Solving Methods for Conditional Constraint Satisfaction

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

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

Abstract

Typical real world tasks such as configuration or design exhibit dynamic aspects which require extending the basic constraint satisfaction framework. In this paper we use the notion of conditional constraint satisfaction (CondCSP) as defined by [Mittal and Falkenhainer, 1990] to provide a framework in which these problems can be formulated and solved. It has been shown in [Soininen *et al.*, 1999] that the decision problem for CondCSP is NP-complete similar to standard CSPs. Solving standard CSPs applies various degrees of local consistency to avoid recomputing infeasible branches in the search tree. In a similar manner, we define new types of local consistency aimed at solving conditional CSPs more efficiently. The different algorithms are compared using a series of randomly generated problems.

1 Introduction

Standard constraint satisfaction problems (CSPs) have been around for a while in order to solve highly combinatorial problems that appear in areas such as action planning, task scheduling, resource allocation, and personnel timetabling. Specialized CSP classes have emerged to capture more adequately specific characteristics of various application domains. Conditional constraint satisfaction problem (CondCSP) is one example of a CSP specialization that applies more effectively constraint technology to diagnosis [Sabin *et al.*, 2001], structural design [Gelle, 1998; Gelle and Faltings, 2003], and configuration [Mittal and Falkenhainer, 1990]. Specific to all these application domains is component optionality, which cannot be directly modeled in the standard CSP framework. The representational advantage of CondCSP over standard CSP is that it allows for a variable rather than predefined number of components to be part of the final design, configuration product, or diagnosis system.

Other CSP specializations have been proposed to represent more directly optional components and other characteristics of the configuration and design tasks. A mathematically well-founded framework for expressing conditional existence of variables is given in [Bowen and Bahler, 1991]. Compact descriptions of types of components and replications of identical component instances of a certain type are supported by generative CSPs [Haselböck, 1993; Stumptner *et al.*, 1994; Fleischanderl *et al.*, 1998]. Composite CSPs model aggregate structures and hierarchical organization of the component types [D. Sabin and Freuder, 1996].

Despite increasing interest in the area of improving the representation of the configuration task, there is limited progress in the area of improving, evaluating, and comparing the solving methods that operate on these representations. In this paper we present several solving methods for CondCSPs, and compare their efficiency on a set of randomly generated CondCSP instances. We also show that CondCSPs can be solved with specific solution algorithms significantly faster than the equivalent standard CSPs.

The conditional CSP formalism was introduced by Mittal and Falkenhainer more than ten years ago [Mittal and Falkenhainer, 1990] under the name of dynamic constraint satisfaction. The formalism expresses decisions that are enforced during the course of problem solving as conditional statements embedded in a standard CSP. Mittal and Falkenhainer show that in typical synthesis tasks, such as model composition or configuration, the set of variables relevant to a solution changes dynamically during the problem solving process. Based on this observation they extend the traditional static CSP formalism with *activity constraints*. Activity constraints describe conditions under which a variable is made active, i.e. part of a solution. This class of dynamic CSPs is renamed *conditional constraint satisfaction problems* [Sabin and Freuder, 1998] to (1) capture the nature of the control component that conditionally changes the initial model of the problem, and to (2) distinguish this class of problems from another class of dynamic CSPs that reuses problem solutions when problem changes over time [Dechter and Dechter, 1988; Bessière, 1991; Verfaillie and Schiex, 1994].

In [Gelle, 1998; Gelle and Faltings, 2003], we provide a solving method for a wide class of CondCSP involving discrete and continuous variables that generates stan-

dard CSP problem spaces from a mixed CondCSP problem statement, and uses standard local consistency to prune their respective solution spaces. Another way to solve CondCSPs is to adapt standard consistency techniques to the conditional domain such that these techniques handle both types of constraints in a CondCSP, standard, or compatibility constraints, and activity constraints [Sabin, 2003].

In [Soininen *et al.*, 1999], it is shown that CondCSPs are more expressive in the sense of knowledge representation than CSPs. The authors argue that it is difficult to capture dynamic aspects in a standard CSP, in which all variables are assigned values in every solution. One way to deal with optionally selecting subsets of variables from the entire variable set, is to add a special value NULL to the domains of the optional variables. However, when a conditional CSP is reformulated into a standard CSP using NULL values, a simple change in the conditional CSP representation cannot be accomplished by a simple update[1] in the corresponding representation of the NULL-based standard CSP reformulation. Thus, modularity of representation cannot be maintained in the reformulated standard CSP. The authors also propose a generalization of the CondCSP that is shown to be in NP while allowing the definition of disjunctions and default negation on constraints.

[Sabin, 2003] describes an algorithm which reformulates CondCSPs into standard CSPs. She then provides evidence that unless an efficient way to exploit the specific structure of the reformulated CSP is found, directly solving the CondCSP is significantly faster than solving the reformulated CSP. This is due to the increase in the arity of the reformulated constraints in the reformulated problem.

The paper is structured as follows. In section two, we will give a formal specification of a conditional CSP together with a small real-world example. We then introduce two different methods for solving conditional CSPs, each of which has three versions corresponding to backtrack search (BT), forward-checking (FC), and maintaining arc-consistency (MAC). In the fourth section, we give experimental results generated from a set of randomly generated problems. We conclude with an outlook to interesting open questions in the area of solving CondCSPs.

2 Conditional Constraint Satisfaction Problems (CondCSP)

First, we recall the original definitions of conditional constraint satisfaction problems introduced in [Mittal and Falkenhainer, 1990]. An instance $\mathcal{P}$ of CondCSP is of the form $\langle \mathcal{V}, \mathcal{D}, \mathcal{V}_\mathcal{I}, \mathcal{C}_\mathcal{C}, \mathcal{C}_\mathcal{A} \rangle$, where $\mathcal{V} = \{v_1, \ldots, v_n\}$ is the set of *variables* and $\mathcal{D} = \{D_1, \ldots, D_n\}$ is the set of *domains* of the variables providing a set $D_i = \{d_{i1}, \ldots, d_{ij}\}$ of *values* for each variable v_i. The set $\mathcal{V}_\mathcal{I}$, $\mathcal{V}_\mathcal{I} \subseteq \mathcal{V}$, is the set of *initial variables* of $\mathcal{P}$, $\mathcal{C}_\mathcal{C}$ is the set of *compatibility constraints*, and $\mathcal{C}_\mathcal{A}$ the set of *activity constraints*. We assume that all these sets are finite.

In contrast to a standard CSP, a variable of $\mathcal{P}$ has an *activity status* that determines the variable participation in solutions. A variable is *active* iff it is assigned a value in the course of problem solving. The set of *initial variables*, $\mathcal{V}_\mathcal{I}$, are the variables which have to be assigned a value in every solution of $\mathcal{P}$. Thus, these variables are always active. The non-initial or *hidden variables*, $\mathcal{V} - \mathcal{V}_\mathcal{I}$, have their activity status set by the activity constraints. Hidden variables have their initial activity status undefined, based on which they do not participate in solutions. Activity constraints act upon these variables in two ways: (1) by making them active or including them in problem solutions, (2) or by explicitly excluding them from solutions. Otherwise, the variables remain hidden.

A compatibility constraint c with arity j specifies the set of allowed combinations of values for a set of variables $v_1, \ldots, v_j$ as a subset of the Cartesian product of the domains of the variables. We denote the subset by $c(v_1, \ldots, v_j)$, i.e. $c(v_1, \ldots, v_j) \subseteq D_1 \times \ldots \times D_j$. A compatibility constraint of $\mathcal{P}$ is *relevant* to the search process iff all the variables it constrains are active.

In addition, the activity constraints of $\mathcal{P}$ allow the manipulation of variable activity. According to the definitions in [Mittal and Falkenhainer, 1990], we distinguish between *require*, or what we call *activity of inclusion* constraints, and *require not*, or what we call *activity of exclusion* constraints. An inclusion activity constraint of form $c \overset{incl}{\rightarrow} v$ makes v active or includes it into the search space iff c constrains active or included variables and holds in $\mathcal{P}$. We call c *activation condition*, shortly *condition*. In general, the condition has the form of a compatibility constraint. An exclusion activity constraint of the form $c \overset{excl}{\rightarrow} v$ *excludes* variable v iff c involves only active variables and holds in $\mathcal{P}$.

In order to simplify the discussion and comparison of different algorithms, we impose some restrictions on the form of the constraints in this paper. We assume that the compatibility constraints are binary. Furthermore, we restrict activity constraints to the form $a_l : v_i = d_{ij} \overset{incl}{\rightarrow} v_k$, i.e. to activity constraints whose condition is a unary constraint of the form $v_i = d_{ij}$. Thus, activity constraints are binary too. The variable v_i is called *condition variable*, v_k is the *target variable*. The value d_{ij} is a *condition value* in the domain of v_i and triggers the inclusion or exclusion of the target variable v_k. A solution to a CondCSP is an assignment of active variables that satisfies all compatibility and activity constraints.

An assignment $\mathcal{A}$ for a CondCSP $\mathcal{P} = \langle \mathcal{V}, \mathcal{D}, \mathcal{V}_\mathcal{I}, \mathcal{C}_\mathcal{C}, \mathcal{C}_\mathcal{A} \rangle$ satisfies the constraints in $\mathcal{P}$ iff the following conditions hold:

1. $\mathcal{A}$ satisfies each compatibility constraint $c \in \mathcal{C}_\mathcal{C}$, i.e., if c's variables are active in $\mathcal{A}$ then the variables

[1] A simple update consists of a number of i) addition of constraints, removals of allowed tuples from constraints, additions of values to variable domains, and addition of variables with their domains, or ii) removal of constraints, additions of allowed tuples, additions of values to domains, and additions of variables [Soininen *et al.*, 1999].

constrained by c are assigned values allowed by c; otherwise, c's variables are not all active (we say c is trivially satisfied).

2. $\mathcal{A}$ satisfies each activity constraint $a \in \mathcal{C}_{\mathcal{A}}$, where $a : c \overset{incl}{\rightarrow} v_k$ or $a : c \overset{excl}{\rightarrow} v_k$, i.e., if c's variables are active and satisfied by $\mathcal{A}$ then v_k is active in, or, respectively, excluded from $\mathcal{A}$; otherwise, c's variables are not all active or they are active but c does not hold in $\mathcal{A}$ (we say a is trivially satisfied and does not affect v_k's activity status).

An assignment $\mathcal{A}$ is a solution to a CondCSP iff $\mathcal{A}$'s variables are active and $\mathcal{A}$ satisfies all the constraints.

We use the simplified configuration task of an industrial mixer [Soininen *et al.*, 1999] to illustrate a CondCSP. The components and their properties are represented as variables with values. For example, vessel volume can be large or small (Table 1). The mixer type (conventional mixer, and reactor, tank) and the types of mixing processes (dispersion, suspension, and blending) are represented as values of the variables mixer and mixing processes. The components cooler and condenser are optional. Thus, they are not part of the set of initial variables but are included or excluded by activity constraints $\mathcal{C}_{\mathcal{A}} = \{a_1, a_2, a_3\}$ shown in Table 2. Additionally, compatibility between components is defined by the constraints $\mathcal{C}_{\mathcal{C}} = \{c_1, c_2\}$. Given the above mixer CondCSP, the assignment $\mathcal{A}_1$=$\{Mi = m, Mp = b, V = s\}$ is a solution, i.e. a configuration, since its variables are active and it satisfies the constraints. The assignment $\mathcal{A}_2$=$\{Mi = r, Mp = s, V = s, Coo = coo1\}$ is another solution with a different set of active variables.

However, $\mathcal{A}_3$=$\{Mi = m, Mp = b, V = s, Coo = coo1\}$ is not a solution since the assignment $Coo = coo1$ is not justified by Coo being in the set of initial variables or by being included by an inclusion activity constraint.

The basic task in the CondCSP framework is to find solutions or to show that there is no solution to a given CondCSP. In the former case we can distinguish between finding one solution (the first one) and enumerating all solutions. In the mixer example, 14 solutions are found (Figure 3). Basically, there are four types of solutions: solutions without cooler nor condenser, those containing a cooler or a condenser, and one containing both.

In the remainder of this paper we will concentrate on particular solving methods and compare their efficiency in terms of execution times and various consistency check measures.

Table 1: Variables of the mixer configuration model

Variable	**Domain**
Mp(Mixing process) $\in \mathcal{V}_{\mathcal{I}}$	{d,s,b}
Mi(Mixer) $\in \mathcal{V}_{\mathcal{I}}$	{m,r,t}
Coo(Cooler)	{coo1, coo2}
Con(Condenser)	{con1,con2}
V(Volume) $\in \mathcal{V}_{\mathcal{I}}$	{l,s}

Table 2: Compatibility and activity constraints of the mixer configuration model.

$$
\begin{array}{rcl}
a_1 & = & (Mi = r \overset{incl}{\rightarrow} Coo) \\
a_2 & = & (Mp = d \overset{incl}{\rightarrow} Con) \\
a_3 & = & (Coo = coo1 \overset{excl}{\rightarrow} Con) \\
a_4 & = & (V = l \overset{incl}{\rightarrow} Con) \\
c_1(Con, V) & = & \{(\text{con1,l}),(\text{con2,l}),(\text{con2,s})\} \\
c_2(Mi, V) & = & \{(\text{r,s}),(\text{m,s}),(\text{t,l})\}
\end{array}
$$

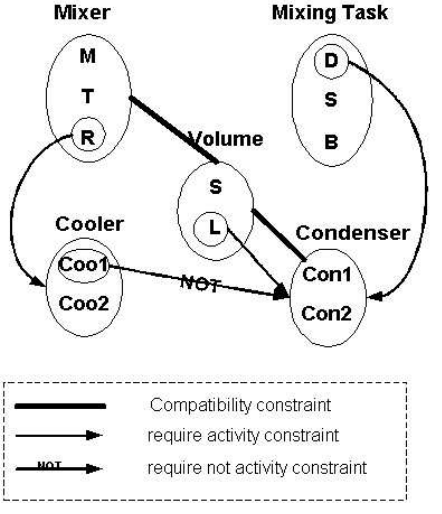


Table 3: *All solutions of the mixer configuration task.*

Mi	**Mp**	**V**	**Coo**	**Con**
r	d	s	coo2	con2
r	b	s	coo1	-
r	b	s	coo2	-
r	s	s	coo1	-
r	s	s	coo2	-
m	d	s	-	con2
t	s	l	-	con1
t	b	l	-	con1
t	d	l	-	con1
t	s	l	-	con2
t	b	l	-	con2
t	d	l	-	con2
m	b	s	-	-
m	s	s	-	-

3 Solving Methods for Conditional Constraint Satisfaction

We draw our attention to three solving methods. The first two have been developed to operate on conditional CSP representations. The third one first reformulates the original CondCSP into a standard CSP and then uses standard CSP solving methods.

The first algorithm, GDCSP [Gelle, 1998; Gelle and Faltings, 2003], has been developed in the context of solving mixed conditional CSPs, i.e. constraint-based

problems that involve discrete and continuous variables. In particular constraints defined on both types of variables are well mapped to the model of activity constraints. The algorithm proceeds by transforming the conditional problem into a set of standard CSPs that involve only variables with domains and compatibility constraints defined on them. It successively combines activation conditions to determine the variables and compatibility constraints of each generated standard CSP. The result of the first step in the algorithm is a tree of generated standard CSPs. To solve them, any kind of standard CSP search method can be applied. An improvement to this basic algorithm consists of interleaving the problem tree generation with standard local consistency checks, thereby pruning some of the CSPs in the tree that are locally inconsistent. For details please refer to [Gelle, 1998; Gelle and Faltings, 2003].

In Figure 1, we show the entire tree of CSPs generated by the algorithm GDCSP. In this algorithm, a_3 is first transformed into a compatibility constraint[2]. We start from the set of initially active variables $\mathcal{V}_\mathcal{I}$. The order of activation in the remaining set of inclusion activity constraints $\{a_1, a_2, a_4\}$ is arbitrary since the variables in all activation conditions are part of $\mathcal{V}_\mathcal{I}$ and thus active. We choose to activate the constraints in the order a_1, a_2, and a_4. The generation results in six problem spaces. Even the simple backtrack search method detects the inconsistency between $Mi = r$ and $V = l$ and does not create those specific problem spaces. It can also be seen how redundant problem spaces are generated in the second branch with $Mi \in \{b, s\}$ due to the simultaneous activation of a_2 and a_4.

The second algorithm solves discrete conditional CSPs with inclusion and exclusion activity constraints. It extends local consistency methods such as forward-checking (FC) and maintaining arc-consistency (MAC) to activity constraints and active variables ([Sabin, 2003]). The algorithm maintains the activity status of all

[2]It has been shown in [Haselböck, 1993] that it is always possible to transform an exclusion activity constraint into a compatibility constraint.

Figure 1: All problem spaces of the mixer example generated by GDCSP.

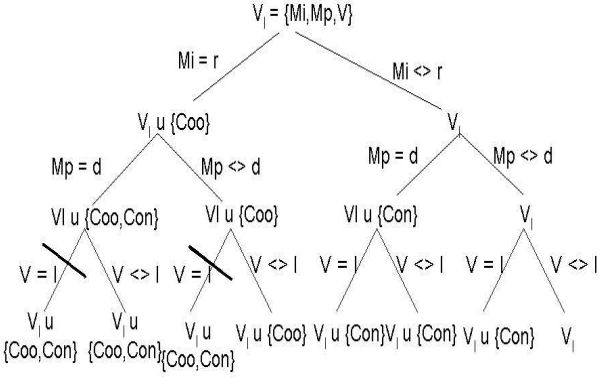
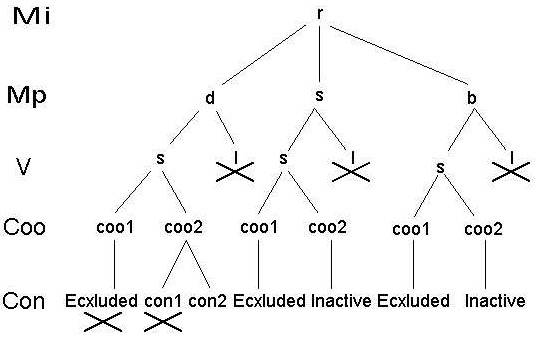


Figure 2: Solution branch $Mi = r$ of the algorithm CCSP-BT.



variables explicitly (active and excluded). When applying BT, activity constraints are checked in the context of the current instantiation of a variable to its value. If this value is consistent with the activity condition, the activity constraint is executed, i.e. a new variable is added to the problem or the respective variable is excluded from the problem. The activity constraint check fails if the inclusion or exclusion of the target variable leads to a contradiction with the activity status of the respective target variable. This happens when the constraint either includes a target variable, but that target variable is already included, or vice versa. The variant algorithm with FC applies in addition forward checking filtering to the variables newly added to the the problem through activity constraints to remove values inconsistent with the current assignments. FC has no effect on variables excluded from the current search context. MAC goes one step further and makes new variables added through activity constraints arc-consistent with all uninstantiated, but active variables. In addition, MAC propagates activity constraints by eliminating condition values that contradict the activity status of the active or excluded variables. The algorithms are explained in detail in [Sabin, 2003].

In figure 2, we show the solution branch with $Mi = r$ solved with the algorithm CCSP-BT, i.e. CCSP applying backtracking. We start with assigning the value r to Mi. This assignment is relevant to a_1 and activates the variable Coo. Then, the value d is assigned to Mp. This in turn activates the variable Con due to the activity constraint a_2. After this, the variable V is assigned the value s, which is consistent with constraint c_2. When the value $coo1$ is assigned to Coo, a conflict is detected since the constraint a_3 excludes the variable Con, which has been activated before. Thus the algorithm backtracks and assigns the value $coo2$ to Coo. As a next step, the value $con1$ is tried and fails with constraint c_1. Eventually the first solution is found with $Con = con2$. The algorithm backtracks and finds the other solutions similarly.

The third way to solve conditional CSPs is to first reformulate the conditional CSP into a standard CSP, and to solve the reformulated problem with standard search

methods. [Sabin, 2003] defines a new reformulation algorithm that deals with the general case in which multiple activity constraints, which might form cycles, activate the same target variable. The reformulation produces non-binary constraints. Therefore, these reformulations are first transformed in binary representations and then solved with standard MAC for binary CSPs.

We implemented all three algorithms within the same object-oriented framework based on C++ and its standard template library. In the following we will compare their efficiency in solving the mixer problem as well as randomly generated discrete problems. For this we employ the following naming scheme: GDCSP-BT, GDCSP-FC, GDCSP-MAC for the first algorithm and its three versions; CCSP-BT, CCSP-FC, CCSP-MAC for the three versions of the second algorithm; and RefCSP-MAC for the reformulated version solved with MAC for standard binary CSPs.

4 Experimental Results

In this section, we report our results on the solution methods CCSP-BT, CCSP-FC, CCSP-MAC and GDCSP-BT, GDCSP-FC, GDCSP-MAC presented in the previous section. As example problems we chose the simple mixer configuration task presented in the previous sections and a set of randomly generated conditional CSPs created with a random generator described in [Wallace, 1996; Sabin, 2003]. The generator creates conditional CSPs pertaining to specific problem classes that are described as follows: number of variables, maximum domain size of the variables, density d_C and satisfiability s_C for compatibility constraints, and activity density d_A and satisfiability s_A for activity constraints. The density characterizes the probability of generating a constraint, while the probability of generating value pairs in a compatibility constraint determines the problem satisfiability. For the conditional problem classes, the density of activity denotes the probability of generating a non-initial variable as target variable, while the probability of generating a value in a domain as condition value characterizes the satisfiability of activation[3].

The mixer problem has 5 variables with a maximum domain size of 3. Its problem space includes 162 possible value combinations and it has 14 solutions. The number of solution spaces found by GDCSP is 6 as shown in Figure 1. Execution time is below 1 ms for all solution methods.

The randomly generated conditional CSPs pertain to the class of problems with 8 variables, 6 values per variable domain (problem size: $8^6 = 1{,}679{,}616$). To maintain size and execution time per reformulated CSP within an acceptable range, density d_C is set to 0.15 and satisfiability s_C varies between 0.1 and 0.9 in steps of 0.1, while the activity satisfiability s_A and activity density are fixed to 0.3. We generated for each satisfiability 100 problem instances and solved them with all algorithm variants. It can be noticed in Figure 3 that the execution times differ dependent on the type of filtering used. GDCSP algorithms generating trees of CSPs show faster execution times than their corresponding CCSP variants that use the same filtering. We conjecture that GDCSP gets rid of entire standard CSPs that prove to be locally inconsistent in the problem space tree while CCSP rediscovers them during value enumeration.

It is interesting to note that the execution time curves for both BT algorithms show a similar shape with peaks at satisfiability 0.3, 0.8 and to a lesser extent at 0.6. The FC and MAC algorithms provide a smoother curve with a weak peak at satisfiability 0.4. Note that CCSP-MAC is resolved faster than CCSP-FC while the same is not true for the corresponding GDCSP variants. With increasing satisfiability the GDCSP-MAC variant generates more overhead due to larger pruning than the GDCSP-FC variant.

When we compare the most efficient method of CCSP, CCSP-MAC to the effort of solving the same problem classes as reformulated CSPs, we can state that the di-

[3]The entire model includes further parameters, which we do not describe here, for a complete description please refer to [Wallace, 1996; Sabin, 2003]

Figure 3: Execution times of GDCSP and CCSP solving classes of 100 problems with 8 variables of 6 values each and with $d_C = 0.15$, s_C varying between 0.1 and 0.9 and $s_A = d_A = 0.3$.

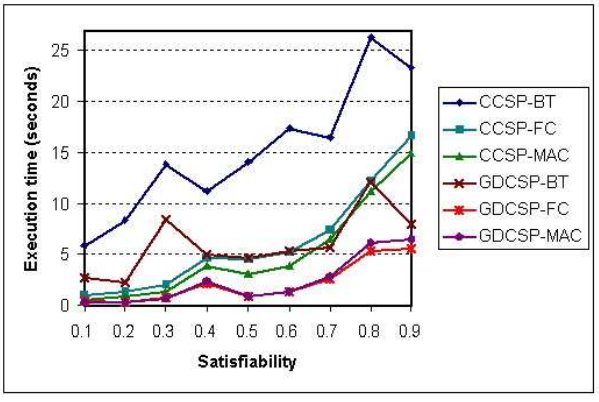


Figure 4: Execution times of the reformulated CSP (RefCSP) compared to CCSP-MAC solving the same problem classes.

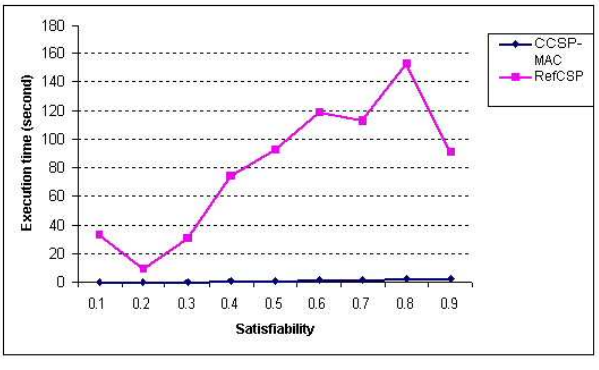
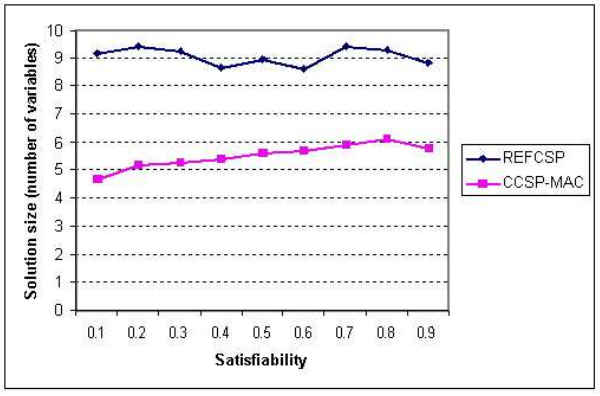


Figure 5: Average solution size, i.e. number of variables, of the reformulated CSPs (refCSP) and the CondCSPs.



rect solving methods, in this case CCSP-MAC, clearly outperform standard solving methods for the reformulated CSP. This can be explained by an increase in the domain size of variables as well an increase in variables participating in the solution (Figure 5).

5 Conclusion

We presented two solving methods for CondCSPs, the GDCSP method generating trees of standard CSPs and the CCSP method adapting forward checking (FC) and maintaining arc consistency (MAC) to the conditional domain. Since real-world problem can often be formulated as conditional CSPs, there is a need to improve solving methods for this kind of CSPs. In addition, we provide evidence on randomly generated CSPs that solving the reformulated CSPs with standard CSP methods is much less efficient (Figure 4). Given this evidence, we would like to stimulate further research in the area of special classes of CSPs such as CondCSPs. Further research topics comprise the application of improved solving methods to real-world configuration problems, elaborate ways to measure the difficulty of solving conditional CSPs, and heuristics applied to conditional CSP solving.

References

[Bessière, 1991] C. Bessière. Arc-consistency in dynamic constraint satisfaction problems. In *Proceedings of the 9th AAAI*, pages 221–226, 1991.

[Bowen and Bahler, 1991] J. Bowen and D. Bahler. Conditional existence of variables in generalized constraint networks. In *Proceedings of AAAI-91*, pages 215–220, 1991.

[D. Sabin and Freuder, 1996] D. Sabin and E. C. Freuder. Configuration as composite constraint satisfaction. *Proceedings of the Artificial Intelligence and Manufacturing Research Planning Workshop*, pages 153– 1612, 1996.

[Dechter and Dechter, 1988] R. Dechter and A. Dechter. Belief maintenance in dynamic constraint networks. In *Proceedings of AAAI-88*, pages 37–42, 1988.

[Fleischanderl *et al.*, 1998] G. Fleischanderl, G. Friedrich, A. Haselböck, H. Schreiner, and M. Stumptner. Configuring large systems using generative constraint satisfaction. In *IEEE Intelligent Systems*, volume 13, pages 59–68, 1998.

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

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

[Haselböck, 1993] A. Haselböck. *Knowledge-based Configuration and Advanced Constraint Technologies*. PhD thesis, Technical University of Vienna, 1993.

[Mittal and Falkenhainer, 1990] S. Mittal and B. Falkenhainer. Dynamic constraint satisfaction problems. In William Dietterich, Tom; Swartout, editor, *Proceedings of the 8th National Conference on Artificial Intelligence*, pages 25–32. MIT Press, 1990.

[Sabin and Freuder, 1998] M. Sabin and E. C. Freuder. Detecting and resolving inconsistency and redundancy in conditional constraint satisfaction problems. In *Web-published papers of the CP'98 Workshop on Constraint Problem Reformulation*, Pisa, Italy, October 1998.

[Sabin *et al.*, 2001] M. Sabin, R. D. Russell, and I. Miftode. Using constraint technology to diagnose errors in networks managed with spectrum. In *Proceedings of the IEEE International Conference on Telecommunications*, 2001.

[Sabin, 2003] Mihaela Sabin. *Towards More Efficient Solution of Conditional Constraint Satisfaction Problems*. Ph.D. Thesis, University of New Hampshire, Durham, NH 03824, U.S.A., 2003.

[Soininen *et al.*, 1999] T. Soininen, E. Gelle, and I. Niemelä. A fixpoint definition for dynamic constraint satisfaction. *Principles and Practice of Constraint Programming, CP'99*, 1999.

[Stumptner *et al.*, 1994] M. Stumptner, A. Haselbck, and G.Friedrich. Cocos - a tool for constraint-based, dynamic configuration. *Proceedings 10th CAIA, San Antonio*, 1994.

[Verfaillie and Schiex, 1994] G. Verfaillie and T. Schiex. Solution reuse in dynamic constraint satisfaction problems. In *Proceedings of the 12th AAAI*, pages 307–312, Seattle, WA, 1994.

[Wallace, 1996] R. J. Wallace. Random CSP generator. http://www.cs.unh.edu/ccc/code.html, Constraint Computation Center, University of New Hampshire, Durham, 1996.