Configuring Random CSP Generators to Favor a Particular Consistency Algorithm

Daniel J. Geschwender   Robert J. Woodward
Berthe Y. Choueiry

Constraint Systems Laboratory
University of Nebraska-Lincoln, USA
{dgeschwe|rwoodwar|choueiry}@cse.unl.edu

TR-UNL-CSE-2014-0003

July 30, 2014

Abstract

In Constraint Processing, no single consistency algorithm always outperforms all others, but the problem type and characteristics often determine the most appropriate algorithm. Our goal is to understand and determine what problem features lead to better performance of one algorithm over another. As a first step in that direction, we utilize an algorithm configurator to set the parameters of a random problem generator. The configurator is set to maximize the ratio between the execution times of two particular algorithms for computing the minimal constraints. Experimentally, we generated instances that ran 1000 times faster for one algorithm over the other. The parameters returned by the configurator give insight into what makes one algorithm outperform the other.

Contents

1 Introduction

Constraint Processing is an expressive and powerful framework for modeling and solving constrained combinatorial problems. Solving a Constraint Satisfaction Problem (CSP) is in general NP-complete, and it is often solved through search and/or constraint propagation. Many search strategies and propagation algorithms exist, each with their strengths and effectiveness. An important research direction is the selection of the appropriate algorithms 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 algorithm. By investigating what type of problems an algorithm performs best on, we may improve our abilities to select the right algorithm for in the appropriate context.

In this paper, we demonstrate the utility of algorithm configuration in the random generation of CSP instances. By taking a suitably parameterized random CSP generator, we use an algorithm configurator to tune the parameters such that a given algorithm performs favorably on the generated instance. With this technique we generated instances that run over 1000 times faster on one algorithm than another. We also see what parameters are responsible for this performance difference.

This paper is organized as follows. Section 2 discusses background of CSPs and algorithm configuration. Section 3 details the consistency algorithms used in this paper. Section 4 explains the method used to configure the CSP generator. Section 5 describes our experimental setup and Section 6 analyzes our results. Finally, in Section 7 we conclude and give ideas for future research.

2 Background

A Constraints 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 whether a solution exists is in general NP-complete. A constraint is defined by its *scope* and *relation*. The scope of a constraint is the set of variables that it covers. The *arity* of a constraint is the cardinality of its scope. The relation is a set of tuples of domain values from variables in the scope. These tuples specify the allowed combinations of domain values (i.e., supports). The *tightness* of a constraint is the ratio of forbidden tuples (i.e., conflicts) to all possible tuples. As tightness increases, the problem becomes more constrained and the number of solutions decreases. As constraint tightness decreases, the problem becomes under-constrained and the number of solutions increases. At either extreme, the problems are relatively easy to solve. Difficult problems are found at the *phase transition* located between over and under-constrained problems [Cheeseman *et al.*, 1991]. In that region, it is not immediately apparent when a solution is at hand, and the cost of problem solving sharply increases.

A *local consistency property* guarantees that the values of all combinations of a given number of CSP variables (alternatively, the tuples of all combinations of a given size of relations) are consistent with the constraints that apply to them. This condition is necessary but not sufficient for the values (or the tuples) to appear in a solution to the CSP. A *propagation algorithm*, which enforces a given consistency property, typically proceeds by deleting inconsistent domain values (or relation tuples). Some consistency properties are stronger than others. The corresponding algorithms result in greater amounts of pruning, often at the expense of an increased cost. Strategies for selecting the right property to enforce in a given context and the most appropriate algorithm for the chosen property are at the center of active research [Chmeiss and Sais, 2004; Epstein *et al.*, 2005; Régin, 2005; Stergiou, 2009; Balafrej *et al.*, 2013].

Algorithm configuration is an optimization technique used to improve the performance of a target algorithm on a particular set of instances by tuning the algorithm's input parameters. Algorithm configuration is typically used to reduce the runtime of the target algorithm, and it can also be used to improve the algorithm's solution quality. Various configuration strategies are used to build configurations, including racing procedures [Birattari *et al.*, 2002], local search [Hutter *et al.*, 2009], and model-based optimization [Hutter *et al.*, 2011]. Algorithm configuration is most effective when as much as possible of the internals of the algorithm are exposed as parameters, increasing the flexibility of the configuration. This approach is particularly useful in that it allows for detailed automated exploration of the parameter space.

3 Consistency Algorithms Considered

The two algorithms considered in our configuration are PerTuple and AllSol [Karakashian *et al.*, 2010; 2012; 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], Figure 1. The importance of minimality was established for knowledge compilation [Gottlob, 2012] and achieving higher levels of consistency [Karakashian *et al.*, 2013]. In particular, we are interested in applying AllSol and PerTuple locally to the clusters of a tree decomposition [Geschwender *et al.*, 2013].

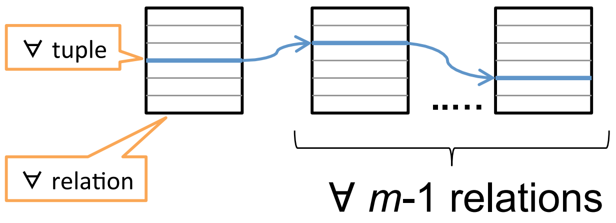

Figure 1: Minimality: every tuple in every relation extends to a full solution

PerTuple loops over all tuples of every relation of the CSP. For each tuple, it performs a backtrack search to find the first solution in which the tuple appears. If no solution is found involving the tuple, the tuple is removed from the relation. If a solution is found, all tuples involved in the solution are saved. PerTuple terminates after having removed or saved every tuple. The number of search processes executed by PerTuple is linear in the total number of tuples in the relations. Intuitively, PerTuple should perform well when each search can be quickly completed, that is, instances that are small or instances that are large but located away from the phase transition. In contrast, AllSol executes a single backtrack search to enumerate all the solutions. For every solution identified, the involved tuples are saved. AllSol terminates after exploring the entire search tree. Tuples not appearing in a solution are removed. AllSol only performs a single but exhaustive backtrack search. Thus, AllSol should outperform PerTuple in large problems that are around or above the phase transition where most of the search space is explored anyway. The situations is depicted in Figures 2 and 3. The backtrack search utilized by both AllSol and PerTuple operates on the dual CSP and uses both forward checking and dynamic variable ordering.

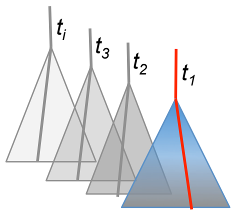

Figure 2: PerTuple performs a new solution search for every tuple

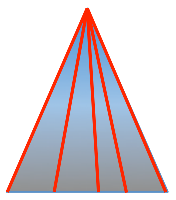

Figure 3: AllSol performs a single exhaustive solution search

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 experiment. We previously explored a portfolio approach to automatically choose one of the two algorithms based on problem features [Geschwender *et al.*, 2013]. While we achieved some success, we found a flagrant imbalance in our test cases: PerTuple was disproportionately over-represented in our tests on benchmark instances. By configuring a random CSP generator, we are able to specifically generate instances that significantly favor AllSol over PerTuple.

4 Configuration of RBGenerator

We use an algorithm configurator that guides a random CSP generator in order to generate instances on which we execute PerTuple and AllSol to test their performances. After comparing their performances on the generated instances, the configurator selects new parameters for the CSP generator in order to influence the performances. Figure 4 shows the various components of the configuration system.

4.1 RBGenerator

We use the random CSP generator RBGenerator [Xu *et al.*, 2007]. This generator is based on the model RB, which 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

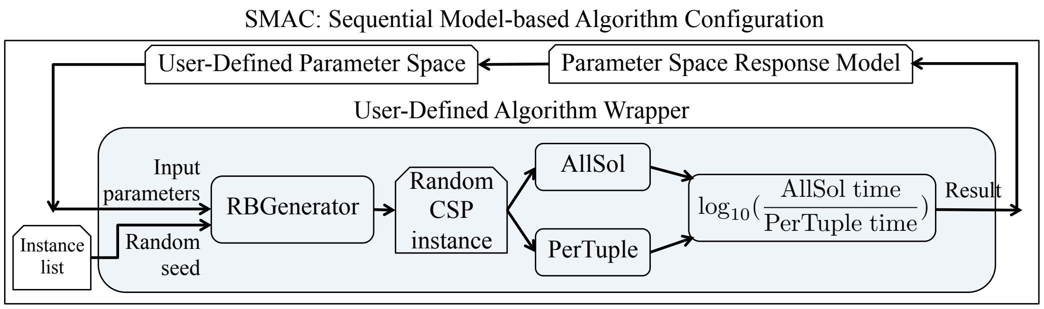
SMAC: Sequential Model-based Algorithm Configuration
User-Defined Parameter Space
Parameter Space Response Model
User-Defined Algorithm Wrapper
Input parameters
RBGenerator
Random CSP instance
AllSol
PerTuple
$\log_{10}(\frac{\text{AllSol time}}{\text{PerTuple time}})$
Result
Instance list
Random seed

Figure 4: Operation of the configurator

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 the phase transition

6. *forced* indicates whether or not instances are forced satisfiable

7. *merged* indicates whether or not constraints of similar scopes are joined

A strength of the RBGenerator is that it guarantees an asymptotic phase transition under certain parameter conditions. When an asymptotic phase transition exists, the threshold can be exactly determined. In addition, the parameters it provides give a lot of flexibility to the configuration process. Adjusting the arity through k allows some control over the structure of the constraints. The impact of problem size can be determined by independently manipulating the parameters for variables (n) and domain size (α). Constrainedness of the problem can be controlled by the parameters r and δ. While r is fairly straightforward in controlling the number of constraints, δ is particularly useful in that it allows control over the problem's position in relation to the phase transition. It can specifically create problems at the transition, or push above or below by a given amount. $forced$ is useful in that it allows a guarantee that even highly constrained problems have at least a single solution. *merged* allows for consolidation of the constraints if there is overlap.

4.2 Sequential Model-based Algorithm Configuration

To tune the parameters fed into RBGenerator, we use of the algorithm configurator SMAC (Sequential Model-based Algorithm Configuration) [Hutter *et al.*, 2011]. We give SMAC a description of the input parameters and acceptable ranges for them as well as a default parameter configuration. We also give it a list of instances to use for configuration. In this case, the list of instances is a set of 30 random seeds for the RBGenerator. Finally, we give SMAC a custom algorithm wrapper that handles the execution of RBGenerator, PerTuple, and AllSol. SMAC takes an initial default configuration, performs an algorithm execution, and determines its performance based on the wrapper output. Then, it iteratively repeats the process, selecting new configurations and evaluating them. Internally, SMAC maintains a continually developing regression model of the parameter space. This model is based around a random forest classifier. New configurations are selected using the regression model and attempt to balance exploration and exploitation. The parameter exploration is also tied to the random seed that is provided to SMAC on launch.

4.3 Algorithm Wrapper

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

5 Experiment Setup

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 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. Table 1 lists the parameter ranges and defaults that were used on both tests. For both fixed and adjustable, 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. This results in ten different paths through the parameter space and a better picture of the effects of the configuration.

Table 1: Parameter ranges and defaults

Adjustable Size			**Fixed Size**		
Parameter	**Range**	**Default**	**Parameter**	**Range**	**Default**
k	[2,10]	3	k	[2,10]	3
n	[2,20]	10	n	[16,16]	16
α	[0.1,10.0]	1.0	α	[1.0,1.0]	1.0
r	[0.1,10.0]	1.0	r	[0.1,10.0]	1.0
δ	[-1000,1000]	10	δ	[-1000,1000]	10
forced	{y,n}	n	*forced*	{y,n}	n
merged	{y,n}	n	*merged*	{y,n}	n

During the course of one configuration run, RBGenerator will generate numerous instances, SMAC will iteratively build a regression model of the parameter space, and the parameters will be fine-tuned. Initially, all runs will begin with the default parameter settings. The configuration seed affects where the parameters will go next. With each new set of parameters, up to 30 new instances, each with a different instance seeds, are built by RBGenerator and tested. After each set of runs, SMAC's regression model integrates the new results and selects a new parameter configuration that is expected to do well.

In some cases, the parameters selected may be invalid and cause a crash. For example, certain values of α, r, and δ in combination may be invalid because they result in tightness above one or below zero. Crashed runs are ignored and new parameter values are selected.

To prevent the algorithm wrapper from stalling during configuration, we set time limits on its components. RBGenerator is allowed to run for five minute to generate a CSP instance. PerTuple and AllSol are allowed to run for 20 minutes each while they enforce minimality on the CSP. If RBGenerator exceeds its time limit, the entire run is considered crashed. If PerTuple and AllSol exceed their time limit, their runtime is reported as 20 minutes.

Restricting the time limit allows for comparison between runs that terminate and those that do not.

SMAC is allowed to run configuration runs for four days. After that point, it takes the best configuration and validates the results by running on all 30 RBGenerator instance seeds.

We run all our tests on a computer cluster of 7232 Intel Xeon cores in 452 nodes. Each configuration run was allocated a single core of an Intel Xeon E5-2670 2.60GHz processors and given 3 GB memory.

6 Results

In all four cases, the configurator is able to find parameter settings that cause the desired algorithm to significantly outperform the other. Table 2 shows the results of the configuration in each of the four tests, across all ten seeds. The column iters. reports the number of iterations of SMAC before stopping (four days). The seven parameters are listed in the next columns. The tightness, calculated by $1 - e^{\frac{-\alpha}{r}} + \frac{\delta}{1000}$, is given in next column. Consistent indicates whether or not the instances had solutions. Speedup indicates how many times one algorithm runs faster than the other (running time of slower algorithm over that of faster algorithm). The reported speedup for each configuration seed is the average speedup across 30 different instances generated with 30 different instance seeds. We also provide the *coefficient of variation* (CoV), which is the ratio of the standard deviation to the mean. A CoV value of less than 50% indicates that the variance across instance seeds is low. Consequently, the results are more dependent on the parameter configuration than on the random variation between the 30 instances.

The maximum speedup found is bolded for each of the four tests. All four tests realize a speedup of at least 100 times, enough to definitively show there are classes of problems heavily suited to either algorithms. However, when configuring for PerTuple, we achieve speedups of over 1000 times. This fact may indicate that PerTuple has a stronger affinity for a particular problem class than AllSol. It is also worth noting that the fixed and adjustable problem size causes little change in the achieved speedup. Adjustable problem size parameters allow discovery of only marginally better configurations.

Note that some of the configuration seeds led to configurations with no speedup (indicated by ‘n/a’), either because of crashes or timeouts. This lack of progress is the result of particular seeds yielding flawed initial parameters.

Table 2: Configured parameters and the resulting speedups

		seed	iters	*k*	*n*	*a*	*r*	*δ*	*forced*	*merged*	tightness	consistent	speedup	CoV
Adjustable Size	**AllSol**	1	4330	4	19	0.75	9.91	578	n	y	0.65	no	101.36	14.55%
		2	2434	7	18	0.20	5.88	-9	y	y	0.02	yes	36.30	6.49%
		3	2722	5	17	0.68	9.18	759	n	y	0.83	no	75.73	8.10%
		4	1451	2	20	1.74	8.28	309	n	y	0.50	no	**348.43**	4.83%
		5	393	2	12	1.45	4.43	-54	n	n	0.23	yes	11.21	32.69%
		6	270	3	14	0.78	1.52	-100	y	n	0.30	yes	9.63	27.00%
		† 7	599	2	20	1.75	8.12	302	y	n	0.50	yes	289.63	27.61%
		8	454	3	8	1.70	1.00	-155	n	n	0.66	yes	13.72	77.29%
		9	1864	5	19	0.72	7.70	825	n	n	0.91	no	80.70	3.56%
		10	2712	4	16	0.86	9.95	646	n	y	0.73	no	115.01	17.31%
	PerTuple	1	248	2	9	1.28	5.29	-162	n	n	0.05	yes	378.29	15.12%
		† 2	562	2	17	0.79	0.61	-394	y	n	0.33	yes	**4,627.35**	78.67%
		3	202	3	16	1.01	0.29	-190	n	n	0.78	yes	296.94	102.49%
		4	88	2	14	0.77	2.69	-235	y	n	0.01	yes	531.85	15.82%
		5	905	2	12	0.85	6.19	-98	y	y	0.03	yes	442.20	12.32%
		‡ 6	305	4	13	9.80	8.05	542	n	n	n/a	n/a	n/a	n/a
		7	145	2	19	0.36	3.59	291	n	y	0.39	no	0.97	17.70%
		8	138	2	14	0.87	1.25	-334	n	n	0.17	yes	1,510.80	40.12%
		∗ 9	95	4	11	1.11	7.61	-52	y	n	n/a	n/a	n/a	n/a
		10	332	3	18	0.99	0.18	-780	y	y	0.22	yes	63.75	73.30%
Fixed Size	**AllSol**	11	1198	4	16	1.00	8.35	799	n	n	0.91	no	103.05	7.73%
		12	1049	4	16	1.00	9.74	832	n	n	0.93	no	98.89	5.45%
		13	1148	4	16	1.00	9.33	826	n	n	0.93	no	107.16	6.87%
		14	1057	4	16	1.00	9.00	811	n	n	0.92	no	89.80	13.29%
		15	1005	4	16	1.00	8.36	794	n	y	0.91	no	87.45	11.21%
		16	1115	4	16	1.00	7.20	764	n	y	0.89	no	92.09	8.51%
		17	989	4	16	1.00	7.18	757	y	n	0.89	yes	87.79	10.54%
		18	980	4	16	1.00	8.59	808	n	n	0.92	no	102.17	6.20%
		19	954	4	16	1.00	9.94	840	y	y	0.94	yes	**109.75**	4.05%
		20	1168	4	16	1.00	7.73	786	n	y	0.91	no	100.21	7.66%
	PerTuple	∗ 11	89	4	16	1.00	5.13	-62	y	n	n/a	n/a	n/a	n/a
		12	120	2	16	1.00	1.22	-361	y	y	0.20	yes	311.41	35.37%
		13	82	2	16	1.00	2.21	-276	y	y	0.09	yes	69.74	28.92%
		14	60	2	16	1.00	3.00	-265	n	n	0.02	yes	47.06	15.05%
		‡ 15	33	5	16	1.00	0.46	757	y	n	n/a	n/a	n/a	n/a
		† 16	173	3	16	1.00	0.14	-950	y	n	0.05	yes	40.36	102.40%
		‡ 17	101	7	16	1.00	9.91	-377	n	n	n/a	n/a	n/a	n/a
		‡ 18	131	8	16	1.00	4.41	292	n	n	n/a	n/a	n/a	n/a
		19	188	2	16	1.00	0.74	-481	n	n	0.26	yes	**1,300.63**	27.01%
		† 20	176	3	16	1.00	0.18	-967	n	n	0.03	yes	69.71	116.22%

*: all instances timeout, †: one or two instances crash, ‡: all instances crash

By continuing to select parameters causing crashes and timeouts, SMAC has no useful data on which to build its model. On those seeds, SMAC continues to blindly select new parameters that cause crashes and timeouts.

Figures 5 and 6 show the improvements over the course of the configuration. Configuring for AllSol tends to see smaller improvements being made, while PerTuple makes fewer, large improvements. AllSol configuration also tends to find improvements early on. In all cases, there is significant variation between seeds. However, configuration runs of AllSol with fixed problem size all converge by the end.

The parameter settings obtained by the configuration processes give insight into what problems each algorithm works well on. Two parameters in particular seem correlated with the algorithm speedups: r and δ. When configuring for AllSol, r is set at an average of around eight (8) and δ is a large positive value around 600. For PerTuple, r is generally lower, around three (3), and δ a large negative value around -200. There are exceptions to this, such as adjustable-PerTuple-7 or fixed-PerTuple-15, but those exceptions are almost all poor speedups or crashed runs.

Figure 7 shows the effect of both r and δ on the performance of the algorithms. The data shown here includes the intermediate configurations tested on the way to the final configurations. Figure 8 shows the combined effect of the parameters.

All of the AllSol configurations of fixed problem size end up with extremely similar parameter configurations as was hinted at by the convergence of their speedups. By restricting the parameters, fewer paths through the parameter space provide viable speedups. Thus, the configuration runs tended to converge.

The r parameter sets the number of constraints while δ influences the number of allowed tuples for constraints. Thus, a configuration with a small r and negative δ yields significantly under-constrained problems, while a configuration with a large value of r and positive δ produces a highly constrained problem. Not only are we able to produce problems favoring both algorithms, we can also determine what makes them 'favorable.'

For real-world problems, the phase transition is not explicitly defined and not all constraints are necessarily of uniform tightness. However, the kappa parameter, introduced by Gent *et al.*, is a measure of the constrainedness of a problem and can approximate the phase transition [1996]. Thus our findings have applicability to real-world instances.

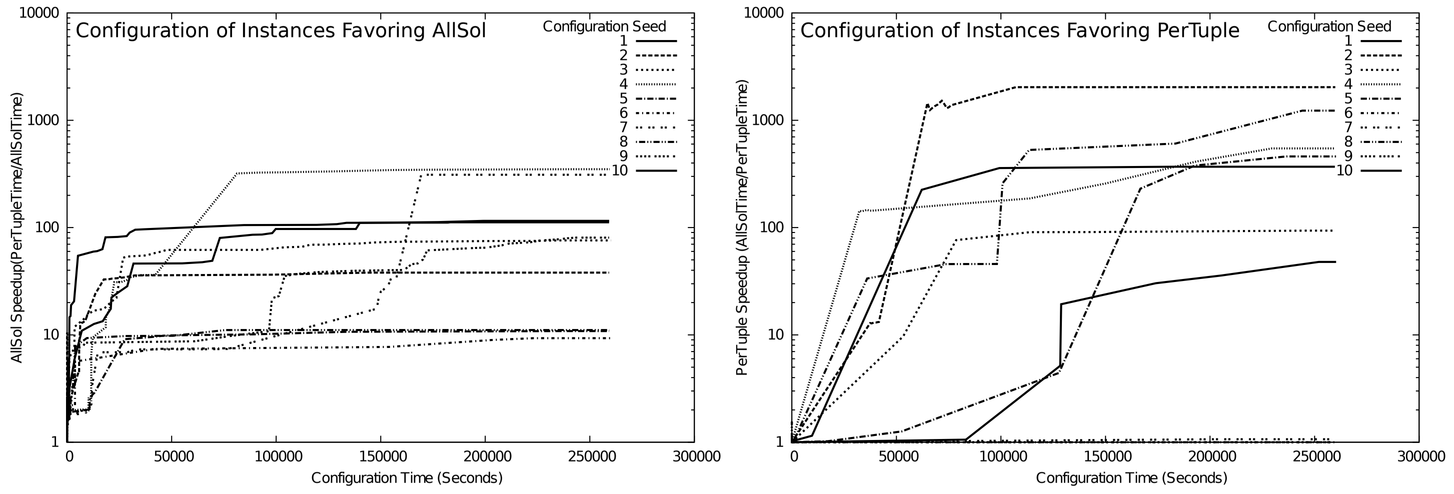

Figure 5: Configuration improvement with adjustable problem size

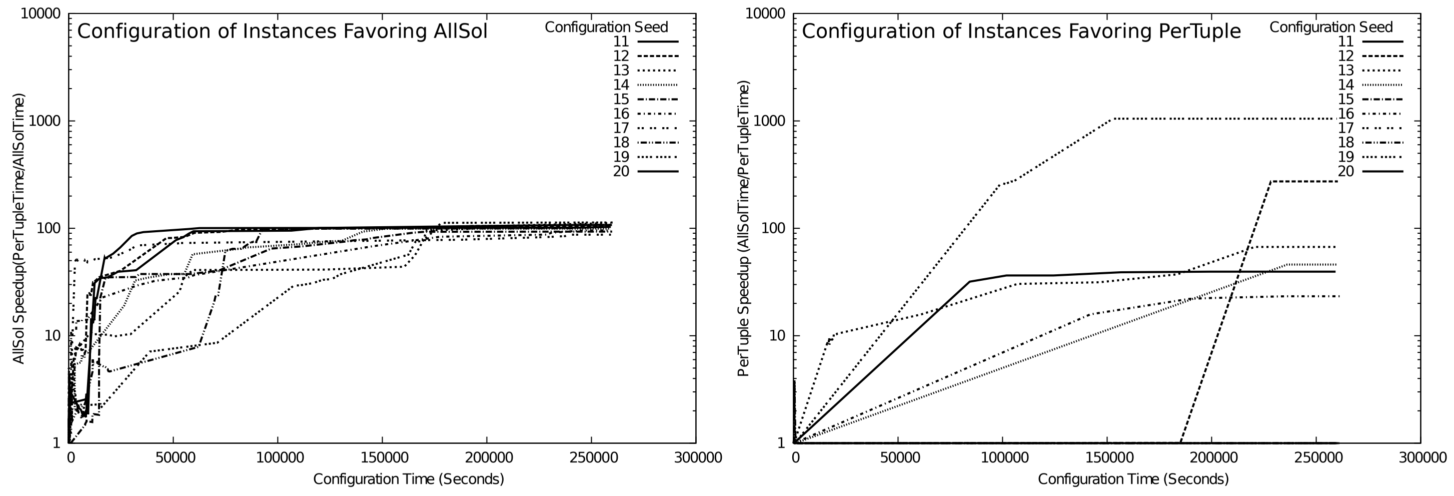

Figure 6: Configuration improvement with fixed problem size

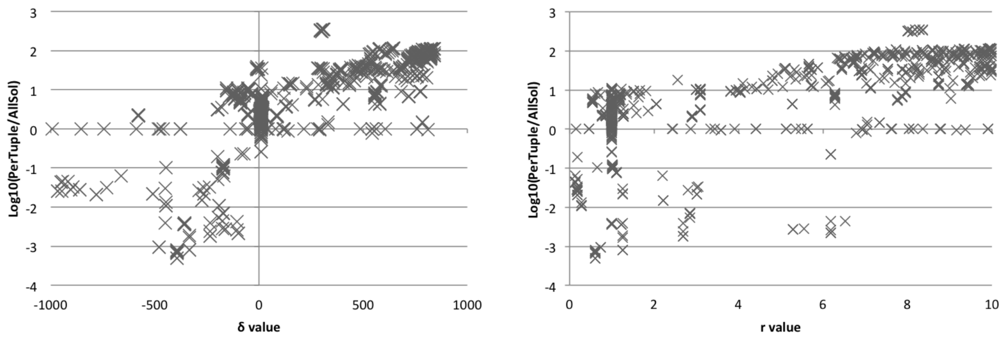

Figure 7: Effects of r and δ

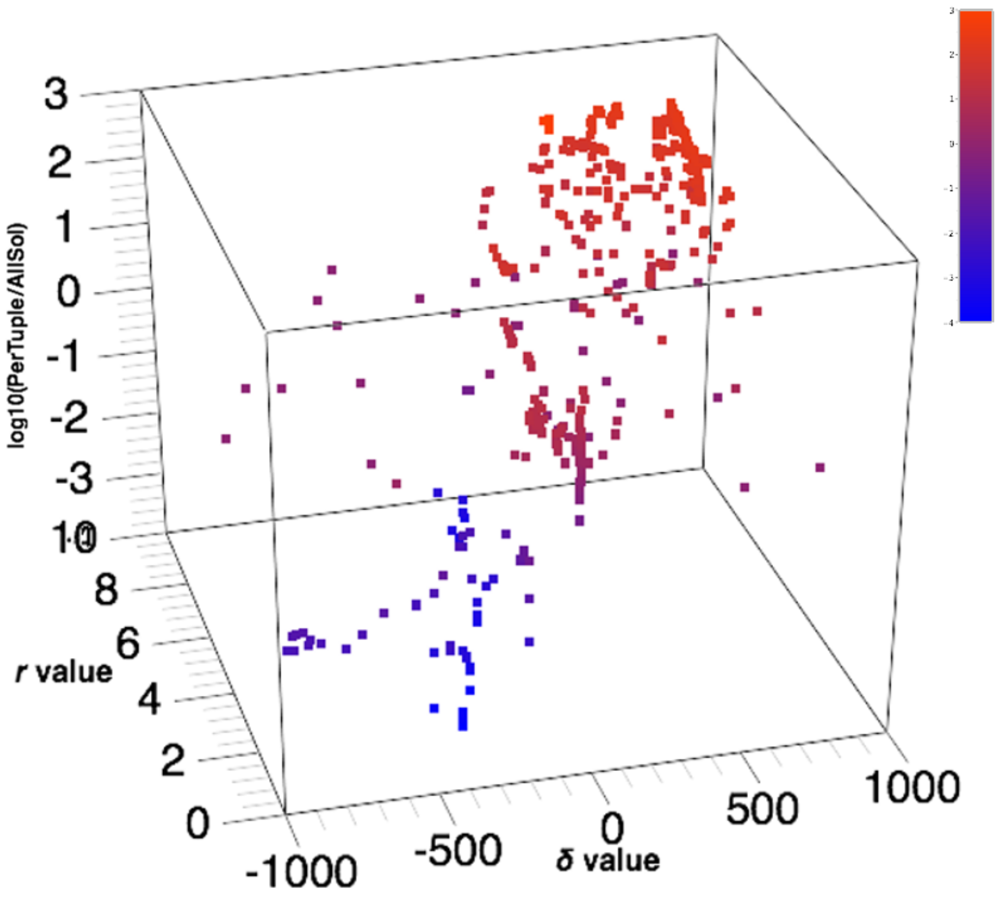

Figure 8: Combined effects of r and δ

7 Conclusions & Future Work

In our work, we have confirmed that pockets of CSPs exist that favor both PerTuple and AllSol. We have also identified the parameters leading to this situation and generated such problems. The parameter settings important to this algorithm choice are the parameters that control the number and tightness of constraints. Highly constrained problems favor AllSol and under-constrained problems favor PerTuple, consistent with our intuitions of the algorithms.

Our use of algorithm configuration for the understanding of the performance of consistency algorithms is at an early stage but shows great promise. An obvious research direction is to apply this approach to other consistency algorithms to ensure it generalizes. Any two algorithms for enforcing the same consistency (e.g., AC3.1 and AC4) could be dropped into this framework to identify a niche of problems where one excelled over the other. It would also be possible to compare algorithms for enforcing different consistencies. This task is accomplished by running the propagation algorithm followed by search and comparing the total time. Making use of a CSP generator with a greater number of parameters may also prove interesting. RBGenerator provides a sufficient number for this task, but having even more parameters provides even more flexibility to the configurator. Finally, this work has application to algorithm selection for CSPs by providing insight into what features of a problem cause one algorithm to outperform another.

Acknowledgments

This research was 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 were conducted on the equipment of the Holland Computing Center at the University of Nebraska - Lincoln.

References

[Balafrej *et al.*, 2013] Amine Balafrej, Christian Bessiere, Remi Coletta, and El Houssine Bouyakhf. Adaptive Parameterized Consistency. In *Proceed-*

ings of the International Conference on Principles and Practice of Constraint Programming, pages 143–158, 2013.

[Birattari *et al.*, 2002] Mauro Birattari, Thomas Stützle, Luis Paquete, and Klaus Varrentrapp. A Racing Algorithm for Configuring Metaheuristics. In *Proceedings of the Genetic and Evolutionary Computing Conference*, volume 2, pages 11–18, 2002.

[Cheeseman *et al.*, 1991] Peter Cheeseman, Bob Kanefsky, and William M. Taylor. Where the Really Hard Problems Are. In *Proceedings of the International Joint Conference on Artificial Intelligence*, pages 331–337, 1991.

[Chmeiss and Sais, 2004] Assef Chmeiss and Lakhdar Sais. Constraint Satisfaction Problems: Backtrack Search Revisited. In *Proceedings of 16th IEEE International Conference on Tools with Artificial Intelligence*, pages 252–257, 2004.

[Epstein *et al.*, 2005] Susan Epstein, Richard Wallace, Eugene Freuder, and Xingjian Li. Learning Propagation Policies. In *Proceedings of the Second International Workshop on Constraint Propagation And Implementation*, pages 1–15, 2005.

[Gent *et al.*, 1996] Ian P. Gent, Ewan MacIntyre, Patrick Prosser, and Toby Walsh. The Constrainedness of Search. In *Proceedings of the Thirteenth AAAI Conference on Artificial Intelligence*, pages 246–252, 1996.

[Geschwender *et al.*, 2013] Daniel Geschwender, Shant Karakashian, Robert Woodward, Berthe Y. Choueiry, and Stephen D. Scott. Selecting the Appropriate Consistency Algorithm for CSPs Using Machine Learning Classifiers. In *Proceedings of the Twenty-seventh AAAI Conference on Artificial Intelligence*, pages 1611–1612, 2013.

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

[Hutter *et al.*, 2009] Frank Hutter, Holger H. Hoos, Kevin Leyton-Brown, and Thomas Stützle. ParamILS: an Automatic Algorithm Configuration Framework. *Journal of Artificial Intelligence Research*, 36(1):267–306, 2009.

[Hutter *et al.*, 2011] Frank Hutter, Holger H. Hoos, and Kevin Leyton-Brown. Sequential Model-based Optimization for General Algorithm Configuration. In *Proceedings of the Fifth Conference on Learning and Intelligent Optimization*, pages 507–523. Springer, 2011.

[Karakashian *et al.*, 2010] Shant Karakashian, Robert J. Woodward, Christopher Reeson, Berthe Y. Choueiry, and Christian Bessiere. A First Practical Algorithm for High Levels of Relational Consistency. In *Proceedings of the Twenty-fourth AAAI Conference on Artificial Intelligence*, pages 101–107, 2010.

[Karakashian *et al.*, 2012] Shant Karakashian, Robert J. Woodward, Berthe Y. Choueiry, and Stephen D. Scott. Algorithms for the Minimal Network of a CSP and a Classifier for Choosing Between Them. TR-UNL-CSE-2012-0007, 2012.

[Karakashian *et al.*, 2013] Shant Karakashian, Robert J. Woodward, and Berthe Y. Choueiry. Improving the Performance of Consistency Algorithms by Localizing and Bolstering Propagation in a Tree Decomposition. In *Proceedings of the Twenty-seventh AAAI Conference on Artificial Intelligence*, pages 466–473, 2013.

[Karakashian, 2013] Shant Karakashian. *Practical Tractability of CSPs by Higher Level Consistency and Tree Decomposition*. PhD thesis, University of Nebraska-Lincoln, 2013.

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

[Régin, 2005] Jean-Charles Régin. AC-*: A Configurable, Generic and Adaptive Arc Consistency Algorithm. In *Proceedings of the Eleventh International Conference on Principles and Practice of Constraint Programming*, volume 3709 of *LNCS*, pages 505–519, 2005.

[Stergiou, 2009] Kostas Stergiou. Heuristics for Dynamically Adapting Propagation in Constraint Satisfaction Problems. *AI Communications*, 22(3):125–141, 2009.

[Xu *et al.*, 2007] Ke Xu, Frederic Boussemart, Fred Hemery, and Christophe Lecoutre. Random Constraint Satisfaction: Easy Generation of Hard (Satisfiable) Instances. *Artificial Intelligence*, 171(8):514 – 534, 2007.