

Modeling with Metaconstraints and Semantic Typing

André Ciré

University of Toronto

John Hooker

Carnegie Mellon University

Tallys Yunes

University of Miami

Monash University/NICTA, May 2014

Exploiting Problem Structure

- You can't solve NP-hard problems without exploiting special structure.

Exploiting Problem Structure

- You can't solve NP-hard problems without exploiting special structure.
- For SAT solvers:
 - Careful encoding of problem in SAT form
 - This has become a minor industry

Exploiting Problem Structure

- You can't solve NP-hard problems without exploiting special structure.
- For SAT solvers:
 - Careful encoding of problem in SAT form
- For MIP solvers:
 - Careful choice of variables for tight formulation
 - Addition of valid inequalities
 - SOS1, SOS2, symmetry-breaking constraints, etc.
 - Solver parameters (e.g., which cuts?)

Conveying structure to the solver(s)

- Formulate problem with global constraints or **metaconstraints** to reveal structure
- Automatically convert these to **optimal formulation** for the solvers(s)
 - Best choice of **variables**.
 - Reformulation of **constraints**.
 - For **effective propagation** or **tight relaxation**
 - Best choice of **domain filters**.
 - Generation of **valid inequalities**

Conveying structure to the solver(s)

- Formulate problem with global constraints or **metaconstraints** to reveal structure
- Automatically convert these to **optimal formulation** for the solvers(s)
 - Best choice of **variables**.
 - Reformulation of **constraints**.
 - For **effective propagation** or **tight relaxation**
 - Best choice of **domain filters**.
 - Generation of **valid inequalities**
- However, metaconstraints pose a fundamental problem of **variable management...**

Variable management problem

- Reformulation typically introduces **new variables**
 - Different metaconstraints may introduce variables that are functionally **the same variable**
 - ...or related in some other way.
 - Recognizing these relationships is essential to obtaining a good model (e.g., a tight continuous relaxation)
 - How can the solver “understand” what is going on in the model?

Variable management problem

- Reformulation typically introduces **new variables**
 - Different metaconstraints may introduce variables that are functionally **the same variable**
 - ...or related in some other way.
 - Recognizing these relationships is essential to obtaining a good model (e.g., a tight continuous relaxation)
 - How can the solver “understand” what is going on in the model?
- Proposal: Model with **semantic typing of variables**.

Semantic typing

- **Semantic typing** assigns a different meaning to each variable...
 - By associating the variable with a multi-place **predicate** and **keyword**.
 - The keyword “**queries**” the relation denoted by the predicate.
- **Advantage:**
 - This allows the solver to **deduce relationships** between variables, both original or introduced.
 - It is also good modeling practice.

How variables are introduced

- The solver may reformulate a constraint containing **general integer variable** x_i in terms of 0-1 variables y_{ij} , where

$$x_i = \sum_j j y_{ij}$$

- y_{ij} s may be **equivalent to other variables that appear** in the model or reformulations of other constraints.

How variables are introduced

- A model may include **two formulations** of the problem that use related variables.
 - Common in CP, because it strengthens **propagation**.

How variables are introduced

- A model may include **two formulations** of the problem that use related variables.

- Common in CP, because it strengthens **propagation**.

- For example,

x_i = job assigned to worker i

y_j = worker assigned to job j

- Solver should generate **channeling constraints** to relate the variables to each other:

$$j = x_{y_j}, \quad i = y_{x_i}$$

How variables are introduced

- The solver may reformulate a **disjunction of linear systems**

$$\bigcup_k A_k x \geq b^k$$

using a convex hull (or big- M) formulation:

$$A_k x^k \geq b^k y_k, \quad \text{all } k$$

$$x = \sum_k x^k, \quad \sum_k y_k = 1$$

$$y_k \in \{0,1\}, \quad \text{all } k$$

- Other constraints may be based on **same set of alternatives**, and corresponding auxiliary variables (y_k etc.) should be equated.

How variables are introduced

- A nonlinear or global solver may use **McCormick factorization** to replace nonlinear subexpressions with auxiliary variables
 - ... to obtain a linear relaxation.

How variables are introduced

- A nonlinear or global solver may use **McCormick factorization** to replace nonlinear subexpressions with auxiliary variables
 - ... to obtain a linear relaxation.
 - For example, bilinear term xy can be linearized by replacing it with new variable z and constraints

$$\begin{aligned} L_y x + L_x y - L_x L_y &\leq z \leq L_y x + U_x y - L_x U_y \\ U_y x + U_x y - U_x U_y &\leq z \leq U_y x + L_x y - U_x L_y \end{aligned}$$

$$\text{where } x \in [L_x, U_x], \quad y \in [L_y, U_y]$$

- Factorization of different constraints may create variables for identical subexpressions.
- They should be identified to get a tight relaxation.

How variables are introduced

- The solver may reformulate different **global constraints** from CP by introducing variables that have the same meaning.

How variables are introduced

- The solver may reformulate different **global constraints** from CP by introducing variables that have the same meaning.
 - For example, **sequence** constraint limits how many jobs of a given type can occur in given time interval:

$\text{sequence}(x), \quad x_i = \text{job in position } i$

and **cardinality** constraint limits how many times a given job appears

$\text{cardinality}(x), \quad x_j = \text{job in position } j$

Both may introduce variables

$y_{ij} = 1 \text{ when job } j \text{ occurs in position } i$

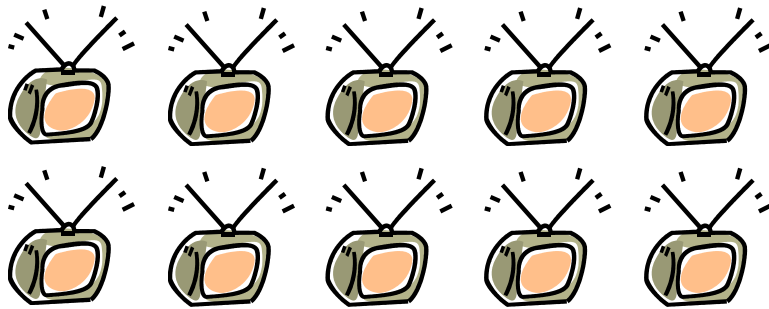
that should be identified.

How variables are introduced

- The solver may introduce equivalent variables while interpreting metaconstraints designed for **classical MIP modeling situations**:
 - Fixed-charge network flow
 - Facility location
 - Lot sizing
 - Job shop scheduling
 - Assignment (3-dim, quadratic, etc.)
 - Piecewise linear

Motivating example

- Allocate 10 advertising spots to 5 products



x_i = how many spots
allocated to product i

$y_{ij} = 1$ if j spots
allocated to product i



A



B



C



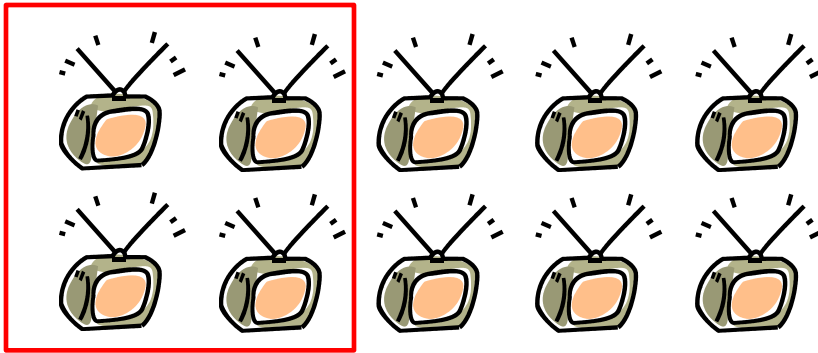
D



E

Motivating example

- Allocate 10 advertising spots to 5 products



≤ 4 spots per product

x_i = how many spots
allocated to product i

$y_{ij} = 1$ if j spots
allocated to product i



A



B



C



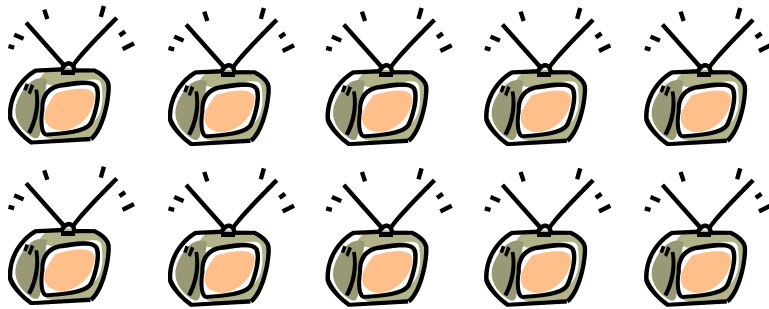
D



E

Motivating example

- Allocate 10 advertising spots to 5 products



≤ 4 spots per product

Advertise ≤ 3 products

x_i = how many spots
allocated to product i

$y_{ij} = 1$ if j spots
allocated to product i



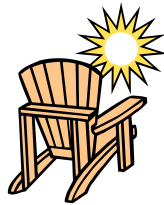
A



B



C



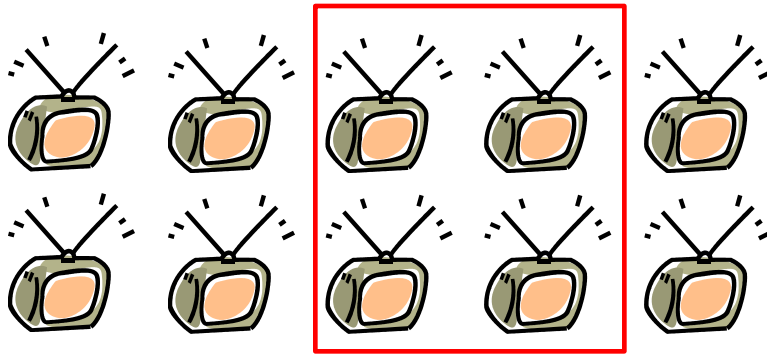
D



E

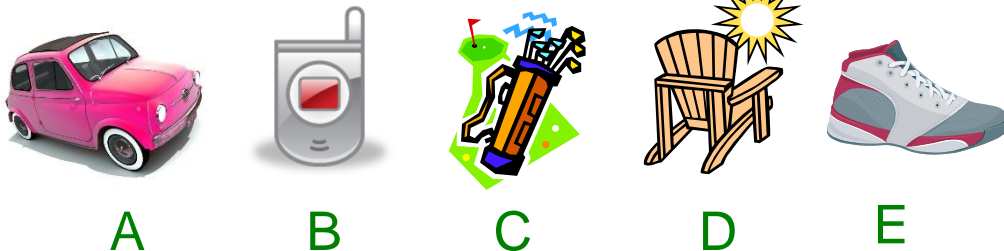
Motivating example

- Allocate 10 advertising spots to 5 products



x_i = how many spots
allocated to product i

$y_{ij} = 1$ if j spots
allocated to product i



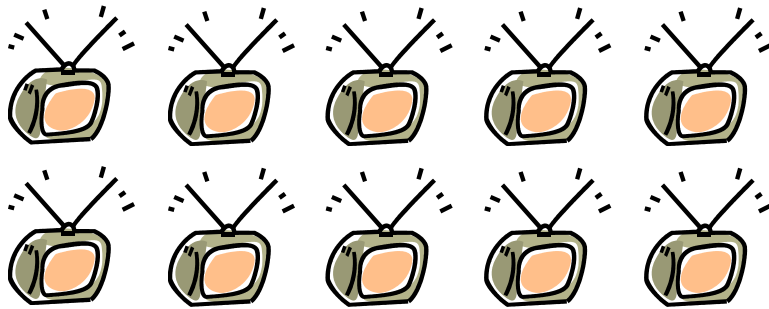
≤ 4 spots per product

Advertise ≤ 3 products

≥ 4 spots for at least
one product

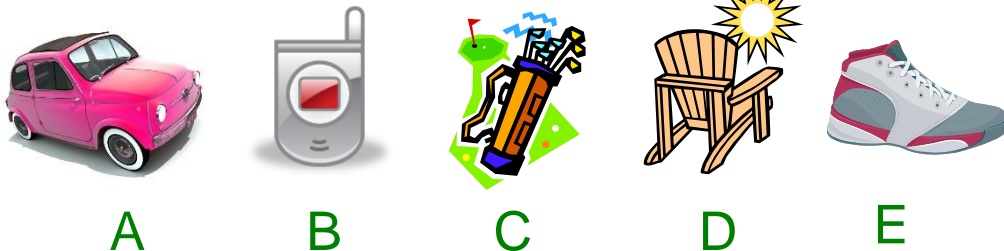
Motivating example

- Allocate 10 advertising spots to 5 products



x_i = how many spots
allocated to product i

$y_{ij} = 1$ if j spots
allocated to product i



≤ 4 spots per product

Advertise ≤ 3 products

≥ 4 spots for at least
one product

P_{ij} = profit from
allocating j spots
to product i

Objective:
maximize profit

Motivating example

```
spots in {0..4}  
product in {A,B,C,D,E})
```

Index sets

Motivating example

```
spots in {0..4}  
product in {A,B,C,D,E}  
data P{product,spots}
```

Data input

Motivating example

`spots in {0..4}`

`product in {A,B,C,D,E}`

`data P{product,spots}`

Declaration of variable x_i

`x[i] is howmany spots allocate(product i)`


Motivating example

```
spots in {0..4}  
product in {A,B,C,D,E}
```

```
data P{product,spots}
```

Declaration of variable x_i

```
x[i] is howmany spots allocate(product i)
```



This makes it
a variable
declaration



Motivating example

```
spots in {0..4}  
product in {A,B,C,D,E}
```

```
data P{product,spots}
```

```
x[i] is howmany spots allocate(product i)
```

Declaration of variable x_i

This is the
semantic type




^

Motivating example

```
spots in {0..4}  
product in {A,B,C,D,E}  
data P{product,spots}
```

Declaration of variable x_i

```
x[i] is howmany spots allocate(product i)
```



Indicates an
integer quantity

Other
keywords:
howmuch
whether

^

Motivating example


```
spots in {0..4}  
product in {A,B,C,D,E}
```

```
data P{product,spots}
```

```
x[i] is howmany spots allocate(product i)
```

Declaration of variable x_i

How many of
what?



^


Motivating example

```
spots in {0..4}  
product in {A,B,C,D,E}
```

```
data P{product,spots}
```

```
x[i] is howmany spots allocate(product i)
```

Declaration of variable x_i



2-place predicate
associated with
variable x

Every variable is
associated with a
predicate that
gives it meaning

Motivating example

```
spots in {0..4}  
product in {A,B,C,D,E}  
data P{product,spots}
```

```
x[i] is howmany spots allocate(product i)
```

Declaration of variable x_i

Other term of the
predicate




Motivating example

```
spots in {0..4}  
product in {A,B,C,D,E}  
data P{product,spots}
```

```
x[i] is howmany spots allocate(product i)
```

Declaration of variable x_i

Associates
index of **x[i]** with
index set **product**



Motivating example

$$\max \sum_i P_{ix_i}$$

spots in {0..4}

product in {A,B,C,D,E}

data P{product,spots}

x[i] is howmany spots allocate(product i)

maximize sum{product i} P[i,x[i]] Objective function

Motivating example

$$\max \sum_i P_{ix_i}$$
$$\sum_i x_i \leq 10$$

```
spots in {0..4}
product in {A,B,C,D,E}
data P{product,spots}
x[i] is howmany spots allocate(product i)
maximize sum{product i} P[i,x[i]]
sum{product i} x[i] <= 10 10 spots available
```

Motivating example

$$\max \sum_i P_{ix_i}$$
$$\sum_i x_i \leq 10$$

```
spots in {0..4}
product in {A,B,C,D,E}
data P{product,spots}
x[i] is howmany spots allocate(product i)
maximize sum{product i} P[I,x[i]]
sum{product i} x[i] <= 10
y[i,j] is whether allocate(product i, spots j)
```

Declare y_{ij}

Indicates 0-1
variable

Motivating example

$$\max \sum_i P_{ix_i}$$
$$\sum_i x_i \leq 10$$

```
spots in {0..4}
product in {A,B,C,D,E}
data P{product,spots}
x[i] is howmany spots allocate(product i)
maximize sum{product i} P[i,x[i]]
sum{product i} x[i] <= 10
y[i,j] is whether allocate(product i, spots j)
```

Declare y_{ij}

Associated with
same predicate
as $x[i]$

Motivating example

$$\max \sum_i P_{ix_i}$$

$$\sum_i x_i \leq 10, \quad \sum_i y_{i0} \geq 2$$

spots in {0..4}

product in {A,B,C,D,E}

data P{product,spots}

x[i] is howmany spots allocate(product i)

maximize sum{product i} P[i,x[i]]

sum{product i} x[i] <= 10

y[i,j] is whether allocate(product i, spots j)

sum{product i} y[i,0] >= 2 At most 3 products advertised

Motivating example

$$\max \sum_i P_{ix_i}$$

$$\sum_i x_i \leq 10, \sum_i y_{i0} \geq 2, \sum_i y_{i4} \geq 1$$

spots in {0..4}

product in {A,B,C,D,E}

data P{product,spots}

x[i] is howmany spots allocate(product i)

maximize sum{product i} P[i,x[i]]

sum{product i} x[i] <= 10

y[i,j] is whether allocate(product i, spots j)

sum{product i} y[i,0] >= 2

sum{product i} y[i,4] >= 1 At least 1 product gets ≥ 4 spots

Motivating example

```
spots in {0..4}
product in {A,B,C,D,E}
data P{product,spots}
```

x[i] is howmany spots **allocate**(product i)

```
maximize sum{product i} P[i,x[i]]
```

```
sum{product i} x[i] <= 10
```

y[i,j] is whether **allocate**(product i, spots j)

```
sum{product i} y[i,0] >= 2
```

```
sum{product i} y[i,4] >= 1
```

```
{product i} sum{spots j} y[i,j] = 1
```

```
{product i} x[i] = sum{spots j} j*y[i,j]
```

$$\max \sum_i P_{ix_i}$$

$$\sum_i x_i \leq 10, \quad \sum_i y_{i0} \geq 2, \quad \sum_i y_{i4} \geq 1$$

$$\sum_j y_{ij} = 1, \quad x_i = \sum_j jy_{ij}, \quad \text{all } i$$

Solver generates linking constraints because

x[i] and **y[i,j]** are associated with the same predicate.

Motivating example

$$\max \sum_i P_{ix_i}$$

$$\sum_i x_i \leq 10, \sum_i y_{i0} \geq 2, \sum_i y_{i4} \geq 1$$

$$\sum_j y_{ij} = 1, x_i = \sum_j jy_{ij}, \text{ all } i$$

spots in $\{0..4\}$

product in $\{A,B,C,D,E\}$

data $P\{\text{product}, \text{spots}\}$

$x[i]$ is howmany spots allocate (product i)

maximize $\text{sum}\{\text{product } i\} P[i, x[i]]$

This constraint must be linearized. Solver generates

$$z_i = \sum_{j=0}^4 P_{ij} y'_{ij}, \sum_{j=0}^4 y'_{ij} = 1, x_i = \sum_{j=0}^4 jy'_{ij}, \text{ all } i$$

$y'[i,j]$ is whether allocate (product i, spots j)

Motivating example

$$\max \sum_i P_{ix_i}$$

$$\sum_i x_i \leq 10, \sum_i y_{i0} \geq 2, \sum_i y_{i4} \geq 1$$

$$\sum_j y_{ij} = 1, x_i = \sum_j jy_{ij}, \text{ all } i$$

spots in {0..4}

product in {A,B,C,D,E}

data P{product,spots}

x[i] is howmany spots allocate (product i)

maximize sum{product i} P[i,x[i]]

This constraint must be linearized. Solver generates

$$z_i = \sum_{j=0}^4 P_{ij} y'_{ij}, \sum_{j=0}^4 y'_{ij} = 1, x_i = \sum_{j=0}^4 jy'_{ij}, \text{ all } i$$

y'[i,j] is whether allocate(product i, spots j)

y and *y'* are identified because they have the same type:

y[i,j] is whether allocate(product i, spots j)

Predicates and relations

Predicate **allocate** denotes 2-place **relation** (set of tuples).
Schematically indicated by:

1	2
product	spots
i	x_i

Predicates and relations

Predicate **allocate** denotes 2-place **relation** (set of tuples).
Schematically indicated by:

1	2
product	spots
i	x_i

Column corresponding to a variable must be a **function** of other columns.

Predicates and relations

Predicate **allocate** denotes 2-place **relation** (set of tuples).
Schematically indicated by:

1	2
product	spots
i	x_i

Declaration of **x[i]** as

howmany spots allocate (product i)

and **y[i,j]** as

whether allocate (product i, spots j)

query the relation for how many and whether.

Predicates and relations

Predicate **allocate** denotes 2-place **relation** (set of tuples).
Schematically indicated by:

1	2
product	spots
i	x_i

Declaration of **x[i]** as

howmany spots allocate (product i)

and **y[i,j]** as

whether allocate (product i, spots j)

query the relation for how many and whether.

In general, **keywords** are **queries** (analogous to relational database)

Predicates and relations

Relation table reveals channeling constraints. For example,

x[i] is which job assign(worker i)

y[j] is which worker assign(job i)

1	2
job	worker
j, x_i	i, y_j

We can read off the channeling constraints

$$j = x_i = x_{y_i}$$

$$i = y_j = y_{x_i}$$

Predicates and relations

If several jobs can be assigned to a worker, we declare

`z[i] is whichset job assign(worker i)`

The channeling constraints are

$$j \in Z_{y_i}$$

Previous work

- **Model management** uses semantic typing to help combine models and use inheritance.
 - Originally inspired by object-oriented programming
Bradley & Clemence (1988)
 - *Quiddity*: a rigorous attempt to analyze conditions for variable identification
Bhargava, Kimbrough & Krishnan (1991)
 - **SML** uses typing in a structured modeling framework
Geoffrion (1992)
 - **Ascend** uses strongly-typed, object-oriented modeling
Bhargava, Krishnan & Piela (1998)

Previous work

- Our semantic typing differs:
 - **Less ambitious** because it doesn't attempt model management.
 - There is only one model.
 - **More ambitious** because we recognize relationships other than equivalence.
 - We manage variables **introduced by solver**.

Previous work

- Modeling systems that convey some structure to solver:
 - All CP modelers (OPL, CHIP, etc.) use **global constraints**.
 - AIMMS uses **typed index sets**.
 - Zinc/MiniZinc (G12 system) reformulates **metaconstraints** for specific solvers.
 - OPL, Xpress-Kalis, Comet, etc., use **interval variables**.
 - SAT solver SymChaff uses high-level **AI planning language PDDL**.
 - Lopes and Fourer (2009) use **UML** (Unified Modeling Language) to model multistage stochastic LPs with recourse.
 - SIMPL has **full metaconstraint capability**.

Previous work

- However, **none of these systems** deals systematically with the variable management problem.
 - We address it with semantic typing of variables.

Assignment problem

```
worker in {1..m}
job in {1..n}
data C{worker,job}
x[i] is which job assign(worker i)
minimize sum{worker i} C[i,x[i]]
alldiff{x[*]}
```

$$\min \sum_i c_{ix_i}$$

$$\text{alldiff}(x_1, \dots, x_n)$$

Assignment problem

```
worker in {1..m}
job in {1..n}
data C{worker,job}
x[i] is which job assign(worker i)
minimize sum{worker i} C[i,x[i]]
alldiff{x[*]}
```

$$\min \sum_i c_{ix_i}$$

$$\text{alldiff}(x_1, \dots, x_n)$$

Objective function
is formulated

$$\max \sum_i c_{ij} y_{ij}, \quad x_i = \sum_j y_{ij}, \quad \text{all } i$$

$y[i,j]$ is whether assign(worker i, job j)

Assignment problem

```
worker in {1..m}
job in {1..n}
data C{worker,job}
x[i] is which job assign(worker i)
minimize sum{worker i} C[i,x[i]]
alldiff{x[*]}
```

$$\min \sum_i c_{ix_i}$$

$$\text{alldiff}(x_1, \dots, x_n)$$

Objective function
is formulated

$$\max \sum_i c_{ij} y_{ij}, \quad x_i = \sum_j y_{ij}, \text{ all } i$$

$y[i,j]$ is whether assign(worker i, job j)

Alldiff
is formulated

$$\sum_j y'_{ij} = 1, \text{ all } i, \quad \sum_i y'_{ij} = 1, \text{ all } j, \quad x_i = \sum_j jy'_{ij}, \text{ all } i$$

$y'[i,j]$ is whether assign(worker i, job j)

Solver identifies y and y' to create classical AP.

Latin squares

j

	2	3	1
i	3	1	2
	1	2	3

Numbers in every row and column are distinct.

We will use **three** formulations to improve propagation.

Latin squares

	<i>j</i>		
	2	3	1
<i>i</i>	3	1	2
	1	2	3

$\text{alldiff}(x_{i1}, \dots, x_{in}), \text{ all } i$

$\text{alldiff}(x_{1j}, \dots, x_{nj}), \text{ all } j$

$\text{alldiff}(y_{i1}, \dots, y_{in}), \text{ all } i$

$\text{alldiff}(y_{1k}, \dots, y_{nk}), \text{ all } k$

$\text{alldiff}(z_{j1}, \dots, z_{jn}), \text{ all } j$

$\text{alldiff}(z_{1k}, \dots, z_{nk}), \text{ all } k$

Numbers in every row and column are distinct.

We will use **three** formulations to improve propagation.

`row, col, num in {1..n}`

`x[i,j] is which num assign(row i, col j)`

`y[i,k] is which col assign(row i, num k)`

`z[j,k] is which row assign(col j, num k)`

Latin squares

	j		
	2	3	1
i	3	1	2
	1	2	3

$\text{alldiff}(x_{i1}, \dots, x_{in}), \text{ all } i$

$\text{alldiff}(x_{1j}, \dots, x_{nj}), \text{ all } j$

$\text{alldiff}(y_{i1}, \dots, y_{in}), \text{ all } i$

$\text{alldiff}(y_{1k}, \dots, y_{nk}), \text{ all } k$

$\text{alldiff}(z_{j1}, \dots, z_{jn}), \text{ all } j$

$\text{alldiff}(z_{1k}, \dots, z_{nk}), \text{ all } k$

Numbers in every row and column are distinct.

We will use **three** formulations to improve propagation.

`row, col, num in {1..n}`

`x[i,j] is which num assign(row i, col j)`

`y[i,k] is which col assign(row i, num k)`

`z[j,k] is which row assign(col j, num k)`

`{row i} alldiff{x[i,*]}; {col j} alldiff{x[*,j]}`

`{row i} alldiff{y[i,*]}; {num k} alldiff{y[*,k]}`

`{col j} alldiff{z[j,*]}; {num k} alldiff{z[*,k]}`

Latin squares

The predicate **assign** denotes the 3-place relation

1	2	3
num	col	row
k, x_{ij}	j, y_{ik}	i, z_{jk}

```
row, col, num in {1..n}
x[i,j] is which num assign(row i, col j)
y[i,k] is which col assign(row i, num k)
z[j,k] is which row assign(col j, num k)
{row i} alldiff{x[i,*]}; {col j} alldiff{x[*,j]}
{row i} alldiff{y[i,*]}; {num k} alldiff{y[*,j]}
{col j} alldiff{z[j,*]}; {num k} alldiff{z[*,k]}
```

Latin squares

The predicate **assign** denotes the 3-place relation

1	2	3
num	col	row
k, x_{ij}	j, y_{ik}	i, z_{jk}

We can read off the channeling constraints:

$$k = x_{z_{jk} y_{ik}}, \quad j = y_{z_{jk} x_{ij}}, \quad i = z_{y_{ik} x_{ikj}}, \quad \text{all } i, j, k$$

which can be propagated.

Latin squares

```
{row i} alldiff{x[i,*]}; {col j} alldiff{x[* ,j]}  
{row i} alldiff{y[i,*]}; {num k} alldiff{y[* ,j]}  
{col j} alldiff{z[j,*]}; {num k} alldiff{z[* ,k]}
```

The 3 formulations generate 3 identical MIP models:

$$x_{ij} = \sum_k k \delta_{ijk}^x; \quad \sum_k \delta_{ijk}^x = 1, \text{ all } i, j; \quad \sum_j \delta_{ijk}^x = 1, \text{ all } i, k; \quad \sum_i \delta_{ijk}^x = 1, \text{ all } j, k$$

$$y_{ik} = \sum_j j \delta_{ijk}^y, \quad \sum_j \delta_{ijk}^y = 1, \text{ all } i, k; \quad \sum_k \delta_{ijk}^y = 1, \text{ all } i, j; \quad \sum_i \delta_{ijk}^y = 1, \text{ all } j, k$$

$$z_{jk} = \sum_i i \delta_{ijk}^z, \quad \sum_i \delta_{ijk}^z = 1, \text{ all } j, k; \quad \sum_k \delta_{ijk}^z = 1, \text{ all } i, j; \quad \sum_j \delta_{ijk}^z = 1, \text{ all } i, k$$

Latin squares

```
{row i} alldiff{x[i,*]}; {col j} alldiff{x[* ,j]}  
{row i} alldiff{y[i,*]}; {num k} alldiff{y[* ,j]}  
{col j} alldiff{z[j,*]}; {num k} alldiff{z[* ,k]}
```

The 3 formulations generate 3 identical MIP models:

$$\begin{aligned}x_{ij} &= \sum_k k \delta_{ijk}^x; \quad \sum_k \delta_{ijk}^x = 1, \text{ all } i, j; \quad \sum_j \delta_{ijk}^x = 1, \text{ all } i, k; \quad \sum_i \delta_{ijk}^x = 1, \text{ all } j, k \\y_{ik} &= \sum_j j \delta_{ijk}^y, \quad \sum_j \delta_{ijk}^y = 1, \text{ all } i, k; \quad \sum_k \delta_{ijk}^y = 1, \text{ all } i, j; \quad \sum_i \delta_{ijk}^y = 1, \text{ all } j, k \\z_{jk} &= \sum_i i \delta_{ijk}^z, \quad \sum_i \delta_{ijk}^z = 1, \text{ all } j, k; \quad \sum_k \delta_{ijk}^z = 1, \text{ all } i, j; \quad \sum_j \delta_{ijk}^z = 1, \text{ all } i, k\end{aligned}$$

The solver declares $\delta_{ijk}^x, \delta_{ijk}^y, \delta_{ijk}^z$
whether assign(row i, col j, num k)

So it treats them as the same variable and generates only 1 MIP model.

Multiple **which** variables

In general, an n -place predicate that denotes the relation

1	...	k	$k + 1$...	n
term ₁	...	term _{k}	term _{$k+1$}	...	term _{n}
$i_1, x_{i(1)}^1$...	$i_k, x_{i(k)}^k$	i_{k+1}	...	i_n

for **which** variables, where $i(j) = i_1 \cdots i_{j-1} i_{j+1} \cdots i_n$

generates the channeling constraints

$$i_j = x_{x_{i(1)}^1 \cdots x_{i(j-1)}^{j-1} x_{i(j+1)}^{j+1} \cdots x_{i(k)}^k i_{k+1} \cdots i_n}^j, \text{ all } i_1, \dots, i_n, j = 1, \dots, k$$

Multiple *whether* variables

whether keywords serve as projection operators on the relation.

$y[i,j,d]$ is *whether* `assign(worker i, job j, day d)`

Project out d :

$y1[i,j]$ is *whether* `assign(worker i, job j)`

Project out j and d :

$y2[i]$ is *whether* `assign(worker i)`

Short forms

Declare x_i to be cost of activity i :

$x[i]$ is *howmuch* cost(activity i)

which is short for the formal declaration

$x[i]$ is *howmuch* cost cost(activity i)

in which a new term **cost** is generated

Declare x to be cost:

x is *howmuch* cost

which is short for

x is *howmuch* cost cost()

Piecewise linear

Piecewise linear function $z = f(x)$
Breakpoints in A , ordinates in C

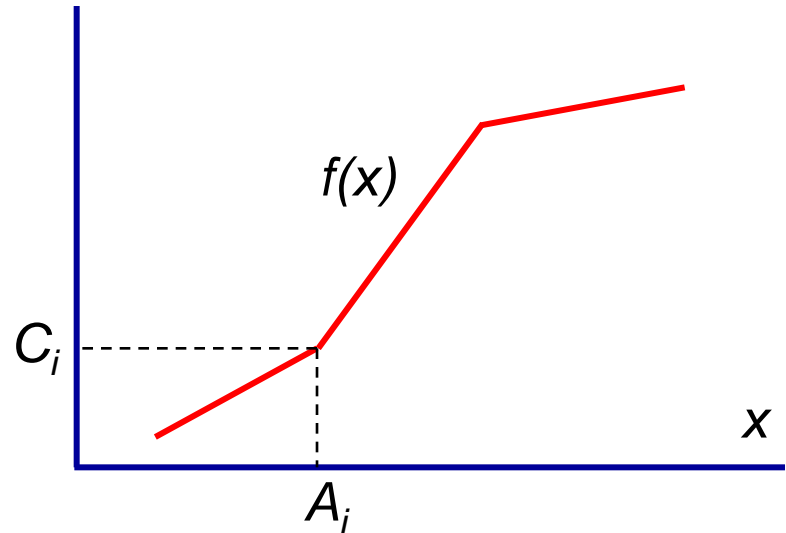
x is howmuch output

index in $\{1..n\}$

data $A, C\{\text{index}\}$

z is howmuch cost

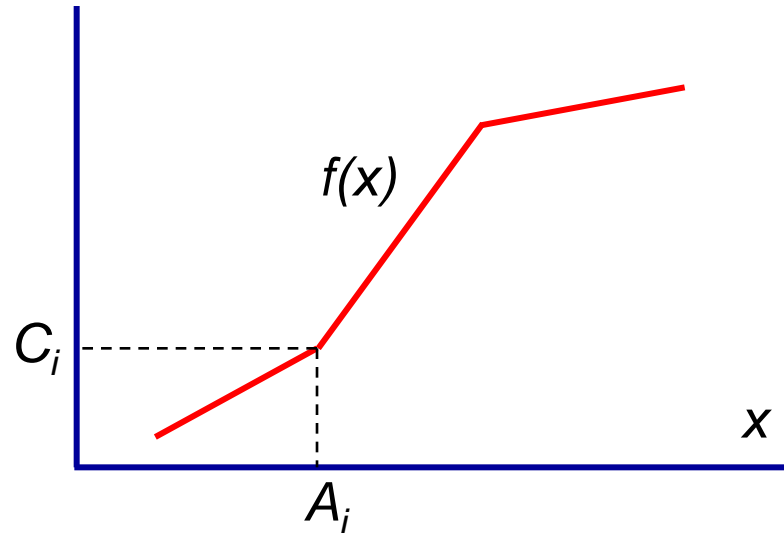
$\text{piecewise}(x, z, A, C)$ this metaconstraint defines $z = f(x)$



Piecewise linear

Piecewise linear function $z = f(x)$
Breakpoints in A , ordinates in C

***x** is howmuch output
index in {1..n}
data **A**, **C**{index}
z is howmuch cost
piecewise(**x**, **z**, **A**, **C**)*



Solver generates the model

$$x = a_1 + \sum_{i=1}^{n-1} \bar{x}_i, \quad z = c_1 + \sum_{i=1}^{n-1} \frac{c_{i+1} - c_i}{a_{i+1} - a_i} \bar{x}_i$$

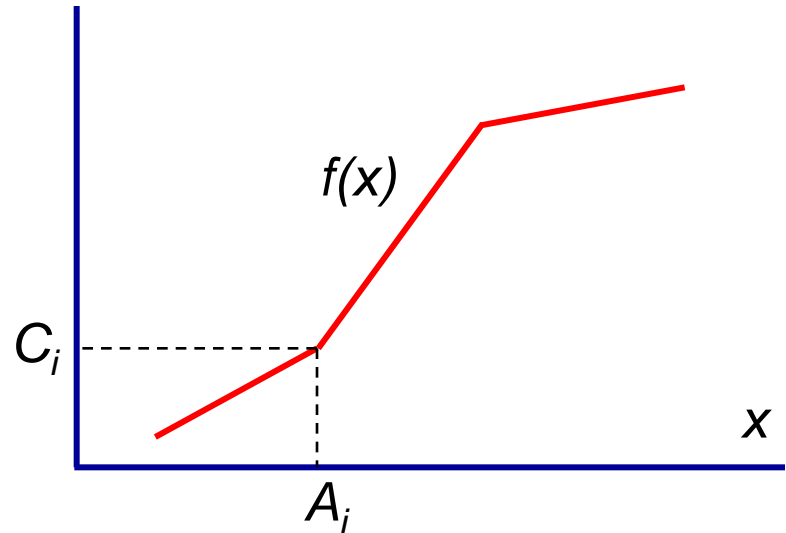
$$(a_{i+1} - a_i)\delta_{i+1} \leq \bar{x}_i \leq (a_{i+1} - a_i)\delta_i, \quad \delta_i \in \{0,1\}, \quad i = 1, \dots, n-1$$

We need to declare auxiliary variables δ_i, x_i

Piecewise linear

Piecewise linear function $z = f(x)$
Breakpoints in A , ordinates in C

*x is howmuch output
index in $\{1..n\}$
data $A, C\{\text{index}\}$
z is howmuch cost
piecewise(x, z, A, C)*



piecewise constraint induces solver to declare a new index set that associates **index** with **A**, and use it to declare δ_i, x_i

*xbar[i] is howmuch output.A(index i)
delta[i] is whether lastpositive output.A(index i)*

Both declarations create predicates inherited from **output** and **A**

Piecewise linear

Suppose there is another piecewise function on the same break points

x is howmuch output

index in $\{1..n\}$

data $A, C\{\text{index}\}$

z is howmuch cost

piecewise(x, z, A, C)

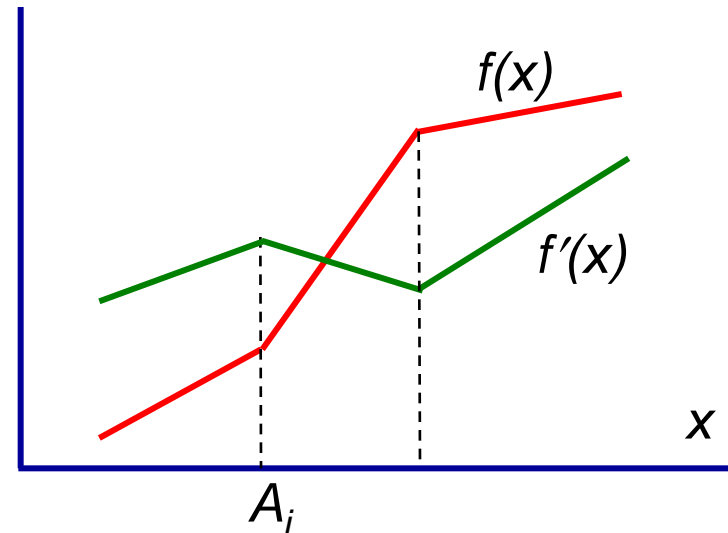
data $C'\{\text{index}\}$

z' is howmuch profit

piecewise(x, z', A, C')

$x'[i]$ is howmuch cost output. $A(\text{index } i)$

$\text{delta}'[i]$ is whether lastpositive output. $A(\text{index})$



Piecewise linear

Suppose there is another piecewise function on the same break points

x is howmuch output

index in {1..n}

data A,C{index}

z is howmuch cost

piecewise(x,z,A,C)

data C'{index}

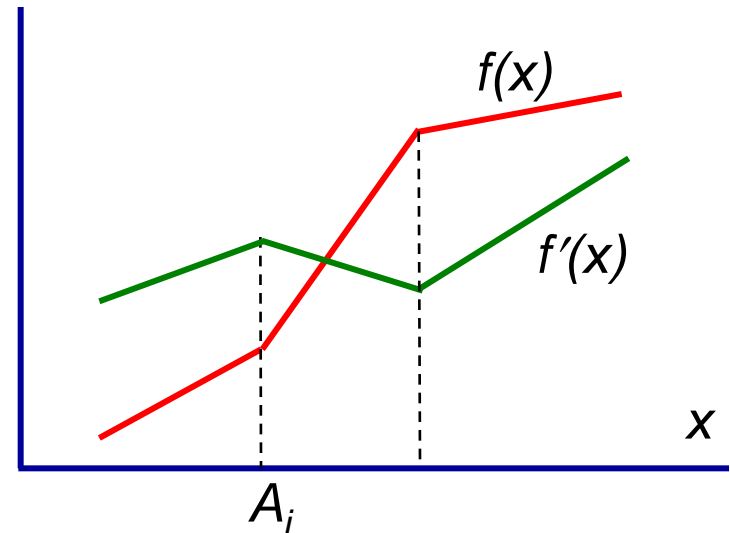
z' is howmuch profit

piecewise(x,z',A,C')

x'[i] is howmuch cost output.A(index i)

delta'[i] is whether lastpositive output.A(index)

The solver creates variables δ'_i and x'_i with same types as δ_i and x_i and so identifies them.



Because new piecewise constraint is associated with the same x and A , solver again creates **output.A**.

Interval variables

$\text{cumulative}(x, D, R, L)$

$x_j \subseteq W_j$, all j

Each job j runs for a time interval x_j .

We wish to schedule jobs so that total resource consumption never exceeds L .

`job in {1..n}`

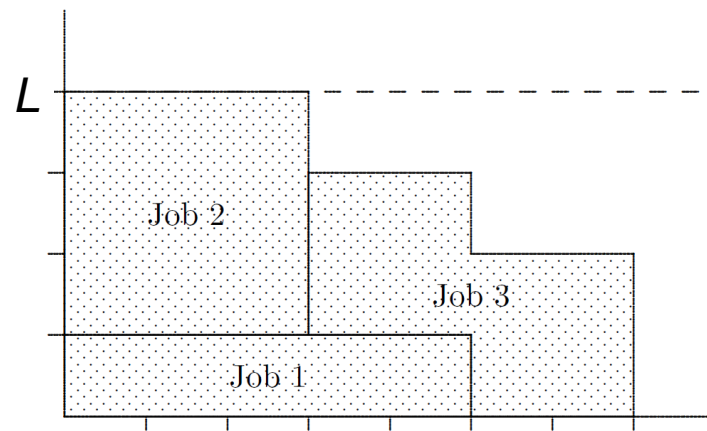
`time in {t..T}`

`data W,D,R{job} window, duration, resource`

`running in [time,time] makes running` an interval variable

`x[j] is when running sched(job j) subset W[j]`

`cumulative(x,D,R,L)`



Interval variables

$\text{cumulative}(x, D, R, L)$

$x_j \subseteq W_j$, all j

Each job j runs for a time interval x_j .

We wish to schedule jobs so that total resource consumption never exceeds L .

`job in {1..n}`

`time in {t..T}`

`data W,D,R{job} window, duration, resource`

`running in [time,time] makes running an interval variable`

`x[j] is when running sched(job j) subset W[j]`

`cumulative(x,D,R,L)`

Solver generates the model

$$\sum_t \delta_{jt} = 1, \text{ all } j; \quad \sum_j R_j \phi_{jt} \leq L, \text{ all } t$$

$$\phi_{jt} \geq \delta_{jt'}, \text{ all } t, t' \text{ with } 0 \leq t - t' < D_j, \text{ all } j$$

`delta[j,t] is whether running.start sched(job j, time t)`

`phi[j,t] is whether running sched(job j, time t)`

Interval variables

Suppose we want finish times
to be separated by at least T_0

```
job in {1..n}
time in {t..T}
data W,D,R{job}
running in [time,time]
x[j] is when running sched(job j) subset W[j]
cumulative(x,D,R,L)
{job j, job k | j<>k} |x[j].end - x[k].end| >= T0
delta[j,t] is whether running.start sched(job j, time t)
phi[j,t] is whether running sched(job j, time t)
```

$\text{cumulative}(x, D, R, L)$

$x_j \subseteq W_j, \text{ all } j$

$|x_j^{\text{end}} - x_k^{\text{end}}| \geq T_0, \text{ all } j, k, j \neq k$

Interval variables

Suppose we want finish times
to be separated by at least T_0

job in {1..n}

time in {t..T}

data W,D,R{job}

running in [time,time]

x[j] is when running sched(job j) subset W[j]

cumulative(x,D,R,L)

{job j, job k | j<>k} |x[j].end - x[k].end| >= T0

delta[j,t] is whether running.start sched(job j, time t)

phi[j,t] is whether running sched(job j, time t)

Solver generates

$$\varepsilon_{jt} + \varepsilon_{kt'} \leq 1, \text{ all } t, t' \text{ with } 0 < t' - t < T_0, \text{ all } j, t \text{ with } j \neq k$$

epsilon[j,t] is whether running.end sched(job j, time t)

$\text{cumulative}(x, D, R, L)$

$$x_j \subseteq W_j, \text{ all } j$$

$$|x_j^{\text{end}} - x_k^{\text{end}}| \geq T_0, \text{ all } j, k, j \neq k$$

Interval variables

Variables δ_{jt} and ε_{jt} are related by an offset.

Solver associates **running.end** in declaration of ε_{jt} with **running.start** in declaration of δ_{jt} and deduces

$$e_{j,t+D_j} = \delta_{jt}, \text{ all } j, t$$

$\text{delta}[j,t]$ *is whether* **running.start** **sched**(job j , time t)

$\text{phi}[j,t]$ *is whether* **running** **sched**(job j , time t)

$\text{epsilon}[j,t]$ *is whether* **running.end** **sched**(job j , time t)

Interval variables

Variables δ_{jt} and ε_{jt} are related by an offset.
Solver associates **running.end** in declaration of ε_{jt}
with **running.start** in declaration of δ_{jt} and deduces

$$e_{j,t+D_j} = \delta_{jt}, \text{ all } j, t$$

Solver also associates **running.end** in declaration of ε_{jt}
with **running** in declaration of ϕ_{jt} and deduces
the redundant constraints

$$\phi_{jt} \geq \varepsilon_{jt'}, \text{ all } t, t' \text{ with } 0 \leq t' - t < D_j, \text{ all } j$$

$\text{delta}[j,t]$ is whether **running.start** sched(job j , time t)
 $\text{phi}[j,t]$ is whether **running** sched(job j , time t)
 $\text{epsilon}[j,t]$ is whether **running.end** sched(job j , time t)

TSP with Side Constraints

Traveling salesman problem with missing arcs and precedence constraints.

city, position in $\{1..n\}$

data D{city, city}

data Prec{city, city}

data Succ{city}

Distances

Prec[i, j] = 1 if i must precede j

Succ[j] = set of successors of city j

$$\min \sum_i D_{is_i}$$

alldiff(x), circuit(s)

$x_i < x_j$, all i, j with $\text{prec}_{ij} = 1$

$s_i \in \text{Succ}_i$

TSP with Side Constraints

Traveling salesman problem with missing arcs and precedence constraints.

city, position in $\{1..n\}$

data $D\{\text{city}, \text{city}\}$ Distances

data $\text{Prec}\{\text{city}, \text{city}\}$ $\text{Prec}[i, j] = 1$ if i must precede j

data $\text{Succ}\{\text{city}\}$ $\text{Succ}[j]$ = set of successors of city j

Two variable systems:

$x[i]$ is which position ordering(city i)

$s[i]$ is successor city ordering(city i) subset $\text{Succ}[i]$

$$\min \sum_i D_{is_i}$$

$$\text{alldiff}(x), \text{circuit}(s)$$

$$x_i < x_j, \text{ all } i, j \text{ with } \text{prec}_{ij} = 1$$

$$s_i \in \text{Succ}_i$$

TSP with Side Constraints

Traveling salesman problem with missing arcs and precedence constraints.

`city, position in {1..n}`

`data D{city, city}` Distances

`data Prec{city, city}` `Prec[i,j]=1` if *i* must precede *j*

`data Succ{city}` `Succ[j]` = set of successors of city *j*

Two variable systems:

`x[i]` is which position ordering(city *i*)

`s[i]` is successor city ordering(city *i*) `subset Succ[i]`

Precedence constraints require **x** variables

`prec{city i, city j | Prec[i,j] = 1}: x[i] < x[j]`

Missing arc constraints (implicit in data **Succ**) require **s** variables

$$\min \sum_i D_{is_i}$$

`alldiff(x), circuit(s)`

$x_i < x_j$, all *i, j* with `precij = 1`

$s_i \in \text{Succ}_i$

TSP with Side Constraints

Traveling salesman problem with missing arcs and precedence constraints.

`city, position in {1..n}`

`data D{city, city}` Distances

`data Prec{city, city}` `Prec[i,j]=1` if *i* must precede *j*

`data Succ{city}` `Succ[j]` = set of successors of city *j*

Two variable systems:

`x[i]` is which position ordering(city *i*)

`s[i]` is successor city ordering(city *i*) `subset Succ[i]`

Precedence constraints require **x** variables

`prec{city i, city j | Prec[i,j] = 1}: x[i] < x[j]`

Missing arc constraints (implicit in data `Succ`) require **s** variables

`min sum {city i} D[i,s[i]]` Objective function

$$\min \sum_i D_{is_i}$$

`alldiff(x), circuit(s)`

$x_i < x_j$, all *i, j* with `precij = 1`

$s_i \in \text{Succ}_i$

TSP with Side Constraints

The solver can give `alldiff(x)` a conventional assignment model using z_{ik} = whether city i is in position k .

`z[i,k]` is whether ordering(city i , position k)

TSP with Side Constraints

The solver can give `alldiff(x)` a conventional assignment model using z_{ik} = whether city i is in position k .

`z[i,k]` is whether ordering(city i , position k)

For `circuit(s)`, the solver can introduce w_{ij} = whether city i immediately precedes city j .

`w[i,j]` is whether successor ordering(city i , city j)

TSP with Side Constraints

The solver can give `alldiff(x)` a conventional assignment model using z_{ik} = whether city i is in position k .

`z[i,k]` is whether `ordering(city i, position k)`

For `circuit(s)`, the solver can introduce w_{ij} = whether city i immediately precedes city j .

`w[i,j]` is whether `successor ordering(city i, city j)`

Declaration of `z` tells solver that predicate is `ordering(city,position)`, not `ordering(city,city)`.

TSP with Side Constraints

The solver can give `alldiff(x)` a conventional assignment model using z_{ik} = whether city i is in position k .

`z[i,k]` is whether ordering(city i , position k)

For `circuit(s)`, the solver can introduce w_{ij} = whether city i immediately precedes city j .

`w[i,j]` is whether successor ordering(city i , city j)

Declaration of `z` tells solver that predicate is `ordering(city,position)`, not `ordering(city,city)`.
Solver generates cutting planes in `w`-space and `s`-space.

TSP with Side Constraints

The solver can give `alldiff(x)` a conventional assignment model using z_{ik} = whether city i is in position k .

`z[i,k]` is whether `ordering(city i, position k)`

For `circuit(s)`, the solver can introduce w_{ij} = whether city i immediately precedes city j .

`w[i,j]` is whether `successor ordering(city i, city j)`

Declaration of `z` tells solver that predicate is `ordering(city,position)`, not `ordering(city,city)`.
Solver generates cutting planes in `w`-space and `s`-space.

The `successor` keyword tells solver how `z` and `w` relate.

$$\phi_{jt} \geq \varepsilon_{jt'}, \text{ all } t, t' \text{ with } 0 \leq t' - t < D_j, \text{ all } j$$

TSP with Side Constraints

Suppose we also have constraints on which city is in position k .
Simply declare

$y[k] = \text{which city ordering}(\text{position } k)$

The solver generates the channeling constraints between $y[k]$
and $x[i] = \text{which position is city } i$

TSP with Side Constraints

Suppose we also have constraints on which city is in position k .
Simply declare

$y[k] = \text{which city ordering}(\text{position } k)$

The solver generates the channeling constraints between $y[k]$
and $x[i] = \text{which position is city } i$

The solver can also introduce a second (equivalent) objective function

$\min \sum \{\text{position } k\} D[y[k], y[k+1]]$

which may improve bounding.

Pros and Cons of Semantic Typing

- Pros

- Conveys problem structure to the solver(s)
 - ...by allowing use of metaconstraints
- Incorporates state of the art in formulation, valid inequalities
- Allows solver to expand repertory of techniques
 - Domain filtering, propagation, cutting plane algorithms
- Good modeling practice
 - Self-documenting
 - Bug detection

Pros and Cons of Semantic Typing

- Cons
 - Modeler must be familiar with a large collection of metaconstraints
 - Rather than few primitive constraints

Pros and Cons of Semantic Typing

- Cons

- Modeler must be familiar with a large collection of metaconstraints

- Rather than few primitive constraints

- *Response*

- *Modeler must be familiar with the underlying **concepts** anyway*
 - *Modeling system can offer sophisticated help, improve modeling*

Pros and Cons of Semantic Typing

- Cons

- Modeler must be familiar with a large collection of metaconstraints
 - Rather than few primitive constraints
- *Response*
 - *Modeler must be familiar with the underlying **concepts** anyway*
 - *Modeling system can offer sophisticated help, improve modeling*
- OR, SAT community is not accustomed to high-level modeling
 - Typed languages like Ascend never really caught on.

Pros and Cons of Semantic Typing

- Cons

- Modeler must be familiar with a large collection of metaconstraints
 - Rather than few primitive constraints
- *Response*
 - *Modeler must be familiar with the underlying **concepts** anyway*
 - *Modeling system can offer sophisticated help, improve modeling*
- OR, SAT community is not accustomed to high-level modeling
 - Typed languages like Ascend never really caught on.
- *Response*
 - *Train the next generation!*