An enhanced features extractor for a portfolio of constraint solvers

Roberto Amadini, Maurizio Gabbrielli, Jacopo Mauro

An Enhanced Features Extractor for a Portfolio of Constraint Solvers

Roberto Amadini
University of Bologna/INRIA

Maurizio Gabbrielli
University of Bologna/INRIA

Jacopo Mauro
University of Bologna/INRIA

ABSTRACT

Recent research has shown that a single arbitrarily efficient solver can be significantly outperformed by a *portfolio* of possibly slower on-average solvers. The solver selection is usually done by means of (un)supervised learning techniques which exploit features extracted from the problem specification. In this paper we present an useful and flexible framework that is able to extract an extensive set of features from a Constraint (Satisfaction/Optimization) Problem defined in possibly different modeling languages: MiniZinc, FlatZinc or XCSP.

Categories and Subject Descriptors

I.2 [**Computing Methodologies**]: Artificial Intelligence—*Constraint Programming, Machine Learning, Algorithm Portfolios.*

1. INTRODUCTION

It is well recognized within the field of Constraint Programming that different solvers are better when solving different problem instances, even within the same problem class [8]. It has also been shown in other areas, such as SATisfiability testing and Integer Linear Programming that the best on-average solver can be out performed by a portfolio of possibly slower on-average solvers. A portfolio approach [8] for constraint solving can be seen as a methodology that exploits the significant variety in performances observed in different algorithms and combines them in order to create a globally better solver, dubbed a *portfolio solver*. A crucial step for the performance of a portfolio solver is the selection of one or more solvers composing the portfolio for solving a specific problem instance. Such a selection process is usually performed by using Machine Learning techniques based on *features* extracted from the instances that need to be solved.

Although several portfolio approaches have been extensively studied and used in the SAT solving field (e.g. [19, 11]) to the best of our knowledge the only Constraint Satisfaction Problems (CSP) portfolio solver is CPHydra [17]. CPHydra deals with problems expressed only in the XCSP format [18], it uses a rather small portfolio (just 3 solvers) and exploits only a limited number of features (36 to be exact, extracted by using Mistral solver [12]). Even worse, we are not aware of a single portfolio solver for Constraint Optimization Problems (COPs). There are several reasons for the lack of CSP/COP portfolio solvers. First of all, the CSP/COP solving fields are more complex than SAT: constraints can be arbitrary complex (e.g. global constraints like *regular* or *bin-packing*) and some of them are supported by only a few solvers. Moreover, no standard input language nor immediately available big dataset exists for CSPs or COPs.

We deem that, at least from the implementation and tool development point of view, there is still a big gap between CSP/COP w.r.t. the SAT field. In this paper we then address this problem by presenting a framework that is able to extract an exhaustive set of 155 features from a CSP/COP instance specified in the MiniZinc format [16] but supporting also other formats like XCSP and FlatZinc [4] through a simple pre-processing phase. Hopefully this framework will be the starting point for the development of future CSP/COP portfolio solvers that rely on the extracted features for their solver selection procedure.

2. FRAMEWORK

In this section we present the technical details of our framework by introducing its main ingredients: the compiler `xcsp2mzn` and the features extractor `mzn2feat` (which includes the parser `fzn2feat`). Moreover, we provide a summary of the set of features that `mzn2feat` can extract.

Note that we decided to use MiniZinc as primary source format of our tool because it is nowadays the most used, supported and general language to specify constraint problems. MiniZinc supports also optimization problems and is the source format used in the MiniZinc challenge [14], the only surviving international competition to evaluate the performances of constraint solvers. Our framework however offers full compatibility with XCSP and FlatZinc. Indeed, on one hand, we developed a compiler `xcsp2mzn` for converting problem instances from XCSP to MiniZinc by preserving the most important global constraints. On the other hand, the feature extractor tool called `mzn2feat` supports natively the FlatZinc format and can extract the features from every FlatZinc model.

2.1 `xcsp2mzn`

While MiniZinc is nowadays the most used language to

encode CSPs, XCSP was mainly used in the past for the International Constraint Solver Competition (ICSC) [6], which ended in 2009. Nevertheless, the ICSC dataset is by far the biggest dataset of CSP instances existing today. Hence, in order to exploit such a dataset for building better portfolios, we developed a compiler from XCSP to MiniZinc. `xcsp2mzn` was developed by adapting `x4g` [13], a converter from XCSP to Gecode [7] used in particular to support the XCSP abridged notation. Since we focused mainly on CSP we did not consider XCSP extensions like weighted constraints or quantifiers over constraints. All the code is written in C++ using the well known `libxml2` libraries. Exploiting the fact that MiniZinc is more expressive than XCSP (i.e. the majority of the primitive constraint of XCSP are also primitive constraints of MiniZinc) the translation was straightforward. The only notable difference was the compilation of extensional constraints (i.e. relations explicitly expressed in terms of all the allowed or not allowed tuples) which are a native feature in XCSP only. To overcome this limitation we used the *table* global constraint for encoding the allowed set of tuples and a conjunction of disjunctions of inequalities for mapping the forbidden set of tuples. As far as global constraints are concerned, XCSP supports the majority of the global constraints defined in the Global Constraint Catalog [5]. Since in this catalog there are hundreds of global constraints, for simplicity we decided to support only the subset of the global constraints used in the ICSC.

2.2 `mzn2feat`

The tool `mzn2feat` allows to extract from a MiniZinc model a set of 155 features: 144 are static features obtained by parsing the source problem instance, while 11 are dynamic and are obtained by running the Gecode solver for a short run. Since the complexity of the MiniZinc language (in particular the possibility of using control flow statements) makes the extraction of the syntactical features quite difficult, we decided to not process directly the MiniZinc instances. We instead compile them to FlatZinc [15], a lower level language having a syntax that is mostly a subset of MiniZinc and that can be obtained from MiniZinc by using the `mzn2fzn` tool provided by the MiniZinc suite. The compilation to FlatZinc was performed using the Gecode redefinitions for global constraints. This allowed us to keep track of how and what global constraints are used without decomposing them.[1] To extract the static features we developed a parser, called `fzn2feat`, using the standard Flex and Bison tools. The dynamic features were collected instead launching Gecode interpreter `fz` for 2 seconds runs.

Summarizing, given a generic MiniZinc model M in input, `mzn2feat` does the following: *i)* first, translates M into the corresponding FlatZinc F_M specification by using Gecode global redefinitions; *ii)* extracts static features from F_M by means of `fzn2feat`; *iii)* extracts dynamic features from F_M by running the `fz` interpreter of Gecode for 2 seconds.
We remark that step *ii)* is applicable to every FlatZinc model F (possibly ignoring the unknown solver-specific redefinitions). Moreover, steps *ii)* and *iii)* are totally independent and therefore they could be parallelized or done in reverse order. For instance, it could be useless to compute the static features if the given instance is solved by Gecode while trying to compute the dynamic features.

[1]Without using solver specific redefinitions, during the compilation process some global constraints are indeed decomposed into basic constraints. For instance the `alldifferent` global constraint is decomposed by default into a conjunction of inequalities, from which it is impossible to uniquely recover the original global constraint.

2.3 Features description

In this section we present an overview of the numeric features extracted by `mzn2feat`; for a more detailed description we defer the interested reader to [2]. `mzn2feat` tries to collect a set of features as exhaustive and general as possible, taking inspiration from and adapting those presented in [9, 3]. Although some of these features are quite generic (e.g., the number of variables or constraints), others are specific to FlatZinc (e.g. search annotations) or to Gecode (the global constraints features). For more details about these technicalities please see [15, 4, 7].

Static Features

Thanks to `fzn2feat` we are able to extract 144 static features grouped in the different categories listed below. In the following we will denote with NV the number of variables and with NC the number of constraints of a given problem. Moreover, we will denote respectively by min, max, avg, CV, and H the minimum, maximum, average, variation coefficient and entropy of a set of values.

Variables (27): the number of variables NV; the number cv of constants; the number av of aliases; the ratio $\frac{av+cv}{NV}$; the ratio $\frac{NV}{NC}$; the number of *defined* variables (i.e. defined as a function of other variables); the number of *introduced* variables (i.e. auxiliary variables introduced during the FlatZinc conversion); sum, min, max, avg, CV, and H of the: variables domain size, variables degree, domain size to degree ratio.

Domains (18): the number of: boolean variables bv and the ratio $\frac{bv}{NV}$; float variables fv and the ratio $\frac{fv}{NV}$; integer variables iv and the ratio $\frac{iv}{NV}$; set variables sv and the ratio $\frac{sv}{NV}$; array constraints ac and the ratio $\frac{ac}{NC}$; boolean constraints bc and the ratio $\frac{bc}{NC}$; int constraints ic and the ratio $\frac{ic}{NC}$; float constraints fc and the ratio $\frac{fc}{NC}$; set constraints sc and the ratio $\frac{sc}{NC}$.

Constraints (27): the total number of constraints NC, the ratio $\frac{NC}{NV}$, the number of constraints using specific FlatZinc annotations; the logarithm of the product of the: constraints domain[2] and constraints degree; sum, min, max, avg, CV, and H of the: constraints domain, constraints degree, domain to degree ratio.

Global Constraints (29): the total number gc of global constraints, the ratio $\frac{gc}{NC}$ and the number of global constraints for each one of the 27 equivalence classes in which we have grouped the 47 global constraints that Gecode natively supports.

Graphs (20): once built the Constraint Graph CG and the Variable Graph VG we compute min, max, avg, CV, and H of the: CG nodes degree, CG nodes clustering coefficient, VG nodes degree, VG nodes diameter.[3]

Solving (11): the number of *labeled* variables (i.e. the

[2]We define the domain of a constraint as the product of the domains size of each variable that occurs in such constraint.
[3]With diameter of a node x we mean the maximum among the minimum distances between x and each other different node $y \neq x$.

variables to be assigned); the solve goal; the number of search annotations; the number of variable choice heuristics; the number of value choice heuristics.

Objective (12):[4] the domain dom, the degree deg, the ratios $\frac{dom}{deg}$ and $\frac{deg}{NC}$ of the variable v that has to be optimized; the degree de of v in the variable graph, its diameter di, $\frac{de}{di}$, and $\frac{di}{de}$. Moreover, named μ_{dom} and σ_{dom} the mean and the standard deviation of the variables domain size and μ_{deg} and σ_{deg} the mean and the standard deviation of the variables degree, we compute $\frac{dom}{\mu_{dom}}$, $\frac{deg}{\mu_{deg}}$, $\frac{dom-\mu_{dom}}{\sigma_{dom}}$, and $\frac{deg-\mu_{deg}}{\sigma_{deg}}$.

Dynamic features

For each problem we extract the following 11 dynamic features: the number of solutions found; the number p of propagations performed; the ratio $\frac{p}{NC}$; the number e of nodes expanded in the search tree; the number f of failed nodes in the search tree; the ratio $\frac{f}{e}$; the maximum depth of the search stack; the peak memory allocated; the CPU time needed for converting from MiniZinc to FlatZinc; the CPU time required for static features computation; the total CPU time needed for extracting all the features.

The first 8 features are collected by `mzn2feat` executing short runs (2 seconds) of Gecode `fz` interpreter with default parameters and `-s` and `-time` options. The last 3 are instead timing features computed by means of the `time` command of the Unix Bash shell.

3. CONCLUSIONS AND EXTENSIONS

In this work we presented a framework that is able to extract an extensive set of features from both satisfaction and optimization problems defined in possibly different modeling languages: MiniZinc, FlatZinc or XCSP. We deem that our work could serve as a building block for the creation of a modern constraint solver adopting a portfolio approach.

Indeed, it should be pretty straightforward to build a CSP portfolio solver exploiting the features extracted by `mzn2feat`: the only requirement is that each constituent solver of the portfolio must support MiniZinc format.

The set of features we propose was tested using the best CSP portfolio approaches [1]. Preliminary empirical results presented in [2] show that portfolio techniques exploiting the new set of features are effective and competitive with state of the art CSP portfolio techniques.

An interesting future direction of this work concerns the improvement of the quality of the features, possibly though appropriate feature filtering techniques. Our framework is flexible enough to allow without a great effort to add new features (e.g. dynamic features computed using local search algorithm as done by SATzilla [20]) as well as to select a proper subset of them as done in [10] for SAT and CP problems.

Since `mzn2feat` is able to process COPs encoded in MiniZinc format, one of the most promising extension of our work is to assemble and analyze portfolios of COP solvers. In our opinion the effectiveness of portfolios in the satisfaction field can also be reflected in the world of combinatorial optimization. Of course, in order to evaluate a COP solver new metrics should also be considered. In fact, often in real world it is better to get a good solution in a short time rather than consume too much time to find the optimal value. Starting from this assumption, it may be reasonable to develop new kinds of portfolio approaches that take into consideration also the solution quality.

[4]These features make sense only for optimization problems, a default value is given in case of satisfaction problems.

4. REFERENCES

[1] R. Amadini, M. Gabbrielli, and J. Mauro. An Empirical Evaluation of Portfolios Approaches for Solving CSPs. In *CPAIOR*, 2013.

[2] R. Amadini, M. Gabbrielli, and J. Mauro. Features for Building CSP Portfolio Solvers. *CoRR*, abs/1308.0227, abs/1308.0227, 2013.

[3] A. Arbelaez, Y. Hamadi, and M. Sebag. Continuous Search in Constraint Programming. In *ICTAI*, 2010.

[4] R. Becket. Specification of FlatZinc - Version 1.6. `http://www.minizinc.org/downloads/doc-1.6/flatzinc-spec.pdf`.

[5] N. Beldiceanu, M. Carlsson, S. Demassey, and T. Petit. Global Constraint Catalogue: Past, Present and Future. *Constraints*, 12(1):21–62, 2007.

[6] Third International CSP Solver Competition 2008. `http://www.cril.univ-artois.fr/CPAI09/`.

[7] GECODE flatzinc. `http://www.gecode.org/flatzinc.html`.

[8] C. P. Gomes and B. Selman. Algorithm portfolios. *Artif. Intell.*, 126(1-2):43–62, 2001.

[9] F. Hutter, L. Xu, H. H. Hoos, and K. Leyton-Brown. Algorithm Runtime Prediction: The State of the Art. *CoRR*, 2012.

[10] C. Kroer and Y. Malitsky. Feature filtering for instance-specific algorithm configuration. In *ICTAI*, 2011.

[11] Y. Malitsky and M. Sellmann. Instance-Specific Algorithm Configuration as a Method for Non-Model-Based Portfolio Generation. In *CPAIOR*, 2012.

[12] Mistral. `http://www.4c.ucc.ie/~ehebrard/mistral/doxygen/html/main.html`.

[13] M. Morara, J. Mauro, and M. Gabbrielli. Solving XCSP problems by using Gecode. In *CILC*, 2011.

[14] MiniZinc Challenge 2012. `http://www.minizinc.org/challenge2012/results2012.html`.

[15] N. Nethercote. Converting MiniZinc to FlatZinc. `http://www.minizinc.org/downloads/doc-1.6/mzn2fzn.pdf`.

[16] N. Nethercote, P. J. Stuckey, R. Becket, S. Brand, G. J. Duck, and G. Tack. MiniZinc: Towards a Standard CP Modelling Language. In *CP*, 2007.

[17] E. O'Mahony, E. Hebrard, A. Holland, C. Nugent, and B. O'Sullivan. Using case-based reasoning in an algorithm portfolio for constraint solving. *AICS 08*, 2009.

[18] O. Roussel and C. Lecoutre. XML Representation of Constraint Networks: Format XCSP 2.1. *CoRR*, abs/0902.2362, 2009.

[19] L. Xu, F. Hutter, H. Hoos, and K. Leyton-Brown. Evaluating Component Solver Contributions to Portfolio-Based Algorithm Selectors. In *SAT*, 2012.

[20] L. Xu, F. Hutter, H. H. Hoos, and K. Leyton-Brown. SATzilla-07: The Design and Analysis of an Algorithm Portfolio for SAT. In *CP*, 2007.