Swarm and Evolutionary Computation 64 (2021) 100896

Contents lists available at ScienceDirect

Swarm and Evolutionary Computation

journal homepage: www.elsevier.com/locate/swevo

Survey Paper

Grammatical evolution for constraint synthesis for mixed-integer linear programming

Tomasz P. Pawlak [a,*], Michael O'Neill [b]

[a] Institute of Computing Science, Poznan University of Technology, Poznań, Poland
[b] School of Business, University College Dublin, Dublin, Ireland

ARTICLE INFO

Keywords:
Mathematical programming
Model acquisition
Constraint learning
High-level modeling language
Operations research

ABSTRACT

The Mixed-Integer Linear Programming models are a common representation of real-world objects. They support simulation within the expressed bounds using constraints and optimization of an objective function. Unfortunately, handcrafting a model that aligns well with reality is time-consuming and error-prone. In this work, we propose a Grammatical Evolution for Constraint Synthesis (GECS) algorithm that helps human experts by synthesizing constraints for Mixed-Integer Linear Programming models. Given relatively easy-to-provide data of available variables and parameters, and examples of feasible solutions, GECS produces a well-formed Mixed-Integer Linear Programming model in the ZIMPL modeling language. GECS outperforms several previous algorithms, copes well with tens of variables, and seems to be resistant to the curse of dimensionality.

1. Introduction

1.1. Background

The Mixed-Integer Linear Programming (MILP) models [1] are a common representation for a real-world object that consists of three parts: (1) variables of the object specified with domains (real or integer) and bounds on their values, (2) linear constraints representing the relationships between these variables, and (3) a linear objective function of these variables representing the outcome of this object. For instance, for a diet plan, the variables may represent the quantities of food items, the constraints might represent the lower bounds on nutrients delivered by the food items, and the objective function could represent the cost of the food. MILP models are quite popular in business and academia, e.g., the NEOS Solver Server [2] reports 36% of the submitted models in 2019 were MILP. A solver is a software tool that *solves* the model by assigning values to variables that minimize (maximize) the objective function subject to the constraints. For example, it finds the diet plan of the minimal cost that meets the nutritional constraints.

MILP models are typically handcrafted by a modeling expert in collaboration with domain experts. This is because sharing the competencies in modeling, and the object being modeled by a single expert is not common in practice. The modeling expert gains information on the object by interviewing the domain experts. As things like personal feelings, and incomplete knowledge of the domain experts may hide some details from the modeling expert, modeling often requires several iterations to bring satisfactory alignment of MILP models with reality. To further complicate matters, many real-world objects are not linear and the non-linear relationships need to be linearized or approximated to meet the requirement of the MILP model. These are advanced techniques and implementing them is error-prone. The errors in MILP models often remain undetected until the optimal solution to the model turns out inapplicable in practice, requiring another iteration of modeling. All these challenges increase the cost of modeling and optimization services.

ZIMPL [3] is a high-level modeling language for MILP models that facilitates modeling by compactly representing common constructs, e.g., sums and quantifiers. The ZIMPL interpreter automatically linearizes common non-linear functions, e.g., absolute value, min, max. ZIMPL transforms into an LP format [4], a low-level modeling language supported by all major solvers. Therefore, a MILP model specified in ZIMPL can be solved by virtually any solver.

ZIMPL, though helpful, does not diminish all challenges in modeling and the burden on the experts remains high. In this study, we propose to help the experts further. Rather than handcraft the MILP model, we propose an approach to automate the synthesis of MILP models in ZIMPL from underlying data about the problem. We assume that the dimension sets, the parameters, and the variables of the object are given. For instance, for the diet plan, one dimension is a set of food items and another is a set of nutrients, the parameters consist of volumes of nutrients in food items, and the variables represent quantities of food items in the diet plan. We also assume that a training set of examples of feasible solutions is available, e.g., the set of exemplary diet plans meeting all

* Corresponding author.
E-mail addresses: tpawlak@cs.put.poznan.pl (T.P. Pawlak), m.oneill@ucd.ie (M. O'Neill).

https://doi.org/10.1016/j.swevo.2021.100896
Received 15 August 2020; Received in revised form 2 April 2021; Accepted 21 April 2021
Available online 28 April 2021

nutrition constraints. A diet advisor may easily collect such data during her service, however, transforming this data into a MILP model requires proper technical training.

Building a MILP model can be decomposed into two largely-independent tasks, (1) the design of the objective function, and (2) the design of the constraints. The latter task of constraint design is more demanding because the number of constraints is usually large, while a typical model consists of only one objective function. Hence, in this work, we focus our attention towards constraint synthesis.

1.2. Contributions

The primary contributions of this study relate to the verification of the main research hypothesis: *the MILP constraints in ZIMPL can be synthesized from the underlying problem data using Grammatical Evolution (GE) [5]*.

More precisely, the contributions are:

- The formalization of the *Constraint Synthesis Problem* (CSP) in Section 2.3
- The proposition in Section 4 of the *Grammatical Evolution for Constraint Synthesis* (GECS) algorithm for CSP
- The empirical verification of the properties of GECS using fourteen real-world and four synthetic CSPs in Section 5.

GECS first generates a problem-specific context-free grammar from the input data, then runs GE to synthesize the constraints. GE is an evolutionary algorithm that uses integer vectors as genotypes and transforms them into code using the given grammar. GE has proved effective in many code synthesis problems [5–7].

GECS is not the first algorithm for CSP, however, to our knowledge it is the first one that synthesizes MILP constraints in a high-level modeling language. The use of the high-level language allows for the generation of constraints that automatically adapt to the data and facilitates the synthesis of large sets of related constraints. This offers a great advantage over contemporary algorithms, most of which fine-tune the weights and produce independent constraints stuck to the training examples. As empirical evidence shows, this also makes GECS resistant to the curse of dimensionality [8] that all other referenced algorithms suffer from. Section 3 discusses the variants of CSP and compares GECS to contemporary algorithms. Section 5.3 confirms empirically the superiority of GECS to two other algorithms in the terms of the test-set performance. Section 6 discusses the advantages and disadvantages of GECS in the context of other algorithms. Section 7 concludes this work and outlines possible extensions to GECS.

Appendix A shows the best models synthesized by GECS in this work. Appendix B lists the abbreviations and the symbols used in the text.

2. Constraint synthesis problem

2.1. Terminology

We define several distinct formal objects that share common names in the literature. To make things clear, we use the term *problem* to refer to the Constraint Synthesis Problem (CSP), the term *model* to refer to the MILP model that in fact consists of the input and the output of the CSP, and the term *solution* to refer to the solution of the MILP model. We also use the terms *model* and *set of constraints* interchangeably, as the latter is an essential part of the former and we do not synthesize other parts of the model.

2.2. Definitions

Let $m = (P, S, \mathbf{x}, g, C)$ be a model in ZIMPL [3,9], where P is a set of parameters p, S is a set of dimension sets s, $\mathbf{x}$ is a vector of variables x, $g(\mathbf{x}) : \mathbb{R}^n \to \mathbb{R}$ is an objective function, and C is a set of constraints $c(\mathbf{x}) : \mathbb{R}^n \to \{0, 1\}$. In ZIMPL, P and S are inputs to m. Parameter $p \in P$ is a number or a string literal or an array thereof indexed using $s \in S$. The dimension set s consists of numbers or string literals or tuples thereof; s may be indexed by another $s' \in S \setminus \{s\}$ and parameterized using $p \in P$. We use the adjective *dimension* because s typically corresponds to a dimension of a modeled object. Variable $x \in \mathbf{x}$ attains either real or integer or binary value or is an array thereof indexed using $s \in S$. The domain of x may be ranged by numeric constants and $p \in P$. The objective function $g(\mathbf{x})$ and the constraints $c(\mathbf{x}) \in C$ are piecewise linear functions w.r.t. $\mathbf{x}$. A specific value of $\mathbf{x}$ such that $\forall_{c \in C} c(\mathbf{x}) = 1$ is referred to as *feasible solution*, and the set $f(C) = \{\mathbf{x} : \forall_{c \in C} c(\mathbf{x}) = 1\}$ as *feasible region*. The *optimal solution* to m is $\mathbf{x} \in f(C)$ that minimizes $g(\mathbf{x})$. For $f(C) = \emptyset$, m is infeasible.

Note that g is optional in m: if not given, g can be simply substituted with a constant function $g(\mathbf{x}) = 1$ and all other definitions apply respectively, e.g., all feasible solutions are optimal w.r.t. g. As we do not use g, we skip it later on.

The below snippet shows the diet plan model in ZIMPL (namely the *zdiet* model from Section 5):

```
set Food := {"Oatmeal","Chicken","Eggs","Milk","Pie","Pork"};
set Nutr := {"Energy", "Protein", "Calcium"};
set Attr := Nutr + {"Servings", "Price"};
param need[Nutr]:=<"Energy">2000,<"Protein">55,<"Calcium">800;
param data[Food * Attr] :=
          |"Servings","Energy","Protein","Calcium","Price"|
|"Oatmeal"|         4,     110,        4,         2,      3|
|"Chicken"|         3,     205,       32,        12,     24|
|"Eggs"   |         2,     160,       13,        54,     13|
|"Milk"   |         8,     160,        8,       284,      9|
|"Pie"    |         2,     420,        4,        22,     20|
|"Pork"   |         2,     260,       14,        80,     19|;
#                        (kcal)       (g)       (mg) (cents)
var x[<f> in Food] integer >= 0 <= data[f, "Servings"];
minimize cost: sum <f> in Food: data[f, "Price"] * x[f];
subto constraint1: forall <i> in Nutr:
  sum <j> in Food: data[j,i] * x[j] >= need[i];
```

In the above model snippet in ZIMPL, $P = \{\texttt{need}, \texttt{data}\}$ $S = \{\texttt{Food}, \texttt{Nutr}, \texttt{Attr}\}$; `Food` stands for the dimension set of available food items, `Nutr` for nutrients, `Attr` for food attributes, `need` stores the minimum daily intake of nutrients, and `data` stores the values of the attributes of food items. `x[Food]` is an array of variables representing the numbers of food items `f` $\in$ `Food` to eat. The `constraint1` verifies for each nutrient whether the food items in the diet plan provide an intake greater than is required.

To solve a ZIMPL model m, the ZIMPL interpreter transforms it into an equivalent model in LP format m' (cf. [4]), and then m' is fed into a solver (in this work [4]). The transformation to m' may introduce auxiliary variables $\mathbf{x}'$ and set C' of auxiliary linear constraints that implement non-linear features of ZIMPL, e.g., absolute value, min and max functions, and conditional expressions. $\mathbf{x}'$ and C' do not change the meaning of the solution to m, i.e., $\mathbf{x} \in f(C) \iff \exists_{\mathbf{x}'} : [\mathbf{x}, \mathbf{x}'] \in f(C \cup C')$. For that reason, we assume that the transformation to the LP format is purely technical and continue discourse using ZIMPL. For more details on ZIMPL, the reader is referred to [3,9].

2.3. Problem

Let X be a set of examples of feasible solutions $\mathbf{x}$. Given the inputs P, S, $\mathbf{x}$ as a ZIMPL snippet, and the input X as a matrix, the *Constraint Synthesis Problem* (CSP) is to find a set of constraints C that maximizes:

$$F_1(C) = 2rq/(r+q) \tag{1}$$

$$r = |f(C) \cap X|/|X| \tag{2}$$

$$q = |f(C) \cap X|/|f(C)| \tag{3}$$

where $F_1(C)$ is F_1-score, r is recall, and q is precision [10, Ch.10] of the partitioning of solution space induced by C.

r does not require calculating $f(C)$ and reduces to the fraction of $\mathbf{x} \in X$ that satisfies the constraints in C. This reduction does not apply to q, where the cardinality of $f(C)$ is important. Note that $f(C)$ may be infinite even if it is bounded, e.g., when real variables exist. Using an infinite $f(C)$ and a finite X at the same time results in $q = 0$ and a negatively biased assessment. For that reason, we substitute $f(C)$ in q with its sample $\hat{f}(C) \subseteq f(C)$. Since $\hat{f}(C)$ may be arbitrarily small part of $f(C)$ and it may happen that $\hat{f}(C) \cap X = \emptyset$ even if $f(C) \cap X \neq \emptyset$, we also use an intersection operator with tolerance $\cap_t$. This results in:

$$\hat{q} = |\hat{f}(C) \cap_t X|/|\hat{f}(C)| \tag{4}$$

$$\hat{f}(C) \cap_t X = \left\{ \mathbf{x} \in \hat{f}(C) : \min_{\mathbf{x}' \in X} d(\mathbf{x}, \mathbf{x}') \leq t \right\} \tag{5}$$

$$d(\mathbf{x}, \mathbf{x}') = \sum_i \frac{|x_i - x'_i|}{|x_i| + |x'_i|} \tag{6}$$

where d is Canberra distance [11], the divisions in d and $\hat{q}$ return 0 if the divisor is 0, t is a threshold on the distance between $\mathbf{x} \in \hat{f}(C)$ and the closest $\mathbf{x}' \in X$. Canberra distance is a weighted L_1 metric that normalizes the magnitudes of values of individual variables to avoid domination of a single variable with large magnitude in the value of the distance. Thanks to normalization, we found in a preliminary experiment a single formula $t = \frac{n}{\ln |X|}$ that for $|\hat{f}(C)| = |X|$ provides $F_1(C^*) \approx 1$ for all ground truth C^* in this study and smaller t decreases $F_1(C^*)$; n is the dimensionality of $\mathbf{x}$.

2.4. Sampling the feasible region

To obtain a uniformly distributed sample $\hat{f}(C)$ of $f(C)$ for an arbitrary C and for large n we use the Hit-and-Run (HaR) algorithm [12]:

1. Relax the domains of integer variables in $\mathbf{x}$ to real
2. Calculate the initial example $\mathbf{x}_0$ as the optimal solution to C supplemented with a random objective function $g(\mathbf{x}) = \mathbf{w} \cdot \mathbf{x}$, where $\mathbf{w} \sim \mathcal{U}([-1, 1]^n)$; $\mathcal{U}(Z)$ stands for the uniform distribution over Z
3. Draw uniformly a random vector $\Delta\mathbf{x} \sim \mathcal{U}([-1, 1]^n)$
4. $\mathbf{x}_{i+1} \leftarrow \mathbf{x}_i + \lambda\Delta\mathbf{x}$, where $\lambda \sim \mathcal{U}([\lambda_{\min}, \lambda_{\max}])$ and $\lambda_{\min}$ and $\lambda_{\max}$ are extreme values for which $\mathbf{x}_{i+1} \in f(C)$
5. Round $\mathbf{x}_{i+1}$ on integer variables to the closest feasible solution w.r.t. L_1 metric and store $\mathbf{x}_{i+1}$ in $\hat{f}(C)$
6. Increment i and go back to step 3 unless $\hat{f}(C)$ reached the requested cardinality.

Note that the above HaR algorithm differs from the definition in [12] in that it begins in step 2 from the optimal solution w.r.t. a random objective function rather than a randomly drawn solution. This is because for large C and n drawing a feasible solution is virtually impossible and it must be constructed somehow. Steps 1 and 5 are added for handling integer variables, as HaR in [12] is defined for the real domain.

2.5. Example

We show a practical instance of a CSP by synthesizing the constraints for the diet plan MILP model. The inputs P, S, $\mathbf{x}$ are given in the form of lines 1–14 of the above ZIMPL snippet and the input X is given as the matrix:

x[Oatmeal]	x[Chicken]	x[Eggs]	x[Milk]	x[Pie]	x[Pork]
0	0	0	4	2	2
0	0	2	2	2	2
0	3	0	2	2	2
0	0	2	8	0	2
0	0	0	8	2	2
⋮	⋮	⋮	⋮	⋮	⋮

For the sake of argument, assume that $|X| = 400$, but for brevity, we show only the first five examples. Lines 16–17 of the snippet in Section 2.2 show the output — the best C found in this study that scores $F_1(C) = 1$. C is equivalent to the ground truth C^* in the zdiet model and differs only in the names of the constraint and the temporary variables.

2.6. Properties

CSP is a kind of one-class classification problem [13], where the aim is to identify the inherent properties of the positive class (the feasible region) using examples of this class rather than separating it from other classes. The main difficulty in CSP lies in finding the boundary of the feasible region expressed using constraints. Given no information about the shape of the ground truth feasible region, it can be only estimated based on the input to the CSP. The threshold t on d in Eq. (5) plays the role of the estimator by locating the optimal boundary of the feasible region in the minimal distance t from all examples in X.

Assuming $X = f(C^*)$, i.e., the training set consists of all feasible solutions to the optimal model, CSP is a problem of finding the α-shape $\alpha(X)$ of X [14] for some $\alpha \in \mathbb{R}$. $\alpha(X)$ is a generalization of the convex hull $co(X)$ that for $\alpha = 0$ reduces to $co(X)$, for $\alpha > 0$ it holds that $X \subseteq co(X) \subseteq \alpha(X)$, and for $\alpha < 0$ it holds that $X \subseteq \alpha(X) \subseteq co(X)$. The facets of $\alpha(X)$ are piecewise linear constraints and $X = \alpha(X)$ for some α. Hence, α exists such that $r(\alpha(X)) = q(\alpha(X)) = F_1(\alpha(X)) = 1$.[1] Calculating $\alpha(X)$ relies on Delaunay triangulation and this has $O(|X|^{\lfloor n/2 \rfloor})$ facets in dimension n [15], hence finding $\alpha(X)$ for arbitrary n is NP-hard and so CSP is.

A CSP is ill-posed, since two different models C_1 and C_2 may be optimal w.r.t. the same instance of the CSP. Some constraints in C_1, C_2, or both may be redundant and do not contribute to the shape of the feasible region, i.e., it may hold that $f(C_1) = f(C_2)$ and so $F_1(C_1) = F_1(C_2)$ even if $C_1 \neq C_2$. Dropping redundancy directly is difficult because a single constraint in ZIMPL may correspond to many hyperplanes in the solution space, some of which may be redundant and others may not. Rather than redundancy removal at all costs, we focus on the simplicity of representation and incorporate this preference into the fitness function by mixing two criteria: F_1 and the number of characters L in the ZIMPL code:

$$\hat{F}_1(C) = F_1(C) - 10^{-6} L \tag{7}$$

The constant 10^{-6} is small enough to ensure the lexicographic order of these criteria in all the CSPs in this study, i.e., for any C_1 and C_2 if $F_1(C_1) < F_1(C_2)$ then $\hat{F}_1(C_1) < \hat{F}_1(C_2)$ holds, and if $F_1(C_1) = F_1(C_2)$ the relative ranks of C_1 and C_2 depend only on L. This may affect the search path in the space of models visited by a synthesis algorithm, but should not prevent it from finding any C^* optimal w.r.t. F_1 and then reducing its size.

3. Related work

In this section we first discuss the alternatives to ZIMPL, then review different formulations of a CSP, and finally survey the works on the synthesis of Mathematical Programming (MP) models.

3.1. Modeling languages

Arguably, the most widespread and recognized high-level language for MP models is AMPL [16], e.g., the NEOS Solver Server [2] reports that 58% of jobs submitted in 2019 used AMPL. Another well-recognized language is GAMS [17], with an 18% market share in 2019 according to [2]. Both, AMPL and GAMS support Linear Programming and

[1] Abuse of notation for brevity; $\alpha(X)$ is the feasible region of the set of constraints $C = f^{-1}(\alpha(X))$.

Non-Linear Programming (LP/NLP) models, transcode to other formats, e.g., LP format [4], and integrate well with many solvers. However, we do not use AMPL and GAMS in this study because the AMPL and GAMS tools, in contrast to ZIMPL, are proprietary software and we require access to the internals of the language interpreter.

3.2. Similar problems

A one-class CSP, as posed in Section 2.3, was previously formulated in several other works e.g., [18–21]. However, the qualitative difference between this work and the previous art [18–21] lies in the generality of the resulting constraints. In this work, the constraint is a high-level expression made of symbols of dimension sets from S, parameters from P, and variables from $\mathbf{x}$, and so valid for any S, P, and $\mathbf{x}$ with the same symbols. In contrast, the constraint in [18–21] is a low-level linear expression of a fixed vector of variables. Consider for instance the best-found constraint for the zdiet CSP from the previous section. It is general enough to model any set of lower bounds for nutrients in the diet plan involving any set of food items and any set of nutrients. On the contrary, a single constraint in [18–21] would represent a single lower bound for a single nutrient in the diet plan with a fixed number and characteristics of food items.

A CSP is sometimes formulated in a two-class way, e.g., in [22–25], where examples of feasible and infeasible solutions are to be separated using constraints. It might be considered a simpler problem than a one-class CSP, as the location of the boundary between classes lies between the examples. However, as shown in [26] the problem of *determining* whether a fixed number $k \geq 2$ of linear constraints separating two sets of examples exist is NP-complete. It is NP-complete even if k varies in the range $[2, n^2]$, where n is the number of variables [27].

Syntax-guided program synthesis (SyGuS) [28] tackles the problem of synthesis of a program in an arbitrary language, where background theory, a correctness predicate, and grammar are given in a predefined format as input. SyGuS is similar to a CSP in that both restrict the syntax of the output. In contrast to a CSP, the training set is optional in SyGuS, however, it may involve a verification oracle that produces counterexamples for invalid models. The algorithms for SyGuS from the work [28] are restricted to integer variables and cannot produce some of the MILP models in this study.

CSP is similar to learning polytopes [29] in the sense that the facets and the interior of the polytope correspond to the MILP constraints and the feasible region in CSP, respectively. However, the constraints in CSP are high-level, i.e., every single constraint may correspond to any number of facets of the polytope and so by synthesizing a single constraint one can learn the entire polytope at once.

Rather than synthesizing an explicit model from examples, Galassi et al. [30] construct a new feasible solution directly from the feasible examples. They teach the deep neural network to produce solutions consistent with the examples provided. This type of problem formulation is useful when one seeks other solutions than already known, but cannot provide explanations nor insight on the object the solutions refer to.

De Raedt et al. [31] survey other variants of the constraint synthesis problem, its applications, and algorithms.

3.3. Synthesis of MP models

We classify the works on the synthesis of MP models based on the representation, and begin from these on LP/NLP [1] as the closest to the topic of this study, then advance to Constraint Programming (CP) [32] and Satisfiability Modulo Theories (SMT) [33].

Linear/Non-Linear Programming

GOCCS [34] is a strongly-typed genetic programming (GP) [35] system that from a one-class training set produces a set of constraints configurable within LP/NLP. GOCCS is susceptible to the curse of dimensionality [8] and its performance decreases rapidly for more than five variables.

ESOCCS [19,36] produces user-configurable LP/NLP models from a one-class training set using evolutionary strategy [37]. ESOCCS outperforms GOCCS when compared using several synthetic benchmarks. It was successfully applied to fully-automated modeling and optimization of a rice farm.

CMA-ESOCCS [21] is a similar to ESOCCS algorithm based on covariance matrix adaptation evolutionary strategy [38]. It has smaller computation cost than ESOCCS and offers similar learning performance.

CSC4.5 [18] creates MILP models from one-class data using a hybrid of expectation-maximization [39] and C4.5 decision tree [40]. CSC4.5 produces oversize models with constraints involving a single variable each, hence unable to represent relationships between the variables.

OCCALS [20] synthesizes MILP models using x-means [41] and local search [42] from one-class data. It outperforms GOCCS, ESOCCS, and CSC4.5, however, it still suffers from the curse of dimensionality for seven or more variables.

EOCCA [43] is a fast constructive approach backed by Principal Components Analysis [44] and x-means [41] that produces Mixed-Integer Quadratic Constraints from one-class data. EOCCA consistently beats ESOCCS on F_1-score and computation time, however, it still suffers from the curse of dimensionality for over 7–9 variables.

Lombardi et al. [24] propose to learn ordinary machine learning models: C4.5 decision tree [40] and a multilayer neural network [45] and then transform them into Mixed-Integer NLP, CP, and SMT models. It uses two-class training sets of exemplary feasible and infeasible schedulings to learn the scheduling model. The downsides of this work are a lack of tuning of the learning algorithms to a CSP and an experimental evaluation limited to a single problem domain.

Pawlak and Krawiec [22] encode the problem of two-class synthesis of LP/NLP constraints using the MILP problem and solves it optimally using an off-the-shelf solver. However, this method overfits and is computationally costly, often requiring terminating the solver before the optimum is reached.

GenetiCS [23] is another strongly-typed GP system. In contrast to GOCCS, GenetiCS requires two classes of examples. The evaluation in [34] shows that GenetiCS requires up to 60% more training information than GOCCS to achieve similar performance.

IncaLP [25] is another algorithm that hybridizes the MILP encoding from the work [22] with incremental learning of INCAL [46] (see below). It achieves similar test-set-based performance to [22] but works one-two orders of magnitude faster. Despite the speed improvement, it still suffers from the same issues as [22].

The above-mentioned algorithms produce low-level constraints involving weights specific to the training examples. In contrast, GECS produces high-level constraints configurable using sets and parameters, and thus adaptable to different objects of the same class by simply providing new values for the parameters. See Section 6 for a more detailed discussion.

Constraint Programming

Model Seeker [47] finds CP-like constraints expressed in Prolog language using a one-class training set and a handcrafted library of templates of constraints. This library is a piece of domain knowledge to be supplied with the problem, and Model Seeker is limited to the constraints available in this library.

Conacq [48] uses version space learning [49] to learn a single CP constraint using a two-class training set. An extension [50] adds interactive queries of an expert for the classification of artificially-created examples. Another extension [51] adds support for arguments provided by the expert to explain her decisions. Conacq is limited to finite-domain

variables and does not support weights nor nonlinear transformations of the variables. The resulting constraint is NP-hard to solve.

QuAcq [52] synthesizes CP models using a two-class set and interactive queries to an expert. QuAcq is asymptotically optimal in the number of queries for constraints involving only $=$ and $\neq$ comparisons.

Satisfiability Modulo Theories

Learning Modulo Theories [53] is a framework for the synthesis of SMT models from one-class data and linear real arithmetic background theory. The assessment using two synthetic problems shows a considerable amount of background knowledge required to synthesize efficiently.

INCAL [46] is an exact algorithm that synthesizes SMT models involving linear real arithmetic from a two-class training set. INCAL encodes the synthesis problem as an SMT and solves it using an off-the-shelf solver.

The work [54] employs inductive logic programming to learn a set of first-order clauses for weighted MAX-SAT theories using one-class examples of solutions and user preferences. The proposed system learns clauses and their weights compliant with the underlying models. The evaluation using relatively easy problem instances shows that the accuracy of this method highly depends on the amount of the available examples and noise.

4. Constraint synthesis algorithm

Grammatical Evolution for Constraint Synthesis (GECS), the main contribution of this study, is the algorithm solving CSP posed in Section 2.3. The input to GECS is the ZIMPL snippet consisting of the definitions of the sets of parameters P, dimension sets S, and variables $\mathbf{x}$, and the matrix of examples, in the reference implementation given in the CSV format. GECS assumes that the ZIMPL snippet is complete and consists of all symbols available for use in the model. GECS yields a ready-to-use well-formed MILP model in ZIMPL.

GECS is based on the PonyGE2 tool [55] and extends it with the ZIMPL interpreter, problem-specific grammars, and a custom fitness function. Fig. 1 shows the flowchart of GECS. It operates in three steps:

1. Extract symbols from the given ZIMPL snippet
2. Generate a problem-specific grammar using these symbols
3. Run PonyGE2 employed with that grammar and the custom fitness function.

Section 4.1 details how we extend the ZIMPL interpreter to extract input information from the ZIMPL snippet in step 1. Section 4.2 discusses grammar generation in step 2. Step 3 runs the PonyGE2 tool, briefly discussed in Section 4.3. The implementation of GECS is open-source.[2]

4.1. Symbol extraction

We extend the ZIMPL interpreter [3,9] to extract descriptors of the symbols from the ZIMPL code. A descriptor is a tuple of (*name*, *type*, *indexes*, *values*, *domain*), where *name* is the symbol name, *type* is either the parameter, set, or variable, *indexes* is the list of types of symbol indexes (`S` for string, `N` for number), *values* applies to sets only and is the list of the types of set elements (`S`, `N`, or tuple thereof), and *domain* applies to variables only and consists of the variable domain (`R` for real, `Z` for integer). Table 1 shows the descriptors extracted from the ZIMPL snippet for the zdiet CSP from Section 2.2.

Technically, the extended ZIMPL interpreter transcodes the ZIMPL code into a Python class for processing by GECS.

[2] https://github.com/tomash87/GECS

Table 1
Descriptors of symbols extracted from ZIMPL snippet for zdiet CSP; refers to empty list, n/a to "not applicable".

Name	Type	Indexes	Values	Domain
`Food`	Set		S	n/a
`Nutr`	Set		S	n/a
`Attr`	Set		S	n/a
`need`	Parameter	S	n/a	n/a
`data`	Parameter	SS	n/a	n/a
`x`	Variable	S	n/a	Z

4.2. Problem-specific grammar generation

GECS generates a problem-specific context-free grammar in Backus-Naur form (BNF) based on the descriptors extracted from the ZIMPL code, such that the models derivable from this grammar correctly use the types of the symbols in indexes and constraints.

4.2.1. Grammar template

GECS generates the grammar from the template designed such that all benchmark models in Section 5 are derivable. The main parts of the template are briefly discussed below and the full template is available as supplementary online material[3] due to its size.

The axiom of the grammar is `<subto>` non-terminal and the corresponding `<subto>` rule has two productions:

```
1 <subto> ::='subto name:' <constr1>';\n'
2         |  'subto name:' <constr1>';\n'<subto>
```

Both productions produce a single constraint, however, the second one recursively calls `<subto>` to produce another constraint. This way the total number $|C|$ of produced constraints is unbounded, but $|C|$ is negatively biased by the exponentially decreasing probability of producing exactly $|C|$ constraints: $(1/2)^{|C|}$. The successive occurrences of the term `name` in the resulting model are replaced by sequential names `constraint1`,..., `constraint`$|C|$ after model generation to drop duplicates.

4.2.2. Hierarchy of constraints

The template divides each constraint into a hierarchy of four levels of constraint expressions and allows for different structures at each level, e.g., `forall` quantifier, `sum` operator, `vif` conditional expression, linear expression, etc. The `forall` quantifier, and the `sum` operator by definition iterate over a set expression and introduce temporary variables or tuple thereof. The temporary variables store elements of that set expression and index the arrays of variables, parameters, and other sets in the next level constraint expression. A newly introduced temporary variable is assigned with the first unused name from the list `i`, `j`,..., and duplicate symbols are disallowed by design. The number and types of temporary variables are extracted from the descriptor of the set symbol in that expression. Every use of the `forall` quantifier is followed with a non-terminal of the next constraint level and the successive levels may use temporary variables introduced in the previous levels. The rule carries the information on the level l and the types of temporary variables t introduced in the previous levels of the expression in the name of the

[3] https://github.com/tomash87/GECS/blob/master/grammars/ZIMPL-dedicated.bnf

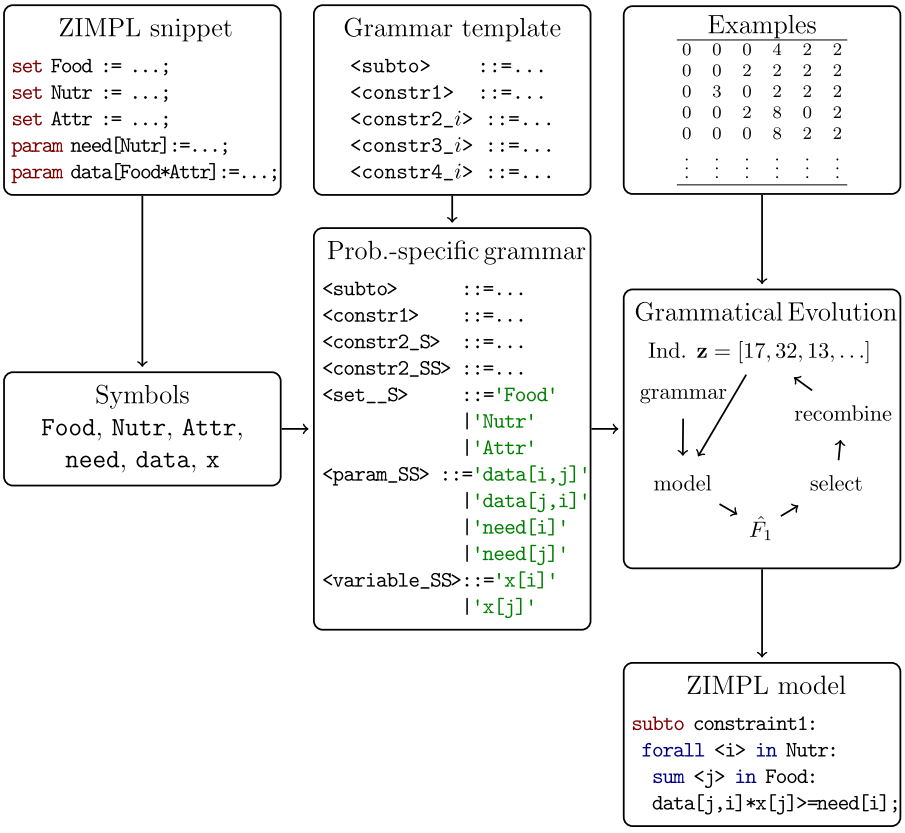

Fig. 1. The flowchart of GECS. The ZIMPL snippet and the matrix of examples are the input, and the ZIMPL model is the output.

left-hand side non-terminal: <constr*l_t*>. The templates for constraint expressions at levels 1–4 are shown below:

```
<constr1>  ::='forall <v> in '<sexpr__t>':'<constr2_t>
  |<sum_simplZ_>' >= 1'
<constr2_i>::='forall <v> in '<sexpr_i_t>':'<constr3_it>
  |<sum_i> <cmp> <cexpr0_i>
  |<sum_i><arithmetic_op><sum_or_var_i><cmp><cexpr0_i>
  |<lexpr_i> <cmp> <cexpr0_i>
  |<vabs_i>' >= 1'
  |<vif_i>
<constr3_i>::='forall <v> in '<set_i_t>':'<constr4_it>
  |<lexpr_i> <cmp> <cexpr0_i>
  |<sum_simpl_i> <cmp> <cexpr0_i>
  |<sum_simpl_i> '-' <sum_simpl_i> <cmp> <param_i>
<constr4_i>::=<vif_i>
```

where v is a comma-separated list of temporary variables introduced at the current level, t is a string of types of these variables, and i is a string of types of temporary variables inherited from the previous levels. A production at level l is created from the template for all combinations of types t of temporary variables up to length $2(l-1)$ if all non-terminals called by this production exist. A certain production may not exist if no symbols referenced by this production exist in the input ZIMPL snippet.

4.2.3. *Constraint structure*

Below, we outline the meaning of the non-terminals involved in the productions of the constraint expressions at all levels.

The <sexpr_*i_t*> non-terminal produces a set expression that consists of a set symbol from S and an optional `with` predicate for selection of its subset. The rule is produced for each $s \in S$ that satisfies two conditions: if s is an array then all types of its indexes are in i, and the list of types of elements of s equals t. For the zdiet CSP, the grammar contains non-terminal <sexpr_S_S>, where the first `S` refers to string temporary variable `i` introduced in the previous level constraint expression and the suffix `S` refers to string temporary variable `j` introduced in the current level by the `forall` or `sum` statement to store string elements of a set. Hence, <sexpr_S_S> evaluates to an array of sets of strings indexed with a string or simply a set of strings since the use of `i` is optional. For the zdiet CSP, the available options are `Food`, `Nutr`, `Attr`.

The template of <sum_*i*> rule corresponds to the (weighted) sum of variables over a set expression, where temporary variables of types i are inherited from the previous level, and <sum_*i*> may introduce new temporary variables to iterate over the set. In the grammar for the zdiet CSP, an exemplary non-terminal <sum_S> may evaluate to `sum` <j> in Food: data[j,i]*x[j], i.e., the sum of the food items weighted by the attribute stored in `i`.

The <sum_simpl Z_*i*> non-terminal is a variant of <sum_*i*> that reduces to a simple sum of variables over a set. The term `Z` is optional and restricts the sum to integer variables only.

The <cexpr*z*_*i*> rule produces a constant expression, where z is either 0 or empty string and signals whether producing 0 is allowed or not by this rule; i is the list of types of temporary variables inherited from the previous level constraint expression. The <cexpr*z*_*i*> non-terminal may evaluate to either a positive integer, a parameter $p \in P$ optionally indexed with available temporary variables, a function of a set $s \in S$ that evaluates to a number, a numeric temporary variable, or an expression thereof. In the zdiet CSP the options for an exemplary <cexpr_S> non-terminal are: `1`, `2`, `3`, `need[i]`, `card`(s), where $s \in S$.

The <sum_or_var_*i*> non-terminal represents an alternative of <sum_*i*> and <variable_*i*>, and the latter evaluates to variable name optionally indexed using temporary variables of types in i, e.g., `x[i]` for <variable_S> in the zdiet CSP.

The <lexpr_i> non-terminal is a weighted linear expression of variables, optionally indexed with temporary variables of types in i. For the zdiet CSP, e.g., <lexpr_SS> may evaluate to `data[j,i]*x[j] - need[i]*x[j]`.

The <vabs_i> non-terminal produces the absolute value of a linear expression with optional use of temporary variables of types in i.

The <vif_i> non-terminal evaluates to an indicator expression `vif <variableZ_i>==1 then <constr_i> end`, where <variableZ_i> is an integer variable and <constr_i> is another constraint that holds only if that variable equals 1.

The <param_i> non-terminal evaluates to a parameter $p \in P$ optionally indexed using temporary variables of types in i.

Last but not least, <cmp> and <arithmetic_op> evaluate to mathematical operators:

```
1 <cmp>            ::= '<=' | '==' | '>='
2 <arithmetic_op> ::=  '+' | '-'
```

4.2.4. Grammar folding

To simplify the generated grammar, each call to a non-terminal with only one production in the corresponding rule is substituted with this production, and the unused rules are removed from the grammar. Generally speaking, grammar folding is beneficial to the performance of Grammatical Evolution, as shown in the work [56].

4.3. Grammatical evolution

GECS searches for the ZIMPL model for the given CSP using Grammatical Evolution (GE) [5] equipped with the grammar from the previous section. For the readers unfamiliar with GE, we outline some essential basics below. For further details, we refer the reader to the documentation of the PonyGE2 framework [55] that we extend with extra classes to handle ZIMPL, grammar generation, and calculation of fitness from Eq. (7).

GE is a population-based stochastic optimization algorithm with general rules of conduct typical to evolutionary computation. First, the population of ZIMPL models is randomly initialized, then a loop of selection and recombination runs until a termination criterion is satisfied. This typically occurs when the maximum number of iterations is reached.

In GE the individual is a vector $\mathbf{z} = [z_1, z_2, \ldots] \in \mathbb{N}^*$ of non-negative integers called codons and the given BNF grammar acts as a mapping function onto the ZIMPL code. Derive the ZIMPL code using these steps:

1. Assign the code with the axiom of the grammar (here: <subto>)
2. Terminate unless the code contains a non-terminal and $\mathbf{z} \neq [\,]$
3. For the first non-terminal <N> seek for the rule r with <N> on the left-hand side
4. Substitute <N> with a production chosen out of the productions in r by indexing from 0 to $u-1$ all productions in r and selecting the one with the number $z_1 \bmod u$
5. Remove z_1 from $\mathbf{z}$
6. Go back to step 2.

The termination condition in step 2 may stop the loop before substituting all non-terminals, resulting in an invalid ZIMPL code. This is a protection against very large (infinite) derivation sequences resulting in oversize (infinite) ZIMPL codes. In this case, the ZIMPL code cannot be evaluated and is assigned with the worst possible fitness value.

A derivation-tree based initialization called Position Independent Grow (PI Grow) is adopted [57], which is reverse mapped back to generate the corresponding genotype vector $\mathbf{z}$. The vector $\mathbf{z}$ is manipulated using a range of recombination operators. PonyGE2 supports e.g., one-point and two-point crossover and mutation borrowed from genetic algorithms [58] and subtree crossover and mutation from genetic programming [59] applied to the derivation trees. In Section 5.1, we look for the setting of the operators that maximize performance in solving a CSP.

GE works with any parent selection operator, and in this work, we use tournament selection [58].

5. Experiment

We seek the answers to four experimental questions:

- What is the best parameter setting for GECS?
- How well does GECS scale with the dimensions of CSPs?
- How well does GECS compare to its competitors?
- How well do the synthesized models work in optimization?

We use eighteen MILP models in ZIMPL as ground truth in eighteen benchmark CSPs. Table 2 shows the statistics of these models: types, numbers, and dimensionality of the involved symbols. The prefix in the name of the model denotes its source: 'a' for the artificial benchmarks from the previous work [20], 'g' and 'z' for the examples from documentation of Gurobi solver [60] and ZIMPL [9], respectively. The 'a' models implement unions of intersecting multidimensional cubes and simplexes, respectively. The 'g' and 'z' models correspond to basic versions of real-world problems: gdiet and zdiet implement the product mix problem [1] occurring in e.g., production planning; gfacility and zfacility implement the facility location problem [61] common in e.g., company expansion planning; gnetflow is the network flow problem [61] known from e.g., production line design; gworkforce implements the assignment problem [61] occurring in e.g., timetabling; zsteiner is the Steiner tree problem [62] tacked in e.g., design of circuits; ztsp is the traveling salesman problem [63] from e.g., logistics; gsudoku [64] and zqueens* [64] implement classic games. Note that gdiet and zdiet implement the same problem differently and thus constitute different instances of a CSP. The same applies to gfacility and zfacility, and zqueens*.

The training set X, the validation set V, and the test set T are sampled independently without replacement from the feasible region $f(C^*)$ of the ground truth model using the HaR algorithm; $|V| = |T| = \min\{2000, |f(C^*)|\}$ and $|X|$ is set on per experiment basis.

The input to each benchmark CSP is composed of X and the ground truth model in ZIMPL stripped of the constraints.

Table 3 -A shows the parameters of GECS set to different values than the defaults in PonyGE2 [55] and fixed in all experiments. Table 3-B shows the parameters subject to tuning in Section 5.1.

The reported statistics are calculated using V and T and averaged over 25 runs with different random seeds. X, V, and T differ for each seed.

5.1. What is the best parameter setting for GECS?

GECS performance is found to be largely-independent of parameter values. The combination of the *variable one-point* crossover, the *int flip per ind* mutation, the population size of *500*, and *60* generations turns out the best setting. However, the Kruskal-Wallis test for the difference of this setting vs six others yields $p > 0.999$, and we conclude insignificance of this difference.

We found these values by seeking the best parameter setting from Table 3-B on $\hat{F}_1$ calculated using V. The parameter POPULATION_SIZE/GENERATIONS in fact combines two parameters set such that the computational budget POPULATION_SIZE×GENERATIONS=30000 is constant. To avoid the combinatorial explosion of parameter settings, we tune them one-by-one. First, we look for the best crossover operator given the remaining parameters set to their defaults in Table 3-B. Then, we tune the mutation operator given the best-found crossover and the defaults for the rest. Finally, we tune simultaneously the population size and the number of generations given the best-found search operators. Table 4 reports the results. In this experiment $|X| = \min\{400, |f(C^*)|\}$ for all benchmarks.

Table 2
Statistics of benchmarks: numbers and dimensionality of symbols (0D means plain symbol, 1D, 2D, 3D mean 1D, 2D, 3D arrays, resp.), $|\hat{f}(C^*)|$ is the total number of feasible integer solutions, n is dimensionality.

Model	Sets	Parameters	Variables	$\lvert\hat{f}(C^*)\rvert$	n
acube_3^2	2×0D	1×0D	1×1D real, 1×1D binary	372	5
acube_5^2	2×0D	1×0D	1×1D real, 1×1D binary	≥ 5000	7
asimplex_3^2	2×0D	3×0D	1×1D real, 1×1D binary	11	5
asimplex_5^2	2×0D	3×0D	1×1D real, 1×1D binary	2	7
gdiet	2×0D	3×1D, 1×2D	1×1D real	3	9
gfacility	2×0D	3×1D, 1×2D	1×1D binary, 1×2D real	≥ 5000	25
gnetflow	2×0D	2×2D, 1×3D	1×3D real	≥ 5000	50
gsudoku	1×0D, 1×1D	1×0D, 1×2D	1×3D binary	≥ 5000	729
gworkforce	2×0D	2×1D, 1×2D	1×2D real	≥ 5000	98
zdiet	3×0D	1×1D, 1×2D	1×1D integer	2249	6
zfacility	3×0D	3×1D, 1×2D	2×1D binary	≥ 5000	40
zqueens1	2×0D	1×0D	1×1D integer	92	8
zqueens2	2×0D, 1×2D	1×0D	1×2D binary	≥ 5000	64
zqueens3	2×0D, 1×2D	1×0D	1×2D binary	≥ 5000	64
zqueens4	2×0D, 1×2D	1×0D	1×2D binary	≥ 5000	64
zqueens5	2×0D	1×0D	1×2D binary	≥ 5000	64
zsteiner	4×0D, 1×1D	1×1D	1×1D binary	53	7
ztsp	3×0D, 1×1D	2×1D	1×1D binary	≥ 5000	45

Table 3
Parameters of GECS: (A) fixed and different than defaults in PonyGE2; (B) subject to tuning; defaults underlined.

(A) Parameter	Value
CACHE	True
CROSSOVER_PROBABILITY	0.9
MAX_GENOME_LENGTH	200
MAX_INIT_TREE_DEPTH	8
MAX_TREE_DEPTH	13
TOURNAMENT_SIZE	5
(B) Parameter	**Tuning set**
CROSSOVER	subtree (S), fixed_twopoint (F2), variable_onepoint (V1)
MUTATION	subtree (S), int_flip_per_ind (I), int_flip_per_codon (C)
POPULATION_SIZE/GENERATIONS	250/120, 500/60, 750/40

5.2. How well does GECS scale with the dimensions of CSPs?

GECS scales linearly on the mean $\hat{F}_1$-score on the test set T w.r.t. the size of the training set X for nearly half of the CSPs from Table 2. For the remaining CSPs, there are no clear trends, and the $\hat{F}_1$-score fluctuates at constant problem-dependent levels. The run-time of GECS is also linear w.r.t. $|X|$, but problem-dependent and significantly larger for highly-dimensional CSPs, e.g., gsudoku.

Fig. 2 (top) shows the mean $\hat{F}_1$-score and 0.95-confidence interval on T for the best-of-run model on X. We verified $|X| \in \{100, 200, \ldots, 800\}$. The zqueens1 and zsteiner CSPs are excluded from this experiment because their ground-truth feasible regions contain only 92 and 53 feasible solutions in total, respectively. Thus, scaling against them cannot be verified in the above-mentioned range.

Fig. 2 (bottom) shows the mean and the 0.95-confidence interval of the run-time of GECS w.r.t. $|X|$. They are obtained using CPython 3.7.1 on Linux x64 running on a heterogeneous grid with 30 Core i5-8500 CPUs, 15 Core i7-4770 CPUs, and 15 Core i7-4790 CPUs. Note that the single-thread performance of all these CPUs is similar, and thus the measured times are comparable with a negligible error. Refer to [65] for the details.

5.3. How well does GECS compare to its competitors?

On the test-sets GECS scores a better mean best-of-run $\hat{F}_1$ on 16 out of the 18 benchmarks than the two state-of-the-art algorithms: OCCALS [20] and ESOCCS [19,36]. The advantage of GECS grows with dimensionality. What is more, for highly dimensional CSPs GECS yields well-formed MILP models where the other algorithms do not terminate in the time budget of 120h. OCCALS and ESOCCS algorithms were chosen as a baseline because they are known to beat several other algorithms (cf. Section 3).

In this comparison, OCCALS and ESOCCS were run with the parameter values shown in Table 5.[4] We use all benchmarks and the same training sets X for OCCALS, ESOCCS, and GECS; $|X| = \min\{400, |f(C^*)|\}$. OCCALS and ESOCCS synthesize models in LP format and we assess them directly using Eq. (7) and test set T, as the technical transformation from ZIMPL is not necessary (cf. Section 2.2). Note that the term L in Eq. (7) refers for OCCALS and ESOCCS to the number of characters in the LP format representation and GECS in the ZIMPL representation. This way we reward the use of the shorter high-level ZIMPL representation, and at the same time keep the magnitude of $\hat{F}_1$ unaffected by this discrepancy of calculating L due to the very small weight of L in

[4] For OCCALS, we picked the parameter values out of $k_{\min} \in \{1, 2, 3\}$ and $c_{\max} \in \{500, 1000\}$ that maximize the mean $\hat{F}_1$-score on the problems from Table 2. For ESOCCS, we use the parameter values being the result of tuning in both [19] and [36]. We use these reasonable defaults and the fixed time budget of 120h per run for all algorithms, as EAs are typically robust to specific parameter values and giving them equal computational budget is far more important for fair comparison [66,67]. Note that ESOCCS suffers from the curse of dimensionality that prevents its termination in 9 out of 18 benchmarks and leads to useless results in 5 out of the remaining 9 benchmarks (cf. Table 6). The tuning of 11 parameters of ESOCCS under these conditions would be time-consuming and unlikely to yield meaningful improvement, nor overcome the course of dimensionality.

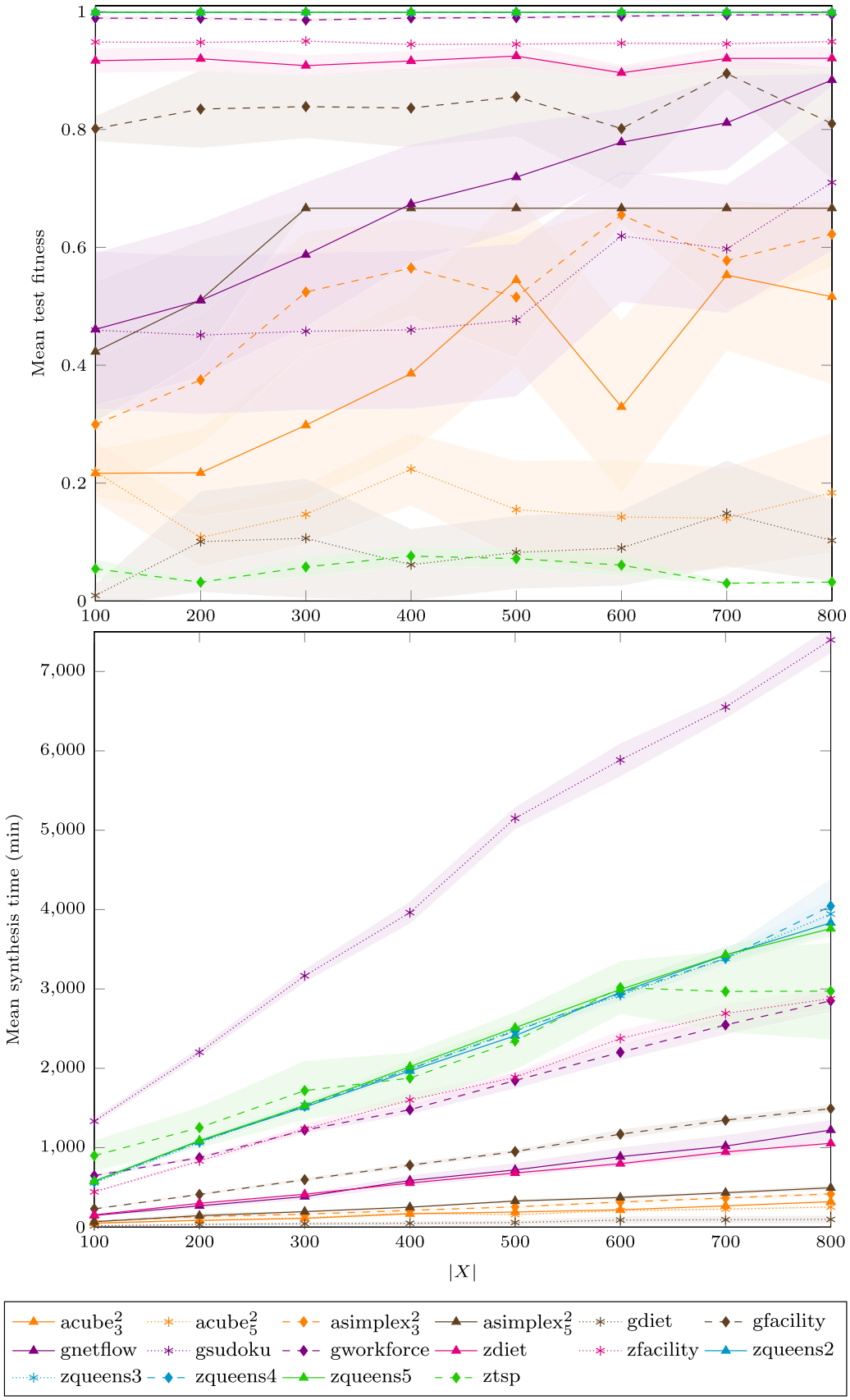

Fig. 2. Mean $\hat{F}_1$ on T of the best-of-run model on X (top) and mean run-time in minutes w.r.t. $|X|$ (bottom); shading reflects 0.95-confidence intervals.

Table 4
Mean $\hat{F}_1$ on V of the best of run model on X, best in bold; bars reflect 0.95-confidence intervals (cell height reflects 0.1); heatmap for $\hat{F}_1$: green for 1, red for 0; overall ranks of parameter settings; Kruskal-Wallis test reports $p > .999$.

	MUTATION: S POP/GEN: 500/60			CROSSOVER: V1 POP/GEN: 500/60		CROSSOVER: V1 MUTATION: I	
	CROSSOVER			MUTATION		POP/GEN	
Problem	S	V1	F2	I	C	250/120	750/40
$acube_3^2$	**0.646**	0.491	0.423	0.412	0.287	0.496	0.373
$acube_5^2$	**0.292**	0.164	0.157	0.211	0.195	0.202	0.153
$asimplex_3^2$	0.642	0.617	**0.664**	0.572	0.561	0.551	0.598
$asimplex_5^2$	0.619	**0.666**	**0.666**	**0.666**	0.597	**0.666**	**0.666**
gdiet	0.056	0.088	0.077	0.076	**0.126**	0.099	0.114
gfacility	0.779	**0.852**	0.800	0.837	0.851	0.803	0.841
gnetflow	0.478	0.683	0.546	0.661	**0.707**	0.632	0.596
gsudoku	0.447	0.446	0.447	0.453	0.450	**0.491**	0.450
gworkforce	0.988	0.988	0.987	0.989	**0.991**	0.987	0.989
zdiet	0.910	**0.932**	0.913	0.914	0.918	0.929	0.912
zfacility	**0.947**	0.943	0.912	**0.947**	0.945	0.923	0.946
zqueens1	**0.996**	0.974	0.978	0.995	0.989	0.938	0.953
zqueens2	**0.999**	**0.999**	**0.999**	**0.999**	**0.999**	**0.999**	**0.999**
zqueens3	**0.999**	**0.999**	**0.999**	**0.999**	**0.999**	**0.999**	**0.999**
zqueens4	**0.999**	**0.999**	**0.999**	**0.999**	**0.999**	**0.999**	**0.999**
zqueens5	**0.999**	**0.999**	**0.999**	**0.999**	**0.999**	**0.999**	**0.999**
zsteiner	**0.959**	0.936	0.934	0.939	0.945	0.930	0.934
ztsp	0.057	0.073	0.069	0.075	0.074	0.077	**0.078**
Rank:	4.139	3.806	4.806	3.444	3.583	4.139	4.083

Table 5
Parameters of OCCALS [20] and ESOCCS [19,36].

Algorithm	Parameter	Value
OCCALS	Min number of clusters k_{min}	1
	Number of constraints per cluster c_{max}	500
ESOCCS	Distribution estimation	Expectation maximization
	Convergence toleration	10^{-11}
	Random initializations	100
	Iteration limit	10^5
	Significance level ρ	0.01
	Population size μ	400
	Offspring to parent ratio λ/μ	3
	Initialization	Bounding box
	Mutation probability p_m	1.0
	Unlabeled to feasible set ratio $\lvert U\rvert/\lvert X\rvert$	2
	Constraint reuse limit r	3

Eq. (7). The input to OCCALS and ESOCCS differ from the input to GECS only in the lack of ZIMPL snippet defining the sets, the parameters, and the variables because OCCALS and ESOCCS do not use this information. To make the comparison fairer, we extend the LP format models produced by OCCALS and ESOCCS with the domains and the bounds of the variables given in the original input to CSP.

Table 6 shows the mean and 0.95-confidence interval of $\hat{F}_1$ on T for the best-of-run model on X found by GECS, OCCALS, and ESOCCS. The negative values occur when the F_1 score is close to 0 and $L > 10^3$. Missing values correspond to runs that did not finish within the 120h time budget. GECS outperforms OCCALS and ESOCCS in 16 out of 18 problems. OCCALS and ESOCCS suffer from the curse of dimensionality [8], as the $\hat{F}_1$ score for them is close to zero for most problems with the number of variables $n > 8$. The only exception is the result of OCCALS for the gnetflow problem of 50 dimensions which we attribute to the specific structure of this problem. To this end, GECS seems to be much more robust to the curse of dimensionality, as it effectively handles even $n = 729$ variables. The Kruskal-Wallis test [68] yields a p-value < 0.001

Table 6
Mean $\hat{F}_1$ on T of the best-of-run model on X; best in bold; bar height reflects 0.95-confidence interval (cell height reflects 0.1); heatmap for means: green for 1, red for 0; mean ranks; p-values of the signed rank test with Bonferroni correction of GECS vs the others, $\leq 0.05/2$ in bold; missing values for the runs unfinished within the 120h limit – for the rank and p-value calculation the missing value is considered worse than all others.

Problem	GECS	OCCALS [20]	ESOCCS [19, 36]
$acube_3^2$	0.386	−0.002	**0.745**
$acube_5^2$	**0.224**	−0.006	0.029
$asimplex_3^2$	**0.565**	0.027	0.034
$asimplex_5^2$	**0.666**	−0.016	−0.007
gdiet	**0.061**	−0.044	−0.000
gfacility	**0.836**	−0.005	−0.001
gnetflow	0.674	**0.924**	
gsudoku	**0.460**		
gworkforce	**0.989**	−0.004	
zdiet	**0.916**	0.886	0.790
zfacility	**0.944**	0.000	
zqueens1	**0.994**	0.652	0.685
zqueens2	**0.999**	−0.021	
zqueens3	**0.999**	−0.003	
zqueens4	**0.999**	−0.004	
zqueens5	**0.999**	−0.003	
zsteiner	**0.937**	0.905	0.903
ztsp	**0.076**	−0.004	
Rank:	1.111	2.361	2.528
p-value:		**0.001**	**0.001**

and so we observe support for the difference between these algorithms. The last row of Table 6 shows the p-values of the post-hoc analysis using the signed rank test with Bonferroni correction [68] and reveals the superiority of GECS over both other algorithms.

5.4. How well do the synthesized models work in optimization?

For 14 out of 18 problems GECS yielded at least once per 25 runs a model in which the objective value of the optimal solution equals the objective value of the optimal solution to the corresponding ground truth model. For 11 out of 18 problems GECS yielded such a model in all 25 runs. However, only in 4 out of 18 problems GECS yielded at least once a model in which the values of variables in the optimal solution equal the values of variables in one of the optimal solutions to the ground truth model.

We collected these statistics by optimizing using the Gurobi solver [4] the models synthesized by GECS for $|X| = 400$ and supplemented with the objective functions from the corresponding ground truth models. Table 7 shows the mean and 0.95-confidence interval of the fraction of the optimal solutions to the synthesized models that equal in the objective value and in the values of variables to one of the optimal solutions to the ground truth models. Technically, we verify whether the optimal solution to the synthesized model lies within the feasible region of the ground truth model and its objective value equals the objective value of the optimal solution to the ground truth model. This is because many different solutions with the same objective value may exist and the solver is free to return any of them. The fractions vary from 0 to 1 depending on the problem and are higher when only the objective value is considered. This may mean that some synthesized models have factually infeasible optimal solutions. This result shows also that for 4 out of 18 problems GECS at least once per 25 runs finds a model with the correct location of the boundary of the feasible region in close proximity of the ground truth optimal solution.

Table 7
Mean fraction of the optimal solutions to the synthesized models equal in the objective value (the middle column), the objective value and values of variables to one of the optimal solutions to the ground truth model (the last column); bar height reflects 0.95-confidence interval (cell height reflects 0.2); heatmap reflects means: red for 0, green for 1.

Problem	Objective	Solution
$acube_3^2$	1.00	0.00
$acube_5^2$	1.00	0.00
$asimplex_3^2$	1.00	0.00
$asimplex_5^2$	1.00	0.00
gdiet	0.00	0.00
gfacility	0.04	0.04
gnetflow	0.16	0.16
gsudoku	1.00	0.00
gworkforce	1.00	1.00
zdiet	0.20	0.20
zfacility	0.00	0.00
zqueens1	1.00	0.00
zqueens2	1.00	0.00
zqueens3	1.00	0.00
zqueens4	1.00	0.00
zqueens5	1.00	0.00
zsteiner	0.00	0.00
ztsp	0.00	0.00

6. Discussion

A MILP model in ZIMPL is general in the sense that it represents an entire class of real-world objects sharing the same constraints and the same objective function and differing only in the values of the parameters and dimensions. For instance, for the zdiet MILP model in ZIMPL, two diet plans with different food dimensions have the same constraints and the objective function and differ only in the food set. This is a qualitative difference w.r.t. the LP format [4] that effectively stores a weighted sum of variables as an objective function and a weighted sum of variables compared to a constant as a constraint, without any particular meaning of the weights. Hence, two models in the LP format for two different food dimensions differ in the constraints and the objective function.

The generality of the ZIMPL representation offers great flexibility in modeling of real-world objects whose details change over time. This also facilitates synthesis of high-quality constraints by GECS, as it effectively looks for the symbols for placeholders in the templates of common structures, e.g., `'sum <j> in '<set_S_S>':' <coeff_var_SS>`, where `<set_S_S>` and `<coeff_var_SS>` are effectively the placeholders for respectively a set symbol and a variable symbol optionally prepended with multiplication by a constant (cf. Section 4.2). This approach is in stark contrast to the previous algorithms for the synthesis of LP/NLP models (cf. Section 3.3) that effectively look for the optimal values for real-valued weights. In Section 5.3 we showed that GECS scores better than two other algorithms that look for the values of weights, especially when the number of variables in the CSP is large ($n > 8$).

Note that GECS seems to be resistant to the curse of dimensionality [8], as its performance does not deteriorate much with n. We hypothesize that this is because n is not really a dimension of a CSP as posed in Section 2.3 and does not influence the size of the resulting ZIMPL model. In contrast, the size of the model in the LP format depends on n, and thus the curse of dimensionality occurs for the algorithms producing models in the LP format.

However, the number of symbols *is* a dimension of a CSP, and as Tables 6 and 7 show, it may impair performance of GECS. It achieves the worst results for the gdiet and ztsp CSP that both feature seven symbols. Note that the number of symbols is not the only factor that influences the difficulty of a CSP: for the gfacility and zfacility CSP with eight and nine symbols, respectively, GECS scores relatively high $\hat{F}_1$. We hypothesize that the other factor is the distribution of *correct* models in the search space of models constrained by the problem-specific grammar (cf. Section 4.2). The verification of this hypothesis requires future work.

We have selected a diverse set of problems with different characteristics as outlined in Table 2. We note that the application of any optimization algorithm to a specific real-world problem requires some degree of tuning to the specific characteristics of that problem. The diversity of problems and settings examined in this study, while cannot capture all possible real-world scenarios that may occur, provide some confidence to suggest the robustness of the algorithm in this regard. The performance of GECS seems to be largely independent of the parameter values, since all parameter settings employed in Section 5.1 lead to similar performance. This is an advantageous property of GECS that allows its use *as is*, without the need for time-consuming problem-specific parameter tuning.

GECS scales linearly with the size of the training set X for some CSPs considered in this study. For other CSPs, its $\hat{F}_1$ scores are roughly constant in the considered range of $|X|$ but consistently ≥ 0.8 and often close to 1.0. There is also a group of difficult CSPs for which the $\hat{F}_1$ score is low and does not change much in the considered range of $|X|$. For 14 out of 18 CSPs GECS produced at least once a model whose objective value of the optimal solution equals the objective value of the optimal solution to the corresponding ground truth model. *These observations partially confirm the main research hypothesis from Section 1: given a large enough training set GECS produces a MILP model equivalent to the ground truth MILP model for some of the CSPs considered in this study.* It also shows that the amount of training information required to reconstruct the ground-truth model depends on that model. For easier ground-truth models, the training set as small as 100 examples suffices, for others it requires much more than 800 examples.

The time-complexity of GECS is linear w.r.t. $|X|$. This is another advantage that opens the possibility to process large training sets. However, the Python-based implementation is slow due to large interpreter overhead. We expect that by rewriting GECS to a compiled language, the run-times shown in Fig. 2 reduce by an order of magnitude.

The input information for GECS is usually easy to provide: the variables, parameters, and dimensions of the object are usually straightforward to identify, and the examples of the feasible solutions can be collected by monitoring the behavior of the object. A synthesized model may be employed as is, to optimize and simulate the modeled object. And, as we have seen in previous studies using a Grammatical Evolution approach for the design of schedulers in the wireless telecommunications networks domain [69], even if the model does not completely fit reality, the human readable nature of the model means it is still easy to augment by the expert — please consult the synthesized models in Appendix A. Hence, we achieved the goal to reduce the burden on the expert in modeling and optimization.

GECS is not a silver bullet. The grammar template from Section 4.2.1 influences the representational bias of GECS. Designing *right* grammar template relies on many decisions that influence the generality, meant as the range of reachable constraints, and the size of the search space. Too *'tight'* grammar template would prevent some constraints to be found, and too *'loose'* may impair search performance. The grammar template from Section 4.2.1 guides well GECS for the CSPs considered in this work, however, solving an other CSP may require extra effort in adapting the grammar template.

GECS deserves future work on robustness to noise to make it applicable to real-world CSPs, where measurement errors and uncertainties typically occur. Although not equipped with an explicit noise-handling

mechanism, we expect GECS to handle well a limited amount of noise, like most Evolutionary Algorithms do [70–72]. The above-mentioned representational bias would prevent overfitting to noisy examples that do not match the parameters and the sets provided. It is also unlikely that the inclusion of the noisy example in the feasible region of the produced model increases recall without deteriorating precision – the components of the $\hat{F}_1$-score fitness function. Hence, GE would keep only the models fitting large-enough part of the examples.

7. Conclusions and future work

We formally posed the Constraint Synthesis Problem for MILP models in ZIMPL high-level modeling language, proposed the GECS algorithm aimed at solving CSP, and verified experimentally its properties and performance w.r.t. the contemporary algorithms. GECS synthesizes MILP models guided by the grammar of ZIMPL and the exemplary solutions. This is a qualitatively different approach than of the majority of previous algorithms, which optimize numerically the weights in the constraints. This mode of work offers GECS great performance boost w.r.t. contemporary algorithms and resistance to the curse of dimensionality. The resulting MILP models may be used 'as is' and adapted to different objects of the same class by simply providing the values of their parameters and dimensions.

Possible extensions to GECS include the synthesis of the objective function in the same way as the constraints. However, this would require extending the training set with the values of the objective function. The input to GECS is currently restricted to fixed values for sets, parameters, and variables. By relaxing this requirement it would be possible to synthesize MILP models from more general data, e.g., corresponding to different instances of the modeled object. Another issue deserving investigation is noise handling. GECS can be also extended for different modeling languages, e.g., AMPL [16] and GAMS [17], by employing their grammars and parsers.

Funding

T.P. Pawlak acknowledges the support of National Science Centre Poland grant 2016/23/D/ST6/03735, and the National Centre for Research and Development Poland grant LIDER/14/0086/L-10/18/NCBR/2019. M. O'Neill acknowledges the support of Science Foundation Ireland grants 13/IA/1850 and 13/RC/2094.

Declaration of Competing Interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Appendix A. The best MILP models

We show the best MILP models w.r.t. $\hat{F}_1$ on test set synthesized by GECS in experiments in Section 5. They are concatenated from the input ZIMPL snippet and the synthesized constraints. We skip the model for the zdiet CSP, since Section 2.2 shows it.

```
# problem: acube_3^2  fitness: 1.000
set N := {1 to 3};
set K := {1 to 2};
param d := 2.7;
var x[<i> in N] real >= i - i*max(K)*d <= i + 2*i*max(K)*d;
var b[<k> in K] binary;
subto constraint1: sum <i> in K: b[i] >= 1;
subto constraint2: forall <i> in K:
  forall <j> in K:
    forall <k> in N: vif b[i] == 1 then x[k] >= k * i end;
```

```
# problem: acube_5^2  fitness: 1.000
set N := {1 to 5};
set K := {1 to 2};
param d := 2.7;
var x[<i> in N] real >= i - i * max(K)*d <= i + 2*i*max(K)*d;
var b[<k> in K] binary;
subto constraint1: forall <i> in N: x[i] >= i;
subto constraint2: forall <i> in K: sum <j> in K: b[j] == 1;
```

```
# problem: asimplex_3^2  fitness: 0.726
set N := {1 to 3};
set K := {1 to 2};
param d := 2.7;
param slope1 := 3.7320508076; # cot(pi/12)
param slope2 := 0.2679491924; # tan(pi/12)
var x[N] real >= -1 <= 2 * max(K) + d;
var b[K] binary;
subto constraint1: forall <i> in K:
  sum <j> in K: 1 * b[j] == (card(K) - 1);
subto constraint2: forall <i> in N with <card(N)> in {2..card(N)}:
  forall <j> in K:
    sum <k> in N: x[i] >= j;
```

```
# problem: asimplex_5^2  fitness: 0.666
set N := {1 to 5};
set K := {1 to 2};
param d := 2.7;
param slope1 := 3.7320508076; # cot(pi/12)
param slope2 := 0.2679491924; # tan(pi/12)
var x[N] real >= -1 <= 2 * max(K) + d;
var b[K] binary;
subto constraint1: forall <i> in K: i * b[i] - 2 * b[i] <= i;
```

```
# problem: gdiet  fitness: 0.939
set CAT := {"kcal", "protein", "fat", "sodium"};
param minNutr[CAT]:=<"kcal">1800,<"protein"> 91,<"fat"> 0,<"sodium"> 0;
param maxNutr[CAT]:=<"kcal">2200,<"protein">1E6,<"fat">65,<"sodium">1779;
set FOOD := {"hamburger", "chicken", "hot dog", "fries", "macaroni",
  "pizza", "salad", "milk", "ice cream"};
param cost[FOOD] := <"hamburger"> 2.49, <"chicken"> 2.89,
  <"hot dog"> 1.50, <"fries">1.89, <"macaroni"> 2.09, <"pizza"> 1.99,
  <"salad"> 2.49, <"milk"> 0.89, <"ice cream"> 1.59;
param nutr[FOOD * CAT] :=
            |"kcal","protein","fat","sodium"|
|"hamburger"|   410,       24,   26,    730 |
|"chicken"  |   420,       32,   10,   1190 |
|"hot dog"  |   560,       20,   32,   1800 |
|"fries"    |   380,        4,   19,    270 |
|"macaroni" |   320,       12,   10,    930 |
|"pizza"    |   320,       15,   12,    820 |
|"salad"    |   320,       31,   12,   1230 |
|"milk"     |   100,        8,  2.5,    125 |
|"ice cream"|   330,        8,   10,    180 |;
var buy[FOOD] real >= 0 <= 1E6;
minimize cost: sum <f> in FOOD: cost[f] * buy[f];
subto constraint1: forall <i> in CAT:
  sum <j> in FOOD: nutr[j,i] * buy[j] >= minNutr[i];
subto constraint2:
  forall <i> in CAT with <(card(FOOD)-2)> in {1..card(FOOD)}:
    sum <j> in FOOD: nutr[j,i] * buy[j] <= maxNutr[i];
subto constraint3:
  forall <i> in FOOD with <(card(CAT)-1)> in {2..(card(CAT)-3)}:
    buy[i] - buy[i] == cost[i];
```

```
# problem: gfacility  fitness: 0.996
set WAREHOUSES := {1 to 4};
param demand[WAREHOUSES] := <1> 15, <2> 18, <3> 14, <4> 20;
set PLANTS := {1 to 5};
param capacity[PLANTS] := <1> 20, <2> 22, <3> 17, <4> 19, <5> 18;
param fixedCosts[PLANTS]:=<1>12000,<2>15000,<3>17000,<4>13000,<5>16000;
param transCosts[WAREHOUSES * PLANTS] :
    |    1,    2,    3,    4,    5|
|  1| 4000, 2000, 3000, 2500, 4500|
|  2| 2500, 2600, 3400, 3000, 4000|
|  3| 1200, 1800, 2600, 4100, 3000|
|  4| 2200, 2600, 3100, 3700, 3200|;
var open[PLANTS] binary;
var transport[WAREHOUSES * PLANTS] real >= 0 <= 1000;
minimize cost: sum <p> in PLANTS: fixedCosts[p] * open[p]
  + sum <w,p> in WAREHOUSES * PLANTS: transCosts[w,p] * transport[w,p];
subto constraint1: forall <i> in PLANTS
  with <(card(WAREHOUSES)-3)> in {2..(card(WAREHOUSES)-3)}:
    forall <j> in WAREHOUSES:
      sum <k> in PLANTS: transport[j,k]
        - sum <k> in PLANTS: open[k] <= demand[j];
subto constraint2: forall <i> in WAREHOUSES
  with <(card(WAREHOUSES)-3)> in {1..(card(PLANTS)-3)}:
    forall <j> in WAREHOUSES:
      sum <k> in WAREHOUSES: transport[k,i]
        - sum <k> in PLANTS: open[j] <= capacity[i];
subto constraint3: sum <i> in WAREHOUSES: open[i] >= 1;
subto constraint4: forall <i> in WAREHOUSES
  with <card(PLANTS)> in {1..(card(PLANTS)-2)}:
    sum <j> in WAREHOUSES: transport[j,i]
      - sum <j> in PLANTS: transport[j,i] <= 1;
subto constraint5: sum <i> in PLANTS: open[i] >= 1;
```

```
# problem: gnetflow  fitness: 0.912
set COMMODITIES := {"Pencils", "Pens"};
set NODES := {"Detroit", "Denver", "Boston", "New York", "Seattle"};
param capacity[NODES * NODES] :=
  <"Detroit", "Boston"> 100, <"Detroit", "New York"> 80,
  <"Detroit", "Seattle"> 120, <"Denver", "Boston"> 120,
  <"Denver", "New York"> 120, <"Denver", "Seattle"> 120
  default 0;
param cost[COMMODITIES * NODES * NODES] :=
  <"Pencils","Detroit","Boston">10,<"Pencils","Detroit","New York">20,
  <"Pencils","Detroit","Seattle">60, <"Pencils","Denver","Boston">40,
  <"Pencils","Denver","New York">40, <"Pencils","Denver","Seattle">30,
  <"Pens","Detroit","Boston">20, <"Pens","Detroit","New York">20,
  <"Pens","Detroit","Seattle">80, <"Pens","Denver","Boston">60,
  <"Pens","Denver","New York">70, <"Pens","Denver","Seattle">30
  default 1e10;
param inflow[COMMODITIES * NODES] :=
  <"Pencils", "Detroit"> 50, <"Pencils", "Denver"> 60,
  <"Pencils", "Boston"> -50, <"Pencils", "New York"> -50,
  <"Pencils", "Seattle"> -10, <"Pens", "Detroit"> 60,
  <"Pens", "Denver"> 40, <"Pens", "Boston"> -40,
  <"Pens", "New York"> -30, <"Pens", "Seattle"> -30
  default 0;
var flow[COMMODITIES * NODES * NODES] real >= 0;
minimize total_flow: sum <c,n1,n2> in COMMODITIES*NODES*NODES:
  cost[c,n1,n2] * flow[c,n1,n2];
subto constraint1: forall <i> in NODES:
  forall <j> in NODES:
    sum <k> in COMMODITIES: flow[k,i,j] <= capacity[i,j];
subto constraint2: forall <i> in COMMODITIES:
  forall <j> in NODES:
    sum <k> in NODES: flow[i,j,k]
      - sum <k> in NODES: flow[i,k,j] == inflow[i,j];
```

```
# problem: gsudoku fitness: 0.998
set N := {1 to 9};
param s := sqrt(max(N));
set S[<n> in N] :=
  {<i,j> in {floor((n-1)/s)*s+1 to (floor((n-1)/s)+1)*s}
    * {((n-1) mod s)*s+1 to (((n-1) mod s)+1)*s}};
param grid[N * N] :=
   | 1, 2, 3, 4, 5, 6, 7, 8, 9|
| 1| 0, 0, 0, 0, 0, 0, 0, 0, 0|
| 2| 0, 0, 0, 0, 0, 0, 0, 0, 0|
| 3| 0, 0, 0, 0, 0, 0, 0, 0, 0|
| 4| 0, 0, 0, 0, 0, 0, 0, 0, 0|
| 5| 0, 0, 0, 0, 0, 0, 0, 0, 0|
| 6| 0, 0, 0, 0, 0, 0, 0, 0, 0|
| 7| 0, 0, 0, 0, 0, 0, 0, 0, 0|
| 8| 0, 0, 0, 0, 0, 0, 0, 0, 0|
| 9| 0, 0, 0, 0, 0, 0, 0, 0, 0|;
var vars[<i,j,v> in N*N*N] integer
  >= (if grid[i,j] == v then 1 else 0 end) <= 1;
subto constraint1: forall <i> in N:
  sum <j,k> in S[i]: vars[k,j,i] <= s;
```

```
# problem: gworkforce fitness: 0.997
set SHIFTS := {"Mon1","Tue2","Wed3","Thu4","Fri5","Sat6","Sun7",
"Mon8", "Tue9", "Wed10", "Thu11", "Fri12", "Sat13", "Sun14"};
set WORKERS := {"Amy", "Bob", "Cathy", "Dan", "Ed", "Fred", "Gu"};
param shiftRequirements[SHIFTS] := <"Mon1">3, <"Tue2">2, <"Wed3">4,
  <"Thu4">2, <"Fri5">5, <"Sat6">4, <"Sun7">4, <"Mon8">2, <"Tue9">2,
  <"Wed10"> 3, <"Thu11"> 4, <"Fri12"> 5, <"Sat13"> 7, <"Sun14"> 5;
param pay[WORKERS] := <"Amy"> 10, <"Bob"> 12, <"Cathy"> 10,
  <"Dan"> 8, <"Ed"> 8, <"Fred"> 9, <"Gu"> 11;
param availability[WORKERS * SHIFTS] :=
  <"Amy","Tue2">1,<"Amy","Wed3">1,<"Amy","Fri5">1,<"Amy","Sun7">1,
  <"Amy","Tue9">1,<"Amy","Wed10">1,<"Amy","Thu11">1,
  <"Amy","Fri12">1,<"Amy","Sat13">1,<"Amy","Sun14">1,
  <"Bob","Mon1">1,<"Bob","Tue2">1,<"Bob","Fri5">1,<"Bob","Sat6">1,
  <"Bob","Mon8">1,<"Bob","Thu11">1,<"Bob","Sat13">1,
  <"Cathy","Wed3">1,<"Cathy","Thu4">1,<"Cathy","Fri5">1,
  <"Cathy","Sun7">1,<"Cathy","Mon8">1,<"Cathy","Tue9">1,
  <"Cathy","Wed10">1,<"Cathy","Thu11">1,
  <"Cathy","Fri12">1,<"Cathy","Sat13"> 1,<"Cathy","Sun14">1,
  <"Dan","Tue2">1,<"Dan","Wed3">1,<"Dan","Fri5">1,<"Dan","Sat6">1,
  <"Dan","Mon8">1,<"Dan","Tue9">1,<"Dan","Wed10">1,<"Dan","Thu11">1,
  <"Dan","Fri12">1,<"Dan","Sat13">1,<"Dan","Sun14">1,
  <"Ed","Mon1">1,<"Ed","Tue2">1,<"Ed","Wed3">1,<"Ed","Thu4">1,
  <"Ed","Fri5">1,<"Ed","Sun7">1,<"Ed","Mon8">1,<"Ed","Tue9">1,
  <"Ed","Thu11">1,<"Ed","Sat13">1,<"Ed","Sun14">1,
  <"Fred","Mon1">1,<"Fred","Tue2">1,<"Fred","Wed3">1,
  <"Fred","Sat6">1,<"Fred","Mon8">1,<"Fred","Tue9">1,
  <"Fred","Fri12">1,<"Fred","Sat13">1,<"Fred","Sun14">1,
  <"Gu","Mon1">1,<"Gu","Tue2">1,<"Gu","Wed3">1,<"Gu","Fri5">1,
  <"Gu","Sat6">1,<"Gu","Sun7">1,<"Gu","Mon8">1,<"Gu","Tue9">1,
  <"Gu","Wed10">1,<"Gu","Thu11">1,<"Gu","Fri12">1,<"Gu","Sat13">1,
  <"Gu","Sun14">1
  default 0;
var x[<w, s> in WORKERS * SHIFTS] real >= 0 <= availability[w, s];
minimize cost: sum <w, s> in WORKERS * SHIFTS: pay[w] * x[w,s];
subto constraint1: forall <i> in SHIFTS:
  sum <j> in WORKERS: x[j,i] == shiftRequirements[i];
```

```
# problem: zfacility fitness: 0.974
set PLANTS := {"A", "B", "C", "D"};
set STORES := {1 .. 9};
set PS     := PLANTS * STORES;
param building[PLANTS] := <"A"> 500, <"B"> 600, <"C"> 700, <"D"> 800;
param capacity[PLANTS] := <"A">  40, <"B">  55, <"C">  73, <"D">  90;
param demand  [STORES] := <1> 10, <2> 14, <3> 17, <4> 8, <5> 9,
                          <6> 12, <7> 11, <8> 15, <9> 16;
param transport[PS] :=
      |  1,  2,  3,  4,  5,  6,  7,  8,  9 |
  |"A"| 55,  4, 17, 33, 47, 98, 19, 10,  6 |
  |"B"| 42, 12,  4, 23, 16, 78, 47,  9, 82 |
  |"C"| 17, 34, 65, 25,  7, 67, 45, 13, 54 |
  |"D"| 60,  8, 79, 24, 28, 19, 62, 18, 45 |;
var x[PS]     binary;  # Is plant p suppling store s ?
var z[PLANTS] binary;  # Is plant p build ?
minimize cost: sum <p> in PLANTS: building[p] * z[p]
  + sum <p,s> in PS: transport[p,s] * x[p,s];
subto constraint1: forall <i> in STORES:
  forall <j> in STORES:
    sum <k> in PLANTS: x[k,j] == 1;
subto constraint2: sum <i,j> in PS: z[i] >= 1;
subto constraint3: forall <i> in STORES:
  forall <j> in STORES:
    forall <k> in PLANTS:
      vif x[k,i] == 1 then 2 * x[k,j] + j * z[k] >= 1 end;
subto constraint4:
  forall <i> in PLANTS with <(card(PS)-2)> in {3..card(PS)}:
    sum <j> in STORES: demand[j] * x[i,j] <= capacity[i];
```

```
# problem: zqueens1 fitness: 0.999
param queens := 8;
set I := {1..queens};
set P := {<i,j> in I * I with i < j};
var x[I] integer >= 1 <= queens;
subto constraint1: forall <i,j> in P: vabs(x[j] - x[i]) >= 1;
```

```
# problems: zqueens2 - zqueens4 (models equal) fitness: 0.999
param columns := 8;
set I   := {1..columns};
set IxI := I * I;
set TABU[<i,j> in IxI] := {<m,n> in IxI with
 (m != i or n != j) and (m == i or n == j or abs(m-i) == abs(n-j))};
var x[IxI] binary;
maximize queens: sum <i,j> in IxI: x[i,j];
subto constraint1: forall <i> in I: sum <j> in I: x[j,i] <= 1;
```

```
# problem: zqueens5 fitness: 1.000
param columns := 8;
set I   := {1..columns};
set IxI := I * I;
var x[IxI] binary;
maximize queens: sum <i,j> in IxI: x[i,j];
subto constraint1: forall <i> in I: sum <j> in I: x[j,i] <= 1;
```

```
# problem: zsteiner fitness: 0.999
set V := {1..5};
set E := {<1,2>, <1,4>, <2,3>, <2,4>, <3,4>, <3,5>, <4,5>};
set T := {1, 3, 5};
param c[E]:=<1,2> 1,<1,4> 2,<2,3> 3,<2,4> 4,<3,4> 5,<3,5> 6,<4,5> 7;
var x[E] binary;
minimize cost: sum <a,b> in E: c[a,b] * x[a,b];
set P[] := powerset(V);
set I   := indexset(P);
subto constraint1: forall <i> in V: sum <j,k> in E: 2 * x[j,k] >= i;
```

```
# problem: ztsp fitness: 0.128
set V := {"Sylt", "Flensburg", "Neumunster", "Husum", "Schleswig",
  "Ausacker", "Rendsburg", "Lubeck", "Westerland", "Segeberg"};
set E := {<i,j> in V * V with i < j};
set P[] := powerset(V \ {ord(V,1,1)});
set K := indexset(P) \ {0};
param px[V] := <"Sylt"> 1, <"Flensburg"> 3, <"Neumunster"> 2,
  <"Husum"> 1, <"Schleswig"> 3, <"Ausacker"> 2, <"Rendsburg"> 1,
  <"Lubeck"> 4, <"Westerland"> 0, <"Segeberg"> 2;
param py[V] := <"Sylt"> 1, <"Flensburg"> 1, <"Neumunster"> 2,
  <"Husum"> 3, <"Schleswig"> 3, <"Ausacker"> 4, <"Rendsburg"> 4,
  <"Lubeck"> 4, <"Westerland"> 1, <"Segeberg"> 3;
defnumb dist(a,b) := sqrt((px[a] - px[b])^2 + (py[a] - py[b])^2);
var x[E] binary;
minimize cost: sum <i,j> in E : dist(i,j) * x[i, j];
subto constraint1: forall <i> in K:
  sum <j,k> in E: x[j,k] + sum <j,k> in E: 3*x[j,k] <= (card(E)-1);
```

Appendix B. List of abbreviations and symbols

BNF	Backus-Naur form of grammar
CP	Constraint Programming
CSP	Constraint Synthesis Problem
GE	Grammatical Evolution
GECS	Grammatical Evolution for Constraint Synthesis
HaR	Hit-and-Run
LP	Linear Programming
MILP	Mixed-Integer Linear Programming
NLP	Non-Linear Programming
SMT	Satisfiability Modulo Theories
ZIMPL	Zuse Institut Mathematical Programming Language
m	Model
P	Set of parameters p
S	Set of dimension sets s
$\mathbf{x}$	Example (a vector of variables x)
X	Training set
T	Test set
V	Validation set
$c(\mathbf{x})$	Constraint
C	Set of constraints (C^* for the optimal C)
$f(C)$	Feasible region of C
$F_1(C)$	F_1-score ($\hat{F}_1(C)$ for approximation)

CRediT authorship contribution statement

Tomasz P. Pawlak: Conceptualization, Methodology, Software, Validation, Formal analysis, Investigation, Resources, Writing - original draft, Visualization, Supervision, Project administration, Funding acquisition. **Michael O'Neill:** Validation, Resources, Writing - review & editing, Funding acquisition.

References

[1] H. Williams, Model Building in Mathematical Programming, Wiley, 2013.
[2] NEOS solver statistics, Accessed 2020-05-26,https://neos-server.org/neos/report.html.
[3] T. Koch, Rapid Mathematical Programming, Technische Universität Berlin, 2004 Ph.D. thesis.
[4] Gurobi Optimization, LLC, Gurobi optimizer reference manual, 2020, http://gurobi.com.
[5] M. O'Neill, C. Ryan, Grammatical evolution: Evolutionary automatic programming in a arbitrary language, Genetic programming, volume 4, Kluwer Academic Publishers, 2003, doi:10.1007/978-1-4615-0447-4.
[6] M. O'Neill, M. Nicolau, A. Agapitos, Experiments in program synthesis with grammatical evolution: A focus on integer sorting, in: C.A. Coello Coello (Ed.), Proceedings of the 2014 IEEE Congress on Evolutionary Computation, Beijing, China, 2014, pp. 1504–1511, doi:10.1109/CEC.2014.6900578.
[7] F.D. O'Neill M., The Elephant in the Room: Towards the Application of Genetic Programming to Automatic Programming, Springer, 2019, pp. 179–192.
[8] R. Bellman, Dynamic programming, Dover Books on Computer Science, Dover Publications, 2013.
[9] T. Koch, Zimpl user guide for version 3.3.6, 2018, https://zimpl.zib.de/.
[10] P. Flach, Machine learning: The art and science of algorithms that make sense of data, Cambridge University Press, New York, NY, USA, 2012.
[11] G.N. Lance, W.T. Williams, Mixed-data classificatory programs i - agglomerative systems., Australian Computer Journal 1 (1) (1967) 15–20.
[12] R.L. Smith, The hit-and-run sampler: A globally reaching markov chain sampler for generating arbitrary multivariate distributions, in: Proceedings of the 28th Conference on Winter Simulation, in: WSC '96, IEEE Computer Society, Washington, DC, USA, 1996, pp. 260–264, doi:10.1145/256562.256619.
[13] S.S. Khan, M.G. Madden, One-class classification: taxonomy of study and review of techniques, Knowl Eng Rev 29 (3) (2014) 345–374, doi:10.1017/S026988891300043X.
[14] H. Edelsbrunner, D. Kirkpatrick, R. Seidel, On the shape of a set of points in the plane, IEEE Trans. Inf. Theory 29 (4) (1983) 551–559, doi:10.1109/TIT.1983.1056714.
[15] R. Seidel, The upper bound theorem for polytopes: an easy proof of its asymptotic version, Computational Geometry 5 (2) (1995) 115–116, doi:10.1016/0925-7721(95)00013-Y.
[16] R. Fourer, D. Gay, B. Kernighan, AMPL: A modeling language for mathematical programming, Scientific Press series, Thomson/Brooks/Cole, 2003.
[17] General Algebraic Modeling System, 2019, https://www.gams.com/.
[18] P. Kudła, T.P. Pawlak, One-class synthesis of constraints for mixed-integer linear programming with C4.5 decision trees, Appl Soft Comput 68 (2018) 1–12, doi:10.1016/j.asoc.2018.03.025.
[19] T.P. Pawlak, Synthesis of mathematical programming models with one-class evolutionary strategies, Swarm Evol Comput 44 (2019) 335–348, doi:10.1016/j.swevo.2018.04.007.
[20] D. Sroka, T.P. Pawlak, One-class constraint acquisition with local search, in: Proceedings of the Genetic and Evolutionary Computation Conference, in: GECCO '18, ACM, New York, NY, USA, 2018, pp. 363–370, doi:10.1145/3205455.3205480.
[21] M. Karmelita, T.P. Pawlak, CMA-ES for one-class constraint synthesis, in: Proceedings of the 2020 Genetic and Evolutionary Computation Conference, in: GECCO' 20, Association for Computing Machinery, New York, NY, USA, 2020, pp. 859–867, doi:10.1145/3377930.3389807.
[22] T.P. Pawlak, K. Krawiec, Automatic synthesis of constraints from examples using mixed integer linear programming, Eur J Oper Res 261 (3) (2017) 1141–1157, doi:10.1016/j.ejor.2017.02.034.
[23] T.P. Pawlak, K. Krawiec, Synthesis of mathematical programming constraints with genetic programming, in: M. Castelli, J. McDermott, L. Sekanina (Eds.), EuroGP 2017: Proceedings of the 20th European Conference on Genetic Programming, LNCS, volume 10196, Springer Verlag, Amsterdam, 2017, pp. 178–193, doi:10.1007/978-3-319-55696-3_12.
[24] M. Lombardi, M. Milano, A. Bartolini, Empirical decision model learning, Artif Intell 244 (Supplement C) (2017) 343–367, doi:10.1016/j.artint.2016.01.005.
[25] E.A. Schede, S. Kolb, S. Teso, Learning linear programs from data, in: 2019 IEEE 31st International Conference on Tools with Artificial Intelligence (ICTAI), 2019, pp. 1019–1026, doi:10.1109/ICTAI.2019.00143.
[26] N. Megiddo, On the complexity of polyhedral separability, Discrete & Computational Geometry 3 (4) (1988) 325–337, doi:10.1007/BF02187916.
[27] A.L. Blum, R.L. Rivest, Training a 3-node neural network is np-complete, Neural Networks 5 (1) (1992) 117–127, doi:10.1016/S0893-6080(05)80010-3.
[28] R. Alur, R. Bodik, G. Juniwal, M.M.K. Martin, M. Raghothaman, S.A. Seshia, R. Singh, A. Solar-Lezama, E. Torlak, A. Udupa, Syntax-guided synthesis, in: 2013 Formal Methods in Computer-Aided Design, 2013, pp. 1–8, doi:10.1109/FMCAD.2013.6679385.
[29] A. Kantchelian, M.C. Tschantz, L. Huang, P.L. Bartlett, A.D. Joseph, J.D. Tygar, Large-margin convex polytope machine, in: Proceedings of the 27th International Conference on Neural Information Processing Systems - Volume 2, in: NIPS'14, MIT Press, Cambridge, MA, USA, 2014, pp. 3248–3256.
[30] A. Galassi, M. Lombardi, P. Mello, M. Milano, Model agnostic solution of csps via deep learning: A preliminary study, in: W.-J. van Hoeve (Ed.), Integration of Constraint Programming, Artificial Intelligence, and Operations Research, Springer International Publishing, Cham, 2018, pp. 254–262.
[31] L.D. Raedt, A. Passerini, S. Teso, Learning constraints from examples, in: AAAI, 2018, pp. 7965–7970.
[32] B. Mayoh, E. Tyugu, J. Penjam, Constraint programming, Nato ASI Subseries F:, Springer, 2013.
[33] C. Barrett, R. Sebastiani, S. Seshia, C. Tinelli, Satisfiability modulo theories, in: Frontiers in Artificial Intelligence and Applications, vol. 185, 1, IOS Press, 2009, pp. 825–885.
[34] T.P. Pawlak, K. Krawiec, Synthesis of constraints for mathematical programming with one-class genetic programming, IEEE Trans. Evol. Comput. 23 (1) (2019) 117–129, doi:10.1109/TEVC.2018.2835565.
[35] D.J. Montana, Strongly typed genetic programming, Evol Comput 3 (2) (1995) 199–230, doi:10.1162/evco.1995.3.2.199.
[36] T.P. Pawlak, Performance improvements for evolutionary strategy-based one-class constraint synthesis, in: Proceedings of the Genetic and Evolutionary Computation Conference, in: GECCO '18, ACM, New York, NY, USA, 2018, pp. 873–880, doi:10.1145/3205455.3205504.
[37] H.-G. Beyer, H.-P. Schwefel, Evolution strategies–a comprehensive introduction, Nat Comput 1 (1) (2002) 3–52, doi:10.1023/A:1015059928466.
[38] N. Hansen, The CMA Evolution Strategy: A Comparing Review, Springer Berlin Heidelberg, Berlin, Heidelberg, 2006, pp. 75–102.
[39] A.P. Dempster, N.M. Laird, D.B. Rubin, Maximum likelihood from incomplete data via the EM algorithm, Journal of the Royal Statistical Society. Series B (Methodological) 39 (1) (1977) 1–38.
[40] J.R. Quinlan, C4.5: Programs for machine learning, Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1993.
[41] D. Pelleg, A. Moore, X-means: Extending k-means with efficient estimation of the number of clusters, in: In Proceedings of the 17th International Conf. on Machine Learning, Morgan Kaufmann, 2000, pp. 727–734.
[42] S. Luke, Metaheuristics, 1st, lulu.com, 2009.
[43] T.P. Pawlak, B. Litwiniuk, Ellipsoidal one-class constraint acquisition for quadratically constrained programming, Eur J Oper Res 293 (1) (2021) 36–49, doi:10.1016/j.ejor.2020.12.018.
[44] C.M. Bishop, Pattern Recognition and Machine Learning, Springer-Verlag, Berlin, Heidelberg, 2006.
[45] S. Haykin, Neural networks: A comprehensive foundation, 2nd, Prentice Hall PTR, Upper Saddle River, NJ, USA, 1998.
[46] S. Kolb, S. Teso, A. Passerini, L.D. Raedt, Learning SMT(LRA) constraints using SMT solvers, in: Proceedings of the Twenty-Seventh International Joint Conference on Artificial Intelligence, IJCAI-18, International Joint Conferences on Artificial Intelligence Organization, 2018, pp. 2333–2340, doi:10.24963/ijcai.2018/323.
[47] N. Beldiceanu, H. Simonis, A model seeker: Extracting global constraint models from positive examples, in: CP'12, volume LNCS 7514, Springer, Quebec City, Canada, 2012, pp. 141–157, doi:10.1007/978-3-642-33558-7_13.
[48] C. Bessiere, R. Coletta, F. Koriche, B. O'Sullivan, A SAT-Based Version Space Algorithm for Acquiring Constraint Satisfaction Problems, Springer Berlin Heidelberg, 2005, pp. 23–34.
[49] T.M. Mitchell, Generalization as search, Artif Intell 18 (2) (1982) 203–226, doi:10.1016/0004-3702(82)90040-6.
[50] C. Bessiere, R. Coletta, B. O'Sullivan, M. Paulin, Query-driven constraint acquisition, in: IJCAI 2007, 2007, pp. 50–55.
[51] K.M. Shchekotykhin, G. Friedrich, Argumentation based constraint acquisition, in: ICDM 2009, The Ninth IEEE International Conference on Data Mining, Miami, Florida, USA, 6–9 December 2009, 2009, pp. 476–482, doi:10.1109/ICDM.2009.62.
[52] C. Bessiere, R. Coletta, E. Hebrard, G. Katsirelos, N. Lazaar, N. Narodytska, C.-G. Quimper, T. Walsh, Constraint acquisition via partial queries, in: IJCAI 2013, 2013, pp. 475–481.
[53] S. Teso, R. Sebastiani, A. Passerini, Structured learning modulo theories, Artif Intell 244 (2017) 166–187, doi:10.1016/j.artint.2015.04.002.
[54] S. Kolb, Learning constraints and optimization criteria, AAAI Workshops, 2016.
[55] M. Fenton, J. McDermott, D. Fagan, S. Forstenlechner, E. Hemberg, M. O'Neill, PonyGE2: grammatical evolution in python, in: Proceedings of the Genetic and Evolutionary Computation Conference Companion, in: GECCO '17, ACM, Berlin, Germany, 2017, pp. 1194–1201, doi:10.1145/3067695.3082469. http://doi.acm.org/10.1145/3067695.3082469.
[56] L. Manzoni, A. Bartoli, M. Castelli, I. Gonçalves, E. Medvet, Specializing context-free grammars with a (1+1)-EA, IEEE Trans. Evol. Comput. (2020), doi:10.1109/TEVC.2020.2983664.
[57] D. Fagan, M. Fenton, M. O'Neill, Exploring position independent initialisation in grammatical evolution, in: Y.-S. Ong (Ed.), Proceedings of 2016 IEEE Congress on Evolutionary Computation (CEC 2016), IEEE Press, Vancouver, 2016, pp. 5060–5067, doi:10.1109/CEC.2016.7748331.
[58] A.E. Eiben, J.E. Smith, Introduction to evolutionary computing, Springer, 2003.
[59] J.R. Koza, Genetic programming: On the programming of computers by means of natural selection, MIT Press, Cambridge, MA, USA, 1992.
[60] Gurobi Optimization, LLC, Functional code examples, Accessed 2020-05-26, http://gurobi.com/resource/functional-code-examples.
[61] M. Conforti, G. Cornuejols, G. Zambelli, Integer programming, Springer Publishing Company, Incorporated, 2014.
[62] T. Polzin, Algorithms for the Steiner problem in networks, Saarland University, Saarbrücken, Germany, 2003 Ph.D. thesis.

[63] A. Schrijver, Combinatorial optimization - Polyhedra and efficiency, Springer, 2003.
[64] J. Bell, B. Stevens, A survey of known results and research areas for n-queens, Discrete Math 309 (1) (2009) 1–31, doi:10.1016/j.disc.2007.12.043.
[65] PassMark Software, cpubenchmark.net, Accessed: 2020-06-08, https://www.cpubenchmark.net/compare/Intel-i5-8500-vs-Intel-i7-4790-vs-Intel-i7-4770/3223vs2226vs1907.
[66] M. Sipper, W. Fu, K. Ahuja, J.H. Moore, Investigating the parameter space of evolutionary algorithms, BioData Min 11 (2) (2018) 1–14, doi:10.1186/s13040-018-0164-x.
[67] Parameter setting in evolutionary algorithms, in: F. Lobo, C.F. Lima, Z. Michalewicz (Eds.), Studies in Computational Intelligence, Springer, 2007, doi:10.1007/978-3-540-69432-8.
[68] G. Kanji, 100 Statistical tests, SAGE Publications, 1999.
[69] M. Fenton, D. Lynch, D. Fagan, S. Kucera, H. Claussen, M. O'Neill, Towards automation & augmentation of the design of schedulers for cellular communications networks, Evol Comput 27 (2) (2019), doi:10.1162/evco_a_00221. Forthcoming
[70] S. Silva, L. Vanneschi, A.I. Cabral, M.J. Vasconcelos, A semi-supervised genetic programming method for dealing with noisy labels and hidden overfitting, Swarm Evol Comput 39 (2018) 323–338, doi:10.1016/j.swevo.2017.11.003.
[71] P. Rakshit, Improved differential evolution for noisy optimization, Swarm Evol Comput 52 (2020) 100628, doi:10.1016/j.swevo.2019.100628.
[72] P. Rakshit, A. Konar, S. Das, Noisy evolutionary optimization algorithms – a comprehensive survey, Swarm Evol Comput 33 (2017) 18–45, doi:10.1016/j.swevo.2016.09.002.