Towards an Optimal Approach to Soft Constraint Problems

Martine Ceberio and Vladik Kreinovich
Department of Computer Science
University of Texas at El Paso, 500 W. University
El Paso, TX 79968, USA, {mceberio,vladik}@cs.utep.edu

Abstract - In traditional constraint satisfaction, constraints are "hard" in the sense that we need to satisfy them all. In many practical situations, however, constraints are "soft" in the sense that if we are unable to satisfy some of them, the corresponding solution is still practically useful. In such situations, it is desirable to satisfy as many high-priority constraints as possible. In this paper, we describe an optimal algorithm for solving the corresponding soft constraint problem.

Keywords—**soft constraints, optimal algorithm**

I. WHY CONSTRAINTS?

In many areas of science and engineering, we are interested in solving design and control problems. Usually, in these problems, the users describe several *constraints* that the desired design or control must satisfy, and our objective is to find a design (correspondingly, a control) that satisfies all these constraints.

In mathematical terms, a design or a control can be usually represented by the values of the relevant numerical parameters $x = (x_1, \ldots, x_n)$. For example, an airplane design can be described in terms of the geometric parameters of the plane, the thickness of the plates that form the airplane's skin, the weight and power of the engine, etc. A typical constraint describes a limitation on some characteristics of this design: e.g., the airplane's speed must exceed a certain threshold, its fuel use must not exceed a certain amount, and the overall cost must be within given limits. Each of the corresponding characteristics y (speed, fuel use, etc.) can be uniquely determined by the design $y = f(x_1, \ldots, x_n)$ for some computable function f. Thus, each constraint can be described as either an inequality of the type $f(x_1, \ldots, x_n) \le y_0$ or $f(x_1, \ldots, x_n) \ge y_0$ – or as an equality $f(x_1, \ldots, x_n) = y_0$ (for example, if we want to design the fastest airplane within a fixed cost).

In many real-life situations, the constraints are consistent, i.e., there exist designs that satisfy all these constraints. In this case, it is desirable to find a design that satisfies a given finite set of inequality/equality-type constraints. The corresponding problem is called the problem of *constraint satisfaction*.

Often, in such situations, there are many different designs that satisfy the given constraints. In this case, it is desirable to select one of these designs. Users can often describe their preference in terms of an *objective function* $g(x_1, \ldots, x_n)$ whose value should be made as large as possible. In such situations, we are interested in maximizing the given function $g(x_1, \ldots, x_n)$ under the given constraints. This problem is called a *constrained optimization problem*.

In general, constraint satisfaction and constraint optimization problems are NP-hard; see, e.g., [KRE 97], [VAV 91]. In practice, however, there exist many efficient tools for solving these problems, including numerous efficient tools that provide validated solution to these problems.

II. WHAT ARE SOFT CONSTRAINTS?

In many practical situations, if we formulate all the users' desired as constraints, we often end up with an inconsistent set of constraints. For example, a user may want to design a plane that is as fast and as fuel-efficient as the existing Airbus or Boeing planes, but that will decrease the noise level to 0.

The reason why constraints are inconsistent is that while some of these constraints are absolute requirements (e.g., safety constraints for a plain), other constraints are simply recommendations, desires, that a user wants to be implemented if possible – but that can be dismissed if it is not possible to satisfy them. Such "not required" constraints are called *soft constraints*.

Soft constraints are an important research topic; see, e.g., the proceedings of the latest conference [PRO 04] and references therein. Our own work in soft constraints is described, e.g., in [BEN 03], [CEB 03].

III. PRIORITY APPROACH TO SOFT CONSTRAINTS: A BRIEF DESCRIPTION

The main idea behind this approach is that if we cannot satisfy all the constraints, we should at least satisfy as many constraints as possible.

One of the natural approaches to soft constraints is therefore to ask the user to *prioritize* their constraints, from the absolutely required to the less required. Once the user sorted all his/her constraints from the most required to the least required, into a sequence $C_1 \succ C_2 \succ \ldots \succ C_n$, we try to find the largest possible value $k = k_{\rm opt}$ for which all the constraints $C_1, C_2, \ldots, C_k$ are still consistent.

IV. PRIORITY APPROACH TO SOFT CONSTRAINTS: A COMPUTATIONAL QUESTION

The existing constraint satisfaction tools enable us, given constraints, either to find a design that satisfies these constraints, or to conclude that the given constraints are inconsistent.

There are many different ways how we can use one of these tools to solve soft constraint problems. For example, we can sequentially apply this tool to constraints sets $\{C_1\}$, $\{C_1, C_2\}$, ..., until we find the first value l for which the corresponding set is inconsistent. Then, the previous value $k_{\rm opt} = l - 1$ is the desired largest k, and the corresponding design is the desired one.

A (potential) problem with this approach is that when the number of constraints is large, the constraint satisfaction tools take a long time to run, so if we have to run the tool for many different sets, we make the process even slower.

Alternatively, we can, e.g., use bisection to find the desired value $k_{\rm opt}$. At each stage of this iterative method, we have an interval $[k^-, k^+]$ that is guaranteed to contain this desired value, i.e., for which the system of the first k^- constraints is consistent while the system of the first k^+ constraints is inconsistent.

Initially, $k^- = 0$ (if we have no constraints, then, of course, the problem is consistent) and $k^+ = n$ (we cannot satisfy all constraints: this is the definition of the soft constraint problem).

Then, sequentially, we test whether the midpoint $k_m \stackrel{\rm def}{=} \lfloor (k^- + k^+)/2 \rfloor$ of the interval is consistent or not, and, depending on the result of this test, replace the original interval with a half-size one: $[k^-, k_m]$ or $[k_m, k^+]$. On each iteration, the interval decreases in half, so after $\log_2(n)$ iteration, we get the interval of width 1 – i.e., we get the desired value $k_{\rm opt}$. The advantage of this technique is that we need fewer iterations to find k, but the disadvantage is that some of these iterations may require analyzing much more constraints – n as opposed to $k \ll n$ – and thus, may take much longer.

Other methods of finding the optimal k are possible. The question is: which of the methods is optimal?

Of course, different methods may be optimal for different cases. What we would like to do is find the methods for which some reasonably defined "worst-case" computation time is optimal. Let us formulate this problem in precise terms.

V. PRIORITY APPROACH TO SOFT CONSTRAINTS: TOWARD FORMALIZING THE COMPUTATIONAL QUESTION

We would like to describe the best strategy of finding $k_{\rm opt}$. In order to do that, we need to explain what we mean by a strategy.

All we can do is check, for several values k_i, whether the corresponding constraint sets are consistent. If we know that constraints sets are consistent for the values $k_1, \ldots, k_m$ and inconsistent for $k = k'_1, \ldots, k'_p$, then this knowledge is equivalent to knowing that the constraints are consistent

for $k^- \stackrel{\rm def}{=} \max(k_1, \ldots, k_m)$ and inconsistent for $k^+ \stackrel{\rm def}{=} \min(k'_1, \ldots, k'_p)$.

In other words, at each stage of the computations, we know that k is within an interval $[k^-, k^+]$ – i.e., that:

- the set of the first k^- inequalities is consistent, while
- the set of the first k^+ inequalities is inconsistent.

If $k^+ = k^- + 1$, then we know that k^- is the desired largest value for which constraints are consistent. If $k^+ > k^- + 1$, then, to continue looking for $k_{\rm opt}$, we must select a value within the interval $[k^-, k^+]$ that we will check next. This value can, in general, depend on the number of a step.

So, in general, we can define the *method* as follows:

Definition 1. *A* method *is a mapping that maps each pair* (I, s)*, where:*

- I *is an integer-valued interval* $[k^-, k^+]$*, where* $0 \le k^- < k^+ \le n$ *and* $k^+ > k^- + 1$*, and*
- s *is a positive integer (= number of step)*

into an integer $k_{\rm next}$ *from the open interval* (k^-, k^+).

One can easily see that each method generates a strategy that eventually leads to the desired value $k_{\rm opt}$: once we check whether the system of constraints is consistent or not, we get a new interval $[k^-, k_{\rm next}]$ or $[k_{\rm next}, k^+]$. For example:

- the sequential search S corresponds to the method in which we select $k_{\rm next} = k^- + 1$;
- bisection B corresponds to the method in which we select $k_{\rm next} = \lfloor (k^- + k^+)/2 \rfloor$.

How can we estimate the worst-case complexity of each method? As we have mentioned, the problem of constraint satisfaction is known to be NP-hard. This means, crudely speaking, that the computation time grows exponentially with the number of constraints. For most NP-hard problems such propositional satisfiability, the actual worst-case complexity of the known algorithms grows as 2^n. For Boolean-type constraints, it is therefore reasonable to assume that the computational complexity of checking the consistency of the system of k constraints is proportional to 2^k.

Similarly, in a more general case of finite constraints in which each variable has $p \ge 2$ different values, we need, in the worst case, p^k checks to check all possible values of these constraint variables. It is therefore reasonable to assume that in general, the computational complexity of checking the consistency of the system of k constraints grows as p^k for some $p \geq 2$.

For every method M and for every value k of $k_{\rm opt}$, we can now define the overall time $T_M(k)$ that this method spends on this case as the sum of the values $p^{k'}$ for all k' for which this method checks consistency.

For each value of $k_{\rm opt}$, the ideal case is when someone's intuition informs us of the correct value $k_{\rm opt}$. In this case, we do not need to check for too many different values k, all we need to do is check whether indeed the system of first k constraints is consistent while the system of the first $k+1$ constraints is inconsistent. In this case, the overall expenses are equal to $p^k + p^{k+1}$.

In real life, no such intuition is available, so we have to test more values k and thus, spend more time. The smaller the "overhead" in comparison with the ideal case, the better. We can therefore define the *overhead* $O_p(M)$ of a method M as

$$O_p(M) \stackrel{\rm def}{=} \max_k \frac{T_M(k)}{p^k + p^{k+1}}.$$

Let $p \geq 2$ be fixed. We say that a method M is *optimal* for this p if out of all methods, it has the smallest possible overhead: $O_p(M_{\rm opt}) = \min\limits_M O_p(M)$. Now, we are ready to describe our main result:

VI. MAIN RESULT

Theorem. *For every $p \geq 2$, the sequential search method S is optimal.*

Proof. For sequential search, if the actual value of $k_{\rm opt}$ is k, we check consistency of sets consisting of 1, 2, …, k, and $k+1$ constraints. As a result, the overall time is equal to

$$T_S(k) = p + p^2 + \ldots + p^{k+1}.$$

Here,

$$T_M(k) = p^{k+1} \cdot (1 + p^{-1} + p^{-2} + \ldots + p^{-k}) <$$

$$p^{k+1} \cdot (1 + p^{-1} + p^{-2} + \ldots) = \frac{p^{k+1}}{(1 - p^{-1})}.$$

Since $p^k + p^{k+1} = p^{k+1} \cdot (1 + p^{-1})$, we conclude that

$$O_S(p) < \frac{1}{(1 - p^{-1}) \cdot (1 + p^{-1})}.$$

Let us now show that every other method has a larger overhead. Indeed, the specific feature of S is that in S, out of all possible integers between k^- and k^+, we always select $k_{\rm next} = k^- + 1$ as the next value to check. This means that in every other method $M \neq S$, there exists an interval in which we select a value $k_{\rm next} > k^- + 2$. In this case, if the actual value $k_{\rm opt}$ is equal to the corresponding k^-, then in this method, we check both k and $\geq k+2$. Later on, we still need to check the value $k+1$ – to make sure that k is indeed the largest consistent value. Thus, for this k, the method M spends at least time $T_M(k) \geq p^k + p^{k+1} + p^{k+2}$. This lower bound can be described as $T_M(k) \geq p^{k+1} \cdot (p + 1 + p^{-1})$, hence

$$O_M(k) = \frac{T_M(k)}{p^{k+1} \cdot (1 + p^{-1})} \geq \frac{p + 1 + p^{-1}}{1 + p^{-1}}.$$

To complete our proof, we must show that

$$\frac{p + 1 + p^{-1}}{1 + p^{-1}} \geq \frac{1}{(1 - p^{-1}) \cdot (1 + p^{-1})}.$$

Multiplying both sides of this inequality by the denominator of the right-hand side, we get an equivalent inequality $p - p^{-2} \geq 1$, i.e., equivalently, that $p^3 - p^2 - 1 \geq 0$. One can show that the equation $p^3 - p^2 - 1 = 0$ has a solution $p_0 \approx 1.47$, and that for $p \geq p_0$, its left-hand side is an increasing function – since its derivative is $3p^2 - 2p = (3p - 2) \cdot p > 0$. Thus, $p^3 - p^2 - 1 > 0$ for all $p \geq p_0$ – in particular, for all $p \geq 2$. Q.E.D.

Comment. From the purely mathematical viewpoint, the above problem is very similar to the following planning problem: the existing AI-based planners either find a plan of given length k or conclude that such a plan is impossible. Based on such a planner, what is the best way to find the shortest plan?

A partial solution to this problem is given in the paper [TRE 01]; in this paper, we also consider the cases when $p < 2$ and when, instead of optimizing the worst-case overhead, we optimize the average-case overhead. It is desirable to extend our soft constraint results to similar cases.

ACKNOWLEDGMENTS

This work was supported in part by NASA under cooperative agreement NCC5-209, by NSF grants EAR-0112968, EAR-0225670, and EIA-0321328, and by NIH grant 3T34GM008048-20S1. This research was partly done when M.C. was a Visiting Researcher at NII, Tokyo, Japan.

The authors want to thank Seetharami R. Seelam for his help.

REFERENCES

[BEN 03] F. Benhamou and M. Ceberio, "Soft Constraints: A Unifying Framework applied to Continuous Soft Constraints", *Proceedings of the ERCIM/CoLogNET Workshop*, 2003.

[CEB 03] M. Ceberio, *Under- and over-constrained numerical CSP: symbolic tools and soft constraints*, Ph.D. Dissertation, Nantes, France, 2003.

[CEB 05] M. Ceberio, O. Kosheleva, V. Kreinovich, G. R. Keller, R. Araiza, M. Averill, and G. Xiang, "Data Processing in the Presence of Interval Uncertainty and Erroneous Measurements: Practical Problems, Results, Challenges", *Abstracts of the Second Scandinavian Workshop on Interval Methods and Their Applications*, Lyngby, Denmark, August 25–27, 2005 (to appear).

[COR 01] Th. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms*, MIT Press, Cambridge, MA, 2001.

[JAU 01] L. Jaulin, M. Keiffer, O. Didrit, and E. Walter, *Applied Interval Analysis*, Springer-Verlag, London, 2001.

[KRE 97] V. Kreinovich, A. Lakeyev, J. Rohn, and P. Kahl, *Computational complexity and feasibility of data processing and interval computations*, Kluwer, Dordrecht, 1997.

[PAP 98] C. H. Papadimitriou and K. Steiglitz, *Combinatorial Optimization: Algorithms and Complexity*, Dover Publications, Inc., Mineola, New York, 1998.

[PRO 04] *Proceedings of the 6th International Workshop on Preferences and Soft Constraints*, Held in conjunction with *10th International Conference on Principles and Practice of Constraint Programming CP'2004*, September 27–October 1, 2004, Toronto, Canada.

[TRE 01] R. A. Trejo, J. Galloway, C. Sachar, V. Kreinovich, C. Baral, and L. C. Tuan, "From Planning to Searching for the Shortest Plan: An Optimal Transition", *International Journal of Uncertainty, Fuzziness, Knowledge-Based Systems (IJUFKS)*, 2001, Vol. 9, No. 6, pp. 827–838.

[VAV 91] S. A. Vavasis, *Nonlinear Optimization: Complexity Issues*, Oxford University Press, New York, 1991.

APPENDIX: WHAT IF CONSTRAINTS ARE NOT PRIORITIZED?

A.1. Informal Introduction to the Problem

What if our constraints are not prioritized? In this case, since all the constraints are equally important, a natural idea is to satisfy as many constraints as possible.

It turns out that the corresponding problem becomes computationally difficult (NP-hard) even in when all the constraints are of the simplest possible type – linear equations (this result is announced in [CEB 05]).

Let us formulate this problem in precise terms.

A.2. Formulation of the Problem in Precise Terms

Let a number $\varepsilon \in (0,1)$ be given.

We are given a finite set of N linear constraints with n unknowns $x_1, \ldots, x_n$:

$$\sum_{j=1}^{n} a_{ij} \cdot x_j = b_i, \;\; i = 1, \ldots, N$$

(where a_{ij} and b_i are given).

Our objective is to check whether it is possible to select, out of these N constraints, a subset of $\geq (1-\varepsilon) \cdot N$ constraints which is consistent (i.e., for which there exist values $x_1, \ldots, x_n$ that satisfy all selected constraints).

We will prove that this problem is NP-hard.

A.3. Proof of NP-hardness

To prove NP-hardness of our problem, we will reduce a known NP-hard problem to the problem whose NP-hardness we try to prove: namely, to the above-described soft constraint problem for non-prioritized constraints.

Specifically, we will reduce, to our problem, the following *subset sum* problem [KRE 97], [PAP 98] that is known to be NP-hard:

- Given:
  - n positive integers $s_1, \ldots, s_n$ and
  - an integer $s > 0$,
- check whether it is possible to find a subset of this set of integers whose sum is equal to exactly s.

For each i, we can take $x_i = 0$ if we do not include the i-th integer in the subset, and $x_i = 1$ if we do. Then the subset problem takes the following form: check whether there exist values $x_i \in \{0, 1\}$ for which

$$\sum_{i=1}^{n} s_i \cdot x_i = s.$$

We will reduce each instance of this problem to the corresponding soft constraints problem. Specifically, we take $N = n/\varepsilon$, and the following N constraints:

- n constraints $x_1 = 0, \ldots, x_n = 0$;
- n constraints $x_1 = 1, \ldots, x_n = 1$;
- $N - 2n$ constraints $\sum s_i \cdot x_i = s$.

Let us show that this soft constraint problem has a solution if and only if the original instance of the subset problem has a solution.

Indeed, if the subset problem has a solution, then all $N-2n$ constraints $\sum s_i \cdot x_i = s$ are satisfied, and for each i, one of the two constraints $x_i = 0$ and $x_i = 1$ is satisfied. Thus, overall, we satisfy $(N - 2n) + n = N - n$ constraints. Since $N = n/\varepsilon$, we have $N - n = N \cdot (1 - \varepsilon)$, so this solution satisfies $N \cdot (1 - \varepsilon)$ constraints.

Vice versa, let us assume that we have a solution that satisfies at least $N \cdot (1 - \varepsilon) = N - n$ linear constraints. For every i, we can have either $x_i = 0$ or $x_i = 1$ (or none), but not both constraints. Thus, no matter what set of values x_i we select, at least n constraints will be not satisfied. Thus, the fact that $\geq N-n$ linear constraints out of N are satisfied means that all the remaining constraints are actually satisfied, i.e.:

- for every i from 1 to n, either $x_i = 1$ or $x_i = 1$, and
- all $N-2n$ constraints $\sum s_i \cdot x_i = s$ are satisfied.

Thus, we have n values $x_i \in \{0,1\}$ for which $\sum s_i \cdot x_i = s$ – i.e., we have a solution to the original subset problem.

This reduction proves that the soft constraint problem for non-prioritized constraints is indeed NP-hard – even in the case of linear constraints.

A.4. How Can We Actually Solve the Soft Constraint Problem for Non-Prioritized Constraints: Idea

Traditional way to solving a constraint satisfaction problem is to use constraint propagation; see, e.g., [JAU 01].

Namely, we start with the intervals $[\underline{x}_1, \overline{x}_1], \ldots, [\underline{x}_n, \overline{x}_n]$ that are guaranteed to contain the actual values of the unknowns $x_1, \ldots, x_n$.

On each iteration, we select a variable x_i and a constraint $f_j(x_1, \ldots, x_n) = 0$, and use the known intervals $[\underline{x}_k, \overline{x}_k]$ for all other variables x_k $(k \neq i)$ to narrow down the interval $[\underline{x}_i, \overline{x}_i]$ into the new interval

$$\mathbf{x}_i^{(j)} = [\underline{x}_i^{(j)}, \overline{x}_i^{(j)}] \stackrel{\text{def}}{=}$$

$$\{x_i : x_i \in [\underline{x}_i, \overline{x}_i] \,\&$$

$$f_j(x_1, \ldots, x_{i-1}, x_i, x_{i+1}, \ldots, x_n) = 0$$

$$\text{for some } x_k \in [\underline{x}_k, \overline{x}_k]\}.$$

Once can easily see that, since the j-th constraint is satisfied, this new interval is guaranteed to contain the actual (unknown) value of x_i.

On each iteration, constraints and variables are selected in such a way that eventually, we use all the constraints and narrow down all the variables.

If the process stalls, we bisect the interval for one the variables into two and try to decrease both resulting half-boxes.

This process cannot be directly applied if we are not 100% sure that all N constraints are valid – and instead, we only know that at least $\geq (1-\varepsilon) \cdot N$ constraints are satisfied (but we do not know which of the original N constraints are satisfied).

Instead, we propose the following modification of this standard constraint propagation procedure. On each iteration, once we selected a variable x_i, instead of selecting a *single* constraint, we try *all* N constraints, and get N resulting intervals $[\underline{x}_i^{(j)}, \overline{x}_i^{(j)}]$. If j-th constraint is satisfied, then $\underline{x}_i^{(j)} \leq x_i \leq \overline{x}_i^{(j)}$ – in particular, $x_i \leq \overline{x}_i^{(j)}$.

We know that at least $N \cdot (1-\varepsilon)$ constraints are satisfied, hence $x_i \leq \overline{x}_i^{(j)}$ for at least $N \cdot (1-\varepsilon)$ different values j.

Hence, if we sort all N upper endpoints $\overline{x}_i^{(j)}$ $(1 \leq j \leq N)$ into an increasing sequence $u_1 \leq u_2 \leq \ldots \leq u_N$, then we can guarantee that x_i is smaller than (or equal to) at least $N \cdot (1-\varepsilon)$ terms in this sequence – i.e., that $x_i \leq u_{N \cdot \varepsilon}$.

Similarly, if we sort all the lower endpoints $\underline{x}_i^{(j)}$ $(1 \leq j \leq N)$ into a decreasing sequence $l_1 \geq l_2 \geq \ldots \geq l_N$, we conclude that $x_i \geq l_{N \cdot \varepsilon}$.

Thus, we can guarantee that $x_i \in [l_{N \cdot \varepsilon}, u_{N \cdot \varepsilon}]$. So, we can take the interval $[l_{N \cdot \varepsilon}, u_{N \cdot \varepsilon}]$ as the desired narrowing of $[\underline{x}_i, \overline{x}_i]$. As a result, we arrive at the following algorithm.

A.5. How Can We Actually Solve the Soft Constraint Problem for Non-Prioritized Constraints: Algorithm

We start with the intervals $[\underline{x}_1, \overline{x}_1], \ldots, [\underline{x}_n, \overline{x}_n]$ that contain the unknowns $x_1, \ldots, x_n$.

On each iteration, we select a variable x_i. For this variable and for each of N constraints, we compute the corresponding interval $[\underline{x}_i^{(j)}, \overline{x}_i^{(j)}]$. Then:

- we sort all N upper endpoints $\overline{x}_i^{(j)}$ $(1 \leq j \leq N)$ into an increasing sequence $u_1 \leq u_2 \leq \ldots \leq u_N$,
- we sort all N lower endpoints $\underline{x}_i^{(j)}$ $(1 \leq j \leq N)$ into a decreasing sequence $l_1 \geq l_2 \geq \ldots \geq l_N$, and
- we take $[l_{N \cdot \varepsilon}, u_{N \cdot \varepsilon}]$ as the new (narrower) interval for x_i.

Similar to the traditional constraint propagation case (of "hard" constraints), we select the variables, e.g., in a cyclic order – $x_1, x_2, \ldots, x_n$, then again $x_1, x_2, \ldots, x_n$, etc. – so that we narrow down the intervals corresponding to *all* the variables.

If the process stalls, we bisect the interval for one the variables into two and try to decrease both resulting half-boxes.

Comment. Out of the entire sorted sequence u_i, we are only interested in a single value $u_{N \cdot \varepsilon}$. It is known (see, e.g., [COR 01]) that there exist special algorithms for producing such a single value – algorithms which are much faster than sorting.