Proceedings of the Twenty-Ninth AAAI Conference on Artificial Intelligence

Characterizing Performance of Consistency Algorithms by Algorithm Configuration of Random CSP Generators

Daniel J. Geschwender and **Robert J. Woodward** and **Berthe Y. Choueiry**
Computer Science and Engineering, University of Nebraska-Lincoln, USA
{dgeschwe|rwoodwar|choueiry}@cse.unl.edu

Abstract

In Constraint Processing, many algorithms for enforcing the same level of local consistency may exist. The performance of those algorithms varies widely. In order to understand what problem features lead to better performance of one algorithm over another, we utilize an algorithm configurator to tune the parameters of a random problem generator and maximize the performance difference of two consistency algorithms for enforcing constraint minimality. Our approach allowed us to generate instances that run 1000 times faster for one algorithm over the other.

Introduction

Constraint Processing is an expressive and powerful framework for modeling and solving constrained combinatorial problems. A Constraint Satisfaction Problem (CSP) is defined by a set of *variables*, their respective *domains*, and a set of *constraints* over the variables restricting the combinations of values that can be assigned to the variables at the same time. A solution to a CSP assigns all variables a value from their respective domains such that no constraint is violated. Determining the satisfiability of a CSP is NP-complete, and a solution is typically found using backtrack search search and/or constraint propagation. Many propagation algorithms exist with widely varied effectiveness. An important research direction is the selection of the appropriate consistency algorithm to employ in solving a given problem instance. As a step in that direction, we consider the inverse question: we investigate what problems are particularly suited to the strengths of a given consistency algorithm. By investigating what type of problems an algorithm performs best on, we aim to improve our abilities to select the right algorithm in the appropriate context.

Consistency Algorithms Considered

We consider two consistency algorithms, PerTuple and AllSol(Karakashian et al. 2010; Karakashian 2013). Both algorithms enforce *constraint minimality*, which guarantees that every tuple in every relation can be extended to a full solution to the CSP (Montanari 1974). The importance of minimality was established for knowledge compilation (Gottlob 2012) and solving difficult CSPs with higher levels of consistency (Karakashian, Woodward, and Choueiry 2013). In particular, we are interested in applying AllSol and PerTuple locally to the clusters of a tree decomposition (Geschwender et al. 2013). The performance of the two algorithms vary widely in practice: Indeed, one algorithm may finish reasonably fast while the other fails to terminate in a given time threshold. This performance difference makes the algorithms ideal candidates for our configuration study.

Configuration of RBGenerator

We use an algorithm configurator that guides a random CSP generator to generate instances on which we execute PerTuple and AllSol. After comparing their performances on the generated instances, the configurator selects new parameters for the CSP generator in order to influence the performances. Figure 1 shows the components of the configuration system.

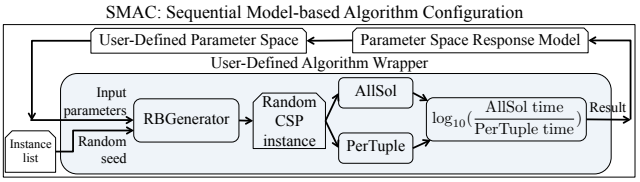

Figure 1: Operation of the configurator

RBGenerator. We use the random CSP generator RBGenerator (Xu et al. 2007), which is based on the model RB and allows for easy generation of hard satisfiable instances at the phase transition. RBGenerator uses the following parameters:

1. $k \geq 2$ denotes the arity of the constraints
2. n denotes the number of variables
3. α determines the domain size $d = n^{\alpha}$ of each variable
4. r determines the number $m = r \cdot n \cdot \ln(n)$ of constraints
5. δ determines the constraint tightness, $t = p_{cr} + \frac{\delta}{1000}$, where $p_{cr} = 1 - e^{-\frac{\alpha}{r}}$ is the location of phase transition
6. *forced* whether instances are forced satisfiable
7. *merged* whether constraints of similar scopes are joined

Sequential Model-based Algorithm Configuration. To tune RBGenerator's parameters, we use of the algorithm configurator "Sequential Model-based Algorithm Configuration" (SMAC) (Hutter, Hoos, and Leyton-Brown 2011). We give SMAC a list of the input parameters with ranges and default values. We provide a list of 30 instances (in this case, random seeds for the RBGenerator). Finally, we give SMAC a custom algorithm wrapper that runs RBGenerator, PerTuple, and AllSol. SMAC takes an initial default configuration, performs an algorithm execution, and determines its performance based on the wrapper output. It iterates the process, selecting new configurations (based on a continually improving regression model) and evaluating them.

Algorithm Wrapper. The algorithm wrapper encapsulates several programs to be run together. Initially, RBGenerator runs with the parameters provided by SMAC, and generates a CSP instance on which our two consistency algorithms PerTuple and AllSol are then executed. The execution times of PerTuple and AllSol are recorded, and are compared by taking the base-10 logarithm of their ratio. Taking the logarithm ensures that equal weights are given to fractional values when the results are averaged in SMAC's model.

Experimental Results

In our experiments, we evaluate how well SMAC is able to generate instances that favor a given algorithm. To this end, we test two cases: those where SMAC is allowed to adjust all parameters (denoted adjustable size), and those where SMAC has a restricted set of parameters (denoted fixed size). For the restricted set of parameters, we fix n to 16 and α to 1 (i.e., 16 variables each with a domain size of 16). Thus, we only allow SMAC to control the constraints and not the size of the CSP. For each the fixed and adjustable cases, we generate instances favoring PerTuple and instances favoring AllSol (for a total of four tests). Each test is performed with ten different configuration runs, each with a different SMAC configuration seed. Each configuration ran for four days on a single core of an Intel Xeon E5-2670 2.60GHz processors and given 3 GB memory.

In all four test cases, the configurator finds parameter settings that cause one algorithm to significantly outperform the other. The average speedup across all 10 seeds for each test is as follows: *adjustable-AllSol:* 108, *adjustable-PerTuple:* 981, *fixed-AllSol:* 98, and *fixed-PerTuple:* 306. Figure 2 shows the progressive improvement of the *adjustable-PerTuple* configuration over time. The x-axis shows the amount of time configuration has run, while the y-axis shows ratio of execution times. The ten lines are the ten different configuration runs, each with a unique seed.

Our approach allowed us to identify the r and δ parameters as important for the configuration. We found that low values of r and δ (fewer, looser constraints) favor PerTuple, and high values of r and δ (more, tighter constraints) favor AllSol. To apply this to real-world problems, a measure such as constrainedness (Gent et al. 1996) may be used. For additional detail on our experiment and results, see our technical report (Geschwender, Woodward, and Choueiry 2014).

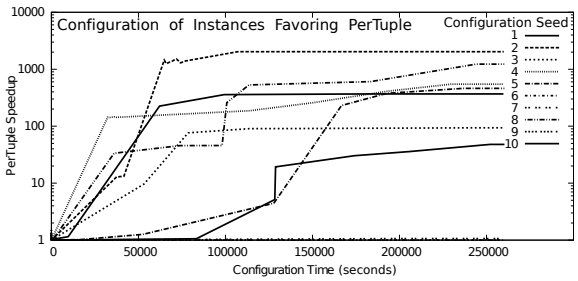

Figure 2: Improvement of the *adjustable-PerTuple* configuration.

Conclusions & Future Work

We successfully used configuration on a random CSP generator to create problem instances favoring PerTuple over AllSol and vice versa. The ratio of the execution time difference was in excess of 100. Additionally we have shown that highly constrained problems favor AllSol and under-constrained problems favor PerTuple. We intend to use this information in building an algorithm portfolio. In the future, we will apply our approach to other algorithms, both for enforcing the same and different levels of consistency. Further, we will use other generators with more configuration parameters in order to capture deeper behaviors of the algorithms.

Acknowledgments: Supported by NSF Grant No. RI-111795. Geschwender was also supported by NSF GRF Grant No. 1041000 and a Barry M. Goldwater Scholarship, and Woodward by NSF GRF Grant No. 1041000 and a Chateaubriand Fellowship. Experiments conducted at UNL's Holland Computing Center.

References

Gent, I. P.; MacIntyre, E.; Prosser, P.; and Walsh, T. 1996. The Constrainedness of Search. In *Proc. of AAAI 1996*, 246–252.

Geschwender, D.; Karakashian, S.; Woodward, R.; Choueiry, B. Y.; and Scott, S. D. 2013. Selecting the Appropriate Consistency Algorithm for CSPs Using Machine Learning Classifiers. In *Proc. of AAAI 2013*, 1611–1612.

Geschwender, D. J.; Woodward, R. J.; and Choueiry, B. Y. 2014. Configuring Random CSP Generators to Favor a Particular Consistency Algorithm. Technical Report TR-UNL-CSE-2014-0003, University of Nebraska-Lincoln.

Gottlob, G. 2012. On Minimal Constraint Networks. *Artificial Intelligence* 191-192:42–60.

Hutter, F.; Hoos, H. H.; and Leyton-Brown, K. 2011. Sequential Model-based Optimization for General Algorithm Configuration. In *Proc. of LION5 2011*, 507–523.

Karakashian, S.; Woodward, R. J.; Reeson, C.; Choueiry, B. Y.; and Bessiere, C. 2010. A First Practical Algorithm for High Levels of Relational Consistency. In *Proc. of AAAI 2010*, 101–107.

Karakashian, S.; Woodward, R. J.; and Choueiry, B. Y. 2013. Improving the Performance of Consistency Algorithms by Localizing and Bolstering Propagation in a Tree Decomposition. In *Proc. of AAAI 2013*, 466–473.

Karakashian, S. 2013. *Practical Tractability of CSPs by Higher Level Consistency and Tree Decomposition*. Ph.D. Dissertation, University of Nebraska-Lincoln.

Montanari, U. 1974. Networks of Constraints: Fundamental Properties and Applications to Picture Processing. *Information Sciences* 7:95–132.

Xu, K.; Boussemart, F.; Hemery, F.; and Lecoutre, C. 2007. Random Constraint Satisfaction: Easy Generation of Hard (Satisfiable) Instances. *Artificial Intelligence* 171(8):514 – 534.