Constraint-based Scheduling & Packing

including Constraint Relationships

Universität Augsburg
University

Agenda

Institute for Software & Systems Engineering

Grundlagen:

- Constraint Satisfaction (Optimisation) Problems
- Funktionsweise von Constraint-Lösern und Sprachen (MiniZinc)
- Einsatz im Scheduling

ISSE-Entwicklungen:

- Constraint Relationships für Soft Constraints
- Entwickelte Fallstudien
- Sprachunterstützung / Features

Constraint Programming: Einordnung

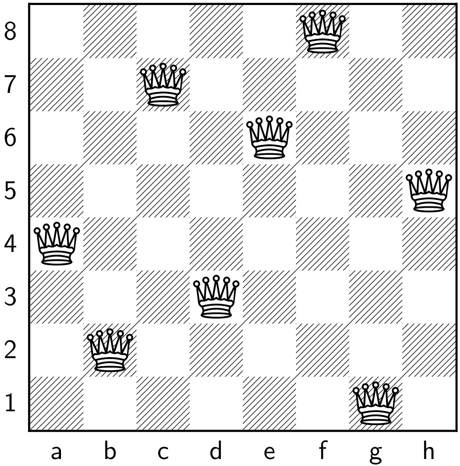
Institute for Software & Systems Engineering

- 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 and Produzenten, ...)
- Deklarativ, aber Constraints sind an Algorithmen geknüpft (**?**)!

Ein prototypisches CSP

Institute for
Software & Systems
Engineering

Und noch eins



Institute for Software & Systems Engineering

Definition (Constraint-Problem)

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

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

Domänen typischerweise: `int`, `bool`, (`float`)

Was ist ein **Constraint**? Ein *boolesches* Prädikat über einer Belegung von X:
$X = \{x, y, z\}$

- $x < y$
- $x + 5 = z - y$
- `alldifferent`$([x, y, z])$
- $\forall$ und $\exists$ nur über endlichen Wertebereichen

Beispiel

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 !=
% 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 $(x \rightarrow 1, y \rightarrow 2, z \rightarrow 0)$

Constraint-Algorithmen

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

Systematische Suche

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

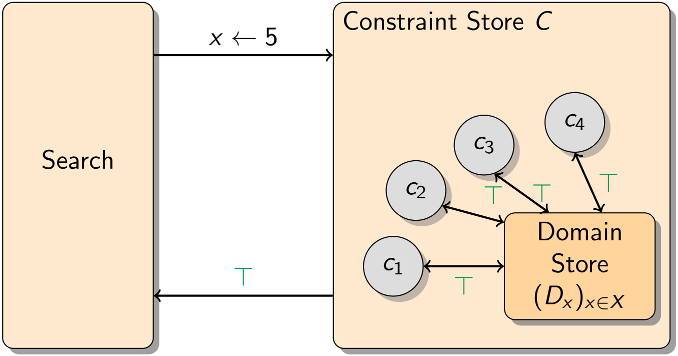
Ausnützen der Konjunktivität: Wenn eine partielle Zuweisung bereits einen Constraint verletzt, wird die letzte Zuweisung rückgängig gemacht (*backtracking*) und neuer Wert versucht.

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

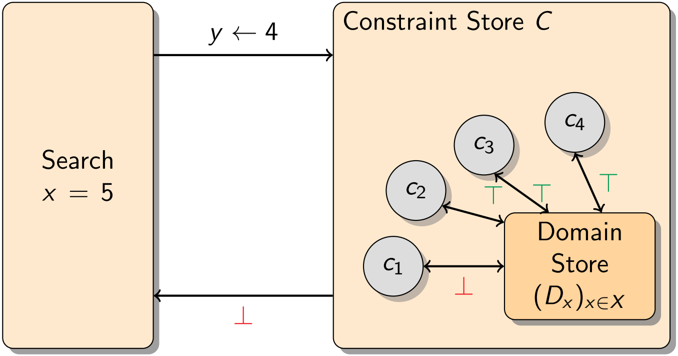
$\rightarrow$ in der Praxis:

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

Architektur von Constraint-Lösern



Architektur von Constraint-Lösern

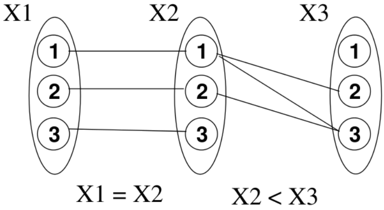


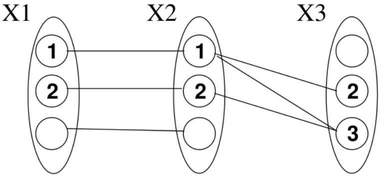
Constraint-Propagation

- 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 $D_{x_i} = \{1, \ldots, 10\}$
- 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.

Constraint-Propagation: Beispiel

Institute for Software & Systems Engineering





Entfernen von Werten, die zu keiner Lösung führen können.

Globale Constraints

- 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!
- $\texttt{alldifferent}(\mathtt{x}_1, \mathtt{x}_2, \mathtt{x}_3)$

Task-Zuweisung in Practice

- Taskzuweisungsproblem (*task allocation problem*)
  - n Roboter
  - m Tasks
  - Gebe jedem Roboter einen *unterschiedlichen* Task, um den Gewinn zu maximieren (Unterschied zu Kompensation und Strafe)
- Beispielproblem:
  - $n = 4$, $m = 5$

	t1	t2	t3	t4	t5
r1	7	1	3	4	6
r2	8	2	5	1	4
r3	4	3	7	2	5
r4	3	1	6	3	6

Task-Zuweisung: Modell

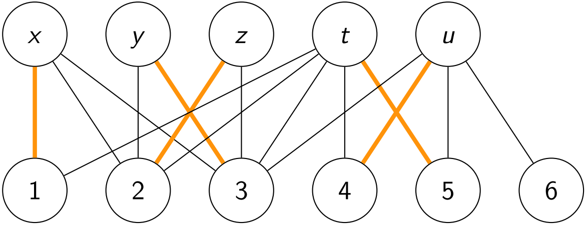
```
% problem data
int: n; set of int: ROBOTS = 1..n;
int: m; set of int: TASKS = 1..m;
array[ROBOTS,TASKS] of int: profit;

% decisions
array[ROBOTS] of var TASKS: allocation;

% goal
solve maximize sum(r in ROBOTS) (profit[r, allocation[r]] );

% have robots work on different tasks
constraint alldifferent(allocation);
```

AllDifferent – Machbarkeit

Institute for Software & Systems Engineering

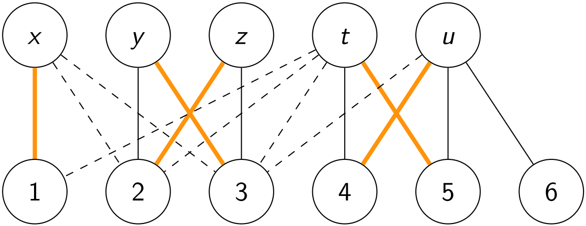
- Erster bekannter Propagator
- Basiert auf Matching in bipartiten Graphen
- Laufzeit ist polynomiell!

```
var {1,2,3}: x;     var {2,3}: y;    var {2,3}: z;
var {1,2,3,4,5}: t; var {3,4,5,6}: u;
constraint alldifferent([x,y,z,t,u]);
```



AllDifferent – Propagierung

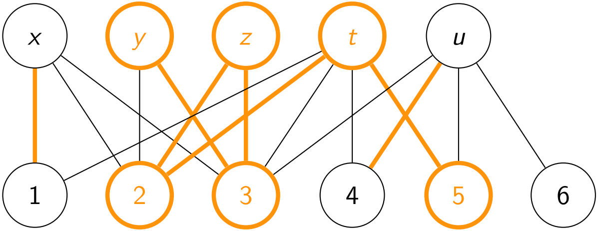
```
var {1,2,3}: x;     var {2,3}: y;    var {2,3}: z;
var {1,2,3,4,5}: t; var {3,4,5,6}: u;
constraint alldifferent([x,y,z,t,u]);
```




```
var {1}: x;     var {2,3}: y;    var {2,3}: z;
var {4,5}: t;    var {4,5,6}: u;
constraint alldifferent([x,y,z,t,u]);
```

AllDifferent – Algorithmus

```
var {1,2,3}: x;     var {2,3}: y;    var {2,3}: z;
var {1,2,3,4,5}: t; var {3,4,5,6}: u;
constraint alldifferent([x,y,z,t,u]);
```

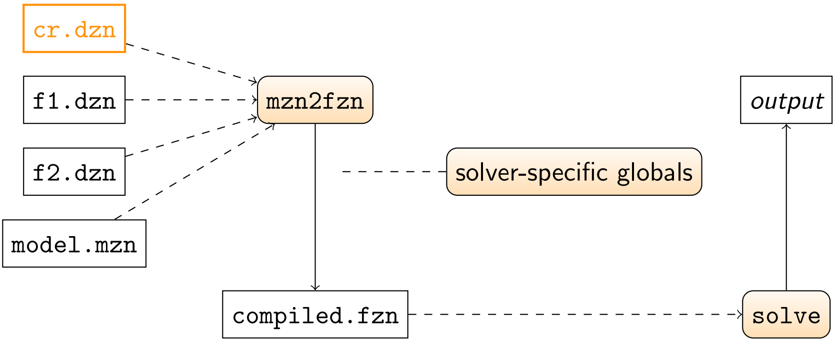


Augmentierender Pfad:
$y \to 3 \to z \to 2 \to t \to 5$

How can we actually use all that?

Institute for Software & Systems Engineering

Architecture



Scheduling & Packing

Institute for Software & Systems Engineering

Soft-Constraints

Präferenzen im Constraint Solving

Institute for Software & Systems Engineering

Constraint-Problem (X, D, C)

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

In der Praxis: überbestimmte Probleme

$((\{\mathrm{x}, \mathrm{y}, \mathrm{z}\}, \mathrm{D_x} = \mathrm{D_y} = \mathrm{D_z} = \{1, 2, 3\}), \{\mathrm{c}_1, \mathrm{c}_2, \mathrm{c}_3\})$ mit

$$\mathrm{c}_1 : \mathrm{x} + 1 = \mathrm{y}$$
$$\mathrm{c}_2 : \mathrm{z} = \mathrm{y} + 2$$
$$\mathrm{c}_3 : \mathrm{x} + \mathrm{y} \leq 3$$

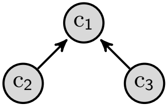
- Nicht alle Constraints können gleichzeitig erfüllt werden
  - e. g., c_2 erzwingt $\mathrm{z} = 3$ und $\mathrm{y} = 1$, im Konflikt mit c_1
- Ein Agent wählt zwischen Zuweisungen, die $\{\mathrm{c}_1, \mathrm{c}_3\}$ oder $\{\mathrm{c}_2, \mathrm{c}_3\}$ erfüllen.

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

Constraint Relationships

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



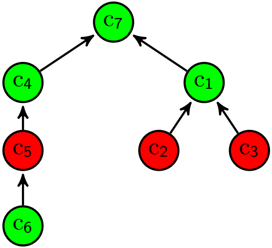
Benefits

- 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)

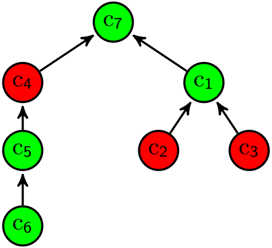
Single-Predecessor-Dominance (SPD) Lifting

Institute for
Software & Systems
Engineering

isWorseThan-Relation für Mengen verletzter Constraints



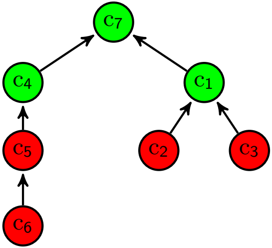
$\rightsquigarrow_{R}^{\mathrm{SPD}}$



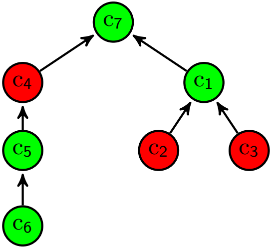
Transitive-Predecessors-Dominance (TPD) Lifting

Institute for Software & Systems Engineering

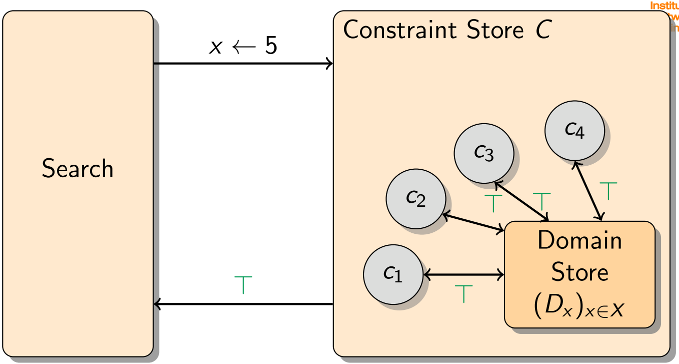
isWorseThan-Relation für Mengen verletzter Constraints



$\rightsquigarrow^{\mathrm{TPD}}_{R}$

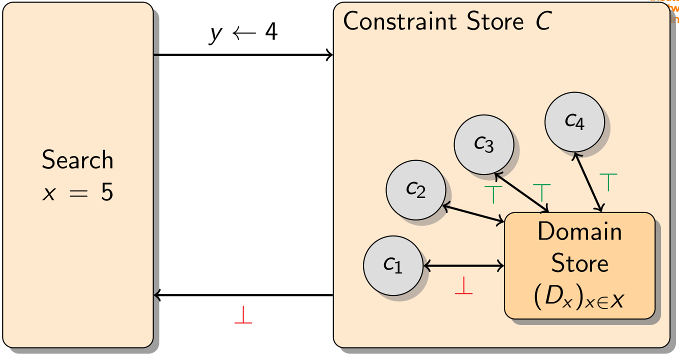


Traditional Constraint Solving



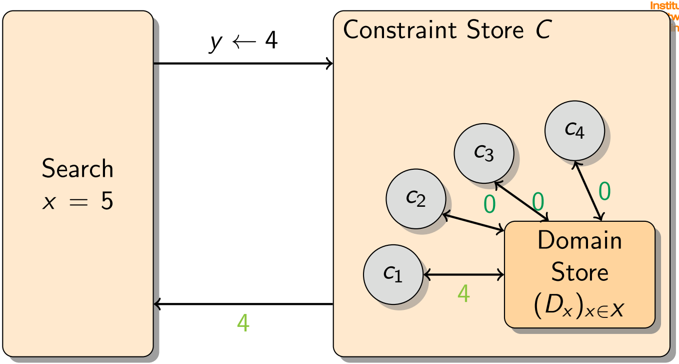
- Eine Menge von Erfüllungsgraden, $\mathbb{B} = \{\perp, \top\}$
- Eine Kombinationsoperation $\wedge$
- Ein neutrales Element $\top$
- Eine partielle Ordnung $(\mathbb{B}, \leq_{\mathbb{B}})$ mit $\top <_{\mathbb{B}} \perp$

Klassisches Constraint-Solving



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

Soft-Constraint-Solving



- Eine Menge von Erfüllungsgraden, e.g., $\{0, \ldots, k\}$
- 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).

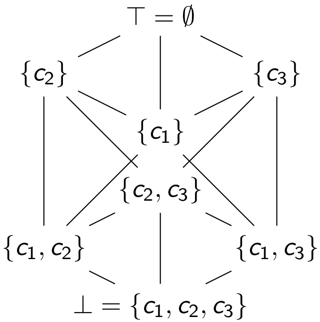
SoftConstraints in MiniZinc

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
```

Search types

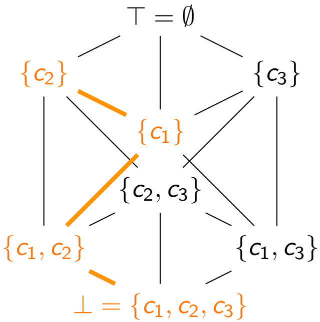
The whole valuation space (partially ordered)




```
%
% 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

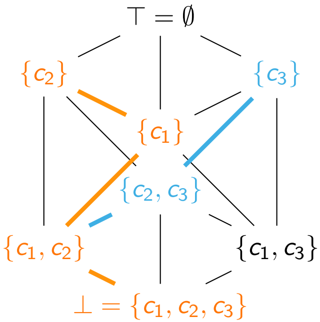
The whole valuation space (partially ordered)




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

Search types: 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     );
```

Case Studies

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

Mentor Matching: Model

```
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;
```

Mentor Matching: Preferences

```
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 |];
```

Mentor Matching: Refinements

Institute for Software & Systems Engineering

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.

Mentor Matching: Real Instance

- 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
```

Mentor Matching: Real Instance

Institute for Software & Systems Engineering

- 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

Exam Scheduling

Institute for Software & Systems Engineering

Goal: Assign exam dates to students such that

- Each student likes their appoints (*approves of it*)
- The number of distinct dates is minimized (to reduce time investment of teachers)

Illustrates some core ideas of constraint relationships:

- No preference of any student should be weighted higher than another one's
- Solution (exam schedule) is a shared decision

Exam Scheduling: Core Model

See `exam-scheduling-approval.mzn`:

```
% Exam scheduling example with just a set of
% approved dates and *impossible* ones
include "globals.mzn";
include "soft_constraints/soft_constraints.mzn";

int: n; set of int: STUDENT = 1..n;
int: m; set of int: DATE = 1..m;
array[STUDENT] of set of DATE: possibles;
array[STUDENT] of set of DATE: impossibles;

% the actual decisions
array[STUDENT] of var DATE: scheduled;

int: minPerSlot = 0; int: maxPerSlot = 4;
constraint global_cardinality_low_up(scheduled % minPerSlot, maxPerSlot
constraint forall(s in STUDENT) (not (scheduled[s] in impossibles[s]));
```

Exam Scheduling: Preferences

See `exam-scheduling-approval.mzn`:

```
% have a soft constraint for every student
nScs = n;
penalties = [ 1 | n in STUDENT]; % equally important in this case

constraint forall(s in STUDENT) (
     (scheduled[s] in possibles[s]) <-> satisfied[s] ) ;
var DATE: scheduledDates;
% constrains that "scheduledDates" different
% values (appointments) appear in "scheduled"
constraint nvalue(scheduledDates, scheduled);

% search variants
solve
:: int_search(satisfied, input_order, indomain_max, complete)
search minimize_lex([scheduledDates, violateds]); % pro teachers
%search minimize_lex([violateds, scheduledDates]); % pro students
```

Exam Scheduling: Real Instance

Institute for Software & Systems Engineering

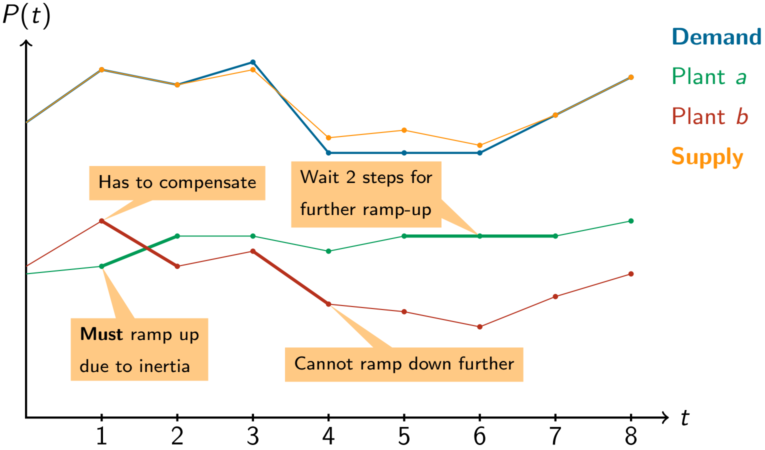
- Collected preferences of 33 students
- over 12 possible dates (6 days, morning and afternoon)
  - *Approval* set
  - *Impossible* set

- Aggregated via approval voting (has nice voting-theoretical properties!)
- At most 4 per appointment
- Immediately (61 msec) found an optimal solution that
  - Is approved by *every* student
  - Is achieved with the minimal number of 9 dates
- Used Strategy:

```
search minimize_lex([violateds, scheduledDates]); % pro students
```

Power Plant Scheduling

Goal: Schedule plants such that they meet the demand; See `unitCommitment.mzn`



Power Plant Scheduling: Core Model

```
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;

constraint obj = sum(w in WINDOW) ( abs( sum(p in PLANTS)
          (supply[w, p]) - demand[w] ) );
```

Power Plant Scheduling: Soft Constraints

```
% 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;
```

→ Library works with MIP (*Mixed Integer Programming*) as well!

Language Features

a

Language Features: Suitable Weighting

a

Language Features: Consistency Checks

Institute for Software & Systems Engineering

a

Language Features: Variable Ordering

a

Language Features: Redundant Constraints

Institute for Software & Systems Engineering

a

Language Features: Custom Search

Institute for Software & Systems Engineering

a

Quellen I

Institute for
Software & Systems
Engineering