Automatic Cost Function Learning with Interpretable Compositional Networks

Florian Richoux[1] and Jean-François Baffier[2]

[1]Université de Nantes, France / JFLI, CNRS, NII, Japan
[2]RIKEN AIP, Japan
florian.richoux@polytechnique.edu, jf@baffier.fr

Abstract

Cost Function Networks (CFN) are a formalism in Constraint Programming to model combinatorial satisfaction or optimization problems. By associating a function to each constraint type to evaluate the quality of an assignment, it extends the expressivity of regular CSP/COP formalisms but at a price of making harder the problem modeling. Indeed, instead of the regular constraints set, one must provide a set of cost functions that are not always easy to define. Here we first propose two clear definitions of Weighted CSP and CFN, separating these in two different problems considered so far to be similar, and we then propose a method to automatically learn a CF of a constraint, given a function deciding if assignments are valid or not. This is to the best of our knowledge the first attempt to automatically learn CFs. Our method aims to learn CFs in a supervised fashion, trying to reproduce the Hamming distance, by using a variation of neural networks we named Interpretable Compositional Networks, allowing us to get explainable results, unlike regular artificial neural networks. We experiment it on 5 different constraints to show its versatility. Experiments show that functions learned on small dimensions scale on high dimensions, outputting a perfect or near-perfect Hamming distance for most constraints. Our system can be used to automatically generate CFs and then having the expressivity of CFN with the same modeling effort than for CSP/COP.

1 Introduction

Twenty years separate Freuder's papers [Fre97] and [Fre18], both about the grand challenges the Constraint Programming (CP) community must tackle, *"to be pioneers of a new usability science and to go on to engineering usability"* [Fre07].

Unlike other paradigms to handle combinatorial problems such as Mixed-Integer Programming, CP lacks a "Model and Run" approach [Pug04, Wal03] to make it more accessible and then more broadly used. We are still far from the original Holy Grail of CP: *"the user states the problem, the computer solves it"* [Fre97]. Some progress has been made though, like in automatic problem modeling [Fre18]: one can cite, among other works, constraint detection described in a natural language [KLT16], automatic constraint model production from an

abstract constraint specification language [AMJ+11], and acquiring constraint networks from examples classified by the user [BKLO17].

This paper contributes to the efforts made by the CP community in automatic problem modeling. We focus on Cost Function Networks (CFN), a formalism in CP, like Constraint Satisfaction Problems (CSP) and Constrained Optimization Problems (COP), with the particularity that it associates a cost function to each constraint type. Where a CSP instance is a network of constraints, *i.e.*, a network of predicates expressing if an assignment satisfies or not each constraint, a CFN instance is a network of cost functions expressing if an assignment satisfies the constraints or, if not, how close it is to satisfy them. Thus, CFN allows us to express a finer structure about the problem: the cost functions network is an ordered structure over invalid assignments a solver can exploit efficiently to improve the search. This is illustrated with Experiment 3 in Section 4.2.

In this paper, we propose a method to automatically learn cost functions, a direction that, to the best of our knowledge, had not been explored by the CP community yet.

2 Preliminaries

In the literature, CFN and Weighted CSP (WCSP) are synonyms [ZGDGS09, BBdG+11]. Some papers like [ATA+12] present CFN to be the formalism and WCSP the problem of finding an assignment minimizing the combined cost function of a given CFN instance. Since it is rarely a good thing in Science to have two different names for the same notion, we start this paper by proposing clear, distinct definitions of CFN and WCSP.

2.1 Definitions of WCSP and CFN

We propose to keep the definition of a WCSP from Lee and Leung [LL12]: a **WCSP** is a tuple (V, D, F) where V is a finite set of variables, D a finite set of domains, one for each variable in V, each domain being the set of values a variable can take, and finally a finite set F composed of cost functions with different scopes $\{x_1, \ldots, x_n\} \subseteq V$. Cost functions replace constraints in classic CSP/COP formalisms.

Let D_f be the Cartesian product of the domain of variables $x_1, \ldots, x_n$ involved in a cost function $f \in F$. Thus a cost function is a function $f : D_f \to \{0, k\}$ where $k \in \mathbb{N} \cup \{\infty\}$ is the special cost for incorrect assignments, *i.e.*, variable assignments violating the constraint expressed by f. Thus, an assignment λ is valid if and only if $f(\lambda) < k$ holds. The function f allows us to rank valid assignments, then expressing a soft constraint.

A **WCSP instance** is then represented by a collection of cost functions in F over variables in V. Let $\vec{x}_f$ be variables in V involved in a given cost function f. An assignment Λ verifies a WCSP instance if and only if, for all cost functions $f \in F$, the projection λ of the assignment Λ on $\vec{x}_f$ is such that $f(\lambda) < k$.

WCSP is an optimization problem where the goal is to find a solution minimizing the sum of the cost functions. Thus, all WCSP instance share the same objective function o over all variables in V defined as

$$o(x_1, \ldots, x_{|V|}) = \bigoplus_{f \in F} (f(\vec{x}_f))$$

with $\oplus$ the operation defined as $\oplus(a, b) = min(k, a + b)$. Since this objective function is invariant (its size changes with the cardinality of F, but its formulation never changes), there is no need to consider it to be part of WCSP instances.

A **CFN** is also defined by a tuple (V, D, F) with the same sets as WCSP. The difference we propose lies in the interpretation of cost functions f. In this paper, cost functions defined in a CFN are functions $f : D_f \rightarrow \mathbb{R}^+$. An assignment λ is valid if and only if $f(\lambda) = 0$ holds. All other strictly positive outputs of f lead to forbidden assignments.

Therefore, unlike WCSP, CFN is considering hard (or crisp) constraints only. Strictly positive outputs of f are then interpreted like **preferences over invalid assignments**: the closer $f(\lambda)$ is to 0, the closer λ is to be a valid assignment for f. This is the fundamental characteristic behind our proposed definition of CFN.

Like for WCSP, a **CFN instance** is represented by a collection of cost functions in F over variables in V. Here, however, an assignment Λ verifies a CFN instance if and only if, for all cost functions $f \in F$, the projection λ of the assignment Λ on $\vec{x}_f$ is such that $f(\lambda) = 0$.

Observe that CFN is then not necessarily an optimization problem: without any given optimization function, CFN is a satisfaction problem where no orderings between solutions exist.

Thus we can also deal with optimization problems with a CFN by considering the tuple (V, D, F, o) with o an objective function to optimize, exactly the same way COP extends CSP.

3 Method design

The main result of this paper is to propose a method to automatically learn a cost function of a constraint, to make the modeling of a combinatorial problem as a CFN easier. We are, in essence, tackling a regression problem, where the goal is to find a function that outputs a target value. Before diving into the description of our method, we need to introduce some essential notions.

Also, we make the difference in this paper between our method, *i.e.*, the proposed methodology and model to learn cost function, and our system, *i.e.*, the implementation of our method.

3.1 Definitions

Since the vocabulary used in fields such as CP and Operational Research can sometimes diverge, we fix here two important definitions used in this paper. A **configuration** of a constraint is an assignment of each variable in the constraint. A configuration may or may not satisfy the constraint. A **solution** is a configuration satisfying the constraint.

We propose a method to automatically learn a cost function of a constraint from the constraint *concept*. Like described in [BKLO17], the **concept** of a constraint is a Boolean function that, given a configuration c, will output *true* or *false* if c satisfies the constraint of not, respectively. Thus, with our method, the only information asked from the user about the constraint he or she wants to learn a cost function from is its associated concept.

Our method learns cost functions in a supervised fashion, searching for a function computing the *Hamming cost* of each configuration. The **Hamming cost** of a configuration c is the minimum number of variables in c to reassign to get a solution. In other words, it is the Hamming distance from c to its closest solution. If c is a solution, then its Hamming cost is 0.

Knowing the number of variables to change to get a solution seems to be useful information to give to the solver, although this is discussed in Section 5.

We need here to introduce the notion of *constraint space*. Given the number of variables and their domains size, the **constraint space** of a constraint instance is the set of couples (c, b) where c is a configuration and b the Boolean output of the concept applied on c. Such constraint spaces are automatically generated from a given concept. Spaces are said to be **complete** if and only if they contain all possible configurations, and **incomplete** otherwise.

We consider a cost function to be a (non-linear) combination of elementary operations. We train a model to learn what combination of operations is fitting best a *training set*. Complete spaces are intuitively good training sets since it is easy to compute the exact Hamming cost of their configuration. We can also consider configurations from incomplete spaces where their Hamming cost has been approximated regarding the subset of solutions in the constraint space, in the case the exact function computing the Hamming cost of a configuration is unknown.

We now have everything we need to introduce our model to learn cost functions.

3.2 Main result

To learn a cost function as a non-linear combination of elementary operations, we build a model inspired by Compositional Pattern-Producing Networks (CPPN). CPPNs, introduced in [Sta07], are themselves a variation of artificial neural network. While neurons in regular neural networks usually contain sigmoid-like functions only (such as ReLU, or Rectified Linear Unit), CPPN's neurons can contain many other kinds of functions: sigmoids, Gaussians, trigonometric functions, and linear functions among others. CPPN are often used to generate 2D or 3D images by applying the function modeled by a CPPN over the space of possible inputs. We build our model by taking these two principles from CPPN: having neurons containing one operation among many possible ones, and taking an input space by considering one by one all its possible elements.

Due to their interpretable nature, we named our variation of artificial neural network **Interpretable Compositional Networks**.

The main idea of our model is to divide it into four layers, each of them having a specific purpose and composed of their neurons, each of them containing a unique operation. All neurons from a layer are linked to all neurons from the next layer. The weight on each link is purely binary: its value is either 0 or 1. A weight between neurons n_1 and n_2 with the value 1 means that the neuron n_2 from layer $l + 1$ will take as input the output of the neuron n_1 from layer l. A weight with the value 0 means that n_2 will discard the output of n_1.

Figure 1 is a schematic representation of our model. It takes as input an n-ary configuration, *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 i-th value of the input. Then, is an operation is selected (*i.e.*, it has an outgoing weight equals to 1), it outputs a vector of n integers. For instance, one of our 18 transformation operations is "*Number of* c_j *such that* $i<j$ *and* $c_i=c_j$", with c_i and c_j respectively the i-th and j-th value of the configuration c. This layer is composed of both linear and non-linear operations.

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 component-wise on all i-th element of our k vectors to get at the end one vector of n integers. We have considering only 2 arithmetic operations so far: the addition and the multiplication.

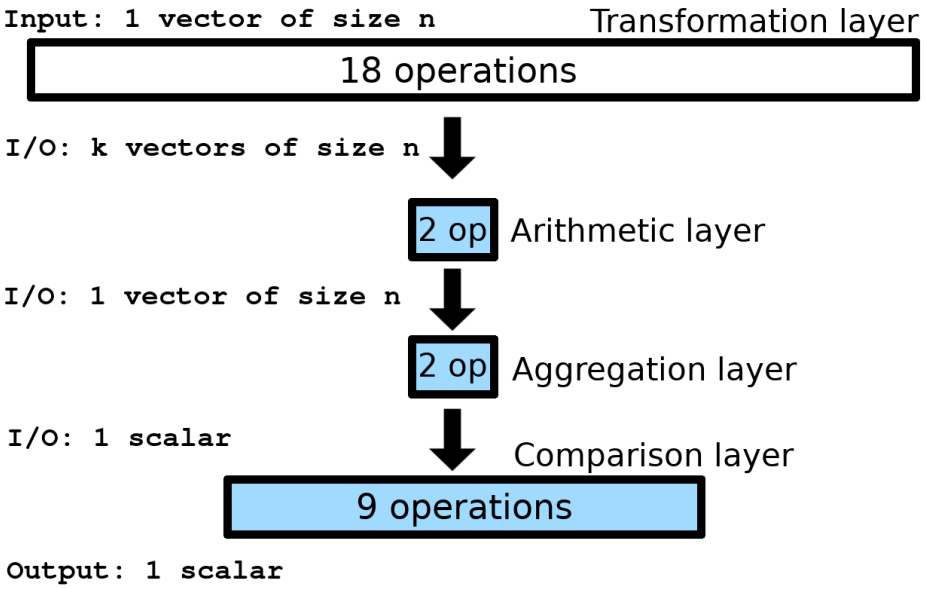

Figure 1: Our 4-layer model. Layers with a blue background have mutually exclusive operations.

The output of the arithmetic layer is given to the **aggregation layer**. Then, this layer crunches the whole vector into a unique integer. At this 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.

Finally, the computed scalar is transmitted to the **comparison layer** with 9 operations. This layer allows comparing the computed value so far with an external parameter value, or the number of variables of the problem, or the domain size, among others. For instance, one comparison operation is the *Euclidian division of the difference between the input and the parameter by the domain size*.

All elementary operations in our model are generic: we do not choose them to fit one or several particular constraints. A comprehensive list of the 18 transformation and 9 comparison operations is available at github.com/richoux/LearningCostFunctions/tree/IJCAI_v1.1/utils/print.cpp Both layers contain the identity function among their operations.

To have simple models of cost functions, operations of the arithmetic, the aggregation, and the comparison layers are mutually exclusive, meaning that exactly one operation is selected for each of these layers. However, many operations from the transformation layer can be selected to compose the cost function. Combined with the choice of having binary weights, it allows us to have a very comprehensible combination of elementary operations to model a cost function: once a cost function is learned, its combination of operations is readable and intelligible by a human being. Thus, once the model of a cost function is learned, the user has the choice to run the network in a feed-forward fashion to compute the cost function, or to re-implement it directly in a programming language. A user can use our system to find cost functions automatically, but he or she can also use it as a decision support system to find promising cost functions that he or she may modify and adapt by hand.

3.3 Learning with Genetic Algorithms

Like any neural network, learning a cost function with our model is to learn the value of its weights. To do so, we use a genetic algorithm. Indeed, using a back-propagation-based algorithm is not be possible since some of our operations are not derivable without easy trick to get around this issue (unlike ReLU, not derivable at 0 but one considers its derivative at 0 is 0).

Since our weights are binary, we represent individuals of our genetic algorithm by a binary vector of size 29, each bit corresponding to one operation in the four layers. Since arithmetic and aggregation layers contain only two mutually exclusive operations, these operations are represented by one bit for each layer. For the transformation and comparison layers, the i-th bit set to 1 means the i-th operation is selected to be part of the cost function. The 9 operations of the comparison layer being also mutually exclusive, the last 9 bits of our chromosomes are such that exactly one is set to 1, the others to 0.

We randomly generate an initial population of 100 individuals and check and fix them, so all of them verify the mutually exclusive constraint of the comparison layer. Then, we run the genetic algorithm to produce 400 generations before outputting its best individual according to our fitness function.

The **fitness function** is the loss function of our supervised learning: the sum, for each configuration of the training set, of the absolute value of the difference between the expected Hamming cost and the Hamming cost computed by the cost function encoded by the individual. Besides, decimal penalties are added in our loss function for each selected transformation operation. Hence it encourages the learning of simple functions, composed of as few transformation operations as possible, and can be seen as a regulation to limit overfitting.

Our genetic algorithm is rather simple: **Selection** is done by a tournament selection between two individuals. **Variation** is done by a one-point crossover operation and a one-flip mutation operation, both crafted to always produce new individuals verifying the mutually exclusive constraint of the comparison layer. The crossover rate is fixed at 0.8, and exactly one bit is mutated for each selected individual. **Replacement** is done by an elitist merge, keeping 5% of the best individuals from the old generation into the new one, and a deterministic tournament truncates the new population to 100 individuals. We use the framework EVOLVING OBJECTS [KMRS02] to code our genetic algorithm.

4 Experiments

To show the versatility of our method, we tested it on five very different constraints: AllDifferent, Ordered, LinearSum, NoOverlap1D, and Minimum. According to XCSP specifications [BLAP16] (see also xcsp.org/specifications) , those global constraints belong to four different families: Comparison (AllDifferent and Ordered), Counting/Summing (LinearSum), Packing/Scheduling (NoOverlap1D) and Connection (Minimum). Always according to XCSP specifications, these five constraints are among the twenty most popular and common constraints. We give a brief description of those five constraints below:

AllDifferent ensures that variables must all be assigned to different values.

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 indexes $i, j \in \{1, n\}$, $i < j$ implies $x_i \leq x_j$.

LinearSum ensures that the equation $x_1 + x_2 + \ldots + x_n = p$ holds, with the parameter p a given integer.

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 indexes $i, j \in \{1, n\}$ 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.

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.

4.1 Experimental protocols

To have experimental evidence of the efficiency of our system, we conducted three different experiments.

All experiments have been done on a computer with a Core i7 6700K CPU and 48GB of RAM, running on Ubuntu 18.04. Programs have been compiled with GCC with the 03 optimization option. Our entire system, its C++ source code, experimental setups and results files are accessible at github.com/richoux/LearningCostFunctions/tree/IJCAI_v1.1/.

4.1.1 Experiment 1: scaling

The first experiment consists in learning cost functions on a small, complete constraint space, composed of about 500 configurations. It is then possible to compute the Hamming distance between each configuration with its closest solution. The goal of this experiment is to show that learned cost functions scale to high-dimensional constraints, making sufficient the use of our system on small constraint instances to get efficient cost functions on any number of variables.

We run 100 cost function learnings on the same complete constraint space, for each of the five constraints presented above. We then analyze the frequency of cost functions we get and compute the errors of the most frequent ones, on a test set of 100 sampled configurations (containing 25 solutions) composed of 100 variables on domains of size 100, belonging to constraint spaces of size $100^{100} = 10^{200}$ (compare to constraint spaces of size around 500 used to learn cost functions).

4.1.2 Experiment 2: learning over incomplete spaces

If for any reason, it is not possible to build a complete constraint space, a robust system must be able to learn effective cost functions on large, incomplete spaces where the exact Hamming cost of their configurations is unknown.

In this experiment, we sample 100 solutions and 100 non-solutions on large constraint spaces, approximate the Hamming cost of each non-solution by computing their Hamming distance with the closest solution among the 100 ones, and learn cost functions on these 200 configurations and their estimated Hamming cost. Then, we evaluate the most frequently learned cost function for each constraint over the same test sets than Experiment 1.

4.1.3 Experiment 3: using learned CFs to solve problems

The goal of this experiment is to assess that learned cost function can effectively be used to solve problems.

We use a local search solver to solve Sudoku and consider the mean and median run-time, as well as other metrics, to compare a pure CSP model (so without cost functions), a CFN model with the most frequently learned cost function from Experiment 1 and run through our neural network, a CFN model with the same cost function but directly hard-coded in C++ and a CFN model with an efficient hand-crafted cost function.

Sudoku is a puzzle game presented like a 9×9 grid where each cell of the grid must be filled up with a number from 1 to 9, such that each row and each column contains precisely each number once. Besides the grid is composed of 9 smaller squares of size 3×3, which must also be filled with each number exactly once. In other words, all numbers of each row, column, and square must be different, which is correctly modeled by the AllDifferent constraint. Usually, the grid is pre-filled with a few numbers, preventing from finding a trivial solution. In this paper, to have randomly generated Sudoku instances, we have pre-filled the entire grid randomly with the expected total number of 1s, 2s, etc., and ask the solver to find a permutation satisfying all AllDifferent constraints described above.

4.2 Results

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

4.2.1 Experiment 1

As written in Section 3, our loss function is the sum of the absolute value of the difference between the expected Hamming cost of a configuration c and the Hamming cost estimated for the cost function on c. The loss function is then normalized with the size of the constraint space used for training, giving us the training error of the constraint space, *i.e.*, the average difference between expected and estimated Hamming costs. Thus, a cost function f with a training error of 2 means that f estimations on configurations used for training are on average +2 or -2 from the real Hamming cost.

In this experiment, we learn 100 times a cost function for each constraint instance. Table 1 shows for each constraint instance the median and mean training errors of the 100 learned cost functions, as well as the training error of the most frequently learned cost function, and its frequency in parenthesis. In this experience, the most frequently learned cost function was systematically the one with the lowest training error.

Table 2 gives the most frequent cost functions learned on small complete constraint spaces for each constraint instance, showing that cost functions outputted by our system are readable

Constraints	median	mean	most freq.
all_different-4-5	0	0.03	0 (97)
ordered-4-5	0.08	0.08	0.08 (100)
linear_sum-3-8-12	0.01	0.05	0.01 (74)
no_overlap-3-8-2	0.14	0.19	0.11 (50)
minimum-4-5-3	0	0.04	0 (88)

Table 1: Median, mean and most frequent training error over 100 runs (with frequency in parenthesis) of learned cost functions over small complete constraint spaces.

Constraints	Most frequent cost function
all_different-4-5	$Count_{>0}(\ \lvert\{c_j : i < j\ \wedge\ c_i = c_j\}\rvert\ )$
ordered-4-5	$Count_{>0}(\ Max(0,\ c_i\ -\ c_{i+1})\ )$
linear_sum-3-8-12	$1 + \frac{abs\left(\sum(c_i) - p\right)}{\lvert D\rvert}$
no_overlap-3-8-2	$1 + \frac{abs\left(\sum(\lvert\{c_j : c_j < c_i + p\}\rvert \times \lvert\{c_j : c_j \geq c_i\ \wedge\ c_j \leq c_i + p\}\rvert) - p\right)}{\lvert D\rvert}$
minimum-4-5-3	$Count_{>0}(\ Max(0,\ p\ -\ c_i)\ )$

Table 2: Most frequent cost function found for each constraint over small complete constraint spaces, with c_i the i-th element of a configuration, p a given parameter, $|D|$ the domain cardinality and abs the absolute value.

and intelligible for a human being. These cost functions correspond to the last column of Table 1. Notice that the most frequent cost function for all_different-4-5, learned 97 over 100 runs, is actually two equivalent functions f_1 and f_2 expressed differently, learned 58 and 39 times, respectively. For a configuration $c = (c_1, c_2, \ldots, c_n)$, f_1 and f_2 are defined as follows:

$$f_1(c) := Count_{>0}(\ \lvert\{c_j : i < j\ \wedge\ c_i = c_j\}\rvert\ )$$
$$\textit{and}$$
$$f_2(c) := Count_{>0}(\ \lvert\{c_j : i > j\ \wedge\ c_i = c_j\}\rvert\ )$$

Learning cost functions over small complete constraint spaces of about 500 configurations takes about 10 seconds on our hardware.

Table 1 shows good performances, but it might be due to overfitting on those small constraint spaces. To check if learned cost functions do not overfit and can scale to constraint instances on higher dimensions, we use the most frequent cost function learned on each constraint for estimating the Hamming cost of 20,000 random configurations sampled from high-dimensional constraint spaces. Those sets are our test sets. Notice we do not have validation sets, *i.e.*, sets to fix the values of the parameters of our genetic algorithm before a final evaluation on test sets since we did basic parameter tunings on training sets only.

For AllDifferent, LinearSum and Minimum, it is easy to define by hand a function computing the Hamming cost of any configuration c without generating the whole constraint space. For these constraints, we tested the corresponding cost function on spaces with 100 variables and domains of size 100.

all_diff	ord	lin_sum	no_ol	min
0	1.27	0.03	2.68	0

Table 3: Mean error over 20,000 configurations in high dimensions of learned cost functions over small complete constraint spaces.

Constraints	median	mean	most freq.
all_different-6-6	0.44	0.44	0.44 (99)
ordered-6-6	0.44	0.46	0.44 (66)
linear_sum-6-6-12	2.03	1.70	0.85 (37)
no_overlap-6-18-2	2.33	2.39	2.29 (48)
minimum-6-6-3	0.59	0.59	0.59 (78)

Table 4: Median, mean and most frequent training error over 100 runs (with frequency in parenthesis) of learned cost functions over incomplete constraint spaces.

For Ordered and NoOverlap1D, since these two constraints are intrinsically combinatoric, finding a function computing the exact Hamming cost of any configuration is not trivial. Therefore, using Latin hypercube sampling to have a good diversity of configurations, we sampled 10,000 solutions and 10,000 non-solutions in constraint spaces of ordered-12-18 (so 18^{12} configurations, *i.e.*, about 1.15×10^{15}) and no_overlap-10-35-3 ($35^{10} \simeq 2.75 \times 10^{15}$ configurations). Then we approximate the Hamming cost of each non-solution, considering the closest solution among the 10,000 sampled solutions.

Table 3 presents the mean error of the most frequently learned cost function for each constraint type, over 20,000 configurations sampled from constraint instances previously introduced. The perfect score of 0 for AllDifferent and Minimum shows that our system has been able to learn the exact Hamming cost over a small constraint space of about 500 configurations. For LinearSum, the cost function only has a total error of 758 over 20,000 configurations, giving a mean error of 0.03 over one configuration. As written previously, we only choose generic operations in our neural network. Describing accurately the Hamming cost for LinearSum requires a particular operation: computing the difference of the smallest value among variables with the highest value in the domain (or the opposite), test if this difference is sufficient to reach the expected sum, and if not, iterate with the second (and third, and so forth) smallest value among variables.

Ordered and NoOverlap1D do not show such good results. For Ordered, a mean error of 1.27 on configurations with 12 variables is still honorable: it means that on average, the difference between the expected and estimated Hamming cost over 12 variables is a bit more than one variable. However, the mean error of 2.68 for NoOverlap1D, considering the constraint instance has 10 variables, is not so good. NoOverlap1D is certainly the most intrinsically combinatoric over our 5 constraints, partly explaining why it is harder to learn a correct cost function for it. One could think that our system is overfitting the training constraint space for NoOverlap1D, but results for Experiment 2 show it is not the case.

4.2.2 Experiment 2

To test if our system can learn efficient cost functions over incomplete constraint space, we learned 100 times a cost function over constraint instances listed in Table 4.

all_diff	ord	lin_sum	no_ol	min
0	1.80	0.03	2.02	0

Table 5: Mean error over 20,000 configurations in high dimensions of learned cost functions over incomplete constraint spaces.

CF	mean	median	std dev	min	max
no CF	1044	764	727	250	3546
learned	383	331	268	57	1812
hard-coded	175	145	107	46	662
hand-crafted	149	125	107	26	608

Table 6: Run-times metrics in milliseconds over 100 runs to solve Sudoku with a local search solver using no cost functions (pure CSP), the most frequently learned cost function for AllDifferent in Experiment 1 run through the interpretable compositional network, the same function but hard-coded in C++, and a hand-crafted cost function.

At first glance, results in Table 4 seam not as good as results from Table 1. However, since we are dealing with incomplete constraint spaces here, *i.e.*, with missing configurations and solutions, the Hamming cost of each configuration is approximated. This approximation is voluntarily very rough, since we only performed a Latin hypercube sampling of 100 solutions and 100 non-solutions in these constraint spaces, giving training sets of 200 elements only when full constraint spaces contain 46,656 configurations (except for no_overlap-6-18-2 with 34,012,224 configurations).

To have a better estimation of the efficiency of cost functions learned on these incomplete spaces, we need to evaluate them on the same test sets used for Experiment 1.

Table 5 confirms the robustness of our system learning cost function on incomplete constraint spaces: the most frequently learned cost functions for AllDifferent, LinearSum, and Minimum are the same as in Experiment 1.

We made further investigations for Ordered and NoOverlap1D, learning cost functions over larger constraint instances, sampling either 200 or 2000 configurations. Instances were ordered-6-9, 9-9, 9-13, 12-12 and 12-18, and no_overlap-6-18-2, 6-24-3, 9-27-2, 9-36-3, 12-36-2 and 12-48-3. Results for Ordered show improvement of the mean test error and strong stability for NoOverlap1D with a mean error of around 2.2. Hence this leads us to think that our system does not suffer from overfitting for these constraints (since mean test errors would have been worst otherwise), but that our model is not able to learn a good cost function with the current set of elementary operations.

4.2.3 Experiment 3

The goal of this experiment is not to be state-of-the-art in terms of run-times for solving Sudoku, but to compare the average run-times of the same solver on four nearly identical Sudoku models presented in Section 4.1.3. For the model with a hand-crafted cost function, we implemented the *primal graph based violation cost* of AllDifferent from [PRB01]. This function simply outputs the number of couples with identical values within a given configuration. To run this experiment, we used the framework GHOST [RUB16], which includes a local search algorithm able to handle both CSP and CFN models.

Table 6 shows that cost functions are well exploited by the solver. We run 100 solving of Sudoku for each model and compute the mean and median run-time in milliseconds, as well as the standard deviation, the shortest and the longest run-time: run-times from cost function-based models are significantly better than the pure CSP model ones. We can also estimate the overload of computing the cost function through the interpretable compositional network (ran in a feed-forward fashion), compare to a hard-coded version of the same cost function. We recall that one advantage of our method is to output intelligible cost functions, letting the choice to the user to compute this function through the interpretable compositional network or to let him/her the possibility to code it himself/herself. Results from Table 6 show that the overload is such that run-times of cost functions executed through the interpretable compositional network are slightly more than twice longer than run-times of their hard-coded version.

More importantly, we see that the most frequently learned cost function trying to reproduce the Hamming cost finds solutions almost as quickly as the carefully hand-crafted cost function from [PRB01]. Thus, our method can be used to automatically find cost functions with an equivalent efficiency than human-defined ones.

5 Discussions and conclusion

In this paper, we present a method to learn cost functions automatically, given a constraint concept only, upon a model based on interpretable compositional networks, an original variation of neural networks. To the best of our knowledge, this is the first attempt to learn cost functions automatically.

We have tested our system over 5 different constraints. It finds the perfect cost function (in our case, the Hamming cost) for 2 of those constraints (AllDifferent and Minimum), and a near-perfect cost function for 1 constraint (LinearSum). For these 3 constraints, cost functions learned over a small, complete constraint space (about 500 configurations) perfectly scale on high-dimension constraint instances (10^{200} configurations). We show the robustness of our system by learning cost functions over incomplete constraint space (200 configurations belonging to spaces of more than 46,000 configurations), and it can find the same cost functions learned on small, complete constraint spaces, leading to the same performances on high-dimension constraint instances.

One of the most significant results in this paper is that our system outputs interpretable results, unlike regular artificial neural networks. Cost functions outputted by our system are intelligible. This allows our system to have operating modes: 1. a fully automatic system, where cost functions are learned and called within our system, being completely transparent to the user who only needs to furnish a concept function for each constraint, to the regular sets of variables V and domains D, and 2. a decision support system, where the user can look at a set of proposed cost functions, pick up and modify the one he or she prefers.

We made this system modular and easy to modify. Thus, an experienced user with special needs can add or remove operations in the system to learn more specific cost functions.

The current limitation of our system is that it struggles to learn high-quality cost function for very combinatorial constraints, such as Ordered and, in particular, NoOverlap1D. By combining results from Experiments 1 and 2, we can conclude that our system is not overfitting but need more diverse and expressive operations to learn a high-quality cost function for such constraints.

An extension of our work would be to do reinforcement learning rather than supervision learning based on the Hamming cost. Indeed, even if the Hamming cost seems to be a natural metric to tell how far a configuration is to be a solution in most cases, it could also be too restrictive, like stressed by [PRB01]. Learning via reinforcement learning would allow finding cost functions that are more adapted to the chosen solver.

References

[AMJ+11] Özgür Akgün, Ian Miguel, Christopher Jefferson, Alan Frisch, and Brahim Hnich. Extensible automated constraint modelling. In *25th Conference on Artificial Intelligence (AAAI'11)*, pages 4–11, 2011.

[ATA+12] David Allouche, Seydou Traoré, Isabelle André, Simon De Givry, George Katsirelos, Sophie Barbe, and Thomas Schiex. Computational protein design as a cost function network optimization problem. In *International Conference on Principles and Practice of Constraint Programming (CP'12)*, pages 840–849, 2012.

[BBdG+11] Christian Bessière, Patrice Boizumault, Simon de Givry, Patricia Gutierrez, Samir Loudni, Jean-Philippe Métivier, and Thomas Schiex. Decomposing global cost functions. In *11th Workshop on Preferences and Soft Constraints (Soft'11) of the International Conference on Principles and Practice of Constraint Programming (CP'11)*, pages 16–30, 2011.

[BKLO17] Christian Bessière, Frederic Koriche, Nadjib Lazaar, and Barry O'Sullivan. Constraint acquisition. *Artificial Intelligence*, 244:315–342, 2017.

[BLAP16] Frederic Boussemart, Christophe Lecoutre, Gilles Audemard, and Cédric Piette. XCSP3: An Integrated Format for Benchmarking Combinatorial Constrained Problems. *arXiv e-prints*, 2016.

[Fre97] Eugene C. Freuder. In pursuit of the holy grail. *Constraints*, 2(1):57–61, 1997.

[Fre07] Eugene C. Freuder. Holy grail redux. *Constraint Programming Letters*, 1:3–5, 2007.

[Fre18] Eugene C. Freuder. Progress towards the holy grail. *Constraints*, 23(2):158–171, 2018.

[KLT16] Zeynep Kiziltan, Marco Lippi, and Paolo Torroni. Constraint detection in natural language problem descriptions. In *25th International Joint Conference on Artificial Intelligence (IJCAI'16)*, pages 744–750, 2016.

[KMRS02] Maarten Keijzer, J. J. Merelo, G. Romero, and M. Schoenauer. Evolving Objects: A General Purpose Evolutionary Computation Library. *Artificial Evolution*, 2310:829–888, 2002.

[LL12] J. Lee and Mario Leung, Ka Lun. Consistency Techniques for Flow-Based Projection-Safe Global Cost Functions in Weighted Constraint Satisfaction. *Journal of Artificial Intelligence Research*, 43:257–292, 2012.

[PRB01] Thierry Petit, Jean-Charles Régin, and Christian Bessière. Specific filtering algorithms for over-constrained problems. In *International Conference on Principles and Practice of Constraint Programming (CP'01)*, 2001.

[Pug04] Jean-François Puget. Constraint programming next challenge: Simplicity of use. In *International Conference on Principles and Practice of Constraint Programming (CP'04)*, pages 5–8, 2004.

[RUB16] Florian Richoux, Alberto Uriarte, and Jean-François Baffier. GHOST: A combinatorial optimization framework for real-time problems. *IEEE Transactions on Computational Intelligence and AI in Games*, 8(4):377–388, 2016.

[Sta07] Kenneth O. Stanley. Compositional pattern producing networks: A novel abstraction of development. *Genetic Programming and Evolvable Machines*, 8(2):131–162, 2007.

[Wal03] Mark Wallace. Languages versus packages for constraint problem solving. In *International Conference on Principles and Practice of Constraint Programming (CP'03)*, pages 37–52, 2003.

[ZGDGS09] Matthias Zytnicki, Christine Gaspin, Simon De Givry, and Thomas Schiex. Bounds arc consistency for weighted CSPs. *Journal of Artificial Intelligence Research*, 35:593–621, 2009.