Experimental results in constraint relaxation

Walter Hower[1] and Stephan Jacobi[2]

[1] CRP – Gabriel Lippmann, 162a avenue de la Faïencerie, L-1511 Luxembourg
hjm-w.hower@t-online.de
[2] Dornenweg 4-6, D-53819 Neunkirchen, Germany
stephan@jacobi-software.de

Abstract. Relaxation is meant as a modification of a constraint network such that the network permits more solutions; for instance, a formerly inconsistent network may become consistent. A lot of algorithms in this area try to localize constraints or even whole priority levels of constraints that must be removed to allow global consistency. However, the problem when removing entire (levels of) constraints is the high degree of violation of the original problem. (It is often the case that constraints do not permit only a few tuples which are essential to form a globally consistent solution. Removing entire levels of constraints is much farther away from the original problem than additionally permitting just the necessary tuples.) This article evaluates a more sophisticated fine-grained approach (working with tuples) that should be able to detect and solve even complex inconsistencies automatically. Besides, an optional cost function can be applied to qualify the solutions by their costs in terms of the violation of the original specification. The current paper focusses on the experimental results obtained so far.

1 Background and Concepts

1.1 Introduction

The constraint satisfaction problem (CSP) comprises a set of n variables with associated finite domains and a set of allowable value assignments ("constraints") to a subset of the variables; then, in order to get the globally consistent solution, we need to compute the set of all n-tuples consistent with the given constraints ([6]).

The current work focusses on *constraint relaxation* — how to enable a CSP to permit more solutions. The most prominent application of this feature is the conflict resolution of inconsistent constraint networks ([5]). If we cannot establish consistency the algorithm is able to detect the source of the inconsistency and to provide a conflict-resolution set of possible relaxations. A potential cost function may include the cardinality of this "candidate set" of relaxation candidates as a parameter to calculate the cost of a specific relaxation. (The cost function may allow to measure the degree of violation of the original problem.)

We like to report on the experimental results obtained — confirming the corresponding algorithms published in [7][3] (and [8]).

[3] the reason why we have to omit some parts of the formal framework here

The article is structured as follows: In the rest of this section we first make some remarks on global consistency; then, a motivating example follows which illustrates the interesting inconsistency aspect which we treat here. The subsequent section describes the system, followed by experimental results. Related work and final remarks conclude the present paper.

1.2 Global Consistency

The time complexity of the algorithms presented in [6] and [7] is linear in the size of the set of (original) constraints.[4] (It does not artificially "synthesize" redundant constraints.) This set is of course exponentially bounded by $O(2^n)$, where n is the problem size. Anyway, when we have a polynomial dependence between the problem size and the number of restrictive constraints, the algorithm will behave polynomially, in this regard — without currently considering the size of the domains. (The arc-consistency algorithm by D. Waltz worked in linear time and not quadratically because in its computer vision area the constraint graph of binary relations was not a complete one. It was a planar graph where the number of restrictive constraints only linearly depends on the number of the variables.)

The linear dependence of the complexity on the number of restrictive constraints enables the algorithm to handle just those computation steps which are really needed; therefore, it is very amenable to sparse networks.

1.3 Inconsistency

The computation of the globally consistent solution, as sketched before, may yield an inconsistency which can be recognized as the generation of an empty set during one of the two phases of the algorithm. (During the descending phase, some kind of pre-filtering is done — in the sense of local (in-)consistency; the ascending phase generates the global (in-)consistency.)[5] Usually, inconsistencies occur during the ascending phase, since pre-filtering is much less restrictive (and does not ensure satisfiability). So, we will only consider the ascending phase here.

To motivate briefly the main ideas of our framework already presented in [7], we illustrate the main features of the algorithm by the following example. It models a typical project management problem, namely resource allocation. In this example we have 8 activities to manage; the dependency graph of these activities is illustrated in Figure 1. Each of the activities takes a specific period of time to be completed and starts at a discrete point of time.

The time periods of each activity are given in Table 1 as values of a duration function.

[4] $O(|C^{orig}|)$, because it just visits the $|C^{orig}|$ constraints at most only twice

[5] The levels are indicated by the cardinalities of the index sets (referring to the arities) of the constraints. We start at the level of those constraints with the highest arity and descend in a certain manner reaching the domains (1-ary constraints) in the end; afterwards, we ascend analogously.

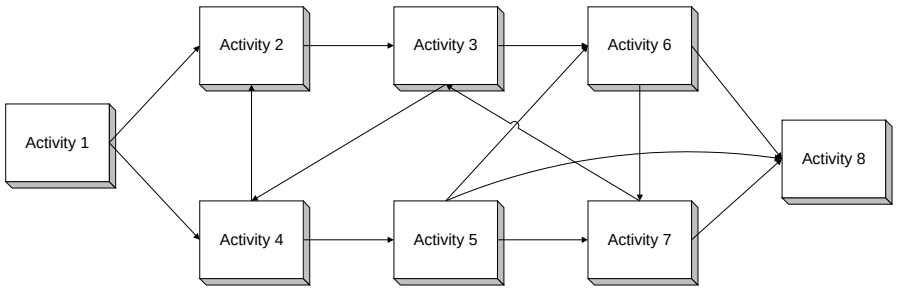

Fig. 1. dependency graph of activities

$$dur(A_1) = 3,\ dur(A_2) = 2,\ dur(A_3) = 5,\ dur(A_4) = 4,$$

$$dur(A_5) = 1,\ dur(A_6) = 2,\ dur(A_7) = 4,\ dur(A_8) = 3$$

Table 1. duration of example activities

In this example we have chosen to model the activities as variables $A_1, \ldots, A_8$ in the constraint network. The domain of each variable is then a set of discrete starting points for the activity in question. For simplicity we define the domains uniformly to be a set of 24 time points $\{t_1, t_2, \ldots, t_{24}\}$, where $t_i < t_j$ for all $i < j$.

The constraint definition is derived from the dependency graph as follows. For each activity in the graph take the node itself and every activity where the node has an incoming edge arriving at this specific activity. This construction leads to the constraint set shown in Table 2. The associated graphical representation is illustrated by Figure 2.

$$\begin{aligned}
C_{A_4,A_5} &= A_5 \geq A_4 + dur(A_4) \\
C_{A_1,A_3,A_4} &= A_4 \geq A_1 + dur(A_1) \wedge A_4 \geq A_3 + dur(A_3) \\
C_{A_1,A_2,A_4} &= A_2 \geq A_1 + dur(A_1) \wedge A_2 \geq A_4 + dur(A_4) \\
C_{A_2,A_3,A_7} &= A_3 \geq A_2 + dur(A_2) \wedge A_3 \geq A_7 + dur(A_7) \\
C_{A_3,A_5,A_6} &= A_6 \geq A_3 + dur(A_3) \wedge A_6 \geq A_5 + dur(A_5) \\
C_{A_5,A_6,A_7} &= A_7 \geq A_5 + dur(A_5) \wedge A_7 \geq A_6 + dur(A_6) \\
C_{A_5,A_6,A_7,A_8} &= A_8 \geq A_5 + dur(A_5) \wedge A_8 \geq A_6 + dur(A_6) \wedge \\
&\quad A_8 \geq A_7 + dur(A_7)
\end{aligned}$$

Table 2. constraint definitions

Now assume that the relaxation costs will be given by a set of annotated cost

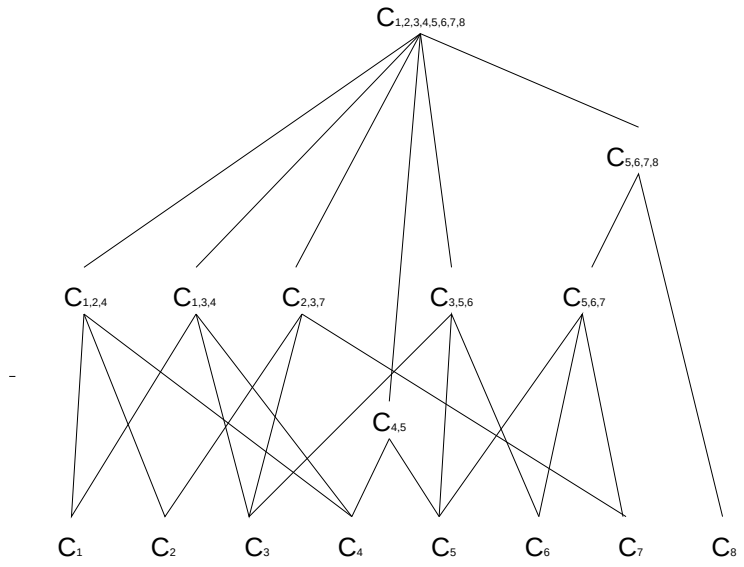

Fig. 2. associated constraint network of the example

functions as given in Table 3.

$cost(C_{A_4,A_5}) = 21,\ cost(C_{A_1,A_3,A_4}) = 43,\ cost(C_{A_1,A_2,A_4}) = 22,\ cost(C_{A_2,A_3,A_7}) = 37,\ cost(C_{A_3,A_5,A_6}) = 12,\ cost(C_{A_5,A_6,A_7}) = 54,\ cost(C_{A_5,A_6,A_7,A_8}) = 122$

Table 3. cost functions

To keep the example simple, we use a set of constant cost functions. Nevertheless, real applications may use much more sophisticated functions that take the context and domain values into consideration for their computation.

The dependency graph exhibits two cycles, namely activities A_2, A_3, A_4 and A_3, A_6, A_7. These cycles cause the inconsistency of the constraint network.

Now, the solver traverses through the network and produces a number of relaxations with their associated costs. The solver will try to find legal assignments for each variable of each constraint. Whenever it is not possible to find such assignments it will generate a so-called "candidate set", i.e. a set of assignments the solver may choose as good candidates for a legal assignment. The solving process can continue with this "candidate set" and finally, when the global solution has been reached, the relaxation costs for these global solutions are being calculated. Actually, the solver generates 47,025 possible relaxations, but there are only two classes of relaxations. The first relaxes C_{A_1,A_3,A_4} and C_{A_2,A_3,A_7}, where the second one additionally relaxes C_{A_4,A_5}. One member of each class

is shown in Table 4. The graphical representation of the relaxation results are illustrated in Figure 3 and Figure 4.

Global solutions: $(t_1, t_8, t_{10}, t_4, t_{10}, t_{15}, t_{17}, t_{21})$
Relaxation of C_{A_1,A_3,A_4} with (t_1, t_{10}, t_4) has costs of 43
Relaxation of C_{A_2,A_3,A_7} with (t_8, t_{10}, t_{17}) has costs of 37
Total costs of relaxation: 80
Global solution: $(t_5, t_{12}, t_{13}, t_8, t_{10}, t_{18}, t_{20}, t_{24})$
Relaxation of C_{A_4,A_5} with (t_8, t_{11}) has costs of 21
Relaxation of C_{A_1,A_3,A_4} with (t_5, t_{13}, t_8) has costs of 43
Relaxation of C_{A_2,A_3,A_7} with (t_{12}, t_{13}, t_{20}) has costs of 37
Total costs of relaxation: 101

Table 4. relaxations and costs

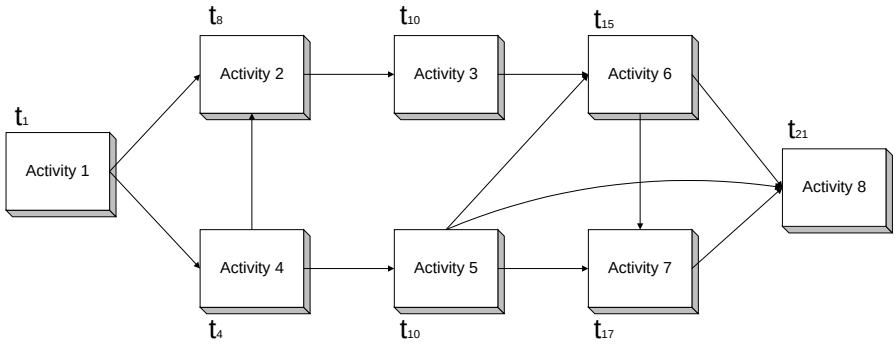

Fig. 3. resulting dependency graph for the first relaxation

The example should just sketch the basic ideas. It does not cover all possible cases of inconsistencies the algorithm is able to handle. For instance, here the inconsistency occurs first at the n-ary constraint. It is also possible that the inconsistency already occurs at lower-ary stages or that inconsistencies occur several times. Nevertheless, it is a real world example of a typical CSP. Another (the *traffic lights*) example can be found in [7].

2 System Design and Implementation

2.1 Equi-Join Optimization

The equi-join is performed for several purposes and especially very often. So, the performance of this operation has a deep impact on the whole system. The

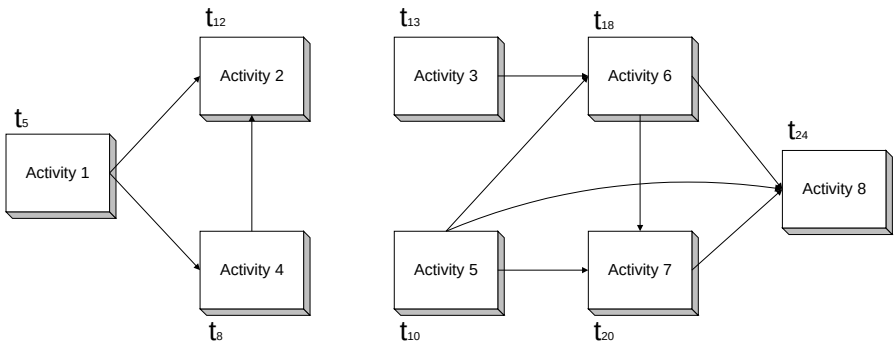

Fig. 4. resulting dependency graph for the second relaxation

first versions of the constraint solver used a simple strategy for calculating the equi-join. It just sorted the constraints by the cardinality of the set of consistent tuples and started with the two smallest sets. This intermediate set was joined with the next set of tuples and so on. Finally, the result was intersected with the set of "admissible" tuples to get the set of "consistent" tuples.[6]

This approach has several disadvantages that became clear during evaluation. First, the intermediate sets may become much larger than the result set finally produced due to weak join conditions. Second, too much work was done by generating the result set in case the admissible tuples are available.

If the tuple sets are taken in the order implied by the defined order of constraints due to their indices, this may reflect inefficient join conditions and may result in an explosion of the intermediate results. E.g., consider the case that an equi-join is performed among $C_{1,2}, C_{3,4}, C_{4,5,6}$. Taking the order given, the first operation would result in the cross product $C_{1,2} \times C_{3,4}$, whereas the equi-join of $C_{1,2,3,4} \bowtie C_{4,5,6}$ may result in a small subset for $C_{1,2,3,4}$. In practise it showed up that this case arises in many situations and reflects a real performance and space problem.

To eliminate the generation of inefficient and large intermediate sets, a cost-based join strategy generator was developed that calculates an optimizing join strategy by ordering the tuple sets according to a cost function that has been developed by taking the three factors *number of join conditions*, *potential size of the equi-join*, and the *size of the resulting index set in comparison to the "target" index* into account. This cost function takes the size of the intersection of the indices, the size of the sets to join, and the arity of the resulting index (the union

[6] The terminology is taken from [7]: The term "admissible" tuples refers in this context to the set of admissible assignments to the variables of a given constraint. It is given by the definition of the CSP. The primary task of the solver is to compute the set of "consistent" tuples which (are a subset of the set of "admissible" tuples and) are consistent with all constraints connected.

of the two indices) into account.

2.2 Description of Candidate Set Generators

The generators can be classified as *failure generators* or as *refinement generators*, whereas the first class is being used to generate initially a set of tuples, and the second class is used to refine the set of tuples generated, if possible. So, every generator has two exits (failure and success). Each generator has a potential parameter, namely the candidate set to refine. That parameter is obviously only useful for refinement generators and will be ignored in the other class.

The Dynamic Candidate Set Generator (DCSG) configuration makes it possible to change the configuration of candidate set generators easily. So, the candidate set generators can be used in every possible combination and as often as needed. This part provides an overview of the candidate set generators used in our experiments. One of the generators are described in greater detail to give an impression of the way candidate set generators work.

The candidate set generators used are:

1. **Equi-join Of Downlinks**
   This candidate set generator **TEquiJoinOfDownlinks** will take all the downlinks of the inconsistent constraints and performs the equi-join among them. The assumption behind this generator is that the inconsistency may arise due to the intersection of the equi-join of the downlinks and the set of admissible tuples. If this case occurs, this generator will produce a non-empty candidate set from the downlinks. During the tests, this generator has always been used as root generator in order to determine the source of the inconsistency. If the equi-join of the downlinks is not empty, it is obvious that an inconsistency with the set of admissible tuples exists. This is totally different to the case where an inconsistency exists among the downlinks.

2. **Largest Equi-join of Downlinks**
   This generator works very similiar to **Equi-join of Downlinks**. The major difference is that it omits determines sets of downlinks which capture all variables and performs the equi-join among them. The largest set of downlinks with an non-empty equi-join is used as candidate set.

3. **Project and Intersect Uplinks**
   The generator **TProjectAndIntersectUplinks** collects the uplinks of the given constraint and projects the set of admissible tuples to the index of the inconsistent constraint. If no restrictive constraint exists or the number of uplinks is less than two the generator will fail.
   Afterwards the generator will search the largest combination of uplinks with a non-empty intersection of the projected sets of admissible tuples. If such a set exists it will be passed to the registered refinement generator. (The generator does not work as a refinement generator; it ignores a given candidate

set if there is one.)

4. **Projection And Join Of Uplinks**

   The generator **TProjectAndJoinOfUplinks** (cf. Algorithm 5) is a specialization of **TProjectAndIntersectUplinks**. The first step is identical to the former generator, but if there is a non-empty intersection of uplinks the intermediate candidate set is intersected with the set of admissible tuples.

5. **Set Of Admissible Tuples**

   The generator **TAdmissibleTuples** is a trivial candidate set generator that simply returns the set of admissible tuples as a candidate set. This may act as a last chance strategy for hard cases where other generators would probably fail.

6. **Smallest Candidate Set**

   TSmallestCandidateSet is a decision generator that can be used for a simple decision between a given candidate set and the set of admissible tuples, whatever the smaller set is. This refinement generator may be used whenever the candidate set gets much too large and no "intelligent" approach of cutting down this set is appropriate.

7. **Intersection With Admissible Tuples**

   The refinement generator **TIntersectAdmissible** can be used to reduce the size of the result set by intersecting it with the set of admissible tuples. If this intersection is non-empty the intersection will be passed to the registered refinement generator, otherwise the original candidate set is passed to the failure generator. To prevent the candidate set from getting too small again, the refinement generator intersects only if the candidate set size is at least the double of the size of the set of admissible tuples.

8. **Lower Or Upper Half Of The Network**

   This strategy generator just looks at the arity of the inconsistent constraint and calls the failure generator if the constraint is in the lower half and the refinement generator if the constraint is located in the upper half.[7] This may help to guide the generation process in order to produce optimal results in terms of relaxation costs. The idea behind is that if an inconsistency occurs

[7] The arity of the constraints ranges from $1 \ldots n$, where the term "lower half" refers to constraints with an arity in the range $1 \ldots \lfloor \frac{n}{2} \rfloor$, and the term "upper half" refers to constraints with an arity between $\lfloor \frac{n}{2} \rfloor + 1$ and n.

at an early stage it may be better to emphasize on upward compatibility, whereas at later stages the downward compatibility may be appropriate.

9. **Union With Admissible Tuples**

   This refinement generator is used to extend the given candidate set by taking the union among the set of admissible tuples of the inconsistent constraint. It can only work as refinement generator, i.e. it must be called with a non-empty candidate set. If the union does not extend the candidate set the generator will fail, otherwise it is interpreted as success.

1 **function** *TProjectAndJoinOfUplinks (TConstraint* C_I*, TTupleSet* CS*)*
2 **if** $C_I^{ul} = \emptyset$ **then return** failure (C_I, CS)
3 **if** $CS = \emptyset$ **then** $CS \leftarrow C_I^{adm}$
4 $\tilde{CS} \leftarrow \emptyset$
5 $\forall X \in \wp(C_I^{ul})$
6 $\quad CS_T \leftarrow (\bigcap_{J \in X} \pi_I(C_J^{adm})) \cap CS$
7 $\quad$ **if** $|CS_T| > |\tilde{CS}$ **then** $\tilde{CS} \leftarrow CS_T$
8 **if** $\tilde{CS} \neq \emptyset$ **then return** success (C_I, $\tilde{CS}$)
9 $\tilde{CS} \leftarrow \emptyset$
10 $\forall X \in \wp(C_I^{ul})$
11 $\quad CS_T \leftarrow \bigcap_{J \in X} \pi_I(C_J^{adm})$
12 $\quad$ **if** $|CS_T| > |\tilde{CS}$ **then** $\tilde{CS} \leftarrow CS_T$
13 **if** $\tilde{CS} \neq \emptyset$ **then return** success (C_I, $\tilde{CS}$) **else return** failure (C_I,CS)

Algorithm5. candidate set generator *TProjectAndJoinOfUplinks*

3 Experimental Results

The relaxation algorithm and moreover the quality of solutions depend on several parameters. For our experimental results we have chosen some parameters which have a significant impact on the behavior of the algorithm. The results have been gained during over 6000 runs of the solver.

The tests were performed using a test-case generator. A short description of the parameters of the test-case generator are given before the results are discussed.

1. **Problem size** n
   The problem size is surely the most important factor of a CSP, since the number of possible constraints exponentially grows as a function of n. So, the problem size is a scaling factor for the whole problem.

2. **Domain size** d
   The domain size d also is a fundamental parameter for the basic problem. It determines the number of legal assignments to the variables. This is a slight simplification here, since each variable can be assigned a different domain. Nevertheless, the possibility to do so is in no way meaningful for test-case generation. But it is clear that increasing the domain size exponentially increases the size of the cross product (of the tuples) and thereby increases the search space for the solver.

3. **Restrictive Constraints**
   The number of possible constraints in CSPs may be overwhelming. Consider the case when you have a problem size of 10. The CSP will permit the definition of 1,023 ($= 2^{10} - 2^0$) constraints. In practical applications usually the opposite occurs; you have many variables, but only very few constraints that are restrictive, i.e. you have a sparse constraint satisfaction network. The algorithm used here offers optimized support for such cases, since it will not generate or even traverse the undefined (or non-restrictive) constraints. So, the number of really restrictive constraints is a major factor in the evaluation of the algorithm, since it scales along the complexity of sparse networks to high density networks.

4. **Number Of Inconsistent Constraints**
   The number of inconsistent constraints is measured relative to the number of restrictive constraints. It is used whenever a decision must be made during test-case generation whether a constraint should get consistent or inconsistent at runtime.

5. **Number Of Global Solutions**
   While the number of restrictive constraints somehow relates to the problem size, the number of global solutions relates to the domain size. It specifies the size of the subset of the cross-product over all domains $d_1 \times d_2 \times \ldots \times d_n$. Since the test-bed generator will hopefully produce an inconsistent CSP, it is a parameter that affects the generation indirectly. First of all, a set of virtually globally consistent solutions is being generated. This set is divided into a number of different partitions. For every restrictive constraint within the network a set of partitions is randomly chosen to form the set of admissible tuples, whereas the union of all tuples of the chosen partitions projected to the index set of the constraint in question is the set of admissible tuples.

6. **Number Of Partitions**
   The number of partitions is directly related to the set of virtually consistent tuples defined by the *number of global solutions*. The set of global solutions is divided into the given number of partitions. Each partition is assigned a probability factor, such that the sum of all probability factors is exactly 1. It is obvious that the number of partitions cannot be greater than the number

of global solutions.

7. **Randomization Factor**
   The randomization factor is a pair of values and determines the minimal and maximal number of additionally generated tuples for each constraint.

3.1 Consistency And Resolution Performance

Interesting properties of an algorithm are always its time and space requirements. Since global consistency is NP-complete in general it is interesting to see whether or not the algorithm also shows an exponential behaviour in practise — especially, since the algorithm takes only the constraints into account that are really restrictive.

Figure 5 shows the time used for establishing consistency as a function of the parameters *constraints* and *inconsistent*. The performance of the solver is quite good-looking and the average time required to solve a single constraint is about 0.01 sec. This behaviour can be explained with the optimized handling of equi-joins, as it is described in Section 2.1.

Concluding, it can be stated that the consistency times confirm the algorithm being in good shape in general — nearly every constraint could be computed within a tenth of a second. The most influencing parameter on the times turned out to be the parameters *Number Of Global Solutions* and *Randomization Factor*, whereas the parameter *Number Of Partitions* shows a systematic influence only for $n = 10$.

3.2 Resolution Quality

The most interesting question with regard to a relaxation solver is of course how good are the relaxations it generates with regard to the given cost function.

Figure 6 and Figure 7 illustrate the candidate-set generator usage and performance.

Please note that the diagrams have logarithmically scaled y-axes; in some diagrams some values are missing. This is due to the fact that in such diagrams 0-values cannot be plotted. But on the other hand, it does not make sense to use linear scaled y-axes, since the values vary too much.

Figure 6 shows the number of calls to each candidate-set generator. Every generator has been called, except the generator *epai* (Extended Projection and Intersection). The candidate-set generator *ejod* has been called for every inconsistency, since it acts as root generator. Remarkable is the curve of the candidate-set generator *adm* (the set of admissible tuples) which acts as a last chance handler in the DCSG configuration. The usage of this generator increases with increasing number of constraints and inconsistent constraints.

Remarkable is the kind of inconsistencies the solver had to process; almost every call to *ejod* leads to a call to *lejod* which was called in the case of failure of the former one. The major part of inconsistencies has not been an incompatibility

of the equi-join of the downlinks and the admissible tuples itself; even the equi-join has been empty. This case is much harder to handle than the other one.

Figure 7 shows the time usage of the different candidate set generators. The most expensive generator is *lejod*. Interesting is that the average times are almost identical to the maximal times. This implies that there are no "easy" cases. The variance of the generator *ejod* is much greater. All other generators perform under 0.1 sec. in the average case.

Concluding this group it becomes obvious that most of the time used for candidate-set generation has been consumed by the generators *lejod* and *ejod*. Although this generators have been optimized several times, since they turned out to be the major factors for resolution, they are still slow. On the other hand, the counterparts of *lejod* for the uplinks, namely *piou* and *pjou*, do not show such a behaviour. This is surely related to the fact that projection is much less expensive in contrast to the equi-join operation.

There are two possible sources for optimizations implied by the results; first, equi-join operations should be delayed as long as possible and reasonable in terms of candidate-set quality, and second, expensive refinements do not make sense and should be omitted.

But nevertheless, the time needed to produce a candidate set is one thing, the quality of the candidate set is the other.

Figure 8 illustrates the costs depending on the number of constraints and percentage of inconsistent constraints. The costs of each relaxation is likely the same as calculated for the test-case before. It increases with the number of constraints affected. The solver found most of the global solutions that were given by the CSP generator. A lot of solutions with very high relaxation costs have been filtered out, but surprisingly, the solver found in some cases solutions with less than 5% of the costs of all other solutions.

The in-depth analysis of the experimental results gave us the affirmation that multi-level candidate-set generation is a good way to resolve inconsistencies and to provide high quality solution for such problems.

3.3 Conclusion

The performance of the solving process is good, whereas the performance of inconsistency resolution is in some cases poor. But, in the average case it shows a reasonable behaviour which may be influenced by the candidate-set generator configuration or even by the generators themselves.

Inconsistency resolution is not only a question of efficiency, the other key factor is relaxation quality. In this regard the tested candidate-set generator configuration produced qualitatively very good results, sometimes with the drawback of long computation times.

Nevertheless, the results are quite promising for the relaxation approach pursued here.

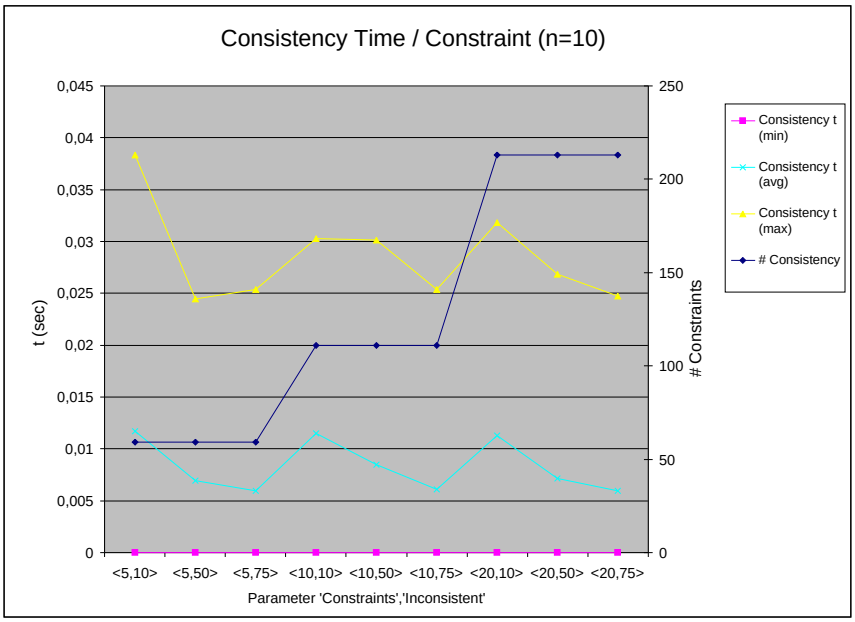

Fig. 5. parameters *Constraints*, *Inconsistent* (problem size n=10)

4 Discussion

4.1 Related Work

[1] interactively computes consistent subsets of user wishes in an ATMS-like manner. [4] utilizes CLP(FD) for (partial) arc-consistency, avoiding an ATMS framework. [2] also deals with local consistency. [3] presents an interesting interactive approach for binary constraints where relaxation is done via additional "tradeoff" constraints. [11] does filtering of inconsistent values in a cost minimization framework (using lower bounds), accepting a non-optimal output (due to the complexity). [12] exploits preferences to retract constraints, in interaction with the user. [10] copes with the MAX-CSP (with entire constraints). [9] treats constraint optimization. The focus of our work, however, is the search for the source of global inconsistency dealing with fine-grained tuples.

4.2 Final Remarks

We have presented a sensible framework for constraint relaxation. It first tries to compute the globally consistent solution by just exploiting the explicitly given constraints (without artificially "synthesizing" redundant constraints). In case of an inconsistency it proposes conflict candidates to relax only those constraints involved in the inconsistency. Besides many other approaches this one works on

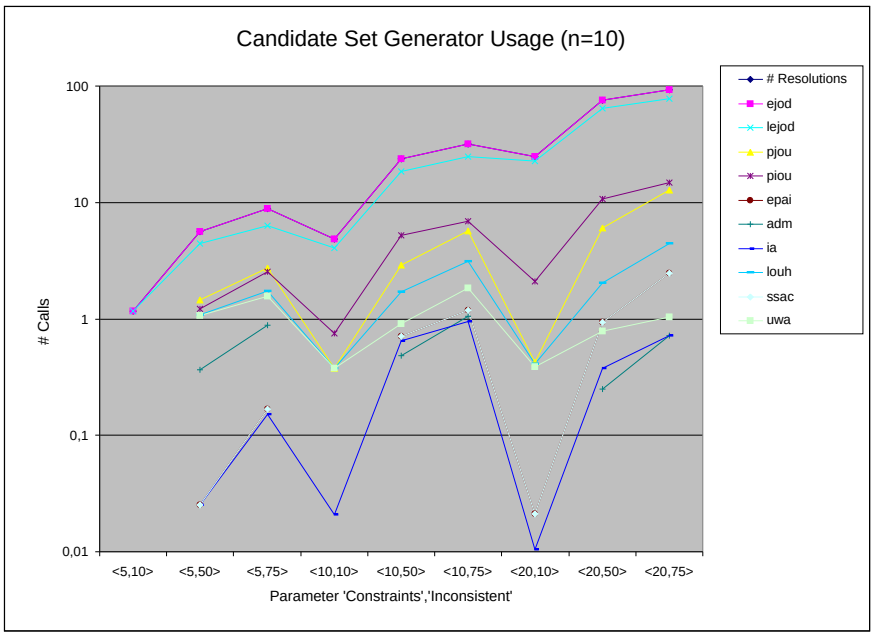

Fig. 6. number of calls to each CS-generator (problem size n=10)

a tuple-level which ensures relaxations with relatively low degrees of violation to the original problem. Moreover, by choosing one relaxation from the set of calculated relaxations, at least one globally consistent solution becomes available. (It is trivial that always a relaxation exists where all constraints have to be modified by adding an additional tuple. But this case is impossible to occur since there always exists at least one solution that remains at least one constraint unchanged.) The algorithm produces a minimal cost relaxation with respect to the tuples yielded by the generators. It is really encouraging that the heuristic generator functions well. The current work illustrates that constraint relaxation based on candidate sets generated heuristically may work efficiently producing quite promising results in terms of "relaxation costs". (The approach of inconsistency resolution is a general one that does not exploit any domain specific knowledge; any usage of such knowledge may offer additional potentials for further optimizations and quality improvements.) The algorithm was able to resolve all inconsistencies in reasonable time.[8] The candidate set generator in the configuration actually used for the tests proved to work effectively; the solver was able to find a set of relaxations minimal in the terms of the given cost function. In some circumstances it found some solutions with a fraction of the costs all other relaxations have.

[8] No test-case took more than 5 hours of computation time. In most cases the computation took less than 10 min. per test-case — without special tuning.

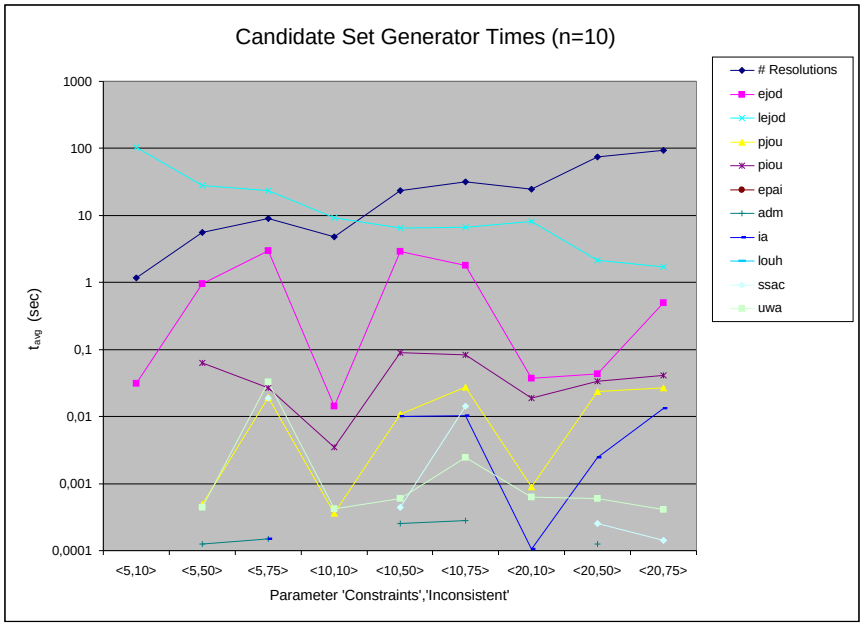

Fig. 7. average time for each CS-generator (problem size n=10)

Constraint relaxation and inconsistency resolution is an important aspect in many areas where combinatorial problems are to be solved. This work may offer a practicable way to solve such problems in real applications. Additionally, the approach provides many degrees of freedom to integrate domain specific knowledge for further enhancements.

References

1. Jérôme Amilhastre, Hélène Fargier, and Pierre Marquis. Consistency restoration and explanations in dynamic CSPs—Application to configuration. *Artificial Intelligence*, 135(1–2):199–234, Elsevier Science B.V., 2002
2. S. Bistarelli, U. Montanari, F. Rossi, T. Schiex, G. Verfaillie, and H. Fargier. Semiring-Based CSPs and Valued CSPs: Frameworks, Properties, and Comparison. *Constraints*, 4(3):199–240, Kluwer Academic Publishers, 1999
3. Eugene C. Freuder and Barry O'Sullivan. Modeling and Generating Tradeoffs for Constraint-Based Configuration. *UICS'01: User-Interaction in Constraint Satisfaction*, CP 2001 Workshop Proceedings, pp. 43–57, Paphos, Cyprus, 2001
4. Yan Georget, Philippe Codognet, and Francesca Rossi. Constraint Retraction in CLP(FD): Formal Framework and Performance Results. *Constraints*, 4(1):5–42, Kluwer Academic Publishers, 1999
5. Walter Hower. On Conflict Resolution in Inconsistent Constraint Networks. Master's thesis ("Diplomarbeit"), Fachbereich Informatik, Universität Kaiserslautern,

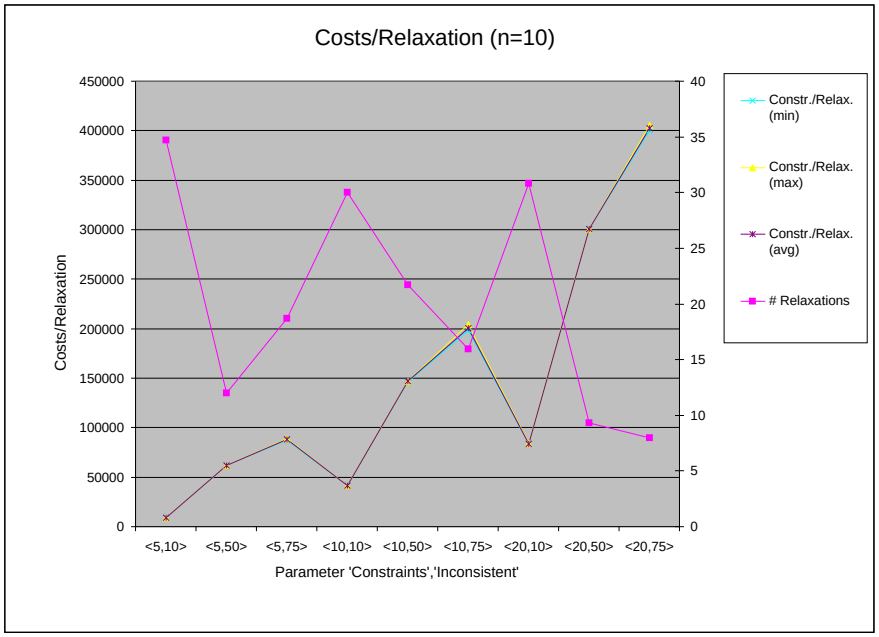

Fig. 8. costs per relaxation (problem size n=10)

Germany, 1989

6. Walter Hower. Revisiting global constraint satisfaction. *Information Processing Letters*, 66(1):41–48, Elsevier Science Publishers, B.V., Amsterdam, The Netherlands, 1998 (pre-print: Global constraint satisfaction revisited. Technical Report 97-02, Department of Computer Science, University College Cork, National University of Ireland, 1997)
7. Walter Hower and Stephan Jacobi. Fine-grained conflict resolution in constraint satisfaction problems. *Journal of Experimental & Theoretical Artificial Intelligence*, 10(1):37–47, Taylor & Francis, London, UK, 1998
8. Stephan Jacobi. Fine-grained constraint relaxation. Master's thesis ("Diplomarbeit"), Fachbereich Informatik, Universität Koblenz-Landau, Germany, 1999
9. Javier Larrosa and Rina Dechter. Boosting Search with Variable Elimination in Constraint Optimization and Constraint Satisfaction Problems. *Constraints*, Kluwer Academic Publishers, 2002 (in press)
10. Javier Larrosa and Pedro Meseguer. Partition-Based Lower Bound for Max-CSP. *Constraints*, Kluwer Academic Publishers, 2002 (in press)
11. Thierry Petit, Jean-Charles Régin, and Christian Bessière. Specific Filtering Algorithms for Over-Constrained Problems. In T. Walsh (Ed.): *Principles and Practice of Constraint Programming – CP 2001*, 7th International Conference, Paphos, Cyprus, Proceedings, pp. 451–463, LNCS 2239, Springer-Verlag, Berlin/Heidelberg, 2001
12. Yan Qu and Stephen Beale. Cooperative Resolution of Over-Constrained Information Requests. *Constraints*, 7(1):29–47, Kluwer Academic Publishers, 2002

This article was processed using the LaTeX macro package with LLNCS style