Rewriting Numeric Constraint Satisfaction Problems for Consistency Algorithms

Claudio Lottaz
AI-Laboratory, Computer Science Department
Swiss Federal Institute of Technology
CH-1015 Lausanne (Switzerland)
lottaz@lia.di.epfl.ch

Abstract

Reformulating constraint satisfaction problems (CSPs) in lower arity is a common procedure when computing consistency. Lower arity CSPs are simpler to treat than high arity CSPs. Several consistency algorithms have exponential complexity in the CSP's arity, others only work on low arity CSPs.

Much work in constraint satisfaction has concentrated on binary CSPs, since in a theoretical view any CSP on discrete domains can be reformulated in binary form. Although this is not true for numeric CSPs the constraints of which are equalities and inequalities specified using mathematical expressions, it has been shown that rewriting such CSPs in terms of ternary constraints is possible as long as only unary and binary operators occur in the mathematical expressions. Nevertheless, very few methods to actually perform this task automatically have been suggested so far, the reformulation is often done by hand.

In this paper we present algorithms to rewrite numeric CSPs in terms of ternary constraints by introducing auxiliary variables. Since the complexity of consistency algorithms also depends on the number of variables involved, we suggest heuristics, to keep the number of introduced auxiliary variables low and to eliminate unnecessary variables from the original CSP.

Keywords: numeric constraint satisfaction problems, consistency algorithms, reformulation, symbolic algebra

1 Introduction

Several constraint satisfaction algorithms focus on numeric constraint satisfaction problems (CSPs). A numeric CSP is defined by a set of variables, their domains and the set of constraints which must be satisfied for any solution. Thereby the domains of the variables are intervals in $\Re$ and the constraints are expressed as closed mathematical expressions (equalities and inequalities).

Such CSPs can accurately model many engineering and design problems from domains such as mechanical, electrical and civil engineering.

Constraint satisfaction techniques mainly fall into two categories: search algorithms and consistency algorithms. Search algorithms determine single solutions to a given CSP. For this task they employ various kinds of intelligent backtracking and splitting techniques. Consistency algorithms eliminate parts from search space in which no solution can be expected due to local inconsistencies. Thus they are approximating the solution space of a CSP. In certain cases, consistency algorithms can compute the solution space of a numeric CSP exactly.

The arity of a constraint is the number of variables it involves. The arity of a CSP is equal to the arity of the highest-arity constraint. Reformulation of numeric CSPs in lower arity before computing consistency is a common procedure because CSPs of lower arity are considerably simpler to treat. Therefore certain consistency algorithms such as 2-consistency as described in [5, 6] only accept ternary constraints and 2B- as well as 3B-consistency [7] are based on primitive constraints which are also ternary. The complexity of other consistency algorithms such as $(r, r-1)$-relational consistency [9] are exponential in the arity of the given CSP and therefore low-arity CSPs are treated much more efficiently.

Much work in constraint satisfaction has focussed on binary CSPs, since theoretically any CSP on discrete domains can be reformulated in binary form. The suggested methods to reformulate discrete CSPs [4, 3, 1], however, are not applicable to numeric CSPs, because the continuous domains of their variables cannot be enumarated and thus the domains and constraints for the hidden variable or the dual graph representation cannot be expressed.

Although numeric CSPs cannot be transformed into binary form, it has been shown that rewriting such CSPs in terms of ternary constraints is possible as long as only unary and binary operators occur in the constraints. It is intuitively clear that any mathematical expression built using unary and binary operators can be rewritten in ternary form by introducing an auxiliary variable for each intermediary result generated by a binary operator. This method, of course, possibly generates many auxiliary variables. So far, very few algorithms to perform the task of reformulating numeric CSPs in ternary form automatically have been suggested, the rewriting is often done by hand.

Since the performance of all above mentioned consistency algorithms strongly depends on the number of variables involved in the given CSP, algorithms which introduce a small number of auxiliary variables are needed. For the same reason it is interesting to see if it is possible to eliminate unnecessary variables from the original CSP. When formalizing problems engineers and designers often use intermediary variables and constants which make the CSP more readable and reusable. However, in the context of computing consistency the elimination of such variables may be beneficial.

The algorithms suggested in this paper are particularly suitable for consistency algorithms which treat ternary constraints without any restriction on the complexity of the constraints' expressions. Examples of such algorithms are described in [5, 6, 9, 10]. We improve existing suggestions, such as Algorithm 3, by removal of unnecessary intermediary variables from the original CSP and by optimizing the number of auxiliary variables introduced.

The next section contains theoretical considerations about the relation between constraint arity and consistency algorithms. The third section is dedicated to the description of algorithms for rewriting numeric CSPs in ternary form and for eliminating unnecessary variables of the original CSP. Finally we show experimental results before drawing our conclusions.

2 Constraint arity and consistency algorithms

Treating high-arity constraints in consistency algorithms directly is very complex. Algorithms which enforce consistency on the symbolic level, face important analytic problems when finding extrema, intersections and the like. Algorithms which use explicit spatial representations of the constraints spend an outrageous effort to store these. The following considerations give an intuition about the benefits of computing $(r, r-1)$-relational consistency on low arity CSPs instead of high-arity ones.

In [9] it has been shown that, for a CSP of arity r, given certain partial convexity restrictions, $(r, r-1)$-relational consistency is equivalent to global consistency, i.e., all solutions can be found without backtracking. Enforcing $(r, r-1)$-relational consistency, however, requires an algorithm with computational complexity $O(n^{2r-1})$, where n is the number of variables in the CSP. As long as the convexity restrictions needed for global consistency keep satisfied in the rewritten CSP, we can expect the results to be equivalent. Thus, since the algorithm's complexity is exponential in r, rewriting a numeric CSP in lower arity before computing $(r, r-1)$-relational consistency has the potential to accelerate the calculation.

Numeric CSPs expressed as mathematical expressions using unary and binary operators can be rewritten with lower arity by introducing auxiliary variables. It is intuitively clear that by introducing auxiliary variables for the intermediary results of all binary operators in a constraint, the constraint's arity can be reduced at the expense of increasing the number of variables (the auxiliary variables) and constraints (the definitions of the auxiliary variables).

A constraint which defines an auxiliary variable that actually helps to reduce the arity of another constraint has at least arity three. Otherwise the auxiliary variable would replace an expression which depends only on one variable by a new variable and could thus not reduce the arity of any constraint. Therefore, a numeric CSP rewritten in the above manner has at least arity three. On the other hand arity three can always be reached by replacing every binary operator by auxiliary variables.

So, on one hand decreasing the arity of a CSP before computing $(r, r-1)$-relational consistency reduces the computational complexity. On the other hand there is a trade-off between decreasing arity and increasing the number of variables of the CSP by introducing auxiliary variables in order to achieve the lower arity. For deciding whether rewriting CSPs in lower arity pays off, we estimate how many auxiliary variables are introduced. Since $(r, r-1)$-relational consistency's complexity is exponential in r, we only consider the case when the arity is reduced as much as possible, i.e. as mentioned before, to an arity of three.

In the worst case rewriting a numeric CSP of arity $r > 3$ in ternary form requires the addition of $O(m)$ auxiliary variables, where m is the number of binary operators in the CSP. In this case the complexity $O((n+m))^5)$ for (3,2)-relational consistency on the rewritten constraint set compares to the complexity $O(n^{2r-1})$ for $(r, r-1)$-relational consistency on the original CSP. In practical problems the exponential influence of r can be expected to outweigh by far the polynomial influnce of m.

Moreover, several consistency algorithms for numeric CSPs were developed specifically for ternary CSPs [5, 6, 7, 9], since any numeric CSP expressed with closed mathematical expressions in unary and binary coperators can be rewritten using ternary constraints exclusively. Nevertheless, very few methods to actually perform this task automatically have been suggested so far, and the known methods in general generate far too many auxiliary variables, thus impliying inefficient subsequent computation of consistency. In fact the CSPs are often rewritten by hand although this may take a long time for large examples.

3 Rewriting numeric CSPs

Although the complexity considerations above show that rewriting numeric CSPs has the potential to accelerate consistency algorithms considerably, the gain in performance strongly depends on the number of variables in the transformed CSP. In this section we present heuristics to keep the number of variables in the rewritten CSP low.

3.1 Eliminating unnecessary intermediary variables

Designers and engineers use constants and intermediary variables to keep their mathematical formulas easier to read and adaptable to other contexts. Some of these unnecessary variables can and should be eliminated from the CSP by substitution before computing consistency.

An intermediary variable a is defined in the CSP as the result of a functional expression $a = f(x_1, \ldots, x_n)$. It can be removed from the CSP by eliminating its definition from the CSP and by substituting a wherever it occurs in the remaining constraints of the CSP by $f(x_1, \ldots, x_n)$. However, substitution of a is only enough to keep the CSP equivalent, when a is a constant. Otherwise the information contained in the domain of a is lost and thus the new CSP is less restrictive. Therefore, in addition the constraints $\overline{a} > f(x_1, \ldots, x_n)$ and $\underline{a} < f(x_1, \ldots, x_n)$ must be added where $\underline{a}$ and $\overline{a}$ are the lower and upper bound of the domain of a.

Candidates for potentially unnecessary variables can be found in any equality. Solving an equality for one of its variables a yields the definition $f(x_1, \ldots, x_n)$ for a. However, this is only valid if $f(x_1, \ldots, x_n)$ is functional. Otherwise, for instance if the original equation was quadratic in a, substitution of a by $f(x_1, \ldots, x_n)$ in the CSP is not equivalent. In the case of a quadratic expression, substitution of a by $f(x_1, \ldots, x_n)$ implies the loss of one of the two possible solutions for a.

If a is a valid candidate for substitution, it should be substituted if its substitution does not increase the arity of any constraint to more than three. Removing a variable the substitution of which renders any constraint C non-ternary is unlikely to be useful in the context of making CSPs ternary, because it would imply an additional subsequent generation of at least one auxiliary variable in order to rewrite C in ternary form.

For illustration consider the following small example, a simplified problem from civil engineering:

$$\begin{array}{rcl} u & < & (3.18e^{-5}H_s + 0.0054)S \\ H_s & > & 137.7 - 0.08633S + 5.511e^{-5}S^2 - 8.358e^{-9}S^3 \\ p & = & u + 9.62 \cdot 10^{-5}(0.0417W)^{1.5161} \\ H_b & > & 0.077(pW^2)^{0.3976} \\ H_b & > & 0.0168(SW^3)^{0.2839} \end{array}$$

This system of constraints contains the intermediary variable p defined as $u + 9.62 \cdot 10^{-5}(0.0417W)^{1.5161}$. Thus the definition of p only involves the variables u and W. The only occurrence of p is in $H_b > 0.077(pW^2)^{0.3976}$. Substituting p in this ternary constraint leaves it ternary because W is involved in both, the definition of p and the constraint where p is to be substituted. Therefore, p is an unnecessary intermediary variable and its elimination accelerates computing consistency.

Since the substitution of a constant or an intermediary variable in the above described manner may decrease the arity of a constraint, unnecessary variables are eliminated iteratively until no more changes occur. The algorithm is shown in Algorithms 1 and 2.

3.2 Making constraints ternary

The so far suggested methods for rewriting CSPs in ternary form introduce far too many auxiliary variables in most cases. In this section we describe a heuristic to generate fewer auxiliary variables.

3.2.1 Introducing auxiliary variables

A simple algorithm to rewrite a given CSP in terms of ternary constraints is suggested in [9, 6]. It replaces any binary operator in a constraint by a new auxiliary variable which represents its result. This step is iterated until all constraints have arity three or less. (see Algorithm 3). Thereby the operands

```
procedure eliminate-unnecessary(constraints)
    while changes occur do
        foreach equality $C \in$ constraints do
            foreach variable $a \in C$ do
                if check-subs(a, C, constraints) then
                    remove $C$ from constraints
                    substitute $a$ in constraints
```

Algorithm 1: *Eliminating constants and unnecessary intermediary variables from numeric CSPs.*

```
function check-subs(a, C, constraints)
    $f \leftarrow$ solve $C$ for $a$
    if $f$ is not functional then return false
    foreach $C' \in$ constraints do
        $C'' \leftarrow$ substitute $a$ in $C'$ by $f$
        if arity $C''$ > arity $C'$ then
            if arity $C'' > 3$ then return false
    return true
```

Algorithm 2: *This function checks if substituting* a *defined in* C *is likely to be useful for subsequent computation of consistency.*

of the chosen operator do not contain any binary operator in order to avoid introducing non-ternary constraints when defining auxiliary variables.

```
procedure simple-ternarize(C)
    while C is not ternary do
        choose a subexpression $e : x_i \circ x_j$ of $C$
        substitute $e$ in $C$ by new variable $x_{n+1}$
        add the constraint $x_{n+1} = x_i \circ x_j$
```

Algorithm 3: *Simple algorithm to make one constraint ternary. x_i and x_j do not contain binary operators.*

This algorithm shows that it is always possible to rewrite a numeric CSP expressed using unary and binary constraints in terms of ternary form but it generates far too many auxiliary variables for various reasons. First, it unnecessarily introduces binary constraints if x_i or x_j are constants or involve the same variable. Second, it does not allow the introduction of complex definitions for auxiliary variables, since only one binary operator is allowed. Finally, it does not reuse auxiliary variables in other constraints or subexpressions.

Some implementations improve upon the last critique about not reusing auxiliary variables by avoiding duplicate definitions. This allows for some optimization, however, current algorithms do not try to provoke the reuse of auxiliary variables when choosing the expressions to define these. Therefore many opportunities for reusing auxiliary variables are missed.

We suggest a more general algorithm to perform the task of rewriting numeric CSPs in ternary form: In the first step the constraints which already have ternary form are sorted out and are no longer manipulated. In the second step the algorithm searches for an expression in two variables which occurs in one of the n-ary constraints. The third step is to substitute the expression found in step two in all non-ternary constraints. These three steps must be repeated until the list of non-ternary constraints is empty. Algorithm 4 illustrates this procedure. In step two subexpressions involving exactly two variables are chosen because these expressions have the potential to decrease the arity of a constraint, and at the same time they do not add non-ternary constraints to the system.

When a new auxiliary variable is added, its domain must be determined as well. Interval-arithmetic provides utilities to find upper and lower bounds for the auxiliary variables according to their definitions and the domains of the variables involved in their definition. However, when variables occur several times in an expression which defines a new auxiliary a, the domains of a may be overestimated. This is inherent to interval arithmetics and not crucial for several consistency algorithms. On the other hand interval arithmetic guarantees that no solutions to the original numeric CSP are lost due to underestimation of domains.

```
function make-ternary(constraints)
    $ternaries = \emptyset$
    $i \leftarrow 1$
    while $constraints \neq \emptyset$ do
        foreach $C \in constraints$ do
            if arity C $\leq 3$ then
                remove $C$ from constraints
                add $C$ to ternaries
        In constraints find $f(x, y)$ in two variables
        add $aux_i = f(x, y)$ to ternaries
        substitute $f(x, y)$ by $aux_i$ in constraints
        $i = i + 1$
    return ternaries
```

Algorithm 4: *Make numeric CSPs ternary.*

3.2.2 Defining auxiliary variables

In order to find expressions for defining auxiliary variables, we must find subexpressions in two variables occurring in the CSP. This is performed by traversing the expression tree defined by the CSP. Whenever the traversing algorithm visits subexpressions involving exactly two variables, it stores them into a list instead of descending further into the expression tree. Thus no subexpressions of expressions in two variables are considered.

What makes traversing an expression tree more complex than expected is that addition and multiplication are commutative. When we encounter the expression $a+b+c$, we have to consider $a+b$, $a+c$ and $b+c$ as possible subexpressions. In fact, computer algebra systems like Maple V treat addition and multiplication as n-ary operators. Finding all subexpressions implies considering all subsets of an operator's operands. Algorithm 5 returns subexpressions in two variables occurring in an expression. Running it on each constraint determines the candidates for defining auxiliary variables.

As soon as the candidate expressions are determined we must decide which is the best expression to be used. Since we want to decrease the arity of all constraints below four, the sum of the arities of all non-ternary constraints seems to be a reasonable criteria. Therefore we choose the candidate expression which decreases the arity of the most of the non-ternary constraints, breaking ties in favor of the candidates which generate the simplest constraints after substitution, i.e., the constraints with the fewest operands. In order to determine the best candidate expression, all candidates are substituted in the non-ternary constraints and the one yielding the best result is chosen.

```
function find-subexpressions(expr)
    if expr involves 0 / 1 variable then return ($\emptyset$);
    if expr involves 2 variables then return (expr);

    subs $\leftarrow \emptyset$
    foreach subset S of expr's operands do
        e $\leftarrow$ expression with expr's operator on S
        subs $\leftarrow$ subs $\cup$ find-subexpressions(e)

    return (subs)
```

Algorithm 5: *Finding all expressions in two variables.*

3.2.3 Complexity considerations

Algorithms 1 and 2 eliminate unnecessary variables from a CSP. The outermost loop of Algorithms 1 is executed once for each constraint, i.e. c times, in the worst case, when all constraints in the CSP can be eliminated. The second loop is called for each remaining constraint and the innermost loop is called r times in the worst case, where r is the arity of the CSP. Therefore $check - subs$ is called $O(rc^2)$ times. $check - subs$ performs its task in $O(c)$ and thus the complexity of the elimination of unnecessary variables is cubic in the number of constraints in the CSP and linear in its arity.

The simple algorithm for making one constraint ternary (Algorithm 3) replaces all but two binary operators by auxiliary variables, because a constraint involving three variables has at least two binary operators. Thereby we consider the constraints in normalized form, i.e. with zero on the righthand side. The algorithm has to be launched for each constraint in the CSP. Hence the overall complexity is $O(mc)$ where m is the number of binary operators and c is the number of constraints in the CSP.

In order to estimate the complexity of our suggestion to rewrite a CSP in ternary form, we give the number of times the substitution of a candidate in the whole CSP is performed. In the worst case Algorithm 4 also introduces m auxiliary variables. For each of these, Algorithm 5, called on all constraints, finds $O(c \cdot 2^{RD})$ subexpressions in the worst case, where R is the maximum arity of operands, and D is the maximum depth of expressions. Given that R and D are bounded, this yields that substitution is called $O(mc)$ times. The complexity of the substitution itself is difficult to estimate, because we use the symbolic algebra package Maple V to perform this task.

However, our experimental results show that Algorithms 4 and 5 can be used on problems of considerable size. Moreover, the computation of consistency by far outweighs this symbolic pretreatment in time cost.

4 Experimental Results

We implemented the above mentioned algorithms in Maple V, making use of Maple's facilities for symbolic manipulation of expressions, namely its sophisticated methods for substituting subexpressions in equalities and inequalities. The algorithms are used in ***Space*Solver**, an Internet-based solver for numeric CSPs, accessible at *http://liawww.epfl.ch/˜lottaz/SpaceSolver/*.

4.1 Examples

The experiments we present in this section involve constraint sets which model problems in mechanical, electrical and civil engineering, namely:

- statics in a school building (Example 1)
- a kinematic pair (two gearwheels, Example 2)
- kinematics in a robot arm (Example 3)
- transistors in an electronic circuit (Example 4)
- dynamics in two stacked gyms (Example 5)
- ventilation in a computer building (Example 6)

The characteristics of these examples are summarized in Table 1. Examples 1 and 2 are borrowed from [9], Examples 3 and 4 are borrowed from [2, 11], and Examples 5 and 6 are taken from [8]

Example	$\|V\|$	$\|C\|$	arity
1	6	5	3
2	9	8	7
3	12	12	9
4	12	12	5
5	11	15	5
6	39	41	6

Table 1: *Characteristics of our examples. $|V|$: number of variables, $|C|$ number of constraints, 'arity': arity of the highest-arity constraint.*

4.2 Removing unnecessary variables

Table 2 compares the four variants for removal of unnecessary variables we suggest in the context of making numeric CSPs ternary. As expected, removing constants and unnecessary intermediary variables which depend on one other variable never causes problems and always allows smaller CSPs after the rewriting in ternary form. Such removals only occur in Examples 5 and 6 in our sample but are expected to be quite frequent in engineering problems.

Example		1	2	3	4	5	6
Eliminate constants	$\lvert R\rvert$	0	0	0	0	6	13
	$\lvert A\rvert$	0	4	9	17	4	31
	$\lvert V\rvert$	6	13	21	29	15	70
	time	0.1s	0.5s	1.7s	6.1s	1.1s	61.3s
Eliminate depend on 1	$\lvert R\rvert$	0	0	0	0	8	20
	$\lvert A\rvert$	0	4	9	17	4	31
	$\lvert V\rvert$	6	13	21	29	13	63
	time	0.1s	0.4s	1.8s	6.2s	1.2s	61.3s
Eliminate depend on 2	$\lvert R\rvert$	1	2	2	0	8	27
	$\lvert A\rvert$	0	7	8	17	4	38
	$\lvert V\rvert$	5	14	18	29	13	63
	time	0.3s	0.8s	1.6s	6.2s	1.2s	75.4s
Eliminate depend on 3	$\lvert R\rvert$	1	2	2	0	8	28
	$\lvert A\rvert$	0	7	8	17	4	35
	$\lvert V\rvert$	5	14	18	29	13	59
	time	0.2s	0.8s	1.8s	6.2s	1.2s	67.4s

Table 2: *Compare the four variants of our rewriting algorithms, namely: eliminate unnecessary variables which depend on zero (constants), one, two and three other variables respectively.* $|R|$*: removed variables,* $|A|$*: auxiliary variables added when making ternary,* $|V|$*: variables in the rewritten CSP, 'time': time needed for the rewrite.*

Removing intermediary variables which depend on two other variables occurs surprisingly often in our sample (Examples 1, 2, 3 and 6). It is interesting to note that in Example 2 the removal of an unnecessary intermediary variable causes additional auxiliary variables to be added during the process of making the CSP ternary. The removal of this more complex unnecessary variable prevents *make-ternary* from recognizing certain reoccurring subexpressions and thus the result gets worse than without the removal. However, in all other cases the removal of the unnecessary intermediary variable makes the resulting CSP smaller.

Removing intermediary variables which depend on three other variables is rarely possible, because all constraints where it is to be replaced must not involve any other variable but those which occur in the unnecessary variable's definition. However, the only case observed in our examples allows a reformulation of Example 6 smaller by four variables.

The running times of the algorithms for the rewrite (including $make-ternary$) as they are given in Table 2 show, that these reformulation algorithms are not very time consuming compared to enforcing high degrees of consistency. All variants of the rewrite are run and the best result is chosen for subsequent computation. The running times are measured on a SUN Ultra 60.

4.3 Rewriting examples in ternary form

Table 3 compares the simple algorithm given in Algorithm 3 with the Algorithms 4 and 5 for rewriting a numeric CSP in ternary form. It shows that choosing the expressions to define new auxiliary variables carefully results in important gains in number of variables, and thus accelerates the execution of consistency algorithms.

The first column in Table 3 gives the number of variables of the CSPs rewritten by Algorithm 3 when the constants were substituted beforehand. The second column shows the time needed for rewriting the CSP in ternary form using all variants for removal of unnecessary intermediary variables with Algorithms 1 and 2 before rewriting with Algorithms 4 and 5. The third column gives the number of variables of the smallest CSPs found. The last column contains the estimated speed-up for (3,2)-relational consistency according to its complexity ($O(n^5)$) and the sizes of the CSPs, i.e. $\frac{t_{simple}}{t_{heuristic}} = (\frac{n_{simple}}{n_{heuristic}})^5$.

Only in Example 2 our algorithms cannot improve the performance of consistency because the structure of this CSP allows for too little freedom to choose the auxiliary variables. In our sample we observe that small examples have less potential for improvement while in larger ones the search for the best expressions to define auxiliary variables pays off. In Example 4, Algorithm 3 generates many auxiliary variables with definitions depending on only one variable, because the constraints contain many coefficients and constants in the expressions. The same effect occurs in Example 3 and Example 5, but in addition several opportunities for reusing auxiliary variables are missed. In Example 6 the effects mentioned before accumulate with the fact that the simple algorithm cannot define an auxiliary variable with more than one binary operator.

It turns out in our experiments that the time needed to enforce consistency largely outweighs the effort for rewriting a numeric CSP in ternary form in the way we suggest.

Example	n_{simple}	$t_{rewrite}$	$n_{heuristic}$	$\frac{t_{simple}}{t_{heuristic}}$
1	6	0.7 s	5	2.5
2	13	2.5 s	13	1.0
3	29	6.9 s	18	10.9
4	53	34.7 s	29	20.4
5	18	4.7 s	15	5.1
6	99	265.4 s	59	13.3

Table 3: *Compare Algorithm 3 to Algorithms 4 and 5. 'n_{simple}'/'$n_{heuristic}$': variables in the CSPs rewritten by Algorithm 3 and Algorithms 4/5 respectively, '$t_{rewrite}$': time needed by Algorithms 4/5, '$\frac{t_{simple}}{t_{heuristic}}$': expected speed-up for (3,2)-relational consistency.*

4.4 Comparing $(r, r-1)$-relational consistency on original CSP to (3,2)-relational consistency on rewritten CSPs

In Section 2 it was claimed that the exponential influence of a CSP's arity outweighs the increase of variables due to rewriting the CSP in lower arity when computing $(r, r-1)$-relational consistency. Given the above mentioned results about what we can reach in rewriting CSPs automatically, let us illustrate this claim with some runtime estimations for our examples. Table 4 gives an estimation of the ratio of expected running times between $(r, r-1)$-relational consistency on the original CSPs and $(3, 2)$-relational consistency on the CSPs rewritten in ternary form based on their respective computational complexity. Since the theoratical complexity of $(r, r-1)$-relational consistency is known to be $O(n^{2r-1})$ and the one of $(3, 2)$-relational consistency is thus $O(n^5)$, we compute an estimation of the ratio beween the aforementioned ratio as

$$\frac{t_o}{t_r} = \frac{n_o^{2r_o-1}}{n_r^5}$$

where t_o is the runtime for $(r, r-1)$-relational consistency on the original CSP, n_o is its number of variables and r_o its arity. t_r represents the runtime of $(3, 2)$-relational consistency on the corresponding rewritten CSP and n_r is the number of variables including auxiliary variables of the rewritten CSP. ‘

Example	n_o	r_o	n_r	$O(\frac{t_o}{t_r})$
1	6	3	5	2.45
2	9	7	13	$6.85 \cdot 10^6$
3	12	9	18	$1.17 \cdot 10^{12}$
4	12	5	29	$2.52 \cdot 10^2$
5	11	5	15	$3.11 \cdot 10^3$
6	39	6	59	$7.15 \cdot 10^8$

Table 4: *Compare performance of $(r, r-1)$-relational consistency on original CSP to $(3, 2)$-relational consistency of rewritten CSP. n_o gives the number of variables in the original CSP and r_o its arity. n_r is the number of variables including auxiliary variables of the CSP in ternary form. $O(\frac{t_o}{t_r})$ is an estimation of the ratio between executaion times based on their respective theoratical complexity.*

$O(\frac{t_o}{t_r})$ in Table 4 gives a rough estimation of the ratio between the runtime of $(r, r-1)$-relational consistency on the original CSP and $(3, 2)$-relational consistency on the corresponding CSP in ternary form. On one hand the used estimations of t_o and t_r may not be directly comparable, because we use worst case complexity, thus dividing the two at least abstracts an unknown multiplicative constant. On the other hand, the convexity restrictions for global consistency are typically not satisfied and therefore the pruning achieved may not be the same. Nevertheless, the values for $O(\frac{t_o}{t_r})$ are large enough to conjecture that calculation on lower arities allows for significant gains in performance. In fact, $(r, r-1)$-relational consistency is intractable on most examples.

5 Conclusions

When considering numeric CSPs it is necessary to treat non-binary CSPs, because the reformulation in binary form as it is suggested for discrete domains are not applicable to CSPs on continuous domains. However, theoretical considerations about complexity of reaching $(r, r-1)$-relational consistency for r-ary numeric constraint satisfaction problems reveal that reformulation of such CSPs in terms of ternary constraints is very promising. In fact the complexity $O(n^{2r-1})$ for the computation on the original CSP compares to $O((n+m)^5)$ for the computation on the ternary CSP, where n is the number of variables and m is the number of binary operators in the CSP. It can be expected that in practical problems m is substantially less than would be needed to reverse the gain in complexity.

Reformulating numeric CSPs in ternary form is simple as long as the introduction of an arbitrary number of auxiliary variables is acceptable. However, in the case where the result should be minimal in the number of variables, the task is difficult. We suggest a heuristic to determine good candidate expressions to define auxiliary variables. Our tests show that the automatic rewriting introduces fewer auxiliary variables than the straight forward manner in acceptable running times.

For better readability and reusability, designers and engineers are likely to use variables which are not needed to keep the arity of the CSP low. We suggest a method to eliminate such unnecessary variables automatically in order to accelerate subsequent computing of consistency. Although the elimination of constants and unnecessary variables depending on one variable is always useful, it turns out that the substitution of complex unnecessary variables may cause problems, because the subsequent rewriting in terms of ternary constraints may fail to recognize and reuse the more complex expressions.

Acknowledgments

This work was performed within a project funded by the Swiss Priority Programme in Computer Science (SPP-IF). The author would also like to thank Djamila Sam-Haroud, Steven Willmott, Boi Faltings, Christian Bliek and Marc Torrens for encouragement and helpful discussions.

References

[1] F. Bacchus and P. Van Beek. On the conversion between non-binary and binary constraint satisfaction problems. In *National Conference on Arti-*

ficial Intelligence (AAAI), pages 311–318, Menlo Park CA, Madison WI, July 1998. AAAI Press.

[2] F. Benhamou, D. McAllester, and P. Van Hentenryck. CLP(intervals) revisited. In Maurice Bruynooghe, editor, *Logic Programming - Proceedings of the 1994 International Symposium*, pages 124–138, Cambridge MA, London, 1994. MIT Press.

[3] R. Dechter. On the expressivenes of networks with hidden variables. In *National Conference on Artificial Intelligence (AAAI)*, pages 556–562, Menlo Park CA, Madison WI, 1990. AAAI Press.

[4] R. Dechter and I. Meiri. Experimental evaluation of preprocessing techniques in constraint satisfaction problems. In N. S. Sridharan, editor, *Internation Joint Conference on Artificial Intelligence (IJCAI)*, pages 271–277. Morgan Kaufmann, August 1989.

[5] B. V. Faltings and E. M. Gelle. Local consistency for ternary numeric constraints. In *Internation Joint Conference on Artificial Intelligence (IJCAI)*, volume 1, pages 392–397, 1997.

[6] E. M. Gelle. *On the Generation of Locally Consistent Solution Spaces in Mixed Dynamic Constraint Problems.* Phd-thesis no. 1826, Swiss Federal Institute of Technology in Lausanne, 1998.

[7] O. Lhomme. Consistency techniques for numerical CSPs. In *Internation Joint Conference on Artificial Intelligence (IJCAI)*, volume 1, pages 232–238, 1993.

[8] C. Lottaz, D. Clément, B. V. Faltings, and I. F. C. Smith. Constraint-based support for collaboration in design and construction. *Journal of Computing in Civil Engineering*, 13(1):23–35, January 1999.

[9] D. Sam-Haroud. *Constraint Consistency Techniques for Continuous Domains.* Phd-thesis no. 1826, Swiss Federal Institute of Technology in Lausanne, Switzerland, 1995.

[10] D. Sam-Haroud and B. V. Faltings. Consistency techniques for continuous constraints. *Constraints*, 1(1&2):85–118, September 1996.

[11] P. Van Hentenryck, L. Michel, and Y. Deville. *Numerica - A Modelling Language for Global Optimization.* MIT Press, Cambridge MA, London, 1997.