Automatic Error Function Learning with Interpretable Compositional Networks

Florian Richoux, Jean-François Baffier

Automatic Error Function Learning with Interpretable Compositional Networks

Florian Richoux[1,3*] and Jean-François Baffier[2,3]

[1*]AIST, Tokyo, Japan.
[2]IIJ Research Lab, Tokyo, Japan.
[3]JFLI, CNRS, Tokyo, Japan.

*Corresponding author(s). E-mail(s): florian@richoux.fr;
Contributing authors: jf@baffier.fr;

Abstract

In Constraint Programming, constraints are usually represented as predicates allowing or forbidding combinations of values. However, some algorithms can exploit a finer representation: error functions. By associating a function to each constraint type to evaluate the quality of an assignment, it extends the expressiveness of regular Constraint Satisfaction Problem/Constrained Optimization Problem formalisms. Their usage comes with a price though: it makes problem modeling significantly harder, since users must provide a set of error functions that are not always easy to define. Here, we propose a method to automatically learn an error function corresponding to a constraint, given its predicate version only. This is, to the best of our knowledge, the first attempt to automatically learn error functions for hard constraints. In this paper, we also give for the first time a formal definition of combinatorial problems with hard constraints represented by error functions. Our method aims to learn error functions in a supervised fashion, trying to reproduce either the Hamming or the Manhattan distance, by using a graph model we named Interpretable Compositional Networks. This model allows us to get interpretable results. We run experiments on 7 different constraints to show its versatility. Experiments show that our system can learn functions that scale to high dimensions, and can learn fairly good functions over incomplete spaces. We also show that learned error functions can be used efficiently to represent constraints in different classic problems.

Keywords: Combinatorial Satisfaction, Combinatorial Optimization, Constraint Programming, Problem Modeling, Error Function, Interpretable Learning

1 Introduction

Twenty years separate Freuder's papers [1] and [2], both about the grand challenges Constraint Programming (CP) must tackle *"to be pioneer of a new usability science and to go on to engineering usability"* [3].

To respond to the lack of a "Model and Run" approach in CP [4, 5], several languages have been developed since the late 2000's, such as ESSENCE [6], XCSP [7] and MiniZinc [8]. However, they require users to have deep expertise on global constraints and to know how well these constraints, and their associated mechanisms such as propagators, are suiting the solver. We are still far from the original Holy Grail of CP: *"the user states the problem, the computer solves it"* [1].

This paper makes a contribution in automatic CP problem modeling. We focus on Error Function-based Constraint Satisfaction and Optimization Problems we define in the next section. Compared to classical Constraint Satisfaction and Constrained Optimization Problems, they rely on a finer structure about the problem: the cost functions network, which is an ordered structure over invalid assignments (in our case) that some solvers, such as constraint-based local search solvers, can exploit efficiently to improve the search.

In this paper, we propose a method to learn error functions automatically, taking as input a user-provided Constraint Satisfaction / Constrained Optimization Problem model, and outputing a corresponding Error Function-based Constraint Satisfaction / Optimization Problem model. This is a direction that, to the best of our knowledge, had not been explored in Constraint Programming.

The motivation of this work starts with this analysis: from a runtime and scalability point of view, some solvers can greatly benefit from working on a problem modeled within the Error Function-based Constraint Satisfaction / Optimization Problem formalisms, rather than the classical ones. However, while defining constraints as predicates is fairly intuitive for users, finding good constraint representations through error functions can be more subtle. We believe that the strong points of Constraint Programming lies in the simplicity for users to model problems, as well as a strong decoupling between the modelization and the resolution of a problem. That motivated this work on a method to convert, in a transparent way for users, a regular Constraint Satisfaction / Constrained Optimization Problem model into an equivalent error function-based model.

2 Error Function-based Constraint Satisfaction and Optimization Problems

Constraint Satisfaction Problems (CSPs) and Constrained Optimization Problems (COPs) are constraint-based problems defined upon a classical hard

constraint network, where constraints can be seen as predicates allowing or forbidding some combinations of variable assignments [9] .

Likewise, Error Function-based Constraint Satisfaction Problems (EF-CSPs) and Error Function-based Constrainted Optimization Problems (EF-COPs) are constraint-based problems defined upon a specific hard constraint network named cost function network [10] or semi-ring constraint network [11]. Both networks are equivalent for the purpose of this paper: cost function networks exactly correspond to semi-ring constraint networks with a totally ordered cost structure [10].

Constraints are then represented by cost functions $f : D_1 \times D_2 \times \ldots \times D_n \to E$, where D_i is the domain of the i-th variable in the constraint scope, n the number of variables (*i.e.*, the size of this scope) and E the set of possible costs.

A cost function network is a quadruplet $\langle V, D, F, S \rangle$ where V is a set of variables, D the set of domains for each variable, *i.e.*, the sets of values each variable can take, F the set of cost functions and S a cost structure. A cost structure is also a quadruplet $S = \langle E, \oplus, \bot, \top \rangle$ where E is the totally ordered set of possible costs, $\oplus$ a commutative, associative, and monotone aggregation operator and $\bot$ and $\top$ are the neutral and absorbing elements of $\oplus$, respectively.

In Constraint Programming, cost functions are often associated to soft constraints: they can be interpreted as preferences over valid or acceptable assignments. However, this is not necessarily the case: it actually depends on the cost structure. For instance, the classical cost structure

$$S_{t/f} = \langle \{true, false\}, \wedge, true, false \rangle$$

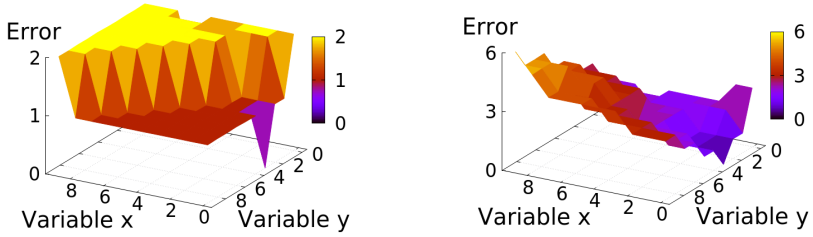
make the cost function network equivalent to a classical constraint network, so dealing with hard constraints.

Here, we consider particular cost functions that also represent hard constraints only, by considering the additive cost structure $S_+ = \langle \mathbb{R} \cup \{\infty\}, +, 0, \infty \rangle$. The additive cost structure produces useful cost function networks capturing problems such as Maximum Probability Explanation (MPE) in Bayesian networks and Maximum A Posteriori (MAP) problems in Markov random fields [12].

We name **error function** a cost function defined in a cost function network with the additive cost structure S_+. Intuitively, error functions are preferences over *invalid* assignments. Let f_c be an error function representing a constraint c and $\vec{x}_c$ an assignment of variables in the scope of c. Then $f_c(\vec{x}_c) = 0$ iff $\vec{x}_c$ satisfies the constraint c. For all invalid assignments $\vec{i}_c$, $f_c(\vec{i}_c) > 0$ holds such that the closer $f_c(\vec{i}_c)$ is to 0, the closer $\vec{i}_c$ is to satisfy c.

The goal of this paper is not to study the advantages of such cost function networks over regular constraint networks. Some constraint-based local search methods such as Adaptive Search exploit this structure efficiently and

show state-of-the-art experimental results, both in sequential [13] and parallel solving [14]. Such question would deserve a deep investigation which is out of the scope of this paper. However, we can give a short illustration of the advantage of cost function networks over regular constraint networks. Figure 1 shows the search landscapes of the same constraint network from a regular constraint network (CSP landscape, Figure 1a) and cost function network (EF-CSP landscape, Figure 1b) point of view. The network is composed of the constraints AllDifferent(x, y), $x \leq y$ and $x + 2y = 6$. Error functions used for Figure 1b have been learned with our system. We can see that the CSP landscape is mostly composed of large plateaus with an error measure (the number of violated constraints) between 0 and 2. On the other hand, the EF-CSP landscape is more in relief with slopes toward the solution, with a broader scope of error values, between 0 and 6, allowing richer comparisons of variable assignments.



(a) CSP landscape (b) EF-CSP landscape

Fig. 1: Search landscapes of a small constraint network.

The term "error function" has been used in the Constraint Programming literature in the same sense as in this paper. Borning et al. [15] are the first, to the best of our knowledge, to use this term. It also appears in the constraint-based local search literature, like in Codognet et al. [13] describing the local search algorithm Adaptive Search. We can also find the equivalent term "penalty function" [16] for local search algorithms in Constraint Programming. However, penalty function can also refer to functions representing soft constraints in Mathematical Programming [17]. Therefore, to avoid confusions with cost functions or penalty functions for soft constraints, since our study deals with hard constraints only, we opted for the name "error function".

Let $\vec{x}$ be a variable assignment, and denote by $\vec{x}_c$ the projection of $\vec{x}$ over variables in the scope of a constraint c. We can now define the Error Function-based Constraint Satisfaction and the Error Function-based Constrainted Optimization Problems.

Problem: Error Function-based Constraint Satisfaction Problem
Input: A cost function network $\langle V, D, F, S_+ \rangle$.
Question: Does a variable assignment $\vec{x}$ exist such that $\forall f_c \in F,\ f_c(\vec{x}_c) = 0$ holds?

Problem: Error Function-based Constrained Optimization Problem
Input: A cost function network $\langle V, D, F, S_+ \rangle$ and an objective function o.
Question: Find a variable assignment $\vec{x}$ maximizing or minimizing the value of $o(\vec{x})$ such that $\forall f_c \in F,\ f_c(\vec{x}_c) = 0$ holds.

Thanks to their constraint structure, problems modeled by an EF-CSP or an EF-COP can be solved by some solvers faster than if they were modeled by a CSP or a COP, as shown by results of Experiment 3 in Section 6.2.2. Another way to consider it: with the same computation budget, a solver could solve larger EF-CSP or EF-COP problems. However, we do not obtain this gain for free: this is a trade-off with modeling simplicity. Indeed, it is not always easy to find good error functions to describe constraints.

To have a better grasp of problems modeled as EF-CSP or EF-COP, let's consider the example of the Magic Square 3×3 problem. Magic Square is a $n \times n$ grid that must be filled up with all numbers from 1 to n^2 (thus, all numbers must appear exactly once in the grid), such that the sum of each row, each column, and the two diagonals must be equal to the same constant p.

If we want to model the Magic Square 3×3 problem as a CSP, we have to declare its constraint network $\langle V, D, C \rangle$, with V and D the sets of variables and domains, like introduced for cost function networks, and C a set of constraints. We need two constraints only to model this problem: AllDifferent, which enforces all variables to be pairwise different, and LinearSum, representing a linear equation where the sum of variables must be equals to a constant parameter.

Constraint network for Magic Square 3×3

Variables V	$\{v_1, \ldots, v_9\}$, one variable for each cell in the grid
Domains D	$\{1, \ldots, 9\}$ for each variable in V
Constraints C	$\text{AllDiff}(v_1, \ldots, v_9)$ $\text{LinearSum}(v_{3i+1}, v_{3i+2}, v_{3i+3}, 15)$, with $0 \leq i \leq 2$ (rows) $\text{LinearSum}(v_{1+i}, v_{4+i}, v_{7+i}, 15)$, with $0 \leq i \leq 2$ (columns) $\text{LinearSum}(v_1, v_5, v_9, 15)$ (1^{st} diagonal) $\text{LinearSum}(v_3, v_5, v_7, 15)$ (2^{nd} diagonal)

The predicates representing the constraints can be expressed as follows:

$$\text{AllDiff}(v_1, \ldots, v_n) \text{ is true} \Leftrightarrow \forall i, j,\ (i \neq j) \Rightarrow (v_i \neq v_j)$$
$$\text{LinearSum}(v_1, \ldots, v_n, p) \text{ is true} \Leftrightarrow v_1 + \ldots + v_n = p$$

The corresponding cost function networks to solve this problem as an EF-CSP is the following one.

Cost function network for Magic Square 3×3

Variables V	$\{v_1, \ldots, v_9\}$, one variable for each cell in the grid
Domains D	$\{1, \ldots, 9\}$ for each variable in V
Error functions F	$f_{\text{AllDiff}}(v_1, \ldots, v_9)$ $f_{\text{LinearSum}}(v_{3i+1}, v_{3i+2}, v_{3i+3}, 15)$, with $0 \leq i \leq 2$ (rows) $f_{\text{LinearSum}}(v_{1+i}, v_{4+i}, v_{7+i}, 15)$, with $0 \leq i \leq 2$ (columns) $f_{\text{LinearSum}}(v_1, v_5, v_9, 15)$ (1^{st} diagonal) $f_{\text{LinearSum}}(v_3, v_5, v_7, 15)$ (2^{nd} diagonal)
Cost structure	S_+

We can define the error functions above as follows:

$$f_{\text{AllDiff}}(v_1, \ldots, v_n) = \sum_{j=1}^{n} \#\{v_i \mid i < j \ \wedge \ v_i = v_j\}$$

$$f_{\text{LinearSum}}(v_1, \ldots, v_n, p) = |v_1 + \ldots + v_n - p|$$

This paper focuses on what Freuder calls the "ease of use" aspect of Constraint Programming [3]. It proposes a way to automatically learn error functions. Users provide the usual constraint network $\langle V, D, C \rangle$, and our systems computes the equivalent cost function networks $\langle V, D, F, S_+ \rangle$. Learned functions composing the set F are independent of the number of variables in constraints scope, and are expressed in an interpretable way: users can understand these functions and easily modify them at will. This way, users can have the powerness of EF-CSPs and EF-COPs with the same modeling effort as for CSPs and COPs.

3 Related works

This work belongs to one of the three directions for Constraint Programming identified by Freuder [3]: *Automation*, *i.e.*, "automating efficient and effective modeling and solving". To the best of our knowledge, few efforts have been done on the modeling side.

There is more research done in this direction in other related research fields, such as Mathematical Programming. Paulus et al [18] proposes an interesting paper where a combinatorial optimization module is directly integrated into a neural network as a layer, learning both the constraints and their costs from data. The main difference with our work is that their method is learning linear constraints only, with a fixed number of variables, when we can deal with arbitrary constraint representations that are independent of the size of their scope.

Another interesting study is proposed by Kumar et al [19]. In their paper, the authors learn from data the constraints and the objective function of

Mixed-Integer Linear Programs. Like for [18], they learn linear constraints only over a fixed number of variables, when we learn error functions of arbitrary constraints independent of the number of variables. A strong limitation of the method in their paper is that each model candidate obtained during the exploration must be solved, to find a (quasi-) optimal global solution for matching it with training examples. Our loss function in this paper does not have this issue because it is independent of any global solutions, since we learn error functions contraint by constraint, so we only have to consider local solutions of each constraint individually, which are greatly easier to find.

A second direction for Constraint Programming described by Freuder [3], and more loosely related to our work, is *Acquisition*, for "acquiring a complete and correct representation of real problems". Remarkable efforts on this topic have been done by Bessiere's research team, for instance with constraints learning by induction from positive and negative examples [20], with interactive queries asked to users [21], and with constraint network learning also through with interactive queries [22].

Model Seeker from Beldiceanu and Simonis [23] is a passive learning system taking positive examples only, which are certainly easier for users to provide. It transforms examples into data adapted to the Global Constraint Catalog [24], then generate structured candidates by grouping variables into regular subsets. For instance, with an array of n values, Model Seeker can reshape this array into $a \times b$ matrix candidates where a and b are divisors of n. Model Seeker then simplifies candidates by eliminating dominated ones, and calls the tool Constraint Seeker [25] to get the global constraint fitting the best the candidates. Model Seeker is particularly efficient to find the model of problems with a regular inner structure that can be expressed by the repetition of short constraints. However, this tool cannot learn models that are independent of the number of variables in the examples, unlike our method.

Teso [26] gives a good survey on Constraint Learning with this interesting remark: "A major bottleneck of [constraint-based problem modeling] is that obtaining a formal constraint theory is non-obvious: designing an appropriate, working constraint satisfaction or optimization problem requires both domain and modeling expertise. For this reason, in many cases a modeling expert is hired and has to interact with domain expert to acquire informal requirements and turn them into a valid constraint theory. This process can be expensive and time-consuming."

We can consider that Constraint Acquisition, or Constraint Learning, focuses on modeling expertise and puts domain expertise on background: users would not be able to understand and modify a learned model without the help of a modeling expert. The goal of these systems is mainly to simplify the interaction between the domain and the modeling experts.

Our work is taking the opposite direction: we focus on domain expertise and put modeling expertise on background. With our system, users always have the control over constraints' representation, which can be modified at

will to fit needs related to their domain expertise. *Constraint Implementation Learning* is what best describes this research topic.

In a recent article, Teso et al. [27] make a survey of four different approaches for the completion of partially-specified problems via supervised learning. One approach, Structured-Output Prediction, is about predicting an output structure from an input (often structured itself). This is somehow connected to our method; the main difference being that we are not trying to predict an output structure from the input, but to find a structured function returning an expected value regarding the input.

A survey from Deshwal et al. [28] gives an overview on some Structured-Output Prediction works which are close to our method such as *cost function learning approaches*. However, these methods look to learn linear cost functions to evaluate a structure inference regarding the training data. Learning these linear functions boils down to learning their variable coefficients, like the works of Kumar et al [19] and Paulus et al [18] previously cited in this section. Our method aims not to learn coefficients but the right set of elementary operations composing an error function, which may lead to non-linear combinations. More importantly, these Structured-Output Prediction methods need to call an inference solver to compute an output prediction. Such calls are costly from a runtime point of view, and are repeated all along the learning process. Our method does also call a solver, but only once and with a very aggressive runtime (200ms). Actually, the solver call *is* the whole learning process in our work, as explained in Section 4.3.

Finally, we can mentioned the work from Domshlak et al [29], in which they learn the weights of Soft Constraint Satisfaction Problems modeled upon semi-ring constraint networks. Their work diverge in several points with our, since they deal with computin weights of a whole soft constraints networks, given a Conditional Preference-net as input, when we aim to learn from data an interpretable expression of hard constraints individually, and their method is size-dependent unlike our.

4 Method design

The main contribution of this paper is to propose a method to automatically learn from data an error function representing a constraint, to ease the modeling of EF-CSP/EF-COP.

We are tackling a symbolic regression problem since the goal is looking for the model of a function in a space of mathematical representation. Such problems are often handled by Genetic Programming methods, but we explain in Section 4.3 why we do not use Genetic Programming to learn error functions in this work.

Our method can be summed up as follows: it learns from data the parameter θ of the model we propose in this paper, model called **Interpretable Compositional Network** (ICN). An ICN_θ parameterized by θ represents an error function. In this paper, we consider error functions to be a (potentially

non-linear) combination of elementary operations such as max, $\sum$, etc (see Appendix A for a complete list). Here, θ can be seen as a vector of Boolean values indicating which elementary operations will be part of the composition. How elementary operations are composed is actually fixed by the architecture of the ICN; thus the only thing to decide is what elementary operations will be part of the error function.

In a nutshell, our method follows this workflow:

1. Users provide a regular constraint programming model in the CSP or COP formalism, *i.e.*, a constraint network.
2. The goal of our method is to convert this constraint network into a cost function network, expressing constraints as error functions. Error functions are learned successively: we consider constraints one by one, and we learn an error function for a given constraint independently of the other ones. Thus, for each constraint in the given model, we generate a training set. This set contains assignments of variables in the scope of the constraint (usually, all possible assignments), together with a Boolean for each assignment telling if it satisfies or not the constraint.
3. We learn the parameter θ of ICN_θ in a supervised fashion to find the error function fitting the best the generated training set. In Section 4.3, we show that learning θ can also be seen as solving a combinatorial optimization problem.

In this paper, we denote by **method** the methodology we propose to learn error functions, and by **system** the implementation of our method.

Before diving into the description of our model and our method, and before introducing some essential notions in the next section, it is important to stress two points:

1) While converting a constraint network into a cost function network, it is not necessary to find an error function for each constraint instance of the constraint network: Finding one error function for each constraint type is sufficient, since error functions we learn are independent of the number of variables and the size of their domain.

Thus, with the example of the Magic Square 3×3 problem given in Section 2, even if the constraint network contains 9 constraint instances (1 AllDifferent and 8 LinearSum), we only need to run our method twice to learn an error function for AllDifferent, and an error function for LinearSum.

2) Neither in our method nor our system, we apply a mechanism forcing learned error functions not to approximate their target constraint, *i.e.*, outputing 0 on candidates that do not satisfy the constraint, or a value strictly higher than 0 on solutions.

However in practice, we never observe this behavior. Moreover, we could very easily get around this problem by always considering the given concept on top of our error functions, then forcing the value 0 on solutions, and adding the value 1 to the output error on non-solution candidates. This simple mechanism could be added to prevent learn error function from approximating their target constraint.

4.1 Definitions

We propose a method to automatically learn an error function from the *concept* of a constraint. As described in Bessiere et al. [30], the **concept** of a constraint is a Boolean function that, given an assignment $\vec{x}$, outputs *true* if $\vec{x}$ satisfies the constraint, and *false* otherwise. Concepts are the predicate representation of constraints mentioned at the beginning of Section 2.

Our method learns error functions in a supervised fashion, searching for a function computing either the *Hamming cost* or the *Manhattan cost* of each assignment. The **Hamming cost** of an assignment $\vec{x}$ is the minimum number of variables in $\vec{x}$ to reassign to get a **solution**, *i.e.*, a variable assignment satisfying the considered constraint. In other words, it is the Hamming distance from $\vec{x}$ to its closest solution. If $\vec{x}$ is a solution, then its Hamming cost is 0. The **Manhattan cost** of an assignment $\vec{x}$ is the minimum number of value incrementations or decrementations in $\vec{x}$ to perform to get a solution. It corresponds to the Manhattan distance from $\vec{x}$ to its closest solution. As for the Hamming cost, if $\vec{x}$ is a solution, its Manhattan cost is 0.

Given the number of variables of a constraint and their domain, the **constraint assignment space** is the set of couples $(\vec{x}, b)$ where $\vec{x}$ is an assignment and b the Boolean output of the concept applied on $\vec{x}$. Such constraint assignment spaces can be generated from concepts. These spaces are said to be **complete** if and only if they contain all possible assignments, *i.e.*, all combinations of possible values of variables in the scope of the constraint, given their domain. Otherwise, spaces are said to be **incomplete**.

A constraint assignment space is a training set as mentioned at the beginning of Section 4. Complete spaces are intuitively good training sets since it is easy to compute the exact Hamming and Manhattan costs of their elements. We also consider assignments from incomplete spaces where their Hamming and Manhattan costs have been approximated regarding a subset of solutions in the constraint assignment space.

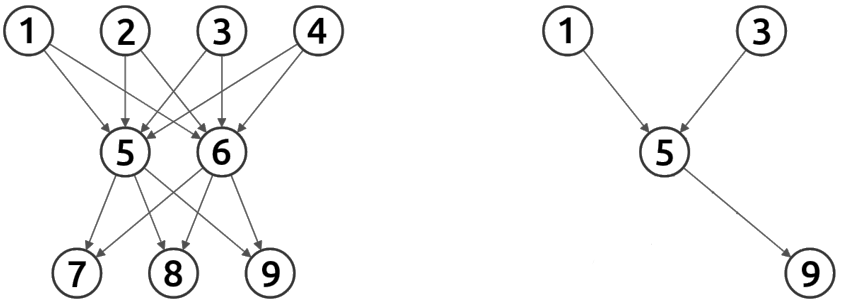
4.2 Interpretable Compositional Network

One of the main contributions of this work is the design of a new model we called Interpretable Compositional Network (ICN).

An ICN is a directed acyclic multipartite graph composed of k independent sets organised in such a way that vertices of an independent set are exclusively connected to all vertices of a unique independent set in the graph. Thus, if we abstract each independent set by a vertex, then we obtain a directed line.

ICNs are networks in the graph theory way, *i.e.*, a graph where vertices or arcs have attributes. In an ICN, vertices are actually elementary operations, and arcs represent the composition of two elementary operations. An ICN_θ parameterized by θ is an ICN where we only consider vertices selected by θ (or enabled by θ), as well as all arcs between those vertices. Figure 2 illustrates the example of a short ICN on the left (Figure 2a) with 9 vertices, and the equivalent ICN_θ on the right (Figure 2b) parameterized by θ selecting vertices

1, 3, 5 and 9. These parameter θ can be represented by the Boolean vector $\{1,0,1,0,1,0,0,0,1\}$ where the 1^{st}, 3^{rd}, 5^{th} and 9^{th} dimension are set to 1, and all others to 0.



(a) A simple ICN

(b) The same ICN_θ parameterized by $\theta = \{1,0,1,0,1,0,0,0,1\}$

Fig. 2: Examples of ICN and ICN_θ

In its shape, an ICN looks like a neural network: the hierarchical organisation between independent sets exactly corresponds to the hierarchical structure of layers of neurons. Due to this similarity, we call *layers* the independent sets within an ICN. However, the similarity with neural networks stops here: the main difference between ICNs and neural networks is that there are no activation functions within ICNs (the elementary operation of a vertex cannot be considered to be an activation function). A second main difference is that ICNs do not have weights on arcs connecting two vertices. We first introduced ICNs in a poster paper as a variant of neural networks [31]. Although ICN is deeply inspired from a variant of neural networks, the correct way to interpret our model is to see it as a particular directed acyclic graph.

In this paper, our ICN model is composed of four layers, each of them having a specific purpose and composed of vertices with a unique operation for each.

Figure 3 is a schematic representation of our network. It takes as input an assignment of n variables, *i.e.*, a vector of n integers. The first layer, called **transformation layer**, is composed of 18 transformation operations, each of them applied element-wise on each element of the input vector. This element-wise computation property is what allows our model to be able to compute the error function of a constraint of an arbitrary size, *i.e.*, with an arbitrary number of variables in the constraint scope. This is a very powerful feature that enables learning error functions over a small constraint assignment spaces that scale to larger spaces.

Such transformation operations are for instance the maximum between the i-th and $(i+1)$-th elements of the input vector, or the number of j-th elements

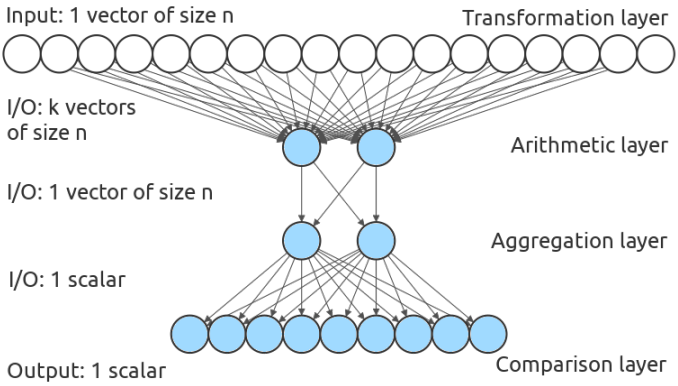


Fig. 3: Our 4-layer network. Layers with blue vertices have mutually exclusive operations.

of the vector smaller than the i-th element such that $j > i$ holds. This layer is composed of both linear and non-linear operations. If an operation is selected, it outputs a vector of n integers.

Example 1 Consider one of our 18 transformation operations: "*Number of* $x[j]$ *such that* $j < i$ *and* $x[i] = x[j]$," with $x[i]$ and $x[j]$ respectively the i-th and j-th value of the assignment $\vec{x}$. Giving the assignment $(3, 1, 3, 4, 3, 1, 2)$ as input, this transformation operation outputs the vector $(0, 0, 1, 0, 2, 1, 0)$.

If k transformation operations are selected, then the next layer gets k vectors of n integers as input. This layer is the **arithmetic layer**. Its goal is to apply a simple arithmetic operation in a component-wise fashion on all i-th element of our k vectors to get one vector of n integers at the end, combining previous transformations into a unique vector. We have considered only 2 arithmetic operations: the addition and the multiplication.

Example 2 Consider the addition as the arithmetic operation, and as inputs the two vectors $(0, 0, 1, 0, 2, 1, 0)$ and $(2, 0, 1, 0, 2, 0, 0)$. Then the arithmetic layer outputs the vector $(2, 0, 2, 0, 4, 1, 0)$.

The output of the arithmetic layer is given to the **aggregation layer**. This layer crunches the whole vector into a unique integer. At the moment, the aggregation layer is composed of 2 operations: *Sum* computing the sum of input values and $Count_{>0}$ counting the number of input values strictly greater than 0.

Example 3 Consider the aggregation operation $Count_{>0}$ applied on $(2, 0, 2, 0, 4, 1, 0)$. Then, the aggregation layer outputs 4, since 4 values in the input are strictly greater than 0.

Finally, the computed scalar is transmitted to the **comparison layer** with 9 operations. Examples of these operations are the identity, or the absolute value of the input minus a given parameter. Except with the identity, this layer compares its input with an external parameter value, or the number of variables of the problem, or the domain size, among others.

Example 4 Consider the comparison operation $\max(0\ ,\ input - parameter)$. Assume that we have the parameter $p = 1$ and the value 4 as input. The comparison layer outputs 3.

All elementary operations in our model are generic: we do not choose them to fit one or several particular constraints. The complete list of elementary operations in our ICN model is given in Appendix A.

Although an in-depth study of the elementary operations properties would be interesting, this is out of the scope of this paper: its goal is to show that learning interpretable error functions via a generic ICN is possible. There is no reason to reduce ICN to its current 31 elementary operations or even a 4-layer architecture. Such elements can be changed by users to best fit their needs.

To have simple models of error functions, operations of the arithmetic, the aggregation, and the comparison layers are mutually exclusive, meaning that precisely one operation is selected for each of these layers. However, many operations from the transformation layer can be selected to compose the error function. This allows us to have a very comprehensible combination of elementary operations to model an error function, making it readable and intelligible by a human being. For instance, the most frequenly learned error function for AllDifferent is $Count_{>0}\big(\#\{x[j] \mid j < i \text{ and } x[j] = x[i]\}\big)$. It corresponds to the selection of the elementary operation $\#\{x[j] \mid j < i \ \wedge \ x[j] = x[i]\}$ (we named $Count^{l}_{=}$) in the transformation layer, followed by any elementary operation from the arithmetic layer (it does not change anything since we only have one operation from the transformation layer), then the $Count_{>0}$ elementary operation from the aggregation layer, and finally the identity in the comparison layer.

Example 5 Consider the three assignments (1, 2, 3, 4), (1, 2, 3, 2) and (1, 1, 2, 1).

The outputs of $Count^{l}_{=}$ on these three assignments are respectively (0, 0, 0, 0), (0, 0, 0, 1) and (0, 1, 0, 2).

Assume the sum operation has been selected in the arithmetic layer. Since we only have one elementary operation selected from the transformation layer, this layer directly outputs the vectors (0, 0, 0, 0), (0, 0, 0, 1) and (0, 1, 0, 2) it receives as input.

The $Count_{>0}$ operation from the aggregation layer outputs the scalar 0, 1 and 2, respectively.

Finally, the identity operation from the comparison layer does not change these values. These are the error of the three assignments above for the AllDifferent constraint, where (1, 2, 3, 4) only satisfies the constraint, and where one and two variables must be changed respectively in (1, 2, 3, 2) and (1, 1, 2, 1) to get a satisfying assignment.

Once the model of an error function is learned, users have the choice to run the network in a feed-forward fashion to compute the error function, or to re-implement it directly in a programming language. Users can use our system to find error functions automatically, but they can also use it as a decision support system to find promising error functions they can modify and adapt by hand.

4.3 Learning EF-CSP/EF-COP models through an EF-COP model

As written in the introduction of Section 4, our error function learning problem is a symbolic regression problem, a family of problems usually tackled through Genetic Programming.

The issue with Genetic Programming is that, in addition of having a huge search space of mathematical representations, we cannot guarantee learning error functions that are independent of the input size, *i.e.*, of the number of variables in the scope of the target constraint. This property is mandatory to have a unique function able to compute the error of a given constraint on 3, 30 or 300 variables. This property also allows us to quickly learn error functions over small constraint scopes in order to use them for computing the error of their constraints over large scopes.

What we do in this work is actually solving a *biased symbolic regression problem*, where the bias is induced by the structure of the ICN we provide. The architecture of an ICN gives a common shape to error functions and drastically reduces the search space of mathematical representations. More importantly, it enforces the input size independence property thanks to its transformation layer which is composed of size-independent elementary operations only. To guarantee this property with Genetic Programming, one would need to guide the function learning to make sure that the function inputs are decomposed in such a way that the length of its inputs does not matter. This would need tailored crossover and mutation operators, introducing a bias in the learning which is somewhat against the philosophy behind Genetic Programming, attempting to answer the question "how can computers be made to do what needs to be done, without being told exactly how to do it?" [32].

4.3.1 Hamming and Manhattan cost estimations

Given a constraint assignment space, our method aims to learn the parameter θ such that the ICN_θ model represents an error function as close as possible from the Hamming cost or the Manhattan cost, defined in 4.1.

If the constraint assignment space is complete, then the Hamming and Manhattan costs of each assignment can be pre-computed before learning θ. Otherwise, the incomplete constraint assignment space is composed of randomly drawn assignments. Then, we can only pre-compute an approximation of the costs of the sampled assignments by computing their Hamming and Manhattan distances regarding the few solutions that has been sampled.

If we consider to set Θ of all parameters θ, the goal is to find the parameter $\theta^* \in \Theta$ such that the ICN_{θ^*} model represents an error function computing exactly the Hamming cost or the Manhattan cost. However, finding this best parameter θ^* is difficult, and may be even impossible with our current ICN architecture. This is why we can only expect finding the best *empirical* parameter $\hat{\theta} \in \Theta$ leading to an error function that computes at best either the Hamming cost or the Manhattan cost on the given constraint assignment space.

The parameter $\hat{\theta}$, learned in a supervised fashion, is computed by the formula in Equation 1. It corresponds to a parameter θ minimizing the sum of the loss function (Equation 2) and a regularization (Equation 3).

$$\hat{\theta} = \underset{\theta \in \Theta}{\text{argmin}} \left(\text{loss}\left(\text{ICN}_\theta(\vec{x})\right) + \text{regularization}(\text{ICN}_\theta) \right) \tag{1}$$

The loss function described in Equation 2 is the minimum between the normalized difference of the value computed by ICN_θ over all assignments in the training set and their Hamming cost, and the normalized difference with their Manhattan cost.

$$\text{loss}\left(\text{ICN}_\theta(\vec{x})\right) = \min\left(\text{Hamming}\left(\text{ICN}_\theta(\vec{x})\right), \text{Manhattan}\left(\text{ICN}_\theta(\vec{x})\right)\right) \tag{2}$$

with

$$\text{Hamming}\left(\text{ICN}_\theta(\vec{x})\right) = \frac{\sum_{\vec{x} \in X} \mid \text{ICN}_\theta(\vec{x}) - \text{Hamming}(\vec{x}) \mid}{n}$$

$$\text{Manhattan}\left(\text{ICN}_\theta(\vec{x})\right) = \frac{\sum_{\vec{x} \in X} \mid \text{ICN}_\theta(\vec{x}) - \text{Manhattan}(\vec{x}) \mid}{n \times (\mid D \mid -1)}$$

where X is the constraint assignment space, ICN_θ $(\vec{x})$ the output of the ICN_θ model giving $\vec{x} \in X$ as an input, and $\text{Hamming}(\vec{x})$ and $\text{Manhattan}(\vec{x})$ respectively the pre-computed Hamming and Manhattan costs of $\vec{x}$ (that is approximated if X is incomplete). To make a fair comparison between the Hamming and Manhattan costs, we normalize these costs in $[0, 1]$ by dividing the difference of Hamming costs with the number n of variables in the scope of the constraint, and by dividing the difference of Manhattan costs with n times the difference between the maximal value and the minimal value in the domain of variables (corresponding to the cardinality of the domain minus 1 in our context).

The regularization described in Equation 3 outputs a value in $]0, 0.9]$ to favor short error functions, *i.e.*, an ICN_θ with a parameter θ selecting as few

elementary operations as possible. This regularization acts like a tie-breaker if two or more ICN_θ have to same loss on the training set.

$$\text{regularization}(\text{ICN}_\theta) = 0.9 \times \frac{\text{Number of selected vertices in ICN}_\theta}{\text{Total number of vertices in ICN}_\theta} \quad (3)$$

4.3.2 Learning $\hat{\theta}$ as an optimization problem

In our a first work, we used genetic algorithms to learn $\hat{\theta}$ via supervised learning [31]. In this paper, we present a more efficient and elegant way to learn them: we can actually model the problem of learning the error functions of an EF-CSP/EF-COP model as an EF-COP.

Indeed, learning an error function modeled by an ICN corresponds to selecting the right vertices in such a way that it both satisfies some conditions and minimizes the difference with the Hamming or Manhattan cost on the training set. Therefore, learning an error function is simultaneously a combinatorial optimization problem itself, and a symbolic regression problem we can tackled by supervised learning.

This combinatorial optimization problem can be modeled by the following EF-COP model:

Cost function network and objective function for learning $\hat{\theta}$

Variables V	One variable for each vertex in the ICN
Domains D	$\{0, 1\}$ for each variable, representing their selection
Error functions F	Mutual exclusion, No empty layer, Parameter-specific operations
Cost structure	S_+
Objective function	Minimizing the sum of the loss function (Equation 2) and the regularization (Equation 3)

The "Mutual exclusion" constraint is applied on the three last layers, *i.e.*, the arithmetic, the aggregation and the comparison layers. It ensures that exactly one vertex in each of these layers is selected. This simply corresponds to the linear equation $\sum_{v \in layer} v = 1$. We use $\mid 1 - \sum_{v \in layer} v \mid$ as the error function for this contraint.

The "No empty layer" constraint only concerns the first layer, where at least one vertex need to be selected. This can be represented by the linear inequation $\sum_{v \in layer} v \geq 1$ and can be represented by the error function $\max(0, 1 - \sum_{v \in layer} v)$.

Some elementary operations within our ICN involve the value of a parameter p in their computation. Indeed, some constraints need some parameters

to be defined, such as the linear equation with the constant at the right hand side of the equation. The "Parameter-specific operations" constraint enforces elementary operations to be part of the operation combination if and only if we are dealing with a constraint involving one or some parameters. This constraint is easily expressed as follows: considering that m is the number of selected parameter-specific elementary operations, if the target constraint contains some parameters, then we must have $m \geq 1$, otherwise $m = 0$ must hold. The error function for this constraint is then a combination of both previous error functions regarding if the target constraint contains at least one parameter or not.

Modeling and solving this EF-COP model has been done using the framework GHOST [33]. Notice that besides our training and tests sets, we do not have validation sets simply because we did not performed any parameter tuning: we use the solver of GHOST with its default parameter values for all experiments in this paper.

5 Attempts and failures

We strongly believe exposing explored directions, attempts, and reasons for their failure can be highly beneficial to the scientific community. Before showing our experimental results, we sum up in this short section our principal attempts and failures to represent error functions before coming up with a representation through Interpretable Compositional Networks.

5.1 Series of sinusoids

Let f be an error function we aim to learn. Our first idea was to represent error functions as a sum of p sinusoids, such that

$$f(\vec{x}) := \sum_{k=0}^{p} \left(a_k.cos(\vec{x}.2\pi.k) + b_k.sin(\vec{x}.2\pi.k) \right)$$

Thus, learning f boiled down to learning coefficients a_i, b_i.

Although any functions, even nonperiodic ones, can be represented by such a sum, error functions we want to learn might be too complicated and too high dimensional to be easily expressed by a reasonably small sum of sinusoids. Moreover, such a representation of f does not allow to express relations among variables in $\vec{x}$, unlike ICN. Furthermore, even if we could learn such a representation of f in a small dimensional space, it was unclear how to extend this function to higher dimensions.

We tried two approaches to learn a_i, b_i coefficients: multivariate interpolation and genetic algorithms.

Interpolation

We tried different tools for doing multivariate interpolation such as CHEBPOL [34] and SPLINTER [35], but it didn't lead to satisfying results, even

when we sampled 10% of the constraint assignment space, which is a huge part of an incomplete space.

Genetic Algorithm

We start with a population of 100 random individuals, where an individual is the vector of real values $(a_0, b_0, \ldots, a_p, b_p)$. Those coefficient are assigned to random values from a normal distribution of mean 0 and standard deviation $\sqrt{n}$, with n the number of variables of the target constraint. The fitness function was to maximize a parameter called *empirical correlation length*, multiplied by the mean of f on samples from a random walk such that $f(\vec{x}) \neq 0$. The idea was to get a function f with a low ruggedness (in other words, a smooth function) such that its value concerning non-solutions is high.

The ruggedness of a landscape can be computed by doing a random walk from a random assignment, and compute the empirical correlation length l (see Hoos and Stützle [36], Chapter 5). To do so, we need first to define the empirical autocorrelation function $r(i)$ for a given distance i.

$$r(i) := \frac{\frac{1}{(m-i)} \cdot \sum_{k=1}^{m-i} \left(f_k - \overline{f}\right) \cdot \left(f_{k+i} - \overline{f}\right)}{\frac{1}{m} \cdot \sum_{k=1}^{m} \left(f_k - \overline{f}\right)^2}$$

with m the length of the random walk, f_k the value of f on the k-th assignment of the random walk, and $\overline{f}$ the mean of those f_k.

We can then compute the empirical correlation length l for a distance of 1, as long as $r(1) \neq 0$ holds:

$$l := \frac{1}{ln(|r(1)|)}$$

Intuitively, the higher the value l, the smoother the function f.

However, we needed to avoid learning a flat function projecting every assignment to the value 0 (it would be very smooth but also completely useless for the solver). To avoid this situation, the fitness function of the genetic algorithm was to maximize l times the mean of f over assignments in the random walk that are not solutions (ie, $f(\vec{x}) \neq 0$). Thus, the algorithm was supposed to try learning smooth functions that severely penalize non-solution assignments.

These two approaches, multivariate interpolation and genetic algorithms, failed mainly because of the same reason: expressing error functions with a sum of p sinusoids led to either functions far from being satisfying ($p < 40$) or with too many coefficients to learn ($p \geq 40$).

5.2 CPPN

Before expressing error functions with ICNs, we tried to represent them with a Compositional Pattern-Producting Network (CPPN) [37], a variant of neural networks from which ICN is inspired. Our CPPN architecture was a two-layer

network composed of units with an activation function among the identity, the absolute value, the sine function, the hyperbolic tangent, the cubic hyperbolic tangent, a sigmoid function, and a Gaussian function.

EF-COP

We first tried to tackle the problem of learning error functions represented by a CPPN as an optimization problem modeled as an EF-COP. We have one variable for each unit in the network to express its selection (or its absence) to model the error function. Domains are then binary. A constraint assured that outputs of the CPPN given some assignments as input are greater than or equals to the Hamming distance of these assignments. To have smooth functions, we also considered the objective function based on the computation of the landscape ruggedness, as introduced in Section 5.1 above. However, such an EF-COP model revealed itself to be inefficient due to a unique constraint over all variables, which was quite artificial and led to a poor (actually, an absent) constraint network that neither a complete solver nor a constraint-based local search can exploit. Moreover, computing the ruggedness as an objective function to have smooth functions tended to output very flat functions.

We also felt that activation functions of our CPPN where too arbitrate to express error functions. We then had the idea to replace these activation functions with more meaningful ones in the context of comparing and evaluating values in a assignment, leading to our current Interpretable Compositional Networks.

6 Experiments

To show the versatility of our method, we tested it on seven different constraints: AllDifferent, Ordered, LinearSum, LinearLessThan, LinearGreaterThan, NoOverlap1D, and Minimum. According to XCSP specifications [7][1], those global constraints belong to four different families: Comparison (AllDifferent and Ordered), Counting/Summing (LinearSum, LinearLessThan, LinearGreaterThan), Packing/Scheduling (NoOverlap1D) and Connection (Minimum). Again according to XCSP specifications, these constraints are among the twenty most popular and common constraints. We give a brief description of those seven constraints below:

- **AllDifferent** ensures that variables must all be assigned to different values.
- **LinearSum** ensures that the equation $x_1 + x_2 + \ldots + x_n = p$ holds, with the parameter p a given integer.
- **LinearLessThan** ensures that the inequation $x_1 + x_2 + \ldots + x_n \leq p$ holds, with the parameter p a given integer.
- **LinearGreaterThan** ensures that the inequation $x_1 + x_2 + \ldots + x_n \geq p$ holds, with the parameter p a given integer.

[1] see also http://xcsp.org/specifications

- **Minimum** ensures that the minimum value of an assignment verifies a given numerical condition. In this paper, we choose to consider that the minimum value must be greater than or equals to a given parameter p.
- **NoOverlap1D** is considering variables as tasks, starting from a certain time (their value) and each with a given length p (their parameter). The constraint ensures that no tasks are overlapping, *i.e.*, for all indices $i, j \in \{1, n\}$ with n the number of variables, we have $x_i + p_i \leq x_j$ or $x_j + p_j \leq x_i$. To have a simpler code, we have considered in our system that all tasks have the same length p.
- **Ordered** ensures that an assignment of n variables $(x_1, \ldots, x_n)$ must be ordered, given a total order. In this paper, we choose the total order $\leq$. Thus, for all indices $i, j \in \{1, n\}$, $i < j$ implies $x_i \leq x_j$.

6.1 Experimental protocols

We conducted three experiments, with two of them requiring samplings. These samplings have been done using Latin hypercube sampling to have a good diversity among drawn assignments. We draw assignments until we get k solutions and k non-solutions.

Due to stochastic learning, all learning and testing have been done 100 times, but over the same pre-computed training sets, to not let the randomness of sampled sets impact the results in some way. We did not re-run batches of experiments to keep the ones with the best results, as it should always be the case with such experimental protocols.

We have hold-out test sets of assignments from larger dimensions to evaluate the quality of our learned error functions. Like written at the end of Section 4.3.2, we do not have any validation sets since we use the default value of the solver parameters.

All experiments have been done on a computer with a Core i9 9900 CPU and 32 GB of RAM, running on Ubuntu 20.04. Programs have been compiled with GCC with the 03 optimization option. Our entire system, its C++ source code, experimental setups, and the results files are accessible on GitHub[2].

6.1.1 Experiment 1: scaling

The goal of this experiment is to show that learned error functions scale to high-dimensional constraints, indicating that learned error functions are independent of the size of the constraint scope.

For this experiment, error functions are learned upon a small, complete constraint assignment space, composed of about 500∼600 assignments and containing about 10∼20% of solutions. For each constraint, we run the learning algorithm 100 times on the same pre-computed complete constraint assignment space. Then, we compute the test error of these learned error functions over a sampled test set. Sampled test sets contain 10,000 solutions and 10,000

[2] https://github.com/richoux/LearningErrorFunctions/tree/2.1

non-solutions, with 100 variables on domains of size 100, belonging to a constraint assignment space of size $100^{100} = 10^{200}$ (except for NoOverlap1D and Ordered, as explained below), thus greatly larger than training spaces containing 500∼600 assignments.

For AllDifferent, LinearSum, LinearLessThan, LinearGreaterThan and Minimum, it is easy to define by hand a formula computing the Hamming and Manhattan costs of any assignment $\vec{x}$ without generating the whole constraint assignment space. For these constraints, we tested the corresponding error function on spaces with 100 variables and domains of size 100.

Whereas for Ordered and NoOverlap1D, since these two constraints are intrinsically combinatorial, finding a formula computing the exact Hamming and Manhattan costs of any assignment is not trivial. Therefore, we sampled 10,000 solutions and 10,000 non-solutions in constraint assignment spaces of Ordered with 12 variables and domains of size 18 (so 18^{12} assignments, *i.e.*, about $1.15e^{15}$) and NoOverlap1D with 10 variables and domains of size 35 ($35^{10} \simeq 2.75e^{15}$ assignments). Then we approximate the Hamming and Manhattan costs of each non-solution, considering the closest solution among the 10,000 sampled solutions. It was not possible to build test sets of higher dimensions for these two constraints since sampling 10,000 solutions is challenging: for Ordered, we estimate the solution rate to be $8.6e^{-10}$ (to make this number concrete, after 100 billion samplings, one can expend finding 86 solutions); for NoOverlap1D, the solution rate is about $3.6e^{-9}$. On a regular computer, it took us a bit more than 10 hours to generate the test set of Ordered. Knowing that such an execution time grows exponentially, generating test sets of higher dimensions would take an unreasonable amount of time.

We show normalized mean training and test errors: first, we compute the mean error among all assignments composing the training or the test set. Normalization is done like in Equation 2: errors from error functions approximating the Hamming cost are divided by the number of variables composing the assignments, and errors from error functions approximating the Manhattan cost are divided by the number of variables composing the assignments times to difference between the maximal and the minimal value in the domains.

Considering normalized errors is important: for instance, having a Hamming-like mean error of 5 on assignments with 10 variables and 100 variables is significantly different: looking at their normalized error allows to realized that the first one implies a mean error every 2 variables, the second a mean error every 20 variables.

6.1.2 Experiment 2: learning over incomplete spaces

If, for some reasons, the users want to learn the error function of a constraint directly over a large number of variables, rather than working on this constraint over few variables, then it may be not possible to build a complete constraint assignment space within a reasonable time. However, a robust system must be able to learn effective error functions upon large, incomplete spaces where the exact Hamming and Manhattan costs of their assignments is unknown.

In this experiment, we built pre-sampled training spaces by sampling 10,000 solutions and 10,000 non-solutions on large constraint assignment spaces of size between 10^{12} and 10^{13}, and with solution rates from 0.15% to $2e^{-7}$%. Then, we approximate the Hamming and Manhattan costs of each non-solution by computing their Hamming and Manhattan distances with the closest solution among the 10,000 ones, learn error functions on these 20,000 assignments and their approximated Hamming and Manhattan costs. Like for Experiment 1, we run the learning algorithm 100 time on the same pre-sampled incomplete spaces, so that each run relies on the same training set. Finally, we evaluate the learned error functions over the same test sets than Experiment 1.

6.1.3 Experiment 3: learned error functions to solve problems

The goal of this experiment is to assess that learned error function can effectively be used to solve 3 classic combinatorial problems. We modeled these problems and solve them using the framework GHOST [33].

We consider the mean and median run-time to compare different representations of our constraints. We take as baseline a pure CSP model where constraints are predicates. We also consider an EF-CSP model with an efficient hand-crafted error function for AllDifferent. We compare those with two models using error functions learned with our system: a), our EF-CSP model using the most frequently learned error function from the previous experiments and computed by running the ICN graph in a fast-forward fashion, and b), our EF-CSP model with the same error function but directly hard-coded in C++. The solver and its parameters remain the same: the only thing that changes in these four different models is the expression of the constraint. Notice that for LinearSum, our system learned the canonical and optimal error function, so the hand-crafted and hard-coded versions are the same.

All problem instances have been solved 100 times each, using the constraint-based local search solver within the framework GHOST, in sequential mode. For each run, we set a timeout of 60 seconds. If no solutions have been found within 60 seconds, we consider the run to be unsolved.

Sudoku

Sudoku is a puzzle game where the player must fill a grid with numbers from 1 to n, n being the size of the side of the grid, such that all numbers in the same row, the same column and the same sub-square must be different. Sudoku can be modeled as a satisfaction problem using the AllDifferent constraint only. We run 100 resolutions of random 9×9, 16×16 and 25×25 Sudoku grids.

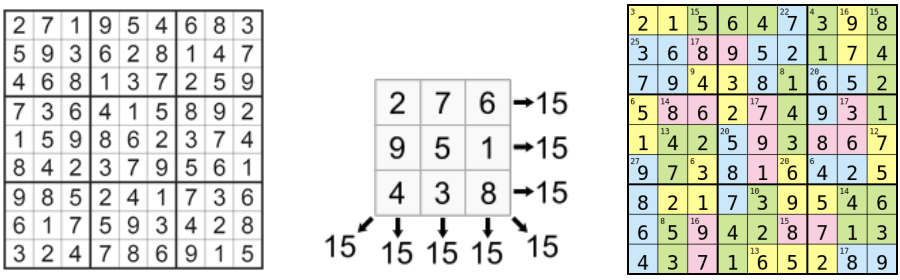
Magic Square

Magic Square is a $n \times n$ grid that must be filled up with all numbers from 1 to n^2 (thus, all numbers must appear exactly once in the grid), such that the sum of each row, each column, and the two diagonals must be equal to a constant p. We can make a better model than the one given as an example at the end of Section 2, by avoiding using the AllDifferent constraint by randomly filling

up the grid with all expected numbers and ask the solver to find a correct permutation. The constraints over the rows, columns, and the two diagonal are modeled with LinearSum since the value of p only depends on n and is known to be $p = n(n^2 + 1)/2$. We run 100 resolutions of 25×25 and 30×30 Magic Square grids.

Killer Sudoku

Killer Sudoku is the same as Sudoku but in such a way that the grid is paved with blocks of cells, named cages, usually composed of 2, 3, or 4 cells. Each cage is associated an integer, and the sum of numbers in their cells must be equals to their integer. A killer Sudoku instance starts with an empty grid, cages preventing from trivial solutions. AllDifferent constraints are used to model the regular Sudoku rules of this puzzle game, and LinearSum constraints are modeling cages. We run 100 resolutions of a 9×9 Killer Sudoku grid.



(a) Sudoku 9×9 (b) Magic Square 3×3 (c) Killer Sudoku 9×9

Fig. 4: Our classic combinatorial problems as benchmarks.

6.2 Results

In this part, we denote by n the number of variables, d the domain size, and p the value of a possible parameter. Constraint instances are denoted by *name-n-d[-p]*.

6.2.1 Experiments 1 & 2

Tables 1 and 2 show the training errors of Experiments 1 and 2, respectively, where error functions have been learned 100 times for each constraint. The first column contains the normalized mean training error of the most frequently learned error function among the 100 runs, with its frequency in parenthesis. Next columns concern the median and the mean together with its standard deviation.

Learning an error function is done quickly: we set a timeout of 200ms to learn an error function over complete constraint assignment spaces from

Experiment 1, and 30s over incomplete constraint assignment spaces from Experiment 2. Learnings have been done in parallel using 16 threads.

Table 1: Training errors (100 runs) of Experiment 1, over small and complete constraint assignment spaces.

Constraints	most freq	median	mean ± std dev
AllDifferent-4-5	0 (100)	0	0 ± 0
LinearSum-3-8-12	0 (74)	0	0.005 ± 0.014
LinearLessThan-3-8-12	0 (85)	0	0.008 ± 0.021
LinearGreaterThan-3-8-12	0 (92)	0	0.003 ± 0.012
Minimum-4-5-3	0 (82)	0	0.005 ± 0.016
NoOverlap1D-3-8-2	0.007 (34)	0.007	0.009 ± 0.003
Ordered-4-5	0.009 (80)	0.009	0.013 ± 0.010

Table 3 contains the normalized mean test errors of error functions learned with Experiments 1 and 2, with their median, mean and standard deviation. The normalized mean test error of the most frequently learned error function for each constraint in each experiment has been isolated in the first column of number, for comparison.
Comparing Table 1 with the first half of Table 3 lead us to conclude that our system is able to learn error functions that scale for most constraint, namely AllDifferent, LinearSum, LinearLessThan, LinearGreaterThan and Minimum. Our system has been able to find the exact Hamming distance or Manhattan distance for these constraints.

Observe that LinearLessThan has a surprizingly high mean and standard deviation. This is due to an error function that show an excellent score on the training set but a very poor score on the test sets. Our system was overfitting this training set twice within 100 runs. In practice, this problem could be alleviated by letting more than 200ms to the solver to find an optimal solution, by taking a slightly larger training set or eventually by running a second time the error function learning, if the user realises that it performs poorly on the

Table 2: Training errors (100 runs) of Experiment 2, over large and incomplete constraint assignment spaces.

Constraints	most freq	median	mean ± std dev
AllDifferent-12-12	0.018 (91)	0.018	0.019 ± 0.001
LinearSum-12-12-42	$2e^{-4}$ (63)	$2e^{-4}$	0.015 ± 0.031
LinearLessThan-12-12-42	$4e^{-4}$ (50)	0.005	0.022 ± 0.033
LinearGreaterThan-12-12-42	0.027 (46)	0.034	0.032 ± 0.004
Minimum-12-12-6	0.019 (24)	0.021	0.022 ± 0.004
NoOverlap1D-8-32-3	0.030 (19)	0.030	0.029 ± 0.002
Ordered-12-12	0.019 (99)	0.019	0.019 ± 0.001

Table 3: Test errors (100 runs) in high dimensions of error functions learned with Experiments 1 and 2.

Exp.	Constraints	most freq	median	mean ± std dev
1	AllDifferent-100-100	0	0	0 ± 0
	LinearSum-100-100-5279	0	0	0.001 ± 0.004
	LinearLessThan-100-100-5279	0	0	19.790 ± 117.033
	LinearGreaterThan-100-100-5279	0	0	0.003 ± 0.007
	Minimum-100-100-30	0	0	0.026 ± 0.123
	NoOverlap1D-10-35-3	0.062	0.062	0.060 ± 0.004
	Ordered-12-18	0.035	0.035	0.045 ± 0.022
2	AllDifferent-100-100	0.006	0.006	0.007 ± 0.001
	LinearSum-100-100-5279	0	0	0.186 ± 0.659
	LinearLessThan-100-100-5279	0	0.001	0.224 ± 0.692
	LinearGreaterThan-100-100-5279	0.020	0.020	0.020 ± $1e^{-4}$
	Minimum-100-100-30	0.200	0.200	0.208 ± 0.100
	NoOverlap1D-10-35-3	0.050	0.041	0.041 ± 0.004
	Ordered-12-18	0.037	0.037	0.045 ± 0.022

test set. Since the learning is done very quickly (as well as testing the error function), this is not a major issue for the real usage of our system.

Althought not perfect, results are good for NoOverlap1D and Ordered, which are clearly the most intrinsically combinatorial constraints among our seven ones. Our system is able to learn error functions with a low test errors and a low standard deviation of performance between the different learned functions. However, since their training errors are significantly lower than their test errors, one could think that our system is overfitting here. Results from Experiment 2 lead us to another conclusion.

First, let's analyse the results of our five first constraints over large, incomplete training sets. It is important to stress that the real Hamming and Manhattan costs in these training sets are unknown and roughtly approximated on purpose. Nevertheless, comparing Table 2 with the second half of Table 3 shows us that our system is able to find high-quality error functions for AllDifferent, LinearSum, LinearLessThan and LinearGreaterThan. This illustrates that our system can learn efficient error function over incomplete constraint assignment spaces. Observe that the learned error functions for the Minimum constraint over incomplete spaces perform poorly, while it learns a perfect error function over complete spaces. The reason is the following: over complete spaces, learned error function for Minimum reproduce the Hamming cost, which is an excellent choice. However, sampled assignments that constitute the incomplete space guide the solver toward Manhattan-like error functions, which is a poor choice for this constraint. This shows that learning over incomplete spaces often produces high-quality error functions, but can sometimes lead to poorly learned ones.

Let's focus now on NoOverlap1D and Ordered. Looking at their scores on the first and second part of Table 3, we can see that their performances are similar and very homogeneous, with a low standard deviation. This is explained but the fact that their error functions learned over both complete and incomplete spaces are very similar, showing that our system was not overfitting their complete training sets. The reason explaining the difference between their training errors in Table 1 and the first half of Table 3 is because their training spaces from Experiment 1 were too small for these highly combinatorial constraints, containing too few different combinations and Hamming/Manhattan cost patterns. In other words, those small spaces does not contain very diverse assignments, penalizing the learning. This explains why results for NoOverlap1D are better over incomplete spaces than complete spaces. However, we think that the elementary operations composing our ICN model are not rich enough to properly express the complexity of NoOverlap1D and Ordered.

We give in appendix the list of the most frequently learned error function for each constraint, both over complete and incomplete spaces.

6.2.2 Experiment 3

The goal of this experiment is not to be state-of-the-art in terms of run-times for solving Sudoku, Magic Square and Killer Sudoku, but to compare the average run-times of the same solver on three or four nearly identical models, depending on the problem, presented in Section 6.1.3. For models with a hand-crafted error function of AllDifferent, we implemented the *primal graph based violation error* from Petit et al. [38]. This function simply outputs the number of couples with identical values within a given assignment. For LinearSum, we do not know better hand-crafted error functions than the one learned the most frequently by our system.

To run this experiment, we used the framework GHOST [33], which includes a constraint-based local search algorithm able to handle both CSP and EF-CSP models.

Table 4 shows that EF-CSP models clearly outperformed their equivalent CSP model, except for smaller Magic Square instances. We run 100 solving of different problem instances for each model and compute the mean and median run-time in seconds, as well as the standard deviation and the success rate, *i.e.*, the number of runs out of 100 that found a solution within 60 seconds. Rows in gray indicates that the success rate is below 100%.

We can estimate the overload of computing the error function through the Interpretable Compositional Network (ran in a feed-forward fashion), compare to a hard-coded version of the same error function. We recall that one advantage of our method is to output intelligible error functions, letting the choice to users to compute this function through the Interpretable Compositional Network or to let them the possibility to code it themselves. Results from Table 4 show that the overload is such that run-times of error functions executed through the interpretable compositional network are between 30% and 80% longer than run-times of their hard-coded version.

Table 4: Run-times in seconds over 100 runs to solve classic problems with 4 different representations of constraints (3 for Magic Square).
Rows in gray means that some runs hit the 60-second timeout.

Problem	Error Function	mean ± std dev	median	success (%)
Sudoku 9 × 9	none (CSP)	2.6 ± 10.6	0.1	97
	fast-forward	$2e^{-2} \pm 1e^{-2}$	$2e^{-2}$	100
	hard-coded	$1e^{-2} \pm 7e^{-3}$	$1e^{-2}$	100
	hand-crafted	$1e^{-2} \pm 5e^{-3}$	$1e^{-2}$	100
Sudoku 16 × 16	none (CSP)	59.9 ± 0	59.9	1
	fast-forward	0.9 ± 0.2	0.8	100
	hard-coded	0.5 ± 0.1	0.5	100
	hand-crafted	0.4 ± 0.1	0.3	100
Sudoku 25 × 25	none (CSP)	-	-	0
	fast-forward	28.5 ± 14.6	25.7	94
	hard-coded	17.3 ± 10.3	14.0	100
	hand-crafted	9.4 ± 5.7	7.6	100
Magic Square 25 × 25	none (CSP)	10.4 ± 5.5	8.5	100
	fast-forward	9.7 ± 9.8	6.1	100
	hard-coded	6.8 ± 5.5	5.1	100
Magic Square 30 × 30	none (CSP)	29.9 ± 11.4	27.3	96
	fast-forward	19.1 ± 14.3	16.2	98
	hard-coded	14.9 ± 11.5	11.7	100
Killer Sudoku 9 × 9	none (CSP)	-	-	0
	fast-forward	2.4 ± 2.2	1.6	100
	hard-coded	1.5 ± 1.1	1.1	100
	hand-crafted	1.1 ± 0.9	0.8	100

The same difference of performance is observed between hard-coded and hand-crafted version of error functions: we see that the most frequently learned error function by our system, once hard-coded in C++, finds solutions within between 30% and 80% more time that as the carefully hand-crafted error function from Petit et al. [38]. Although perfectible, these results are encouraging and show that our method can be used to automatically find error functions that are usable in practice.

7 Conclusion

In this paper, we give a formal definition of Error Function-based Constraint Satisfaction and Optimization Problems, and we present a method to learn error functions automatically upon a model based on Interpretable Compositional Networks, a particular directed acyclic graph. To the best of our knowledge, this is the first attempt to learn error functions for hard constraints automatically.

We have tested our system over 7 different constraints. In Experiment 1 over small and complete training spaces, it finds the exact Hamming or Manhattan costs for 5 of them. The beauty of our method is that we learn error functions by solving an Error Function-based Constrainted Optimization Problem. Error functions of these 7 constraints have been learned over constraint assignment space composed of 500∼600 assignments and perfectly scale on high-dimension constraint instances with 10^{200} assignments. In Experiment 2, we show the robustness of our system by learning error functions over incomplete constraint assignment space containing 20,000 randomly drawn assignments from spaces of about 10^{12} assignments. It finds high-quality error functions for 4 out of 5 constraints that had a perfect error function in the previous Experiment 1.

With the analysis of our results, we conclude it is better to use our system over complete spaces for simple constraints such as AllDifferent, LinearSum and Minimum. For more complex constraints, like NoOverlap1D and Ordered, experiments show that very small training spaces are too restricted and do not contain enough of diverse assignments, and the current set of elementary operations composing our ICN model is certainly not expressive enough.

Results from Experiment 3 show two things. First, while using a constraint-based local search solver, there is a real gain to model Constraint Programming problems with EF-CSP models rather than the classical CSP models. Second, our system learns high-quality error functions that can be used in practice to efficiently express contraints in combinatorial problems.

These two points imply that our method allows users to get the power of error function-based models for free, leveraging the difficulty of their modeling: users can get an EF-CSP or an EF-COP model with the same modeling effort as for classical CSP and COP models. Like Freuder [3] wrote: "*This research program is not easy because 'ease of use' is not a science.*" However, we believe our result is a step toward the 'ease of use' of Constraint Programming.

One of the most significant results in this paper is that our system outputs interpretable results. Error functions output by our system are intelligible. This allows our system to have two operating modes: 1) a fully automatic system, where error functions are learned and called within our system, being completely transparent to users who only need to furnish a concept function for each constraint, in addition to the regular sets of variables V and domains D, and 2) a decision support system, where users can look at a set of proposed error functions, pick up and modify the one they prefer.

We made this system modular, allowing users with special needs to add or remove operations in the system to learn more specific error functions.

An extension of our work would be to do reinforcement learning rather than supervision learning based on the Hamming or Manhattan cost. Indeed, even if these costs seem natural metrics to tell how far an assignment is to be a solution for constraint-based local search solvers, it could also be too restrictive. Learning via reinforcement learning would allow finding error functions that are more adapted to the chosen solver.

Another interesting extension is the theoretical study of the properties of elementary operations and their combinations, together with the properties of the search landscape such combinations imply, such as the ruggedness, the solution density, the presence of funnels, the solution symetries, etc. Having a deeper knowledge of these properties would help selecting the right elementary operations for an ICN regarding the type of constraints users aim to represent.

Statements and Declarations

Conflict of interest

The authors have no competing interests to declare that are relevant to the content of this article.

Data and code availability

Our entire system, its C++ source code, experimental setups, and the results files are accessible on the following GitHub repository: https://github.com/richoux/LearningErrorFunctions/tree/2.1

License

References

[1] Freuder, E.C.: In pursuit of the holy grail. Constraints **2**(1), 57–61 (1997). https://doi.org/10.1023/A:1009749006768

[2] Freuder, E.C.: Progress towards the holy grail. Constraints **23**(2), 158–171 (2018). https://doi.org/10.1007/s10601-017-9275-0

[3] Freuder, E.C.: Holy grail redux. Constraint Programming Letters **1**, 3–5 (2007)

[4] Puget, J.-F.: Constraint programming next challenge: Simplicity of use. In: Proceedings of the International Conference on Principles and Practice of Constraint Programming (CP 2004), pp. 5–8. Springer, Heidelberg (2004)

[5] Wallace, M.: Languages versus packages for constraint problem solving. In: Proceedings of the International Conference on Principles and Practice

of Constraint Programming (CP 2003), pp. 37–52. Springer, Heidelberg (2003). https://doi.org/10.1007/978-3-540-45193-8_3

[6] Frisch, A., Harvey, W., Jefferson, C., Martínez-Hernández, B., Miguel, I.: ESSENCE: A constraint language for specifying combinatorial problems. Constraints **13**, 268–306 (2008)

[7] Boussemart, F., Lecoutre, C., Audemard, G., Piette, C.: XCSP3: An Integrated Format for Benchmarking Combinatorial Constrained Problems. arXiv e-prints **abs/1611.03398**, 1–238 (2016)

[8] Nethercote, N., Stuckey, P.J., Becket, R., Brand, S., Duck, G.J., Tack, G.: Minizinc: Towards a standard cp modelling language. In: Proceedings of the International Conference on Principles and Practice of Constraint Programming (CP 2007), pp. 529–543. Springer, Heidelberg (2007)

[9] Bessière, C.: Constraint reasoning. In: A Guided Tour of Artificial Intelligence Research vol. 2, pp. 153–183 (2020)

[10] Cooper, M., Givry, S., Schiex, T.: Valued constraint satisfaction problems. In: A Guided Tour of Artificial Intelligence Research vol. 2, pp. 185–207 (2020)

[11] Meseguer, P., Rossi, F., Schiex, T.: Soft constraints. In: The Handbook of Constraint Programming, pp. 279–326. Elsevier, Amsterdam (2006)

[12] Hurley, B., O'sullivan, B., Allouche, D., Katsirelos, G., Schiex, T., Zytnicki, M., Givry, S.D.: Multi-language evaluation of exact solvers in graphical model discrete optimization. Constraints **21**(3), 413–434 (2016). https://doi.org/10.1007/s10601-016-9245-y

[13] Codognet, P., Diaz, D.: Yet another local search method for constraint solving. In: Proceedings of the International Symposium on Stochastic Algorithms: Foundations and Applications (SAGA 2001), pp. 73–90 (2001)

[14] Caniou, Y., Codognet, P., Richoux, F., Diaz, D., Abreu, S.: Large-scale parallelism for constraint-based local search: The costas array case study. Constraints **20**(1), 30–56 (2015). https://doi.org/10.1007/s10601-014-9168-4

[15] Borning, A., Freeman-Benson, B., Wilson, M.: Constraint hierarchies. In: Constraint Programming, pp. 75–115 (1994)

[16] Galinier, P., Hao, J.: A general approach for constraint solving by local search. J. Math. Model. Algorithms **3**(1), 73–88 (2004)

[17] Mezura-Montes, E., Coello Coello, C.A.: Constraint-handling in nature-inspired numerical optimization: Past, present and future. Swarm and Evolutionary Computation **1**(4), 173–194 (2011)

[18] Paulus, A., Rolínek, M., Musil, V., Amos, B., Martius, G.: Comboptnet: Fit the right np-hard problem by learning integer programming constraints. In: Proceedings of the 38th International Conference on Machine Learning (ICML 2021), pp. 8443–8453. PMLR, Online (2021)

[19] Kumar, M., Kolb, S., De Raedt, L., Teso, S.: Learning mixed-integer linear programs from contextual examples. arXiv e-prints **abs/2107.07136**, 1–11 (2021)

[20] Bessiere, C., Coletta, R., Koriche, F., O'Sullivan, B.: A SAT-Based Version Space Algorithm for Acquiring Constraint Satisfaction Problems. In: Proceedings of the 16th European Conference on Machine Learning (ECML 2005), pp. 23–34. Springer, Heidelberg (2005). https://doi.org/10.1007/11564096_8

[21] Bessiere, C., Coletta, R., O'Sullivan, B., Paulin, M.: Query-driven constraint acquisition. In: Proceedings of the 20th International Joint Conference on Artificial Intelligence (IJCAI 2007), pp. 50–55. IJCAI/AAAI Press, Palo Alto (2007)

[22] Bessiere, C., Coletta, R., Hebrard, E., Katsirelos, G., Lazaar, N., Narodytska, N., Quimper, C., Walsh, T.: Constraint acquisition via partial queries. In: Proceedings of the 23rd International Joint Conference on Artificial Intelligence (IJCAI 2013), pp. 475–481. IJCAI/AAAI Press, Palo Alto (2013)

[23] Beldiceanu, N., Simonis, H.: A model seeker: Extracting global constraint models from positive examples. In: Proceedings of the International Conference on Principles and Practice of Constraint Programming (CP 2012), pp. 141–157. Springer, Heidelberg (2012)

[24] Beldiceanu, N., Carlsson, M., Demassey, S., Petit, T.: Global constraint catalog: Past, present and future. Constraints **12**, 21–62 (2007)

[25] Beldiceanu, N., Simonis, H.: A constraint seeker: Finding and ranking global constraints from examples. In: Proceedings of the International Conference on Principles and Practice of Constraint Programming (CP 2011), pp. 12–26. Springer, Heidelberg (2011)

[26] Teso, S.: Constraint learning: An appetizer. In: Reasoning Web: Explainable Artificial Intelligence, pp. 232–249. Springer, Heidelberg (2019). https://doi.org/10.1007/978-3-030-31423-1_7

[27] Teso, S., Bliek, L., Borghesi, A., Lombardi, M., Yorke-Smith, N., Guns, T., Passerini, A.: Machine learning for combinatorial optimisation of partially-specified problems: Regret minimisation as a unifying lens. arXiv e-prints **abs/2205.10157**, 1–12 (2022)

[28] Deshwal, A., Doppa, J.R., Roth, D.: Learning and inference for structured prediction: A unifying perspective. In: Proceedings of the 28th International Joint Conference on Artificial Intelligence (IJCAI 2019), pp. 6291–6299. IJCAI/AAAI Press, Palo Alto (2019)

[29] Domshlak, C., Prestwich, S., Rossi, F., Venable, K., Walsh, T.: Hard and soft constraints for reasoning about qualitative conditional preferences. J. Heuristics **12**, 263–285 (2006)

[30] Bessiere, C., Koriche, F., Lazaar, N., O'Sullivan, B.: Constraint acquisition. Artificial Intelligence **244**, 315–342 (2017). https://doi.org/10.1007/3-540-45470-5

[31] Richoux, F., Baffier, J.-F.: Error function learning with interpretable compositional networks for constraint-based local search. In: Proceeding of the 2021 Genetic and Evolutionary Computation Conference (GECCO 2021), pp. 137–138. ACM, New York (2021). https://doi.org/10.1145/3449726.3459464

[32] Koza, J.R.: Survey of genetic algorithms and genetic programming. In: Wescon Conference Record, pp. 589–594 (1995)

[33] Richoux, F., Uriarte, A., Baffier, J.-F.: GHOST: A combinatorial optimization framework for real-time problems. IEEE Transactions on Computational Intelligence and AI in Games **8**(4), 377–388 (2016). https://doi.org/10.1109/TCIAIG.2016.2573199

[34] Gaure, S.: chebpol: Multivariate Interpolation. https://rdrr.io/cran/chebpol/man/chebpol-package.html. [Online; accessed 17 October 2022] (2019)

[35] Grimstad, B.: SPLINTER. https://github.com/bgrimstad/splinter. [Online; accessed 17 October 2022] (2016)

[36] Hoos, H.H., Stützle, T.: Stochastic Local Search: Foundations and Applications. Elsevier, Amsterdam (2005)

[37] Stanley, K.O.: Compositional Pattern Producing Networks: A Novel Abstraction of Development. Genetic Programming and Evolvable Machines **8**(2), 131–162 (2007). https://doi.org/10.1007/s10710-007-9028-8

[38] Petit, T., Régin, J.-C., Bessiere, C.: Specific filtering algorithms for over-constrained problems. In: Proceedings of the International Conference on Principles and Practice of Constraint Programming (CP 2001). Springer, Heidelberg (2001). https://doi.org/10.1007/3-540-45578-7_31

Appendix A List of elementary operations

A.1 Transformation layer

- Identity

$$id(x[i]) := x[i]$$

- Number of elements on the right equals to $x[i]$

$$Count^r_=(x[i]) := \#\{x[j] \mid j > i \wedge x[j] = x[i]\}$$

- Number of elements on the right smaller than $x[i]$

$$Count^r_<(x[i]) := \#\{x[j] \mid j > i \wedge x[j] < x[i]\}$$

- Number of elements on the right greater than $x[i]$

$$Count^r_>(x[i]) := \#\{x[j] \mid j > i \wedge x[j] > x[i]\}$$

- Number of elements on the left equals to $x[i]$

$$Count^l_=(x[i]) := \#\{x[j] \mid j < i \wedge x[j] = x[i]\}$$

- Number of elements on the left smaller than $x[i]$

$$Count^l_<(x[i]) := \#\{x[j] \mid j < i \wedge x[j] < x[i]\}$$

- Number of elements on the left greater than $x[i]$

$$Count^l_>(x[i]) := \#\{x[j] \mid j < i \wedge x[j] > x[i]\}$$

- Number of elements equals to $x[i]$ + param

$$Count_{=+p}(x[i]) := \#\{x[j] \mid x[j] = x[i] + param\}$$

- Number of elements smaller than $x[i]$ + param

$$Count_{<+p}(x[i]) := \#\{x[j] \mid x[j] < x[i] + param\}$$

- Number of elements greater than $x[i]$ + param

$$Count_{>+p}(x[i]) := \#\{x[j] \mid x[j] > x[i] + param\}$$

- max(0, $x[i]$ - param)
- max(0, param - $x[i]$)
- max(0, $x[i] - x[i+1]$)
- max(0, $x[i+1] - x[i]$)
- Number of elements equals to $x[i]$

$$Count_{=}(x[i]) := \#\{x[j] \mid x[j] = x[i]\}$$

- Number of elements smaller than $x[i]$

$$Count_{<}(x[i]) := \#\{x[j] \mid x[j] < x[i]\}$$

- Number of elements greater than $x[i]$

$$Count_{>}(x[i]) := \#\{x[j] \mid x[j] > x[i]\}$$

- Number of elements greater than or equals to $x[i]$ AND less than or equals to $x[i]$ + param

$$Count_{>=<+p}(x[i]) := \#\{x[j] \mid x[j] \geq x[i] \wedge x[j] \leq x[i] + param\}$$

A.2 Arithmetic layer

- Sum of the i-th element of each vector $\vec{x}_j$: $\forall i \in \{1, n\} \sum_{j=1}^{k} x_j[i]$
- Product of the i-th element of each vector $\vec{x}_j$: $\forall i \in \{1, n\} \prod_{j=1}^{k} x_j[i]$

A.3 Aggregation layer

- $\sum_{i=1}^{n} x[i]$
- $Count_{>0}(\vec{x}) := \#\{x[i] \mid x[i] > 0\}$

A.4 Comparison layer

- $id(x) = x$
- $\mid x - param \mid$
- $\max(0, param - x)$
- $\max(0, x - param)$
- $Euclidian_p(x)$:= If(x = param) then 0 else $1 + \frac{|x-param|}{maximal\ domain\ size}$
- $Euclidian(x)$:= If(x = 0) then 0 else $1 + \frac{x}{maximal\ domain\ size}$
- | x - number of variables|
- max(0, number of variables - x)
- max(0, x - number of variables)

Appendix B Most frequently learned error functions

Elementary operations from the transformation layer are applyed element-wise. Therefore, an operation like $Count^l_{=}(\vec{x})$ is applied on each element $x[i]$ of the vector $\vec{x}$, producing a transformed vector.

We denote by n, d and p the number of variables, the domain size and the value of the parameter, respectively.

B.1 Complete spaces

$AllDifferent:$	$Count_{>0}\big(Count^l_{=}(\vec{x})\big)$
$LinearSum:$	$\mid \sum_{i=1}^{n} x[i] - p \mid$
$LinearLessThan:$	$\max(0, \sum_{i=1}^{n} x[i] - p)$
$LinearGreaterThan:$	$\max(0, p - \sum_{i=1}^{n} x[i])$
$Minimum:$	$Count_{>0}\big(\max(0, p - x[i])\big)$
$NoOverlap1D:$	$\max\Big(0, \sum_{i=1}^{n}\big(Count^l_{=}(\vec{x}) + Count_{<+p}(\vec{x})\big) - n\Big)$
$Ordered:$	$\sum_{i=1}^{n}\big(\max(0, x[i] - x[i+1])\big)$

B.2 Incomplete spaces

$AllDifferent:$	$Count_{>0}\big(Count_{=}(\vec{x}) + Count^r_{=}(\vec{x})\big)$
$LinearSum:$	$\mid \sum_{i=1}^{n} x[i] - p \mid$
$LinearLessThan:$	$\max(0, \sum_{i=1}^{n} x[i] - p)$
$LinearGreaterThan:$	$\max\Big(0, p - \sum_{i=1}^{n}\big(id(\vec{x}) \times \max(0, x[i+1] - x[i])\big)\Big)$
$Minimum:$	$\max\Big(0, \sum_{i=1}^{n}\big(Count_{>+p}(\vec{x}) + \max(0, p - x[i])\big) - p\Big)$
$NoOverlap1D:$	$\max\Big(0, \sum_{i=1}^{n}\big(Count_{>=<+p}(\vec{x}) + Count_{<+p}(\vec{x})\big) - p\Big)$
$Ordered:$	$\sum_{i=1}^{n}\big(Count^r_{<}(\vec{x})\big)$