Assumption-Based Pruning in Conditional CSP

Felix Geller and Michael Veksler
{felix,veksler}@il.ibm.com

IBM Research Laboratory
Haifa, Israel

Abstract. A conditional constraint satisfaction problem (CCSP) is a variant of the standard constraint satisfaction problem (CSP). CCSPs model problems where some of the variables and constraints may be conditionally inactive such that they do not participate in a solution. Recently, algorithms were introduced that use MAC at their core to solve CCSP. We extend MAC with a simple assumption-based reasoning. The resulting algorithm, Activity MAC (AMAC), is able to achieve significantly better pruning than existing methods. AMAC is shown to be more than two orders of magnitude more efficient than CondMAC on certain problem classes. Our algorithm is most naturally expressed using a variant of the CCSP representation that we refer to as Activity CSP (ACSP). ACSP introduces activity variables which explicitly control the presence of other variables in the solution. Common aspects of CCSP, such as activity clustering and disjunction, are easily captured by ACSP and contribute to improved pruning by AMAC.

1 Introduction

Standard constraint satisfaction problems (CSP) are being used to represent and solve complex combinatorial problems in many areas. Over the years, several specialized CSP variants were introduced to accommodate specific classes of problems.

1.1 Conditional CSP and historical overview

Conditional CSP (CCSP [1], CondCSP [2] or originally DCSP [3]) is a specialization in which it is possible to conditionally disable parts of the problem. This structural conditionality of CCSP is useful in several problem areas, such as configuration [3] and hardware test generation [4].

CCSP adds to CSP the notion of variable activity and inactivity. Only active variables are assigned values, while inactive variables are ignored. Over the years, several CCSP solving techniques have been proposed. The original work of Mittal and Falkenhainer [3] offers three solution

methods. The most comprehensive algorithm is a kind of backtrack-search algorithm. It uses an Assumption Based Truth Maintenance System (ATMS) [5] to perform simple deductions, cache constraint checks, and generate nogoods (i.e. learn conflicts). As any classic backtrack-search algorithm, it assigns values to variables, and backtracks once they lead to failure. The second algorithm uses backtrack-search with a limited forward-checking for activity constraints (resembling the backtrack-search used in [6]). The third algorithm reformulates CCSP to CSP, where inactive variables are assigned a special value referred to as *null*. None of the three algorithms use constraint propagation to prune the search space.
In contrast, Gelle and Faltings [2] preprocess the CCSP and then use mixed-CSP techniques combined with MAC [7]. Sabin, Freuder and Wallace [1] introduced CondMAC, a variant of MAC that takes variable activity into account. CondMAC considers only those variables that are active; variables with an activity status which is not determined yet are ignored. CondMAC consults activity constraints to find which variables should be active. Both works [2, 1] suggest that algorithms based on MAC are about two orders of magnitude faster than either pure backtrack-search or *null* reformulated CCSP.

1.2 Disjunction and clustering of activity

Two widespread phenomena in configuration problems are disjunction and clustering of activity. Mittal and Falkenhainer [3] mentioned the necessity of *"disjunction over required* [activity of] *variables"* for expression some types of domain knowledge. However, they did not implement it due to performance issues with such constraints. Soininen, Gelle and Niemelä [6] also observed that CCSP (DCSP) models are too weak to represent some configuration tasks. One of their observations is that *" It is, e.g., difficult to encode, as an activity constraint that ... either a condenser or a cooler should be included"* (where *cooler* and *condenser* are variables). In this case, a valid configuration must have either a cooler or condenser; so it is not valid to have them both deactivated. They conclude that *disjunctive activity constraints* [3] would help. It is interesting to note that most Conditional CSP papers use configuration examples for which *disjunctive activity constraints* would be natural, most notably the car configuration problem [3].
Another important phenomenon is *clustering of activity*. Clustering occurs when several variables have identical activity status in all solutions. We refer to the set of variables with identical activity conditions as an *activity cluster*. We believe that clustering is quite common in CCSPs due to the inherent modularity of the problems, where variables within the same component have the same activity rules.

1.3 The Activity MAC algorithm

This paper introduces a novel Activity MAC (AMAC) algorithm to solve CCSPs. Unlike CondMAC [1] AMAC tries to propagate constraints regardless of variable activity. This early constraint propagation allows significantly better domain pruning than CondMAC.

To enable early constraint propagation, AMAC concurrently checks several assumptions about activity of CCSP elements. The algorithm allows deduction-like information flow between different activity assumptions. This assumption-based reasoning allows AMAC to propagate constraints over the conditionality barrier, which eliminates a substantial number of backtracks at a reasonable cost.
In order to simplify the AMAC algorithm, we present a slightly modified variant of the CCSP representation. To avoid confusion with standard CCSPs, we refer to it as Activity CSP (ACSP). This representation has a new class of variables named *activity variables*. As the name suggests, activity variables control the activity of other variables. The domain of an activity variable is $\{true, false\}$ such that when it is assigned with *true* the controlled variables are all active. CCSP's activity constraints are no longer needed because regular constraints can operate directly on activity variables.
We show that ACSP and CCSP representations are computationally equivalent. Yet, ACSPs can represent important aspects of Conditional CSPs more naturally, such as *activity clustering* and *activity disjunction*. AMAC can easily use these aspects to achieve better pruning and improve performance. This paper does not deal with the nontrivial task of extracting *activity clustering* and *activity disjunction* from CCSPs. It is possible that one would prefer to model problems directly as ACSPs rather than CCSPs, because ACSPs eliminate the cost of extraction.

1.4 Minimality in CCSP and ACSP

The original work by Mittal and Falkenhainer [3] imposes a subset minimality rule over the activity of variables. According to this rule, a complete assignment is a valid solution only if removal of any of the assigned variables will violate at least one constraint. Subset minimality was discovered to be either too strong and impractical or irrelevant in some domains. According to Soininen, Gelle and Niemelä [6] some cases of configuration problems are not interested in plain subset minimality, and they propose a different minimality rule. This led us to believe that it would be useful to separate optimization goals (e.g., subset minimality) from core representation and algorithms. It should be simple and efficient to add goals such as minimality onto the basic algorithms.
In our work, we add minimality on top of the basic ACSP solver through variable and value ordering. ACSPs with added minimality have the minimality semantics of CCSPs. The separation of the ACSP structure and minimality goals makes it possible to define different goals. Possible goals range from optimization of an arbitrary cost function through minimal activity subset, to minimal assignment subset (as in the original [3]).

1.5 Benchmarks

Another interesting observation we made was that popular benchmarks of random CCSPs [2] take neither activity clustering nor activity disjunction into account. As a result, these benchmarks will generate problems

with these properties with a negligible probability. We introduce a modified benchmark that generates problems with clustering and disjunction. On this benchmark, AMAC shows a significant performance improvement over CondMAC.
The rest of our paper is organized as follows, In Section 2 we discuss the benefits of early constraint propagation, in Section 3 we introduce Activity CSP, in Section 4 we describe the workings of Activity MAC, in Section 5 we present our experimental results, and we summarize with the conclusions in Section 6.

2 Early constraint propagation

None of the published solution techniques for Conditional CSP exploit constraint propagation to the full extent. Namely, constraint checking is invoked only after the constraint variables become active. In a sense, this limits the information flow during the solution process only from already active variables to variables whose activity is still undecided. However, as the example in Figure 1 shows, there are benefits in activating conditional constraints ahead of time. The early constraint propagation allows:

- Information flow in 'reverse' direction, from potentially active variables to definitely active ones:
  In the example the domain of x is initially $\{0, \ldots, 100\}$. If active, y can affect x through constraint $x = y$ and produce domain $\{0,\ldots,4\}$ for x. Similarly, z being active implies domain $\{5,\ldots,9\}$ for x. Since either y or z has to eventually be active because of the activity disjunction, we can conclude that x's domain is at most $\{0,\ldots,9\}$. Note, this deduction is due to constraint propagation without any assignments or backtrack steps.
- Early activity conflict detection:
  We detect contradiction between the assignments $v^a_y = \mathit{true}$ and $v^a_z = \mathit{true}$ using the following argument: if we assume $v^a_y = \mathit{true}$, we obtain $\mathrm{D}(x) = \{0, \ldots, 4\}$. Similarly, $v^a_z = \mathit{true}$ implies $\mathrm{D}(x) = \{5, \ldots, 9\}$. Since the two domains are disjoint, we learn that $v^a_1 \wedge$

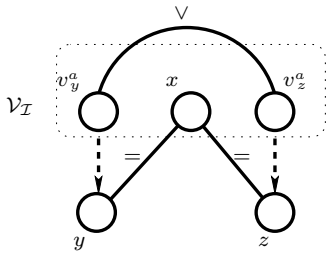


Fig. 1. A conditional CSP problem: $\mathcal{V} = \{x, y, z, v^a_y, v^a_z\}, \mathcal{V}_\mathcal{I} = \{x, v^a_y, v^a_z\}, \mathrm{D}(x) = \{0, \ldots, 100\}, \mathrm{D}(y) = \{0, \ldots, 4\}, \mathrm{D}(z) = \{5, \ldots, 9\}, \mathrm{D}(v^a_y) = \mathrm{D}(v^a_z) = \{\mathit{true}, \mathit{false}\}, \mathcal{C}_\mathcal{C} = \{x = y, x = z, v^a_y \vee v^a_z\}, \mathcal{C}_\mathcal{A} = \{v^a_y = \mathit{true} \xrightarrow{\mathit{incl}} y, v^a_y = \mathit{false} \xrightarrow{\mathit{excl}} y, v^a_z = \mathit{true} \xrightarrow{\mathit{incl}} z, v^a_z = \mathit{false} \xrightarrow{\mathit{excl}} z\}$

$v_2^a = \mathit{false}$. The detection of inconsistency between potential activity assignments lets us avoid backtrack steps.

We provide this sort of enhanced pruning in our solution method.

3 Activity CSP

The increased pruning is most easily gained when clustering and disjunction activity aspects of CCSP are stated explicitly. Hence, we introduce Activity CSP, a variant of CCSP, that allows us to capture the notion of clustering and express our solution algorithm in the natural way.

Before presenting the Activity CSP model, we recall the definition of the CCSP model [3].

3.1 CCSP

In the standard CCSP model, a problem is defined as tuple $\langle \mathcal{V}, \mathcal{V}_\mathcal{I}, \mathcal{D}, \mathcal{C}_\mathcal{C}, \mathcal{C}_\mathcal{A} \rangle$, where $\mathcal{V}$ are the variables, $\mathcal{D}$ are the domains, $\mathcal{V}_\mathcal{I} \subseteq \mathcal{V}$ is subset of initially *active* variables (that have to participate in all solutions). There are two types of constraints:

- $\mathcal{C}_\mathcal{C}$ - compatibility constraints, which specify valid combinations of the variables values.
- $\mathcal{C}_\mathcal{A}$ - activity constraints, which specify the conditions under which variables (in $\mathcal{V} \setminus \mathcal{V}_\mathcal{I}$) become active. Activity constraints are further subdivided into inclusion and exclusion constraints. An inclusion constraint $C \xrightarrow{incl} v$ states that if C holds, the variable is active (required in a solution). Alternatively, $C \xrightarrow{excl} v$ states that if C holds, the variable is not active (excluded from a solution). C is a regular compatibility constraint.

Solution *Sol* is assignment of values to $\mathcal{V}_{Sol} \subseteq \mathcal{V}$, s.t. $\mathcal{V}_\mathcal{I} \subseteq \mathcal{V}_{Sol}$, which satisfies *relevant* constraints. A compatibility constraint is relevant if all its variables are active. An activity constraint $C \xrightarrow{incl} v$ ($C \xrightarrow{excl} v$) is relevant, if C is relevant. An activity constraint $C \xrightarrow{incl} v$ ($C \xrightarrow{excl} v$) is satisfied if either C doesn't hold or v is active (inactive). A solution *Sol* is *minimal* if no assignment that is a proper subset of *Sol* is a solution.

3.2 ACSP

In ACSP, a problem is defined by $\langle \mathcal{V}, \mathcal{V}_\mathcal{I}, \mathcal{V}_\mathcal{A}, \mathcal{D}, \mathcal{C}, \mathcal{A} \rangle$, where $\mathcal{V}, \mathcal{V}_\mathcal{I}$ and $\mathcal{D}$ are the same as in CCSP, while there are two differences:

- $\mathcal{V}_\mathcal{A}$ are explicit *activity* variables with a Boolean domain. The activity variables participate in every solution: $\mathcal{V}_\mathcal{A} \subseteq \mathcal{V}_\mathcal{I}$. $\mathcal{A}$ is a mapping from $\mathcal{V} \setminus \mathcal{V}_\mathcal{I}$ to $\mathcal{V}_\mathcal{A}$. For each variable v, $\mathcal{A}(v)$ specifies its activation condition: v is active iff $\mathcal{A}(v)$ is *true*.
- We do not distinguish between compatibility and activity constraints: any constraint may refer to an activity variable.

The clustering effect is achieved when several variables share the same activation condition.

Solution *Sol* is an assignment of values to $\mathcal{V}_{Sol} \subseteq \mathcal{V}$, s.t.

1. Every active variable is assigned a value:

$$\mathcal{V}_{\mathcal{I}} \subseteq \mathcal{V}_{Sol} \text{ and } \forall v \in \mathcal{V} \setminus \mathcal{V}_{\mathcal{I}} : \ \ v \in \mathcal{V}_{Sol} \text{ iff } Sol(\mathcal{A}(v)) = true$$

2. All *relevant* constraints are satisfied.

Solution *Sol* is (locally) minimal, if no solution can be obtained from *Sol* by changing the value of some activity variables from *true* to *false* (and shrinking $\mathcal{V}_{Sol}$ correspondingly). More formally, for any assignment $S \subset Sol$ with $\mathcal{V}_S$ being the set of the assigned variables, that

1. agrees with *Sol* on the values of variables in $(\mathcal{V} \setminus \mathcal{V}_{\mathcal{A}}) \cap \mathcal{V}_S$: $\forall v \in (\mathcal{V} \setminus \mathcal{V}_{\mathcal{A}}) \cap \mathcal{V}_S \ \ S(v) = Sol(v)$ and
2. assigns less *true* values to activity variables: $\{v : v \in \mathcal{V}_{\mathcal{A}}, S(v) = true\} \subset \{v : v \in \mathcal{V}_{\mathcal{A}}, Sol(v) = true\}$,

such S can't be a solution.

To express the example problem as ACSP, we declare v^a_y, v^a_z as explicit activity variables: $\mathcal{V}_{\mathcal{A}} = \{v^a_y, v^a_z\}$ and replace the activity constraints with activation conditions: $\mathcal{A}(y) = v^a_y, \mathcal{A}(z) = v^a_z$.

3.3 Model equivalence.

We don't lose in expressive power or representation efficiency when we switch from CCSP to ACSP. Moreover, the two models are equivalent:

- From CCSP to ACSP: With each variable $v \in \mathcal{V} \setminus \mathcal{V}_{\mathcal{I}}$, we associate an activity variable v^a, set $\mathcal{A}(v) = v^a$, and replace each activity constraint $C \xrightarrow{incl} v$ ($C \xrightarrow{excl} v$) with compatibility constraint $C' : C \to v^a = true$ ($C' : C \to v^a = false$)
- From ACSP to CCSP: We replace activation condition $\mathcal{A}(v) = v^a$ with pair of activity constraints: $v^a = true \xrightarrow{incl} v$ and $v^a = false \xrightarrow{excl} v$.

Throughout the rest of the paper we use only the ACSP representation.

4 Implementing early constraint propagation

The underlying idea in the early constraint propagation is that we invoke constraints based on assumptions that some activity variables are *true*. In Section 4.1 we develop a framework that enables assumption-based reasoning about variable activity. Then we combine the assumption-based reasoning mechanism with a standard constraint propagation in section 4.2

4.1 Assumption-based reasoning

Activity set. An activity set of a constraint C is the set of activity conditions of its variables, and is denoted AS(C). We call any constraint C with AS(C) $\neq \emptyset$ a *conditional* constraint. We say that an activity set is *violated* if at least one of its variables is assigned *false*. Activity set is said to be *true* if all its variables are assigned *true*.

Shadows. The notion of the variable shadow is central to our solution approach. For a variable $v \in \mathcal{V} \setminus \mathcal{V}_{\mathcal{I}}$ and an activity set AS, shadow $v[AS]$ is a copy of variable v. The domain of the shadow consists of values that the variable may get if the activity set were *true*. For each conditional constraint C, and its variable v, we let the constraint operate on the shadow $v[\mathrm{AS}(C)]$ instead of the original variable v. Initially, $\mathrm{D}(v[AS]) = \mathrm{D}(v)$. Over the course of the algorithm execution, the shadow's domain becomes more restricted than the variable's domain, as long as the activity set is not violated: $\mathrm{D}(v[AS]) \leftarrow \mathrm{D}(v[AS]) \cap \mathrm{D}(v)$. The shadow 'sees' the changes in the variable's domain, and in addition, it is subject to the conditional constraint. If the activity set becomes true, the shadow's domain is kept identical to the variable's domain. If the activity set becomes falsified, the shadow variable is ignored for the rest of the solution process.

We demonstrate constraint parameter substitution on our example problem: Constraint $C_1(x, y)$ $x = y$ has the activity set $\mathrm{AS}(C_1) = \{v^a_y\}$. We switch constraint C_1 to shadows $x[\{v^a_y\}]$, $y[\{v^a_y\}]$. Note that $y[\{v^a_y\}]$ coincides with the variable y, because the shadow's activity set is exactly the variable's activation condition. Constraint $C_2(x, z)$ $x = z$ has the activity set $\mathrm{AS}(C_2) = \{v^a_z\}$. We substitute constraint parameters with $x[\{v^a_z\}]$, z.

Assumption-based decomposition. For each constraint, we make the assumption that its activity set is *true*. Creating and using shadow variables, for the activity set, effectively puts the constraint in a 'sandbox' in which it is run. Several conditional constraints may share the activity set and, possibly, shadows. This fact allows us to obtain assumption-based decomposition of ACSP problem: we partition the problem constraints (and variables) into groups according to their activity sets. Together with corresponding shadows, each group of constraints represents a conventional CSP problem that includes all the variables and constraints whose activity follows from the assumptions. An example of assumption-based decomposition is given in Figure 2. There is at most one shadow variable for each parameter of a conditional constraint.

The sub-problems corresponding to various activity sets can be handled independently. Upon reaching a failure in a sub-problem, we infer that the activity set is violated.

In our example, activations of C_1 and C_2, having different activity sets, are unrelated, even though initially (prior to parameter substitution) the constraints shared a variable. Reaching consistency over C_1, we get $\mathrm{D}(x[\{v^a\}]) = \{0, \ldots, 4\}$ and the activation of C_2 results in $\mathrm{D}(x[\{v^a_z\}]) = \{5, \ldots, 9\}$. Because the original problem's constraints are distributed over several sub-problems, we have to coordinate solving of the sub-problems and enable information exchange between them during the solution process. Propagating domain updates between shadows of the same variable provides the necessary communication link.

The assumption-based decomposition is performed on the conceptual level. It is intended to demonstrate that the conditional CSP may be transformed into a set of conventional CSPs. The resulting problems

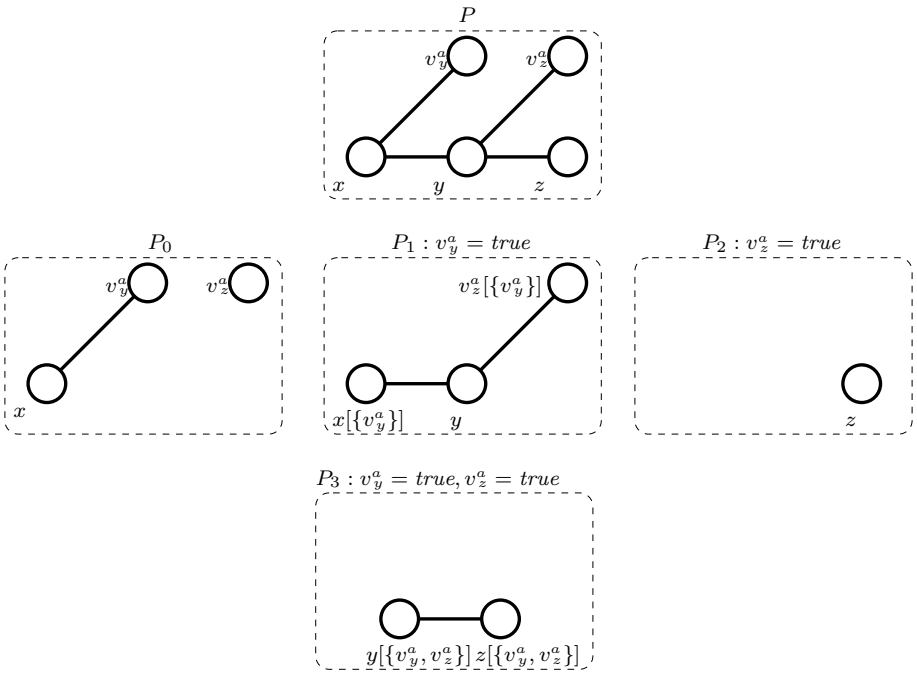


Fig. 2. Assumption-based decomposition of an ACSP: The original problem P contains five variables x, y, z, v_y^a, v_z^a. $\mathcal{V}_{\mathcal{I}} = \{x, v_y^a, v_z^a\}, \mathcal{A}(y) = v_y^a, \mathcal{A}(z) = v_z^a$. Edges denote binary constraints. The constraints' activity sets dictate partition into four sub-problems: P_0 (which is unconditional), P_1 (based on assumption $v_y^a = true$), P_2 ($v_z^a = true$) and P_3 ($v_y^a = true, v_z^a = true$).

can be solved then using standard methods coupled with the shadow synchronization mechanism. The actual implementation does not create an explicit decomposition.

Shadow synchronization rules. Let's assume we have two shadows of the same variable v: $v[AS_1]$, $v[AS_2]$. At some point in the solution process domain of $v[AS_1]$ is modified, and we want to *synchronize* shadow $v[AS_2]$ with $v[AS_1]$. We can assume that none of the variables in $AS_1 \cup AS_2$ is *false* (otherwise, at least one of the shadows is not active in the current partial assignment.) We consider the following cases:

1. If the assumption AS_1 is weaker than AS_2, the changes of $v[AS_1]$ should be reflected in $v[AS_2]$ More precisely, if, given the current partial assignment, all the variables in $AS_1 \setminus AS_2$ are *true* , then

$$\mathrm{D}(v[AS_2]) \leftarrow \mathrm{D}(v[AS_2]) \cap \mathrm{D}(v[AS_1])$$

   If an empty domain results, we conclude that $\bigwedge_{v^a \in AS_2} v^a = false$.
2. Even if the assumptions are incomparable (none is included in another), we still can benefit from comparing shadow domains. Specif-

ically, the fact

$$\mathrm{D}(v[AS_1]) \cap \mathrm{D}(v[AS_2]) = \emptyset$$

implies, that the domains contradict each other and both shadows cannot exist in a solution at the same time. Therefore, we infer that $\bigwedge_{v^a \in AS_1 \cup AS_2} v^a = \mathit{false}$.

In our example, computing $\mathrm{D}(x[\{v_y^a\}]) \cap \mathrm{D}(x[\{v_z^a\}]) = \emptyset$, leads the algorithm to conclusion that at least one of the variables v_y^a, v_z^a has to be *false*.

Essentially, the synchronization rules capture the semantics of the constraint $(\bigwedge_{v^a \in AS_1 \cup AS_2} v^a) \rightarrow (v[AS_1] = v[AS_2])$.

Propagating constraints over activity variables. We consider constraints that refer only to activity variables apart from other constraints. Taking into account the constraints' semantics allows further improvement of pruning. Specifically, we focus on two types of constraints:

1. **Activity Disjunction** ($\bigvee$): $v_1^a \vee v_2^a \vee \ldots \vee v_k^a$
2. **Activity Implication** ($\rightarrow$): $v_1^a \rightarrow v_2^a$,

where v_i^a are activity variables.

Activity Disjunction. Until now we observed how the information flows from more 'certain' shadows to less 'certain' shadows. The propagation of information in the reverse direction is also possible. Suppose we have a disjunction constraint over activity variables: $v_1^a \vee v_2^a \vee \ldots \vee v_k^a$. The constraint imposes the following relationship between variable v and its shadows $v[\{v_1^a\}], v[\{v_2^a\}], \ldots, v[\{v_k^a\}]$: $\mathrm{D}(v)$ can be no larger than $\bigcup_{i=1}^{i=k} \mathrm{D}(v[\{v_i^a\}])$. In our example the algorithm infers that $\mathrm{D}(x)$ can be no larger than $\mathrm{D}(x[\{v_y^a\}]) \cup \mathrm{D}(x[\{v_z^a\}]) = \{0, \ldots, 9\}$ and is reduced w.r.t. the initial $\mathrm{D}(x)$.

Activity Implication. An implication between activity variables induces equivalence between activity sets. We can take advantage of this equivalence to merge shadows under equivalent activity sets into single shadow, and thus reduce the overall number of variables.

4.2 Activity MAC

We integrate the above ideas into a standard propagation-based algorithm (AC-3, see [7].) We call our modified propagation method Activity MAC. This section discusses AMAC implementation details. We consider in turn the preprocessing stage, the search and constraint propagation component, and the treatment of activity disjunctions and implications. For brevity we omit most of the code and concentrate on three important functions: ReachArcConsistency, SynchronizeShadows, ComputeUnionConstraints - see Algorithm 1. Finally, we describe an optimization for the case when there are no activity disjunctions.

Preprocessing stage. The preprocessing stage computes the constraints' activity sets, builds shadows and replaces the constraint parameters with shadows. New constraints and new variables besides shadows are also created. The new constraints are referred to as *internal*

constraints, as opposed to the user constraints that are part of the input problem. One kind of internal constraints are *inference* constraints which serve for manipulation of conclusions of the form $\bigwedge_{v^a \in AS} v^a = \mathit{false}$. We associate an internal Boolean variable v_{AS} with an activity set AS for which $|AS| \geq 2$ and introduce a constraint $\bigwedge_{v^a \in AS} v^a = v_{AS}$. For any two activity sets AS_1, AS_2, s.t. $AS_1 \subseteq AS_2$, we add a redundant constraint $v_{AS_2} \rightarrow v_{AS_1}$. Another type of internal constraints, Union Constraints, is discussed later.

Search and constraint propagation. After the preprocessing stage, AMAC calls the recursive Solve procedure. The recursive Solve procedure implements a standard enumeration algorithm combined with arc consistency, with the following exceptions:

- Only original problem variables, whose existence is not ruled out, are considered for instantiation.
- We can produce a *minimal* solution, if during the variable selection step we always prefer variables whose activity is decided, and when choosing a value for an activity variable, we try *false* before *true*

Our version of ReachArcConsistency incorporates several changes as compared to the standard ReachArcConsistency:

- The constraint queue Q is a priority queue: internal constraints have precedence over user constraints. User constraints are sorted according to activity set size (unconditional constraints first.) Q is initialized to contain all the constraints (including the internal ones) prior to the first call to ReachArcConsistency. On each subsequent invocation, Q holds all the constraints incident to an instantiated variable.
- Handling constraint projection failure (lines 1.10 – 1.12): whenever projection of constraint C with $\mathrm{AS}(C) \neq \emptyset$ fails (discovers an empty set), instead of immediately returning FAILURE, we set $v_{\mathrm{AS}(C)}$ to *false* and propagate this update further.
- Shadow synchronization (lines 1.19 – 1.22): once a constraint parameter's domain is modified, the update is propagated to the shadows of the same variable.

Propagating activity disjunctions and implications.

Activity Disjunction. Let's assume we have a constraint $v_1^a \vee v_2^a \vee \ldots \vee v_k^a$. We define UnionConstraint$(x_0, x_1, \ldots x_k)$ as $x_0 = x_1 \vee \ldots \vee x_0 = x_k$. The UnionConstraint$(x_0, x_1, \ldots x_k)$ propagation is performed by the following operation: $\mathrm{D}(x_0) \leftarrow \mathrm{D}(x_0) \cap \bigcup_{i=1}^{i=k} \mathrm{D}(x_i)$. Assume that for some variable v there are k shadows : $v[AS_1], v[AS_2], \ldots, v[AS_k]$, such that:

$$\forall i, 1 \leq i \leq k : v_i^a \in AS_i \text{ and } \forall j, j \neq i,\ v_i^a \notin AS_j \tag{1}$$

We define AS'_0 to be the intersection of AS_i, $1 \leq i \leq k$. We define k activity sets AS'_i: $\forall i, 1 \leq i \leq k :\ AS'_i = AS'_0 \cup \{v_i^a\}$. For any $0 \leq i \leq k$, we create shadow $v[AS'_i]$ if it doesn't already exist.
The constraint $v_1^a \vee v_2^a \vee \ldots \vee v_k^a$ implies the constraint UnionConstraint$(v[AS'_0], v[AS'_1], \ldots, v[AS'_k])$. Union constraints are created in the function ComputeUnionConstraints which is executed during the preprocessing stage. Note, for a given variable v, the shadow

set Shadows(v) changes during the iteration over disjunction constraints. This means we may need to consider the same disjunction constraint several times. In general, we repeatedly iterate over disjunction constraints until a fixed point.

Activity Implication. An implication constraint induces the equivalence relation between activity sets. Suppose, we are given an implication constraint $v_1^a \rightarrow v_2^a$ and we have activity sets AS_1 and AS_2, such that, $v_1^a \in AS_1, v_2^a \notin AS_1, AS_2 = AS_1 \cup \{v_2^a\}$, then they are equivalent from the point of view of our algorithm: $(\bigwedge_{v^a \in AS_1} v^a) \leftrightarrow (\bigwedge_{v^a \in AS_1 \cup \{v_2^a\}} v^a)$. Thus, we can identify shadow $v[AS_1]$ with $v[AS_1 \cup \{v_2^a\}]$.

Under given implication constraints, for an activity set AS we define *minimal equivalent set* AS_{min} as a set, such that:

1. AS_{min} is equivalent to AS.
2. For any proper subset of AS_{min}, AS', $AS' \subset AS_{min}$, AS' is **not** equivalent to AS.

The minimal equivalent set of an activity set AS is unique. The canonic representation allows for fast equivalence tests. Whenever the algorithm computes an activity set, the set is replaced by its canonic equivalent.

Timely activation of the conditional constraints. If the problem does not contain activity disjunction constraints, the only benefit we gain from the conditional constraint propagation is the early detection of conflicts. A conflict implies that for some activity set AS $\bigwedge_{v^a \in AS} v^a = \mathit{false}$. In order to 'blame' the failure on specific variable, we defer constraint activation until its activation set has at most one free (unassigned) variable.

4.3 Discussion

Preprocessing. Gelle et al. [2] use preprocessing to reduce a CCSP problem to set of conventional CSPs, which are then sequentially solved. Interestingly enough, Sabin et al. [1] propose an idea of interleaving solving several CSPs that result from CCSP reformulation. This quite accurately describes one important aspect of our approach: namely, parallel solving of sub-problems resulting from the assumption-based decomposition.

Comparison with ATMS. The idea of conditioning data on assumptions has been introduced by de Kleer in his seminal paper on Assumption-based Truth Maintenance System [5]. ATMS is a general technique for solving search problems. Roughly speaking, ATMS consists of two components: the solver and the Boolean inference engine. The solution space is explored in parallel, and the solver maintains several contexts corresponding to different assumptions while the inference engine checks for assumption consistency. This resembles our notion of shadows and activity sets and their manipulation through the inference constraints.

```
1: function ReachArcConsistency( )
2:     // Q - is priority constraint queue.
3:     While Q ≠ ∅ do
4:         Select constraint C ∈ Q, s.t. AS(C) is not violated.
5:         Propagate constraint C
6:         If constraint failed then
7:             // some of the variables in Vars(C) have empty domains
8:             If AS(C) = ∅ then
9:                 return FAILURE
10:            else
11:                v_AS(C) ← false
12:                ModifiedVars ← {v_AS(C)}
13:        else
14:            // Propagate changes in C's variables to their shadows
15:            ModifiedVars ← {v|v ∈ Vars(C), D(v) modified by C propagation}.
16:            For all v ∈ ModifiedVars do
17:                // v is a shadow of some variable w
18:                // s.t. v = w[AS] for some activity set AS
19:                For all u ∈ Shadows(w), s.t. u = w[AS'], AS ≠ AS' do
20:                    // Propagate update from v to u
21:                    SynchronizeShadows(w, AS, AS')
22:                    Update ModifiedVars (if either u, v_AS' or v_AS∪AS' has been modified)
23:        // Update Q
24:        For all v ∈ ModifiedVars do
25:            Q ← Q ∪ IncidentConstraints(v)
26:        Q ← Q \ {C}
27:    return SUCCESS
28: procedure SynchronizeShadows(v, AS, AS')
29:    // Propagate changes in v[AS] to v[AS']
30:    If AS \ AS' = ∅ or all variables in AS \ AS' are true then
31:        D(v[AS']) ← D(v[AS']) ∩ D(v[AS])
32:        If D(v[AS']) = ∅ then
33:            v_AS' ← false
34:    else
35:        If D(v[AS']) ∩ D(v[AS]) = ∅ then
36:            v_AS∪AS' ← false
37: function ComputeUnionConstraints( )
38:    UnionConstraints ← ∅
39:    NewConstraints ← ∅
40:    For all v ∈ Vars(P) do
41:        repeat
42:            UnionConstraints ← UnionConstraints ∪ NewConstraints
43:            NewConstraints ← ∅
44:            For all C ∈ ActivityDisjunctionConstraints(P) do
45:                // C = v_1^a ∨ v_2^a ∨ ... ∨ v_k^a
46:                If
                       exist distinct shadows v_1, v_2, ..., v_k ∈ Shadows(v), s.t.
                       ∀i, 1 ≤ i ≤ k : v_i = v[AS_i] for some AS_i and
                                       v_i^a ∈ AS_i, but ∀j, j ≠ i, v_i^a ∉ AS_j
                   then
47:                AS'_0 ← ⋂_{i=1}^{i=k} AS_i
48:                ∀i, 1 ≤ i ≤ k : AS'_i ← AS'_0 ∪ {v_i^a}
49:                For all 0 ≤ i ≤ k do
50:                    Create v[AS'_i] (if it doesn't exist)
51:                // Create UnionConstraint
52:                C' ← UnionConstraint(v[ES'_0], v[ES'_1], ..., v[ES'_k])
53:                If C' ∉ UnionConstraints then
54:                    NewConstraints ← NewConstraints ∪ {C'}
55:        until NewConstraints = ∅
56:    return UnionConstraints
```

Algorithm 1: Activity MAC

ATMS has been applied subsequently to solving CSPs ([8], [9]), where the CSP is encoded using assumptions asserting that a certain, not necessarily Boolean, variable is assigned some value. (Note that in [3] ATMS is mainly used for nogoods recording.) However, for problems with large variable domains, the ATMS technique may be prohibitively expensive. In contrast, in AMAC, the assumptions are of very limited form; they refer only to activity variables and are known in advance. Overall, manipulation of assumptions in AMAC incurs a reasonable polynomial cost.

5 Experimental Results

Our experiments show that AMAC is considerably faster than CondMAC on hard problems. It seems that most of the benefit comes from AMAC's better handling of clustering.

5.1 The solvers

We used the Generation Core [4] CSP solver to perform our experiments. The solver implements AC-3 [7] with random selection of both variables and values. The solver tries to minimize the effects of the heavy tailed phenomena [10], when few bad initial selections cause massive backtracks. As a countermeasure the solver performs restarts after consecutive $1, 2, 4, \ldots, 256$ seconds and then gives up after another 512 seconds (1023 total). We implemented both CondMAC [1] and AMAC solving methods. We believe that the merits of more sophisticated algorithms, like MAC-DBT (Dynamic Backtracking [11]), are orthogonal to the gains AMAC has over CondMAC.

5.2 Benchmarking technique and definitions

Unlike in other CSP domains, there are no publicly known CCSP benchmark problems. The only published benchmarks are random CCSP generators. We use a random CCSP generator similar to the one used in [1, 2] with several modifications. Because we want to give meaningful results for the CCSP community, we define our tests in terms of CCSP.
For simplicity, we test only binary constraints. We define the density of compatibility $d_c = \#constraints \cdot 2/(|\mathcal{V}|^2 - \mathcal{V})$. If $d_c = 1$, there is a constraint between every pair of regular variables. We define satisfiability of compatibility s_c as the ratio of number of valid tuples to the product of domain sizes. If $s_c = 1$, every constraint permits all possible value pairs, such that the CSP is trivially satisfiable. For activity constraints of the type $v_i \in A_i \xrightarrow{incl} v_j$, satisfiability of activation $s_a = |A_i|/|D_i|$. In these benchmarks there is exactly one activity constraint for each $v_i \in \mathcal{V} \setminus \mathcal{V}_\mathcal{I}$. And density of activity is defined $d_a = |\mathcal{V} \setminus \mathcal{V}_\mathcal{I}| \,/\, |\mathcal{V}|$.
We define N_c as the number of variables per conditional cluster. $N_c = 1$ means that all conditional variables (variables in $\mathcal{V} \setminus \mathcal{V}_\mathcal{I}$) have independent activity constraints. On the other extreme, $N_c = |\mathcal{V} \setminus \mathcal{V}_\mathcal{I}|$ means that all conditional variables have exactly the same activity constraints. We

define $N_\vee$ as the total number of binary disjunctions between conditional clusters. A binary disjunction between v_1 and v_2 is a constraint that insures that at least one of v_1 or v_2 are active. This is similar to $true \xrightarrow{incl} 1\{v_1, v_2\}2$ in the language of [6].

5.3 Comparison with other CCSP benchmarks

Let's consider the benchmarks presented in [1]. The benchmark has $N = 10$ variables, $|D_i| = 10$ values in each domain, $1 - 3$ conditional variables, $d_a \in \{0.1, 0.2, 0.3\}$, fixed $d_c = s_c = 0.2$ and $s_a \in [0.1, 0.9]$. The times for both CondMAC and AMAC were below 2 milliseconds. This is almost four orders of magnitude faster than previously reported results. We can only speculate why our implementation of CondMAC is so much faster. The only major algorithmic difference is that we use AC-3 while [1] uses AC-4. Differences in random problem generators may also contribute to the big difference. Other contributing factors could be newer hardware and software engineering differences. Both manual and automatic inspection of the results assured the correctness of our results.

5.4 AMAC vs. CondMAC

In all our tests, we averaged 5000 runs per sample. All our results are for the classic subset minimality [3] optimization goal.

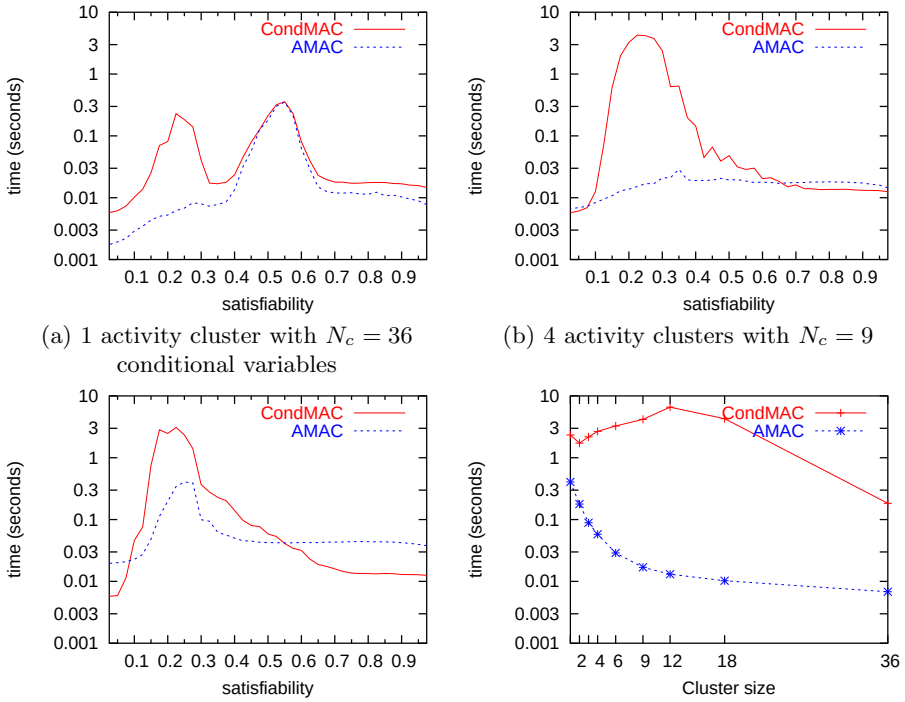
For our clustering benchmarks we chose the following characteristics: $N = |\mathcal{V}| = 48$, $\mathcal{V}_\mathcal{I} = 12$, $s_a = 0.75$. Note that our 36 conditional variables decompose into a reasonable amount of 9 possible N_c values. In Figure 3 we tested $s_c \in [0.025, 0.975]$ using 0.025 intervals. Figure 3(d) shows us that AMAC is more than two orders of magnitude faster than CondMAC for nontrivial clustering.

As expected, AMAC is slower than CondMAC for trivial problems (up to 3 times, Figure 3(d)). In these cases, CondMAC finds a solution/failure without backtracking, while AMAC needlessly explores assumptions.

For disjunction, the difference between AMAC and CondMAC is more than five orders of magnitude. This gap makes it impossible to get comprehensible results. For example with $s_a = 0.3$, $s_c = 0.2$, $d_c = 0.15$, $|\mathcal{V}_\mathcal{I}| = 12$, $|\mathcal{V}| = 48$, $|D_i| = 12$ and two clusters of $N_c = 18$. All CondMAC instances finished within a time limit of 2046 seconds, while the average of AMAC was 2/100 of a second.

6 Conclusions

We have shown that combining assumption-based reasoning with constraint propagation can significantly enhance pruning for Conditional CSP. The increased pruning is supported by problem characteristics like activity clustering and presence of activity disjunction constraints. These problem's aspects are naturally captured by Activity CSP, a variant of CCSP.



(a) 1 activity cluster with $N_c = 36$ conditional variables

(b) 4 activity clusters with $N_c = 9$

(c) 36 independent conditional variables ($N_c = 1$)

(d) $S_c = 0.25$ with different cluster sizes (N_c)

Fig. 3. Clustering: Density of compatibility is $d_c = 0.15$. Satisfiability of activity $s_a = 0.75$. Different values of $s_c \in [0.025, 0.975]$ are displayed.

A possible extension to the ACSP model can be to attach activation conditions to activity variables, such that an activity variable may depend on another activity variable. This enhancement may enable modeling of problems with hierarchical/nested structure.
Another possible extension may support negative activation conditions, this may permit simpler modeling of mutually exclusive alternatives.

7 Acknowledgments

The authors wish to thank Mati Joshua for his substantial contribution. We also wish to thank Eyal Bin, Yehuda Naveh and Gil Shurek for in depth discussions and comments throughout the writing of this paper.

References

1. Sabin, M., Freuder, E.C., Wallace, R.J.: Greater efficiency for conditional constraint satisfaction. In Rossi, F., ed.: CP. Volume 2833 of Lecture Notes in Computer Science., Springer (2003) 649–663

2. Gelle, E., Faltings, B.: Solving mixed and conditional constraint satisfaction problems. Constraints **8** (2003) 107–141
3. Mittal, S., Falkenhainer, B.: Dynamic constraint satisfaction problems. In: Proc. of AAAI-90, Boston, MA (1990) 25–32
4. Bin, E., Emek, R., Shurek, G., Ziv, A.: Using a constraint satisfaction formulation and solution techniques for random test program generation. IBM Systems Journal **41** (2002) 386–402
5. de Kleer, J.: An assumption-based tms. Artif. Intell. **28** (1986) 127–162
6. Soininen, T., Gelle, E., Niemelä, I.: A fixpoint definition of dynamic constraint satisfaction. In Jaffar, J., ed.: CP. Volume 1713 of Lecture Notes in Computer Science., Springer (1999) 419–433
7. Mackworth, A.K.: Consistency in networks of relations. Artificial Intelligence **8** (1977) 99–118
8. de Kleer, J.: A comparison of atms and csp techniques. In: IJCAI. (1989) 290–296
9. McAllester, D.A.: Truth maintenance. In: AAAI. (1990) 1109–1116
10. Gomes, C.P., Selman, B., Crato, N., Kautz, H.A.: Heavy-tailed phenomena in satisfiability and constraint satisfaction problems. Journal of Automated Reasoning **24** (2000) 67–100
11. Jussien, N., Debruyne, R., Boizumault, P.: Maintaining arc-consistency within dynamic backtracking. In: Principles and Practice of Constraint Programming. (2000) 249–261