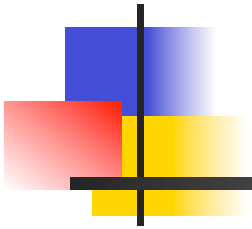
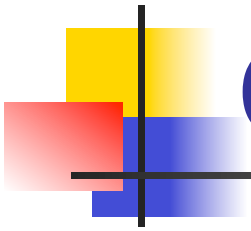


Search Techniques in Constraint Programming

Pascal Van Hentenryck

Brown University

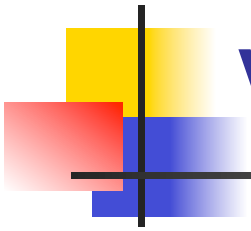


Outline

- Variable/Value Labeling
- Value/Variable Labeling
- Domain splitting
- Symmetry breaking during search
- Randomization and Restarts
- Large Neighborhood Search

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));
}
```



Variable/Value Labeling

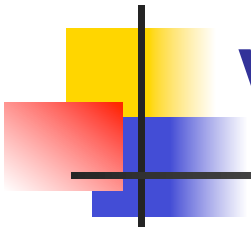
- Two steps
  - choose the variable to assign next
  - choose the value to assign
- Dynamic orderings
  - choose the next variable dynamically
  - choose the next value dynamically

The Queens Problem

select the rqueen with the smallest domain

```
using {

   forall(c in C) by (row[c].getSize())

      tryall<cp>(r in R) by (col[r].getSize())

         cp.post(row[c] == r);

}
```

select the cqueen with the smallest domain



Variable/Value Labeling

- Two steps
  - choose the variable to assign next
  - choose the value to assign
- Dynamic orderings
  - may use lexicographic criterion
- Example
  - start with the queens close to the middle

The Queens Problem

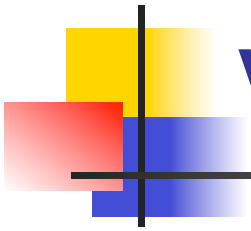
lexicographic ordering

```
using {

   forall(c in C) by (row[c].getSize(),abs(n/2-c))

      tryall<cp>(r in R) by (col[r].getSize())

         cp.post(row[c] == r);

}
```

The Queens Problem

lexicographic ordering

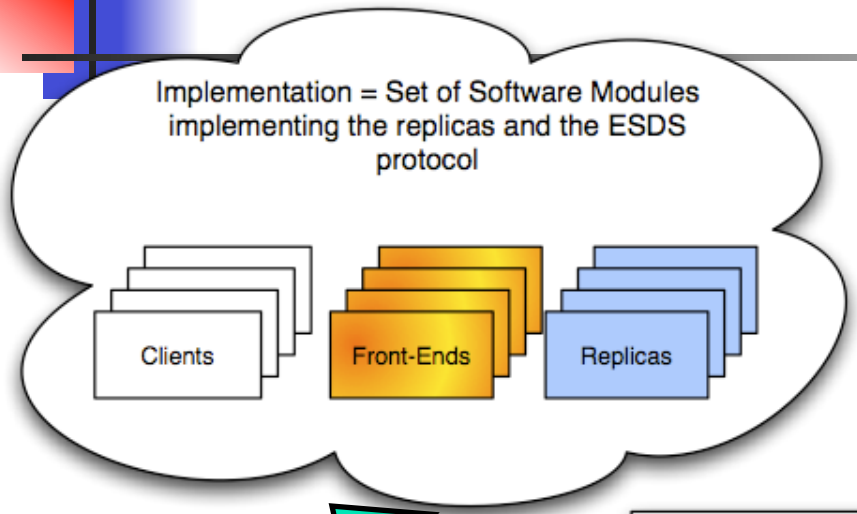
```
using {

   forall(c in C: !row[c].bound())
          by (row[c].getSize(),abs(n/2-c))

      tryall<cp>(r in R: row[c].memberOf(r))
                 by (col[r].getSize())

         cp.post(row[c] == r);

}
```



Variable/Value Labeling

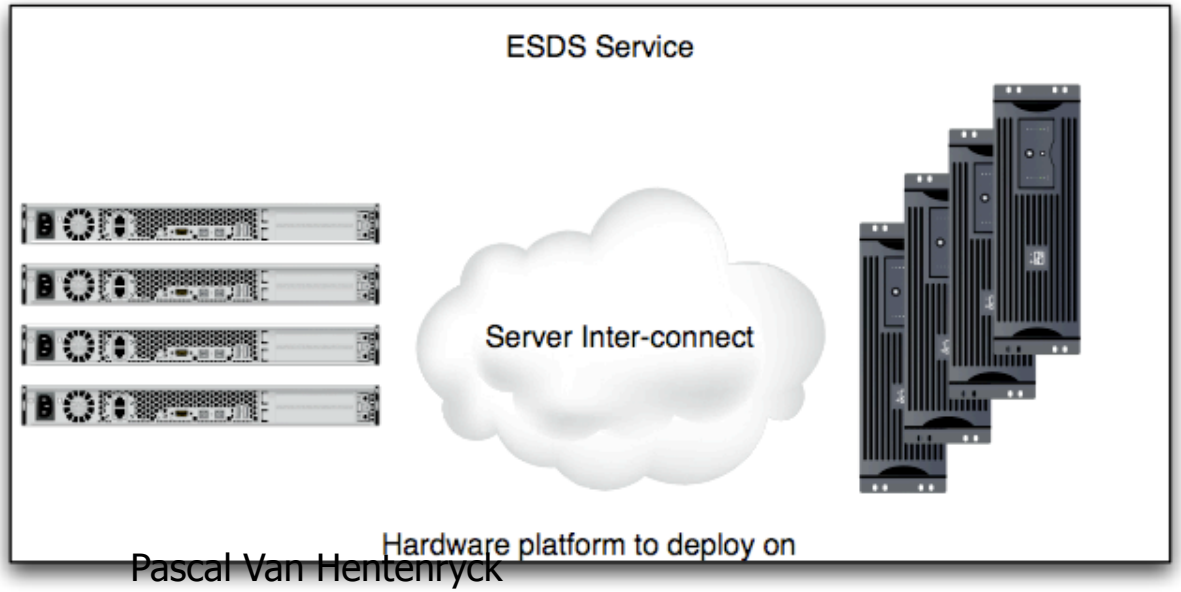
- Variable choice
  - often the most constrained variable
- Value choice
  - often a value that decreases the solution space the least

Serializable Data Services



The Problem

Map the software modules on the hardware to minimize network traffic induced by protocol



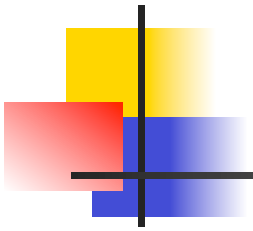
The ESDS Deployment Problem

- Generalized quadratic assignment problem
  - *f* : the communication frequency matrix
  - *h* : the distance matrix (# hops)
  - *x :* the assignment vector (decision variables)
  - C (set of components)
  - S (set of separation constraints), Col (set of co-locations)

$$\min_{x \in \mathbb{N}^n} \sum_{a \in C} \sum_{b \in C} f_{a,b} \cdot h_{x_a, x_b}$$

The CP Model

```
minimize<cp>
    sum(a in C,b in C: a != b) f[a,b] * h[x[a],x[b]]
subject to {
   forall(S in Col,c1 in S,c2 in S: c1 < c2)
      cp.post(x[c1] == x[c2]);
   forall(S in Sep)
     cp.post(alldifferent(all(c in S) x[c]),onDomains);
}
using {
 while (!bound(x))
    selectMax(i in C: !x[i].bound(), j in C)(f[i,j])
      tryall(n in N) by (min(l in N: x[j].memberOf(l))h[n,l])
        cp.post(x[i] == n);
}
```



The Search Procedure

- Key ideas
  - Minimize the largest term of the objective function
    - Pick first modules that communicate frequently
      - Decreasing frequency
    - Assign them to "close-by" locations
      - Increasing hops (distances)

The CP Model

```
minimize<cp>
    sum(a in C,b in C: a != b) f[a,b] * h[x[a],x[b]]
subject to {
   forall(S in Col,c1 in S,c2 in S: c1 < c2)
      cp.post(x[c1] == x[c2]);
   forall(S in Sep)
     cp.post(alldifferent(all(c in S) x[c]),onDomains);
}
using {
 while (!bound(x))
    selectMax(i in C: !x[i].bound(), j in C)(f[i,j])
      tryall(n in N) by (min(l in N: x[j].memberOf(l))h[n,l])
        cp.post(x[i] == n);
}
```

CPLEX Results (V11)

Bench	Opt.	Run Time (sec.)	Time to Opt. (sec.)	# Nodes at End	max Tree Size(MB)	Reduced MIP #Rows	Reduced MIP #Cols	Gap When Optimum Found
SIMPLE2	194	4	2	68		8834	3070	61.34%
SIMPLE1	194	1420	247	4238	2.58	11295	3900	25.77%
SIMPLE0	194	18	12	502		14056	4830	38.49%
fe3c5pc	298	5091	1717	22741	25.04	18442	6312	39.79%
fe3c5pc5	298	5407	960	26360	30.59	19804	6770	37.85%
fe3c5sun	258	7877	210	29216	33.94	20458	6990	88.50%
fe3c6pc5	312	10556	507	35901	41.54	21605	7375	78.21%
fe3c7pc5	322	14712	312	26818	34.38	25356	8635	70.50%
fe3c7pc5CS	322	44677	387	81673	116.00	25356	8635	67.82%
fe3c7pc5CST	338	6154	480	51223	66.44	25290	8610	64.50%
fe3dist	248	4105	3100	33889	34.62	17097	5860	15.44%
SCSS1SNUFE	232	844	135	16350	14.73	17492	5994	70.08%
SCSS2SNUFE	268	1624	90	32846	29.82	17492	5994	83.33%
SCSS2SNCFE	268	3026	1551	72711	55.20	14048	4830	26.12%
HYPER8	512	53906	51381	124918	190.95	31583	10725	5.82%

Results for best settings.

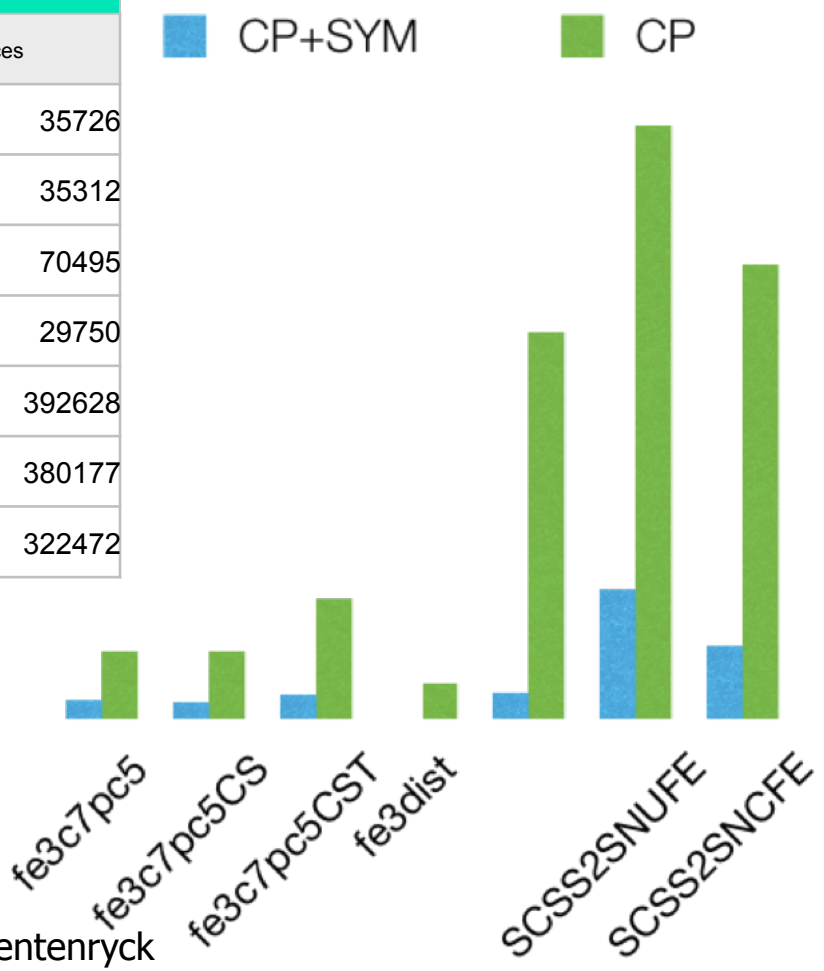
On an Athlon64 @ 2Ghz (Linux)

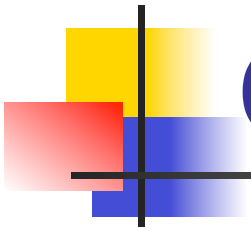
Constraint Programming Results

Bench	T_{end}	#Choices	T_{opt}
SIMPLE2	0.23	2510	0.19
SIMPLE1	1.38	15408	0.19
SIMPLE0	7.75	87491	0.19
fe3c5pc	2.76	14597	0.19
fe3c5pc5	4.64	24130	0.22
fe3c5sun	6.29	30601	0.20
fe3c6pc5	3.54	18547	0.20
fe3c7pc5	7.83	35726	0.20
fe3c7pc5CS	7.77	35312	0.20
fe3c7pc5CST	13.68	70495	0.20
fe3dist	4.16	29750	0.19
SCSS1SNUFE	43.34	392628	0.19
SCSS2SNUFE	66.43	380177	49.82
SCSS2SNCFE	50.83	322472	36.04
HYPER8	65.07	123213	8.06
HYPER16	309.46	513051	254.4

With Symmetry breaking

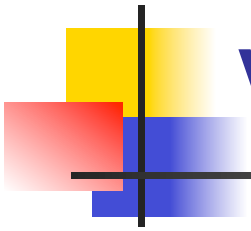
	CP + SYM		CP	
Bench	T_{end}	#Choices	T_{end}	#Choices
fe3c7pc5	2.21	8628	7.83	35726
fe3c7pc5CS	2.07	7949	7.77	35312
fe3c7pc5CST	3.03	13654	13.68	70495
fe3dist	0.31	1708	4.16	29750
SCSS1SNUFE	3.24	20805	43.34	392628
SCSS2SNUFE	14.74	71692	66.43	380177
SCSS2SNCFE	8.48	48740	50.83	322472





Outline

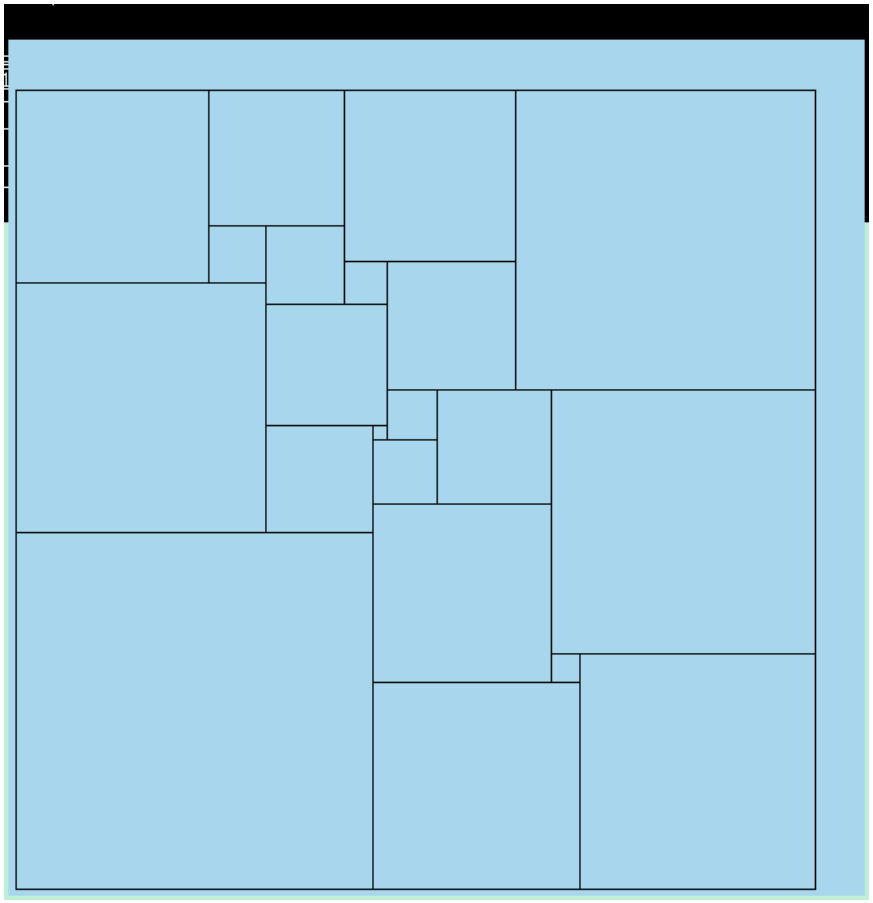
- Variable/Value Labeling
- Value/Variable Labeling
- Domain splitting
- Symmetry breaking during search
- Randomization and Restarts

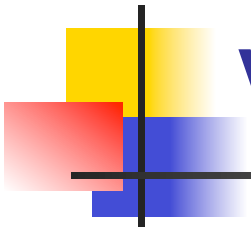


Value/Variable Labeling

- Two steps
  - first choose the value to assign next
  - choose the variable to assign to this value
- Why is it useful?
  - you may know that a value must be assigned
  - often the case in some scheduling and resource allocation problems

The Perfect Square Problem





Value/Variable Labeling

- Take a position t along the x axis
  - we know that a square must be placed there if the existing squares do not fill up the size
- Value/Variable heuristic
  - choose a x coordinate
  - choose a square to position there
  - repeat for all the x coordinates
  - repeat for all the y coordinates

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

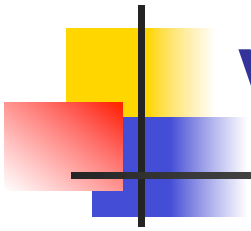
The Perfect Square Search

```
using {

  forall(p in Side)
    forall(i in Square)
      try<cp> cp.post(x[i] == p);|  cp.post(x[i] != p);

  forall(p in Side)
    forall(i in Square)
      try<cp> cp.post(y[i] == p);| cp.post(y[i] != p);
}
```

choose a x coordinate

consider a square

place it on p or do not place it on p

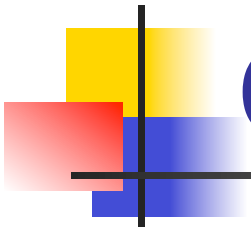


Value/Variable Labeling

- Value/Variable heuristic
  - choose a x coordinate
  - choose a square to position there
  - repeat for all the x coordinates
  - repeat for all the y coordinates
- Optimization
  - choose only positions where a square can be placed

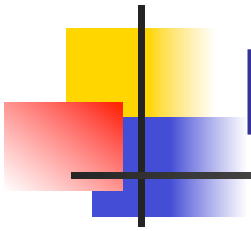
The Perfect Square Search

```
using {
  forall(k in Square)
    selectMin(p in Square: !x[p].bound(),r = x[p].getMin())(r)
      tryall<cp>(i in setof(k in Square)(!x[k].bound() && x[k].getMin() == r))
        cp.post(x[i] == r);
      onFailure
        cp.post(x[i] != r);

  forall(k in Square)
    selectMin(p in Square: !y[p].bound(),r = y[p].getMin())(r)
      tryall<cp>(i in setof(i in Square) (!y[i].bound() && y[i].getMin() == r))
        cp.post(y[i] == r);
      onFailure
        cp.post(y[i] != r);
}
```



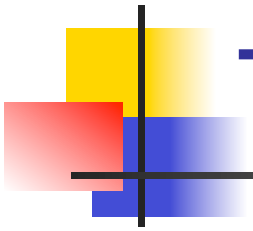
Outline

- Variable/Value Labeling
- Value/Variable Labeling
- Domain splitting
- Symmetry breaking during search
- Randomization and Restarts
- Large neighborhood search



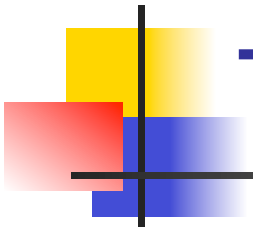
Domain Splitting

- Motivation
  - labeling entails drastic decisions
  - sometimes some weaker commitment is preferable
- Domain splitting
  - partition the domain of a variable



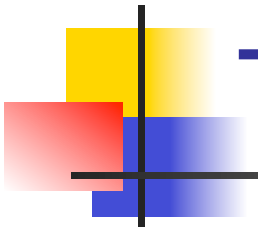
The Magic Square Problem

2	9	4
7	5	3
6	1	8



The Magic Square Problem

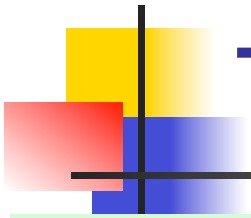
1	90	18	14	119	112	117	118	20	54	8
93	2	99	16	29	92	94	31	68	56	91
39	75	3	5	12	87	95	101	57	97	100
21	83	103	4	86	78	84	58	89	15	50
79	70	98	107	55	51	59	28	25	76	23
24	106	63	32	109	60	9	108	104	34	22
69	49	105	110	61	26	82	7	13	72	77
73	46	102	62	40	36	38	113	81	33	47
71	42	64	114	41	27	44	53	115	52	48
80	65	6	111	74	67	19	37	11	116	85
121	43	10	96	45	35	30	17	88	66	120



The Magic Square Problem

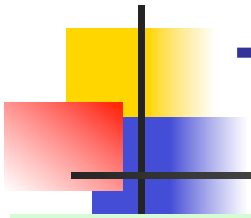
	1	90	18	14	119	112	117	118	20	54	8
	93	2	99	16	29	92	94	31	68	56	91
	39	75	3	5	12	87	95	101	57	97	100
	21	83	103	4	86	78	84	58	89	15	50
	79	70	98	107	55	51	59	28	25	76	23
	24	106	63	32	109	60	9	108	104	34	22
	69	49	105	110	61	26	82	7	13	72	77
	73	46	102	62	40	36	38	113	81	33	47
	71	42	64	114	41	27	44	53	115	52	48
	80	65	6	111	74	67	19	37	11	116	85
	121	43	10	96	45	35	30	17	88	66	120

Labeling a variable is a very strong choice



The Magic Square Model

```
range R= 1..n; range D= 1..n^2; int T=n*(n^2+1)/2;
var<CP>{int} s[R,R](cp,D);
solve<cp> {
   forall(i in R) {
      cp.post(sum(j in R) s[i,j] == T);
      cp.post(sum(j in R) s[j,i] == T);
   }
   cp.post(sum(i in R) s[i,i] == T);
   cp.post(sum(i in R) s[i,n-i+1] == T);
   cp.post(alldifferent(all(i in R,j in R) s[i,j]));
   forall(i in 1..n-1) {
      cp.post(s[i,i] < s[i+1,i+1]);
      cp.post(s[i,n-i+1] < s[i+1,n-i]);
   }
}
```



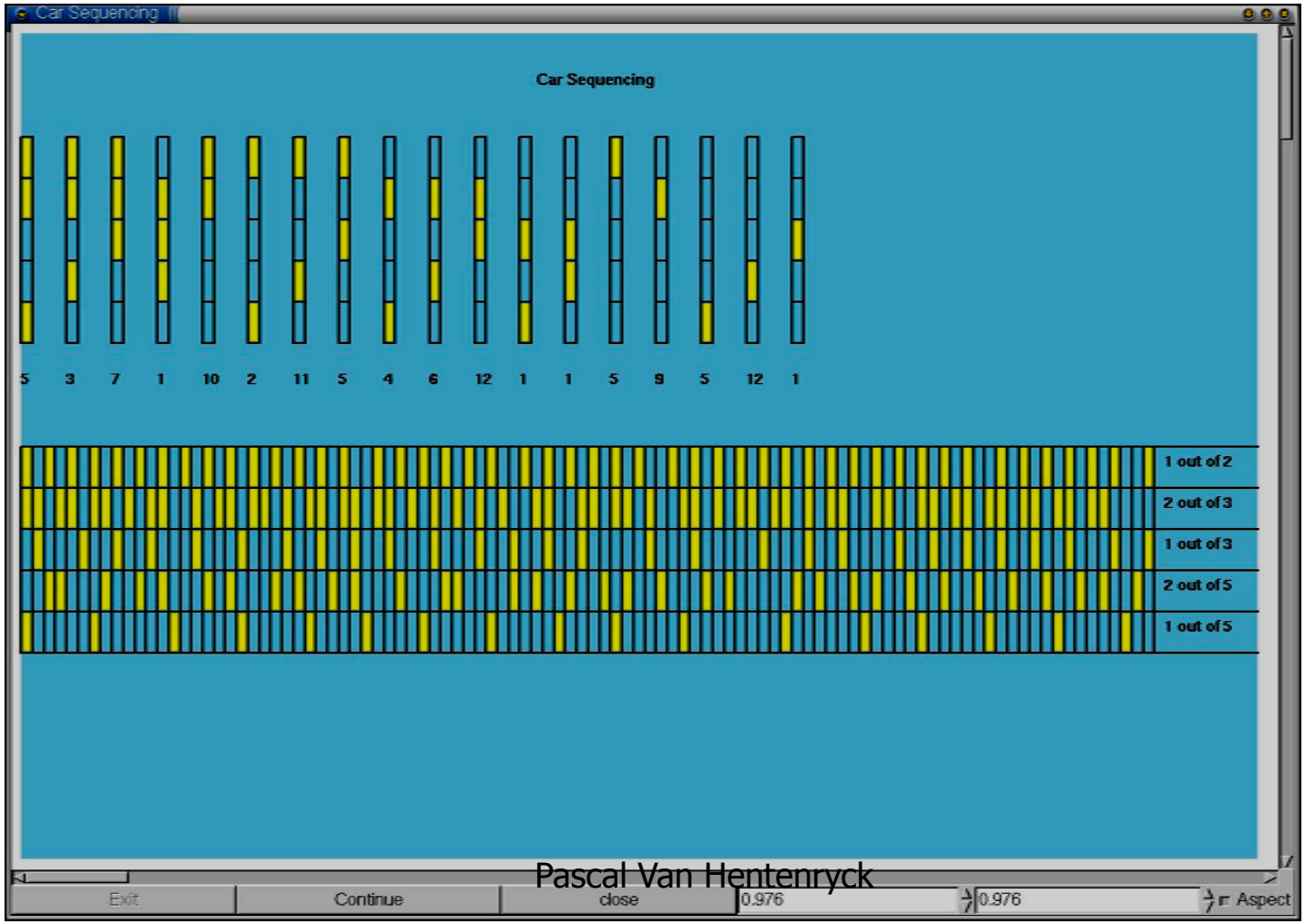
The Magic Square Search

```
using {
   var<CP>{int}[] x = all(i in R,j in R) s[i,j];
   range V = x.getRange();
   while (!bound(x)) {
      selectMin(i in V:!x[i].bound())(x[i].getSize()) {
         int mid = (x[i].getMin()+x[i].getMax())/2;
         try<cp>
            cp.post(x[i] <= mid);
         |
            cp.post(x[i] > mid);
       }
   }
}
```

Car Sequencing

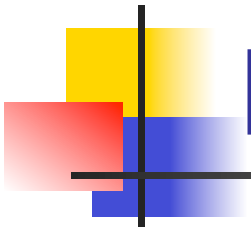
- Cars on an assembly line
- Cars have options (e.g. leather seats)
- Capacity constraints on the production units (2 out of 5)
- Sequencing

Car sequencing



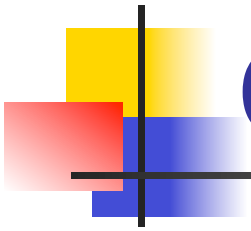
Car Sequencing


```
using {
   forall(o in Options) by (slack[o])
      forall(i in Cars)
         try<cp>
            cp.post(inside(line[i],options[o]));
         |
            cp.post(outside(line[i],options[o]));
}
```



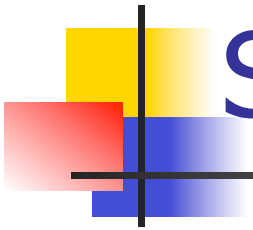
Domain Splitting

- Motivation
  - focus on difficult options
  - decide which slots take this option
- Domain splitting
  - do not assign configurations to slots
  - branch on two cases
    - those configurations with the option
    - those configurations without the option



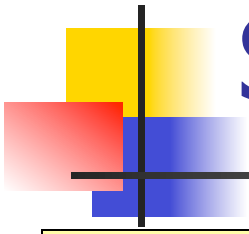
Outline

- Variable/Value Labeling
- Value/Variable Labeling
- Domain splitting
- Symmetry breaking during search
- Randomization and Restarts
- Large neighborhood search



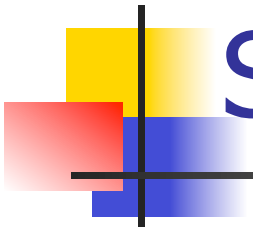
Scene Allocation

- 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](m,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));
```

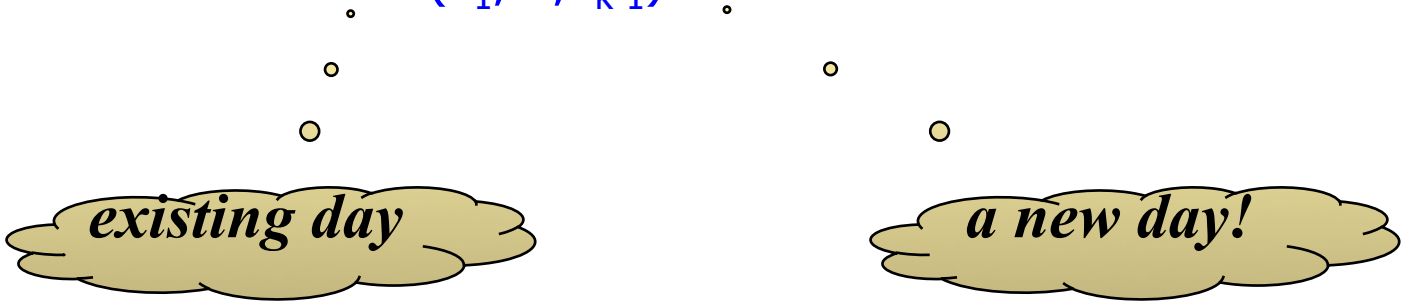


Scene Allocation

- 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

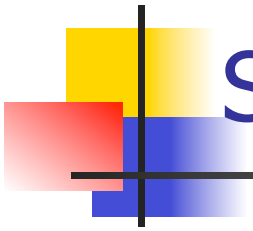
- Value Symmetries
  - the days are interchangeable
  - If s is a solution, p(s) is a solution, where p(s) denotes the solution s where the days have been permuted with permutation p
- How to eliminate value symmetries
  - Consider the scene x_k. Which day must be considered?
    - $1..\max(x_1,\ldots,x_{k-1})+1$



Scene Allocation

```
var<CP>{int} shoot[Scenes](m,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(shoot[Scenes.getLow()] == Days.getLow());
  forall(s in Scenes: s != Scenes.getLow())
     cp.post(shoot[s] <= max(k in 1..s-1) shoot[k] + 1);
}
```

- This eliminates all the value symmetries
  - It interferes with the heuristic



Symmetry Breaking during Search

- Key idea
  - break symmetries during the search
  - dynamically impose the symmetry-breaking constraints
- How?
  - same constraints as in the model case
  - the order is different
  - it is discovered dynamically

Scene Allocation

```
var<CP>{int} shoot[Scenes](m,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
  - It interferes with the heuristic

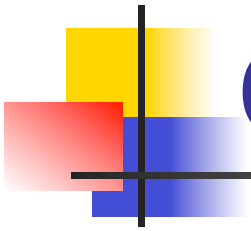
Scene Allocation Search

```
using {
   while (!bound(shoot)) {
      int mday = max(-1,maxBound(shoot));
      selectMin(s in Scenes: !shoot[s].bound())(shoot[s].getSize(),
                                                -sum(a in appears[s]) fee[a])
        tryall<cp>(d in 0..mday + 1)
           cp.post(shoot[s] == d);
  }
}
```

existing days!

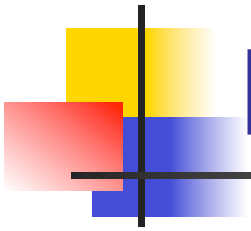
new day!

- This eliminates all the value symmetries
- It does not interfere with the heuristic



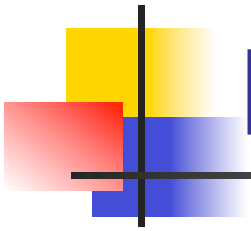
Outline

- Variable/Value Labeling
- Value/Variable Labeling
- Domain splitting
- Symmetry breaking during search
- Randomization and Restarts
- Large neighborhood search



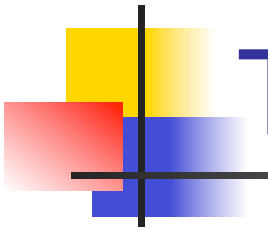
Randomization and Restarts

- Motivation
  - sometimes there is no (obvious) good variable ordering heuristic
  - but there exist good ones
- How to find them?
  - use randomization and restarts
  - choose the variables randomly
  - restarts if too many failures



Randomization and Restarts

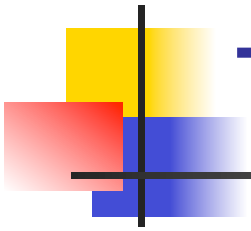
- Clean separation
  - Model
  - Search: and/or tree
  - Search exploration: restarts



The Magic Square Problem

	1	90	18	14	119	112	117	118	20	54	8
	93	2	99	16	29	92	94	31	68	56	91
	39	75	3	5	12	87	95	101	57	97	100
	21	83	103	4	86	78	84	58	89	15	50
	79	70	98	107	55	51	59	28	25	76	23
	24	106	63	32	109	60	9	108	104	34	22
	69	49	105	110	61	26	82	7	13	72	77
	73	46	102	62	40	36	38	113	81	33	47
	71	42	64	114	41	27	44	53	115	52	48
	80	65	6	111	74	67	19	37	11	116	85
	121	43	10	96	45	35	30	17	88	66	120

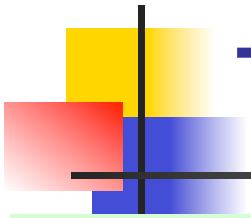
- Where do we start?



The Magic Square Search

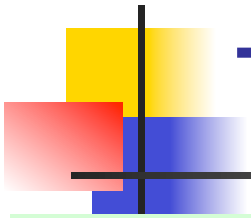
- First we randomize

```
using {
   var<CP>{int}[] x = all(i in R,j in R) s[i,j];
   range V = x.getRange();
   while (!bound(x))
      selectMin[3](i in V:!x[i].bound())(x[i].getSize()) {
         int mid = (x[i].getMin()+x[i].getMax())/2;
         try<cp> cp.post(x[i] <= mid);| cp.post(x[i] > mid);
       }
}
```



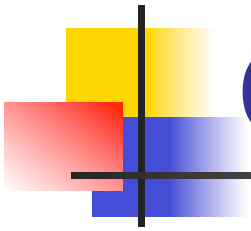
The Magic Square Search

```
cp.restartOnFailureLimit(100);
explore<cp> {
  …
}
using {
  var<CP>{int}[] x = all(i in R,j in R) s[i,j];
  range V = x.getRange();
  while (!bound(x))
    selectMin[3](i in V:!x[i].bound())(x[i].getSize()) {
       int mid = (x[i].getMin()+x[i].getMax())/2;
       try<cp> cp.post(x[i] <= mid)| cp.post(x[i] > mid);
    }
}
```



The Magic Square Search

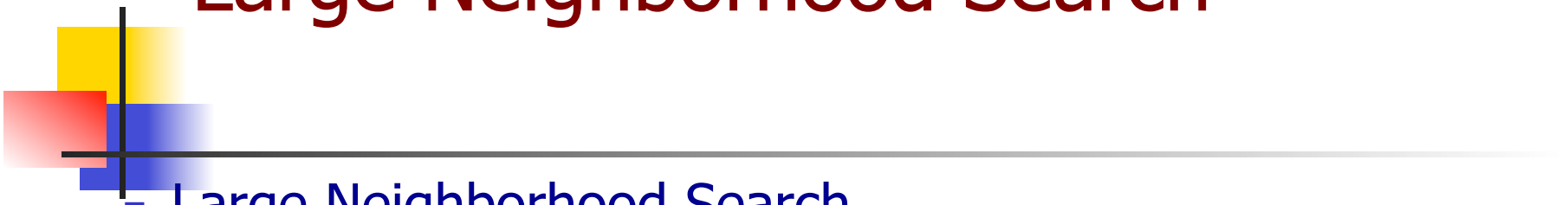
```
cp.restartOnFailureLimit(100);
explore<cp> {
  …
}
using {
  ..
}
onRestart {
  cp.setRestartFailureLimit(cp.getRestartFailureLimit()+100);
}
```



Outline

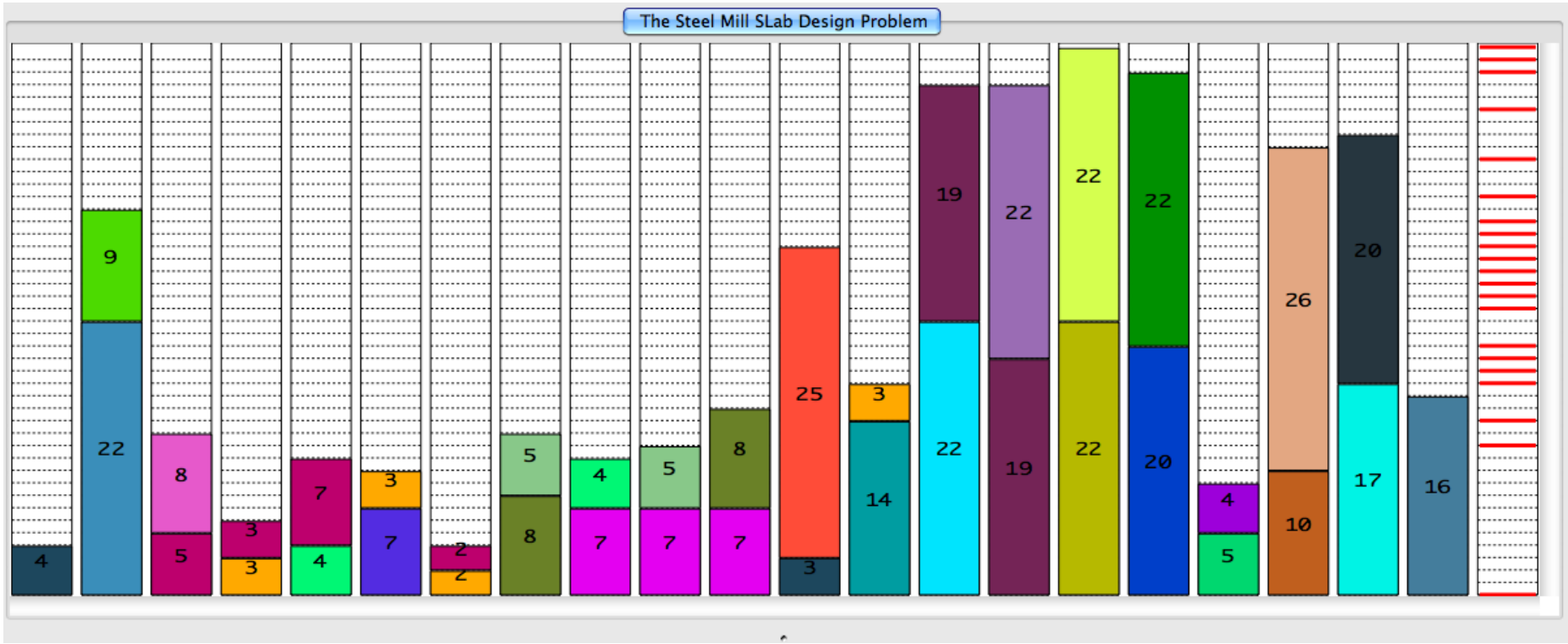
- Variable/Value Labeling
- Value/Variable Labeling
- Domain splitting
- Symmetry breaking during search
- Randomization and Restarts
- Large neighborhood search

Large Neighborhood Search

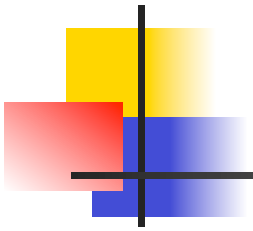


- Large Neighborhood Search
  - Combination of constraint programming and local search
- Start with a CP solution
  - CP is good to find feasible solution
- Iterate two steps
  - Relax a part of the solution (fix the rest)
  - Reinsert the rest using CP
- Strengths
  - Exploit the strength of CP to explore strongly constrained combinatorial spaces

Steel Mill Slab Design



The CP Model



- Basic idea
  - Given a set of orders grouped together, it is easy to choose the slab size which minimize their loss
  - take the smallest size >= to the sum of the order sizes
  - Just precompute these!
- Now the problem consists of
  - allocate orders to slabs
  - satisfying the compatibility constraints
  - minimizing the loss given the precomputed information

Example, loss array

Slab Capacity	12	10	7
Orders	1,3,4	2	5
Load	12	8	6
Loss	0	2	1

Order	1	2	3	4	5
Size	5	8	3	4	6
Color	1	3	2	1	2

load	0	1	2	3	4	5	6	**7**	8	9	**10**	11	**12**
loss	0	6	5	4	3	2	1	0	2	1	0	1	0

For a given load, loss array gives you the corresponding loss

Slab Design

```
set{int} colorOrders[c in Colors] =
   filter(o in Orders) (color[o] == c);
int maxCap = max(i in Caps) capacities[i];
int loss[c in 0..maxCap] =
   min(i in Caps: capacities[i] >= c) capacities[i] - c;

var<CP>{int} x[Orders](cp,Slabs);
var<CP>{int} l[Slabs](cp,0..maxCap);
minimize<cp>
   sum(s in Slabs) loss[l[s]]
subject to {
   cp.post(multiknapsack(x,weight,l));
   forall(s in Slabs)
     cp.post(sum(c in Colors)(or(o in colorOrders[c])(x[o]==s))<= 2);
}
```

element

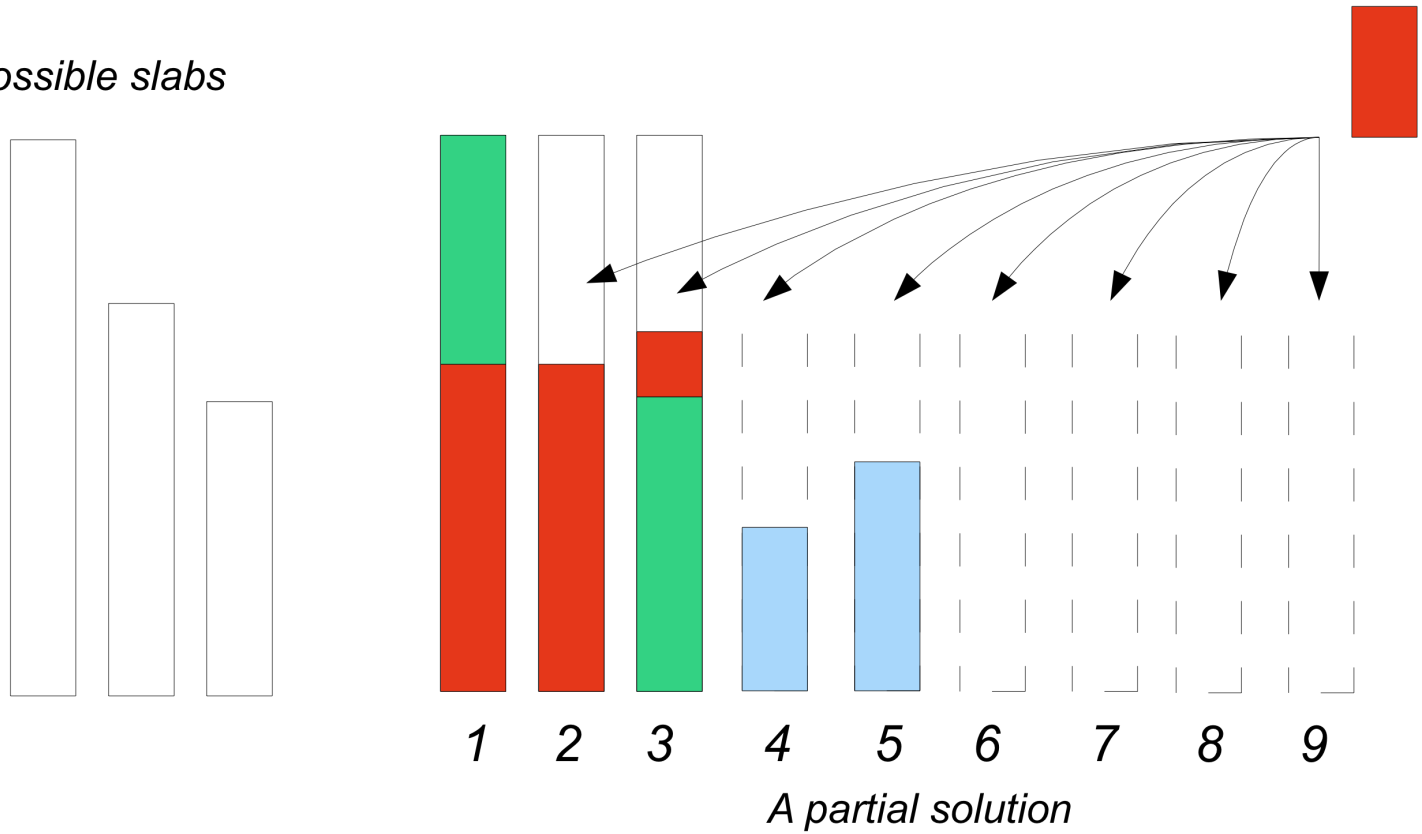
generalized multi-knapsack

cardinality

Slab Design

Branching:
Where to put the order?

Possible slabs

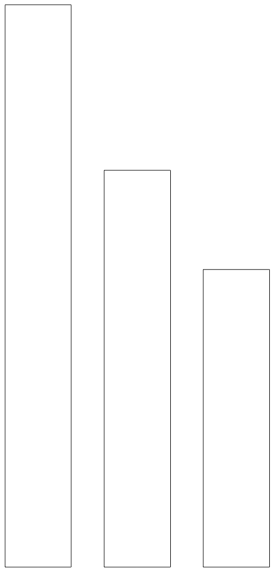


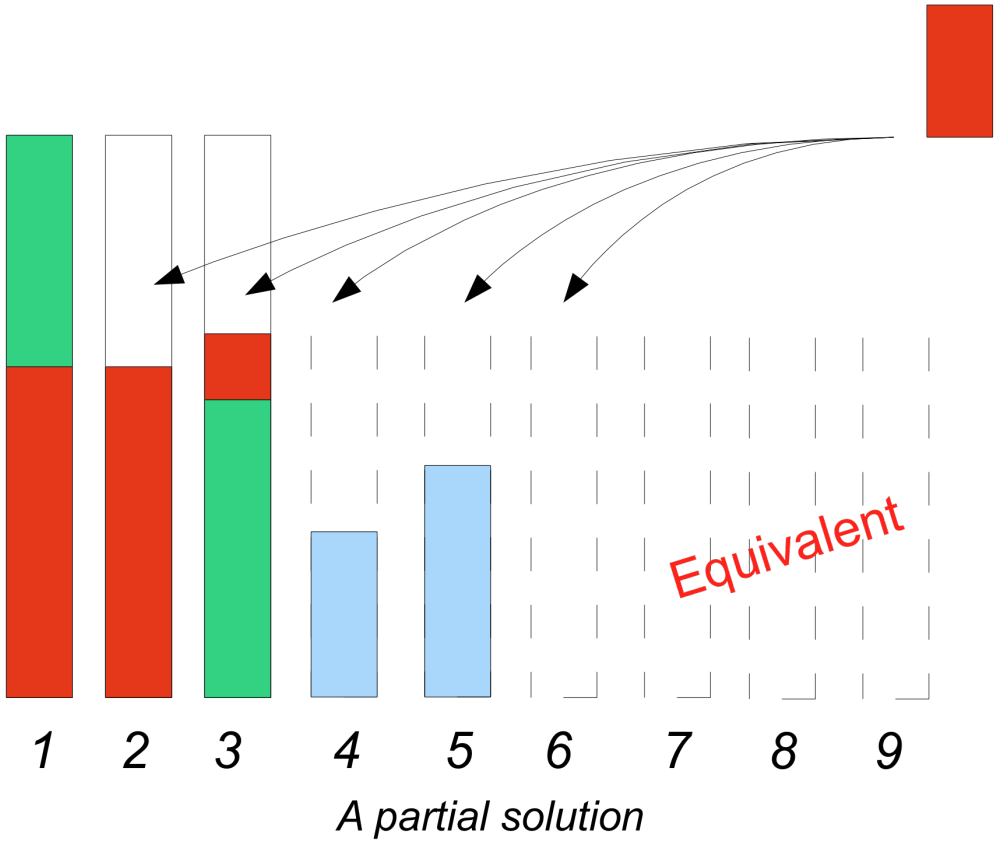
A partial solution

Slab Design

Branching:
Where to put the order?

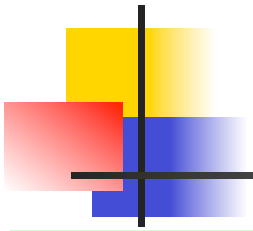
Possible slabs



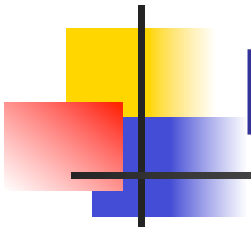


A partial solution

Slab Design



```
var<CP>{int} x[Orders](cp,Slabs);
var<CP>{int} l[Slabs](cp,0..maxCap);

minimize<cp>
    sum(s in Slabs) loss[l[s]]
subject to {
   cp.post(multiknapsack(x,weight,l));
   forall(s in Slabs)
    cp.post(sum(c in Colors) (or(o in colorOrders[c])(x[o]==s))<=2);
}
using {
   forall(o in Orders) by (x[o].getSize(),-weight[o]) {
      int ms = max(0,maxBound(x));
         tryall<cp>(s in Slabs: s <= ms + 1)
           cp.label(x[o],s);
       onFailure
          cp.diff(x[o],s);
   }
}
```

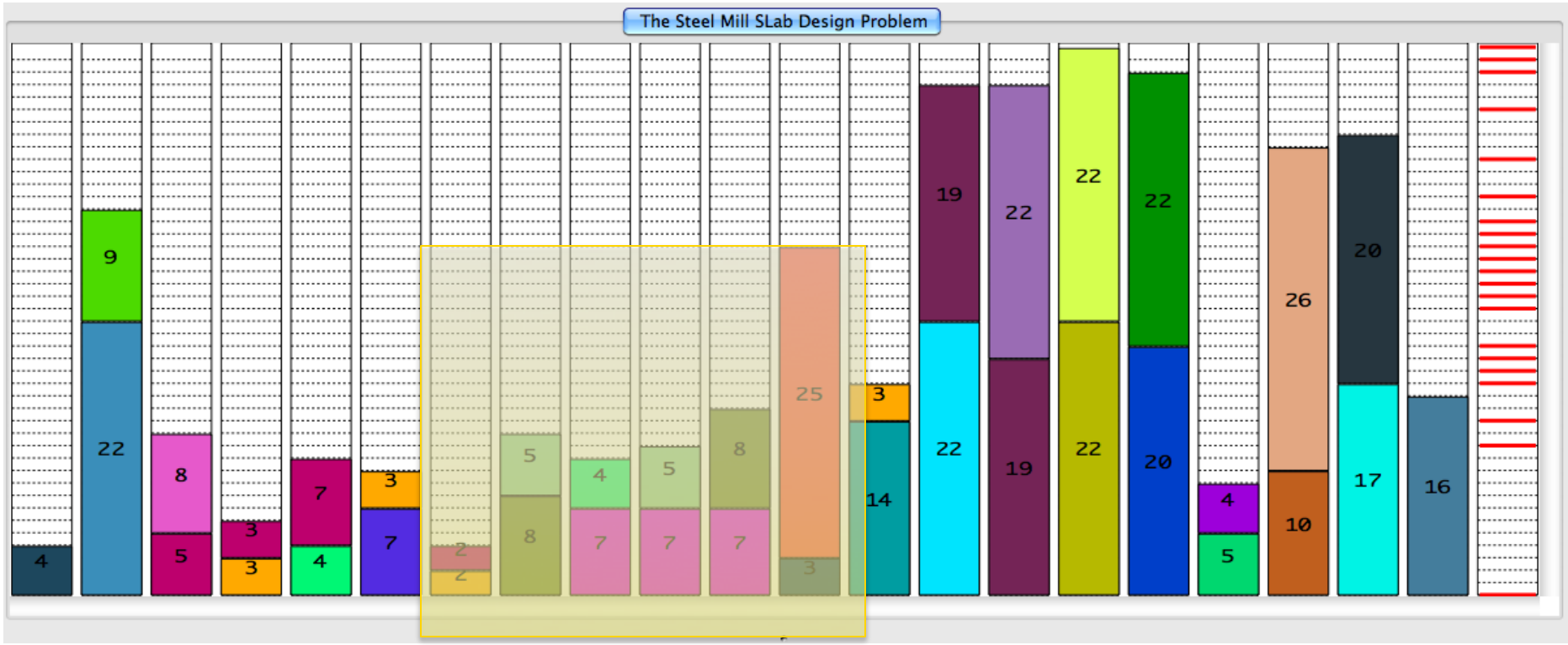


Large Neighborhood Search

- Relax a random set of variables
- Fix the remaining ones
  - to their values in an existing solution
- Repeat after a number of failures

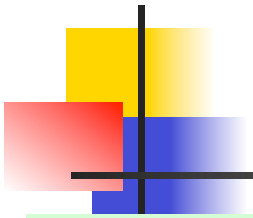
- Note that
  - CP then searches a small search space

Steel Mill Slab Design



Pascal Van Hentenryck

Slab Design



```
cp.lnsOnFailure(6000);
UniformDistribution dist(1..100);
var<CP>{int} x[Orders](cp,Slabs);
var<CP>{int} l[Slabs](cp,0..maxCap);
minimize<cp>
    sum(s in Slabs) loss[l[s]]
subject to { … }
using {
   …
}
onRestart {
   Solution s = CP.getSolution();
   if (s != null)
      forall(o in orders)
       if (dist.get() <= 90)
          cp.post(x[o]==x[o].getSnapshot(s));
}
```

Experimental Results