CompositionalNetworks.jl: a scaling glass-box neural network to learn combinatorial functions

Jean-François BAFFIER[1], Khalil CHRIT[2], Florian RICHOUX[3,4], Pedro PATINHO[2], and Salvador ABREU[2]

[1]IIJ, Japan
[2]NOVA-LINCS, University of Évora, Portugal
[3]AIST, Japan
[4]JFLI, CNRS, Japan

ABSTRACT

Interpretable Compositional Networks (ICNs) are a neural network variant for combinatorial function learning that allows the user to obtain interpretable results, unlike ordinary artificial neural networks. An ICN outputs a composition of functions that scales with the size of the input, allowing a learning phase on relatively small spaces. CompositionalNetworks.jl is a pure Julia package that exploits the language's meta-programming, parallelism and multiple dispatch features to produce learned compositions in mathematical and programming languages such as Julia, C or C++.

Keywords

Julia Language, Constraint Programming, Local Search, Metaheuristics, Neural Network, Metaprogramming, Scalable Machine Learning, Glass-Box Algorithm

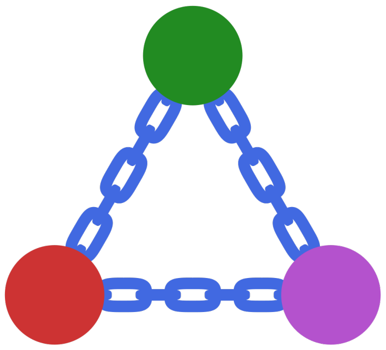


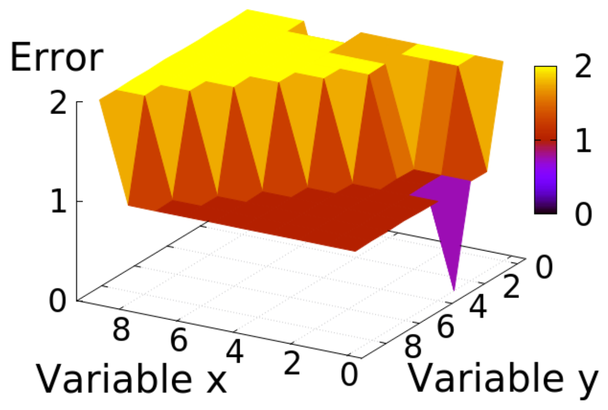
Fig. 1: Logo of the JuliaConstraints organization on GitHub that hosts, among other things, the CompositionalNetworks.jl package.

1. Introduction

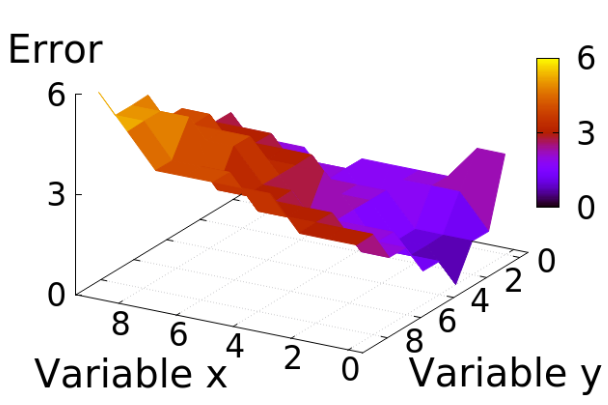
The discipline of combinatorial optimization consists in finding an optimal configuration of elements within a finite set. Such a set is usually subject to constraints that can be represented by functions that are often highly combinatorial. These constraints are mostly formulated as concepts, boolean functions indicating whether each constraint is respected or not. A solution (sometimes called a satisfying solution, depending on the field) is a configuration that respects all the constraints.
Different domains such as operational research, constraint programming, metaheuristics, propose methods to find (satisfactory or optimal) solutions. Among these methods, some can or could benefit from a finer granularity in the evaluation of the impact of each constraint on a given configuration.
In [10], we introduced Interpretable Compositional Networks (ICNs), a neural network variant for combinatorial function learning that allows the user to obtain interpretable results, unlike ordinary artificial neural networks. An ICN outputs a composition of functions that scale with the size of the input, allowing a learning phase on relatively small spaces.
This work consists in a library that implements ICNs in the Julia programming language [3]. Although we present a direct application of ICNs in the following subsection, this neural network framework is simple to extend to learn other combinatorial and non-combinatorial functions.

1.1 Application to Constraint Programming

Constraint Satisfaction Problem (CSP) and Constrained Optimization Problem (COP) are constraint-based problems where constraints can be seen as predicates (*concepts*) allowing or forbidding some combinations of variable assignments. Such a formulation corresponds well to the so-called complete solution methods which guarantee an exploration of all solutions. Unfortunately, due to the highly combinatorial nature of some problems, these methods cannot always converge in a reasonable time.
On the other hand, Constraint-Based Local Search (CBLS) is a family of metaheuristics in which the neighborhood is constructed on the basis of an *error function*, itself a quantitative representation of how far the current configuration is from an admissible solution. We refer to the corresponding problems as Error Function Satisfaction Problem (EFSP) and Error Function Optimization Problem (EFOP). In the case of CBLS, this is computed as a function of the constraints which are not currently satisfied. Error functions may be derived from the constraint satisfaction problem structure, hand-coded, automatically acquired by some machine learning process, or constructed as a combination of these methods. It should be noted that, the best performing systems resort to hand-tuned error functions, as witness [5].
As illustrated by Figure 2, a well-designed error function helps in converging faster towards better-quality solutions. It does, however, introduce an additional layer of complexity to the user.



(a) CSP landscape



(b) EFSP landscape

Fig. 2: Comparison of a Constraint Satisfaction Problem (CSP) and an Error Function Satisfaction Problem (EFSP) landscapes. The finer scale in the error heuristic of the EFSP leads to a better convergence rate.

1.2 Interpretable Compositional Networks (ICN)

In [10], we proposed a neural network inspired by Compositional Pattern-Producing Networks (CPPN) to learn (highly) combinatorial functions as non-linear combinations of elementary operations. CPPNs [12] are themselves a variant of artificial neural networks. While neurons in regular neural networks usually contain sigmoid-like functions only (such as ReLU, *i.e.* Rectified Linear Unit), CPPN's neurons can contain many other kinds of functions: sigmoids, Gaussians, trigonometric functions, and linear functions among others. CPPNs are often used to generate 2D or 3D images by applying the function modeled by a CPPN giving each pixel individually as input, instead of considering all pixels at once. This simple trick allows the learned CPPN model to produce images of any resolution.

We propose our variant by taking these two principles from CPPN: having neurons containing one operation among many possible ones, and handling inputs in a size-independent fashion. Due to their interpretable nature, we named our variant **Interpretable Compositional Networks** (ICN).

Although ICNs are not limited to learning function for EFSP/EFOP, the original structure was designed to learn compositions weighted in accordance to the hamming metric [10]. The hamming distance between a configuration and its closest solution is a meaningful indicator of the number of variables needed to be changed. Manhattan and other variants of minkowski comparison are other example of possible metrics.

An ICN is made of several layers of (possibly exclusive) operations such that the first layers accept a vector as input and the last layer returns a single numerical value. Generally, layers alter there input in three possible ways: *increment*, *decrement*, or *conservation* of the dimension of the input.

In the context of EFSP/EFOP, the output should be non-negative if the constraint is violated and equal to 0 otherwise. As shown by Figure 3, our ICNs are composed of four layers, each of them having a specific purpose and themselves composed of neurons applying a unique operation each. 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. This restriction is crucial to obtain interpretable functions. We refer the reader to [10] for a more comprehensive definition.

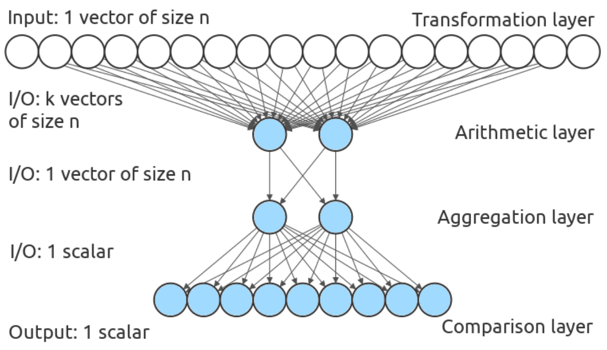


Fig. 3: Scheme of the basic 4-layers ICN model used in EFSP/EFOP. Layers with blue neurons have mutually exclusive operations. the *transformation* layer increases the input dimension, the *arithmetic* and *aggregation* layers reduces it, and the *comparison* layer leaves it untouched. The figure is taken from [10].

In [10] we introduced the concept of ICN and a first implementation for Constraint Programming as a C++ library[1]. The results were evaluated through the GHOST C++ solver [11] and serve as a proof of concept that most of the models give scalable functions, and remain fairly effective using incomplete training sets.

As mentioned in Section 4, CompositionalNetworks.jl generates code for a direct use in Julia, but also exports usable code for both the GHOST(C++) and AdaptiveSearch (C)[2]. The code exported in C++, that is the *operations* of each neuron, is strongly inspired by our C++ library.

This work is part of a collection of articles that introduce the theoretical background of ICN [10], a user-friendly implementation in Julia, and an extensive collection of benchmarks [1]. Please note that the default parameters, and semi-automated learning parameters for advanced users, in CompositionalNetworks.jl will improve along the results in [1] computed from ICNBenchmarks.jl on an HPC cluster.

[1]`https://github.com/richoux/LearningUtilityFunctions`

[2]This feature is a WIP at the moment this article is submitted, but is expected to be completed in the coming weeks.

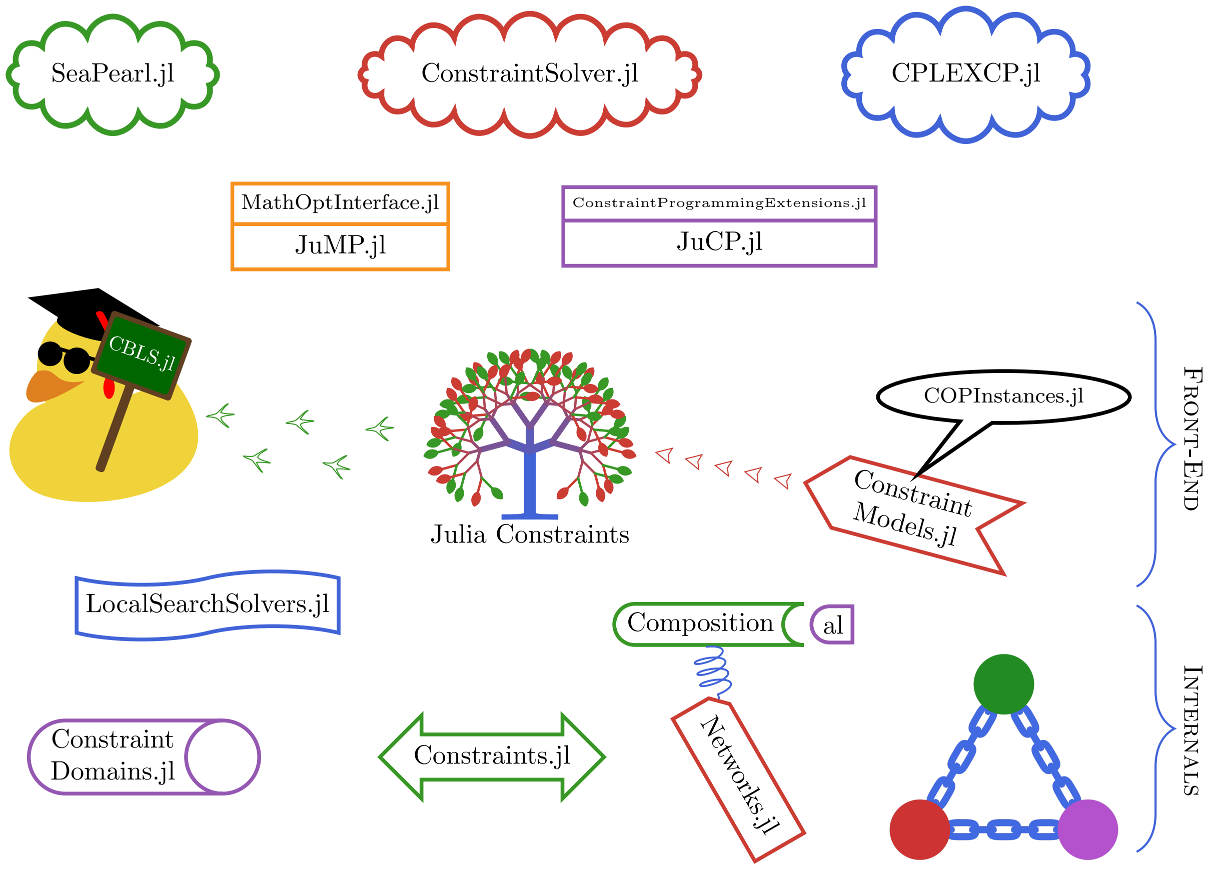


Fig. 4: Overview of the Constraint Programming ecosystem in Julia, including JuliaConstraints. *Front-End* and *Internals* refer to the latest. Note that, at the moment of the writing, `ConstraintProgrammingExtension.jl`, `JuCP.jl`, and `COPInstances.jl` are temporary names.

2. The example of the JuliaConstraints framework

The CompositionalNetworks.jl package is part of the JuliaConstraints GitHub organization that proposes a first collection of Julia packages for Constraint-Based Local Search (CBLS), a subfield of Constraint Programming. All packages fall under the MIT license. The main goal of this framework is to provide a set of high level semi-automatic tools which strike a good compromise between efficiency and ease of modeling.

2.1 History of JuliaConstraints

The LocalSearchSolvers.jl package is a Constraint-Based Local Search framework started in Fall 2020 and inspired by other CBLS solvers such as GHOST (C++) and AdaptiveSearch (C) that allows users to tune their own solver.
During the development of LocalSearchSolvers.jl, we decided to split the code and functionality of the original framework into several independent packages for ease of maintenance and in hopes of providing common tools for other constraint programming packages in Julia. These tools were collected into the JuliaConstraints ecosystem.
The global state of the Constraint Programming ecosystem is presented in Figure 4. Beside the CBLS framework of JuliaConstraints detailed below, other solvers are complete search methods, such as ConstraintSolver.jl[3], CPLEXCP.jl[4], and SeaPearl.jl [4].

State of the packages in JuliaConstraints

—Stable packages
  —CompositionalNetworks.jl: internal tool to provide semi-automated error functions. Also, a standalone to learn compositions for combinatorial functions[5]

—Beta packages (usable with frequent breaking changes)
  —ConstraintDomains.jl: creation of discrete, continuous, and arbitrary domains
  —Constraints.jl: generation of constraints from a boolean concept or an error function. Also, list usual constraints and their properties
  —LocalSearchSolvers.jl: A CBLS framework in Julia with built-in parallel and distributed scalability
  —CBLS.jl: A MOI/JuMP wrapper for LocalSearchSolvers
  —ConstraintModels.jl: list of CP models for CBLS.jl and LocalSearchSolvers.jl

[3]`https://github.com/Wikunia/ConstraintSolver.jl`
[4]`https://github.com/dourouc05/CPLEXCP.jl`
[5]At the time of submission, CompositionalNetworks.jl is not yet release as `v1`. Some minor changes are expected from the results of the benchmarks in [1] that will be submitted at the end of October.

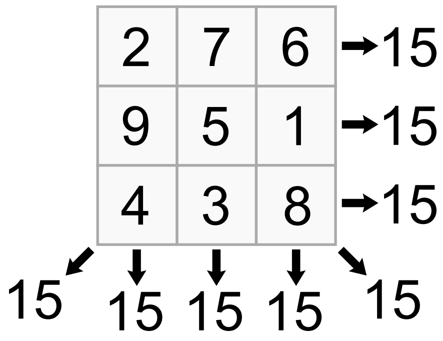


Fig. 5: A solved magic-square of size 3. The *magic sum* is equal to 15. Image from Wikipedia's user PHIDAUEX.

2.2 The *JuMP* ecosystem

JuMP is a modeling language accessible through its main package JuMP.jl [6] and many supporting Julia packages. Beneath JuMP is an abstraction layer called MathOptInterface (MOI) [8]. One can make a parallel between MOI/JuMP and FlatZinc/MiniZinc [9], common modeling tools of the Constraint Programming community. Recently, the ConstraintProgrammingExtensions.jl package that extends MOI/JuMP to the Constraint Programming world was made available. It is likely to become the recommended unified JuMP modeling extension for the Julia Constraint Programming solvers.

2.3 A simple example with `CBLS.jl`

Our CBLS framework uses three levels of modeling syntax: a *raw* Julia syntax, and both MOI and JuMP syntaxes. In this article, we will only use the highest level one, the JuMP syntax.
A Magic Square of order n is composed of n^2 distinct values ranging from 1 to n^2 layed out as a square array X. These values need to be arranged such that the sums of each diagonal, row and column be equal to the same value, the magic sum Σ.

$$\sum_{j=1}^{n} X_{ij} = \sum_{i=1}^{n} X_{ij} = \sum_{j=1}^{n} X_{ii} = \sum_{j=1}^{n} X_{i,n-i+1} = \frac{n(n^2+1)}{2} = \Sigma$$

In ConstraintModels.jl, we use the following code to generate a magic-square instance.

```julia
# Import CBLS.jl and JuMP.jl
using CBLS, JuMP

function magic_square(n::Integer)
model = Model(CBLS.Optimizer)
N = n^2
Σ = n * (N + 1) / 2
@variable(model, 1 <= X[1:n, 1:n] <= N, Int)
@constraint(model, vec(X) in AllDifferent())
for i in 1:n
  @constraint(model, X[i,:] in Linear(Σ))
  @constraint(model, X[:,i] in Linear(Σ))
end
@constraint(model,
  [X[i,i] for i in 1:n] in Linear(Σ)
)
@constraint(model,
  [X[i,n+1-i] for i in 1:n] in Linear(Σ)
)
  return model, X
end
```

3. A flexible implementation

As mentioned in Subsection 1.2, CompositionalNetworks.jl extends the work of the first C++ library used for prototyping in [10]. We decided to use some features of the Julia language to ease provisioning flexibility and broader features to ICNs.

3.1 The Julia language

The Julia language was introduced in [3] and purports to be a new take on *mathematical programming*. Some key features of the language, according to the official website, which led us to consider it as an implementation vehicle:

Fast: Julia was designed from the beginning for high performance. Julia programs compile to efficient native code for multiple platforms via LLVM [7].

Dynamic: It is dynamically typed, feels like a scripting language, and has good support for interactive use.

Reproducible: Reproducible environments make it possible to recreate the same Julia environment every time, across platforms, with pre-built binaries.

Composable: It uses multiple dispatch as a paradigm, making it easy to express many object-oriented and functional programming patterns.

General-purpose: It provides metaprogramming, asynchronous I/O, debugging, logging, profiling, a package manager, and more. One can build entire Applications and Microservices in Julia.

Open source: It is an open source project with over 1,000 contributors. It is made available under the MIT license and the source code is available on GitHub.

3.2 Layers of operations

A layer is defined by a non-empty collection of (possibly mutually exclusive) operations. Our basic ICN (for EFSP/EFOP) is composed of four layers: *transformation*, *arithmetic* (exclusive), *aggregation* (exclusive), and *comparison* (exclusive). An illustration is given as Figure 3. The `Layer` structure stores these operations.

```julia
struct Layer
  functions::LittleDict{Symbol, Function}
  exclusive::Bool
end
```

The most complex layer is probably the *transformation* layer. An operation (neuron) of this layer is defined over a vector of values and an optional parameter. As it is often simpler to define a transformation as a function over a specific index of the input vector, we provide meta-programming utility functions to generate vectorized methods: `lazy` and `lazy_param`.

```julia
# Transformation defined for a vector
tr_identity(x; param=nothing) = identity(x)

# Transformation defined for the index of a vector
tr_count_eq(i, x; param=nothing) =
  count(y -> x[i] == y, x) - 1
tr_count_eq_param(i, x; param) =
  count(y -> y == x[i] + param, x)

# Generating vectorized versions
lazy(tr_count_eq)
lazy_param(tr_count_eq_param)
```

When a concept is provided to an ICN, the user can provide an optional parameter input with the keyword argument `param`. A layer will be generated with or without the parametric operations (neurons) based on `param`'s value. Follow a shortened version of the transformation layer generator.

```julia
function transformation_layer(param=false)
  transformations = LittleDict{Symbol,Function}(
    :identity => tr_identity,
    :count_eq => tr_count_eq,
    # ...
  )
  if param
    tr_param = LittleDict{Symbol, Function}(
      :count_eq_param => tr_count_eq_param,
      # ...
    )
    transformations = LittleDict(
      union(transformations, tr_param)
    )
  end
  return Layer(transformations, false)
end
```

All methods required for the definition of additional layers are all available. An (unweighted) ICN is simply generated from a sequence of layers. After a learning phase using a genetic algorithm from `Evolutionary.jl`[6], this ICN can output a composition in various forms.

3.3 Composition

Once the ICN is weighted, we store its output as a composition with three forms. A collection of `Symbols` per layer of the ICN that allows the generation of code as a pre-process in any language, assuming the operations encoding in that language are available. CompositionalNetworks.jl currently supports Julia, C, and C++.
A composition also stores a Julia object of type `Function` to apply the composition in a dynamic fashion.

```julia
struct Composition{F<:Function}
  code::Dict{Symbol,String}
  f::F
  symbols::Vector{Vector{Symbol}}
end
```

As mentioned above, an ICN outputs a composition, *i.e.* a mathematical function composed of several basic operations. The `code` function returns the definition of a composition in either a mathematical or a programming language. From `v1` of CompositionalNetworks.jl, `code` accepts `:maths` (default), `:Julia`, `:C`, `:Cpp` as values for the `lang` keyword argument.

```julia
function code(
  c::Composition, lang=:maths;
  name="composition"
)
  return get!(
    c.code, lang, generate(c, name, Val(lang))
  )
end
```

[6]`https://github.com/wildart/Evolutionary.jl`

4. How to use CompositionalNetworks.jl

This section covers the essential steps to use ICN in a Julia environment. The actions required are few, which allow users to apply or export compositions effortlessly.

4.1 Installation and import

CompositionalNetworks.jl will keep compatibility with the latest stable and LTS releases of Julia[7]. We recommend the users to install the latest version of Julia[8] and to call it with `julia -t auto` which launch Julia with all available threads on the local machine. Installing CompositionalNetworks.jl from the Julia REPL is as simple as:

```julia
using Pkg
Pkg.update()
Pkg.add("CompositionalNetworks")
```

Alternatively, one can install from the package REPL (simply press ‘]’ in a Julia session) in a single command.

```
add CompositionalNetworks
```

Finally, loading the package is done by calling:

```julia
using CompositionalNetworks
```

Note that from Julia `v1.7`, calling `using` on an uninstalled package will prompt the user to install it. Hence, the last command is the only one strictly required.

4.2 Explore, learn, and compose

It is far from practical to apply the internals of CompositionalNetworks.jl step-by-step. We provide a user-friendly sequence of higher-level actions that fits most of expected uses of this package:

—*explore* a search space according to a collection of variables domains, a metric, and a concept with an optional parameter
—*learn* the weights of an ICN on that space according to the same metric
—*compose* the elementary operations according to the weights of the ICN

The whole sequence can be executed through `explore_learn_compose`. That function accepts an increasingly large collection of (keyword) arguments:

—`domains`: domains of the variables that define the training space
—`concept`: the concept of the targeted constraint
—`param`: an optional parameter of the constraint
—`search`: either `:flexible`,`:partial` or `:complete` search. Flexible search will use `search_limit` and `solutions_limit` to determine if the search space needs to be partially or completely explored
—`global_iter`: number of learning iterations

[7]Note that at the time of writing, Julia has just released `v1.7`. Compatibility with last LTS will start from the next LTS release.
[8]`https://julialang.org/downloads/`

—`local_iter`: number of generations in the underlying genetic algorithm
—`metric`: the metric to measure the distance between a configuration and known solutions
—`pop_size`: size of the population in the genetic algorithm
—`complete_search_limit`: used in conjunction with `solution_limit` to determine if a flexible search is set to complete or partial
—`solutions_limit`: see above
—`sampler`: optional sampler function for a collection of configurations
—`configurations`: optional collection of configurations to use as a training set
—`memoize`: if true, use a memoized version of the metric

Each of those actions is semi-automated through default parameters and machine learning over a large collection of benchmarks [1].

```
function explore_learn_compose(
  domains,
  concept,
  param=nothing;
  global_iter=Threads.nthreads(),
  local_iter=100,
  metric=:hamming,
  pop_size=400,
  search=:flexible,
  complete_search_limit=1000,
  solutions_limit=100,
  sampler=nothing,
  configurations=explore(
    domains, concept, param;
    search, complete_search_limit, solutions_limit
  ),
  memoize=true,
)
  dom_size = maximum(domain_size, domains)
  solutions, non_sltns = configurations
  return learn_compose(
    solutions,
    non_sltns,
    dom_size,
    param;
    local_iter,
    global_iter,
    metric,
    pop_size,
    sampler,
    memoize,
  )
end
```

A long-term goal of CompositionalNetworks.jl is to make the three first arguments optional too. We hope that such improvement will be used within JuliaConstraints, for instance within the following more advanced examples.

4.3 Advanced examples

Large scale use of CompositionalNetworks.jl is made within JuliaConstraints and appears in:

—Constraints.jl: the learning script can be found in `/src/learn.jl` and can be called as `learn_from_icn()`.
—ICNBenchmarks.jl: an extensive collection of ICN benchmarks run on an HPC cluster. This package is not registered in the general registry of Julia packages.

5. Future challenges

We envisage several directions for future improvement of ICN within CompositionalNetworks.jl. Follows a non-comprehensive list of the most challenging features.

New operations in existing layers. As the experiments have shown in [10, 1], some concepts are too complex to be covered with the current set of operations. The most direct solutions is to provide a broader collection of elementary operations.

New layers. Applied to EFSP/EFOP or not, it is likely that adding some layers of operation could cover a wider spectrum of combinatorial functions (with some cost for the interpretability and computation time).

Reinforcement learning. Some limitations in the ICN parametrization and metrics could be solved at runtime with dynamic learning. In the JuliaConstraints context, this could occur at the solver execution.

Apply ICN to other fields and problems. Contributions and suggestions are more than welcome!

6. Problems, constraints, and compositions zoo

The problems modelled, and the compositions extracted in this section are subject to future changes and improvements of the JuMP/JuCP packages. However, the keys ideas are presented here and examples will be updated accordingly within JuliaConstraints in the future. We welcome GitHub issues and pull requests.

6.1 A few combinatorial models

At the moment, ConstraintModels.jl hosts more than a dozen models. Along with *Magic Square* introduced in Section 2.3, we present models for two other classical optimization problems: *Golomb ruler* and *minimum cut*. The choice was made to cover different type of constraints, with error functions learned from CompositionalNetworks.jl.
Note that a model for the minimum cut problem in a network requires a matrix (graph) as input. Additionally, we add the optional argument of the interdiction of some links in the cut, which lead to an intractable problem called *Network Interdiction* [2].

```
function mincut(graph, source, sink, interdiction)
  m = JuMP.Model(CBLS.Optimizer)
  n = size(graph, 1)
  separator = n + 1

  @variable(m, 0 <= X[1:separator] <= n, Int)

  @constraint(m,
    [X[source], X[separator], X[sink]] in Ordered()
  )
  @constraint(m, X in AllDifferent())

  obj(x...) = o_mincut(graph, x...; interdiction)
  @objective(m, Min, ScalarFunction(obj))

  return m, X
end
```

In contrast, an instance of the Golomb ruler problem does not require specific dataset. However, this problem exists as both satisfaction and optimization variants.

```
function golomb(n, L)
  m = JuMP.Model(CBLS.Optimizer)

  @variable(m, 0 <= X[1:n] <= L, Int)

  @constraint(m, X in AllDifferent())

  # optional for output readability
  @constraint(m, X in Ordered())

  # No two pairs have the same length
  for i in 1:(n - 1), j in (i + 1):n
    for k in i:(n - 1), l in (k + 1):n
      (i, j) < (k, l) || continue
      @constraint(m,
        [X[i], X[j], X[k], X[l]] in DistDifferent()
      )
    end
  end

  # Add objective
  @objective(m, Min, ScalarFunction(maximum))

  return m, X
end
```

6.2 Constraints and compositions

Along with the magic-square model in Section 1, the models in this zoo use the following constraints: `:all_different`, `:dist_different`, `:linear`, and `:ordered`[9].

The *AllDifferent* constraint ensures that all the values of a given configuration are unique. This constraint is representative of Constraint Programming and is used in a large amount of models.

```
function all_different(x;
  X = zeros(length(x), 1), param=nothing, dom_size
)
  tr_in(Tuple([tr_count_eq_left]), X, x, param)
  for i in 1:length(x)
    X[i,1] = ar_sum(@view X[i,:])
  end
  return ag_count_positive(@view X[:,1]) |> (
      y -> co_identity(
        y; param, dom_size, nvars=length(x)
      )
    )
end
```

DistDifferent is constraint ensuring that $|x[1]-x[2]| \neq |x[3]-x[4]|$ for any vector x of size 4.

```
function dist_different(x)
  return abs(x[1] - x[2]) != abs(x[3] - x[4])
end
```

Linear (also called *Sum* or *LinearSum* in the literature) Global ensures that the sum of the values of x is equal to a given parameter $param$. Note that this version is a simplification of a linear sum of values with some coefficients.

```
function linear(x; param, dom_size)
  return abs(sum(x) - param) / dom_size
end
```

Ordered is a constraint ensuring that all the values of x are ordered (here in a decreasing order).

```
function ordered(x;
  X = zeros(length(x), 1), param=nothing, dom_size
)
  tr_in(
    Tuple([tr_contiguous_vals_minus]), X, x, param
  )
  for i in 1:length(x)
    X[i,1] = ar_sum(@view X[i,:])
  end
  return ag_count_positive(@view X[:,1]) |> (
      y -> co_identity(
        y; param, dom_size, nvars=length(x)
      )
    )
end
```

7. Acknowledgements

We're grateful to Ricardo Rosa and Pranshu Malik for the logo of JuliaConstraints presented as Figure 1.

8. References

[1] Jean-François Baffier, Khalil Chrit, Florian Richoux, Pedro Patinho, and Salvador Abreu. Interpretable composition networks: A scaling glass-box neural network to learn combinatorial functions. *IJCAI21-DSO Special issue of annals of mathematics and artificial intelligence on Data Science meets Optimization*, 2, 2022.

[2] Jean-François Baffier, Vorapong Suppakitpaisarn, Hidefumi Hiraishi, and Hiroshi Imai. Parametric multiroute flow and its application to multilink-attack network. *Discrete Optimization*, 22:20 – 36, 2016. doi:10.1016/j.disopt.2016.05.002. SI: ISCO 2014.

[3] Jeff Bezanson, Alan Edelman, Stefan Karpinski, and Viral B Shah. Julia: A fresh approach to numerical computing. *SIAM review*, 59(1):65–98, 2017.

[4] Félix Chalumeau, Ilan Coulon, Quentin Cappart, and Louis-Martin Rousseau. Seapearl: A constraint programming solver guided by reinforcement learning. *CoRR*, abs/2102.09193, 2021. 2102.09193.

[5] Philippe Codognet, Danny Munera, Daniel Diaz, and Salvador Abreu. Parallel local search. In Youssef Hamadi and Lakhdar Sais, editors, *Handbook of Parallel Constraint Reasoning*, pages 381–417. Springer, 2018. doi:10.1007/978-3-319-63516-3_10.

[6] Iain Dunning, Joey Huchette, and Miles Lubin. Jump: A modeling language for mathematical optimization. *SIAM Review*, 59(2):295–320, 2017. doi:10.1137/15M1020575.

[7] Chris Lattner and Vikram Adve. Llvm: A compilation framework for lifelong program analysis & transformation. In *International Symposium on Code Generation and Optimization, 2004. CGO 2004.*, pages 75–86. IEEE, 2004.

[9]The code provided here will be updated following the results of the benchmark in [1]. In particular, for `:dist_different` and `:linear`, the current code is a hand-written function.

[8] Benoit Legat, Oscar Dowson, Joaquim Garcia, and Miles Lubin. Mathoptinterface: a data structure for mathematical optimization problems. *INFORMS Journal on Computing*, in press.

[9] Nicholas Nethercote, Peter J. Stuckey, Ralph Becket, Sebastian Brand, Gregory J. Duck, and Guido Tack. Minizinc: Towards a standard cp modelling language. In Christian Bessière, editor, *Principles and Practice of Constraint Programming – CP 2007*, pages 529–543, Berlin, Heidelberg, 2007. Springer Berlin Heidelberg.

[10] Florian Richoux and Jean-François Baffier. Automatic cost function learning with interpretable compositional networks, 2020. 2002.09811.

[11] Florian Richoux, Alberto Uriarte, and Jean-François Baffier. Ghost: A combinatorial optimization framework for rts-related problems. *IEEE Transactions on Computational Intelligence and AI in Games*, PP(99):1–1, 2016. doi:10.1109/TCIAIG.2016.2573199.

[12] Kenneth O. Stanley. Compositional Pattern Producing Networks: A Novel Abstraction of Development. *Genetic Programming and Evolvable Machines*, 8(2):131–162, 2007.