Constraint Programming

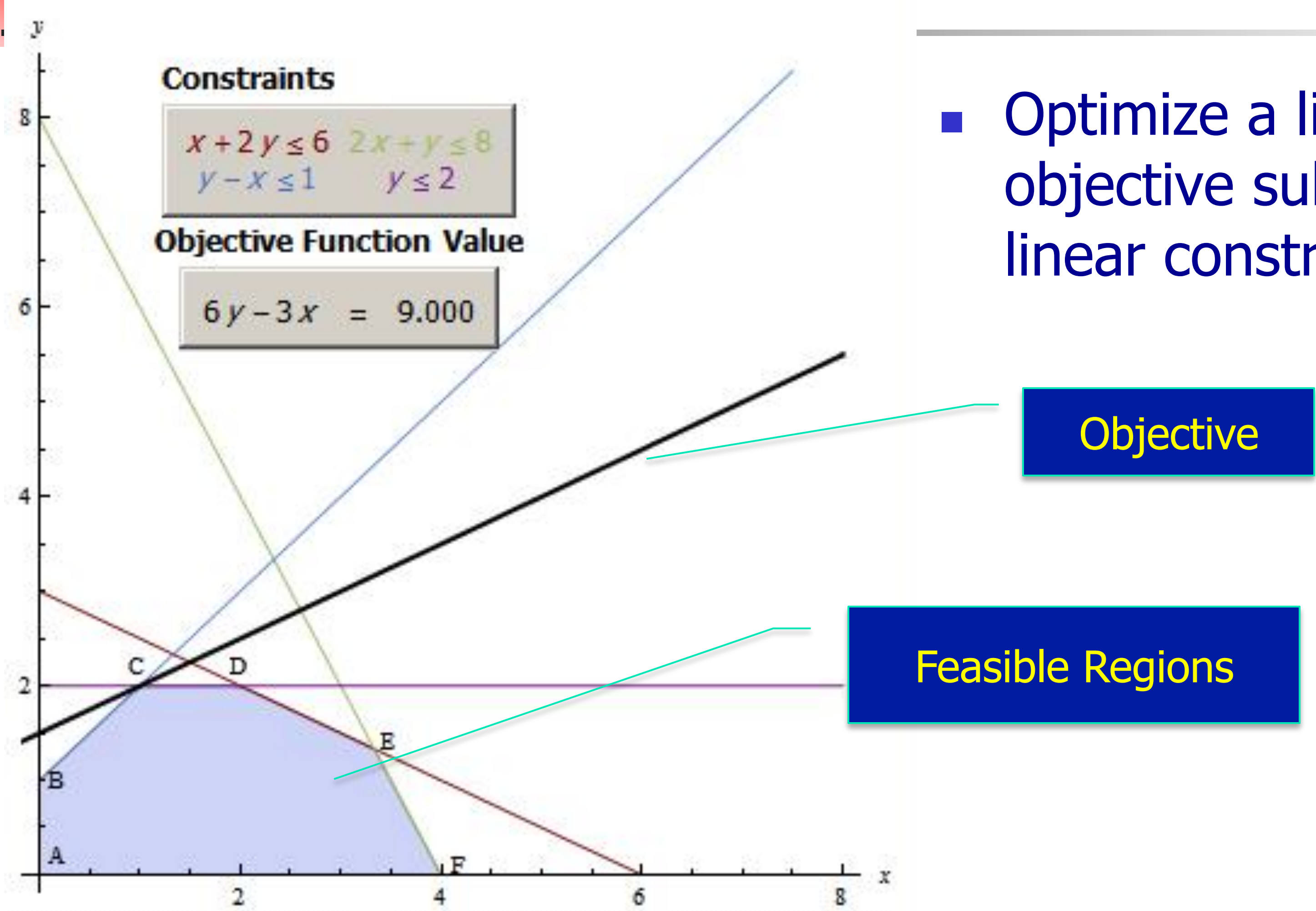
Pascal Van Hentenryck

Brown University

Constraint Programming

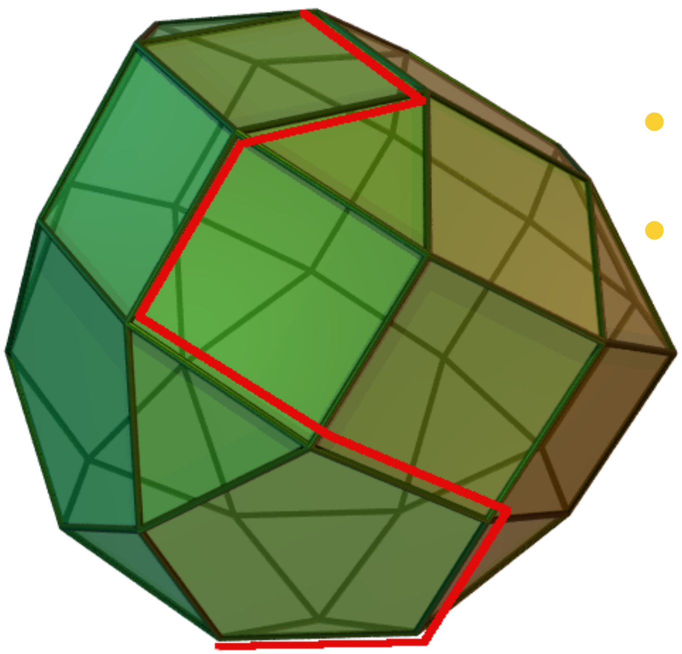
- Two main contributions
  - A new approach to combinatorial optimization
    - Orthogonal and complementary to standard OR methods
    - Combinatorial versus numerical
    - Feasibility versus optimality
  - A new language for combinatorial optimization
    - Rich language for constraints
    - Language for search procedures
    - Vertical extensions

Linear Programming



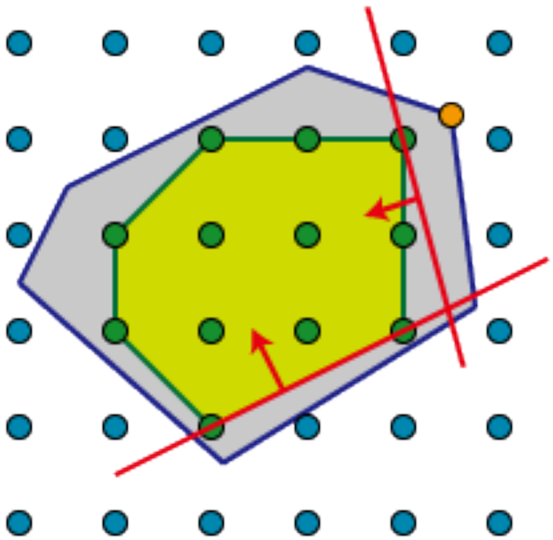
- Optimize a linear objective subject to linear constraints

Linear Programming



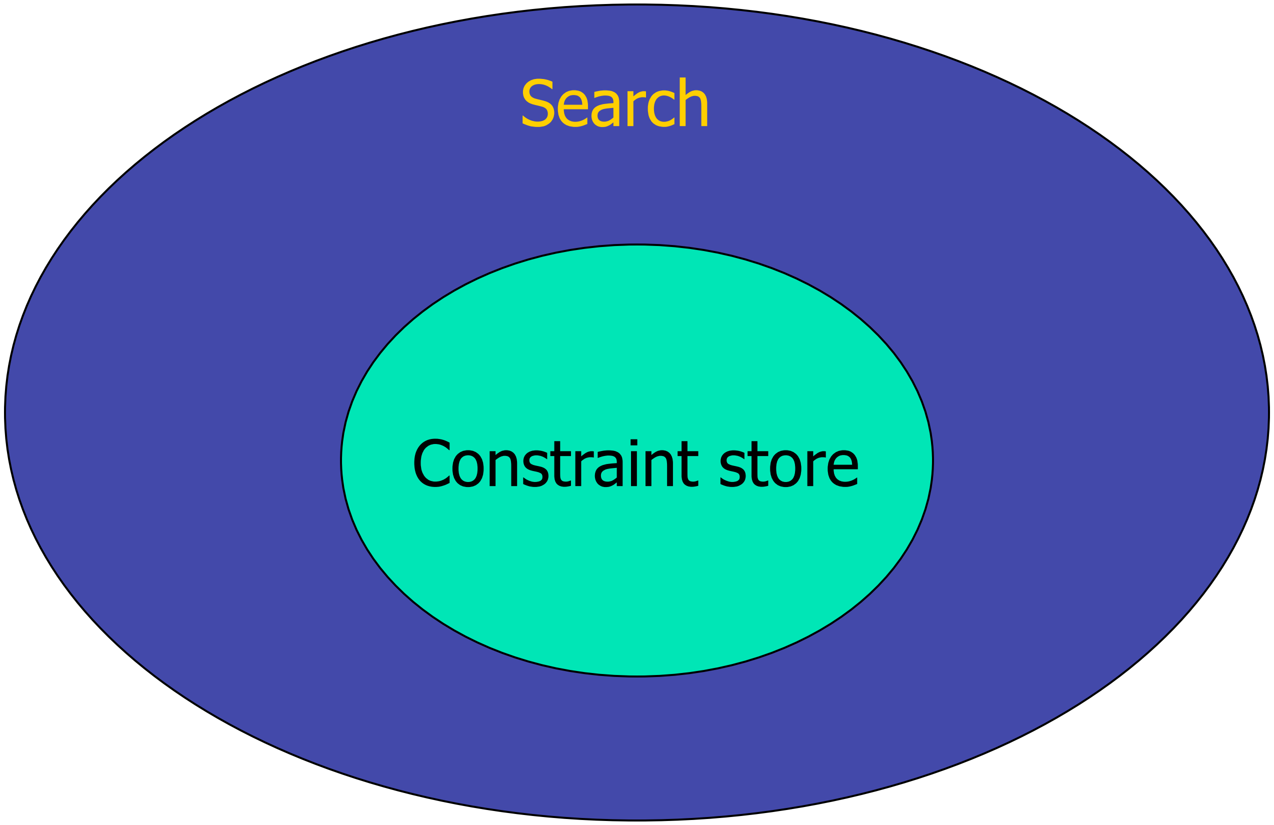
- Many dimensions
- Simplex algorithm
  - Moving from vertex to vertex
  - Geometry – Algebra links
  - Invented in the 40s in the US

Mixed Integer Programming



- Some variables must take integer values
  - From P to NP-complete
- Branch and cut and bound
  - Tighten the polyhedron
  - Branch when stuck
  - Use the LP to bound to prune
  - Tremendous progress in last 20 years

Computational Model



Computation Model

- Branch and Prune
  - Pruning: reduce the search the space as much as possible
  - Branching: when no pruning is available, decompose the problem into subproblems
- Fundamental novelty
  - How to prune
  - How to branch

Computational Model

- Iterate Branch and Prune Steps
  - Prune: eliminate infeasible configurations
  - Branch: decompose into subproblems
- Prune
  - Represent the search space explicitly: domains
  - Use constraints to reduce possible variable values
- Branch
  - Use heuristics based on feasibility information
- Main focus:constraints and feasibility

Coloring

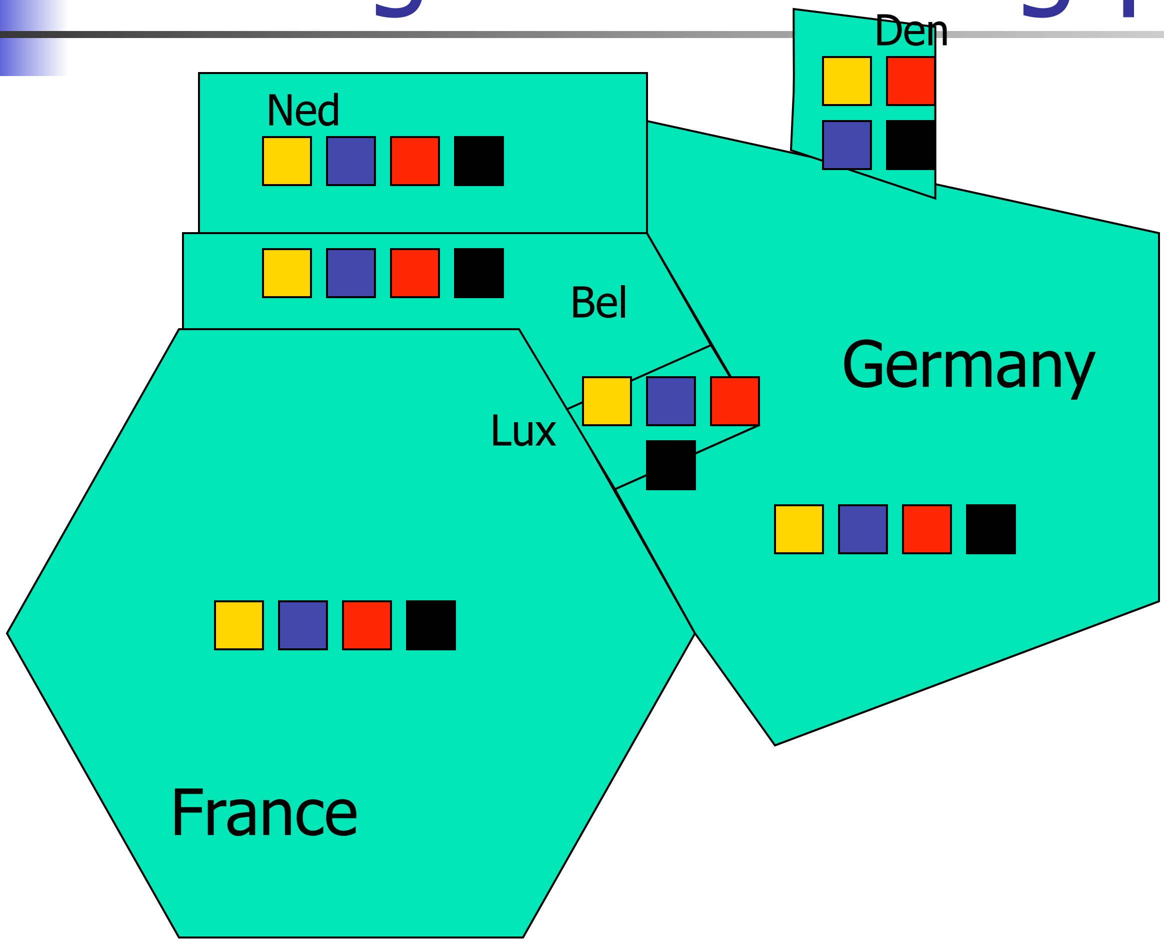
- Color a map of (part of) Europe: Belgium, Denmark, France, Germany, Netherlands, Luxembourg
- Use at most 4 colors

Coloring

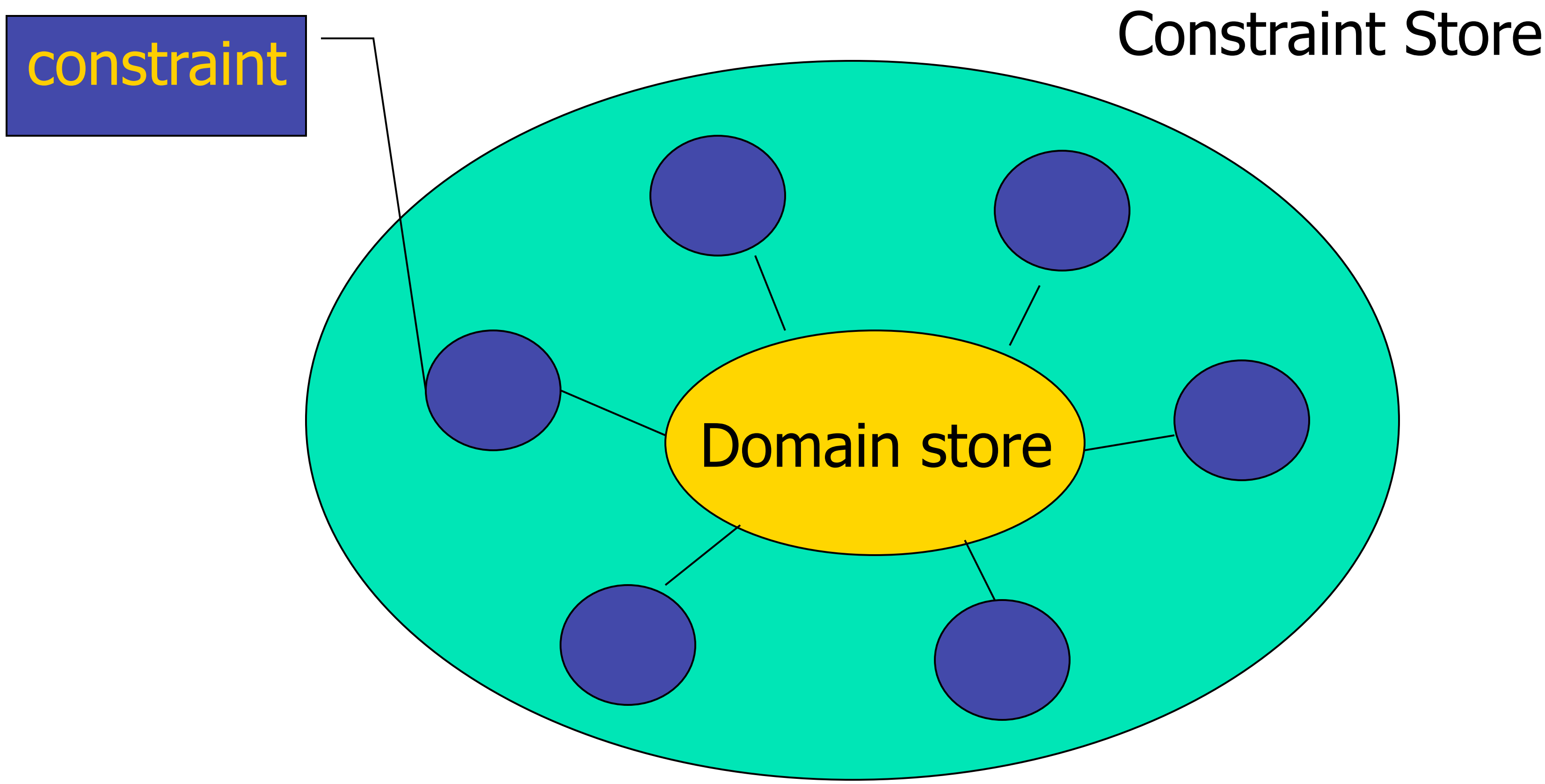
```
import cotfd;
Solver<CP> cp();
enum Countries =
   {Belgium,Denmark,France,Germany,Netherlands,Luxembourg};

var<CP>{int} color[Countries](cp,1..4);
solve<cp> {
   cp.post(color[France]     != color[Belgium]);
   cp.post(color[France]     != color[Luxembourg]);
   cp.post(color[France]     != color[Germany]);
   cp.post(color[Luxembourg] != color[Germany]);
   cp.post(color[Luxembourg] != color[Belgium]);
   …
}
```

Solving the coloring problem



Computational Model



Computational Model

What does a constraint do?

- Feasibility checking
  - can the constraint be satisfied given the domains of its variables
- Pruning
  - remove values from the domains if they do not appear in any solution of the constraint.

Constraint Solving

- General (fixpoint) algorithm is

```
repeat
  select a constraint c
  if c is infeasible wrt the domain store
     return infeasible
  else
     apply pruning algorithm of c
until no value can be removed
```

Constraints

- Specialized to each constraint type:

$$3x+10y+2z + 4w = 4$$

x in {0,1}, y in {0,1,2}, z in {0,1,2}, w in {0,1}

Simple bound reasoning (BC) gives

y in {0}

Domain reasoning (AC) gives

x in {0}, y in {0}, z in {0,2}, w in {0,1}

Constraints

- Domain Consistency
  - The holy grail
- A constraint is domain-consistent if, for every variable x and every value in the the domain of x, there exist values in the domains of the other variables such that the constraint is satisfied for these values

Constraint Programming

- Domain store
  - For each variable: what is the set of possible values?
  - If empty for any variable, then infeasible
  - If singleton for all variables, then solution
- Constraints
  - Capture interesting and well studied substructures
  - Need to
    - Determine if constraint is feasible wrt the domain store
    - Prune "impossible" values from the domains

Constraint Solving

- Constraint solving is declarative
  - every constraint is domain-consistent
  - the domains are as large as possible
  - greatest fixpoint
- Constraint solving algorithms
  - significant research subarea
  - many different algorithms
  - well understood at this point

Branching

- Once constraint solving is done, apply the search method

```
Choose a variable x with non-
  singleton domain (d1, d2, … di)
For each d in (d1, d2, … di)
  add constraint x=di to problem
```

Constraint Programming

- Two main contributions
  - A new approach to combinatorial optimization
    - Orthogonal and complementary to standard OR methods
    - Combinatorial versus numerical
    - Feasibility versus optimality
  - A new language for combinatorial optimization
    - Rich language for constraints
    - Language for search procedures
    - Vertical extensions

Central Theme

Combinatorial Application

=

Model + Search

- The model
  - what the solutions are and their quality
- The search
  - how to find the solutions

Central Theme

- The model
  - represents the combinatorial structure of the problem as explicitly as possible

Central Theme

Combinatorial Application

=

Model + Search

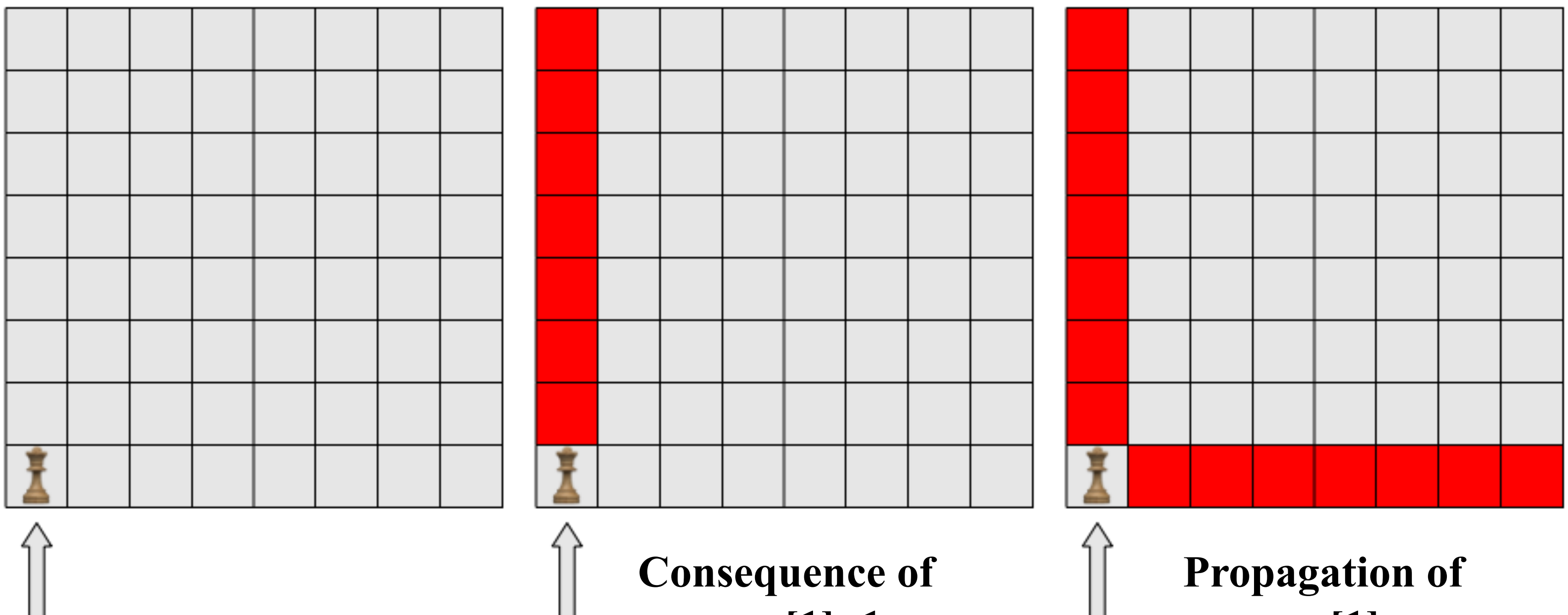
- The search
  - nondeterministic program + exploration strategy
  - exploration strategy: DFS, LDS, BFS, ...

The Queens Problem

```
int n = 8;
range R = 0..n-1;
var<CP>{int} queen[R](cp,R);
solve<cp> {
   forall(i in R,j in R: i<j) {
      cp.post(queen[i]      != queen[j]);
      cp.post(queen[i] + i != queen[j] + j);
      cp.post(queen[i] - i  != queen[j] - j);
   }
} using {
     forall(q in R)
        tryall<cp>(r in R)
             cp.post(queen[q] == r);
}
```

Nondeterminism

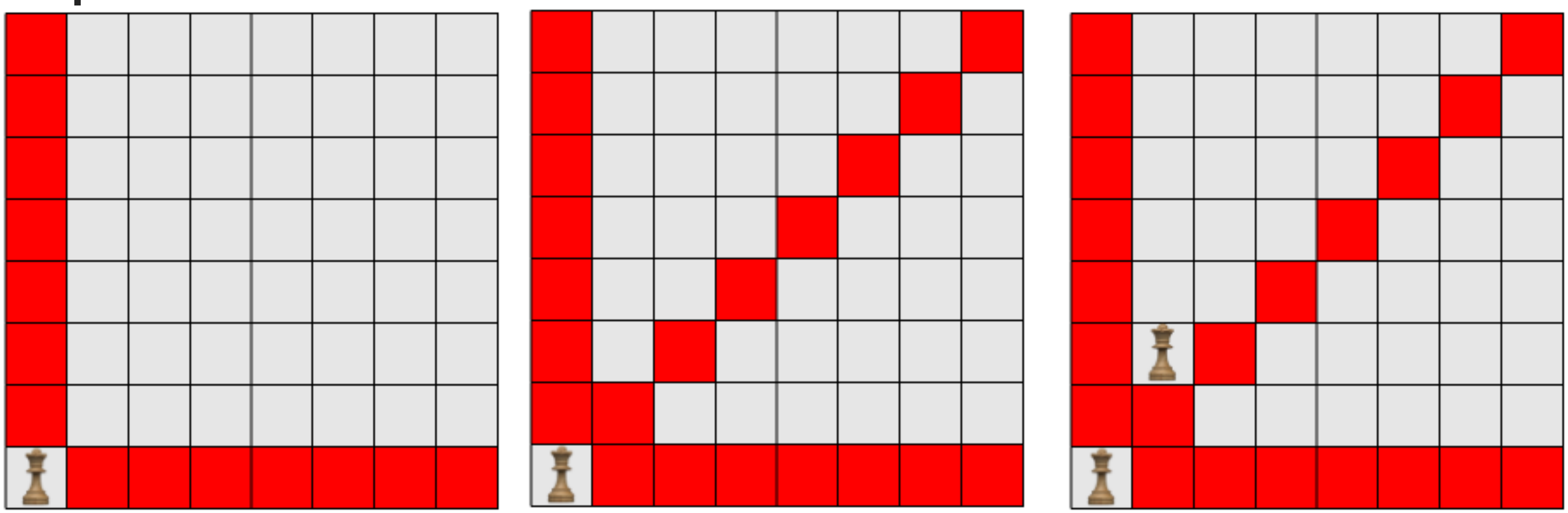
The Queens Problem



Consequence of
queen[1]=1

Propagation of
queen[1] <> queen[2]
queen[1] <> queen[3]
…
queen[1] <> queen[8]

The Queens Problem



Propagation of

queen[1]+1<> queen[2]+2

queen[1]+1<> queen[3]+3

…

queen[1]+1 <> queen[8]+8

No more inference

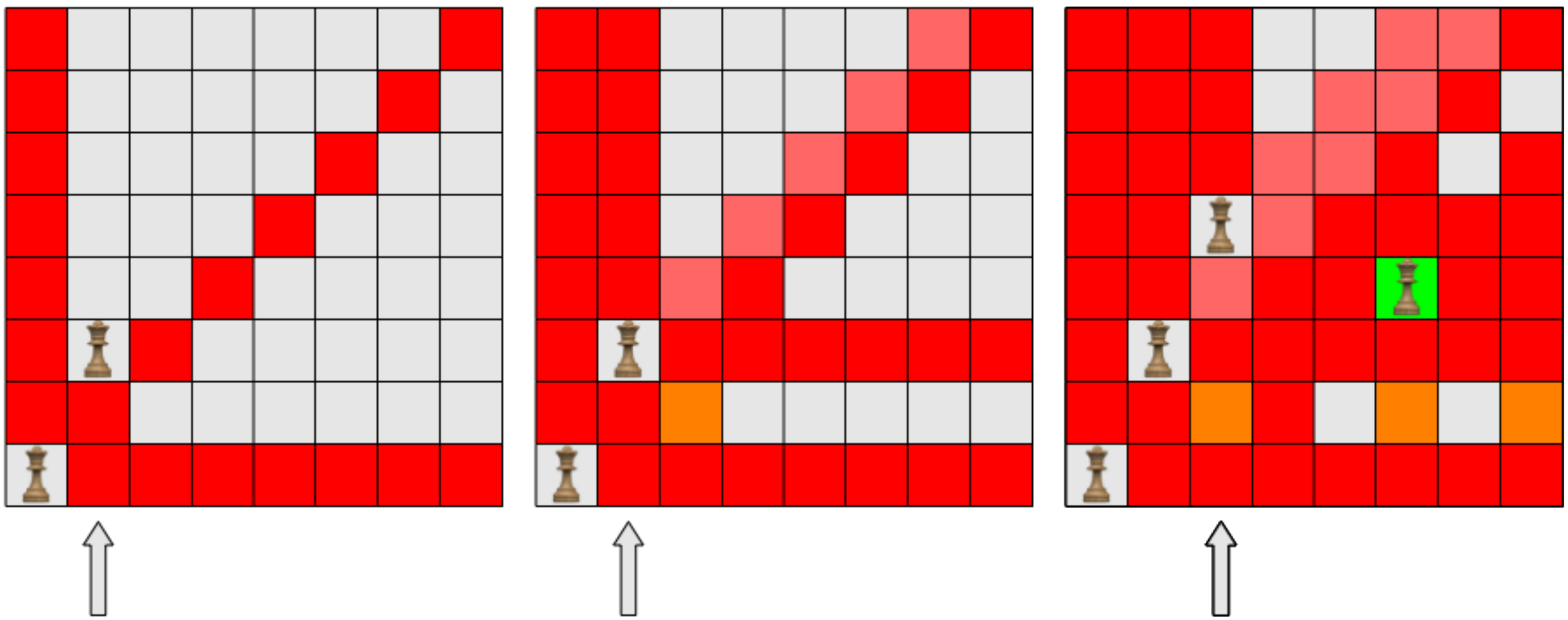
Place another queen

Questions

Which one ?

On which tile ?

The Queens Problem



Failure!
Go back to last choice
Try an alternative!

The Search

Search Procedure

=

Nondeterministic Program + Exploration Strategy

- Nondeterministic program
  - specify (implicitly) an and-or tree
- Exploration strategy
  - specify how to explore the tree

The CP Language

- A rich constraint language
  - Arithmetic, higher-order, logical constraints
  - Global constraints for natural substructures
- Specification of a search procedure
  - Definition of search tree to explore
  - Specification of exploration strategy
- Separation of concerns
  - Constraints and search are separated

Constraint Programming

- What is a constraint?
  - numerical inequalities and equations
  - combinatorial/global constraints
    - natural subproblems arising of many applications
    - a set of activities A cannot overlap in time
  - logical/threshold combinations of these
  - reification
  - …

Send More Money

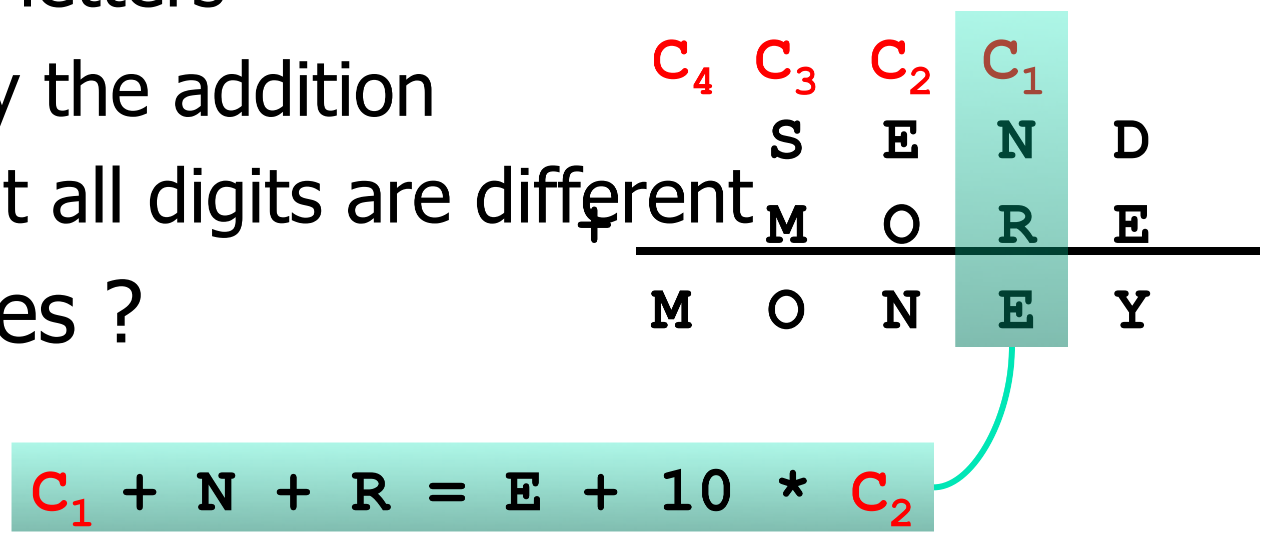
- Assign
  - Digits to letters
  - to satisfy the addition
  - and all digits are different
- Approaches ?
  - Direct
  - Carry

```
    S E N D
+   M O R E
-----------
  M O N E Y
```

```
              S*1000+E*100+N*10+D
+             M*1000+O*100+R*10+E
= M*10000+O*1000+N*100+E*10+Y
```

Send More Money

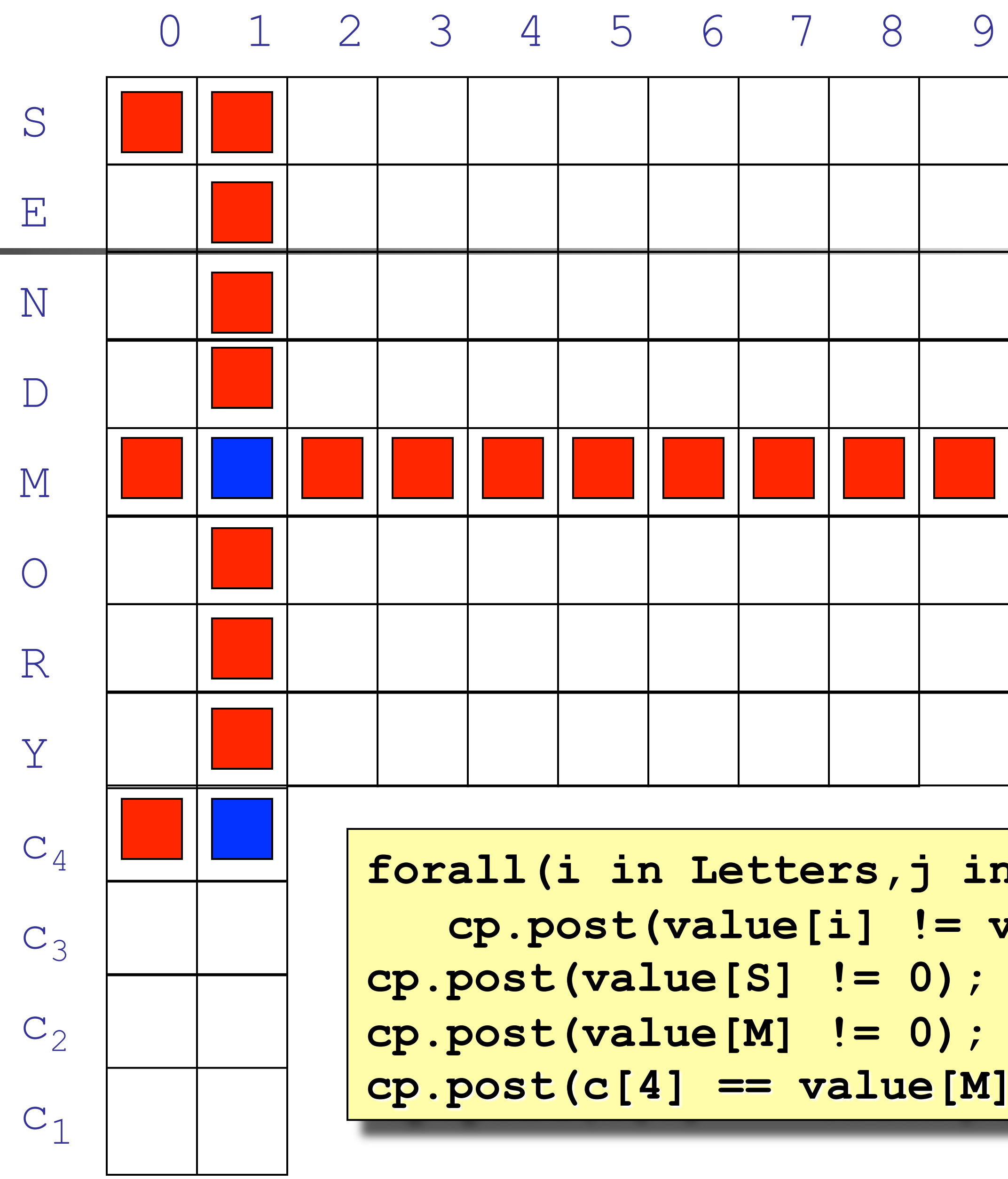
- Assign
  - Digits to letters
  - to satisfy the addition
  - such that all digits are different
- Approaches ?
  - Direct
  - Carry



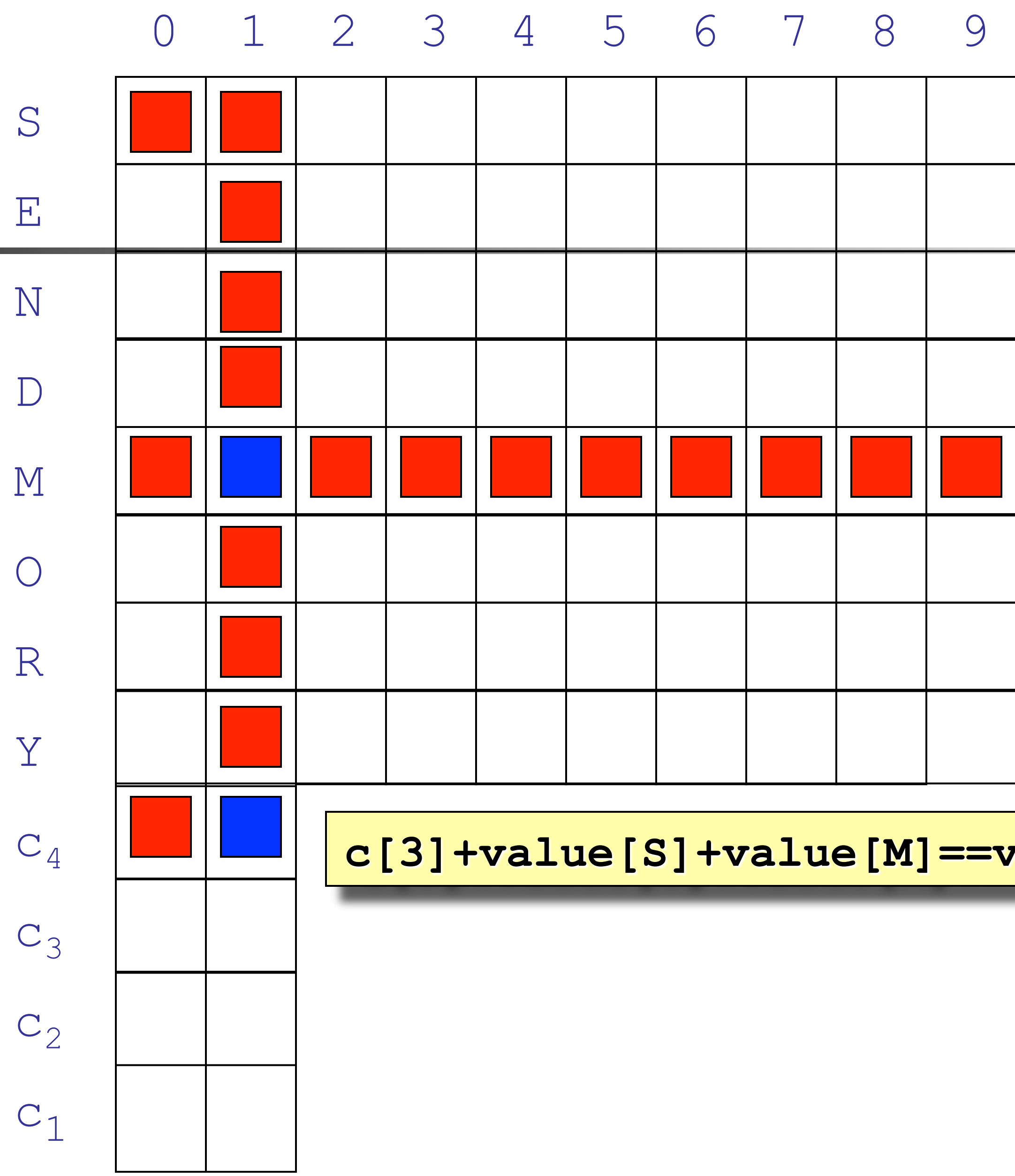
Send More Money [Carry]

```
enum Letters = {S,E,N,D,M,O,R,Y};
range Digits = 0..9;
range Bin    = 0..1;
var<CP>{int} value[Letters](cp,Digits);
var<CP>{int} c[1..4](cp,Bin)
solve<cp> {
   forall(i in Letters, j in Letters: i < j)
       cp.post(value[i] != value[j]);
   cp.post(value[S] != 0);
   cp.post(value[M] != 0);
   cp.post(c[4]                    == value[M]);
   cp.post(c[3]+value[S]+value[M] == value[O]+ 10  * c[4]);
   cp.post(c[2]+value[E]+value[O] == value[N] + 10 * c[3]);
   cp.post(c[1]+value[N]+value[R] == value[E] + 10 * c[2]);
   cp.post(     value[D]+value[E] == value[Y] + 10 * c[1]);
}
```

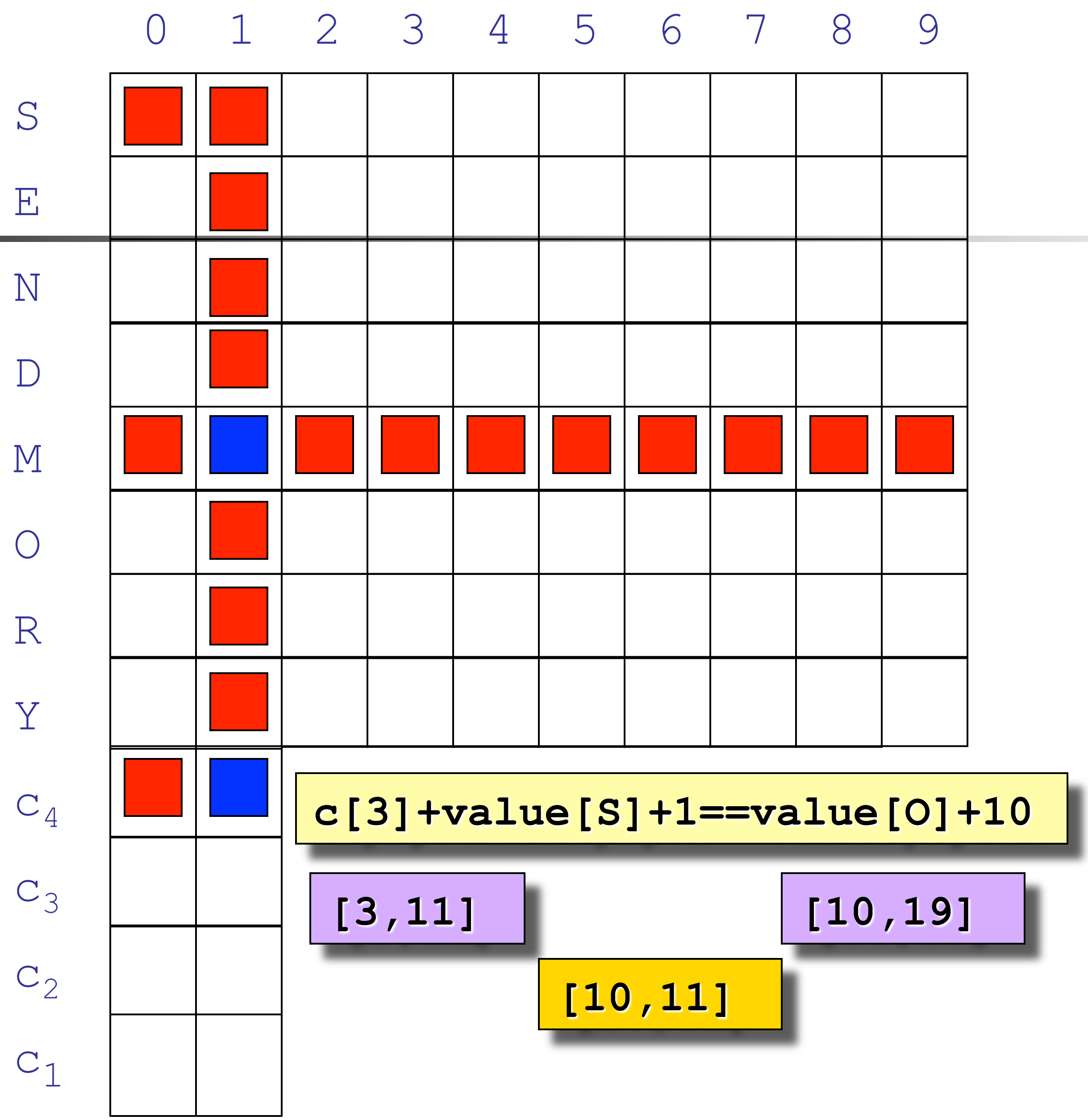
	0	1	2	3	4	5	6	7	8	9
S										
E										
N										
D										
M										
O										
R										
Y										
C_4										
C_3										
C_2										
C_1										

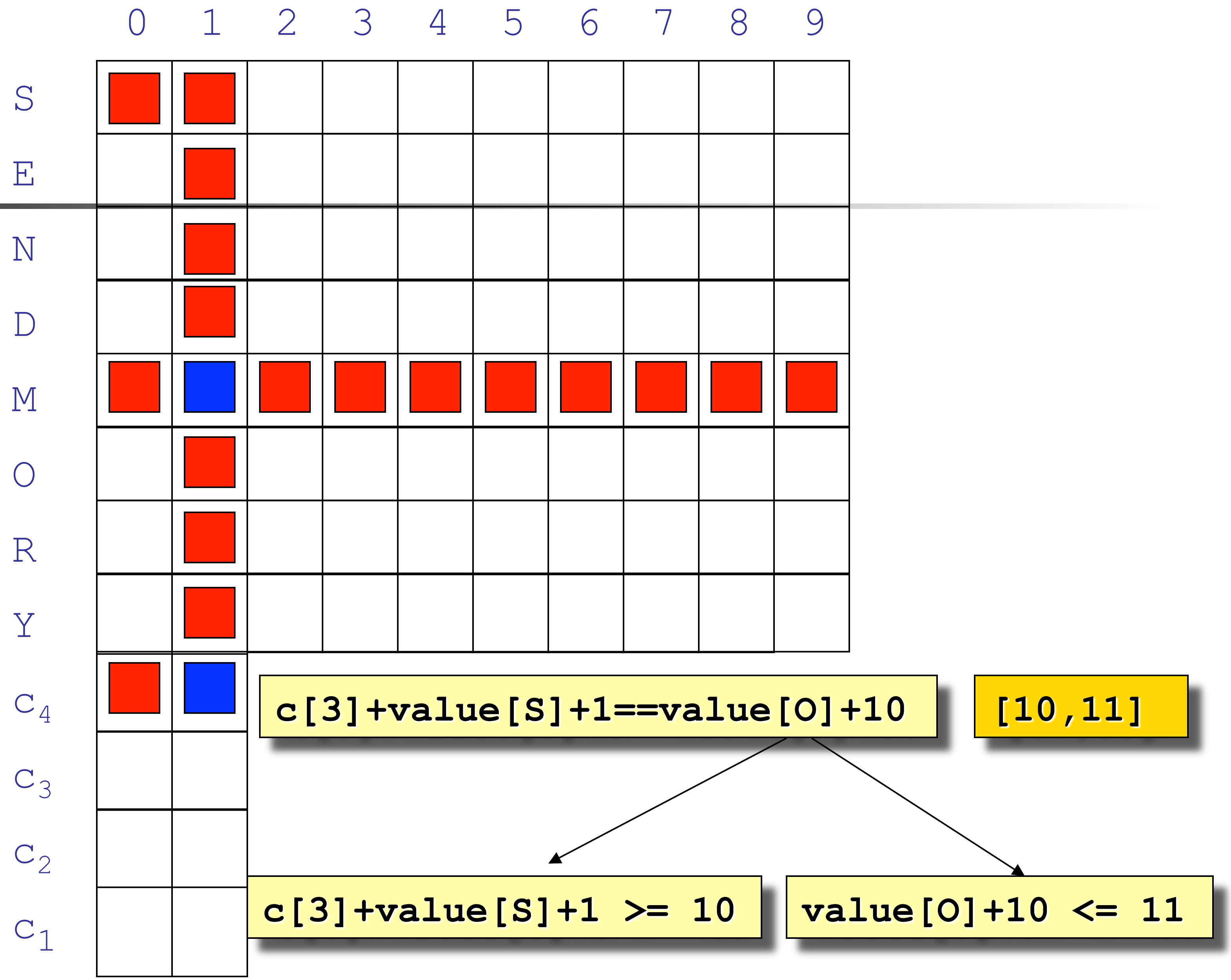


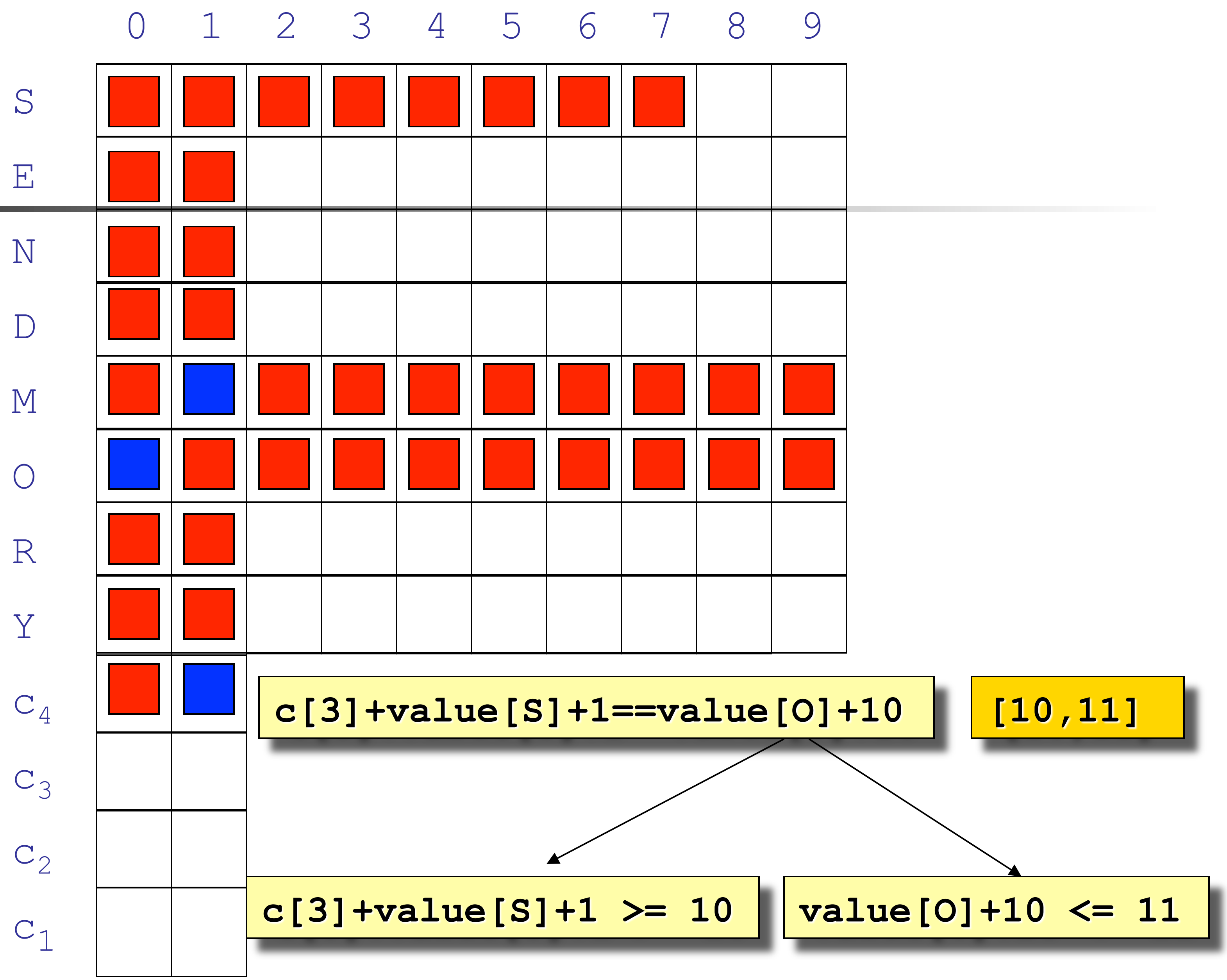

```
forall(i in Letters,j in Letters:i<j)
   cp.post(value[i] != value[j]);
cp.post(value[S] != 0);
cp.post(value[M] != 0);
cp.post(c[4] == value[M]);
```

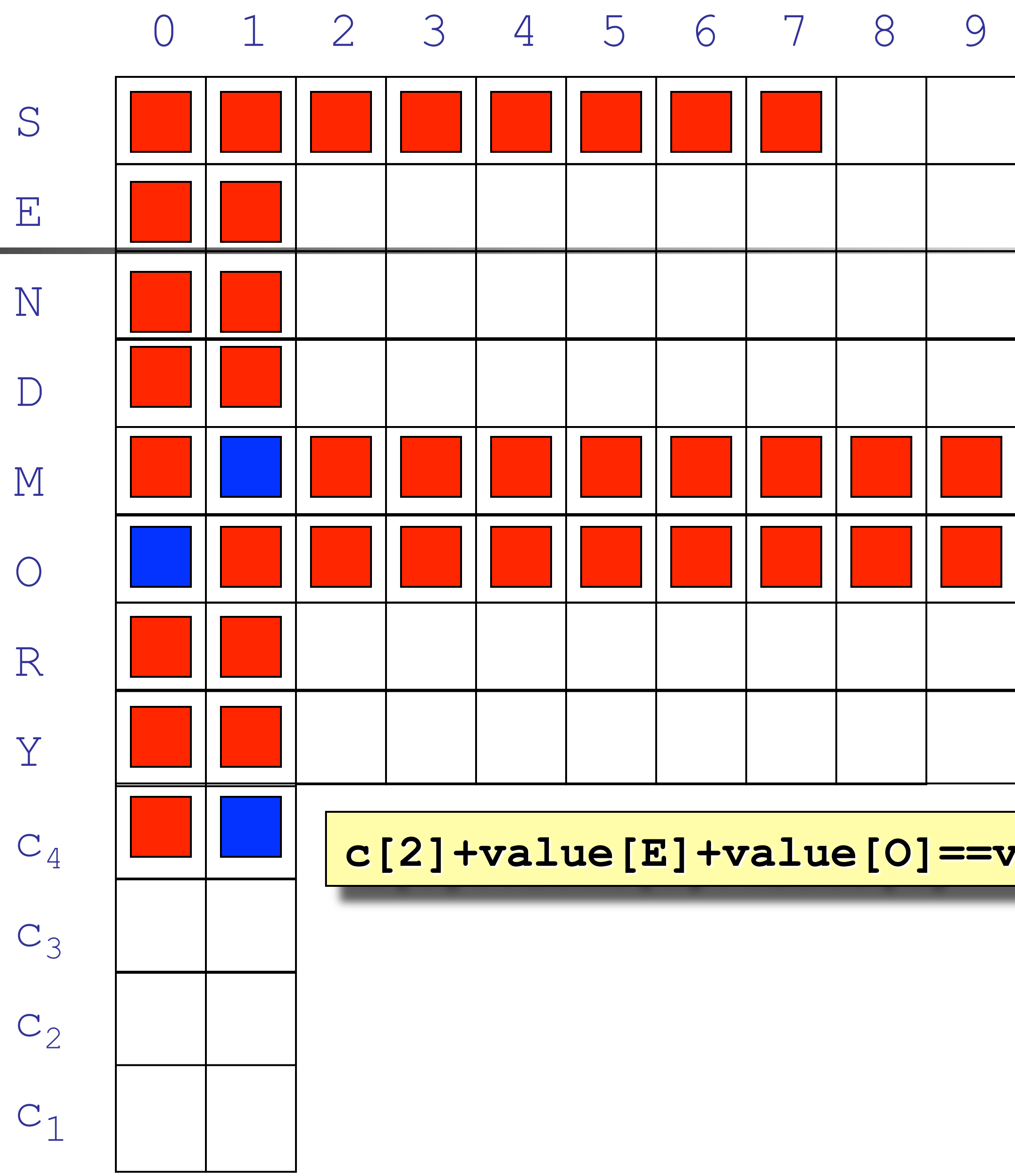


`c[3]+value[S]+value[M]==value[O]+10*c[4]`

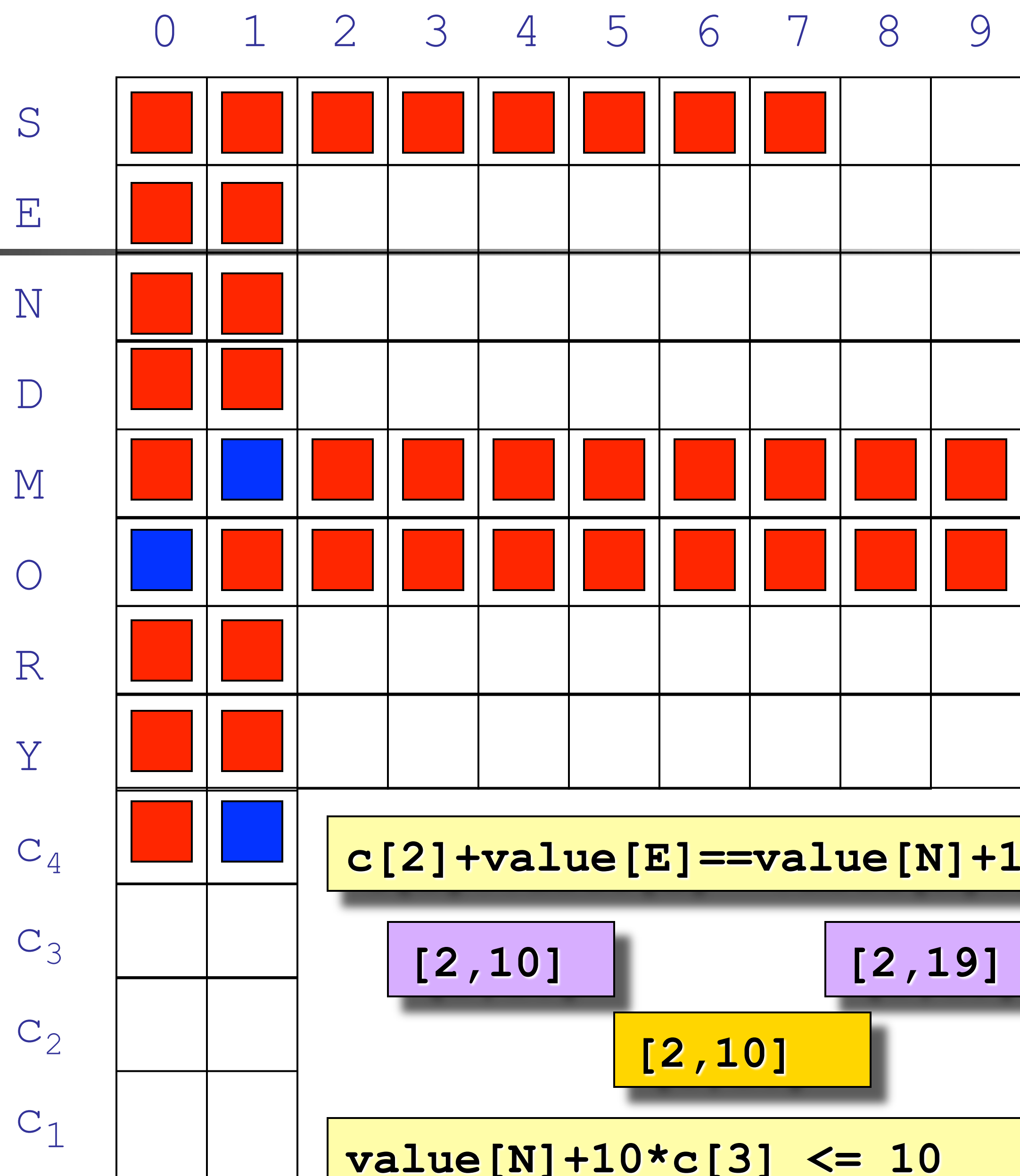

0 1 2 3 4 5 6 7 8 9
S
E
N
D
M
O
R
Y
c4
c3
c2
c1
c[3]+value[S]+1==value[O]+10
[3,11]
[10,19]
[10,11]


0 1 2 3 4 5 6 7 8 9
S
E
N
D
M
O
R
Y
C4
C3
C2
C1
c[3]+value[S]+1==value[O]+10
[10,11]
c[3]+value[S]+1 >= 10
value[O]+10 <= 11


0 1 2 3 4 5 6 7 8 9
S
E
N
D
M
O
R
Y
c4
c3
c2
c1
c[3]+value[S]+1==value[O]+10
[10,11]
c[3]+value[S]+1 >= 10
value[O]+10 <= 11




```
c[2]+value[E]+value[O]==value[N]+10*c[3]
```

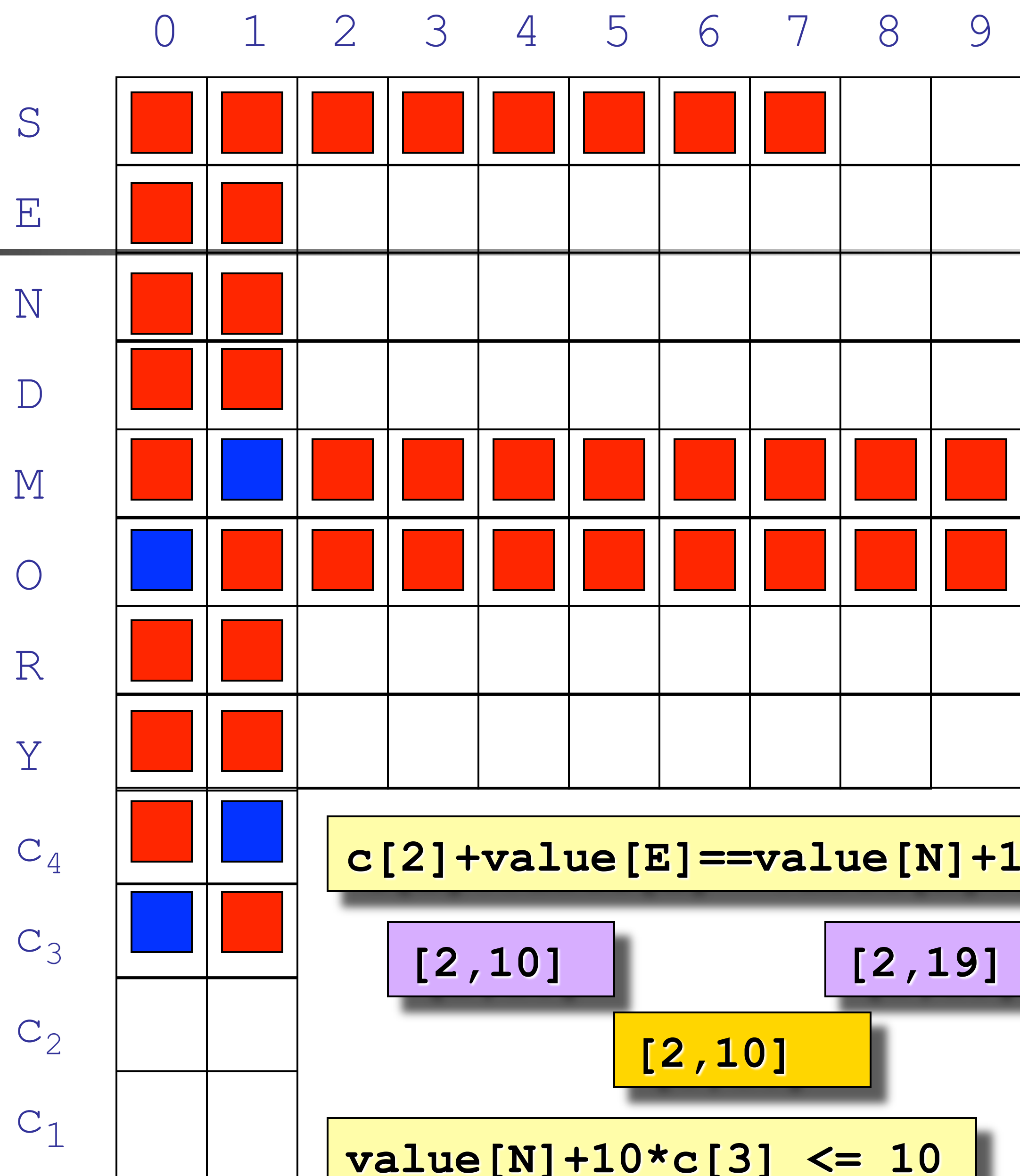


`c[2]+value[E]==value[N]+10*c[3]`

[2,10] [2,19]

[2,10]

`value[N]+10*c[3] <= 10`



c[2]+value[E]==value[N]+10*c[3]

[2,10]

[2,19]

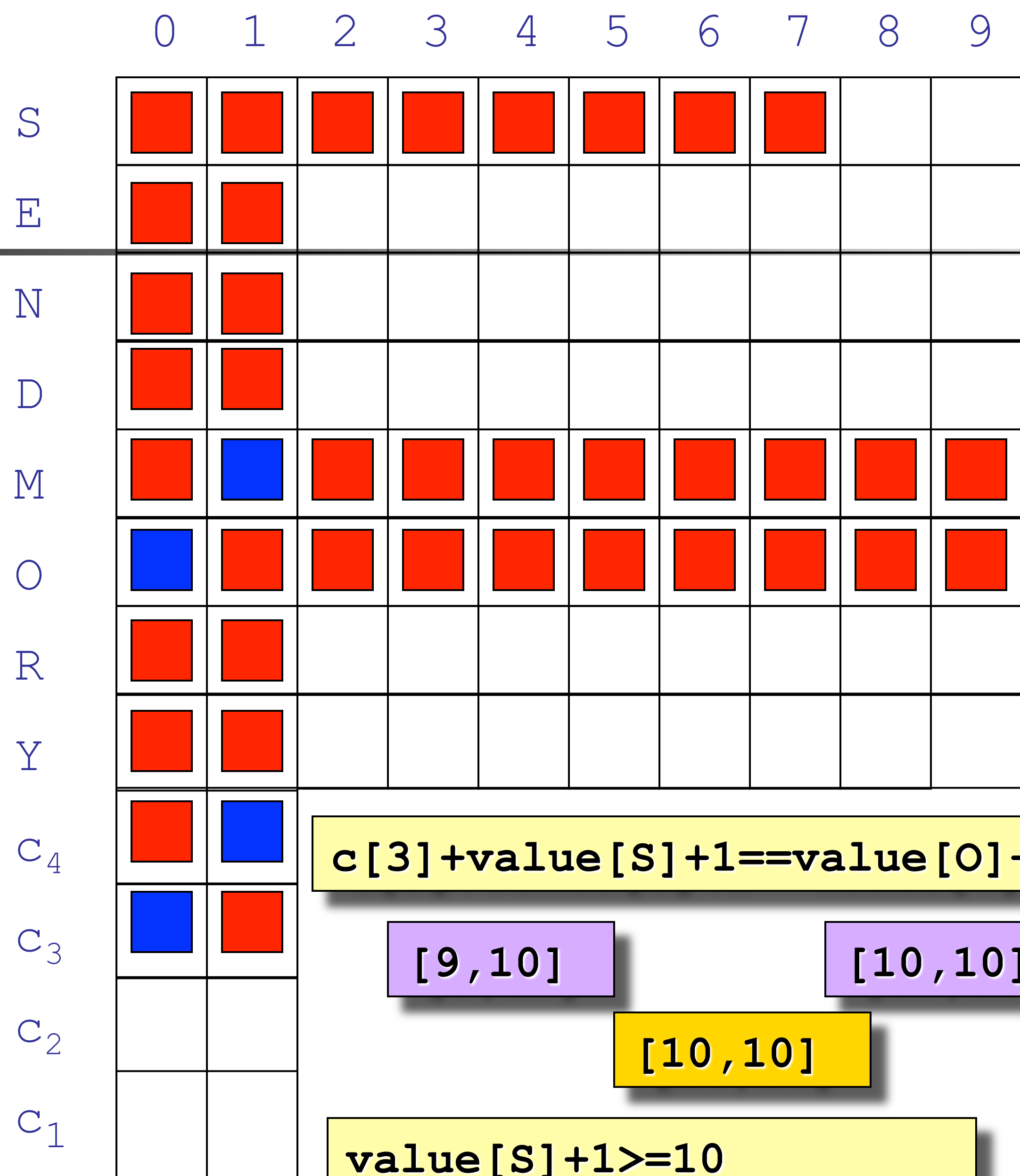
[2,10]

value[N]+10*c[3] <= 10

c[3] <= 0

Send More Money [Carry]

```
enum Letters = {S,E,N,D,M,O,R,Y};
range Digits = 0..9;
range Bin    = 0..1;
var<CP>{int} value[Letters](cp,Digits);
var<CP>{int} c[1..4](cp,Bin);
solve<cp> {
   forall(i in Letters, j in Letters: i < j)
       cp.post(value[i] != value[j]);
   cp.post(value[S] != 0);
   cp.post(value[M] != 0);
   cp.post(c[4]                    == value[M]);
   cp.post(c[3]+value[S]+value[M] == value[O]+ 10  * c[4]);
   cp.post(c[2]+value[E]+value[O] == value[N] + 10 * c[3]);
   cp.post(c[1]+value[N]+value[R] == value[E] + 10 * c[2]);
   cp.post(     value[D]+value[E] == value[Y] + 10 * c[1]);
};
```



c[3]+value[S]+1==value[O]+10

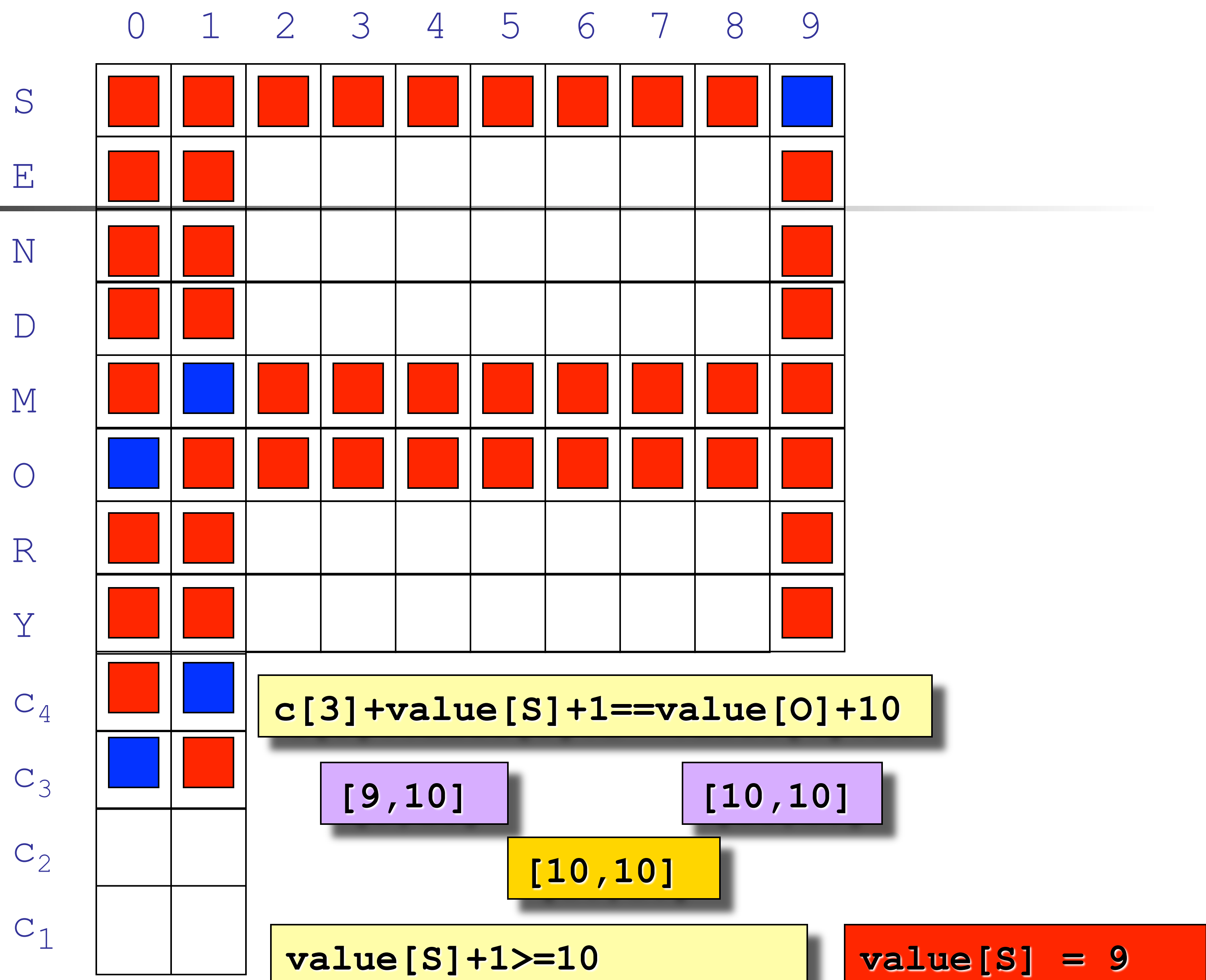
[9,10]

[10,10]

[10,10]

value[S]+1>=10

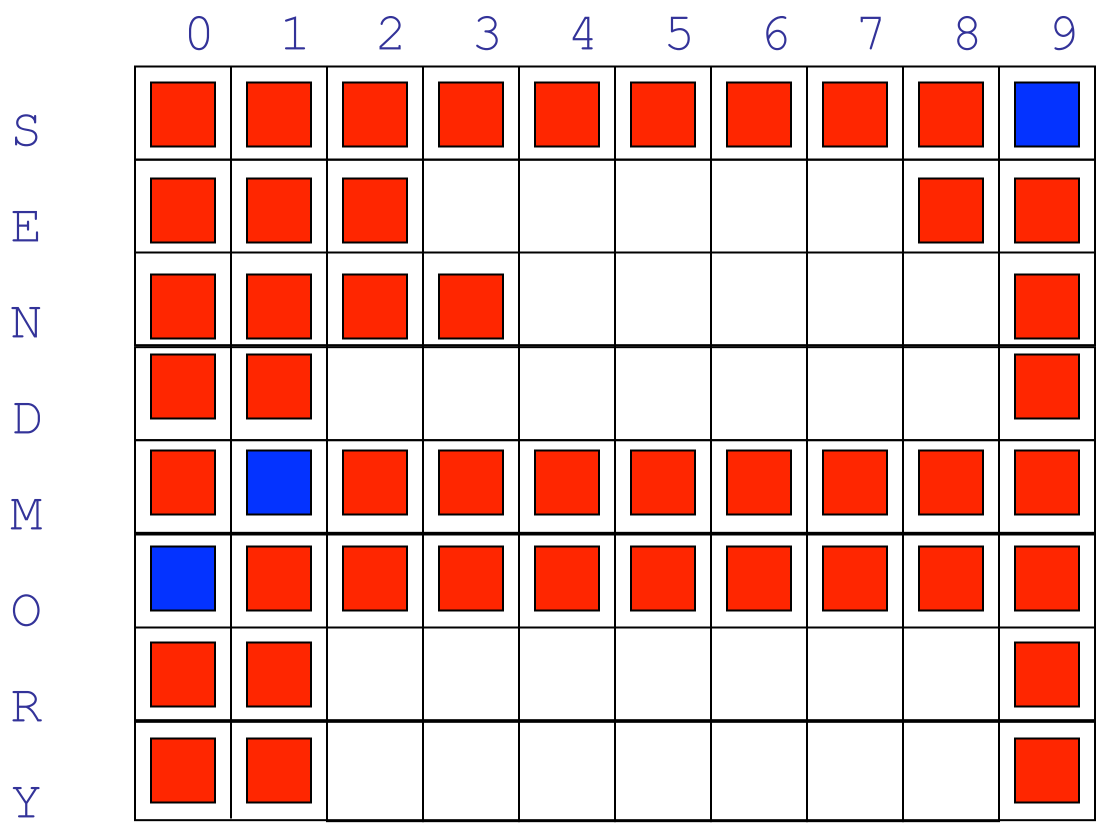
value[S] = 9


0 1 2 3 4 5 6 7 8 9
S
E
N
D
M
O
R
Y
c_4
c_3
c_2
c_1
c[3]+value[S]+1==value[O]+10
[9,10]
[10,10]
[10,10]
value[S]+1>=10
value[S] = 9

Send More Money [Direct]

```
enum Letters = {S,E,N,D,M,O,R,Y};
range Digits = 0..9;
var<CP>{int} value[Letters](cp,Digits);
explore<cp> {
   forall(i in Letters, j in Letters: i < j)
       cp.post(value[i] != value[j]);
   cp.post(value[S] != 0);
   cp.post(value[M] != 0);
   cp.post(value[S]*1000+value[E]*100+value[N]*10+value[D]+
           value[M]*1000+value[O]*100+value[R]*10+value[E]==
           value[M]*10000+value[O]*1000+value[N]*100+
           value[E]*10+value[Y]);
}
```

Send More Money [Direct]



Pascal Van Hentenryck

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
?	?	?	?	?

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));
}
```

Reification

- Reification
  - Allow constraints inside constraints
  - Replace the constraint by a 0/1 variables representing the truth value of the constraint

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)
```

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)
```

- Assume s[0]=1

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

Magic Series

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

- Now s[1]>0

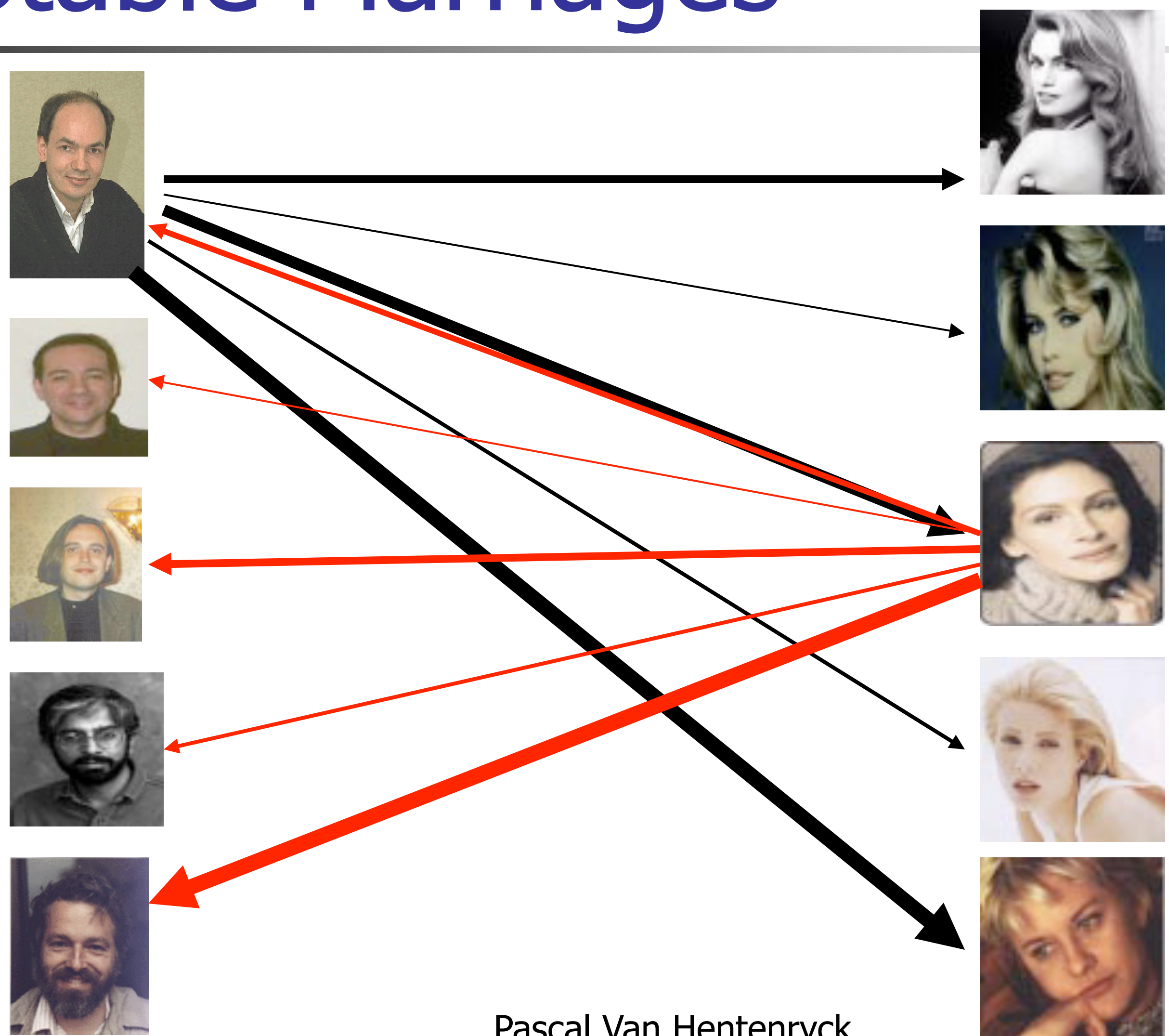
```
1     ==                       (s[2]==0)+(s[3]==0)+(s[4]==0)
s[1] == 1           +(s[1]==1)+(s[2]==1)+(s[3]==1)+(s[4]==1)
s[2] ==              (s[1]==2)+(s[2]==2)+(s[3]==2)+(s[4]==2)
s[3] ==              (s[1]==3)+(s[2]==3)+(s[3]==3)+(s[4]==3)
s[4] ==              (s[1]==4)+(s[2]==4)+(s[3]==4)+(s[4]==4)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

Reification

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

- Reification
  - reasons about constraint entailment
  - is a constraint always true or always false?

Stable Marriages



Stable Marriages

- A marriage is stable between James and Kathryn provided that
  - Whenever James prefers another woman, say Anne, to Kathryn, then Anne prefers her spouse to James;
  - Whenever Kathryn prefers another man, say Laurent, to James, then Laurent prefers his spouse to Kathryn.

Stable Marriages

```
enum Men = {Richard,James,John,Hugh,Greg};
enum Women = {Helen,Tracy,Linda,Sally,Wanda};

int preferm[Men,Women];
int preferw[Women,Men];

var<CP>{Women} wife[Men](cp,Women);
var<CP>{Men} husband[Women](cp,Men);

solveall<cp> {
      …
}
```

Stable Marriages

- Two types of constraints
  - The solution is a collection of marriages
    - If John is married to Jane, then Jane must be married to John
    - The husband of the wife of George is George
  - Stability rules

Stable Marriages

```
enum Men = {Richard,James,John,Hugh,Greg};
enum Women = {Helen,Tracy,Linda,Sally,Wanda};
int preferm[Men,Women] = …;
int preferw[Women,Men] = …;
var<CP>{Women} wife[Men](cp,Women);
var<CP>{Men} husband[Women](cp,Men);

explore<cp> {
   forall(i in Men)
      cp.post(husband[wife[i]] == i);
   forall(i in Women)
      cp.post(wife[husband[i]] == i);
   …
}
```

Element constraint

element

Stable Marriages

Implication

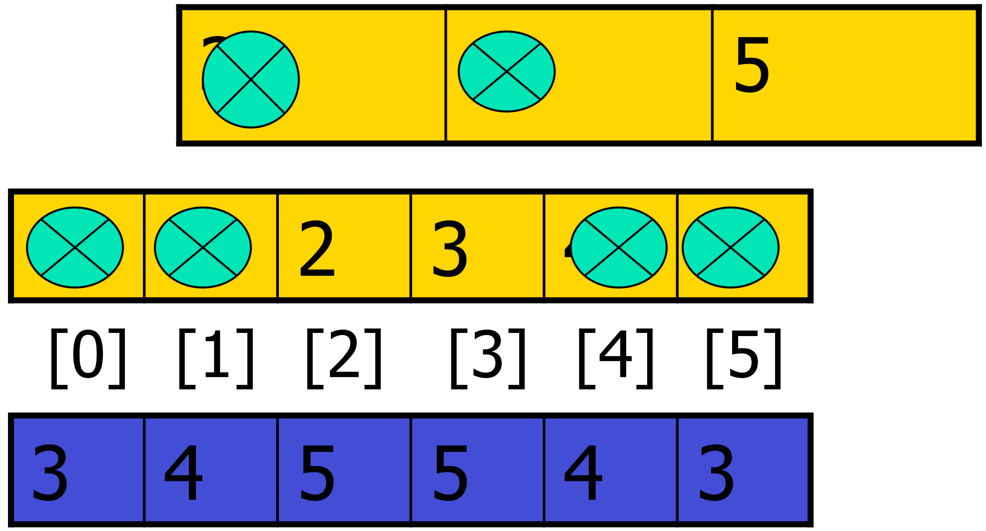
```
explore<cp> {
   forall(i in Men)
      cp.post(husband[wife[i]] == i);
   forall(i in Women)
      cp.post(wife[husband[i]] == i);

   forall(i in Men,j in Women)
     cp.post(preferm[i,j] > preferm[i,wife[i]] =>
             preferw[j,husband[j]] > preferw[j,i]);
   forall(i in Men,j in Women)
     cp.post(preferw[j,i] < preferw[j,husband[j]] =>
             preferm[i,wife[i]] < preferm[i,j]);
}
```

Stable Marriages

- Element constraints
  - ability to index an array/matrix with a decision variable or an expression;
- Logical constraints
  - ability to express any logical combination of constraint
  - see also reification

The Element Constraint



- X : variable
- Y : variable
- C : array
- Constraint: X = C[Y]
- X <> 3
- Y <> 1 & Y <> 4

The Element Constraint

- Facility location: want a constraint that customer c can be assigned to warehouse i only if warehouse open. (open[i]=1 if warehouse i is open)

IP: x[c,i] is 1 if customer c is assigned to i

x[c,i] <= open[w]

CP:w[c] is the warehouse customer c is assigned to (not a 0,1 variable)

open[w[c]] = 1;

Sudoku

5	3			7				
6			1	9	5			
	9	8					6	
8				6				3
4			8		3			1
7				2				6
	6					2	8	
			4	1	9			5
				8			7	9

Sudoku

combinatorial constraint

array comprehension

```
range R = 1..9;
var<CP>{int} S[R,R](cp,R);
solve<cp> {
   // constraints for fixed positions
   forall(i in R)
      cp.post(alldifferent(all(j in R) S[i,j]),onDomains);
   forall(j in R)
      cp.post(alldifferent(all(i in R) S[i,j]),onDomains);
   forall(i in 0..2,j in 0..2)
      cp.post(alldifferent(all(r in i*3+1..i*3+3,
                               c in j*3+1..j*3+3)S[r,c]),
              onDomains);
}
```

Global Constraints

- Recognize some combinatorial substructures arising in many practical applications
  - alldifferent is a fundamental building block
  - many others (as we will see)
- Make modeling easier and more natural
  - Declarative
  - Compositionality

The alldifferent Constraint

- Most well-known global constraint.

```
alldifferent(x,y,z)
```

states that x, y, and z take on different values. So x=2, y=1, z=3 would be ok, but not x=1, y=3, z=1.

- Very useful in many resource allocation, time tabling, sport scheduling problems

Sudoku

			1		2	9		
				9		3		1
					8			6
				3				
	6	2						
	7	9		1	6			
		8		6				7
		4				1	9	
					4		2	

Sudoku

			1		2	9		
				9		3		1
					8			6
				3				
	6	2						
	7	9		1	6			
		8		6				7
		4		2		1	9	
					4		2	

Sudoku

8	3	6	1		2	9		4
2	4		6	9		3	8	1
	9		3	4	8	2		6
	8			3			6	
	6	2					1	
	7	9		1	6		4	
9	2	8	5	6	1	4	3	7
6	5	4	7	2	3	1	9	8
7	1	3	9	8	4	6	2	5

Sudoku

8	3	6	1	5	2	9		4
2	4		6	9		3	8	1
	9		3	4	8	2		6
	8			3			6	
	6	2					1	
	7	9		1	6		4	
9	2	8	5	6	1	4	3	7
6	5	4	7	2	3	1	9	8
7	1	3	9	8	4	6	2	5

Pascal Van Hentenryck

Sudoku

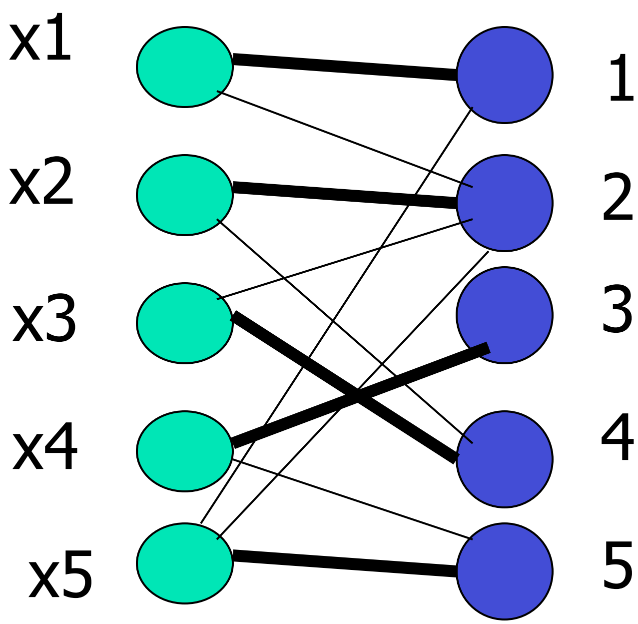
8	3	6	1	5	2	9	7	4
2	4	5	6	9		3	8	1
1	9	7	3	4	8	2	5	6
4	8	1		3		7	6	
	6	2		7			1	
	7	9		1	6		4	
9	2	8	5	6	1	4	3	7
6	5	4	7	2	3	1	9	8
7	1	3	9	8	4	6	2	5

Sudoku

8	3	6	1	5	2	9	7	4
2	4	5	6	9	7	3	8	1
1	9	7	3	4	8	2	5	6
4	8	1	2	3	5	7	6	9
5	6	2	4	7	9	8	1	3
3	7	9	8	1	6	5	4	2
9	2	8	5	6	1	4	3	7
6	5	4	7	2	3	1	9	8
7	1	3	9	8	4	6	2	5

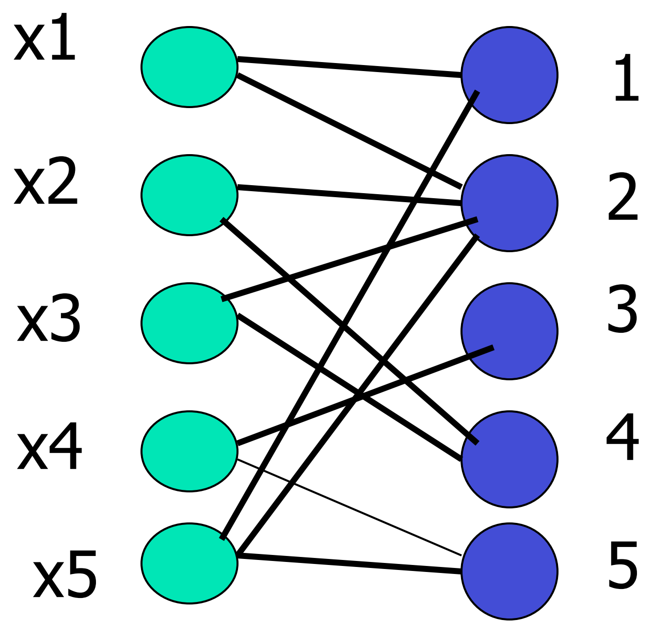
Alldifferent: Feasibility

- Feasibility? Given domains, create domain/variable bipartite graph



Alldifferent: Pruning

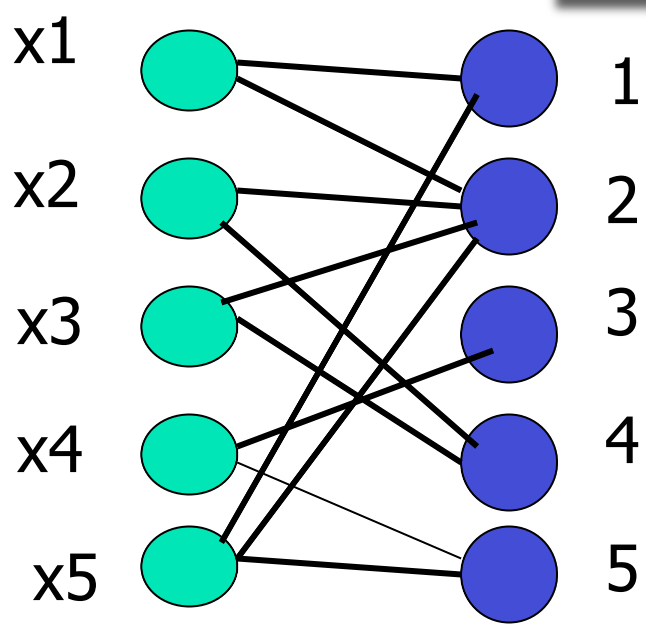
- Pruning? Which edges are in no solution?



Alldifferent: Pruning

- The benefits of globality

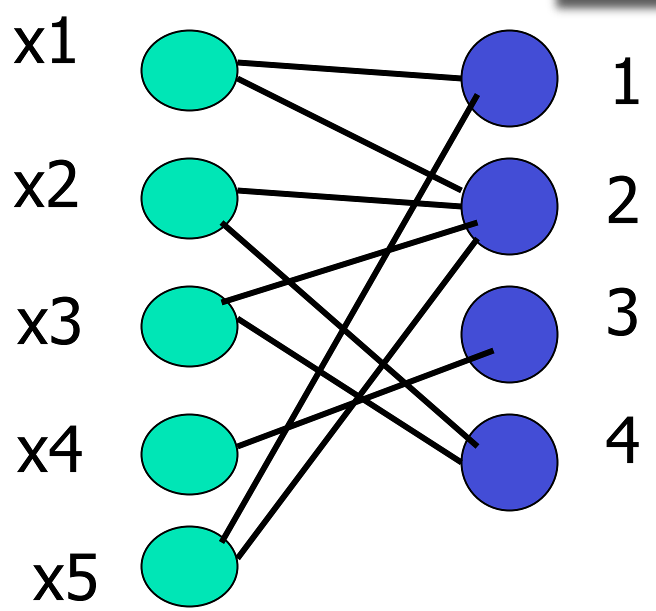
```
forall(i in 1..4, j in i..5)
         cp.post(x[i] != x[j]);
```



Alldifferent: Pruning

- The benefits of globality

```
forall(i in 1..4, j in i..5)
        cp.post(x[i] != x[j]);
```



Global Constraints

- Feasibility/Pruning algorithms
  - specialized to each type of constraints
- Many different algorithms
  - alldifferent: matching
  - cardinality constraints: flow
  - knapsack constraints: dynamic programming (DP)
  - one-machine scheduling: sorting + dominance
  - cumulative scheduling: DP + dominance
  - Linear constraint with objective: primal/dual simplex

Global Constraints: Summary

- Recognize some combinatorial substructures arising in many practical applications
- Make modeling easier and more natural
- Encapsulate strong pruning algorithms
  - Efficiency: exploit the substructure
- Declarative
  - details are hidden (how)
  - pruning is specified (what)
- Compositionality and extensibility

Table Constraints

(x,y,z) in

1	1	5
1	2	4
2	2	3

Domain store

x,y in {1,2},z in {3,4}

Table Constraints

(x,y,z) in

1	1	5
1	2	4
2	2	3

Feasibility

Domain store

x,y in {1,2},z in {3,4}

Table Constraints

(x,y,z) in

1	1	5
1	2	4
2	2	3

Pruning

⟶ $y \neq 1$

Domain store

x,y in {1,2},z in {3,4}

Constraint Programming

- Two main contributions
  - A new approach to combinatorial optimization
    - Orthogonal and complementary to standard OR methods
    - Combinatorial versus numerical
    - Feasibility versus optimality
  - A new language for combinatorial optimization
    - Rich language for constraints
    - Language for search procedures
    - Vertical extensions

Computational Model

- Iterate Branch and Prune Steps
  - Prune: eliminate infeasible configurations
  - Branch: decompose into subproblems
- Prune
  - Represent the search space explicitly: domains
  - Use constraints to reduce possible variable values
- Branch
  - Use heuristics based on feasibility information
- Main focus:constraints and feasibility

Euler Knight

- The problem
  - use a knight to visit all the positions in a chessboard exactly once.
- Abstraction
  - Travelling salesman problem
  - Vehicle routing (UPS, ...)

Euler Knight

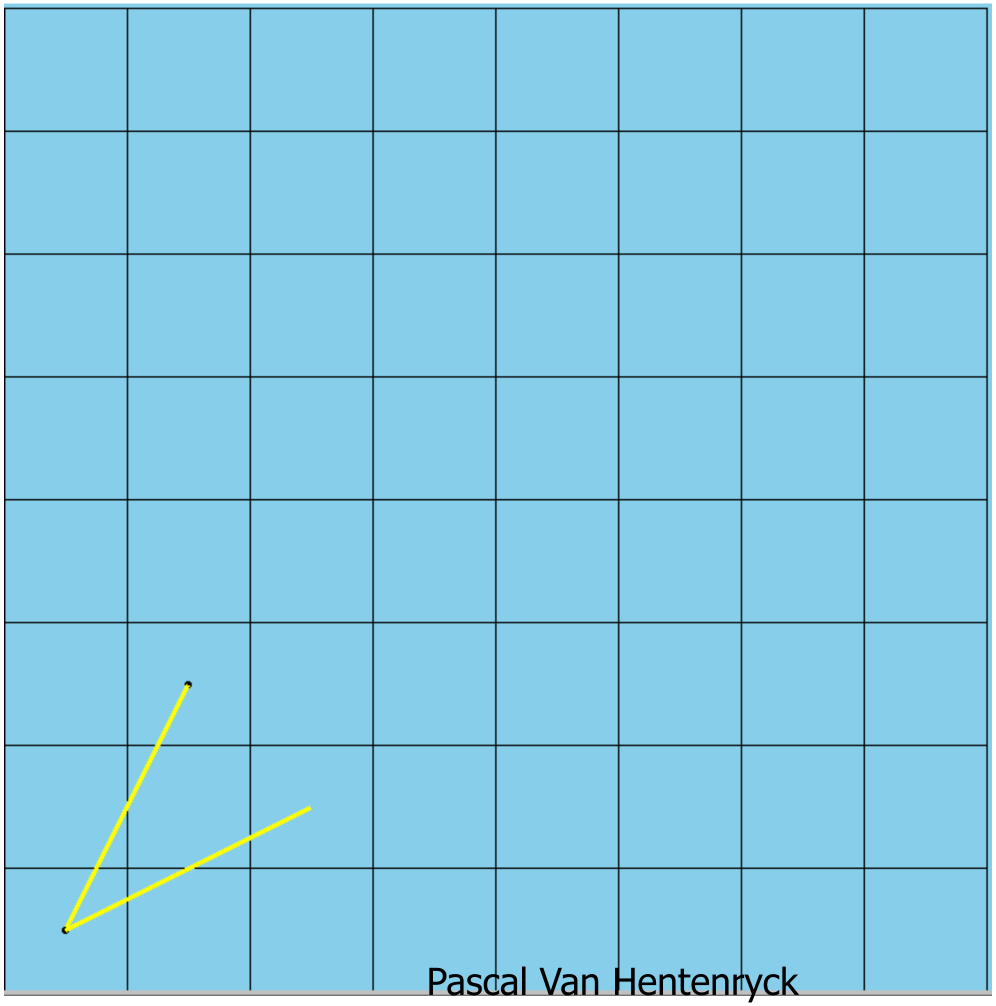
```
range Chessboard = 1..64;
var<CP>{int} jump[i in Chessboard](cp,Knightmoves(i));

solve<cp>
  cp.post(circuit(jump));
```

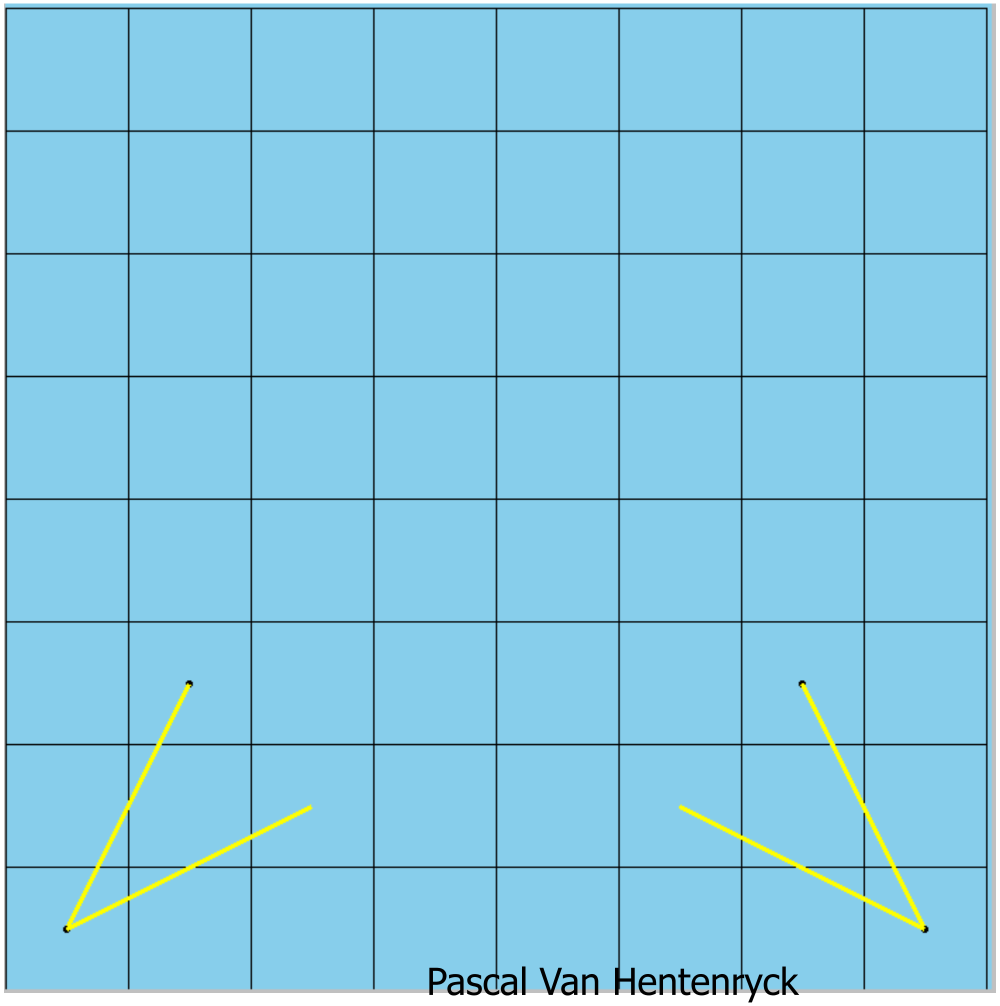
Euler Knight

```
function set{int} Knightmoves(int i) {
   set{int} S;
   if (i % 8 == 1)
      S = {i-15,i-6,i+10,i+17};
   else if (i % 8 == 2)
      S = {i-17,i-15,i-6,i+10,i+15,i+17};
   else if (i % 8 == 7)
      S = {i-17,i-15,i-10,i+6,i+15,i+17};
   else if (i % 8 == 0)
      S = {i-17,i-10,i+6,i+15};
   else
      S = {i-17,i-15,i-10,i-6,i+6,i+10,i+15,i+17};
   return filter(v in S) (v >= 1 && v <= 64);
}
```

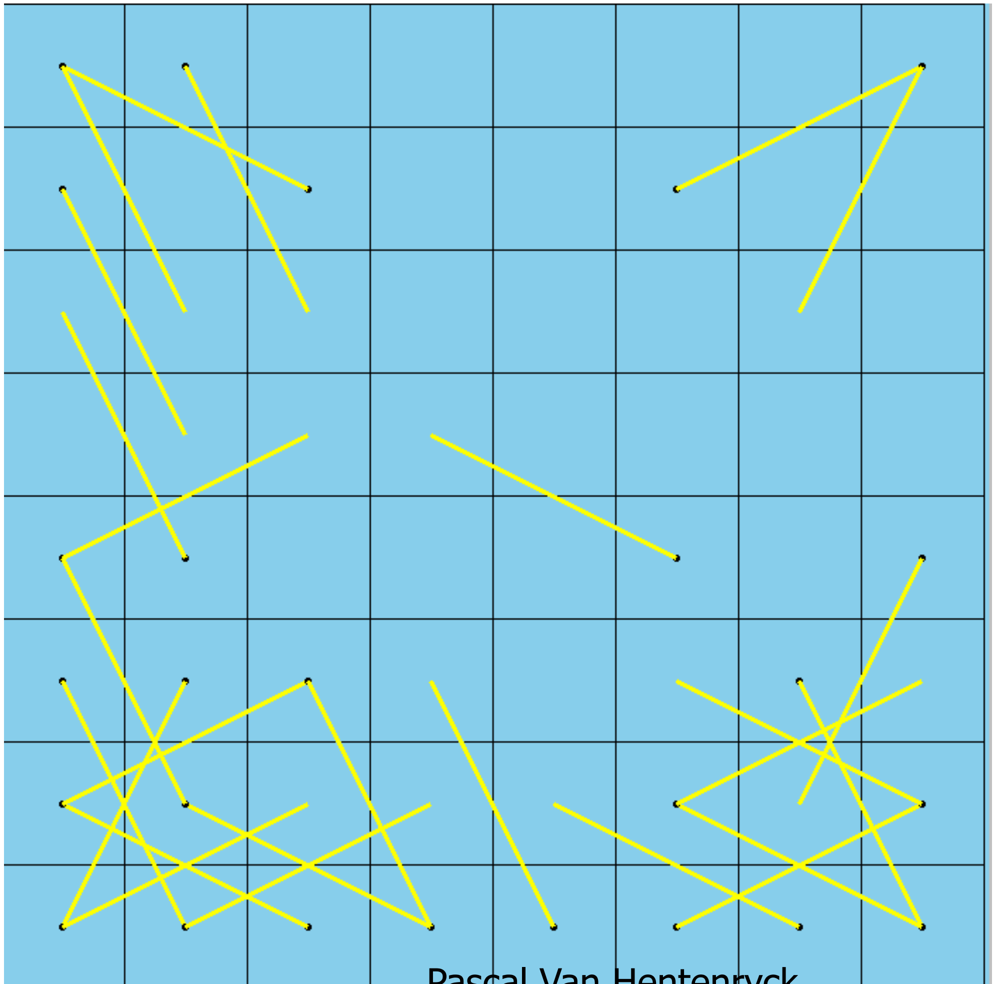
Euler Knight



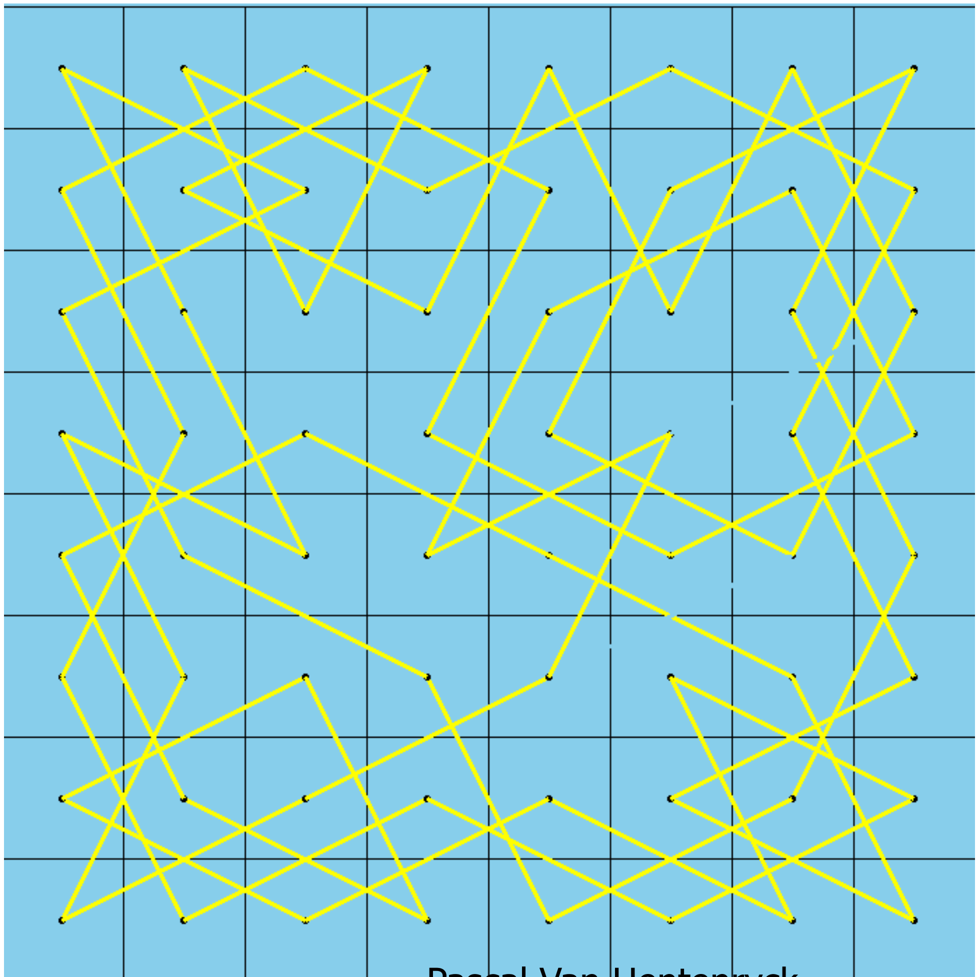
Euler Knight



Euler Knight



Euler Knight



Coloring

- Color a map of (part of) Europe: Belgium, Denmark, France, Germany, Netherlands, Luxembourg
- No two adjacent countries same color
- Minimize the number of colors

Coloring

objective function

```
enum Countries =
   {Belgium,Denmark,France,Germany,Netherlands,Luxembourg};

var<CP> color[Countries](cp,1..4);
minimize<cp>
   max(c in Countries) color[c]
subject to {
   cp.post(color[France]     != color[Belgium]);
   cp.post(color[France]     != color[Luxembourg]);
   cp.post(color[France]     != color[Germany]);
   cp.post(color[Luxembourg] != color[Germany]);
   cp.post(color[Luxembourg] != color[Belgium]);
   …
}
```

Finding optimal solutions

- Constraint programs can find optimal solutions. Typically works by finding a feasible solution and adding a constraint that future solutions must be better than it. Repeat until infeasible: the last solution found is optimal

Strengths of CP: Computational

- Capture combinatorial substructures directly in the language
  - Uses specialized algorithms to prune the search space for each of them
- Uses the pruning information to branch in an informed manner

Strength of CP: Language

Constraint Program

=

Model + Search

- The model
  - what the solutions are and their quality
- The search
  - how to find the solutions