Greater Efficiency for Conditional Constraint Satisfaction

Mihaela Sabin[1], Eugene C. Freuder[2], and Richard J. Wallace[2]

[1] Department of Mathematics and Computer Science
Rivier College, 420 Main Street, Nashua, NH 03060, USA
msabin@rivier.edu
[2] Cork Constraint Computation Center, Department of Computer Science
University College Cork, Cork, Ireland
{e.freuder,r.wallace}@4c.ucc.ie

Abstract. A conditional constraint satisfaction problem (CCSP) extends a standard constraint satisfaction problem (CPS) with a condition-based component that controls what variables participate in problem solutions. CCSPs adequately represent configuration and design problems in which selected subsets of variables, rather than the entire variable set, are relevant to final solutions. The only algorithm that is available for CCSP and operates directly on the original, unreformulated CCSP statement has been basic backtrack search. Reformulating CCSPs into standard CSPs has been proposed in order to bring the full arsenal of CSP algorithms to bear. One reformulation approach adds null values to variable domains and transforms CCSP constraints into CSP constraints. However, a complete null-based reformulation of CCSPs has not been available. In this paper we provide more advanced algorithms for CCSP and a full null-based reformulation into standard CSP. Thorough testing reveals that the advanced algorithms perform up to two orders of magnitude better than plain backtracking, but that realizing practical advantages from reformulation is problematic. The advanced algorithms extend forward checking and maintaining arc consistency to CCSPs. The null-based reformulation improves on the preliminary findings in [1] by removing the limitation on multiple activation, and by localizing changes. It identifies and addresses a difficulty presented by activity cycles.

1 Introduction

There are many important and complex tasks to which constraint satisfaction has been successfully applied. As a result, specialized constraint satisfaction problem (CSP) classes have emerged to cope more directly with specific characteristics of various application domains. Qualifiers such as partial, dynamic, hierarchical, composite, interval, mixed, and others characterize CSP specializations that have been studied in the last decade. The conditional constraint satisfaction is another specialization developed to cope with the special features of diagnosis and configuration problems.

F. Rossi (Ed.): CP 2003, LNCS 2833, pp. 649–663, 2003.

Conditional CSP extends standard CSP with a condition-based component that models dynamic changes of problem solutions with predefined conditions. The formalism has been introduced in [2] under the name of dynamic CSP. It integrates classical constraint satisfaction with a special type of constraint, *activity constraints*, responsible for selecting those variables that should participate in solutions. The formalism has been originally motivated by synthesis tasks such as product configuration, in which not all cataloged components are present in every single configured product. This class of dynamic CSPs is renamed *conditional constraint satisfaction problems (CCSPs)* [3] 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 [4,5,6].

Since its first formalization in 1990, conditional constraint satisfaction paradigm has been used for modeling not only configuration problems, but also diagnosis [7], design [1], and network management [8] problems. Despite increasing interest in the area of representing application problems as CCSPs, little progress has been made in the area of improving direct solving methods that operate on CCSP representations. In contrast with other CSP specializations, no standard CSP solving method, except for backtrack search [1], has been adapted to the conditional domain. The lack of specialized, direct solving methods is compounded by the fact that a benchmark test base for this type of problems is extremely limited [9,10], although very much needed in experimental evaluations.

In this paper we present two advanced methods for solving CCSPs that use local consistency methods of forward checking and maintaining arc consistency. Solving methods find values for the set of active variables. These are obtained from the initial set of variables that are assigned values in every solution, and variables that are newly incorporated into the problem via activity constraints. The technical challenges encountered and overcome in extending forward checking and maintaining arc-consistency to CCSP are to: (1) keep track of variables' activity status as determined by consistency checking of activity constraints, (2) enforce chosen level of consistency when checking both compatibility and activity constraints, (3) in case of maintaining arc consistency, extend arc consistency with activation consistency along activity constraints.

The opportunity of importing efficient standard algorithms, whose behavior has been extensively tested, raises new challenges. Are there available similarly comprehensive experimental studies for evaluating CCSP solving? The reality of many application domains, such as configuration or diagnosis, is that either real-life problems data is not publicly available or problem examples are too simple. A practical approach that overcomes this difficulty and has proved very successful for benchmarking standard solving algorithms is to use randomly generated CSPs. This is the approach we consider in this paper to evaluate empirically the proposed algorithms. We extend a random standard CSP generation model [11] to produce random activity constraints, and use the model to implement a random conditional CSP generator. We generate large and diverse problem populations to conduct experimental studies that time algorithm execution, and

count search operations specific to standard and conditional CSP solving. The testing reveals that the advanced algorithms perform up to two orders of magnitude better than plain backtracking.

An alternative approach to directly solving CCSP is to reformulate a CCSP into an equivalent standard CSP. This approach has the advantage of bringing to bear a mature constraint technology developed in the standard domain. The first reformulation of conditional CSP into standard CSP has been mentioned by Mittal and Falkenhainer [2], although they have not presented a full description of the transformation. They consider the addition of a special value, called "null", to the domains of all variables which are not initially active. A variable instantiation with "null" indicates that the variable does not participate in the problem solution. The feasibility of obtaining a null-based CSP reformulation of a CCSP has been examined in-depth by Gelle [1]. She develops a null-based reformulation algorithm that imposes the following limitation on CCSPs: non-initial variables are activated by at most one activity constraint. A transformation of multiple activations of the same variable has not been considered on the grounds of an additional limitation, i.e., the transformation does not preserve locality of change [1,9].

We have developed an algorithm of null-based reformulation that removes these limitations. The algorithm (1) transforms multiple activations, and, (2) preserves locality of change by allowing a less restrictive local change than the one defined in [9]. Moreover, we have identified a new difficulty with null-based reformulation introduced by activity cycles. We have developed an alternative null-based reformulation algorithm that overcomes this difficulty at the cost of not preserving locality of change. We have evaluated experimentally the performance of solving the reformulated standard problem and compared it with results obtained from applying direct solving methods to the original problem. The findings show that the advanced solving methods are faster by one to two orders of magnitude than solving the equivalent, reformulated problem.

2 Conditional CSP: An Example

Before we recall the definition of the conditional CSP, we give an example of a simple product configuration task for which we develop a CCSP representation. The insights of the modeling exercise facilitate the introduction of the basic concepts of the CCSP framework. The example is a simplified version of an example introduced by [2] and specifies a car configuration task (Figure 1). The specifications include:

- required components, that participate in all final car configurations, with their values;
- optional components, that can be optionally selected according to certain configuration requirements, with their values;

– configuration requirements of compatibility, that restrict the values of the selected components according to product assembly requirements and promotional sales strategies;
– configuration requirements for selecting optional components, that express customer preferences and additional requirements with regard to assembling and selling the product.

Given the specified components and requirements, the task of configuration is to assign values to selected components in such a way that requirements pertinent to what is selected are satisfied. To obtain a CCSP representation of the car

Required components and their values
- comfort package has luxury, deluxe, and standard values
- frame has convertible and sedan values

Optional components and their values
- sunroof has sr1 and sr2 values
- air conditioner has ac1 and ac2 values

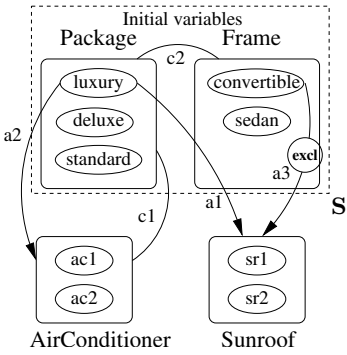
Configuration requirements of compatibility among component values
1. standard comfort package is not compatible with ac2 air conditioner
2. luxury comfort package is not compatible with ac1 air conditioner

Configuration requirements for selecting optional components
1. luxury comfort package includes sunroof option
2. luxury comfort package includes air conditioner option
3. convertible frame excludes sunroof option

Fig. 1. A simple car configuration task example

configuration example, we identify problem variables, values, and constraints. We apply the following modeling guidelines and produce the CCSP representation in Figure 2.

– Configuration task components and their values correspond to problem *variables* and their associated *domains of values*. Required components, which are part of any configuration solution, are distinguished as *initial variables*. Initial variables have the property of being initially *active* or *included* in the problem search space. Optional components have their *activity status* initially undefined as they are not selected to either participate in, that is, be included, or to explicitly not participate in, or be excluded from, problem solutions.
– The requirements of component compatibility are modeled as *compatibility constraints*, which restrict the combinations of allowed values assigned to selected components.
– *activity constraints* change the initial variable set according to certain conditions. These conditions control which optional components get selected in a configuration, and which optional components are removed from a configuration.



Activity Constraints

$a_1 : Package = luxury \xrightarrow{\text{incl}} Sunroof$

$a_2 : Package = luxury \xrightarrow{\text{incl}} AirConditioner$

$a_3 : Frame = convertible \xrightarrow{\text{excl}} Sunroof$

Compatibility Constraints

$c_1^{disallowed} : \{(luxury\ ac1)\}$

$c_2^{disallowed} : \{(standard\ convertible)\}$

Solution Set

Frame	*Package*	*Sunroof*	*AirConditioner*
convertible	*deluxe*	EXCL	UNDEF
sedan	*luxury*	sr_1	a_2
sedan	*luxury*	sr_2	a_2
sedan	*luxury*	UNDEF	UNDEF
sedan	*standard*	UNDEF	UNDEF

Fig. 2. Conditional CSP representation of the car configuration task example

This description of modeling a configuration task as a CCSP identifies five problem components. Thus, a conditional constraint satisfaction problem, $\mathcal{P} = \langle \mathcal{V}, \mathcal{D}, \mathcal{V}_\mathcal{I}, \mathcal{C}_\mathcal{C}, \mathcal{C}_\mathcal{A} \rangle$, involves a set of variables, $\mathcal{V} = \{v_1, \ldots, v_n\}$, which, if active, can take on discrete values from their corresponding finite domains $\mathcal{D} = \{D_{v_1}, \ldots, D_{v_n}\}$, a non-empty set of initially active variables, called initial variables, $\mathcal{V}_\mathcal{I}$, $\mathcal{V}_\mathcal{I} \subseteq \mathcal{V}$, a set of compatibility constraints, $\mathcal{C}_\mathcal{C}$, and a set of activity constraints, $\mathcal{C}_\mathcal{A}$. All sets are finite.

The CCSP model in Figure 2 has four variables, two compatibility constraints $\{c_1, c_2\}$, and three activity constraints, $\{a_1, a_2, a_3\}$, of which a_1 and a_2 are inclusion activity constraints, and a_3 is an exclusion activity constraint. Two of the problem variables, *Package* and *Frame* are initial variables and, therefore, active. They participate in all solutions and define the initial search problem with which the solving process starts. The non-initial variables, *AirConditioner* and *Sunroof*, have their activity status initially undefined. Their participation in solutions is determined by activity constraints.

We say that a compatibility constraint c is consistent with an instantiation $\mathcal{I}$ of the constraint variables iff either not all constraint variables are active, or constraint variables are active and c satisfies $\mathcal{I}$. For example, the instantiation $Package = standard$ and $Frame = convertible$ trivially satisfies c_1 since the constraint variable *AirConditioner* is not active.

An inclusion activity constraint, $a : a_{cond} \xrightarrow{\text{incl}} v_t$, has an activation condition, a_{cond}, which is a regular constraint defined on a set of condition variables, and a target variable, v_t. We say that a is consistent with an instantiation $\mathcal{I}$ of the activation variables of a_{cond} iff either (1) not all condition variables are active, or $\mathcal{I}$ is inconsistent with a_{cond}, or (2) all condition variables are active, $\mathcal{I}$ satisfies a_{cond}, and v_t is active. The example's activity constraints of inclusion are

a_1 and a_2. The instantiation $Package = luxury$ makes $AirConditioner$ active according to a_1, and $Sunroof$ active according to a_2. In both cases, condition variable $Package$ is active and the instantiation satisfies the activation condition of a_1 and a_2.

Given an exclusion activity constraint, $a : a_{cond} \xrightarrow{\text{excl}} v_t$, we say that a is consistent with an instantiation $\mathcal{I}$ of the activation variables a_{cond} iff (1) either not all condition variables are active, or $\mathcal{I}$ is inconsistent with a_{cond}, or (2) all condition variables are active, $\mathcal{I}$ satisfies a_{cond}, and v_t is not active. a_3 is an example of an exclusion activity constraint. The instantiation $Frame = convertible$ makes $Sunroof$ not active, since condition variable $Frame$ is active and the instantiation satisfies the activation condition of a_3. Note that this instantiation does not involve condition variables of either of the inclusion activity constraints.

A solution to a CCSP $\mathcal{P}$ is an instantiation of a set of active variables such that all compatibility and activity constraints are satisfied. All solutions to the example problem are listed in Figure 2.

3 Solving Methods

The domain of standard CSPs benefits from a rich collection of thoroughly tested algorithms. In contrast, the study of conditional CSPs is still in its infancy with little research directed to specialized solving methods that operate directly on CCSP representations. Following the model of other CSP specializations, we develop adaptations of the most representative standard CSP methods for the conditional domain:

- modified backtrack search algorithm (CondBT) that handles both types of activity constraints,
- new forward checking algorithm (CondFC) that propagates compatibility constraints over active variables, and
- new maintaining arc-consistency algorithm (CondMAC) that propagates both compatibility and activity constraints.

In Section 4, the relative performance of the proposed methods is analyzed experimentally by using random CCSPs. We show that (1) the run-time complexity order in the standard domain holds in the conditional domain, i.e., CondBT < CondFC < CondMAC, and that (2) the advanced algorithms CondFC and CondMAC are faster by up to two orders of magnitude than CondBT. The full descriptions of the algorithms can be found in [12]. In the following we use a running example to describe the algorithms' behavior.

Backtrack search is the only algorithm that has been adapted for conditional constraint satisfaction [2,1]. The proposed adaptation, however, handles only activity constraints of inclusion. Activity constraints of exclusion are reformulated as compatibility constraints [13]. We modify the algorithm, what we call CondBT, to handle both types of activity constraints as given in the original problem representation. Figure 3 shows the search tree for finding all solutions to the example problem in previous section. The algorithm maintains an agenda

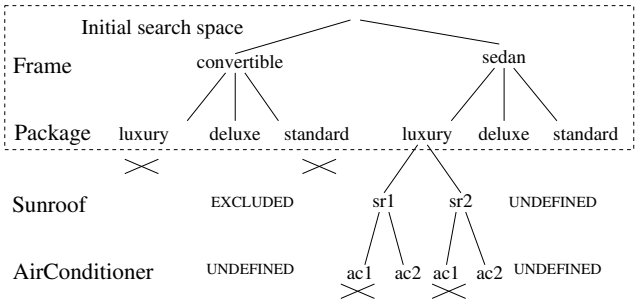


Fig. 3. CondBT search trace on the sample problem in Figure 2

of future variables, which await instantiation. Therefore, only active variables are stored in the agenda. The agenda's initial set is the set of the problem's initial variables. The algorithm's implementation uses recursion to traverse the search tree. For each active variable instantiation, the algorithm first checks the compatibility constraints and then the activity constraints. The backtrack search trace in Figure 3 has the initial search space defined by $Frame$ and $Package$.

A compatibility constraint is checked only if it involves the current variable and previously instantiated variables, called past variables. Otherwise, no compatibility constraint check is performed. If the current instantiation is consistent with the value assignment of the past variables, the constraint is satisfied. For example, when search reaches the instantiation $Package = luxury$, with past variable $Frame = convertible$, both c_1 and c_2 are satisfied.

An activity constraint is checked only if its activation condition is defined on the current variable and past variables. Otherwise, the activity constraint is discarded and no check is performed. Checking the consistency of the activation condition has two possible outcomes: either (1) current instantiation violates the activation condition, in which case the constraint does not "trigger" or has no effect on the activity status of its target variable, or (2) activation condition holds, in which case its effect has to be determined. In the first case, the constraint is trivially satisfied. In the second case, the constraint satisfiability depends on matching the constraint type (of inclusion or exclusion) with the activity status of the target variable (included, excluded, or undefined). The constraint fails if an inclusion (or exclusion) activity constraint targets an already excluded (or included) variable. The constraint holds if the activity status of the target variable is consistent with the type of activation. The constraint also holds if the target variable's activity status is undefined. In this case, the activity constraint has the effect of setting the target's status to active (or included) or to excluded.

Let us consider that the current search point is $Frame = convertible$. Note that the activity status of $AirConditioner$ and $Sunroof$ is undefined. The only constraint checked at this point is the exclusion activity constraint a_3: its activation condition is satisfied, and the exclusion of $Sunroof$ takes effect, that is, its activity status becomes excluded. The algorithm proceeds deeper in the tree by choosing the next future variable in the agenda, and instantiates $Package$ with

luxury. As shown before, compatibility constraint checking is successful and the algorithm continues with checking the activity constraints. Activity constraint a_1 is checked and it fails: the inclusion of *Sunroof* conflicts with its current activity status. *luxury* instantiation is found inconsistent, *luxury* is removed from *Package*'s domain, and the search goes sideways in the tree to the next value, *deluxe*, in the domain of the current variable. Constraint checking results in finding the first solution to the example problem: $Frame = convertible$, $Package = deluxe$.

Forward checking in the conditional context (CondFC) enforces look-ahead consistency [14] along compatibility constraints and prunes inconsistent values from the domains of future variables. When activity constraints come into play and newly activated variables are added to the set of future variables in the agenda, consistency propagation is reiterated to involve these variables as well: values which are inconsistent with the current partial solution are filtered from the newly active variables. In Figure 4 we use the same sample problem to trace CondFC execution. Let us consider that the current instantiation is $Package = luxury$. *Frame* has been assigned the value *sedan*, but the propagation of c_2 compatibility constraint did not find any inconsistent value in the domain of *Package*. When *luxury* is tried for *Package* the only applicable constraints are: a_1 includes *Sunroof* and a_2 includes *AirConditioner*. Both constraints are satisfied and the search space grows with these two variables. Forward checking prunes *ac*1 from the domain of *AirConditioner* by propagating c_1.

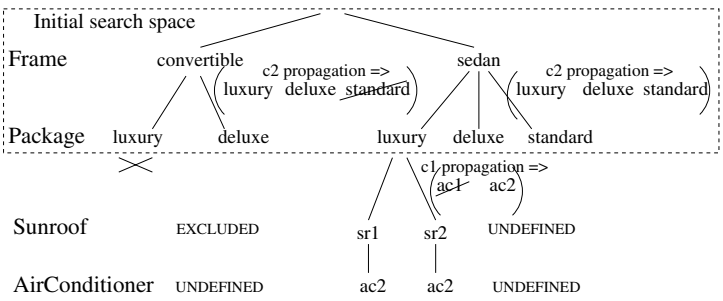


Fig. 4. CondFC search trace on the sample problem in Figure 2

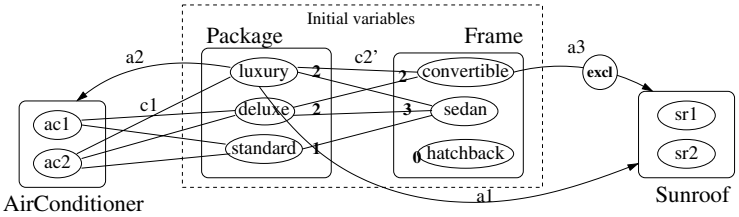
The level of consistency enforced by CondFC can be extended to arc consistency over all future variables, which are both directly and indirectly connected via compatibility constraints to the current instantiation node in the search tree. Arc consistency processing has received constant attention in the research community since Mackworth' seminal paper on consistency in constraint networks [15]. Combining backtrack search with arc consistency has produced one of the most effective solving algorithm for binary standard CSPs, maintaining arc consistency (MAC) [16,17,18,19]. CondMAC is MAC's analog for binary conditional CSP. It uses arc consistency over binary compatibility constraints[1] and a new

[1] CondMAC implementation uses AC-4 arc consistency algorithm [20].

form of local consistency over binary activity constraints, called *activation consistency*.

A modified version of the running example (Figure 5) is used to exemplify the execution of CondMAC. Prior to launching backtrack search, the agenda of initial variables is made arc consistent. The figure shows that there is only one compatibility constraint, c_2', which participates in the computation of the support counters associated with (and shown next to) the values of the initial variables. The support value of 0 for *hatchback* on c_2' indicates that *hatchback* is inconsistent with *Package*'s values and can be removed for *Frame*'s domain. Having completed this preliminary phase, in Figure 6 we show how local consistency is interleaved with backtrack search in CondMAC.

The improvement of CondMAC over CondFC consists of (1) making the newly included variables arc consistent along compatibility constraints and (2) propagating activity constraints to further remove condition values that contradict activity status of problem variables. When *convertible* is assigned to *Frame*, the other value left in its domain, *sedan*, is eliminated. Along the compatibility constraint c_2', *sedan* supports all three values at *Package*. Its elimination propagates via c_2' and support counters of *luxury*, *deluxe*, and *standard* are decremented. Consequently, *standard*'s support counter becomes 0, which shows its inconsistency with the partial solution $Frame = convertible$. The value is removed and no more arc consistency propagation takes place at this point. Following the checking of compatibility constraints, we check activity constraints whose condition variables are active. a_3 qualifies, it is satisfied, and *Sunroof* is marked as excluded from the search tree rooted at $Frame = convertible$. However, there is another activity constraint, a_1, whose condition involves future variable *Package*, and which conflicts with a_3. To maintain activation consistency over future variables, a_1's condition value, *luxury*, which is found inconsistent with *convertible*, is also removed from the domain of *Package*. All value removals due to enforcing activation consistency are propagated via arc consistency over compatibility constraints defined on future variables. In this example, removal of *luxury* propagates on c_2' and results in *convertible* losing one more



c_2' : {(*luxury convertible*)(*luxury sedan*)
(*deluxe convertible*)(*deluxe sedan*)(*standard sedan*)}

Fig. 5. Modified version of sample problem: an additional value, *hatchback*, in *Frame*, and updated c_2', which leaves *hatchback* with no support at *Package*. Values participating in compatibility constraints have associated support counters

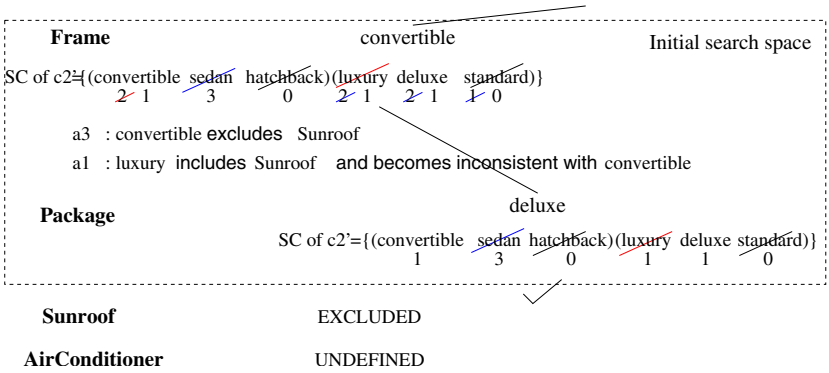


Fig. 6. CondMAC search trace on the sample problem in Example 5

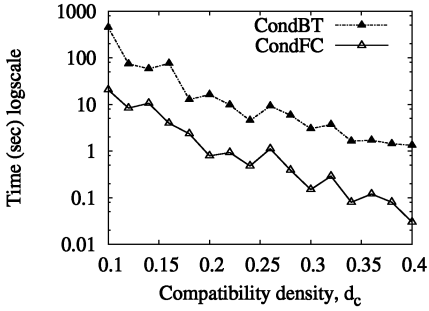
support, down to 1 at this point. With this level of consistency achieved, search continues with the instantiation of *Package* with the only value left in its domain, *deluxe*. Applicable constraints (that is, $c_2{}'$ only) are checked and satisfied, and the first solution to the problem is found.

4 Experimental Evaluation

In our experiments we use Freuder and Wallace's model of constant probability of inclusion for generating random CSPs [21,11], extended with additional parameters that collect activity information, called *activity parameters*, for a specialized class of binary CCSPs. The class restricts both compatibility and activity constraints to binary constraints. Binary activity constraints are defined on a single condition variable, with an associated unary activation constraint, and the usual target variable. As a general practice, the most prevalent experimental design for studying algorithm performance using random standard CSPs involves varying density and satisfiability parameters. In the context of random CCSPs, these parameters are the probability of generating compatibility constraints, denoted d_c, and the probability of generating allowed pairs in a compatibility constraint, denoted s_c. Specific to CCSP, we are interested in generating combinations of parameter values for those activity parameters that control the amount of activity a problem exhibits. These parameters are:

- *density of activity*, denoted d_a, is the probability of generating a non-initial variable as a target variable.
- *satisfiability of activation*, denoted s_a, is the probability of generating a value in a domain as a condition value. The number of condition values in a domain measures the satisfiability of the activation condition defined on that domain.

The three algorithms for solving CCSPs, CondBT, CondFC, and CondMAC, were tested in experiments covering diverse populations of randomly generated problems. The algorithms' implementations handle binary constraints. This restriction is imposed by the binary CondMAC algorithm and the binary random



Test suite design: problem of 10 variables, with 8 values per domain, fixed satisfiability of compatibility, $s_c = 0.25$, density of activity, $d_a = 0.3$, and satisfiability of activation, $s_a = 0.3$. Compatibility density, d_c, varies in the range $[0.1 \ldots 0.4]$ in increments of 0.02. For each of the 16 (d_c, s_c, d_a, s_a) topological classes 100 instances were generated.

Fig. 7. CondBT and CondFC execution time for variable compatibility density, d_c

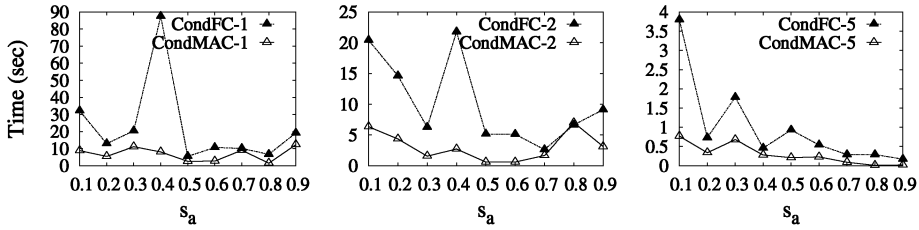


Fig. 8. CondFC and CondMAC execution time for variable satisfiability of activation, s_a and three values of density of activity: $d_a = 0.1$ (left), $d_a = 0.2$ (middle), and $d_a = 0.5$ (right). Fixed compatibility topology: $d_c = s_c = 0.2$

CCSPs used during testing. The experimental analysis has two types of studies. In the first category we measured execution time of each algorithm for finding minimum size solutions, that is, solutions that have a minimum number of variables that are assigned values. In the second category we run the algorithms to find all solutions, and we collected measures that are representative of algorithm effort: number of backtracks and compatibility checks as well as some new measurements specific to CCSP, such as number of condition checks, included and excluded variables, activity constraints that redundantly set variables' activity status, and activity constraints whose action conflict with variables' activity status.

Relative time performance of CondBT and CondFC is shown in Figure 7. We observe that CondFC runs one to two orders of magnitude faster than CondBT. Relative time performance of CondFC and CondMAC has been studied on a larger test suite that consists of 81 problem classes corresponding to all (d_a, s_a) activity parameter combinations, with d_a and s_a varying in the $[0.1 \ldots 0.9]$ range in 0.1 increments. Figure 9 shows time variation with s_a for three d_a values. Compatibility parameters are fixed: $d_c = s_c = 0.2$. There are 100 instances per problem class, each of 10 variables with domains of 10 values. The main result supported by the data is that CondMAC consistently outperforms CondFC.

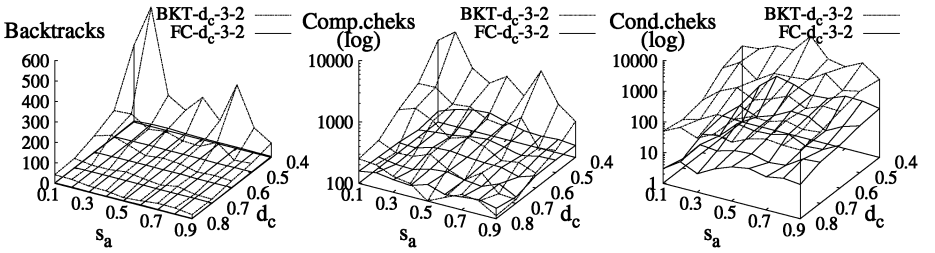


Fig. 9. Comparison between CondFC and CondFC effort measured as the number of backtracks (left), compatibility checks (middle), and condition checks (right). Variation of effort with density of compatibility, d_c, and satisfiability of activation, s_a. Fixed $s_c = 0.3$ and $d_a = 0.2$

Finally, the experimental study in Figure 9 evaluates algorithm efficiency measured by counting the number of backtracks, compatibility checks, and condition checks performed by CondBT and CondFC when searching for all solutions. The problem space considered for this study, as defined by (d_c, s_c, d_a, s_a), had 2,025 problem classes: 5 d_c values in $[0.4 \ldots 0.8]$ range, 5 s_c values in $[0.1 \ldots 0.4]$, and 81 d_a and s_a combined values in $[0.1 \ldots 0.9]$ range. Figure 9 synthesizes CondBT vs. CondFC comparison results for only 45 classes, with fixed $s_c = 0.3$ and $d_a = 0.2$. The study results show that CondFC outperforms CondBT on all measures and for all problem topologies. Similar to standard CSP solvers, all effort measures counted for CondBT and CondFC increase with problem satisfiability, s_c, and decrease with problem density, d_c. As problems exhibit more conditionality (larger d_a and s_a), CondBT and CondFC perform more condition checks, obviously, but fewer backtracks. CondFC is better than CondBT by one to two orders of magnitude on the number of backtracks and compatibility checks, and up to three orders of magnitude on the number of condition checks.

5 Reformulation

The prominence and maturity of the constraint satisfaction classical paradigm motivates our interest in reformulating the conditional CSP representation into a standard CSP. This reformulation requires the addition of a special value, called "null", to the domains of non-initial variables, and the transformation of compatibility and activity constraints into ordinary constraints [2,1,13,9]. A null-based reformulation of conditional CSPs is presented and studied in depth in [1,9]. However, this transformation is limited in the following key respects:

1. it does not transform multiple activations of the same variables,
2. it does not preserves locality of change: when the original problem changes with the addition of another activity constraint to a multiple activation cluster, which has already been reformulated, the reformulation cannot be updated locally,
3. it does not handle activity cycles.

To address these three limitations of the null-based reformulation, we have developed two alternative transformations. One removes the first two limitations; the other removes the third. Both algorithms synthesize non-binary ordinary constraints whose arity increases with the number of activity constraints in a multiple activation cluster or in an activity cycle.

Given the reformulation algorithms that overcome the limitations with multiple activations, locality of change, and activity cycles, we are interested in evaluating the relative efficiency of solving with standard methods the reformulated problem. The test suite we designed for this purpose has random conditional CSPs of 8 variables with domains of 6 values. The problems are organized in nine classes, each corresponding to a s_c value in $[0.1 \ldots 0.9]$. The other three problem generation parameters were fixed. Conditional CSPs were solved with CondMAC. Their non-binary null-based reformulations, obtained with the reformulation algorithm that handles activity cycles, were first transformed into a binary constraint representations and then were solved with the MAC algorithm for binary CSPs. The execution time results are shown in Figure 10, on a normal scale (left) and logscale (right). We observe that solving binary null-based reformulations is much slower, up to two orders of magnitude, than solving original, conditional CSPs directly. Two conclusions can be drawn from these findings. First, greater efficiency in solving conditional CSPs lies with algorithms that operate on the original representation. Second, much has to be learned about what is specific to null-based reformulations and how standard methods can more efficiently exploit these representations.

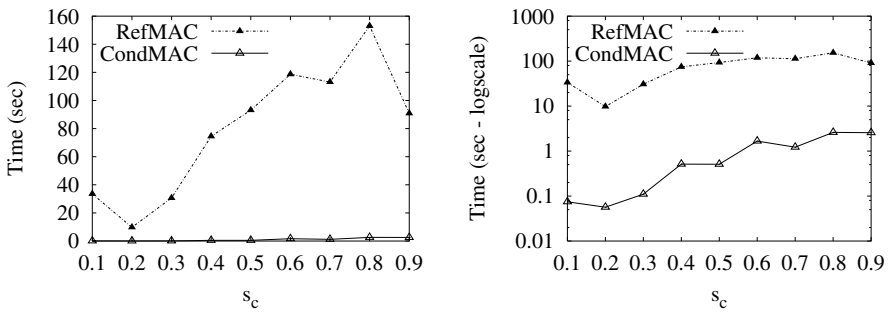


Fig. 10. Execution time of CondMAC and RefMAC - standard MAC for solving equivalent null-based, standard reformulations. Original conditional CSPs have 8 variables and 6-value domains. 100 problem instances per topological class: variable satisfiability of compatibility, s_c, in $[0.1 \ldots 0.9]$ and fixed $d_c = 0.15$, $d_a = s_a = 0.3$

6 Conclusion and Future Work

CCSPs are extensions to standard CSPs that have proved useful in representing configuration and diagnosis problems. In contrast to other CSP extensions, CCSP has not benefited from adaptations of efficient CSP solving algorithms to

improve CCSP solving. Moreover, experimental analysis of the efficiency of available CCSP solvers has been extremely limited. In this paper we presented two advanced algorithms for CCSP that adapt forward checking and maintaining arc consistency to keep track of variables' activity status and to enforce local consistency along compatibility and activity constraints. We studied their efficiency experimentally and shown an improvement of up to two orders of magnitude over plain backtrack search. An alternative approach to directly solving CCSP is to reformulate it into an equivalent standard CSP. We studied a null-based reformulation of CCSPs, addressed its limitations, and provided experimental evidence that the proposed direct methods are more efficient.

We envision two directions for our future work. Real-life configuration and diagnosis problems are formulated as non-binary CCSPs. We want to generalize the current implementations of CondFC and CondMAC to handle non-binary constraints and take advantage of efficient non-binary local consistency algorithms [19,22,23]. In [1,10] a reformulation method has been proposed that generates a set S of standard CSPs equivalent to the original CCSP. Conventional local consistency methods are then applied on intermediate problems generated along the way to producing S in order to reduce S and solve its members more efficiently with CSP solving algorithms. The method can be further improved with a hybrid approach that interleaves CSP solving, rather than just preliminary local consistency, with reformulation[2]. We are interested in a more comprehensive study of CCSP solving that will include this hybrid approach and facilitate new advances in solving and reformulating CCSPs.

Acknowledgments

This work has received support from Science Foundation Ireland under Grant 00/PI.1/C075, from the National Science Foundation under Grant No. IRI-9504316, and from Trilogy Software, Inc.

References

1. Gelle, E.: On the generation of locally consistent solution spaces. Ph.D. Thesis, Ecole Polytechnique Fédérale de Lausanne, Switzerland (1998)
2. Mittal, S., Falkenhainer, B.: Dynamic constraint satisfaction problems. In: Proceedings of the Eighth National Conference on Artificial Intelligence. (1990)
3. Sabin, M., Freuder, E.: 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 (1998)
4. Dechter, R., Dechter, A.: Belief maintenance in dynamic constraint networks. In: Proceedings of AAAI-88. (1988) 37–42
5. Bessière, C.: Arc-consistency in dynamic constraint satisfaction problems. In: Proceedings of the 9th AAAI. (1991) 221–226

[2] E. Gelle, ABB Corporate Research Ltd., Switzerland, personal communication.

6. Verfaillie, G., Schiex, T.: Solution reuse in dynamic constraint satisfaction problems. In: Proceedings of the 12th AAAI, Seattle, WA (1994) 307–312
7. Sabin, D., Sabin, M., Russell, R., Freuder, E.: A constraint-based approach to diagnosing software problems in computer networks. In Montanari, U., ed.: Principles and Practice of Constraint Programming - CP'95, Lecture Notes of Computer Science 976, Springer Verlag (1995)
8. Sabin, M., Russell, R., Miftode, I.: Using constraint technology to diagnose errors in networks managed with spectrum. In: Proceedings of the IEEE Internationl Conference on Telecommunications, Bucharest, Romania (2001)
9. Soininen, T., Gelle, E., Niemelä, I.: A fixpoint definition for dynamic constraint satisfaction. Principles and Practice of Constraint Programming, CP'99 (1999)
10. Gelle, E., Faltings, B.: Solving mixed and conditional constraint satisfaction problems. Constraints **8** (2003) 107–141
11. Wallace, R.: Random CSP Generator. Constraint Computation Center, University of New Hampshire, Durham, NH, U.S.A. (1996) http://www.cs.unh.edu/ccc/code.html.
12. Sabin, M.: Solving and Reformulation of Conditional Constraint Satisfaction Problems. PhD thesis, University of New Hampshire, Durham, NH, U.S.A. (2003)
13. Haselböck, A.: Knowledge-based Configuration and Advanced Constraint Technologies. PhD thesis, Technical University of Vienna (1993)
14. Haralick, R., Elliott, G.: Increasing tree search efficiency for constraint satisfaction problems. Artificial Intelligence **14** (1980) 263–313
15. Mackworth, A.: Consistency in networks of relations. Artificial Intelligence **8** (1977)
16. Sabin, D., Freuder, E.: Contradicting conventional wisdom in constraint satisfaction. In Borning, A., ed.: Principles and Practice of Constraint Programming, Lecture Notes in Computer Science. Volume 874. Springer (1994) (PPCP'94: Second International Workshop, Orcas Island, Seattle, USA).
17. Grant, S., Smith, B.: The phase transition behavior of maintaining arc consistency. Technical Report 92.95, School of Computing, University of Leeds (1995) (A revised and shortened version appears in Proceedings ECAI'96, pp. 175-179, 1996).
18. Bessière, C., Régin, J.C.: MAC and combined heuristics: Two reasons to forsake FC (and CBJ?) on hard problems. In Freuder, E., ed.: Principles and Practice of Constraint Programming. Lecture Notes of Computer Science. Volume 1118., Springer (1996) 61–75 (CP'96: Second International Conference, Boston, MA, USA).
19. Bessière, C., Régin, J.C.: Arc consistency for general constraint networks: preliminary results. In: Proceedings IJCAI'97, Nagoya, Japan (1997) 398–404
20. Mohr, R., Henderson, T.: Arc and path consistency revisited. Aritificial Intelligence **28** (1986) 225–233
21. Freuder, E., Wallace, R.: Partial constraint satisfaction. Artificial Intelligence **58** (1992)
22. Bessière, C., Régin, J.C.: Refining the basic constraint propagation algorithm. In: Proceedings IJCAI'01, Seattle, WA (2001) 309–315
23. Bessière, C., Meseguer, P., Freuder, E., Larrosa, J.: On forward checking for nonbinary constraint satisfaction. Artificial Intelligence **141** (2002) 205–224