Identifying inconsistent CSPs by Relaxation

Tomas Nordlander[1], Ken Brown[2] and Derek Sleeman[1]

[1]Department of Computing Science, University of Aberdeen, Scotland, UK
{tnordlan, sleeman }@csd.abdn.ac.uk
[2]Cork Constraint Computation Centre, Department of Computer Science,
University College Cork, Ireland
k.brown@cs.ucc.ie

Abstract. How do we identify inconsistent CSPs quickly? This paper presents relaxation as one possible method; showing how we can generate relaxed CSPs which are easier to prove inconsistent. We examine different relaxation strategies based on constraint graph properties, and we show that removing constraints of low tightness is an efficient strategy which is also simple to implement.

Introduction

The MUSKRAT (Multistrategy Knowledge Refinement and Acquisition Toolbox) framework aims to unify problem solving, knowledge acquisition and machine learning in a single computational framework [1]. Given a set of Knowledge Bases (KBs) and Problem Solvers (PSs), the system tries to identify which KBs could be combined with which PS(s) to solve a given task. We propose to represent the KBs and PSs as CSPs, which we can combine to get larger CSPs. If a combined CSP has a solution, then the original combination could be used to solve the task; if the CSP does not have a solution, then the combination cannot be used. Identifying a suitable combination thus requires examining a series of CSPs, and rejecting inconsistent ones until we find one with a solution. Proving CSPs inconsistent can be a lengthy process. We would like to find a way to identify inconsistent CSPs quickly. One method might be to relax a CSP to obtain one that is easier to prove inconsistent. If we can prove the relaxed CSP is inconsistent, then we know that the original was also inconsistent. If the relaxed CSP has a solution, then the original CSP represents a plausible combination for the original task. This paper investigates different relaxation strategies for random binary CSPs, and suggests that removing constraints with low tightness is an effective method for identifying inconsistent problems.

Background

White & Sleeman [1] proposed a Meta Problem Solver, which checks if combinations of KBs with a PS, represented as a CSP, are plausible. However, this approach was

unable to verify that no discarded combinations contained a solution; i.e. it was unable to guarantee the properties of the Meta Problem Solvers.

Constraint Satisfaction Problems and associated methods are an effective way of modelling and solving combinatorial problems [2, 3], and there exist a number of efficient toolkits and languages (e.g. [4, 5]). A CSP may be proven to be inconsistent by search, or by attempting to enforce various forms of consistency [6]. The idea of relaxing CSPs has received considerable attention [7-9], but focused on changing the problem to introduce solutions. Relaxing problems is also a common technique in mathematical programming, e.g. to obtain better bounds in optimisation [10]. There has been extensive research on randomly generated CSPs [7, 11], which show a phase transition between soluble and insoluble problems, with the hardest problems being at the transition point. Problem classes in binary CSPs can be described by a tuple <n,m,c,t>, where n is the number of variables and m is the number of values in each domain, c is the number of constraints, and t is the number of forbidden tuples in each constraint (a measure of problem tightness).

Experiments

Our aim is to relax a given CSP to generate a new CSP which is easier to solve, with the intention of quickly discarding inconsistent CSPs. In this paper, we examine different relaxation strategies, trying to identify the best way to relax individual problems. We generate random CSPs, then relax them in different ways and measure the computation times and solution results. First we will describe the software, then we present the experiments.

The CSP-SUITE [12] used in these experiments is written in SICStus Prolog [5] and consists of three modules. The *Generating* module creates random binary CSPs. We want to identify individual constraints to relax, so to introduce variety we allow the tightness of individual constraints in a problem to vary rather than be constant. Thus we use the 5-tuple <n,m,c,[t_μ,r]>, where t_μ is the average number of forbidden tuples, with the number for each constraint selected uniformly at random from the range [t_μ-r, t_μ+r]. The *Relaxing* module generates relaxed CSPs from an original by removing a specified number of constraints according to nine different strategies. (1) Random simply chooses the constraints randomly. (2) Greedy Search considers each constraint in turn, removing it, solving the relaxed CSP, and replacing the constraint. It then selects the constraint whose removal gave the best performance improvement, removes it, and repeats the whole process on the resulting CSP. (3) Greedy Ordering uses the first iteration of Greedy Search to generate an ordering list for all constraints then removes constraints in the order suggested. (4, 5) Node Degree selects constraints in ascending or descending order of the degree of their associated variables in the constraint graph. (6, 7) Isolate Node selects high or low degree nodes and removes a series of constraints incident on those nodes (i.e. it tries to remove variables from the problem). (8, 9) Tightness removes constraints in ascending or descending order of their tightness. Note that strategies (2) and (3) would not be applicable in our eventual framework, since they must solve many CSPs each time. They are useful here as reference points, showing what might be achievable. The

Solving module simply solves the CSPs using the finite domain library [13], recording search effort. Since the library does not report constraint checks, we use the resumption statistic instead. We have confirmed that resumptions correlate well with cpu time [14].

First, we compare all strategies. In Figure 1, for the problem class <20,10,133, [65, 15]>, we show the resumption profit achieved (i.e. how much easier it is to solve the relaxed CSP compared to the original) by each strategy, removing up to 60 constraints each time. The problem class is in the over-constrained region, and in all cases we considered, remained over-constrained even after removing the constraints. The graph shows that removing low tightness constraints is the most profitable of the applicable strategies, for this problem class. We assume that although such constraints are easy to satisfy, they are likely to be redundant in the proof of inconsistency, since they rule out so few combinations, and thus they introduce many extra branches for no benefit.

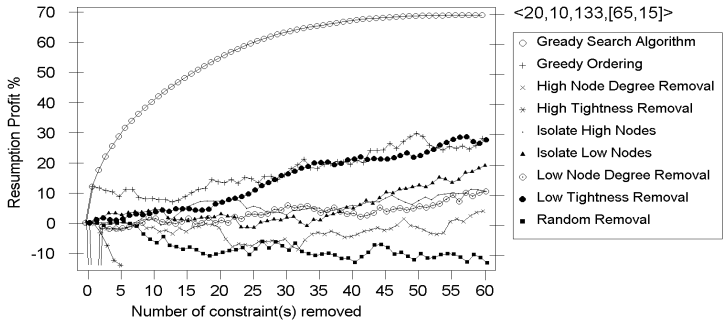

Fig. 1. Relaxation Strategies when removing up to 60 constraints

We then concentrate on the low tightness strategy, and Figure 2 illustrates its effect on 4 different problem classes. Whilst the graph shows a negative result for the <20,10,133,[45,15]> curve, the others are positive. In the best case we can create relaxed CSPs which are up to 45% easier to solve than the original CSPs, still without introducing a solution. The graphs suggest that Low Tightness Removal strategy works better on CSPs with high tightness and wide distribution

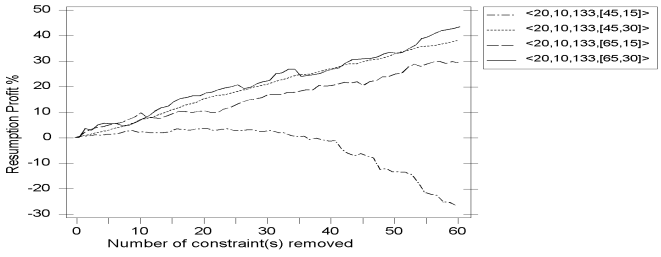

Fig. 2. Low Tightness Removal Strategy when removing 60 constraints

It might be thought that removing constraints from an overconstrained problem simply moves the problem leftwards up the hardness curve towards the transition

point. This is in fact the case for the Random Removal, which essentially follows the path of the standard curve. Does the same effect happen for Low Tightness Removal - i.e. is there a point at which removing constraints is no longer profitable? In Figure 3 we plot the search effort against the number of constraints in problems obtained by relaxing an original problem using Low Tightness Removal, and we include Random for comparison. We start with a particular problem class on the right hand side of the curve, remove constraints according to each strategy, and solve the new problem after each removal. In three cases, we avoid any significant hardness peak with the low tightness strategy, and for the fourth a peak comes only after removing approximately 40 constraints. That peak is further to the left than for Random, and the peak normally coincides with the solubility transition. Figure 4 shows the transition curves, and we can see that in four cases, the transition point for Low Tightness Removal is later (further left) than for Random. This gives us some confidence that we can use our relaxation strategy reliably without introducing new solutions.

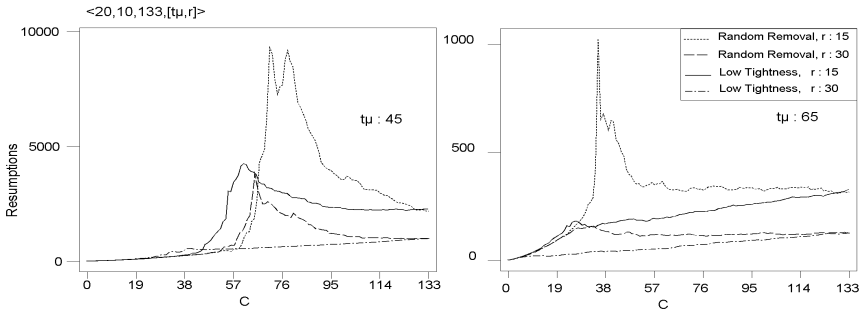

Fig. 3. Search effort for Random Removal vs. Low Tightness Removal

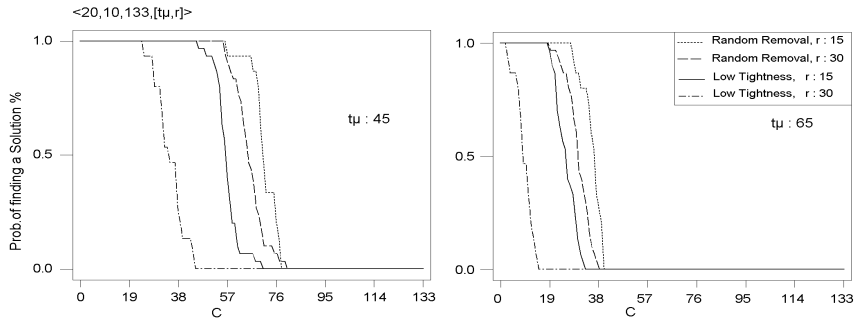

Fig. 4. Transition phase for Random Removal vs. Low Tightness Removal

Summary and Future Work

We have shown that relaxing CSPs can be useful for detecting inconsistency. In particular, we have shown that the simple strategy of removing constraints of low tightness is effective, reducing the time to prove inconsistency.

Future work will include: investigating the performance of Low Tightness on problems nearer the hardness peak; investigating non-binary and global constraints; other recognisable characteristics of CSPs (e.g. [15]); investigation of more theoretical concepts (e.g. higher consistency, problem hardness); real-world problems such as scheduling; heuristic methods, thus sacrificing our guarantee of correctness; and a case-library for selecting the appropriate relaxation strategy.

Acknowledgements

This work is supported under the EPSRC's grant number GR/N15764/01 and the Advanced Knowledge Technologies Interdisciplinary Research Collaboration, which comprises the Universities of Aberdeen, Edinburgh, Sheffield, Southampton and the Open University. We thank Mats Carlsson at SICTStus for helpful discussions about resumption, constraint checks and search effort.

References

1. S. White and D. Sleeman, "A Constraint-Based Approach to the Description & Detection of Fitness-for-Purpose." *ETAI*, vol. 4, pp. 155-183, 2000.
2. R. Barták, "Online Guide to Constraint Programming: http://kti.ms.mff.cuni.cz/~bartak/constraints/binary.html," 1998.
3. E. Tsang, "Foundations of Constraint Satisfaction." London & San Diego: Academic Press, 1993.
4. ILOG Solver, 5.3 ed. Paris: ILOG Inc., http://www.ilog.com/, 2003
5. SICStus Prolog, 3.9.1 ed. Kista: Swedish Institute of Computer Science, http://www.sics.se/sicstus/, 2001
6. E. Freuder, "Synthesizing constraint expressions", CACM, 1978, pp. 958--966.
7. E. Freuder and R. Wallace, "Partial Constraint Satisfaction", *Artificial Intelligence*, vol. 58, pp. 2170, 1992.
8. S. Bistarelli, U. Montanari, and F. Rossi, "Constraint Solving over Semi-rings", IJCAI'95, 1995, pp. 624--630.
9. T. Schiex, H. Fargier, and G. Verfaillie, "Valued Constraint Satisfaction Problems: hard and easy proble", IJCAI'95, 1995, pp. 631-637.
10. J. Hooker, "Logic-based methods for optimization : combining optimization and constraint satisfaction." New York, 2000, pp. 209.
11. E. MacIntyre, P. Prosser, B. Smith, and T. Walsh., "Random Constraint Satisfaction: Theory meets Practice", CP-98, 1998, pp. 325-339.
12. CSP-Suite, 1.90 ed. Aberdeen: University of Aberdeen, http://www.csd.abdn.ac.uk/~tnordlan/Proglog%20programs, 2002
13. SICStus, "Constraint Logic Programming over Finite Domains," in *SICStus Prolog User's Manual*, vol. Release 3.9.1, R. 3.9.1, Ed. Kista: Intelligent Systems Laboratory Swedish Institute of Computer Science, 2002, pp. 345-381.
14. T. E. Nordlander, "First Year Report," Aberdeen University, Aberdeen October 2002.
15. T. Walsh, "Search on high degree graphs", IJCAI-2001, 2001, pp. 266-274.