1-1-2006

Evaluation of solving methods for conditional constraint satisfaction problem

Mihaela C. Sabin
University of New Hampshire, Manchester, mihaela.sabin@unh.edu

Evaluation of Solving Methods for Conditional Constraint Satisfaction Problems

Mihaela Sabin
Department of Mathematics and Computer Science, Rivier College
420 Main Street, Nashua, NH 03060, U.S.A., msabin@rivier.edu

Esther Gelle
ABB Switzerland Ltd, Corporate Research
CH-5405 Baden, Switzerland, esther.gelle@ch.abb.com

Abstract

Conditional constraint satisfaction problems (CondCSPs) adequately capture problem change at solving time by conditionally identifying those variables and constraints that are relevant to final solutions. Real-world tasks with dynamic behavior, such as configuration, design, diagnosis, planning, and hardware test generation, have been modeled more naturally with CondCSPs. Such interest has been matched by the development of more effective algorithms that depart from classical backtracking and incorporate local consistency checking. Although performance results have been reported for these specialized algorithms, the experimental analysis has been conducted separately, using different test suites, and little is known about the algorithms' relative performance. In this abstract we present a CondCSP solver that implements direct and reformulation-based algorithms, each of which using forward checking and maintaining local consistency. In our experimental analysis we have considered randomly generated CondCSPs of diverse topologies in terms of problem density and satisfiability of the standard and conditional problem components. Execution time results show that there is not one winner but that reformulation solving in conjunction with forward checking performs better on problems with larger solution sets, while direct solving in conjunction with maintaining arc consistency is always preferred over direct solving using forward checking.

A conditional constraint satisfaction problem (CondCSP) extends the standard CSP with a condition-based component that models problem change by allowing for "on-the-fly" selection of subsets of variables that participate in problem solutions. The formalism, introduced by (Mittal & Falkenhainer 1990) under the name of dynamic CSP, has been renamed to conditional CSP (Sabin & Freuder 1998) to qualify the control component that models dynamic changes of the solution space with predefined conditions. The original application domain in (Mittal & Falkenhainer 1990) is product configuration, in which a changing rather than fixed number of components are part of final solutions. Selecting optional components to assemble configuration variants is naturally represented in CondCSP with condition-based constraints. More recently, conditional constraint satisfaction has been adapted to the planning domain (Do & Kambhampati 2000; Ambite *et al.* 2005; Miguel, Jarvis, & Shen 2000) and hardware test generation (Geller & Veksler 2005).

Each application domain has produced specialized algorithms for solving CondCSPs. Following the example of standard CSP solving, local consistency methods have been incorporated in CondCSP solvers and improved performance results have been reported. Although this solving approach has been observed across application domains, little is known about the algorithms' relative performance. Moreover, in striking contrast with the state-of-the-art of standard CSP solving, CondCSP class lacks systematic findings with regard to how algorithm efficiency correlates with problem topology, such as density, satisfiability, and conditionality. This challenge is compounded by an almost inexistent library of CondCSP benchmark problems.

To address these challenges, we have developed a CondCSP solver that includes two representative algorithms that have initially been proposed, implemetned, and evaluated separately. One algorithm has direct solving methods that adapt standard consistency checking, such as forward checking and maintaining arc consistency, to the special constraints that enforce conditionality in a CondCSP (Sabin 2003; Sabin, Freuder, & Wallace 2003). The other algorithm reformulates the original problem into intermediate CondCSPs with incrementally lesser conditionality as they are ultimately transformed into standard CSPs. Standard consistency checking is interleaved with problem reformulation to eliminate inconsistent subproblems and solve the resulting standard CSPs (Gelle 1998; Gelle & Faltings 2003). To overcome the lack of publicly known benchmark problems, we have used random CondCSPs and designed test suites for both direct and reformulation solving algorithms.

The first complete description of CondCSP backtrack search (Gelle 1998) solves a partially reformulated CondCSP, in which activity constraints of exclusion are rewritten as compatibility constraints. (Sabin 2003) proposed CondCSP analogs to CSP backtrack ($CondBt$), forward checking ($CondFc$), and maintaining arc consistency ($CondMac$) search algorithms. $CondMac$ interleaves backtrack search with maintaining arc consistency (Mac) adapted to propagate consistency checking on both compatibility and activity constraints in the original CondCSP. Experimental evaluation on random CondCSPs (Wallace 1996) shows up to two orders of magnitude of efficiency improvement over plain backtrack search.

A different solving approach is to successively process activation conditions of inclusion into an equivalent reformulation (Gelle 1998; Gelle & Faltings 2003). The reformulation algorithm, Gt, generates a tree whose internal nodes are CondCSPs and the leaves are standard CSPs. Gt reformulates inclusion activity constraints into compatibility constraints by creating intermediate CondCSPs with lesser conditionality until the leaves level is reached. Standard consistency checking is interleaved with tree generation to eliminate inconsistent subproblems and to solve in the end the resulting standard CSPs. Experimental evaluation results for Gt using forward checking were reported for solving a bridge design problem.

These algorithms contribute to CondCSP solving, but their development and evaluation took place separately and used different implementation frameworks (C++ vs. Lisp) and different experimental test suites. For the purpose of examining these algorithms' relative performance and how problem topology impacts performance we have integrated both implementations within the same solver.

We have analyzed experimentally the relative performance of the two types of algorithms by using randomly generated CondCSPs. The random CondCSP generator (Wallace 1996) extends the random standard CSP generator with *activity parameters*. In addition to the standard parameters of density and satisfiability of the problem compatibility constraints, d_c and s_c, we have d_a, *density of activity*, the probability of generating a non-initial variable as a target variable, and s_a, *satisfiability of activity*, the probability of generating a value in a domain as a condition variable. The test suites we designed generated different problem topologies with a problem size of number of variables $n = 10$, and number of values per domain $d_{size} = 10$. Density and satisfiability of compatibility and activity were varied.

We designed two test suites for our experiments. We compared the execution time, in number of seconds, for $CondFc$, $CondMac$, $GtFc$ and $GtMac$. In the first test suite density parameters were kept constant, $d_c = 0.3$ and $d_a = 0.5$, while satisfiability parameters varied: $s_c \in [0.2 \ldots 0.4]$, and $s_a \in [0.5 \ldots 0.8]$ (increments of 0.1). In the second test suite compatibility density was increased to $d_c = 0.5$, and the variation of the compatibility satisfiability was shifted up into the interval $s_c \in [0.5 \ldots 0.7]$. The activity parameters were maintained the same as in the first test suite.

The problems in the first test suite have lower density of compatibility and variable satisfiability of compatibility in a lower range than the problems in the second suite. We observe that in all cases $CondMac$ outperforms $CondFc$ while $GtFc$ outperforms $GtMac$ in most cases. There are two contributing factors in support of the latter observation. First, Gt transforms all exclusion constraints into compatibility constraints prior to the tree generation. Thus, the advantage of $MacActivity$ pruning in detecting activation conflicts does not manifest in Gt. Secondly, it is known that Mac's performance deteriorates with increasing problem density. In the case of Gt, the compatibility density of the resulting standard CSPs is larger than the original CondCSP's compatibility density, since all activity constraints have been reformulated into compatibility constraints. Overall, the pruning power of $MacCompatibility$ is overcome by the implementation overhead which, in the end, does not always pay off.

Another observation is that $GtFc$ has the best performance, thus outperforming $CondMac$, in the second test suite for problems with higher density and satisfiability of compatibility. Finally, we observe that, in general, decreasing satisfiability of activity costs both $CondFc$ and $CondMac$ more, while just the opposite is seen for Gt algorithms. Our test suites show that the number of solutions increases with decreasing satisfiability of activity. Gt's advantage of pruning whole subproblems during reformulation copes better with larger solution spaces (in the tens of thousands) than the direct solving methods.

References

Ambite, J.; Knoblock, C.; Muslea, M.; and Minton, S. 2005. Conditional constraint networks for interleaved planning and information gathering. *Intelligent Systems, IEEE [see also IEEE Intelligent Systems and Their Applications]* 20(2):25–33.

Do, M. B., and Kambhampati, S. 2000. Solving planning-graph by compiling it into CSP. In *Artificial Intelligence Planning Systems*, 82–91.

Gelle, E., and Faltings, B. 2003. Solving mixed and conditional constraint satisfaction problems. In *Constraints, 8(2):107–141, 2003.*

Gelle, E. 1998. *On the generation of locally consistent solution spaces*. Ph.D. Thesis, Ecole Polytechnique Fédérale de Lausanne, Switzerland.

Geller, F., and Veksler, M. 2005. Assumption-based pruning in conditional CSP. In *Principles and Practice of Constraint Programming*.

Miguel, I.; Jarvis, P.; and Shen, Q. 2000. Flexible graphplan. In *Proceedings of the Fourteenth European Conference on Artificial Intelligence*, 506–510.

Mittal, S., and Falkenhainer, B. 1990. Dynamic constraint satisfaction problems. In *Proceedings of the 8th National Conference on Artificial Intelligence*, 25–32. The MIT Press.

Sabin, M., and Freuder, E. 1998. Detecting and resolving inconsistency and redundancy in conditional constraint satisfaction problems. In *CP'98 Workshop on Constraint Problem Reformulation*.

Sabin, M.; Freuder, E. C.; and Wallace, R. J. 2003. Greater efficiency for conditional constraint satisfaction. In *Principles and Practice of Constraint Programming*, 649–663.

Sabin, M. 2003. *Towards Improving Solving of Conditional Constraint Satisfaction Problems.* Ph.D. Dissertation, University of New Hampshire, Durham, NH, U.S.A.

Wallace, R. 1996. *Random CSP Generator*. Constraint Computation Center, University of New Hampshire, Durham, NH, U.S.A.