



Constraint-based Scheduling & Packing

including Constraint Relationships

Bisher:

- Mechanismen für eigennützige Agenten
- Optimierungsprobleme bei Auszahlung
- Systematische Ansätze für SOAS
- RIA: Zustände über Constraints beschrieben

In dieser Vorlesung:

- Prinzipien Diskreter Optimierung
- Constraint Programming
- Propagationen
- Suchstrategien + Heuristiken

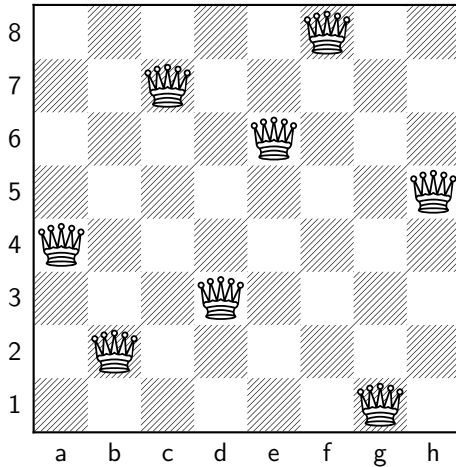
- Generischer Ansatz zur Lösung von **Erfüllbarkeitsproblemen**
- Ausnützen von **Struktur** von logischen Bedingungen (?)
- Konzentration auf *endliche* Wertebereiche und Erfüllbarkeit (?, Kap. 5)

- Generischer Ansatz zur Lösung von **Erfüllbarkeitsproblemen**
- Ausnützen von **Struktur** von logischen Bedingungen (?)
- Konzentration auf *endliche* Wertebereiche und Erfüllbarkeit (?, Kap. 5)
 - Im Gegensatz z.B. zu *linearer Programmierung*, *konvexe Optimierung*
 - Verallgemeinert Boolesche Erfüllbarkeitsprobleme (SAT)
 - Scheduling-Probleme
 - Reorganisationen
 - Zuweisungsprobleme (z.B. Frequenzen an Sender, Energie an Produzenten, ...)

- Generischer Ansatz zur Lösung von **Erfüllbarkeitsproblemen**
- Ausnützen von **Struktur** von logischen Bedingungen (?)
- Konzentration auf *endliche* Wertebereiche und Erfüllbarkeit (?, Kap. 5)
 - Im Gegensatz z.B. zu *linearer Programmierung*, *konvexe Optimierung*
 - Verallgemeinert Boolesche Erfüllbarkeitsprobleme (SAT)
 - Scheduling-Probleme
 - Reorganisationen
 - Zuweisungsprobleme (z.B. Frequenzen an Sender, Energie an Produzenten, ...)
- Deklarativ, aber Constraints sind an Algorithmen geknüpft (!)?

Ein prototypisches CSP





Definition (Constraint-Problem)

Ein **Constraint-Problem** (X, D, C) ist beschrieben durch

- *Variablen* X , *Domänen* $D = (D_x)_{x \in X}$ (endlich), *Constraints* C

Definition (Constraint-Problem)

Ein **Constraint-Problem** (X, D, C) ist beschrieben durch

- Variablen X , Domänen $D = (D_x)_{x \in X}$ (endlich), Constraints C

Technisch ist ein Constraint $c \in C$ mit $\text{Scope } \text{sc}(c) \subseteq X$ eine Einschränkung auf gültige Tupel des kartesischen Produktes der Domänen des Scopes:

$$c \subseteq \prod_{x \in \text{sc}(c)} D_x.$$

Definition (Constraint-Problem)

Ein **Constraint-Problem** (X, D, C) ist beschrieben durch

- Variablen X , Domänen $D = (D_x)_{x \in X}$ (endlich), Constraints C

Technisch ist ein Constraint $c \in C$ mit $\text{Scope } \text{sc}(c) \subseteq X$ eine Einschränkung auf gültige Tupel des kartesischen Produktes der Domänen des Scopes:

$$c \subseteq \prod_{x \in \text{sc}(c)} D_x.$$

Constraints sind entweder:

- **Extensional**; direkte Auflistung der Menge (z.B. $\{1, 2\}, \{2, 3\}$)
- **Intensional**; Syntaktische Vorschrift gegeben (z.B. $x + 1 = y$)

Definition (Constraint-Problem)

Ein **Constraint-Problem** (X, D, C) ist beschrieben durch

- *Variablen* X , *Domänen* $D = (D_x)_{x \in X}$ (endlich), *Constraints* C

Technisch ist ein Constraint $c \in C$ mit *Scope* $sc(c) \subseteq X$ eine Einschränkung auf gültige Tupel des kartesischen Produktes der Domänen des Scopes:

$$c \subseteq \prod_{x \in sc(c)} D_x.$$

Constraints sind entweder:

- **Extensional**; direkte Auflistung der Menge (z.B. $\{1, 2\}, \{2, 3\}$)
- **Intensional**; Syntaktische Vorschrift gegeben (z.B. $x + 1 = y$)

Wir verwenden beide in *funktionaler* Form: $c : [X \rightarrow D] \rightarrow \mathbb{B} = \{\text{tt}, \text{ff}\}$, wobei $[X \rightarrow D]$ die Menge aller Abbildungen von X nach D ist.

Problem

CSP (X, D, C) mit

- $X = \{x, y, z\}$
- $D_x = D_y = \{0, 1, 2\}, D_z = \{0, 1\}$
- C
 - $c_1 : x \neq y, y \neq z, x \neq z$
 - $c_2 : x + 1 = y$

Problem

CSP (X, D, C) mit

- $X = \{x, y, z\}$
- $D_x = D_y = \{0, 1, 2\}, D_z = \{0, 1\}$
- C
 - $c_1 : x \neq y, y \neq z, x \neq z$
 - $c_2 : x + 1 = y$

Problem

CSP (X, D, C) mit

- $X = \{x, y, z\}$
- $D_x = D_y = \{0, 1, 2\}$, $D_z = \{0, 1\}$
- C
 - $c_1 : x \neq y, y \neq z, x \neq z$
 - $c_2 : x + 1 = y$

```
var 0..2: x;  
var 0..2: y;  
var 0..1: z;
```

```
% c1
```

```
constraint x != y /\ y != z /\ x != z;
```

```
% c2
```

```
constraint x + 1 = y;
```

```
solve satisfy;
```

Welche Zuweisung ist eine Lösung dieses Problems?

- $\Theta = \{(x \rightarrow 1, y \rightarrow 2, z \rightarrow ?), (x \rightarrow 0, y \rightarrow 1, z \rightarrow ?)\}$ erfüllen c_2 ;

Problem

CSP (X, D, C) mit

- $X = \{x, y, z\}$
- $D_x = D_y = \{0, 1, 2\}$, $D_z = \{0, 1\}$
- C
 - $c_1 : x \neq y, y \neq z, x \neq z$
 - $c_2 : x + 1 = y$

```
var 0..2: x;  
var 0..2: y;  
var 0..1: z;
```

```
% c1
```

```
constraint x != y /\ y != z /\ x != z;
```

```
% c2
```

```
constraint x + 1 = y;
```

```
solve satisfy;
```

Welche Zuweisung ist eine Lösung dieses Problems?

- $\Theta = \{(x \rightarrow 1, y \rightarrow 2, z \rightarrow ?), (x \rightarrow 0, y \rightarrow 1, z \rightarrow ?)\}$ erfüllen c_2 ;
- $(x \rightarrow 0, y \rightarrow 1, z \rightarrow ?)$ lässt sich aber zu keiner Lösung erweitern, da z entweder 0 oder 1 sein muss und somit garantiert c_1 verletzt

Problem

CSP (X, D, C) mit

- $X = \{x, y, z\}$
- $D_x = D_y = \{0, 1, 2\}$, $D_z = \{0, 1\}$
- C
 - $c_1 : x \neq y, y \neq z, x \neq z$
 - $c_2 : x + 1 = y$

```
var 0..2: x;  
var 0..2: y;  
var 0..1: z;
```

```
% c1
```

```
constraint x != y /\ y != z /\ x != z;
```

```
% c2
```

```
constraint x + 1 = y;
```

```
solve satisfy;
```

Welche Zuweisung ist eine Lösung dieses Problems?

- $\Theta = \{(x \rightarrow 1, y \rightarrow 2, z \rightarrow ?), (x \rightarrow 0, y \rightarrow 1, z \rightarrow ?)\}$ erfüllen c_2 ;
- $(x \rightarrow 0, y \rightarrow 1, z \rightarrow ?)$ lässt sich aber zu keiner Lösung erweitern, da z entweder 0 oder 1 sein muss und somit garantiert c_1 verletzt
- Also ist die einzige Lösung $(x \rightarrow 1, y \rightarrow 2, z \rightarrow 0)$

Wesentliche Kernrichtungen für Algorithmen:

- Systematische (vollständige) Suche (“Try”)
 - Backtracking
 - Branch & Bound

Wesentliche Kernrichtungen für Algorithmen:

- Systematische (vollständige) Suche (“Try”)
 - Backtracking
 - Branch & Bound
- Constraint-Propagation, Inferenz
 - Einfache, lokale Konsistenzchecks (*Logische Schlüsse*)
 - Reduktion der Domänen

Wesentliche Kernrichtungen für Algorithmen:

- **Systematische** (vollständige) Suche (“**Try**”)
 - Backtracking
 - Branch & Bound
- Constraint-**Propagation**, Inferenz
 - Einfache, lokale Konsistenzchecks (*Logische Schlüsse*)
 - **Reduktion** der *Domänen*
- **Relaxierung**
 - Löse einfachere Teilprobleme
 - Nehme Ergebnis als *Schranken*

Wesentliche Kernrichtungen für Algorithmen:

- **Systematische** (vollständige) Suche (“Try”)
 - Backtracking
 - Branch & Bound
- Constraint-**Propagation**, Inferenz
 - Einfache, lokale Konsistenzchecks (*Logische Schlüsse*)
 - **Reduktion** der *Domänen*
- **Relaxierung**
 - Löse einfachere Teilprobleme
 - Nehme Ergebnis als *Schranken*
- **Lokale** (*heuristische*) Suche
 - *Min-Conflicts*-Heuristik
 - Large-neighborhood Search
 - Tabu-Suche / Simulated Annealing

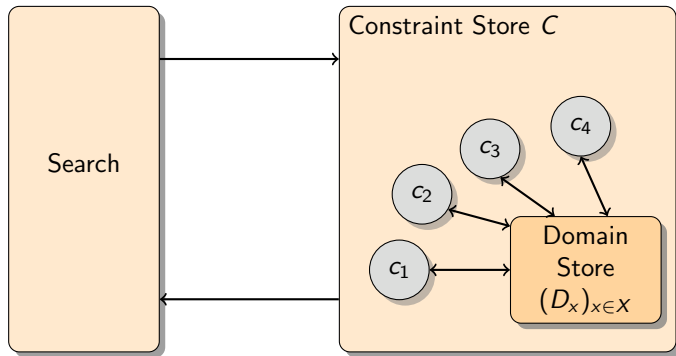
Partielle Zuweisungen schrittweise um ein Variablen-Wert-Paar erweitert.

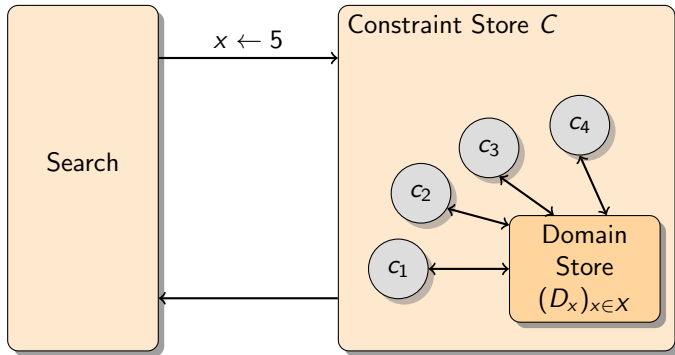
Ausnützen der Konjunktivität: Wenn eine partielle Zuweisung bereits **einen** Constraint verletzt, wird die letzte Zuweisung **rückgängig** gemacht (*backtracking*) und neuer Wert versucht.

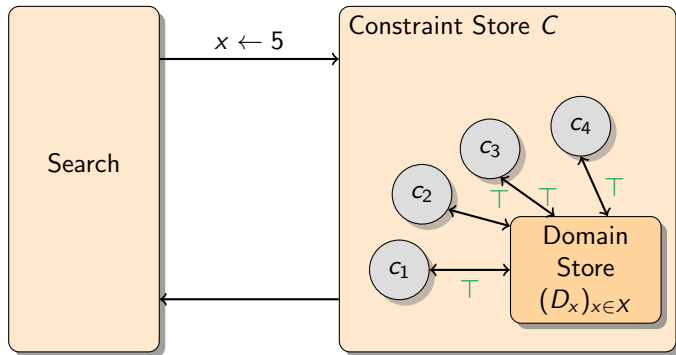
Im schlimmsten Fall exponentielle Exploration aller vollständigen Zuweisungen $O(|D|^{|X|})$.

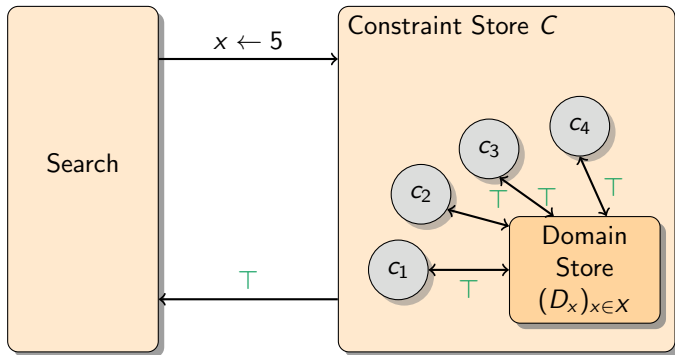
→ in der Praxis:

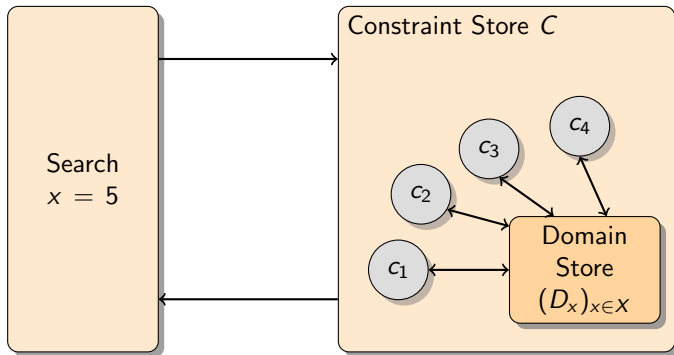
- **Einschränkung der Lösungsraums durch Propagation**
- Frühzeitiges Abschneiden von “**Sackgassen**”
- Frühzeitiges Probieren von **vielversprechenden** Kandidaten

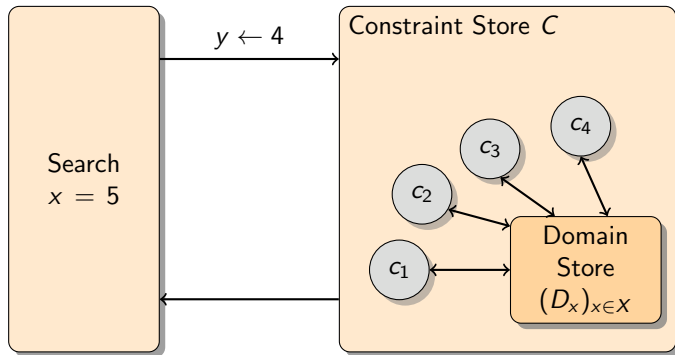


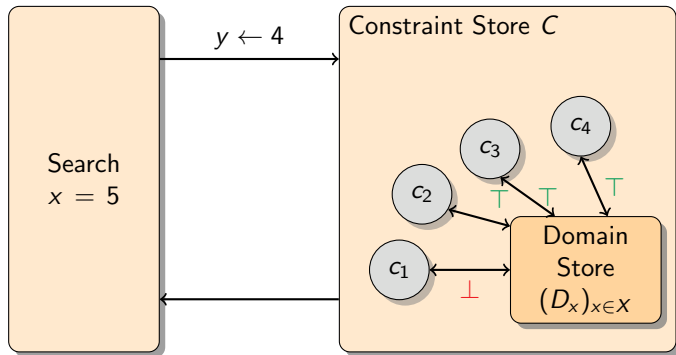


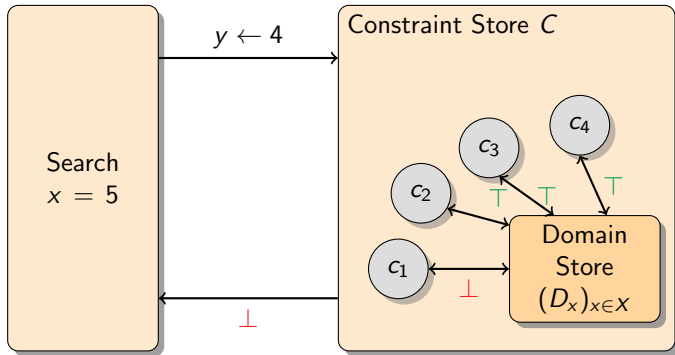








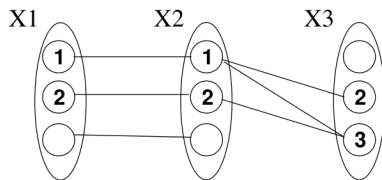
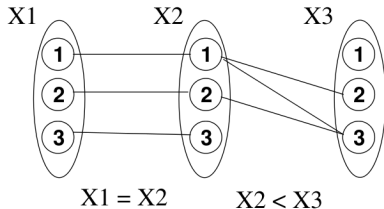




- Nutze Constraints, um Suchraum einzugrenzen
- Idee: Entferne alle Werte aus Domänen, die in keiner Lösung vorkommen können (*Domain Store*)

- Nutze Constraints, um Suchraum einzugrenzen
- Idee: Entferne alle Werte aus Domänen, die in keiner Lösung vorkommen können (*Domain Store*)
- $|x_1 - x_2| > 5$ für $x_1, x_2 \in X$ und $D_{x_i} = \{1, \dots, 10\}$
- Welche Werte sind nicht möglich?
- jeweils 5 und 6

- Nutze Constraints, um Suchraum einzugrenzen
- Idee: Entferne alle Werte aus Domänen, die in keiner Lösung vorkommen können (*Domain Store*)
- $|x_1 - x_2| > 5$ für $x_1, x_2 \in X$ und $D_{x_i} = \{1, \dots, 10\}$
- Welche Werte sind nicht möglich?
- jeweils 5 und 6
- Propagierungsschritte beeinflussen einander
 - “Kettenreaktion”
 - Fixpunktalgorithmus \rightarrow Keine Propagierung möglich
 - Wenn Domäne nur mehr einen Wert enthält, *muss* dieser zugewiesen werden.



Entfernen von Werten, die zu keiner Lösung führen können.

- Betrachten wir folgendes einfaches Problem
 - $X = \{x_1, x_2, x_3\}$
 - $(D_x)_{x \in X} = \{1, 2\}$
 - $C : x_1 \neq x_2, x_2 \neq x_3, x_1 \neq x_3$
- Ist dieses Problem nach Constraint-Propagation mit binären Constraints lösbar?

- Betrachten wir folgendes einfaches Problem
 - $X = \{x_1, x_2, x_3\}$
 - $(D_x)_{x \in X} = \{1, 2\}$
 - $C : x_1 \neq x_2, x_2 \neq x_3, x_1 \neq x_3$
- Ist dieses Problem nach Constraint-Propagation mit binären Constraints lösbar?
- Ja, für jedes $d \in D_x$ gibt es einen Partner

- Betrachten wir folgendes einfaches Problem
 - $X = \{x_1, x_2, x_3\}$
 - $(D_x)_{x \in X} = \{1, 2\}$
 - $C : x_1 \neq x_2, x_2 \neq x_3, x_1 \neq x_3$
- Ist dieses Problem nach Constraint-Propagation mit binären Constraints lösbar?
- Ja, für jedes $d \in D_x$ gibt es einen Partner
- Insgesamt allerdings nicht, da mindestens 3 unterschiedliche Werte nötig
- \rightarrow daher globale Constraints, die eine größere Menge von Variablen im Auge betrachten können
- Und *spezialisierte* Propagationsalgorithmen haben!
- `alldifferent(x_1, x_2, x_3)`

Häufig auftretende Constraints samt Propagierungsalgorithmen. Gesammelt im *Global Constraints Catalogue*¹

`alldifferent(VARIABLES)` Erfordert, dass alle Variablen **unterschiedliche** Werte haben, z.B: [1, 2, 3]

¹<http://www.emn.fr/z-info/sdemasse/gccat/sec5.html>

Häufig auftretende Constraints samt Propagierungsalgorithmen. Gesammelt im *Global Constraints Catalogue*¹

`alldifferent(VARIABLES)` Erfordert, dass alle Variablen **unterschiedliche** Werte haben, z.B: [1, 2, 3]

`allexact(VARIABLES)` Erfordert, dass alle Variablen den **gleichen** Wert haben, z.B: [5, 5, 5]

¹<http://www.emn.fr/z-info/sdemasse/gccat/sec5.html>

Häufig auftretende Constraints samt Propagierungsalgorithmen. Gesammelt im *Global Constraints Catalogue*¹

`alldifferent(VARIABLES)` Erfordert, dass alle Variablen **unterschiedliche** Werte haben, z.B: [1, 2, 3]

`allexact(VARIABLES)` Erfordert, dass alle Variablen den **gleichen** Wert haben, z.B: [5, 5, 5]

`nvalue(VARIABLES, N)` Erfordert, dass genau *N* verschiedene Werte vorkommen, z.B: ([1, 2, 1], 2)

¹<http://www.emn.fr/z-info/sdemasse/gccat/sec5.html>

Häufig auftretende Constraints samt Propagierungsalgorithmen. Gesammelt im *Global Constraints Catalogue*¹

`alldifferent(VARIABLES)` Erfordert, dass alle Variablen **unterschiedliche** Werte haben, z.B: [1, 2, 3]

`allexact(VARIABLES)` Erfordert, dass alle Variablen den **gleichen** Wert haben, z.B: [5, 5, 5]

`nvalue(VARIABLES, N)` Erfordert, dass genau N verschiedene Werte vorkommen, z.B: ([1, 2, 1], 2)

`at_most(N, VARIABLES, V)` Erfordert, dass höchstens N Variablen den Wert V annehmen z.B: (2, [2, 2, 4], 2)

¹<http://www.emn.fr/z-info/sdemasse/gccat/sec5.html>

Constraint-Problem (X, D, C)

- Variablen X , Domänen $D = (D_x)_{x \in X}$, Constraints C

In der Praxis: überbestimmte Probleme

$((\{x, y, z\}, D_x = D_y = D_z = \{1, 2, 3\}), \{c_1, c_2, c_3\})$ mit

$$c_1 : x + 1 = y$$

$$c_2 : z = y + 2$$

$$c_3 : x + y \leq 3$$

- Nicht alle Constraints können gleichzeitig erfüllt werden

Constraint-Problem (X, D, C)

- Variablen X , Domänen $D = (D_x)_{x \in X}$, Constraints C

In der Praxis: überbestimmte Probleme

$((\{x, y, z\}, D_x = D_y = D_z = \{1, 2, 3\}), \{c_1, c_2, c_3\})$ mit

$$c_1 : x + 1 = y$$

$$c_2 : z = y + 2$$

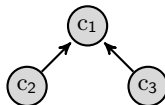
$$c_3 : x + y \leq 3$$

- Nicht alle Constraints können gleichzeitig erfüllt werden
 - e.g., c_2 erzwingt $z = 3$ und $y = 1$, im Konflikt mit c_1
- Ein Agent wählt zwischen Zuweisungen, die $\{c_1, c_3\}$ oder $\{c_2, c_3\}$ erfüllen.

Welche Zuweisungen $v \in [X \rightarrow D]$ sollen bevorzugt werden von einem Agenten (oder sogar einer Menge von Agenten)?

Ansatz (?)

- Definiere Relation R über Constraints C um anzugeben, welche Constraints wichtiger sind als andere, e. g.
 - c_1 wichtiger als c_2
 - c_1 wichtiger als c_3



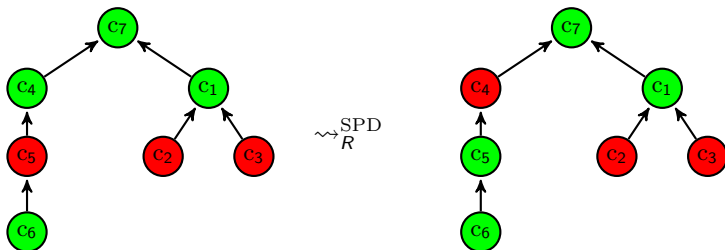
Benefits

- Qualitativer** Formalismus — einfach zu spezifizieren
 - Hebe diese Relation auf Verletzungsmengen
 - Dominanzeigenschaften** regulieren den Tradeoff “Hierarchie vs. Egalitär”
 - Single-Predecessors-Dominance (SPD) vs. Transitive-Predecessors-Dominance (TPD)

Worsening-Relation für Mengen verletzter Constraints

$$V \rightsquigarrow_R^{\text{SPD}} V \uplus \{c\}$$

$$V \uplus \{c\} \rightsquigarrow_R^{\text{SPD}} V \uplus \{c'\} \quad \text{if } c \rightarrow_R c'$$



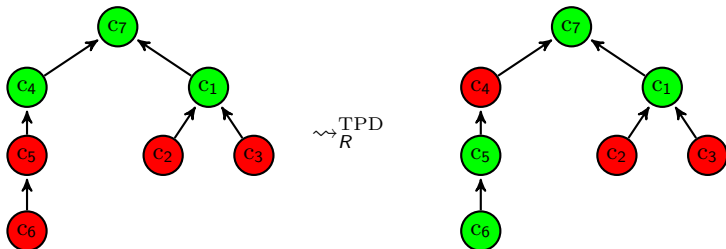
Ordnungsrelation über Zuweisungen

$$w <_R^{\text{SPD}} v \iff \{c \in C \mid v \not\models c\} (\rightsquigarrow_R^{\text{SPD}})^+ \{c \in C \mid w \not\models c\}$$

Worsening-Relation für Mengen verletzter Constraints

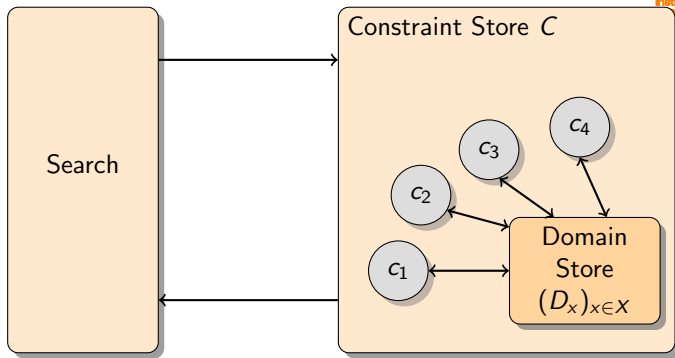
$$V \rightsquigarrow_R^{\text{TPD}} V \uplus \{c\}$$

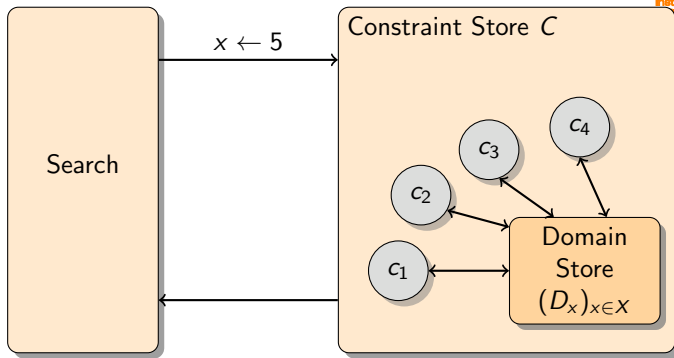
$$V \uplus \{c_1, \dots, c_k\} \rightsquigarrow_R^{\text{TPD}} V \uplus \{c'\} \quad \text{if } \forall c \in \{c_1, \dots, c_k\}. c \rightarrow_R^+ c'$$

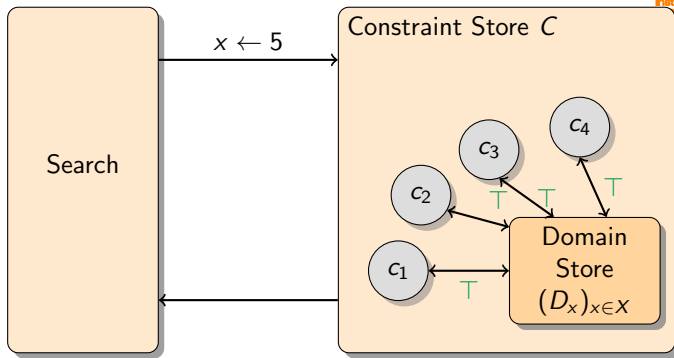


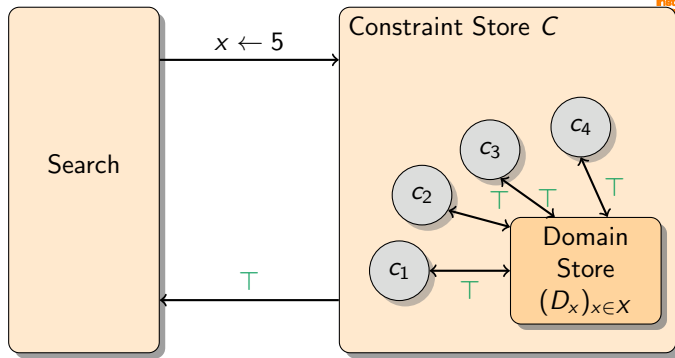
Ordnungsrelation über Zuweisungen

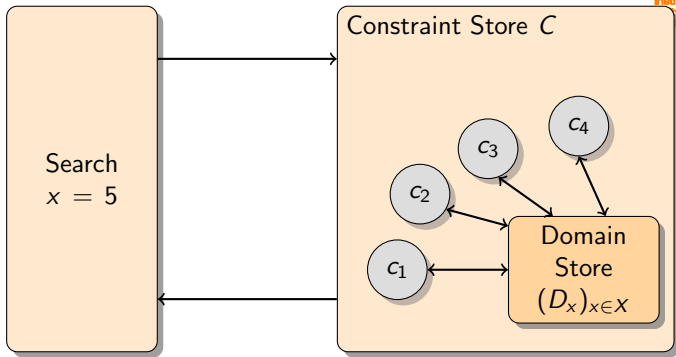
$$w <_R^{\text{TPD}} v \iff \{c \in C \mid v \not\models c\} (\rightsquigarrow_R^{\text{TPD}})^+ \{c \in C \mid w \not\models c\}$$

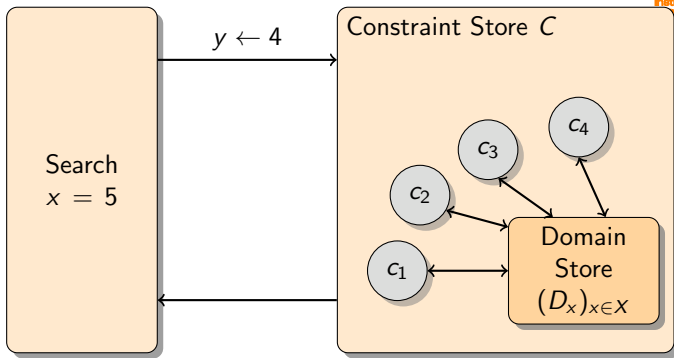


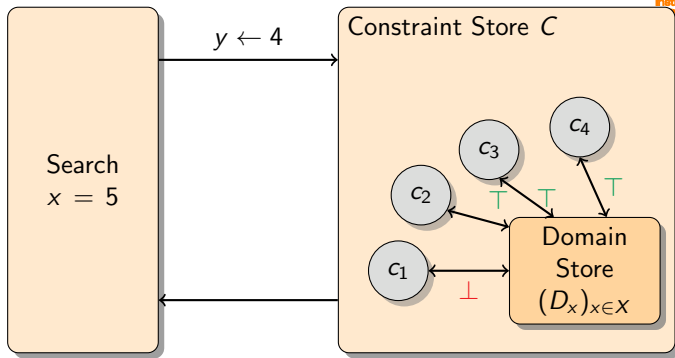


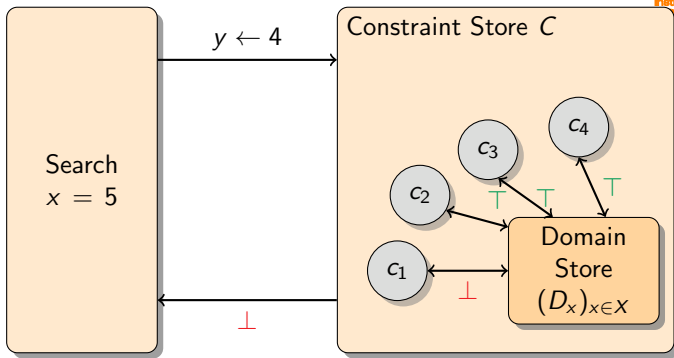


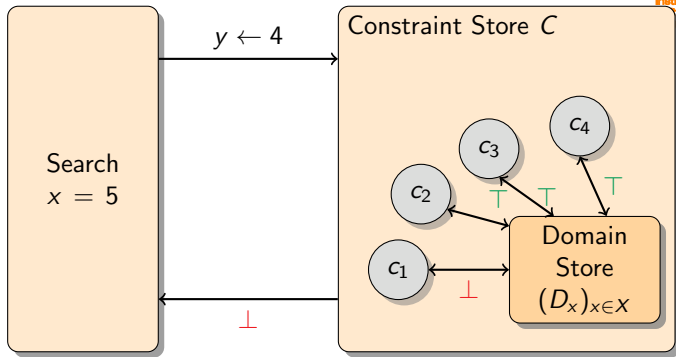




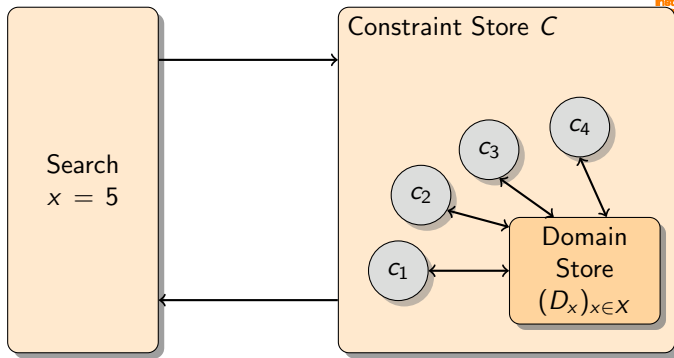


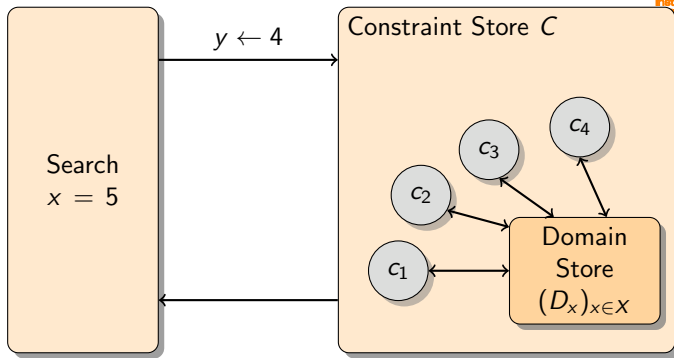


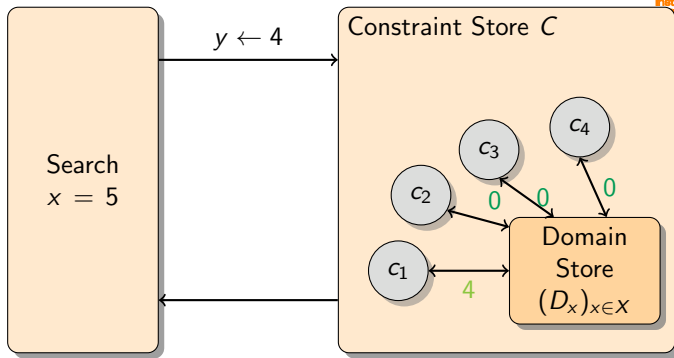


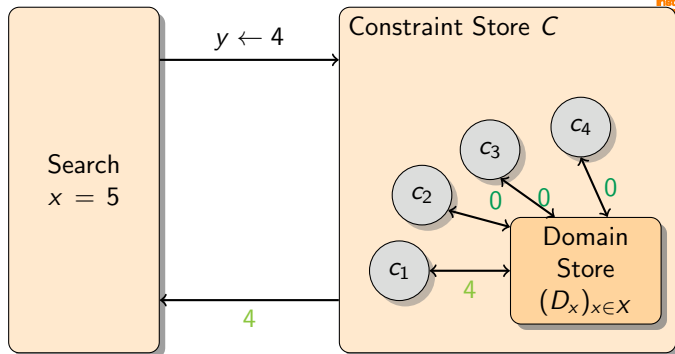


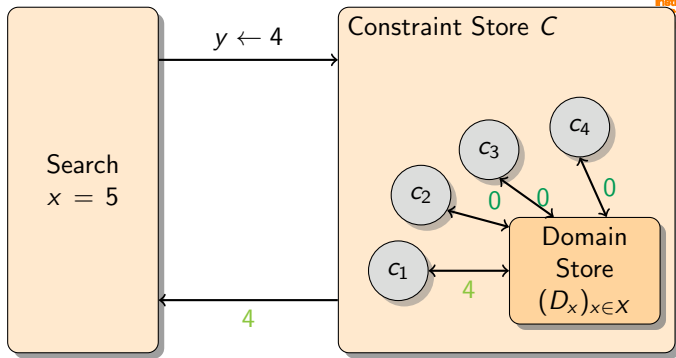
- Eine Menge von Erfüllungsgraden,
 $\mathbb{B} = \{\perp, \top\}$
- Eine Kombinationsoperation \wedge
- Ein neutrales Element \top
- Eine partielle Ordnung $(\mathbb{B}, \leq_{\mathbb{B}})$ mit
 $\top <_{\mathbb{B}} \perp$











- Eine Menge von Erfüllungsgraden, e.g., $\{0, \dots, k\}$
- Eine Kombinationsoperation $+$
- Ein neutrales Element 0
- Eine partielle Ordnung (\mathbb{N}, \geq) mit 0 als Top

Eine **valuation structure** (?),
wenn die Ordnung total ist,
sonst eine **partial valuation
structure** (?) (PVS).

```
% 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";

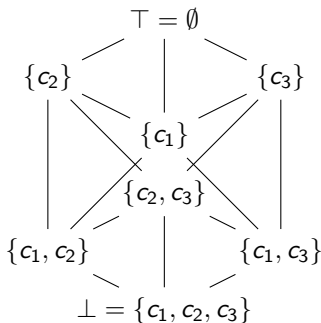
var 1..3: x; var 1..3: y; var 1..3: z;

% 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
```

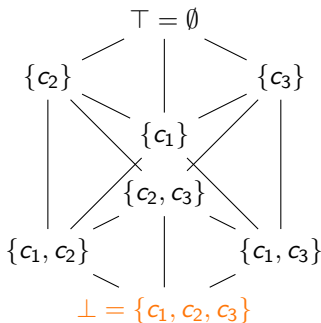
The whole valuation space (partially ordered)



```
%  
% 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

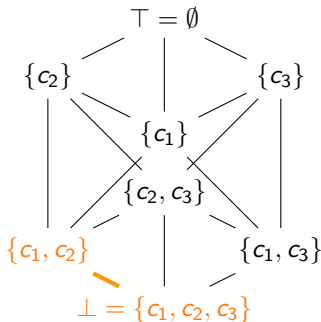
The whole valuation space (partially ordered)



```
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

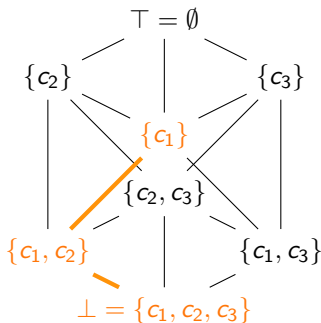
The whole valuation space (partially ordered)



```
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

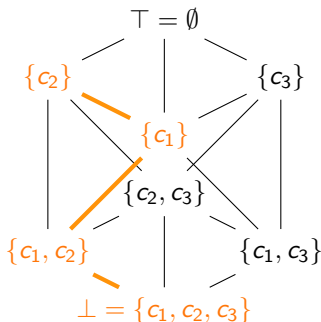
The whole valuation space (partially ordered)



```
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

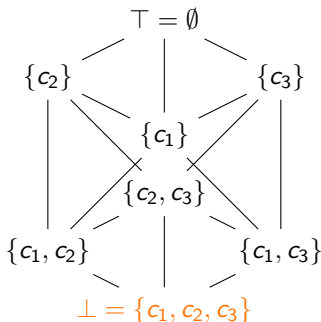
The whole valuation space (partially ordered)



```
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

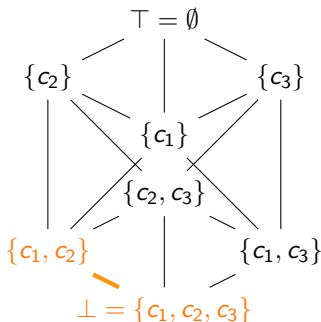
The whole valuation space (partially ordered)



```
function ann: strictlyBetterBAB(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

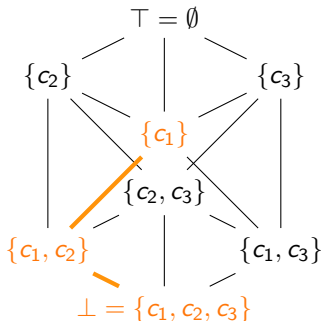
The whole valuation space (partially ordered)



```
function ann: strictlyBetterBAB(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

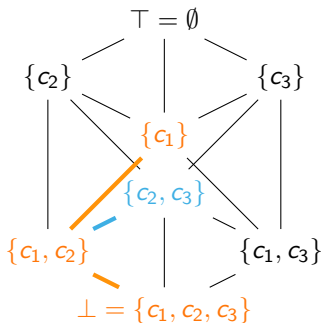
The whole valuation space (partially ordered)



```
function ann: strictlyBetterBAB(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

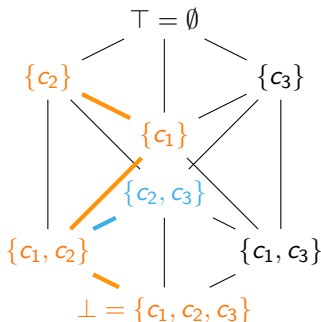
The whole valuation space (partially ordered)



```
function ann: strictlyBetterBAB(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

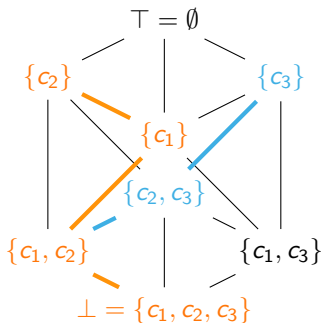
The whole valuation space (partially ordered)



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

Applied to domains where

- Certain properties should really capture **preferences**, not constraints
- at design time, it is **unclear** whether an instance is actually solvable
- Solution space is *combinatorial*
 - Discrete choices
 - Additional hard constraints

Illustrative case studies

- Mentor Matching
- Exam Scheduling
- Power Plant Scheduling
- Multi-User Multi-Displays

```
int: n; set of int: STUDENT = 1..n;
int: m; set of int: COMPANY = 1..m;

% assign students to companies
array[STUDENT] of var COMPANY: worksAt;

% insert relationships of students and companies here

int: minPerCompany = 2; int: maxPerCompany = 3;
constraint global_cardinality_low_up (
    worksAt, [c | c in COMPANY],
    [minPerCompany | c in COMPANY],
    [maxPerCompany | c in COMPANY]);

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

```
n = 3; m = 3;
int: brenner = 1;
int: teufel = 2;
int: fennek = 3;

int: cupgainini = 1;
int: gsm = 2;
int: junedied = 3;

% specify soft constraints, order by relationship
constraint worksAt[teufel] = junedied <-> satisfied[teufJune];
constraint worksAt[teufel] = cupgainini <-> satisfied[teufCap];
constraint worksAt[teufel] = gsm <-> satisfied[teufGsm];

constraint worksAt[fennek ] in {cupgainini, gsm} <-> satisfied[fenFavs];
constraint worksAt[fennek ] in {junedied} <-> satisfied[fenOK];

crEdges = [| teufGsm, teufCap | teufGsm, teufJune
           | fenOK, fenFavs |];
```

Split company and student preferences:

```
% first, our students' preferences
var int: penStud = sum(sc in 1..lastStudentPref)
    (bool2int(not satisfied[sc]) * penalties[sc]);

% now companies' preferences
var int: penComp = sum(sc in lastStudentPref+1..nScs)
    (bool2int(not satisfied[sc]) * penalties[sc]);
```

Optimize lexicographically

```
solve
:: int_search([ satisfied[mostImpFirst[i]] | i in SOFTCONSTRAINTS],%...
%search minimize_lex([penStud, penComp]) /\ if % ...
search minimize_lex([penComp, penStud]) /\ if % ...
```

Mentor Matching: Priority Example

Taken from example: student-company-matching.mzn

```
solve
:: int_search([ satisfied[mostImpFirst[i]] | i in SOFTCONSTRAINTS],%...
search minimize_lex([penStud, penComp]) /\ if %...
```

```
worksAt = [1, 3, 2, 3], penalty Students: 8, penalty Companies: 6
-----
=====
```

```
solve
:: int_search([ satisfied[mostImpFirst[i]] | i in SOFTCONSTRAINTS],%...
search minimize_lex([penStud, penComp]) /\ if %...
```

```
worksAt = [1, 3, 1, 2], penalty Students: 10, penalty Companies: 4
-----
=====
```

Here, company 1 (cupgainini) wanted to have student 3, and company 2 (APS) did not have any preferences whatsoever (so accepted student 4 instead of 3). Student 4 would have liked company 3 (junedied) better, though.

- Collected data from winter term

Example

"the favorites":

1. JuneDied-Lynx- HumanIT
2. Cupgainini

"I could live with that":

3. Seamless-German
4. gsm systems
5. Yiehlke

"I think, we won't be happy":

6. APS
7. Delphi Databases

- Gave precedence to **students**
 - After all, what should companies do with unhappy students?
- Search space: 7 companies for 16 students $\rightarrow 7^{16} = 3.3233 \cdot 10^{13}$
- Led to a constraint problem with
 - 77 student preferences (soft constraints) from 16 students
 - of a total of 114 soft constraints (37 company preferences)
- *Proved* optimal solution
 - 4 minutes compilation
 - another 2m 12s solving time

a

```
include "soft_constraints/soft_constraints_noset.mzn";
include "soft_constraints/cr_types.mzn";
include "soft_constraints/cr_weighting.mzn";
% ground penalties using the appropriate weighting
penalties = [weighting(s, SOFTCONSTRAINTS, crEdges, true)
              | s in SOFTCONSTRAINTS];

int: T = 5; set of int: WINDOW = 1..T;
array[WINDOW] of float: demand = [10.0, 11.3, 15.2, 20.7, 19.2];

int: P = 3; set of int: PLANTS = 1..P;
array[PLANTS] of float: pMin = [12.0, 5.0, 7.3];
array[PLANTS] of float: pMax = [15.0, 11.3, 9.7];

array[WINDOW, PLANTS] of var 0.0..15.0: supply;
var float: obj;

% minimize the absolute error between supply and demand
constraint obj = sum(w in WINDOW) ( abs( sum(p in PLANTS)
                                         (supply[w, p]) - demand[w] ) );
```

```
% ground penalties using the appropriate weighting
penalties = [weighting(s, SOFTCONSTRAINTS, crEdges, true)
             | s in SOFTCONSTRAINTS];
[...]

% some soft constraints
constraint supply[1, 2] >= 6.0 <-> satisfied[1];
constraint supply[2, 2] >= 6.0 <-> satisfied[2];

% constraint time step 1 seems more urgent
nCrEdges = 1;
crEdges = [| 2, 1 |];

% could do something more sophisticated here
solve minimize obj + penSum;
```

a

a

a

a

a

a

a

