Constraint-based Scheduling & Packing

including MiniBrass

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

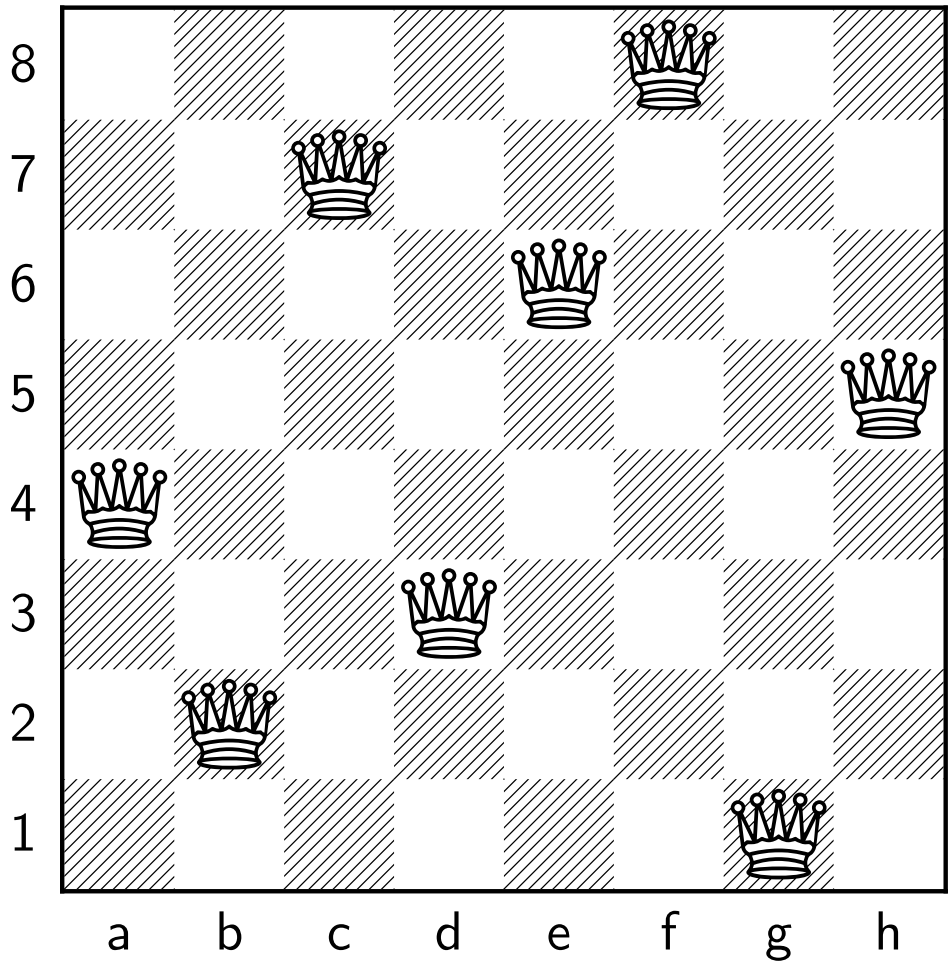
- Generischer Ansatz zur Lösung von Erfüllbarkeitsproblemen
- Ausnützen von **Struktur** von logischen Bedingungen (Tsang, 1993)
- Konzentration auf *endliche* Wertebereiche und Erfüllbarkeit (Russell and Norvig, 2010, Kap. 5)
  - Im Gegensatz z.B. zu *linearer Programmierung*, *konvexe Optimierung*
  - Verallgemeinert Boolesche Erfüllbarkeitsprobleme (SAT)
  - **Scheduling-Probleme**
  - Reorganisationen
  - Zuweisungsprobleme (z.B. Frequenzen an Sender, Energie and Produzenten, ...)
- Deklarativ, aber Constraints sind an Algorithmen geknüpft (Rossi et al., 2006)!

Ein prototypisches CSP

Institute for
Software & Systems
Engineering

Und noch eins

Institute for
Software & Systems
Engineering



Basics

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

Institute for Software & Systems

Problem

CSP (X, D, C) mit

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

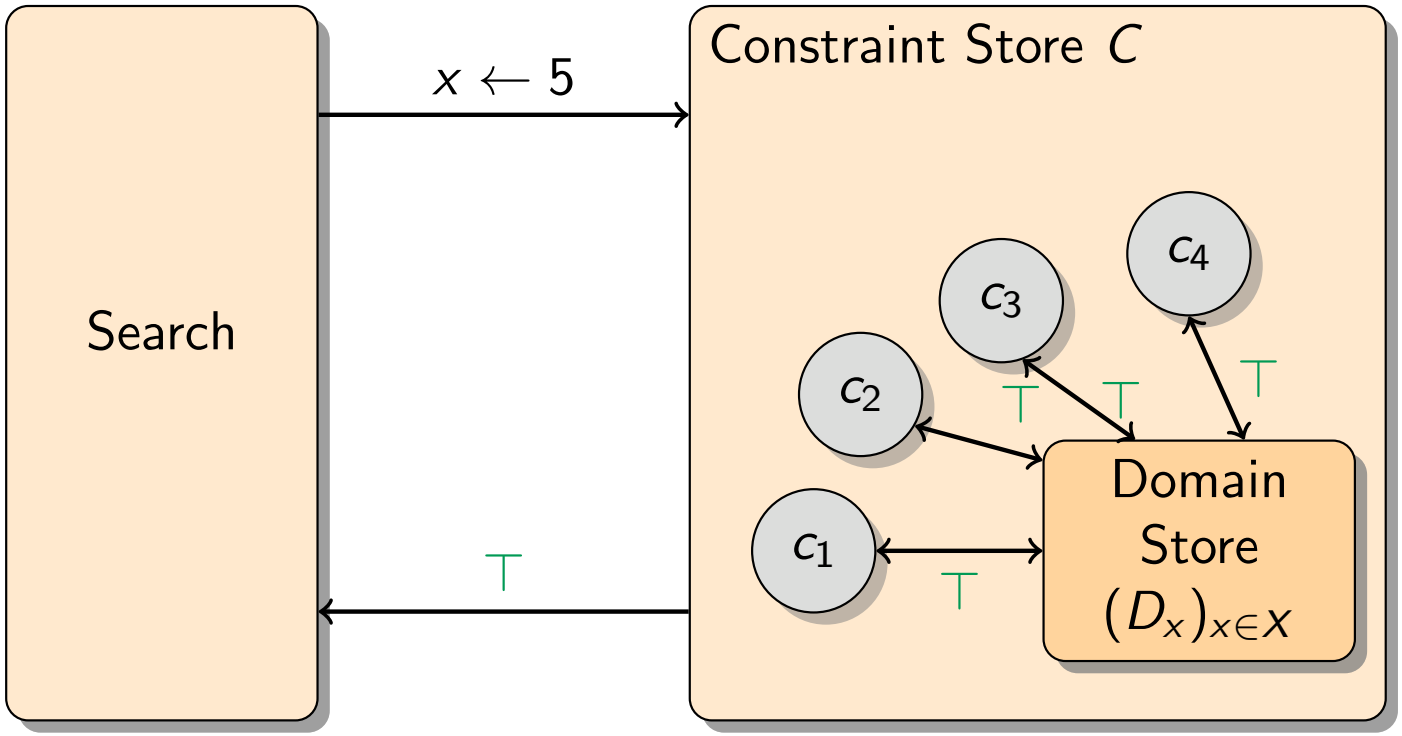
```
var 0..2: x;
var 0..2: y;
var 0..1: z;

% c1
constraint x != y /\ y != z /\ x != z;
% c2
constraint x + 1 = y;

solve satisfy;
```

Welche Zuweisung ist eine Lösung dieses Problems?

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

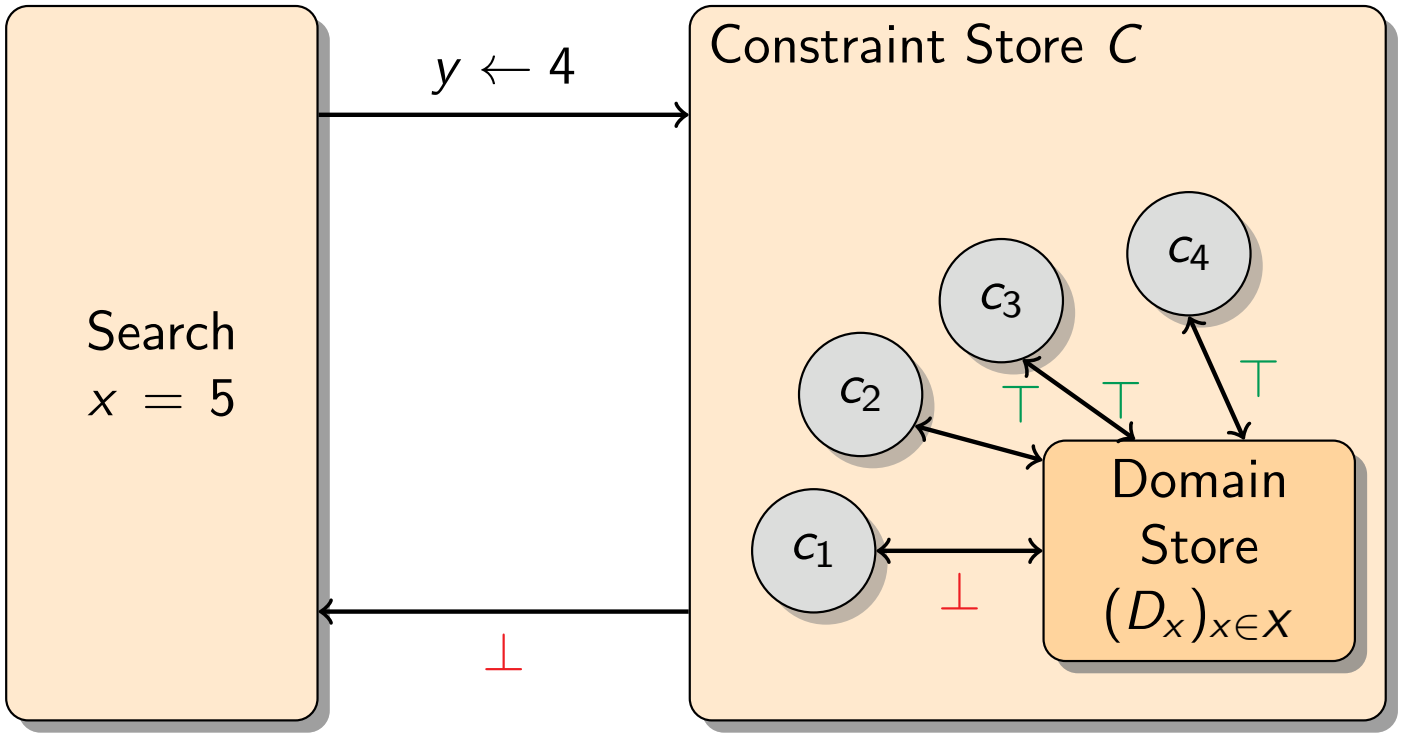
Architektur von Constraint-Lösern

Institute for
Software & Systems
Engineering



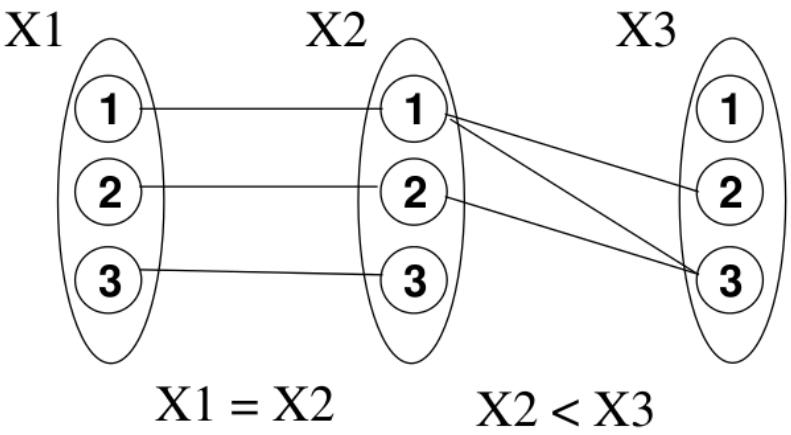
Architektur von Constraint-Lösern

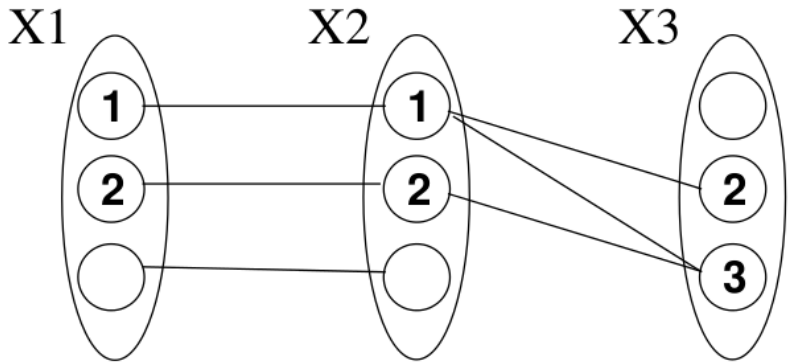
Institute for
Software & Systems
Engineering



Constraint-Propagation: Beispiel

Institute for
Software & Systems
Engineering





Entfernen von Werten, die zu keiner Lösung führen können.

Globale Constraints

Institute for Software & Systems Engineering

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

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

Institute for Software & Systems Engineering

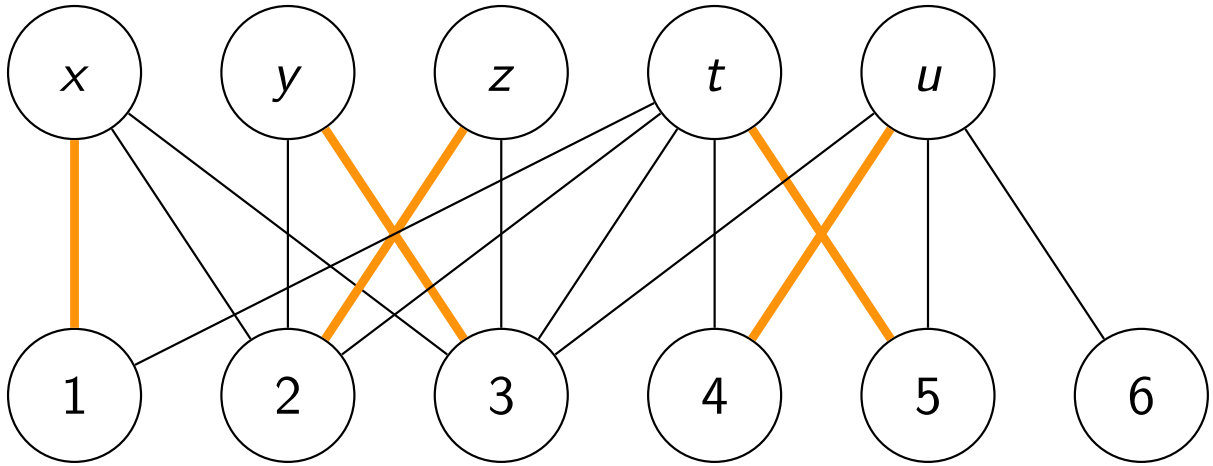
```
% problem data
int: n; set of int: ROBOTS = 1..n;
int: m; set of int: TASKS = 1..m;
array[ROBOTS,TASKS] of int: profit;

% 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

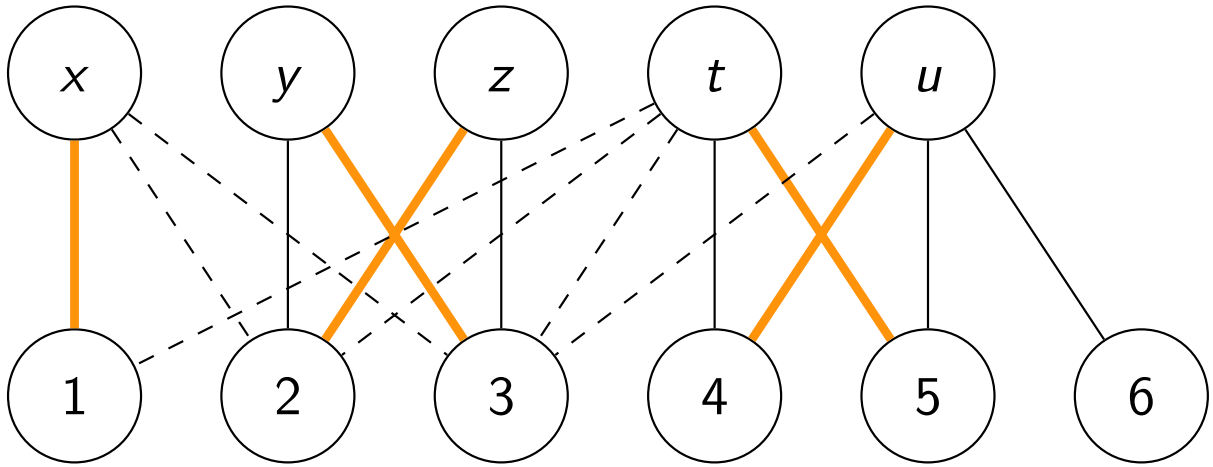
- Erster bekannter Propagator
- Basiert auf Matching in bipartiten Graphen
- Laufzeit ist polynomiell!

```
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]);
```

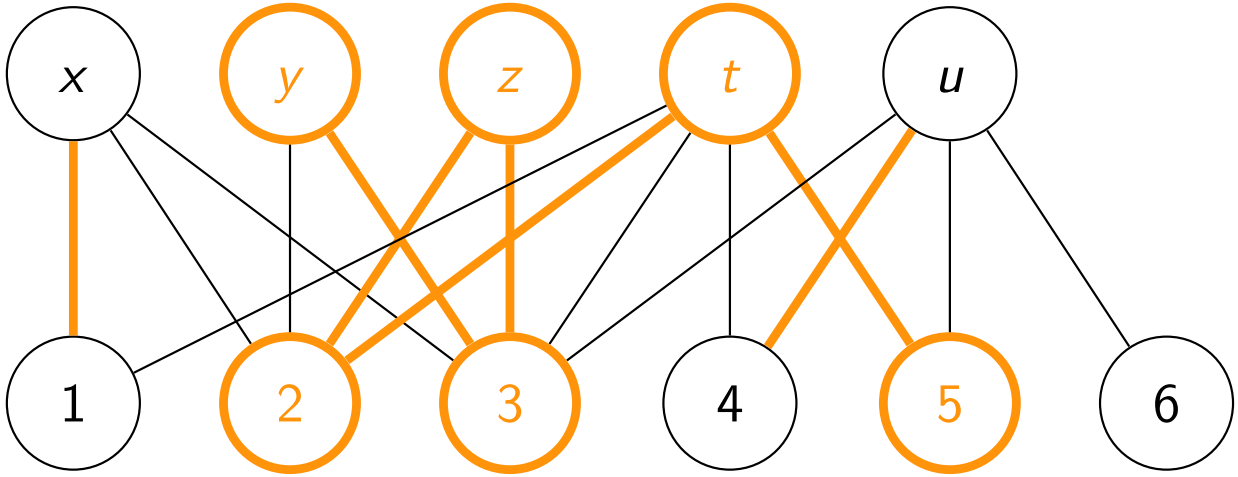



```
var {1}: x;     var {2,3}: y;    var {2,3}: z;
var {4,5}: t;    var {4,5,6}: u;
constraint alldifferent([x,y,z,t,u]);
```

AllDifferent – Algorithmus

Institute for
Software & Systems

```
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:
$y \to 3 \to z \to 2 \to t \to 5$

Let that sink in

Institute for Software & Systems Engineering

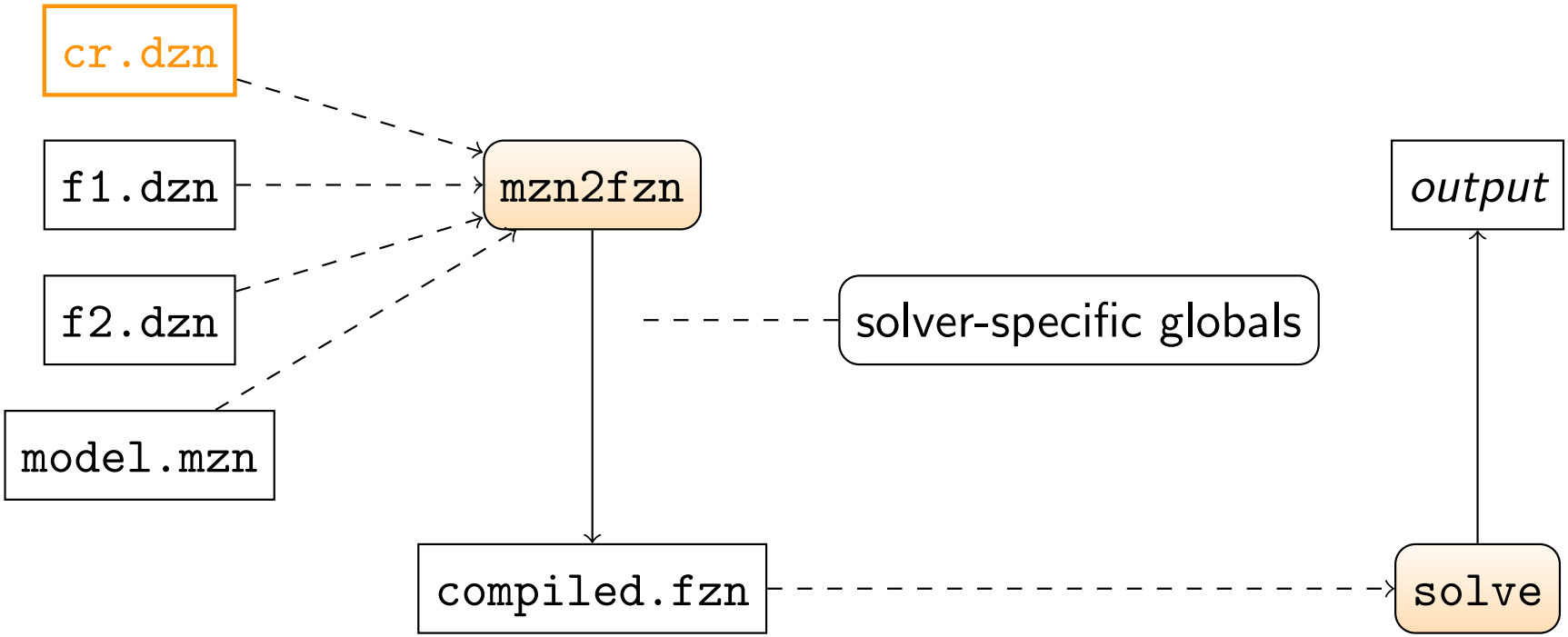
Take away message

Global constraints are the essence of constraint programming!

How can we actually use all that?

Institute for Software & Systems Engineering

Architecture



Why MiniZinc?

Institute for
Software & Systems
Engineering

Rationale

One modeling language – many solvers

Supported Solvers

- Gecode (CP)
- JaCoP (CP)
- Google Optimization Tools (CP)
- Choco (CP)
- G12 (CP/LP/MIP)

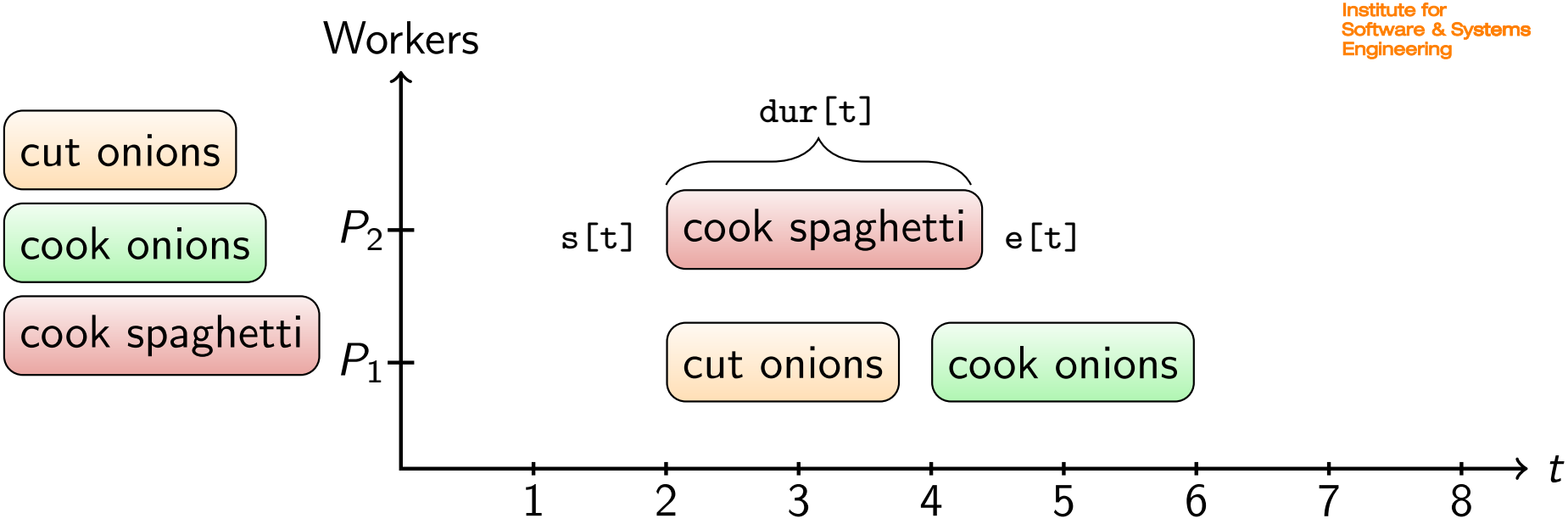
Scheduling & Packing

Institute for
Software & Systems
Engineering

- A very important application for CP
- Strong support (global constraints) in MiniZinc
  - Involved propagators – fortunately encapsulated by solvers!
- Planning and scheduling are coming together and constraint satisfaction may serve as a bridge between them
- Many applications in practice (production style problems)
- Overview papers:
  - Bartak'03 (Barták, 2003)
  - Bartak'10 (Barták et al., 2010)
  - Baptiste'06 (Baptiste et al., 2006)

Intro to Scheduling

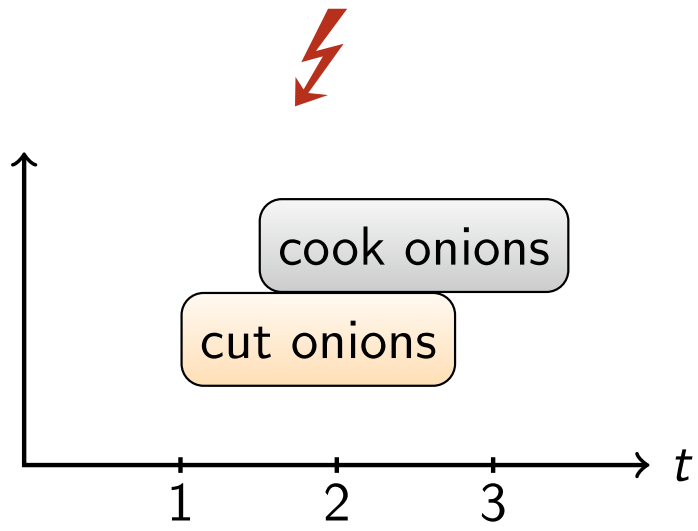
Institute for
Software & Systems
Engineering

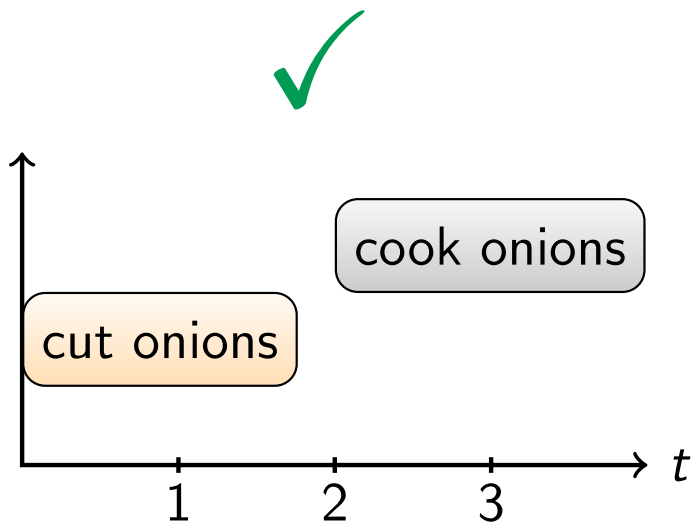



```
int: T; set of int: TASK = 1..T; set of int: HORIZON = 0..10;

array[TASK] of var HORIZON: s; % start times
array[TASK] of var HORIZON: e; % end times
array[TASK] of HORIZON: du; % durations (fixed as constants!)

constraint forall(t in TASK) (s[t] + du[t] = e[t]); % consistency
constraint e[cut_on] <= s[cook_on]; % precedence
```

Non-overlaps (Manually)

Institute for
Software & Systems
Engineering

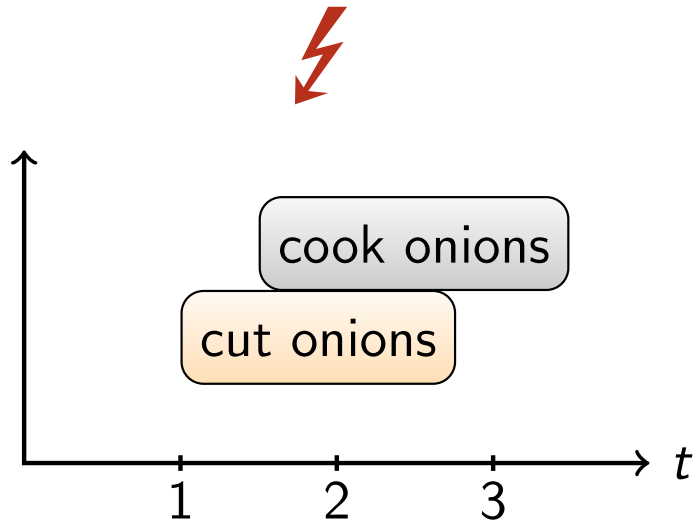


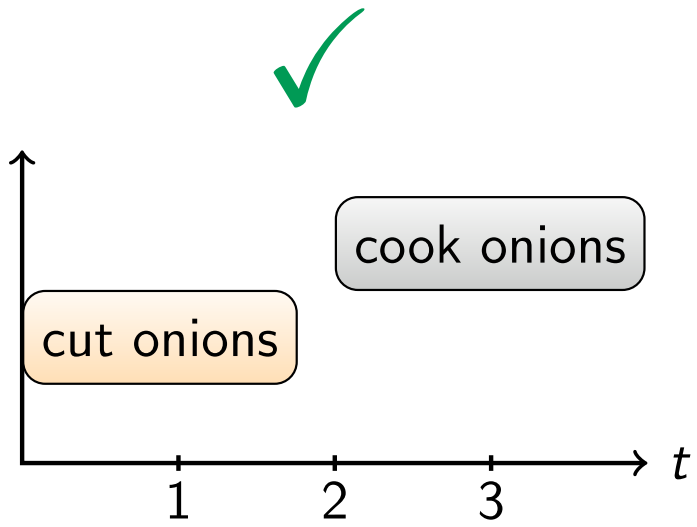



```
predicate nonoverlap(var TASK: task1, var TASK: task2) =
            s[task1] + d[task1] <= s[task2] \/
            s[task2] + d[task2] <= s[task1];


constraint nonoverlap(cut_on, cook_on);
```

Non-overlaps (Global Constraint)

Institute for
Software & Systems
Engineering

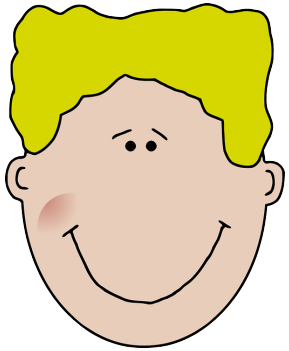


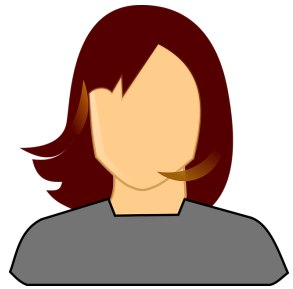



```
include "disjunctive.mzn"; % global definition from standard library
predicate disjunctive(array [int] of var int: s,
                      array [int] of var int: d)

constraint nonoverlap( [ s[cut_on], s[cook_on] ],
                       [ du[cut_on], du[cook_on] ]);
```

A small example

Institute for
Software & Systems
Engineering



- Two brothers, Alfred and Ben; both need to
  - Read about an algorithm before
  - programming it on a computer
- Their mother needs to do work on a computer as well
- Only one computer is available

Model

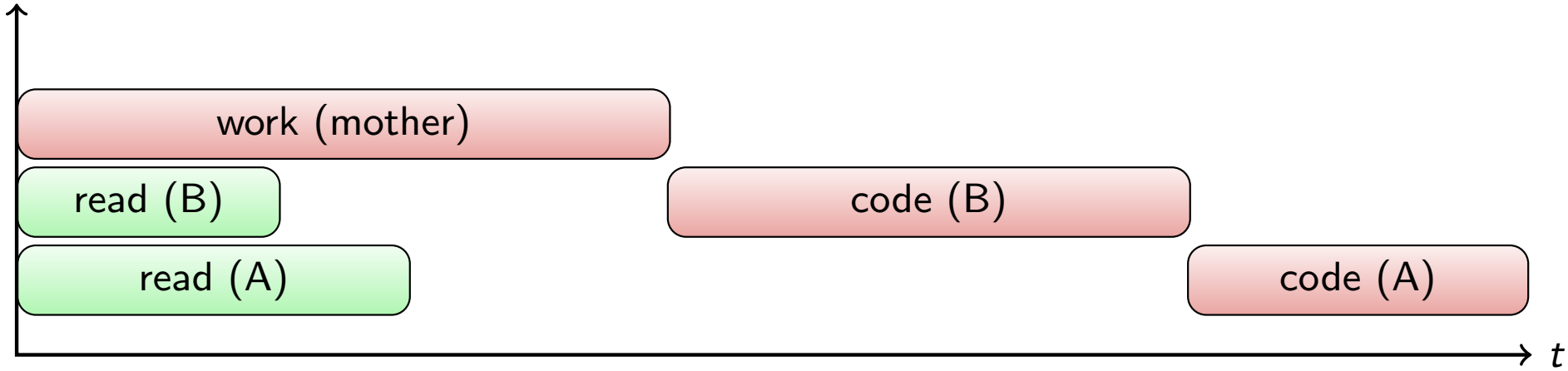
```
int: T = 5; set of int: TASK = 1..T;
set of int: HORIZON = 0..50;
array[TASK] of var HORIZON: s; % start times
array[TASK] of var HORIZON: e; % end times
var HORIZON: makeSpan;
int: readAlfred = 1; int: codeAlfred = 2;
int: readBen = 3; int: codeBen = 4; int: workMother = 5;

% durations
array[TASK] of int: d = [5, 2, 3, 8, 12];

constraint forall(t in TASK) (s[t] + d[t] = e[t]);
constraint forall(t in TASK) (e[t] <= makeSpan);
constraint e[readAlfred] <= s[codeAlfred] /\ e[readBen] <= s[codeBen];

set of TASK: conflicting = {codeAlfred, codeBen, workMother};
constraint disjunctive( [s[t] | t in conflicting],
                        [d[t] | t in conflicting] );
solve minimize makeSpan;
```

Solution

Institute for
Software & Systems
Engineering

```
int: readAlfred = 1; int: codeAlfred = 2;
int: readBen = 3; int: codeBen = 4; int: workMother = 5;
set of TASK: conflicting = {codeAlfred, codeBen, workMother};
array[TASK] of int: d = [5, 2, 3, 8, 12];
```

```
s = array1d(1..5 ,[0, 20, 0, 12, 0]);
e = array1d(1..5 ,[5, 22, 3, 20, 12]);
makeSpan = 22;
```

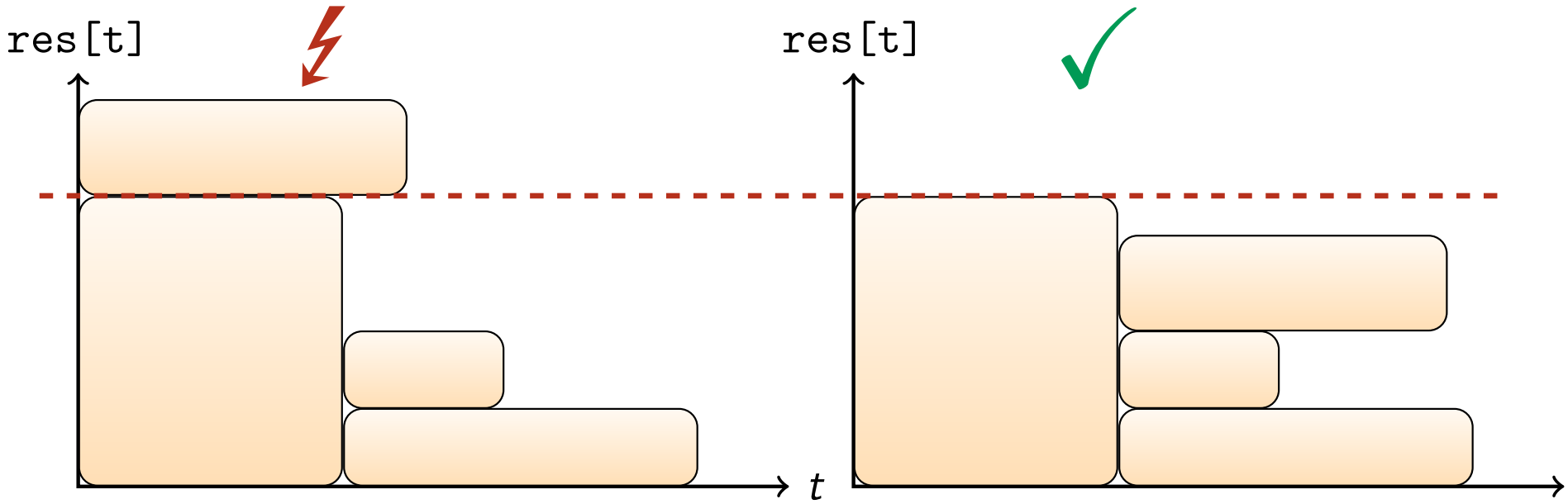


Cumulative Scheduling

Institute for
Software & Systems
Engineering

- `disjunctive` is useful for unary resources
- It is actually a special case of `cumulative`
  - I have m copies of the same resource
  - No more than m can be used at once!
  - Example: m workers, m bulldozers, m tools, ...

- Assume task `t` takes `res[t]` resources



Cumulative Global Constraint

Institute for
Software & Systems
Engineering

```
predicate cumulative(array [int] of var int: s,
                     array [int] of var int: d,
                     array [int] of var int: r,
                     var int: b)
```

MiniZinc Docs

Requires that a set of tasks given by start times `s`, durations `d`, and resource requirements `r`, never require more than a global resource bound `b` at any one time.

Household Revisited

Institute for
Software & Systems
Engineering

How about buying a new computer?

```
predicate cumulative(array [int] of var int: s,
                     array [int] of var int: d,
                     array [int] of var int: r,
                     var int: b)
```

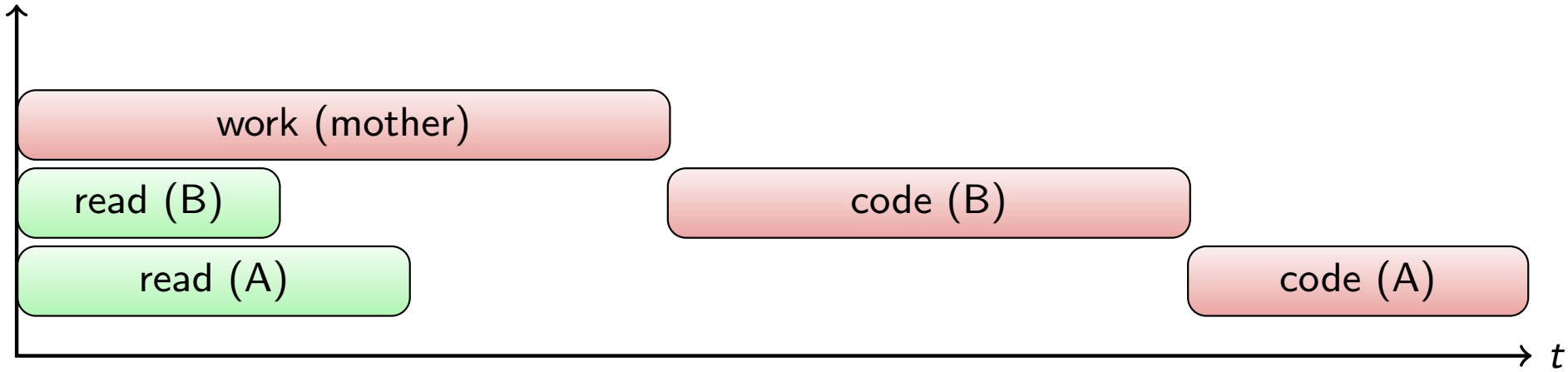
```
set of TASK: conflicting = {codeAlfred, codeBen, workMother};
int: nComputers = 2;

constraint cumulative( [ s[t] | t in conflicting],
                       [ d[t] | t in conflicting],
                       [ 1   | t in conflicting],
                       nComputers );
```

Solution (1 Computer)

Institute for Software & Systems Engineering

```
int: readAlfred = 1; int: codeAlfred = 2;
int: readBen = 3; int: codeBen = 4; int: workMother = 5;
set of TASK: conflicting = {codeAlfred, codeBen, workMother};
array[TASK] of int: d = [5, 2, 3, 8, 12];
```

```
s = array1d(1..5 ,[0, 20, 0, 12, 0]);
e = array1d(1..5 ,[5, 22, 3, 20, 12]);
makeSpan = 22;
```

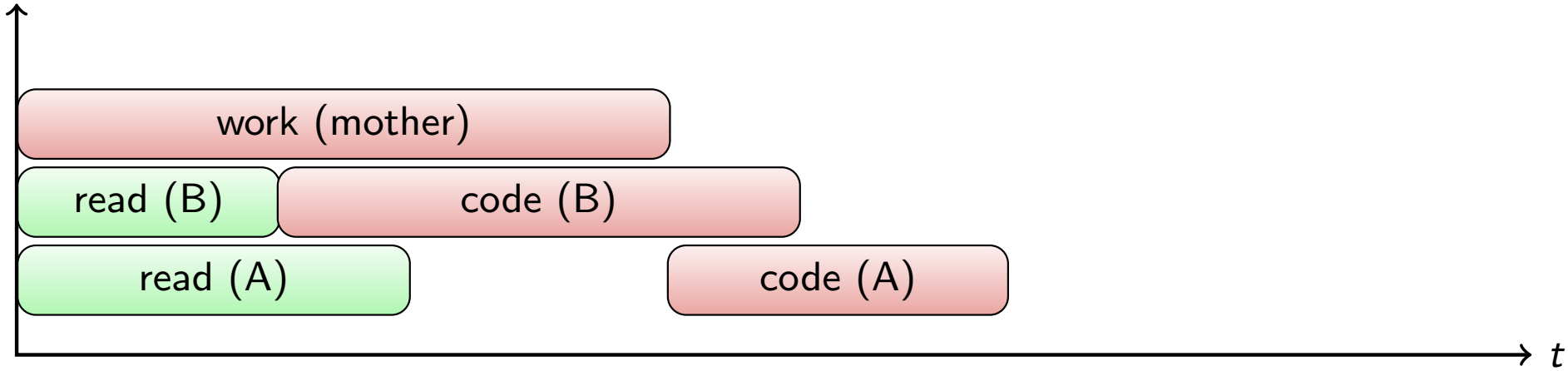


Solution (2 Computers)

Institute for Software & Systems Engineering

```
int: readAlfred = 1; int: codeAlfred = 2;
int: readBen = 3; int: codeBen = 4; int: workMother = 5;
set of TASK: conflicting = {codeAlfred, codeBen, workMother};
array[TASK] of int: d = [5, 2, 3, 8, 12];
```

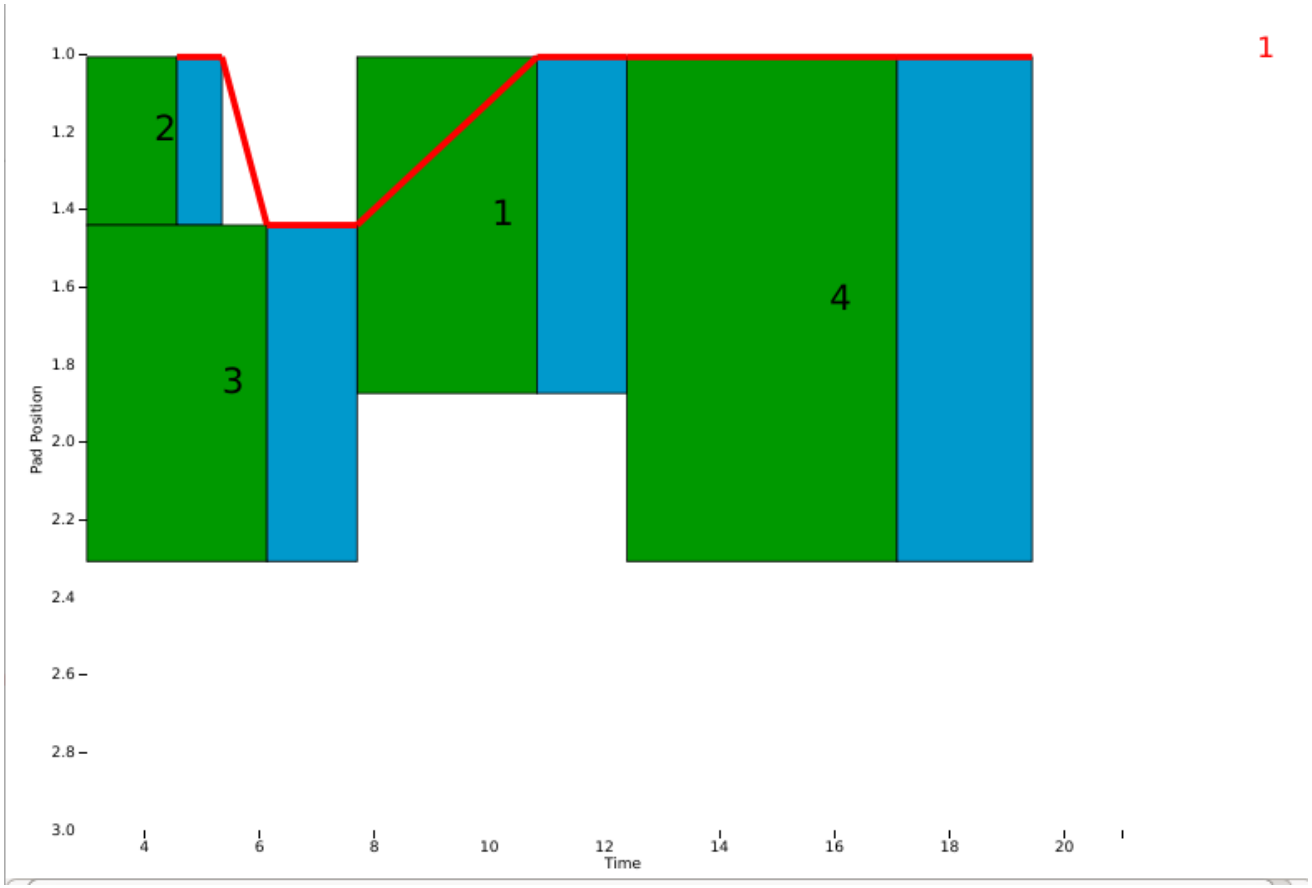
```
s = array1d(1..5 ,[0, 11, 0, 3, 0]);
e = array1d(1..5 ,[5, 13, 3, 11, 12]);
makeSpan = 13;
```



Port Scheduling

Institute for Software & Systems Engineering

Combines many scheduling and packing constraints:



Scheduling: Outlook

Institute for
Software & Systems
Engineering

There's clearly much more to scheduling (that could be useful for us)

- Optional tasks $\rightarrow$ brings us to planning problems
- State-based models: Captures transitions of finite state machines (FSM)
- Sequence-dependent Scheduling
  - Times for cooling down
  - Color changes

Take away

Rather easy to capture since we're dealing only with constraint models!

Soft-Constraints

Institute for
Software & Systems
Engineering

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.

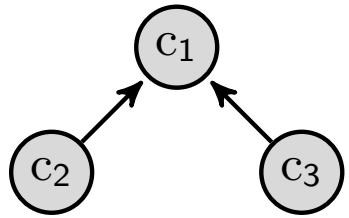
Welche Zuweisungen $v \in [X \to D]$ sollen bevorzugt werden von einem Agenten (oder sogar einer Menge von Agenten)?

Constraint Relationships

Institute for Software & Systems Engineering

Ansatz (Schiendorfer et al., 2013)

- 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



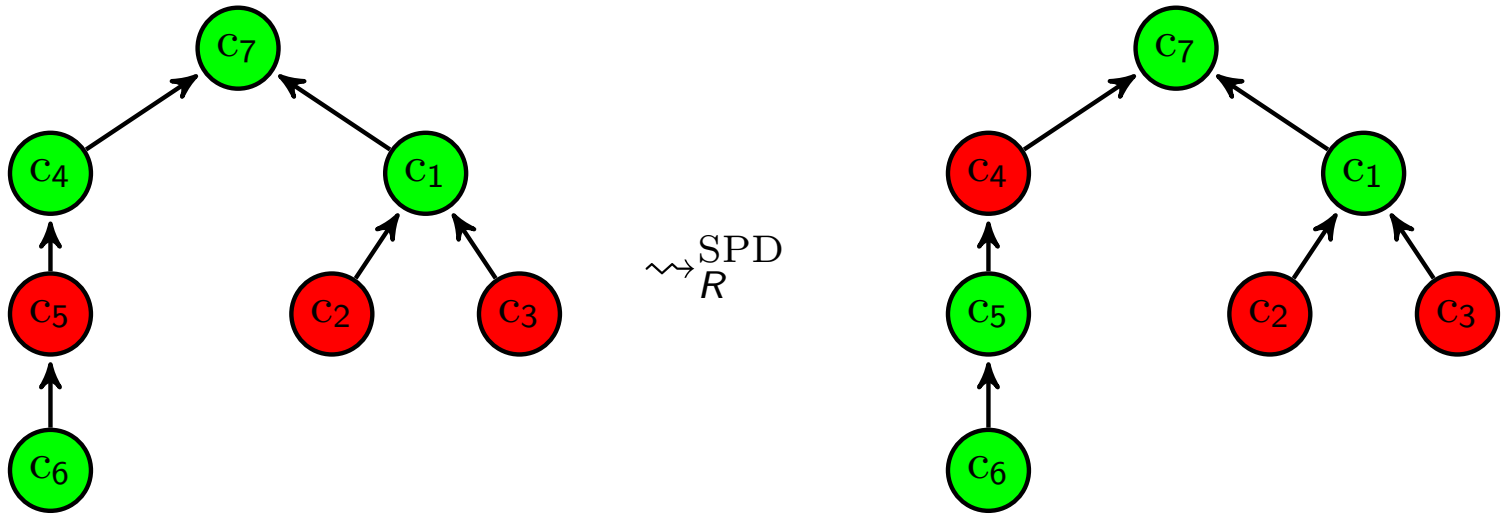
Benefits

- 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

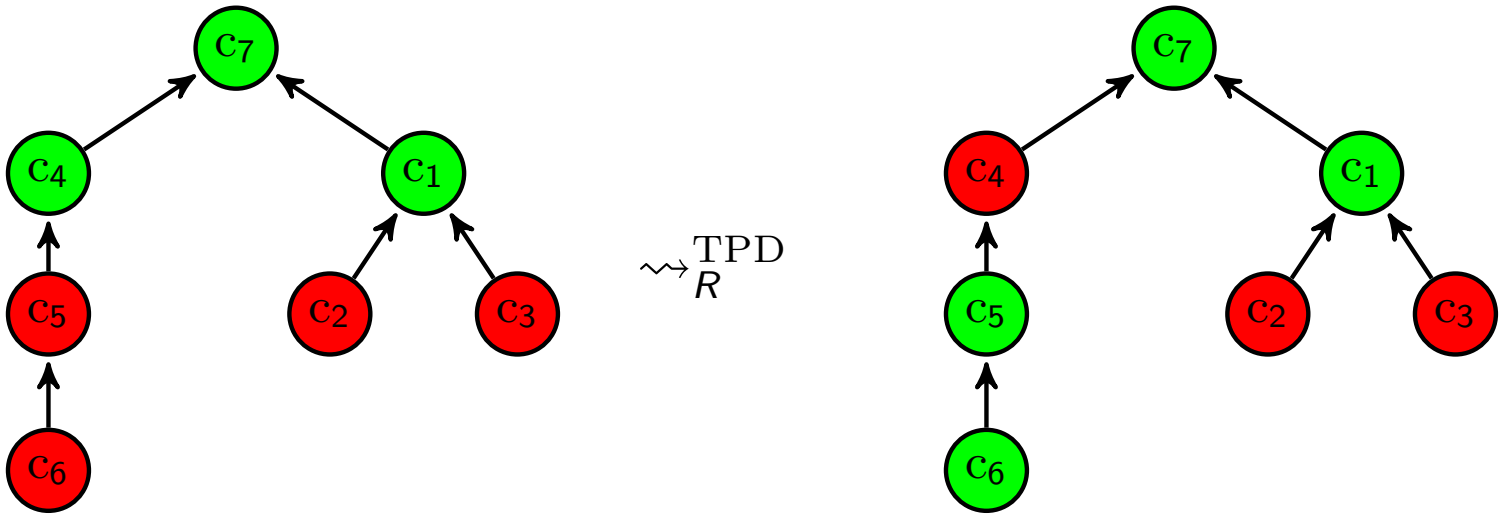
isWorseThan-Relation für Mengen verletzter Constraints



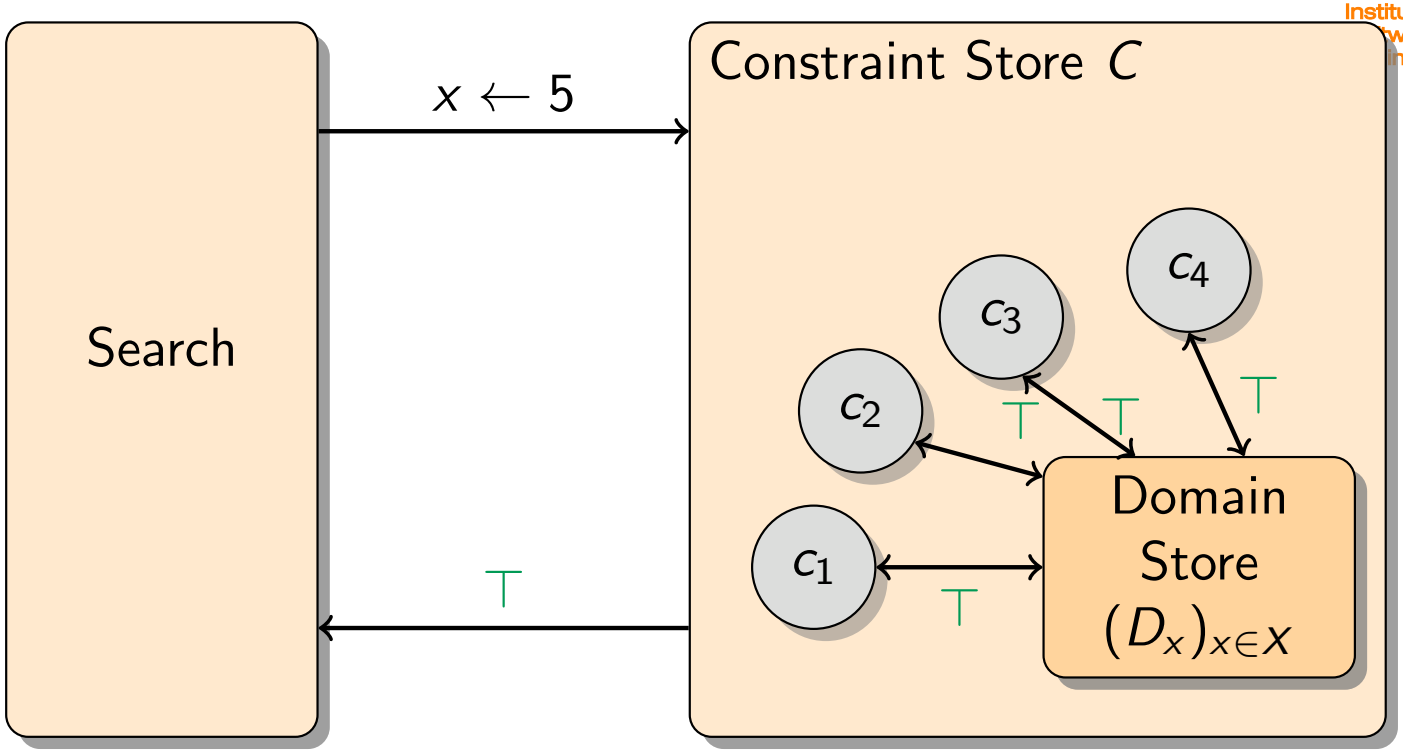
Transitive-Predecessors-Dominance (TPD) Lifting

Institute for Software & Systems Engineering

isWorseThan-Relation für Mengen verletzter Constraints

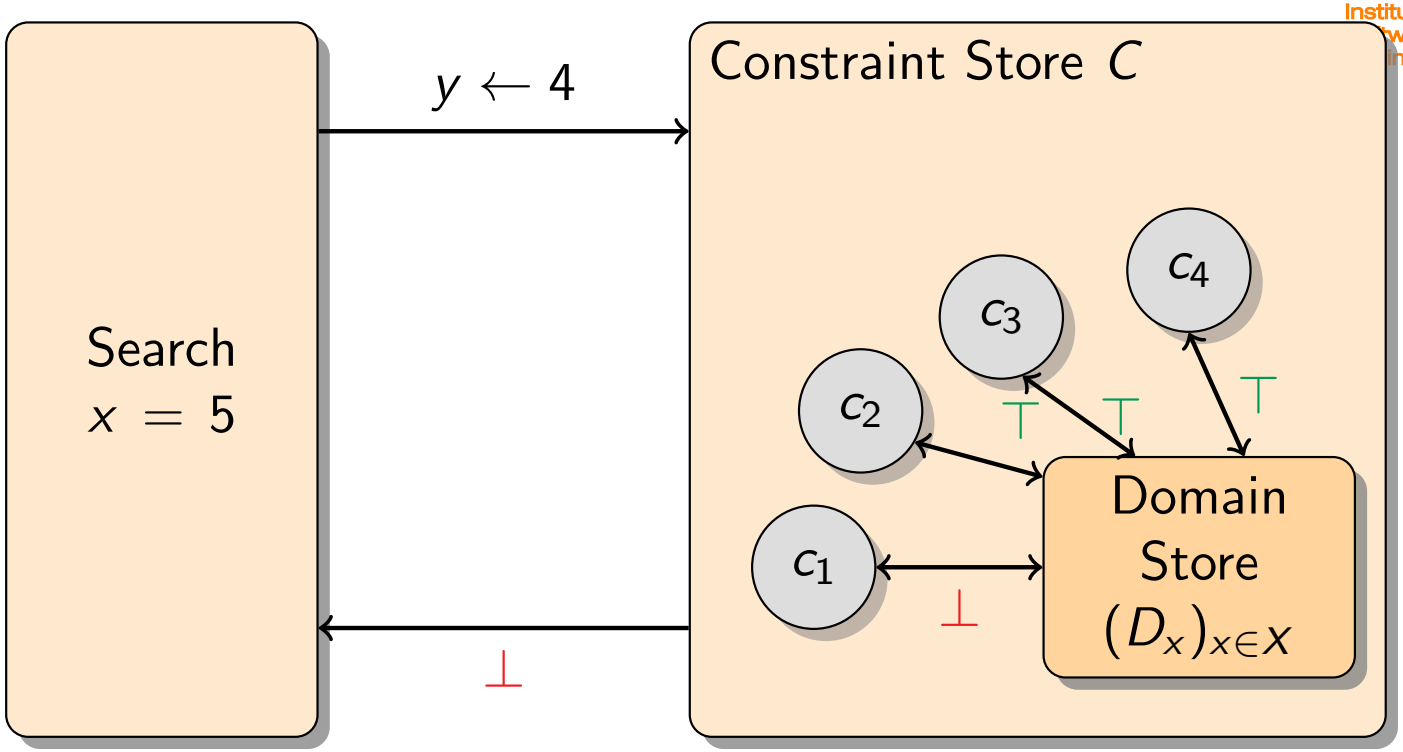


Traditional Constraint Solving



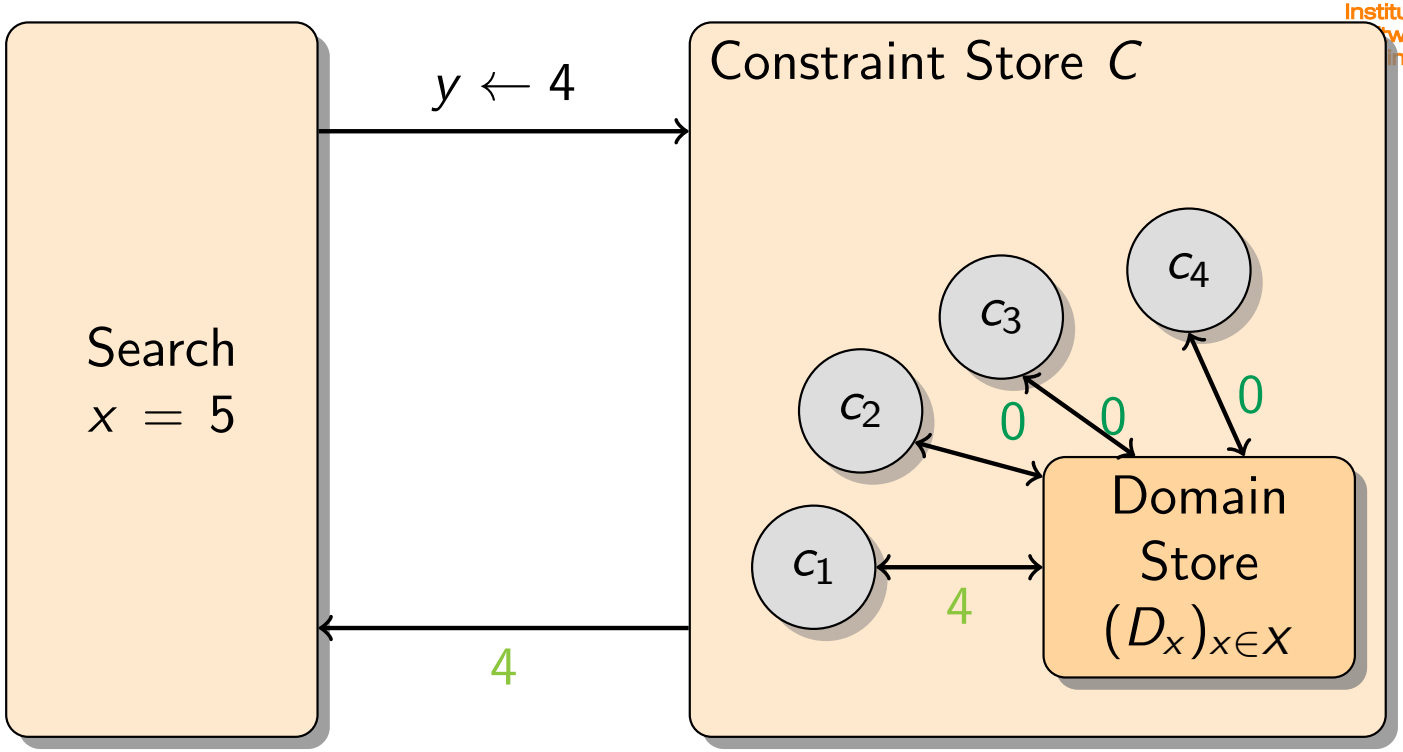
- Eine Menge von Erfüllungsgraden, $\mathbb{B} = \{\bot, \top\}$
- Eine Kombinationsoperation $\wedge$
- Ein neutrales Element $\top$
- Eine partielle Ordnung $(\mathbb{B}, \leq_{\mathbb{B}})$ mit $\top <_{\mathbb{B}} \bot$

Klassisches Constraint-Solving



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

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

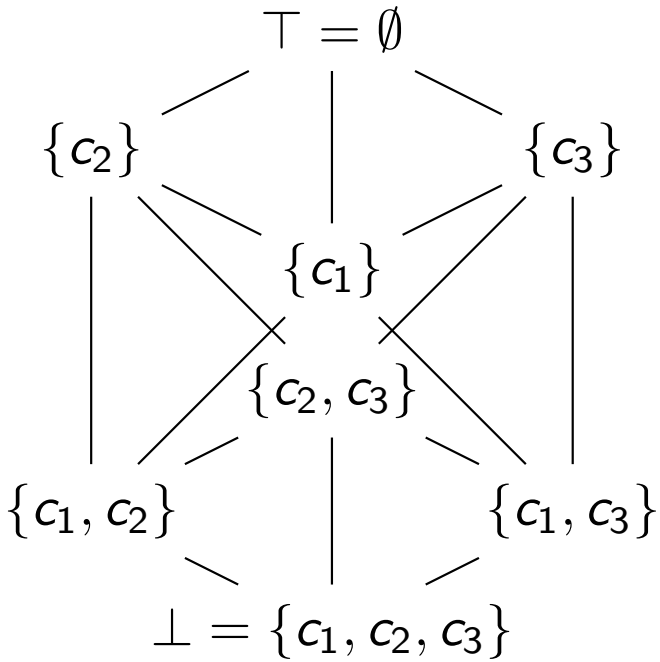
SoftConstraints in MiniZinc

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)

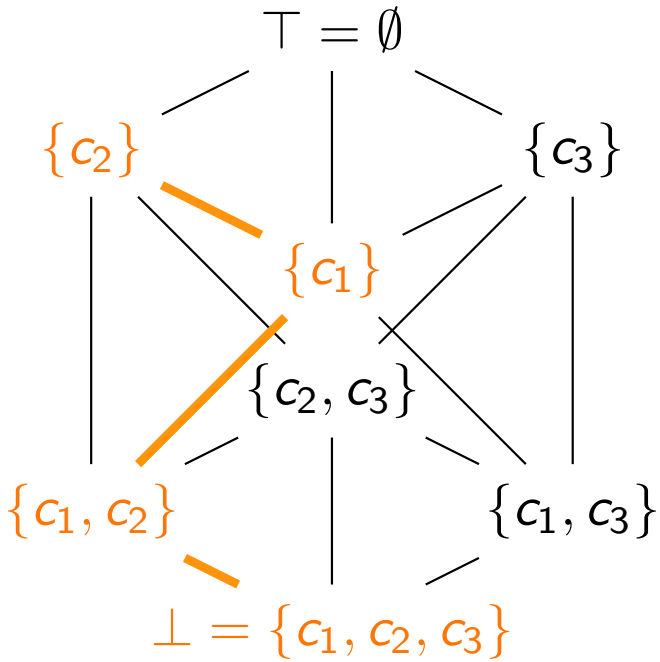



```
%
% Typical Optimization Routine (Branch and Bound):
%
%  1. Look for the first feasible solution
%  2. Impose restrictions on the next feasible solution
%  3. Repeat
```

Search types: Strictly better

Institute for Software & Systems Engineering

The whole valuation space (partially ordered)

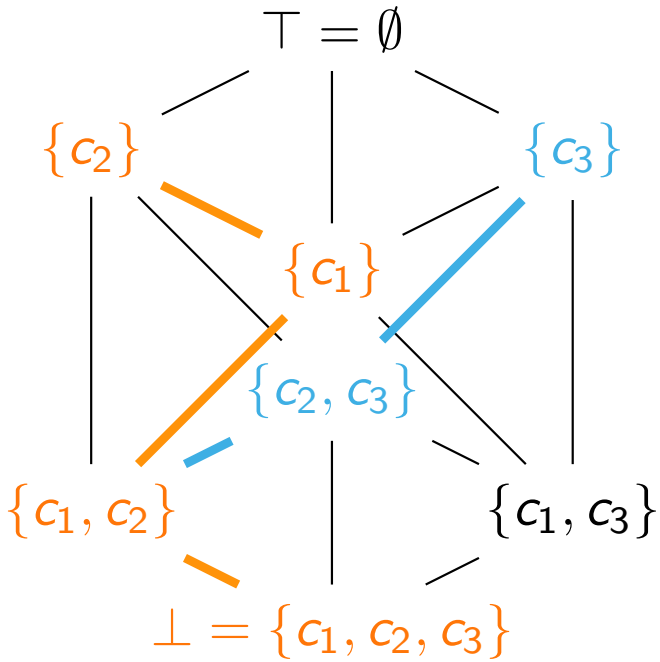



```
function ann: strictlyBetterBAB(var set of SOFTCONSTRAINTS: vScs)
     = repeat(
         if next() then
           commit() /\
           post(isWorse(sol(vScs), vScs))
         else break endif     );
```

Search types: Only not dominated

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(vScs, sol(vScs)) \/ vScs = sol(vScs)))
         else break endif     );
```

Case Studies

Institute for Software & Systems Engineering

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 (found in `example-problems`)

- Mentor Matching
- Exam Scheduling
- Power Plant Scheduling

Mentor Matching

Institute for Software & Systems Engineering

Goal: Assign mentees (e.g. students) to mentors (e.g. companies) such that

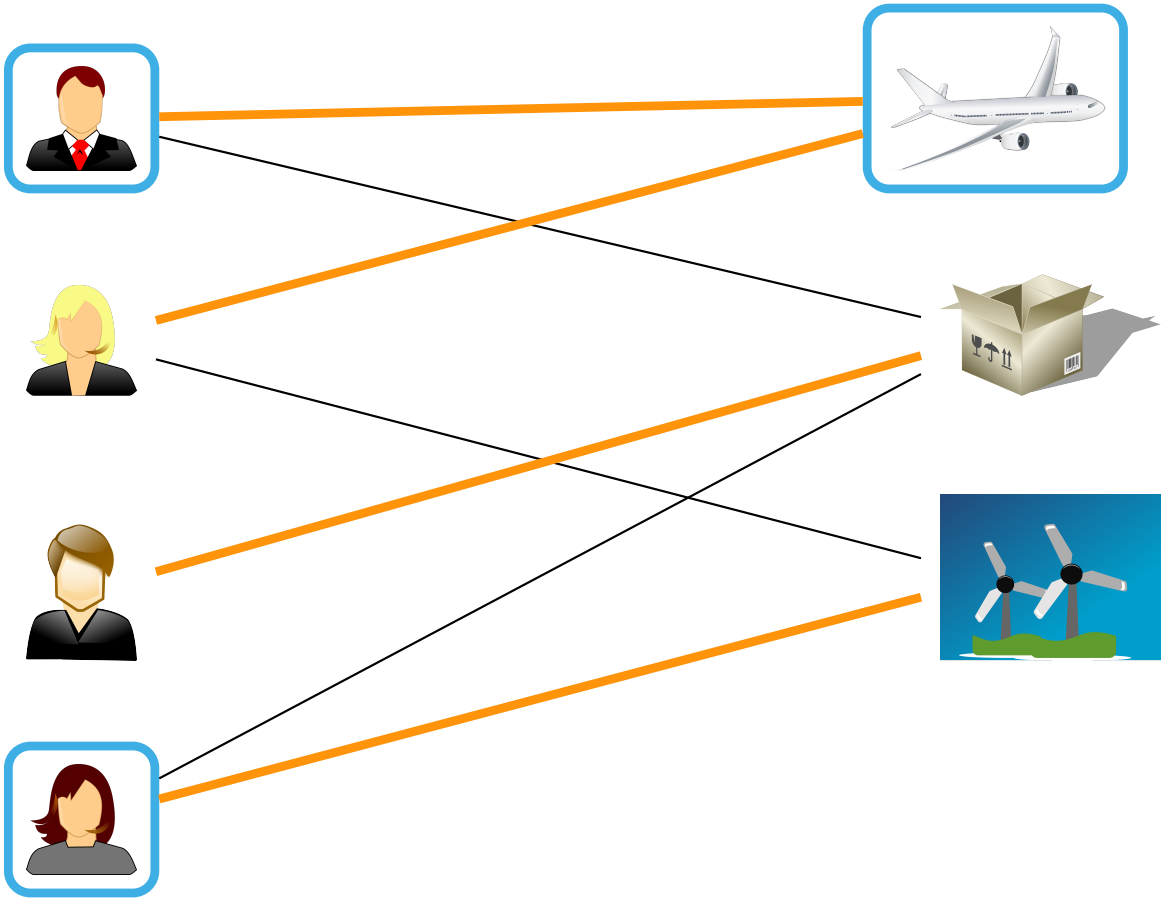
- Students are most satisfied with their mentors
- Companies are satisfied with their mentees
- Two-sided preferences

So far, sounds like a typical *stable matching* problem, but:

- We do not have a 1:1 mapping (companies advise several students)
- Additional constraints are present
  - Each company has to advise at least l, at most u students
  - The number of advised students *should* be roughly equal per company (fairness)
  - Students actually despising a company should not be forced to go there (*hard exclusion* of solutions)

Mentor Matching: Example

Institute for
Software & Systems
Engineering



This assignment respects the students' preferences (edges) but ignores the companies' preferences. ok, it's not really a *matching* since companies supervise more than one student ...

Mentor Matching: Model

Institute for Software & Systems

```
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

Institute for Software & Systems Engineering

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

Institute for Software & Systems Engineering

- 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)

At least 3 options have to be selected

		Approve	Absolutely not
12 February 2016	Morning	○	○
12 February 2016	Afternoon	○	○
18 February 2016	Morning	○	○
18 February 2016	Afternoon	○	○
…	…	○	○

Name	

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

Exam Scheduling: Core Model

Institute for Software & Systems Engineering

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

Institute for Software & Systems Engineering

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

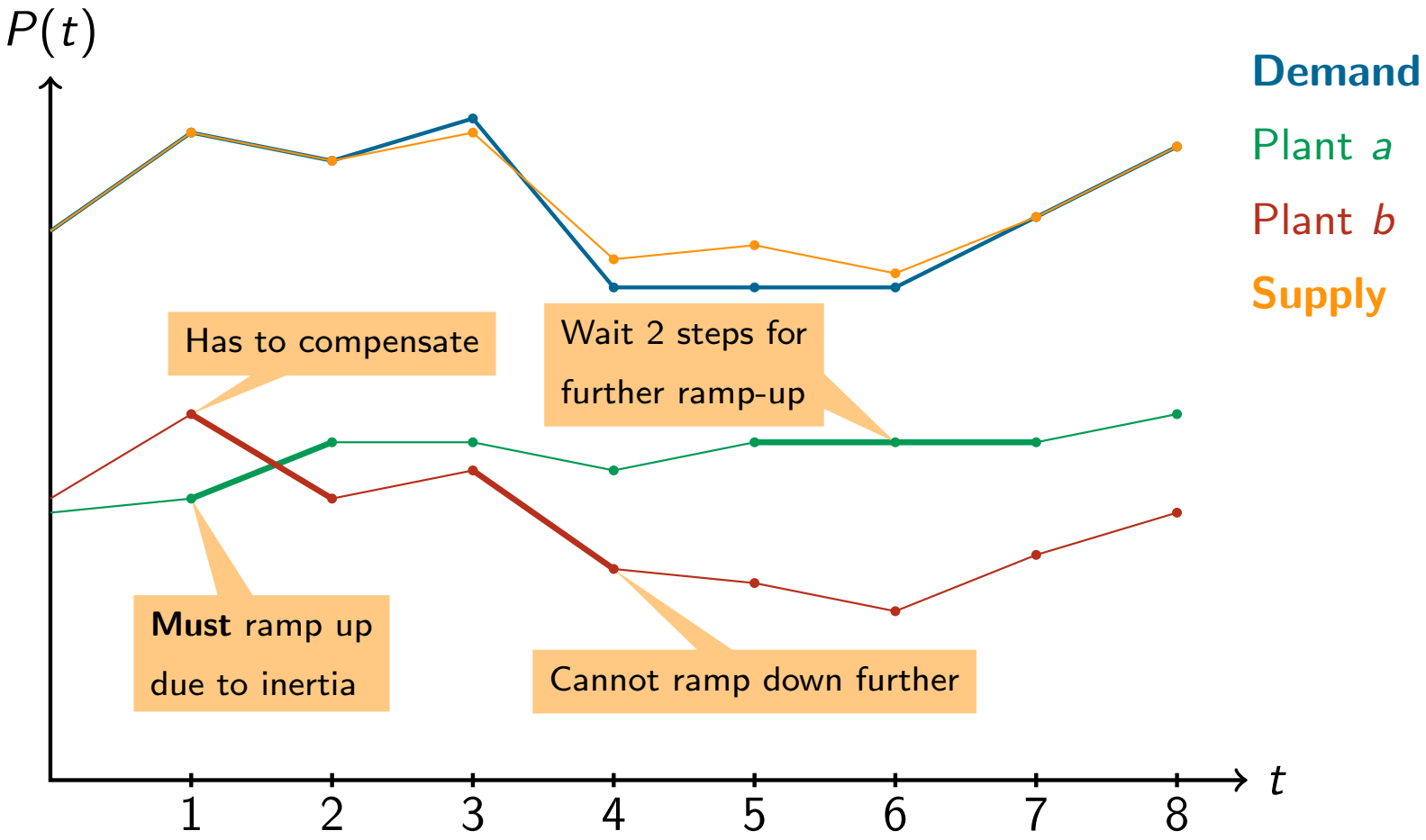
- Aggregated via approval voting (has nice voting-theoretical properties!)
- At most 4 per appointment
- Immediately (61 msec) found an optimal solution that
  - Is approved by *every* student
  - Is achieved with the minimal number of 9 dates
- Used Strategy:

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

Power Plant Scheduling

Institute for Software & Systems Engineering

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

Institute for Software & Systems Engineering

```
% 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

Institute for Software & Systems Engineering

a

Language Features: Suitable Weighting

Institute for Software & Systems Engineering

a

Language Features: Consistency Checks

Institute for Software & Systems Engineering

a

Language Features: Variable Ordering

Institute for
Software & Systems
Engineering

a

Language Features: Redundant Constraints

Institute for Software & Systems Engineering

a

Language Features: Custom Search

Institute for Software & Systems Engineering

a

References I

Baptiste, P., Laborie, P., Le Pape, C., and Nuijten, W. (2006).
Constraint-based scheduling and planning.
Handbook of Constraint Programming, 22:759–798.

Barták, R., Salido, M. A., and Rossi, F. (2010).
Constraint satisfaction techniques in planning and scheduling.
Journal of Intelligent Manufacturing, 21(1):5–15.

Barták, R. (2003).
Constraint-based scheduling: An introduction for newcomers.
In *In Intelligent Manufacturing Systems 2003*, pages 69–74. IFAC Publications, Elsevier Science.

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.

References II

Rossi, F., Beek, P. v., and Walsh, T. (2006).
Handbook of Constraint Programming (Foundations of Artificial Intelligence).
Elsevier Science Inc., New York, NY, USA.

Russell, S. and Norvig, P. (2010).
Artificial Intelligence: A Modern Approach.
Prentice Hall series in artificial intelligence. Prentice Hall.

Schiendorfer, A., Steghöfer, J.-P., Knapp, A., Nafz, F., and Reif, W. (2013).
Constraint Relationships for Soft Constraints.
In Bramer, M. and Petridis, M., editors, *Proc. 33rd SGAI Int. Conf. Innovative Techniques and Applications of Artificial Intelligence (AI'13)*, pages 241–255. Springer.

Tsang, E. (1993).
Foundations of constraint satisfaction, volume 289.
Academic press London.