Using Constraints with Memory to Implement Variable Elimination

Martí Sánchez[1] and **Pedro Meseguer**[2] and **Javier Larrosa**[3]

Abstract. Adaptive consistency is a solving algorithm for constraint networks. Its basic step is variable elimination: it takes a network as input, and produces an equivalent network with one less variable and one new constraint (the join of the variable bucket). This process is iterated until every variable is eliminated, and then all solutions can be computed without backtracking. A direct, naive implementation of variable elimination may use more space than needed, which renders the algorithm inapplicable in many cases. We present a more sophisticated implementation, based on the projection with memory of constraints. When a variable is projected out from a constraint, we keep the supports which that variable gave to the remaining tuples. Using this data structure, we compute a set of new factorized constraints, equivalent to the new constraint computed as the join of the variable bucket, but using less space for a wide range of problems. We provide experimental evidence of the benefits of our approach.

1 Introduction

Adaptive consistency (ADC) [3] is a solving algorithm for constraint networks. It performs complete inference, solving the network without searching its state space. Its basic step is variable elimination: it takes as input a constraint network and produces an equivalent network with one less variable and one new constraint, computed as the join of the variable bucket (that is, all constraints that mention that variable). This step is iterated until all variables are eliminated, and then all solutions can be found without backtracking. Its basic operations are projection and join of constraints (see Section 2). Its temporal and spatial complexities are exponential in $w*$, the width of the induced constraint graph. This approach has been shown useful in several contexts [1, 6, 4, 5].

Since solving constraint networks is an NP-complete problem, finding an algorithm with exponential temporal complexity is not a surprise. However, the exponential complexity in space renders it difficult for general use. In addition, a naive implementation may require more space than strictly needed. To overcome this, in this paper we present a new form to implement variable elimination, which allow us to save some space with respect to the naive implementation. Obviously, we cannot circumvent the exponential complexity in space in the worst case, but we believe that our approach could provide benefits in many problems not falling in such category.

To overcome this issue, we have pursued the following idea. When eliminating a variable x_i, instead of computing a single new constraint g_i, which could be very large, we compute a set of smaller constraints which are equivalent –as set– to g_i. Only the forbidden tuples (nogoods) of these constraints are stored. These constraints are factorized, in the sense that if a tuple is forbidden, it appears in one constraint only. In many cases, these constraints require less storage than g_i.

This is done using the projection with memory of constraints. When a variable is projected out from a constraint, we keep the supports which that variable gave to the remaining tuples. When a variable is going to be eliminated, the constraints in the variable bucket are projected with memory. Looking the combinations of supports with empty intersections, we compute the nogoods that form the new factorized constraints. This process is detailed in Section 3.

The paper is organized as follows. In Section 2, we define the basic concepts and recall the algorithm of adaptive consistency. In Section 3, we present the conceptual elements required for our approach. We describe its practical usage in Section 4, while the experimental results appear in Section 5. Finally, Section 6 contains some conclusions of this work.

2 Preliminaries

A constraint network is defined by a triple (X, D, C), where $X = \{x_1, \ldots, x_n\}$ is a set of n variables, $D = \{D_1, \ldots, D_n\}$ is a collection of finite domains such that each variable x_i takes values in D_i, and C is a set of constraints. A constraint $c \in C$ is a relation defined on the subset of variables $var(c) \subseteq X$ (called its *scope*), declaring the allowed value combinations. If $var(c) = \{x_{i_1}, \ldots, x_{i_k}\}$, the constraint c is a subset of the cartesian product $D_{i_1} \times \ldots \times D_{i_k}$. In other words, a constraint c is a set of tuples over $var(c)$.

A tuple t is a set of values corresponding to a set of variables $var(t) \subseteq X$. The pair (x_i, a) means that variable x_i takes value $a \in D_i$. The projection of tuple t over the set $A \subset var(t)$, noted $t[A]$, is the subtuple which contains only the assignments to variables in A. The projection over the empty set $t[\emptyset]$ produces the empty tuple λ. Given two tuples t and t' with the same assignment to the common variables, their join is a new tuple $t \cdot t'$ which contains the assignments of both t and t'. Clearly, $var(t \cdot t') = var(t) \cup var(t')$. Solving a constraint network means finding a tuple t such that $var(t) = X$ and for all $c \in C$, $t[var(c)] \in c$.

We define two operations on constraints:

- Projecting out a variable. Given a constraint c such that $x_i \in var(c)$, projecting out variable x_i from c, noted $c \Downarrow_{x_i}$, is a new constraint with scope $var(c) - \{x_i\}$ defined as $\{t | \exists a \in D_i, t \cdot (x, a) \in c\}$.
- Join. Given two relations c_j and c_k, its join $c_j \bowtie c_k$ is a new relation with scope $var(c_j) \cup var(c_k)$ defined as $\{t = t' \cdot t'' | \; t' \in c_j \wedge t'' \in c_k\}$.

[1] IIIA-CSIC, Campus UAB, 08193 Bellaterra, Spain
[2] IIIA-CSIC, Campus UAB, 08193 Bellaterra, Spain
[3] Dep. LSI, UPC, Jordi Girona 1-3, 08034 Barcelona, Spain

```
function ADC($X, D, C$) return tuple
1  for each $i = n..1$ do
2      $B_i := \{c \in C | \, x_i \in var(c)\}$;
3      $g_i := (\bowtie_{c \in B_i} c) \Downarrow_{x_i}$;
4      if $g_i = \emptyset$ then return NULL;
5      $C := C \cup \{g_i\} - B_i$;
6  $t = \lambda$
7  for each $i = 1..n$ do
8      select $a \in D_i$ such that $t \cdot (x_i, a) \in (\bowtie_{c \in B_i} c)$;
9      $t := t \cdot (x_i, a)$;
10 return $t$
```

Figure 1. ADC algorithm. It takes as input a constraint network, (X, D, C), and returns tuple t, a solution.

Adaptive consistency [3] is an algorithm to solve constraint networks. It is an instance of a more general algorithm called Bucket Elimination [2]. ADC performs complete inference, replacing subsets of constraints by new inferred constraints until the problem is trivially solved. The algorithm of adaptive consistency appears in Figure 1. Given a static variable ordering, variables are processed from last to first. When processing variable x_i, all constrains in the problem mentioning x_i form the set B_i (called its bucket). All the constraints in B_i are joined and x_i is projected out from the result, obtaining a new constraint g_i that does not mention x_i. If g_i is empty, the network has no solution and ADC stops, returning NULL. Otherwise, x_i and B_i are substituted by g_i, obtaining an equivalent problem with one less variable. Because of this, this step is called variable elimination. The process iterates, eliminating one variable in turn, until no variables remain. If an empty constraint is generated, the problem has no solution. Otherwise, a solution is built in a backtrack-free manner, instantiating variables from first to last and getting values consistent with the new generated constraints. This last step can easily be modified to compute all solutions of the network.

Usually, it is assumed that ADC uses constraints in positive form, that is, storing permitted tuples. However, when constraints are very loose this is not the best option. A better idea is to use constraints in negative form, that is, storing forbidden tuples. A negative constraint $\bar{c}$ is a constraint stored in negative form (if $t \in \bar{c}$, t is forbiden by the constraint). A positive constraint can be negated $\neg c = \bar{c} = \{t \notin c\}$. Double negation produces the original set of tuples.

ADC can be redefined to work exclusively with negative information. Join and projection need to be extended to negative constraints:

$$\bar{c}_j \bowtie \bar{c}_k = \{t = t' \cdot t'' | \, t' \in \bar{c}_j \vee t'' \in \bar{c}_k\}$$

$$\bar{c}_j \Downarrow_{x_i} = \{t | \forall a \in D_i, t \cdot (x_i, a) \in \bar{c}_j\}$$

This new version, called ADC-, is just like standard ADC but the set of original constraints are assumed to be negative, and the negative form of join and projection should always be used.

Although both ADC and ADC- have the same worst case time and space complexity, the question is which one is more efficient in practice. We have observed that it depends on the problem to be solved. Problems with very loose constraints are better solved with ADC- because ADC generates large positive constraints, and the opposite occurs for problems with not very loose constraints. This can be observed in Figure 3 (left), which shows the relative performance of ADC and ADC- with respect to the number of tuples generated, for the random binary class $\langle n = 7, m = 5, p_1 = 1\rangle$ and varying tightness.

3 Constraints with Memory

We aim at a new implementation of variable elimination, which could work in practice more efficiently than both ADC and ADC-. To do this, we propose two new operations on constraints.

The first operation is *projecting out a variable with memory*. Given a positive constraint c with $x_i \in var(c)$, projecting out x_i from c with memory, $c \Downarrow^m_{x_i}$, produces a set of pairs $\{(S_k, T_k)\}$ where S_k is a non-empty subset of D_i and T_k is a non-empty subset of tuples of $c \Downarrow_{x_i}$, defined as follows. Tuple $t \in T_k$ iff $\forall a \in S_k$, $t \cdot (x_i, a) \in c$. If $t \in T_k$, S_k is said to be its support set. Observe that sets T_k are mutually disjoint, although this is not necessarily the case for sets S_k. Projecting out a variable with memory differs from standard projection in that it *remembers* for each tuple those supporting values in D_i. Besides, tuples with the same support set are grouped together. In the following, c^i denotes the result of projecting out x_i from constraint c with memory. We say that c^i is a constraint with memory about x_i. Observe that c^i always denotes a positive constraint with memory. Abusing notation, we will write $t \in c^i = \{(S_k, T_k)\}$ meaning $\exists k$ such that $t \in T_k$.

The join operation can be extended to constraints with memory. Given two constraints with memory $c^i_j = \{(S_p, T_p)\}$ and $c^i_k = \{(S'_p, T'_p)\}$, their *join* $c^i_j \bowtie c^i_k$ is a new constraint with memory $c^i_l = \{(S''_p, T''_p)\}$. A tuple $t'' \in T''_s$ iff $t'' = t \cdot t'$, $t \in T_q, t' \in T'_r$ such that $S_q \cap S'_r = S''_s \neq \emptyset$.

The second operation is *extracting nogoods from constraints with memory*, that can be applied to one or two constraints. Given a constraint with memory c^i, its set of *original nogoods*, $\oslash c^i$, is a set of tuples t with scope $var(c^i)$ defined as $\{t | \, t \notin c^i\}$, which is equivalent to $\bar{c} \Downarrow_{x_i}$. Given two constraints with memory $c^i_j = \{(S_p, T_p)\}$ and $c^i_k = \{(S'_p, T'_p)\}$, their *inferred nogoods*, $c^i_j \oslash c^i_k$, is a set of tuples with scope $var(c^i_j) \cup var(c^i_k)$ defined as,

$$\{t'' = t \cdot t' | \, t \in T_q, t' \in T'_r, S_q \cap S'_r = \emptyset\}$$

In words, the inferred nogoods are those tuples that were permitted by either c^i_j and c^i_k but are not permitted by their join. Observe that the operation $\oslash$ produces a negative constraint.

Example 1 Let us consider the 3-queens problem with the usual formulation (variables are rows and values are columns denoted as a, b, c). It has three constraints with the following permitted tuples,

c_1

x_1	x_2
a	c
c	a

c_2

x_1	x_3
a	b
b	a
b	c
c	b

c_3

x_2	x_3
a	c
c	a

Projecting out x_1 with memory we get,

c^1_1

$S : x_1$	$T : x_2$
$\{a\}$	$\{c\}$
$\{c\}$	$\{a\}$

c^1_2

$S : x_1$	$T : x_3$
$\{a, c\}$	$\{b\}$
$\{b\}$	$\{a, c\}$

Its join $c^1_1 \bowtie c^1_2$ is empty. Original and inferred nogoods are,

$\oslash c^1_1$

x_2
b

$\oslash c^1_2$

x_2

$c^1_1 \oslash c^1_2$

x_2	x_3
c	a
c	c
a	a
a	c

what means that value b is forbidden for x_2 because c_1, no value is forbidden for x_2 because c_2, and the pairs ca, cc, aa, ac are forbidden for x_2, x_3 because of the combined action of c_1 and c_2. □

4 Adaptive Consistency Revisited

We present ADC-F, a new implementation of adaptive consistency. The idea is to work with negative constraints and constraints with memory simultaneously. The bucket of a variable is formed by negative constraints only, and the result of variable elimination is a new set of negative constraints. When processing a bucket, some of these constraints are turned positive (when projecting out the variable to eliminate), producing new negative constraints using $\oslash$ operation.

The pseudocode for ADC-F appears in Figure 2. Given a constraint network (X, D, C), where C is a set of negative constraints, ADC-F works as follows. As for ADC, we assume a static variable ordering given by the variable indexes. First, there is a loop to extract original nogoods from initial constraints (linea 1-4). Variables are processed from last to first (line 1). For each variable x_i, we extract in F_i the original nogoods of constraints mentioning x_i when projecting out x_i (lines 2, 3), and those constraints are removed from C (line 4). After processing all variables, the set C is set to the original C plus the original nogoods (line 5). Second, there is other loop to extract inferred nogoods from those constrains composing the bucket of each variable. Variables are processed again from last to first (line 6). For each variable x_i, its bucket B_i is computed as the set of all (negative) constraints having x_i in their scope (line 7). Next, for every constraint in B_i, its elimination with memory is computed and stored in M_i (line 8), and the set N_i is initialized (line 9). Now, the algorithm enters a loop (lines 10-14). In each iteration two constraints from M_i are selected (line 11). Their inferred nogoods are computed; if they forbid everything in the corresponding scope, there is no solution and NULL is returned (line 12). Otherwise, they are added to N_i (line 13). Their join is computed and added to M_i (line 14). The process is repeated until M_i contains a singleton. At this point the set N_i contains all the relevant negative information, without mentioning variable x_i. Thus, N_i can replace B_i in C (line 15). Third, there is the initialization of t (line 16) and another loop (lines 17-19) to reconstruct a solution, which is finally returned (line 20).

```
function ADC-F($X, D, C$) return tuple
1   for each $i = n..1$ do
2       $H_i := \{\bar{c}_l | \bar{c}_l \in C, x_i \in var(\bar{c}_l)\}$;
3       $F_i := \{\oslash c_l^i = \bar{c}_l \Downarrow_{x_i} | \bar{c}_l \in H_i\}$;
4       $C := C - H_i$;
5   $C := (\cup_{i=1}^n H_i) \cup (\cup_{i=1}^n F_i)$;
6   for each $i = n..1$ do
7       $B_i := \{\bar{c}_l | \bar{c}_l \in C, x_i \in var(\bar{c}_l)\}$;
8       $M_i := \{c_l^i = \bar{c}_l \Downarrow_{x_i}^m | \bar{c}_l \in B_i\}$;
9       $N_i := \emptyset$;
10      while $|M_i| > 1$ do
11          $(c_j^i, c_k^i)$:=popTwo($M_i$);
12          if $\neg(c_j^i \oslash c_k^i) = \emptyset$ then return NULL;
13          $N_i := N_i \cup \{c_j^i \oslash c_k^i\}$;
14          $M_i := M_i \cup \{c_j^i \bowtie c_k^i\}$;
15      $C := C \cup N_i - B_i$;
16  $t = \lambda$
17  for each $i = 1..n$ do
18      select $a \in D_i$ such that $t \cdot (x_i, a) \in (\bowtie_{c \in B_i} c)$;
19      $t := t \cdot (x_i, a)$;
20  return $t$
```

Figure 2. ADC-F implementation of adaptive consistency. C is the set of constraints in negative form.

Example 2 Let us see how the process works in the 4-queens problem, with the usual formulation: rows are variables and columns are values a, b, c, d. The initial constraints of the problem are:

$c_{12}(x_1, x_2)$
$\{ac, ad, bd, ca, da, db\}$

$c_{13}(x_1, x_3)$
$\{ab, ad, ba, bc, cb, cd, da, dc\}$

$c_{14}(x_1, x_4)$
$\{ab, ac, ba, bc, cb, cd, db, dc\}$

$c_{23}(x_2, x_3)$
$\{ac, ad, bd, ca, da, db\}$

$c_{24}(x_2, x_4)$
$\{ab, ad, ba, bc, cb, cd, da, dc\}$

$c_{34}(x_3, x_4)$
$\{ac, ad, bd, ca, da, db\}$

None of them generates original nogoods ($\oslash c_{ij}^1 = \emptyset$). Elimination of x_1:

c_{12}^1

$S : x_1$	$T : x_2$
$\{a\}$	$\{c\}$
$\{a, b\}$	$\{d\}$
$\{c, d\}$	$\{a\}$
$\{d\}$	$\{b\}$

c_{13}^1

$S : x_1$	$T : x_3$
$\{a, c\}$	$\{b, d\}$
$\{b, d\}$	$\{a, c\}$

c_{14}^1

$S : x_1$	$T : x_4$
$\{a, b, c\}$	$\{c\}$
$\{a, c, d\}$	$\{b\}$
$\{b\}$	$\{a\}$
$\{c\}$	$\{d\}$

$h_1 = c_{12}^1 \bowtie c_{13}^1$

$S : x_1$	$T : x_2 x_3$
$\{a\}$	$\{cb, cd, db, dd\}$
$\{b\}$	$\{da, dc\}$
$\{c\}$	$\{ab, ad\}$
$\{d\}$	$\{aa, ac, ba, bc\}$

$\bar{n}_1(x_2, x_3) = c_{12}^1 \oslash c_{13}^1$
$\{bb, bd, ca, cc\}$

$\bar{n}_2(x_2, x_3, x_4) = h_1 \oslash c_{14}^1$
$\{aaa, aad, abc, aca, acd, adc, baa, bad, bca, bcd, cba, cbd, cda, cdd, dab, dba, dbd, dcb, dda, ddd\}$

Bucket $B_2 = \{\bar{c}_{23}, \bar{c}_{24}, \bar{n}_1(x_2, x_3), \bar{n}_2(x_2, x_3, x_4)\}$. Two constraints have the same scope, $\bar{c}_{23}$ and $\bar{n}_1(x_2, x_3)$, so we perform its join as their union, producing $\bar{g}_1(x_2, x_3)$. Now, $B_2 = \{\bar{g}_1(x_2, x_3), \bar{c}_{24}, \bar{n}_2(x_2, x_3, x_4)\}$. Elimination of x_2:

$g_1^2(x_2, x_3)$

$S : x_2$	$T : x_3$
$\{a\}$	$\{c, d\}$
$\{d\}$	$\{a, b\}$

c_{24}^2

$S : x_2$	$T : x_4$
$\{a, c\}$	$\{b, d\}$
$\{b, d\}$	$\{a, c\}$

$n_2^2(x_2, x_3, x_4)$

$S : x_2$	$T : x_3 x_4$
$\{a, b\}$	$\{ba, da, bd, dd\}$
$\{a, b, c\}$	$\{ab, cb\}$
$\{b, c, d\}$	$\{bc, dc\}$
$\{c, d\}$	$\{aa, ca, ad, cd\}$

$h_2 = g_1^2(x_2, x_3) \bowtie c_{24}^2$

$S : x_2$	$T : x_3 x_4$
$\{a\}$	$\{cb, cd, db, dd\}$
$\{d\}$	$\{aa, ac, ba, bc\}$

$\bar{n}_3(x_3, x_4) = g_1^2(x_2, x_3) \oslash c_{24}^2$
$\{ab, ad, bb, bd, ca, cc, da, dc\}$

$\bar{n}_4(x_3, x_4) = h_2 \oslash n_2^2(x_2, x_3, x_4)$
$\{ba, cd\}$

Bucket $B_3 = \{\bar{c}_{34}, \bar{n}_3(x_3, x_4), \bar{n}_4(x_3, x_4)\}$. The three constraints have the same scope, so we perform its join as their union, producing $\bar{g}_2(x_3, x_4)$. Now, $B_3 = \{\bar{g}_2(x_3, x_4)\}$. Elimination of x_3:

$g_2^3(x_3, x_4)$

$S : x_3$	$T : x_4$
$\{a\}$	$\{c\}$
$\{d\}$	$\{b\}$

The x_4 variable is trivially eliminated. The problem has solution which can be obtained assigning variables in reverse order. □

We are going to show that ADC-F is equivalent to ADC. First, we prove a technical lemma to show that joining two constraints and projecting out variable x with memory is equivalent to projecting out x from each constraint with memory and subsequently performing the join.

Lemma 1 $(c \bowtie c') \Downarrow_x^m = c \Downarrow_x^m \bowtie c' \Downarrow_x^m$.

Proof. Notation: $c \Downarrow_x^m = \{(S_i, T_i)\}$, $\quad c' \Downarrow_x^m = \{(S'_j, T'_j)\}$,
$c \Downarrow_x^m \bowtie c' \Downarrow_x^m = \{(S''_k, T''_k)\}$, $\qquad (c \bowtie c') \Downarrow_x^m = \{\mathcal{S}_l, \mathcal{T}_l\}$
$\Rightarrow$) If $\tau \in \mathcal{T}_l$ and $a \in \mathcal{S}_l$, then $\tau \cdot (x, a) \in c \bowtie c'$. We can write $\tau = t \cdot t'$, where $t = \tau[var(c) - \{x\}]$ and $t' = \tau[var(c') - \{x\}]$. Then, $t \in T_i$, $t' \in T'_j$, and $a \in S_i \cap S'_j = S''_k$. So, $\tau = t \cdot t' \in T''_k$ and $a \in S''_k$.
$\Leftarrow$) If $t'' \in T''_k$ and $a \in S''_k$, then $t'' = t \cdot t'$, with $t \in T_i$, $t' \in T'_j$, $a \in S_i \cap S'_j$. Then, $t'' \cdot (x, a) \in c \bowtie c'$, so $t'' \in \mathcal{T}_l$ and $a \in \mathcal{S}_l$. $\square$

Theorem 1 *The set of constraints $F_i \cup N_i$ is equivalent to (forbids the same tuples as) $g_i = (\bowtie_{c \in B_i} c) \Downarrow_{x_i}$.*

Proof. In this proof, B_i and g_i refer to the bucket and the join projecting out variable x_i of the ADC algorithm. By construction, the set H_i (line 2 of Figure 2) is included in the bucket B_i. Then, F_i contains those forbidden tuples by constraints in H_i, for all values of x_i. These tuples are obviously forbidden by the join g_i. Therefore, we rest to consider the set N_i and g_i.

Applying lemma 1 we see that the unique constraint that remains in M_i is g_i, with memory about x_i. At each point that two constraints c^i_j and c^i_k are joined, we extract in $c^i_j \oslash c^i_k$ those tuples which cannot belong to the join because there is no common support in the eliminated variable. Therefore, the set F_i contains the tuples that are forbidden by the initial constraints, and the set N_i contains the tuples that are discovered forbidden in the join process. So its union forbids the same tuples as g_i. $\square$

Theorem 2 *If constraints $\{c_1, \ldots, c_p\}$ are processed ordered by increasing arity, the set $F_i \cup N_i$ is factorized: a tuple forbidden by a constraint is not forbidden by any other constraint.*

Proof. Let us assume that tuple t is found forbidden when c^i_{k+1} enters the join process. This means that t was allowed by c^i_{k+1} and by $\bowtie c^i_1, \ldots, c^i_k$, but t did not have a common support in x_i. If t was allowed by $\bowtie c^i_1, \ldots, c^i_k$, it was not found forbidden in a previous step. If tuple t has been eliminated when joining c_{k+1}, it will not appear in the new join ($\bowtie c^i_1, \ldots, c^i_{k+1}$). Therefore, it cannot be eliminated in the future. The condition that constraints must be considered by increasing arity is to assure that forbidden tuples are generated by increasing arity. $\square$

ADC-F, as presented in Figure 2, admits some improvements. First, in the last iteration of the loop of lines 10-14, the join $c^i_j \bowtie c^i_k$ (line 14) is not needed because its result will not be used in the next iteration. So it can be saved. Second, if c^i_k is one of the constraints to be processed in the last iteration of the loop of lines 10-14, the unique operation to be done is the inferred nogoods (line 13). In this case, the tuples of c^i_k which are supported by every value of x_i are not needed: they will not generate any tuple in the result because their support will never have an empty intersection with the support of any other tuple of other constraint. Therefore, we know that from the set B_i, we can choose one constraint for which universally supported tuples should not be generated. As a direct heuristic, we choose the constraint of highest arity in B_i as the last one to be processed.

5 Experimental Results

We have evaluated ADC, ADC- and ADC-F on random binary problems, SAT problems, n-queens and Schur's lemma. In all the problems and instances tested, space was a more important concern than time. A constraint can quickly grow to an intractable size. So we have tested all the instances until the program ran out of memory.

When eliminating a variable ADC-F adds up to a linear number of new constraints with respect to the size of the bucket. So the sequence of joins is an important decision. We have worked on several heuristics and two showed to bring substantial benefits: *minimum resulting arity* and *maximum expected nogoods generated*. The first one selects the two functions of smaller scope. The second one selects the two functions that have smaller support sets in its tables of memory projection, smaller supports are intended to produce more empty intersections.

5.1 SAT and Random Problems

A binary random problem class is defined by the tuple $\langle n, m, p_1, p_2 \rangle$, where n is the number of variables, m is the number of values per variable, p_1 is the problem connectivity and p_2 is the constraint tightness. With random problems the advantages of ADC-F with respect to ADC can be controlled by the tightness and the connectivity of the generated problems. ADC-F has its greater gain when the tightness is inferior to 0.5 and connectivity is close to 1. In that case, a gain of 3 orders of magnitude is reached. Connectivity is also an important parameter because combined with loose constraints can make a variable elimination very expensive for ADC. We observed in our experiments that no matter the connectivity of the graph, the positive representation is more advantageous when tightness is greater than 0.5 . This can be seen in Figure 3 on the left where ADC and ADC-F lines cross.

An extreme case of loose constraints is the SAT problem modelled as a CSP using the model of one variable per logical variable, and each clause a constraint with a single negative tuple (the combination that forbids the clause) [7]. In Figure 3 on the right we can appreciate a gain of 3 to 1 orders of magnitude as the number of clauses grows (in this case the connectivity of the graph also grows). It is interesting to notice that ADC-F maintains a constant gain of one order of magnitude even when the number of clauses grows, that is because there is a single forbidden tuple in every constraint.

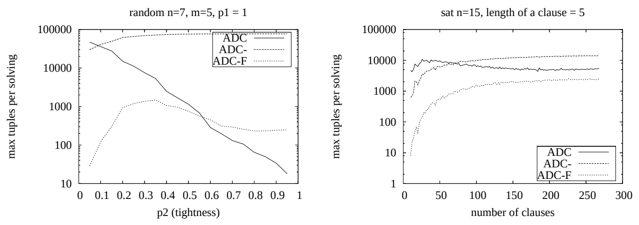

Figure 3. On the left: results for the random binary class $\langle n = 7, m = 5, p_1 = 1 \rangle$. On the right: results for 5-SAT instances.

5.2 Shur's Lemma and n-queens

The problem is to put n balls labelled from 1 to n into 3 boxes so that for any triple of balls x, y, z with $x + y = z$, not all are in the same box. 23 is the greater number of balls that can be placed into three boxes. The problem is modelled having $3n$ binary variables each one indicating whether there is a ball in a particular box. In the model a particular variable appears in few constraints (usually $n-1$ constraints) of arity 3, but of different scopes. In this case the heuristic *minimum arity* caused the best performance. ADC-F could solve one more instance than ADC. When $n=7$, ADC-F requires 2.88 times less tuples and is 11.2 times faster.

The n-queens problem is to place n queens into a $n \times n$ chessboard in such a way that none of them attack each other. The n-queens is modelled with n variables each one with n values. In each elimination we have to build a constraint involving all the variables because each variable is related to all other variables. In this case the heuristic *minimum arity* did not have much benefits. The heuristic *minimum number of expected tuples* reduced considerably the number of tuples, a bit less than an order of magnitude. The factorization does not have any effect for small arity constraints because when n grows, nogoods also grow in size. For example, the smallest nogoods different from those contained in the original constraints for 8-queens are of arity 4. Figure 5 reports the results of this experiment. ADC-F generates 6.93 times less tuples and is about 30 times faster. In the 8-queens the first elimination is the most expensive. We have tested all the instances until the program ran out of memory. ADC-F could solve one more instance than ADC, that is shown in Figure 4 as the line execution corresponding to $n=8$ does not appear in the ADC plots.

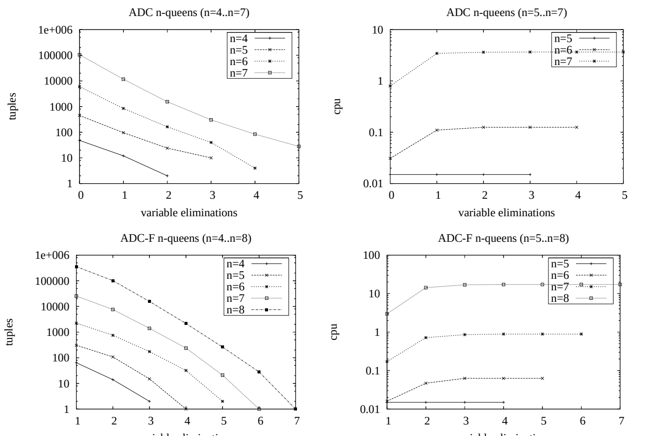

Figure 4. Results for the n-queens problem.On top, ADC tuples and CPU time. On bottom, ADC-F tuples and CPU time.

5.3 Discussion

Although ADC is highly inefficient with loose constraints, ADC- offers no improvements in practice. Only when constraints to join are really loose (for example in random problems when tightness is below 0.1), ADC- improves over ADC. The intuitive reason is that, when ADC joins two constraints the resulting number of tuples is bounded by $|c_i|.|c_j|$. This is not the case for ADC-, where the negative version of join generates an exponential number of tuples with respect to the non-common variables.

ADC-F deals also with negative information. The difference with ADC- is that at every two by two join it generates a negative constraint filled with nogoods that will in fact factorize other bigger nogoods that will not appear lately. ADC-F can perform a linear number of those factorizations at every bucket. This set of factorized negative constraints when joined together are equal to the negation of the positive constraint that ADC would generate. Imagine a variable linked to many loose constraints. ADC generates a single constraint containing all the allowed combinations. Instead, ADC-F generates a set of negative constraints that contain the factorized forbidden combinations. For example, if a single value is forbidden at any moment it only appears once inside a unary constraint. Moreover it can be proved that in very special cases, the elimination of a variable may not generate a constraint of arity equal to all its neighbors when performing ADC-F, but a smaller arity constraint. This advantage of factorizing nogoods raises the power of eliminating a variable linked to loose constraints and can reach gains of several orders of magnitude in random problems and SAT, and of one order of magnitude in n-queens and Shur's lemma.

ADC-F joins constraints in the bucket and generates constraints of nogoods at the same time. At the last join, when only two constraints remain, ADC-F only needs to generate nogoods as the positive join is subsumed by all the generated nogoods. Because of this fact we can always choose a constraint of the bucket that when computing its projection with memory its completely permitted tuples (the ones that are supported by every value of the eliminated variable) will be skipped. This constraint will be the one of largest arity. When projecting with memory a negative constraint, we can compute at the same time the $\oslash$ operation (that is the tuples that are fobidden by all values of the eliminated variable) and also the tuples supported by all values of the eliminated variable.

6 Conclusions

The theoretical complexity of ADC is exponential with respect to the width of the induced graph, which is very sensitive to the arity of constraints and does not take into account its tightness. In this work, we show how sensitive is ADC not only to the arity but also to the tightness of constraints. Especially, very loose constraints and variables linked to many loose constraints can make the algorithm impractical in many cases. We have described ADC-F that eliminates a variable by returning a set of constraints that does not mention that variable and that represent a set of factorized nogoods in such a way that variables are eliminated in a compact way, sometimes with exponential savings. As general conclusion, when constraints have more permitted than forbidden tuples, ADC-F is the preferred choice. Otherwise, when constraints have more forbidden that permitted tuples, classical ADC may perform better.

REFERENCES

[1] U. Bertele and F. Brioschi, *Nonserial Dynamic Programming*, Academic Press, 1972.
[2] R. Dechter, 'Bucket elimination: A unifying framework for reasoning', *Artifical Intelligence*, **113**, 41–85, (1999).
[3] R. Dechter and J. Pearl, 'Network-based heuristics for constraint satisfaction problems', *Artifical Intelligence*, **34**, 1–38, (1987).
[4] G. Gottlob, N. Leone, and F. Scarcello, 'A comparison of structural csp decomposition methods', *Artificial Intelligence*, **124**(2), 243–282, (2000).
[5] J. Larrosa and E. Morancho, 'Solving still life with soft constraints and variable elimination', in *Proc. CP-2003*, pp. 466–479, (2003).
[6] J. Pearl, *Probabilistic Inference in Intelligent Systems. Networks of Plausible Inference*, Morgan Kaufmann, San Mateo, CA, 1988.
[7] T. Walsh, 'Sat v csp', in *Proc. CP-2000*, pp. 441–456, (2000).