CFN format (.cfn suffix)

Release 1.0.0

INRAE

Sep 26, 2025

CONTENTS

With this JSON-compatible format, it is possible:

- to give a name to variables and functions.
- to associate a local label to every value that is accessible inside toulbar2 (among others for heuristics design purposes).
- to use decimal and possibly negative costs.
- to solve both minimization and maximization problems.
- to debug your **.cfn** files: the parser gives a cause and line number when it fails.
- to use gzip'd or xz compressed files directly as input (.cfn.gz and .cfn.xz).
- to use dense descriptions for dense cost tables.

In a **cfn** file, a Cost Function Network is described as a JSON object with extra freedom and extra constraints.

Freedom:

- the double quotes around strings are not compulsory: both `"problem"` and `problem` are strings.
- double quotes can also be added around numbers: both `1.20` and `"1.20"` will be interpreted as decimal numbers.
- the commas that separate the fields inside an array or object are not compulsory. Any separator will do (comma, white space). So `[1, 2]` or `[1,2]` or `[1 2]` are all describing the same array.
- the delimiters for objects and arrays (`{}` and `[]`) can be used arbitrarily for both types of items.
- the colon (`:`) that separates the name of a field in an object from the contents of the field is not compulsory.
- It is possible to comment a line with a `#` the first position of a line.

Constraints:

- strings should not start with a character in `0123456789-.+` and cannot contain `/#[]{}:,` or a space character (tabs…).
- numbers can only be integers or decimals. No scientific notation.
- the order of fields inside an object is compulsory and cannot be changed.

A CFN is an object with 3 data: a definition of the main problem properties (tag `problem`), of variables and their domains (tag `variables`) and of cost functions (tag `functions`), in this order:

```
{ "problem": <problem properties>,
  "variables": <variables and domains>,
  "functions": <functions descriptions> }
```

Problem properties:

An object with two fields:

1. `"name"` : the name of the problem.
2. `"mustbe"` : specifies the direction of optimization and a global (upper/lower) bound on the objective. This is the concatenation of a comparator (> or <) immediately followed by a decimal number, described as a string. The comparator specifies the direction of optimization:

- "<": we are minimizing and the decimal indicates a global upper bound (all costs equal to or larger than this are considered as unfeasible).
- ">": we are maximizing and and the decimal indicates a global lower bound (all costs equal to or less than this are considered as unfeasible).

The number of significant digits in the decimal number gives the precision that will be used for all cost computations inside toulbar2.

As an example, `"mustbe": "<10.00"` means that the CFN describes a function where all costs larger than or equal to 10.00 are considered as infinite. All costs will also be handled with 2 digits of precision after the decimal point.

The two fields must appear in this order:

```
{ "name": "test_problem", "mustbe": "<-12.100" }
```

or

```
{test.problem <12.100}
```

in a more concise non-JSON-compatible form.

Variables and domains:

An object with as many fields as variables. All fields must have different names. The contents of a variable field can be an array or an integer. An array gives the sequence of values (defined by their name) of the variable domain. An integer gives the domain cardinality, without naming values (values are represented by their position in the domain, starting at 0). If a negative domain size is given, the variable is an interval variable instead of a finite domain variable and it has domain [0,-domainsize-1].

```
{ "fdv1": ["a", "b", "c"], "fdv2" : 2, "iv1" : -100}
```

defines 3 variables, two finite domain variables and 1 interval variable. The first domain variable has 3 values, `"a"` `"b"` and `"c"`. the second has two anonymous values and the interval variable has domain [0,99].

As an extra freedom, it is possible to give no name to variables. This can be achieved using an array instead of an object. The example above can therefore be written:

```
[[a b c] 2 -100]
```

or even just

```
[3 2 -100]
```

in a dense non JSON-compatible format.

Functions:

An object with as many fields as functions. Every function is an object with different possible fields. All functions have a `scope` which is an array of variables (names or indices). The rest of the fields depends on the type of the cost function: table cost function or global (including arithmetic functions).

Table cost functions:

Sparse functions format:* useful for functions that are dominantly constant. A numerical `defaultcost` must be given after the scope. The `costs` table must be an array of tuple costs: a sequence of value names or indices followed by a numeric cost or `inf` to represent a forbidden tuple. The `defaultcost` is used to define the cost of any missing tuple.

```
{"scope": ["fdv1", "fdv2"],
 "defaultcost": 0.234,
 "costs": ["a", 0, 5,
           "a", 1, 6.2,
           "c", 0, -7.21] }
```

is a possible sparse function definition. Here only 3 tuples are defined with their costs. All 3 remaining tuples will have cost `0.234`.

Dense function format: if the `defaultcost` tag is absent, a complete lexicographically ordered list of costs is expected instead.

```
{"scope": [ "fdv1", "fdv2" ],
 "costs": [4.2, 3.67, -12.1, 7.1, -3.1, 100.2] }
```

describes the 6 costs of the 6 tuples insides the cartesian product of the two variables `"fdv1"` and `"fdv2"`. To assign costs to tuples, all possible tuples of the cartesian product are lexicographically ordered using the declared value order in the domain of each variable. In the example above, the order over the six pairs will be `("a",0) ("a",1) ("b",0) ("b",1) ("c",0) ("c",1)` that will be associated to the costs `4.2, 3.67, -12.1, 7.1, -3.1` and `100.2` in this order. This lexicographic ordering is used for all arities.

Shared function format: If instead of an array, a string is given for the cost table, then this string must be the name of a yet undefined function. The actual function will have the same cost table as the future indicated function (on the specified scope). The domain sizes of the two functions must match.

```
{"scope": [ "v1", "v3" ],
 "costs": "f12" }
```

defines a function on variables `v1` and `v3` that will have the same cost table as the function i:code:*f12* that must be defined later in the file.

Global and arithmetic cost functions

These functions are defined by a `scope`, a `type` and `parameters`. The `type` is a string that defines the specific function to use, the `parameters` is an array of objects. The composition of the `parameters` depends on the `type` of the function.

At this point, in maximization mode, most of the global cost functions have restricted usage (with the exception of wregular).

Arithmetic functions:

These functions have all arity 2 and it is assumed here that these variables are called x and y . The values are considered as representing their index in the domain and are therefore integer. The `type` can be either:

- `">="` : with `parameters` array $[cst, \delta]$ where cst and δ are two costs, to express cost function $max(0, y + cst - x \leq \delta ? y + cst - x : upperbound)$. This is a soft inequality with hard threshold δ.
- `">"`: similar with a strict inequality and semantics $max(0, y + 1 + cst - x \leq \delta ? y + 1 + cst - x : upperbound)$
- `"<="`: similar with an inverted inequality and semantics: $max(0, x - cst - y \leq \delta ? x - cst - y : upperbound)$
- `"<"`: similar with a strict inequality and semantics $max(0, x - cst + 1 - y \leq \delta ? x - cst + 1 - y : upperbound)$
- `"="`: similar with an equality and semantics: similar with a strict inequality and semantics $|y + cst - x| \leq \delta ? |y + cst - x| : upperbound)$
- `"disj"`: takes a `parameters` array $[cstx, csty, w]$ to express soft binary disjunctive cost function with semantics $((x \geq y + csty) \vee (y \geq x + cstx))?0 : w)$
- `"sdisj"`: takes a `parameters` array $[cstx, csty, xmax, ymaxwxwy]$ to express a special disjunctive cost function with three implicit constraints $x \leq xmax$, $y \leq ymax$ and $(x < xmax \wedge y < ymax) \Rightarrow (x \geq y + csty \vee y \geq x + cstx)$ and an additional cost function $((x = xmax)?wx : 0) + ((y = ymax?wy : 0)$.

Example : arithmetic function with >= operator :

```
"arith0": {"scope": ["v5", "v6"],
           "type": ">=",
           "params": [1, 3]}
```

Global cost functions:

We use an informal syntactical description of each global cost function below. the "|" is used for alternative keywords and parentheses together with ?, * and + to denote optional or repeated groups of items (+ requires that at least one repetition exists). For more details on semantics and implementation, see:

1. Lee, J. H. M., & Leung, K. L. (2012). Consistency techniques for flow-based projection-safe global cost functions in weighted constraint satisfaction. *Journal of Artificial Intelligence Research*, 43, 257-292. *Artificial Intelligence*, 238, 166-189. 2. Allouche, D., Bessiere, C., Boizumault, P., De Givry, S., Gutierrez, P., Lee, J. H., … & Wu, Y. (2016). Tractability-preserving transformations of global cost functions. *Artificial Intelligence*, 238, 166-189.

Using a flow-based propagator:

- `salldiff`" with parameters array `[metric: "var"|"dec"|"decbi" cost: cost]` expresses a soft alldifferent with either variable-based (`var` keyword) or decomposition-based (`dec` and `decbi` keywords) cost semantic with a given `cost` per violation (`decbi` decomposes into a complete binary cost function network).
  - example :

    ```
    "f1": {"scope": ["v1" "v2" "v3" "v4"],
           "type": "salldiff",
           "params": {"metric": "var" "cost": 0.7}}
    ```

    generates a cost of 0.7 per variable assignment that needs to be changed for all variables to take a different value.
- "`sgcc`" with parameters array `[metric:"var"|"dec"|"wdec" cost: cost bounds: [[value lower_bound upper_bound (shortage_weight excess_weight)?]*]` expresses a soft global cardinality constraint with either variable-based (`var` keyword) or decomposition-based (`dec` keyword) cost semantic with a given `cost` per violation and for each value its `lower` and `upper` bound (`value shortage` and `excess weights` penalties must be given iff `wdec` is used).
  - example :

    ```
    name: {scope: [v1 v2 v3 v4]
           type: sgcc
           params: {
              metric: wdec
              cost: 0.5
              bounds: [[0 1 2 0.2 0.2]
                       [1 3 4 0.2 0.1]]
              }
          }
    ```

- "`ssame`" with parameters array `[cost: cost vars1: [(variable)*] vars2: [(variable)*]]` to express a permutation constraint on two lists of variables of equal size with implicit variable-based cost semantic
  - example :

    ```
    name: {scope: [v1 v2 v3 v4]
           type : ssame
           params : {
    ```

(continues on next page)

(continued from previous page)

```
          cost : 6.2
          vars1 : [v1 v2]
          vars2 : [v3 v4]
          }
      }
```

- "sregular" with parameters array [metric: "var"|"edit" cost: cost starts: [(state)*] ends: [(state)*] transitions: [(start-state symbol_value end_state)*] to express a soft regular constraint with either variable-based (var keyword) or edit distance-based (edit keyword) cost semantics with a given cost per violation followed by the definition of a deterministic finite automaton with arrays of initial and final states, and an array of state transitions where symbols are domain values indices.
  - example :

```
name: {scope: [v1 v2 v3 v4]
       type : sregular
       params : {
          metric: var
          cost: 1.0
          nb_states: 2
          starts: [0]
          ends: [0 1]
          transitions: [[0 0 0][0 1 1][1 1 1]]
          }
      }
```

Global cost functions using a dynamic programming DAG-based propagator:

- "sregulardp" with parameters array [metric: "var" cost: cost nb_states: nb_states starts: [(state)*] ends: [(state)*] transitions: [(start_state value_index end_state)*] to express a soft regular constraint with a variable-based (var keyword) cost semantic with a given cost per violation followed by the definition of a deterministic finite automaton with arrays of initial and final states, and an array of state transitions where symbols are domain value indices.
  - example: see sregular above.
- "sgrammar"|"sgrammardp" with parameters array [metric: "var"|"weight" cost: cost nb_symbols: nb_symbols nb_values: nb_values start: start_symbol terminals: [(terminal_symbol value (cost)?)*] non_terminals: [(nonterminal_in nonterminal_out_left nonterminal_out_right (cost)?)*] to express a soft/weighted grammar in Chomsky normal form. The costs inside the rules and terminals should be used only with the weight metric.
  - example:

```
name: {scope: [v1 v2 v3 v4]
       type : sgrammardp
       params: {
          metric : var
          cost : 1.012
          nb_symbols : 4
          nb_values : 2
          start : 0
          terminals : [[1 0][3 1]]
          non_terminals : [[0 0 0][0 1 2][0 1 3][2 0 3]]
```

(continues on next page)

(continued from previous page)

```
        }
    }
```

- "samong"|"samongdp" with parameters array [metric: "var" cost: cost min: lower_bound max: upper_bound values: [(value)*]] to express a soft among constraint to restrict the number of variables taking their value into a given set of value indices
  - example:

```
name: {scope: [v1 v2 v3 v4]
       type : samong
       params: {
          metric : var
          cost : 1.0
          min: 2
          max: 2
          values: [0]
          }
    }
```

- "salldiffdp" with parameters array [metric: "var" cost: cost] to express a soft alldifferent constraint with variable-based ("var" keyword) cost semantic with a given cost per violation (decomposes into samongdp cost functions)
  - example:

```
name: {scope: [v1 v2 v3 v4]
       type: salldiffdp
       params: {
          metric: var
          cost: 0.7
          }
    }
```

- "sgccdp" with parameters array [metric: "var" cost: "cost" bounds: [(value lower_bound upper_bound)*]] to express a soft global cardinality constraint with variable-based ("var" keyword) cost semantic with a given cost per violation and for each value its lower and upper bound (decomposes into samongdp cost functions)
  - example:

```
name: {scope: [v1 v2 v3 v4]
       type: sgccdp
       params: {
          metric: var
          cost: 1.1
          bounds: [[0 0 1] [1 2 3]]
          }
    }
```

- "max|smaxdp" with parameters array [defaultcost: defcost tuples: [(variable value cost)*]] to express a weighted max cost function to find the maximum cost over a set of unary cost functions associated to a set of variables (by default, defCost if unspecified)
  - example:

```
name: {scope: [v1 v2 v3 v4]
       type : smaxdp
       params: {
          defaultcost: 3
          tuples: [[0 0 4] [1 1 3][2 2 2][3 3 1]]
          }
       }
```

- "MST"|"smstdp" with empty parameters expresses a hard spanning tree constraint where each variable is assigned to its parent variable index in order to build a spanning tree (the root being assigned to itself)
  - example:

```
name: { scope: [v1 v2 v3 v4]
        type: MST params: []}
```

Global cost functions using a cost function network-based propagator (decompose to bounded arity table cost functions):

- "wregular" with parameters nb_states: nbstates starts: [[state cost]*] ends: [[state cost]*] transitions: [[state value_index state cost]*] to express a weighted regular constraint with weights on initial states, final states, and transitions, followed by the definition of a deterministic finite automaton with number of states, list of initial and final states with their costs, and list of weighted state transitions where symbols are domain value indices
  - example :

```
name: {scope: [v1 v2 v4 v3]
       type : wregular
       params: {
          nb_states: 4
          starts : [[0 0.0][1 0.5]]
          ends : [[2 -1.0] [3 0.0]]
          transitions : [[0 0 1 0.5][0 1 2 0.0]
                         [2 0 2 1.0][1 1 3 -1.0]]
          }
       }
```

- "walldiff" with parameters array [hard|lin|quad] cost to express a soft alldifferent constraint as a set of wamong hard constraint (hard keyword) or decomposition-based (lin and quad keywords) cost semantic with a given cost per violation.
  - example:

```
name: {scope: [v1 v2 v3 v4]
       type : walldiff
       params: {
          metric: lin
          cost: 0.8
          }
      }
```

- "wgcc" with parameters metric: hard|lin|quad cost: cost bounds: [[value lower_bound upper_bound]*] to express a soft global cardinality constraint as either a hard constraint (hard keyword) or with decomposition-based (lin and quad keyword) cost semantic with a given cost per violation and for each value its lower and upper bound
  - example:

```
name: {scope: [v1 v2 v3 v4]
       type : wgcc
       params: {
          metric: lin
          cost: 3.3
          bounds: [[0 0 1][1 2 2][2 0 1]]
          }
      }
```

- "wsame" with parameters a `metric: hard|lin|quad cost: cost` to express a permutation constraint on two lists of variables of equal size (implicitly concatenated in the scope) using implicit decomposition-based cost semantic
  - example:

```
name: { scope: [v1 v2 v3 v4]
        type : wsame
        params: {
           metric: lin
           cost: 3.3
           }
      }
```

- "wsamegcc" with parameters array `metric: hard|lin|quad cost: cost bounds: [[value lower_bound upper_bound]*]` to express the combination of a soft global cardinality constraint and a permutation constraint.
  - example:

```
name: {scope: [v1 v2 v3 v4]
       type : wsamegcc
       params: {
          metric: lin
          cost: 3.3
          bounds: [[0 0 1][1 0 1][2 0 1][3 0 0]]
          }
      }
```

- "wamong" with parameters `metric: hard|lin|quad cost: cost values: [(value)*] min: lower_bound max: upper_bound` to express a soft among constraint to restrict the number of variables taking their value into a given set of values.
  - example:

```
name: {scope: [v1 v2 v3 v4]
       type: wamong
       params: {
          metric: lin
          cost: 1
          values: [0]
          min: 1
          max: 1
          }
      }
```

- "wvaramong" with parameters array `metric: hard cost: cost values: [(value)*]` to express a hard among constraint to restrict the number of variables taking their value into a given set of values to be equal to the last variable in the scope.
  - example:

```
name: {scope: [v1 v2 v3 v4 v5]
       type: wvaramong
       params: {
          metric: hard
          cost: 12.0
          values: [1]
          }
      }
```

- "woverlap" with parameters `metric: hard|lin|quad cost: cost comparator: comparator to: righthandside]` overlaps between two sequences of variables X, Y (i.e. set the fact that Xi and Yi take the same value (not equal to zero))
  - example:

```
name: {scope: [v1 v2 v3 v4]
       type: woverlap
       params: {
          metric: hard
          cost: 2.01
          comparator: ">"
          to: 1
          }
      }
```

- "wdiverse" with parameters `distance: integer values: [(value)*]` to express a hard diversity constraint using a dual encoding such that there is a given minimum Hamming distance to a given variable assignment (values).
  - example:

```
name: { scope: [v1 v2 v3 v4]
        type : wdiverse
        params: {
           distance: 2
           values: [0 1 0 1]
           }
      }
```

- "whdiverse" with parameters `distance: integer values: [(value)*]` to express a hard diversity constraint using a hidden encoding such that there is a given minimum Hamming distance to a given variable assignment (values).
  - example:

```
name: { scope: [v1 v2 v3 v4]
        type : whdiverse
        params: {
           distance: 2
           values: [0 1 0 1]
```

(continues on next page)

(continued from previous page)

```
            }
      }
```

- "wtdiverse" with parameters distance: integer values: [(value)*] to express a hard diversity constraint using a ternary encoding such that there is a given minimum Hamming distance to a given variable assignment (values).
  - example:

```
name: { scope: [v1 v2 v3 v4]
        type : wtdiverse
        params: {
           distance: 2
           values: [0 1 0 1]
           }
      }
```

- "wsum" parameters metric: hard|lin|quad cost: cost comparator: comparator to: righthandside to express a soft sum constraint with unit coefficients to test if the sum of a set of variables matches with a given comparator and right-hand-side value.
  - example:

```
name: {scope: [v1 v2 v3 v4]
       type: wsum
       params: {
          metric: quad
          cost: 1.0
          comparator: "<="
          to: 4
          }
      }
```

- "wvarsum" with parameters metric: hard cost: cost comparator: comparator to express a hard sum constraint to restrict the sum to be comparator to the value of the last variable in the scope.
  - example:

```
mywsum: {scope: [v1 v2 v3 v4]
         type : wvarsum
         params: {
            metric: hard
            cost: 3
            comparator: "=="
            }
        }
```

  Comparators: let us note <> the comparator, K the right-hand-side (to:) value associated to the comparator, and Sum the result of the sum over the variables. For each comparator, the gap is defined according to the distance as follows:
  - if <> is == : gap = abs(K - Sum)
  - if <> is <= : gap = max(0,Sum - K)
  - if <> is < : gap = max(0,Sum - K - 1)

  - if <> is != : gap = 1 if Sum != K and gap = 0 otherwise
  - if <> is > : gap = max(0,K - Sum + 1);
  - if <> is >= : gap = max(0,K - Sum);

Warning: the decomposition of `wsum` and `wvarsum` may use an exponential size (sum of domain sizes). list_size1 and list_size2 must be equal in `ssame`.

Global cost functions using a dedicated propagator:

- "knapsack" with parameters `capacity: capacity weights: [(coefficient)*]` to express a hard global reverse knapsack constraint (i.e., a linear constraint on 0/1 variables with >= operator) where capacity and coefficients (one for each variable in the scope) are positive or negative integers. Use negative numbers to express a linear constraint with <= operator. See below a simple example encoding `v1+v2+v3+v4 >= 1`.
  - example:

```
myknapsack: {scope: [v1 v2 v3 v4]
         type : knapsack
         params: {
            capacity: 1
            weights: [1 1 1 1]
            }
        }
```

- "knapsackv" with parameters `capacity: capacity weightedvalues: [([variable value coefficient])*]` to express a hard global reverse knapsack constraint (i.e., a generalized linear constraint on domain variables with >= operator) where capacity and coefficients are positive or negative integers. Use negative numbers to express a generalized linear constraint with <= operator. Variables can be names or indices in the whole problem. They must also belong to the scope. See below a simple example encoding `(v1=1)+(v2=1)+(v3=1)+(v4=1) >= 1`.
  - example:

```
myknapsackv: {scope: [v1 v2 v3 v4]
         type : knapsackv
         params: {
            capacity: 1
            weightedvalues: [[v1 1 1] [v2 1 1] [v3 1 1] [v4 1 1]]
            }
        }
```

- "salldiffkp" with parameters array `[metric: "hard" cost: inf]` to express a hard alldifferent constraint (decomposes into knapsackv cost functions)
  - example:

```
name: {scope: [v1 v2 v3 v4]
       type: salldiffkp
       params: {
          metric: hard
          cost: inf
          }
      }
```

- "clique" with parameters `rhs: 1 values: [([(value)*])*]` to express a hard global clique constraint to restrict the number of variables taking their value into a given set of values (one set per variable) to at most

1 occurrence for all the variables. A clique of binary constraints must also be added to forbid any two variables from using both the restricted values.

- example:

```
f01: { scope: [v0 v1] defaultcost: 0 costs: [1 1 inf]}
f02: { scope: [v0 v2] defaultcost: 0 costs: [1 1 inf]}
f03: { scope: [v0 v3] defaultcost: 0 costs: [1 1 inf]}
f12: { scope: [v1 v2] defaultcost: 0 costs: [1 1 inf]}
f13: { scope: [v1 v3] defaultcost: 0 costs: [1 1 inf]}
f23: { scope: [v2 v3] defaultcost: 0 costs: [1 1 inf]}
myclique: {scope: [v0 v1 v2 v3]
         type : clique
         params: {
            rhs: 1
            values: [[1], [1], [1], [1]]
            }
        }
```

- "cfnconstraint" with parameters cfn: cost-function-network lb: cost ub: cost duplicatehard: value strongduality: value to express a hard global constraint on the cost of an input weighted constraint satisfaction problem in cfn format such that its valid solutions must have a cost value in [lb,ub[.
  - "duplicatehard" (0|1): if true then it assumes any forbidden tuple in the original input problem is also forbidden by another constraint in the main model (you must duplicate any hard constraints in your input model into the main model).
  - "strongduality" (0|1): if true then it assumes the propagation is complete when all channeling variables in the scope are assigned and the semantic of the constraint enforces that the optimum and ONLY the optimum on the remaining variables is between lb and ub.
  - example :

```
name: {scope: [v1 v2 v4]
       type : cfnconstraint
       params: {
          cfn:
            {
            problem: {name: "subcfn", mustbe: "<1000.0"}
            variables: {v1:2, v2:2, v4:2}
            functions: {
               {scope: [v1], costs: [0.0, -3.0]},
               {scope: [v2], costs: [-1.0, 0.0]},
               {scope: [v4], costs: [0.0, 2.0]}}
            }
          lb : -1.0
          ub : 0.0
          duplicatehard: 0
          strongduality: 0
          }
       }
```

Warning: the same floatting-point precision and optimization sense (minimization or maximization) should be used by the encapsulated cost function network and the main model. Warning: the list of variables of the encapsulated cost function network should be exactly the same as the scope (and with the same order).