COUNT-CP, a CP constraint learner for the holy grail 2021 challenge

Mohit Kumar, Samuel Kolb, Tias Guns

KU Leuven, Belgium

{firstname.lastname@cs.kuleuven.be}

Abstract: We introduce the COUNT-CP constraint learner for CP problems. Count-CP is based on the idea behind Mohit Kumar et al's COUNT-OR, a constraint learning algorithm for personnel rostering problems: empirically finding the suitable upper and lower bounds for relevant expressions `lb <= expr <= ub`. COUNT-OR learns bounds for expressions that occur frequently in rostering problems: aggregation operations on slices of tensors. In contrast, COUNT-CP uses a grammar over decision variables or pairs of decision variables suitable for expressing constraints typical for CP problems. Additionally, COUNT-CP introduces an additional step for generalizing learned models across different instance sizes. COUNT-CP is implemented using the CPMpy constraint programming and modeling environment.

Our proposal consists of two parts:

- Part 1: learning constraints for instances of the same size
- Part 2: learning generalized constraint models

Part 1: Learning constraints for instances of the same size

Constraint bias

Previous work (Kumar et al.) on learning for personnel rostering focussed on learning constraints over expressions that aggregate over various slices of a tensor, e.g.,

```
foreach i, j:
    0 <= sum(X[i, j, :]) <= 1
```

In this work, we consider the case of learning constraints of the following form:

```
lb <= expr <= ub
```

where `expr` is a numeric expression over one or two decision variables, such as `x + y` or `abs(x-y)`. To learn these constraints, we defined a grammar that captures expressions frequently occurring in CP problems.

Expression grammar

We consider both *unary* expressions, that is expression involving one individual variable, and *binary* expressions involving two variables.

To construct our grammar, we look at unary and binary expressions that can be expressed in the generic CPMpy modeling language. The expressions are the same for other modeling languages.

We consider the unary operators identity and absolute value, as well as the binary operators addition and subtraction. Unary expressions are generated using unary operators, while binary expressions are generated by combining both unary and binary operators. More concretely, we have the following grammar rules:

In [3]:
```
# [x, abs(x)]
def unary_operators():
    yield lambda x: x  # identity
    yield abs

# [x+y, x-y, y-x, |x|+|y|, |x|-|y|, |y|-|x|]
def binary_operators():
    for u in unary_operators():
        yield lambda x, y: u(x) + u(y)
        yield lambda x, y: u(x) - u(y)
        yield lambda x, y: u(y) - u(x)

# the unary expressions = unary operators
def generate_unary_expr(x):
    for u in unary_operators():
        yield u(x)

# the binary expressions = unary operators wrapped around binary
# e.g. for binary x-y: x-y, abs(x-y), abs(abs(x)-abs(y))
def generate_binary_expr(x, y):
    for b in binary_operators():
        for u in unary_operators(): # includes identity
            yield u(b(x, y))
```

Observe how this does not include the 'traditional' constraint biases `x != y`, `x <= y`, `x < y`, etc. The reason is that we have expressions that subsume those, namely correspondingly `abs(x - y) >= 1`, `x - y <= 0` and `x - y <= -1`. Hence, the above constraint bias (the inequalities over that grammar) can learn those and more.

One of the few unary/binary constraints it can not learn is `x != c`, for some constant `c` which lies between (exclusive) the lower and upperbound of `x`. We believe it will be very rare that a constraint model intentionally excludes one individual value, without that value being specified as the 'input data'. Hence, it is not part of our bias.

The bias also does not include n-ary global constraints such as `alldifferent()` or `increasing()`, however, these have decompositions into binary constraints, meaning that we can learn the decomposed versions. Also not included are tertiary constraints such as `x + y = z` or, equivalently, bounds on `x + y - z` for arbitrary triples. We leave it open whether these constructs are commonly used, and how to best manage the large number of candidates.

Finally, we also don't support element constraints like `list[x] == y`; they typically interact between different variable matrices (e.g. `list` and some variables `x`, `y`), for which we have not yet defined a grammar.

For this challenge this simple grammar already allowed us to learn the large majority of constraints. However, for more complicated problems, the grammar can trivially be extended with additional unary (e.g., `x*x`, `mod(x, 2)`), binary or n-ary operators (e.g., sums over lists). This increased expressiveness would, however, incur an additional computational cost during learning.

Constraint learning

Our learning algorithm only makes use of **positive** data, that is, the given true solutions. As all solutions have the same size, we can easily represent them as a tensor. More specifically, let there be `N` examples, where each example is a list of size `d`, then we represent the positive instances as an `N x d` matrix.

Similarly, for `N x d1 x d2` for an `d1 x d2` shaped matrix of decision variables. Without loss of generality, we can represent it as an `N x d` matrix, with `d = d1 x d2`, that is, a flattened matrix in which every row is an example and every column is one element of the `d1 x d2` shaped matrix of decision variables. If solutions contain multiple sets of decision variables, we can convert them into `N x d` matrix separately and concatenate these matrices. For example, if every solution contains two sets of decision variables of size `d1` and `d2`, respectively, we represent the solutions by two matrices of size `N x d1` and `N x d2` and concatenate them to obtain an `N x d` matrix where `d = d1 + d2`. From now on we assume an `N x d` representation of the solutions.

We will generate all possible expressions from the grammar, apply it on all possible elements of the matrices, and compute the resulting bounds. This can be seen as a convex hull of all possible expressions around the given solutions. This is a valid, though potentially overly restrictive, representation of constraints which the data satisfies.

Our approach has four phases:

1. Generate and compute expressions
2. Translate bounded expressions to constraints
3. Remove implied constraints
4. Learning the objective function

1. Generate and compute expressions

Given a tensor (list, matrix, or higher dimensionality) of decision variables, we generate all possible unary expressions, and for each expression, we apply it on each of the columns of our corresponding `N x d` matrix of solutions.

For every pair of expression `expr` and column index `i`, we obtain a list of `N` values and can easily compute the min and max values. This way, we obtain a data structure with tuples `(expr, i, lb, ub)`, where `lb` and `ub` are the lower and upper bounds of an expression applied to a decision variable `i` across all positive examples.

Next, we do the same for binary expressions over all *pairs* of columns in the `N x d` solution matrix, and we compute and store the bounds of the resulting lists of values. In the resulting

data structure we then store the bounds as `(expr, (i1, i2), lb, ub)`, where `i1` and `i2` are the indices of the two decision variables (or columns) to which the expression was applied.

(From a technical point of view, unary index tuples `(expr, i, lb, ub)` are represented as `(expr, (i,), lb, ub)`. This representation also allows extensions to n-ary expressions `(expr, (i1, i2, ..., in), lb, ub)`.)

Currently, constraints are computed for each set of decision variables separately. For problem type `03`, this means that we do not learn binary constraints that mix the `customer` and `warehouse` decision variables. We imposed this restriction to decrease the computation time for our experiments, however, removing this restriction allows more expressive constraints to be learned.

2. Translate bounded expressions to constraints

We read in the challenge data, extract from the `formatTemplate` the number of list/matrix variables and their shape, as well as their lower and upper bound.

Next, we create corresponding CPMpy decision variables with corresponding domains.

On those, we can then apply the same expressions as used in step 1. CPMpy overloads Python operators, so applying the operators results in valid CPMpy expressions and constraints. However, for many expressions, the computed lower or upper bound is trivial: it is equivalent to the lower or upper bound of the expression given the domains of the variables.

To detect these trivial constraints with non-restricting bounds, we can make use of CPMpy's mechanism to create auxiliary variables with tight domains for arbitrary expressions. Only if the learned bounds are different from those computed by CPMpy for the generic expression, the corresponding constraint is added to a candidate model.

Note that this does not change the set of solutions covered, it just makes it easier to inspect the models and slightly more efficient to execute it for the solver.

3. Remove implied constraints

While we already removed constraints with trivial bounds, we can still have implied constraints, such as `abs(x + y) >= 2` and `x + y >= 2` while x and y can only take positive values. More complex examples exist.

We consider a constraint `c1` to be *implied by* a constraint `c2` if all solutions of `c2` are also solutions of c1. That is, `c1` constrains a subset (or equal) of `c2`. Logically, we have `c2 -> c1`. This means that `not (c2 -> c1)` == `c2 and not c1` can not be true. In other words, we can use a constraint solver to search for a counter example for which `c2 and not c1`. If it finds one, the `c1` is not implied by `c2`. If it finds no such solution, then `c1` is implied, and it can safely be removed. This will again not change the set of accepted solutions, but it will make the learned constraint models more interpretable.

More generally, for a set of learned constraints `C`, we check for each one in turn whether it is implied by the other constraints or not, and remove it if it is. The pseudo-code is as follows:

```
C = [...] # list of constraints
i = 0
while i < len(C):
    m = Model(C[:i] + C[i+1:] + ~all(C[i]))
    if m.solve():
        # skip it, not implied
        i += 1
    else:
        del C[i]
```

where `C[:i] + C[i+1:]` consists of all constraints except `C[i]` and `~all(C[i])` is the negation of `C[i]` (even when it is a list of constraints).

With a lazy clause generation solver like ortools CP-SAT, generating (or disproving) such counter examples is pretty efficient.

Note that the *order* in which the constraints are tested for removal matters... We assume that the grammar rules are ordered from 'simple' to 'complex', and hence we traverse the list of constraints from the end (more complex) to the beginning (the unary constraints, simple).

Improving this ordering of constraints through more elaborate estimates of which constraint should be preferred over other equivalent ones could be useful.

4. Learning the objective function

We feel the challenge has a rather loose interpretation of 'objective function', given that there are multiple solutions given and hence it does not seem to be a function that has to be optimized.

Instead, one could see learning the objective function as an *equation discovery* problem: given a solution, what equation generates the given value. Equations could be generated using a simple grammar of n-ary expressions in CPMpy, i.e., `sum, min, max`. However, in this challenge, the objective seems to be simply `max` in all types with objective functions except one.

The other one, type `03`, involves weighted sums over projected variable representations. We could add weighted sums over matching matrix dimensions, but with the limited time available we decided to manually provide the required objective function for this problem.

Results for local learning

The following table shows the results for this local learning approach:

In [5]:
```
import pandas as pd
pd.set_option("display.max_rows", None)
pd.set_option('precision', 2)
df = pd.read_csv("merged.csv")
df = df[df.model_used == 'instance level']
```

```
df['percentage_pos'] = df['percentage_pos'].round(2)
df['percentage_neg'] = df['percentage_neg'].round(2)
df[['type','percentage_pos', 'percentage_neg']].groupby(['type']).mean()
```

Out[5]:

type	percentage_pos	percentage_neg
1	100.0	100.00
2	100.0	99.91
4	100.0	99.68
7	100.0	77.63
8	100.0	99.93
10	100.0	99.96
11	100.0	100.00
13	100.0	99.50
14	100.0	100.00
15	100.0	99.95
16	100.0	100.00

Generally we see huge reduction from initial set of constraints to filtered (non-trivial, non-implied) set of constraints. In part, the initial set of constraints could already be reduced by pre-processing expressions in the grammar, e.g., for positive variables `x` and `y`, expressions `x + y` and `abs(x + y)` are equivalent and one of them can be excluded before learning.

As per construction, our approach always satisfies all positive examples.

The last column shows the percentage of negative examples that we *reject*. That is, our constraint model correctly answers 'UNSAT' when we force the decision variables to equal the negative example.

It shows that our local constraint learning approach achieves >99% accuracy on many constraint types, with a reasonable number of constraints in the model. The exception is `type07`, where a limited visual inspection did not reveal to the authors what a missing constraint could be.

Part 2: Learning generalized constraint models

Sequence bias

As mentioned earlier, our constraint learning approach can learn constraints such as `alldifferent` by learning the individual constraints `abs(x - y) >= 1` between pairs of decision variables. This approach, however, does not scale to problems of different sizes as the constraints are "hardcode" for individual pairs in the instance.

To overcome this limitation and learn constraints that are independent of the problem size, our aim is to find *index groups*, groups of decision variables or pairs of variables, that share a constraint.

For example, `alldifferent` can be encoded as:

```
forall pairs (x[i], x[j]):
    abs(x[i] - x[j]) >= 1
```

An increasing sequence can be encoded as:

```
forall sequential pairs (x[i], x[i+1]):
    x[i+1] - x[i] >= 0
```

Sequence grammar

The algorithm of part 1 provides us with all unary and binary expressions. In this second step, we group decision variables or pairs of decision variables into sequences. To learn these constraints we use a grammar of sequence generators.

We consider the unary sequences:

- `all` : all variables
- `even` : all variables with even indices
- `odd` : all variables with odd indices

The pairwise sequences are:

- `all` : all pairs of variables
- `sequence` : all sequential pairs of variables `x[i], x[i+1]`
- `even` : all sequential pairs with even indices
- `odd` : all sequential pairs with odd indices

Sequences can also be generated using additional background information, such as the pairs given in the inputData of type01.

Generalized constraint learning

To learn generalized constraints, we first generate all sequences of variables and for every expression we combine the lower and upper bounds of the individual expressions.

For example, for the sequence of even variables, we compute the minimal lower bound and maximal upper bound of all entries `(expr, (i,), lb, ub)` -- obtained in part 1 -- where `i` is even. The result is an entry `(expr, (forall even vars), lb_min, ub_max)`, where `(forall even vars)` is the generator expression and `lb_min` and `ub_max` are the minimal lower bound and maximal upper bound across indices.

Next, as in part 1, for each instance we filter out the trivial lower and upper bounds and look for redundant entries. This redundancy check uses the same principle as the one in part 1, except that we consider groups of constraints at ones: all constraints generated by a generator.

Finally, we compare the generalized constraints across *instances*. If for any instance a combination `(expr, generator, _, _)` has been filtered out, we do not include it in the final model. Otherwise, we once more compute the minimal lower and upper bounds across

all entries `(expr, generator, lb, ub)` to obtain the final generalized constraints of the form `(expr, generator, lb_min, lb_max)`. This time `lb_min` and `ub_max` are the minimal lower bound and maximal upper bound across instances.

A shortcoming of this method is that currently sequences are generated over all decision variables. This shortcoming could be addressed by first partitioning variables into groups, e.g., rows of a matrix, blocks of variables, diagonal elements, etc., using a suitable grammar, and using a generator within those partitions. We believe this extension would allow us to tackle most of the problems in this competition.

A second improvement would be the inclusion of symbolic bounds in the constraints. Instead of using numeric constants `lb` and `ub` in the constraints `lb <= expr <= ub`, we could try to replace them with symbolic constants from the input data or from the problem size. (This would also be helpful when implementing the partitioning described in the previous step.)

In [6]:
```
import pandas as pd
pd.set_option("display.max_rows", None)
pd.set_option('precision', 2)
df = pd.read_csv("merged.csv")
df = df[df.model_used == 'type level']
df['percentage_pos'] = df['percentage_pos'].round(2)
df['percentage_neg'] = df['percentage_neg'].round(2)
df[['type','percentage_pos', 'percentage_neg']].groupby(['type']).mean()
```

Out[6]:

type	percentage_pos	percentage_neg
1	100.0	100.00
2	100.0	99.85
4	100.0	96.66
7	100.0	0.00
8	100.0	98.25
10	100.0	99.96
11	100.0	0.00
13	100.0	98.34
14	100.0	100.00
15	100.0	99.16
16	100.0	100.00

The performance deteriorates very slightly when using generalised models. The exceptions are `type07` and `type11`. It's possible that we are supposed to use inputData in these cases to learn generalised models.

Note that the generalised models learned here can also be applied on the instances for which no positive/negative examples are given.

Conclusion

We implemented a constraint learner through an inequality-based learner over a constraint grammar involving unary and binary expressions over decision variables. Key techniques used are grammars, matrix operations to efficiently extract bounds, and the CPMpy modeling environment.

Our local learning approach achieves high accuracy and automatically reduces the size of the learned model by removing trivial and implied constraints. We also investigated automated ways to generalize the learned constraints, with decent initial success.

Much future work remains, especially with respect to reducing the set of local constraints further to an 'elementary' set; as well as generalizing constraints across instances of different size.