Noname manuscript No.
(will be inserted by the editor)

Qualitative Soft Constraints

Alexander Schiendorfer · **Alexander Knapp** ·
Wolfgang Reif

Received: date / Accepted: date

Abstract Over-constrained problems are ubiquitous in industrial and other real-world problems. Plenty of formalisms have been proposed and abstracted in algebraic structures. Specialized areas diverged such as cost function networks or semiring-based soft constraints. Integration with conventional constraint problems are implemented using soft global constraints. Most of them involve a cost variable / cost functions. This is not good, if costs can be defined by different agents/modelers that lack a common currency idea. We propose a formalism that is purely qualitative, relies on the satisfaction/dissatisfaction of soft constraints. Preferences are written as a directed acyclic graph. We show how to proceed from this graph by suggesting set orderings over sets of violated constraints, prove that these orderings are mathematically enforced by common axioms of monoidal soft constraints, discuss the relationship to other formalisms in terms of expressiveness, implement various optimization strategies using MiniZinc/MiniSearch, and provide experimental results.

Keywords Soft Constraints · Over-constrained Problems · Modeling Formalisms · MiniZinc

Argument: There are plenty bespoke algorithms for weighted CSPs but there is little support for higher-level modeling languages such as MiniZinc, Essence or OPL.

Many articles discuss how to adequately model preferences and the comparison of such formalisms. But support for common modeling language exploiting features of conventional constraint solvers are rare.

This research is partly sponsored by the German Research Foundation (DFG) in the project “OC-Trust” (FOR 1085).

Alexander Schiendorfer
University of Augsburg
E-mail: schiendorfer@isse.de

Alexander Knapp
University of Augsburg
E-mail: knapp@isse.de

Wolfgang Reif
University of Augsburg
E-mail: reif@isse.de

1 Introduction

Many (perhaps most) industrial combinatorial optimization problems occurring in practice tend to be over-constrained, according to [4]. Usually, this fact leads to several refinements of an initial constraint model by manually weakening or dropping constraints until a solution can be found. The situation is more severe if the concrete instance, i.e., all parameters to a problem are known only at runtime when a system has to make autonomous decisions. Simply failing with `unsatisfiable` is not an option, a compromise solution is necessary.

The problem has been recognized for many years (see Section 2) but still lacks ready-to-use implementations for modern constraint platforms. Most often, softening constraints is achieved by encoding violations in one or more cost variables that need to be minimized, leading to the framework of weighted CSP. Assigning weights is comparatively simple, expressive, and enables techniques such as soft global constraints [4] or soft arc consistency [2]. However, there are certain drawbacks of expressing graded satisfaction only in terms of weights.

better use sac-revisited

- Why do we really want a graph of constraints?
  - Numbers are hard to pick
  - Often, there is more than one cost value and an aggregation has to be performed, most likely using a weighted sum of cost values. When "emulating" a lexicographic ordering over these objectives, one has to pick sufficiently large weights to give precedence to a more important objective. Not only may this cause overflows and other numerical issuesbut also lead to rather weak propagation of bounds in CP-solvers. Moreover, choosing suitable weights is hard if the objectives' domains do not show natural bounds but depend on choices made for other decision variables.
  - Clear and well-defined semantics such as lexicographic ordering.

cite minisearch

cite Pierre Schaus LNS

Constraint relationships have first been introduced in combination with a translation to weighted CSP in [9]. Theoretical concerns, such as the relationship to algebraic frameworks and hierarchical constraints have been addressed in [5] and [8], respectively. This paper, by contrast, aims to present the formalism and its relations to related concept in a unified way as well as to provide a reusable implementation using MiniZinc/MiniSearch.

2 Related Work

Pioneering attempts to generalize the rigid constraint formalism were offered by the formalism of *partial constraint satisfaction* [3]. The core idea was to define a metric that measures the distance d of an assignment θ to the solution space of the original problem with $d = 0$ indicating solutions. Proposed choices included the number of domain items to be added to make θ feasible or the number of *violated constraints* of θ. The latter is now better known as *Max-CSP*.

Valued constraints took on that idea to label constraints with *values* from a totally ordered set that can be seen as penalties for violating constraints.

Similarly, *semiring-based* soft constraint frameworks labels each assignment with a satisfaction degree of a so-called *c-semiring*, i.e., an algebraic structure composed of a multiplication operator for *combining* satisfaction degrees of several soft constraints as well as an addition operator (i.e., a supremum) that induces a partial order for ranking solutions.

- T. Petit, J.C. Régin, and C. Bessière. Meta-constraints on violations for over constrained problems.
  [6] introduces reified variables for cost values, we rely on that technique
- Conflict-Directed A* Search for Soft Constraints
  [7] based on reified variables, uses propagation in combination with A star, i.e., in principle a best first search where propagation of reified variables may lower the objective (if one satisfied soft constraint propagates the violation of many others, try another assignment first). Not a classical branch and bound approach since the node with the highest promising objective value is branched on. Requires quite some manual bookkeeping. Not clear how much benefit this brings over conventional search using variable/value heuristics.
- [4] soft global constraints
- [1] translate weighted CSP to SMT $\rightarrow$ relevant for evaluation! What can be expressed there, what not?

3 Foundations

As usual, a constraint problem $CP = (X, D, C)$ is described by a set of decision variables X, their associated family of domains of possible values $D = (D_x)_{x \in X}$, and a set of constraints C that restrict valid assignments. An assignment θ is a mapping from X to D, written as $\theta \in [X \rightarrow D]$, such that each variable x maps to a value in D_x. A constraint $c \in C$ is understood as a map $c : [X \rightarrow D] \rightarrow \mathbb{B}$ where we write $\theta \models c$ for $c(\theta) = \mathit{true}$. We call an assignment θ a solution if $\theta \models c$ holds for all $c \in C$ and write the set of all solutions of a constraint problem CP as $\mathrm{sol}(CP)$. If the main task of CP is to find a solution, we call it a *constraint satisfaction problem* (CSP).

We obtain *constraint optimization problems* (COP) by adding an objective function $f : [X \rightarrow D] \rightarrow P$ where $(P, \leq_P)$ is a partial order, i.e., $\leq_P$ is a reflexive, antisymmetric, and transitive relation over P. Elements of P are interpreted as *solution degrees*, denoting quality. Without loss of generality, we interpret $m <_P n$ as m being inferior to n and restrict our attention to maximization problems. Hence, a solution degree m is optimal with respect to a constraint problem CP, if for all solutions $\theta \in \mathrm{sol}(CP)$ it holds either that $f(\theta) \leq_P m$ or $f(\theta) \parallel_P m$, expressing incomparability. It is *reachable* if there is a solution $\theta \in \mathrm{sol}(CP)$ such that $f(\theta) = m$.

- In a PVS, e.g., ε_M is trivially optimal.
- Non-reachable optimal solution degrees emerge, e.g., as upper bounds or from the supremum operator in c-semirings.
- Thus, the answer returned from dynamic programming for c-semirings usually just returns an upper bound that is not necessarily reachable.

3.1 Category Theory

This is just verbatim from Wikipedia, Abstract Algebra, better reformulate

- Free algebras can be shown to exist using the Adjoint Functor Theorem. See [Poi92] A. Poigné. Basic category theory.

- Algebraic structures, with their associated homomorphisms, form mathematical categories. Category theory is a powerful formalism for analyzing and comparing different algebraic structures.
- Mathematical categories are composed of *objects* (e.g,. algebraic structures) and *arrows* (also called morphisms) between them. Each arrow f admits a domain A and codomain B, both being objects, and is written as $f : A \to B$. For each arrow $f : A \to B$ and $g : B \to C$ there has to be a composite arrow $(g \circ f) : A \to C$. Arrow composition $\circ$ needs to be associative and for each object A, there has to be an identity arrow $\mathrm{id}_A : A \to A$ acting as "neutral element" with respect to composition, i.e., $\mathrm{id}_A \circ f = f \circ \mathrm{id}_A = f$. The most straightforward example is given by the category Set, where objects are sets, arrows are functions, composition is function composition, and the identity arrows are just the identity function. A slightly more elaborate example is given by PO, the category of partially-ordered sets, that has partial orders as objects and partial order homomorphisms (i.e., monotone functions) as arrows. Note that this definition is proper since monotone functions are closed under function composition, i.e., if $\varphi : |P| \to |Q|$ and $\psi : |Q| \to |R|$ are monotone functions, so is $\psi \circ \varphi$.
- In categorical arguments, it is important to distinguish algebraic structures from their underlying components, e.g., underlying sets, functions, or relations. Taking partially-ordered sets as examples, we refer to the structure $P = (|P|, \leq_P)$ as *partial order*, to $|P|$ as the *underlying set*, and to the binary, reflexive, antisymmetric, and transitive relation $\leq_P \subseteq |P| \times |P|$ as *ordering* (relation). All applications dealt with in this paper are examples of so-called *concrete categories* where objects are sets (perhaps) with additional structures and arrows are set-theoretic functions satisfying certain axioms (preserving structure as in homomorphisms).
- More technically, $|\cdot|$ is a an example of a *functor*, i.e., a mapping F between categories C and D that sends every C-object A to a D-object $F(A)$ and every C-arrow $f : A \to B$ to a D-arrow $F(f) : F(A) \to F(B)$. In this case, $|\cdot| : \mathrm{PO} \to \mathrm{Set}$ is a mapping such that if $P = (X, \leq_P)$, $|P| = X$ and each PO-homomorphism $\varphi : P \to Q$ is mapped to its underlying function $|\varphi| : |P| \to |Q|$.

4 Implementation

```
% Library predicate:
% -----------------
% Implements single predecessor
% dominance on sets of constraints
% (upper smyth ordering with partial
%  order inversion)
% ---------------------------------
include "alldifferent_except_0.mzn";

predicate spd_worse(var set of int: lhs, var set of int: rhs,
                    set of int: softConstraints,
                    array[int, 1..2] of int: edges
) :: promise_total =
let {
  int: le = min(index_set_1of2(edges));
  int: ue = max(index_set_1of2(edges));

  var set of int: lSymDiff = lhs diff rhs;
  var set of int: rSymDiff = rhs diff lhs;
  set of int: softConstraints0 = {0} union softConstraints;
  % 0 represents noVal for constraints not in the right-hand-side
```

```
  var set of int: rUndefined = softConstraints diff rSymDiff;

  % I need to make the dominance explicit by a function
  array[softConstraints] of var softConstraints0: witness;

  % collect all predecessors such that succ in lessThans[pred]
  % iff succ less than pred
  array[softConstraints] of set of softConstraints: dominators =
    [ {succ | succ in softConstraints where exists(e in le..ue)
         (edges[e,1] = pred /\ edges[e,2] = succ)} | pred in softConstraints];

} in (
  lhs != rhs /\
  alldifferent_except_0(witness) /\
  forall(s in ub(rUndefined)) (s in rUndefined -> witness[s] = 0) /\
  forall(s in ub(rSymDiff)) (s in rSymDiff -> (witness[s] in lSymDiff /\
                                               witness[s] in dominators[s]))
);
```

```
% Library predicate:
% ------------------
% Implements transitive predecessor
% dominance on sets of constraints
% ---------------------------------

predicate tpd_worse(var set of int: lhs, var set of int: rhs,
                    set of int: softConstraints,
                    array[int, 1..2] of int: edges
) =
let {
  int: le = min(index_set_1of2(edges));
  int: ue = max(index_set_1of2(edges));

  var set of int: lSymDiff = lhs diff rhs;
  var set of int: rSymDiff = rhs diff lhs;

  % collect all predecessors such that succ in lessThans[pred]
  % iff succ less than pred
  array[softConstraints] of set of softConstraints: dominateds =
    [ {succ | succ in softConstraints where exists(e in le..ue)
         (edges[e,1] = succ /\ edges[e,2] = pred)} | pred in softConstraints];

  array[softConstraints] of var bool: isDominated = [ exists(d in ub(lSymDiff))
        (d in lSymDiff /\ s in dominateds[d])  | s in softConstraints];


} in (
  lhs != rhs /\
  forall(r in ub(rSymDiff)) (r in rSymDiff -> isDominated[r])
);
```

Some points that should be emphasized

1. Variable ordering can be used to assign reified variables first.
2. Order them decreasingly by importance (can be done using data manipulation in MiniZinc)
3. Assign `true` first, hoping to find a solution to all soft constraints.
4. Discuss/prove that this search tree, in a static ordering, assures that violation degrees are tried in TPD-order.
5. SPD-order can probably achieved by using a restarts strategy → use new scope every time, add only those soft constraints that are supposed to hold.
6. Discuss similarities/differences to conflict-directed A* search.
7. *Quite a nifty point:* in addition to posting the constraint `xpd_better(lb, violatedScs)`, meaning that the set of violated soft constraints should be better in the next solution,

we can post something like `penalty(lb) > penalty(violatedScs)` *as well*! Why does that make sense?

- *If* we find a solution θ_2 that is XPD-better than θ_1, then it is also penalty-better (by implication).
- `violatedScs` is a bounded set variable (at least below by $\emptyset$, above by `soft-constraints`). From these bounds, the solver can immediately derive bounds for `penalty(violatedScs)`. Propagation of reified variables during search (one satisfied constraint implying the violation of another etc.) is then properly handled. Search can stop, once the best case minimal penalty violation cannot go below the imposed upper bound for violation.
- This constraint is *redundant*, i.e., if for any solution θ_2 XPD-better θ_1 holds, penalty better holds as well.
- But we do not have a *propagator* for XPD-better. The set variable `violatedScs` is bounded below by the number of minimally (definitely, already) violated soft constraints, above by the maximally possible violations (cmp. α, ζ). We could, in principle, build a simple bounds propagator for XPD-better that restricts the domain of `violatedScs` to values strictly above the last found lower bound.
- It's unreasonable to assume a default behavior like bounds propagation on a user-defined predicate such as XPD-better. It could be that the predicate is only true "somewhere" in the middle between lower and upper set-bound. Then bounds propagation would incorrectly cut partial assignments.
- But in lieu of a dedicated XPD-propagator, we can benefit from the redundant penalty constraint. For instance, suppose we have seen a solution violating $\{c_2, c_3\}$, with penalty 2. Assume we are in a search tree and the lower bound for our `violatedScs` is $\{c_1\}$ violated constraints, and the upper bound be $\{c_1, c_2, c_3\}$. Thus, the penalty is at least 3, at most 5. We can cut the search since we posted `penalty(violatedScs) < 2`.
- It turns out that propagation is smarter than I thought ... since the witness is just a bunch of additional variables we get much more propagation than initially thought, which is nice.
- The penalty trick still cuts down the number of failures.

look for reification of global constraints paper, there is also something about that in one of the MZ functions papers!

8. Global constraints aren't as easily reified. Probably in the future . Better use a restarts approach for now.
9. It turns out that, since XDP-better is formulated as a MiniZinc predicate in terms of a conjunction of other (global) constraints, we inherit some propagation capability.
10. MiniZinc is expressive enough to encapsulate utility methods as MiniZinc functions. That includes a consistency check to guarantee that the entered graph is not cyclic, that edges and vertex sets are consistent (no edge pointing to an integer other than one supported by the set of vertices), as well as a function that calculates the transitive closure of a given graph. This is necessary to obtain a partial order (see *free* partial order over a dag).

4.1 Search

The first search strategy corresponds to classical branch-and-bound (BAB) search in propagation engines. For every found solution, a constraint is imposed that the next solution has to be strictly better.

```
function ann: strictlyBetterBAB(var set of SOFTCONSTRAINTS: violatedScs) =
       repeat(
           if next() then
               let {
                 set of SOFTCONSTRAINTS: lb = sol(violatedScs);
               } in (
                 print("Intermediate solution:") /\ print() /\
                 commit() /\ post(spd_better(lb, violatedScs,
                                 SOFTCONSTRAINTS, edges))
               )
           else break endif
       );

[...]

search strictlyBetterBAB_TPD(violatedScs) /\
   if hasSol() then print("Final solution: ") /\ print()
   else print("No solution found\n") endif;
```

While this procedure yields optimal solutions, it is not ideal for partially ordered objectives since another optimum need not be better than the current solution. Instead, we need to impose that any coming solution *must not be dominated* by *any* solution seen so far. Technically, we have a set of lower bounds (the satisfaction degrees of previous solutions) $L = \{l_1, \ldots, l_m\}$ and require that it must not be the case that $\exists l \in L : \text{obj} \leq_M l$ for any partial valuation structure M. The next solution must either be strictly better than any one of the maxima of L or incomparable to all of them. We would also like to remove lower bounds from L when we find stricter ones (as one would do with totally ordered objective spaces) but this is not as easy due to partiality (which constraint was to blame/should stop being imposed).

```
function ann: onlyNotDominated(var set of SOFTCONSTRAINTS: violatedScs) =
       repeat(
           if next() then
               let {
                 set of SOFTCONSTRAINTS: lb = sol(violatedScs);
               } in (
                 print("Intermediate solution:") /\ print() /\
                 commit() /\ post(not (spd_better(violatedScs, lb,
                                    SOFTCONSTRAINTS, edges)
                                    \/ violatedScs = lb ) )
               )
           else break endif
       );
```

The difference is best explained by an example. Consider the following oversimplified constraint model.

```
var 1..3: x;

constraint x = 1 <-> violated[1];
constraint x = 2 <-> violated[2];
constraint x = 3 <-> violated[3];

solve
:: int_search([x], input_order, indomain_max, complete)
search strictlyBetterBAB_TPD(violatedScs);
```

We explore `x` in a decreasing order. This results in the sequence $\langle\{3\},\{2\},\{1\}\rangle$ of satisfaction degrees. $\{3\}$ and $\{2\}$ both dominate $\{1\}$ but are incomparable; $\{1\} <_M \{3\}$, $\{1\} <_M \{2\}$ but $\{2\} \parallel_M \{3\}$. The reachable optima of this problem are clearly $\{\{2\},\{3\}\}$

```
Intermediate solution:Obj: 1 by violating {3..3 } : x -> 3
----------
```

==========

Each assignment to `x` violates precisely one soft constraint.

5 Evaluation

5.1 Tailoring MiniZinc to Accomodate Softness

Here we'll have a short introduction into how we actually modeled our benchmark problems. Includes how we added several notions of constraints, different levels (few: 3-5, medium: 5-10, many: 10+)

What are the aspects that need motivation?

1. Enhanced propagation by redundant constraints
2. Search heuristics
3. Comparing BaB with LNS (depending on MiniSearch's progress)

1. **Factors**: type of heuristics / variable ordering, enhanced propagation
2. **Dependent Variables**: runtime / failures / nb. problems solved / optimal value after 5 minutes

5.2 Overhead from Softness

Here, we'll compare the plain MiniZinc problems with softened but more constrained problems. Central question: How much harder / slower will it be to find (optimal) solutions? We shall, of course, present the altered models online! Including some documentation about how we modified the original problems.

1. Measure if a problem that was easy in the beginning is still easy
2. Measure if hard problems get tremendously harder (play a bit with search heuristics)

5.3 Comparison with Related Work

Here, we'll demonstrate that we do not really have a competing product, since no solver truly implements algebraic structures for soft constraints. So nothing is really offered in a qualitative setting.

But in the quantitative realm (with all its disadvantages), there are two competing solvers that offer bespoke solutions for weighted CSP. Even though we do not want to really compete in terms of runtime, we hope to see the differences

1. WSimply
2. Toulbar2, perhaps interfaced by Numberjack

Figure out workflow of WSimply with MiniZinc models – that'd be ideal!

Try a model with Numberjack

Find a cool scheduling/packing problem involving lots of global constraints and make a translation to WCSP/Numberjack Format for toulbar2

6 Conclusion

References

1. Ansótegui, C., Bofill, M., Palahí, M., Suy, J., Villaret, M.: Solving Weighted CSPs with Meta-Constraints by Reformulation into Satisfiability Modulo Theories. Constraints **18**(2), 236–268 (2013)
2. Cooper, M., Schiex, T.: Arc Consistency for Soft Constraints. Artificial Intelligence **154**(1), 199–227 (2004)
3. Freuder, E.C., Wallace, R.J.: Partial Constraint Satisfaction. Artif. Intell. **58**(1–3), 21–70 (1992)
4. van Hoeve, W.J.: Over-constrained Problems. In: Hybrid Optimization, pp. 191–225. Springer (2011)
5. Knapp, A., Schiendorfer, A., Reif, W.: Quality over Quantity in Soft Constraints. In: Proc. 26$^{\text{th}}$ Int. Conf. Tools with Artificial Intelligence (ICTAI'2014), pp. 453–460 (2014)
6. Petit, T., Régin, J.C., Bessière, C.: Meta-Constraints on Violations for Over Constrained Problems. In: Proc. 12$^{\text{th}}$ Int. Conf. Tools with Artificial Intelligence (ICTAI'2000), pp. 358–365 (2000)
7. Sachenbacher, M., Williams, B.C.: Conflict-Directed A* Search for Soft Constraints. In: Proc. 3$^{\text{rd}}$ Int. Conf. Integration of Artificial Intelligence and Operations Research Techniques in Constraint Programming (CPAIOR'2006), pp. 182–196. Springer (2006)
8. Schiendorfer, A., Knapp, A., Steghöfer, J.P., Anders, G., Siefert, F., Reif, W.: Partial Valuation Structures for Qualitative Soft Constraints. In: R.D. Nicola, R. Hennicker (eds.) Software, Services and Systems - Essays Dedicated to Martin Wirsing on the Occasion of His Emeritation, Lect. Notes Comp. Sci. 8950. Springer (2015)
9. Schiendorfer, A., Steghöfer, J.P., Knapp, A., Nafz, F., Reif, W.: Constraint Relationships for Soft Constraints. In: M. Bramer, M. Petridis (eds.) Proc. 33$^{\text{rd}}$ SGAI Int. Conf. Innovative Techniques and Applications of Artificial Intelligence (AI'13), pp. 241–255. Springer (2013)