

# A Meta Constraint Satisfaction Optimization Problem for the Optimization of Regular Constraint Satisfaction Problems

Sven Löffler, Ke Liu<sup>a</sup> and Petra Hofstedt

Brandenburg University of Technology Cottbus-Senftenberg, Department of Mathematics and Computer Science, MINT, Programming Languages and Compiler Construction Group, Konrad-Wachsmann-Allee 5, 03044 Cottbus, Germany

Keywords: Constraint Programming, CSP, Decomposition, Optimization, CSOP, Regular CSP.

Abstract: This paper describes a new approach on optimization of regular constraint satisfaction problems (rCSPs) using an auxiliary constraint satisfaction optimization problem (CSOP) that detects areas with a potentially high number of conflicts. The purpose of this approach is to remove conflicts by the combination of regular constraints with intersection and concatenation of their underlying deterministic finite automata (DFAs). This, eventually, often allows to significantly speed-up the solution process of the original rCSP.

## 1 INTRODUCTION

Constraint programming (CP) is a powerful method to model and solve NP-complete problems in a declarative way. Typical research problems in CP are rostering, graph coloring, optimization and satisfiability (SAT) problems (Marriott, 1998).

Since the search space of CSPs is very big and the solution process often needs an extremely high amount of time we are always interested in improvement and optimization of the solution process. Mostly, a CSP in practical can be described in various ways; and consequently, the problem can be modeled by different combinations of constraints, which results in the diversity of resolution speed and behavior. Hence, the diversity of models and constraints for a given CSP offers us an opportunity to improve the problem solving by using another model in which a subset of constraints can be replaced with a faster constraint which combines the original constraints.

Our approach based on the idea that CSPs can be modeled as (Löffler et al., 2017) or transformed into (Löffler et al., 2018) rCSPs. Nevertheless, there are also several rCSP descriptions for a given problem which also have a diversity in their resolution speed and behavior. The solving speed and behavior depends amongst other things from the number of backtracks in the depth first search of the solving process. Because only overlapping constraints can lead to backtracks in a rCSP, we developed a CSOP which

detects highly overlapping parts of a CSP which can be substituted by a singleton regular constraint.

## 2 PRELIMINARY


In this section, we introduce the necessary definitions, methods and theoretical considerations which are the basis of our approach. The basis for all of our considerations is the regular membership constraint (in the following regular constraint called) and its propagation algorithm defined and explained in (Pesant, 2004).

We consider furthermore constraint satisfaction problems (CSPs), regular constraint satisfaction problems (rCSPs), constraint satisfaction optimization problems (CSOPs) and sub-CSPs which are defined as follows.

**CSP** (Dechter, 2003) A constraint satisfaction problem (CSP) is defined as a 3-tuple  $P = (X, D, C)$  with  $X = \{x_1, x_2, \dots, x_n\}$  is a set of variables,  $D = \{D_1, D_2, \dots, D_n\}$  is a set of finite domains where  $D_i$  is the domain of  $x_i$  and  $C = \{c_1, c_2, \dots, c_m\}$  is a set of primitive or global constraints containing between one and all variables in  $X$ .

**rCSP** (Löffler et al., 2017) A regular constraint satisfaction problem (CSP) is a CSP  $P = (X, D, C)$  like defined before, with only regular constraints in  $C$ .

Additionally, we consider only rCSPs with a strict variable order  $x_1, \dots, x_n$ . This means that in each constraint  $c \in C$ , the variables are also ordered by their index, where  $x_i$  occur earlier as  $x_j \forall i, j \in \{1, \dots, n\}, i < j$ .

<sup>a</sup>  <https://orcid.org/0000-0002-5256-9253>

*CSOP* (Rossi et al., 2006; Tsang, 1993) A constraint satisfaction optimization problem (CSOP)  $P_{opt} = (X, D, C, f)$  is defined as a CSP with an optimization function  $f$  that maps each solution to a numerical value.

*Sub-CSP* Let  $P = (X, D, C)$  be a CSP. For  $C' \subseteq C$  we define  $P_{sub} = (X', D', C')$  such that  $X' = \bigcup_{c \in C'} vars(c)$  with corresponding domains  $D' = \{D_i \mid x_i \in X'\} \subseteq D$ .

## 2.1 Definitions and Methods

For the further concepts and methods we assume that a CSP  $P = (X, D, C)$  is given.

In our description we use the two functions  $vars(c)$  and  $cons(x)$ , where the method  $vars$  has a constraint  $c \in C$  as input and returns all variables  $X' \subseteq X$  which are covered by this constraint. Similarly, the method  $cons$  has a variable  $x \in X$  as input and returns all constraints  $C' \subseteq C$  which cover this variable.

We say two constraints  $c_i, c_j \in C, i \neq j$  *overlap* if the intersection of variables  $vars(c_i)$  and  $vars(c_j)$  is not empty.

We define the maximal size  $size(P)$  of a CSP  $P = (X, D, C)$  as the product of all cardinalities of the domains of the CSP  $P$ .

$$size(P) = \prod_{i=1}^{|X|} |D_i| \quad (1)$$

The size of a variable and of a constraint can be defined similar, i.e.  $size(x_i) = |D_i|$  and  $size(c) = \prod_{x_i \in vars(c)} |D_i|$ .

## 2.2 Theoretical Consideration

In this paper, it is important to distinguish between local (Apt, 2003) and global consistency (Dechter, 2003). Local consistency guarantees that each value of a variable in the scope of a certain constraint is at least part of one of its solutions. A CSP is locally consistent if all of its constraints are locally consistent. In contrast, global consistency implies that each value of the variable of the CSP can be extended to at least one solution of the entire CSP. Therefore, global consistency is a much stronger enforcement of consistency. In particular, search interleaved with global consistency is backtracking free.

For a given CSP  $P$ , we can separate the propagators of the constraints into two sets: the one set ensure local consistency for the constraints and the other one not (e.g. bound consistency). In the following we name a constraint locally consistent if the constraint has only propagators which enforce local consistency.

Examples for both categories based on their implementation in Choco Solver (Prud'homme et al., 2017) are for locally consistent constraints: *arithm*, *count* or *regular* and for not locally consistent constraints: *cumulative* or *sum*.

Consider a CSP  $P$  with only one constraint  $c_1$ , which has a propagator that ensures local consistency, and this implies that  $P$  must be backtracking free. The question is what leads to a not backtracking free CSP? If we add another constraint  $c_2$ , which has a propagator that ensures local consistency, to  $P$  then there are two possibilities.

*Case 1:* The two constraints  $c_1$  and  $c_2$  do not overlap. It is clear that this cannot lead to a fail because we can decompose such a CSP into two separate CSPs  $P_1$  and  $P_2$ , solving them individually and merge the results together. Hence, each solution of  $P_1$  contains no value assignment for a variable of  $P_2$  and vice versa, every element of the cross product of the solutions of  $P_1$  and  $P_2$  is a solution of  $P$ .

*Case 2:* The two constraints  $c_1$  and  $c_2$  overlap. Depending on different things such as the types of constraints  $c_1$  and  $c_2$ , search strategy, propagation order, variable order etc. the CSP has a fail or not.

Thus, the origins for fails in a CSP  $P$  with only locally consistent constraints are overlapping constraints. This leads to the consideration that areas of the CSP  $P$  with a high number of overlapping constraints have the potential to be responsible for a large number of fails.

Because the regular constraint is locally consistent based on the propagation algorithm used in (Prud'homme et al., 2017) and discussed in (Pesant, 2004), we can assume that a rCSP needs only backtracks if their exist overlapping variables.

For the reason that only overlapping constraints leads to backtracking in the search process we assume that the areas in the CSP with a high density of constraints leads to a high number of backtracks. If we substitute all constraints of an area with a high density of constraints by only one locally consistent (regular) constraint we can reduce the number of backtrackings significantly and so we can improve the solving speed of the CSP.

## 3 THE META CSOP FOR THE OPTIMIZATION OF rCSPs

In this section, we present an approach to detect highly overlapping areas in a CSP  $P$  using a Meta CSOP  $P_{opt}$ .

### 3.1 A General CSOP to Find Maximal Overlapping Sub-CSPs

The idea behind this approach is to find sub-CSPs  $P' = \{P_1, \dots, P_k\}$  in the original CSP  $P$  which are responsible for a large number of fails inside the backtrack search and small enough to replace each of them by a singleton regular constraint. We assume that a sub-CSP with a large number of overlapping constraints is likely to incur fails during backtrack search.

For detecting  $k$  sub-CSPs  $\{P_1, \dots, P_k\}$  of a given CSP  $P = (X, D, C)$  where  $X = \{x_1, \dots, x_n\}$ ,  $D = \{D_1, \dots, D_n\}$  and  $C = \{C_1, \dots, C_m\}$  we use a CSOP  $P_{opt} = (X', D', C', f)$  where  $X' = \{x'_j | \forall j \in \{1, \dots, m\}\} \cup x_{opt}$ ,  $D = \{D_j = \{0, \dots, k\} | \forall j \in \{1, \dots, m\}\} \cup D_{opt} = \{0, \dots, \infty\}$ ,  $C = \{cspOverlapSplit(X', M, k, s, D, v)\}$  and  $f = maximize(x_{opt})$  which finds an optimized decomposition of  $P$  into  $k$  sub-CSPs with maximum number of overlapping.

For each constraint  $c_j \in C$  a variable  $x_j$  with domain  $D_j = \{0, \dots, k\}$  is created. The value assignment  $v_j$  of each variable  $x_j \in X'$  of  $P_{opt}$  represents that the corresponding constraint  $c_j \in C$  of  $P$  is part of sub-CSP  $P_{v_j}$  with  $v_j$  is greater null, otherwise the corresponding constraint  $c_j$  is not part of a sub-CSP.

The *cspOverlapSplit* constraint will be explained in the next section. The objective function  $f$  maximizes the  $x_{opt}$  variable with domain  $D_{opt} = \{1, \dots, \infty\}$  which is also explained in the following section.

### 3.2 The cspOverlapSplit-Constraint

The *cspOverlapSplit* constraint is a newly developed constraint which split a set of constraints (from a CSP  $P$ ) in maximum  $k$  sub-sets (representing sub-CSPs  $P_1, \dots, P_k$ ) where the sum of overlapping variables inside of each sub-set is maximum and the size of each sub-CSP is smaller or equal a given value.

The algorithm gets a set of variables  $X'$ , a two-dimensional array of integers  $M$ , the maximum number of sub-CSPs  $k$ , a maximum sub-CSP size  $s$  (so that  $size(P_l) \leq s | \forall l \in \{1, \dots, k\}$ ), the domains  $D$  of the original CSP  $P$  and a  $n$ -dimensional vector  $v = v_1, \dots, v_n$  as input.

The variables  $X'$  must be the variables like described in the last section. Each variable  $x_j \in X'$  represents the corresponding constraint  $c_j \in C$  of the given CSP  $P$ . If the value of variable  $x_j$  is set to a value  $v$  it means that the constraint  $c_j$  is part of the sub-CSP  $v$  and therefore not included in other sub-CSPs.

The two-dimensional array  $M$  illustrates which variables are covered by which constraints. For each

constraint  $c_j \in C$  of CSP  $P$  exist one line which contains all variables  $X_c$  which are covered by the constraint ( $X_c = \{x_i | x_i \in vars(c)\}$ ). So the entry  $M_{j,l} = i$  follows from the fact that constraint  $c_j$  contains variable  $x_i$  at  $l$ -th position.

Each value  $v_i \in v | \forall i \in \{1, \dots, n\}$  represents the number of constraints which cover variable  $x_i \in X$  in  $P$ .

---

Algorithm 1: *cspOverlapSplit*.

---

**Input:** variables  $X'$ ,  $\text{int} \times \text{int} M$ ,  $\text{int} k$ ,  $\text{int} s$ , Domains  $D$ ,  $\text{intVector } v = v_1, \dots, v_n$

- 1 Create  $k + 1$  integer vectors  $a_0, \dots, a_k$  of size  $|X|$
- 2 **forall**  $x_i \in X'$  **do**
- 3     **if** (*isInstantiated*( $x_i$ )) **then**
- 4          $\text{int } v = x_i.\text{getValue}()$
- 5         **forall**  $j \in \{1, \dots, |vars(c_i)|\}$  **do**
- 6              $a_v[M_{i,j}]++$
- 7  $\text{int } \text{maxValue} = 0$
- 8 **forall**  $l \in \{1, \dots, k\}$  **do**
- 9      $\text{maxValue} = \text{maxValue} +$   
            $\text{calculateValueSubCsp}(a_0, \dots, a_k, l, D, s, v)$
- 10 update upperBound of  $x_{opt}$  to  $\text{maxValue}$

---

Algorithm 1 shows the propagation algorithm of the *cspOverlapSplit* constraint. In line 1,  $k + 1$  integer vectors of size  $|X|$  (number of variables in the original CSP  $P$  plus 1) are created. The vectors  $a_1, \dots, a_k$  represent the sub-CSPs  $P_1, \dots, P_k$  and the vector  $a_0$  represents the variables which could not assignment to a sub-CSP. For each vector  $a_l$  the  $i$ -th entry ( $\forall i \in \{1, \dots, |X|\}$ ) represents if the  $i$ -th variable  $x_i \in X$  of CSP  $P$  is part of sub-CSP  $P_l$  ( $a_l[i] > 0$ ) or not ( $a_l[i] = 0$ ). If the value  $a_l[i]$  is greater than zero then it represents also the number of covering constraints of  $x_i$  in sub-CSP  $P_l$ .

In lines 2 to 6, the variables  $x_i \in X'$  (representation of  $c_i$  in the original CSP  $P$ ) are checked if there are already instantiated. If a variable  $x_i$  is instantiated to value  $v$  then it means that constraint  $c_i \in C$  of  $P$  and all of it variables ( $vars(c_i)$ ) are part of the sub-CSP  $P_v$ . If such assignments exist, the algorithm increase all integer values in  $a_v$  by one which represent variables covered by the constraint  $c_i$  (line 5 and 6).

In lines 7 to 9, the new upper bound of the optimization variable will be calculated, based on the assignments for the sub-CSPs. The calculation process will be explained below.

Finally, in line 10, the upper bound of the optimization variable will be updated.

*Remark* Finding the optimized solution for the CSOP  $P_{opt}$  is also very time consuming and in this case may not useful. Our algorithm find a compro-

mise between a good solution and a short execution time.

Algorithm 2: *calculateValueSubCsp*.

---

**Input:** integer vectors  $a_0, \dots, a_k$ , integer  $l$ , domains  $D$ , integer  $s$ , integer vector  $v = v_1, \dots, v_n$

**Output:** The maximum integer influence of the sub-CSP  $P_l$  to  $x_{opt}$ .

---

```

1 Integer list doms = new sorted list
2 Integer list values = new sorted list
3 Integer currentSize = 1
4 Integer value = 0
5 forall  $i \in \{1, \dots, n\}$  do
6   Integer additionalValue =  $v_i$ 
7   forall  $a_j \in \{a_0, \dots, a_k\} | (j \neq l)$  do
8     additionalValue = additionalValue -  $a_j[i]$ 
9   if  $a_l[i] \geq 1$  then
10    doms.add( $|D_i|$ )
11    currentSize = currentSize *  $|D_i|$ 
12    if (additionalValue > 1) then
13      value = value + additionalValue2
14  else
15    values.add(additionalValue)
16 if currentSize >  $s$  then
17   fails()
18 Integer domValue = getNextDomain(doms,
19   sort( $D$ ))
20 currentSize = currentSize * domValue
21 while currentSize ≤  $s$  do
22   value = value + (values.next())2
23   domValue = getNextDomain(doms,
24   sort( $D$ ))
25   currentSize = currentSize * domValue
26 return value

```

---

Algorithm 2 calculates the maximum value which is possible for the subCSP  $P_l$ .

In lines 1-4, the needed variables are instantiated.

The integer list *doms* will be filled with the domainsizes of all variables which are part of the sub-CSP  $P_l$  starting with the smallest. The integer list *values* will be filled with the possible values which variables (outside of  $P_l$ ) can add to the result value whenever they are added to  $P_l$ . The *currentSize* represents the size of the sub-CSP  $P_l$  with the current variable and constraint selection  $size(P_l)$ . The *value* is the maximum integer influence of the sub-CSP  $P_l$  to  $x_{opt}$ .

In lines 6-8, the possible number of constraints which cover the variable  $x_i$  in the sub-CSP  $P_l$  will be calculated (*additionalValue*). For this the number of constraints which covers variable  $x_i$  in  $P$  ( $v_i$ ) will be

reduced by every constraint which is assignment to an other sub-CSP (line 8).

If the variable  $x_i$  is part of the sub-CSP  $P_l$  ( $a_l[i] \geq 1$ ) and more than one constraint cover this variable in sub-CSP  $P_l$  (*additionalValue* > 1) then the *additionalValue* will be added to the result *value* and the *currentSize* value will be updated, otherwise only the *additionalValue* will be added to the *values* list, lines 9-15.

We are interested in an area with a high number of overlapping so if a variable is covered by only one constraint in a sub-CSP it does not infect the number of overlapping, and therefore the *value* will not change (line 12). If the variable is covered by more than one constraint in the sub-CSP then we consider it in a quadratic way (line 13).

If the variable  $x_i$  is no variable of the sub-CSP  $P_l$  at the moment then it is not clear if *additionalValue* is the best value or there are other variables with a higher value. For this reason the *additionalValue* is added to the *values* list, from which we will take the best values later.

If the *currentSize* of the sub-CSP  $P_l$  is greater than the maximum allowed sub-CSP size  $s$  then the CSP is temporary not satisfiable and the propagator fails (line 16 and 17).

At this point the *value* is only calculated based on the variable assignments to the sub-CSP  $P_l$  which are already done. In lines 18-23, the *value* will be increased by a value which can be reached by variables which are not in  $P_l$  yet but can be added to the sub-CSP  $P_l$ .

In lines 18 and 19, the smallest domain which is not part of sub-CSP  $P_l$  will be selected and the *currentSize* will be increased by this value.

While the *currentSize* is not as big as the given maximum CSP size  $s$ , the *value* will be increased by the next value from the list *values* with descending order. This will be repeated until the *currentSize* is getting to big (lines 20-23).

With this calculation we obtain a *value* which might be bigger as the real possible value because we say always that the variable with the smallest size leads to the biggest impact to *value*. It is clear that the real value cannot be higher so that *value* can be used as upper bound of our optimization variable  $x_{opt}$ .

### 3.3 The Complexity of the CspOverlapSplit Algorithm

In this section we prove the complexity of the *cspOverlapSplit* algorithm. We first prove the complexity of the *calculateValueSubCsp* algorithm which is used in the other one.

In lines 5-8, there are two nested loops which leads to a complexity class of  $O(n * (k + 1))$ . In line 10, there is an inclusion into a sorted list with the maximum length of  $n - 1$  if all other variables have been included before. The other efforts in lines 1-17 can be ignored for the complexity class, the complexity is in  $O(n * (k + 1 + \log(n)))$  for this part.

The *getNextDomain* method has a complexity of  $n$  in worst case if the next domain is  $D_i$  with  $D_i \geq D_j | \forall j \in \{1, \dots, n\}, j \neq i$  where  $x_i$  is the only variable which is not part of sub-CSP  $P_i$ . In this case the while loop in lines 20-23 will be ignored because after this all variables are part of the sub-CSP  $P_i$ , *currentSize* = *size(P)* and logically *s* is chosen smaller as *size(P)*.

If the loop is not ignored then the effort of *getNextDomain* and the loop is accumulated in  $O(n)$  in the worst case. This leads to a complexity of  $O(n * (k + 1 + \log(n)) + n) \in O(n * (k + 1 + \log(n)))$  for the *calculateValueSubCsp* algorithm.

The *cspOverlapSplit* algorithm is partitioned into two parts. The first part (lines 1-6) has two nested loops, where the first is repeated  $|X'| \leq |C| = m$  times. The inner loop is repeated  $\text{vars}(c_i) \leq |X| = n$  times. This leads to the complexity class  $O(m * n)$  for the first part.

The second part repeats  $k$  times the *calculateValueSubCsp* algorithm, so it is in  $O(k * (n * (k + 1 + \log(n))))$ .

Finally, the overall complexity is in  $O(n * m + n * k * (k + 1 + \log(n))) \in O(n * (m + k^2 + k * \log(n)))$ . Depending on  $\log(n)$  or  $k$  is bigger, this can be reduced to  $O(n * (m + \log^2(n)))$  or  $O(n * (m + k^2))$ .

For a static number of sub-CSPs  $k$ , the algorithm can be propagated in quadratical time, which is an acceptable time for such constraints and which is in the same complexity class like the arc consistency propagation algorithm of the *allDifferent* constraint or the propagator of the *cumulative* constraint.

### 3.4 Integration of the Algorithm into the Regularization Process

This section explains where we use the Meta CSOP problem in the regularization process.

Figure 1 shows the regularization process from a general CSP  $P$  to an optimized rCSP  $P'_{reg}$ . It starts with a general CSP  $P = (X, D, C)$  which have different kinds of constraints  $c \in C$  like *count*, *allDifferent* or *sum*, then  $P$  will be transformed into a rCSP  $P_{reg}$ . This can be realized by direct transformations as explained in (Löffler et al., 2018) or with the detection of all solutions of a sub-CSP of  $P$  which will be convert to a regular constraint.

After getting the rCSP  $P_{reg}$  we optimize it ( $P'_{reg}$ ) by

the intersection and minimization of the DFAs which are used in the regular constraints. By doing this, we reduce the number of constraints in  $P_{reg}$ ; therefore, some potential fails and unwanted redundancy are removed.

Unwanted redundancy means implicit redundancy between two (or more) constraints which leads to a slow down of the solving process.

Do not confound this kind of redundancy with the positive redundancy which is often called in the literature. We expect that not such positive redundancy constraints were added to the model before so that all the negative redundancy can be removed first then we add the positive redundancy. The interplay of positive and negative redundancy would also be an exciting research topic.

These optimizations can be repeated until only one regular constraint is left ( $P''_{reg}$ ) but these may not be useful. The optimization steps are also very time-consuming. Reducing the rCSP to one with only one regular constraint needs maybe more time as solving the original CSP  $P$ . Mostly, it is very time improving to do some optimization steps and solve the rCSP then. To find the perfect point until which the optimization is useful or not is also one of our research areas and will be answered in the future.

The presented Meta CSOP  $P_{opt}$  is part of the optimization steps of the algorithm. It detects the constraints which should be combined (intersected) next. Using  $P_{opt}$  with a maximum sub-CSP size  $s$  lower as the size of the given CSP (*size(P)*) leads automatically to a stopping point in the optimization which is reached before all constraints are combined into one regular constraint.

*Remark:* the algorithm is not only but especially useful for regular constraints. It is also possible to substitute constraints for the *table* constraint or a set of different types of constraints.

## 4 COMBINING OF REGULAR CONSTRAINTS

This section explains how we combine two (or more) regular constraints into a new one.

We assume that we have a ordered rCSP  $P_{reg} = \{X, D, C\}$  like explained in 2.1. Given are two regular constraints  $c_1 = \text{regular}(X_1, M_1)$  and  $c_2 = \text{regular}(X_2, M_2)$  with  $c_1, c_2 \in C$  and the variables  $X_1 \subseteq X$  and  $X_2 \subseteq X$  where  $\forall k \in \{1, 2\} \forall i, j \in \{1, \dots, |X_k|\}, i < j | x_i$  occur earlier in  $c_k$  as  $x_j$ .

There are two cases. Case one: the two constraints cover the same set of variables ( $X_1 = X_2$ ). Case two:

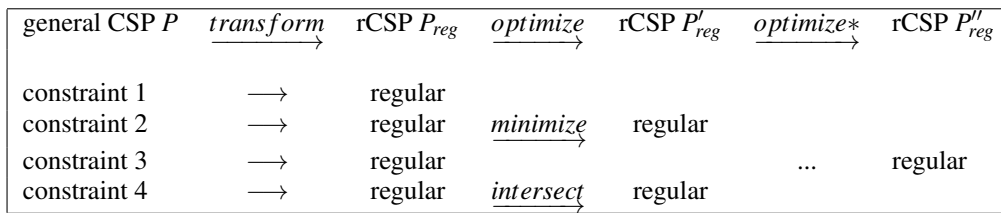


Figure 1: The regularization of CSPs.

the two constraints cover different sets of variables ( $X_1 \neq X_2$ ).

In the first case, we construct the automatons  $M_1$  and  $M_2$  and using then the intersection method of DFAs ( $\cap$ ). The resulting regular constraint is then  $c_{reg} = \text{regular}(X_1, M_1 \cap M_2)$ .

In case two we use the particular shape of the internally used automatons of regular constraints. An inputted automaton  $M$  to a regular constraint  $\text{regular}(X, M)$  will be transformed internally into an automaton  $M'$  which looks like  $M' = M \cap M_{all}$ , where  $M_{all}$  is an automaton which accept all words with length  $|X|$ .  $M'$  is a directed multigraph (Pesant, 2004) and can be partitioned into levels of states  $L = \{l_0, \dots, l_{|X|}\}$ . In each level  $l_i \in L$  are all states of  $M'$  which can be reached after reading  $i$  many variables.

For the combination of two regular constraints  $c_1 = \text{regular}(X_1, M_1)$  and  $c_2 = \text{regular}(X_2, M_2)$  with internal used automatons  $M'_1$  and  $M'_2$ , we search the first index  $i$  where the  $i$ -th variable  $x_i \in X_1$  is not equal to the  $i$ -th variable  $x_i \in X_2$  or vice versa.

Without loss of generality, we assume that  $x_i$  is in  $X_1$  and not in  $X_2$ . For each state  $s_{i-1,k}$  in  $M'_2$  at level  $i-1$  we create a new state  $s_{new,k}$  and change all transitions from all states  $s_{i-1,k}$  to their destination states, in a way that they starts now from state  $s_{new,k}$ . After these, we add the transitions from  $s_{i-1,k}$  to  $s_{new,k}$  with every value in the alphabet of  $M'_1$ .

This will be repeated for all variables  $x$  which are only in  $X_1$  or  $X_2$  and not in the other. Doing this leads to two new automatons  $M'_{new1}$  and  $M''_{new2}$  which can be intersected with the DFA intersection method. The resulting regular constraint is then  $c_{reg} = \text{regular}(X', M'_{new1} \cap M''_{new2})$ , where  $X'$  is the sorted union of  $X_1$  and  $X_2$ .

## 5 EXPERIMENTAL RESULTS

In this section, we present our experimental results of our Meta CSOP when applied to a random benchmark suite. All the experiments are set up on a DELL laptop with an Intel i7-4610M CPU, 3.00GHz, with 16 GB 1600 MHz DDR3 and running under Windows

7 professional with service pack 1. The algorithms are implemented in Java under JDK version 1.8.0\_171 and Choco Solver (Prud'homme et al., 2017).

We create randomly rCSPs with different number of variables (50, 100, 150), different domain sizes (5 or 10), different number of constraints (1 or 1.5 times the number of variables), different maximum sub-CSP size (10,000, 100,000, 1,000,000) different maximum number of sub-CSPs (5 or 10), different time for optimization (1s or 3s), different solution ratio for each regular constraint (0.2-0.4 or 0.4-0.6 or 0.6-0.8) and a solving time of 5 minutes.

With these parameter settings, we try to cover a big area of CSPs with nonempty solution space because CSPs with no solution can be solved very fast often.

We created for each parameter combination rCSPs and solved it without and with regularization. For the regularization we limit the solving time of the Meta CSOP by 1 second and 3 seconds and limit the solving process of the regularized model by 5 minutes. We also solved the original problem without regularization and limit the time of the solving process by the total time which was needed for transforming and solving of the regularization model to got a fair comparison.

The regular constraints cover between 2 and 4 variables and are randomly created. A random regular constraint for variables  $X$  with domains  $D$  was created in the following way.

- Create a matrix  $M \in |X| \times |\cup D|$  which contains all value combinations for the variables  $X$  respectively their domains  $D$ .
- Remove rows from  $M$  until only the given solution ratio (0.2-0.4 or 0.4-0.6 or 0.6-0.8) is left.
- Create an automaton which accepts exactly the inputs given in  $M$ .

With these randomly generated constraints we simulate a not optimized CSP with small constraints. We expect that the given CSP is not modelled in a perfect way so that our regularization approach can optimize it.

Table 1 shows the results of our test benchmark. We only consider the results for one second as optimization time because the results for three seconds

Table 1: The improvements of regularization.

Selection	Programs	Relevant	Failure	Improve.	Constant	Deterior.	Average improve. (%)
All	99	73	3	39	3	28	1.849
Top 10%	99	60	0	39	2	19	3.014
Top 20%	99	26	0	23	2	1	8.297

were not significantly different. We created 99 random regular programs like described before from which 73 were relevant because they were complex enough that the problem was not solved in less than five minutes. For 3 of these 73 relevant programs could not found a useful regularization in one second. For the other 70 programs we speed up the solving process in 39 cases, had 3 times no difference and decrease 28 times the solving speed. All in all we observed an average improvement of 1.849% (inclusive all transformation times).

This sounds not very successfully but it is clear that this approach is not useful for all kind of rCSPs. May some of them were modeled efficiently before so our approach decrease the solving speed for the reason that the transformations also need time. Another reason can it be that the rCSP has to less overlapping constraints which can be combined.

So the question is can we find out for which kind of rCSP our approach is useful in short time? The answer is yes, we can! In line 2 and 3 of table 1 you can see that the average improvement increase if we only consider the programs in which the combination of constraints reduces the total number of constraints by at least 10% respectively 20%.

With our Meta CSOP we detect if a rCSP can be reduced by 10% or 20% in one second. If it is useful we can automatically decide to combine the detected regular constraints and so our rCSP will be improved by 3.014% respectively 8.297%. Where is no risk to slow down the system significantly. If we cannot find a reduction of constraints which is big enough then we do not use the transformations and we only loss one second for checking this. Otherwise we can do the combinations and so we will mostly improve the solving speed. All other optimizations like adding positive redundancy or parallelization can still be executed afterward.

*Remarks:* For some of the researchers 8.297% may sounds also not very improvement. But consider that we used a very general approach in the Meta CSOP based on the size of sub-CSPs. If we specify this in a way that we do not use the size of the sub-CSPs but the size of the automaton and the potential size of the new created automaton then it allows us to choose constraints with a higher precision and with a higher number of variables.

The biggest influence on the number of constraints

which can be combined (and so the biggest influence on the solving speed) has the parameter sub-CSP size. What is not shown in the table is that all models which where created with sub-CSP size equal 1,000,000 reduced the number of constraints at least by 10%.

## 6 CONCLUSION AND FUTURE WORK

We have presented a new way to combine a set of constraints of a rCSP to one new regular constraint by the use of a Meta CSOP. The Meta CSOP is specialized to find sub-CSPs with a maximum size which are predestined for the substitution with a regular constraint. We also have shown how a set of regular constraints can be combined to one new regular constraint by the use of the automaton intersection method. We evaluated our approach by a random benchmark suite.

The experimental results showed that rCSP could be replaced with fewer constraints in a short time. Furthermore the experimental results show that our approach speedups the solving process of special rCSPs.

Future work will include a more specific side condition for the Meta CSOP based on the size of the automaton and not of the sub-CSP size. We also will distinguish between positive and negative redundancy and explain the advantages and disadvantages of constraint combination in detail. Furthermore a comparison between table and regular constraint looks promising. Both constraints are powerful enough to substitute each other constraint in a CSP. It would be interesting to analyze which one is in which situations better.

## REFERENCES

- Apt, K. (2003). *Principles of Constraint Programming*. Cambridge University Press, New York, NY, USA.
- Dechter, R. (2003). *Constraint processing*. Elsevier Morgan Kaufmann.
- Löffler, S., Liu, K., and Hofstedt, P. (2017). The power of regular constraints in csps. In *47. Jahrestagung der Gesellschaft für Informatik, Informatik 2017, Chemnitz, Germany, September 25-29, 2017*, pages 603–614.

- Löffler, S., Liu, K., and Hofstedt, P. (2018). The regularization of cps for rostering, planning and resource management problems. In *Artificial Intelligence Applications and Innovations - 14th IFIP WG 12.5 International Conference, AIAI 2018, Rhodes, Greece, May 25-27, 2018, Proceedings*, pages 209–218.
- Marriott, K. (1998). *Programming with Constraints - An Introduction*. MIT Press, Cambridge.
- Pesant, G. (2004). A regular language membership constraint for finite sequences of variables. In Wallace, M., editor, *Principles and Practice of Constraint Programming - CP 2004*, volume 3258 of *Lecture Notes in Computer Science*, pages 482–495. Springer.
- Prud'homme, C., Fages, J.-G., and Lorca, X. (2017). Choco documentation.
- Rossi, F., Beek, P. v., and Walsh, T. (2006). *Handbook of Constraint Programming*. Elsevier, Amsterdam, First edition.
- Tsang, E. P. K. (1993). *Foundations of constraint satisfaction*. Computation in cognitive science. Academic Press.

