ML-based Decision Support for CSP Modelling with Regular Membership and Table Constraints

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

Keywords: Constraint Programming, CSP, Refinement, Optimizations, Regular Membership Constraint, Table Constraint, ML, Machine Learning.

Abstract: The regular membership and the table constraints are very powerful constraints which allow it to substitute every other constraint in a constraint satisfaction problem. Both constraints can be used very flexible in a huge amount of problems. The main question we want to answer with this paper is, when is it faster to use the regular membership constraint, and when the table constraint. We use a machine learning approach for such a prediction based on propagation times. As learning input it takes randomly generated constraint problems, each containing exactly one table resp. one regular membership constraint. The evaluation of the resulting decision tool with specific but randomly generated CSPs shows the usefulness of our approach.

1 INTRODUCTION

Since the search space of constraint satisfaction problems (CSPs) is very big and the solution process often needs an extremely high amount of time we are always interested in a speed-up of the solution process. There are various ways to describe a CSP in practice and consequently, the problem can be modeled by different combinations of constraints, which results in the differences in resolution speed and behavior.

The regular membership and the table constraints are both very powerful constraints which allow it to substitute every other constraint in a constraint satisfaction problem. Both constraints can be used very flexible in many problems. In this paper we try to answer the question whether a regular or a table constraint is more suitable in a given CSP, based on machine learning.

For our test series we create random CSPs with only one constraint in two different ways with the expectation, that the CSPs created by one way are faster solvable with table constraints, and the others are faster solvable with the regular constraint. Finally, we evaluate our approach with another random set of CSPs and give an outlook of our aims for the future.

In this paper we will use the notion of a "*regular constraint*" synonymously for "*regular membership constraint*" or "*regular language membership constraint*".

2 PRELIMINARIES

In this section, we introduce necessary definitions and methods of constraint programming for our machine learning approach. We consider CSPs which are defined in the following way:

CSP. A constraint satisfaction problem (CSP) is defined as a 3-tuple $P = (X, D, C)$ where $X = \{x_1, x_2, \ldots, x_n\}$ is a set of variables, $D = \{D_1, D_2, \ldots, D_n\}$ is a set of finite domains such that D_i is the domain of x_i, and $C = \{c_1, c_2, \ldots, c_m\}$ is a set of constraints.
Constraint. A constraint $c_j = (X_j, R_j) \in C$ is a relation R_j, which is defined over the variables $X_j \subseteq X$ of the constraint c_j (Dechter, 2003).

Next, we define the two essential constraints for this work, the regular constraint and the table constraint. For the definition of the regular constraint we first need the definition of a deterministic finite automaton (DFA).

Deterministic Finite Automaton. A deterministic finite automaton M is a five tuple $(Q, \Sigma, \delta, q_0, F)$, where:

Löffler, S., Becker, I. and Hofstedt, P.
ML-based Decision Support for CSP Modelling with Regular Membership and Table Constraints.
DOI: 10.5220/0010299109740981
In *Proceedings of the 13th International Conference on Agents and Artificial Intelligence (ICAART 2021) - Volume 2*, pages 974-981
ISBN: 978-989-758-484-8

- Q is a finite set of states,
- Σ is a finite alphabet,
- δ is a transition function $Q \times \Sigma \to Q$,
- $q_0 \in Q$ is a initial state,
- and $F \subseteq Q$ is a set of final states.

A word $(w_1 w_2 \dots w_n) = w \in \Sigma^*$ is accepted by an automaton M iff w is element of the language, which is described by the automaton: $w \in L(M)$.

In this paper, the considered automatons have a certain structure: The set of states Q can be be partitioned into $Q = Q_0 \cup ... \cup Q_{n+1}$, such that transitions are only possible form a state of level Q_i to a state of level Q_{i+1}. Furthermore, Q_0 contains only the initial state q_0 and Q_{n+1} contains only the singleton final state q_{final}.

Maximum Width. The maximum width of such an automaton M corresponds to the maximum number of states in one level:

$$maxWidth(M) = max(|Q_i| \; \forall i \in \{0,...,n\}). \quad (1)$$

Regular Constraint. The regular constraint guarantees for a DFA $M = (Q, \Sigma, \delta, q_0, F)$ and an ordered set of variables $\{x_1,...,x_n\} = X' \in X$ with domains $\{D_1,\dots,D_n\} = D' \subseteq D$, where $D_i \subseteq \Sigma \mid \forall i \in \{1,\dots,n\}$, that for each variable assignment $\phi_j(x_1,...,x_n) = (d_{j,1},...,d_{j,n})$ the word $d_{j,1}...d_{j,n}$ is accepted by the automaton M (van Hoeve and Katriel, 2006).

$$regular(\{x_1,...,x_n\}, M) = \{(w_1,\dots,w_n) \mid \forall i \in \{1,\dots,n\}, \text{where } w_i \in D_i, \; (w_1 \dots w_n) \in L(M)\}. \quad (2)$$

The table constraint is an intensively researched constraint (Bessière and Régin, 1997; Lecoutre, 2011; Lecoutre et al., 2015; Lecoutre and Szymanek, 2006; Lhomme and Régin, 2005; Mairy et al., 2014).

Table Constraint. A positive (resp. a negative) table constraint guarantees for an ordered set of variables $\{x_1,...,x_n\} = X' \in X$ and a list of tuples T, that for each variable assignment $\phi_j(x_1,...,x_n) = (d_{j,1},...,d_{j,n})$ the tuple $t_j = (d_{j,1},...,d_{j,n})$ must (not) be element of the tuple list T. For all positive table constraints follows accordingly:

$$table(\{x_1,...,x_n\}, T) := \{(d_{j,1},...,d_{j,n}) \in T \mid \forall j \in \{1,...,|T|\}\}. \quad (3)$$

For a speed comparison of two constraints c_1 and c_2, we must guarantee that both are *equivalent*. Thus, we define that two constraints $c_1 = (X^1, R^1)$ and $c_2 = (X^2, R^2)$ are *equivalent*, if they cover the same variables $X^1 \widehat{=} X^2$ and describe the same tuples with their relations R^1 and R^2.

Furthermore, two propagators p_1 and p_2 are *equivalent* if they correspond to *equivalent* constraints and propagate the same value eliminations for equal domains resp. domain changes.

2.1 A Regular and a Table Propagator

Both the regular and the table constraint can represent every other constraint and they both reach the same consistency level: Generalized Arc Consistency (GAC) [1]. Thus, they are suitable as competitors for each other.

We use the regular propagator introduced by Gilles Pesant in (Pesant, 2004), for further details we refer to the original source.

The table constraint is implemented by different propagators across different solvers but also within the same solver. For example the Choco Solver (Prud'homme et al., 2017) alone has more than ten different propagators for the table constraint. It follows a list of acronyms, under which the algorithms are known in literature:

- AC2001, AC3, AC3rm, AC3bit+rm, CT, FC, GAC2001, GAC2001+, GAC3rm, GAC3rm+, GACSTR+, MDD+, and STR2+.

The Compact Table algorithm (CT) was chosen here because it is very effective and amongst others default in the Choco Solver (for positive tuple sets with more than 500 tuples), in Oscar (OscaR, 2018) and in the Google OR Tools (LLC, 2019).

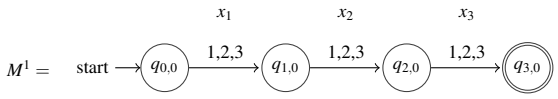
The CT algorithm was introduced in (Demeulenaere et al., 2016) and will only be mentioned here. For a description and details we suggest the original source.

Since the regular propagator and the CT propagator both reach GAC, and they correspond to equivalent constraints (regular and table constraint), they are also equivalent.

3 COMPACTNESS OF REGULAR CONSTRAINTS

The Hypotheses 1 and 2 (see below) for our machine learning approach suggest that the ratio of the number of tuples $|T|$ of a table constraint $c_t = (X, T)$ by the number of transitions $|M.\Delta|$ resp. the width of the corresponding automaton M of an equivalent regular constraint $c_r = (X, M)$ can be indicators for the constraint type which solves the problem faster. We call

[1] At least the two propagators we use for our approach.



$T^1 = \{\{1,1,1\},\{1,1,2\},\{1,1,3\},\{1,2,1\},\{1,2,2\},$
$\{1,2,3\},\{1,3,1\},\{1,3,2\},\{1,3,3\},\{2,1,1\},$
$\{2,1,2\},\{2,1,3\},\{2,2,1\},\{2,2,2\},\{2,2,3\},$
$\{2,3,1\},\{2,3,2\},\{2,3,3\}.\{3,1,1\},\{3,1,2\},$
$\{3,1,3\},\{3,2,1\},\{3,2,2\},\{3,2,3\},\{3,3,1\},$
$\{3,3,2\},\{3,3,3\}\}$

Figure 1: The tuple set T^1 and the automaton M^1 for two equivalent constraints $c_t^1 = (x_1, x_2, x_3, T)$ and $c_r^1 = (x_1, x_2, x_3, M)$, with a compact automaton M.

this ratios CiT (compactness in transitions) resp. CiW (compactness in width):

$$CiT = |T|/|M.\Delta|, \tag{4}$$

$$CiW = |T|/width(M). \tag{5}$$

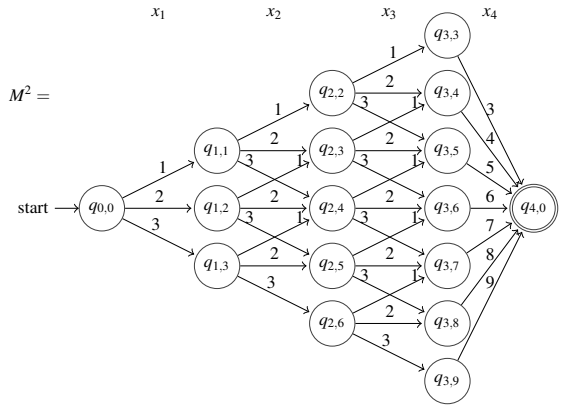
Hypothesis 1. Given a regular constraint $c_r = (X, M)$ and an equivalent positive table constraint $c_t = (X, T)$. If the CiT value for these constraints exceeds a certain threshold, then using the regular constraint c_r is more promising than the table constraint c_t, and otherwise vice versa.

Hypothesis 2. Given a regular constraint $c_r = (X, M)$ and an equivalent positive table constraint $c_t = (X, T)$. If the CiW value for these constraints exceeds a certain threshold, then using the regular constraint c_r is more promising than the table constraint c_t, and otherwise vice versa.

Both hypotheses are motivated by the fact that for finding one solution a table constraint has to unselect all tuples but one via the supports and the regular constraint has to remove all transitions of all paths from the initial state to the final state of the automaton but one (if using the mentioned propagators).

The Figures 1 and 2 show the automatons M^1 and M^2 of regular constraints c_r^1 and c_r^2, and the tuples T^1 and T^2 of equivalent table constraints c_t^1 and c_t^2. The constraints c_r^1 and c_t^1 represented in Figure 1 allow all possible combinations of the values 1, 2, and 3 for the variables x_1, x_2, and x_3, whereas, the constraints c_r^2 and c_t^2 shown in Figure 2 allow all possible combinations of the values 1, 2, and 3 for the variables x_1, x_2 and x_3, where the sum of the variables x_1, x_2 and x_3 is equal to a variable $x_4 \in \{3, ..., 9\}$.

In both cases, there are 27 solutions, but in the first case it's a very compact automaton (with only 9 transitions) and in the second case a bigger automaton (with 34 transitions). So in case one, six transitions must be removed for the instantiation of a solution, while in case two this are 30 transitions. In each case, 26 supports (representing the tuples) must be removed in the corresponding table constraint.



$T^2 = \{\{1,1,1,3\},\{1,1,2,4\},\{1,2,1,4\},\{2,1,1,4\},$
$\{1,1,3,5\},\{1,2,2,5\},\{1,3,1,5\},\{2,1,2,5\},$
$\{2,2,1,5\},\{3,1,1,5\},\{1,2,3,6\},\{1,3,2,6\},$
$\{2,1,3,6\},\{2,2,2,6\},\{2,3,1,6\},\{3,1,2,6\},$
$\{3,2,1,6\},\{1,3,3,7\},\{2,2,3,7\},\{2,3,2,7\},$
$\{3,1,3,7\},\{3,2,2,7\},\{3,3,1,7\},\{2,3,3,8\},$
$\{3,2,3,8\},\{3,3,2,8\},\{3,3,3,9\}\}$

Figure 2: The tuple set T^2 and the automaton M^2 for two equivalent constraints $c_t^2 = (x_1, x_2, x_3, x_4, T)$ and $c_r^2 = (x_1, x_2, x_3, x_4, M)$, with a big automaton M.

Because of the different values for $CiT_1 = 27/9$ vs. $CiT_2 = 27/34$ and $CiW_1 = 27/1$ vs. $CiW_2 = 27/7$, CiT and CiW both beeing bigger for the first problem, both hypothesis suggest that in the first case, the regular constraint and in the second case the table constraint propagates relatively faster.

4 GENERATION OF RANDOM PROBLEM INSTANCES

In this section, we explain the generation of the training data for the machine learning approach.

The main idea is to generate CSPs $P^{all} = \{P^t \cup P^r \cup P^{t*} \cup P^{r*}\}$, which contain only one table (P^t) or regular (P^r) constraint, and then to create an equivalent CSP $P_i^{r*} \in P^{r*}$ for each CSP $P_i^t \in P^t$, which uses a regular constraint instead of a table constraint, and vice versa. All other settings, like search strategies, propagation order etc. are identical for $P_i^t \in P^t$ and $P_i^{r*} \in P^{r*}$ ($P_j^r \in P^r$ and $P_j^{t*} \in P^{t*}$ resp.), so that the time difference in the solution process should result exclusively from the propagation time of the chosen constraint. Thus, the real impact of the used constraints can be checked. In the following, we explain how we create our random CSPs in P^t and P^r, to guarantee the quality and transparency of our data.[2]

[2]The source code used to generate the samples can be found at: https://git.informatik.tu-cottbus.de/loeffsve/

Table 1: The parameters for the generation of CSPs.

Parameter	Range of values
Number of solutions	{1,2, ..., 25000}
Number of variables	{2,3, ..., 21}
Domain Base	{2,3, ..., 200}
Δ domain size	{0, 1, ..., *domainSizeBase*-1}
Ratio of leaks	0-95%
Maximum width	{1,2,...,50}

The CSPs in P^t are created fully randomly, such that the solutions of such a CSP $P_i^t \in P^t$ (resp. $P_i^{r*} \in P^{r*}$) are also completely random (e.g. in number and distribution). In contrast to this, the CSPs in P^r (resp. P^{t*}) are created in a way, such that the used automaton M_i of each CSP $P_i^r \in P^r$ can be represented very compactly ($width(M) \leq 50$).

According to our hypothesis introduced in Section 3, we expect that the CSPs originally created with a regular constraint can be solved faster faster in its regular version, and the CPS originally created with a table constraint can be solved faster with a table constraint on average. Furthermore, we hope to find more accurate hypotheses for the prediction of usefulness of table and regular constraints.

To have a meaningful data set we created 1000 CSPs originally with table constraints and 1000 CSPs originally with regular constraints.

For the generation, solving, and learning we used a computer with a 6 core CPU, 12 threads, at 2.40GHz, with 32 GB RAM memory (2667 MHz), running the Canonical Ubuntu OS in version 18.04.1. The algorithms are implemented in Java under JDK version 1.8.0_231.

For the randomization, we used the *Random* class of the *java.util* package, which uses a 48-bit seed, which is modified using a linear congruential formula, as explained in (Knuth, 1997, Section 3.2.1.). As the initial seed for the problem generation the problem number (1 to 1000) was used. We used the randomly instantiated parameters for the problem generation as presented in Table 1 (in case of table generation the *maxWidth* parameter was not used).

The number of solutions is also the number of tuples in the table constraint and also the number of paths from the initial state to the final state in the automaton of the corresponding regular constraint. The *domainSizeBase* is a base value and the actual domain size of each variable is calculated based on *domainSizeBase* plus or minus a random value between 0 and the Δ*DomainSize* value.

The ratio of leaks is the average of the number of domain values $|D_i|$ divided by the range size of the domain $max(D_i) - min(D_i) + 1$ for all domains $D_i \in D$.

The following subsections explain the creation of the CSPs in P^t and P^r.

4.1 Table based Problem Instances

Algorithm 1 shows the interaction of the parameters for the creation of the variables, or more precisely, for the creation of the domains of the variables. Each entry $values[i]$ represents the domain D_i. The algorithm follows exactly the description of the parameters. Each domain size is in an Δ*DomainSize* (ΔDS) range around the *domainSizeBase* (dB) (Line 3) and the possibility for leaks is equal to r (Line 5 to 7). After the generation of the variables with Algorithm 1, $nbTuples$ different tuples will be generated randomly based on the generated values.

Algorithm 1: *createVariables*.

```
   Data: int: nbVars, dB, ΔDS
         double: r
   Result: int[][]: values
1  int[][] values = new int[nbVars]
2  forall (int i = 0; i < nbVars; i++) do
3  |  values[i] = randomInt(−ΔDS, ΔDS) + dB
4  |  forall (int idx = 0, v = 0;
   |     idx < values[i].length; v++) do
5  |  |  if (randomDouble() > r) then
6  |  |  |  values[i][idx] = v
7  |  |  |  idx++
8  return values
```

4.2 Regular based Problem Instances

For the regular based CSP generation the solutions are not directly generated, rather the automaton M is generated and subsequently checked, to whether the number of paths from the initial state to the final state (so the number of solutions) is in an acceptable range. The *maxWidth* parameter limits the maximum width of the automaton and guarantees that M is compact.

Algorithm 2 shows the generation of automatons for the CSPs P^r. The algorithm takes the domains $D = \{D_1, ..., D_n\}$, the number of variables *nbVars* and the maximum width of the automaton *maxWidth* as input and returns the states of the created automaton in a two dimensional array q, where each sub-array $q[i]$ corresponds to the states Q_i of level i.

First, the variables are instantiated: The value lNS, standing for the number of states of the last level,

code-generation-table-vs-regular.

Algorithm 2: *createAutomaton*.

```
   Data: Domains D = {D_1, ..., D_n}
         int nb_Vars, maxWidth
   Result: Node[][] q
1  repeat
2      int lNS = 1
3      int[] nb_Trans = new int[nb_Vars]
4      Node[][] q = new Node[nb_Vars + 1][]
5      q[0][0] = createFirstState()
6      q[n][0] = createFinalState()
7      forall (int i = 1; i < nb_Vars; i++) do
8          int cNS = randomInt(1, min(lNS *
             |D_{i-1}|, maxWidth))
9          q[i] = createStates(cNS, i)
10         lNS = cNS
11         nb_Trans[i - 1] =
             randomInt(1, min(lNS *
             |D_{i-1}|, 100))
12     nb_Trans[nb_Vars - 1] =
         randomInt(1, min(lNS *
         |D_{nb_Vars-1}|, 100))
13     addTransitions(nb_Trans, D)
14     if (maxWidth > 5) then
15         maxWidth--
16 until checkNbTuples(q);
17 return q
```

is instantiated with value 1. This represents that at level 0 is only one state ($q_{0,0}$, Line 2). Each entry of the integer array *nbTrans* represents the number of transitions from level i to level $i+1$ (Line 3). The two dimensional node array q will contain the states of the automaton and is consequently instantiated with length $n+1$ (Line 4).

For each level $i \in \{1 \text{ to } nbVars\}$, a random number of states is created and added to the *states* array (Lines 8 and 9). The number of states for each level is always between one and the minimum of the *maxWidth* value and the number of states from the last level (*lNS*) multiplied by the domain size of the corresponding variable ($|D_{level-1}|$), because this is the maximum number of states a deterministic automaton (like we use) can reach in a level.

Afterwards, a random number between one and 100 (resp. the maximum number) is generated, which represents the number of transitions from level $i-1$ to level i (Lines 11 and 12). The bounds *maxWidth* and 100 guarantee that the automaton will not be too big.

The *addTransitions* method then adds the previously generated number of transitions to each level (Line 13) and removes all states which are not part of at least one path from the initial state to the current resp. the final state.

The *checkNbTuples* method counts all paths from the initial state to the final state, which corresponds to the number of solutions resp. the number of tuples of an equivalent table constraint and returns *true* if the number of solutions is neither too big nor to small (Line 16). A new automaton will be generated randomly until the number of solutions is acceptable. In most cases the automaton will have too many solutions and not few. To reduce the number of solutions, we reduce the maximum width step by step down to a minimum of five (Lines 14 and 15).

The bounds, which limit the width of the automaton and the number of transitions lead to the point that automaton generated in this way is more compact than automatons, which are created from randomly generated tuples. Consequently, we expect that the CSPs in P^r propagate faster with the regular constraint and the CSPs in P^t propagate faster with the table constraint, on average.

5 MACHINE LEARNING FOR DECISION MAKING

In the following, we briefly outline the creation of classifiers with standard machine learning techniques in our approach.

5.1 Methodology

In order to create learning data we solved each problem instance, both from our tuple based, as well as automaton based instances, with both types constraints (table and regular). For each run we calculated and stored the problem features, propagator state features, and the run times. This provides a set of samples of the following form:

$$(id, \text{used constraint}, (f_1, \ldots, f_n), (r_1, \ldots, r_m)),$$

where f_i is a recorded feature as described in Section 5.2, r_j is a recorded result (e.g., run time in *ns*), used constraint $\in \{\text{table}, \text{regular}\}$, and *id* being a tuple:

$$(\text{seed for problem generation}, \text{modelling base}),$$

$$\text{modelling base} \in \{\text{tuple based}, \text{automaton based}\}.$$

Since the samples represent the following Cartesian product:

$$\{\text{modelling base}\} \times \{\text{used constr.}\} \times n\ (= 1000)$$

we end up with a dataset of 4000 run samples (P^{all}).

Since each problem *id* is run twice, with either a tabular (P_i^t, P_j^{t*}) or regular constraint (P_i^{r*}, P_j^r), this reduces to 2000 actual problem samples (1000 for each modelling base). For each problem *id* we then calculate a number of features, such as the speedup of the faster vs. the slower constraint, and which constraint was the fastest. For these we also deduce the classes we want to predict based upon this data. As of now we try to classify samples where the speedup was bigger than 1.5 according to the faster constraint ("regular", "table"). The rest we try to identify as "irrelevant". The goal of this distinction is to avoid unnecessary transformations when applying this to real problems.

We furthermore prepare the data by standardizing the features using the Scikit-Learn `StandardScaler`. We then compose tables suitable for machine learning resulting in samples of the form:

$$(id, (f_1, \ldots, f_n), class \in \{\text{irrel, regular, table}\}),$$

with f_i being the inputs for our ML approach, and *class* being the label the classifier is supposed to predict.

In the next step, we separate a random subset of 30% of the original data set (by *id*) for testing. The remaining 70% are used for training. This results in a training set of 1400 samples and a testing set of 600 samples. From here we utilise machine learning to train a classifier. We use the Scikit-Learn library for easy-to-use, reliable, and proven machine learning (Pedregosa et al., 2011). From the vast selection of models available in Scikit-Learn we chose the Random Forest Classifier (RFC) for this experiment. RFCs were introduced by (Breiman, 2001), but instead of letting each decision tree vote on the class the scikit-learn implementation averages the probabilistic prediction of each tree in the ensemble (scikit-learn developers, 2020). RFCs tend to deliver reasonably good results with little hyperparameter optimization (Hastie et al., 2009, p. 587 ff.). The RFC is configured with the default parameters provided by Scikit-Learn in version 0.23.

5.2 Problem Features

We collect a set of measurements from the CSP we aim to make a decision for. In total we collect 11 features which are easy to calculate and which we suspect to possibly be indicators of the CSPs suitability for one constraint or the other.

Problem Features. Features inherent to the problem:

- Variable and solution tuple count;
- Minimal, maximum, and average domain size, domain size delta;
- Average number of leaks and avg. rate of leaks;
- Minimal, maximum, and average domain offset to zero.

Propagator Features. Features taken from the propagators internal state:

- Number of zeros and ones in supports for table constraint (cf. (Demeulenaere et al., 2016));
- CiT and CiW, Number of states, edges, and width, minimal, maximum and avg. branching degree with respect to the internal DFA, as well as the number of supports for the regular constraint.

5.3 Evaluation of the Problem Instances

We evaluated the classifiers against the previously separated testing subset, which consists of 291 `table`, 220 `regular` and 89 `irrel` samples. To evaluate the classifiers against the testing data we trained the classifiers on the training data and then apply it to the test data. We trained multiple classifiers on different subsets of features. The first subset only contains problem features. The second subset also includes the propagator features. In an application scenario this would require to first instantiate both propagators, however this can be done efficiently. We then calculated the following metrics based on the predictions and correct classification data:

Precision. The precision is the ratio of true positives to the total number of positively predicted samples. It encodes how well the classifier differentiates the respective label against other labels. A high precision on *regular* and *tabular* labels is desirable, in order not to actively transform a model using a slower constraint.

Recall (Sensitivity). The recall value is the ratio of true positives to the total number of positives. It encodes how sensitive the classifier is in picking up on a certain label. Of special interest is the recall on the *regular* and *tabular* label, as these encode the sensitivity to improvement. However sensitivity is less important than precision, as it's better to miss some improvements than to actively worsen a model.

F1-Score. The F1-Score encodes a weighted average of precision and recall. It is calculated as follows:

$$F1 = 2 * (precision * recall) / (precision + recall).$$

For all metrics holds that 1 is the best value and 0 the worst. The evaluation results for all tested classifiers are displayed in Table 2 and in the confusion matrices in Figure 3. Table 2 displays the calculated metrics as

Table 2: Classification metrics of different classifiers tested against the test data set.

	Precision	Recall	F1-Score	Support
	Problem Features Only			
Irrel.	19.10%	36.17%	25.00%	47
Reg.	14.55%	15.31%	14.92%	209
Tab.	29.21%	24.71%	26.77%	344
	Problem and Propagator Features			
Irrel.	70.79%	91.30%	79.75%	69
Reg.	98.18%	100.0%	99.08%	216
Tab.	99.31%	91.75%	95.38%	315

well as the number of instances classified correspondingly (*support*). Figure 3 visualizes how the classifications are distributed. Ideally all fields are zero but the top-left to bottom-right diagonal. For the classifier based on problem features only there are many misclassfied cases, especially between tabular and regular instances. These misclassifications between *regular* and *tabular* are especially undesirable, as they actively worsen a model. In the matrix for the classifier that includes propagator features we see that there are isolated cases of regular and tabular labels misclassified as irrelevant (left column) and a number of irrelevant instances classified as tabular (top-right field).

Two things become evident when looking at the data: First, while the classifier trained on problem features only does not perform well at all, the classifier that also learns on propagator features performs much better. Second, it appears feasable to predict the better peforming constraint from features extractable from the problem and propagators built to represent it. While the classification is not perfect yet, it does generalize well from the training data onto the test data set. Furthermore, the biggest classification inaccuracies occur with the *irrelevant* class, which naturally does not result in much lost time but the one spent on classification.

The classification takes less than two milliseconds (1.17*ms* on average). Our generated problems have a median run time of 3.25*ms*, while the fastest models have a median run time of 2.21*ms*. This also does not include the feature calculation time, so upon first glance the classification time appears not worth the trade off. However the classification needs to happen only once for a constraint within a real problem and only depends on the RFCs tree depth and count and therefore should scale well across bigger problems. We also believe that for a real problem with larger run times the repeated propagation effort results in a well scaling speedup, with the increasing absolute savings making the classification time irrelevant.

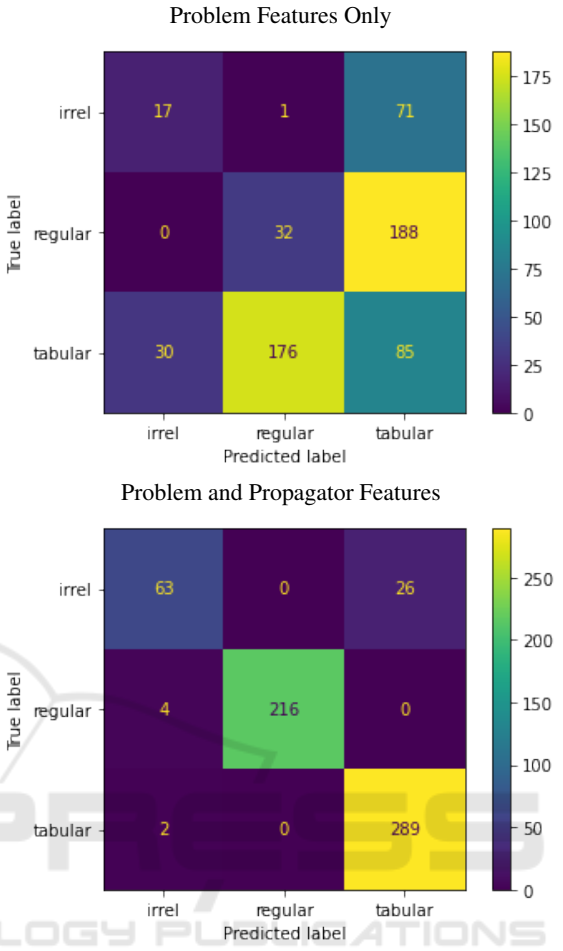


Figure 3: Confusion Matrices over the Classification of the Generated Test Problems.

The Random Forest Classifier (RFC) implemented in Scikit-Learn allows a closer look at the importance of features within the decision trees. While these values are to be taken with a grain of salt (see (scikit-learn developers, 2020)), the forests appear to put high importance on the features of the regular constraint (especially CiT and CiW) and table constraint, as well as the number of variables, supporting the initially stated hypotheses.

In conclusion we proved that it is possible to predict with some certainty whether the tabular or regular constraint will propagate faster based on problem features extracted before attempting to solve the problem. We suspect that these predictions generalize to constraints embedded in real problems and allow to reduce their runtimes.

6 CONCLUSION AND FUTURE WORK

We showed that an ML based approach can be used to predict whether a regular or a table constraint works faster in a CSP with a single constraint. A test bench with 2000 sample CSPs was generated and tested. Based on these results a classifier was trained which managed to distinguish models that profit from one or the other constraint with reasonable precision.

The next step based upon the insights learned with these experiments is to examine, whether the predictions can help speed up solving real problems. Of further interest would be, whether and how the features extracted from the generated problems differ to the ones obtained from subsets of real problems. We would also like to explore further optimizations in the machine learning approach, as well as the possibility of deducing rule-based predictive models from the insights gained from the machine learning experiments.

REFERENCES

Bessière, C. and Régin, J. (1997). Arc consistency for general constraint networks: Preliminary results. In *Fifteenth International Joint Conference on Artificial Intelligence, IJCAI 97, 2 Volumes*, pages 398–404. Morgan Kaufmann.

Breiman, L. (2001). Random forests. *Machine Learning*, 45(1):5–32.

Dechter, R. (2003). *Constraint processing*. Elsevier Morgan Kaufmann.

Demeulenaere, J., Hartert, R., Lecoutre, C., Perez, G., Perron, L., Régin, J., and Schaus, P. (2016). Compact-table: Efficiently filtering table constraints with reversible sparse bit-sets. In Rueher, M., editor, *Principles and Practice of Constraint Programming - 22nd International Conference, CP 2016*, volume 9892 of *Lecture Notes in Computer Science*, pages 207–223. Springer.

Hastie, T., Tibshirani, R., and Friedman, J. H. (2009). *The Elements of Statistical Learning: Data Mining, Inference, and Prediction, 2nd Edition*. Springer Series in Statistics. Springer.

Knuth, D. E. (1997). *The Art of Computer Programming, Volume 2: Seminumerical Algorithms*. Addison-Wesley, Boston, third edition.

Lecoutre, C. (2011). STR2: optimized simple tabular reduction for table constraints. *Constraints An International Journal*, 16(4):341–371.

Lecoutre, C., Likitvivatanavong, C., and Yap, R. H. C. (2015). STR3: A path-optimal filtering algorithm for table constraints. *Artif. Intell.*, 220:1–27.

Lecoutre, C. and Szymanek, R. (2006). Generalized arc consistency for positive table constraints. In Benhamou, F., editor, *Principles and Practice of Constraint Programming - 12th International Conference, CP*, volume 4204 of *Lecture Notes in Computer Science*, pages 284–298. Springer.

Lhomme, O. and Régin, J. (2005). A fast arc consistency algorithm for n-ary constraints. In Veloso, M. M. and Kambhampati, S., editors, *The Twentieth National Conference on Artificial Intelligence and the Seventeenth Innovative Applications of Artificial Intelligence Conference*, pages 405–410. AAAI Press / The MIT Press.

LLC, G. (2019). Google LLC, Google OR-Tools, 2019.

Mairy, J., Hentenryck, P. V., and Deville, Y. (2014). Optimal and efficient filtering algorithms for table constraints. *Constraints An International Journal*, 19(1):77–120.

OscaR (2018). OscaR: Operational research in scala. https://bitbucket.org/oscarlib/oscar. last visited 2019-08-22.

Pedregosa, F., Varoquaux, G., Gramfort, A., Michel, V., Thirion, B., Grisel, O., Blondel, M., Prettenhofer, P., Weiss, R., Dubourg, V., VanderPlas, J., Passos, A., Cournapeau, D., Brucher, M., Perrot, M., and Duchesnay, E. (2011). Scikit-learn: Machine learning in python. *J. Mach. Learn. Res.*, 12:2825–2830.

Pesant, G. (2004). A regular language membership constraint for finite sequences of variables. In Wallace, M., editor, *Principles and Practice of Constraint Programming - CP, 10th International Conference*, volume 3258 of *Lecture Notes in Computer Science*, pages 482–495. Springer.

Prud'homme, C., Fages, J.-G., and Lorca, X. (2017). Choco documentation.

scikit-learn developers (2020). *scikit-learn user guide*, Release 0.23.2 edition.

van Hoeve, W.-J. and Katriel, I. (2006). *Global Constraints*. Elsevier, Amsterdam, First edition. Chapter 6.