Constraint Relationships

Language Features

Universität
Augsburg
University

Overview

These slides show some features of the language/library

To familiarize yourself with basics, consider looking at:

- Step-by-Step enhancing a MiniZinc model (establishes the core elements)
- Case Studies (for some specific examples)

`http://isse-augsburg.github.io/constraint-relationships/`

SoftConstraints in MiniZinc (HelloWorld)

Institute for Software & Systems

```
% X: {x,y,z} D_i = {1,2,3}, i in X
%    * c1: x + 1 = y * c2: z = y + 2 * c3: x + y <= 3
% (c) ISSE
% isse.uni-augsburg.de/en/software/constraint-relationships/
include "soft_constraints/minizinc_bundle.mzn";

var 1..3: x; var 1..3: y; var 1..3: z;

% read as "soft constraint c1 is satisfied iff x + 1 = y"
constraint x + 1 = y <-> satisfied[1];
constraint z = y + 2 <-> satisfied[2];
constraint x + y <= 3 <-> satisfied[3];

% soft constraint specific for this model
nScs = 3; nCrEdges = 2;
crEdges = [| 2, 1 | 3, 1 |]; % read c2 is less important than c1

solve minimize penSum; % minimize the sum of penalties
```

Step 4: Switch to PVS Architecture

Institute for Software & Systems Engineering

Idea Many soft constraint formalisms are generalized by partial valuation structures (Gadducci et al., 2013) that give the codomain of the objective function an *algebraic structure*.

- A partially ordered valuation structure is described by $(M, \oplus_M, \leq_M, \varepsilon_M)$ where
  - M …is a set of violation/satisfaction degrees, e.g., $\mathbb{N}$ for weights, $[0, 1]$ for probabilities etc.
  - $\oplus_M$ …is a binary *combination* operation to aggregate values from M, e.g., $3 \oplus_M 5 \equiv 3 + 5$
  - $\leq_M$ …is a partial *order* over M operation to rank values from M, e.g., $5 \leq_M 3 \equiv 5 \geq 3$, $m \leq_M n$ means *m is worse* than *n*
  - $m \leq_M \varepsilon_M$ …for every $m \in M$, i.e. ε_M is the *best* possible solution, e.g. 0 violation
- Similar (Bistarelli et al., 1999): c-semirings and (total) valuation structures (`toulbar2`)

PVS–Idea

Institute for Software & Systems Engineering

Concrete PVS	M	$\oplus_M$	$\leq_M$	ε_M
Weighted CSP	$\mathbb{N}$	$+$	$\geq$	0
Fuzzy CSP	$[0,1]$	min	$\leq$	1
Constraint Relationships[1]	2^{C_s}	$\cup$	$\subseteq_{\mathrm{SPD}}$	$\emptyset$

Main Idea

Implement search strategies (BaB and LNS) for partially ordered valuation structures. Instantiate for concrete problems.

From now on we rely on MiniSearch (`www.minizinc.org/minisearch/`).

[1] C_s is the set of soft constraints, $\subseteq_{\mathrm{SPD}}$ is the SPD-ordering on sets.

Step 5: Use PVS-based Search

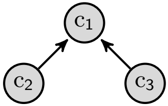
```
% only declare predicate for worsening
predicate isWorse(var set of int: leftViolatedScs,
                  var set of int: rightViolatedScs);
function ann: strictlyBetterBAB(var set of SOFTCONSTRAINTS: violatedScs)
=     repeat(
          if next() then
              let { set of SOFTCONSTRAINTS: lb = sol(violatedScs);} in (
                print("Intermediate solution:") /\ print() /\
                commit() /\ post(isWorse(lb, violatedScs))
              )
          else break endif);
```

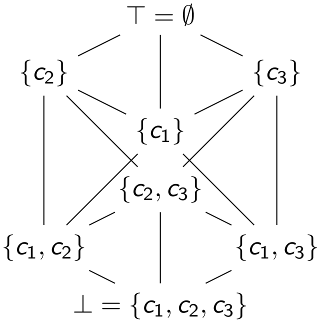
Only relies on `isWorse`!

```
[... inside pvs_spd.mzn]
predicate isWorse(var set of int: leftViolatedScs,
                 var set of int: rightViolatedScs) = (
  spd_worse(leftViolatedScs, rightViolatedScs, SOFTCONSTRAINTS, crEdges)
);
```

Search types

The whole valuation space (partially ordered)

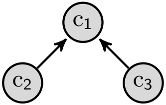


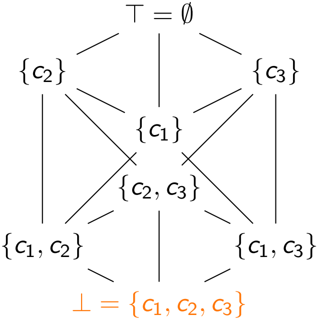



```
%
% Typical Optimization Routine (Branch and Bound):
%
%  1. Look for the first feasible solution
%  2. Impose restrictions on the next feasible solution
%  3. Repeat
```

Search types: Strictly better

The whole valuation space (partially ordered)

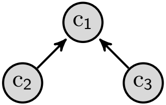


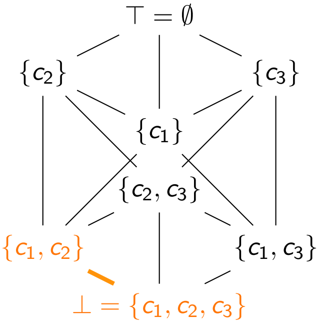



```
function ann: strictlyBetterBAB(var set of SOFTCONSTRAINTS: vScs)
      = repeat(
           if next() then
             commit() /\
             post(isWorse(sol(vScs), vScs))
           else break endif      );
```

Search types: Strictly better

The whole valuation space (partially ordered)



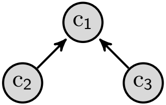


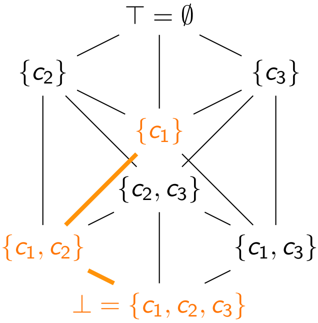

```
function ann: strictlyBetterBAB(var set of SOFTCONSTRAINTS: vScs)
      = repeat(
           if next() then
             commit() /\
             post(isWorse(sol(vScs), vScs))
           else break endif      );
```

Search types: Strictly better

Institute for
Software & Systems
Engineering

The whole valuation space (partially ordered)

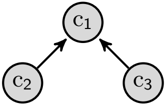


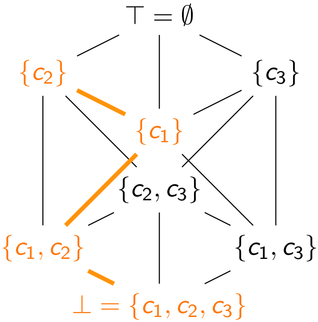



```
function ann: strictlyBetterBAB(var set of SOFTCONSTRAINTS: vScs)
      = repeat(
          if next() then
            commit() /\
            post(isWorse(sol(vScs), vScs))
          else break endif     );
```

Search types: Strictly better

The whole valuation space (partially ordered)



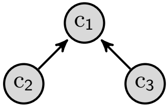


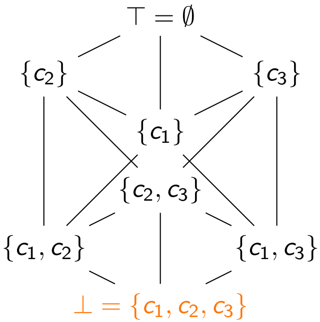

```
function ann: strictlyBetterBAB(var set of SOFTCONSTRAINTS: vScs)
      = repeat(
          if next() then
            commit() /\
            post(isWorse(sol(vScs), vScs))
          else break endif      );
```

Search types: Only not dominated

Institute for Software & Systems Engineering

The whole valuation space (partially ordered)

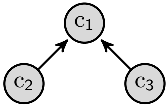


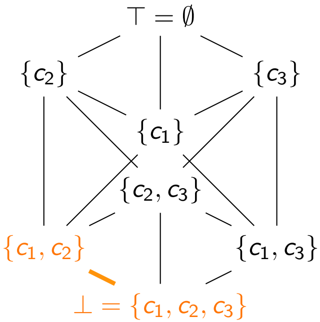



```
function ann: onlyNotDominatedBAB(var set of SOFTCONSTRAINTS: vScs)
      = repeat(
           if next() then
             commit() /\
             post((isWorse(vScs, sol(vScs)) \/ vScs = sol(vScs)))
           else break endif     );
```

Search types: Only not dominated

The whole valuation space (partially ordered)

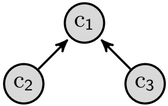


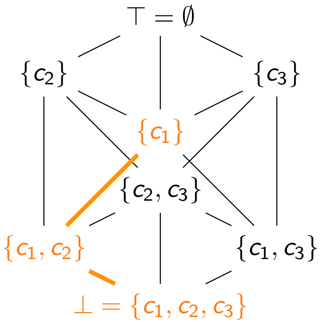



```
function ann: onlyNotDominatedBAB(var set of SOFTCONSTRAINTS: vScs)
      = repeat(
          if next() then
            commit() /\
            post((isWorse(vScs, sol(vScs)) \/ vScs = sol(vScs)))
          else break endif      );
```

Search types: Only not dominated

The whole valuation space (partially ordered)



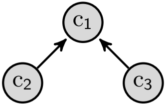


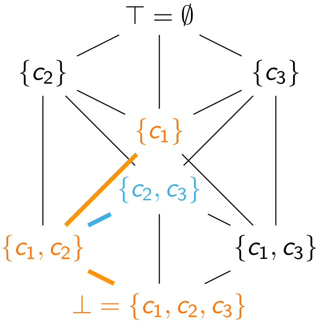

```
function ann: onlyNotDominatedBAB(var set of SOFTCONSTRAINTS: vScs)
      = repeat(
          if next() then
            commit() /\
            post((isWorse(vScs, sol(vScs)) \/ vScs = sol(vScs)))
          else break endif      );
```

Search types: Only not dominated

Institute for Software & Systems Engineering

The whole valuation space (partially ordered)

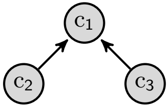


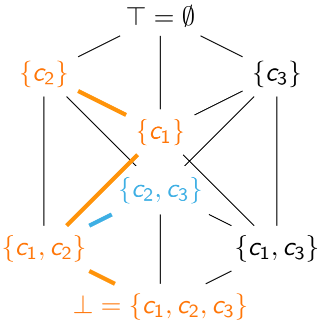



```
function ann: onlyNotDominatedBAB(var set of SOFTCONSTRAINTS: vScs)
      = repeat(
          if next() then
            commit() /\
            post((isWorse(vScs, sol(vScs)) \/ vScs = sol(vScs)))
          else break endif     );
```

Search types: Only not dominated

The whole valuation space (partially ordered)



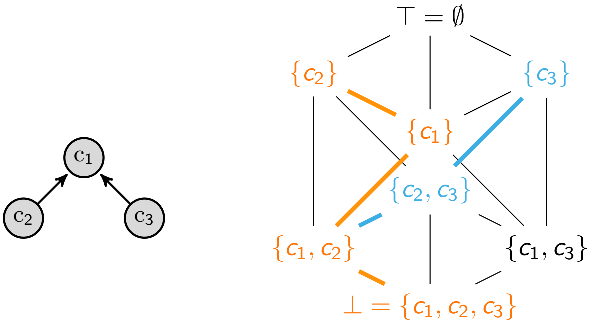



```
function ann: onlyNotDominatedBAB(var set of SOFTCONSTRAINTS: vScs)
      = repeat(
          if next() then
            commit() /\
            post((isWorse(vScs, sol(vScs)) \/ vScs = sol(vScs)))
          else break endif       );
```

Search types: Only not dominated

Institute for Software & Systems Engineering

The whole valuation space (partially ordered)



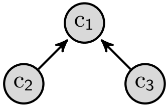

```
function ann: onlyNotDominatedBAB(var set of SOFTCONSTRAINTS: vScs)
      = repeat(
           if next() then
             commit() /\
             post((isWorse(vScs, sol(vScs)) \/ vScs = sol(vScs)))
           else break endif      );
```

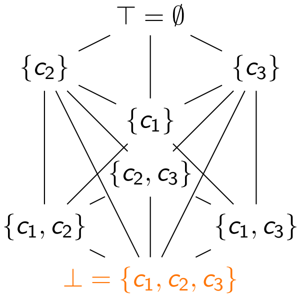
Search Demo: Model

```
include "minisearch.mzn";
include "soft_constraints/soft_constraints.mzn";
include "soft_constraints/pvs_search.mzn";
include "soft_constraints/tpd_worse.mzn";
include "soft_constraints/pvs_tpd.mzn";
nScs = 3; var 1..3: x;

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

nCrEdges = 2; crEdges = [| 2, 1 | 3, 1 |];
% solution degrees are explored in order {3}, {2}, {1}
solve
:: int_search([x], input_order, indomain_max, complete)
search strictlyBetterBAB(violatedScs);
%search onlyNotDominatedBAB(violatedScs);

output [" Obj: \(penSum) violating {\(violatedScs)}: x=\(x)"];
```

Search Demo: Strictly better

Execute `minisearch diff-BAB-sb.mzn`
(explores in order $\{c_3\}, \{c_2\}, \{c_1\}$)

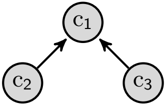


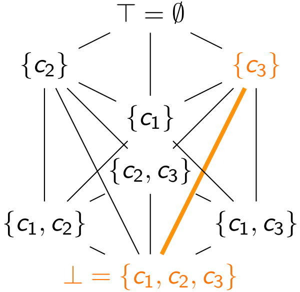



```
% just sees {c_3} then finds no better solution and stops
%
Intermediate solution: Obj: 1 violating {3..3}: x=3
----------
==========
%
```

Search Demo: Strictly better

Execute `minisearch diff-BAB-sb.mzn`
(explores in order $\{c_3\}, \{c_2\}, \{c_1\}$)

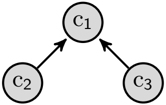


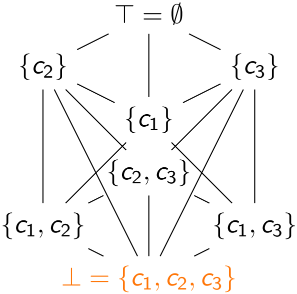



```
% just sees {c_3} then finds no better solution and stops
%
Intermediate solution: Obj: 1 violating {3..3}: x=3
----------
==========
%
```

Search Demo: Only not dominated

Execute `minisearch diff-BAB-ond.mzn`
(explores in order $\{c_3\}, \{c_2\}, \{c_1\}$)

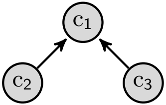


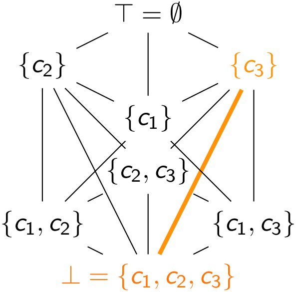



```
% lists both solutions
Intermediate solution: Obj: 1 violating {3..3}: x=3
----------
Intermediate solution: Obj: 1 violating {2..2}: x=2
----------
==========
```

Search Demo: Only not dominated

Execute `minisearch diff-BAB-ond.mzn`
(explores in order $\{c_3\}, \{c_2\}, \{c_1\}$)

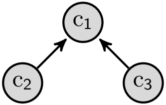


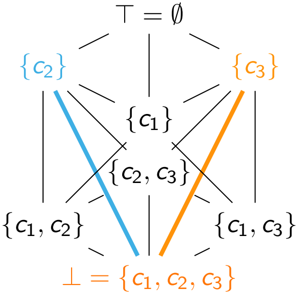



```
% lists both solutions
Intermediate solution: Obj: 1 violating {3..3}: x=3
----------
Intermediate solution: Obj: 1 violating {2..2}: x=2
----------
==========
```

Search Demo: Only not dominated

Execute `minisearch diff-BAB-ond.mzn`
(explores in order $\{c_3\}, \{c_2\}, \{c_1\}$)






```
% lists both solutions
Intermediate solution: Obj: 1 violating {3..3}: x=3
----------
Intermediate solution: Obj: 1 violating {2..2}: x=2
----------
==========
```

Switching PVS Type

Institute for Software & Systems Engineering

Concrete PVS are instantiated by importing the appropriate `pvs_x.mzn` file.

```
include "minisearch.mzn";
include "soft_constraints/soft_constraints.mzn";
include "soft_constraints/spd_worse.mzn"; % tpd_worse.mzn
include "soft_constraints/pvs_spd.mzn"; % pvs_tpd.mzn, pvs_weighted.mzn
include "soft_constraints/pvs_search.mzn";

var 0..10: x; var 0..10: y;
nScs = 2; nCrEdges = 1; crEdges = [| 2, 1 |];
constraint x < y <-> satisfied[1];
constraint x + 4 = y <-> satisfied[2];

solve
search strictlyBetterBAB(violatedScs) /\ print();
output["x = \(x), y = \(y), violatedScs = \(violatedScs)"];
```

Consistency Checks

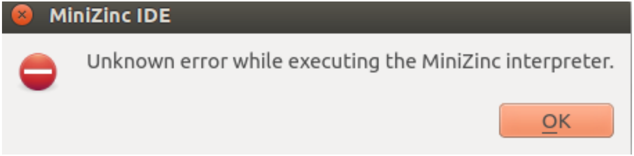
Institute for Software & Systems Engineering

Vital, when designing constraint relationships: avoid cycles!

`model-inconsistent.mzn`

```
include "soft_constraints/soft_constraints.mzn";
include "soft_constraints/spd_worse.mzn";
include "soft_constraints/pvs_spd.mzn";

var 0..10: x; var 0..10:y;

nScs = 2;
nCrEdges = 2; crEdges = [| 2, 1 | 1, 2 |];
constraint x < y <-> satisfied[1];
constraint x + 4 = y <-> satisfied[2];

solve satisfy;

output["x = \(x), y = \(y), violatedScs = \(violatedScs)"];
```

Consistency Checks



Better: Add model checks to detect cyclic relationships!

Consistency Checks: Assertions

model-inconsistent-safe.mzn

```
include "minisearch.mzn";
include "soft_constraints/soft_constraints.mzn";
include "soft_constraints/spd_worse.mzn";
include "soft_constraints/pvs_spd.mzn";
include "soft_constraints/pvs_search.mzn";
include "soft_constraints/cr_consistency.mzn";
var 0..10: x; var 0..10:y;
nScs = 2;
nCrEdges = 2; crEdges = [| 2, 1 | 1, 2 |];

constraint x < y <-> satisfied[1];
constraint x + 4 = y <-> satisfied[2];
constraint assert(consistentCR(SOFTCONSTRAINTS, crEdges),
               "Constraint relationship is not consistent");
solve
search strictlyBetterBAB(violatedScs) /\ print();
output["x = \(x), y = \(y), violatedScs = \(violatedScs)"];
```

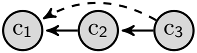
Consistency Checks: Assertions

```
minisearch model-inconsistent-safe.mzn
```

```
minizinc/std/soft_constraints/cr_consistency.mzn:46:
  in call 'assert'
  Assertion failed: Relationship is cyclic!
```

Transitive Closure

For larger constraint relationships, it can be more convenient to just specify a directed acyclic graph and have the closure (all transitive edges) be calculated automatically.




```
include "soft_constraints/soft_constraints.mzn";
include "soft_constraints/cr_types.mzn";
include "soft_constraints/cr_consistency.mzn";
nScs = 3; penalties = [1 | i in 1..nScs];

array[int, 1..2] of int: crTempEdges = [| 3, 2 | 2, 1 ||];
crEdges = transClosure(SOFTCONSTRAINTS, crTempEdges);
nCrEdges = max(crEdges);
var 0..3: x; solve satisfy;

output["crEdges=\(crEdges)\ncrTempEdges=\(crTempEdges)"++
      "\nnCrEdges = \(nCrEdges)"];
```

Variable Ordering

Institute for Software & Systems Engineering

Recall that `soft_constraints.mzn` defines a variable ordering with weights in descending order:

```
% find the sorted permutation of soft constraint instances
array[SOFTCONSTRAINTS] of SOFTCONSTRAINTS: sortPermScs =
    arg_sort(penalties);
% invert, since arg_sort use <= and we need decreasing order
array[SOFTCONSTRAINTS] of SOFTCONSTRAINTS: mostImpFirst =
    [ sortPermScs[nScs-s+1] | s in SOFTCONSTRAINTS];
```

We can use this ordering to try out important constraints early on in search.

Variable Ordering: Demo

Institute for Software & Systems Engineering

`smallexample.mzn`:

```
include "soft_constraints/soft_constraints.mzn";
include "soft_constraints/tpd_worse.mzn";
include "soft_constraints/pvs_tpd.mzn";
var 1..3: x; var 1..3: y; var 1..3: z;

constraint x + 1 = y <-> satisfied[1];
constraint z = y + 2 <-> satisfied[2];
constraint x + y <= 3 <-> satisfied[3];
nScs = 3; nCrEdges = 2; crEdges = [| 2, 1 | 3, 1 |];

solve minimize penSum;
output ["Obj: \(penSum) by x=\(x), y=\(y),z=\(z)"];
```

```
Obj: 4 by x=1, y=1,z=1
----------
Obj: 3 by x=1, y=1,z=3
----------
Obj: 1 by x=1, y=2,z=1
----------
```

Variable Ordering: Demo

Institute for Software & Systems Engineering

smallexample-mif.mzn:

```
include "soft_constraints/soft_constraints.mzn";
include "soft_constraints/tpd_worse.mzn";
include "soft_constraints/pvs_tpd.mzn";
var 1..3: x; var 1..3: y; var 1..3: z;
constraint x + 1 = y <-> satisfied[1];
constraint z = y + 2 <-> satisfied[2];
constraint x + y <= 3 <-> satisfied[3];
nScs = 3; nCrEdges = 2; crEdges = [| 2, 1 | 3, 1 |];
solve
:: int_search([satisfied[mostImpFirst[i]] | i in 1..nScs],
              input_order, indomain_max, complete)
minimize penSum;
output ["Obj: \(penSum) by x=\(x), y=\(y),z=\(z)"];
```

```
Obj: 1 by x=1, y=2,z=1
----------
==========
```

(finds the optimal solution at first try)

Variable Ordering: Demo

Institute for Software & Systems Engineering

This ordering can of course be combined with problem-specific heuristics (e.g. large rectangles first in packing problems, here first-fail on a queens problem).

Example: `minisearch soft-queens.mzn`

```
solve :: seq_search([
        int_search([satisfied[mostImpFirst[i]] | i in SOFTCONSTRAINTS],
                   input_order, indomain_max, complete),
        int_search(
          queens,
          first_fail,
          indomain_median,
          complete
      )] )
    minimize penSum;
```

Redundant Constraints

Institute for Software & Systems Engineering

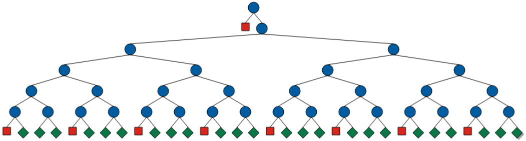
If we are mostly interested in performance (for constraint-relationship-based search), we can add *redundant constraints*, if we use *strictlyBetter* search.

From `redundant-constraints.mzn`:

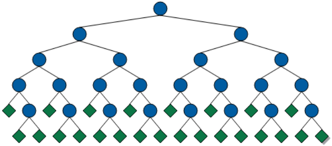
```
constraint spd_worse({1}, violatedScs,SOFTCONSTRAINTS,crEdges);
% if we need to find an actually better solution than {1}
% (with violation 2), the penalty sum has to be strictly
% less than 2
constraint penSum < 2;
```

Redundant Constraints

Without redundant constraint (`redundant-constraints.mzn`):



With redundant weight constraint – less failures (`redundant-constraints-weighted.mzn`):



Custom Search

Search strategies based on PVS are defined in `soft_constraints/pvs_search.mzn`.

Currently supported:

- Branch and bound (BAB)
- Large Neighborhood Search (LNS)

for *strictly better* and *only not dominated*

```
%[... in m2.mzn]
solve
search strictlyBetterBAB(violatedScs) /\ print();

%[... in m2-lns.mzn]
solve
search lns_pvs(violatedScs, dummies, 2, 0.5) /\ print();
```

Summary

This concludes our overview of some language features

Make sure to check out our other slides about:

- Step-by-Step enhancing a MiniZinc model (establishes the core elements)
- Case Studies (for some specific examples)

`http://isse-augsburg.github.io/constraint-relationships/`

References I

Bistarelli, S., Montanari, U., Rossi, F., Schiex, T., Verfaillie, G., and Fargier, H. (1999).
Semiring-Based CSPs and Valued CSPs: Frameworks, Properties, and Comparison.
Constraints, 4(3):199–240.

Gadducci, F., Hölzl, M., Monreale, G., and Wirsing, M. (2013).
Soft constraints for lexicographic orders.
In Castro, F., Gelbukh, A., and González, M., editors, *Proc. 12th Mexican Int. Conf. Artificial Intelligence (MICAI'2013)*, Lect. Notes Comp. Sci. 8265, pages 68–79. Springer.