Modeling Techniques in Constraint Programming

Pascal Van Hentenryck

Brown University

Outline

- Symmetries
- Auxiliary Variables
- Redundant Constraints
- Dual Modeling

Symmetries

- Many problems naturally exhibit symmetries
  - exploring symmetrical solutions is useless
- Many kinds of symmetries
  - variable symmetries
  - value symmetries
- The next slides
  - symmetry-breaking constraints

BIBD

- Balanced incomplete block designs
  - Input: (v,b,r,k,l).
  - Output: a v by b binary matrix with exactly r ones per row, k ones per column, and with a scalar product of value l
- Why BIBD?
  - Combinatorial design
  - Full of variable symmetries

BIBD

```
range Rows = 1..v;
range Cols = 1..b;
var<CP>{bool} M[Rows,Cols](cp);
solve<cp> {
   forall(i in Rows)
      cp.post(sum(x in Cols) M[i,x] == r);
   forall(i in Cols)
      cp.post(sum(x in Rows) M[x,i] == k);
   forall(i in Rows,j in i+1..v)
      cp.post(sum(x in Cols) (M[i,x] && M[j,x]) == 1);
}
```

BIBD

- (7,7,3,3,1)

0	1	1	0	0	1	0
1	0	1	0	1	0	0
0	0	1	1	0	0	1
1	1	0	0	0	0	1
0	0	0	0	1	1	1
1	0	0	1	0	1	0
0	1	0	1	1	0	0

- (7,7,3,3,1)

1	0	1	0	1	0	0
0	1	1	0	0	1	0
0	0	1	1	0	0	1
1	1	0	0	0	0	1
0	0	0	0	1	1	1
1	0	0	1	0	1	0
0	1	0	1	1	0	0

BIBD

- (7,7,3,3,1)

0	1	1	0	0	1	0
1	0	1	0	1	0	0
0	0	1	1	0	0	1
1	1	0	0	0	0	1
0	0	0	0	1	1	1
1	0	0	1	0	1	0
0	1	0	1	1	0	0

- (7,7,3,3,1)

0	1	1	0	0	1	0
1	0	1	0	1	0	0
0	0	1	1	0	0	1
1	0	0	0	0	1	1
0	1	0	0	1	0	1
1	1	0	1	0	0	0
0	0	0	1	1	1	0

BIBD

- How to break the variable symmetries
  - impose an ordering on the variables
- Consider the row
  - impose a lexicographic constraint
- Lexicographic ordering

a:	0 1 1 0 0 1 0	1 1 1 0 0 1 0
b:	1 0 1 0 1 0 0	1 0 1 0 1 0 0
	a <= b	a >= b

BIBD

- (7,7,3,3,1)

0	1	1	0	0	1	0
1	0	1	0	1	0	0
0	0	1	1	0	0	1
1	1	0	0	0	0	1
0	0	0	0	1	1	1
1	0	0	1	0	1	0
0	1	0	1	1	0	0

- (7,7,3,3,1)

0	0	0	0	1	1	1
0	0	1	1	0	0	1
0	1	0	1	1	0	0
0	1	1	0	0	1	0
1	0	0	1	0	1	0
1	0	1	0	1	0	0
1	1	0	0	0	0	1

BIBD

- (7,7,3,3,1)

0	1	1	0	0	1	0
1	0	1	0	1	0	0
0	0	1	1	0	0	1
1	1	0	0	0	0	1
0	0	0	0	1	1	1
1	0	0	1	0	1	0
0	1	0	1	1	0	0

- (7,7,3,3,1)

0	0	0	0	1	1	1
0	0	1	1	0	0	1
0	1	0	1	1	0	0
0	1	1	0	0	1	0
1	0	0	1	0	1	0
1	0	1	0	1	0	0
1	1	0	0	0	0	1

BIBD

- (7,7,3,3,1)

0	0	0	0	1	1	1
0	0	1	1	0	0	1
0	1	0	1	1	0	0
0	1	1	0	0	1	0
1	0	0	1	0	1	0
1	0	1	0	1	0	0
1	1	0	0	0	0	1

- (7,7,3,3,1)

0	0	0	0	1	1	1
0	0	1	1	0	0	1
0	1	0	1	0	1	0
0	1	1	0	1	0	0
1	0	0	1	1	0	0
1	0	1	0	0	1	0
1	1	0	0	0	0	1

BIBD

```
range Rows = 1..v;
range Cols = 1..b;
var<CP>{bool} M[Rows,Cols](cp);
solve<cp> {
  forall(i in Rows)
    cp.post(sum(x in Cols) M[i,x] == r);
  forall(i in Cols)
    cp.post(sum(x in Rows) M[x,i] == k);
  forall(i in Rows,j in i+1..v)
    cp.post(sum(x in Cols) (M[i,x] && M[j,x]) == l);
  forall(i in 1..v-1)
    cp.post(lexleq(all(j in Cols) M[i,j],all(j in Cols) M[i+1,j]));
  forall(j in 1..b-1)
    cp.post(lexleq(all(i in Rows) M[i,j],all(i in Rows) M[i,j+1]));
}
```

rows

column

Scene Allocation

- Shooting scenes for a movie
  - an actor plays in some of the scenes
  - at most k scenes a day
  - each actor are paid each day they play
- Objective
  - minimize the total cost
- Difficulty
  - expressing the objective
  - symmetries

Scene Allocation

```
int   maxScene = …;
range Scenes = …;
range Days  = …;
range Actor = …;
int   fee[Actor] = …;
set{Actor} appears[Scenes] = …;
set{int} which[a in Actor] = setof(i in Scenes) member(a,appears[i]);
var<CP>{int} shoot[Scenes](cp,Days);

minimize<cp>
  sum(a in Actor) sum(d in Days) fee[a] * or(s in which[a]) (shoot[s]==d)
subject to
  cp.post(atmost(all(k in Days) 5,shoot));
```

Scene Allocation

- Value Symmetries
  - the days are interchangeable
  - If s is a solution, p(s) is a solution, where p(s) denotes the solution s where the days have been permuted with permutation p
- How to eliminate value symmetries
  - Consider the first scene x_1. Which day must be candidate assignments?

Scene Allocation

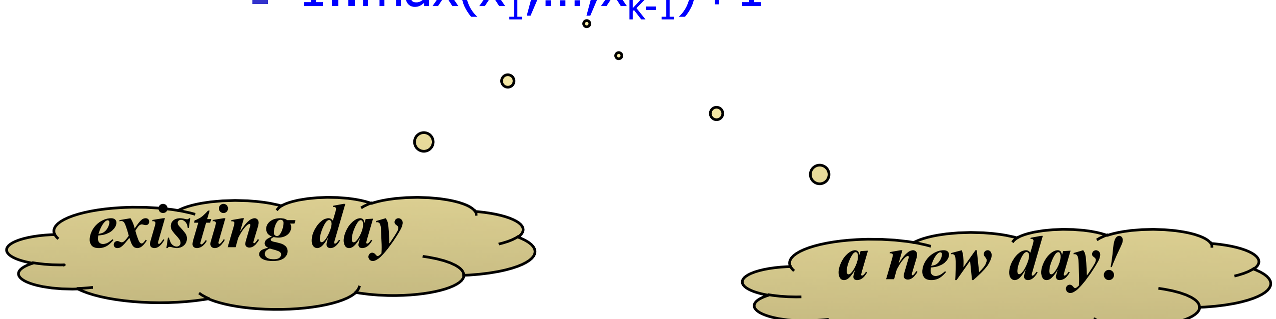
- Value Symmetries
  - the days are interchangeable
  - If s is a solution, p(s) is a solution, where p(s) denotes the solution s where the days have been permuted with permutation p
- How to eliminate value symmetries
  - Consider the first scene x_1. Which day must be candidate assignments?
  - Only one day: say 1
  - All days are interchangeable at this point!

Scene Allocation

- Value Symmetries
  - the days are interchangeable
  - If s is a solution, p(s) is a solution, where p(s) denotes the solution s where the days have been permuted with permutation p
- How to eliminate value symmetries
  - Consider the first scene x_1. Which day must be candidate assignments? Only one day: say 1
  - Consider the second scene x_2. Which day must be considered?
    - either 1 (the day of x_1) or a single new day, say 2.

Scene Allocation

- Value Symmetries
  - the days are interchangeable
  - If s is a solution, p(s) is a solution, where p(s) denotes the solution s where the days have been permuted with permutation p
- How to eliminate value symmetries
  - Consider the scene x_k. Which day must be considered?
    - $1..\max(x_1,\dots,x_{k-1})+1$



Scene Allocation

```
var<CP>{int} shoot[Scenes](cp,Days);

minimize<cp>
  sum(a in Actor) sum(d in Days) fee[a] * or(s in which[a]) (shoot[s]==d)
subject to {
  cp.post(atmost(all(k in Days) 5,shoot));
  cp.post(scene[Scenes.getLow()] == Days.getLow());
  forall(s in Scenes: s != Scenes.getLow())
     cp.post(scene[s] <= max(k in 1..s-1) scene[k] + 1);
}
```

- This eliminates all the value symmetries
  - there is a limitation however

Outline

- Symmetries
- Auxiliary Variables
- Redundant Constraints
- Dual Modeling

Auxiliary variables

- Motivation
  - factorize common expressions and constraints
  - make it easier to state the problem constraints

Auxiliary variables

- Motivation
  - factorize common expressions and constraints
  - make it easier to state the problem constraints

Outline

- Symmetries
- Auxiliary Variables
- Redundant Constraints
- Dual Modeling

Redundant Constraints

- Motivation
  - semantically redundant: do not exclude any solution
  - computationally significant: reduce the search space
- What are redundant constraints?
  - express properties of the solutions (not explicited operationally)

Magic Series

- A series $S = (S_0,\ldots,S_n)$ is magic if S_i is the number of occurrences of i in S

0	1	2	3	4
?	?	?	?	17

Magic Series

- A series $S = (S_0,\ldots,S_n)$ is magic if S_i is the number of occurrences of i in S

0	1	2	3	4
2	1	2	0	0

Magic Series

```
int n = 5;
range D = 0..n-1;
var<CP>{int} s[D](cp,D);
solve<cp> {
  forall(k in D)
    cp.post(s[k] == sum(i in D) (s[i]==k));
}
```

- Redundant constraint
  - Can we find a property of the solution?

Magic Series

- A series $S = (S_0,\ldots,S_n)$ is magic if S_i is the number of occurrences of i in S

0	1	2	3	4
?	?	?	?	17

Magic Series

```
int n = 5;
range D = 0..n-1;
var<CP>{int} s[D](cp,D);
solve<cp> {
  forall(k in D)
    cp.post(s[k] == sum(i in D) (s[i]==k));
  cp.post(sum(k in D) s[k] == n);
}
```

- Redundant constraint
  - there are n numbers in the series

Magic Series

```
int n = 5;
range D = 0..n-1;
var<CP>{int} s[D](cp,D);
explore<cp> {
  forall(k in D)
    cp.post(s[k] == sum(i in D) (s[i]==k));
  cp.post(sum(k in D) s[k] == n);
}
```

- Redundant constraint
  - can I reexpress **`sum(k in D) s[k]`**?

Magic Series

- A series $S = (S_0,\ldots,S_n)$ is magic if S_i is the number of occurrences of i in S

0	1	2	3	4
2	1	2	0	0

Magic Series

```
int n = 5;
range D = 0..n-1;
var<CP>{int} s[D](cp,D);
solve<cp> {
  forall(k in D)
    cp.post(s[k] == sum(i in D) (s[i]==k));
  cp.post(sum(k in D) s[k] == n);
  cp.post(sum(k in D) k*s[k] == n);
}
```

- Redundant constraint
  - can I reexpress `sum(k in D) s[k]`?

Magic Series

```
s[0] == (s[0]==0)+(s[1]==0)+(s[2]==0)+(s[3]==0)+(s[4]==0)
s[1] == (s[0]==1)+(s[1]==1)+(s[2]==1)+(s[3]==1)+(s[4]==1)
s[2] == (s[0]==2)+(s[1]==2)+(s[2]==2)+(s[3]==2)+(s[4]==2)
s[3] == (s[0]==3)+(s[1]==3)+(s[2]==3)+(s[3]==3)+(s[4]==3)
s[4] == (s[0]==4)+(s[1]==4)+(s[2]==4)+(s[3]==4)+(s[4]==4)
s[1] + 2*s[2]+ 3*s[3] + 4*s[4] = 5;
```

Magic Series

```
s[0] == (s[0]==0)+(s[1]==0)+(s[2]==0)+(s[3]==0)+(s[4]==0)
s[1] == (s[0]==1)+(s[1]==1)+(s[2]==1)+(s[3]==1)+(s[4]==1)
s[2] == (s[0]==2)+(s[1]==2)+(s[2]==2)
s[3] == (s[0]==3)+(s[1]==3)
s[4] == (s[0]==4)+(s[1]==4)
s[1] + 2*s[2]+ 3*s[3] + 4*s[4] = 5;
```

Magic Series

- Assume s[0]=2

```
2    ==    (s[1]==0)+(s[2]==0)+(s[3]==0)+(s[4]==0)
s[1] ==    (s[1]==1)+(s[2]==1)+(s[3]==1)+(s[4]==1)
s[2] == 1 +(s[1]==2)+(s[2]==2)
s[3] ==    (s[1]==3)
s[4] ==    (s[1]==4)
s[1] + 2*s[2]+ 3*s[3] + 4*s[4] = 5;
```

```
s[2] >= 1
s[1] + 3*s[3] + 4*s[4] <= 3
```

Magic Series

- Assume s[0]=2

```
2    ==    (s[1]==0)+(s[2]==0)+(s[3]==0)+(0==0)
s[1] ==    (s[1]==1)+(s[2]==1)+(s[3]==1)+(0==1)
s[2] == 1 +(s[1]==2)+(s[2]==2)
s[3] ==    (s[1]==3)
0    ==    (s[1]==4)
s[1] + 2*s[2]+ 3*s[3] + 4*0 = 5;
```

```
s[2] >= 1
s[1] + 3*s[3] + 4*s[4] <= 3
```

Magic Series

- Assume s[0]=2

```
1    ==    (s[1]==0)+(s[2]==0)+(s[3]==0)
s[1] ==    (s[1]==1)+(s[2]==1)+(s[3]==1)
s[2] == 1 +(s[1]==2)+(s[2]==2)
s[3] ==    (s[1]==3)

s[1] + 2*s[2]+ 3*s[3] = 5;
```

```
s[2] >= 1
s[1] + 3*s[3] + 4*s[4] <= 3
```

Magic Series

- Assume s[0]=2

```
1    ==    (s[1]==0)+         +(s[3]==0)
s[1] ==    (s[1]==1)+(s[2]==1)+(s[3]==1)
s[2] == 1 +(s[1]==2)+(s[2]==2)
s[3] ==    (s[1]==3)

s[1] + 2*s[2]+ 3*s[3] = 5;
```

Magic Series

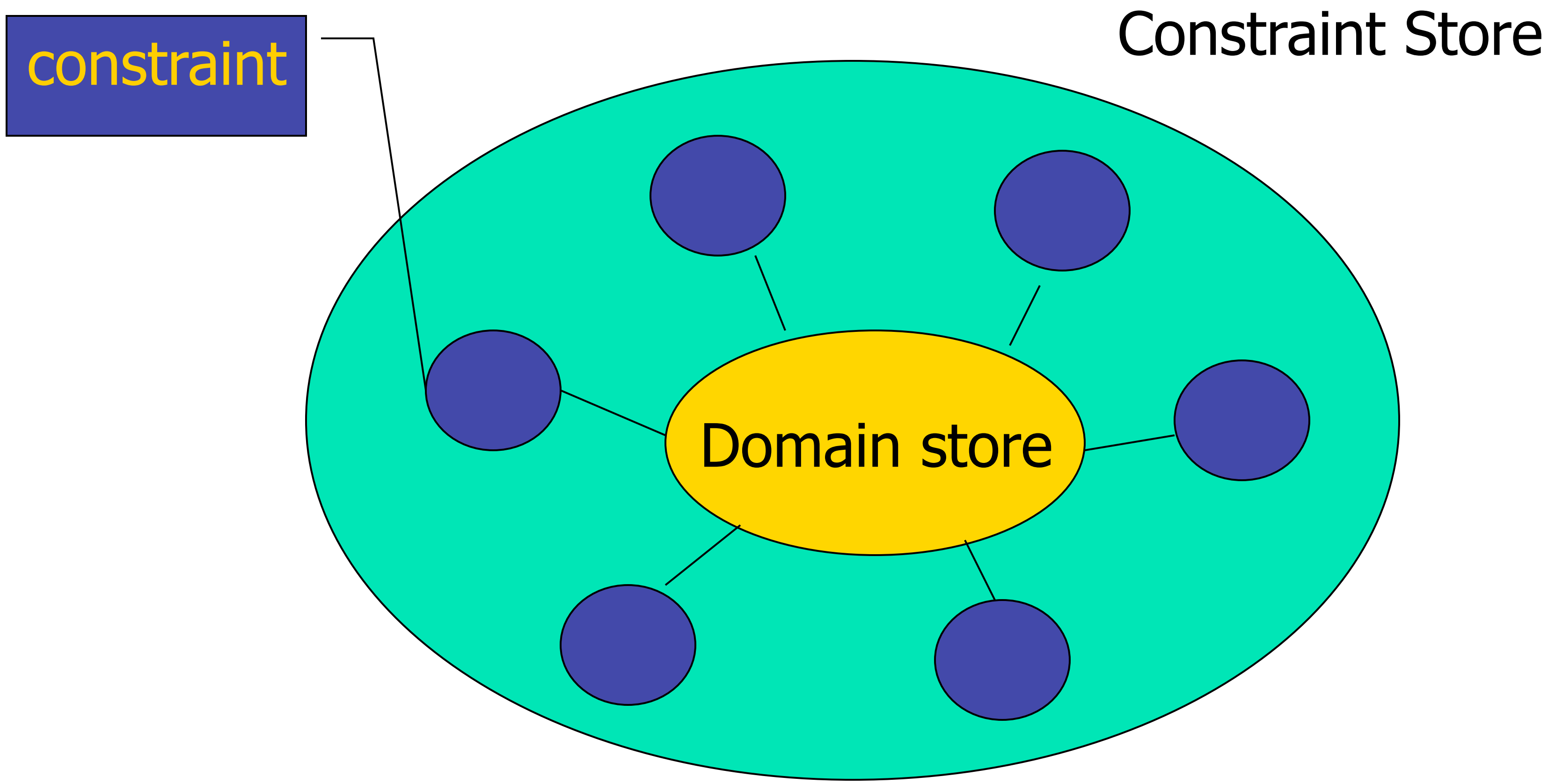
- Assume s[0]=2 and s[1] = 1

```
1    ==    (s[1]==0)+          +(s[3]==0)
1    ==    (s[1]==1)+(s[2]==1)+(s[3]==1)
s[2] == 1 +(s[1]==2)+(s[2]==2)
s[3] ==    (s[1]==3)

s[1] + 2*s[2]+ 3*s[3] = 5;
```

Redundant Constraints

- First role
  - express properties of the solutions
  - boost the propagation of other constraints

Computational Model



Redundant Constraints

- First role
  - express properties of the solutions
  - boost the propagation of other constraints
- Second role
  - provide a more global view
  - combine existing constraints
  - improve communication

Market Split

```
int w[C,V];
int rhs[C];
var<CP>{int} x[V](cp,0..1);

solve<cp> {

   forall(c in C)
      cp.post(sum(v in V) w[c,v] * x[v] == rhs[c]);

}
using { … }
```

- Observe
  - the equations only communicates through the domains

Market Split

```
int w[C,V];
int rhs[C];
var<CP>{int} x[V](cp,0..1);
int alpha = 5;
solve<cp> {

   forall(c in C)
      cp.post(sum(v in V) w[c,v] * x[v] == rhs[c]);
   cp.post(sum(v in V) (sum(c in C) alpha^c * w[c,v])* x[v]
           == sum(c in C) alpha^c * rhs[c]);
}
```

- Redundant constraints
  - combinations of other constraints: surrogate constraints

Market Split

```
int w[C,V];
int rhs[C];
var<CP>{int} x[V](cp,0..1);
solve<cp> {

   forall(c in C)
      cp.post(binaryKnapsack(x,w[c],rhs[c]));

}
```

global Constraint

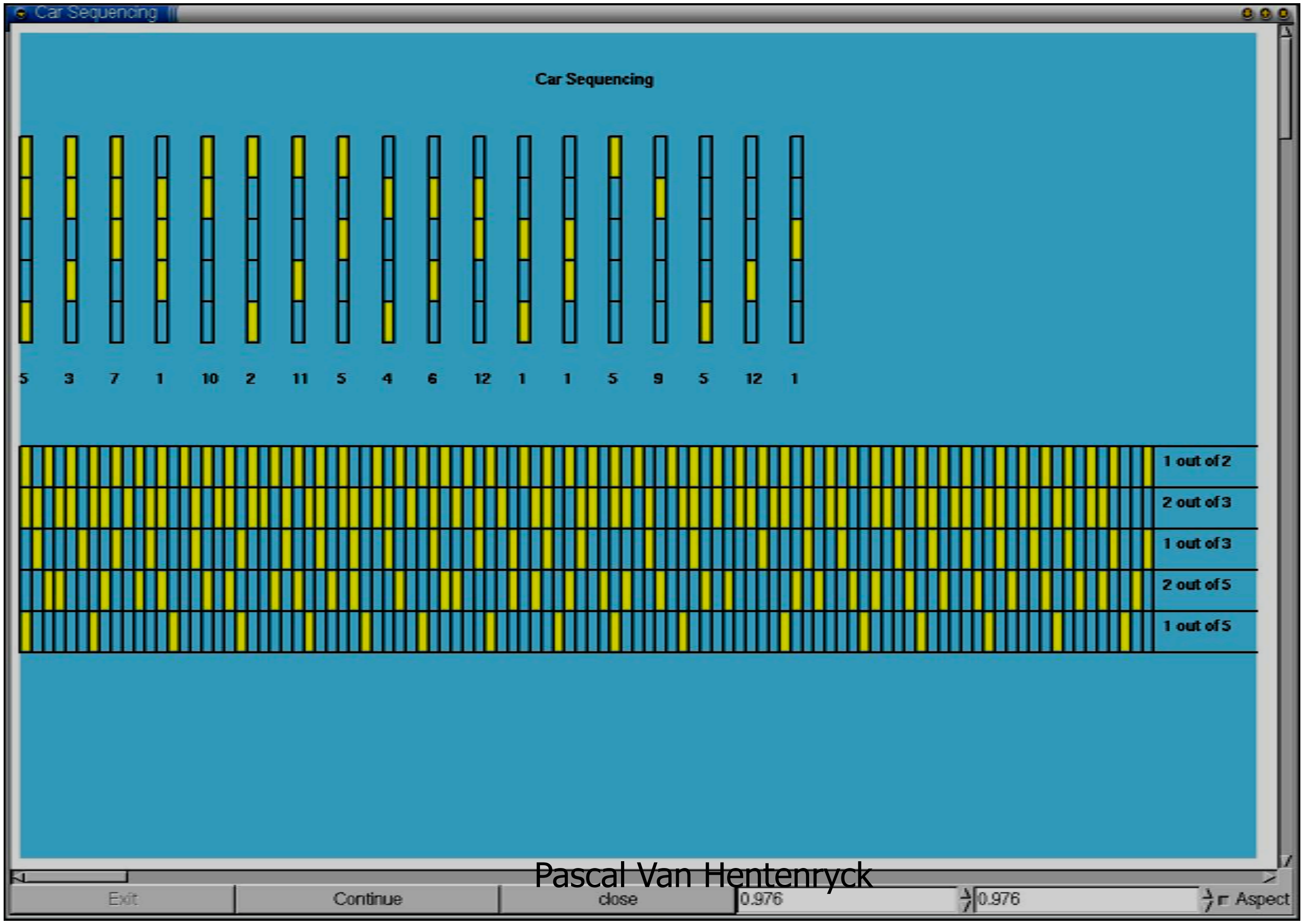
Redundant Constraints

- First role
  - express properties of the solutions
  - boost the propagation of other constraints
- Second role
  - provide a more global view
  - combine existing constraints
- Third role
  - provide a more global view
  - derive a consequence of existing constraints

Car Sequencing

- Cars on an assembly line
- Cars have options (e.g. leather seats)
- Capacity constraints on the production units (2 out of 5)
- Sequencing

Car sequencing



Small example

Options	1	2	3	4	5	demand
Class 1	yes		yes	yes		1
Class 2				yes		1
Class 3		yes			yes	2
Class 4		yes		yes		2
Class 5	yes		yes			2
Class 6	yes	yes				2
capacity	1/2	2/3	1/3	2/5	1/5	

Car Sequencing

```
range Cars = …;
range Configs = …;
range Options = …;
int demand[Configs] = …;
int lb[Options] = …;
int ub[Options] = …;
int requires[Options,Config] = …;

var<CP>{int} line[Cars](cp,Configs);
var<CP>{int} setup[Options,Cars](cp,0..1);
```

Car Sequencing

```
solve<cp> {

   forall(c in Configs)
      cp.post(sum(s in Cars) (line[s] == c) == demand[c]);

   forall(s in Cars,o in Options)
      cp.post(setup[o,s] == requires[o,line[s]]);

   forall(o in Options, s in 1..nbCars-ub[o]+1)
      cp.post(sum(j in s..s+ub[o]-1) setup[o,j] <= lb[o]);
}
```

demand

capacity

Car sequencing

Slots	1	2	3	4	5	6	7	8	9	10	D
Cl 1											1
Cl 2											1
Cl 3											2
Cl 4											2
Cl 5											2
Cl 6											2

Car sequencing

	1	2	3	4	5	6	7	8	9	10	
O1											1/2
O2											2/3
O3											1/3
O4											2/5
O5											1/5

Element:y->x

Capacity constraints

Small example

Options	1	2	3	4	5	demand
Class 1	yes		yes	yes		1
Class 2				yes		1
Class 3		yes			yes	2
Class 4		yes		yes		2
Class 5	yes		yes			2
Class 6	yes	yes				2
capacity	1/2	2/3	1/3	2/5	1/5	

Car sequencing

Element: x -> y

Slots	1	2	3	4	5	6	7	8	9	10	D
Cl 1											1
Cl 2											1
Cl 3											2
Cl 4											2
Cl 5											2
Cl 6											2

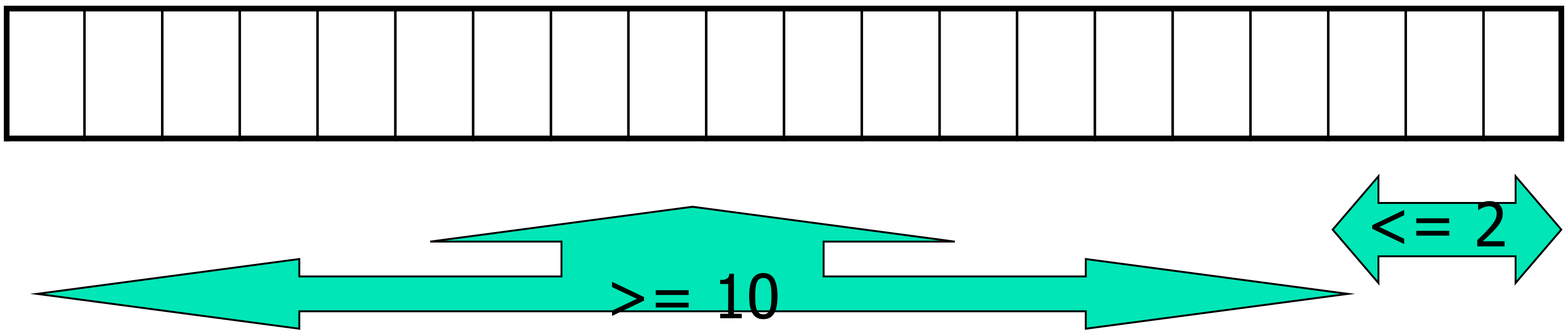
Car Sequencing

```
solve<cp> {

   forall(c in Configs)
      cp.post(sum(s in Cars) (line[s] == c) == demand[c]);

   forall(s in Cars,o in Options)
      cp.post(setup[o,s] == requires[o,line[s]]);

   forall(o in Options, s in 1..nbCars-ub[o]+1)
      cp.post(sum(j in s..s+ub[o]-1) setup[o,j] <= lb[o]);
}
using
   label(line);
```

demand

capacity

Redundant constraints



- Capacity 2/3
- Demand 12

Car Sequencing

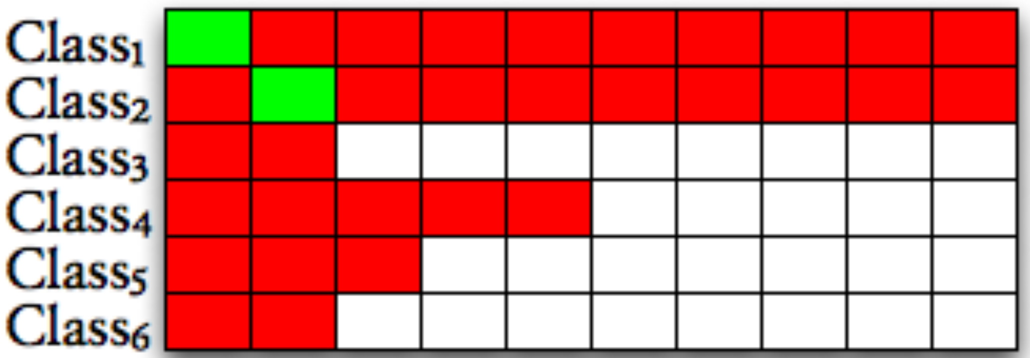
```
solve<cp> {
  forall(c in Configs)
    cp.post(sum(s in Cars) (line[s] == c) == demand[c]);

  forall(s in Cars,o in Options)
    cp.post(setup[o,s] == requires[o,line[s]]);

  forall(o in Options, s in 1..nbCars-ub[o]+1)
    cp.post(sum(j in s..s+ub[o]-1) setup[o,j] <= lb[o]);

  forall(o in Options, i in 1..demand[o])
   cp.post(sum(s in 1..nbCars-i*ub[o]) setup[o,s] >= demand[o]-i*lb[o]);
}
```

Element: x -> y

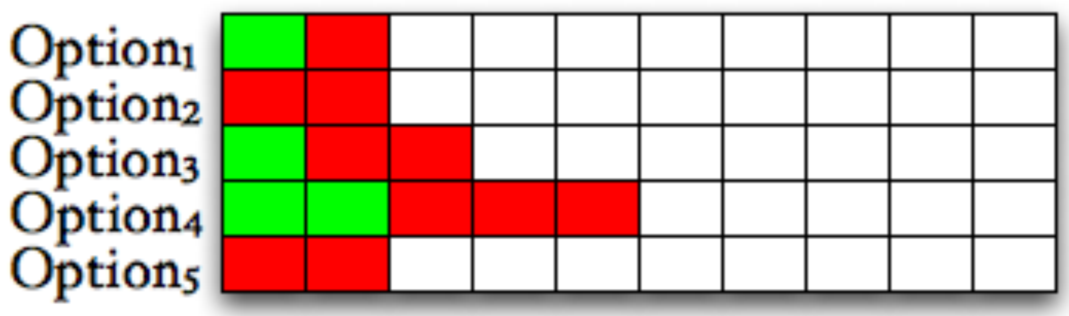
Car sequencing

Slots	1	2	3	4	5	6	7	8	9	10	D
Cl 1											1
Cl 2											1
Cl 3											2
Cl 4											2
Cl 5											2
Cl 6											2

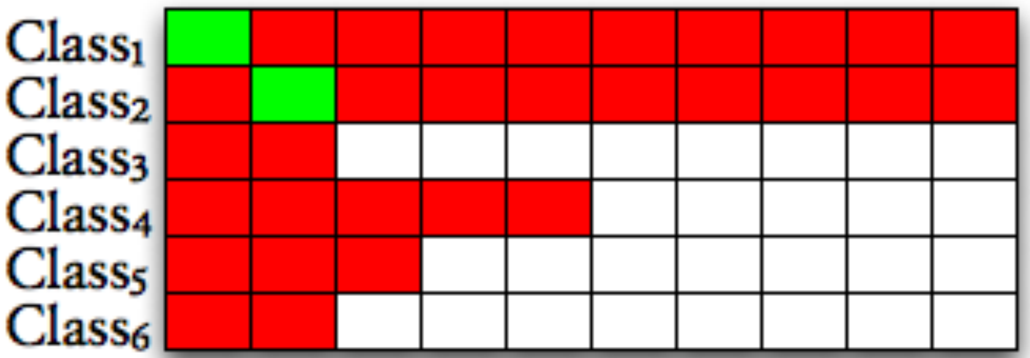
Propagation



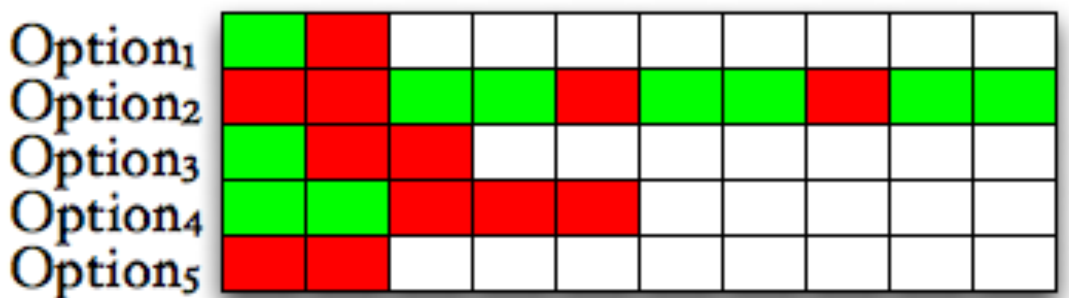
Options	1	2	3	4	5	Demand
Class 1	✔		✔	✔		1
Class 2				✔		1
Class 3		✔			✔	2
Class 4		✔		✔		2
Class 5	✔		✔			2
Class 6	✔	✔				2
Capacity	1/2	2/3	1/3	2/5	1/5	



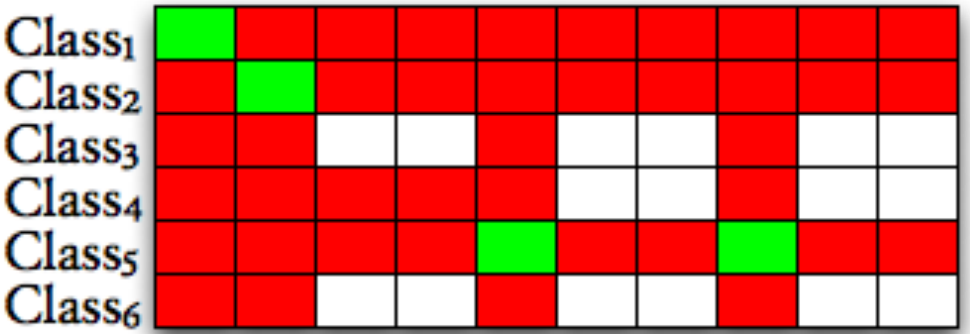
Redundant Impact



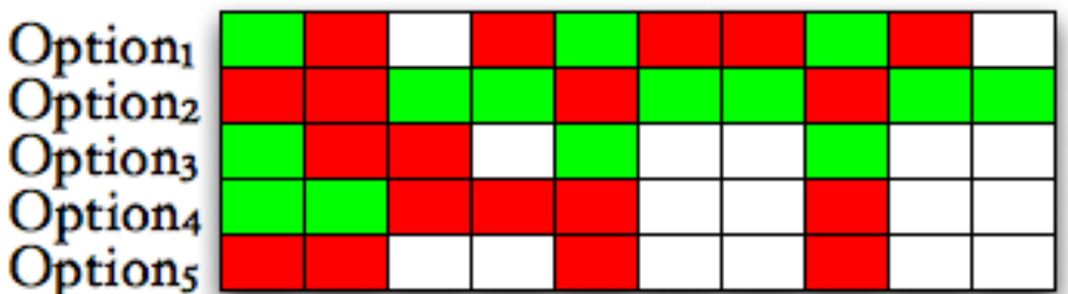
Options	1	2	3	4	5	Demand
Class 1	✔		✔	✔		1
Class 2				✔		1
Class 3		✔			✔	2
Class 4		✔		✔		2
Class 5	✔		✔			2
Class 6	✔	✔				2
Capacity	1/2	2/3	1/3	2/5	1/5	



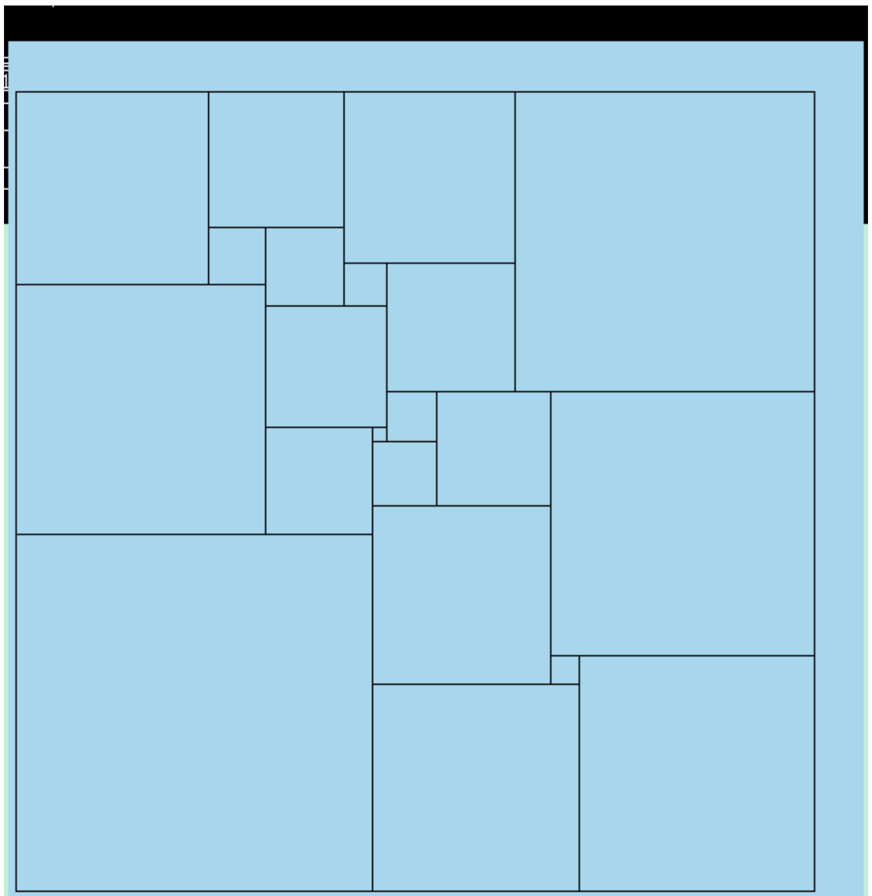
Final Propagation



Options	1	2	3	4	5	Demand
Class 1	✔		✔	✔		1
Class 2				✔		1
Class 3		✔			✔	2
Class 4		✔		✔		2
Class 5	✔		✔			2
Class 6	✔	✔				2
Capacity	1/2	2/3	1/3	2/5	1/5	



The Perfect Square Problem



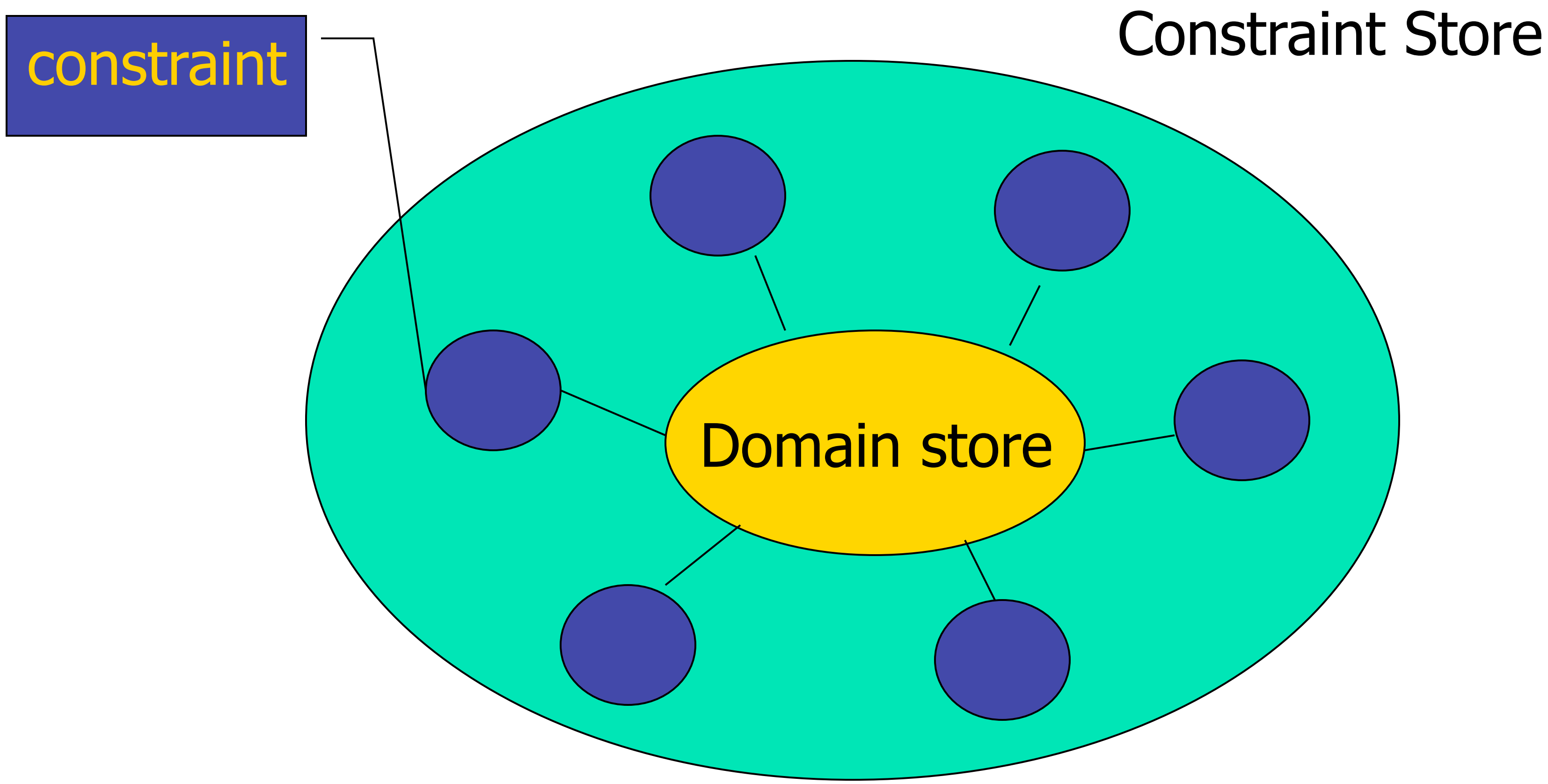
The Perfect Square Model

```
int   s    = 112;range Side = 1..s; range Square = 1..21;
int side[Square] = [50,42,37,35,33,29,27,25,24,19,18,17,16,15,11,9,8,7,6,4,2];
var<CP>{int} x[i in Square](cp,Side);
var<CP>{int} y[i in Square](cp,Side);
solveall<cp> {
  forall(i in Square) {
      cp.post(x[i]<=s-side[i]+1); cp.post(y[i]<=s-side[i]+1); }
  forall(i in Square,j in Square: i<j)
    cp.post(x[i]+side[i]<= x[j] || x[j]+side[j]<=x[i] ||  y[i]+side[i]<=y[j] || y[j]+side[j]<=y[i]);

  forall(p in Side) {
     cp.post(sum(i in Square) side[i]*((x[i]<=p) && (x[i]>=p-side[i]+1)) == s);
     cp.post(sum(i in Square) side[i]*((y[i]<=p) && (y[i]>=p-side[i]+1)) == s);
  }
}
```

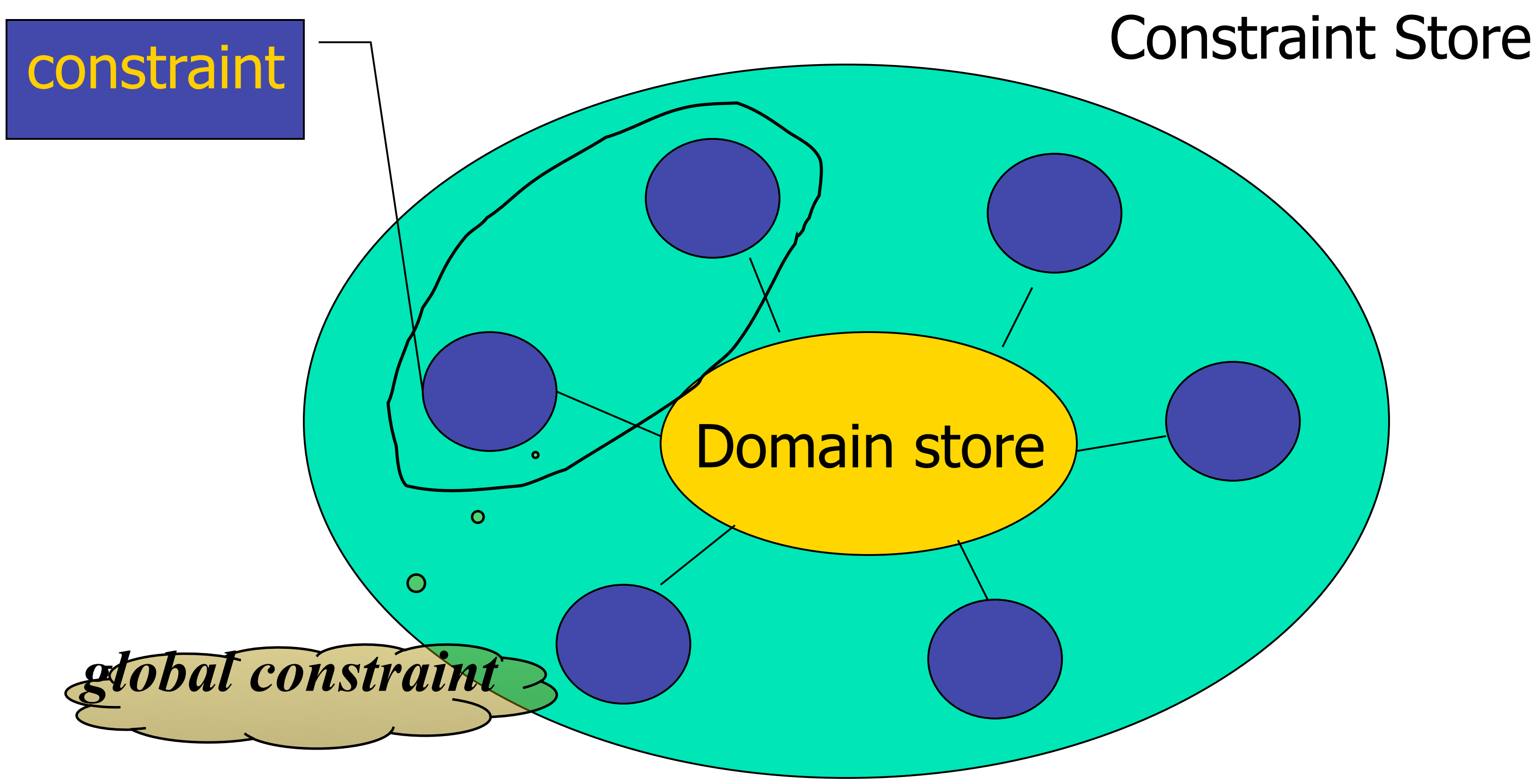
no overlap

Redundant constraints

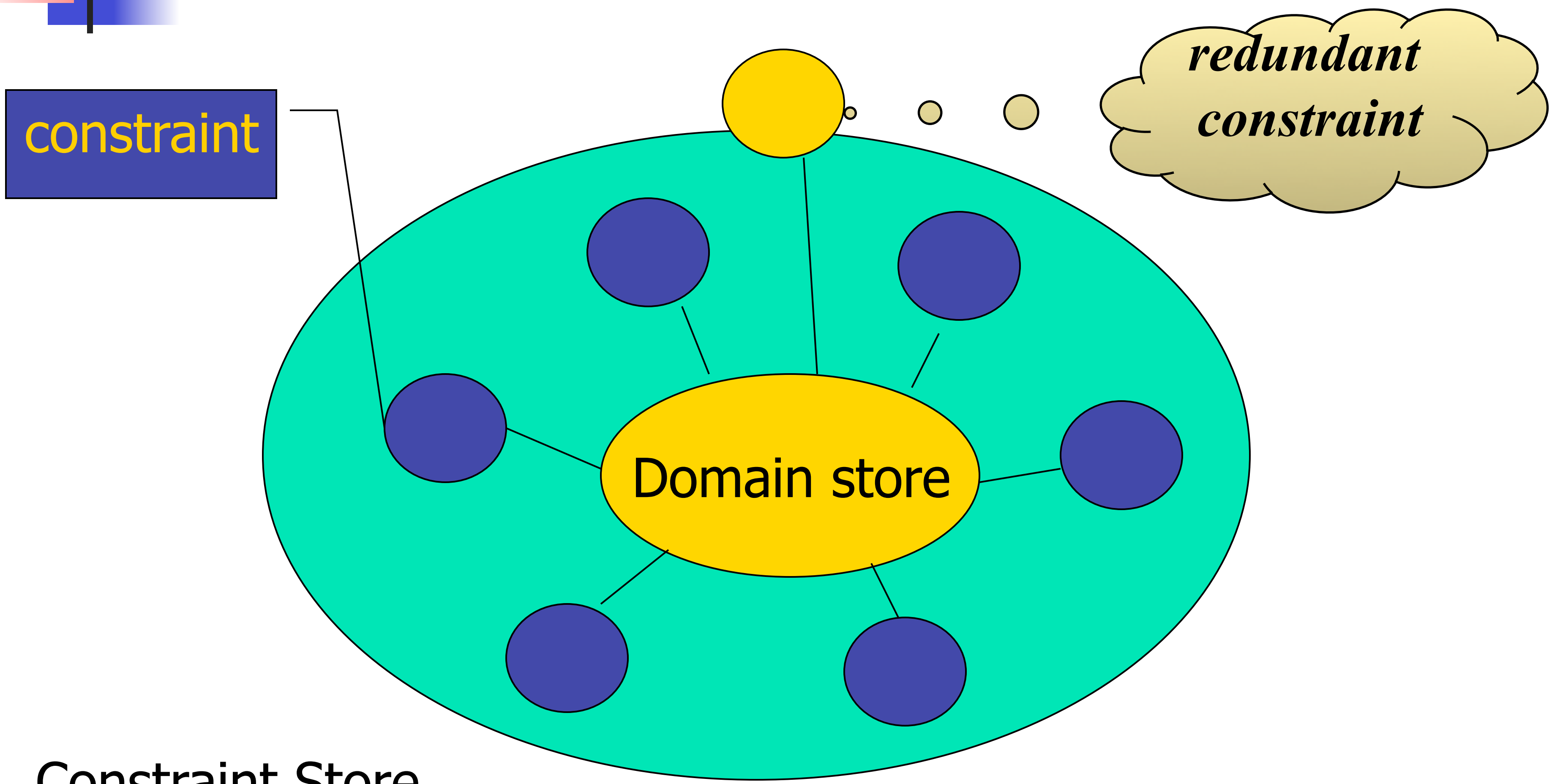
Computational Model



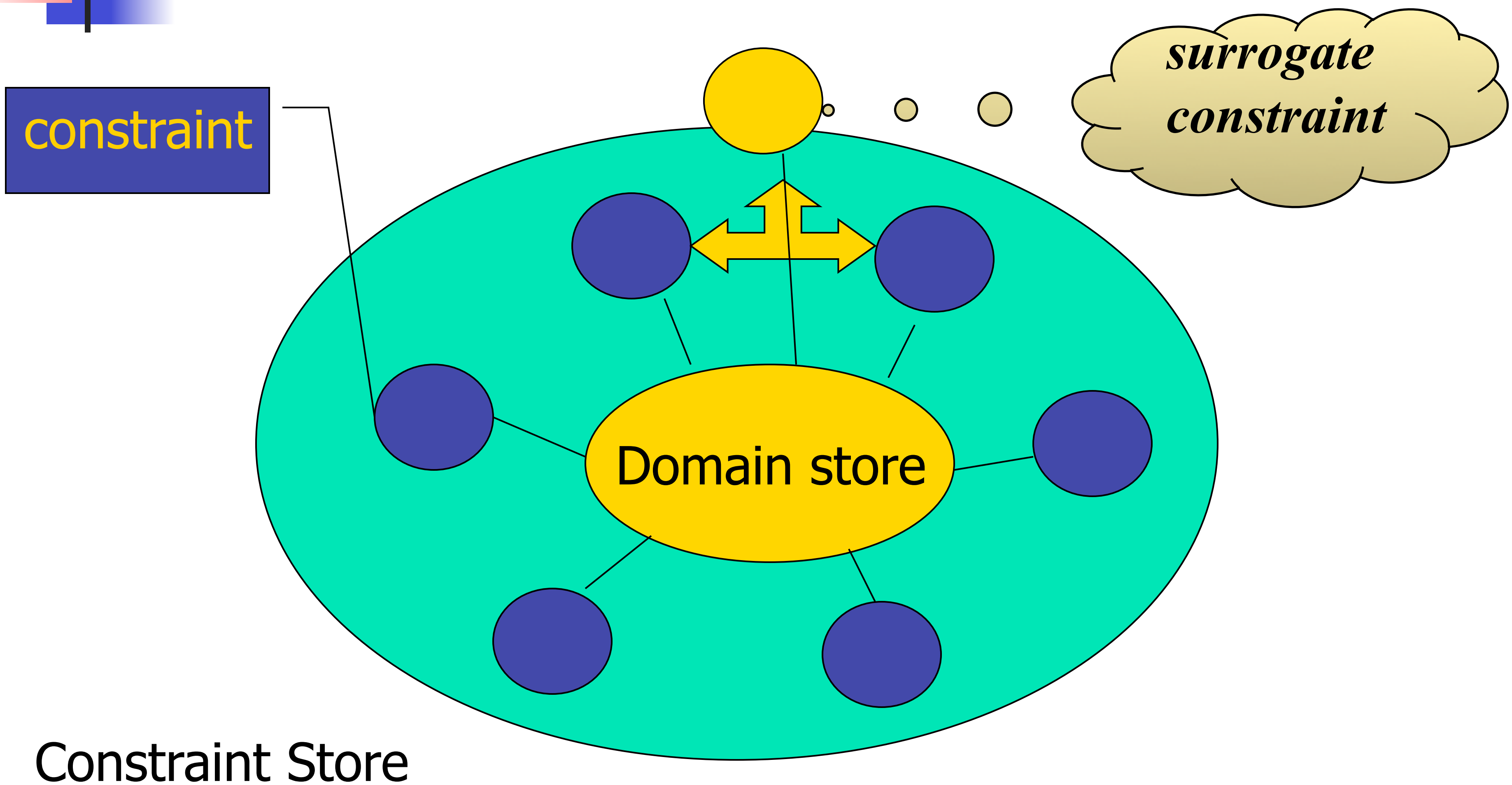
Improving Communication



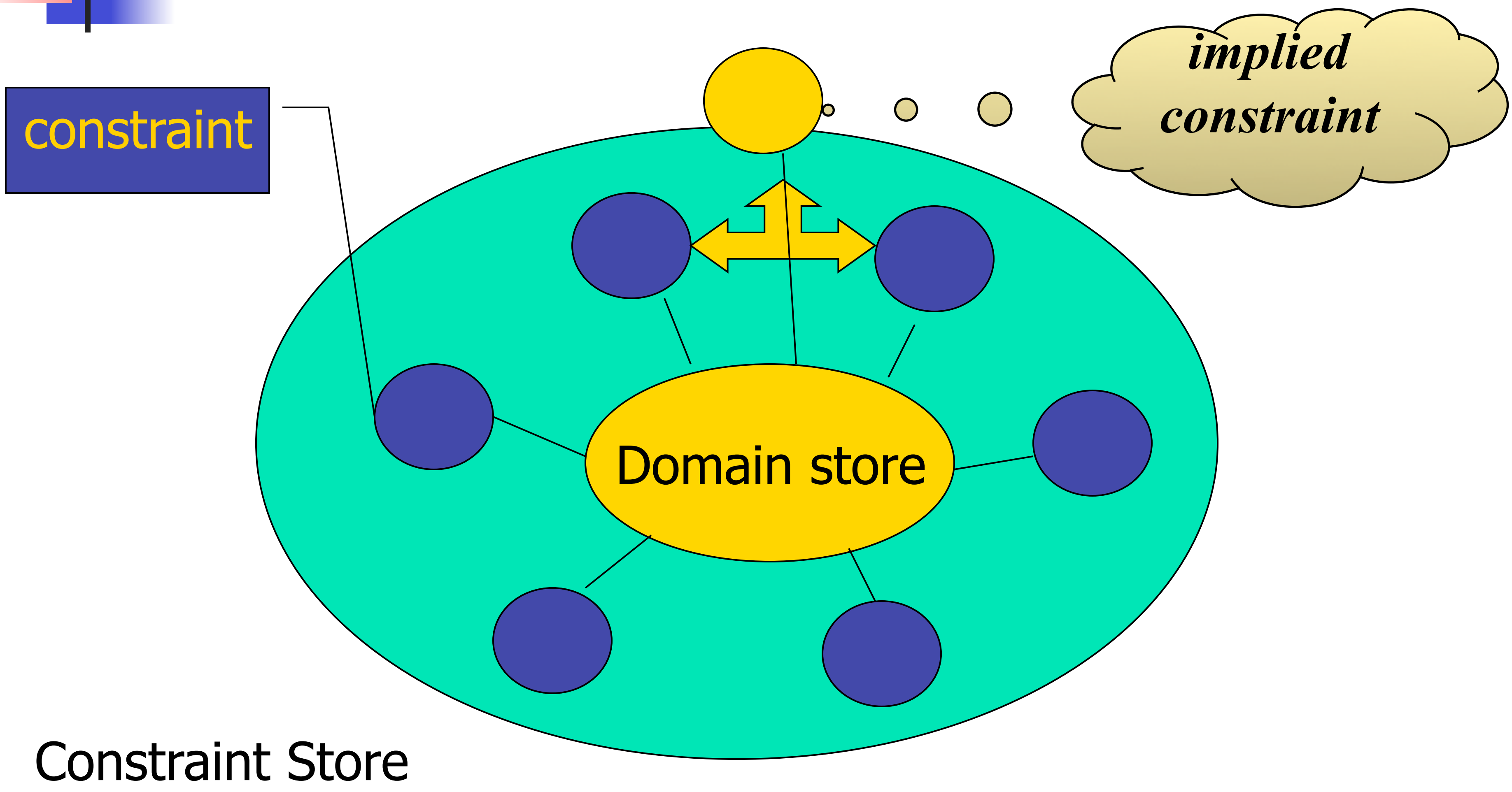
Improving Communication



Improving Communication



Improving Communication



Outline

- Auxiliary Variables
- Redundant Constraints
- Dual Modeling
- Symmetries

Dual Modeling

- Motivation
  - sometimes there are several possible models for the same problem
- Dual modeling
  - use several of them and link their variables

The Queens Problem

```
Solver<CP> cp();
int n = 8;
range R = 1..n;
range C = 1..n;
var<CP>{int} row[C](cp,R);
solve<cp> {
  cp.post(alldifferent(row));
  cp.post(alldifferent(all(k in R) row[k] + k));
  cp.post(alldifferent(all(k in R) row[k] - k));
}
```

The Queens Problem

```
var<CP>{int} row[C](cp,R);
var<CP>{int} col[R](cp,C);
solve<cp> {
   cp.post(alldifferent(row));
   cp.post(alldifferent(all(k in R) row[k] + k));
   cp.post(alldifferent(all(k in R) row[k] - k));

   cp.post(alldifferent(col));
   cp.post(alldifferent(all(k in R) col[k] + k));
   cp.post(alldifferent(all(k in R) col[k] - k));

  forall(r in R,c in C)
       cp.post((row[c] == r) == (col[r] == c));
}
```

Sport Scheduling

	Week 1	Week 2	Week 3	Week 4	Week 5	Week 6	Week 7
period 1	0 vs 1	0 vs 2	4 vs 6	3 vs 6	3 vs 7	1 vs 5	2 vs 4
period 2	2 vs 3	1 vs 7	0 vs 3	5 vs 7	1 vs 4	0 vs 6	5 vs 6
period 3	4 vs 5	3 vs 5	1 vs 6	0 vs 4	2 vs 6	2 vs 7	0 vs 7
period 4	6 vs 7	4 vs 6	2 vs 5	1 vs 2	0 vs 5	3 vs 4	1 vs 3

- There are n teams, n/2 periods, n-1 weeks
- Every team must play against every other team
  - Difficult to express
- A team plays exactly one game per period
- A team can play at most twice in the same period

Sport Scheduling

- Submitted by Bob Daniel to the MIP LIB.
- McAloon, Tretkoff, and Wetzel claim that state-of-the-art MIP packages cannot find a solution for n=14 (1997).
- The model to be described finds a solution in a couple of seconds (n = 14).

Sport Scheduling

- Team variables: a variable for each slot

	Week 1	Week 2	Week 3	Week 4	Week 5	Week 6	Week 7
period 1	- vs -	- vs -	- vs -	- vs -	- vs -	- vs -	- vs -
period 2	- vs -	- vs -	- vs -	- vs -	- vs -	- vs -	- vs -
period 3	- vs -	- vs -	- vs -	- vs -	- vs -	- vs -	- vs -
period 4	- vs -	- vs -	- vs -	- vs -	- vs -	- vs -	- vs

- Game variables: a variable for each game

	Week 1	Week 2	Week 3	Week 4	Week 5	Week 6	Week 7
period 1	-	-	-	-	-	-	-
period 2	-	-	-	-	-	-	-
period 3	-	-	-	-	-	-	-
period 4	-	-	-	-	-	-	-

Sport Scheduling

Add a dummy week

	Week 1	Week 2	Week 3	Week 4	Week 5	Week 6	Week 7	Week 8
period 1	- vs -	- vs -	- vs -	- vs -	- vs -	- vs -	- vs -	- vs -
period 2	- vs -	- vs -	- vs -	- vs -	- vs -	- vs -	- vs -	- vs -
period 3	- vs -	- vs -	- vs -	- vs -	- vs -	- vs -	- vs -	- vs -
period 4	- vs -	- vs -	- vs -	- vs -	- vs -	- vs -	- vs	- vs -

Constraints

- each team plays exactly twice in each period
- all the teams playing in week n are distinct

Sport Scheduling

- A team plays once a week
  - Take all teams in a week and state that they are all different (global constraint)
- A team plays exactly twice in a period
  - Take all the teams in a period and state a cardinality constraint (global constraint)
- Every team plays against all other teams
  - Take all the games and state that they are all different
- Link the games and the teams
  - Table constraint described by a set of possible couples

Sport Scheduling

```
int n = 14;
range Periods = 1..n/2;
range Teams   = 1..n;
range Weeks   = 1..n-1;
range EWeeks = 1..n;
enum Location = {home,away};
range Games = 1..(n/2)*n-1;
tuple triple {int a1; int a2; int a3; }
set{triple} Triples();
forall(i in 1..n,j in 1..n: i < j)
   Triples.insert(triple(i,j,(i-1)*n + j));
Table<CP> t(all(t in Triples) t.a1,
            all(t in Triples) t.a2,
            all(t in Triples) t.a3);
```

Sport Scheduling

```
var<CP>{int} team[Periods,EWeeks,Location](cp,Teams);
var<CP>{int} game[Periods,Weeks](cp,1..n^2);

solve<cp>{
  forall(w in EWeeks)
    cp.post(alldifferent(all(p in Periods,l in Location) team[p,w,l]),onDomains);
  forall(p in Periods)
    cp.post(exactly(all(i in Teams)2,all(w in EWeeks,l in Location) team[p,w,l]),
            onDomains);
  cp.post(alldifferent(all(p in Periods,w in Weeks) game[p,w]),onDomains);

  forall(p in Periods,w in Weeks)
    cp.post(table(team[p,w,home],team[p,w,away],game[p,w],t));
}
using labelFF(all(p in Periods,w in Weeks) game[p,w]);
```

Table Constraints