g_2 is likely to correctly separate S from U.

We use g_1 and g_2 as a two-objective characterization of models and perform selection using the NSGA-II [4] algorithm. By promoting mutually nondominated models, NSGA-II maintains a Pareto front of approximately evenly distributed models in the $g_1 \times g_2$ criteria space. This lowers the risk of focusing on the criterion that happens simpler to optimize at the expense of the other, which would be likely if criteria g_1 and g_2 were aggregated into a scalar fitness function (by e.g., linear combination). Nevertheless, we resort to aggregation when picking the best-of-run model: from all models generated in a run, we select the one that maximizes $f(C) = g_1(C) + g_2(C)$. We choose this approach because, in absence of any other user's preference, the numbers of true-positives and true-negatives are equally important. Also, g_1 and g_2 are estimated from same-sized samples and thus can be justifiably added. In particular, any set of constraints perfectly separating S from U is guaranteed to achieve the optimal value of $f(C) = 2|S|$.

D. Post-Processing

When GP terminates, we simplify the best-of-run model C. To this aim, we remove from C the constraints $c \in C$ for which there does not exist a state $s \in \Pi_{i=1}^n \mathbb{D}_i$ such that: 1) $\neg c(s)$ and 2) no other constraint is violated for s, i.e., $\neg\exists c' \in C \setminus \{c\} : \neg c'(s)$. For such a c, all infeasible states it separates from the feasible ones are also separated by another constraint(s) in C, so removing c does not change the feasible region of the model. We verify the properties 1) and 2), on a validation set of 100 000 uniformly drawn states from the Cartesian product of variables' domains $V = \{s : s \sim \mathcal{U}(\Pi_{i=1}^n \mathbb{D}_i)\}$. The relatively large cardinality of V is intended to provide for sufficient confidence when testing for the properties, particularly when the number of variables is high (as per the *curse of dimensionality*, building a meaningful hypervolume requires an infeasible sample size [23]). Using even larger V for greater confidence is technically feasible, as this step involves only the best-of-run individual.

Finally, the constraints in the best-of-run model are symbolically simplified and transformed to expanded polynomial form[2] for better readability.

V. Experiment

A. Setup

We conduct three experiments. First, we tune the probabilities of engaging the operators from Section IV-B to maximize

[1] If real infeasible states are available, they may be included in U; however, we do not assume their availability for the reasons discussed in Section I.

[2] Using the MathNet.Symbolics library [24].

TABLE I
PARAMETERS OF GP ALGORITHM IN GOCCS

Parameter	Value
Population size	2000
Max generations	50
$MinConstr$, $MaxConstr$	1, 20
$MaxDepth$	3
Instruction sets	L, P
Initialization method	Ramped Half-and-Half [22]
Parent selection method	NSGA-II binary tournament [4]
Offspring selection method	NSGA-II post-selection [4]
Runs per benchmark	30, each with different seed

the final fitness. Then, we use that distribution in GOCCS and analyze the syntactic and semantic properties of the synthesized MP models. Finally, we compare GOCCS to its precursor, GenetiCS [1].

In all experiments, we employ the following benchmarks parameterized with the number of variables n.

$$\text{Ball}n: \quad d^2 \geq \sum_{i=1}^{n}(x_i - i)^2$$
$$\forall_{i=1}^{n} : x_i \in [i - 2d, i + 2d].$$
$$\text{Simplex}n: \quad \forall_{i=1}^{n}\forall_{j=i+1}^{n} : x_i \cot\frac{\pi}{12} - x_j \tan\frac{\pi}{12} \geq 0$$
$$\forall_{i=1}^{n}\forall_{j=i+1}^{n} : x_j \cot\frac{\pi}{12} - x_i \tan\frac{\pi}{12} \geq 0$$
$$\sum_{i=1}^{n} x_i \leq d$$
$$\forall_{i=1}^{n} : x_i \in [-1, 2 + d].$$
$$\text{Cube}n: \quad \forall_{i=1}^{n} : x_i \geq i$$
$$\forall_{i=1}^{n} : x_i \leq i + id$$
$$\forall_{i=1}^{n} : x_i \in [i - id, i + 2id]$$

where $d = 2.7$ to enforce use of noninteger values and so make the benchmarks closer to real-world problems. The benchmarks demand different capabilities from a constraint synthesis algorithm. Solving Ball$n$ requires one quadratic constraint involving all n variables, Cube$n$ requires $2n$ linear constraints involving one variable each, and Simplex$n$ requires $n(n-1)$ linear constraints of two variables each and one linear constraint of n variables. We construct a training set of feasible examples S (see Section II) by uniformly sampling m examples from the feasible region of each benchmark. We consider problem instances with $n \in [3, 7]$ variables and $m \in \{100, 200, 300, 400, 500\}$ examples for each n.

Table I shows the parameters of the GP algorithm common for all setups. Depending on the instruction set (L or P, Section IV-A), GOCCS synthesizes a linear or polynomial model, respectively. The experimental software is open-source.[3]

B. Parameter Tuning

To determine a well-performing mixture of search operators from Section IV-B, we run five GP setups, each engaging one of the search operators with probability 0.6 and the remaining four with probability 0.1 each. We name these setups after the dominating operator:

[3]http://www.cs.put.poznan.pl/tpawlak/link/?GOCCS

TABLE II
(a) RANKS OF MEANS OF BEST-OF-RUN FITNESS ON TRAINING SET WITH $m = 300$, BEST IN BOLD. (b) p-VALUES FOR FRIEDMAN'S TEST AND FOR INCORRECTLY JUDGING A SETUP IN A ROW AS BETTER THAN A SETUP IN A COLUMN, OBTAINED USING SYMMETRY TEST

(a)

Model type	CTX	CTM	CSX	CSM	GCM
Linear models	4.37	3.00	**1.67**	4.23	1.73
Polynomial models	4.27	2.33	**1.27**	4.60	2.53

(b)

Linear models, Friedman's p-value: 2.05×10^{-5}				
	CTX	CTM	CSX	CSM	GCM
CTX					
CTM	0.123		0.203		
CSX	**0.000**	0.141		**0.000**	1.000
CSM	0.999				
GCM	**0.000**	0.180		**0.000**	

Polynomial models, Friedman's p-value: 4.16×10^{-8}				
	CTX	CTM	CSX	CSM	GCM
CTX				0.978	
CTM	**0.007**			**0.001**	0.997
CSX	**0.000**	0.346		**0.000**	0.182
CSM					
GCM	**0.023**			**0.003**	

	CTX	CTM	CSX	CSM	GCM
CTX setup:	0.6	0.1	0.1	0.1	0.1
CTM setup:	0.1	0.6	0.1	0.1	0.1
CSX setup:	0.1	0.1	0.6	0.1	0.1
CSM setup:	0.1	0.1	0.1	0.6	0.1
GCM setup:	0.1	0.1	0.1	0.1	0.6

and compare them on all three benchmarks for $n \in [3, 7]$ and $m = 300$, which results in 15 problem instances of 300 feasible examples each (the results for other m lead to similar conclusions). Table II(a) shows the ranks of means of the best-of-run fitness. CSX setup achieves the best overall rank for both types of models, GCM and CTM setups are the runner-ups for linear and polynomial models, respectively. CTX and CSM setups are the worst for the linear and polynomial models, respectively. We hypothesize that CSX and GCM focus on exploitation, as the former is capable only of mixing the existing constraints and the latter fine-tunes the constants in constraints. In contrast, CTX and CSM may introduce new constraints and thus are more exploratory. Apparently, exploitation is essential to perform well here, probably because finding good constraints is easier with a series of small improvements than by replacing existing constraints with new ones.

Table II(b) shows the outcomes of Friedman's test [25] and *post-hoc* analysis using symmetry test [26] (conclusive p-values in bold). Superiority of CSX and GCM setups to CTX and CSM setups is significant for both linear and polynomial models, so we use CSX setup till the end of this paper.

C. Analysis of the Synthesized MP Models

Next, we assess the syntactic and semantic fidelity of the best-of-run models to the benchmark models from Section V-A.

Table III(a) shows the mean and 0.95-confidence interval of the difference between the numbers of constraints in the synthesized and benchmark models. The results are consistent across linear and polynomial models. For Ball$n$ the synthesized

models have significantly more constraints than the benchmark (two-tailed t-test at $\alpha = 0.05$) and those differences increase with the dimensionality n. For the linear models this is due to GOCCS attempting to approximate the actual quadratic constraint using multiple linear constraints. For polynomial models it may be hard to evolve a single constraint that involves the squares of all variables. For Simplex$n$ and Cube$n$ the differences in the numbers of constraints are not greater than 1 and decrease with n. We attribute this to the *curse of dimensionality* [27]: given the same number of feasible examples in S, a high-dimensional space offers more possibilities for placing the constraints so that they enclose S. Note that for Simplex3, Cube3, and Cube4, and for many values of m and/or model types, the two-tailed t-test concludes equality (underlined values) of the number of constraints in the synthesized models and the benchmarks.

Table III(b) shows the mean and 0.95-confidence interval of the difference between the total number of terms in all constraints of the synthesized model and the corresponding number in the benchmark, after the models have been transformed to the expanded polynomial form.[2] For all benchmarks, the synthesized linear models contain roughly the same number of terms as the benchmark (statistical equality confirmed by two-tailed t-test for all considered combinations of n and m). In contrast, the polynomial models contain significantly more terms than the respective benchmark models.

We also assess the similarity of expressions in the synthesized and actual constraints. To do so, we transform each constraint to the expanded polynomial form. Then, for each pair of weight vectors $\boldsymbol{w}_i^s$ and $\boldsymbol{w}_j^b$ of terms in the synthesized and actual constraints, respectively, we calculate the angle between them, i.e., $\alpha_{ij} = \arccos |\boldsymbol{w}_i^s \cdot \boldsymbol{w}_j^b / (\|\boldsymbol{w}_i^s\| \|\boldsymbol{w}_j^b\|)|$. Next, we formulate an *assignment problem*, in which the objective is to pair $\boldsymbol{w}_i^s$s and $\boldsymbol{w}_j^b s$ so that the mean angle is minimal and each vector is paired at least once

$$\begin{aligned} \min \frac{1}{N} \sum_{ij} \alpha_{ij} b_{ij} \qquad & \text{the mean of angles} \\ \text{subject to } \forall i : \sum_j b_{ij} \geq 1 \qquad & \boldsymbol{w}_i^s \text{ assigned to at least one } \boldsymbol{w}_j^b \\ \forall j : \sum_i b_{ij} \geq 1 \qquad & \boldsymbol{w}_j^b \text{ assigned to at least one } \boldsymbol{w}_i^s \\ \forall b_{ij} \in \{0, 1\} \qquad & \text{indicator of assigning } \boldsymbol{w}_i^s \text{ to } \boldsymbol{w}_j^b \end{aligned} \tag{1}$$

where N is the greater of the numbers of the synthesized and actual constraints. The optimized function varies from 0 (weight vectors pairwise parallel) to $\pi/2$ (weight vectors pairwise orthogonal). Notice that for linear models, α_{ij} in the weight space is the same as the angle between the constraints' hyperplanes in the solution space; for nonlinear models this relationship does not hold.

Table III(c) shows the mean and 0.95-confidence interval of the mean angles, i.e., the optimal value of the function minimized in (2). The angles for Ball$n$ benchmark are higher than for other benchmarks. For Simplex$n$ and Cube$n$, the angles start from about 0.3 rad for linear models and 0.7 rad for polynomial models. All mean angles are significantly different from zero with respect to one-tailed t-test at $\alpha = 0.05$.

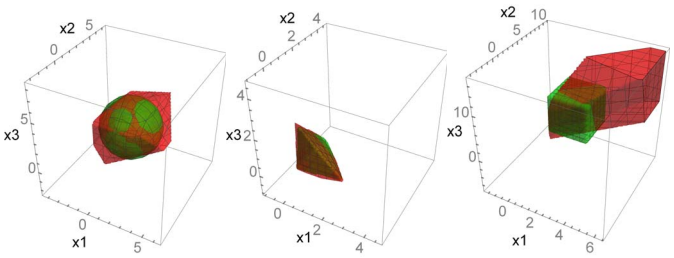

Fig. 3. Feasible regions of models: actual in green and best-found linear in red; from left: Ball3, Simplex3, and Cube3 benchmarks.

For all benchmarks the angle increases with n, which may be a sign of the curse of dimensionality: for high n, it is likely for a model to be correct while being expressed differently than the actual benchmark model. This is supported by the fact that larger sets of examples lead to decrease of angles.

Next, we focus on how well the feasible regions of the benchmarks are reconstructed in the synthesized models. Table III(d) shows the mean and 0.95-confidence interval of the sensitivity of the synthesized models on a test set of 100 000 examples uniformly sampled from the Cartesian product of domains of variables. An example is considered feasible if it falls into the feasible region of a benchmark model, and infeasible otherwise. Sensitivity is equivalent to $g_1(C)$ normalized to [0, 1] range by dividing by $|S|$ (see Section IV-C). The values vary across the benchmarks, in $[0.69, 0.83]$ for Ball$n$ and in $[0.47, 0.58]$ for Cube$n$, while remaining largely independent from m and n. In contrast, for Simplex$n$ sensitivity varies strongly in the entire range $[0, 1]$ and noticeably decreases with n. Recall that the number of constraints in Simplex$n$ grows quadratically with n, and that causes rapid decrease of the ratio of the hypervolume of the feasible region to the hypervolume of the Cartesian product of variables' domains. This makes true positives very rare, even in our relatively large test set.

Table III(e) shows test-set specificity, i.e., $g_2(C)$ normalized to [0, 1] range by dividing by $|U|$. Overall, specificity is high and in most cases well above 0.9. Moreover, for Simplex$n$ and a few pairs of m and n, where $n \geq 6$, the one-tailed t-test concludes equality to 1. Specificity clearly increases with n, likely due to the number of true negatives increasing with n.

We also qualitatively assess the synthesized models by comparing them visually to the benchmark models. Fig. 3 shows the visualizations of the feasible regions of the actual and the best models found in all runs for $n = 3$, $m = 500$ and linear instruction set (L). We omit the polynomial models because of being worse than the corresponding linear ones. Clearly, the feasible region of Simplex3 is reconstructed most faithfully. We attribute this to higher density of distribution of feasible examples than for the remaining benchmarks, as Simplex3 has the smallest volume of the feasible region of all benchmarks for $n = 3$. The synthesized model for Ball3 does not resemble a ball, however, its center of mass is close to the actual ball center and its extreme points are at a distance close to the radius of the actual ball. Reconstruction of Cube3

TABLE III

(a) MEAN DIFFERENCE IN NUMBERS OF CONSTRAINTS AND (b) TERMS IN CONSTRAINTS, BETWEEN THE SYNTHESIZED AND THE ACTUAL MODELS. (c) MEAN ANGLE BETWEEN THE CORRESPONDING CONSTRAINTS IN THE SYNTHESIZED AND THE ACTUAL MODELS. (d) MEAN SENSITIVITY AND (e) SPECIFICITY OF THE SYNTHESIZED MODELS ON TEST SET. HEATMAPS REFLECT VALUES FROM BEST (GREEN) TO WORST (RED). BARS REFLECT 0.95-CONFIDENCE INTERVALS (CELL HEIGHT REFLECTS 1, LARGER VALUES TRUNCATED). UNDERLINED VALUES DIFFER INSIGNIFICANTLY (a) AND (b) FROM 0 WITH RESPECT TO TWO-TAILED t-TEST AT $\alpha = 0.05$, (d) AND (e) FROM 1 WITH RESPECT TO ONE-TAILED t-TEST AT $\alpha = 0.05$

(a)

	$m\backslash n$	Ball$n$ 3	4	5	6	7	Simplex$n$ 3	4	5	6	7	Cube$n$ 3	4	5	6	7
Linear models	100	6.43	8.43	10.30	10.97	11.60	−1.47	−6.70	−13.37	−23.80	−35.63	−1.03	−1.73	−2.27	−3.37	−4.47
	200	7.43	10.03	11.43	13.17	13.07	−1.27	−5.90	−13.47	−22.80	−35.03	−0.40	−0.70	−1.67	−2.67	−3.70
	300	8.30	10.87	12.10	12.70	13.17	−1.67	−5.83	−13.33	−23.10	−35.37	0.07	−0.70	−1.40	−2.80	−4.07
	400	9.07	10.87	13.03	13.13	12.67	−1.57	−6.07	−12.67	−22.93	−35.20	0.03	−0.93	−0.83	−2.93	−3.47
	500	9.03	10.97	12.93	12.97	12.77	−1.53	−5.97	−13.23	−23.20	−35.83	0.20	−0.30	−1.37	−1.90	−3.00
	$m\backslash n$	3	4	5	6	7	3	4	5	6	7	3	4	5	6	7
Poly. models	100	7.13	9.30	10.87	11.80	12.73	0.40	−5.37	−12.07	−21.70	−34.70	−0.70	−1.13	−1.70	−3.03	−4.57
	200	7.67	10.87	12.03	13.67	13.97	0.13	−3.73	−11.97	−21.83	−34.20	0.27	−0.37	−1.67	−2.77	−3.13
	300	8.63	11.70	13.73	14.03	14.30	0.07	−4.40	−11.37	−21.57	−34.53	0.57	0.03	−1.30	−2.37	−3.13
	400	8.57	12.37	13.53	14.43	14.07	−0.20	−3.90	−11.03	−21.37	−34.63	1.23	0.07	−1.17	−1.83	−3.13
	500	9.30	11.63	13.60	14.73	14.33	0.10	−4.83	−11.47	−21.30	−33.53	0.67	−0.17	−0.87	−1.23	−2.63

(b)

	$m\backslash n$	Ball$n$ 3	4	5	6	7	Simplex$n$ 3	4	5	6	7	Cube$n$ 3	4	5	6	7
Linear models	100	0.00	0.00	0.00	0.00	−0.03	0.00	0.00	0.00	−0.03	−0.07	−0.07	−0.03	−0.03	−0.03	−0.10
	200	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	−0.03	−0.07	0.00	0.00	0.00	−0.07	−0.10
	300	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	−0.03	−0.07	0.00	0.00	−0.07	−0.03	−0.10
	400	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	−0.03	0.00	−0.03	0.00
	500	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	−0.03	0.00
	$m\backslash n$	3	4	5	6	7	3	4	5	6	7	3	4	5	6	7
Poly. models	100	4.27	6.73	9.30	12.17	14.67	5.47	6.10	9.53	10.37	10.93	3.17	5.13	6.47	6.57	8.23
	200	4.23	7.33	11.30	13.43	15.60	5.20	9.27	9.10	10.87	14.63	3.93	5.13	6.63	7.40	9.73
	300	4.47	8.63	11.57	14.10	15.43	4.67	7.57	11.13	12.33	12.77	4.03	6.37	7.80	8.77	9.17
	400	4.97	8.47	10.93	14.73	15.97	4.53	9.17	10.63	10.77	13.43	4.83	5.83	7.43	8.70	9.40
	500	5.03	8.57	11.53	14.77	17.00	4.73	7.33	10.83	12.90	14.23	4.77	6.57	7.60	9.43	9.80

(c)

	$m\backslash n$	Ball$n$ 3	4	5	6	7	Simplex$n$ 3	4	5	6	7	Cube$n$ 3	4	5	6	7
Linear models	100	1.01	1.18	1.29	1.36	1.40	0.40	0.50	0.50	0.62	0.67	0.58	0.56	0.64	0.69	0.73
	200	1.00	1.18	1.29	1.35	1.40	0.38	0.44	0.57	0.61	0.64	0.47	0.58	0.64	0.68	0.71
	300	0.99	1.17	1.29	1.35	1.38	0.36	0.42	0.51	0.63	0.66	0.50	0.60	0.64	0.70	0.70
	400	1.00	1.18	1.28	1.35	1.39	0.34	0.43	0.50	0.60	0.67	0.47	0.57	0.61	0.71	0.70
	500	1.00	1.17	1.28	1.34	1.39	0.33	0.44	0.53	0.58	0.71	0.52	0.60	0.63	0.71	0.72
	$m\backslash n$	3	4	5	6	7	3	4	5	6	7	3	4	5	6	7
Poly. models	100	1.09	1.25	1.30	1.35	1.38	0.79	0.81	0.81	0.81	0.87	0.69	0.77	0.78	0.85	0.86
	200	1.06	1.22	1.31	1.33	1.37	0.75	0.80	0.80	0.80	0.90	0.71	0.77	0.79	0.82	0.87
	300	1.08	1.23	1.30	1.34	1.37	0.74	0.78	0.78	0.86	0.88	0.71	0.82	0.83	0.85	0.84
	400	1.07	1.23	1.29	1.34	1.37	0.72	0.82	0.81	0.80	0.93	0.72	0.77	0.81	0.84	0.84
	500	1.07	1.21	1.29	1.35	1.38	0.73	0.77	0.82	0.84	0.88	0.70	0.75	0.83	0.85	0.85

(d)

	$m\backslash n$	Ball$n$ 3	4	5	6	7	Simplex$n$ 3	4	5	6	7	Cube$n$ 3	4	5	6	7
Linear models	100	0.71	0.76	0.75	0.75	0.78	0.92	0.92	0.91	0.53	0.05	0.47	0.48	0.54	0.53	0.53
	200	0.77	0.76	0.77	0.78	0.76	0.96	0.96	0.97	0.40	0.00	0.51	0.52	0.54	0.51	0.58
	300	0.77	0.81	0.80	0.80	0.83	0.97	0.97	0.96	0.41	0.03	0.48	0.51	0.51	0.54	0.55
	400	0.77	0.80	0.80	0.82	0.80	0.97	0.98	0.98	0.38	0.00	0.50	0.51	0.51	0.53	0.53
	500	0.80	0.81	0.80	0.80	0.81	0.97	0.97	0.96	0.57	0.07	0.51	0.48	0.54	0.51	0.58
	$m\backslash n$	3	4	5	6	7	3	4	5	6	7	3	4	5	6	7
Poly. models	100	0.69	0.75	0.72	0.76	0.76	0.91	0.90	0.94	0.50	0.07	0.48	0.51	0.53	0.56	0.51
	200	0.75	0.78	0.76	0.76	0.78	0.95	0.95	0.96	0.35	0.03	0.54	0.49	0.53	0.55	0.58
	300	0.79	0.79	0.79	0.81	0.79	0.95	0.95	0.97	0.50	0.03	0.50	0.49	0.55	0.54	0.51
	400	0.82	0.82	0.80	0.82	0.81	0.96	0.96	0.98	0.42	0.00	0.51	0.55	0.53	0.55	0.56
	500	0.81	0.80	0.81	0.83	0.81	0.96	0.96	0.94	0.57	0.07	0.49	0.48	0.52	0.52	0.58

(e)

	$m\backslash n$	Ball$n$ 3	4	5	6	7	Simplex$n$ 3	4	5	6	7	Cube$n$ 3	4	5	6	7
Linear models	100	0.90	0.91	0.94	0.96	0.97	0.98	0.98	1.00	1.00	0.99	0.88	0.90	0.91	0.92	0.93
	200	0.94	0.94	0.95	0.97	0.98	0.98	0.99	1.00	1.00	1.00	0.90	0.91	0.90	0.93	0.94
	300	0.96	0.95	0.96	0.97	0.98	0.98	0.99	1.00	1.00	1.00	0.92	0.93	0.91	0.92	0.94
	400	0.95	0.95	0.96	0.97	0.98	0.99	0.99	1.00	1.00	1.00	0.90	0.89	0.93	0.93	0.95
	500	0.96	0.95	0.96	0.97	0.98	0.99	0.99	1.00	1.00	1.00	0.91	0.92	0.92	0.94	0.95
	$m\backslash n$	3	4	5	6	7	3	4	5	6	7	3	4	5	6	7
Poly. models	100	0.92	0.92	0.95	0.97	0.98	0.97	0.98	1.00	1.00	1.00	0.88	0.90	0.93	0.93	0.94
	200	0.93	0.94	0.96	0.98	0.99	0.98	0.99	1.00	1.00	1.00	0.89	0.93	0.93	0.94	0.96
	300	0.95	0.94	0.96	0.97	0.99	0.98	0.99	1.00	1.00	0.99	0.91	0.92	0.92	0.94	0.96
	400	0.94	0.95	0.96	0.98	0.98	0.98	0.99	1.00	1.00	1.00	0.92	0.92	0.93	0.94	0.96
	500	0.96	0.96	0.96	0.98	0.98	0.98	0.99	1.00	1.00	1.00	0.91	0.93	0.93	0.95	0.96

is least accurate: the synthesized constraints are not orthogonal to axes and the feasible region extends beyond the actual feasible region. We attribute this to the distribution of models searched by GOCCS: as a working set of constraints (an individual) orthogonal to axes is unlikely to perform well until it bounds the feasible region in all directions, i.e., until it has

n pairs of opposite constraints, it is unlikely to survive in the population long enough for its constants to become well-tuned.

To sum up, when it comes to complexity of synthesized models, the main phenomenon that shapes the outcomes of GOCCS is clearly the curse of dimensionality. Its detrimental effects can be addressed with larger training sets, but only to a certain degree. The number of synthesized constraints does not seem to vary significantly between linear and polynomial models, however, the total number of terms in constraints is significantly smaller for the former. Linear models with few terms are easy to interpret and solvable in polynomial time, so they should be preferred in general. However, polynomial models, though NP-hard to solve and often more complex, can be occasionally more desirable, especially when the domain knowledge suggests their adequacy. Last but not least, the varying complexity of synthesized models does not seem to impact their correctness: GOCCS systematically maintains moderate-to-high sensitivity and high specificity.

D. Importance of Infeasible Examples

In this experiment, we compare GOCCS to a state of the art approach for two-class constraint synthesis problem, GenetiCS [1], and look for a minimum number of infeasible examples required by the latter to synthesize constraints of similar accuracy as GOCCS. To this aim, we apply GOCCS to a training set of $m_f = 500$ feasible examples, and GenetiCS to the same training set of $m_f = 500$ feasible examples and a varying number $m_i \in \{100, 200, 300, 400, 500\}$ of infeasible examples. GenetiCS uses the same representation of constraints and GP operators as GOCCS, but relies on a different fitness function and selection operator (see Section III). GenetiCS is thus the most similar state-of-the-art competitor for GOCCS.

Table IV shows the mean and 0.95-confidence interval of classification accuracy calculated on the same test set as in the previous section. GOCCS achieves the highest accuracy in four and five out of 15 problems for linear and polynomial models, respectively. Accuracy of GenetiCS increases with m_i as expected, and for $m_i = m_f = 500$ it achieves the highest accuracy in 10 out of 15 problems for both linear and polynomial of models. GOCCS and GenetiCS rank similarly when the latter is provided with $m_i = 300$ infeasible examples (the bottom of Table IV). The p-values of one-tailed Wilcoxon signed rank test [25] for significance of difference of pairs of GOCCS and particular GenetiCS setups (see Table IV) are inconclusive for $m_i \leq 300$ except $m_i = 100$ and polynomial models for which GOCCS is better. For $m_i = 500$ GenetiCS is better for both types of models.

In the conclusion, even though GOCCS does not make use of infeasible examples altogether, it is not worse than GenetiCS supplied with roughly one infeasible example for every two of feasible examples, and tends to be better when GenetiCS uses smaller fractions of infeasible examples. This outcome strongly favors GOCCS for many real-world problems, where, as argued in Section I, infeasible examples are few and far between (note that artificially increasing the above ratio by reducing the number of feasible examples will be detrimental due to curse of dimensionality).

TABLE IV
MEAN ACCURACY ON TEST SET, BEST IN BOLD. BARS REFLECT 0.95-CONFIDENCE INTERVALS (CELL HEIGHT REFLECTS 0.1). THE p-VALUES OF ONE-TAILED WILCOXON SIGNED RANK TEST OF DIFFERENCES BETWEEN GOCCS AND PARTICULAR GENETICS SETUPS

		GOCCS	GenetiCS				
	Problem m_i:	0	100	200	300	400	500
Linear models	Ball3	0.95	0.93	0.96	0.96	0.97	**0.97**
	Ball4	0.95	0.92	0.95	0.96	0.97	**0.97**
	Ball5	0.96	0.92	0.95	0.96	0.97	**0.98**
	Ball6	0.97	0.91	0.95	0.96	0.97	**0.97**
	Ball7	**0.98**	0.91	0.95	0.96	0.97	0.97
	Simplex3	0.99	0.97	0.99	0.99	0.99	**0.99**
	Simplex4	0.99	0.97	0.98	0.99	0.99	**0.99**
	Simplex5	**1.00**	0.97	0.98	0.99	0.99	0.99
	Simplex6	**1.00**	0.96	0.98	0.98	0.99	0.99
	Simplex7	**1.00**	0.97	0.98	0.99	0.99	0.99
	Cube3	0.90	0.97	0.99	0.99	0.99	**0.99**
	Cube4	0.92	0.96	0.98	0.98	0.99	**0.99**
	Cube5	0.92	0.95	0.97	0.98	**0.99**	0.98
	Cube6	0.94	0.94	0.97	0.98	0.98	**0.98**
	Cube7	0.95	0.94	0.97	0.98	0.98	**0.98**
	Rank:	3.80	5.80	4.47	3.27	2.33	1.33
	Wilcoxon's p-value:		0.19	0.82	0.16	**0.02**	**0.02**
Polynomial models	Ball3	0.95	0.94	0.96	0.96	0.97	**0.97**
	Ball4	0.96	0.93	0.95	0.96	0.97	**0.97**
	Ball5	0.96	0.92	0.95	0.97	0.97	**0.97**
	Ball6	**0.98**	0.90	0.95	0.97	0.97	0.97
	Ball7	**0.98**	0.91	0.95	0.96	0.97	0.97
	Simplex3	0.98	0.96	0.98	0.99	0.99	**0.99**
	Simplex4	0.99	0.96	0.98	0.98	0.99	**0.99**
	Simplex5	**1.00**	0.96	0.98	0.98	0.99	0.99
	Simplex6	**1.00**	0.97	0.98	0.99	0.99	0.99
	Simplex7	**1.00**	0.97	0.98	0.99	0.99	0.99
	Cube3	0.89	0.96	0.99	0.99	0.99	**0.99**
	Cube4	0.92	0.95	0.97	0.98	0.98	**0.99**
	Cube5	0.93	0.94	0.97	0.98	0.98	**0.98**
	Cube6	0.95	0.93	0.96	0.97	0.98	**0.98**
	Cube7	0.96	0.92	0.96	0.97	0.98	**0.98**
	Rank:	3.47	5.80	4.67	3.40	2.33	1.33
	Wilcoxon's p-value:		**0.01**	0.97	0.67	0.11	**0.02**

VI. CASE STUDY: WINE QUALITY

To illustrate GOCCS in practical settings, we use it to synthesize constraints for a real-world problem of modeling wine quality, combine them with an objective function, optimize the resulting model to determine the optimal proportions of wine ingredients, and compare the results to a baseline method. We use the wine quality database [28] composed of 1599 red and 4898 white wine examples, each described using eleven variables relevant to taste. Each example is supplemented with quality grade Q, calculated as the median of grades assigned by at least three sensory assessors, varying in range 0 (bad) to 10 (excellent). After removing duplicates from the original database, we obtain a dataset of 1359 examples of red wine and a dataset of 3961 examples of white wine, summarized in Table V. As the original database specification does not cite domains of variables, we calculate them as described in Section IV-C and then crop them to $[0, \infty)$, as negative variables have no physical interpretation.

We build separate models for red wine and white wine, using the above sets of examples as feasible states. (Section IV). We use GOCCS with the linear instruction set

TABLE V
VARIABLES IN WINE QUALITY DATASET [28]; DOMAINS CALCULATED AS IN SECTION IV-C (EXCEPT FOR Q), NEGATIVE VALUES REMOVED

		Red wine				White wine			
Variable	Meaning	Min	Mean	Max	Domain	Min	Mean	Max	Domain
FA	Fixed acidity g(tartaric acid)$/dm^3$	4.60	8.31	15.90	[4.589, 15.910]	3.80	6.84	14.20	[3.787, 14.292]
VA	Volatile acidity g(acetic acid)$/dm^3$	0.12	0.53	1.58	[0.000, 1.666]	0.08	0.28	1.10	[0.080, 1.145]
CA	Citric acid g/dm^3	0.00	0.27	1.00	[0.000, 1.000]	0.00	0.33	1.66	[0.000, 1.660]
RS	Residual sugar g/dm^3	0.90	2.52	15.50	[0.757, 15.503]	0.60	5.91	65.80	[0.523, 66.151]
C	Chlorides g(sodium chloride)$/dm^3$)	0.01	0.09	0.61	[0.000, 0.612]	0.01	0.05	0.35	[0.000, 0.416]
FSD	Free sulfur dioxide mg/dm^3	1.00	15.89	72.00	[1.000, 72.029]	2.00	34.89	289.00	[1.800, 289.327]
TSD	Total sulfur dioxide mg/dm^3	6.00	46.83	289.00	[6.000, 289.019]	9.00	137.19	440.00	[8.947, 440.091]
D	Density g/dm^3	0.99	0.997	1.00	[0.990, 1.004]	0.99	0.994	1.04	[0.987, 1.053]
pH	pH	2.74	3.31	4.01	[2.719, 4.010]	2.72	3.20	3.82	[2.716, 3.821]
S	Sulfates g(potassium sulfate)$/dm^3$	0.33	0.66	2.00	[0.273, 2.005]	0.22	0.49	1.08	[0.198, 1.089]
A	Alcohol $vol.\%$	8.40	10.43	14.90	[8.400, 14.931]	8.00	10.59	14.20	[8.000, 14.205]
Q	Quality (dependent variable)	3.00	5.62	8.00	[0.000, 10.000]	3.00	5.85	9.00	[0.000, 10.000]

(L) and parameters from Table I. 30 runs of the method with different seeds result with the best found set of constraints having fitness $f = 2668$ for red wine (the optimum, for a hypothetical model classifying all examples correctly, is 2718) and $f = 7880$ for white wine (the optimum is 7922).

To supplement the synthesized models with objective functions, we conduct least-squares quadratic regression of the dependent variable Q with respect to independent variables from Table V. From the resulting regression function we remove all terms whose coefficients are insignificant with respect to t-test and start over. The regression functions obtained in this way turn out to be not concave for both red and white wine. As most quadratic programming (QP) solvers assume concavity, we remove the minimal subset of terms that prevents a function from being concave and start over. The resulting functions have coefficients of determination $r^2 = 0.39$ and 0.33, respectively.

The synthesized red wine QP model is

$$\begin{aligned}\max\ & -0.01015\mathrm{FA}^2 - 0.7481\mathrm{VA}^2 - 1.681\mathrm{S}^2 + 0.1639\mathrm{FA} + \\ & - 1.836\mathrm{C} - 0.00197\mathrm{TSD} - 0.6832\mathrm{pH} + 3.819\mathrm{S} + 0.3003\mathrm{A} \\ & + 2.873\end{aligned}$$

$$\begin{aligned}\text{subject to}\quad & -\mathrm{VA} + \mathrm{C} \le 0 \\ & 1.669\mathrm{C} - \mathrm{S} \le 0 \\ & \mathrm{VA} + \mathrm{pH} + \mathrm{S} \le 5.449 \\ & \mathrm{RS} - 3.455\mathrm{D} - \mathrm{pH} \le 2.712 \\ & 3.192\mathrm{VA} - \mathrm{RS} - \mathrm{FSD} \le 0 \\ & 2.384\mathrm{CA} - \mathrm{RS} - \mathrm{C} \le 0 \\ & \mathrm{VA} + 2.384\mathrm{RS} + 2.544\mathrm{C} + \mathrm{TSD} - 5.717\mathrm{D} - 16.3\mathrm{A} \le 11.95 \\ & 0.2764\mathrm{FA} + 1.492\mathrm{VA} + 0.3844\mathrm{RS} + \mathrm{C} + \mathrm{D} - \mathrm{pH} + \\ & \qquad 4.712\mathrm{S} \le 9.153 \\ & 2\mathrm{FA} + \mathrm{CA} + 1.686\mathrm{RS} + 0.5031\mathrm{C} + 1.384\mathrm{FSD} - \mathrm{TSD} - \\ & \qquad 4.384\mathrm{pH} + 2.384\mathrm{S} - 2\mathrm{A} \le 3.415 \\ & \mathrm{FA} - 10.17\mathrm{VA} + 4.384\mathrm{CA} - 2.384\mathrm{RS} - \mathrm{C} + 2.384\mathrm{FSD} + \\ & \qquad \mathrm{TSD} + 2.384\mathrm{pH} - 2.384\mathrm{S} \ge 2.384 \\ & 2.384\mathrm{FA} + 0.4379\mathrm{VA} + \mathrm{CA} - 0.3844\mathrm{RS} + 6.123\mathrm{C} + \\ & \qquad 4.136\mathrm{FSD} - 0.3844\mathrm{TSD} - \mathrm{D} - 2\mathrm{S} - 0.6764\mathrm{A} \ge 0.1357.\end{aligned}$$

The objective function is easy to interpret: high amount of alcohol positively contributes to wine quality, amounts of tartaric acid of 8.07 g/dm^3 and sulfates of 1.14 g/dm^3 are optimal, and other ingredients have negative impact on quality. The first constraint states that concentration of sodium dioxide (salt, C) must be not greater than the concentration of acetic acid. The second constraint captures an analogous relationship between (scaled) C and the concentration of sulfates S. The third constraint imposes an upper bound on the sum of concentrations of acetic acid and sulfates in relation to pH: for acidic wines with low pH, higher amounts of acetic acid and sulfates are allowed than for the wines with high pH. Given that D ≈ 1 for all wines, the forth constraint can be rewritten as RS $-$ pH ≤ 6.167, implying that increasing concentration of sugar must be accompanied with increasing value of pH (decreasing acidity) for a wine to remain within the feasible region (and thus essentially pass as a wine). The next two constraints relate the concentrations of sugar, acetic acid, free sulfur dioxide (FSD), citric acid, and sodium chloride. Interpretation of the remaining constraints is more complex.

The best QP model synthesized for white wine is

$$\begin{aligned}\max\ & -0.02784\mathrm{FA}^2 - 1.234\mathrm{CA}^2 - 0.00007981\mathrm{FSD}^2 \\ & - 0.00002581\mathrm{TSD}^2 + 0.4588\mathrm{FA} - 1.332\mathrm{VA} + 1.259\mathrm{CA} \\ & + 0.06963\mathrm{RS} + 0.01358\mathrm{FSD} + 0.006554\mathrm{TSD} - 144.9\mathrm{D} \\ & + 0.8708\mathrm{pH} + 0.6428\mathrm{S} + 0.1996\mathrm{A} + 141.8\end{aligned}$$

$$\begin{aligned}\text{subject to}\quad & \mathrm{D} \le 1.0052 \\ & \mathrm{VA} - \mathrm{C} \ge 0 \\ & \mathrm{CA} + \mathrm{C} - \mathrm{D} \le 0 \\ & -\mathrm{VA} + 0.5283\mathrm{CA} - \mathrm{RS} - 0.2167\mathrm{D} + 2\mathrm{pH} + 2\mathrm{A} \ge 0.6499 \\ & -4.786\mathrm{FA} + 4.893\mathrm{C} + 0.107\mathrm{TSD} - 2.893\mathrm{pH} + \mathrm{S} + \mathrm{A} \le 2 \\ & -\mathrm{FA} - \mathrm{VA} - 0.9435\mathrm{CA} + 2\mathrm{RS} + 2\mathrm{C} + \mathrm{FSD} - \mathrm{TSD} + \\ & \qquad 0.5283\mathrm{D} \le 0.7209 \\ & -2.8\mathrm{VA} - 1.951\mathrm{CA} + 0.051\mathrm{RS} + \mathrm{C} - 0.0565\mathrm{TSD} + \\ & \qquad 1.528\mathrm{D} + 2.264\mathrm{pH} + 0.04894\mathrm{S} + \mathrm{A} \ge 1.127 \\ & 0.2791\mathrm{FA} - 0.1927\mathrm{VA} - 3\mathrm{CA} + 0.2791\mathrm{RS} + \mathrm{C} \\ & \qquad + 1.528\mathrm{FSD} - \mathrm{TSD} - 1.528\mathrm{D} + 0.5283\mathrm{S} - \mathrm{A} \le 0.1316.\end{aligned}$$

TABLE VI
OPTIMAL SOLUTIONS FOR THE WINE MODELS AND THE MOST SIMILAR WINES WITH RESPECT TO CANBERRA DISTANCE.[4] UNDERLINING SIGNALS CHANGES RESULTING FROM ADDING EXTRA CONSTRAINTS

	Red wine					White wine	
	Optimal solution	Optimal w/extra constraints	Most similar	Optimal in convex hull	Most similar	Optimal solution	Most similar
FA	8.070	8.070	8.500	11.147	10.400	8.253	6.800
VA	0.000	0.000	0.340	0.362	0.410	0.080	0.150
CA	0.387	0.434	0.400	0.473	0.550	0.525	0.410
RS	3.675	4.145	4.700	4.332	3.200	35.387	12.900
C	0.000	0.000	0.055	0.084	0.076	0.000	0.044
FSD	14.917	2.155	3.000	19.905	22.000	80.237	79.500
TSD	6.000	6.000	9.000	53.379	54.000	141.983	182.000
D	0.998	0.996	0.997	0.995	1.000	0.987	0.997
pH	2.719	2.719	3.380	3.204	3.150	3.821	3.240
S	1.136	1.136	0.660	0.861	0.890	1.089	0.780
A	14.931	14.931	11.600	14.183	9.900	14.205	10.200
Q (objective)	8.318	8.318	7.000	7.193	6.000	11.194	6.000
Distance to optimal			2.992		0.632		2.534
Percentile in dataset			99%		86%		79%

Given complete wine quality models, it is tempting to ask: what is the characteristics of an ideal wine? To answer this question, we optimize the models with Gurobi solver [29] and obtain the solutions shown in Table VI. For the red wine, the optimal concentration of FSD turns out to be greater than the total concentration of sulfur dioxide (TSD). Because it should hold by definition that FSD $\leq$ TSD, we determined that in the dataset $\min \text{TSD} - \text{FSD} = 3$ and $\max \text{FSD}/\text{TSD} = 0.8571$, and added the following constraints to the QP model:

$$-\text{FSD} + \text{TSD} \geq 3$$
$$\text{FSD} - 0.8571\text{TSD} \leq 0$$

and reran the QP solver. The resulting optimal solution (the third column in Table VI) has the same quality, while featuring noticeably lower FSD and slightly different density and concentrations of citric acid and residual sugar.

In the white wine model, the relation of FSD and TSD is modeled by the fifth and the last of the synthesized constraints. As a result, the QP solver produces an optimal solution that does not violate the domain knowledge.

Table VI juxtaposes the obtained optimal solutions with the examples from the respective datasets that are most similar with respect to Canberra distance.[4] For red wine, the quality of the most similar example is 7—smaller than the estimated quality of 8.318, however, not worse than the quality of 99% of examples in the dataset. For white wine, the most similar example has quality of 6, again smaller than the estimated quality of 11.194, however, not worse than 79% of examples. We attribute the not so accurate prediction of quality to imperfect fitting of regression functions, signaled by relatively low r^2 coefficients. Better prediction performance requires higher-order or nonconcave objective functions, which, however, may prevent use of QP solvers.

Although these optimal solutions are plausible, their practical applicability cannot be *guaranteed* based on the above evidence. Thus, assuming for the sake of argument that feasible wine compositions are expressible using linear constraints, we ask: is it possible to synthesize a set of linear constraints that is *guaranteed* to embrace only feasible solutions? Obviously, the facets of a *convex hull*, the smallest convex superset of the dataset, define such constraints.

Calculating a convex hull is, however, exponential in time and space with respect to dimensionality, due to the exponential number of resulting constraints. We calculated the convex hull of the red wine dataset using the qhull tool [30] which took about 41 h on a Core i7-5960X CPU, consumed over 40GB of RAM and resulted in 102 911 486 constraints. We were unable to calculate the convex hull for white wine dataset due to insufficient computational resources. Optimizing the convex hull-based model for the red wine with respect to the same objective function as above took Gurobi solver [29] 67 h and consumed 118GB of RAM. Comparing these numbers to the average of 27 min per each of 30 runs of GOCCS, 11 constraints of the best model and negligible time needed to optimize this model, clearly points to the advantages of our method in terms of required computational resources, interpretability of the resulting model and ease of its optimization.

The fifth column in Table VI shows the optimal solution to the convex hull-based QP red wine model. The quality of this solution is 7.193, 1.125 short of the optimal solution to the GOCCS red wine model. This is expected, as the convex hull-based QP model, in contrast to GOCCS QP model, by definition encloses the dataset in the tightest way possible, and leaves no margin around the examples. For the same reason, it is much closer to the examples in the dataset than the solution to the GOCCS QP model (see the last row of Table VI). In effect, this solution is "credible" by having quality similar to the best already known solutions from the dataset and requiring only small changes in the existing business process to be applied. However, further improvements in this process cannot be done solely by means of the convex hull, as it remains unchanged after including this solution in the dataset. GOCCS is free from this limitation, as it produces models with margins surrounding examples and thus allowing for improved solutions.

VII. DISCUSSIONS AND CONCLUSION

The experimental evaluation in Section V and the case study in Section VI demonstrate that GOCCS proves effective for synthesis of constraints in a wide range of operating conditions, for problems with various numbers of variables, different distributions of variables, and requiring constraints that vary in number and complexity. The ability of synthesizing constraints from feasible examples only is advantageous, as infeasible states may occur rarely in practice and acquiring a large enough number of them can be costly. GOCCS fares well without them, as evidenced by the comparison with GenetiCS that requires also a sample of infeasible states (Section V-D).

Accurate placement of the boundary between the feasible and infeasible states is particularly important for model optimization, as this is, where the optimal solutions are often located. Loose constraints that overestimate the feasible region may lead to optimal solutions that are practically unrealizable. Too tight constraints, on the other hand, underestimate that

[4]For the red wine, distance to solution to the model with extra constraints.

region and may result in suboptimal solutions. In absence of information on the margins between feasible examples and the constraints, in GOCCS we infer them from the distribution of feasible examples (see Section IV-C). This leads to tight constraints, where feasible examples are densely distributed and more relaxed ones, where they are sparse, which is consistent with the principles of statistical sampling stating that estimates are more precise when the available sample is dense.

The two-objective characterization of desirable properties of constraints seems to not only yield constraints that quite accurately delineate the feasible region, but, in combination with additional checks, naturally keeps complexity at bay. This advantage becomes particularly evident when confronting GOCCS with the naive construction of constraints from the convex hull of feasible examples, which produces exorbitant number of constraints and requires impractical amount of computing resources (Section VI).

The LP/NLP representation of models synthesized by GOCCS is easy to interpret and inspect by humans and directly usable in off-the-shelf solvers. This in turn makes it easy to adjust or correct models, as demonstrated by augmenting the QP model of red wine with extra constraints in Section VI. Thanks to convex objective functions, large LP and QP models can be solved efficiently in polynomial time. This is a qualitative advantage over, e.g., CP models that are NP-hard to solve. Also LP/NLP solvers support both real and integer variables, while CP solvers operate in principle on discrete variables and handle real variables by discretizing them or branching their domains, in either way returning intervals that only approximately locate an optimal solution.

The main challenge for the method remains to be, unsurprisingly, the curse of dimensionality: increasing the number of variables has systematic detrimental effect on GOCCS outcomes. Follow-up work on this issue is our priority, as practical LP/NLP models often feature large numbers of variables. Extending GOCCS with some form of dimensionality reduction seems to be most natural. Other future work includes improvement of GP search performance by employing semantic GP methods [31]–[33] or use of other types of evolutionary algorithms, e.g., as in [34].

REFERENCES

[1] T. P. Pawlak and K. Krawiec, "Synthesis of mathematical programming constraints with genetic programming," in *Proc. 20th Eur. Conf. Genet. Program. (EuroGP)*, vol. 10196, Apr. 2017, pp. 178–193.

[2] T. P. Pawlak and K. Krawiec, Automatic synthesis of constraints from examples using mixed integer linear programming," *Eur. J. Oper. Res.*, vol. 261, no. 3, pp. 1141–1157, 2017.

[3] T. Helmuth, L. Spector, and J. Matheson, "Solving uncompromising problems with lexicase selection," *IEEE Trans. Evol. Comput.*, vol. 19, no. 5, pp. 630–643, Oct. 2015.

[4] K. Deb, A. Pratap, S. Agarwal, and T. Meyarivan, "A fast and elitist multiobjective genetic algorithm: NSGA-II," *IEEE Trans. Evol. Comput.*, vol. 6, no. 2, pp. 182–197, Apr. 2002, doi: 10.1109/4235.996017.

[5] C. Bessiere, R. Coletta, F. Koriche, and B. O'Sullivan, *A SAT-Based Version Space Algorithm for Acquiring Constraint Satisfaction Problems*. Heidelberg, Germany: Springer, 2005, pp. 23–34, doi: 10.1007/11564096_8.

[6] C. Bessiere, R. Coletta, B. O'Sullivan, and M. Paulin, "Query-driven constraint acquisition," in *Proc. IJCAI*, Jan. 2007, pp. 50–55.

[7] C. Bessiere *et al.*, "Constraint acquisition via partial queries," in *Proc. IJCAI*, 2013, pp. 475–481.

[8] A. Aswal and G. N. S. Prasanna, "Estimating correlated constraint boundaries from timeseries data: The multi-dimensional German tank problem," in *Proc. EURO*, 2010, pp. 1–17. [Online]. Available: http://slideplayer.com/slide/7976536/

[9] N. Beldiceanu and H. Simonis, "A model seeker: Extracting global constraint models from positive examples," in *Principles and Practice of Constraint Programming* (LNCS 7514). Quebec City, QC, Canada: Springer, Oct. 2012, pp. 141–157.

[10] S. Kolb, "Learning constraints and optimization criteria," in *Proc. AAAI Workshops*, 2016, pp. 403–409.

[11] S. Teso, R. Sebastiani, and A. Passerini, "Structured learning modulo theories," *Artif. Intell.*, vol. 244, pp. 166–187, Mar. 2017.

[12] J. M. Hernández-Lobato *et al.*, "A general framework for constrained Bayesian optimization using information-based search," *J. Mach. Learn. Res.*, vol. 17, no. 1, pp. 5549–5601, Jan. 2016. [Online]. Available: http://jmlr.org/papers/v17/15-616.html

[13] D. M. J. Tax, "One-class classification: Concept-learning in the absence of counter-examples," Ph.D. dissertation, Faculty Elect. Eng., Math. Comput. Sci. Intell. Syst., Delft Univ. Technol., Delft, The Netherlands, 2001.

[14] C. Cortes and V. Vapnik, "Support-vector networks," *Mach. Learn.*, vol. 20, no. 3, pp. 273–297, 1995, doi: 10.1007/BF00994018.

[15] F. Denis, R. Gilleron, and F. Letouzey, "Learning from positive and unlabeled examples," *Theor. Comput. Sci.*, vol. 348, no. 1, pp. 70–83, 2005, doi: 10.1016/j.tcs.2005.09.007.

[16] J. R. Quinlan, *C4.5: Programs for Machine Learning*. San Francisco, CA, USA: Morgan Kaufmann, 1993.

[17] R. Curry and M. Heywood, "One-class genetic programming," in *Proc. 12th Eur. Conf. Genet. Program. (EuroGP)*, vol. 5481, Apr. 2009, pp. 1–12.

[18] V. L. Cao, M. Nicolau, and J. McDermott, "One-class classification for anomaly detection with kernel density estimation and genetic programming," in *Proc. 19th Eur. Conf. Genet. Program. (EuroGP)*, vol. 9594, Mar./Apr. 2016, pp. 3–18.

[19] Y. Bengio, "Learning deep architectures for AI," *Foundations and Trends® in Machine Learning*, vol. 2, no. 1, pp. 1–127, 2009, doi: 10.1561/2200000006.

[20] L. Manevitz and M. Yousef, "One-class document classification via neural networks," *Neurocomputing*, vol. 70, nos. 7–9, pp. 1466–1481, Mar. 2007, doi: 10.1016/j.neucom.2006.05.013.

[21] D. J. Montana, "Strongly typed genetic programming," *Evol. Comput.*, vol. 3, no. 2, pp. 199–230, 1995, doi: 10.1162/evco.1995.3.2.199.

[22] J. R. Koza, *Genetic Programming: On the Programming of Computers by Means of Natural Selection*. Cambridge, MA, USA: MIT Press, 1992. [Online]. Available: http://mitpress.mit.edu/books/genetic-programming

[23] D. M. J. Tax and R. P. W. Duin, "Uniform object generation for optimizing one-class classifiers," *J. Mach. Learn. Res.*, vol. 2, pp. 155–173, Mar. 2002. [Online]. Available: http://jmlr.org/papers/v2/tax01a.html

[24] C. Rüegg. *Math.NET Symbolics*. Accessed: Jan. 20, 2018. [Online]. Available: http://symbolics.mathdotnet.com/

[25] G. Kanji, *100 Statistical Tests*. London, U.K.: SAGE, 1999.

[26] T. Hothorn, K. Hornik, M. A. van de Wiel, and A. Zeileis. (2015). *Package 'Coin': Conditional Inference Procedures in a Permutation Test Framework*. [Online]. Available: http://cran.r-project.org/web/packages/coin/coin.pdf

[27] R. Bellman, *Dynamic Programming* (Dover Books on Computer Science). Mineola, NY, USA: Dover, 2013.

[28] P. Cortez, A. Cerdeira, F. Almeida, T. Matos, and J. Reis, "Modeling wine preferences by data mining from physicochemical properties," *Decis. Support Syst.*, vol. 47, no. 4, pp. 547–553, 2009.

[29] *Gurobi Optimizer Reference Manual*, Gurobi Optim. Inc., Houston, TX, USA, 2015. [Online]. Available: http://www.gurobi.com

[30] C. B. Barber, D. P. Dobkin, and H. Huhdanpaa, "The quickhull algorithm for convex hulls," *ACM Trans. Math. Softw.*, vol. 22, no. 4, pp. 469–483, Dec. 1996, doi: 10.1145/235815.235821.

[31] T. P. Pawlak, B. Wieloch, and K. Krawiec, "Semantic backpropagation for designing search operators in genetic programming," *IEEE Trans. Evol. Comput.*, vol. 19, no. 3, pp. 326–340, Jun. 2015, doi: 10.1109/TEVC.2014.2321259.

[32] T. P. Pawlak, B. Wieloch, and K. Krawiec, Review and comparative analysis of geometric semantic crossovers," *Genet. Program. Evolvable Mach.*, vol. 16, no. 3, pp. 351–386, Sep. 2015.

[33] T. P. Pawlak and K. Krawiec, Competent geometric semantic genetic programming for symbolic regression and boolean function synthesis," *Evol. Comput.*, Feb. 2017, pp. 1–36.

[34] T. P. Pawlak, "Synthesis of mathematical programming models with one-class evolutionary strategies," *Swarm Evol. Comput.*, May 2018.

Tomasz P. Pawlak received Ph.D. degree from the Poznań University of Technology, Poland, in 2015.

He is currently an Assistant Professor with the Poznań University of Technology. His current research interests include constraint synthesis, one-class classification, evolutionary computation, and genetic programming, semantics and program behavior.

Dr. Pawlak was a recipient of the START Scholarship from Foundation for Polish Science, which is one of the most notable scholarships for young researchers in Poland, and several other awards. More details at www.cs.put.poznan.pl/tpawlak.

Krzysztof Krawiec received the Ph.D. and Habilitation degrees from the Poznań University of Technology, Poland, in 2000 and 2004, respectively.

He is currently an Associate Professor with the Poznań University of Technology. He has authored over 100 publications, including *Evolutionary Synthesis of Pattern Recognition Systems* in 2005 and *Behavioral Program Synthesis With Genetic Programming* in 2016. His current research interests include semantics and program behavior in genetic programming, coevolutionary algorithms and test-based problems, and evolutionary computation for learning game strategies and for synthesis of pattern recognition systems.

Dr. Krawiec is an Associate Editor of *Genetic Programming and Evolvable Machines*. More details at www.cs.put.poznan.pl/kkrawiec.