Discrete Optimization

Constraint Programming: Part I

Goal of the Lecture

- Basic introduction to constraint programming

Constraint Programming

- Computational paradigm
  - use constraints to reduce the set of values that each variable can take
  - remove values that cannot appear in any solution

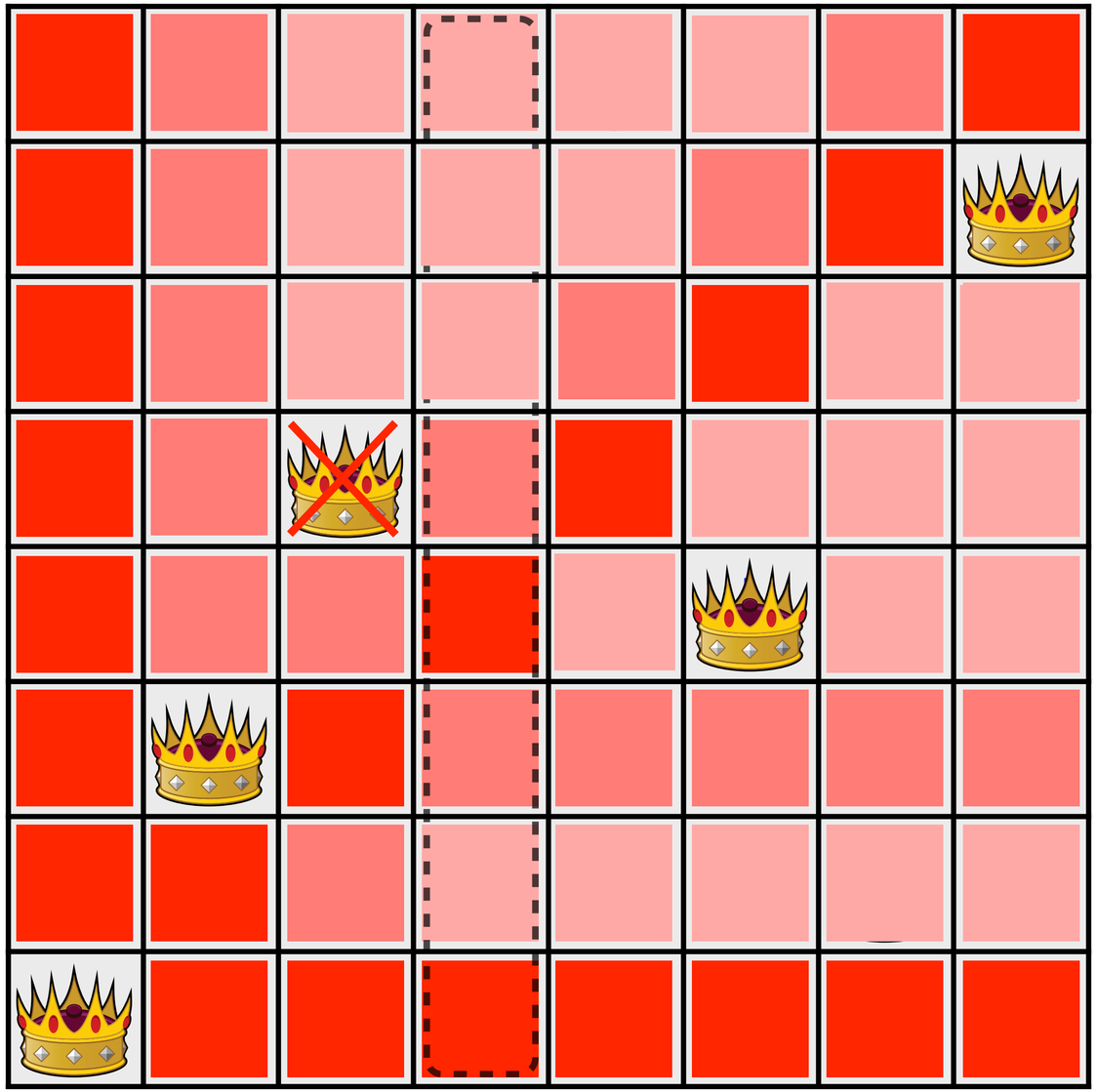
- Modeling methodology
  - convey the structure of the problem as explicitly as possible
  - express substructures of the problem
  - give solvers as much information as possible

The 8-Queens Problem

- Specification
  - place 8 queens on a chessboard so that none of them attach each other
  - two queens attack each other if they are on the same column, row, or diagonal

The 8-Queens Problem

Failure!

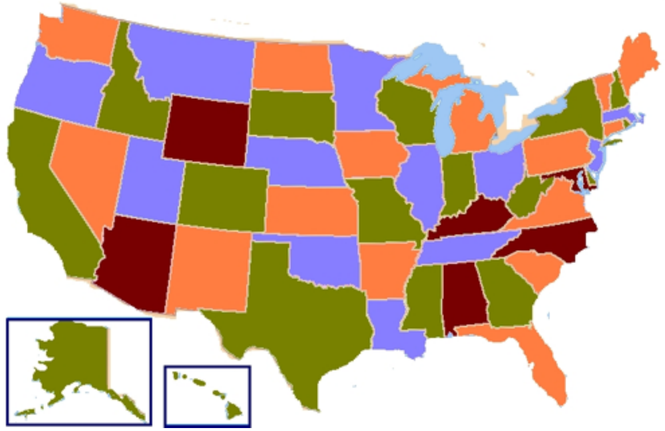


Constraint Programming

- Computational paradigm
  - use constraints to reduce the set of values that each variable can take
  - make a choice when no more deduction can be performed
- What is a choice?
  - there are many choices!
  - for the moment, assume a choice assigns a value to a variable
- Choices can be wrong
  - in optimization, they are often wrong :-(
  - the solver then backtracks,
    i.e., it tries another value

Coloring a Map

- Specification
  - color a map so that no two adjacent territories receive the same color
- The 4 Color Theorem
  - Every map can be colored with just 4 colors
  - Proven by Kenneth Appel and Wolfgang Haken
  - First major theorem proven with a computer



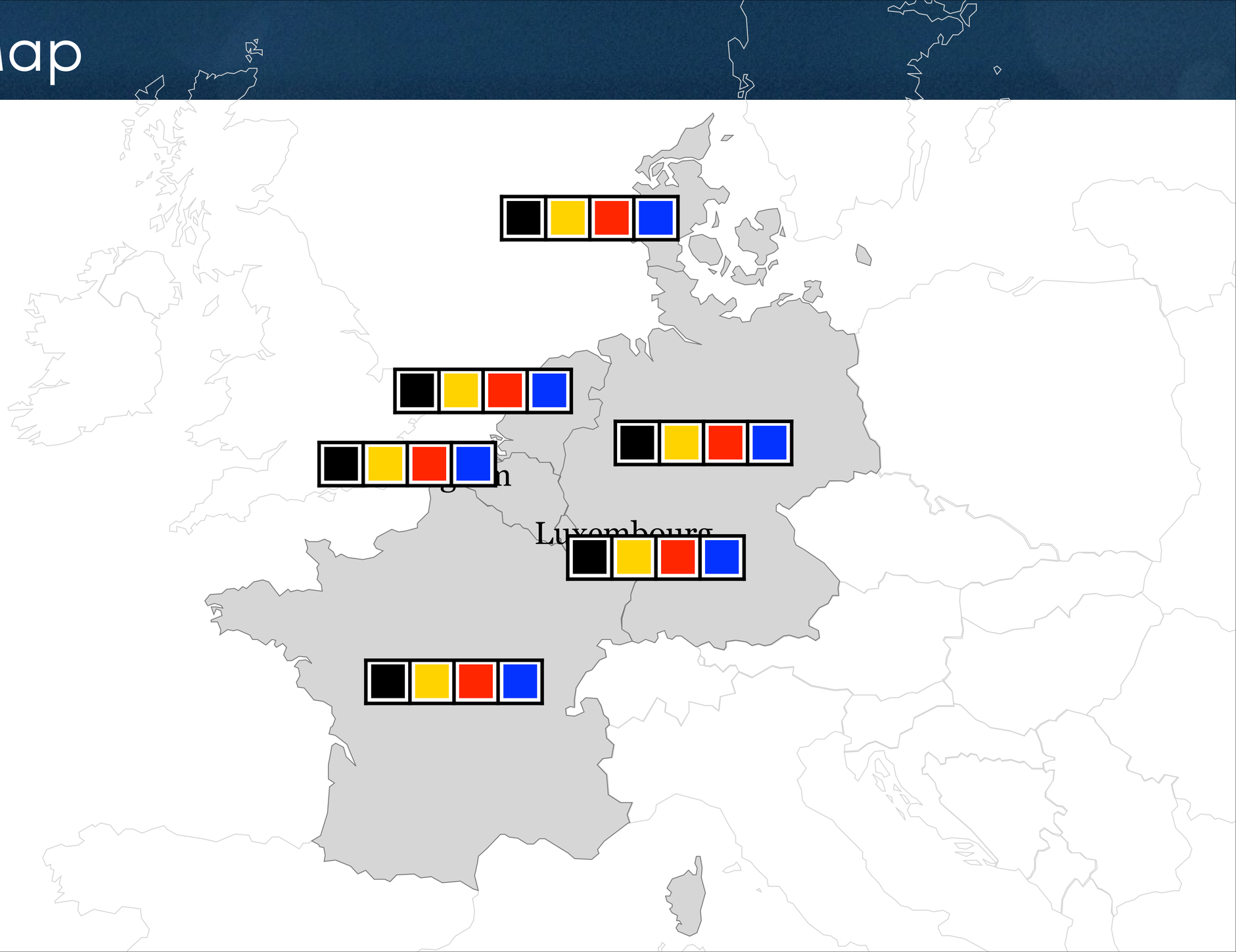
Coloring a Map

- How to color this map with constraint programming?
  - choose the decision variables
  - express the constraints in terms of the decision variables
- What are the decision variables?
  - the color given to each country
- What are the domains of the decision variables?
  - the domain is the set of values that a variable can take
  - four different colors
- How do you express the constraints?
  - specify that two adjacent countries cannot be given the same color

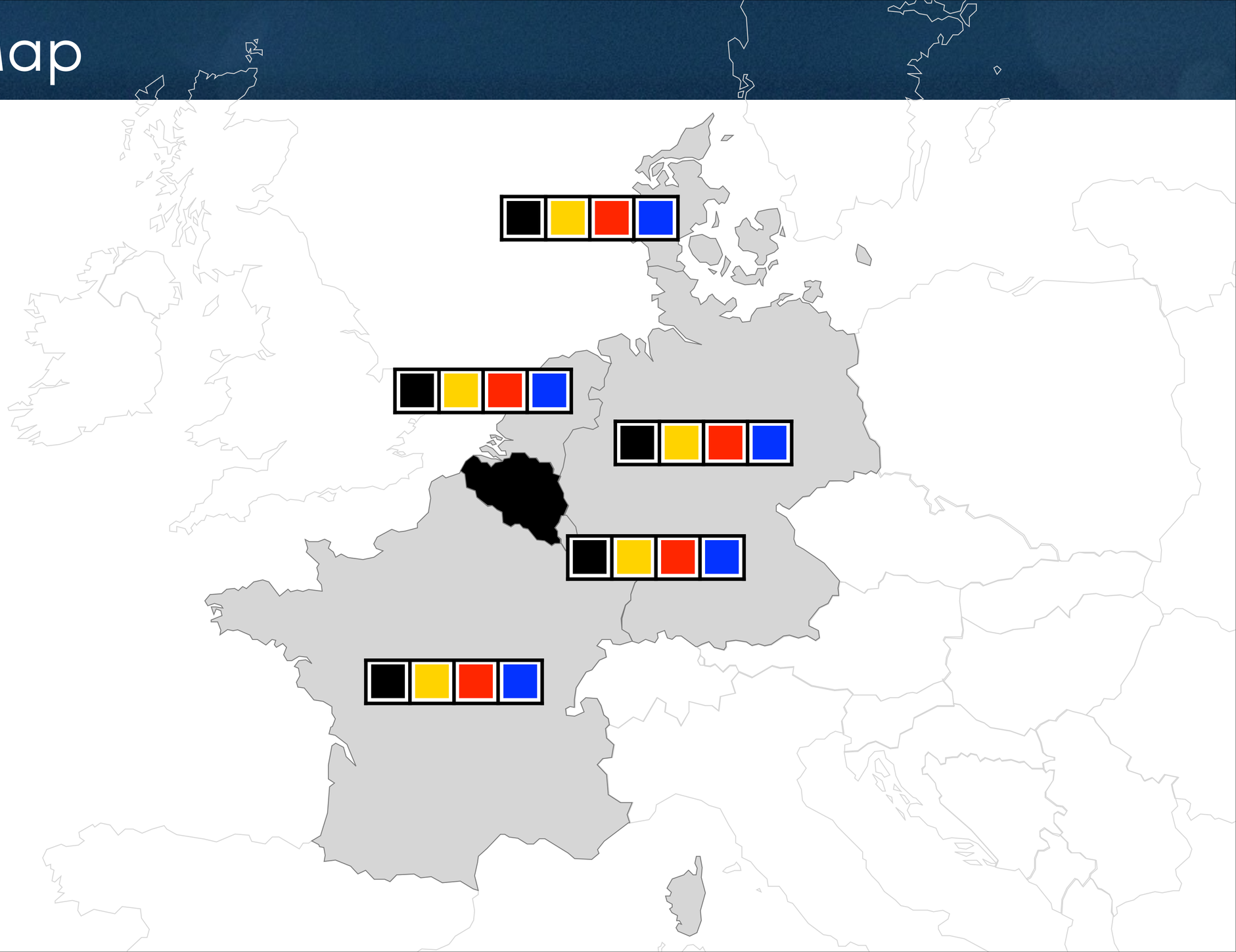
Coloring a Map

```
enum Countries = { Belgium, Denmark, France, Germany,
                   Netherlands, Luxembourg };
enum Colors = { black, yellow, red, blue };
var{Colors} color[Countries];

solve {
  color[Belgium] ≠ color[France];
  color[Belgium] ≠ color[Germany];
  color[Belgium] ≠ color[Netherlands];
  color[Belgium] ≠ color[Luxembourg];
  color[Denmark] ≠ color[Germany];
  color[France] ≠ color[Germany];
  color[France] ≠ color[Luxembourg];
  color[Germany] ≠ color[Netherlands];
  color[Germany] ≠ color[Luxembourg];
}
```

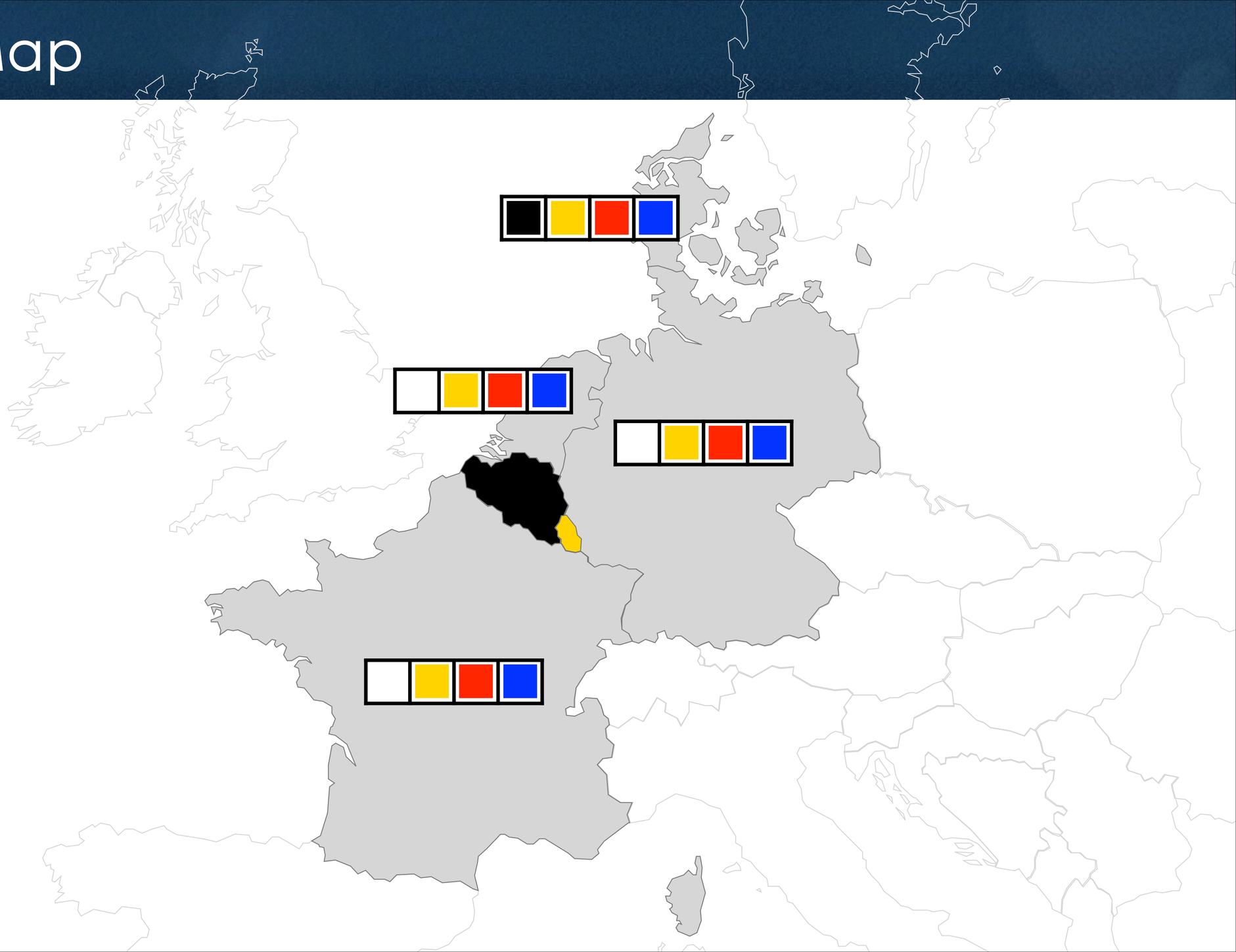
Coloring a Map



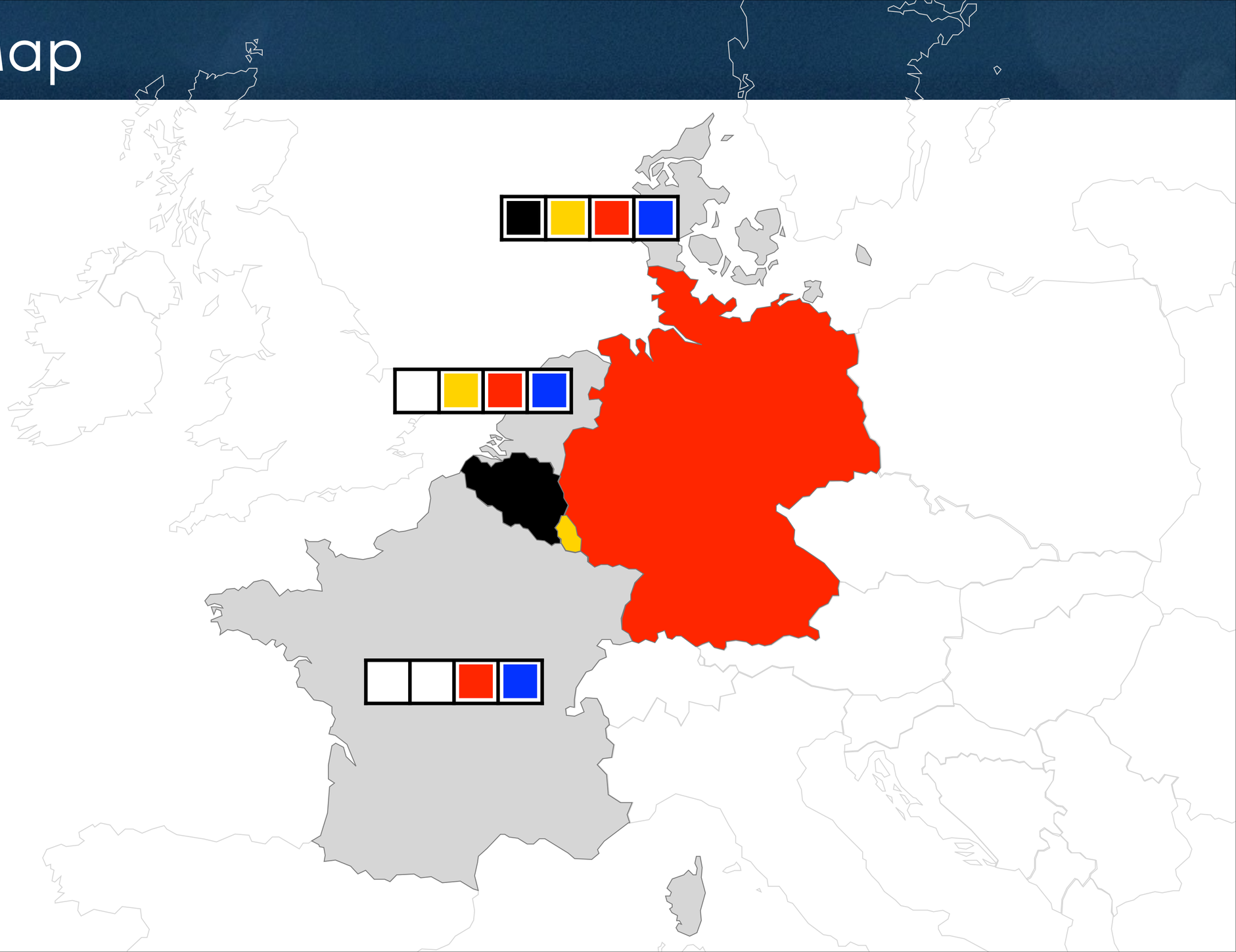
Coloring a Map



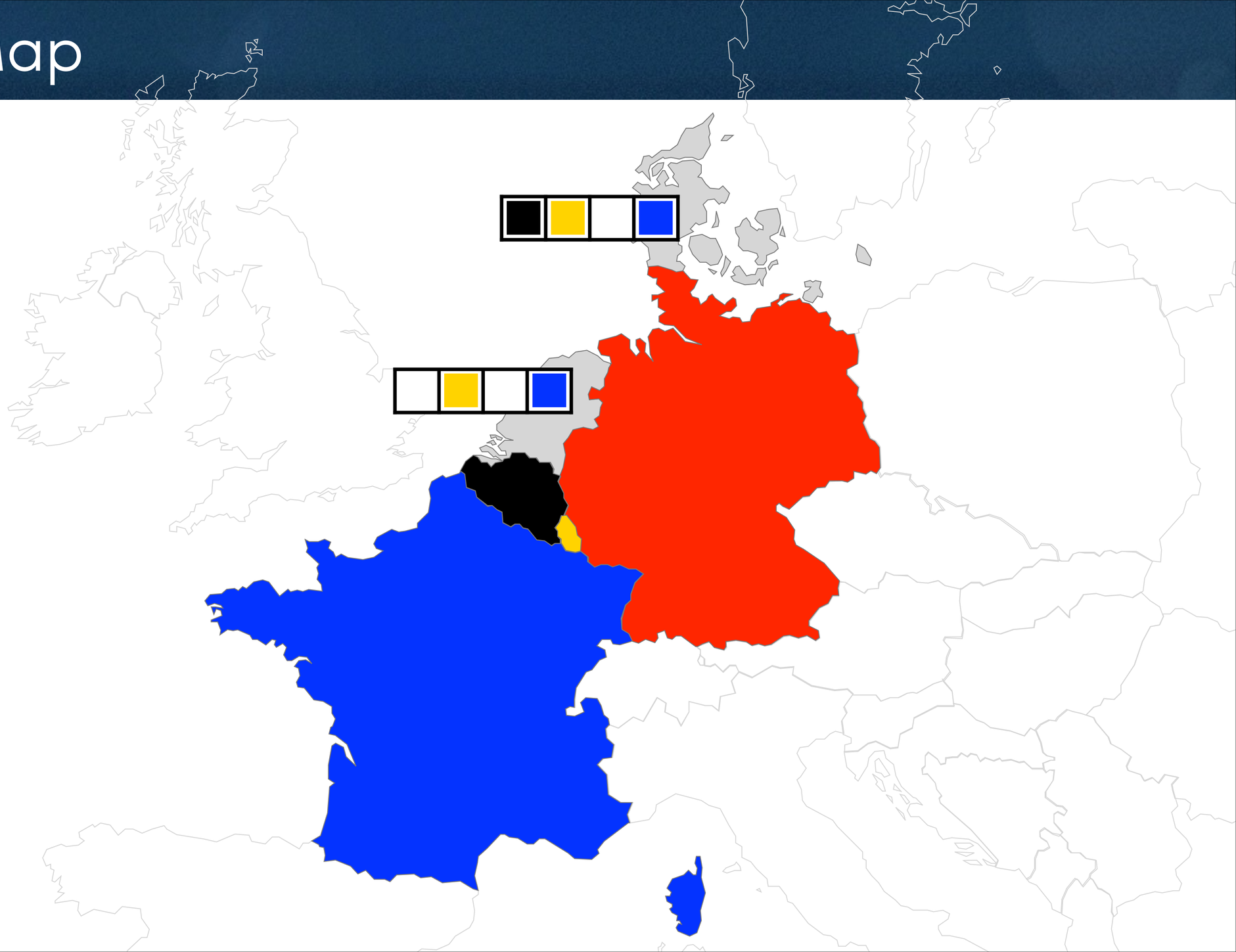
Coloring a Map



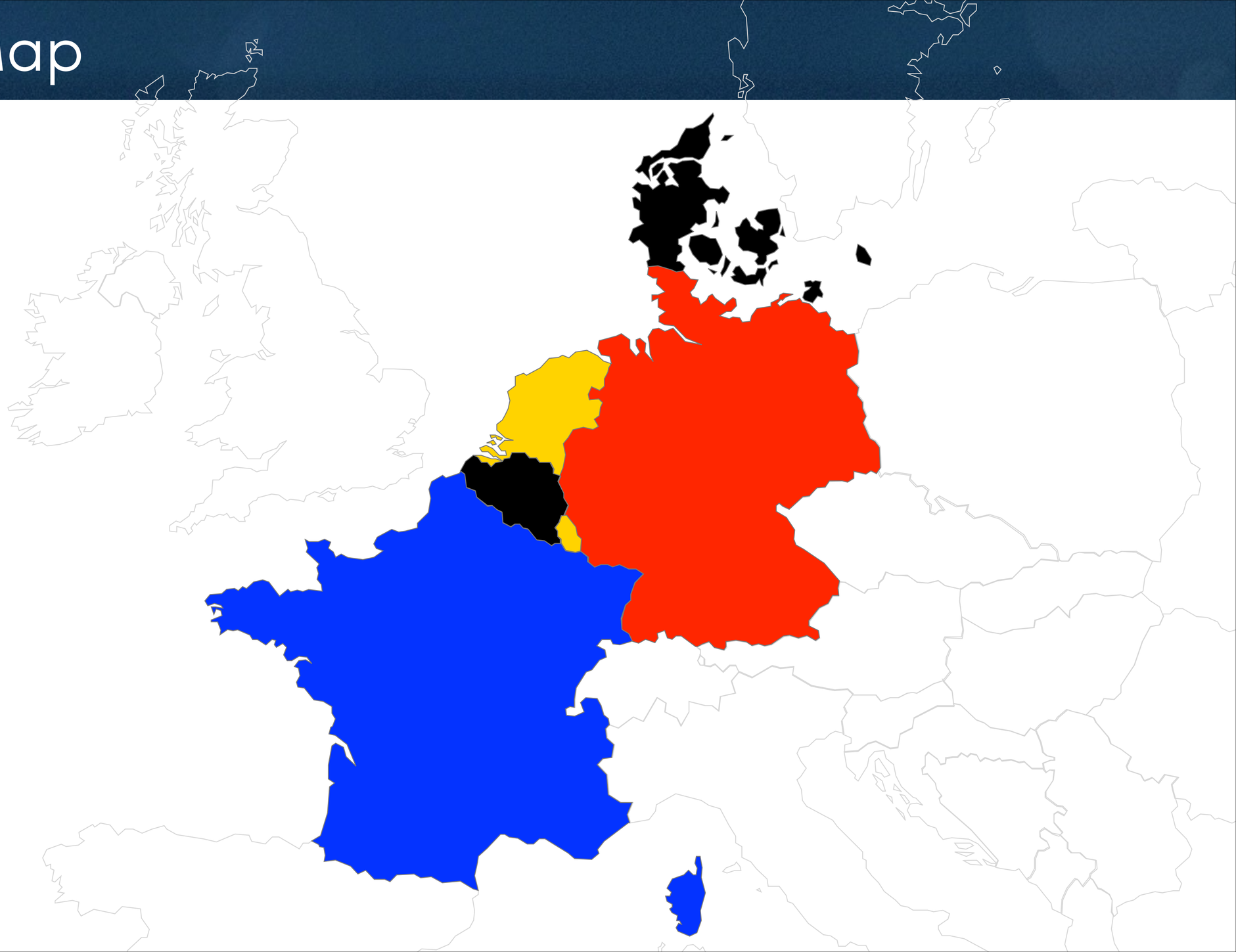
Coloring a Map



Coloring a Map



Coloring a Map



Constraint Programming

- Computational paradigm
  - use constraints to reduce the set of values that each variable can take
  - make a choice when no more deduction can be performed

- What does this mean for the coloring problem?

Coloring a Map

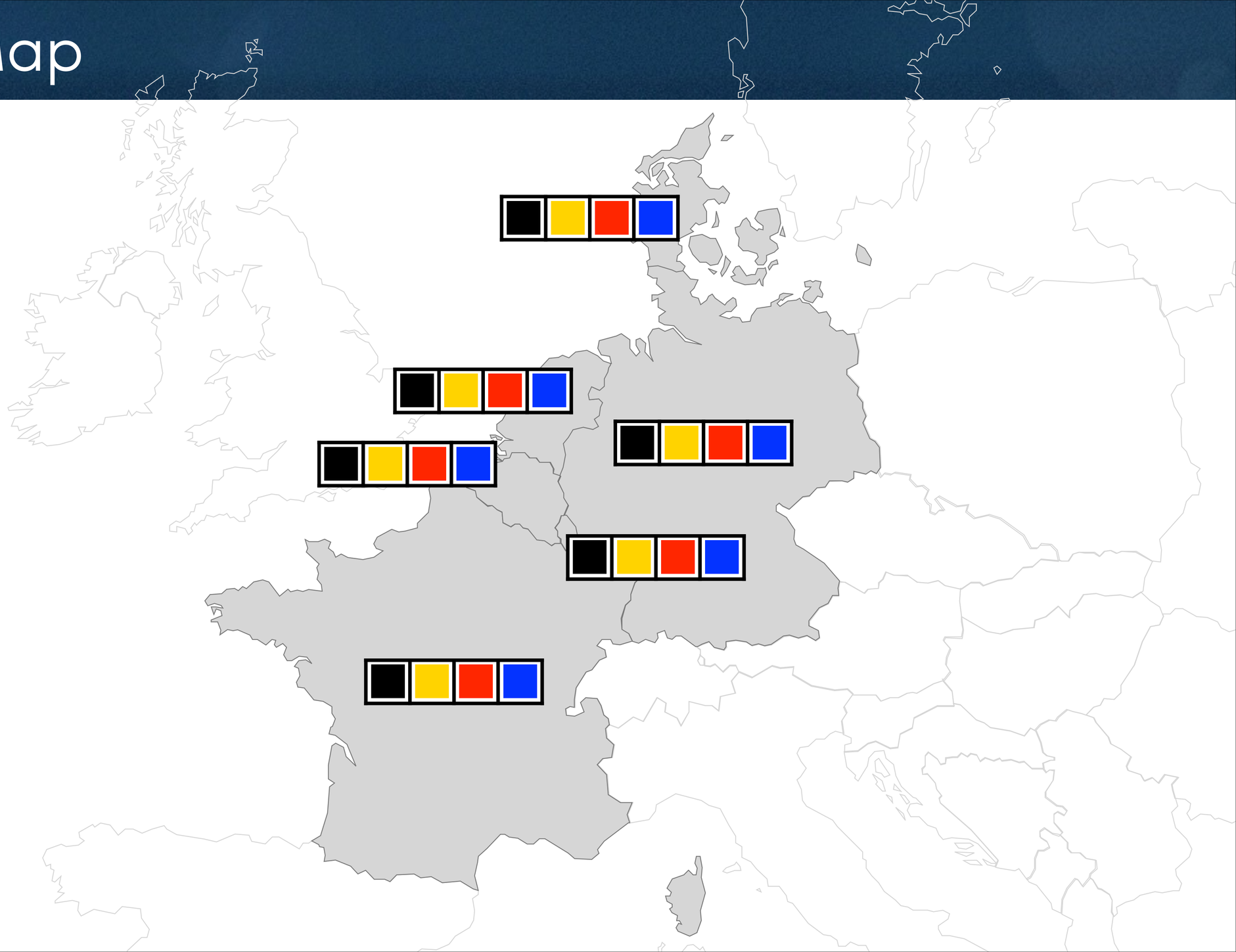
```
enum Countries = { Belgium, Denmark, France, Germany,
                   Netherlands, Luxembourg };
enum Colors = { black, yellow, red, blue };
var{Colors} color[Countries];

solve {
  color[Belgium] ≠ color[France];
  color[Belgium] ≠ color[Germany];
  color[Belgium] ≠ color[Netherlands];
  color[Belgium] ≠ color[Luxembourg];
  color[Denmark] ≠ color[Germany];
  color[France] ≠ color[Germany];
  color[France] ≠ color[Luxembourg];
  color[Germany] ≠ color[Netherlands];
  color[Germany] ≠ color[Luxembourg];
}
```

Constraint Programming

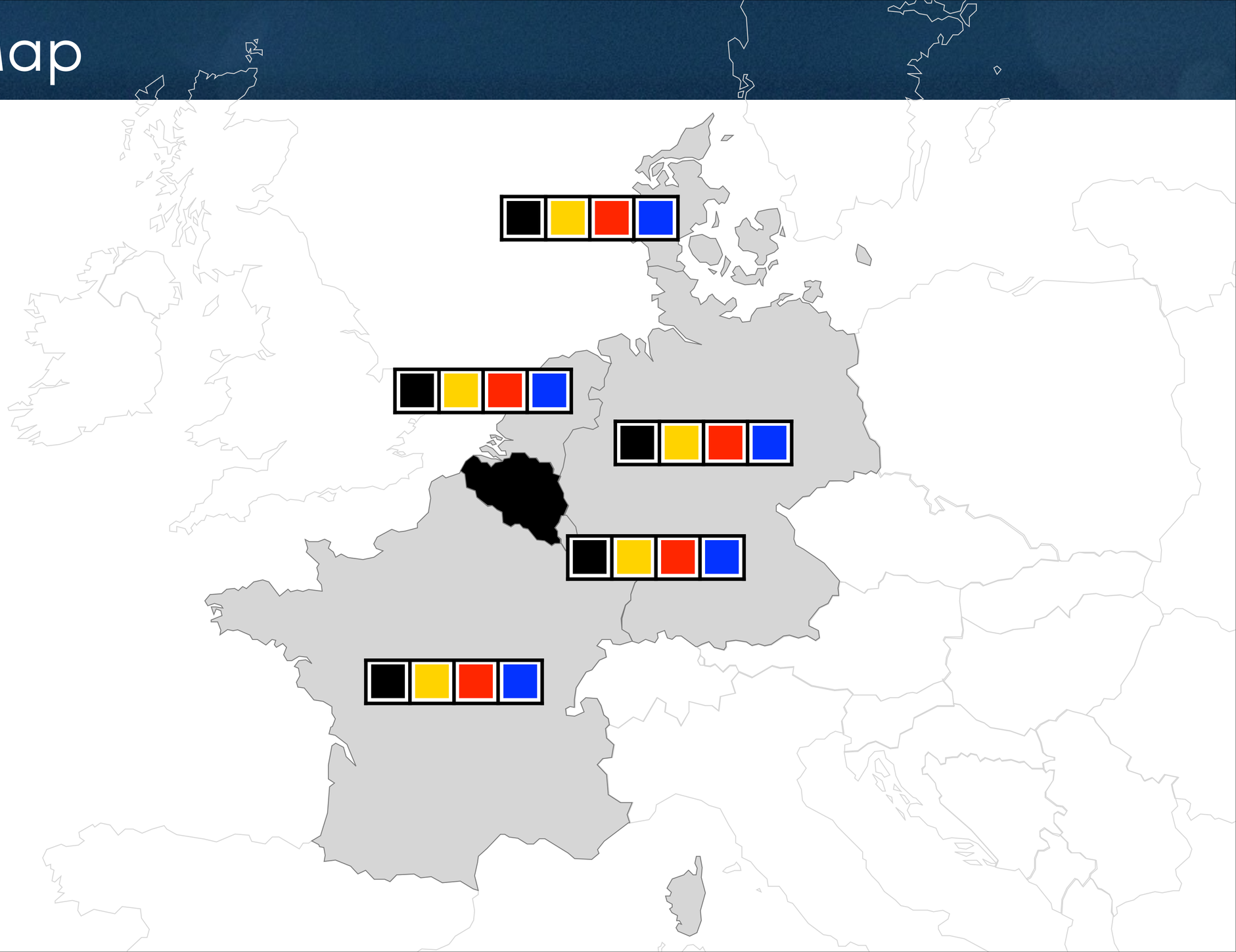
- Computational paradigm
  - use constraints to reduce the set of values that each variable can take
  - make a choice when no more deduction can be performed

- What does this mean for the coloring problem?
  - no value can be removed initially, so the system must make a choice

Coloring a Map



Coloring a Map



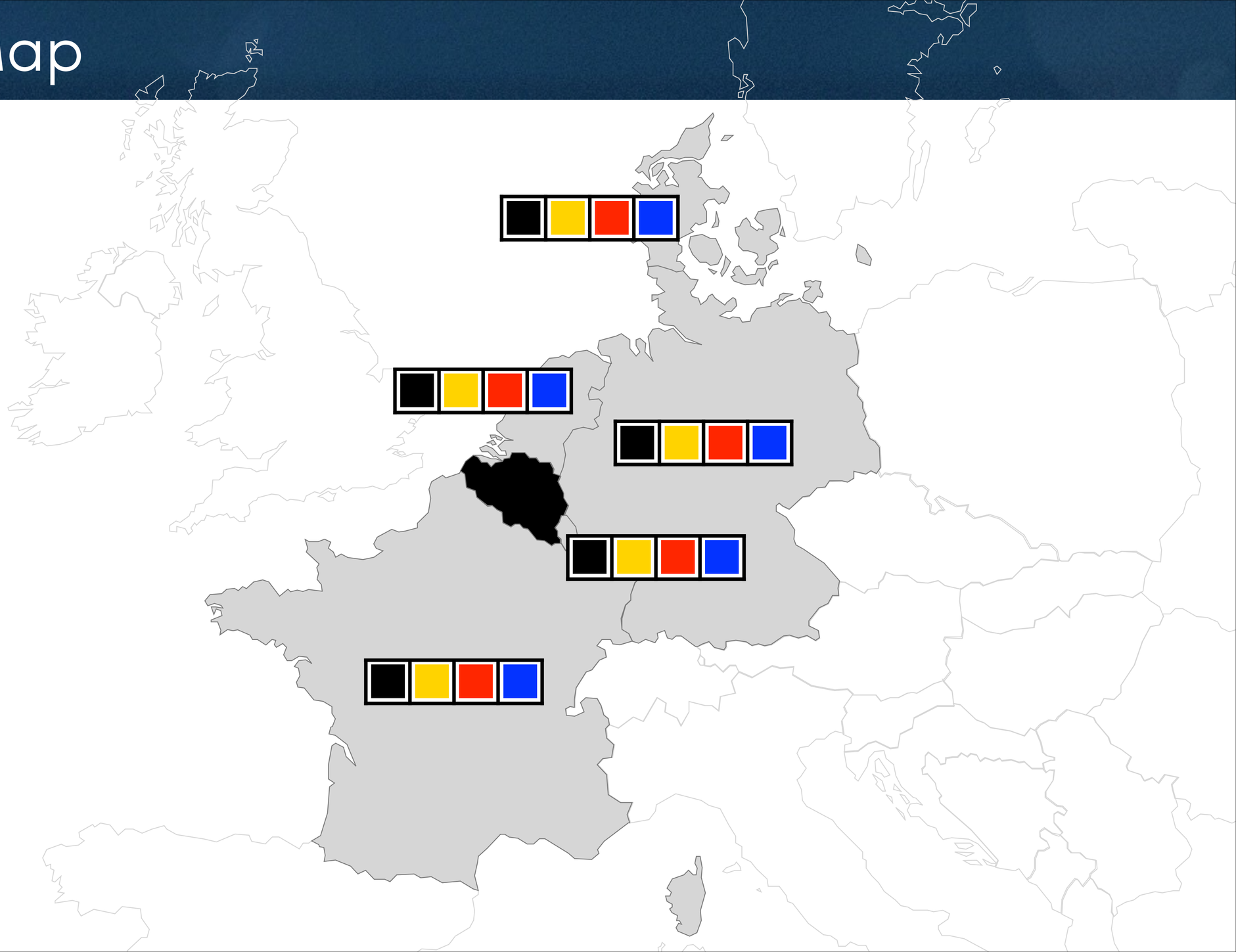
Coloring a Map

```
enum Countries = { Belgium, Denmark, France, Germany,
                   Netherlands, Luxembourg };
enum Colors = { black, yellow, red, blue };
var{Colors} color[Countries];

solve {
 color[Belgium] ≠ color[France];
 color[Belgium] ≠ color[Germany];
 color[Belgium] ≠ color[Netherlands];
 color[Belgium] ≠ color[Luxembourg];
 color[Denmark] ≠ color[Germany];
 color[France] ≠ color[Germany];
 color[France] ≠ color[Luxembourg];
 color[Germany] ≠ color[Netherlands];
 color[Germany] ≠ color[Luxembourg];
}
```

Coloring a Map

```
enum Countries = { Belgium, Denmark, France, Germany,
                   Netherlands, Luxembourg };
enum Colors = { black, yellow, red, blue };
var{Colors} color[Countries];

solve {
 black ≠ color[France];
 black ≠ color[Germany];
 black ≠ color[Netherlands];
 black ≠ color[Luxembourg];
 color[Denmark] ≠ color[Germany];
 color[France] ≠ color[Germany];
 color[France] ≠ color[Luxembourg];
 color[Germany] ≠ color[Netherlands];
 color[Germany] ≠ color[Luxembourg];
}
```

Coloring a Map



Computational Paradigm

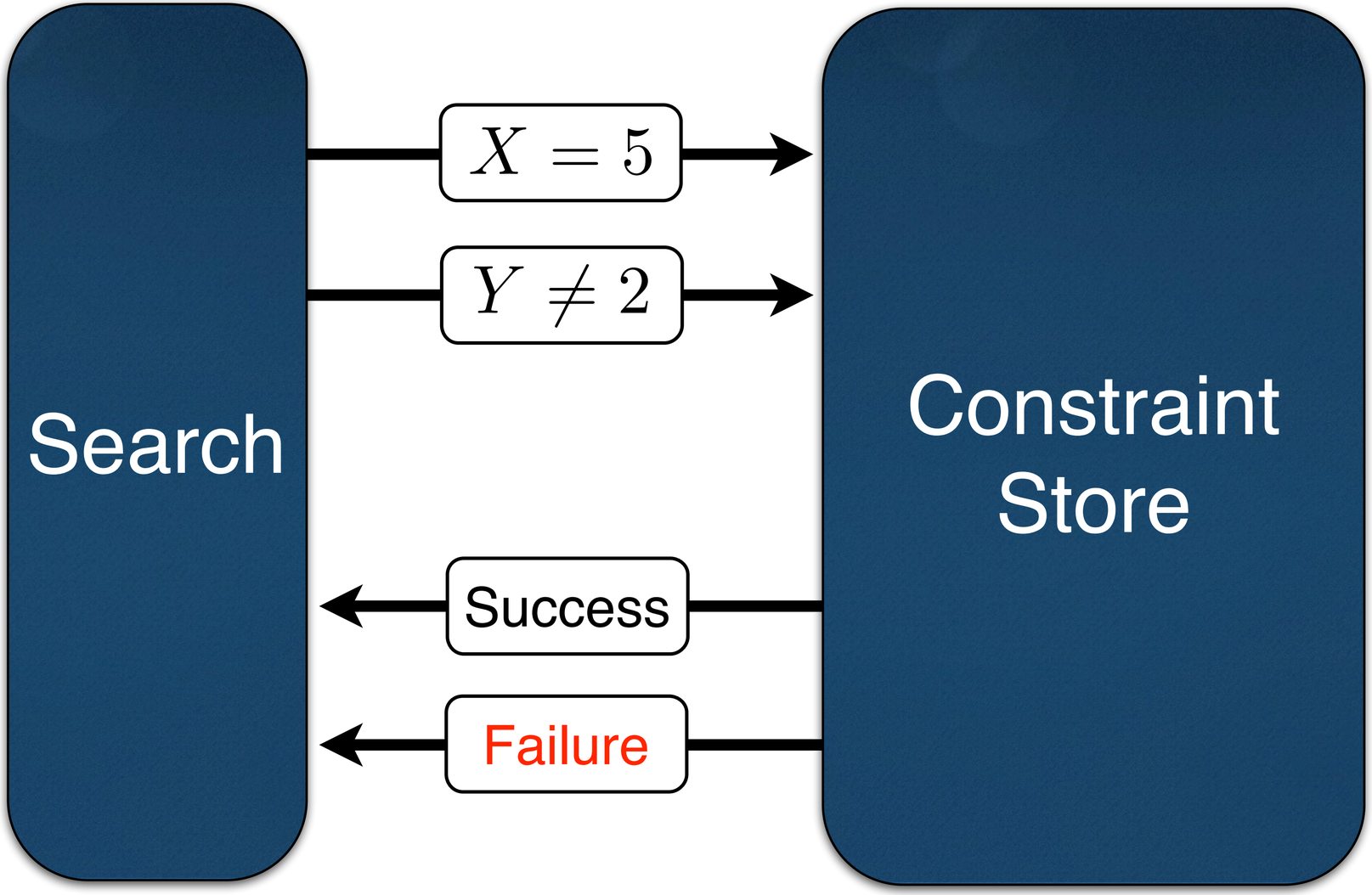
- Branch and prune
  - pruning
    - reduce the search space as much as possible
  - branching
    - decompose the problem into subproblems and explore the subproblems
- Pruning
  - use constraints to remove, from the variable domains, values that cannot belong to any solution
- Branching
  - e.g., try all the possible values of a variable until a solution is found or it can be proven that no solution exists

Computational Paradigm

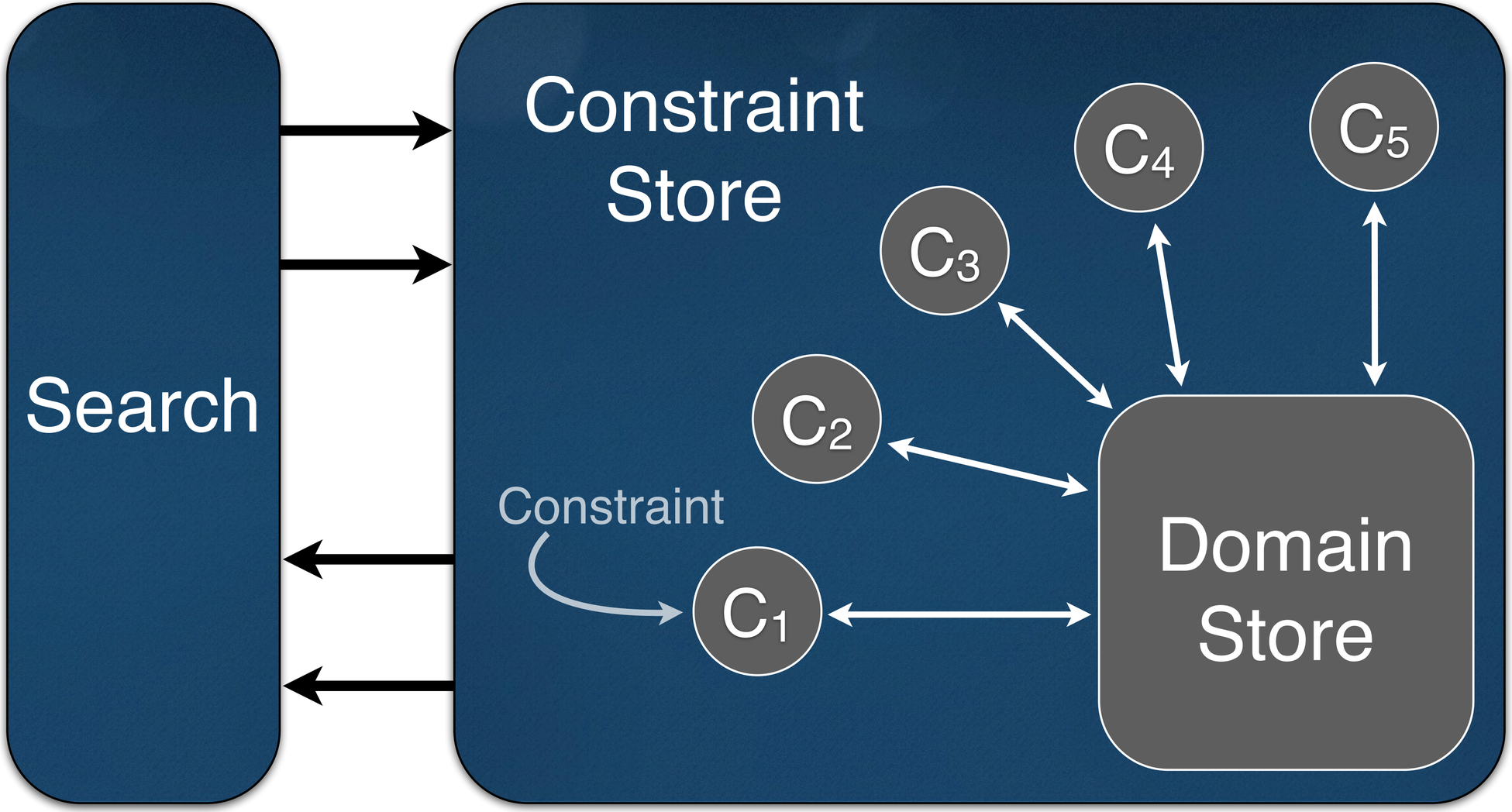
- Complete method, not a heuristic
  - given enough time, it will find a solution to a satisfaction problem
  - given enough time, it will find an optimal solution to an optimization problem

- Focus on feasibility
  - how to use constraints to prune the search space by eliminating values that cannot belong to any solution

Computational Paradigm



Computational Paradigm



Computational Paradigm

- What does a constraint do?
  - feasibility checking
  - pruning
- Feasibility checking
  - a constraint checks if it can be satisfied given the values in the domains of its variables
- Pruning
  - if satisfiable, a constraint determines which values in the domains cannot be part of any solution

Computational Paradigm

- The propagation engine
  - this is the core of any constraint-programming solver
  - a simple (fixpoint) algorithm

```
propagate()
{
  repeat
    select a constraint c;
    if c is infeasible given the domain store then
      return failure;
    else
      apply the pruning algorithm associated with c;
  until no constraint can remove any value from the
  domain of its variables;
  return success;
}
```

Back to the 8-Queens Problem

- What are the decision variables?
  - many possible modelings
    - this is what makes optimization problems interesting :-)
- Here is one modeling
  - associate a decision variable with each column
    - the variable denotes the row of the queens placed in this column
    - no two queens can be placed on the same column so this is valid
- What are the constraints?
  - the queens cannot be placed on the same,
    - row
    - upward diagonal
    - downward diagonal

A Constraint Program for the 8-Queens

- Constraints
  - the queens cannot be placed on the same,
    - row
    - upward diagonal
    - downward diagonal

```
range R = 1..8;
var{int} row[R] in R;
solve {
   forall(i in R,j in R: i < j) {
      row[i] ≠  row[j];
      row[i] ≠  row[j] + (j - i);
      row[i] ≠  row[j] - (j - i);
   }
}
```

A Constraint Program for the 8-Queens

- A simple model for the 8-queens problem
- What happens when the queen in column 1 is assigned the value 1

```
range R = 1..8;
var{int} row[R] in R;
solve {
   forall(i in R,j in R: i < j) {
      row[i] ≠  row[j];
      row[i] ≠  row[j] + (j - i);
      row[i] ≠  row[j] - (j - i);
   }
}
```

```
row[1] ≠  row[2];
...
row[1] ≠  row[8];

row[1] ≠  row[2] + 1;
...
row[1] ≠  row[8] + 7;

row[1] ≠  row[2] - 1;
...
row[1] ≠  row[8] - 7;
```

Computational Paradigm

- What does a constraint do?
  - feasibility checking
  - pruning
- Feasibility checking
  - a constraint checks if it can be satisfied given the values in the domains of its variables
- Pruning
  - if satisfiable, a constraint determines which values in the domains cannot be part of any solution

Computational Paradigm

- Consider two variables X, Y
  - X can take the values 0,1,2
  - Y can take the values 1,2,3

X $\in \{0,\ 1,\ 2\}$

Y $\in \{1,\ 2,\ 3\}$

- The domain of X is the set of values it can take

$$D(X) = \{0,\ 1,\ 2\}$$

- A short hand for ranges of integers

$$[1..5] = \{1,\ 2,\ 3,\ 4,\ 5\}$$

Computational Paradigm

- Consider constraint, X ≠ Y
- Feasibility checking

X $\in \{0,\ 1,\ 2\}$

Y $\in \{1,\ 2,\ 3\}$

$$|D(X) \cup D(Y)| \geq 2$$

$$|\{0,\ 1,\ 2\} \cup \{1,\ 2,\ 3\}| \geq 2$$

$$|\{0,\ 1,\ 2,\ 3\}| \geq 2$$

$$4 \geq 2$$

Computational Paradigm

- Consider constraint, X ≠ Y
- Pruning

X $\in \{\cancel{0}, 1, \cancel{2}\}$

Y $\in \{\cancel{1}, 2, 3\}$

$$D(X) = \{1\}$$
$$\Rightarrow D(Y) \setminus \{1\}$$

$$D(Y) = \{2\}$$
$$\Rightarrow D(X) \setminus \{2\}$$

Until Next Time

Citations

Map_of_USA_four_colours.svg (http://commons.wikimedia.org/wiki/File:Map_of_United_States_vivid_colors_shown.png) of the modification : Derfel73) Dbenbenn derivative work: Tomwsulcer (Map_of_USA_four_colours.svg) [CC-BY-SA-3.0 (http://creativecommons.org/licenses/by-sa/3.0)], via Wikimedia Commons

Discrete Optimization

Constraint Programming: Part II

Goals of the Lecture

- Illustrating more complex constraint propagation
- Showing that constraints have dedicated algorithms

Computational Paradigm

- Branch and prune
  - pruning
    - reduce the search space as much as possible
  - branching
    - decompose the problem into subproblems and explore the subproblems

Computational Paradigm

- Branch and prune
  - pruning
    - reduce the search space as much as possible
  - branching
    - decompose the problem into subproblems and explore the subproblems
- Pruning
  - use constraints to remove, from the variable domains, values that cannot belong to any solution

Computational Paradigm

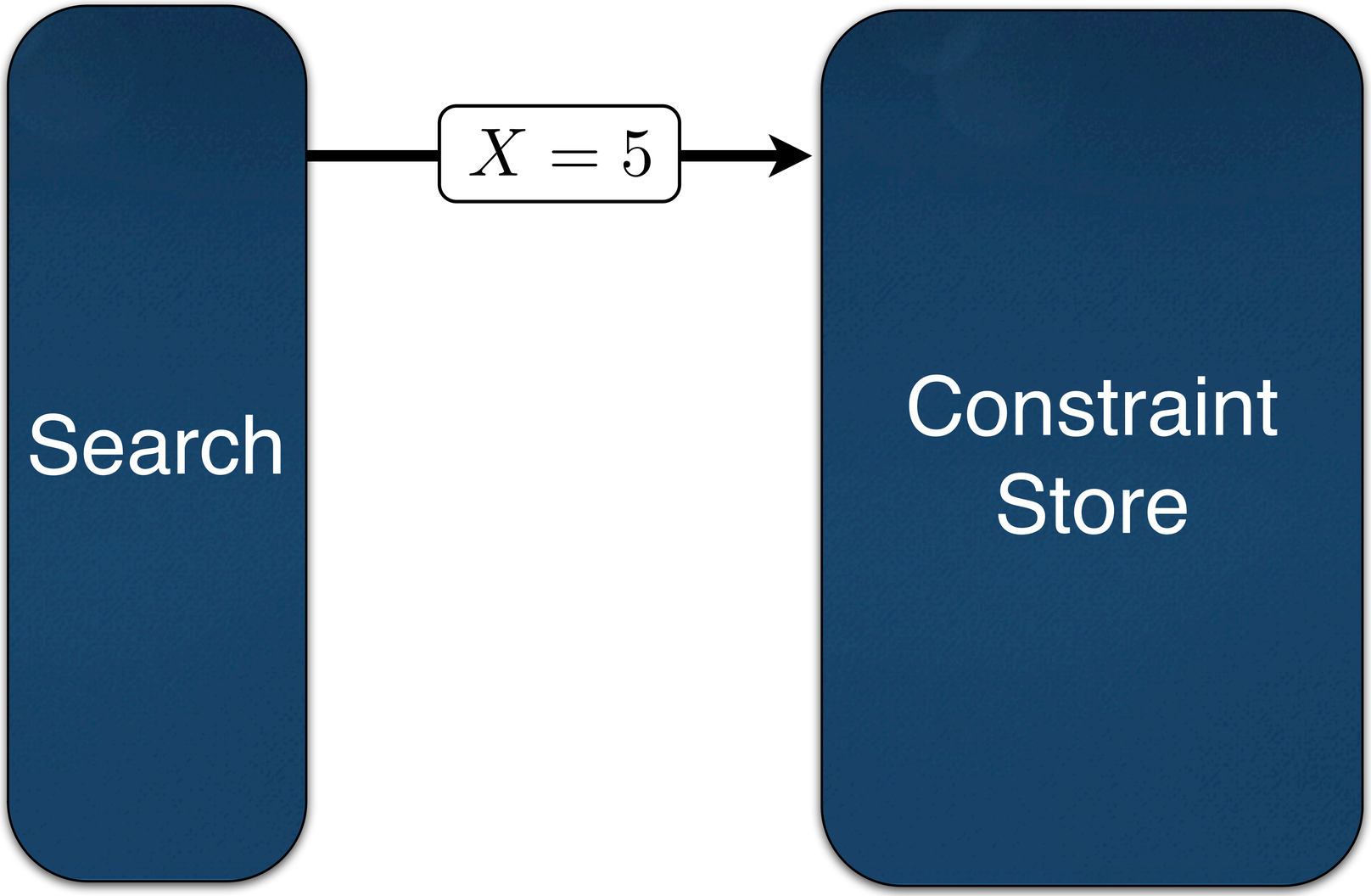
- Branch and prune
  - pruning
    - reduce the search space as much as possible
  - branching
    - decompose the problem into subproblems and explore the subproblems
- Pruning
  - use constraints to remove, from the variable domains, values that cannot belong to any solution
- Branching
  - e.g., try all the possible values of a variable until a solution is found or it can be proven that no solution exists

Computational Paradigm

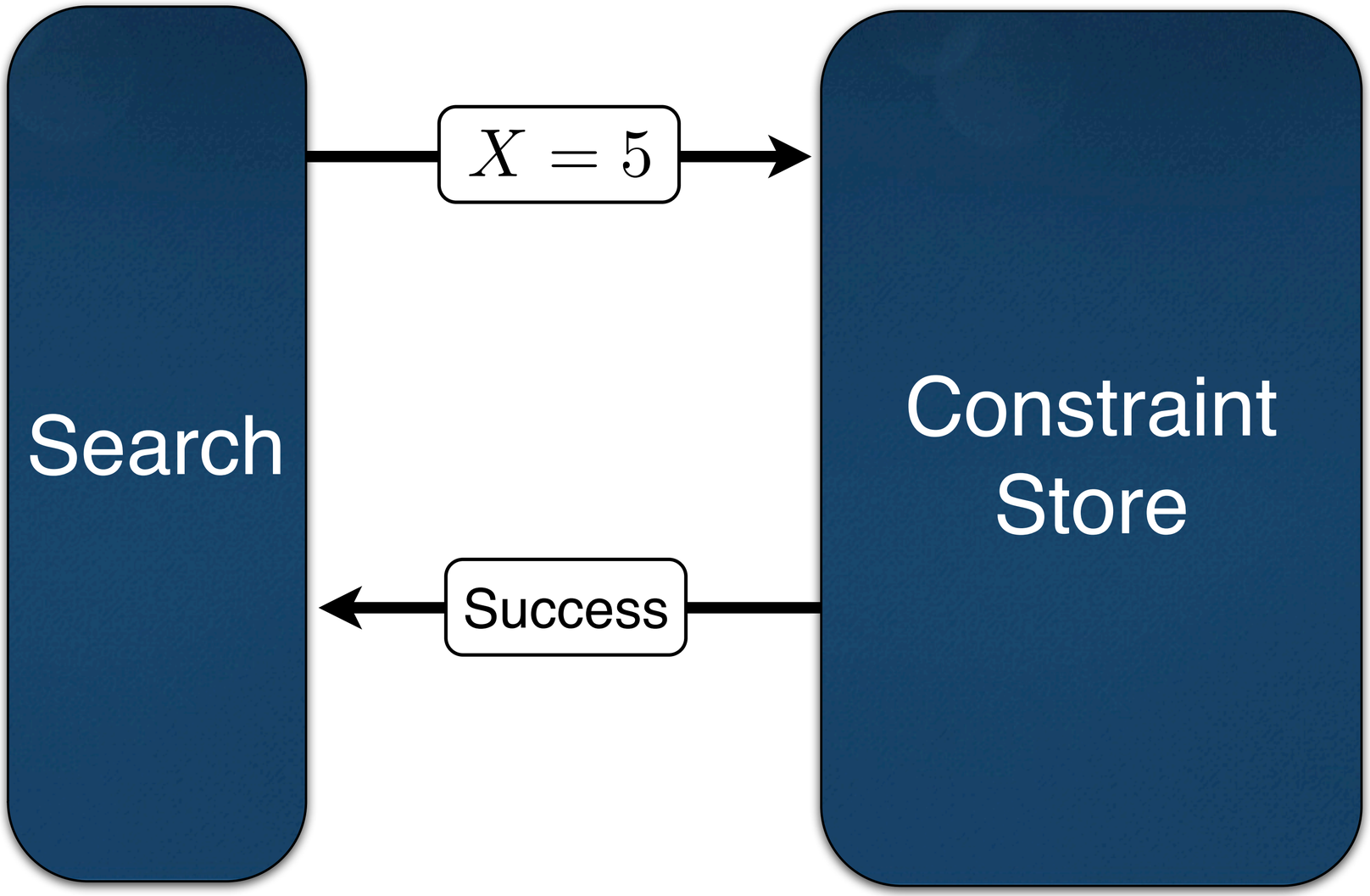
Search

Constraint Store

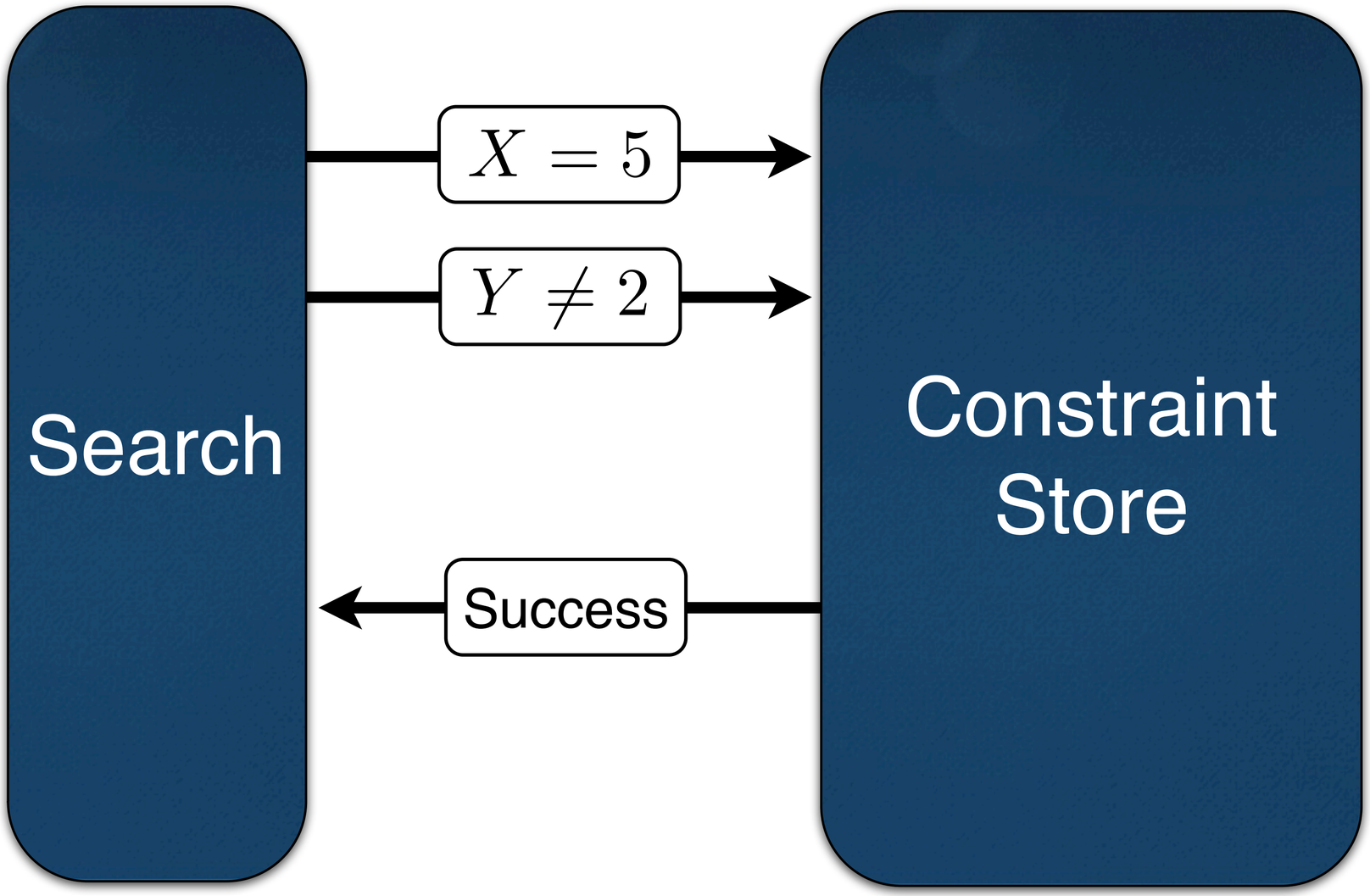
Computational Paradigm



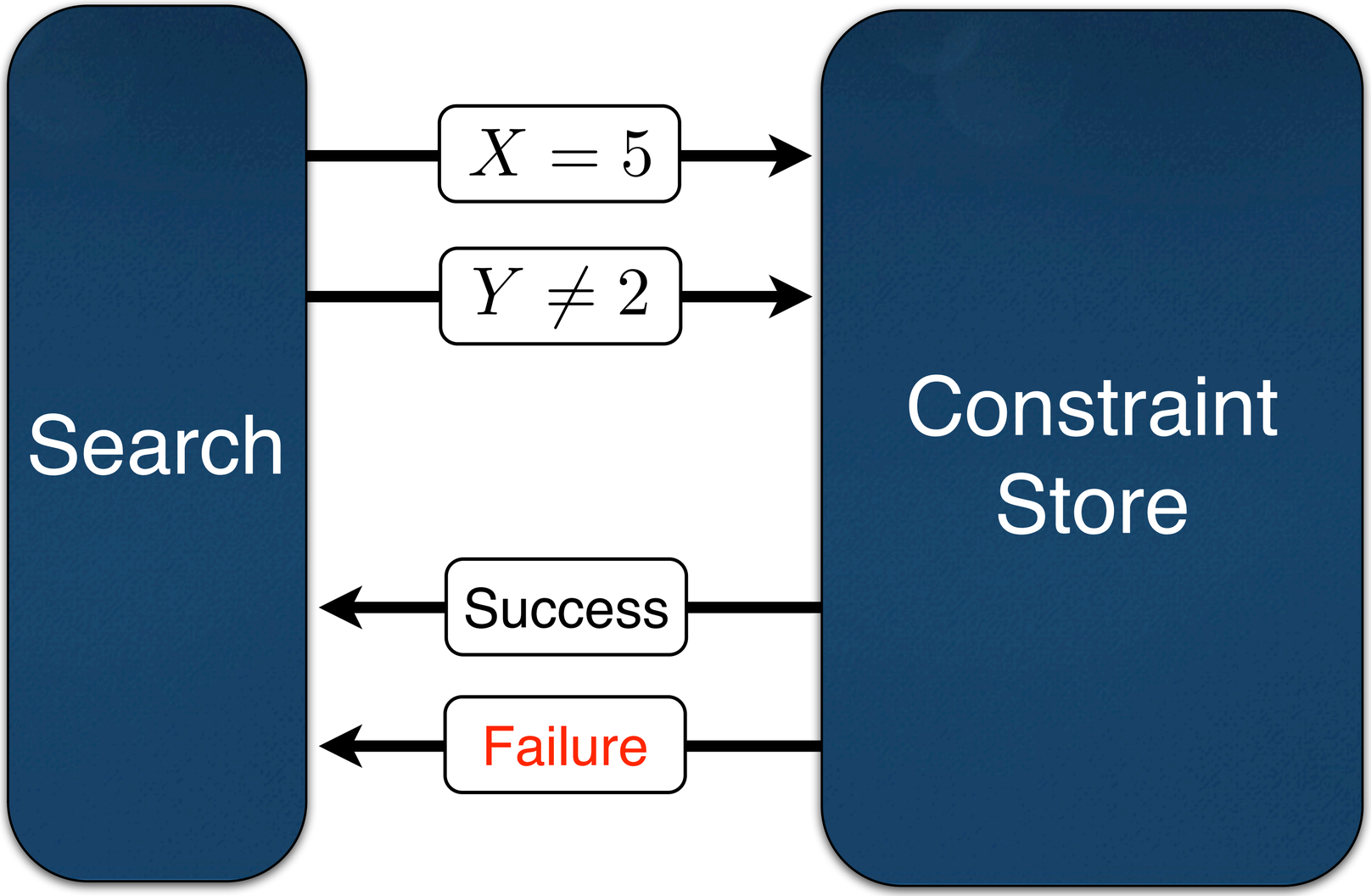
Computational Paradigm



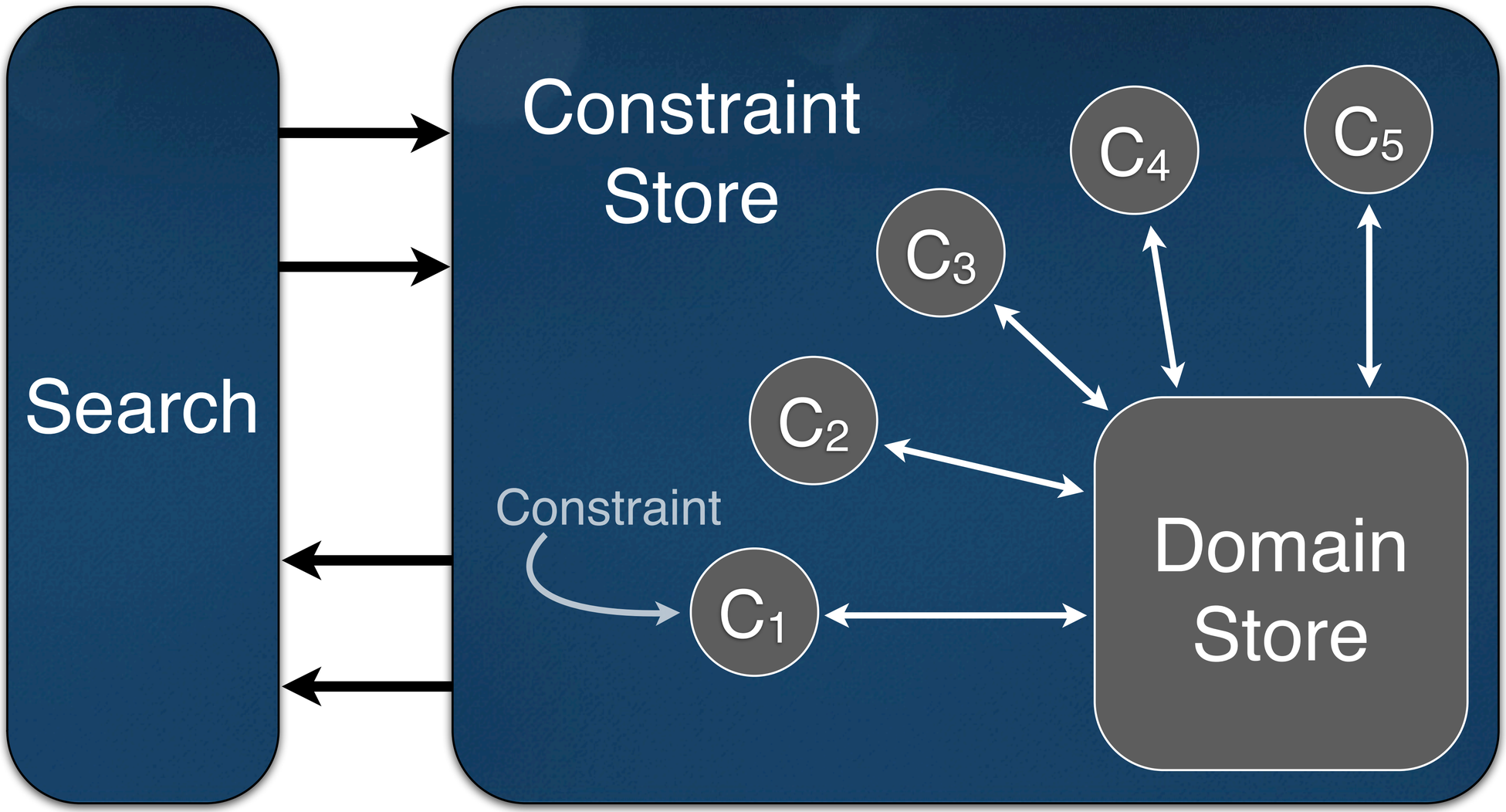
Computational Paradigm



Computational Paradigm



Computational Paradigm



Computational Paradigm

- What does a constraint do?
  - feasibility checking
  - pruning

Computational Paradigm

- What does a constraint do?
  - feasibility checking
  - pruning
- Feasibility checking
  - can a constraint be satisfied given the values in the domains of its variables

Computational Paradigm

- What does a constraint do?
  - feasibility checking
  - pruning
- Feasibility checking
  - can a constraint be satisfied given the values in the domains of its variables
- Pruning
  - if satisfiable, determine which values in the domains cannot be part of any solution

Computational Paradigm

- What does a constraint do?
  - feasibility checking
  - pruning
- Feasibility checking
  - can a constraint be satisfied given the values in the domains of its variables
- Pruning
  - if satisfiable, determine which values in the domains cannot be part of any solution
- The algorithms use dedicated algorithms for each constraint
  - they exploit the structure and properties of the constraint

Send More Money

- Specification
  - assign different digits to letters to satisfy the addition

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

Send More Money

- Specification
  - assign different digits to letters to satisfy the addition

- Model to illustrate constraint propagation
  - no claim that this is a good model: it is not

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

Send More Money

- Basic modeling
  - add carries explicitly like in kindergarten

	C_4	C_3	C_2	C_1
		S	E	N	D
+		M	O	R	E
=	M	O	N	E	Y

Send More Money

- Basic modeling
  - add carries explicitly like in kindergarten

- What are the decision variables
  - there is a variable for each letter to denote the value of the letters
  - there is a variable for each carry

```
  C4  C3  C2  C1
      S   E   N   D
  +   M   O   R   E
 ------------------
= M   O   N   E   Y
```

Send More Money

```
enum Letters = { S, E, N, D, M, O, R, Y};
range Digits = 0..9;
var{int} value[Letters] in Digits;
var{int} carry[1..4] in 0..1;

solve {
  forall(i in Letters, j in Letters: i < j)
     value[i] ≠ value[j];
  value[S] ≠ 0;
  value[M] ≠ 0;
  carry[4]                          = value[M];
  carry[3] + value[S] + value[M] = value[O] + 10 * carry[4];
  carry[2] + value[E] + value[O] = value[N] + 10 * carry[3];
  carry[1] + value[N] + value[R] = value[E] + 10 * carry[2];
             value[D] + value[E] = value[Y] + 10 * carry[1];
}
```

C_4	C_3	C_2	C_1	
	S	E	N	D
+	M	O	R	E
= M	O	N	E	Y

What is the Search Space?

	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										

What is the Search Space?

	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										

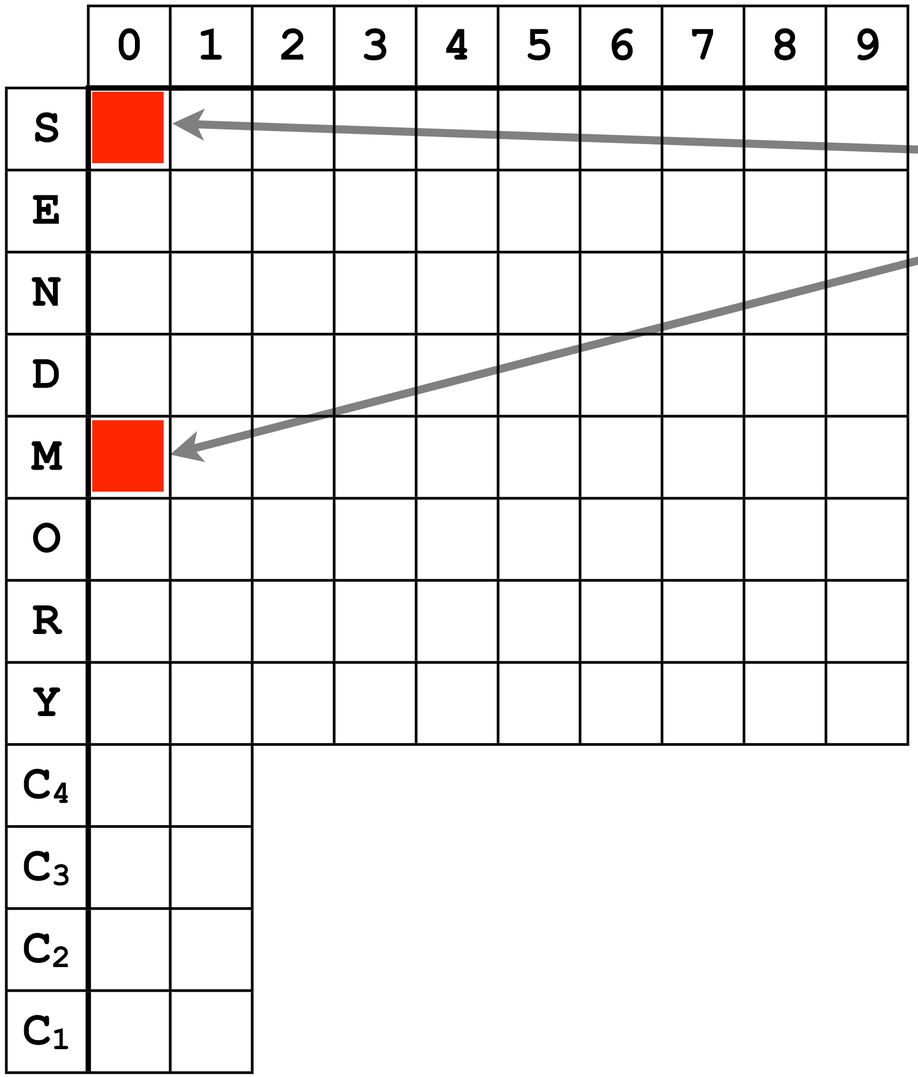
```
forall(i in Letters, j in Letters: i < j)
   value[i] ≠ value[j];
value[S] ≠ 0;
value[M] ≠ 0;
carry[4]                          = value[M];
```

What is the Search Space?

	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										

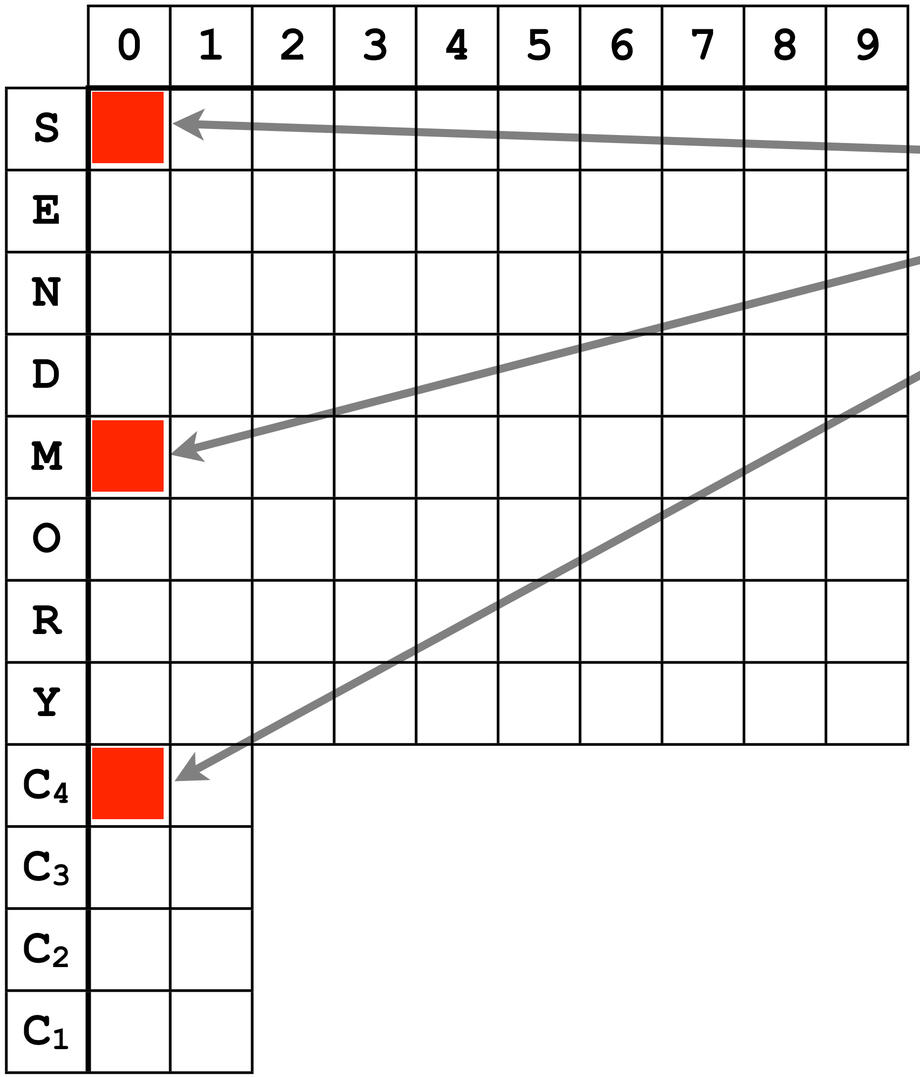
```
forall(i in Letters, j in Letters: i < j)
   value[i] ≠ value[j];
value[S] ≠ 0;
value[M] ≠ 0;
carry[4]                          = value[M];
```

What is the Search Space?



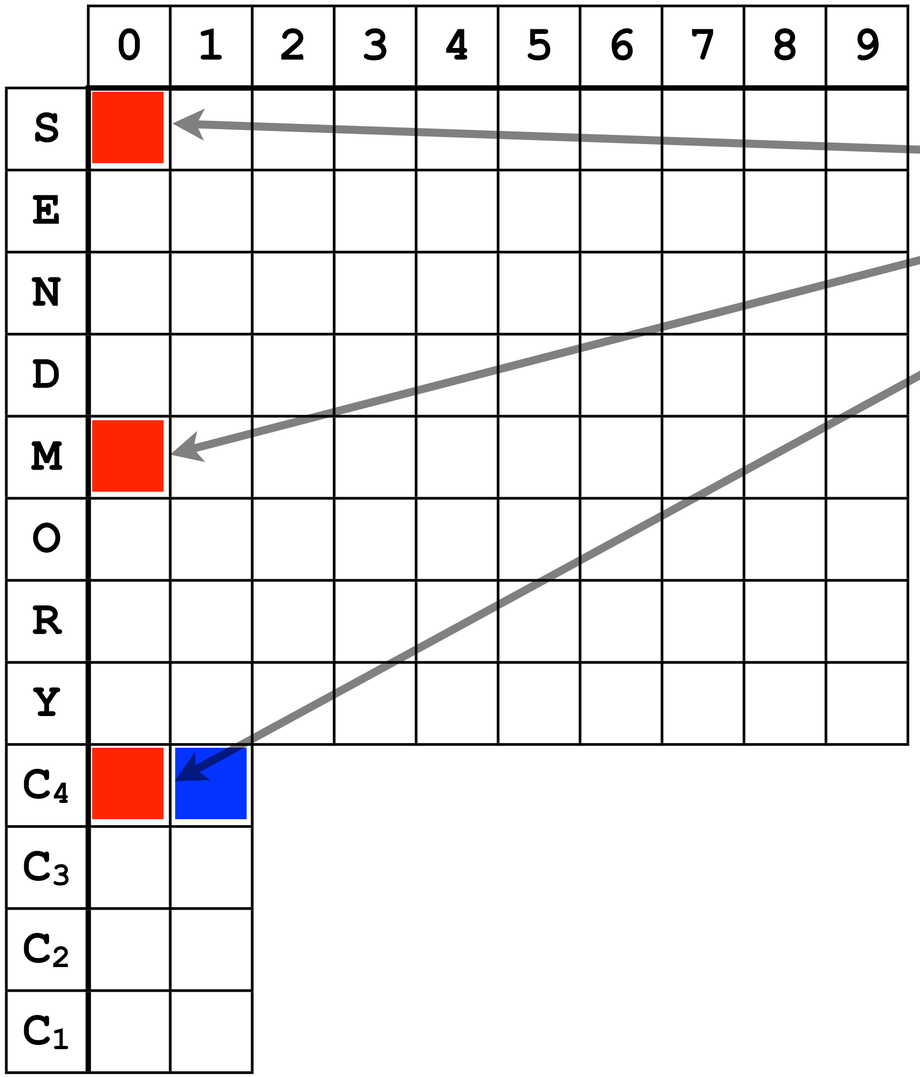

```
forall(i in Letters, j in Letters: i < j)
   value[i] ≠ value[j];
value[S] ≠ 0;
value[M] ≠ 0;
carry[4]                          = value[M];
```

What is the Search Space?



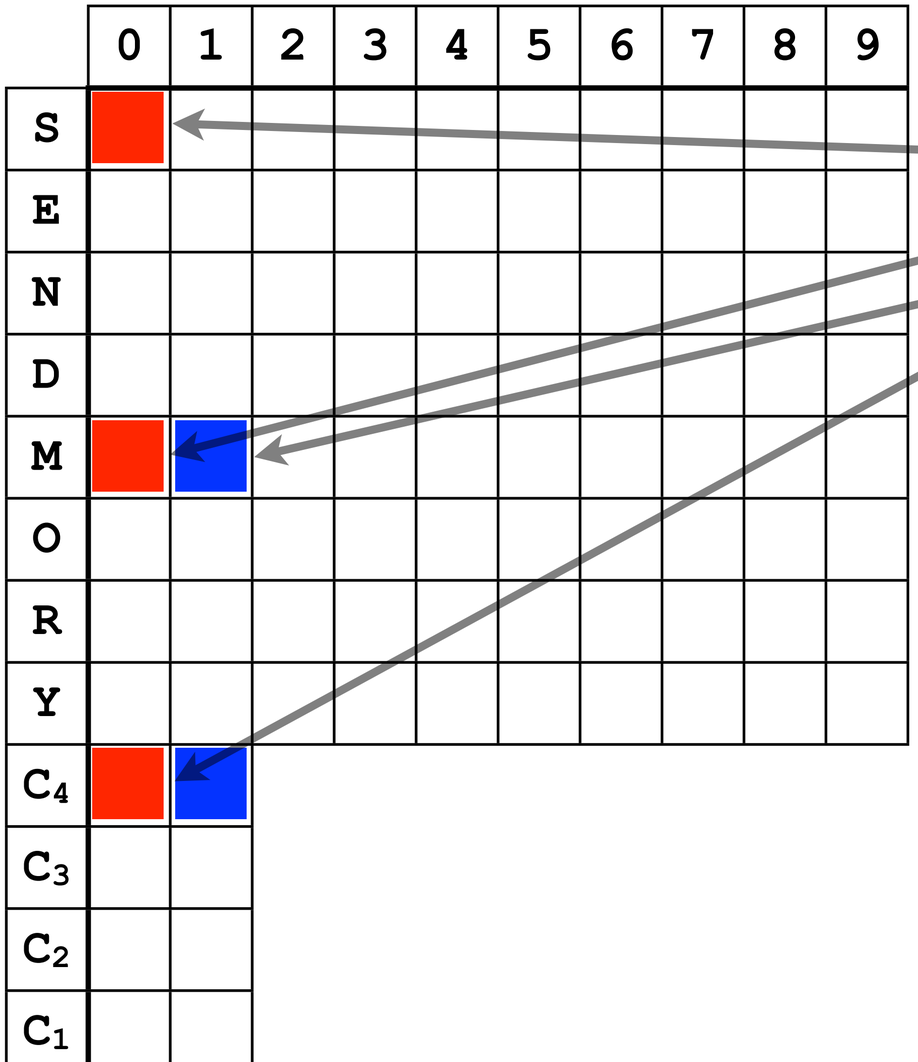

```
forall(i in Letters, j in Letters: i < j)
   value[i] ≠ value[j];
value[S] ≠ 0;
value[M] ≠ 0;
carry[4]                          = value[M];
```

What is the Search Space?



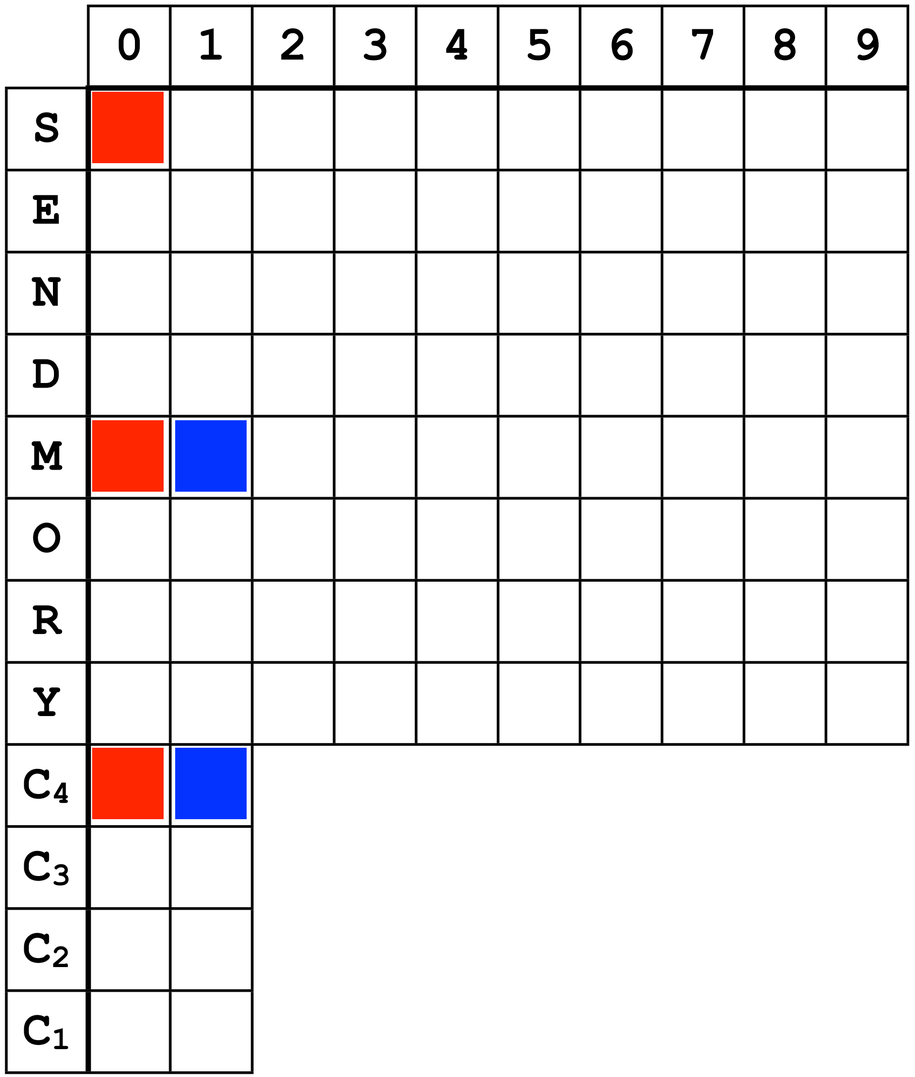

```
forall(i in Letters, j in Letters: i < j)
   value[i] ≠ value[j];
value[S] ≠ 0;
value[M] ≠ 0;
carry[4]                          = value[M];
```

What is the Search Space?



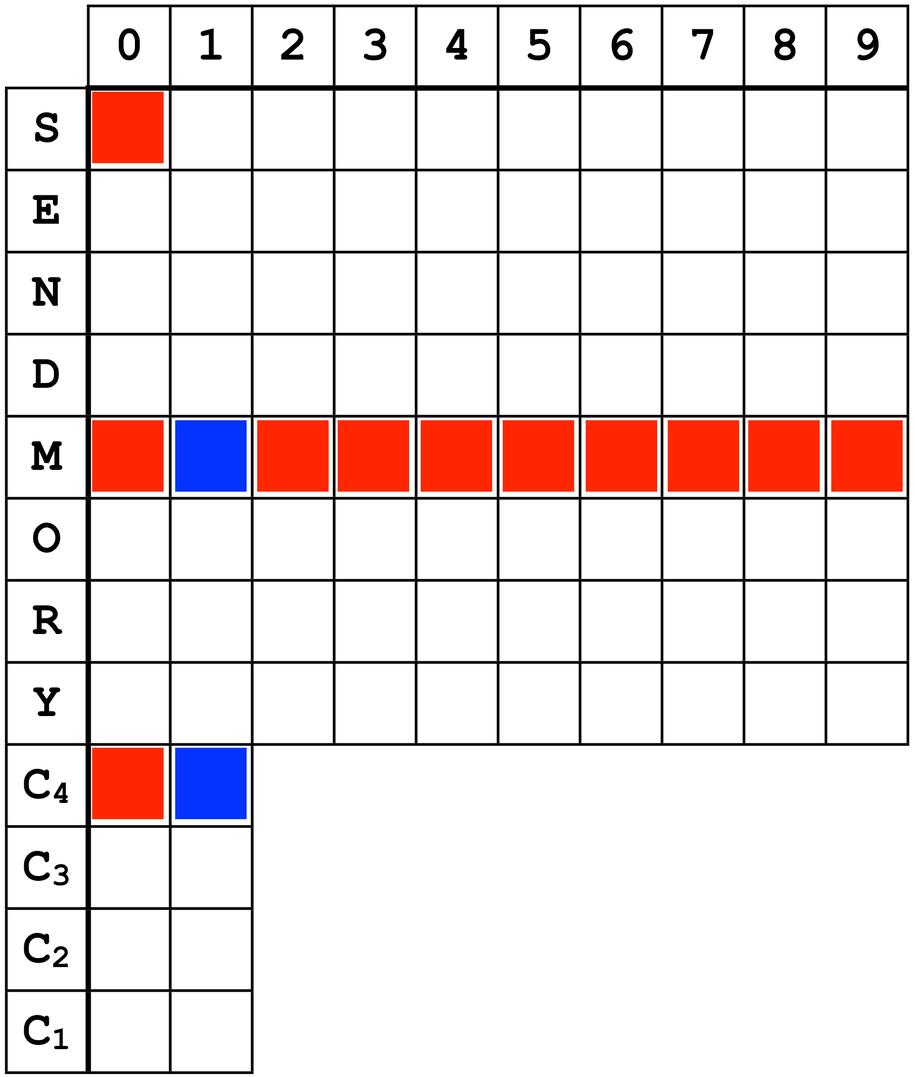

```
forall(i in Letters, j in Letters: i < j)
   value[i] ≠ value[j];
value[S] ≠ 0;
value[M] ≠ 0;
carry[4]                          = value[M];
```

What is the Search Space?



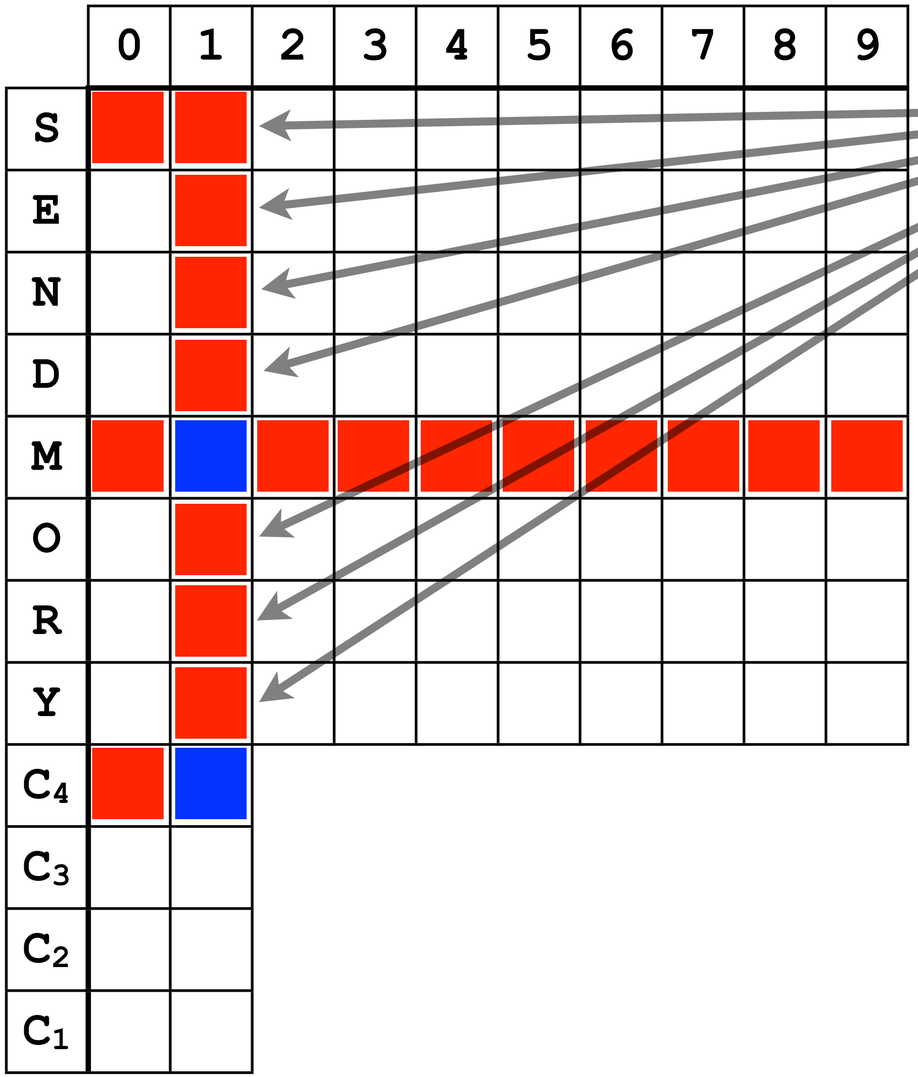

```
forall(i in Letters, j in Letters: i < j)
   value[i] ≠ value[j];
value[S] ≠ 0;
value[M] ≠ 0;
carry[4]                          = value[M];
```

What is the Search Space?



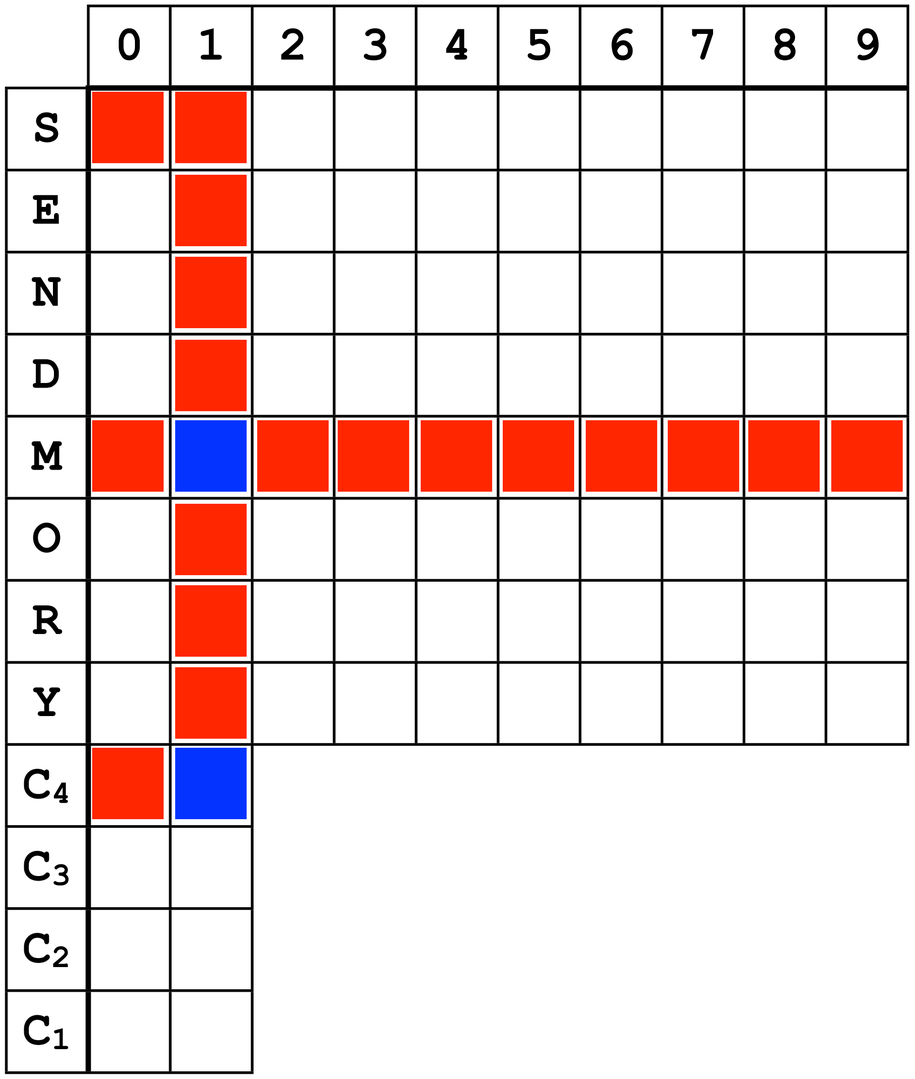

```
forall(i in Letters, j in Letters: i < j)
   value[i] ≠ value[j];
value[S] ≠ 0;
value[M] ≠ 0;
carry[4]                          = value[M];
```

What is the Search Space?



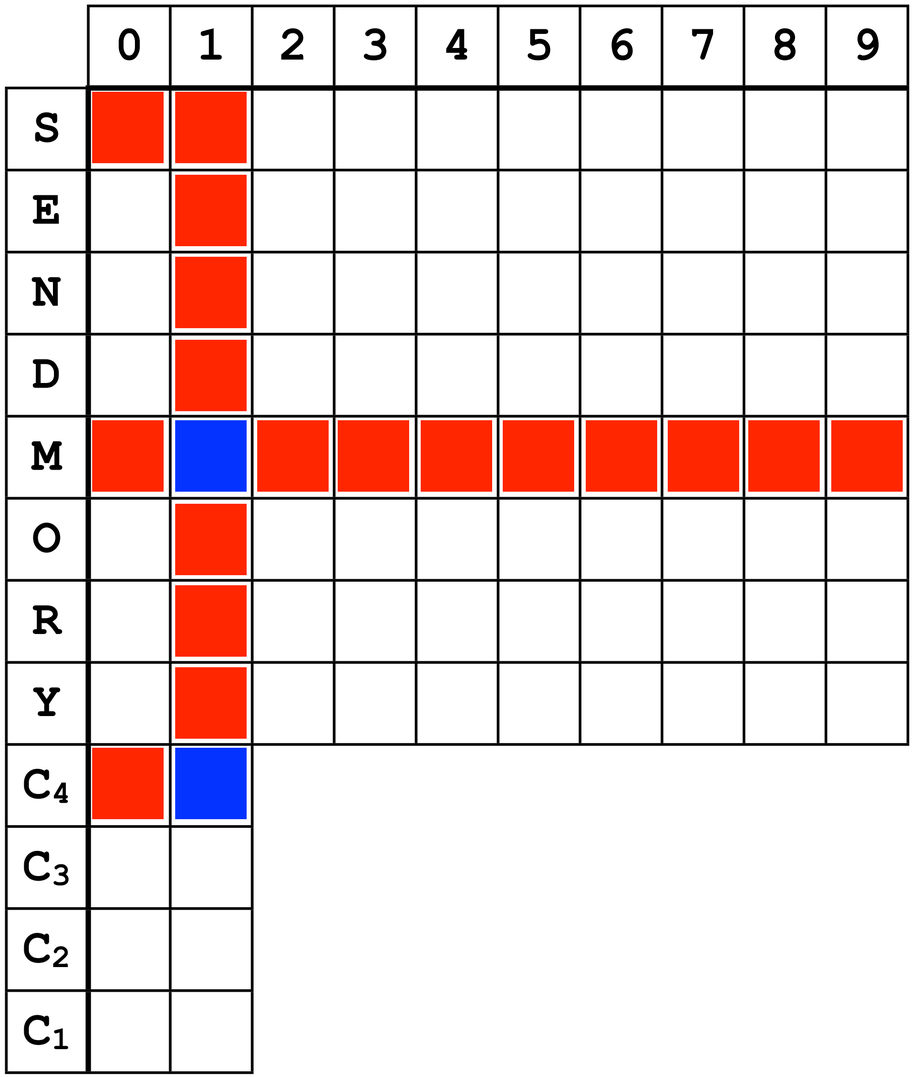

```
forall(i in Letters, j in Letters: i < j)
  value[i] ≠ value[j];
value[S] ≠ 0;
value[M] ≠ 0;
carry[4]                          = value[M];
```

What is the Search Space?

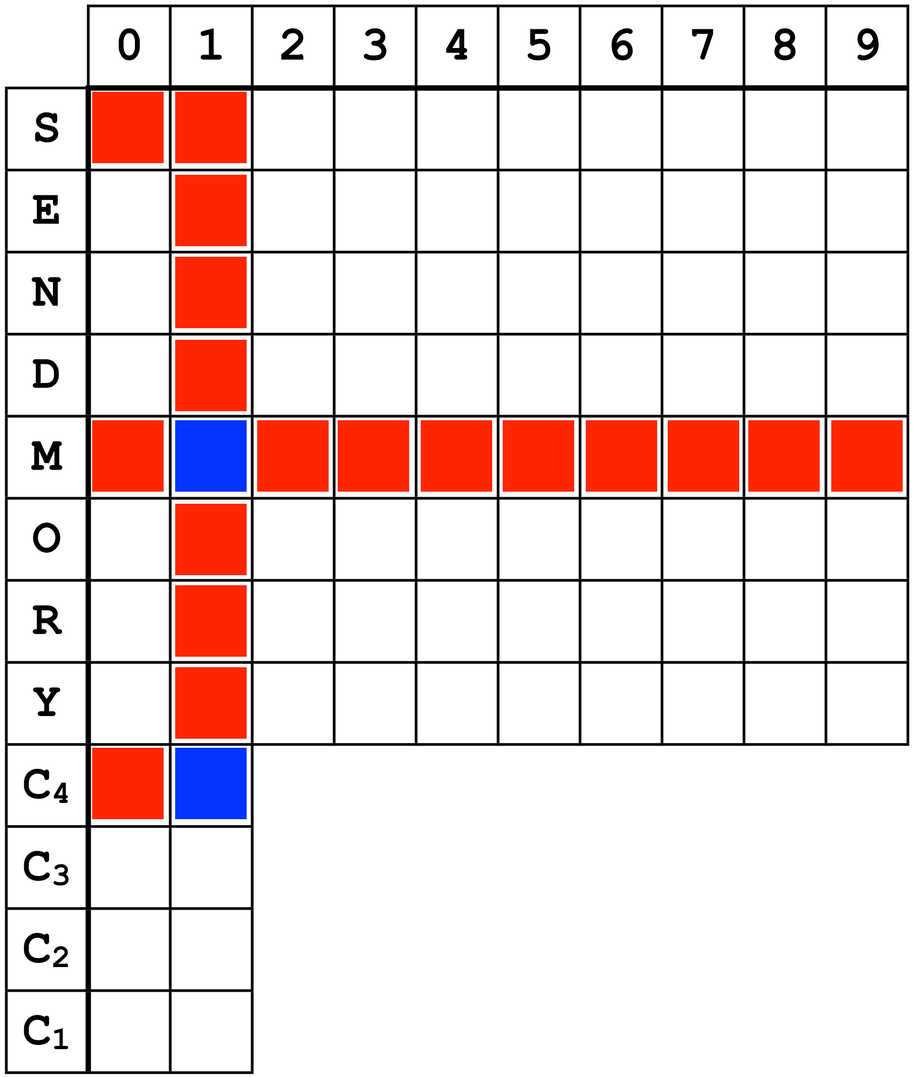



```
forall(i in Letters, j in Letters: i < j)
   value[i] ≠ value[j];
value[S] ≠ 0;
value[M] ≠ 0;
carry[4]                          = value[M];
```

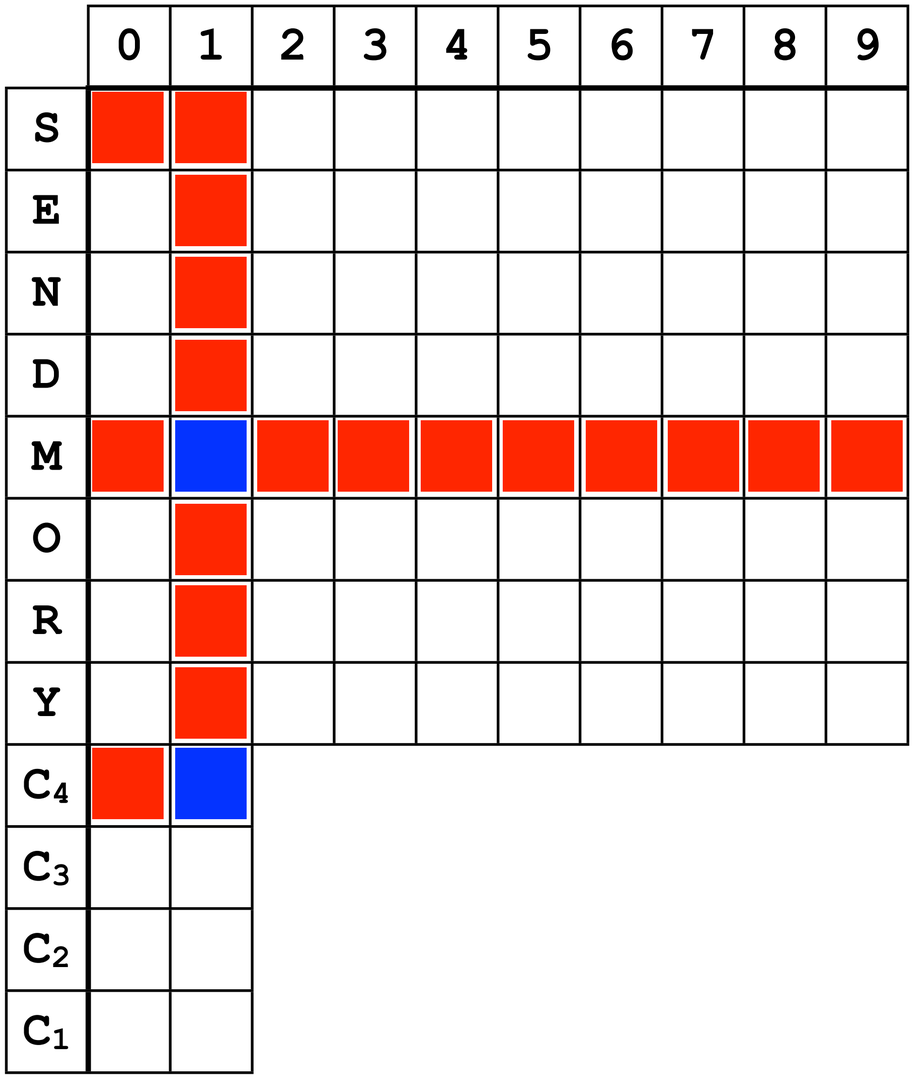
What is the Search Space?



What is the Search Space?

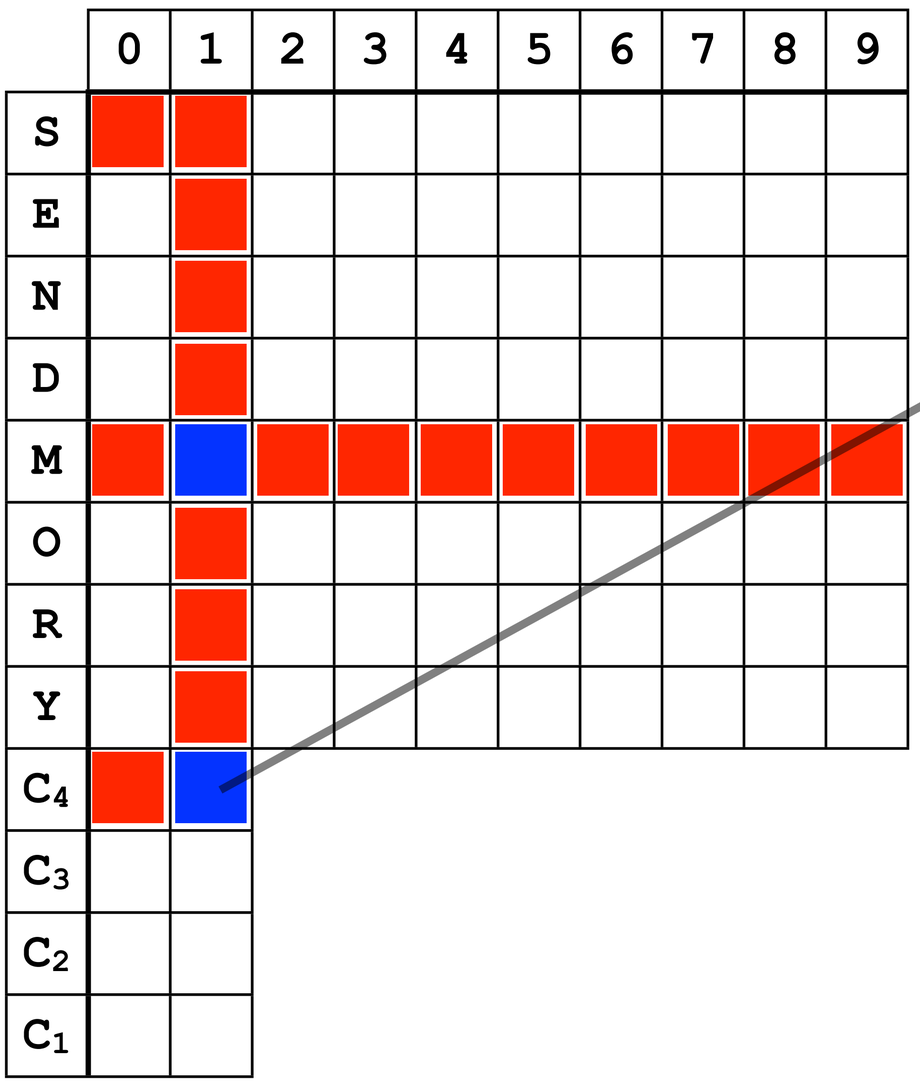


What is the Search Space?



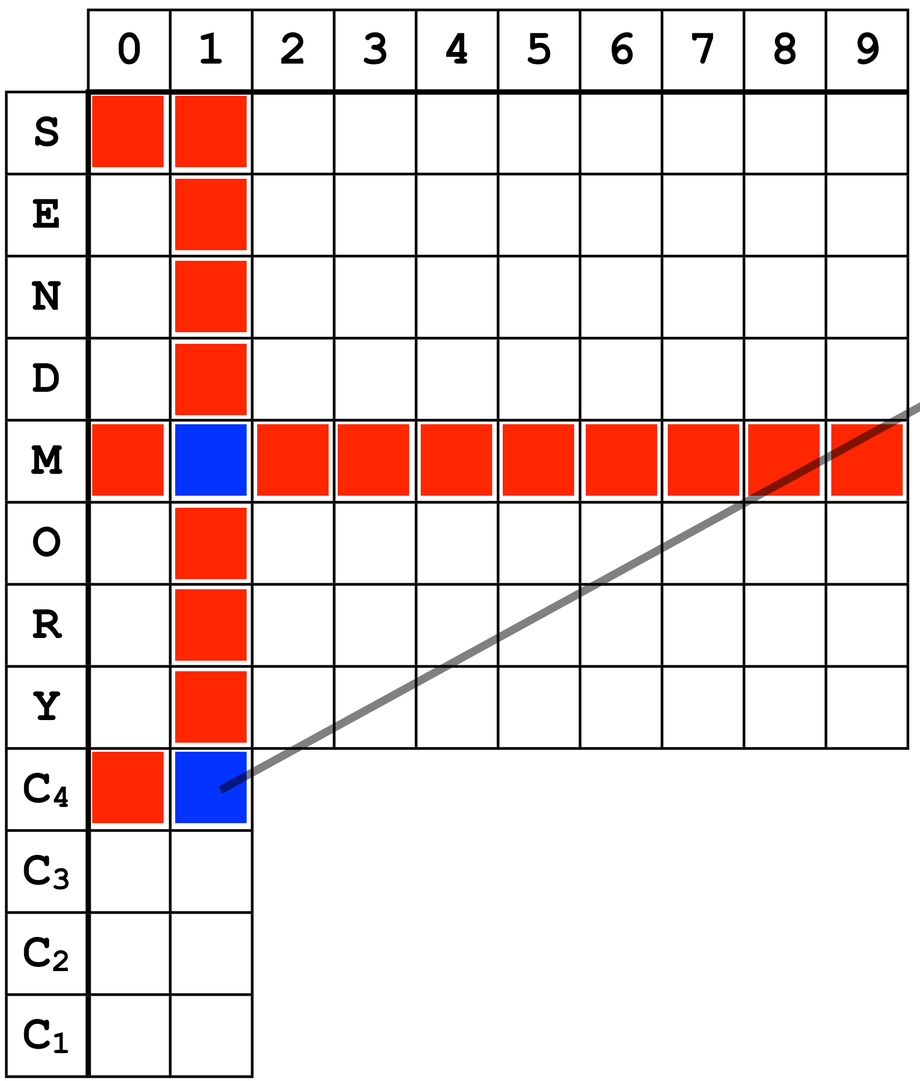

```
carry[3] + value[S] + value[M] = value[O] + 10 * carry[4];
```

What is the Search Space?




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

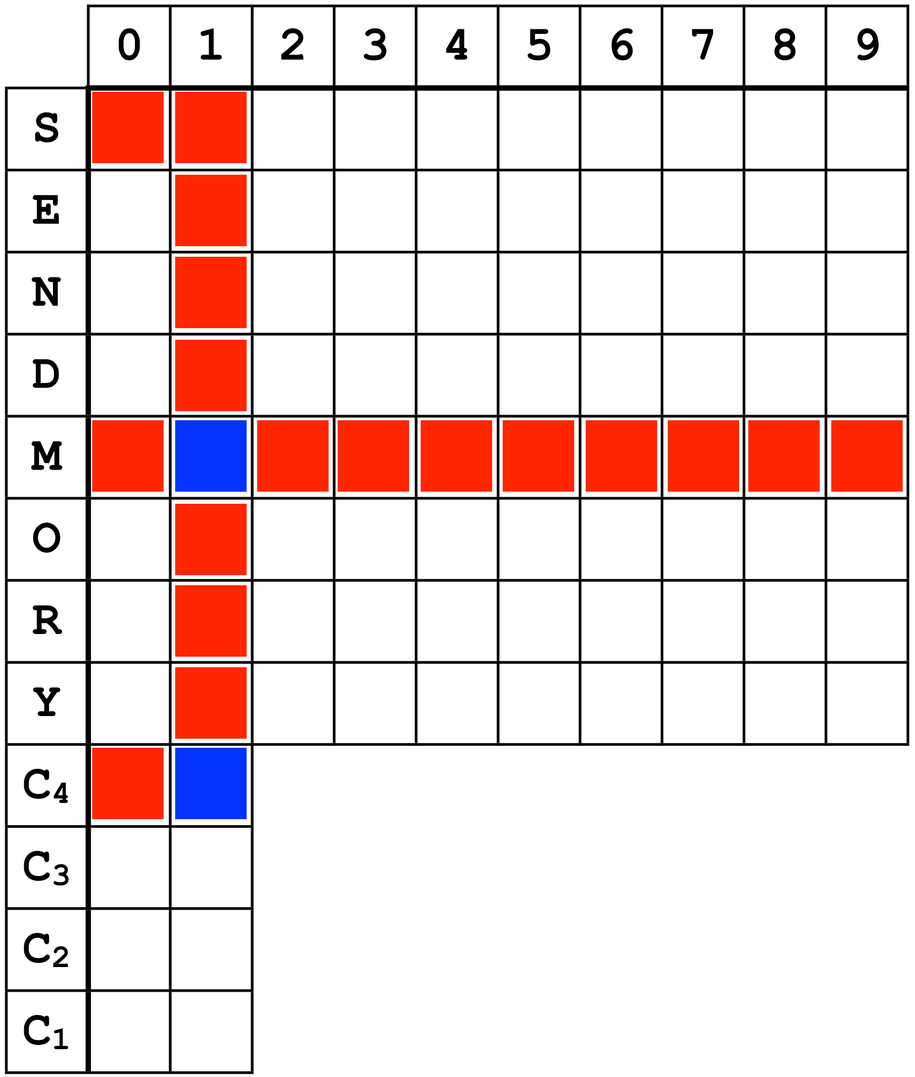
What is the Search Space?




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

```
carry[3] + value[S] + 1        = value[O] + 10 * carry[4];
```

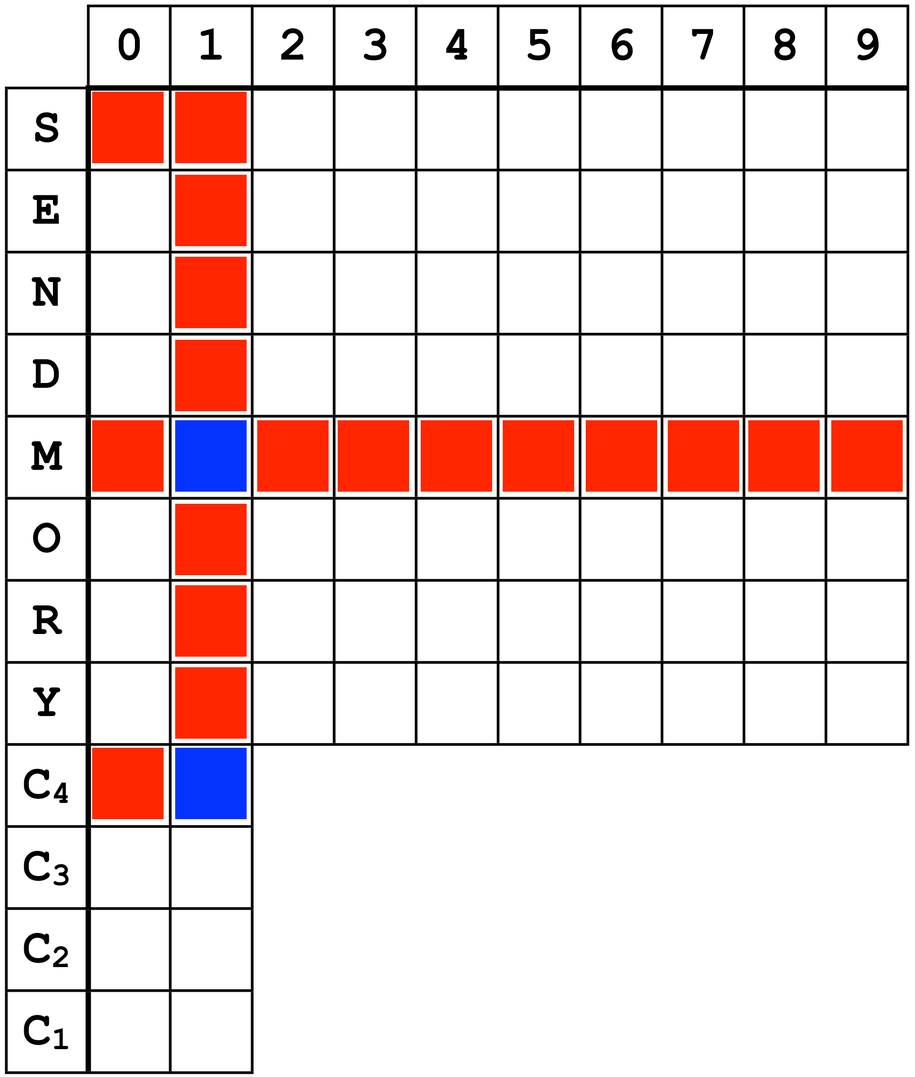
What is the Search Space?




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

```
carry[3] + value[S] + 1        = value[O] + 10 * carry[4];
```

What is the Search Space?

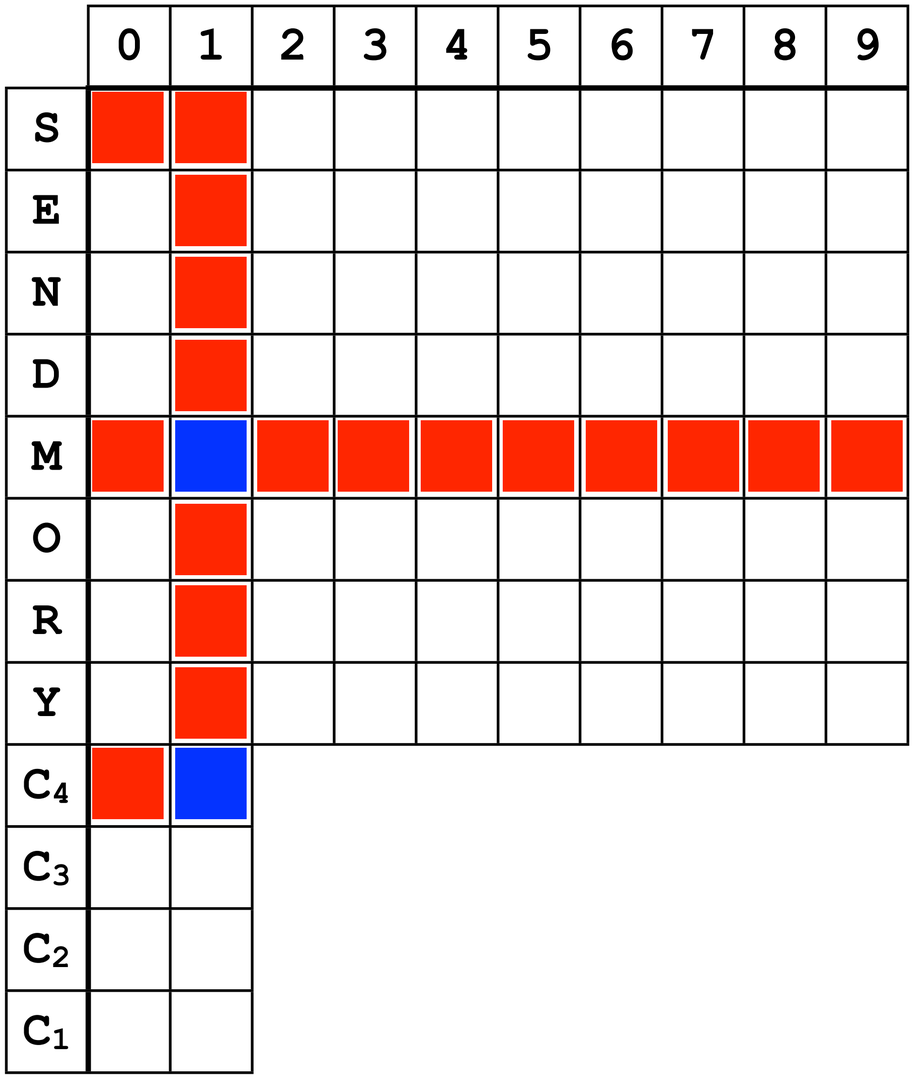



```
carry[3] + value[S] + value[M] = value[O] + 10 * carry[4];
```

```
carry[3] + value[S] + 1        = value[O] + 10 * carry[4];
```

$\in [3..11]$

What is the Search Space?



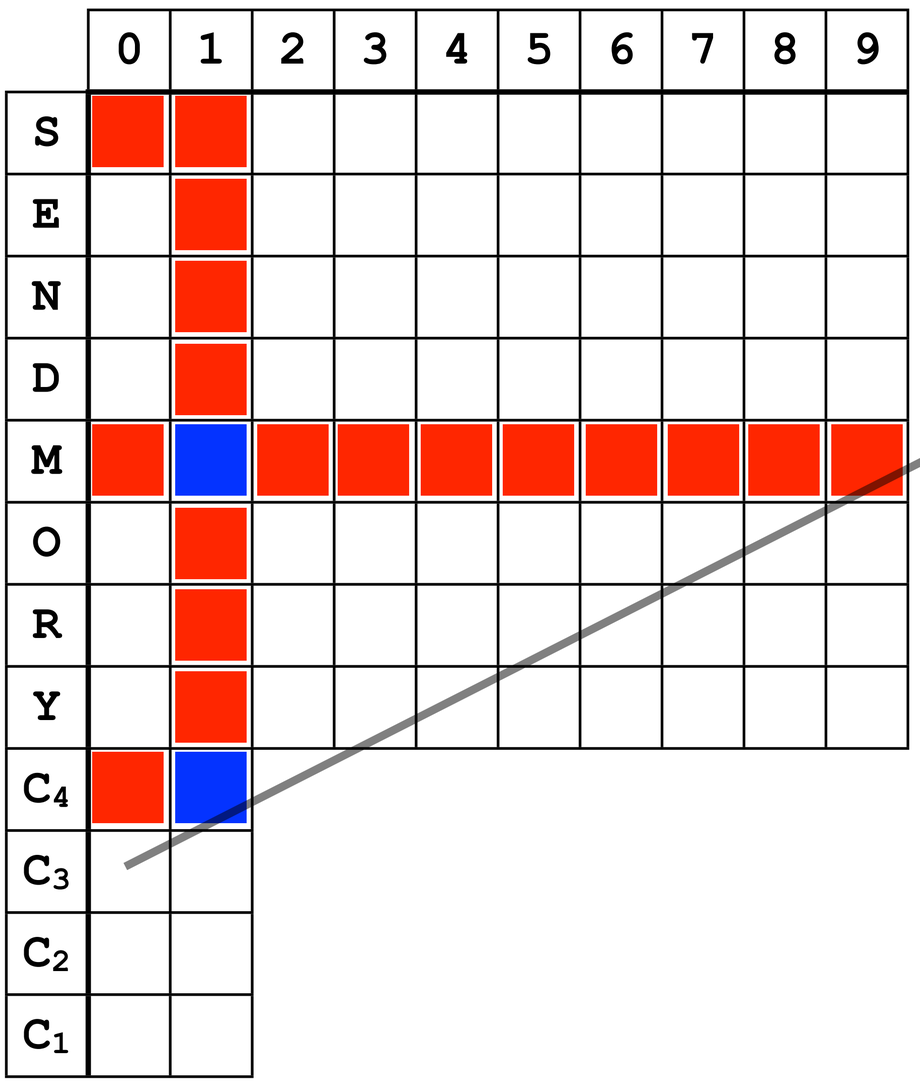

```
carry[3] + value[S] + value[M] = value[O] + 10 * carry[4];
```

```
carry[3] + value[S] + 1        = value[O] + 10 * carry[4];
```

$\in [3..11]$

$\in [0 + 2 + 1 \,..\, 1 + 9 + 1]$

What is the Search Space?



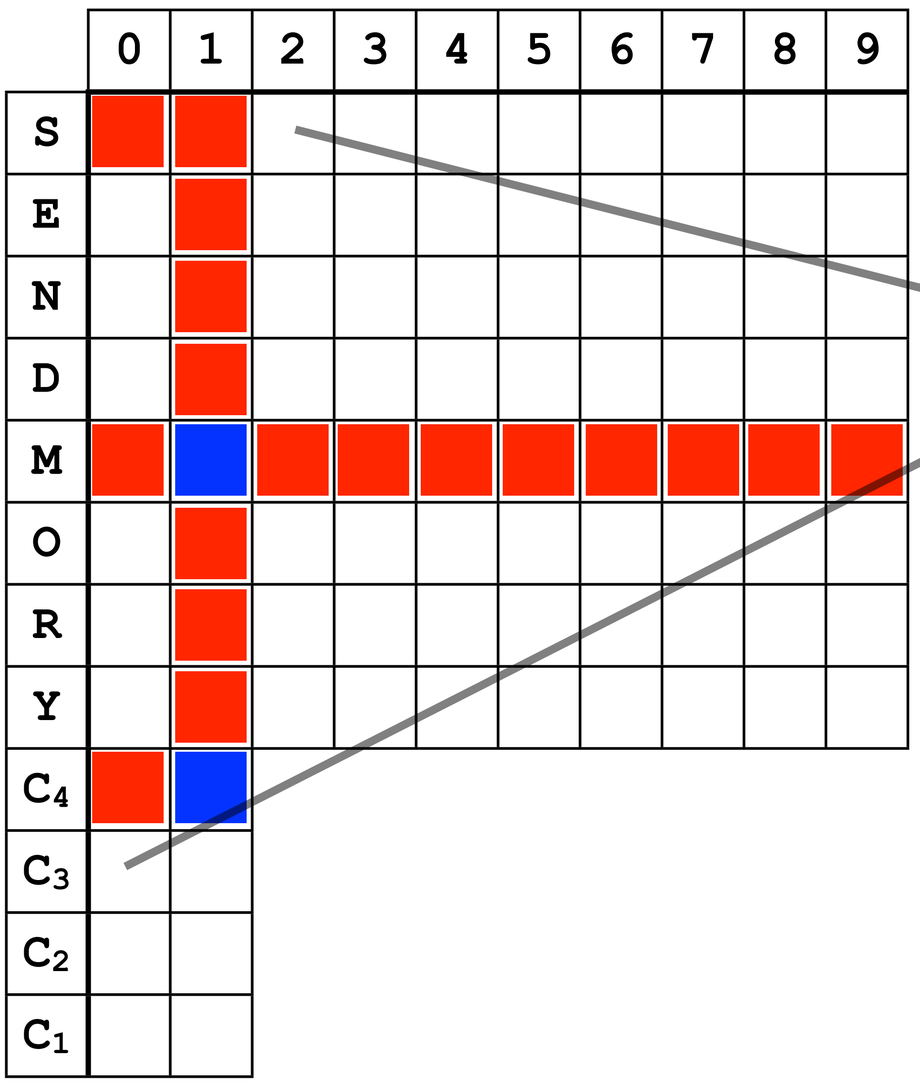

```
carry[3] + value[S] + value[M] = value[O] + 10 * carry[4];
```

```
carry[3] + value[S] + 1        = value[O] + 10 * carry[4];
```

$\in [3..11]$

$\in [0 + 2 + 1 \,..\, 1 + 9 + 1]$

What is the Search Space?



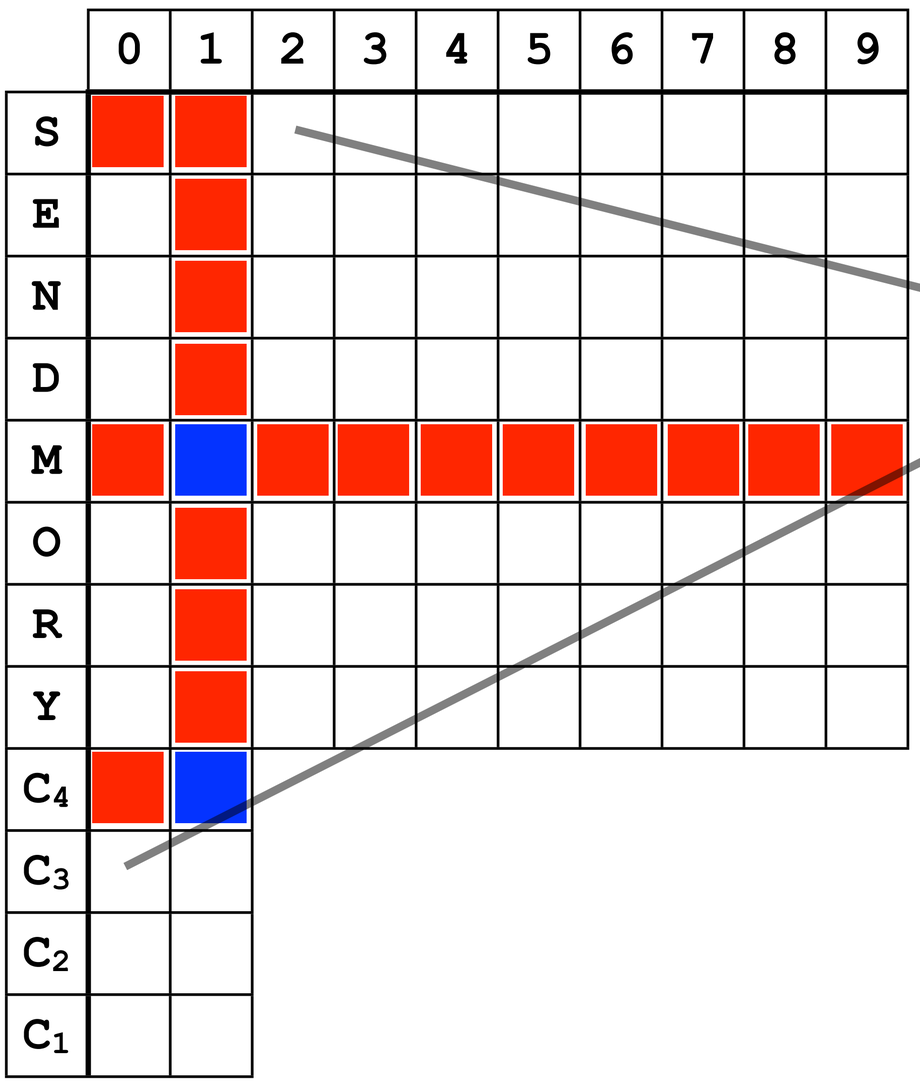

```
carry[3] + value[S] + value[M] = value[O] + 10 * carry[4];
```

```
carry[3] + value[S] + 1        = value[O] + 10 * carry[4];
```

$\in [3..11]$

$\in [0 + 2 + 1 \,..\, 1 + 9 + 1]$

What is the Search Space?



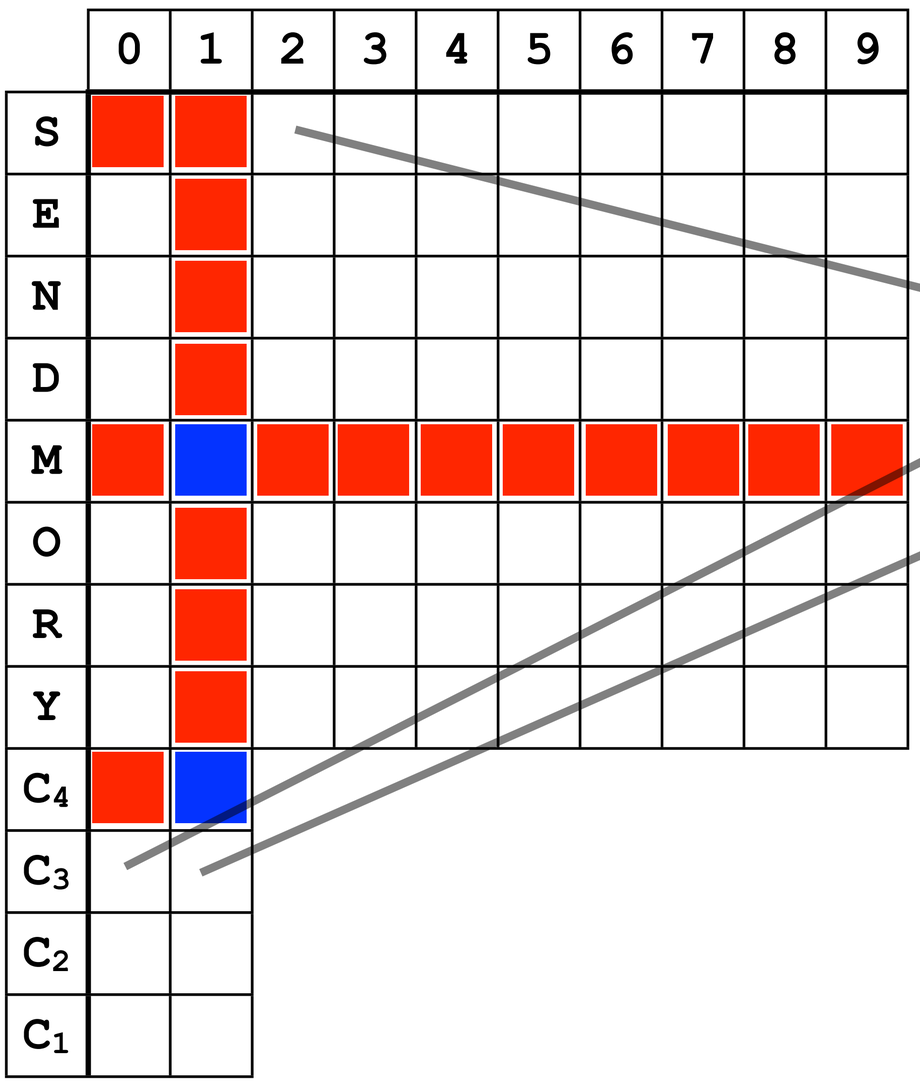

```
carry[3] + value[S] + value[M] = value[O] + 10 * carry[4];
```

```
carry[3] + value[S] + 1        = value[O] + 10 * carry[4];
```

$\in [3 .. 11]$

$\in [0 + 2 + 1 .. 1 + 9 + 1]$

What is the Search Space?



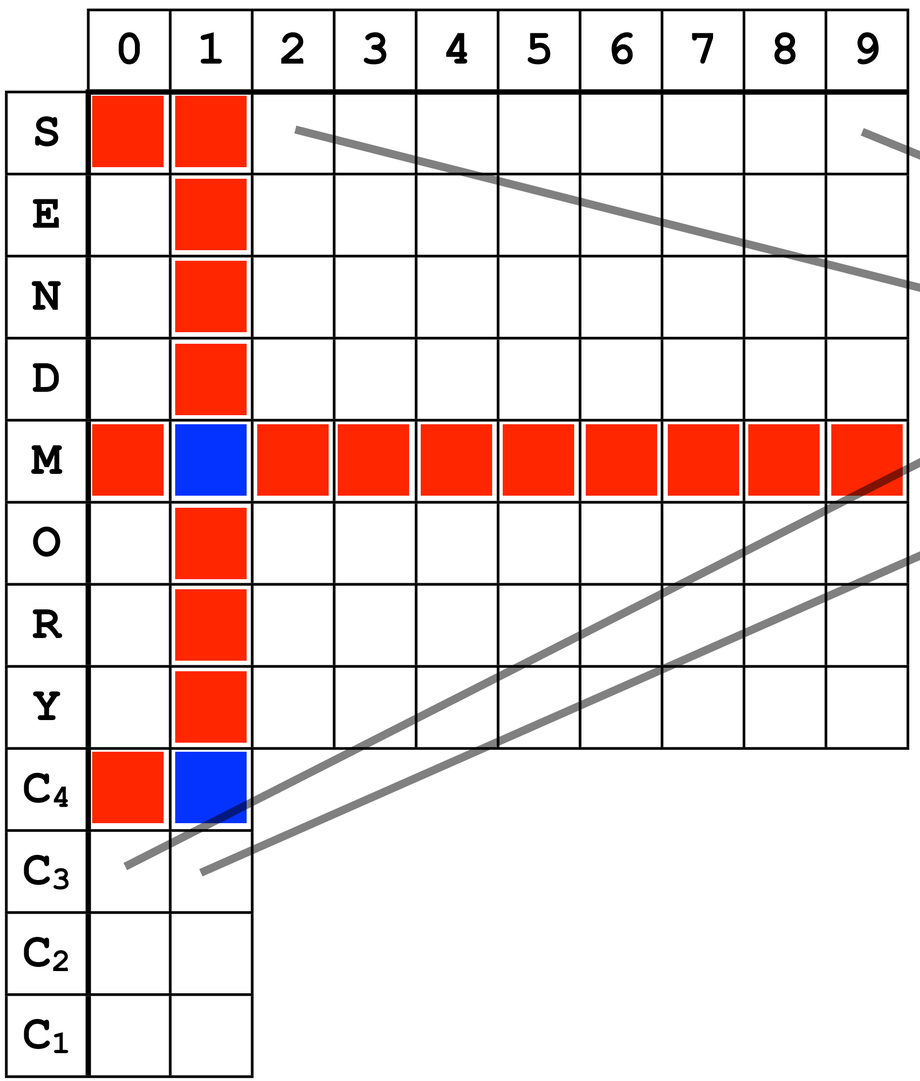

```
carry[3] + value[S] + value[M] = value[O] + 10 * carry[4];
```

```
carry[3] + value[S] + 1        = value[O] + 10 * carry[4];
```

$\in [3 .. 11]$

$\in [0 + 2 + 1 .. 1 + 9 + 1]$

What is the Search Space?



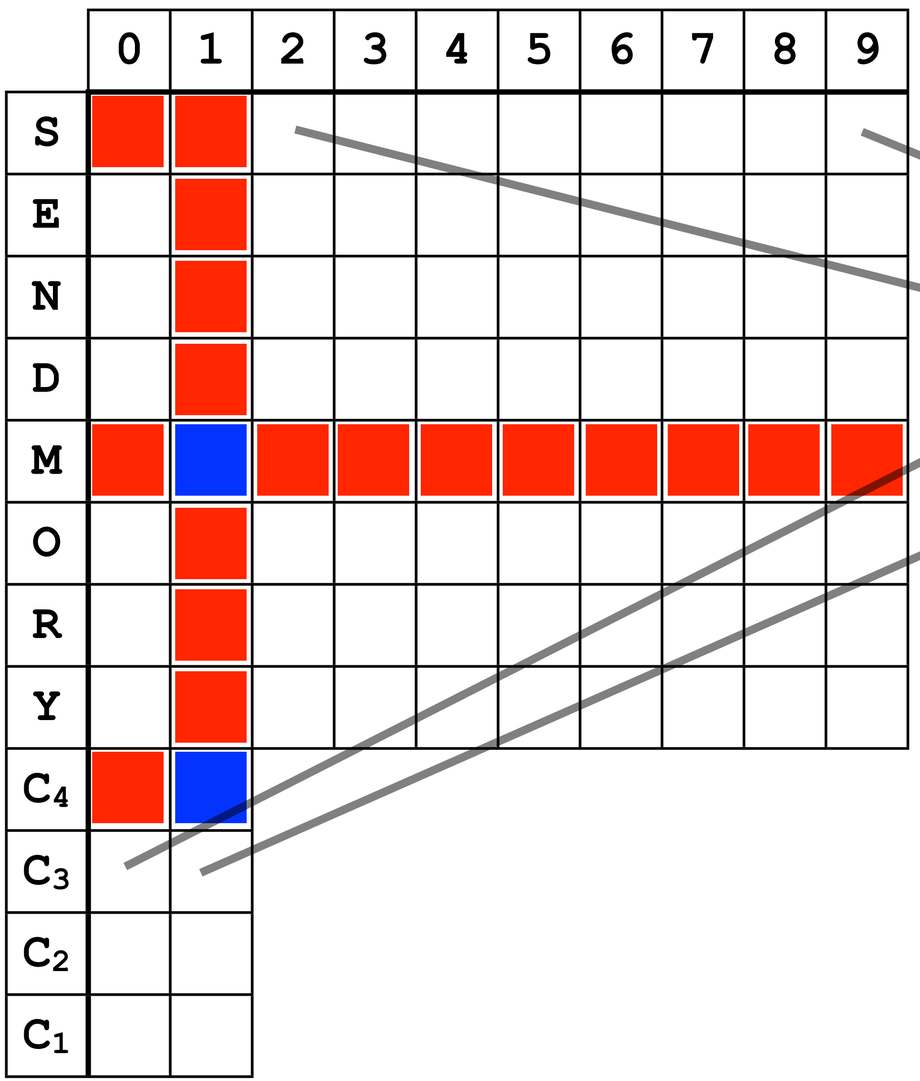

```
carry[3] + value[S] + value[M] = value[O] + 10 * carry[4];
```

```
carry[3] + value[S] + 1        = value[O] + 10 * carry[4];
```

$\in [3 .. 11]$

$\in [0 + 2 + 1 .. 1 + 9 + 1]$

What is the Search Space?



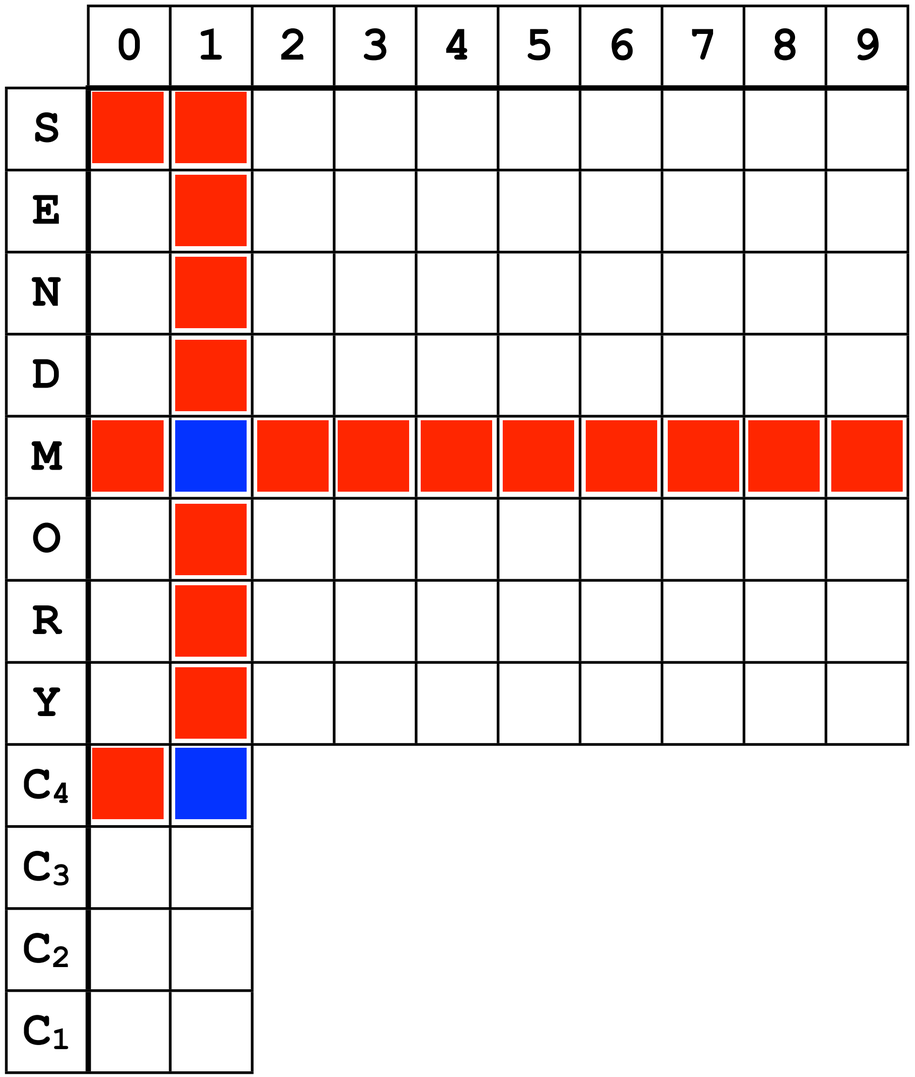

```
carry[3] + value[S] + value[M] = value[O] + 10 * carry[4];
```

```
carry[3] + value[S] + 1        = value[O] + 10 * carry[4];
```

$\in [3..11]$

$\in [0 + 2 + 1 .. 1 + 9 + 1]$

What is the Search Space?




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

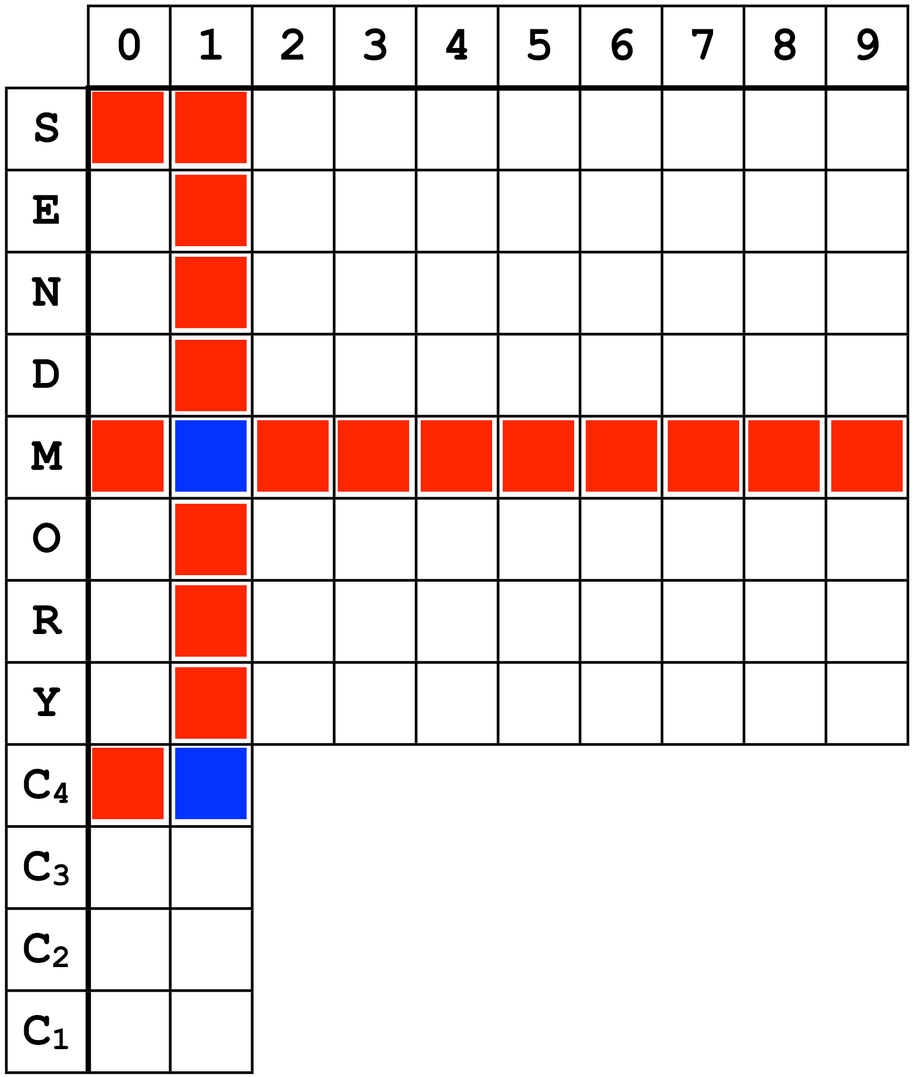
```
carry[3] + value[S] + 1        = value[O] + 10 * carry[4];
```

$\in [3..11]$

$\in [0 + 2 + 1 \,..\, 1 + 9 + 1]$

$\in [10..19]$

What is the Search Space?




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

```
carry[3] + value[S] + 1        = value[O] + 10 * carry[4];
```

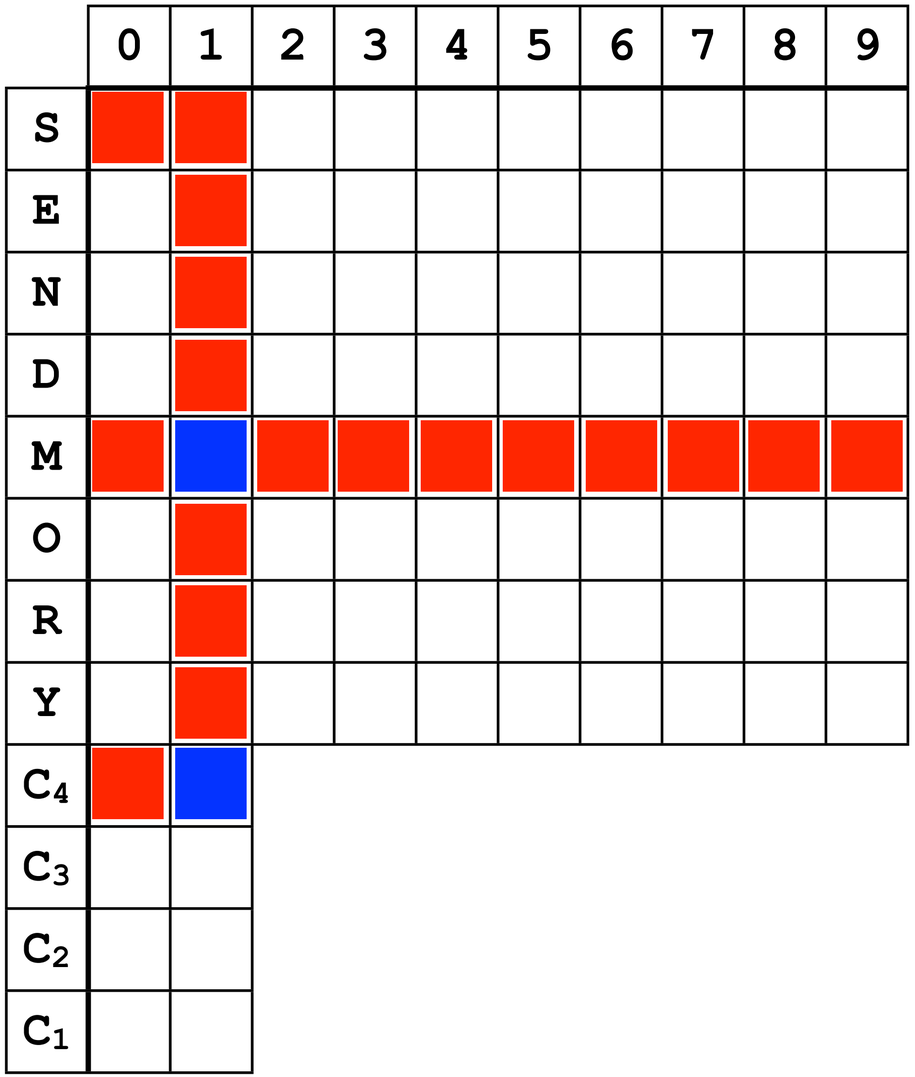
$\in [3..11]$

$\in [0 + 2 + 1 \,..\, 1 + 9 + 1]$

$\in [10..19]$

$$[3..11] \cap [10..19] = [10..11]$$

What is the Search Space?




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

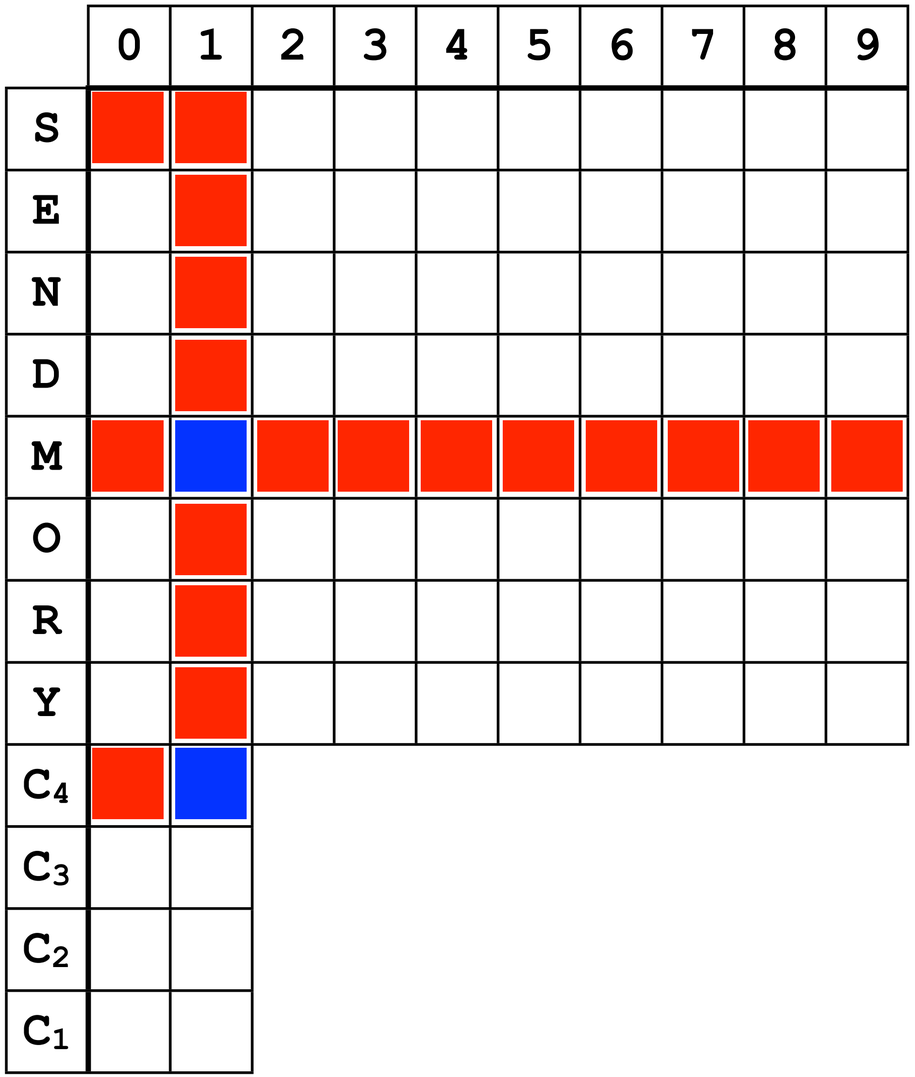
```
carry[3] + value[S] + 1        = value[O] + 10 * carry[4];
```

```
10 <= carry[3] + value[S] + 1 <= 11
```

$\in [0 + 2 + 1 \,..\, 1 + 9 + 1]$

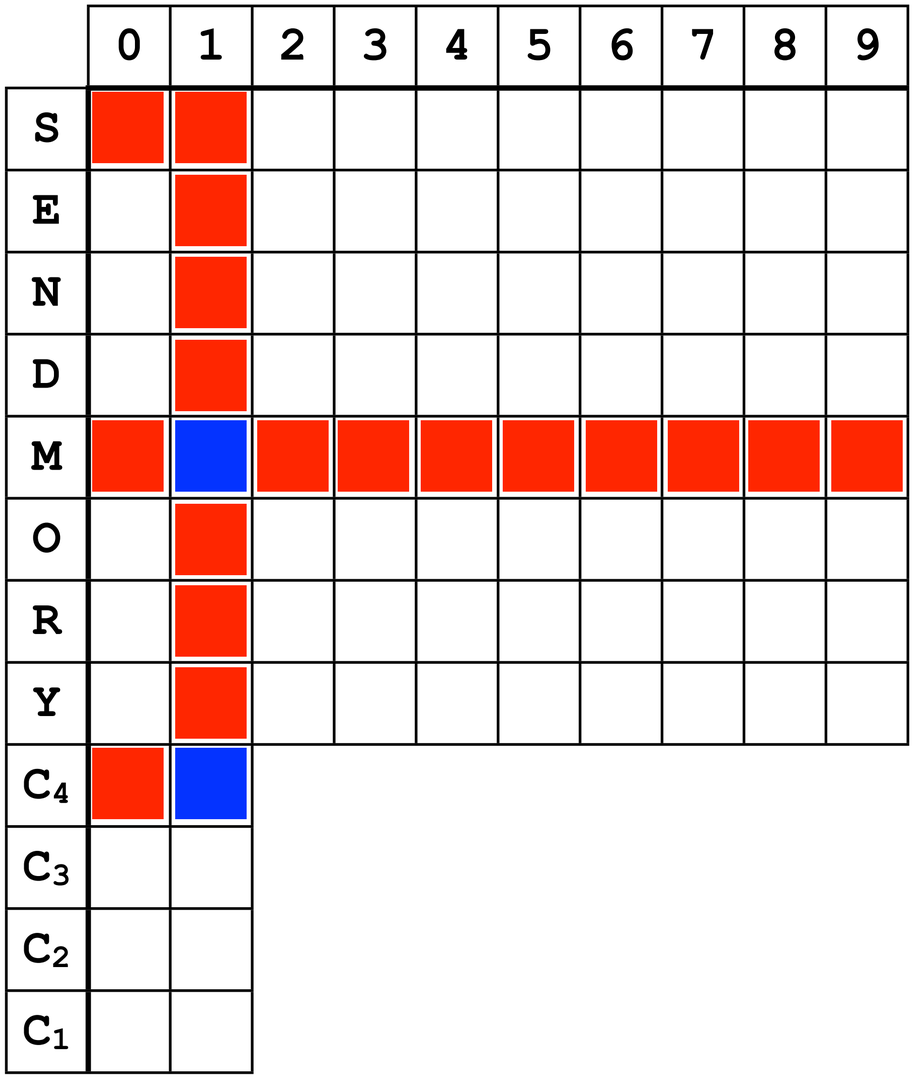
$[3..11] \cap [10..19] = [10..11]$

What is the Search Space?




```
10 <= carry[3] + value[S] + 1 <= 11
```

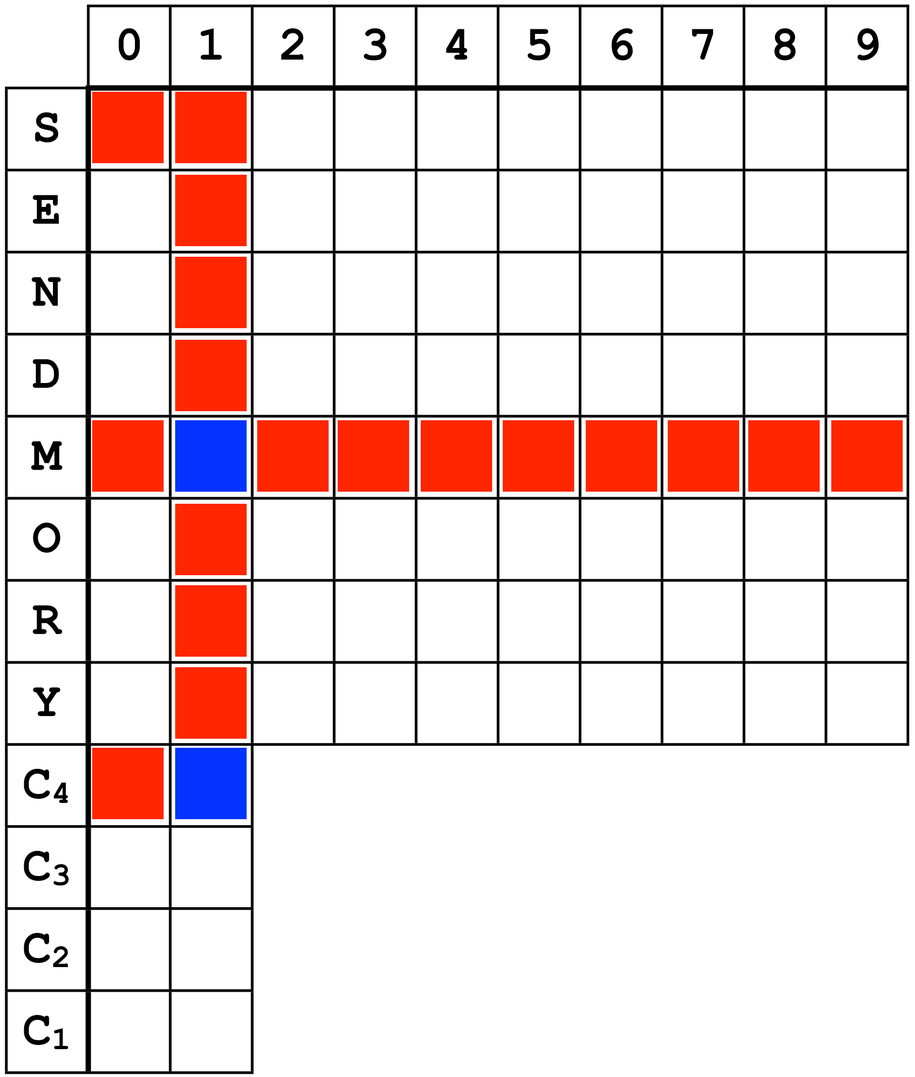
What is the Search Space?




```
10 <= carry[3] + value[S] + 1 <= 11
```

```
10 - 1 <= carry[3] + value[S] <= 11 - 1
```

What is the Search Space?

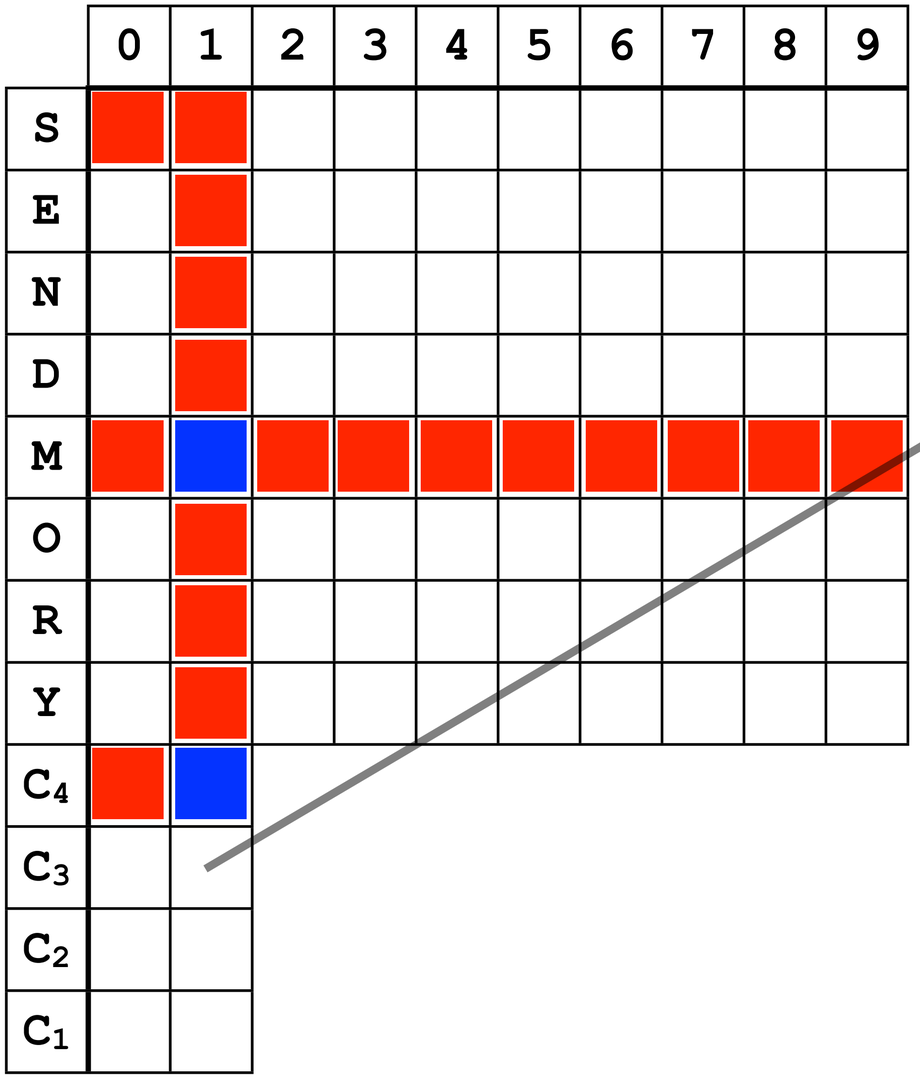



```
10 <= carry[3] + value[S] + 1 <= 11
```

```
10 - 1 <= carry[3] + value[S] <= 11 - 1
```

```
10 - 1 - carry[3] <= value[S] <= 11 - 1 - carry[3]
```

What is the Search Space?

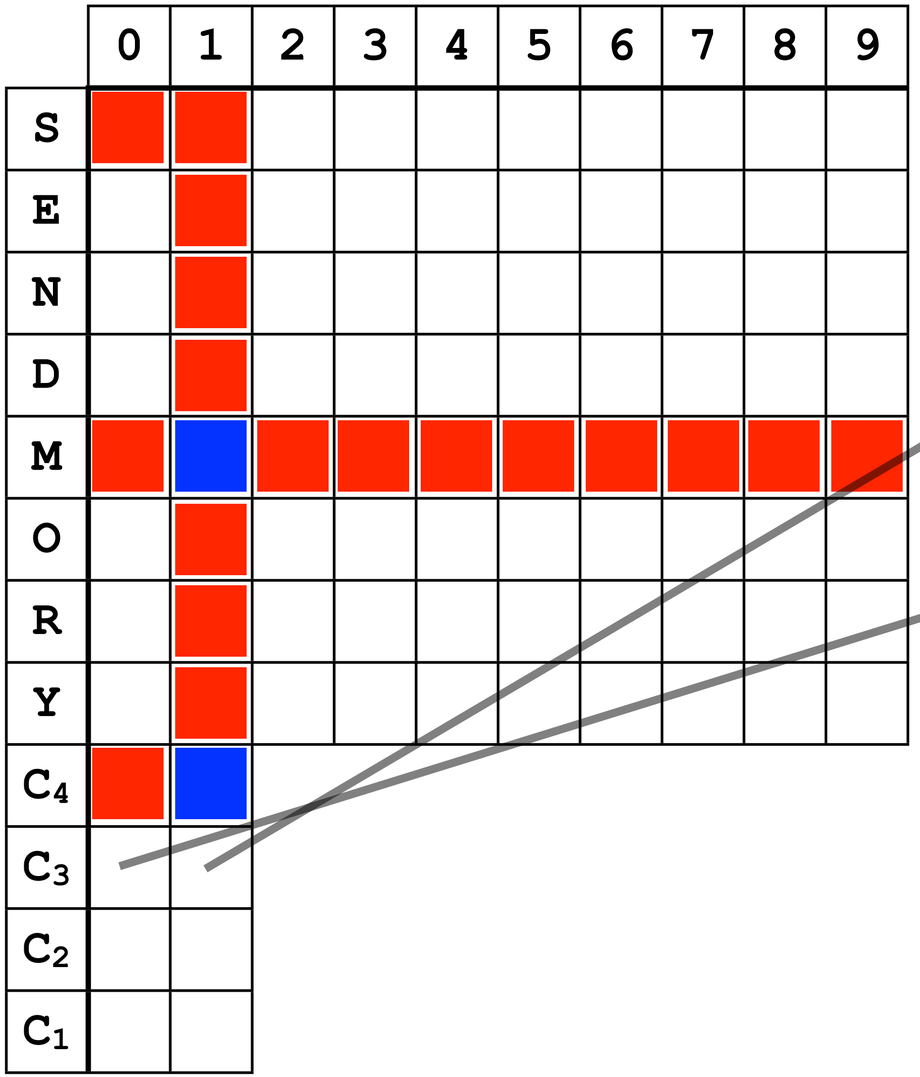



```
10 <= carry[3] + value[S] + 1 <= 11
```

```
10 - 1 <= carry[3] + value[S] <= 11 - 1
```

```
10 - 1 - carry[3] <= value[S] <= 11 - 1 - carry[3]
```

What is the Search Space?

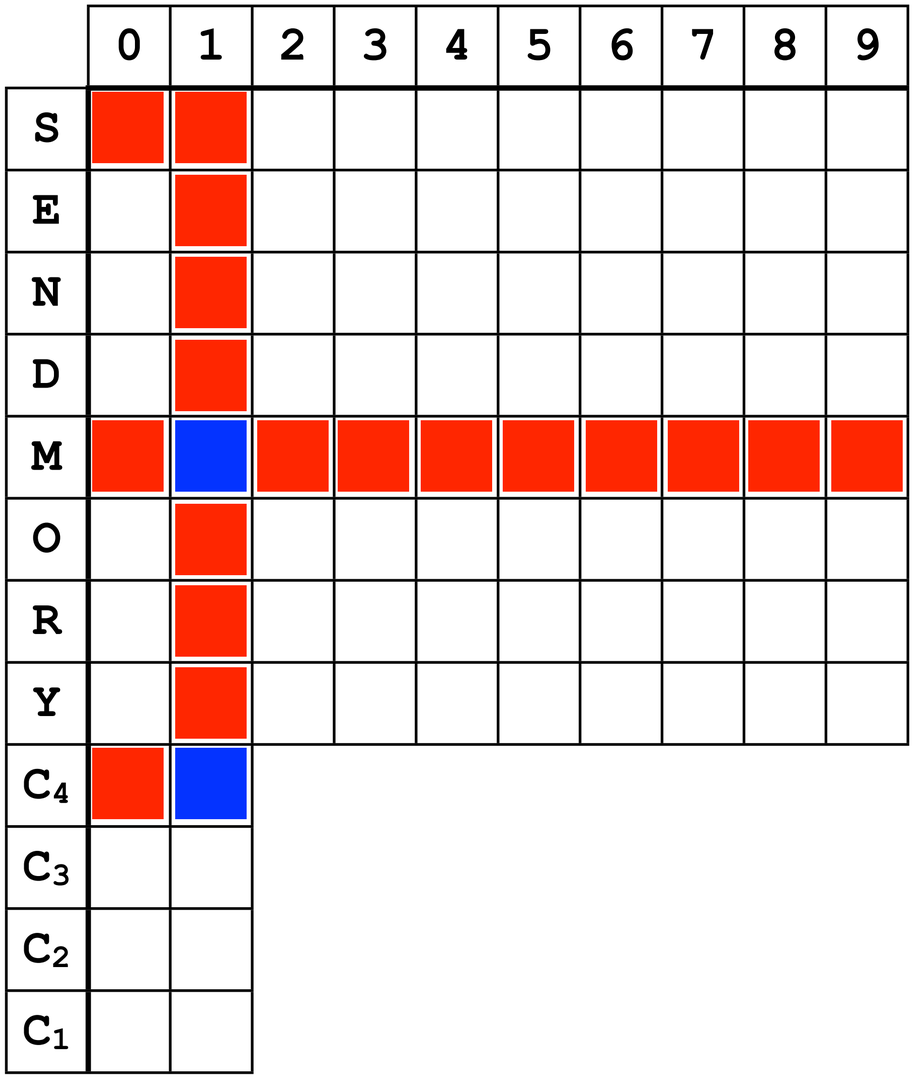



```
10 <= carry[3] + value[S] + 1 <= 11
```

```
10 - 1 <= carry[3] + value[S] <= 11 - 1
```

```
10 - 1 - carry[3] <= value[S] <= 11 - 1 - carry[3]
```

What is the Search Space?



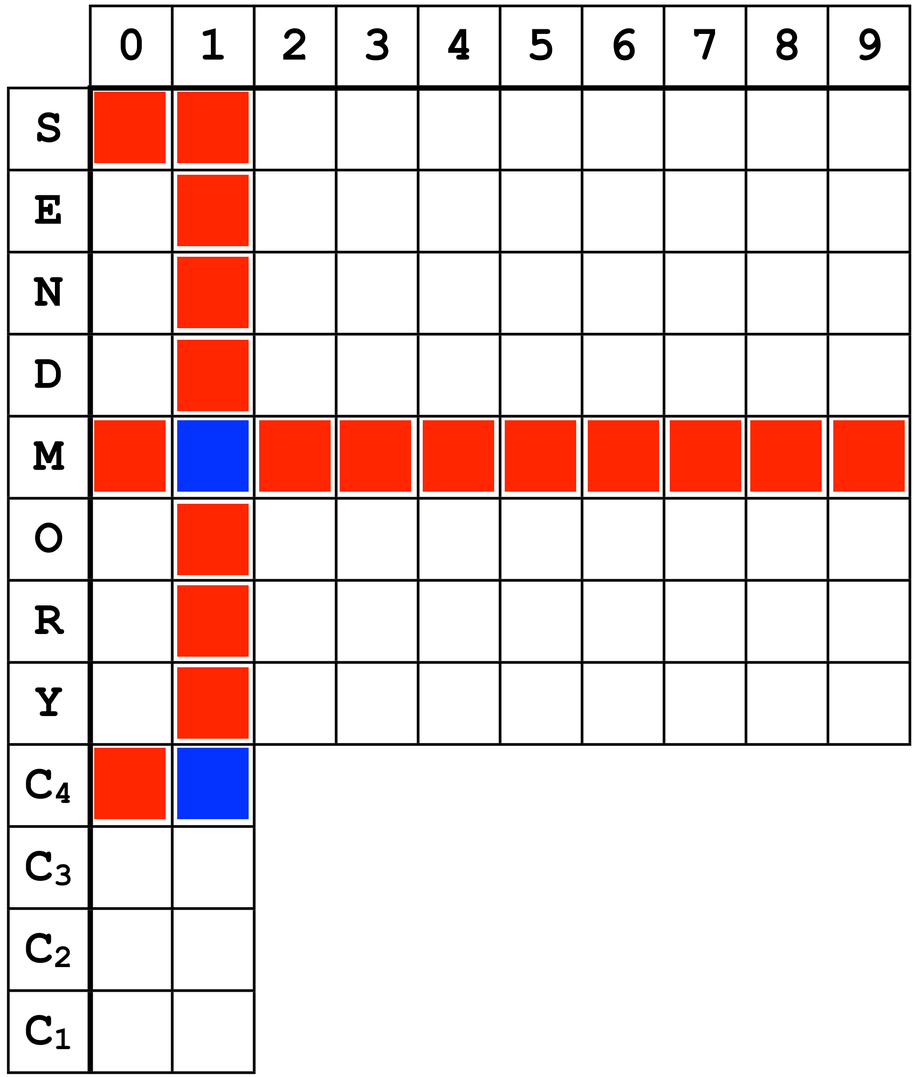

```
10 <= carry[3] + value[S] + 1 <= 11
```

```
10 - 1 <= carry[3] + value[S] <= 11 - 1
```

```
10 - 1 - carry[3] <= value[S] <= 11 - 1 - carry[3]
```

```
10 - 1 - 1 <= value[S] <= 11 - 1 - 0
```

What is the Search Space?




```
10 <= carry[3] + value[S] + 1 <= 11
```

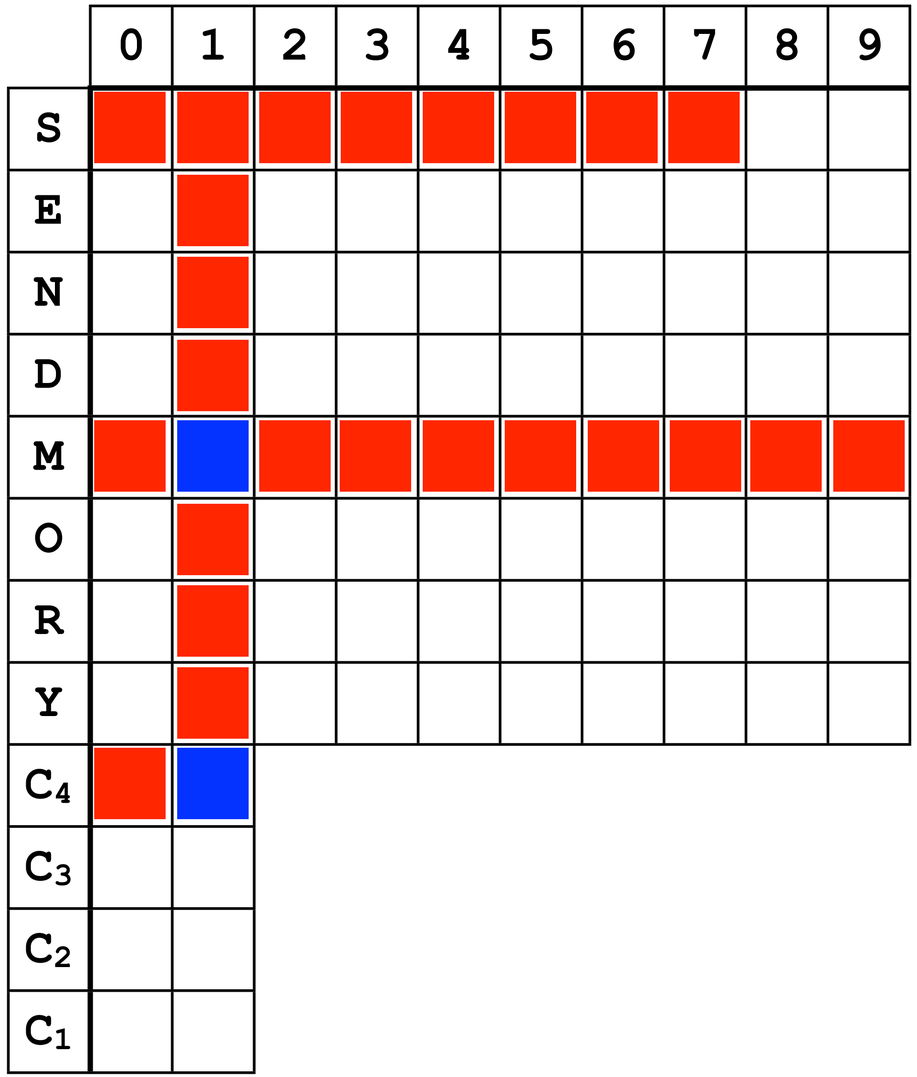
```
10 - 1 <= carry[3] + value[S] <= 11 - 1
```

```
10 - 1 - carry[3] <= value[S] <= 11 - 1 - carry[3]
```

```
10 - 1 - 1 <= value[S] <= 11 - 1 - 0
```

```
8 <= value[S] <= 10
```

What is the Search Space?




```
10 <= carry[3] + value[S] + 1 <= 11
```

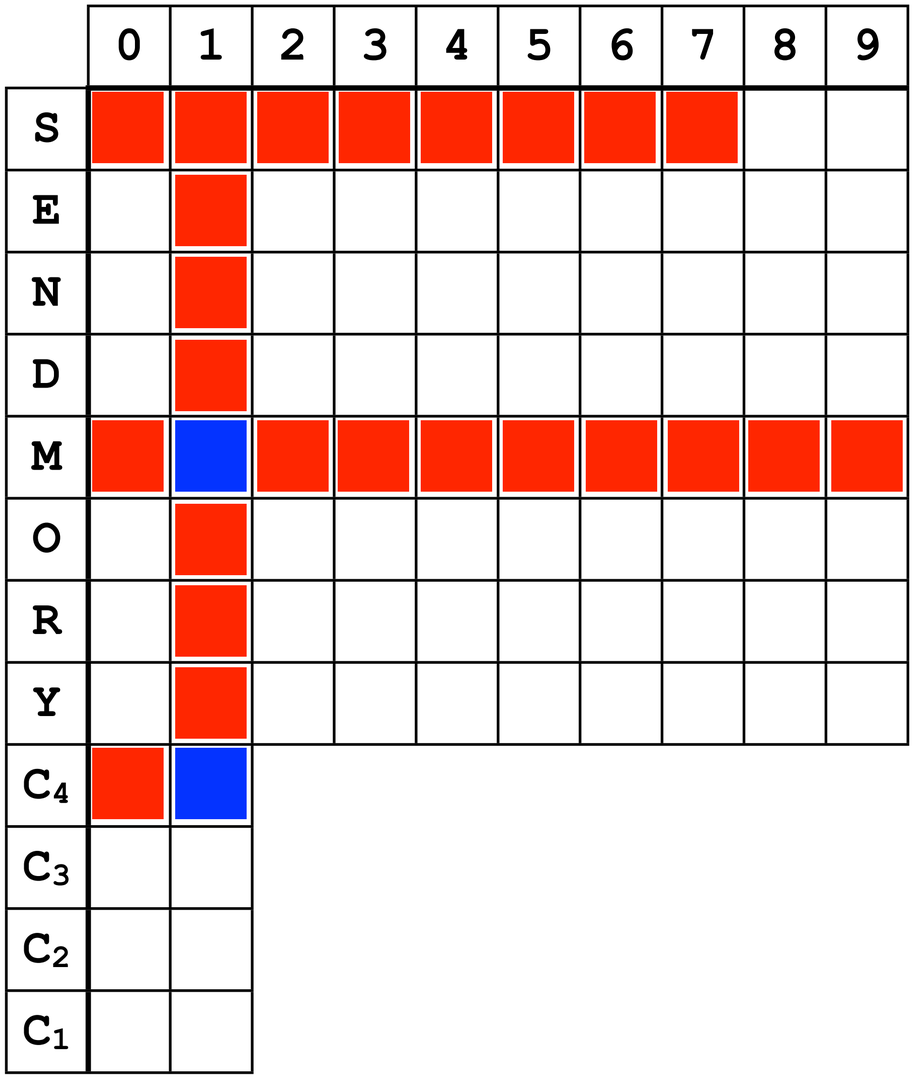
```
10 - 1 <= carry[3] + value[S] <= 11 - 1
```

```
10 - 1 - carry[3] <= value[S] <= 11 - 1 - carry[3]
```

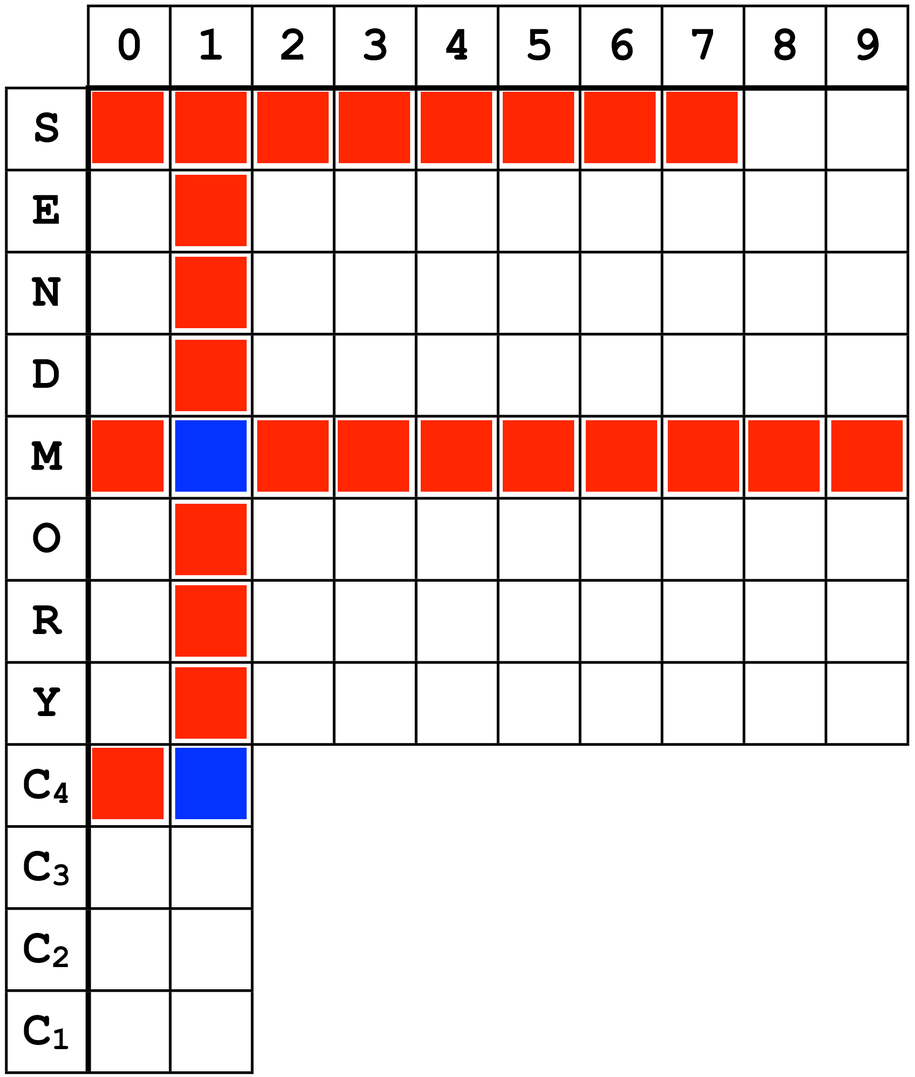
```
10 - 1 - 1 <= value[S] <= 11 - 1 - 0
```

```
8 <= value[S] <= 10
```

What is the Search Space?



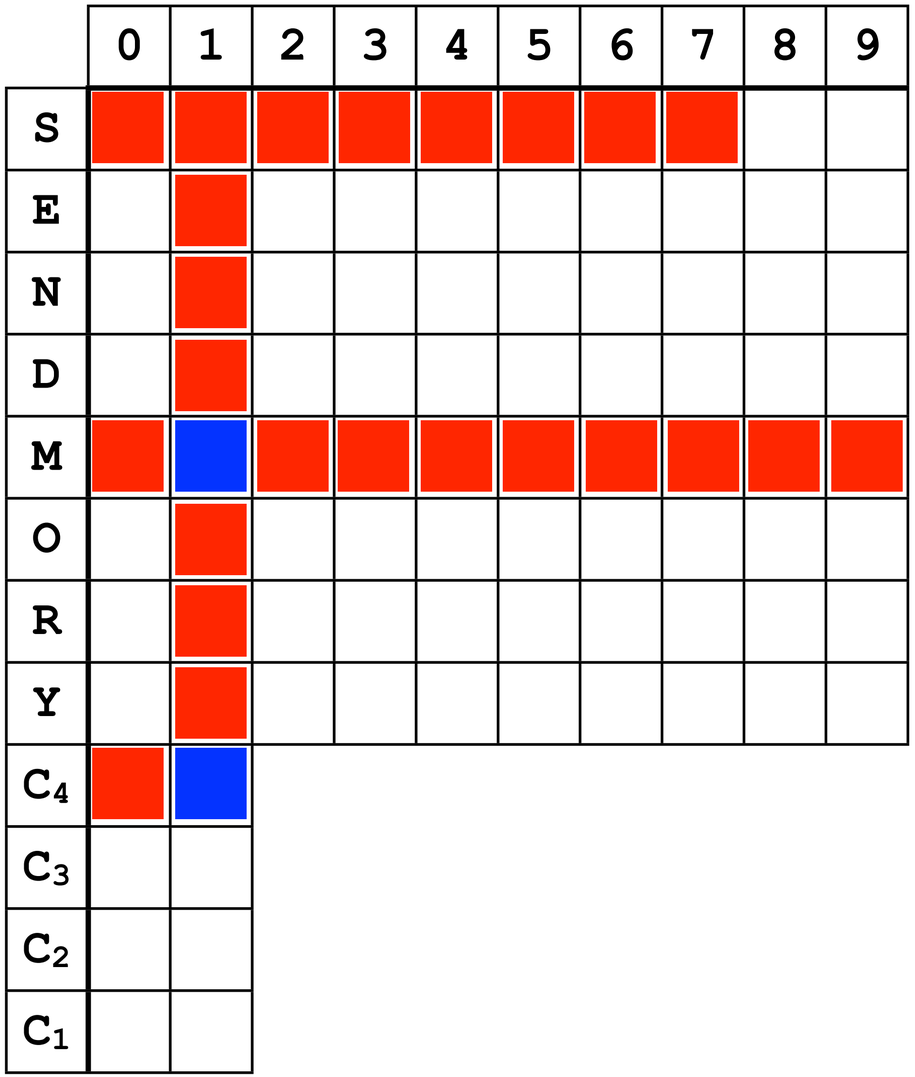
What is the Search Space?




```
carry[3] + value[S] + 1          = value[O] + 10 * carry[4];
```

$\in [10..11]$

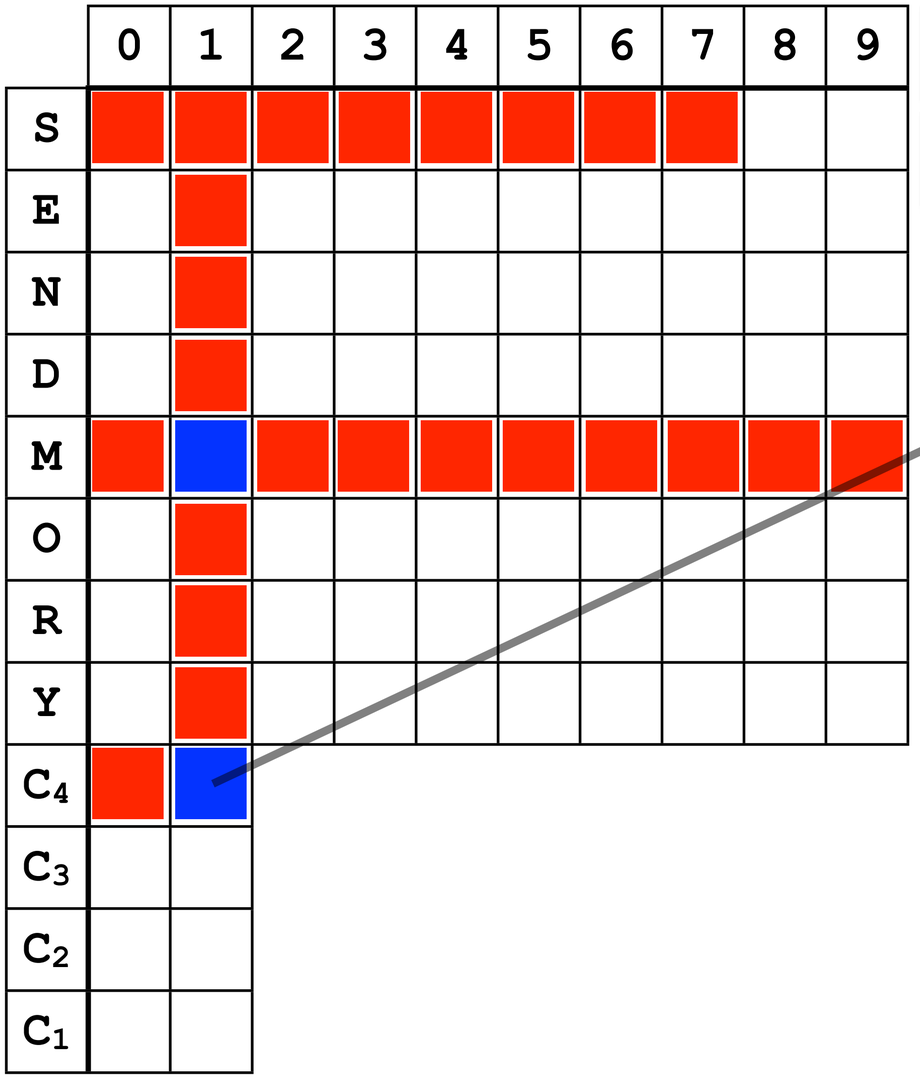
What is the Search Space?




```
carry[3] + value[S] + 1          = value[O] + 10 * carry[4];
```

```
10 <= value[O] + 10 * carry[4] <= 11
```

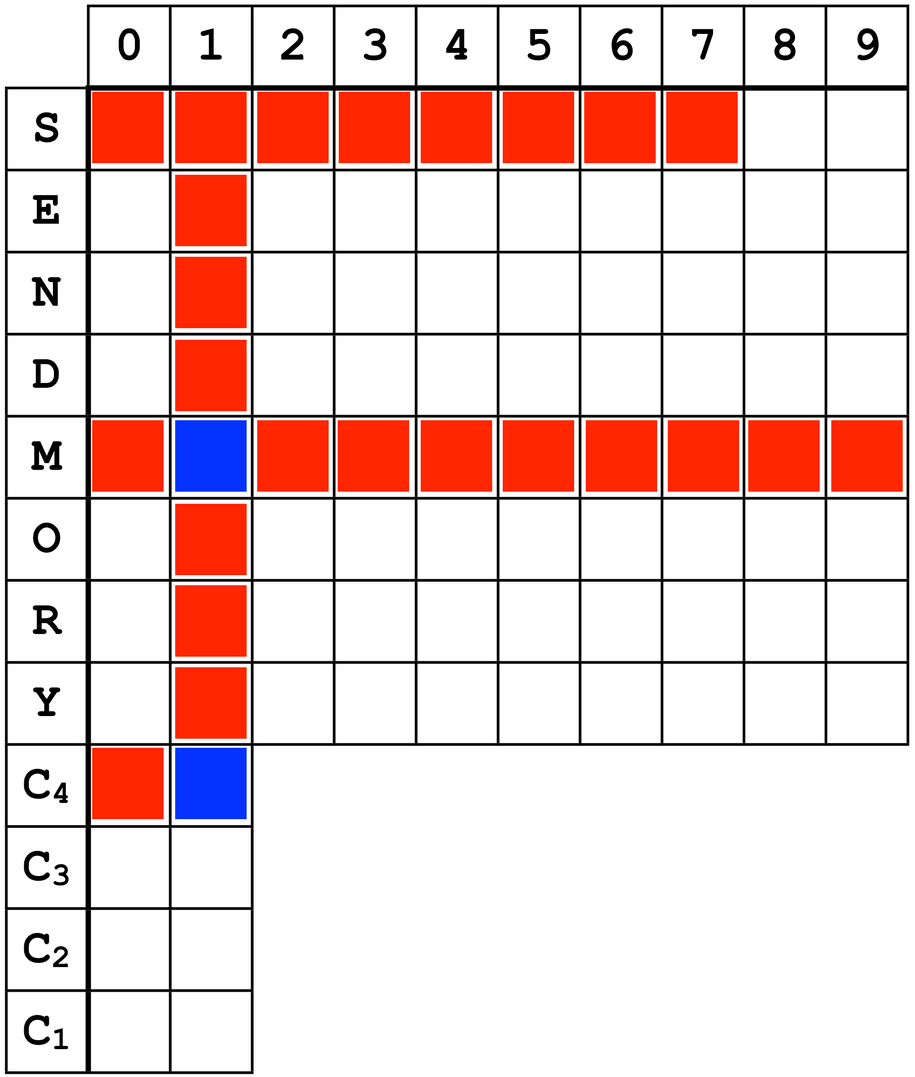
What is the Search Space?




```
carry[3] + value[S] + 1          = value[O] + 10 * carry[4];
```

```
10 <= value[O] + 10 * carry[4] <= 11
```

What is the Search Space?

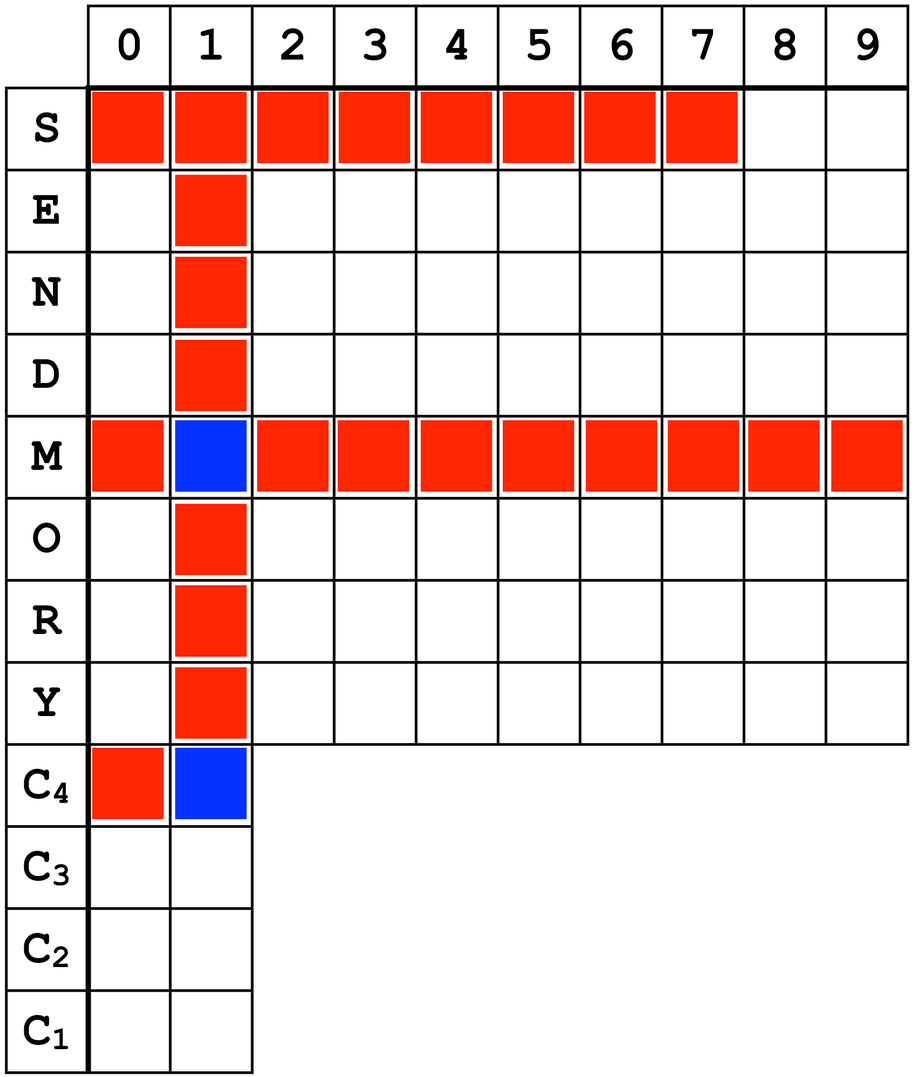



```
carry[3] + value[S] + 1          = value[O] + 10 * carry[4];
```

```
10 <= value[O] + 10 * carry[4] <= 11
```

```
10 <= value[O] + 10 * 1 <= 11
```

What is the Search Space?



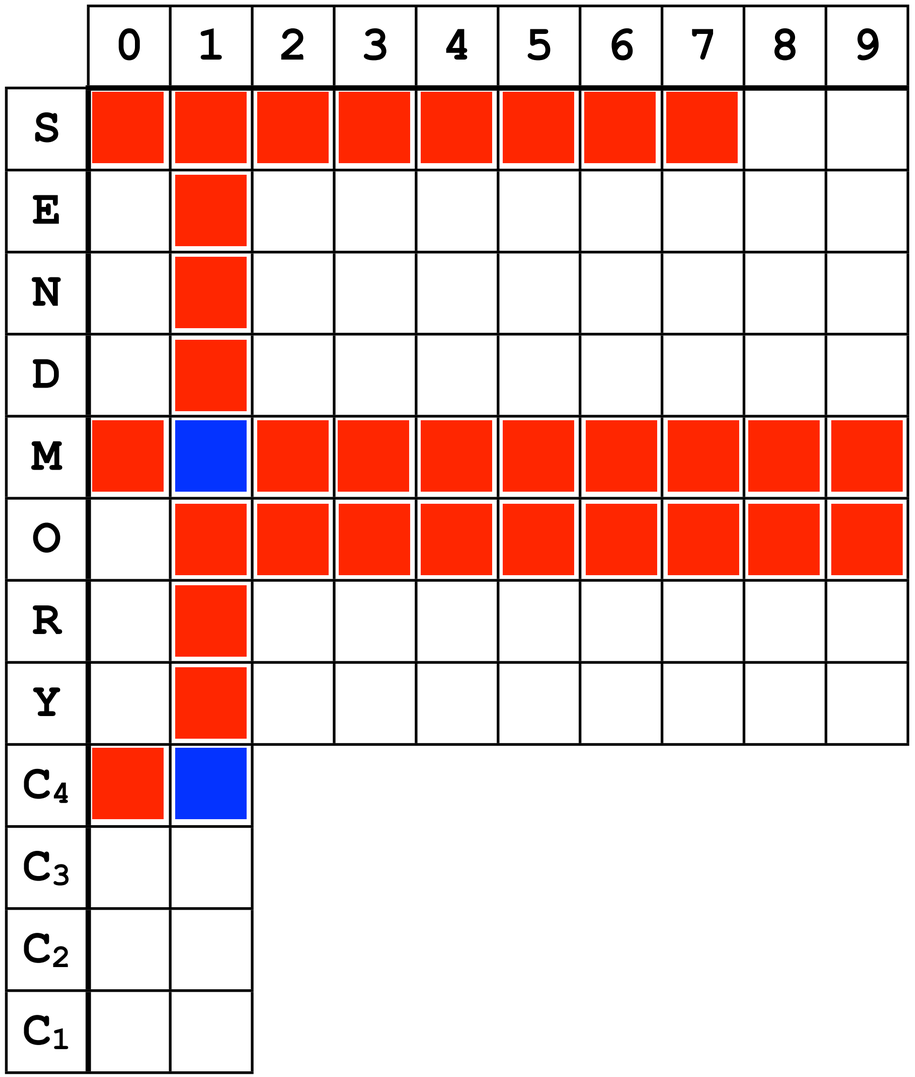

```
carry[3] + value[S] + 1          = value[O] + 10 * carry[4];
```

```
10 <= value[O] + 10 * carry[4] <= 11
```

```
10 <= value[O] + 10 * 1 <= 11
```

```
0 <= value[O] <= 1
```

What is the Search Space?



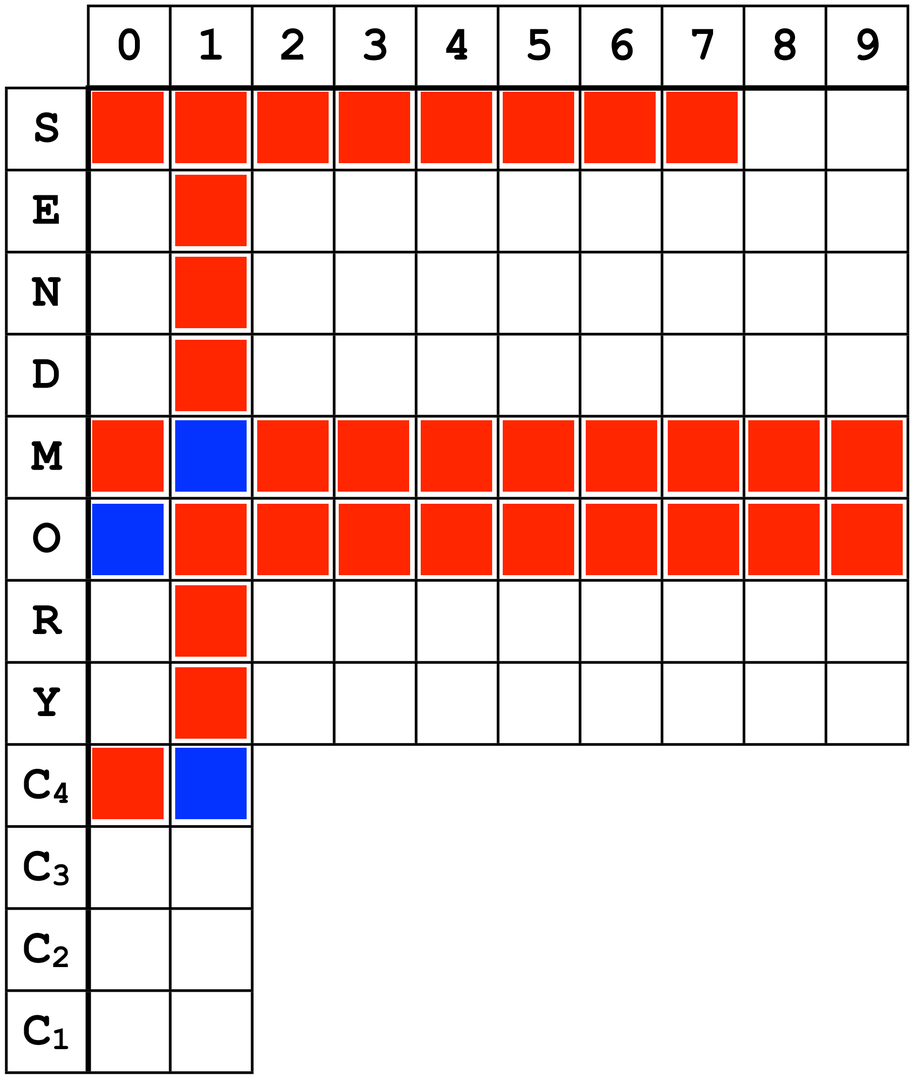

```
carry[3] + value[S] + 1          = value[O] + 10 * carry[4];
```

```
10 <= value[O] + 10 * carry[4] <= 11
```

```
10 <= value[O] + 10 * 1 <= 11
```

```
0 <= value[O] <= 1
```

What is the Search Space?



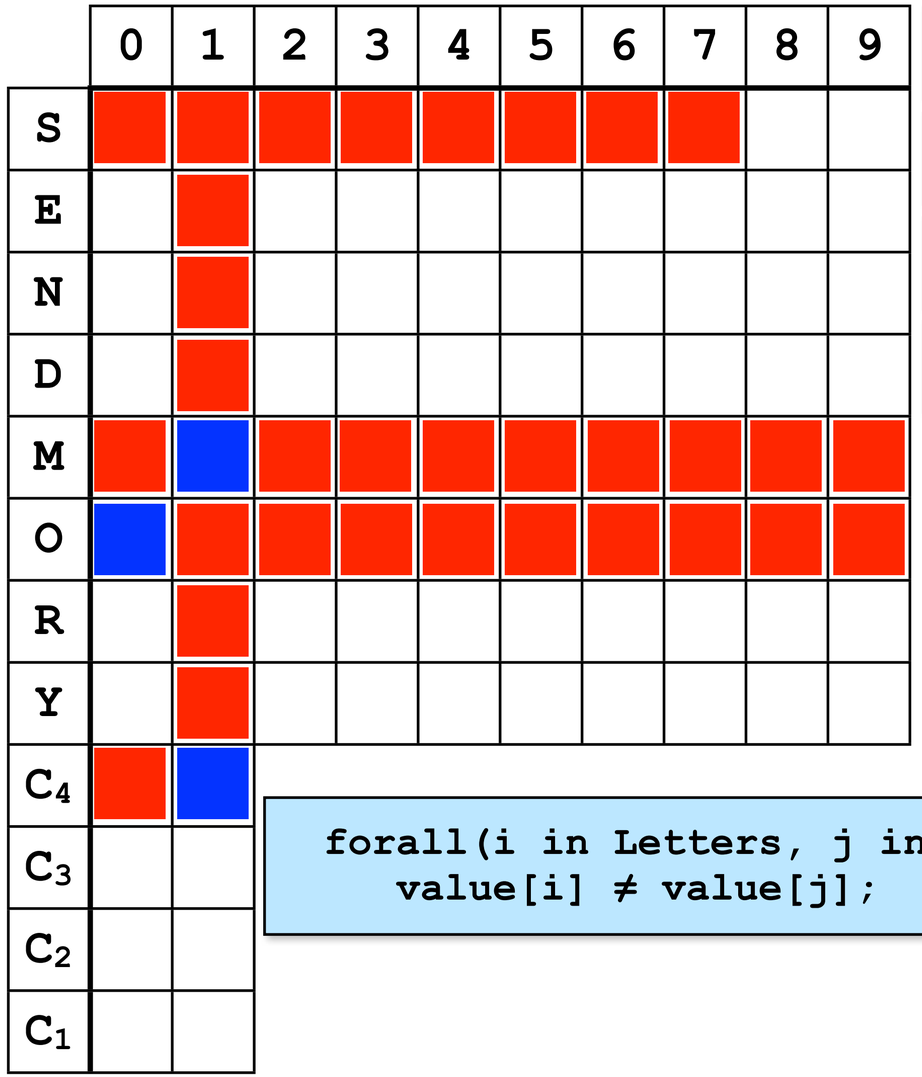

```
carry[3] + value[S] + 1          = value[O] + 10 * carry[4];
```

```
10 <= value[O] + 10 * carry[4] <= 11
```

```
10 <= value[O] + 10 * 1 <= 11
```

```
0 <= value[O] <= 1
```

What is the Search Space?




```
carry[3] + value[S] + 1          = value[O] + 10 * carry[4];
```

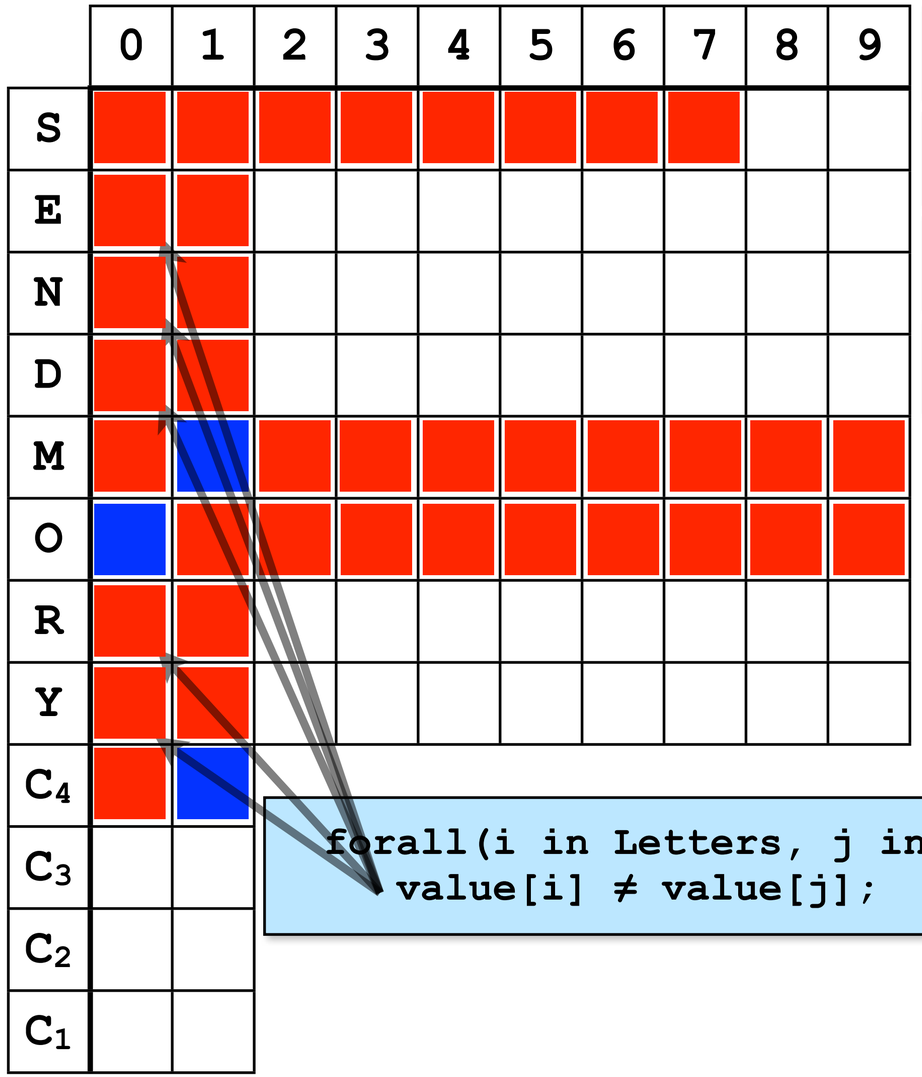
```
10 <= value[O] + 10 * carry[4] <= 11
```

```
10 <= value[O] + 10 * 1 <= 11
```

```
0 <= value[O] <= 1
```

```
forall(i in Letters, j in Letters)
   value[i] ≠ value[j];
```

What is the Search Space?




```
carry[3] + value[S] + 1          = value[O] + 10 * carry[4];
```

```
10 <= value[O] + 10 * carry[4] <= 11
```

```
10 <= value[O] + 10 * 1 <= 11
```

```
0 <= value[O] <= 1
```

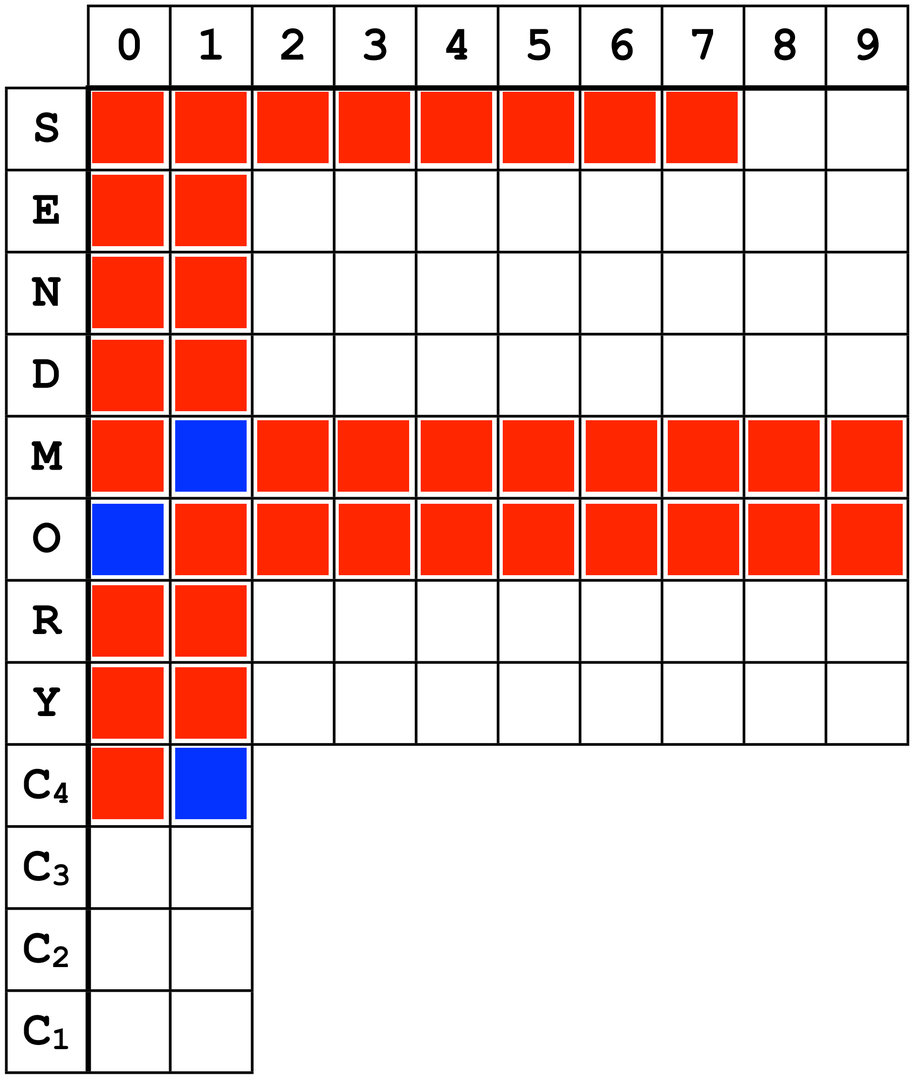
```
forall(i in Letters, j in Letters)
  value[i] ≠ value[j];
```

Send More Money

```
enum Letters = { S, E, N, D, M, O, R, Y};
range Digits = 0..9;
var{int} value[Letters] in Digits;
var{int} carry[1..4] in 0..1;

solve {
  forall(i in Letters, j in Letters: i < j)
     value[i] ≠ value[j];
  value[S] ≠ 0;
  value[M] ≠ 0;
  carry[4]                          = value[M];
  carry[3] + value[S] + value[M] = value[O] + 10 * carry[4];
  carry[2] + value[E] + value[O] = value[N] + 10 * carry[3];
  carry[1] + value[N] + value[R] = value[E] + 10 * carry[2];
             value[D] + value[E] = value[Y] + 10 * carry[1];
}
```

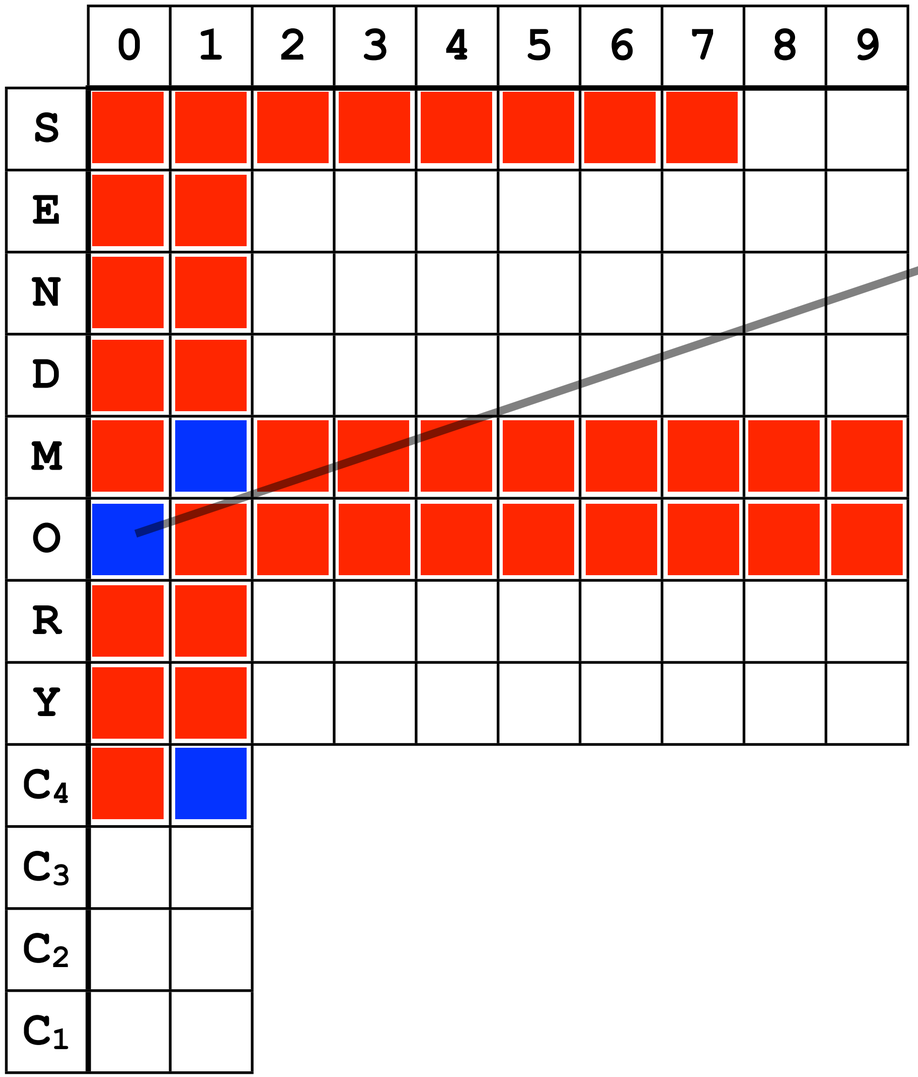
C_4	C_3	C_2	C_1	
	S	E	N	D
+	M	O	R	E
= M	O	N	E	Y

What is the Search Space?



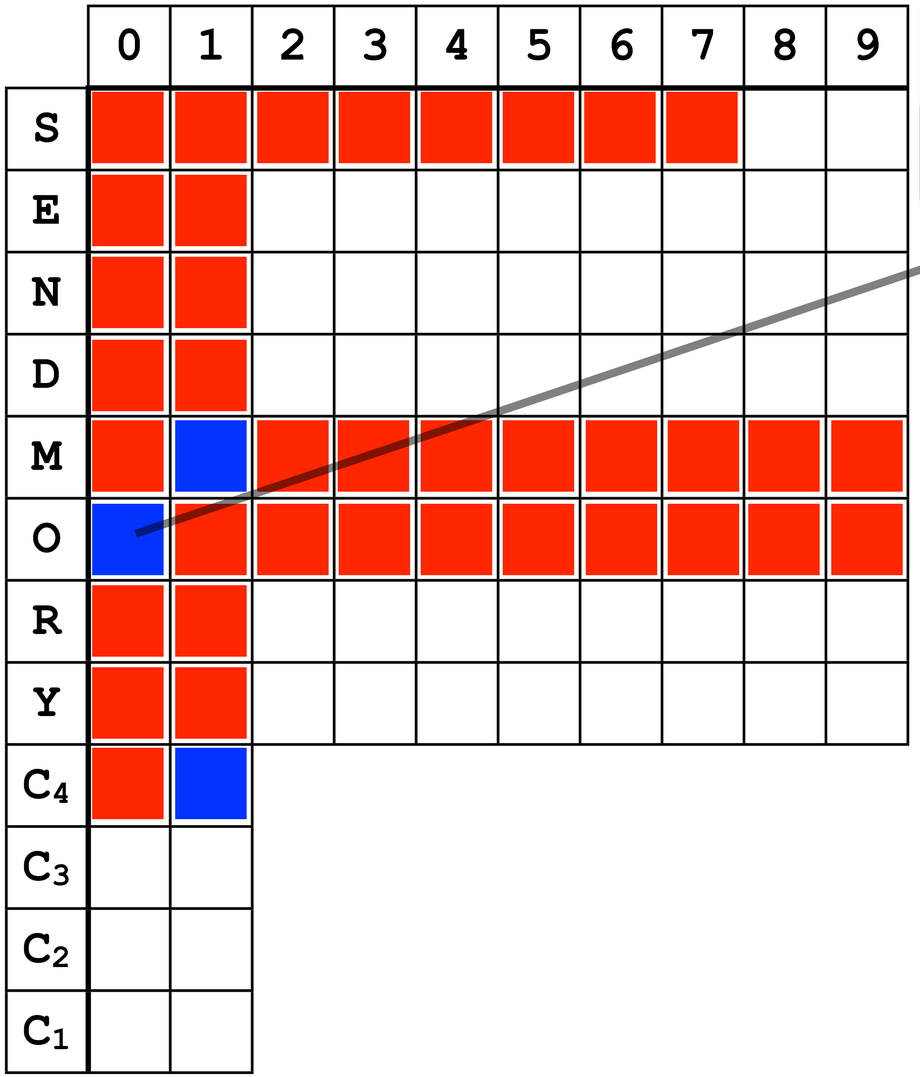

```
carry[2] + value[E] + value[O] = value[N] + 10 * carry[3];
```

What is the Search Space?




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

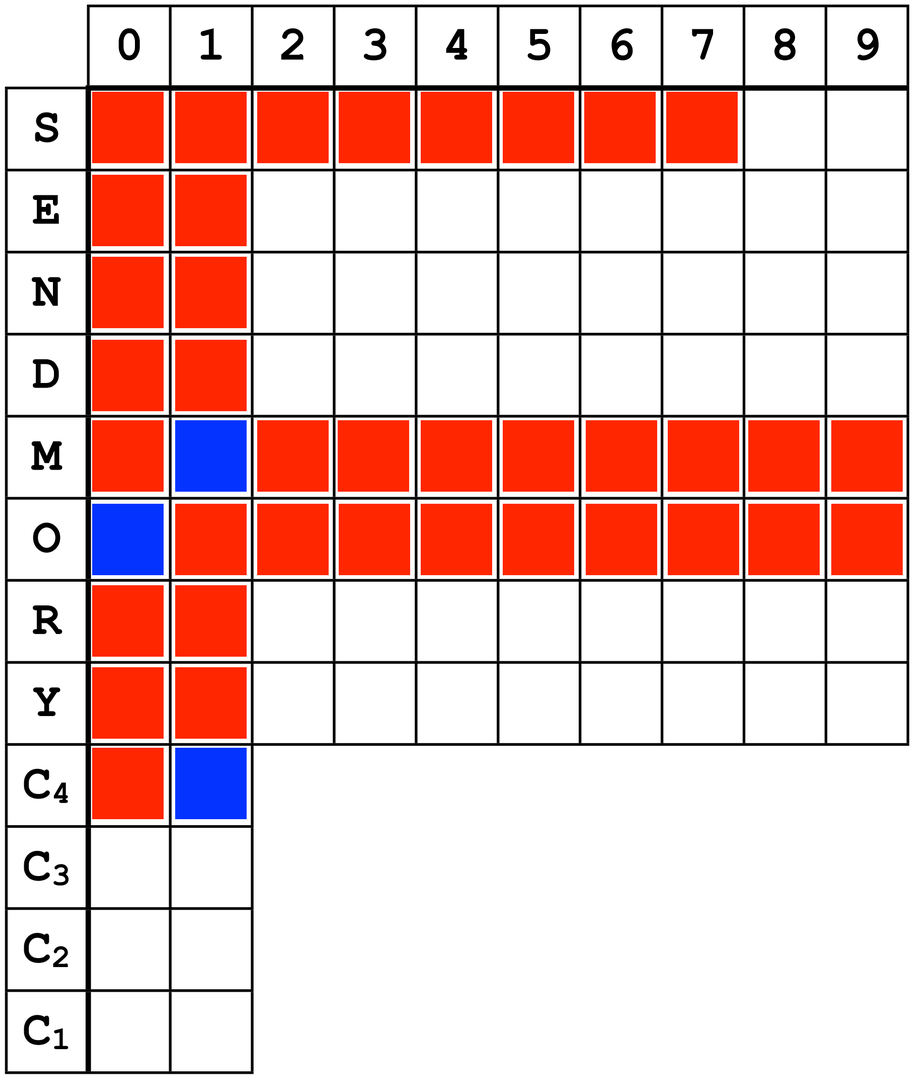
What is the Search Space?




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

```
carry[2] + value[E] + 0        = value[N] + 10 * carry[3];
```

What is the Search Space?

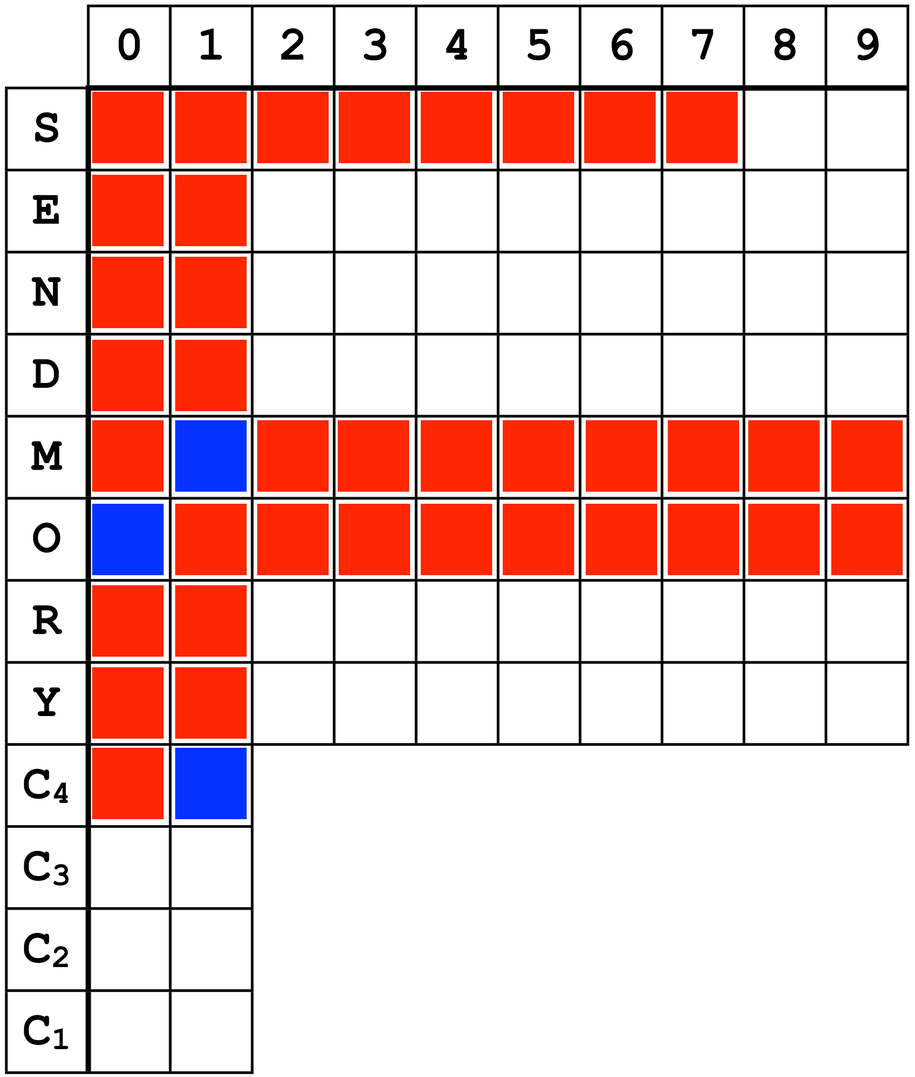



```
carry[2] + value[E] + value[O] = value[N] + 10 * carry[3];
```

```
carry[2] + value[E] + 0        = value[N] + 10 * carry[3];
```

$\in [2..10]$

What is the Search Space?



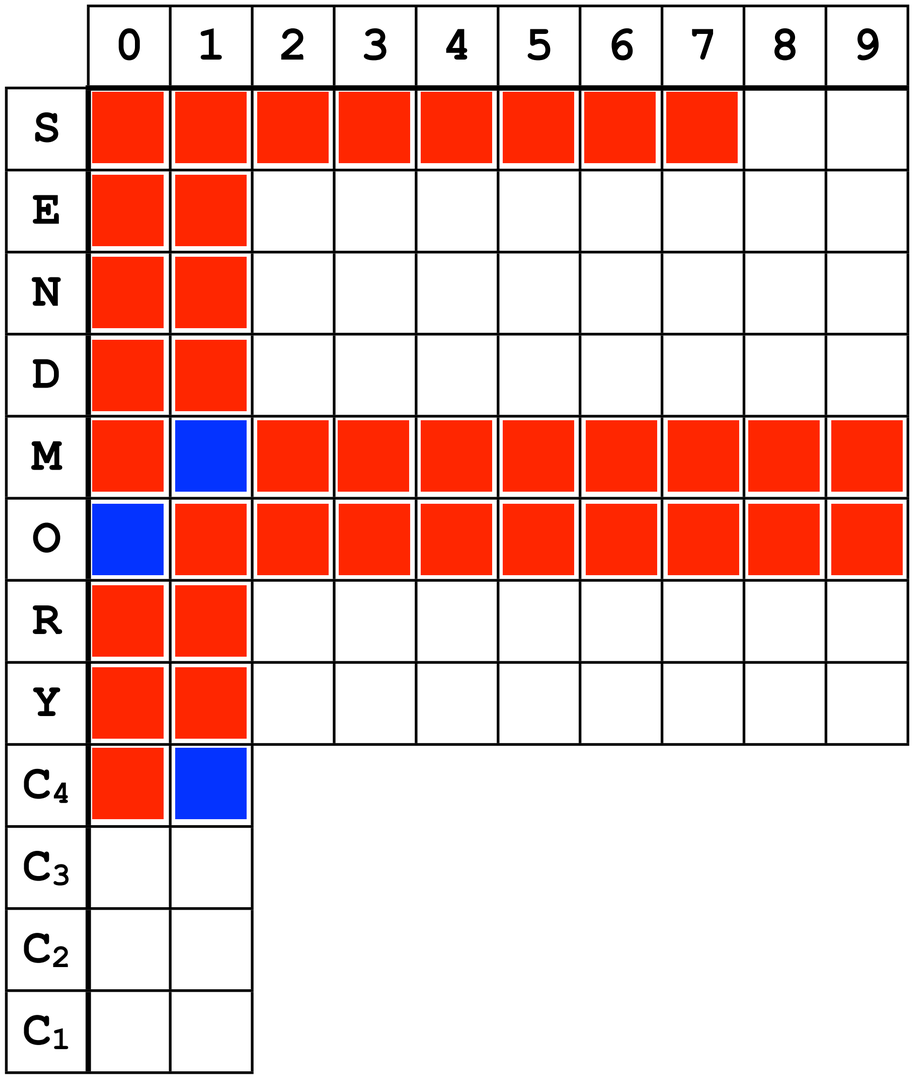

```
carry[2] + value[E] + value[O] = value[N] + 10 * carry[3];
```

```
carry[2] + value[E] + 0        = value[N] + 10 * carry[3];
```

$\in [2..10]$

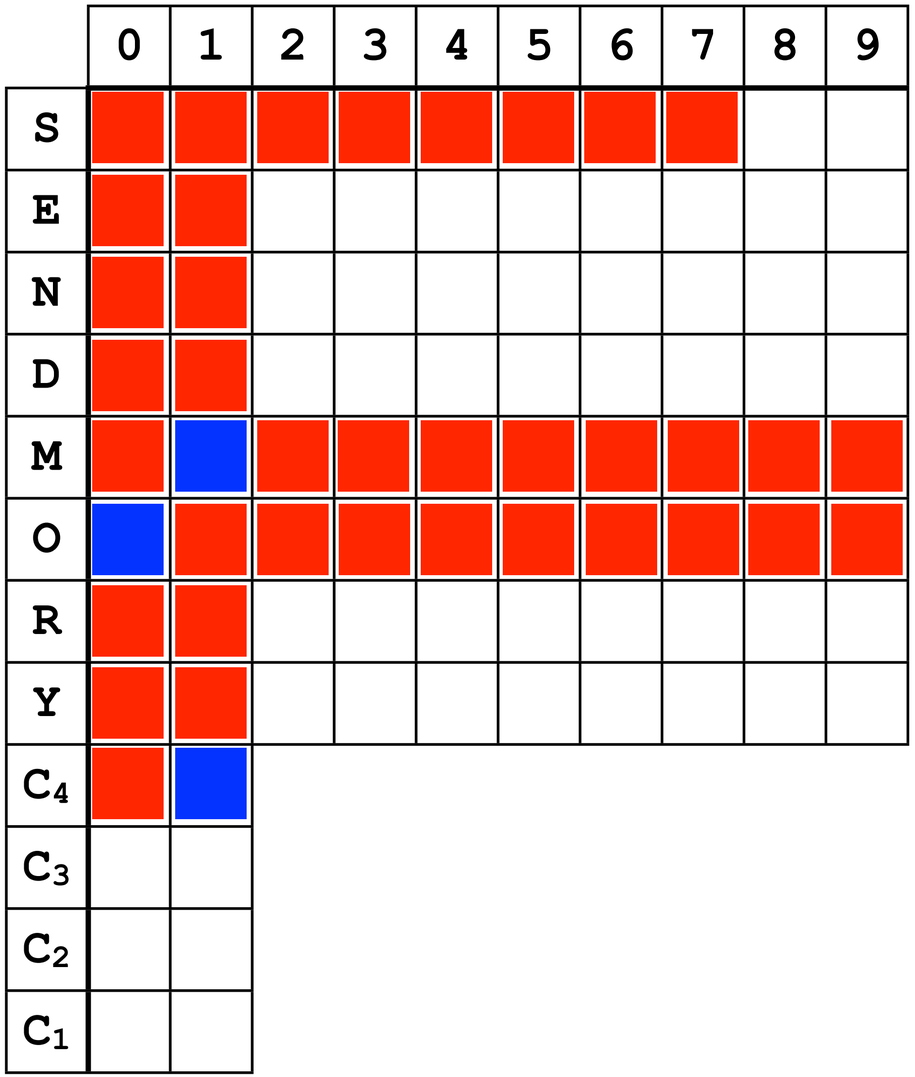
```
2 <= value[N] + 10 * carry[3] <= 10
```

What is the Search Space?




```
2 <= value[N] + 10 * carry[3] <= 10
```

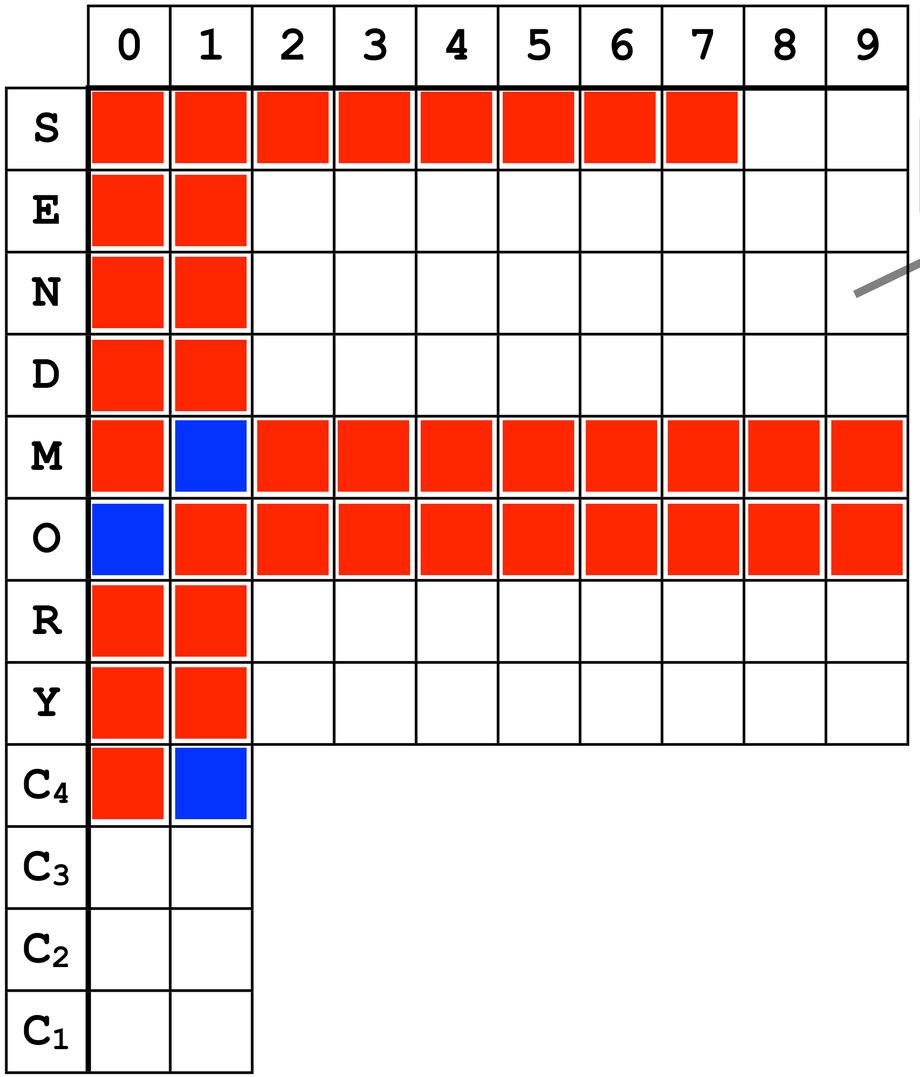
What is the Search Space?




```
2 <= value[N] + 10 * carry[3] <= 10
```

```
2 - value[N] <= 10 * carry[3] <= 10 - value[N]
```

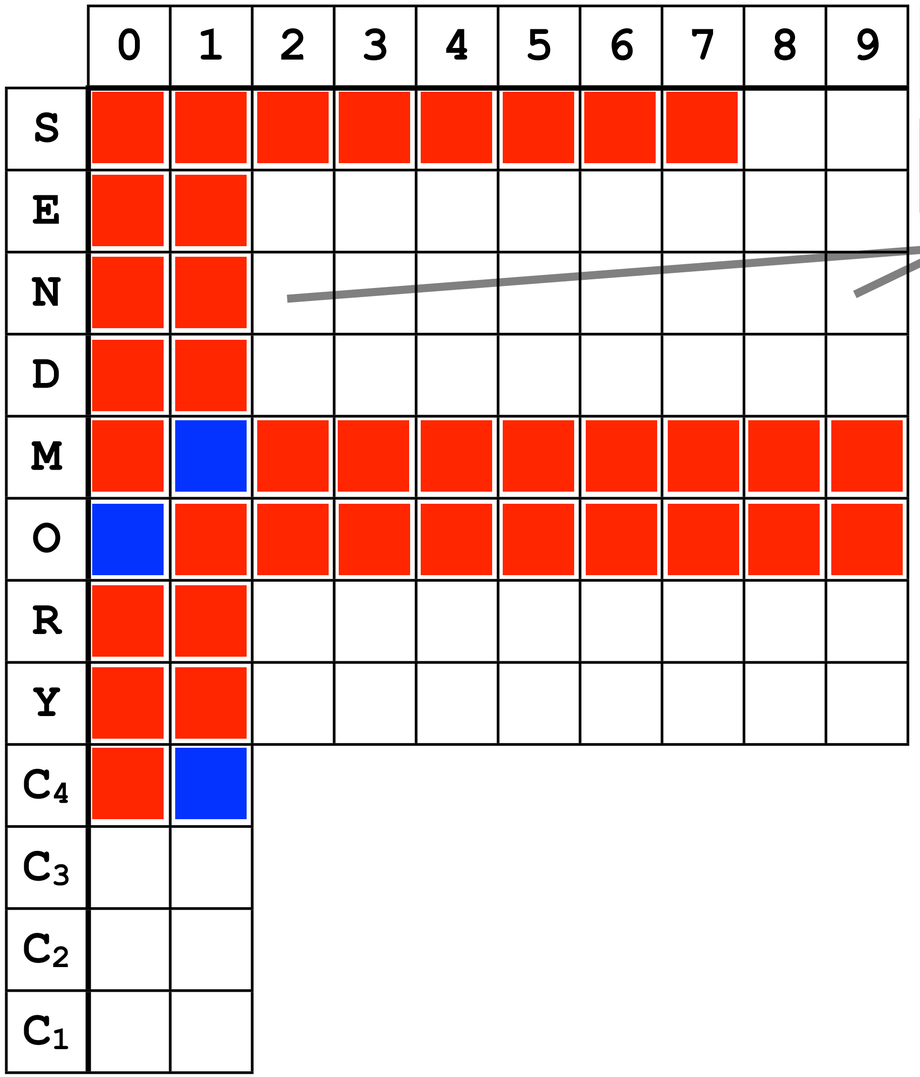
What is the Search Space?




```
2 <= value[N] + 10 * carry[3] <= 10
```

```
2 - value[N] <= 10 * carry[3] <= 10 - value[N]
```

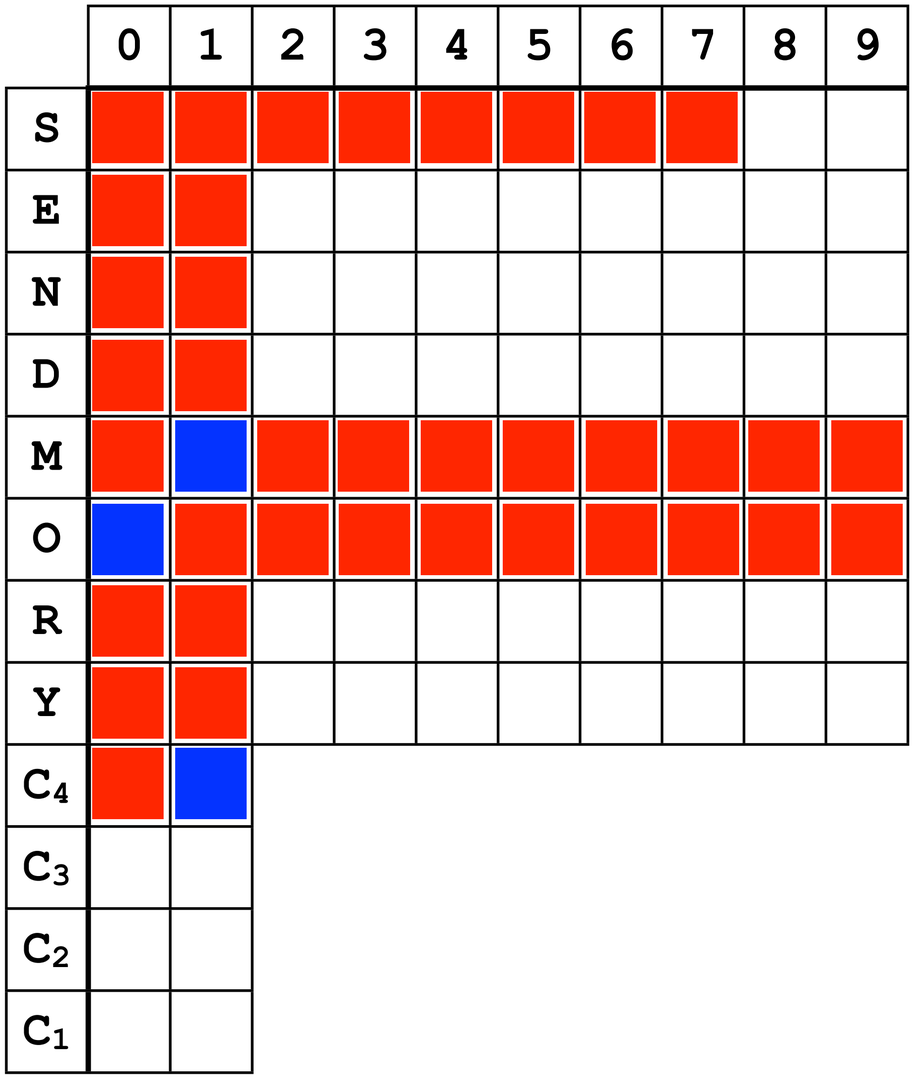
What is the Search Space?




```
2 <= value[N] + 10 * carry[3] <= 10
```

```
2 - value[N] <= 10 * carry[3] <= 10 - value[N]
```

What is the Search Space?

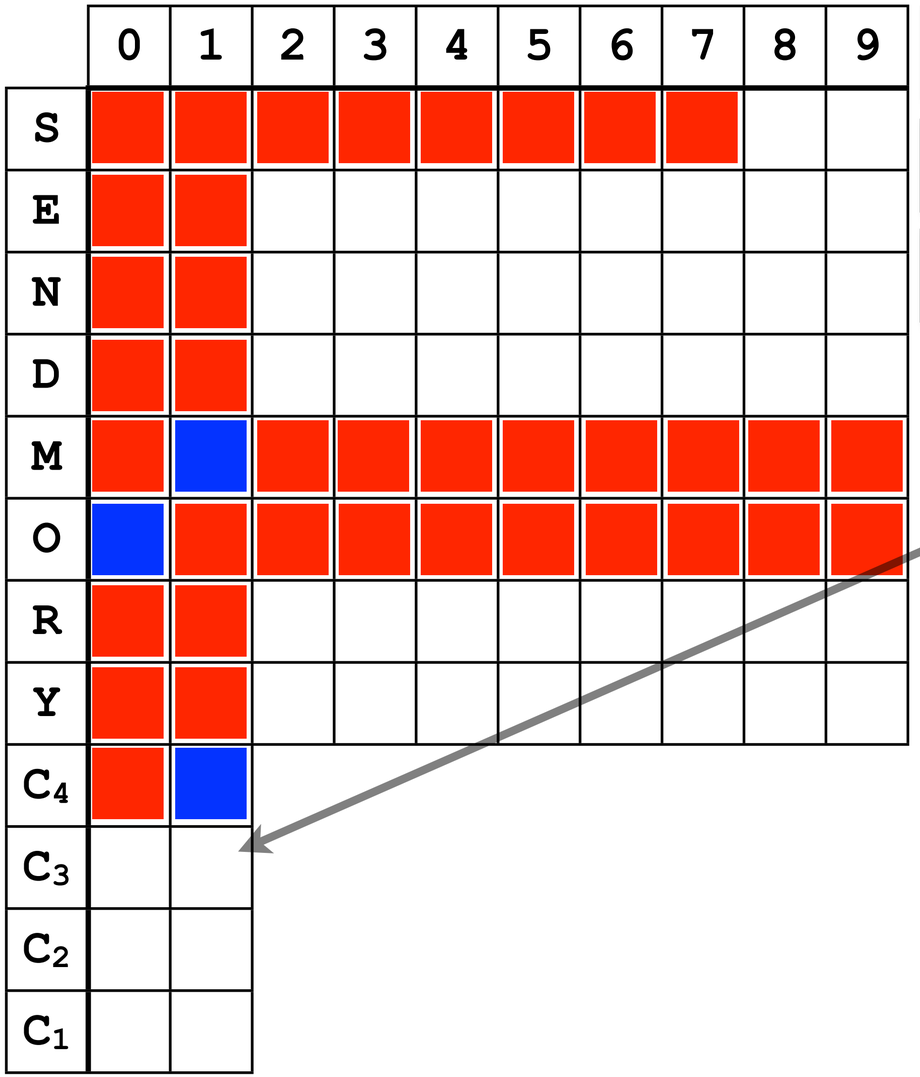



```
2 <= value[N] + 10 * carry[3] <= 10
```

```
2 - value[N] <= 10 * carry[3] <= 10 - value[N]
```

```
-7 <= 10 * carry[3] <= 8
```

What is the Search Space?

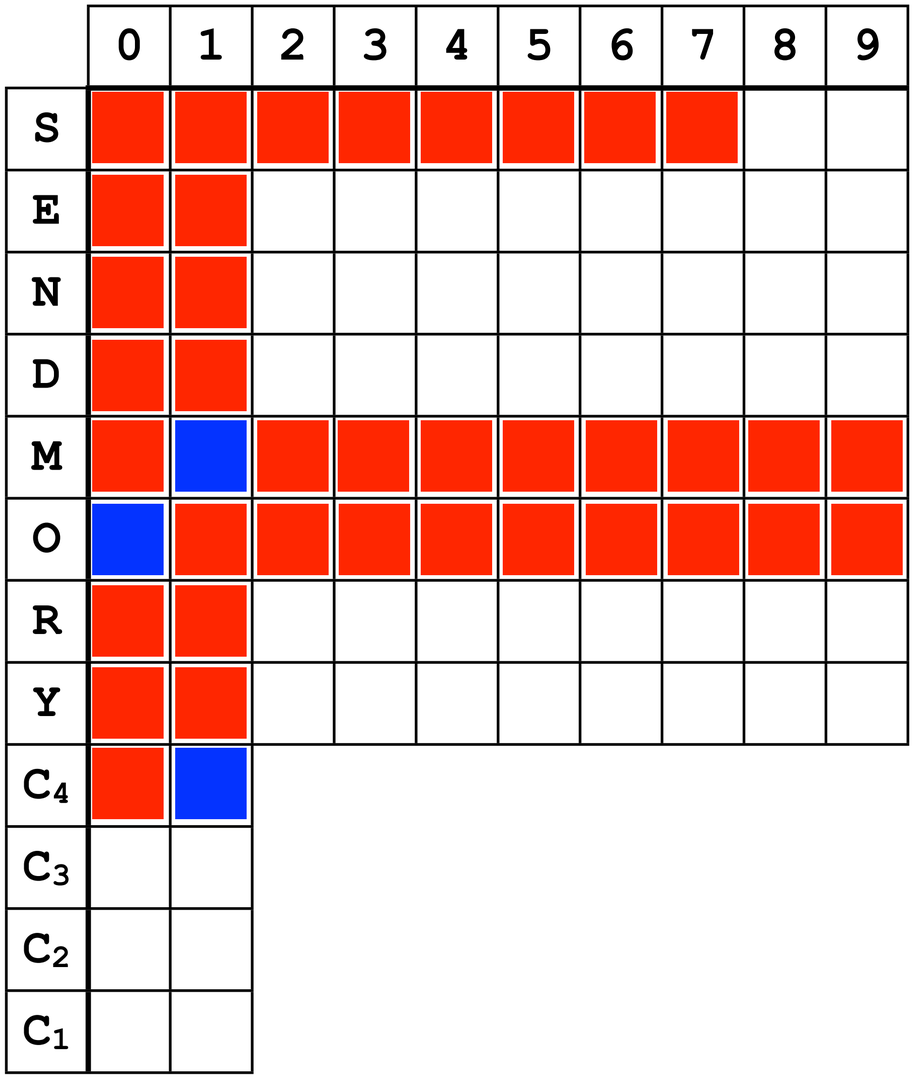



```
2 <= value[N] + 10 * carry[3] <= 10
```

```
2 - value[N] <= 10 * carry[3] <= 10 - value[N]
```

```
-7 <= 10 * carry[3] <= 8
```

What is the Search Space?

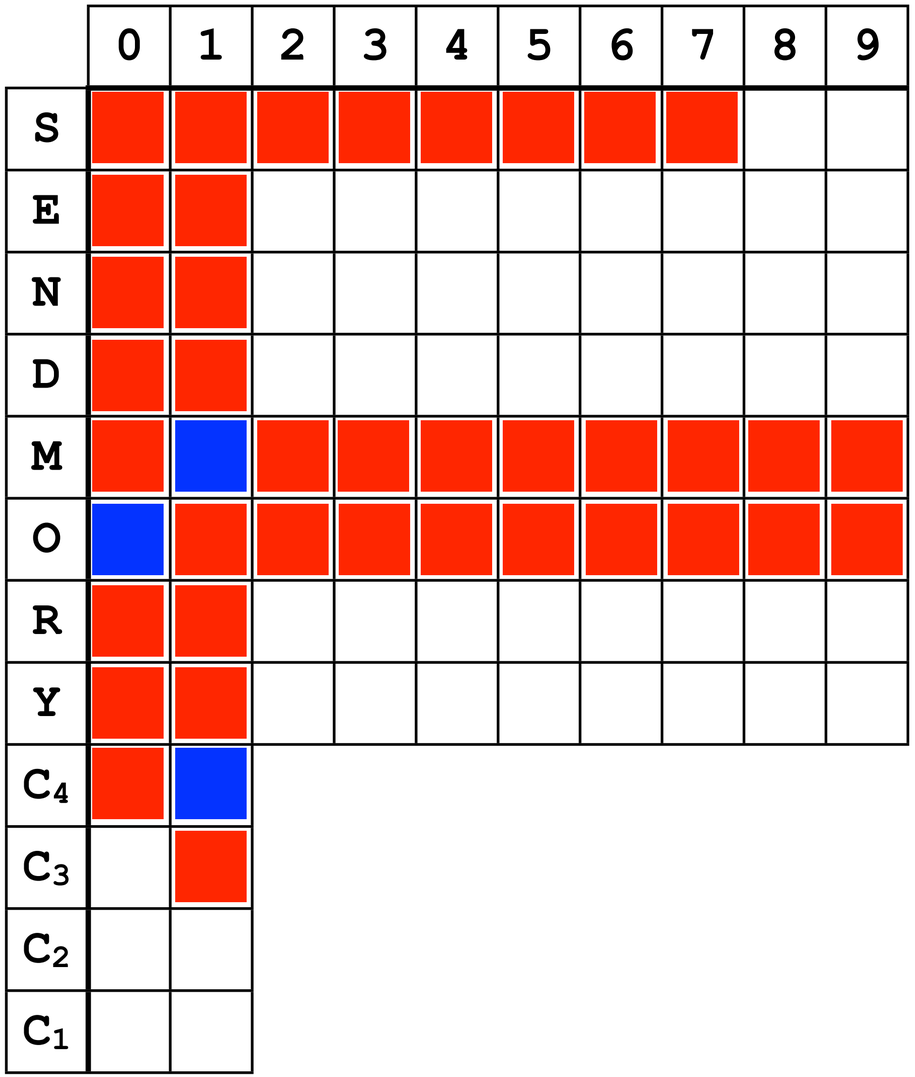



```
2 <= value[N] + 10 * carry[3] <= 10
```

```
2 - value[N] <= 10 * carry[3] <= 10 - value[N]
```

```
-7 <= 10 * carry[3] <= 8
```

What is the Search Space?

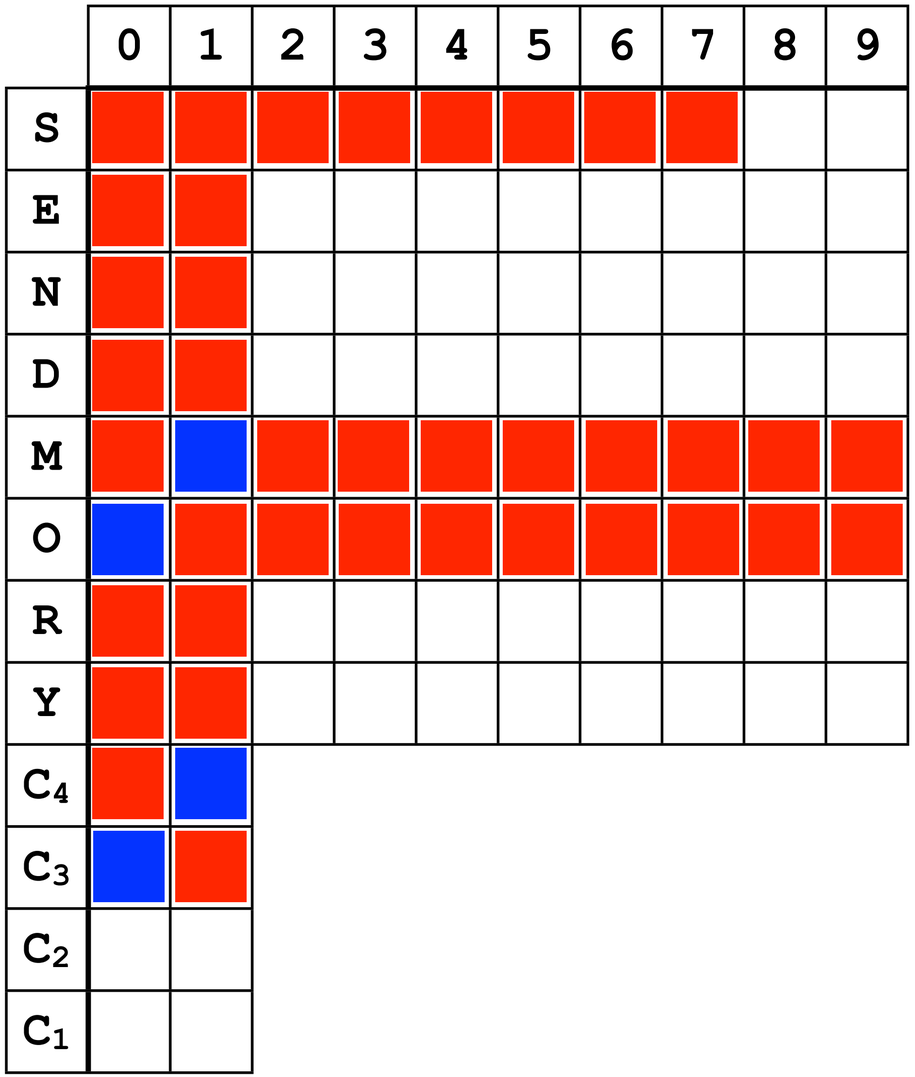



```
2 <= value[N] + 10 * carry[3] <= 10
```

```
2 - value[N] <= 10 * carry[3] <= 10 - value[N]
```

```
-7 <= 10 * carry[3] <= 8
```

What is the Search Space?

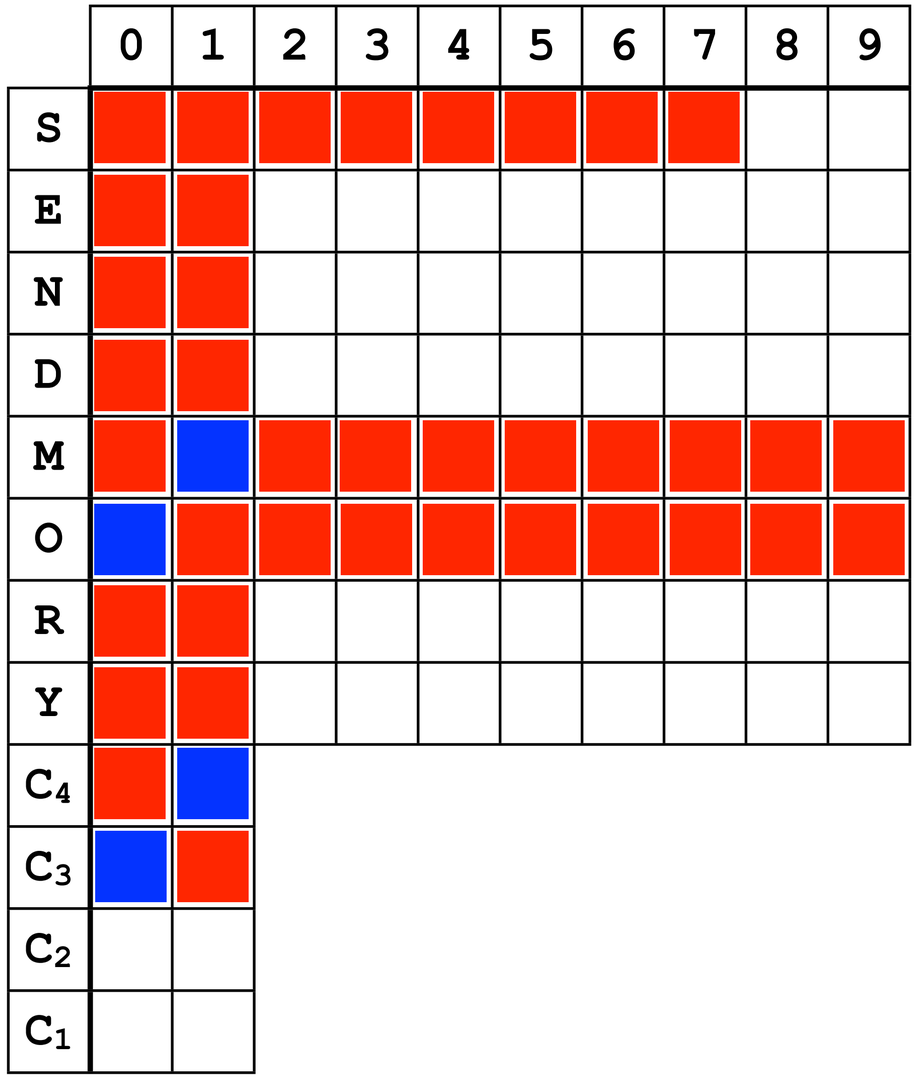



```
2 <= value[N] + 10 * carry[3] <= 10
```

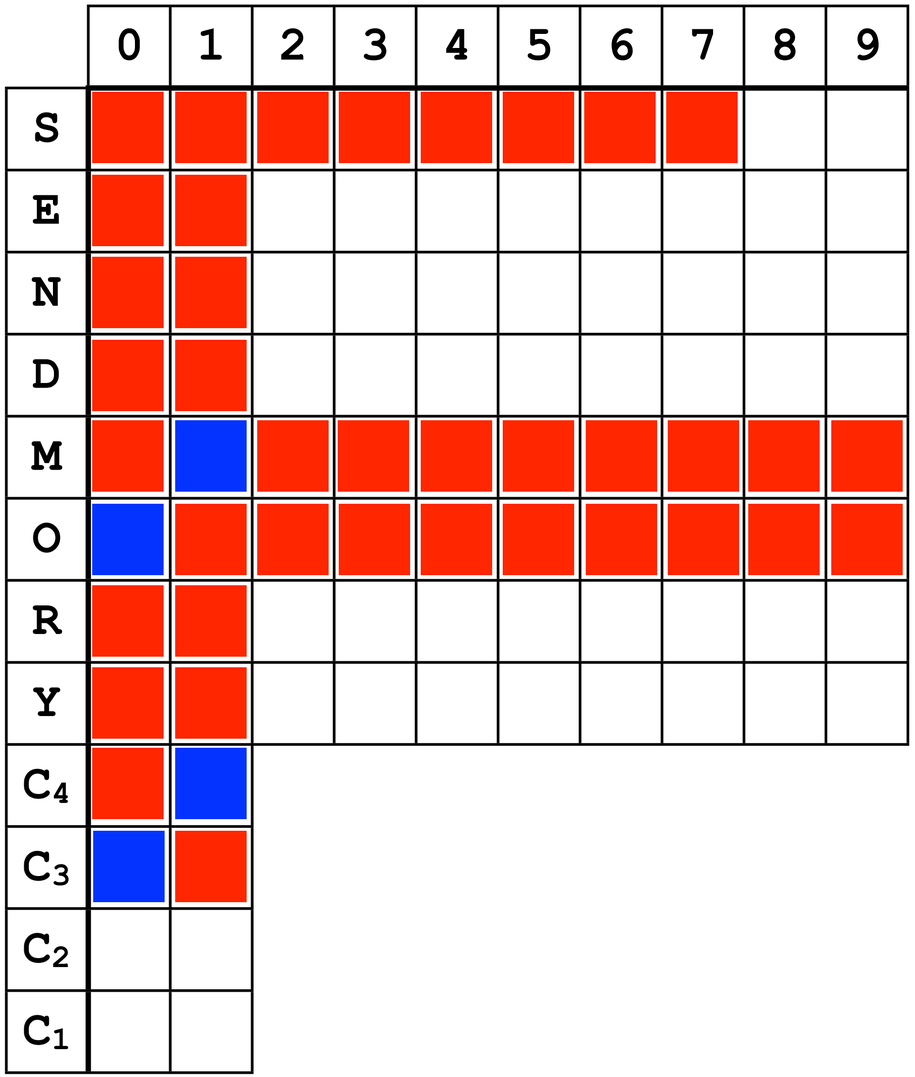
```
2 - value[N] <= 10 * carry[3] <= 10 - value[N]
```

```
-7 <= 10 * carry[3] <= 8
```

What is the Search Space?

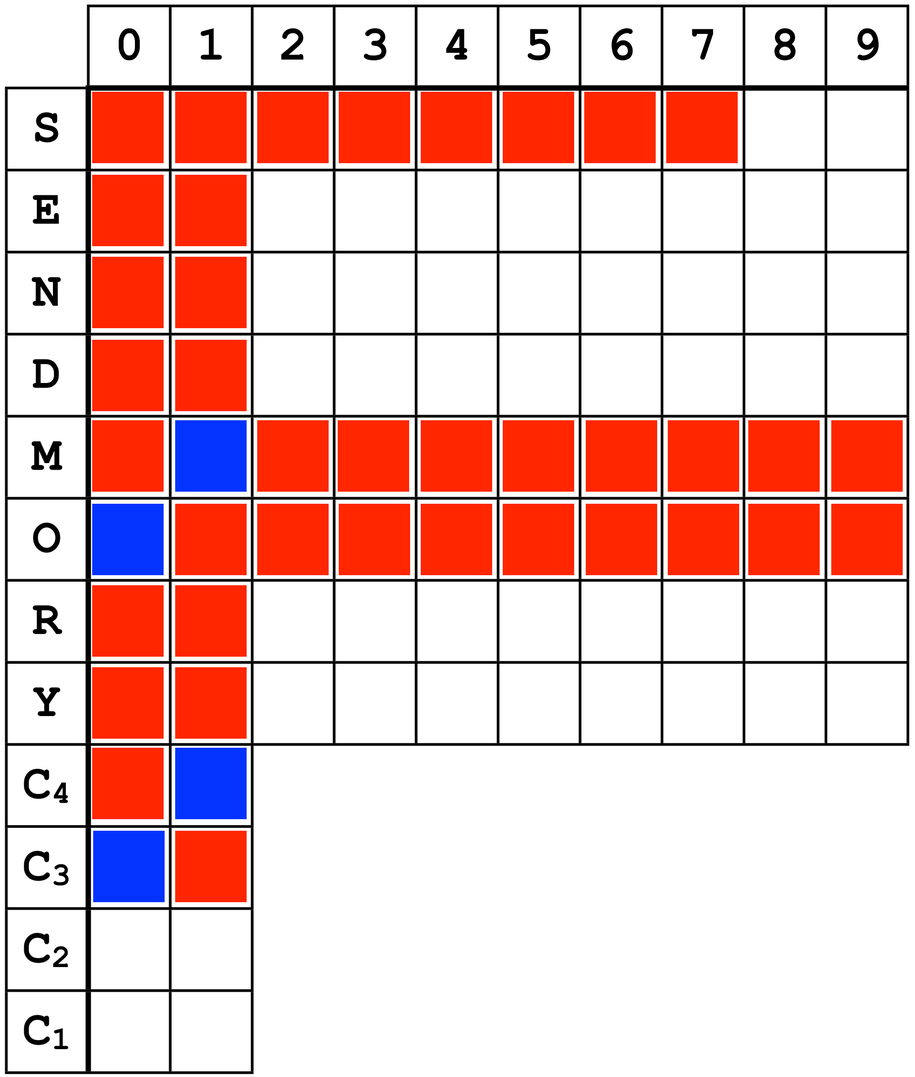


What is the Search Space?



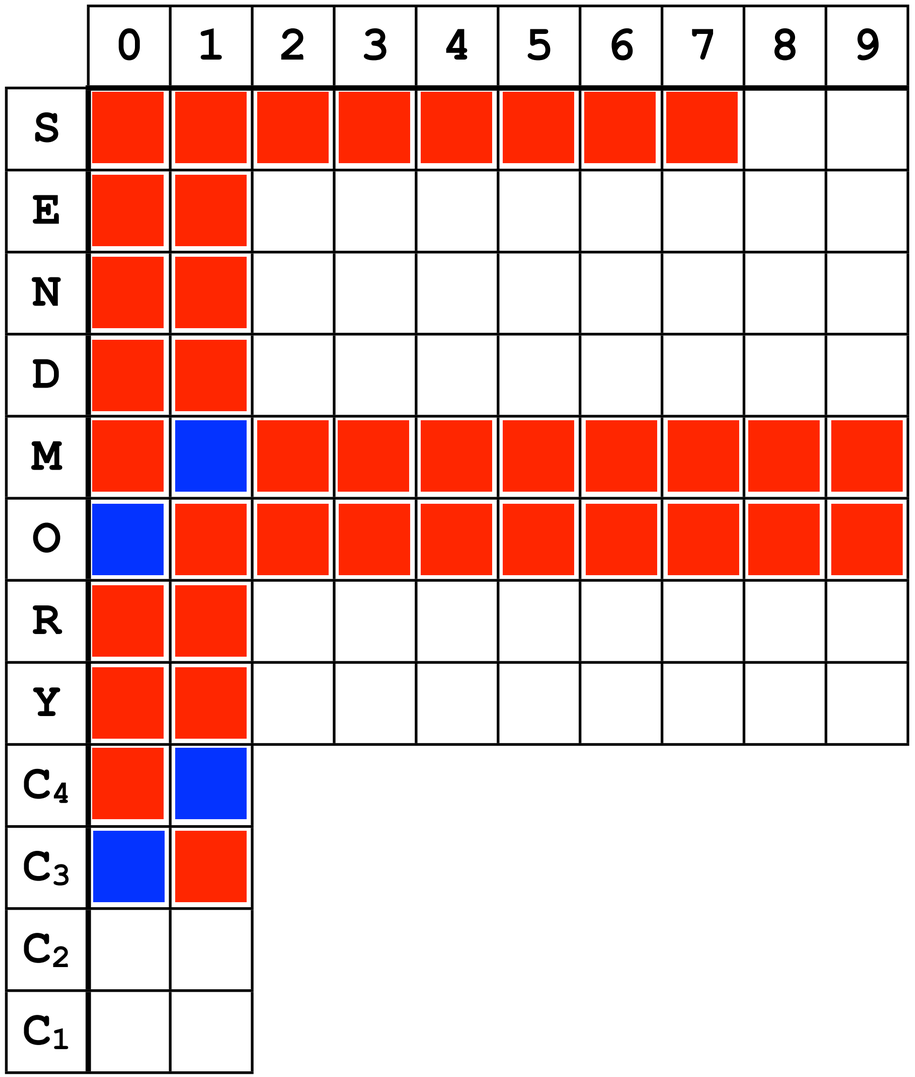

```
enum Letters = { S, E, N, D, M, O, R, Y};
range Digits = 0..9;
var{int} value[Letters] in Digits;
var{int} carry[1..4] in 0..1;

solve {
  forall(i in Letters, j in Letters: i < j)
     value[i] ≠ value[j];
  value[S] ≠ 0;
  value[M] ≠ 0;
  carry[4]                          = value[M];
  carry[3] + value[S] + value[M] = value[O] + 10 * carry[4];
  carry[2] + value[E] + value[O] = value[N] + 10 * carry[3];
  carry[1] + value[N] + value[R] = value[E] + 10 * carry[2];
             value[D] + value[E] = value[Y] + 10 * carry[1];
}
```

What is the Search Space?




```
enum Letters = { S, E, N, D, M, O, R, Y};
range Digits = 0..9;
var{int} value[Letters] in Digits;
var{int} carry[1..4] in 0..1;

solve {
  forall(i in Letters, j in Letters: i < j)
     value[i] ≠ value[j];
  value[S] ≠ 0;
  value[M] ≠ 0;
  carry[4]                          = value[M];
  carry[3] + value[S] + value[M] = value[O] + 10 * carry[4];
  carry[2] + value[E] + value[O] = value[N] + 10 * carry[3];
  carry[1] + value[N] + value[R] = value[E] + 10 * carry[2];
             value[D] + value[E] = value[Y] + 10 * carry[1];
}
```

What is the Search Space?



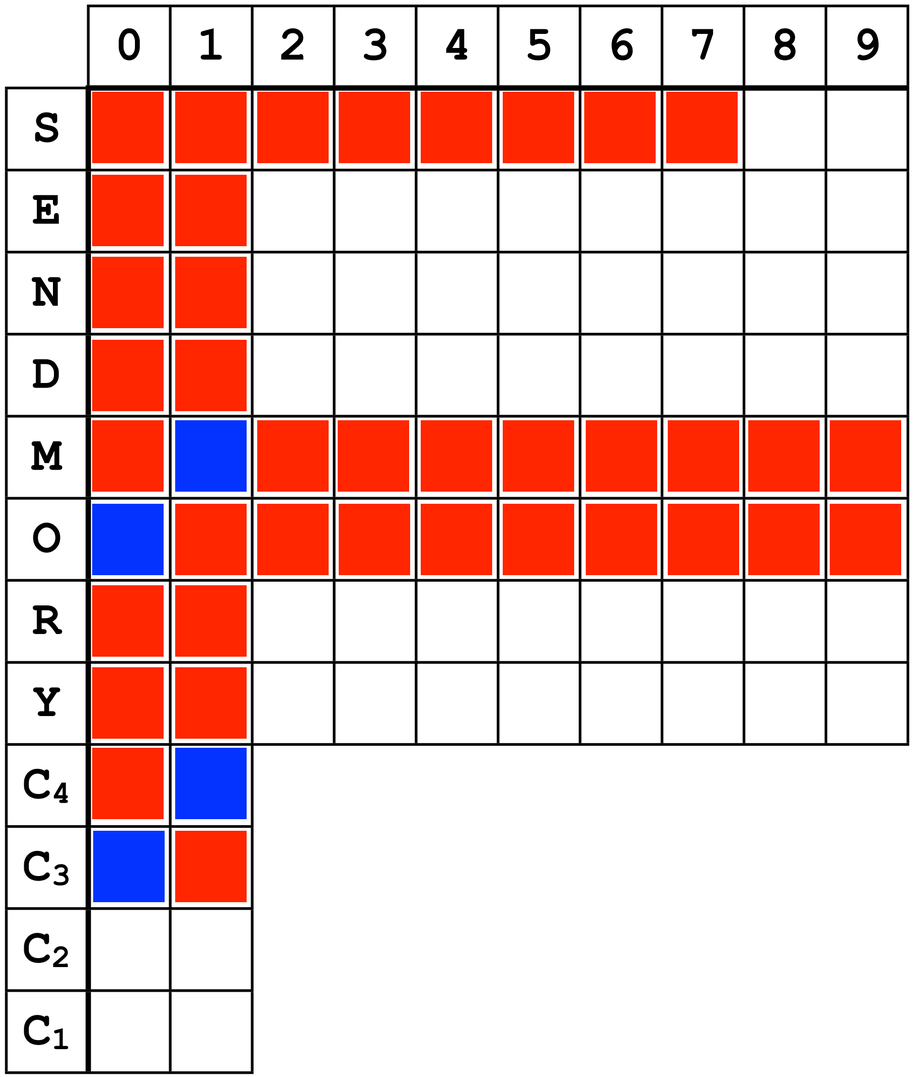

```
enum Letters = { S, E, N, D, M, O, R, Y};
range Digits = 0..9;
var{int} value[Letters] in Digits;
var{int} carry[1..4] in 0..1;

solve {
  forall(i in Letters, j in Letters: i < j)
     value[i] ≠ value[j];
  value[S] ≠ 0;
  value[M] ≠ 0;
  carry[4]                          = value[M];
  carry[3] + value[S] + value[M] = value[O] + 10 * carry[4];
  carry[2] + value[E] + value[O] = value[N] + 10 * carry[3];
  carry[1] + value[N] + value[R] = value[E] + 10 * carry[2];
           value[D] + value[E] = value[Y] + 10 * carry[1];
}
```

Computational Paradigm

- The propagation engine
  - this is the core of any constraint-programming solver
  - a simple (fixpoint) algorithm

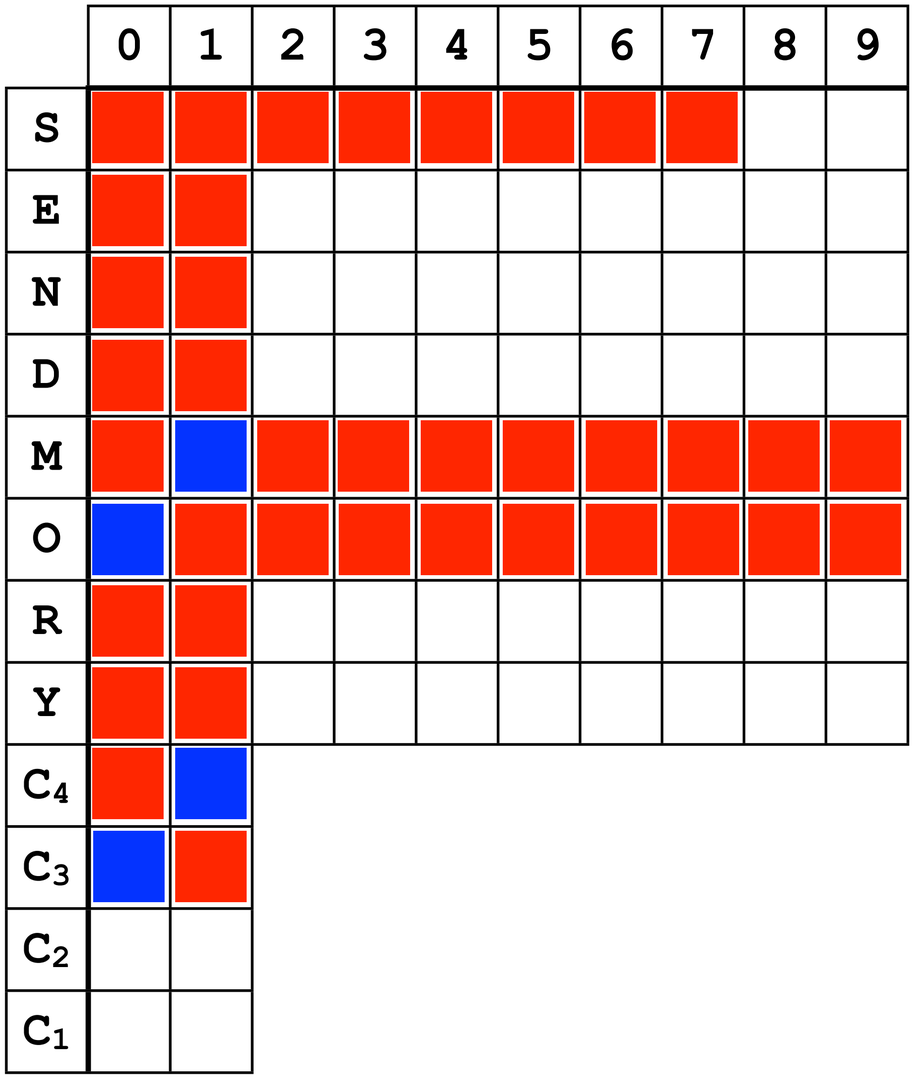
```
propagate()
{
  repeat
    select a constraint c;
    if c is infeasible given the domain store then
      return failure;
    else
      apply the pruning algorithm associated with c;
  until no constraint can remove any value from the
  domain of its variables;
  return success;
}
```

What is the Search Space?



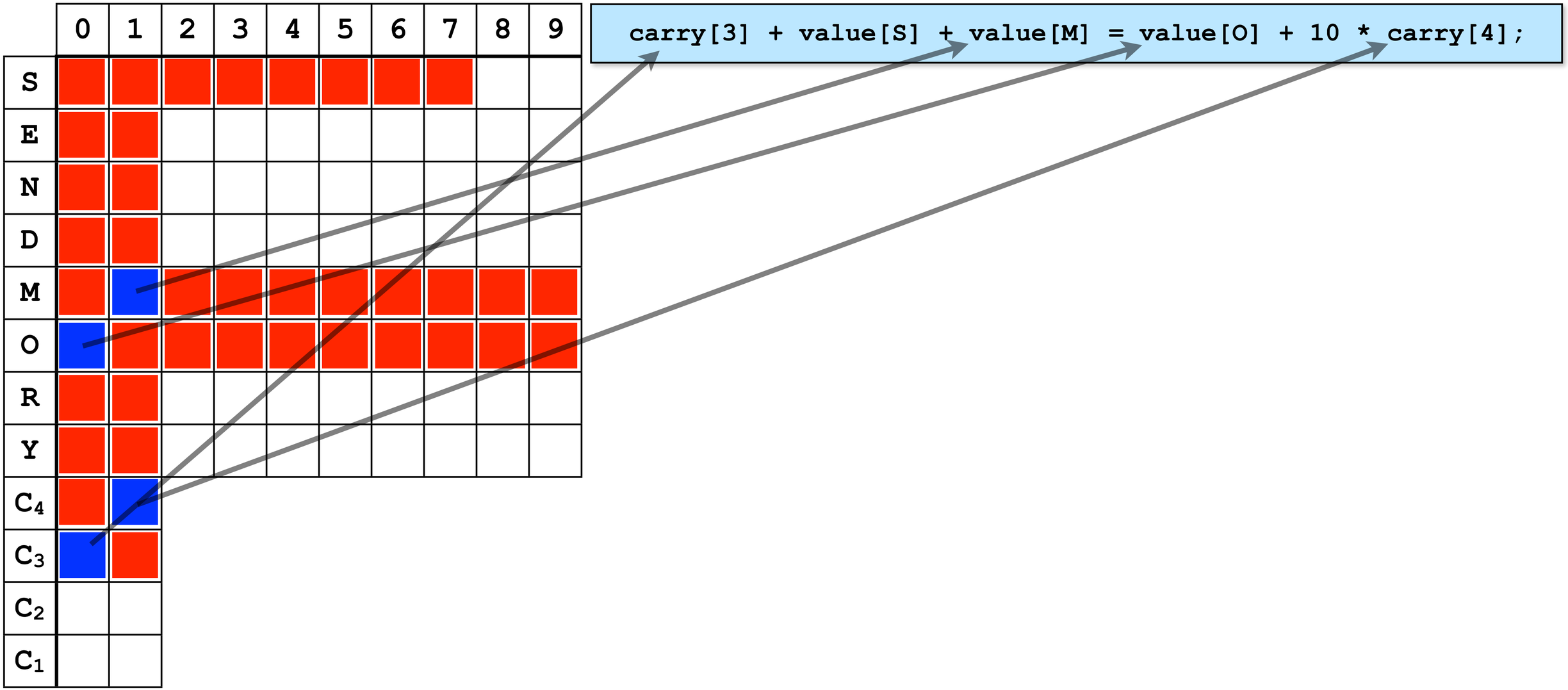

```
enum Letters = { S, E, N, D, M, O, R, Y};
range Digits = 0..9;
var{int} value[Letters] in Digits;
var{int} carry[1..4] in 0..1;

solve {
  forall(i in Letters, j in Letters: i < j)
     value[i] ≠ value[j];
  value[S] ≠ 0;
  value[M] ≠ 0;
  carry[4]                          = value[M];
  carry[3] + value[S] + value[M] = value[O] + 10 * carry[4];
  carry[2] + value[E] + value[O] = value[N] + 10 * carry[3];
  carry[1] + value[N] + value[R] = value[E] + 10 * carry[2];
           value[D] + value[E] = value[Y] + 10 * carry[1];
}
```

What is the Search Space?

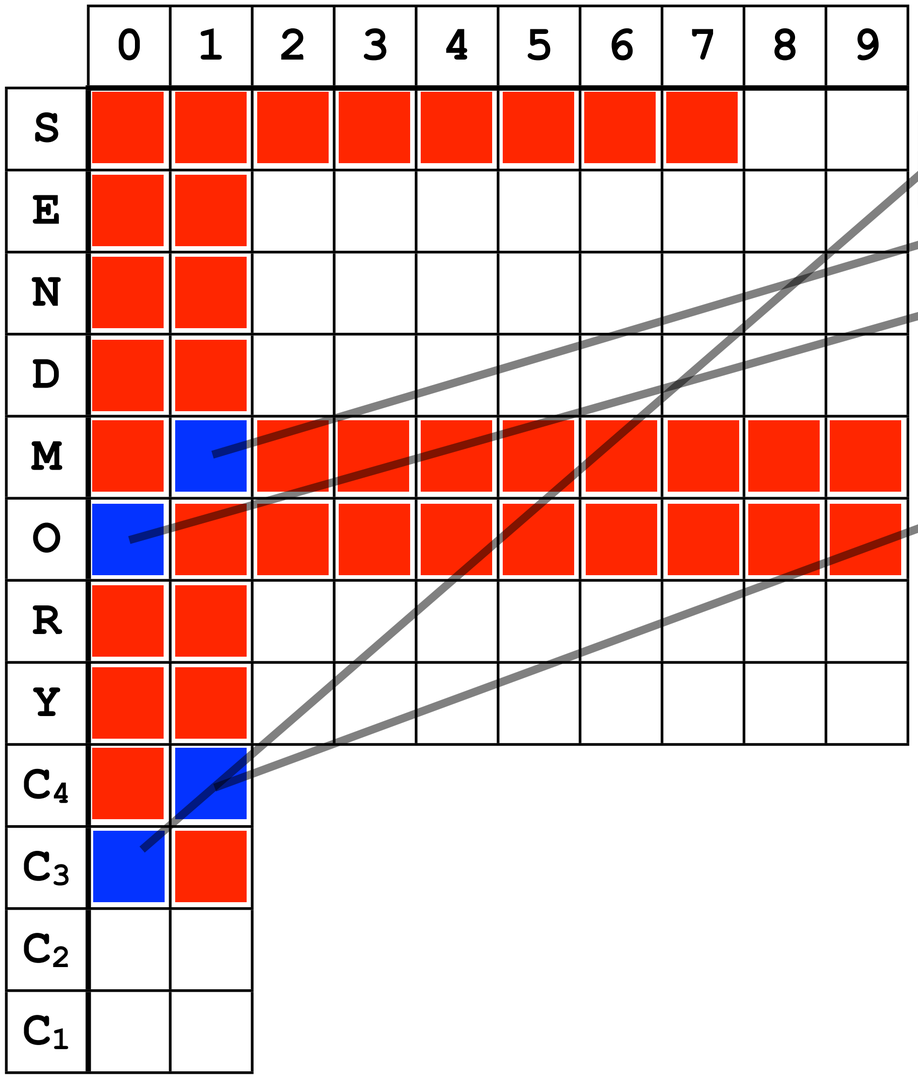



```
carry[3] + value[S] + value[M] = value[O] + 10 * carry[4];
```

What is the Search Space?



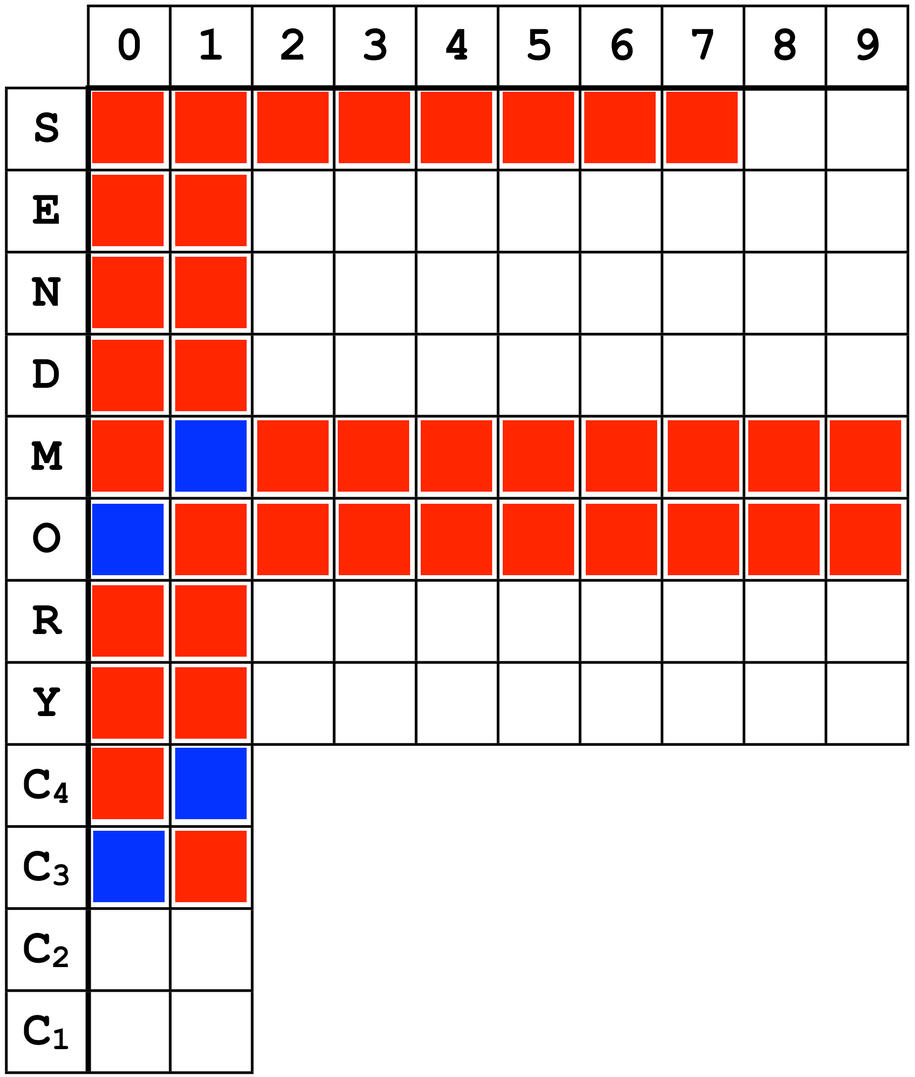
What is the Search Space?




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

```
0 + value[S] + 1        = 0        + 10 * 1;
```

What is the Search Space?

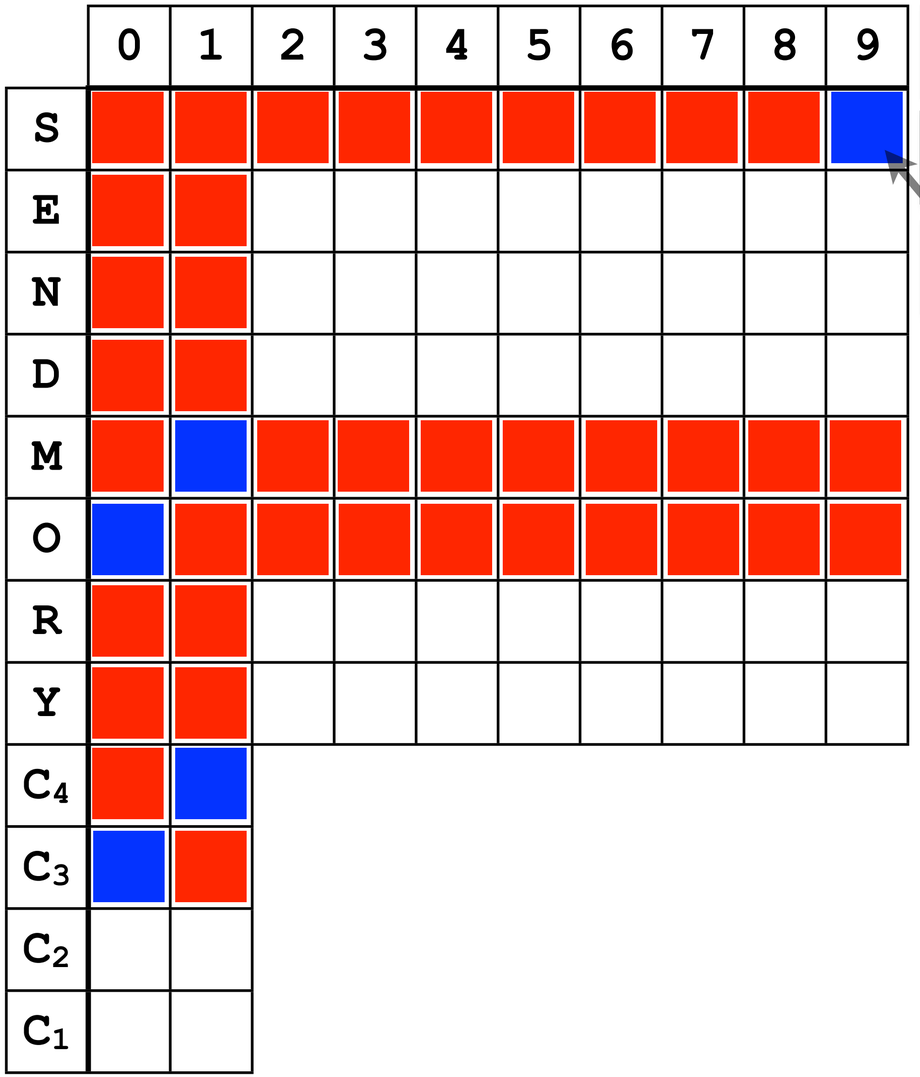



```
carry[3] + value[S] + value[M] = value[O] + 10 * carry[4];
```

```
        0 + value[S] + 1        = 0        + 10 * 1;
```

```
value[S] = 9;
```

What is the Search Space?

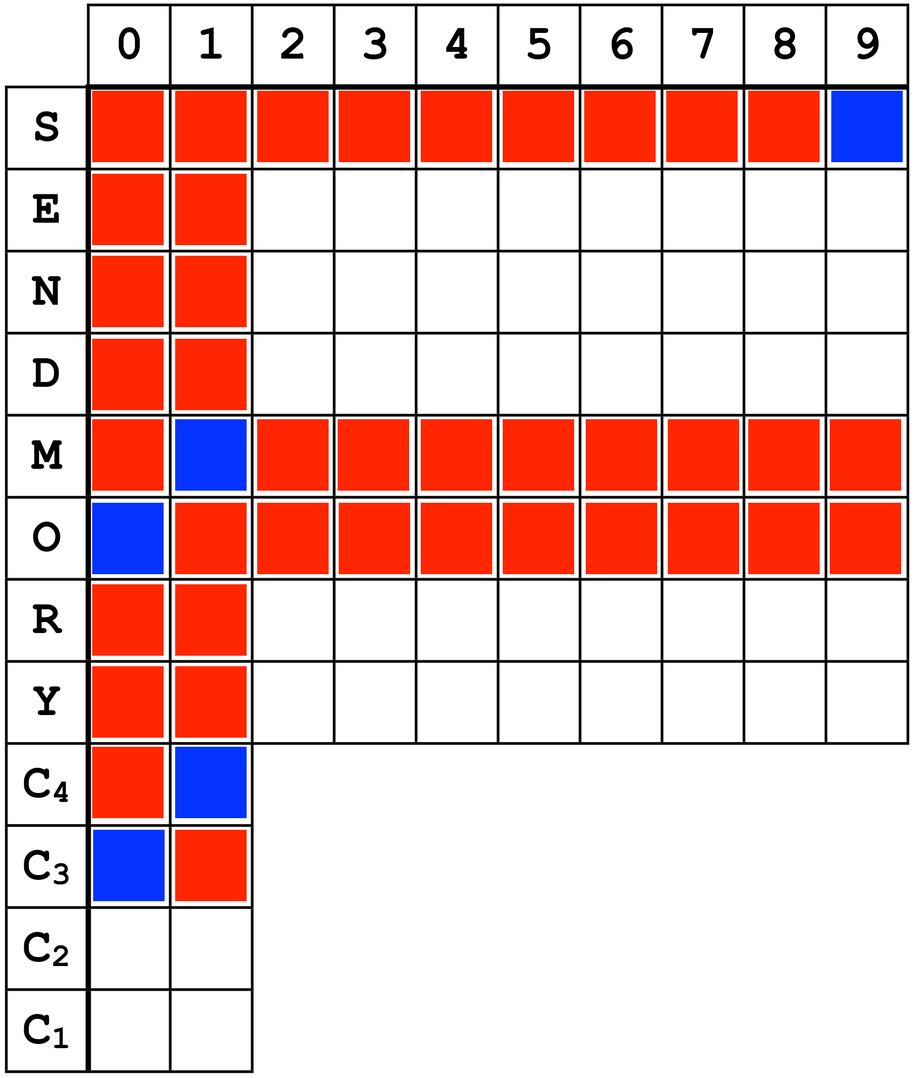



```
carry[3] + value[S] + value[M] = value[O] + 10 * carry[4];
```

```
        0 + value[S] + 1        = 0        + 10 * 1;
```

```
value[S] = 9;
```

What is the Search Space?



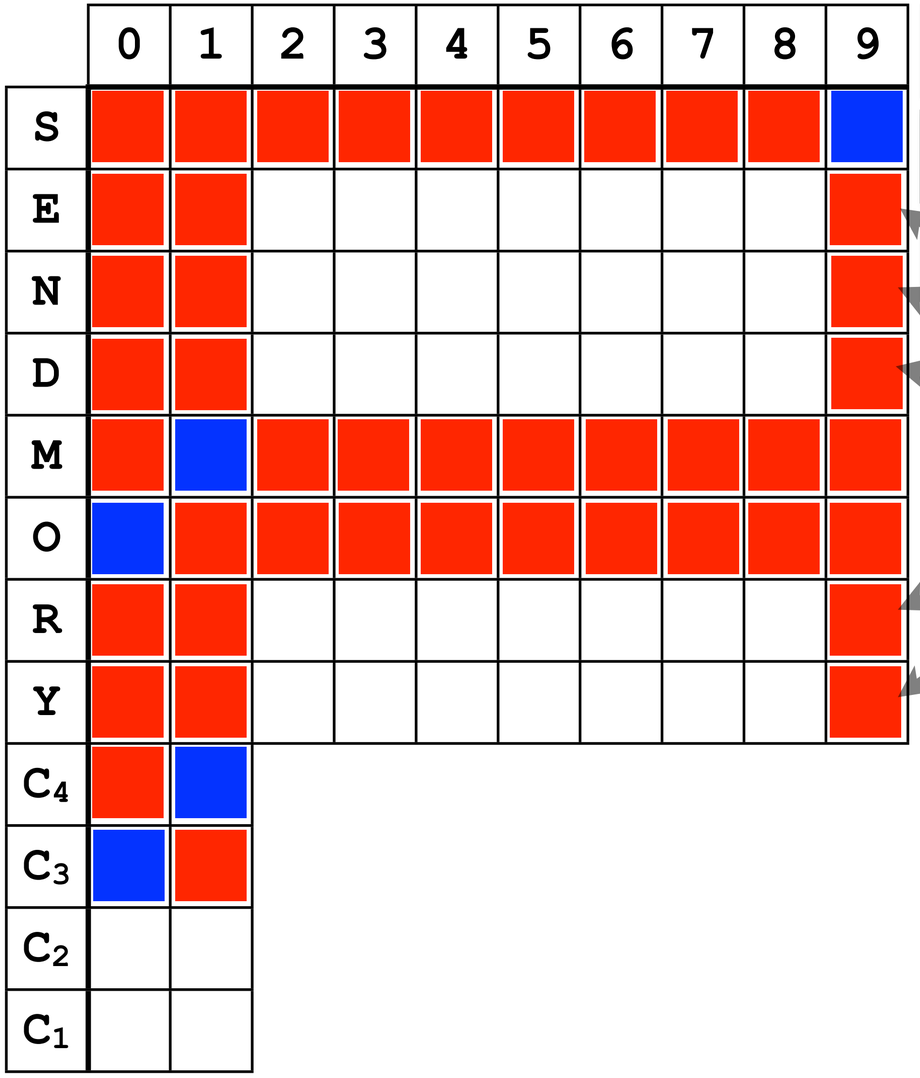

```
carry[3] + value[S] + value[M] = value[O] + 10 * carry[4];
```

```
0 + value[S] + 1        = 0        + 10 * 1;
```

```
value[S] = 9;
```

```
forall(i in Letters, j in Letters: i < j)
   value[i] ≠ value[j];
```

What is the Search Space?



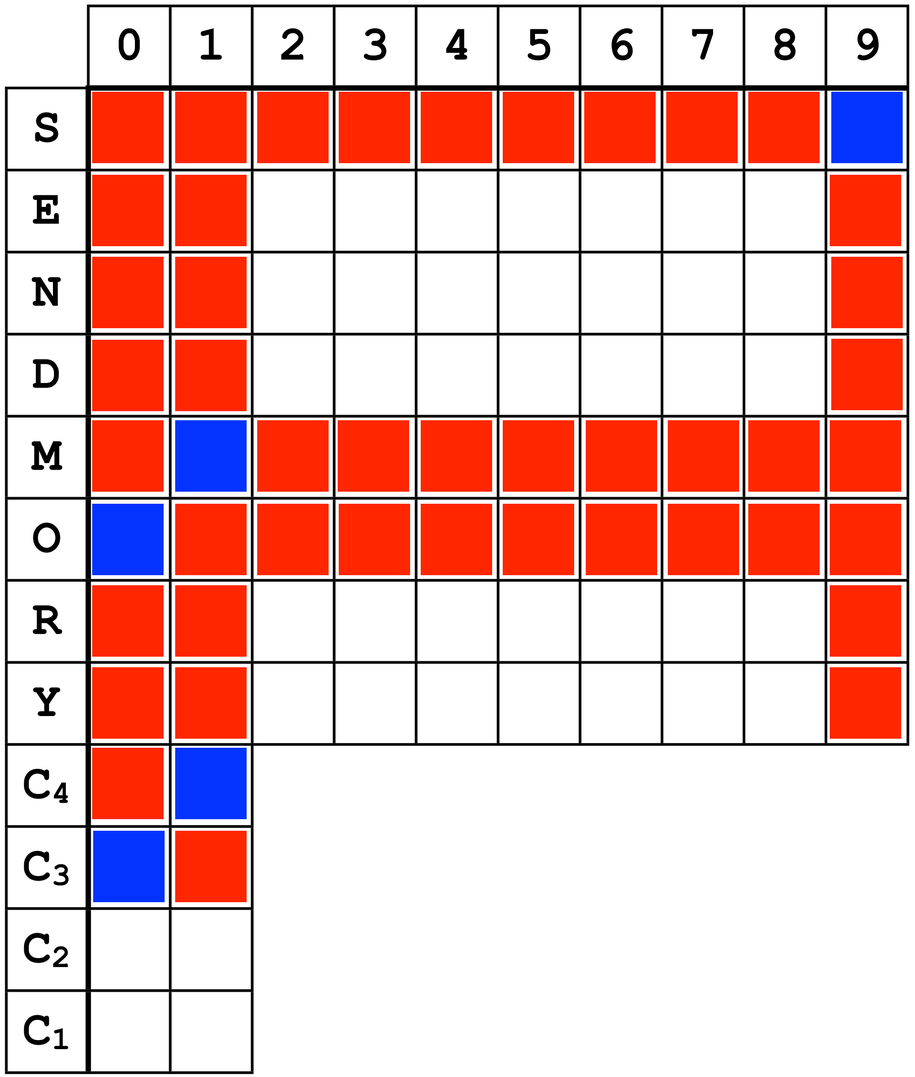

```
carry[3] + value[S] + value[M] = value[O] + 10 * carry[4];
```

```
0 + value[S] + 1        = 0        + 10 * 1;
```

```
value[S] = 9;
```

```
forall(i in Letters, j in Letters: i < j)
  value[i] ≠ value[j];
```

What is the Search Space?



Linear Constraints over Integers

- Consider a constraint

$$a_1x_1 + \ldots + a_nx_n \geq b_1y_1 + \ldots + b_my_m$$

$a_i, b_j \geq 0$ are constants

x_i, y_j are variables with domains $D(x_i), D(y_j)$

Linear Constraints over Integers

- Consider a constraint

$$a_1x_1 + \ldots + a_nx_n \geq b_1y_1 + \ldots + b_my_m$$

$a_i, b_j \geq 0$ are constants

x_i, y_j are variables with domains $D(x_i), D(y_j)$

- Feasibility test

Linear Constraints over Integers

- Consider a constraint

$$a_1x_1 + \ldots + a_nx_n \geq b_1y_1 + \ldots + b_my_m$$

$a_i, b_j \geq 0$ are constants

x_i, y_j are variables with domains $D(x_i), D(y_j)$

- Feasibility test

$$a_1 \max(D(x_1)) + \ldots + a_n \max(D(x_n)) \geq b_1 \min(D(y_1)) + \ldots + b_m \min(D(y_m))$$

Linear Constraints over Integers

- Consider a constraint

$$a_1x_1 + \ldots + a_nx_n \geq b_1y_1 + \ldots + b_my_m$$

$a_i, b_j \geq 0$ are constants

x_i, y_j are variables with domains $D(x_i), D(y_j)$

- Feasibility test

$$a_1 \max(D(x_1)) + \ldots + a_n \max(D(x_n)) \geq b_1 \min(D(y_1)) + \ldots + b_m \min(D(y_m))$$

$$\boxed{\begin{aligned} l &= a_1 \max(D(x_1)) + \ldots + a_n \max(D(x_n)) \\ r &= b_1 \min(D(y_1)) + \ldots + b_m \min(D(y_m)) \end{aligned}}$$

Linear Constraints over Integers

- Consider a constraint

$$a_1x_1 + \ldots + a_nx_n \geq b_1y_1 + \ldots + b_my_m$$

$a_i, b_j \geq 0$ are constants

x_i, y_j are variables with domains $D(x_i), D(y_j)$

- Pruning

$$a_ix_i \geq r - (l - a_i \max(D(x_i))$$

$$x_i \geq \left\lceil \frac{r - (l - a_i \max(D(x_i))}{a_i} \right\rceil$$

$$y_j \leq \left\lfloor \frac{l - (r - b_j \min(D(y_j))}{b_j} \right\rfloor$$

Until Next Time

Discrete Optimization

Constraint Programming: Part III

Goals of the Lecture

- Illustrating the rich modeling language of constraint programming
- Key aspect of constraint programming
  - ability to state complex, idiosyncratic constraints

Magic Series

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

	0	1	2	3	4
Occurrences	2	1	2	0	0

- Can you find a magic series?

Magic Series and Reification

```
int n = 5;
range D = 0..n-1;
var{int} series[D] in D;
solve {
   forall(k in D)
     series[k] = sum(i in D) (series[i]=k);
}
```

- Reification
  - the ability to transform a constraint into a 0/1 variable
  - the variable has the value 1 if the constraint is true and 0 otherwise

Magic Series and Reification

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

	0	1	2	3	4
Occurrences	?	?	?	?	?

Magic Series and Reification

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

- But what is series[1] ?

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

Reification

- What is happening behind the scene?

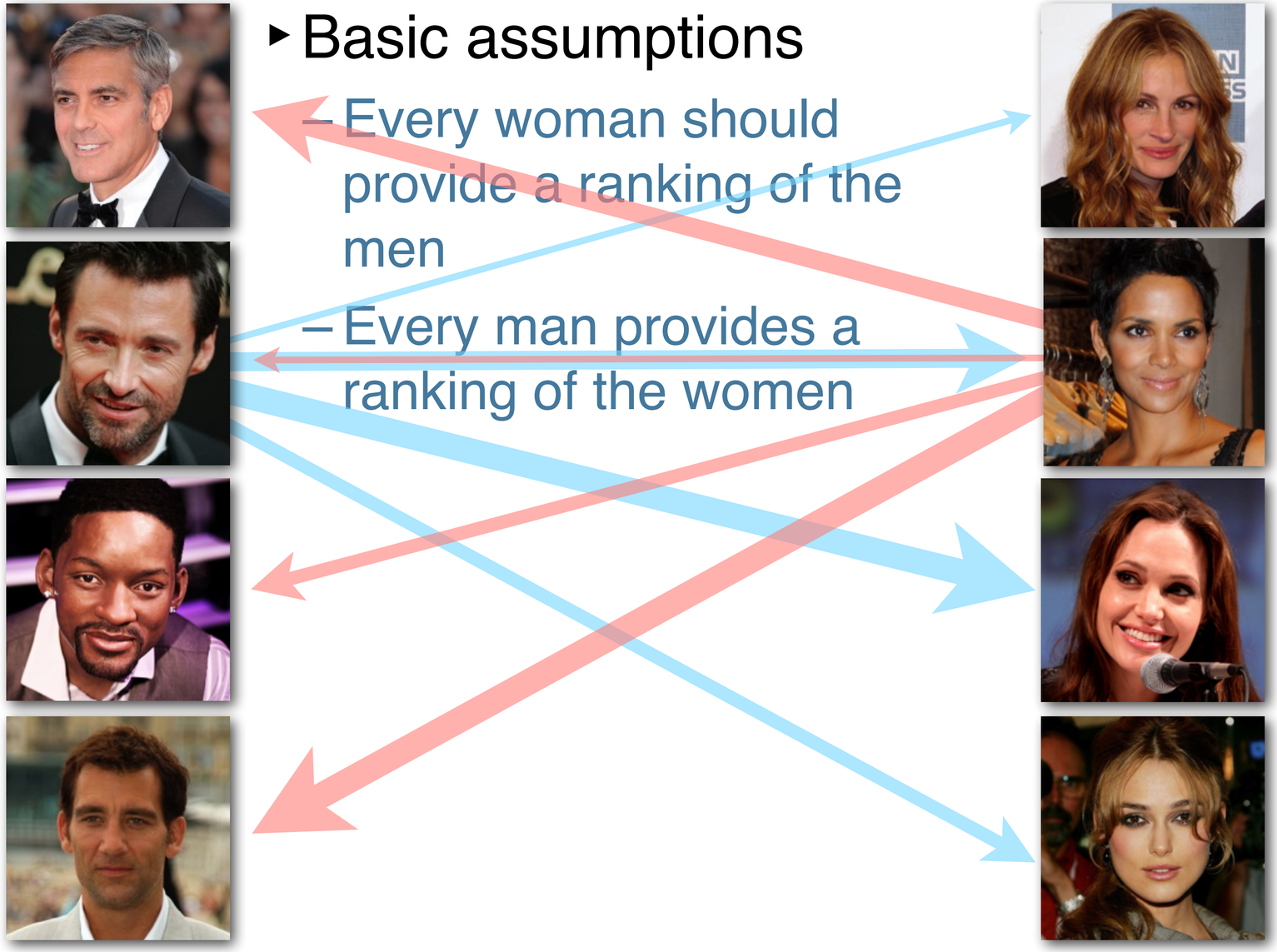
```
int n = 5;
range D = 0..n-1;
var{int} series[D] in D;
solve {
   forall(k in D) {
      var{int} b[D] in 0..1;
      forall(i in D)
         booleq(b[i],series[i],k);
      series[k] = sum(i in D) b[i];
   }
}
```

- Constraint booleq(b,x,v) holds
  if (b=1 and x=v) or (b=0 and x≠v).

$$\text{booleq}(b, x, v) \Leftrightarrow (b = 1 \wedge x = v) \vee (b = 0 \wedge x \neq v)$$

Stable Marriages

- Basic assumptions
  - Every woman should provide a ranking of the men
  - Every man provides a ranking of the women



Stable Marriages

- A marriage between Hugh and Angelina is stable provided that
  - If Angelina prefers another man, say George, over Hugh, then George must prefer his spouse over Angelina
  - If Hugh prefers another woman, say Julia, over Angelina, then Julia must prefer her spouse over Hugh

- These stability rules make the marriage stable!

Stable Marriages

- What are the decision variables?

Stable Marriages

- Data and decision variables

`wrank[Hugh,Julia]` is the ranking of Julia in Hugh's preferences

```
enum Men = {George,Hugh,Will,Clive};
enum Women = {Julia,Halle,Angelina,Keira};

int wrank[Men,Women];
int mrank[Women,Men];
...

var{Women} wife[Men];
var{Men}   husband[Women];
```

`mrank[Julia,Hugh]` is the ranking of Hugh in Julia's preference

Stable Marriages

```
solve {
   forall(m in Men)
      husband[wife[m]] = m;
   forall(w in Women)
      wife[husband[w]] = w;

   ...
}
```

Stable Marriages

```
solve {
   forall(m in Men)
      husband[wif
   forall(w in Wo
      wife[husband[w]] = w;

   forall(m in Men, w in Women)
      wrank[m,w] < wrank[w,wife[m]] => mrank[w,husband[w]] < mrank[w,m];
   forall(w in Women, m in Men)
      mrank[w,m] < mrank[w,husband[m]] => wrank[m,wife[m]] < mrank[m,w];
}
```

m prefers w over his wife

Stable Marriages

```
solve {
   forall(m in Men)
      husband[wife[m]] = m;
   forall(w in Women)
      wife[husband[w]] = w;

   forall(m in Men, w in Women)
      wrank[m,w] < wrank[w,wife[m]] => mrank[w,husband[w]] < mrank[w,m];
   forall(w in Women, m in Men)
      mrank[w,m] < mrank[w,husband[m]] => wrank[m,wife[m]] < mrank[m,w];
}
```

w prefers her husband over m

Stable Marriages

```
enum Men = {George,Hugh,Will,Clive};
enum Women = {Julia,Halle,Angelina,Keira};
int wrank[Men,Women];
int mrank[Women,Men];
...
var{Women} wife[Men];
var{Men}   husband[Women];
solve {
   forall(m in Men)
      husband[wife[m]] = m;
   forall(w in Women)
      wife[husband[w]] = w;

   forall(m in Men, w in Women)
      wrank[m,w] < wrank[w,wife[m]] => mrank[w,husband[w]] < mrank[w,m];
   forall(w in Women, m in Men)
      mrank[w,m] < mrank[w,husband[m]] => wrank[m,wife[m]] < mrank[m,w];
}
```

Stable Marriages

- Two interesting features
  - Element constraint
    - useful in many applications
  - Logical combination of constraints
- The element constraint
  - ability to index an array/matrix with a variable or an expression containing variables
- Logical combination of constraints
  - can be handled by reification for instance

The Basic Element Constraint

- x, y: variables
- c is an array of integers
- constraint x = c[y]

Y $\in \{0, 1, 2, 3, 4, 5\}$

X $\in \{3, 4, 5\}$

Array c

i	0	1	2	3	4	5
c[i]	3	4	5	5	4	3

$X \neq 3$

$Y \neq 4$

$Y \neq 1$

Citations

66ème Festival de Venise (Mostra)(http://www.flickr.com/photos/nicogenin/3902889281/) by Nicolas Genin (http://www.flickr.com/photos/nicogenin/) CC BY-SA 2.0 (http://creativecommons.org/licenses/by-sa/2.0/deed.en)

Hugh Jackman (http://www.flickr.com/photos/58820009@N05/8294109236) by Eva Rinaldi (http://www.flickr.com/photos/evarinaldiphotography/) CC BY-SA 2.0 (http://creativecommons.org/licenses/by-sa/2.0/deed.en)

Wil Smith (http://www.flickr.com/photos/sandrascherer/6775228280/) by Sandra Scherer (http://www.flickr.com/people/sandrascherer/) CC BY-SA 2.0 (http://creativecommons.org/licenses/by-sa/2.0/deed.en)

Clive Owen, (http://www.flickr.com/photos/oneras/251106343/) by Mario Antonio Pena Zapateira (http://www.flickr.com/photos/oneras/) CC BY-SA 2.0 (http://creativecommons.org/licenses/by-sa/2.0/deed.en)

Julia Roberts 2011 Shankbone (http://www.flickr.com/photos/shankbone/5656729480/) by David Shankbone (http://www.flickr.com/people/shankbone/) CC BY 2.0 (http://creativecommons.org/licenses/by/2.0/deed.en)

Angelina Jolie (http://www.flickr.com/photos/gageskidmore/4860509634/) by Gage Skidmore (http://www.flickr.com/photos/gageskidmore/) CC BY-SA 2.0 (http://creativecommons.org/licenses/by-sa/2.0/deed.en)

Halle Berry 10 by German Marin [CC BY 3.0 (http://creativecommons.org/licenses/by/3.0)], via Wikimedia Commons (http://upload.wikimedia.org/wikipedia/commons/f/f8/Halle_Berry_10.jpg)

Keira Knightley (http://www.flickr.com/photos/tonyshek/7271339306/) by Tony Shek (http://www.flickr.com/photos/tonyshek/) CC BY-SA 2.0 (http://creativecommons.org/licenses/by-sa/2.0/deed.en)

Discrete Optimization

Constraint Programming: Part IV

Goals of the Lecture

- Illustrating the rich modeling language of constraint programming
- Key aspect of constraint programming
  - ability to state complex, idiosyncratic constraints

Global Constraints

- Critical features of constraint programming
  - capture combinatorial substructures arising in many applications

Global Constraints

- Critical features of constraint programming
  - capture combinatorial substructures arising in many applications
- Modeling
  - make modeling easier and more natural

Global Constraints

- Critical features of constraint programming
  - capture combinatorial substructures arising in many applications
- Modeling
  - make modeling easier and more natural
- Problem solving
  - convey the problem structure to the solver that does not have to rediscover it
  - give the ability to exploit dedicated algorithms

Global Constraints - AllDifferent

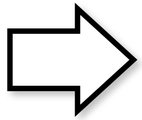
- alldifferent(x_1,...,x_n)
  - specifies that x_1,...,x_n take values that are different

```
range R = 1..8;
var{int} row[R] in R;
solve {
   forall(i in R,j in R: i < j) {
      row[i] ≠  row[j];
      row[i] + i ≠  row[j] + j;
      row[i] - i ≠  row[j] - j;
   }
}
```

Global Constraints - AllDifferent

- alldifferent(x_1,...,x_n)
  - specifies that x_1,...,x_n take values that are different

```
range R = 1..8;
var{int} row[R] in R;
solve {
   forall(i in R,j in R: i < j) {
      row[i] ≠  row[j];
      row[i] + i ≠  row[j] + j;
      row[i] - i ≠  row[j] - j;
   }
}
```



```
range R = 1..8;
var{int} row[R] in R;
solve {
   alldifferent(row);
   alldifferent(all(i in R) row[i]+i);
   alldifferent(all(i in R) row[i]-i);
}
```

Global Constraints - AllDifferent

- alldifferent(x_1,...,x_n)
  - specifies that x_1,...,x_n take values that are different

```
range R = 1..8;
var{int} row[R] in R;
solve {
   forall(i in R,j in R: i < j) {
      row[i] ≠  row[j];
      row[i] + i ≠  row[j] + j;
      row[i] - i ≠  row[j] - j;
   }
}
```

```
range R = 1..8;
var{int} row[R] in R;
solve {
   alldifferent(row);
   alldifferent(all(i in R) row[i]+i);
   alldifferent(all(i in R) row[i]-i);
}
```

array comprehension: collect all the elements in an array

Why global constraints?

- Given
  - a constraint $c(x_1,...,x_n)$
  - x_1 in D_1, ..., x_n in D_n

- Feasibility testing
  - Can I find values in the variable domains such that the constraint holds?

Why global constraints?

- Given
  - a constraint $c(x_1,\ldots,x_n)$
  - x_1 in D_1, ..., x_n in D_n

$$c(x_1, \ldots, x_n)$$
$$D_1 = D(x_1), \ldots, D_n = D(x_n)$$

- Feasibility testing
  - Can I find values in the variable domains such that the constraint holds?

Why global constraints?

- Given
  - a constraint $c(x_1,\ldots,x_n)$
  - x_1 in D_1, ..., x_n in D_n

$$c(x_1, \ldots, x_n)$$
$$D_1 = D(x_1), \ldots, D_n = D(x_n)$$

- Feasibility testing
  - Can I find values in the variable domains such that the constraint holds?

Why global constraints?

- Given
  - a constraint $c(x_1,\ldots,x_n)$
  - x_1 in D_1, ..., x_n in D_n

$$\mathrm{c}(x_1, \ldots, x_n)$$
$$D_1 = D(x_1), \ldots, D_n = D(x_n)$$

- Feasibility testing
  - Can I find values in the variable domains such that the constraint holds?

$$\exists\, v_1 \in D_1, \ldots, v_n \in D_n :$$
$$\mathrm{c}(x_1 = v_1, \ldots, x_n = v_n) = \mathrm{true}$$

Why Global Constraints?

- Given
  - a constraint alldifferent(x_1,...,x_3)
  - x_1 in [1..2], ..., x_3 in [1..2]

- Is this feasible?
  - No, only two values for 3 variables (pigeon hole principle)

Why Global Constraints?

- Given
  - a constraint alldifferent(x_1,...,x_3)
  - x_1 in [1..2], ..., x_3 in [1..2]

$$\text{alldifferent}(x_1, x_2, x_3)$$
$$D_1 = \{1, 2\}, D_2 = \{1, 2\}, D_3 = \{1, 2\}$$

- Is this feasible?
  - No, only two values for 3 variables (pigeon hole principle)

Why Global Constraints?

- Given
  - a constraint alldifferent(x_1,...,x_3)
  - x_1 in [1..2], ..., x_3 in [1..2]

$$\text{alldifferent}(x_1, x_2, x_3)$$
$$D_1 = \{1, 2\}, D_2 = \{1, 2\}, D_3 = \{1, 2\}$$

- Is this feasible?

Why Global Constraints?

- Given
  - a constraint alldifferent(x_1,...,x_3)
  - x_1 in [1..2], ..., x_3 in [1..2]

$$\text{alldifferent}(x_1, x_2, x_3)$$
$$D_1 = \{1, 2\}, D_2 = \{1, 2\}, D_3 = \{1, 2\}$$

- Is this feasible?
  - No, only two values for 3 variables (pigeon hole principle)

Why Global Constraints?

$$\text{alldifferent}(x_1, x_2, x_3)$$
$$D_1 = \{1, 2\}, D_2 = \{1, 2\}, D_3 = \{1, 2\}$$

- Is this feasible?
  - No, only two values for 3 variables (pigeon hole principle)

Why Global Constraints?

$$\text{alldifferent}(x_1, x_2, x_3)$$
$$D_1 = \{1, 2\}, D_2 = \{1, 2\}, D_3 = \{1, 2\}$$

- Is this feasible?
  - No, only two values for 3 variables (pigeon hole principle)
- However $x_1 \neq x_2, x_2 \neq x_3, x_3 \neq x_1$

Why Global Constraints?

$$\text{alldifferent}(x_1, x_2, x_3)$$
$$D_1 = \{1, 2\}, D_2 = \{1, 2\}, D_3 = \{1, 2\}$$

- Is this feasible?
  - No, only two values for 3 variables (pigeon hole principle)
- However $x_1 \neq x_2, x_2 \neq x_3, x_3 \neq x_1$

$$\exists\, v_1 \in D_1, \ldots, v_n \in D_n :$$
$$\text{c}(x_1 = v_1, \ldots, x_n = v_n) = \text{true}$$

Why Global Constraints?

$$\text{alldifferent}(x_1, x_2, x_3)$$
$$D_1 = \{1, 2\}, D_2 = \{1, 2\}, D_3 = \{1, 2\}$$

- Is this feasible?
  - No, only two values for 3 variables (pigeon hole principle)
- However $x_1 \neq x_2, x_2 \neq x_3, x_3 \neq x_1$

$$\exists\, v_1 \in D_1, \ldots, v_n \in D_n :$$
$$\text{c}(x_1 = v_1, \ldots, x_n = v_n) = \text{true}$$

$$\neq (x_1 = 1, x_2 = 2) = \text{true}$$
$$\neq (x_2 = 1, x_3 = 2) = \text{true}$$
$$\neq (x_3 = 1, x_1 = 2) = \text{true}$$

Why Global Constraints?

$$\text{alldifferent}(x_1, x_2, x_3)$$
$$D_1 = \{1, 2\}, D_2 = \{1, 2\}, D_3 = \{1, 2\}$$

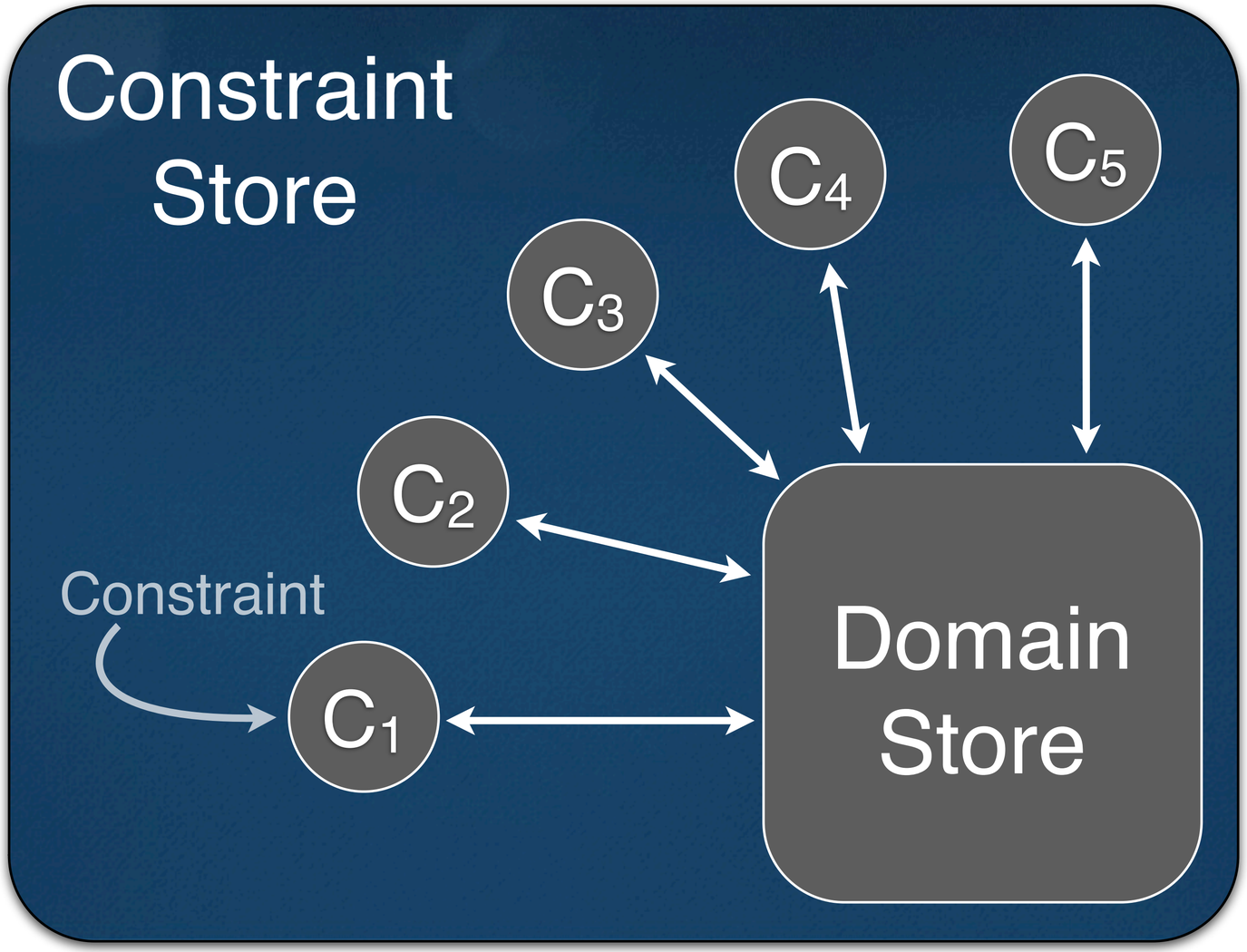
Global constraints make it possible to discover infeasibilities earlier

- However $x_1 \neq x_2, x_2 \neq x_3, x_3 \neq x_1$

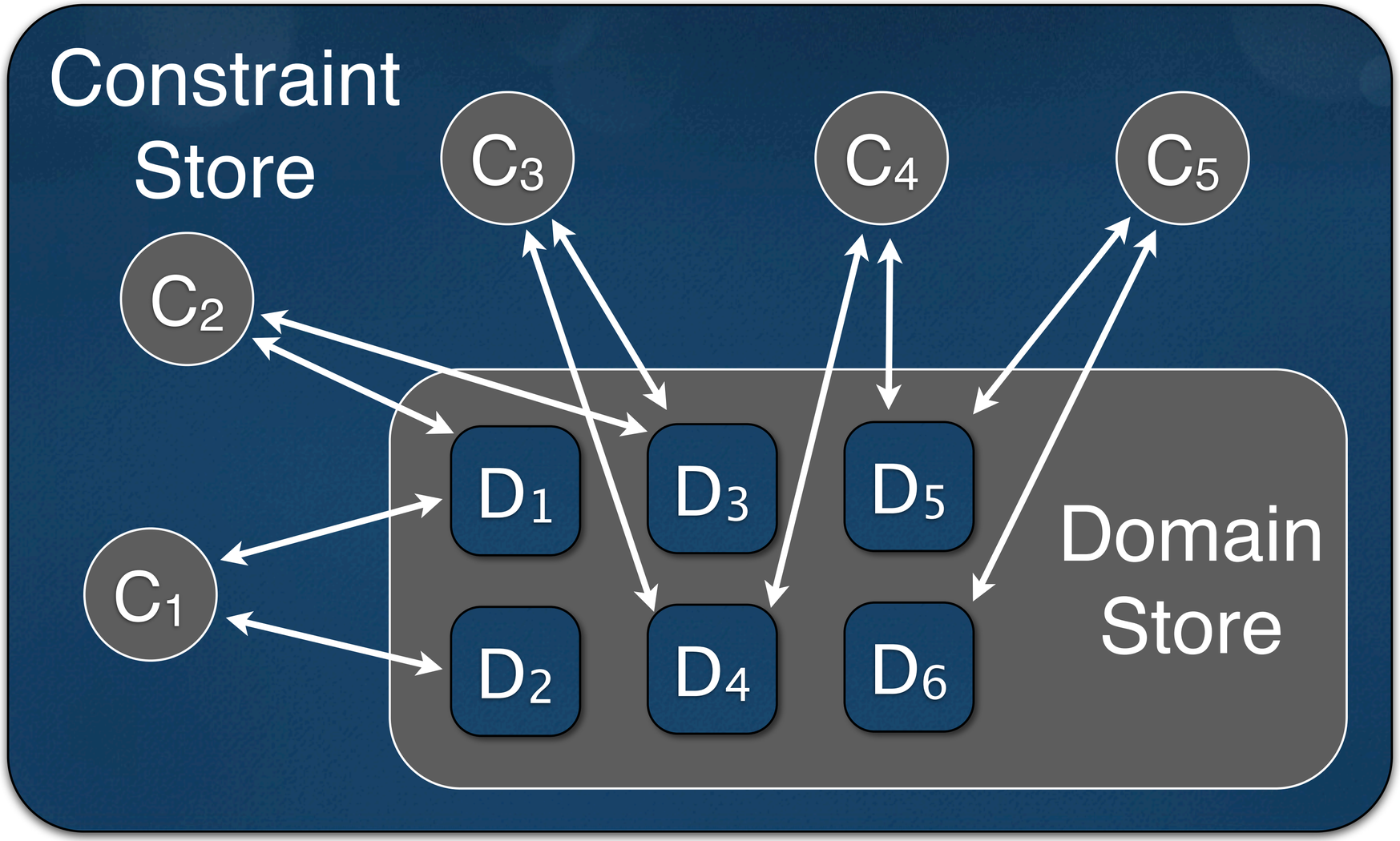
$$\exists\, v_1 \in D_1, \ldots, v_n \in D_n :$$
$$\text{c}(x_1 = v_1, \ldots, x_n = v_n) = \text{true}$$

$$\neq (x_1 = 1, x_2 = 2) = \text{true}$$
$$\neq (x_2 = 1, x_3 = 2) = \text{true}$$
$$\neq (x_3 = 1, x_1 = 2) = \text{true}$$

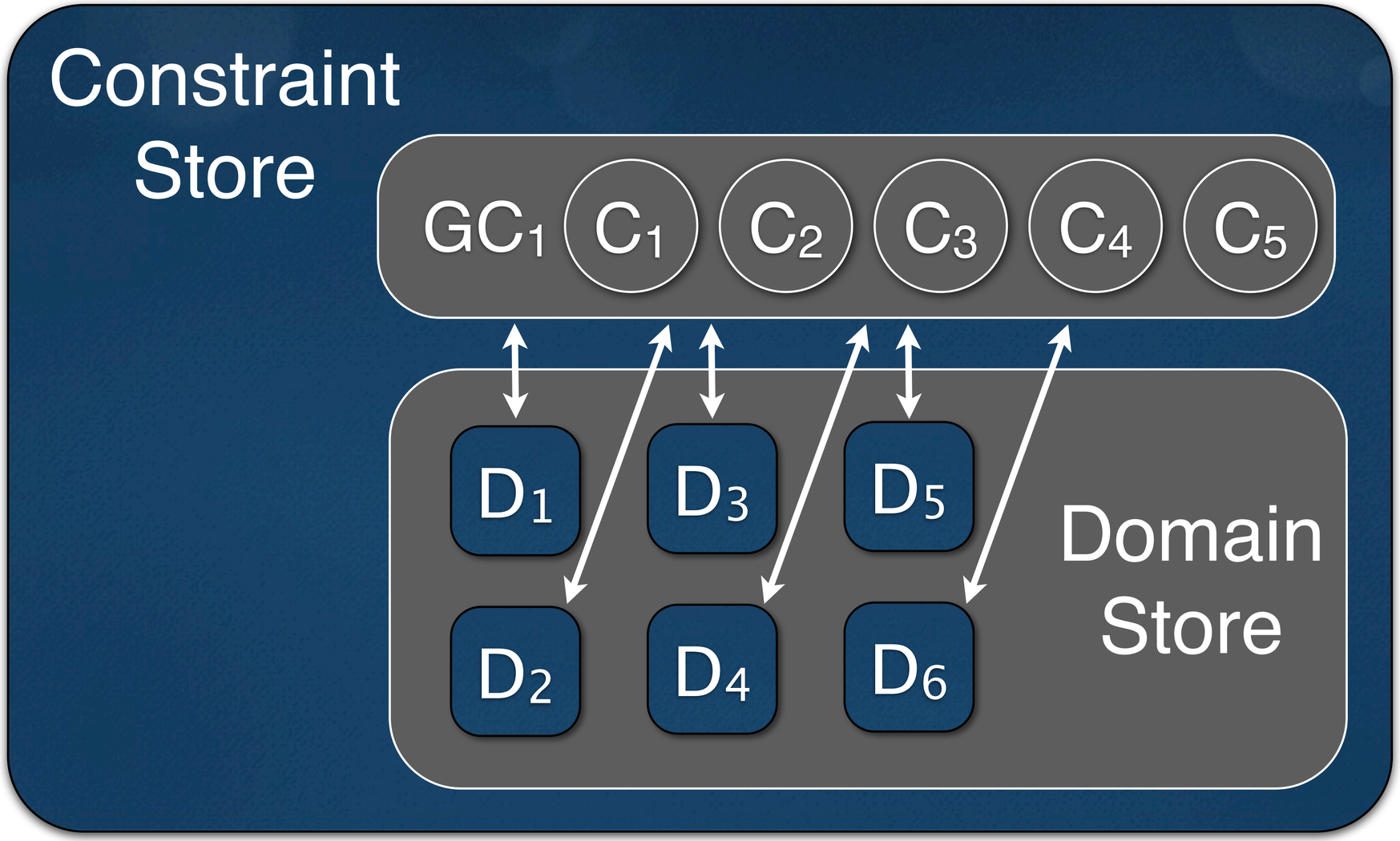
Computational Paradigm



Computational Paradigm



Computational Paradigm



Why Global Constraints?

- Given
  - a constraint $c(x_1,...,x_n)$
  - x_1 in D_1, ..., x_n in D_n

Why Global Constraints?

- Given
  - a constraint $c(x_1,...,x_n)$
  - x_1 in D_1, ..., x_n in D_n
- Pruning

Why Global Constraints?

- Given
  - a constraint $c(x_1,...,x_n)$
  - x_1 in D_1, ..., x_n in D_n
- Pruning
  - Given v_i in D_i, does there exist a solution such that $x_i=v_i$?

Why Global Constraints?

- Given
  - a constraint $c(x_1,...,x_n)$
  - x_1 in D_1, ..., x_n in D_n
- Pruning
  - Given v_i in D_i, does there exist a solution such that $x_i=v_i$?
  - Can I find values in the variable domains such that the constraint holds?

Why Global Constraints?

- Given
  - a constraint $c(x_1,...,x_n)$
  - x_1 in D_1, ..., x_n in D_n
- Pruning
  - Given v_i in D_i, does there exist a solution such that $x_i=v_i$?
  - Can I find values in the variable domains such that the constraint holds?

$$\text{given: } v_i \in D_i,\ x_i = v_i$$

$$\begin{aligned}\exists\, & v_1 \in D_1, \ldots, v_{i-1} \in D_{i-1}, \\ & v_{i+1} \in D_{i+1}, \ldots, v_n \in D_n : \\ & \mathrm{c}(x_1 = v_1, \ldots, x_n = v_n) = \mathrm{true}\end{aligned}$$

Why Global Constraints?

- Given
  - a constraint alldifferent(x_1,...,x_3)
  - x_1 in [1..2], x_2 in [1..2], x_3 in [1..3]

$$\text{alldifferent}(x_1, \ldots, x_n)$$

$$D_1 = \{1, 2\}, D_2 = \{1, 2\}, D_3 = \{1, 2, 3\}$$

Why Global Constraints?

- Given
  - a constraint alldifferent(x_1,...,x_3)
  - x_1 in [1..2], x_2 in [1..2], x_3 in [1..3]

$$\text{alldifferent}(x_1, \ldots, x_n)$$

$$D_1 = \{1, 2\}, D_2 = \{1, 2\}, D_3 = \{1, 2, 3\}$$

- Pruning?

Why Global Constraints?

- Given
  - a constraint alldifferent(x_1,...,x_3)
  - x_1 in [1..2], x_2 in [1..2], x_3 in [1..3]

$$\text{alldifferent}(x_1, \ldots, x_n)$$

$$D_1 = \{1, 2\}, D_2 = \{1, 2\}, D_3 = \{1, 2, 3\}$$

- Pruning?

$$x_3 \neq 1 \quad x_3 \neq 2$$

Why Global Constraints?

- Given
  - a constraint alldifferent(x_1,...,x_3)
  - x_1 in [1..2], x_2 in [1..2], x_3 in [1..3]

$$\text{alldifferent}(x_1, \ldots, x_n)$$

$$D_1 = \{1, 2\}, D_2 = \{1, 2\}, D_3 = \{1, 2, 3\}$$

- Pruning?

$$x_3 \neq 1 \quad x_3 \neq 2$$

Why Global Constraints?

- Given
  - a constraint alldifferent(x_1,...,x_3)
  - x_1 in [1..2], x_2 in [1..2], x_3 in [1..3]

$$\text{alldifferent}(x_1, \ldots, x_n)$$

$$D_1 = \{1, 2\}, D_2 = \{1, 2\}, D_3 = \{1, 2, 3\}$$

- Pruning?

$$x_3 \neq 1 \quad x_3 \neq 2$$

- However $x_1 \neq x_2, x_2 \neq x_3, x_3 \neq x_1$
  - Do not produce any pruning

Why Global Constraints?

Global constraints make it possible to prune the search space more.

$$\text{alldifferent}(x_1, \ldots, x_n)$$

$$D_1 = \{1, 2\}, D_2 = \{1, 2\}, D_3 = \{1, 2, 3\}$$

- Pruning?

  $x_3 \neq 1 \quad x_3 \neq 2$

- However $x_1 \neq x_2, x_2 \neq x_3, x_3 \neq x_1$
  - Do not produce any pruning

Why Global Constraints?

- The million-dollar question
  - Can we detect feasibility and prune global constraints efficiently?

Why Global Constraints?

- The million-dollar question
  - Can we detect feasibility and prune global constraints efficiently?
- It depends on the constraint obviously
  - sometimes we can
  - sometimes we need to relax our standards
    - the pruning may be suboptimal
    - the pruning may take exponential time
    - ....

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

```
range R = 1..9;
var{int} s[R,R] in R;
solve {
 //constraints on fixed positions
 forall(i in R)
   alldifferent(all(j in R) s[i,j]);
 forall(j in R)
   alldifferent(all(i in R) s[i,j]);
 forall(i in 0..2,j in 0..2)
   alldifferent(all(r in i*3+1..i*3+3,
                    c in j*3+1..j*3+3) s[r,c]);
}
```

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				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		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		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
	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		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

Table Constraints

- The simplest global constraint

Table Constraints

- The simplest global constraint

X $\in \{1, 2\}$

Y $\in \{1, 2\}$

Z $\in \{3, 4, 5\}$

Table Constraints

- The simplest global constraint

X $\in \{1, 2\}$

Y $\in \{1, 2\}$

Z $\in \{3, 4, 5\}$

Total Possibilities?

Table Constraints

- The simplest global constraint

X $\in \{1, 2\}$

Y $\in \{1, 2\}$

Z $\in \{3, 4, 5\}$

Total Possibilities?

$$|\{1, 2\}| \times |\{1, 2\}| \times |\{3, 4, 5\}|$$

Table Constraints

- The simplest global constraint

X $\in \{1, 2\}$

Y $\in \{1, 2\}$

Z $\in \{3, 4, 5\}$

Total Possibilities?

$$|\{1, 2\}| \times |\{1, 2\}| \times |\{3, 4, 5\}|$$

$$2 \times 2 \times 3 = 12$$

Table Constraints

- The simplest global constraint

X $\in \{1, 2\}$

Y $\in \{1, 2\}$

Z $\in \{3, 4, 5\}$

Total Possibilities?

$$|\{1, 2\}| \times |\{1, 2\}| \times |\{3, 4, 5\}|$$

$$2 \times 2 \times 3 = 12$$

Table Constraint	X	Y	Z
Combination 1	1	1	5
Combination 2	1	2	4
Combination 3	2	2	3
Combination 4	1	2	3

Table Constraints

- The simplest global constraint

X $\in \{1, 2\}$

Y $\in \{1, 2\}$

Z $\in \{3, 4, 5\}$

$\rightarrow$ $Z \neq 5$

Total Possibilities?

$$|\{1, 2\}| \times |\{1, 2\}| \times |\{3, 4, 5\}|$$

$$2 \times 2 \times 3 = 12$$

Table Constraint	X	Y	Z
Combination 1	1	1	5
Combination 2	1	2	4
Combination 3	2	2	3
Combination 4	1	2	3

Table Constraints

- The simplest global constraint

X $\in \{1, 2\}$

Y $\in \{1, 2\}$

Z $\in \{3, 4, 5\}$

$\rightarrow$ $Z \neq 5$

Total Possibilities?

$$|\{1, 2\}| \times |\{1, 2\}| \times |\{3, 4, 5\}|$$

$$2 \times 2 \times 3 = 12$$

Table Constraint

	X	Y	Z
~~Combination 1~~	~~1~~	~~1~~	~~5~~
Combination 2	1	2	4
Combination 3	2	2	3
Combination 4	1	2	3

Table Constraints

- The simplest global constraint

X $\in \{1, 2\}$

Y $\in \{1, 2\}$

Z $\in \{3, 4, 5\}$

Pruning?

$\rightarrow$ $Z \neq 5$

Total Possibilities?

$$|\{1, 2\}| \times |\{1, 2\}| \times |\{3, 4, 5\}|$$

$$2 \times 2 \times 3 = 12$$

Table Constraint

	X	Y	Z
~~Combination 1~~	~~1~~	~~1~~	~~5~~
Combination 2	1	2	4
Combination 3	2	2	3
Combination 4	1	2	3

Table Constraints

- The simplest global constraint

X $\in \{1, 2\}$

Y $\in \{1, 2\}$

Z $\in \{3, 4, 5\}$

Pruning?

$\rightarrow$ $Z \neq 5$

Total Possibilities?

$$|\{1, 2\}| \times |\{1, 2\}| \times |\{3, 4, 5\}|$$

$$2 \times 2 \times 3 = 12$$

Table Constraint

	X	Y	Z
~~Combination 1~~	~~1~~	~~1~~	~~5~~
Combination 2	1	2	4
Combination 3	2	2	3
Combination 4	1	2	3

Table Constraints

- The simplest global constraint

X $\in \{1, 2\}$

Y $\in \{1, 2\}$

Z $\in \{3, 4, 5\}$

Pruning?

$\rightarrow$ $Z \neq 5$

Total Possibilities?

$$|\{1,2\}| \times |\{1,2\}| \times |\{3,4,5\}|$$

$$2 \times 2 \times 3 = 12$$

Table Constraint

	X	Y	Z
~~Combination 1~~	~~1~~	~~1~~	~~5~~
Combination 2	1	2	4
Combination 3	2	2	3
Combination 4	1	2	3

Table Constraints

- The simplest global constraint

X $\in \{1, 2\}$

Y $\in \{1, 2\}$

Z $\in \{3, 4, 5\}$

Pruning?

$\rightarrow \boxed{Z \neq 5} \rightarrow \boxed{Y \neq 1}$

Total Possibilities?

$$|\{1, 2\}| \times |\{1, 2\}| \times |\{3, 4, 5\}|$$

$$2 \times 2 \times 3 = 12$$

Table Constraint

	X	Y	Z
~~Combination 1~~	~~1~~	~~1~~	~~5~~
Combination 2	1	2	4
Combination 3	2	2	3
Combination 4	1	2	3

Finding Optimal Solutions

```
enum Countries = { Belgium, Denmark, France,
                   Germany, Netherlands, Luxembourg };
var{int} color[Countries] in 1..4;
minimize
  max(c in Countries) color[c]
subject to {
  color[Belgium] ≠ color[France];
  color[Belgium] ≠ color[Germany];
  color[Belgium] ≠ color[Netherlands];
  color[Belgium] ≠ color[Luxembourg];
  color[Denmark] ≠ color[Germany];
  color[France] ≠ color[Germany];
  color[France] ≠ color[Luxembourg];
  color[Germany] ≠ color[Netherlands];
  color[Germany] ≠ color[Luxembourg];
}
```

Optimization in Constraint Programming?

- Focus of constraint programming
  - feasibility

Optimization in Constraint Programming?

- Focus of constraint programming
  - feasibility
- How to optimize?
  - Solve a sequence of satisfaction problems
  - Find a solution
  - Impose a constraint that the next solution must be better

Optimization in Constraint Programming?

- Focus of constraint programming
  - feasibility
- How to optimize?
  - Solve a sequence of satisfaction problems
  - Find a solution
  - Impose a constraint that the next solution must be better
- Guaranteed to find an optimal solution
  - at least theoretically
  - strong when the new constraint reduces the search space
    - scheduling problems are good examples

Discrete Optimization

Constraint Programming: Part V

Goal of the lecture

- Illustrating the modeling techniques in constraint programming
  - symmetry breaking

Symmetries

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

BIBDs

- Balanced Incomplete Block Designs (BIBDs)
  - Input: (v,b,r,k,l)
  - Output: a v by b 0/1 matrix with exactly r ones per row, k ones per column, and a scalar product of value l

- Why BIBDs?
  - example of combinatorial design
  - full of variable symmetries

(3,3,2,2,1)

1	1	0
0	1	1
1	0	1

BIBDs

```
range Rows = 1..v;
range Cols = 1..b;
var{int} m[Rows,Cols] in 0..1;
solve {
   forall(i in Rows)
      sum(y in Cols) m[i,y] = r;
   forall(j in Cols)
      sum(x in Rows) m[x,j] = k;
   forall(i in Rows,j in Rows: j > i)
      sum(x in Cols) (m[i,x] & m[j,x]) = 1;
}
```

BIBDs

(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

BIBDs

(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

BIBDs

- How to break variable symmetries
  - impose an ordering on the variables

- Consider the row symmetries
  - 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

$a \leq b$ $a \geq b$

BIBDs

(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

Lexicographic Ordering

(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

BIBDs

```
range Rows = 1..v;
range Cols = 1..b;
var{int} m[Rows,Cols] in 0..1;
solve {
   forall(i in Rows)
      sum(y in Cols) m[i,y] = r;
   forall(j in Cols)
      sum(x in Rows) m[x,j] = k;
   forall(i in Rows,j in Rows: j > i)
      sum(x in Cols) (m[i,x] & m[j,x]) = 1;
   forall(i in 1..v-1)
      lexleq(all(j in Cols) m[i,j],all(j in Cols) m[i+1,j]);
   forall(j in 1..b-1)
      lexleq(all(i in Rows) m[i,j],all(i in Rows) m[i,j+1]);
}
```

Scene Allocation

- Shooting scenes for a movie
  - an actor plays in some of the scenes
  - at most k scenes can be shot per day
  - each actor is paid by the day
- Objective
  - minimize the total cost
- Symmetries
  - what kind of symmetries do we have here?

Scene Allocation

```
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{int} shoot[Scenes] in Days;

minimize
   sum(a in Actor) sum(d in Days)
      fee[a] * or(s in which[a]) (shoot[s]=d)
subject to
   atmost(all(i in Days) 5,Days,shoot);
```

Scene Allocation

- Value symmetries
  - the days are interchangeable
  - I can swap all the scenes in day 1 and all the scenes in day 2 and I still have a solution
  - nothing can be used to distinguish the days: they are interchangeable
  - if *s* is a solution, then *p(s)* is a solution where the days of *s* have been permuted by permutation *p*
- How can we eliminate these symmetries?
  - Consider the first scene s_1. What are the days that we consider for this scene?

Scene Allocation

- Value symmetries
  - the days are interchangeable
  - if *s* is a solution, then *p(s)* is a solution where the days of *s* have been permuted by permutation *p*
- How can we eliminate these symmetries?
  - consider the first scene s_1. What are the days that we consider for this scene?

Only one day, say day 1

  - cons ch days

All days are interchangeable at this point

Only two days, day 1 and day 2

Scene Allocation

- How can we eliminate these symmetries?
  - consider the first scene s_1. What are the days that we consider for this scene?

    Only one day, say day 1
  - consider the second scene s_2. Which days must be considered for s_2?

    Only two days, day 1 and day 2
  - In general the already used days and one additional new day

existing days

new day

$$D(s_k) = \{1, 2, \ldots, \max(s_1, ..., s_{k-1}) + 1\}$$

Scene Allocation

```
range Scenes = 1..n;
range Days  = 1..m;
range Actor = …;
int fee[Actor] = …;
set{Actor} appears[Scenes] = …;
set{int} which[a in Actor] = setof(i in Scenes) member(a,appears[i]);
var{int} shoot[Scenes] in Days;

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

Citations

66ème Festival de Venise (Mostra)(http://www.flickr.com/photos/nicogenin/3902889281/) by Nicolas Genin (http://www.flickr.com/photos/nicogenin/) CC BY-SA 2.0 (http://creativecommons.org/licenses/by-sa/2.0/deed.en)

Clive Owen, (http://www.flickr.com/photos/oneras/251106343/) by Mario Antonio Pena Zapateira (http://www.flickr.com/photos/oneras/) CC BY-SA 2.0 (http://creativecommons.org/licenses/by-sa/2.0/deed.en)

Julia Roberts 2011 Shankbone (http://www.flickr.com/photos/shankbone/5656729480/) by David Shankbone (http://www.flickr.com/people/shankbone/) CC BY 2.0 (http://creativecommons.org/licenses/by/2.0/deed.en)

Keira Knightley (http://www.flickr.com/photos/tonyshek/7271339306/) by Tony Shek (http://www.flickr.com/photos/tonyshek/) CC BY-SA 2.0 (http://creativecommons.org/licenses/by-sa/2.0/deed.en)

Discrete Optimization

Constraint Programming: Part VI

Goal of the Lecture

- Illustrating modeling techniques in constraint programming
  - redundant constraints

Redundant Constraints

- Motivation
  - **semantically redundant**
    - do not exclude any solution
  - **computationally significant**
    - reduce the search space

Redundant Constraints

- Motivation
  - **semantically redundant**
    - do not exclude any solution
  - **computationally significant**
    - reduce the search space
- How do I find redundant constraints?
  - they express properties of the solutions not captured by the model

Redundant Constraints

- Motivation
  - **semantically redundant**
    - do not exclude any solution
  - **computationally significant**
    - reduce the search space
- How do I find redundant constraints?
  - they express properties of the solutions not captured by the model
- Critical aspect of constraint programming!

Magic Series

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

	0	1	2	3	4
Occurrences	?	?	?	?	?

Magic Series

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

	0	1	2	3	4
Occurrences	?	?	?	?	?

- Can you find a magic series?

Magic Series

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

	0	1	2	3	4
Occurrences	2	1	2	0	0

- Can you find a magic series?

Magic Series

```
int n = 5;
range D = 0..n-1;
var{int} series[D] in D;
solve {
   forall(k in D)
     series[k] = sum(i in D) (series[i]=k);
}
```

Magic Series

```
int n = 5;
range D = 0..n-1;
var{int} series[D] in D;
solve {
   forall(k in D)
     series[k] = sum(i in D) (series[i]=k);
}
```

- Redundant constraints
  - can you find a property of the solution?

Magic Series

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

- The decision variables denote a number of occurrences
- The number of occurrences is bounded

Magic Series

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

	0	1	2	3	4
Occurrences	?	?	?	?	17

- The decision variables denote a number of occurrences
- The number of occurrences is bounded

Magic Series

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

	0	1	2	3	4
Occurrences	?	?	?	?	17

- The decision variables denote a number of occurrences

Magic Series

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

	0	1	2	3	4
Occurrences	?	?	?	?	17

- The decision variables denote a number of occurrences
- The number of occurrences is bounded

Magic Series

```
int n = 5;
range D = 0..n-1;
var{int} series[D] in D;
solve {
   forall(k in D)
     series[k] = sum(i in D) (series[i]=k);
   sum(i in D) series[i] = n;
}
```

Magic Series

```
int n = 5;
range D = 0..n-1;
var{int} series[D] in D;
solve {
   forall(k in D)
     series[k] = sum(i in D) (series[i]=k);
   sum(i in D) series[i] = n;
}
```

- Redundant constraints
  - The decision variables denote a number of occurrences
  - The number of occurrences is bounded

Magic Series

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

	0	1	2	3	4
Occurrences	?	?	?	?	?

Magic Series

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

- What does “series[2] = 3” mean?

	0	1	2	3	4
Occurrences	?	?	?	?	?

Magic Series

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

- What does “series[2] = 3” mean?
  - that there are three “2” in the array “series”

	0	1	2	3	4
Occurrences	?	?	?	?	?

Magic Series

- What does “series[2] = 3” mean?
  - that there are three “2” in the array “series”

	0	1	2	3	4
Occurrences	?	?	?	?	?

Magic Series

- What does “series[2] = 3” mean?
  - that there are three “2” in the array “series”

- sum(i in D) series[i] =
  sum(i in D) i * series[i]

	0	1	2	3	4
Occurrences	?	?	?	?	?

Magic Series

- What does “series[2] = 3” mean?
  - that there are three “2” in the array “series”

- sum(i in D) series[i] =
  sum(i in D) i * series[i]

- Which constraint is stronger?

	0	1	2	3	4
Occurrences	?	?	?	?	?

Magic Series

- What does “series[2] = 3” mean?
  - that there are three “2” in the array “series”

- sum(i in D) series[i] =
  sum(i in D) i * series[i]

- Which constraint is stronger?
  - sum(i in D) series[i] = n

	0	1	2	3	4
Occurrences	?	?	?	?	?

Magic Series

- What does “series[2] = 3” mean?
  - that there are three “2” in the array “series”

- sum(i in D) series[i] =
  sum(i in D) i * series[i]

- Which constraint is stronger?
  - sum(i in D) series[i] = n
  - sum(i in D) i * series[i] = n

	0	1	2	3	4
Occurrences	?	?	?	?	?

Magic Series

```
int n = 5;
range D = 0..n-1;
var{int} series[D] in D;
solve {
   forall(k in D)
     series[k] = sum(i in D) (series[i]=k);
   sum(i in D) series[i] = n;
   sum(i in D) i * series[i] = n;
}
```

Magic Series

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

- The redundant constraint implies
  - series[4] < 2
  - series[3] < 2
  - series[2] < 3
  - series[1] < 6

Magic Series

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

- The redundant constraint implies
  - series[4] < 2
  - series[3] < 2
  - series[2] < 3
  - series[1] < 6

Magic Series

- Assume that series[0] = 2

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

- It follows that series[2] > 0
  - series[1] + 3 series[3] + 4 series[4] <= 3

Magic Series

- Assume that series[0] = 2

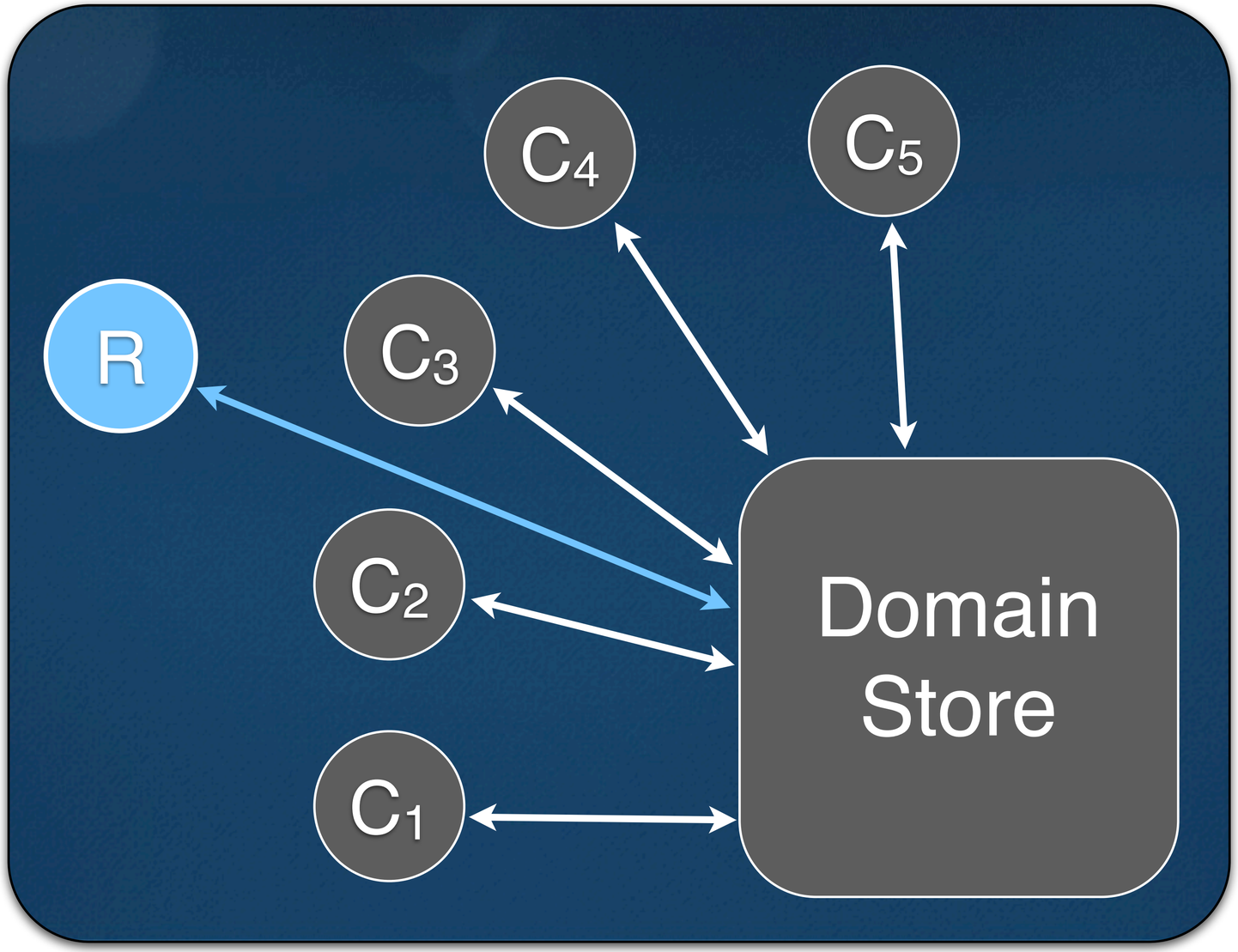
```
2          =    (series[1]=0)+(series[2]=0)+(series[3]=0)+ 1;
series[1]  =    (series[1]=1)+(series[2]=1)+(series[3]=1);
series[2]  = 1 +(series[1]=2)+(series[2]=2);
series[3]  =    (series[1]=3);
0          =    (series[1]=4);
series[1] + 2 series[2] + 3 series[3] = 5
```

- It follows that series[2] > 0
  - series[1] + 3 series[3] + 4 series[4] <= 3
  - series[4] <= 0
  - series[3] <= 1

Redundant Constraints

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

Redundant Constraint



Constraint Store

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 Problems

```
range C = ...;
range V = ...;
int w[C,V] = ...;
int rhs[C];
var{int} x[V] in 0..1;
solve {
   forall(c in C)
      sum(v in V) w[c,v] x[v] = rhs[c];
}
```

Market Split Problems

```
range C = ...;
range V = ...;
int w[C,V] = ...;
int rhs[C];
var{int} x[V] in 0..1;
solve {
   forall(c in C)
      sum(v in V) w[c,v] x[v] = rhs[c];
}
```

- Observe that
  - the constraints only communicate through the domains

Market Split Problems

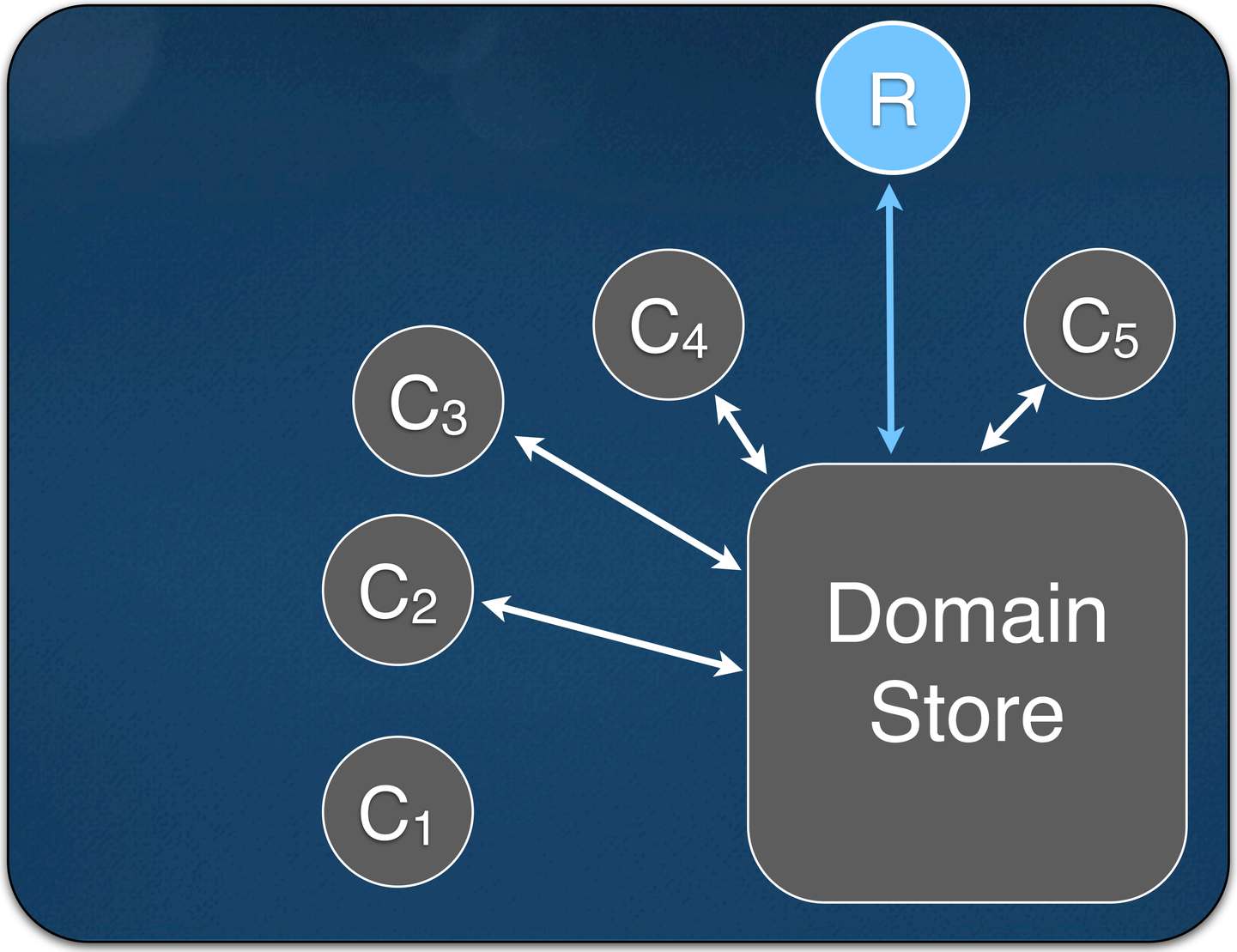
```
range C = ...;
range V = ...;
int w[C,V] = ...;
int rhs[C];
var{int} x[V] in 0..1;
solve {
   forall(c in C)
      sum(v in V) w[c,v] x[v] = rhs[c];
   sum(v in V) (sum(c in C) alpha^c * w[c,v]) * x[v] = sum(c in C) alpha^c * rhs[c];
}
```

Market Split Problems

```
range C = ...;
range V = ...;
int w[C,V] = ...;
int rhs[C];
var{int} x[V] in 0..1;
solve {
   forall(c in C)
      sum(v in V) w[c,v] x[v] = rhs[c];
   sum(v in V) (sum(c in C) alpha^c * w[c,v]) * x[v] = sum(c in C) alpha^c * rhs[c];
}
```

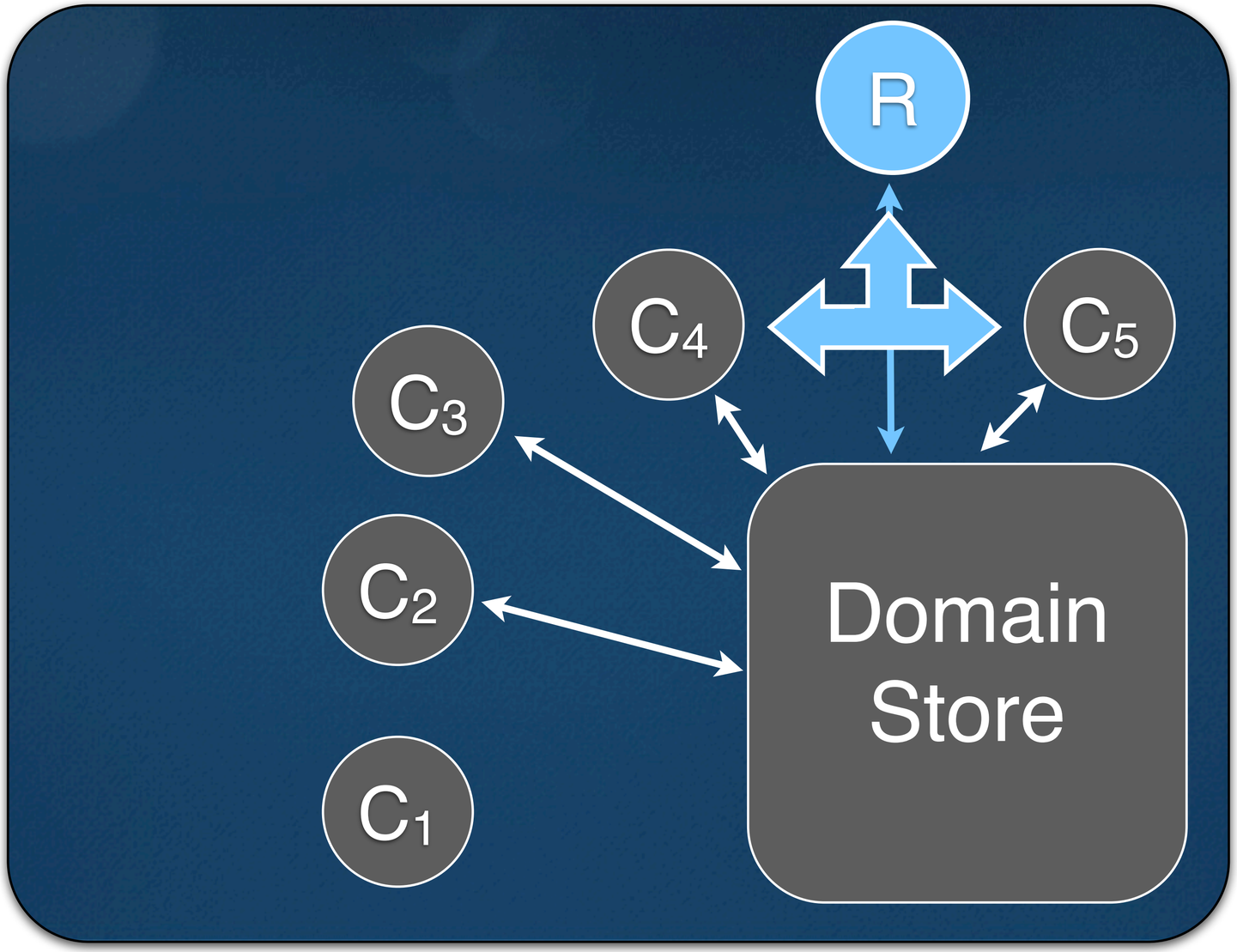
- Surrogate constraints
  - combination of existing constraints

Surrogate Constraint



Constraint Store

Surrogate Constraint



Constraint Store

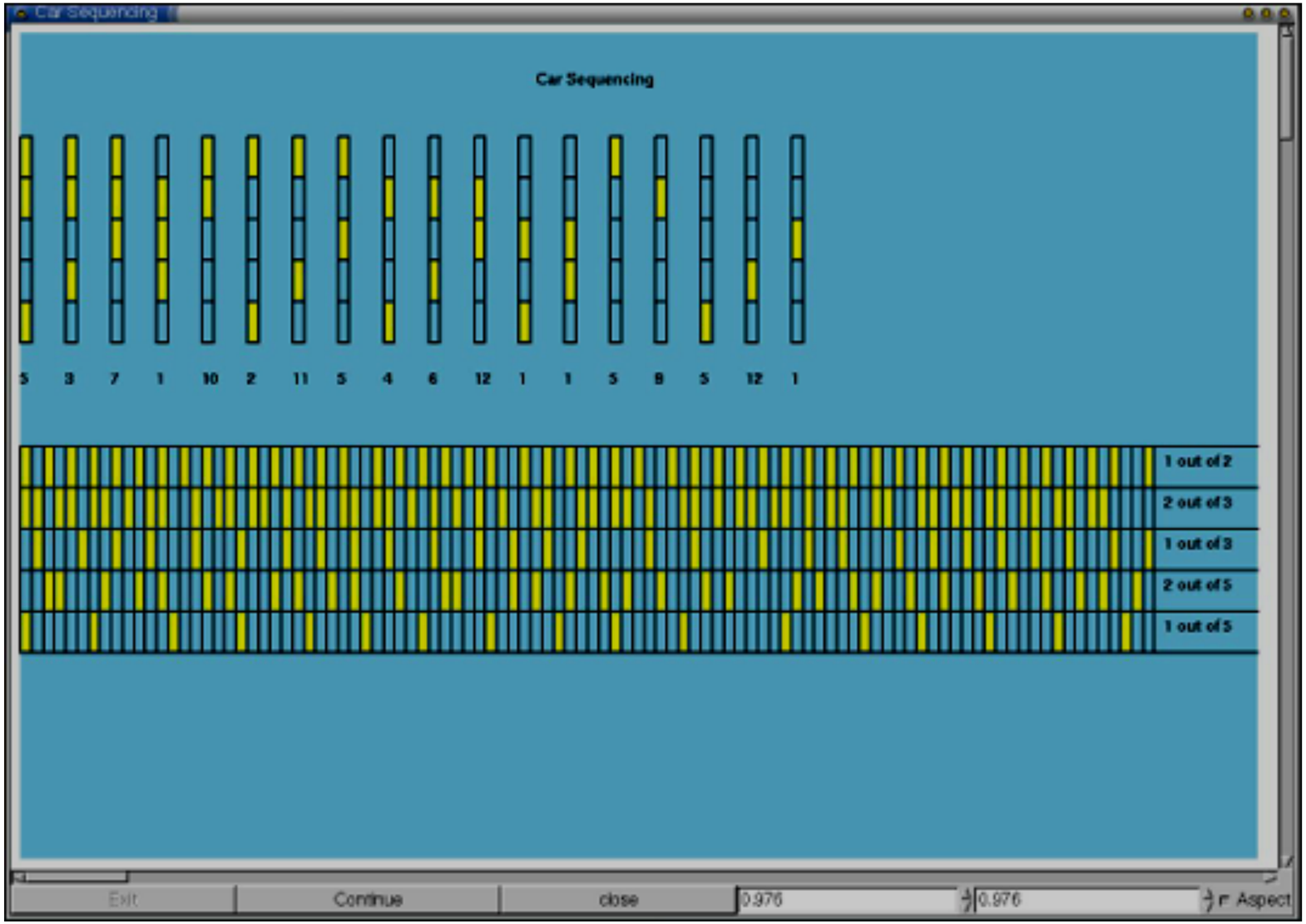
Discrete Optimization

Constraint Programming: Part VII

Car Sequencing

- Cars on an assembly line
- Cars require specific options
  - leather seats, moonroof
- Capacity constraints on the production units
  - at most 2 out of 5 successive cars can require a moonroof
- Sequence all the cars such that the capacity constraints are satisfied

Car Sequencing



Car Sequencing

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 Slots = ...;
range Configs = ...;
range Options = ...;
int demand[Configs] = ...;
int nbCars = sum(c in Configs) demand[c];
int lb[Options] = ...;
int ub[Options] = ...;
int requires[Options,Config] = ...;
var{int} line[Slots] in Configs;
var{int} setup[Options,Slots] in 0..1;

solve {
   forall(c in Configs)
      sum(s in Slots) (line[s] = c) = demand[c];

   forall(s in Slots,o in Options)
      setup[o,s] = requires[o,line[s]];

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

line[s] denotes the type of car sequenced on slot s

setup[o,s]=1 if slot[s] has a car requiring option o

demand constraints

defines the setup variables

capacity constraints

Car Sequencing

Slots	1	2	3	4	5	6	7	8	9	10	Demand
Class 1											1
Class 2											1
Class 3											2
Class 4											2
Class 5											2
Class 6											2

Car Sequencing

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	

Setup	1	2	3	4	5	6	7	8	9	10	Capacity
Option 1											1/2
Option 2											2/3
Option 3											1/3
Option 4											2/5
Option 5											1/5

capacity constraint

element constraint

Car Sequencing

```
range Slots = ...;
range Configs = ...;
range Options = ...;
int demand[Configs] = ...;
int lb[Options] = ...;
int ub[Options] = ...;
int requires[Options,Config] = ...;
var{int} line[Slots] in Configs;
var{int} setup[Options,Slots] in 0..1;

solve {
   forall(c in Configs)
      sum(s in Slots) (line[s] = c) = demand[c];

   forall(s in Slots,o in Options)
      setup[o,s] = requires[o,line[s]];

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

Car Sequencing

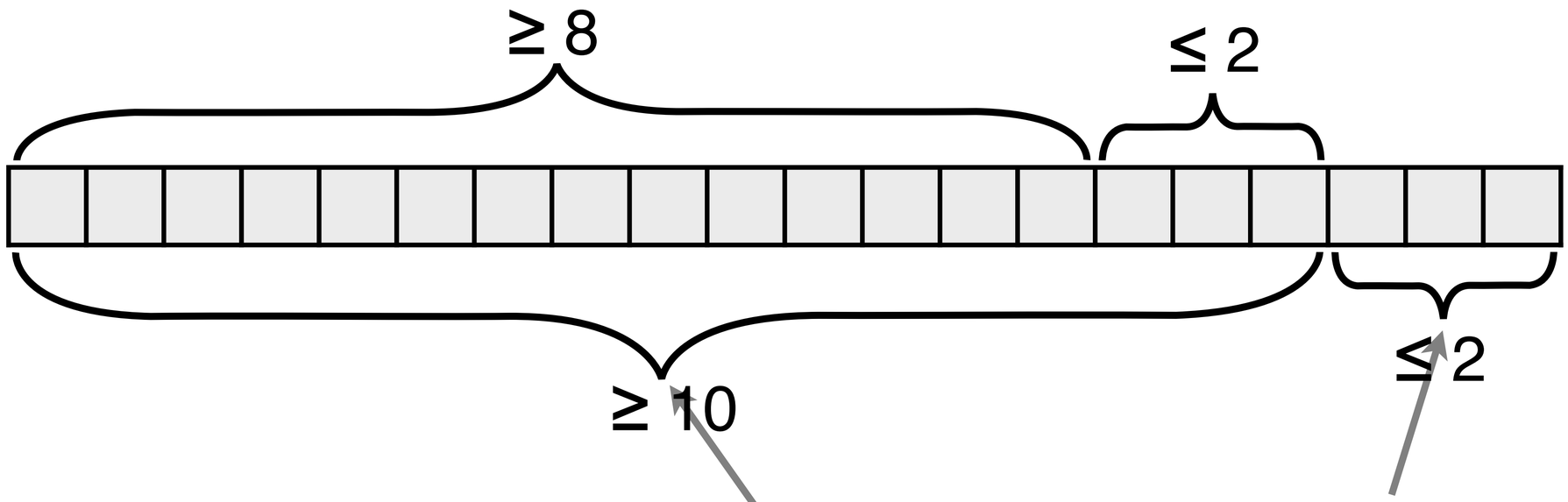
Setup	1	2	3	4	5	6	7	8	9	10	Capacity
Option 1											1/2
Option 2											2/3
Option 3											1/3
Option 4											2/5
Option 5											1/5

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	

Slots	1	2	3	4	5	6	7	8	9	10	Demand
Class 1											1
Class 2											1
Class 3											2
Class 4											2
Class 5											2
Class 6											2

Car Sequencing: Redundant Constraints

- Consider an option *o* with
  - Capacity: 2 out of 3
  - Demand: 12 cars



- How many cars with option *o* can be in the last 3 slots?
- How many cars with option *o* must be in these slots?

Car Sequencing: Redundant Constraints

```
range Slots = ...;
range Configs = ...;
range Options = ...;
int demand[Configs] = ...;
int lb[Options] = ...;
int ub[Options] = ...;
int requires[Options,Config] = ...;
var{int} line[Cars] in Configs;
var{int} setup[Options,Slots] in 0..1;

solve {
   forall(c in Configs)
      sum(s in Slots) (line[s] = c) = demand[c];
   forall(s in Slots,o in Options)
      setup[o,s] = requires[o,line[s]];
   forall(o in Options, s in 1..nbCars-ub[o]+1)
      sum(j in s..s+ub[o]-1) setup[o,s] <= lb[o];

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

}
```

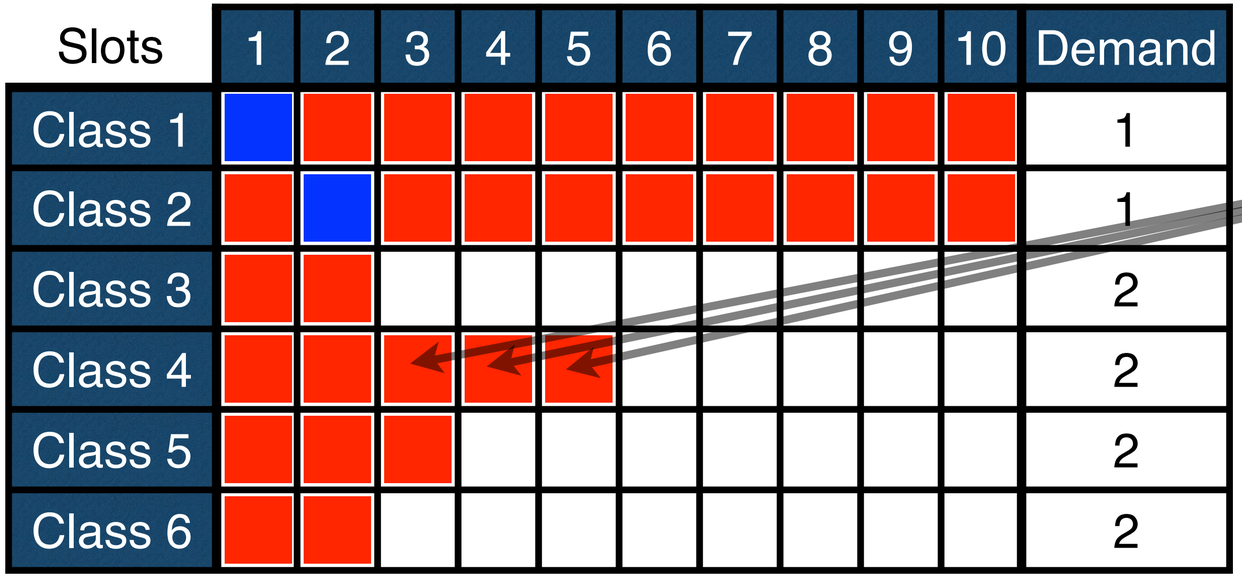
Car Sequencing

Setup	1	2	3	4	5	6	7	8	9	10	Capacity
Option 1											1/2
Option 2											2/3
Option 3											1/3
Option 4											2/5
Option 5											1/5

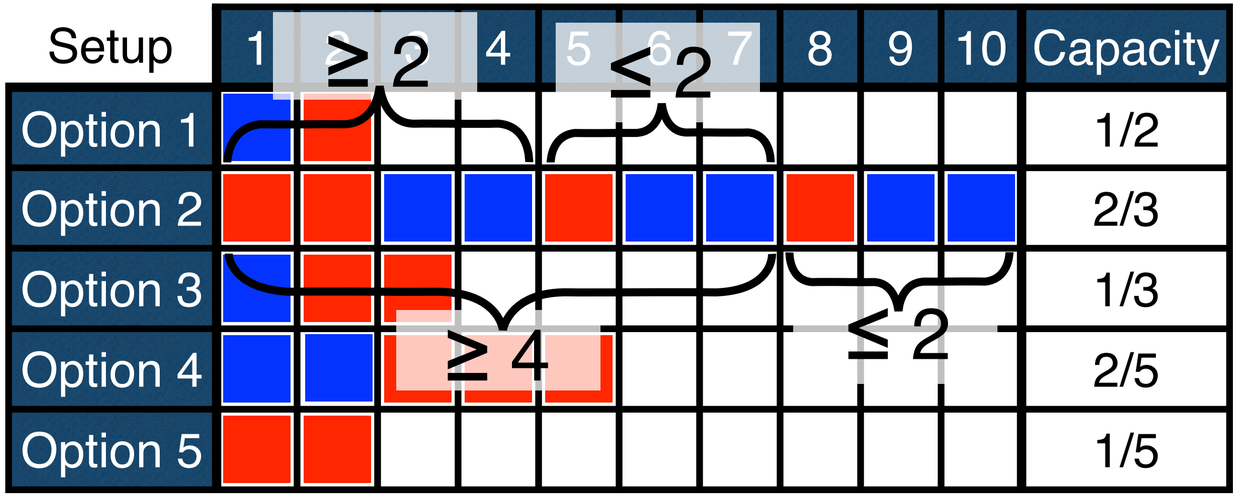
Options	1	2	3	4	5	Demand
Class 1	yes		yes	yes		1
Class 2				yes		1
Class 3		yes			yes	2
Class 4		yes		yes		2
Class 5	yes		yes			2
Class 6	yes	yes				2
Capacity	1/2	2/3	1/3	2/5	1/5	

Slots	1	2	3	4	5	6	7	8	9	10	Demand
Class 1											1
Class 2											1
Class 3											2
Class 4											2
Class 5											2
Class 6											2

Car Sequencing



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

Slots	1	2	3	4	5	6	7	8	9	10	Demand
Class 1											1
Class 2											1
Class 3											2
Class 4											2
Class 5											2
Class 6											2

Setup	1	2	3	4	5	6	7	8	9	10	Capacity
Option 1											1/2
Option 2											2/3
Option 3											1/3
Option 4											2/5
Option 5											1/5

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

Slots	1	2	3	4	5	6	7	8	9	10	Demand
Class 1											1
Class 2											1
Class 3											2
Class 4											2
Class 5											2
Class 6											2

Setup	1	2	3	4	5	6	7	8	9	10	Capacity
Option 1											1/2
Option 2											2/3
Option 3											1/3
Option 4											2/5
Option 5											1/5

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	

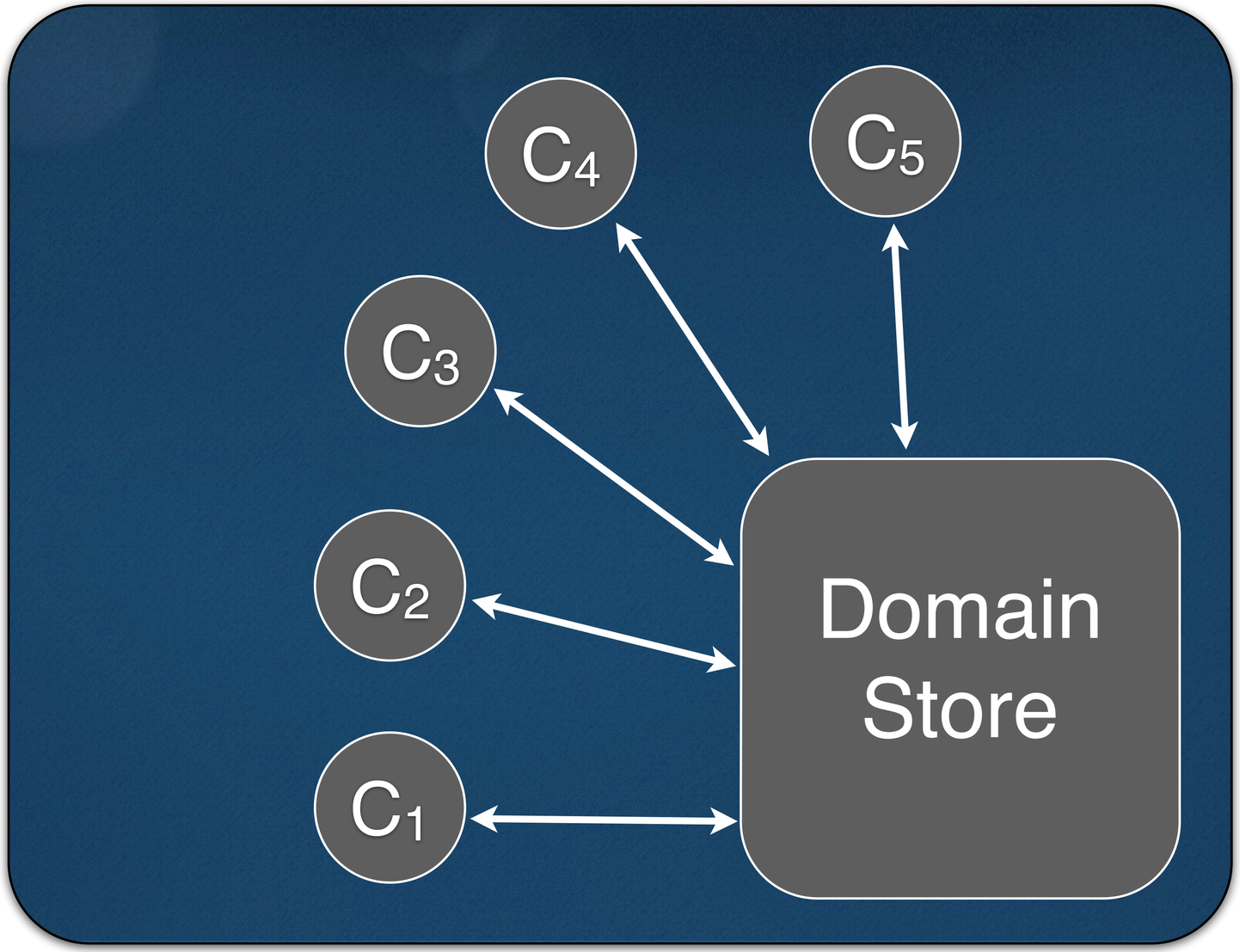
Car Sequencing

Slots	1	2	3	4	5	6	7	8	9	10	Demand
Class 1											1
Class 2											1
Class 3											2
Class 4											2
Class 5											2
Class 6											2

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	

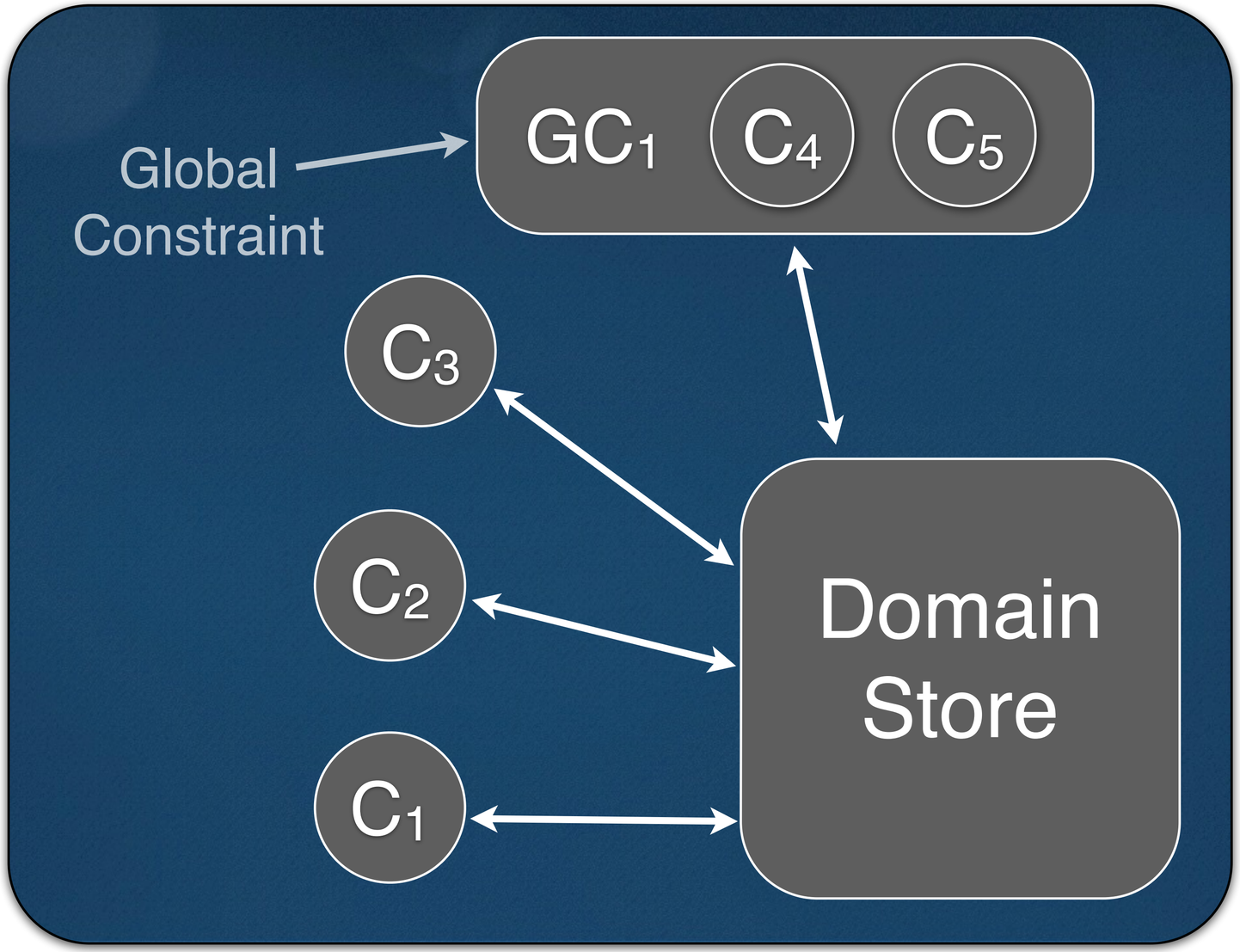
Setup	1	2	3	4	5	6	7	8	9	10	Capacity
Option 1											1/2
Option 2											2/3
Option 3											1/3
Option 4											2/5
Option 5											1/5

Improving Communication



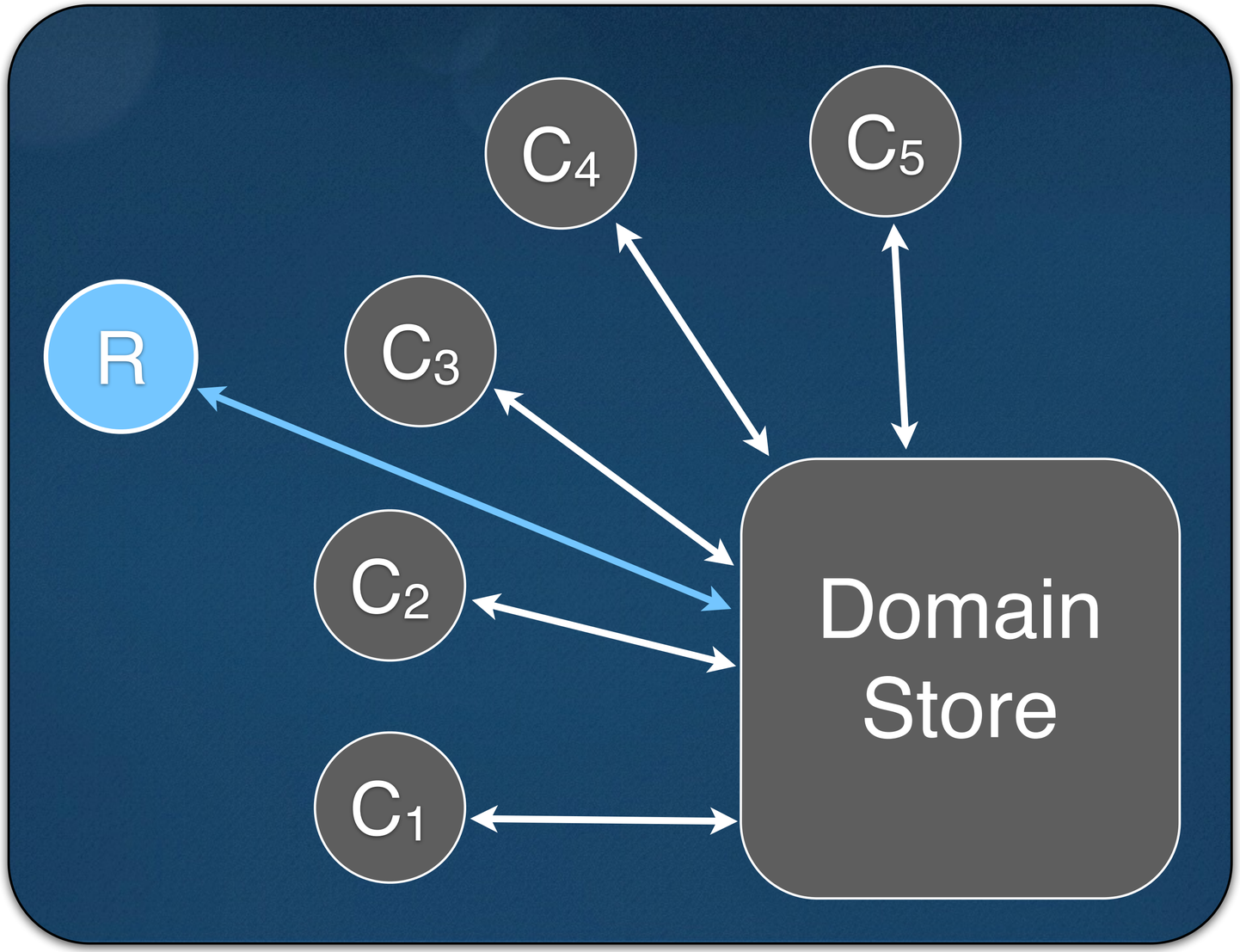
Constraint Store

Improving Communication



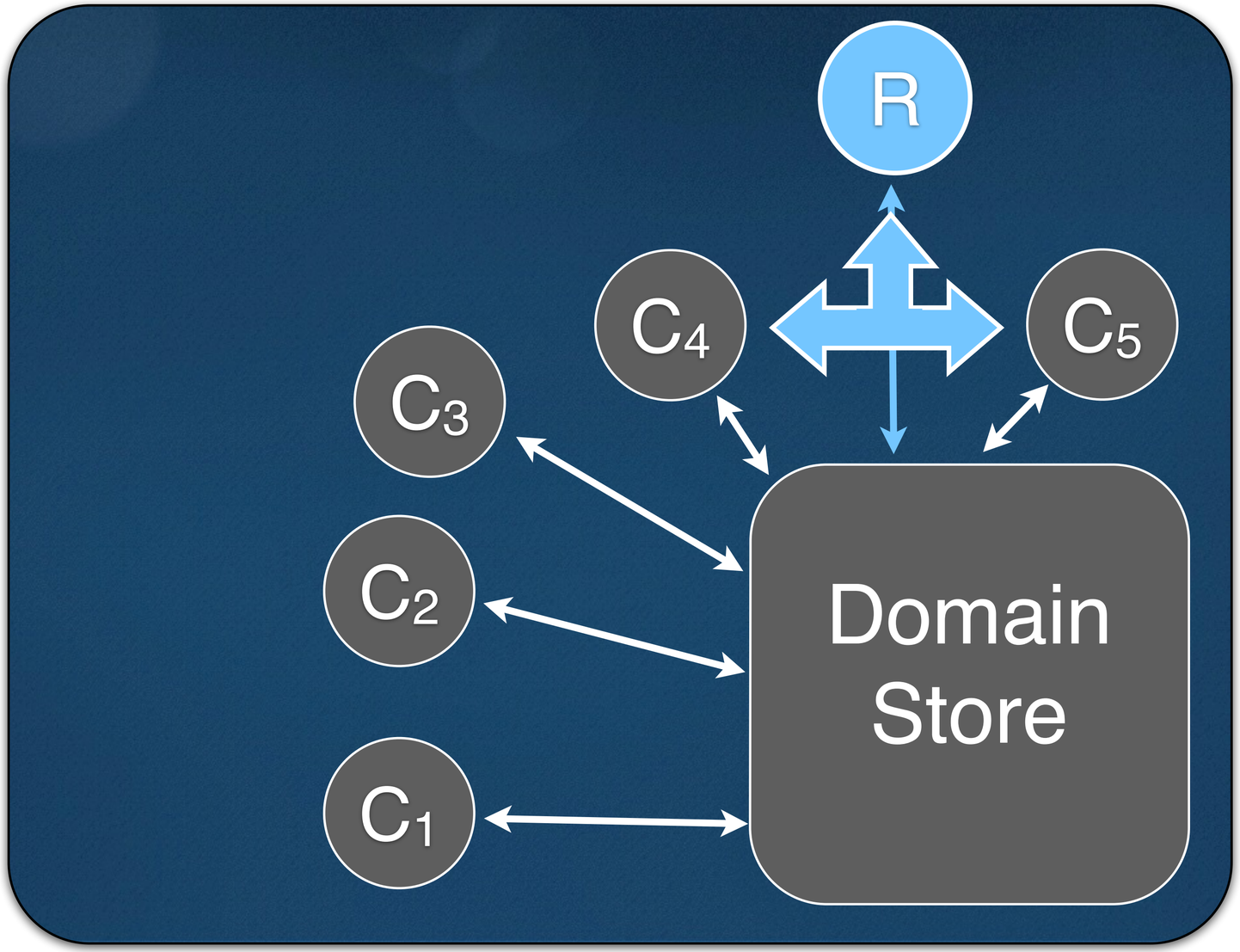
Constraint Store

Improving Communication



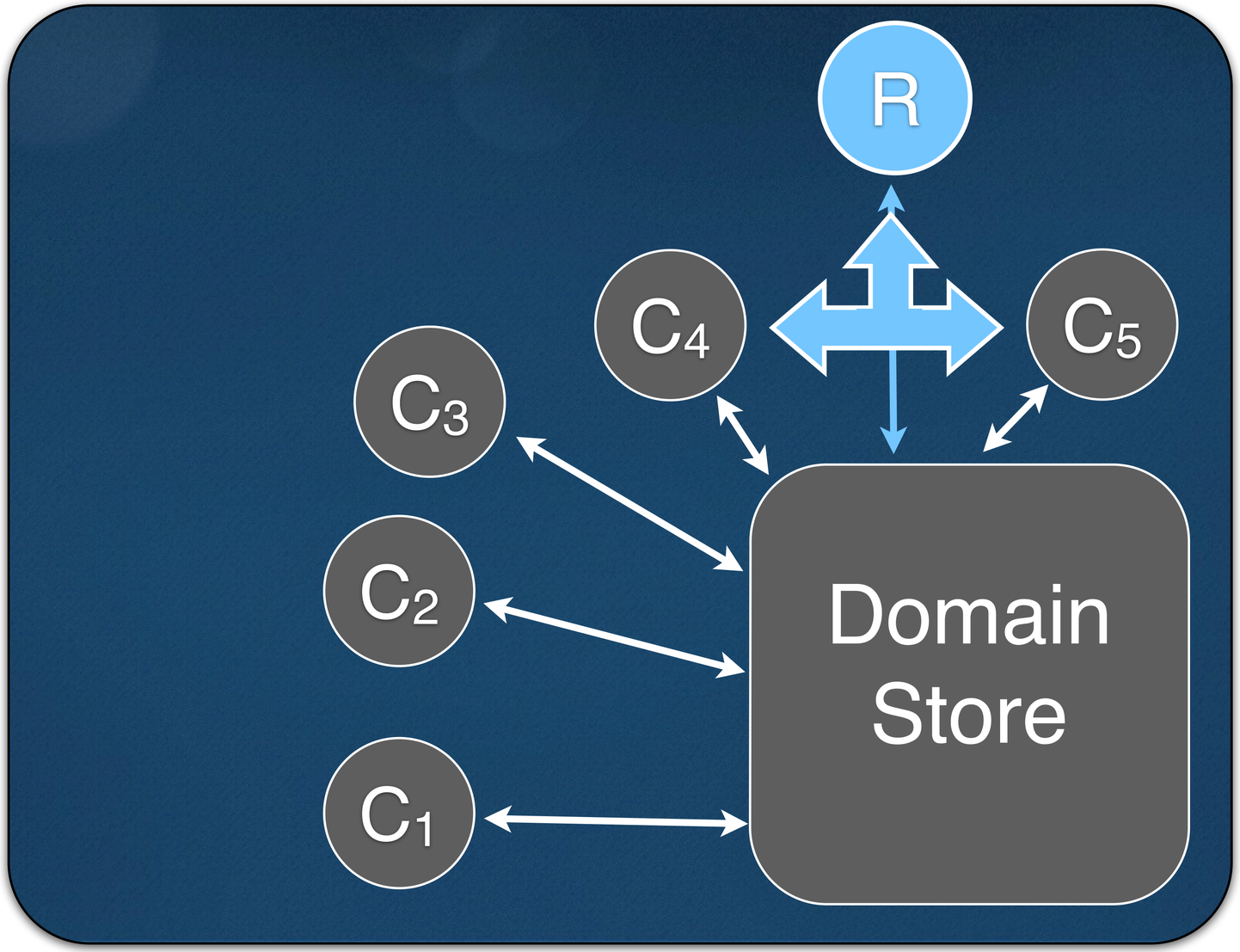
Constraint Store

Improving Communication: Surrogate Constraint



Constraint Store

Improving Communication: Implied Constraint



Constraint Store

Dual Modeling

- Sometimes there are multiple ways of modeling a problem
  - not the same decision variables
- The two models may have complementary strengths
  - hard to choose between them
  - some constraints are easier to express in one model and others in the other one
- Dual modeling
  - the idea of stating multiple models of a problem and linking them with constraints

Back to the 8-Queens Problem

- What are the decision variables?
  - many possible modelings
    - this is what makes optimization problems interesting :-)
- Here is one modeling
  - associate a decision variable with each column
  - the variable denotes the row of the queens placed in this column
- What are the constraints?
  - the queens cannot be placed on the same row
  - the queens cannot be placed on the same upward diagonal
  - the queens cannot be placed on the same downward diagonal

Back to the 8-Queens Problem

- What are the decision variables?
  - many possible modelings
    - this is what makes optimization problems interesting :-)
- Here is another modeling
  - associate a decision variable with each row
  - the variable denotes the column of the queens placed in this row
- What are the constraints?
  - the queens cannot be placed on the same row
  - the queens cannot be placed on the same upward diagonal
  - the queens cannot be placed on the same downward diagonal

The 8-Queens Problem with Dual Modeling

```
range R = 1..8;
range C = 1..8;
var{int} row[C] in R;
var{int} col[R] in C
solve {
   forall(i in R,j in R: i < j) {
      row[i] ≠  row[j];
      row[i] ≠  row[j] + (j - i);
      row[i] ≠  row[j] - (j - i);
   }
   forall(i in C,j in C: i < j) {
      col[i] ≠  col[j];
      col[i] ≠  col[j] + (j - i);
      col[i] ≠  col[j] - (j - i);
   }
   forall(r in R,c in C)
      (row[c] = r) <=> (col[r] = c);
}
```

Citations

Wolfsburg - Volkswagen Assembly Line (http://www.flickr.com/photos/24736216@N07/2994043188/) by Roger Wollstadt (http://www.flickr.com/photos/24736216@N07/) CC BY-SA 2.0 (http://creativecommons.org/licenses/by-sa/2.0/deed.en)

Geely assembly line in Beilun, Ningbo (http://commons.wikimedia.org/wiki/File:Geely_assembly_line_in_Beilun,_Ningbo.JPG) by Siyuwj (Own work) [CC-BY-SA-3.0 (http://creativecommons.org/licenses/by-sa/3.0)], via Wikimedia Commons

Discrete Optimization

Constraint Programming: Part VIII

Goals of the Lecture

- Illustrate how to implement global constraints
  - knapsack
  - alldifferent
- Significant area of research
  - over 100 global constraints proposed so far

The Gold Standard for Pruning

- After pruning
  - if value *v* is in the domain of variable *x*, then there exists a solution to the constraint with value *v* assigned to variable *x*
- Many historical names
  - arc consistency, domain consistency
- Optimal pruning
  - cannot prune more if only domains are considered
- Complexity
  - may not be enforced in polynomial time

Binary Knapsack

- The constraint

$$l \leq \sum_{k \in R} w_k x_k \leq u$$
$$x_k \in \{0, 1\}$$

- Example

$$10 \leq 2x_1 + 3x_2 + 4x_3 + 5x_4 \leq 12$$

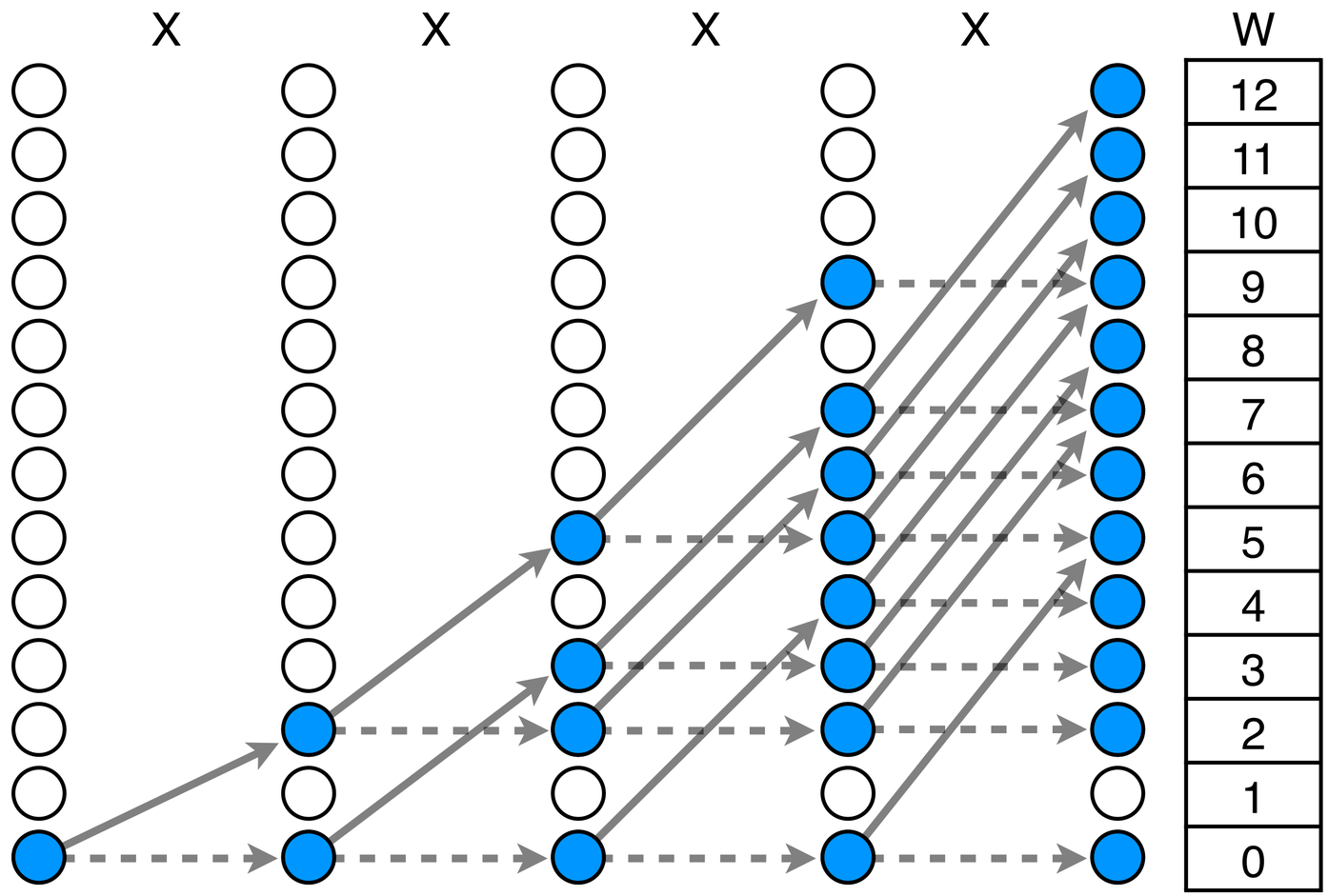
  - feasibility
    - can we find a solution to the constraint?
  - pruning
    - can we eliminate values from the domains?

Binary Knapsack

- Feasibility
  - use dynamic programming for feasibility
  - pseudo-polynomial
- Pruning
  - exploit the dynamic programming table
  - forward phase
    - keep dependency links
  - backward phase
    - update dependency links to keep only feasible values
  - combine feasibility with pruning

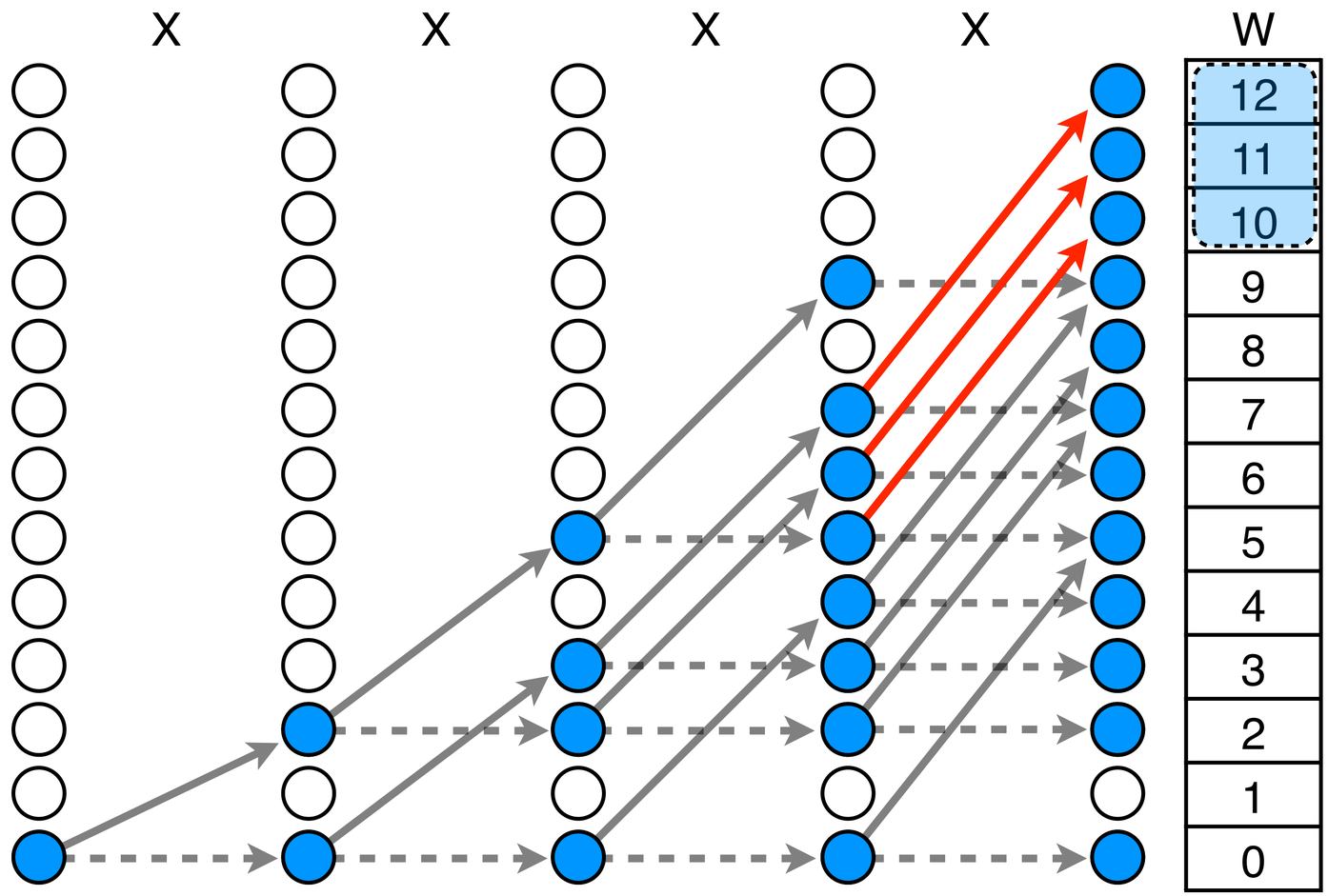
Forward Phase

$$10 \leq 2x_1 + 3x_2 + 4x_3 + 5x_4 \leq 12$$



Backward Phase

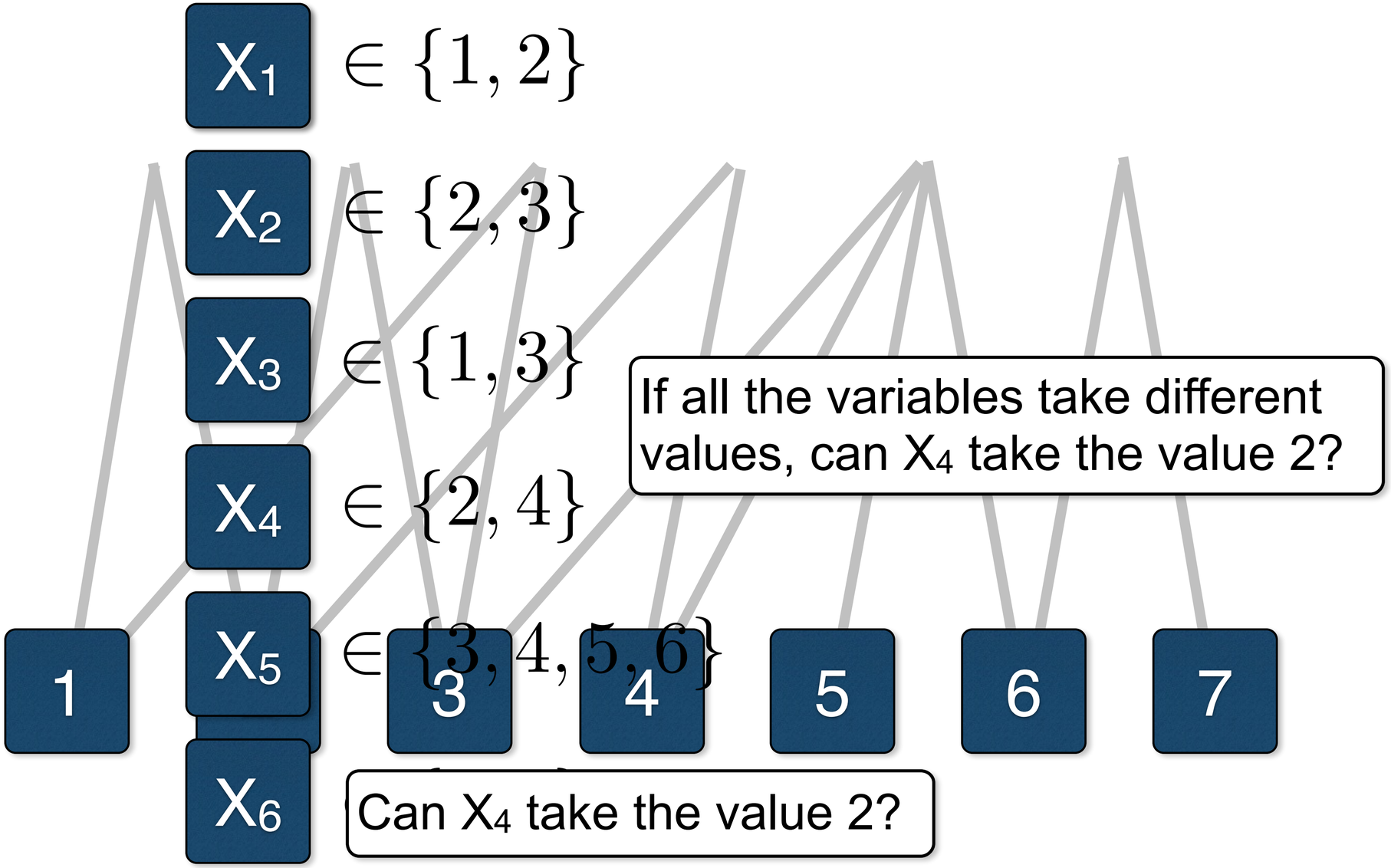
$$10 \leq 2x_1 + 3x_2 + 4x_3 + 5x_4 \leq 12 \quad \boxed{x_4 = 1}$$



Alldifferent Constraint

- The constraint
  - alldifferent(x_1,...,x_n)
- Feasibility
  - can we find values in the domains of the variables so that each two variables are assigned a different value?
- Pruning
  - are there values in the domain of a variable that the variable cannot take, i.e., if the variable takes that value, then there is no solution.

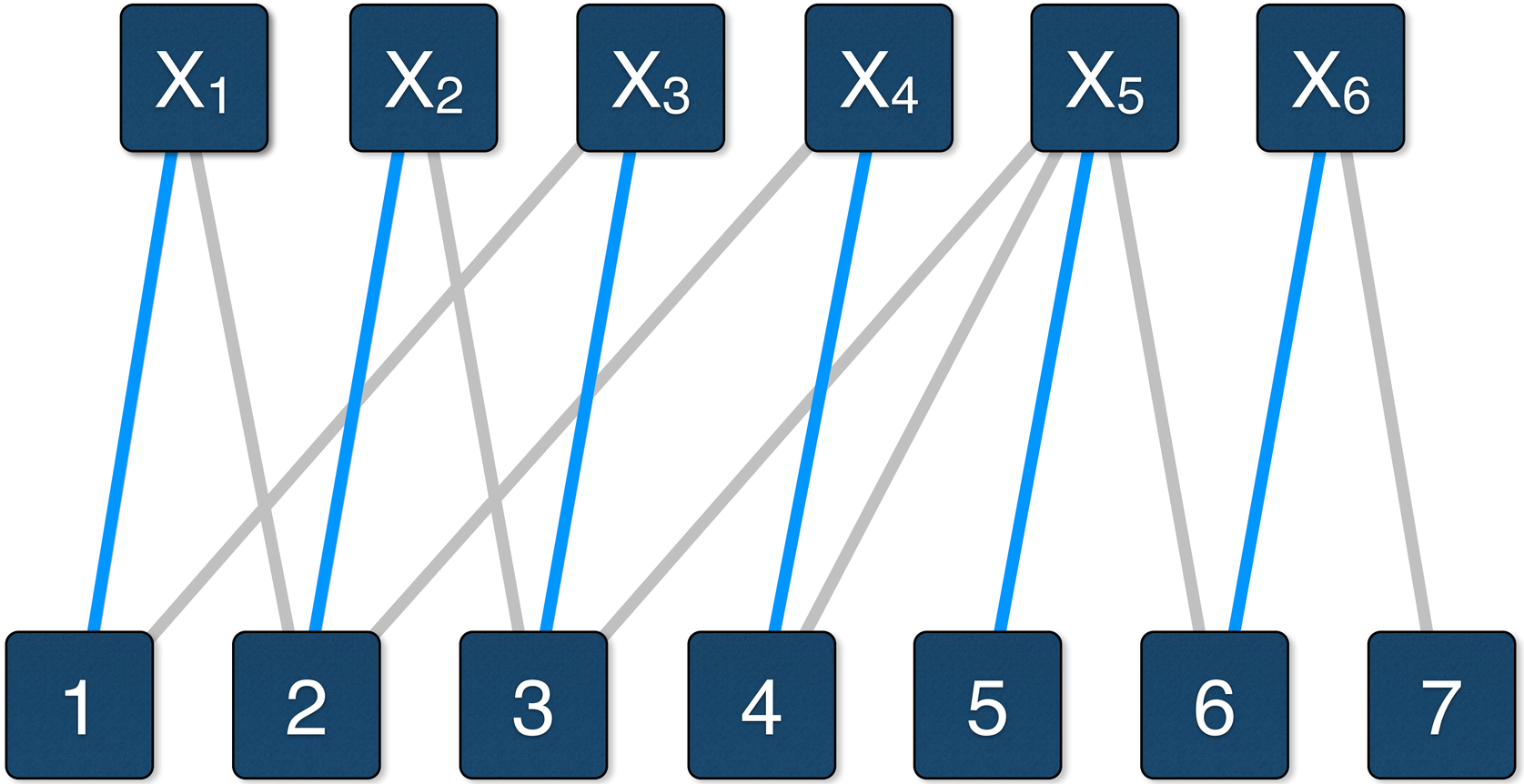
Alldifferent Representation



If all the variables take different values, can X_4 take the value 2?

Can X_4 take the value 2?

Alldifferent Feasibility

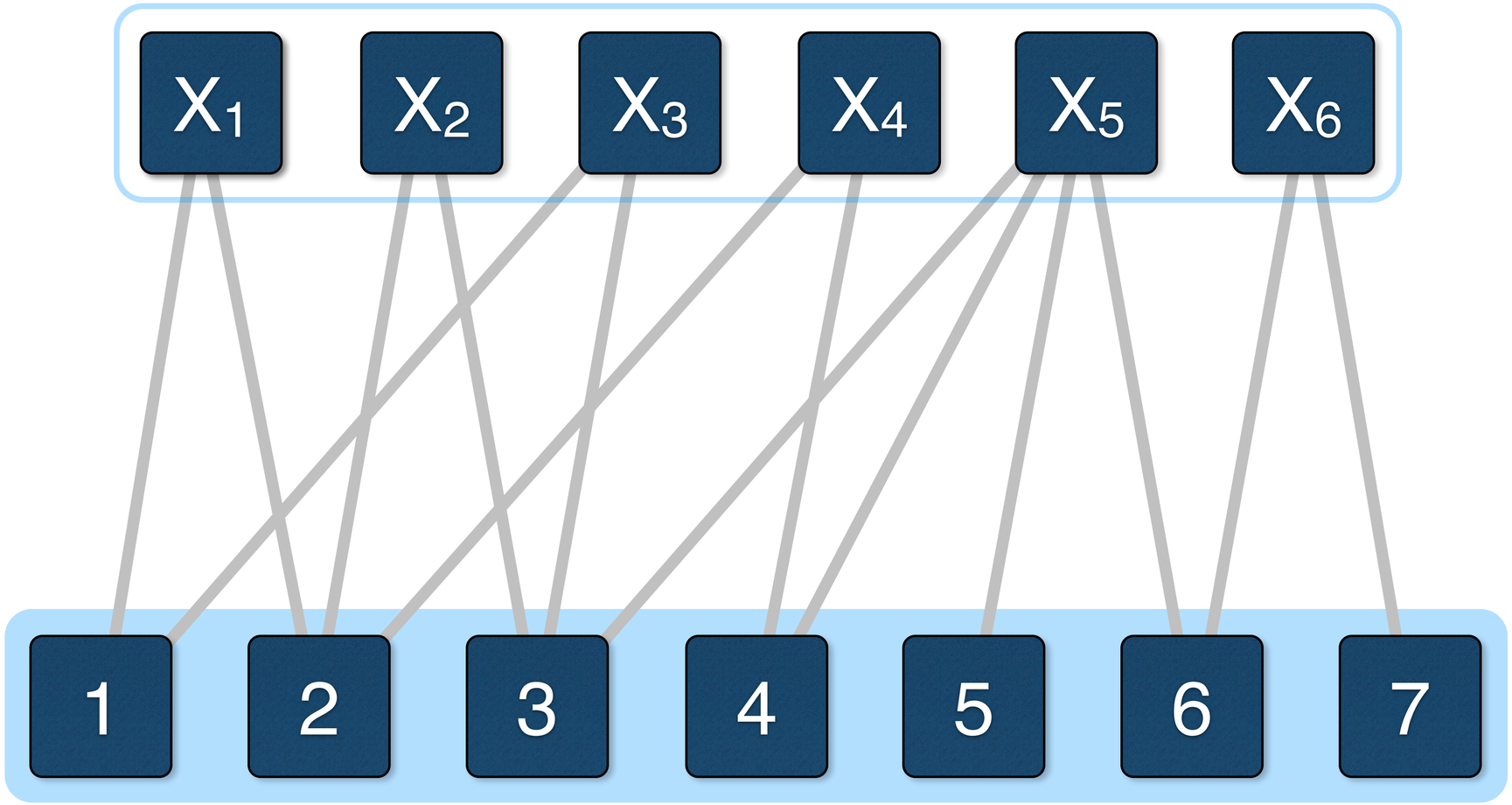


This is a feasible solution!

Alldifferent Constraint

- Bipartite graphs
  - a graph with two types of vertices
  - edges only between vertices of different types
- Alldifferent constraint
  - vertices for the variables
  - vertices for the values
  - edges between variables and values

Alldifferent Feasibility

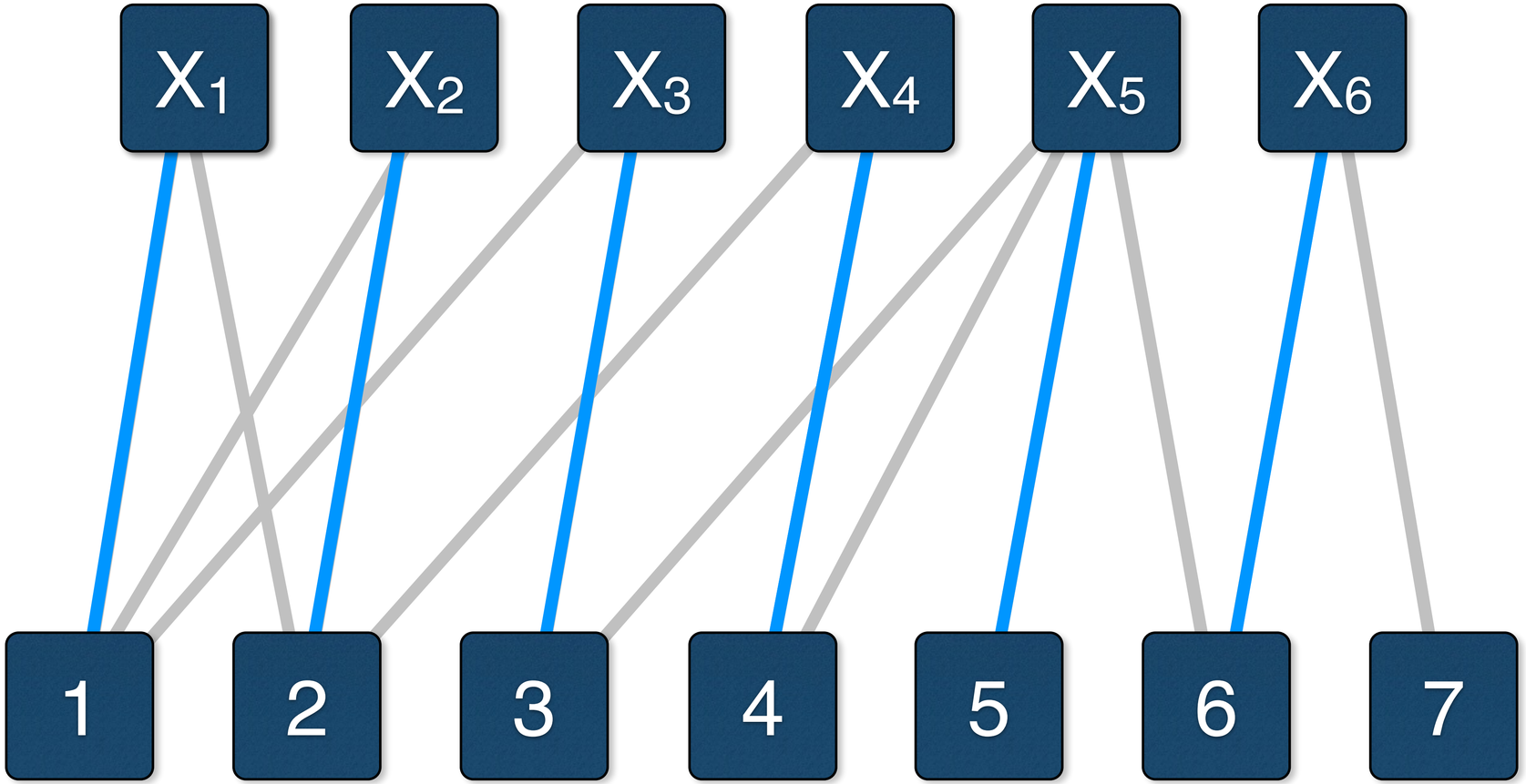


Matching and Alldifferent

- A **matching** for a graph $G=(V,E)$ is a set of edges in E such that no two edges in E share a vertex.
- A **maximum matching** M for a graph G is a matching with the largest number of edges
- Feasibility
  - finding a maximum matching in a bipartite graph

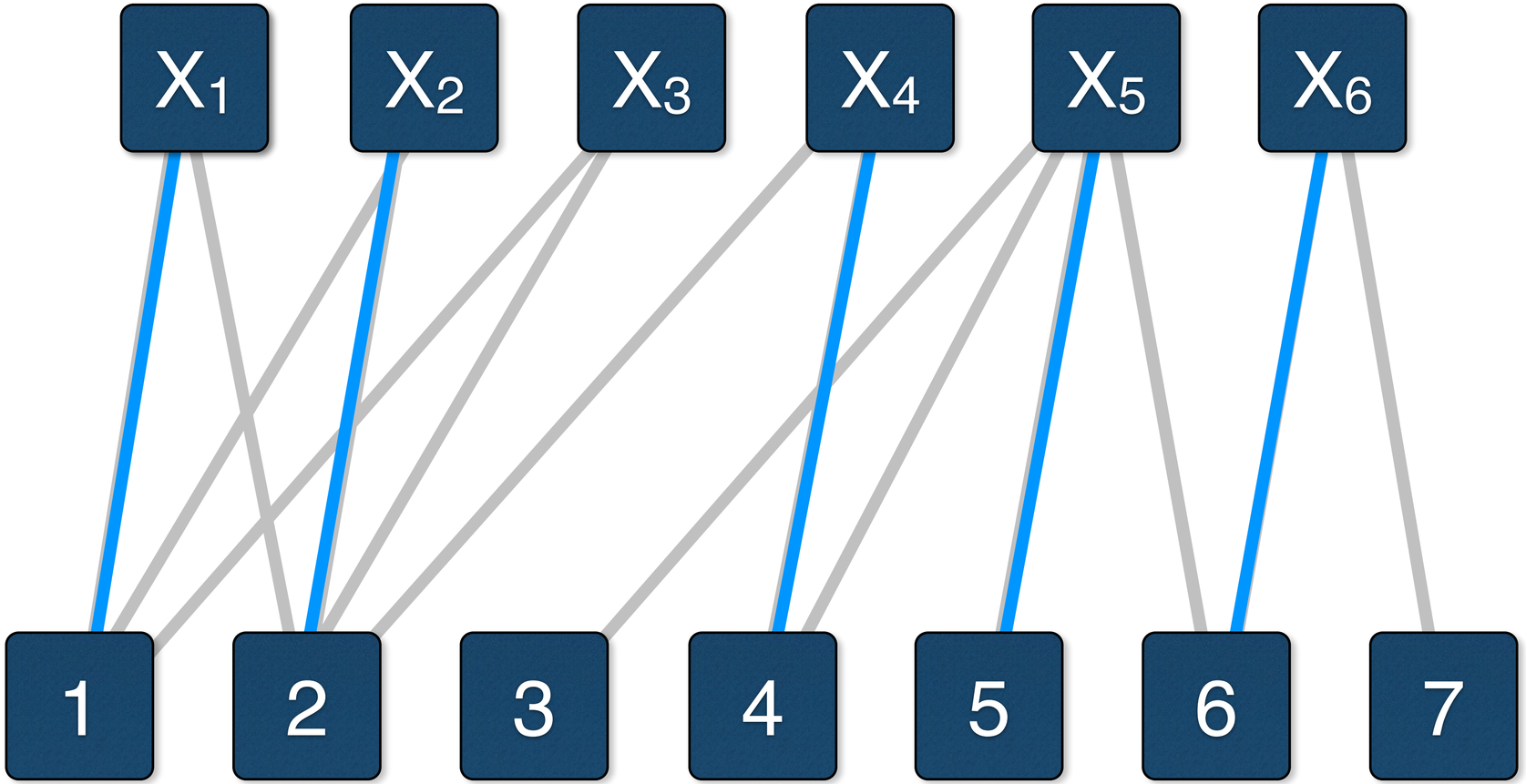
if the maximum matching has a size equal to the number of variables, then the constraint is feasible; otherwise, it is not feasible

Alldifferent Feasibility



The constraint is feasible

Alldifferent Feasibility

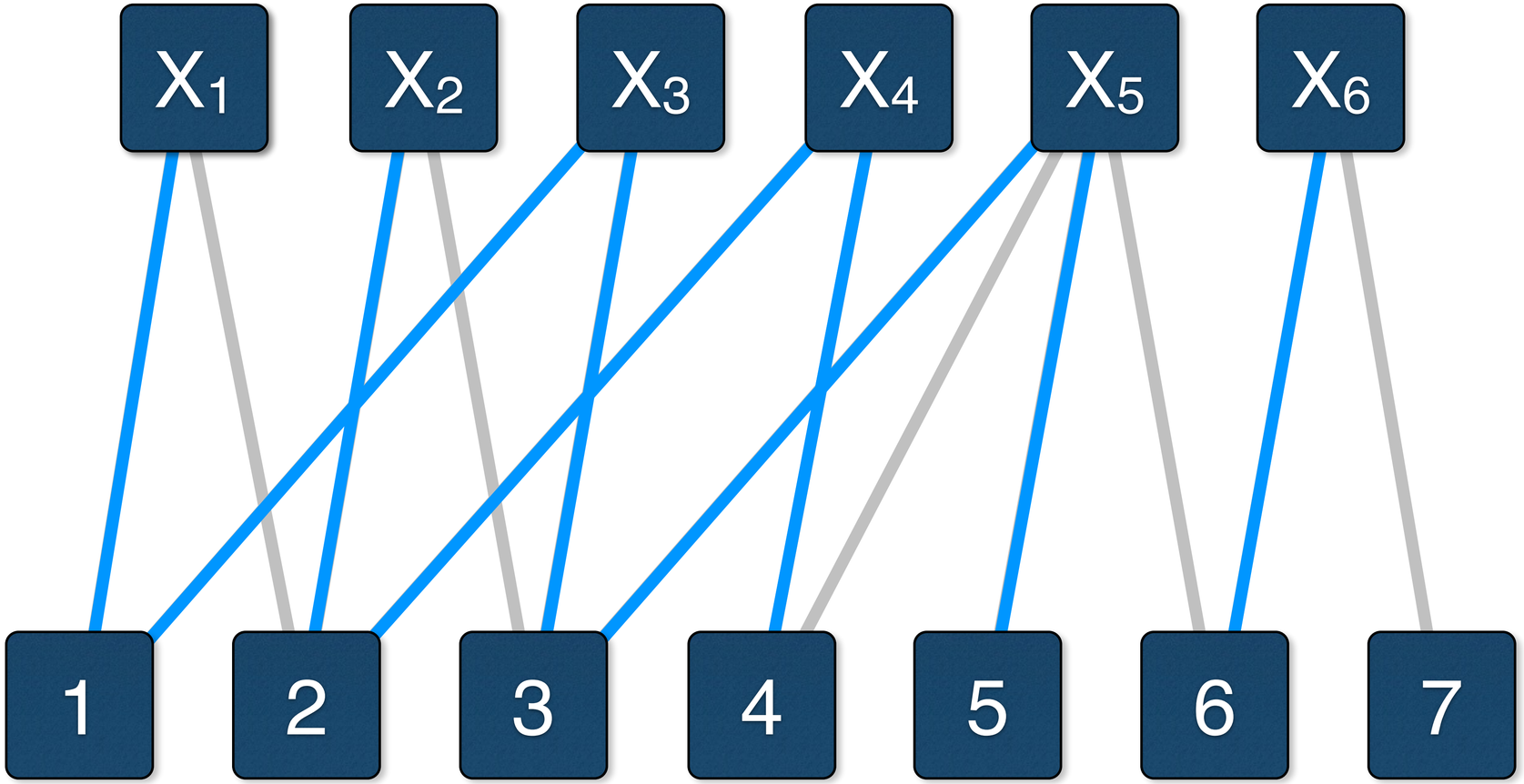


The constraint is not feasible

How to Find a Maximum Matching?

- How to find a maximum matching?
  - start with any matching
  - improve the matching
- When no improvement is possible
  - we have a maximum matching

How to Improve a Matching?



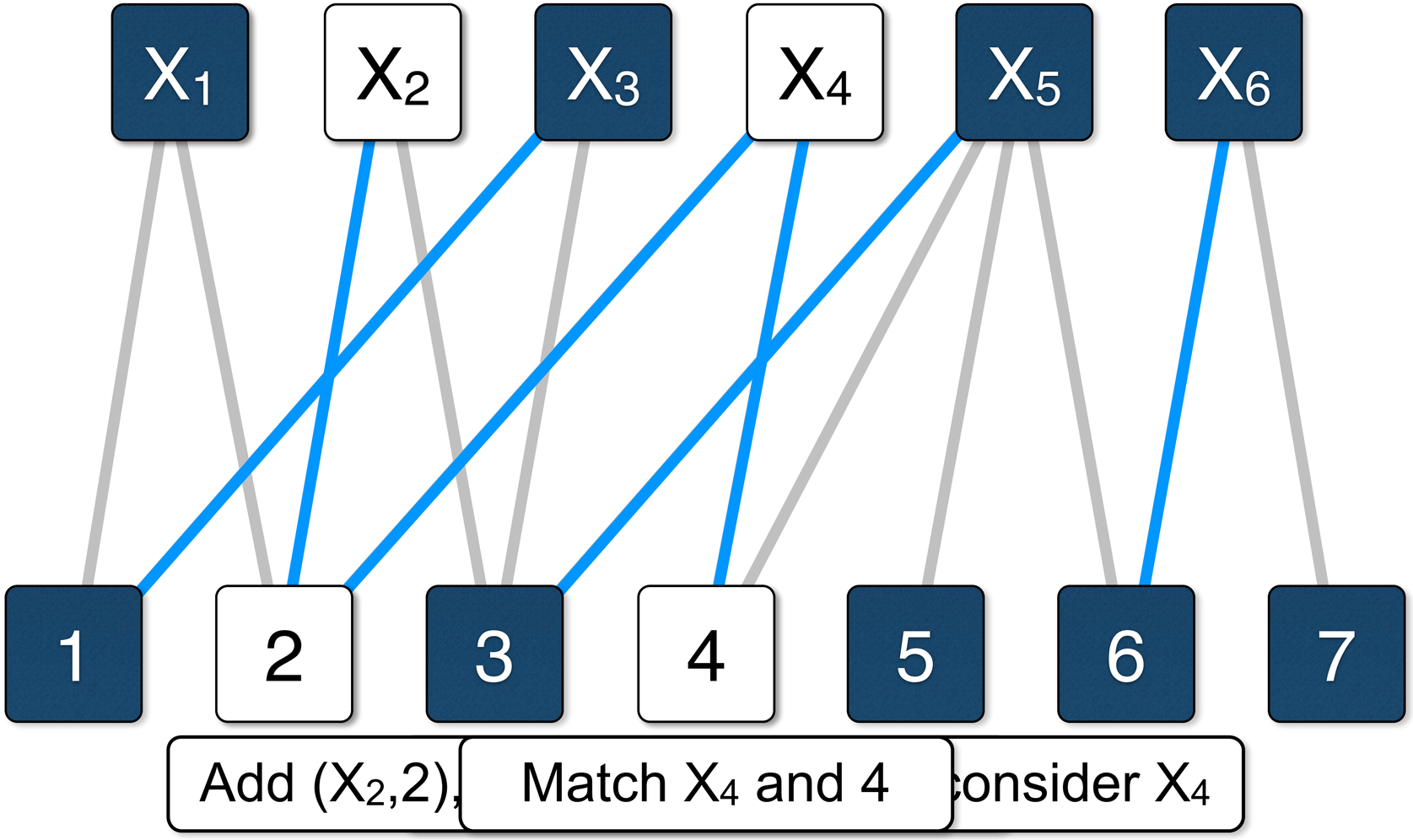
Replace $(X_3,1)$ and $(X_5,3)$ by $(X_1,1)$, $(X_3,3)$, $(X_5,5)$

How to Find a Maximum Matching?

- How to find a maximum matching?
  - start with any matching
  - improve the matching
- How to find an improvement?

1. start from a free vertex x
2. if there is an edge (x,v) where v is not matched, then insert (x,v) in the matching
3. otherwise, take a vertex v matched to y. remove (y,v) and add (x,v) from the matching and restart at step 2 with y instead of x

How to Improve a Matching?



Alternating Paths

- An alternating path P for a matching M is a path from a vertex x in X to a vertex v in V (both of which are free) such that the edges in the path are alternatively in $E \backslash M$ and M.

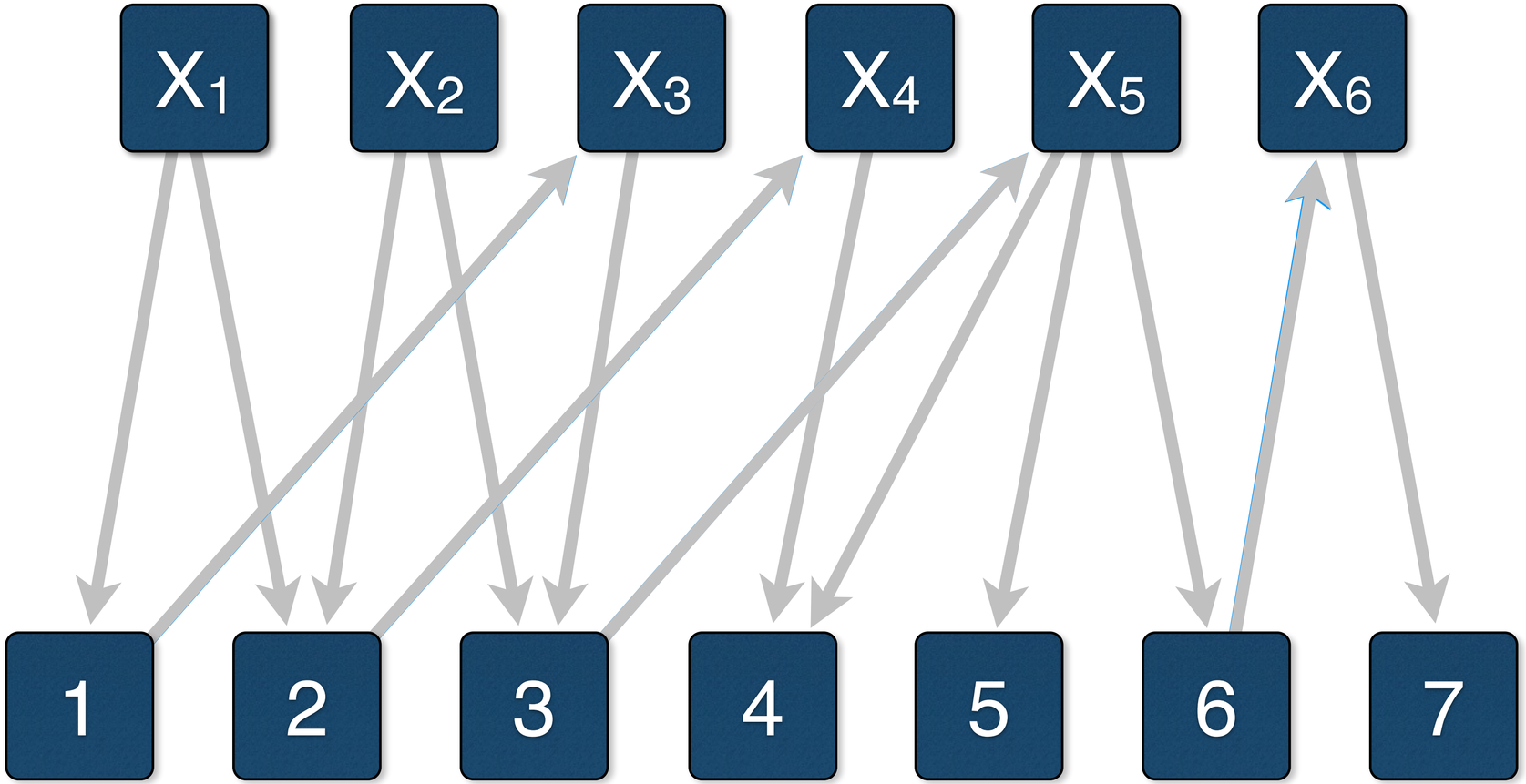
- Odd number of edges
  - why?

- We improve the matching!

How to Find Alternative Paths

- Create a directed graph.
- Given a matching
  - edges in the matching are oriented from bottom to top
  - edges not in the matching are oriented from top to bottom

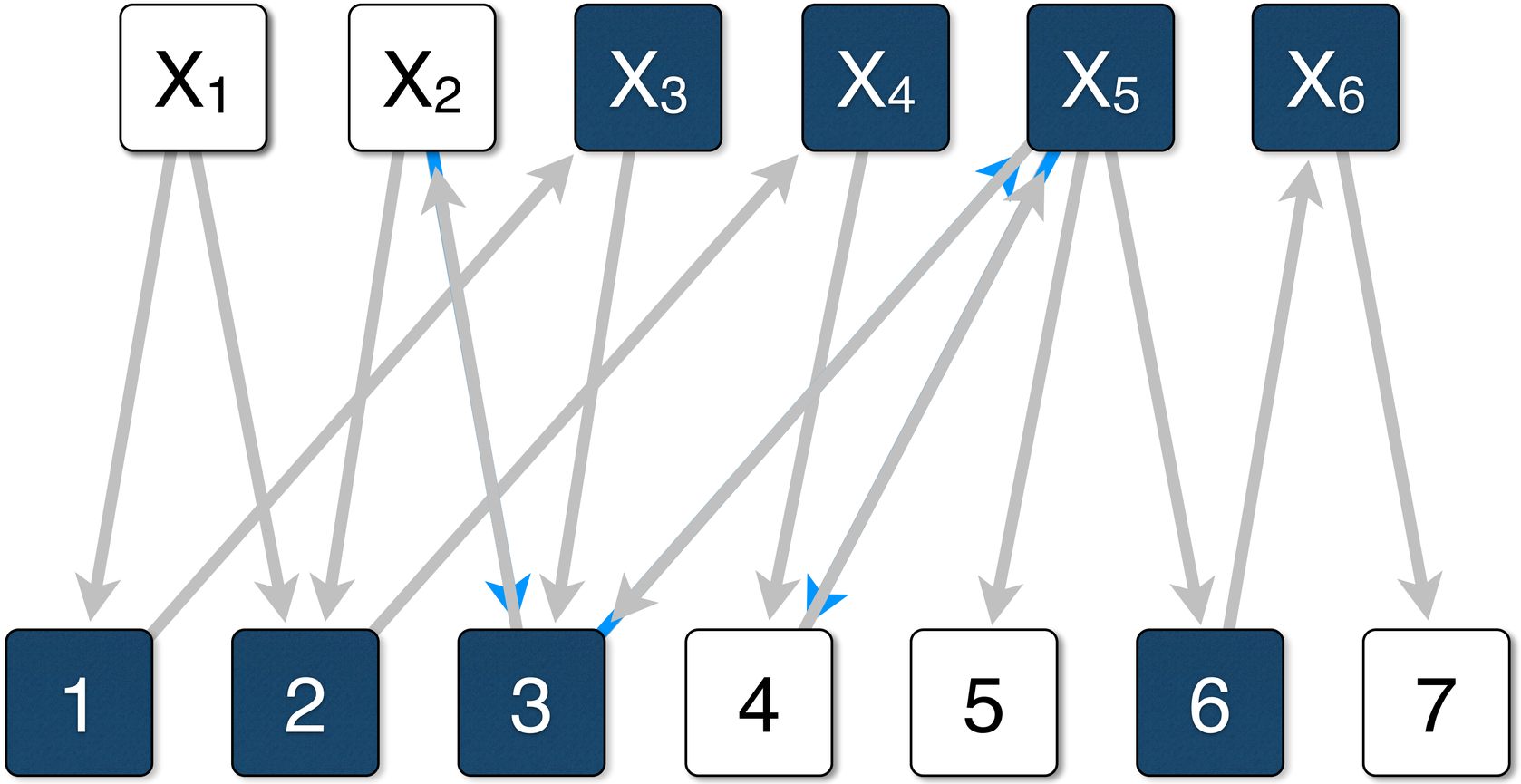
How to Improve a Matching?



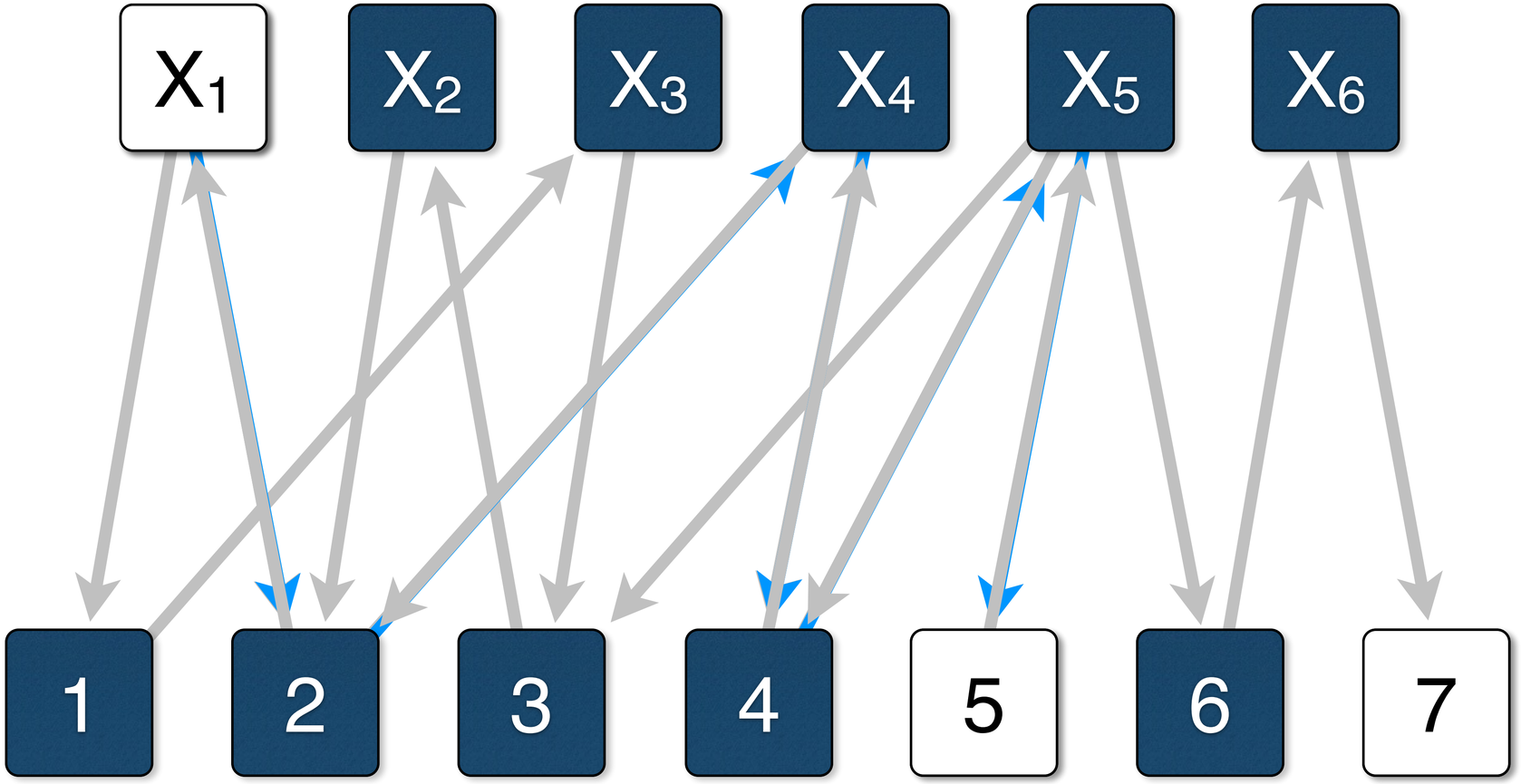
How to Find Alternative Paths

- Create a directed graph.
- Given a matching
  - edges in the matching are oriented from bottom to top
  - edges not in the matching are oriented from top to bottom
- An alternating path is thus a path starting from a free vertex x and ending in another free vertex v

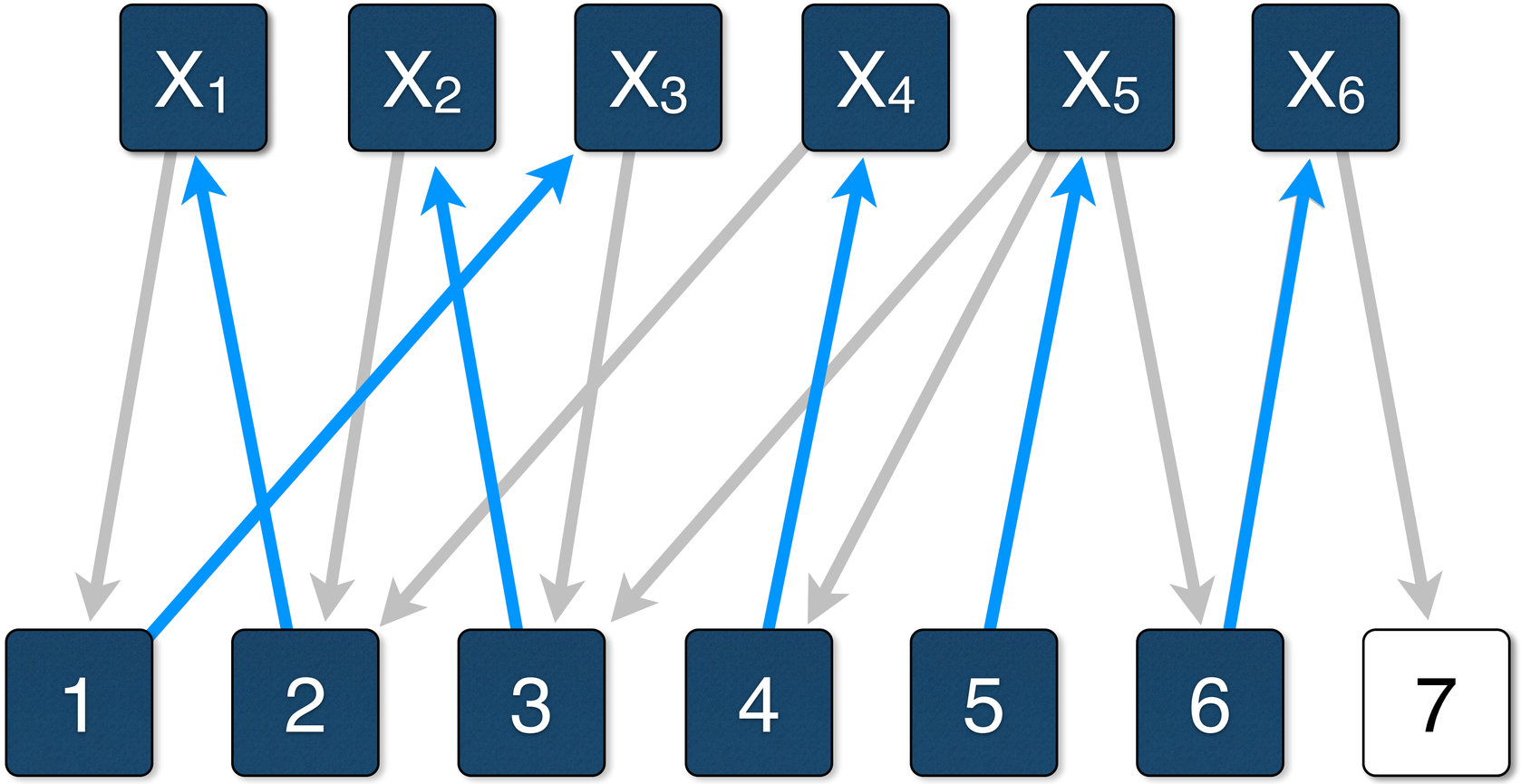
How to Find Alternative Paths



How to Find Alternative Paths



How to Find Alternative Paths



How to Find Alternative Paths

- Given a matching
  - edges in the matching are oriented from bottom to top
  - edges not in the matching are oriented from top to bottom
- An alternating path is thus a path starting from a free vertex x and ending in another free vertex v
- How to find an alternative path?
  - use depth-first or best-first search
  - $O(|X| + |E|)$ where X is the set of vertices and E is the set of edges

Feasibility of the Alldifferent constraint

- Use a bipartite graph
  - vertex set for the variables
  - vertex set for the values
  - edge *(x,v)* if *v* is in *D(x)*
- Feasibility
  - alldifferent is feasible iff the size of the maximum matching is the number of variables
- Finding a maximum matching
  - improve a matching using alternating paths in the directed graph obtained by a proper orientation of the edges

How to Prune?

- Pruning is equivalent to
  - *v* must be removed from the domain of *x* if the edge *(x,v)* appears in no maximum matching
- Only need to look at the edges not present in the maximum matching
- Naive approach,
  - force the edge *(x,v)* in the matching, i.e., remove all other edges *(x,w)*.
  - search for a maximum matching.
  - If it is smaller than the number of variables, *v* must be removed from *D(x)*

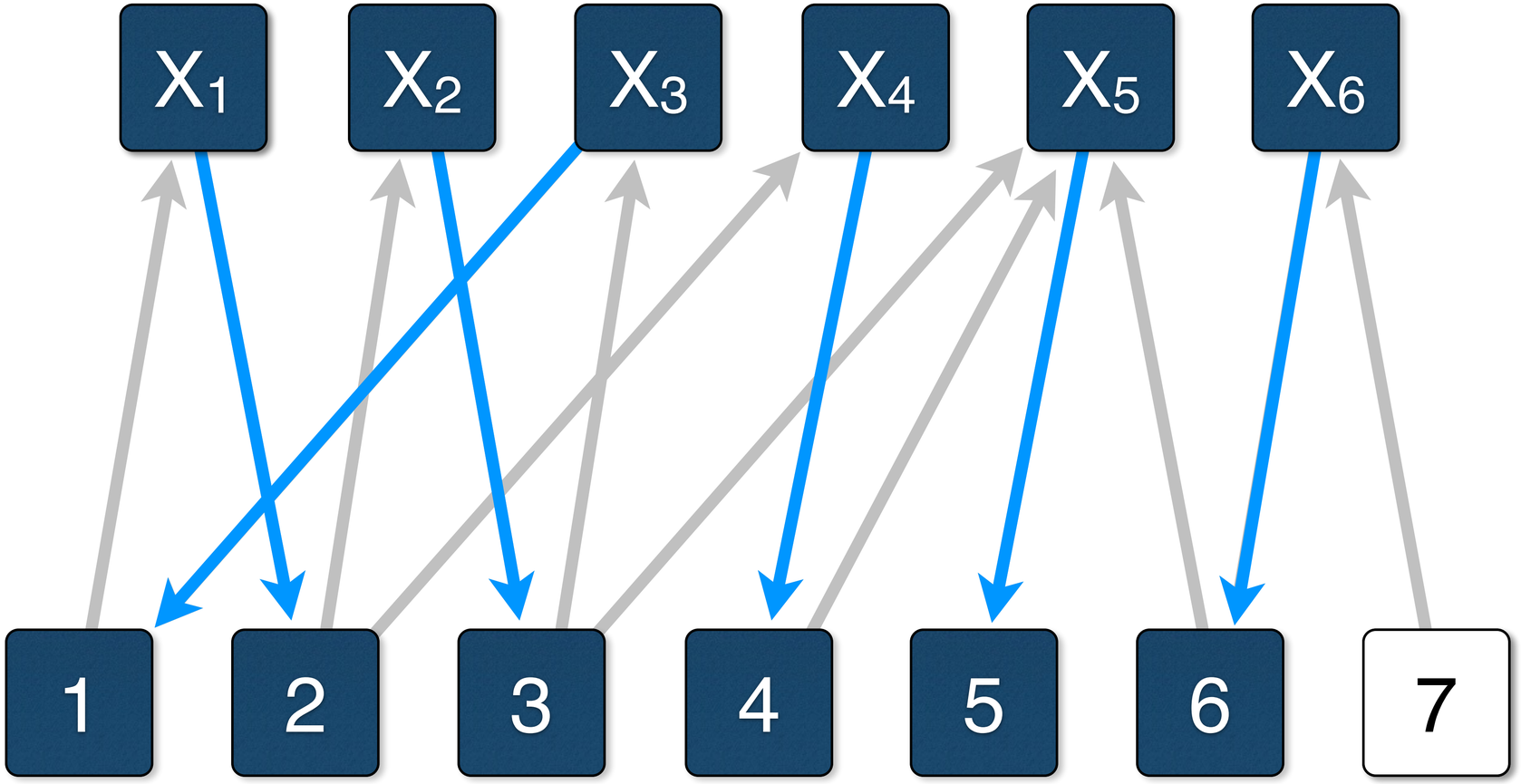
How to Prune?

- Basic property (Berge, 1970)
  - an edge belongs to some but not all maximum matchings iff, given a maximum matching *M*, it belongs to either
    - an even alternating path starting at a free vertex
    - an even alternating cycle
- Note that
  - the edges not in the maximum matching do not belong to all maximum matchings
  - the above property tells us whether they belong to at least one maximum matching
  - the free vertices are ?

Pruning

- Create a directed graph like before but reserve the direction of the edges
- Given a matching
  - edges in the matching are oriented from top to bottom
  - edges not in the matching are oriented from bottom to top

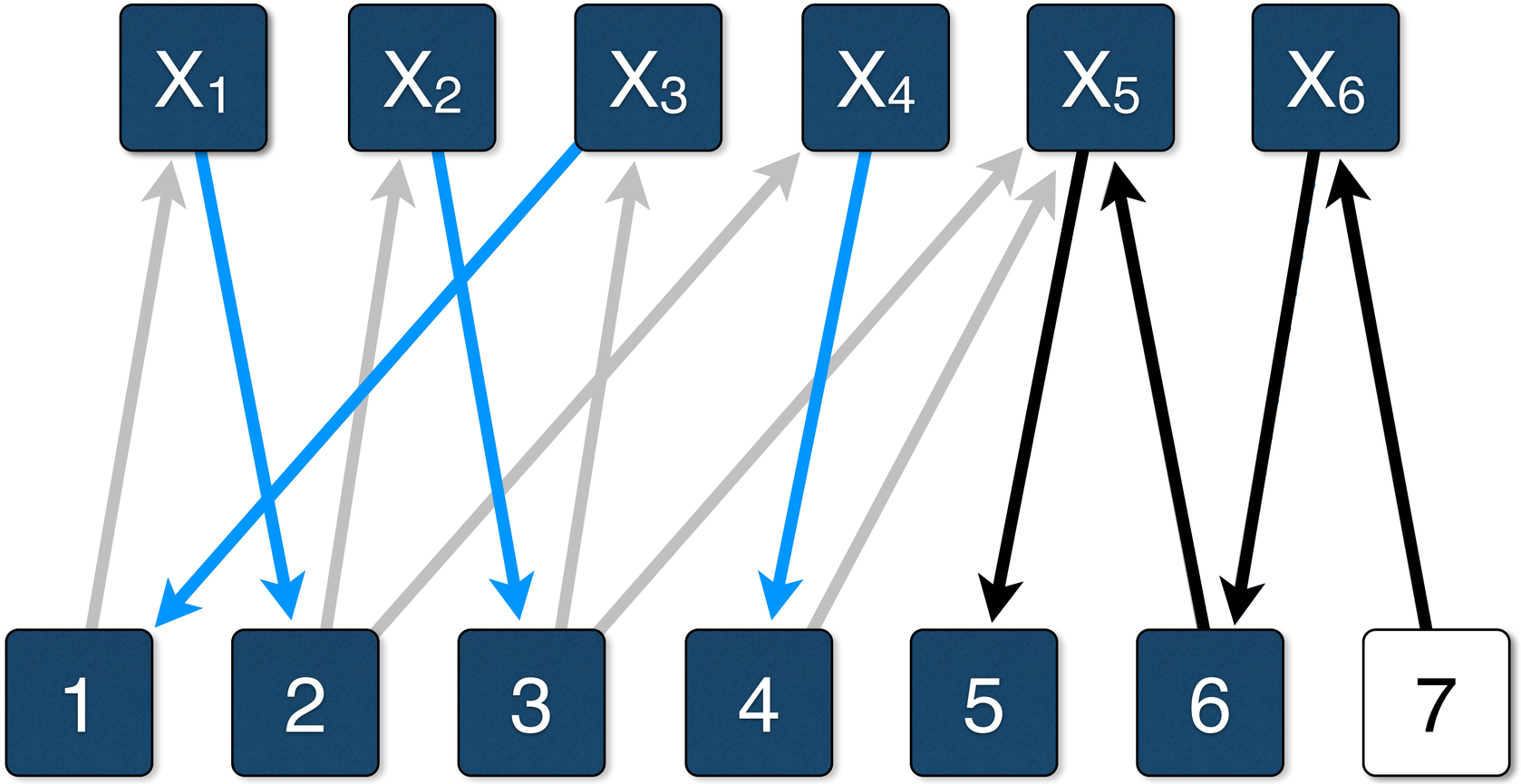
How to Find Alternative Paths



Pruning

- Given a matching M, create a directed graph like before but reserve the direction of the edges
- Search for even alternating path starting from a free vertex: P

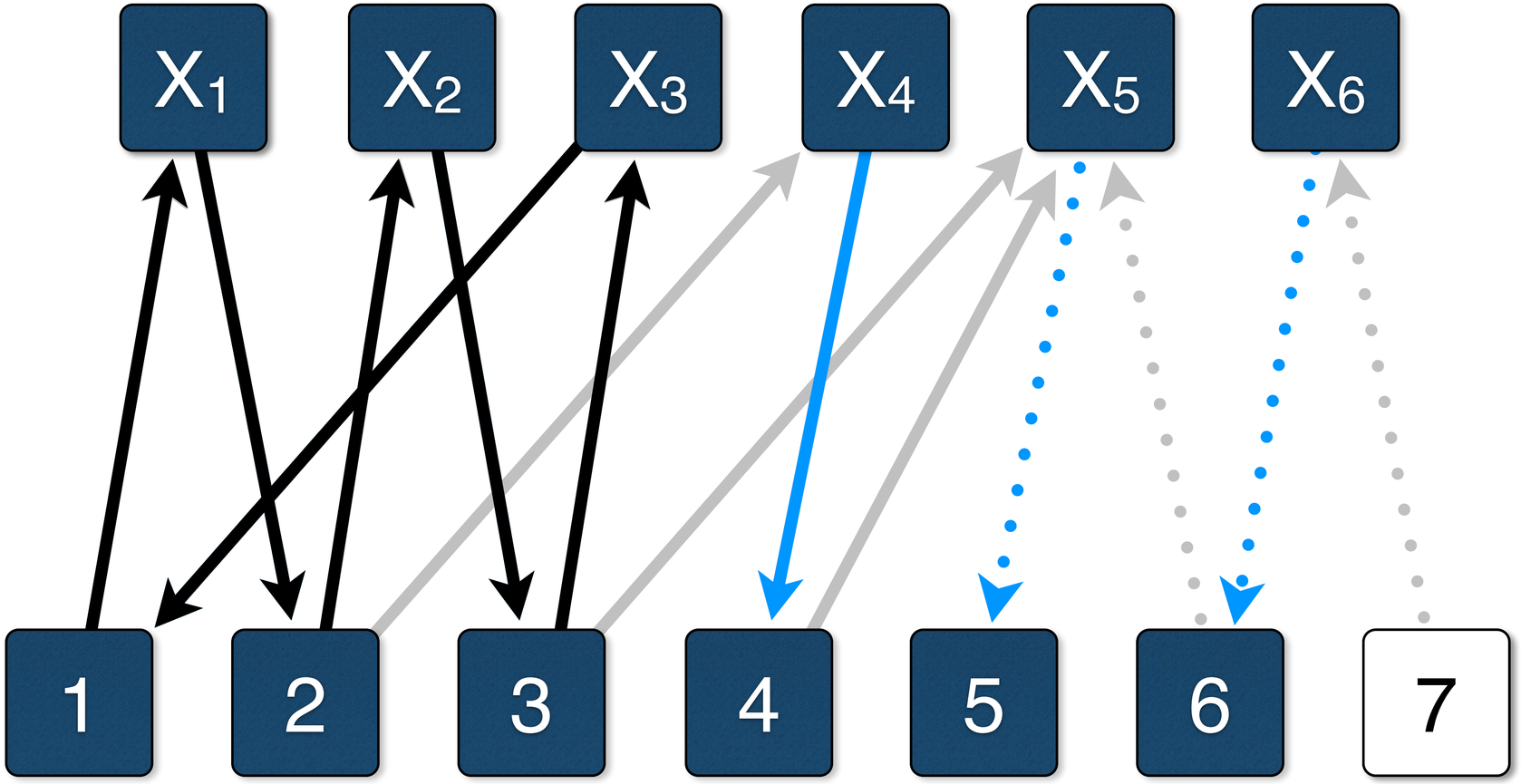
Even Alternating Path from a Free Vertex



Pruning

- Given a matching M, create a directed graph like before but reserve the direction of the edges
- Search for even alternating path starting from a free vertex: P
- Search for all strongly connected components and collect all the edges belonging to them: C

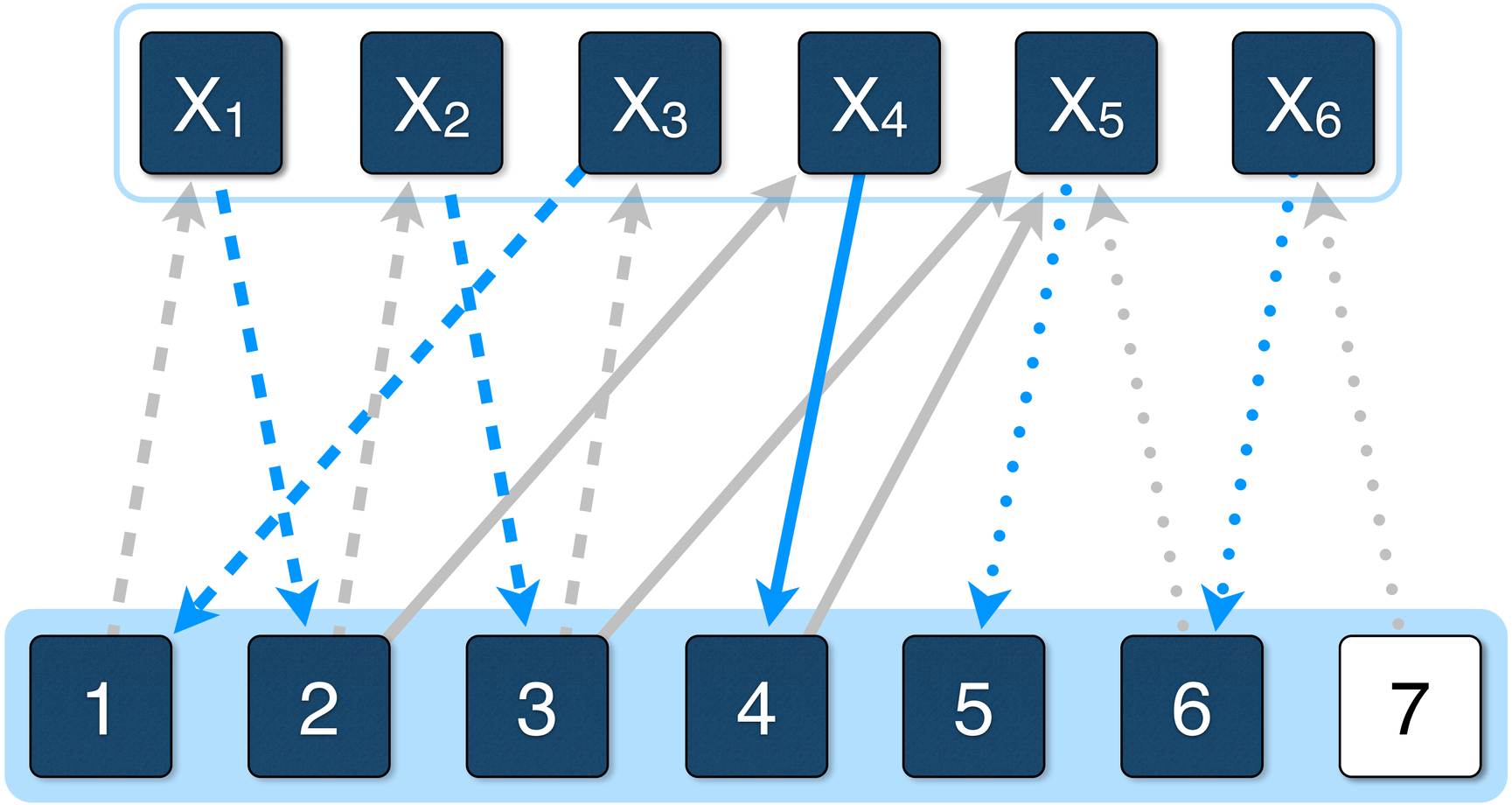
Even Alternating Cycles



Pruning

- Given a matching M, create a directed graph like before but reserve the direction of the edges
- Search for even alternating path starting from a free vertex: P
- Search for all strongly connected components and collect all the edges belonging to them: C
- Remove all edges not in M, P, or C

The Pruned Domains



Pruning

- Given a matching *M*, create a directed graph like before but reserve the direction of the edges
- Search for even alternating path starting from a free vertex: *P*
- Search for all strongly connected components and collect all the edges belonging to them: *C*
- Remove all edges not in *M*, *P*, or *C*
- *Complexity: O((|X| + |V|) * |E|)*

Discrete Optimization

Constraint Programming: Part IX

Goals of the lecture

- Search in constraint programming
  - introduction
  - active research area

Search in constraint programming

- Key idea
  - use feasibility information for branching

Search in constraint programming

- Key idea
  - use feasibility information for branching
- First-fail principle
  - try first where you are the most likely to fail

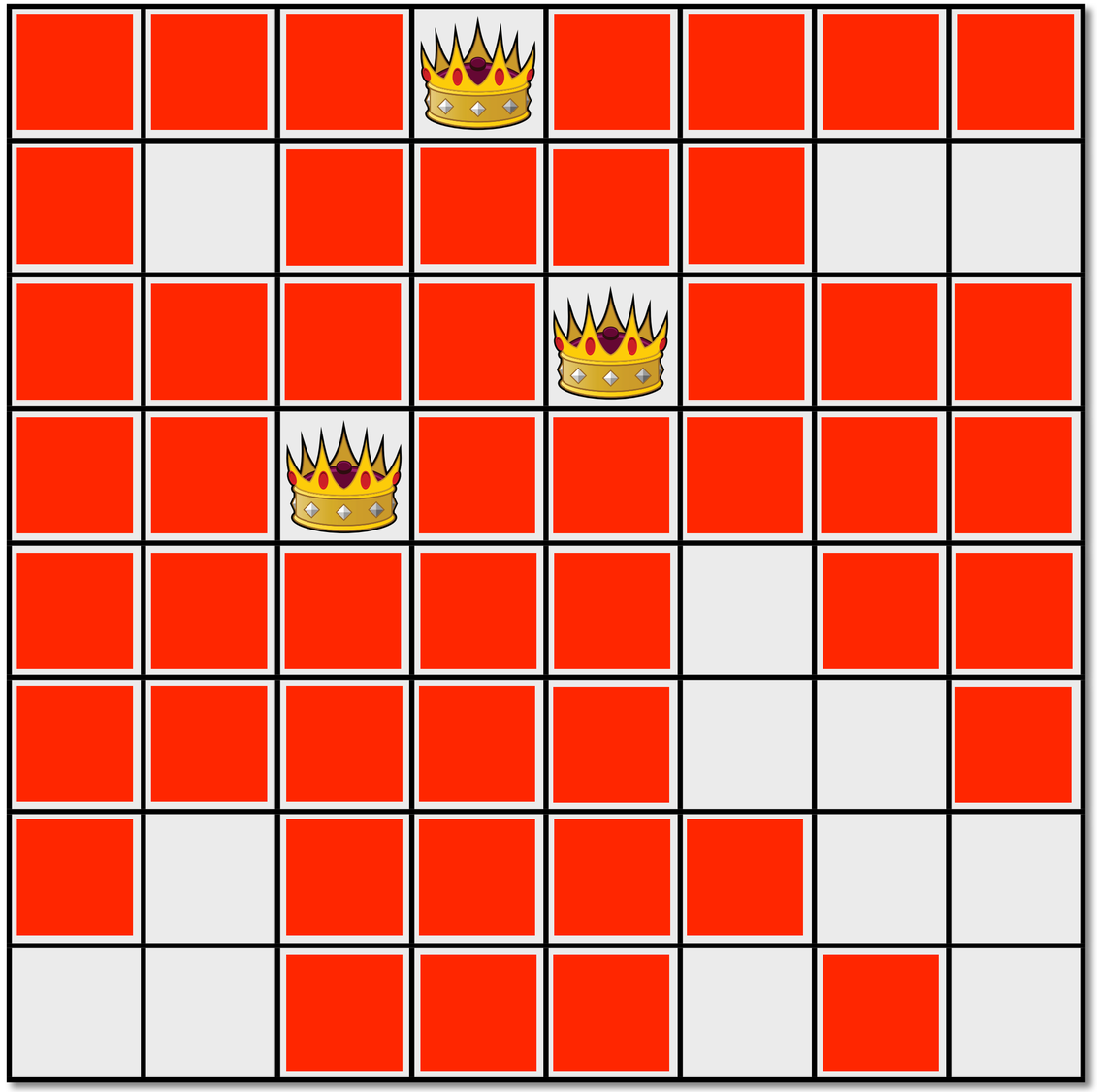
Search in constraint programming

- Key idea
  - use feasibility information for branching
- First-fail principle
  - try first where you are the most likely to fail
- Why the first-fail principle?
  - do not spend time doing easy stuff first and avoid redoing the difficult part

Search in constraint programming

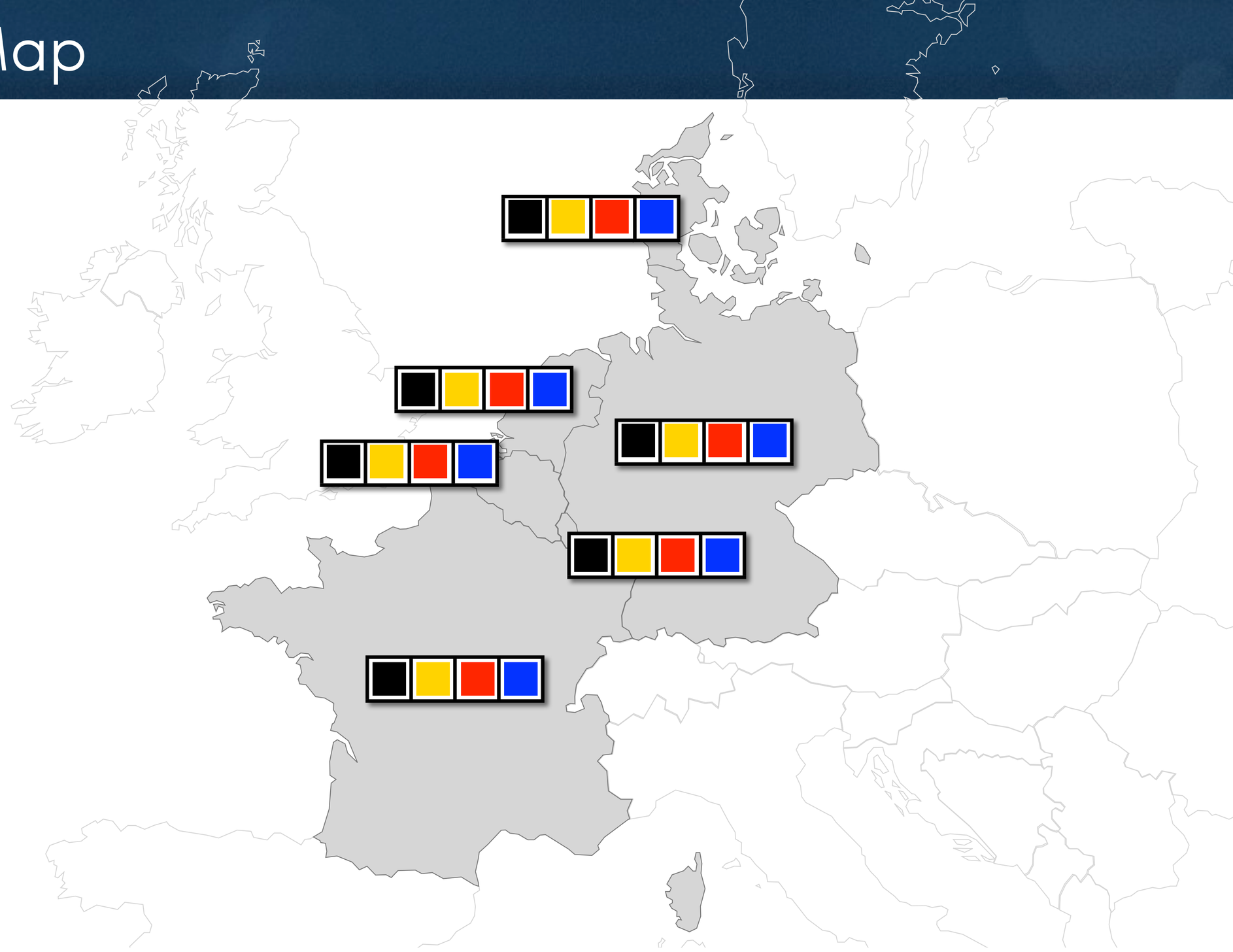
- Key idea
  - use feasibility information for branching
- First-fail principle
  - try first where you are the most likely to fail
- Why the first-fail principle?
  - do not spend time doing easy stuff first and avoid redoing the difficult part
- The ultimate goal
  - creating small search trees

How to approximate the first-fail principle?



Coloring a Map

Coloring a Map



Euler Knight

- The problem
  - use a knight to visit all positions of a chessboard exactly once

- Puzzle with links
  - vehicle routing problems

Euler Knight

```
range Board = 1..64;
var{int} jump[i in Board] in Knightmoves(i);
solve {
   circuit(jump);
}
```

Euler Knight

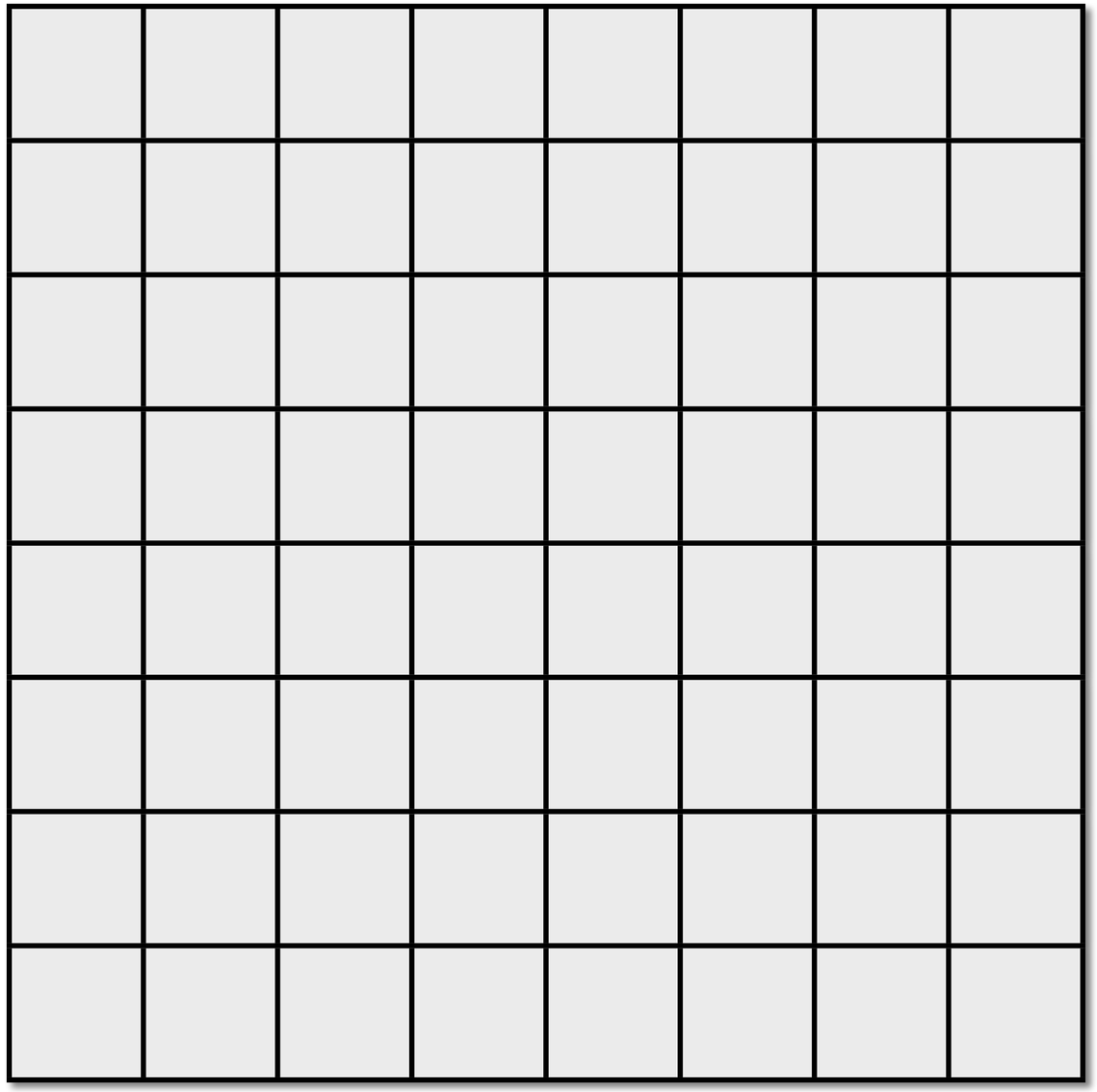
```
range Board = 1..64;
var{int} jump[i in Board] in Knightmoves(i);
solve {
   circuit(jump);
}
```

```
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);
}
```

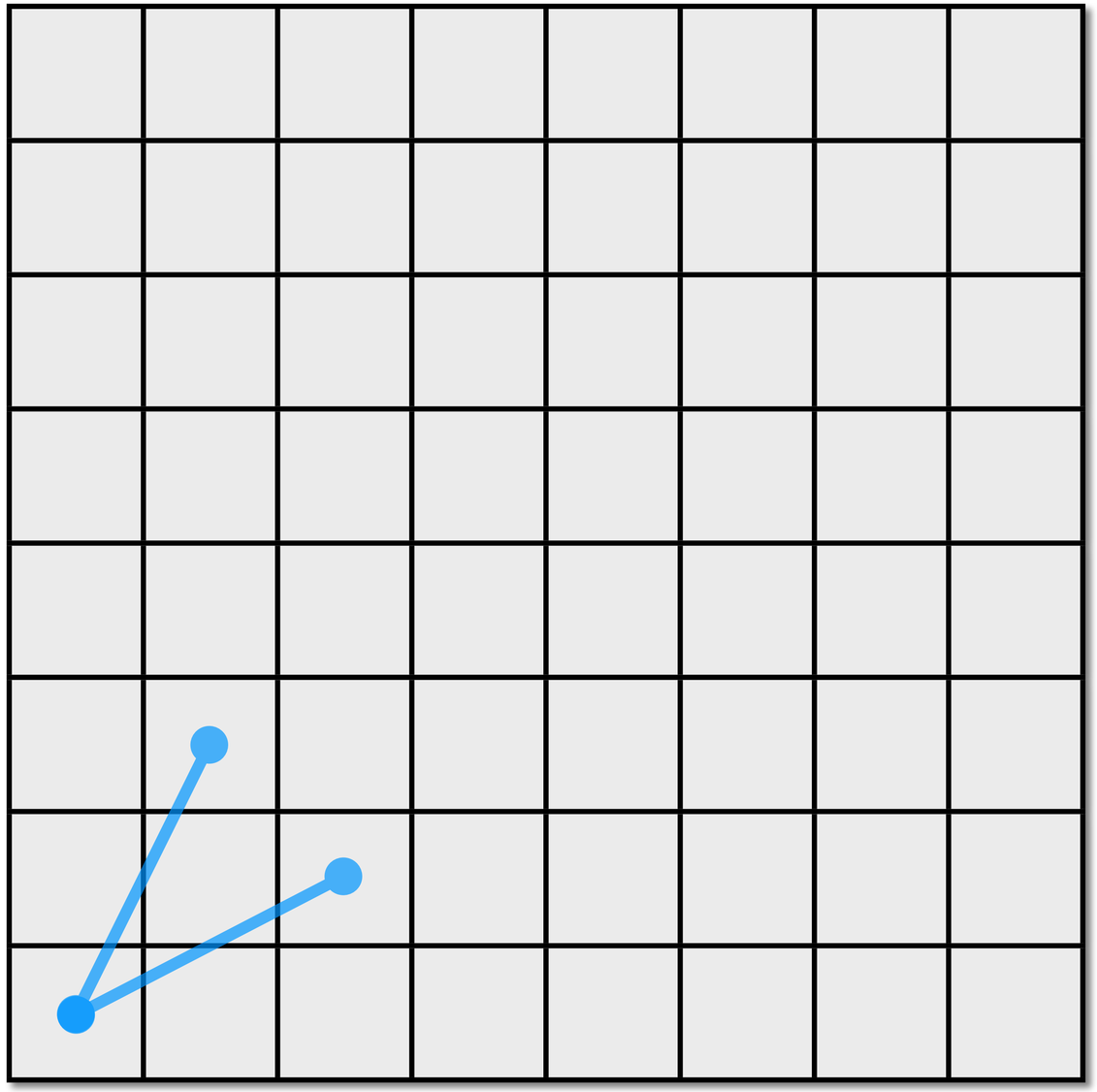
Euler Knight

- First-fail principle on the Euler Knight
  - where do we start?

Euler Knight



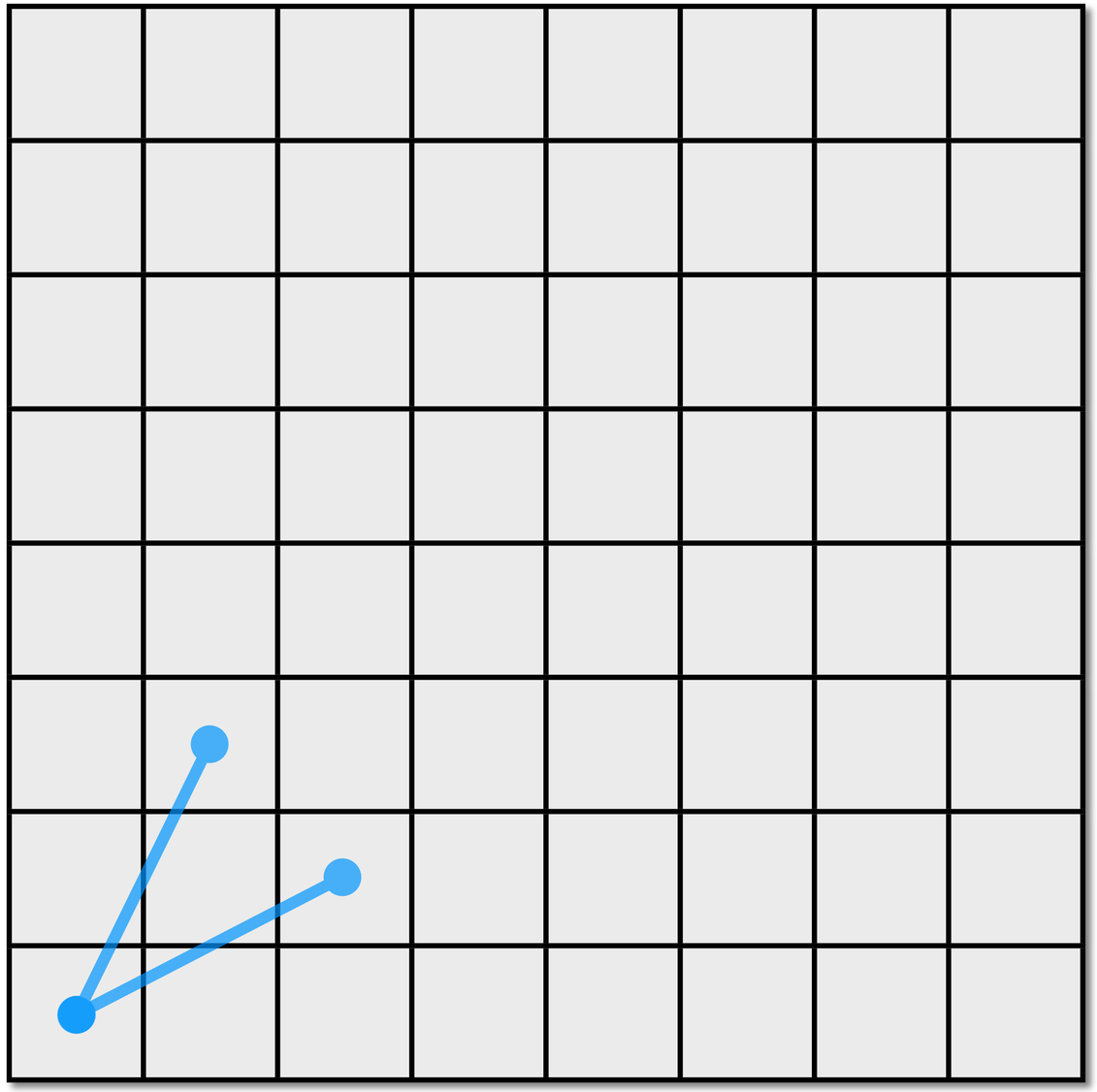
Euler Knight



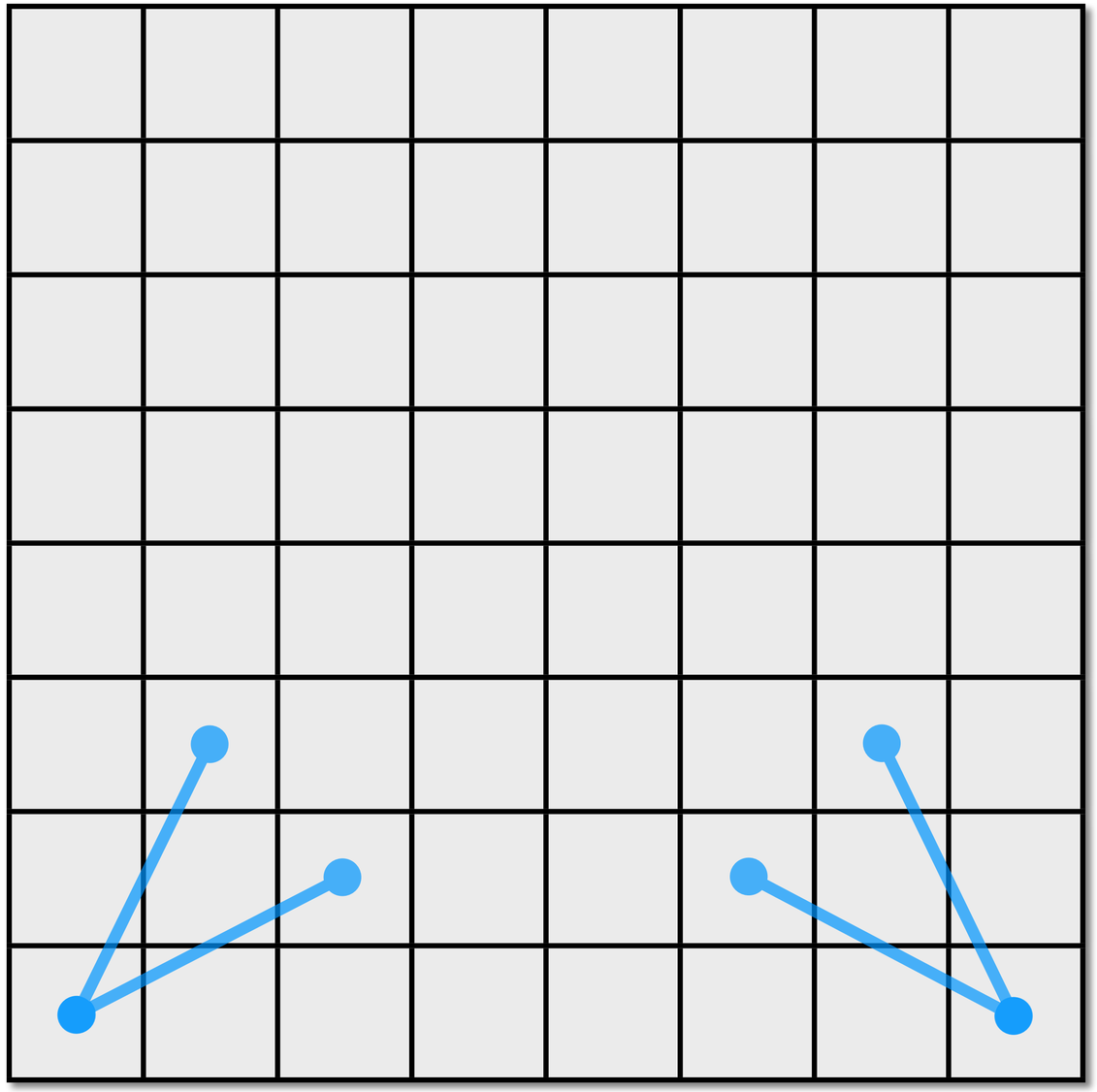
Euler Knight

- First-fail principle on the Euler Knight
  - where do we go next?

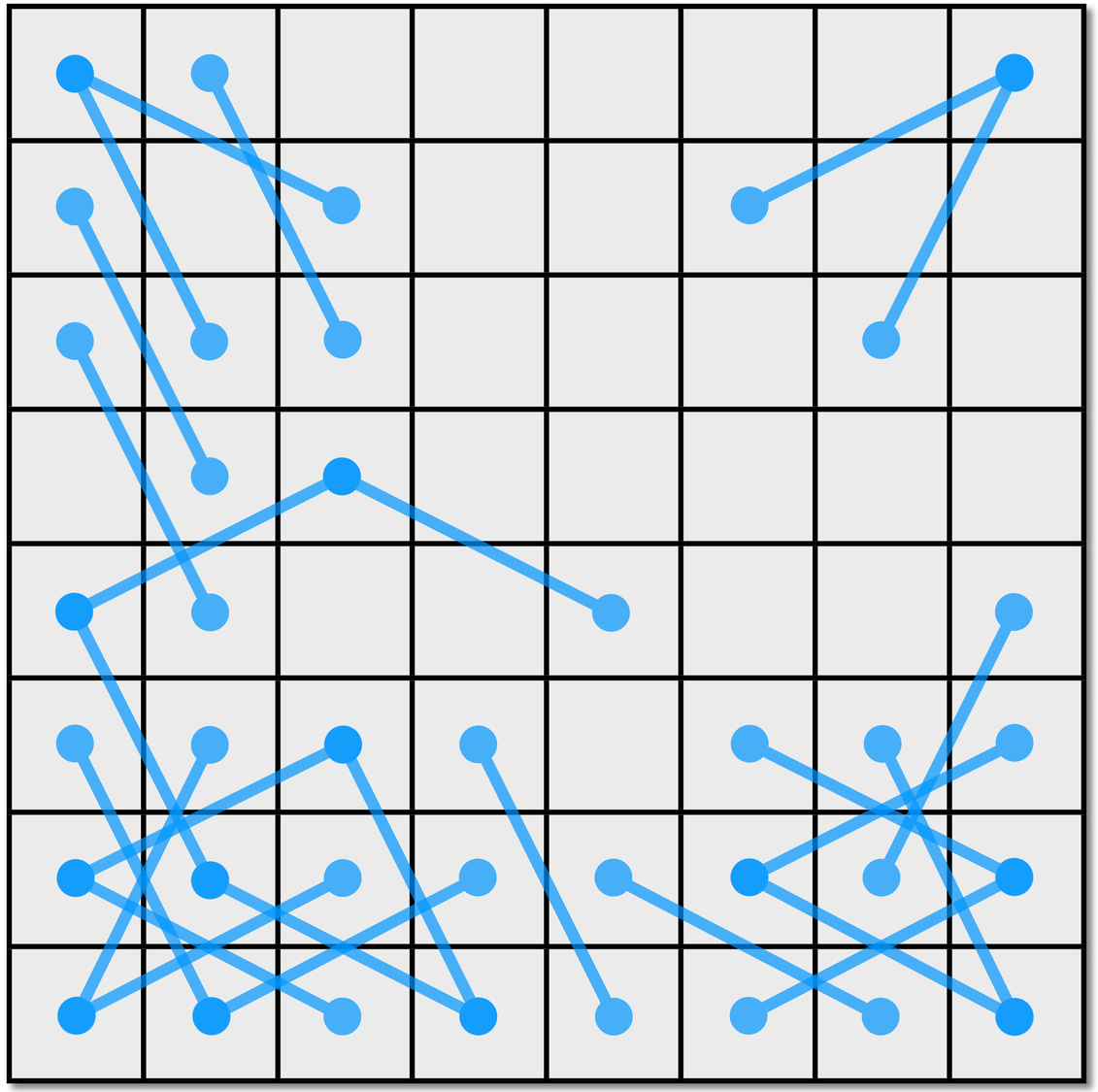
Euler Knight



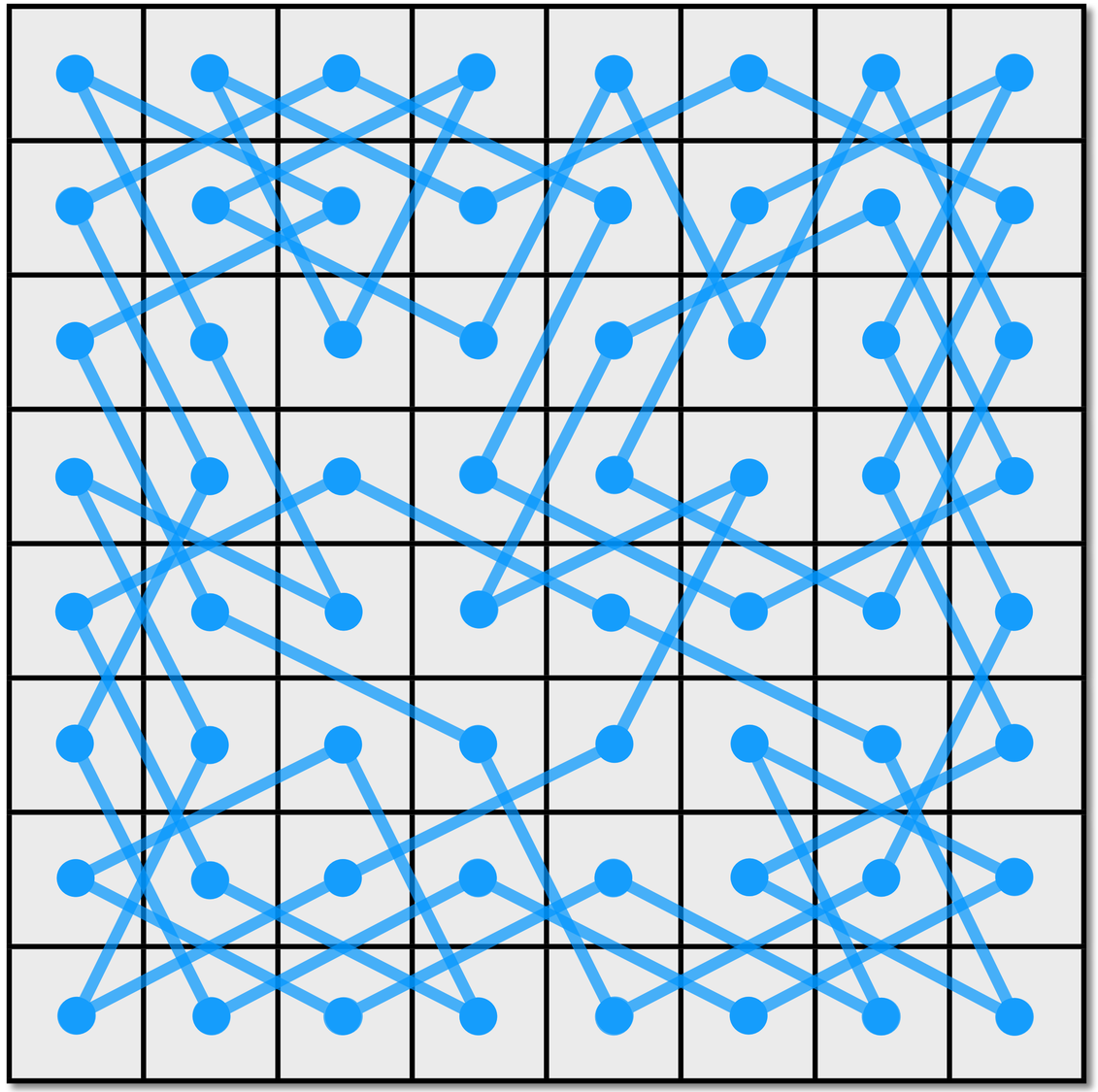
Euler Knight



Euler Knight



Euler Knight



A search procedure for the 8-queens problem

```
range R = 1..8;
var{int} row[R] in R;
solve {
   forall(i in R,j in R: i < j) {
      row[i] ≠  row[j];
      row[i] ≠  row[j] + (j - i);
      row[i] ≠  row[j] - (j - i);
   }
}
using {
   forall(r in R)
      tryall(v in R)
         row[r] = v;
}
```

A search procedure for the 8-queens problem

```
range R = 1..8;
var{int} row[R] in R;
solve {
   forall(i in R,j in R: i < j) {
      row[i] ≠  row[j];
      row[i] ≠  row[j] + (j - i);
      row[i] ≠  row[j] - (j - i);
   }
}
using {
   forall(r in R)
      tryall(v in R)
         row[r] = v;
}
```

iterate over all queens

A search procedure for the 8-queens problem

```
range R = 1..8;
var{int} row[R] in R;
solve {
   forall(i in R,j in R: i < j) {
      row[i] ≠  row[j];
      row[i] ≠  row[j] + (j - i);
      row[i] ≠  row[j] - (j - i);
   }
}
using {
   forall(r in R)
     tryall(v in R)
         row[r] = v;
}
```

nondeterministically explore all values

A search procedure for the 8-queens problem

```
range R = 1..8;
var{int} row[R] in R;
solve {
   forall(i in R,j in R: i < j) {
      row[i] ≠  row[j];
      row[i] ≠  row[j] + (j - i);
      row[i] ≠  row[j] - (j - i);
   }
}
using {
   forall(r in R)
      tryall(v in R)
         row[r] = v;
}
```

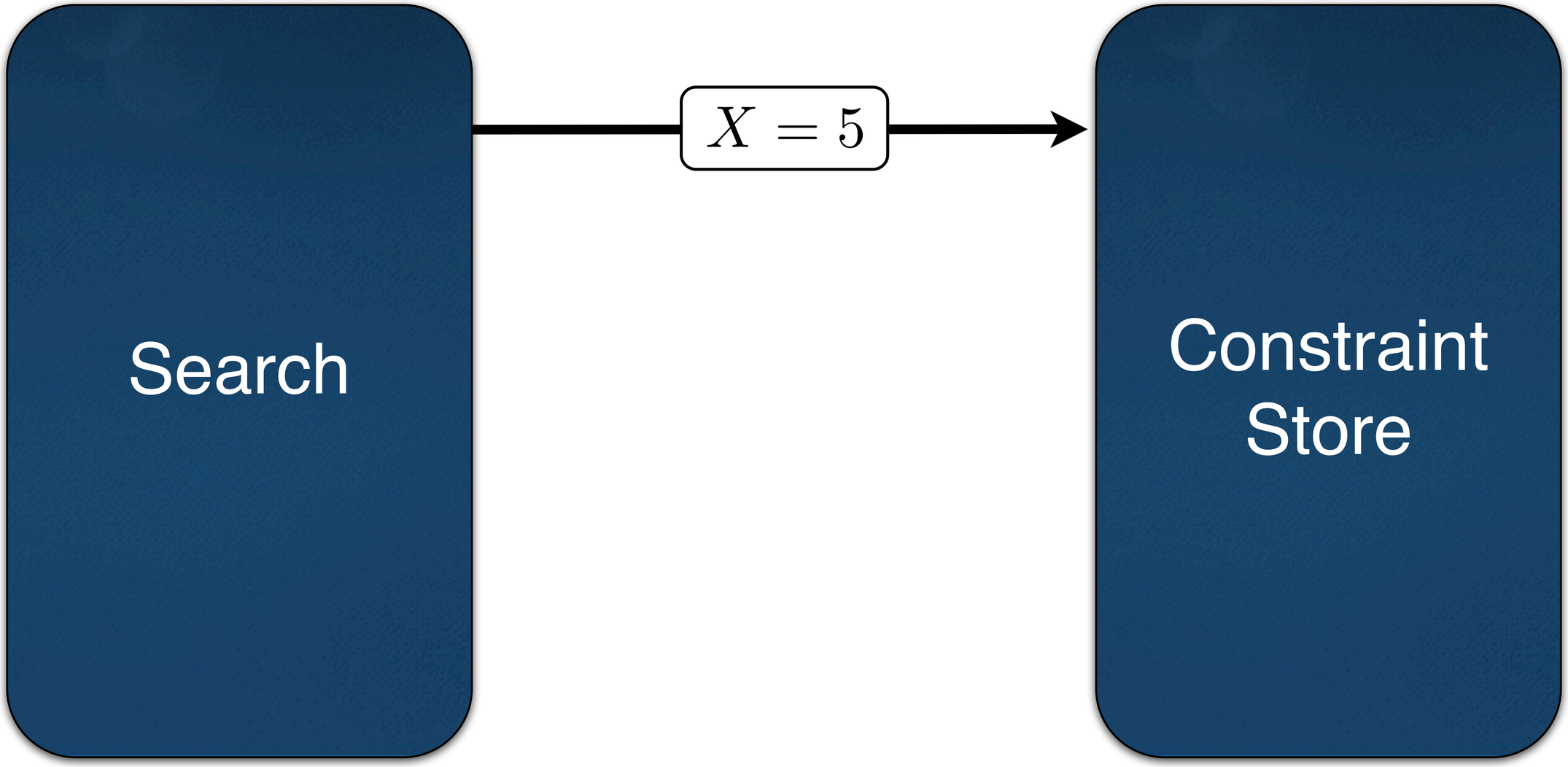
add a constraint to the constraint store

Computational Paradigm

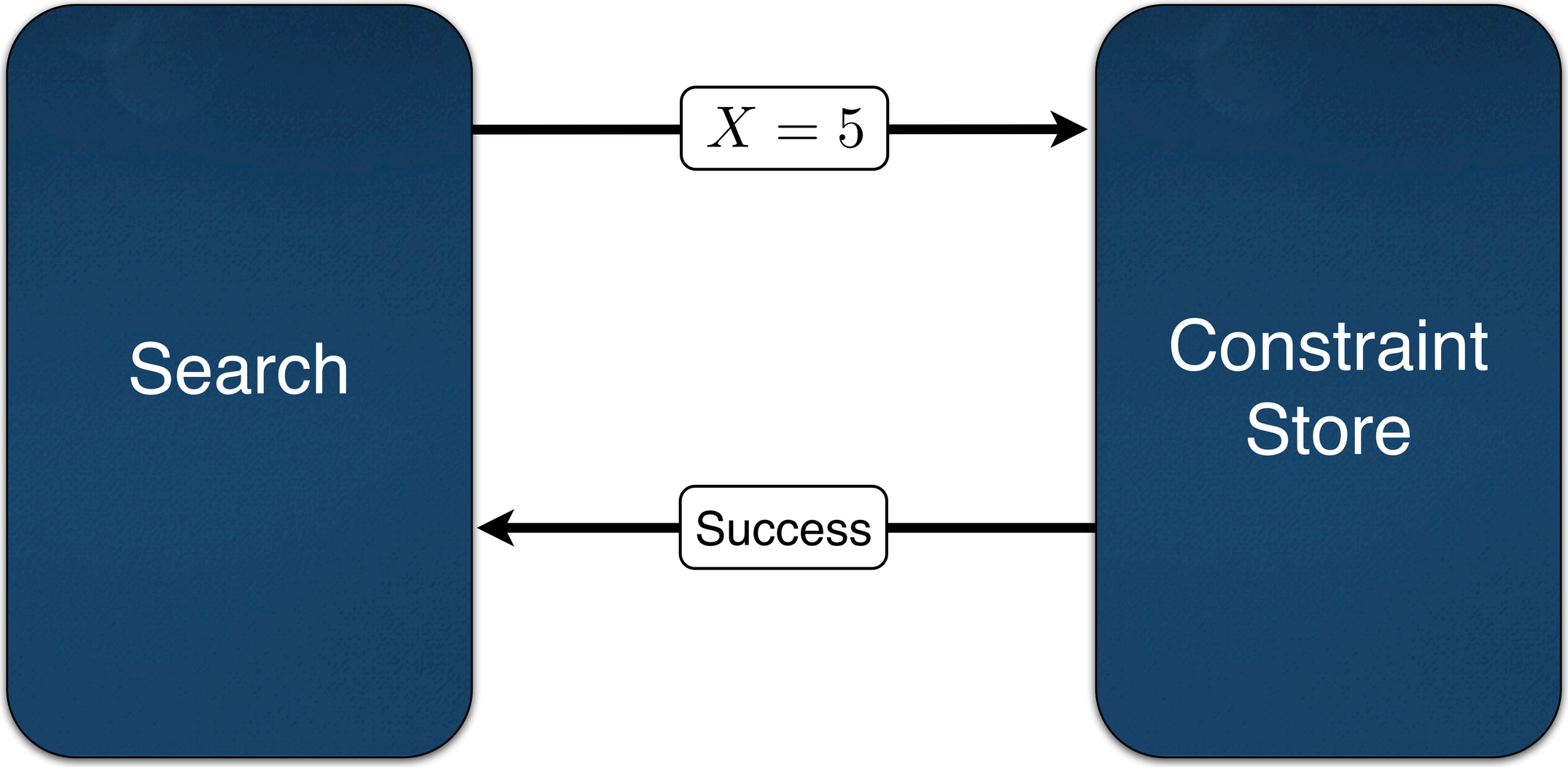
Search

Constraint Store

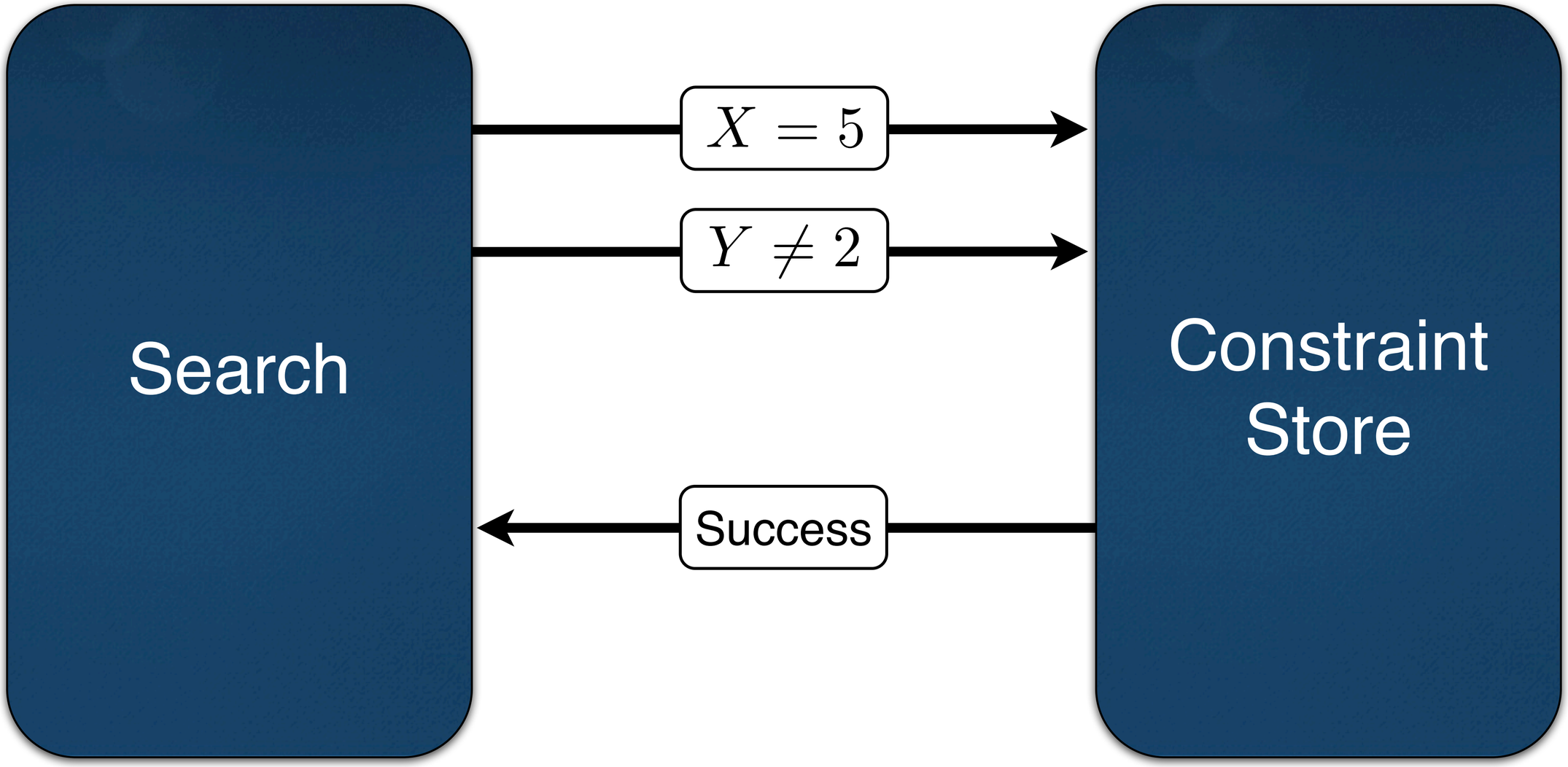
Computational Paradigm



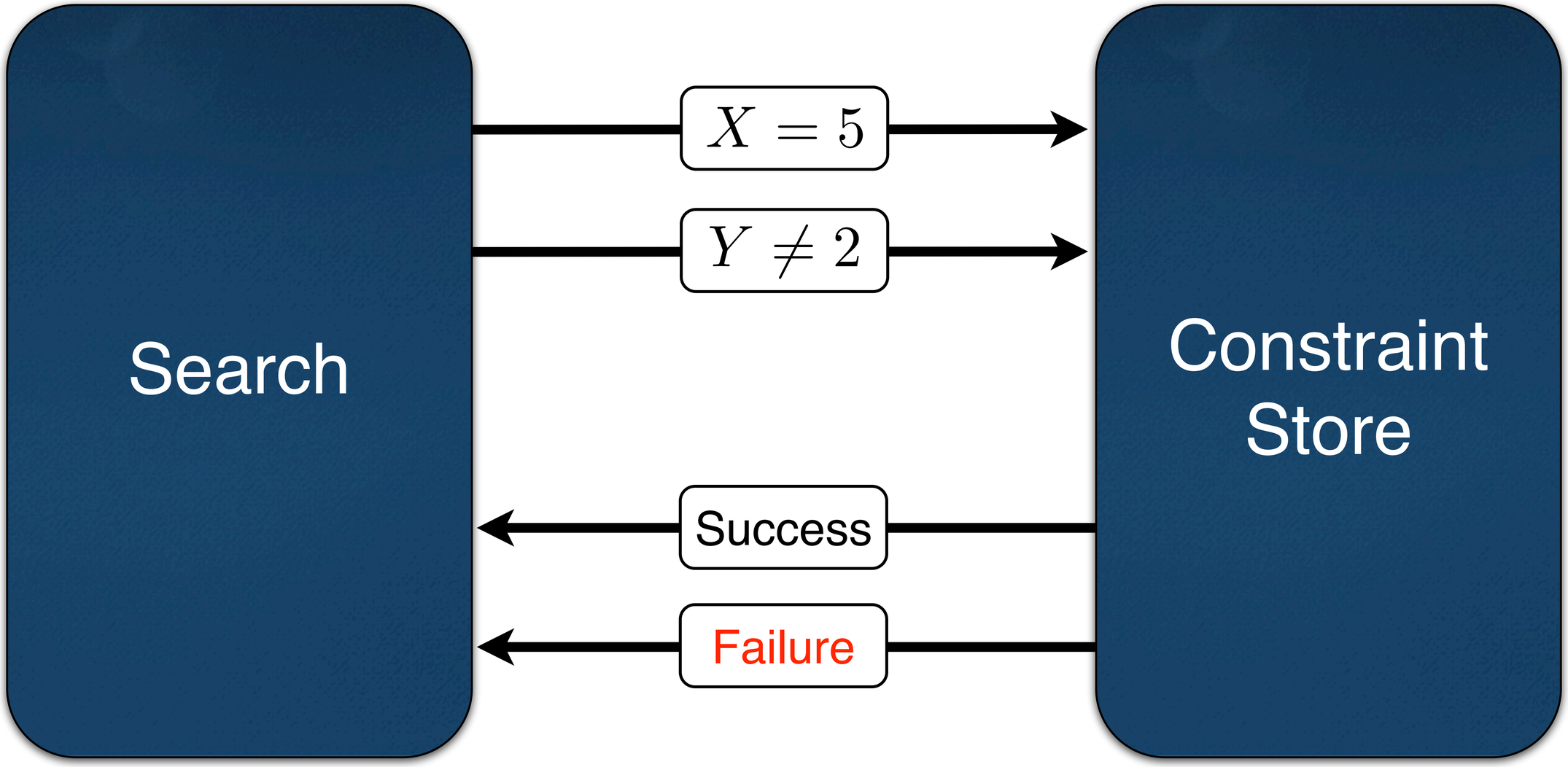
Computational Paradigm



Computational Paradigm



Computational Paradigm



Nondeterministic choice

- When a constraint fails
  - that is, when adding a constraint to the constraint store returns a failure
- the solver goes back to the last tryall
  - and assigns a value that has not been tried before
  - if no such value is left, the system backtracks to an earlier nondeterministic instruction

Understanding nondeterministic computations

```
range R = 1..8;
var{int} row[R] in R;
solve {
   forall(i in R,j in R: i < j) {
      row[i] ≠  row[j];
      row[i] ≠  row[j] + (j - i);
      row[i] ≠  row[j] - (j - i);
   }
}
using {
   forall(r in R)
      tryall(v in R)
         row[r] = v;
}
```

Understanding nondeterministic computations

```
range R = 1..8;
var{int} row[R] in R;
solve {
   forall(i in R,j in R: i < j) {
      row[i] ≠  row[j];
      row[i] ≠  row[j] + (j - i);
      row[i] ≠  row[j] - (j - i);
   }
}
using {
   forall(r in R)
      tryall(v in R)
         row[r] = v;
}
```

iterate over all queens

Understanding nondeterministic computations

```
range R = 1..8;
var{int} row[R] in R;
solve {
   forall(i in R,j in R: i < j) {
      row[i] ≠  row[j];
      row[i] ≠  row[j] + (j - i);
      row[i] ≠  row[j] - (j - i);
   }
}
using {
   tryall(v in R) row[1] = v;
   tryall(v in R) row[2] = v;
   tryall(v in R) row[3] = v;
   tryall(v in R) row[4] = v;
   tryall(v in R) row[5] = v;
   tryall(v in R) row[6] = v;
   tryall(v in R) row[7] = v;
   tryall(v in R) row[8] = v;
}
```

Understanding nondeterministic computations

```
range R = 1..8;
var{int} row[R] in R;
solve {
   forall(i in R,j in R: i < j) {
      row[i] ≠  row[j];
      row[i] ≠  row[j] + (j - i);
      row[i] ≠  row[j] - (j - i);
   }
}
using {
   forall(r in R)
     tryall(v in R)
         row[r] = v;
}
```

Understanding nondeterministic computations

```
range R = 1..8;
var{int} row[R] in R;
solve {
   forall(i in R,j in R: i < j) {
      row[i] ≠  row[j];
      row[i] ≠  row[j] + (j - i);
      row[i] ≠  row[j] - (j - i);
   }
}
using {
   forall(r in R)
     tryall(v in R)
         row[r] = v;
}
```

nondeterministically explore all values

Understanding nondeterministic computations

```
range R = 1..8;
var{int} row[R] in R;
solve {
   forall(i in R,j in R: i < j) {
      row[i] ≠  row[j];
      row[i] ≠  row[j] + (j - i);
      row[i] ≠  row[j] - (j - i);
   }
}
using {
   forall(r in R)
     try row[r] = 1;
     |   row[r] = 2;
     |   row[r] = 3;
     |   row[r] = 4;
     |   row[r] = 5;
     |   row[r] = 6;
     |   row[r] = 7;
     |   row[r] = 8;
     endtry;
}
```

Understanding nondeterministic computations

```
range R = 1..8;
var{int} row[R] in R;
solve {
   forall(i in R,j in R: i < j) {
      row[i] ≠  row[j];
      row[i] ≠  row[j] + (j - i);
      row[i] ≠  row[j] - (j - i);
   }
}
using {
   forall(r in R)
     try row[r] = 1;
     |   row[r] = 2;
     |   row[r] = 3;
     |   row[r] = 4;
     |   row[r] = 5;
     |   row[r] = 6;
     |   row[r] = 7;
     |   row[r] = 8;
     endtry;
}
```

Searching in constraint programming

- variable/value labeling
- value/variable labeling
- domain splitting
- focusing on the objective
- symmetry breaking during search
- randomization and restarts

Variable/Value Labeling

- Two steps
  - choose the variable to assign next
  - choose the value to assign

Variable/Value Labeling

- Two steps
  - choose the variable to assign next
  - choose the value to assign
- First-fail principle
  - choose the variable with the smallest domain

Variable/Value Labeling

- Two steps
  - choose the variable to assign next
  - choose the value to assign
- First-fail principle
  - choose the variable with the smallest domain
- The variable ordering is dynamic
  - reconsider the selection after each choice

Variable/Value Labeling

```
range R = 1..8;
var{int} row[R] in R;
solve {
   forall(i in R,j in R: i < j) {
      row[i] ≠  row[j];
      row[i] ≠  row[j] + (j - i);
      row[i] ≠  row[j] - (j - i);
   }
}
using {
   forall(r in R) by row[r].getSize()
      tryall(v in R)
         row[r] = v;
}
```

Variable/Value Labeling

```
range R = 1..8;
var{int} row[R] in R;
solve {
   forall(i in R,j in R: i < j) {
      row[i] ≠  row[j];
      row[i] ≠  row[j] + (j - i);
      row[i] ≠  row[j] - (j - i);
   }
}
using {
   forall(r in R) by row[r].getSize()
      tryall(v in R)
         row[r] = v;
}
```

select first the variable with the smallest domain

Variable/Value Labeling

- Two steps
  - choose the variable to assign next
  - choose the value to assign
- First-fail principle
  - choose the variable with the smallest domain
  - choose the most constrained variable

Variable/Value Labeling

- Two steps
  - choose the variable to assign next
  - choose the value to assign
- First-fail principle
  - choose the variable with the smallest domain
  - choose the most constrained variable
- Use a lexicographic criterion
  - first the domain size
  - next the proximity to the middle of the board

Lexicographic ordering

```
range R = 1..8;
var{int} row[R] in R;
solve {
   forall(i in R,j in R: i < j) {
      row[i] ≠  row[j];
      row[i] ≠  row[j] + (j - i);
      row[i] ≠  row[j] - (j - i);
   }
}
using {
   forall(r in R)
     by (row[r].getSize(),abs(r-n/2))
      tryall(v in R)
         row[r] = v;
}
```

Dynamic orderings for variable and value choices

```
range R = 1..8;
range C = 1..8;
var{int} row[C] in R;
var{int} col[R] in C
solve {
   forall(i in R,j in R: i < j) {
      row[i] ≠  row[j];
      row[i] ≠  row[j] + (j - i);
      row[i] ≠  row[j] - (j - i);
   }
   forall(i in C,j in C: i < j) {
      col[i] ≠  col[j];
      col[i] ≠  col[j] + (j - i);
      col[i] ≠  col[j] - (j - i);
   }
   forall(r in R,c in C)
      (row[c] = r) <=> (col[r] = c);
}
using {
   forall(r in R) by row[r].getSize()
     tryall(v in R) by col[v].getSize()
        row[r] = v;
}
```

Variable/Value Labeling

- Variable ordering
  - choose the most constrained variable
  - e.g., smallest domain, variable that fails often

Variable/Value Labeling

- Variable ordering
  - choose the most constrained variable
  - e.g., smallest domain, variable that fails often
- Value ordering
  - often choose a value that leaves as many options as possible to the other variables
  - this helps finding a solution

Feasibility versus Optimality

- strong focus on feasibility branching
  - the pruning algorithm provides a lot of information

Feasibility versus Optimality

- strong focus on feasibility branching
  - the pruning algorithm provides a lot of information
- may also focus on the objective function
  - does not change the way search works

The ESDD Deployment Problem

- Generalized quadratic assignment problem
  - f: the communication frequency matrix
  - h: the distance matrix (in hops)
  - x: the assignment vector (decision variables)
  - C: sets of components
  - Sep: separation constraints
  - Col: colocation constraints
  - objective function

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

The CP Model

```
minimize
    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)
     x[c1] = x[c2];
   forall(S in Sep)
     alldifferent(all(c in S) x[c]);
}
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 x[j].memberOf(l)) h[n,l])
         x[i] = n;
}
```

The CP Model

```
minimize
    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)
     x[c1] = x[c2];
   forall(S in Sep)
     alldifferent(all(c in S) x[c]);
}
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 x[j].memberOf(l)) h[n,l])
         x[i] = n;
}
```

select a component i to assign that has the largest communication frequency

The CP Model

```
minimize
    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)
     x[c1] = x[c2];
   forall(S in Sep)
     alldifferent(all(c in S) x[c]);
}
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 x[j].memberOf(l)) h[n,l])
         x[i] = n;
}
```

try the possible value starting first with those minimizing the number of hops to j

Discrete Optimization

Constraint Programming: Part X

Goals of the lecture

- Search in constraint programming
  - introduction
  - active research area

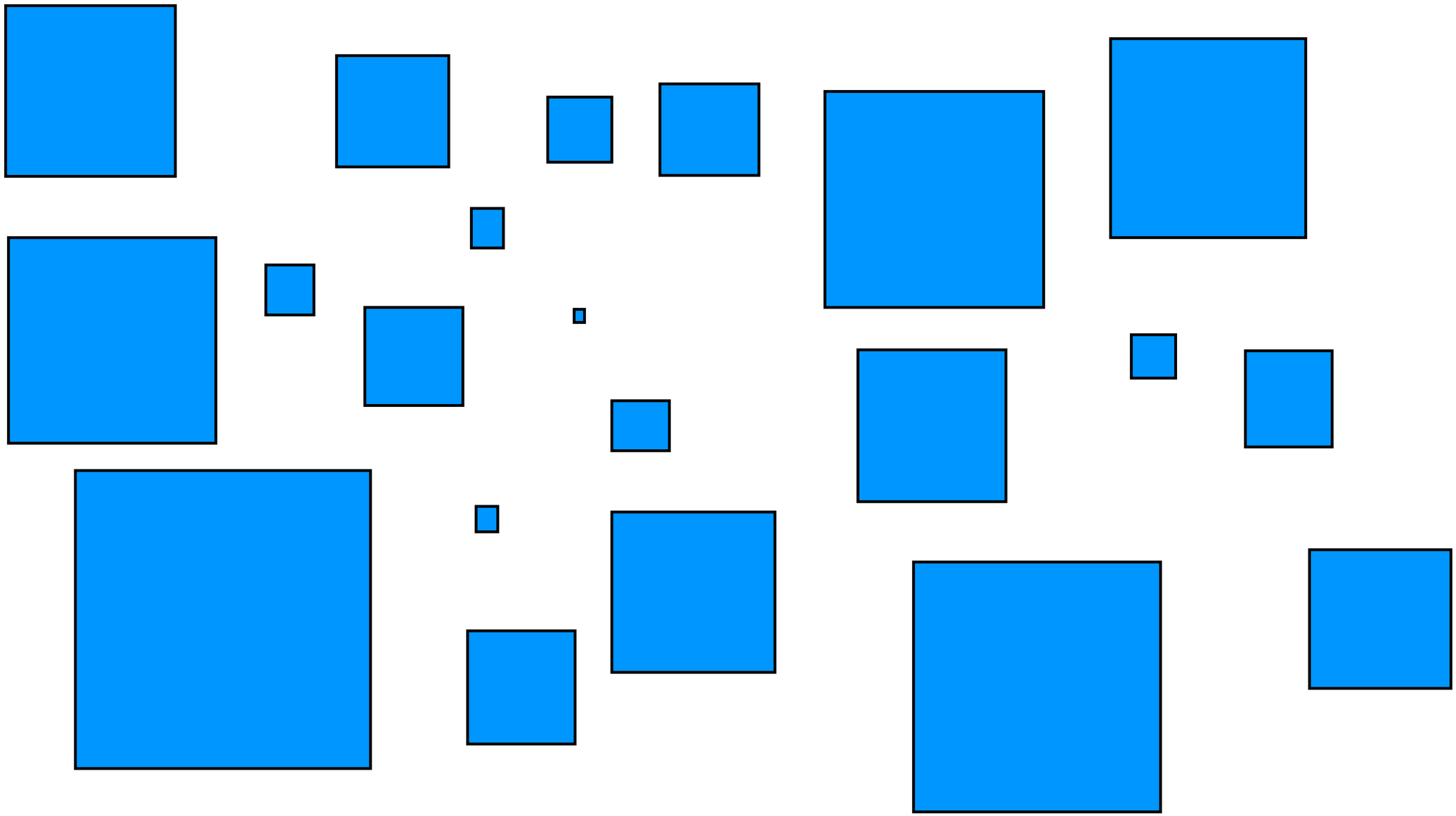
Searching in constraint programming

- variable/value labeling
- focusing on the objective
- value/variable labeling
- domain splitting
- symmetry breaking during search
- randomization and restarts

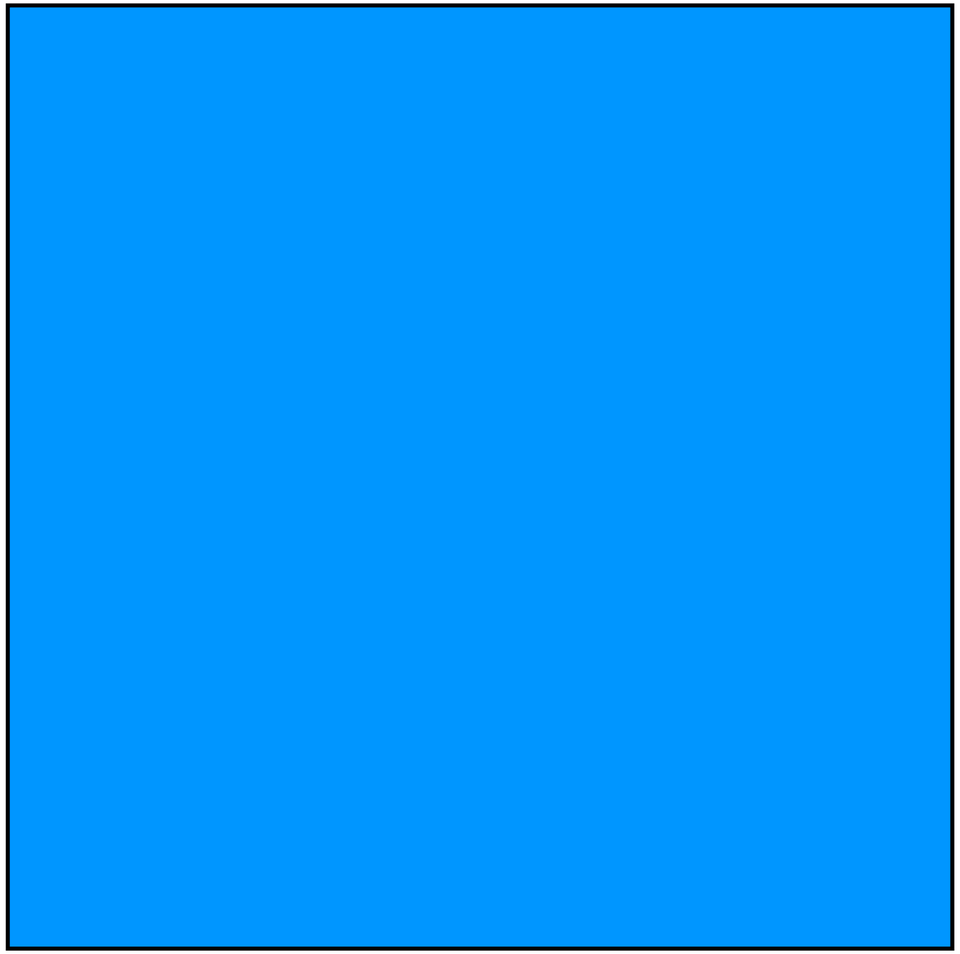
Value/variable labeling

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

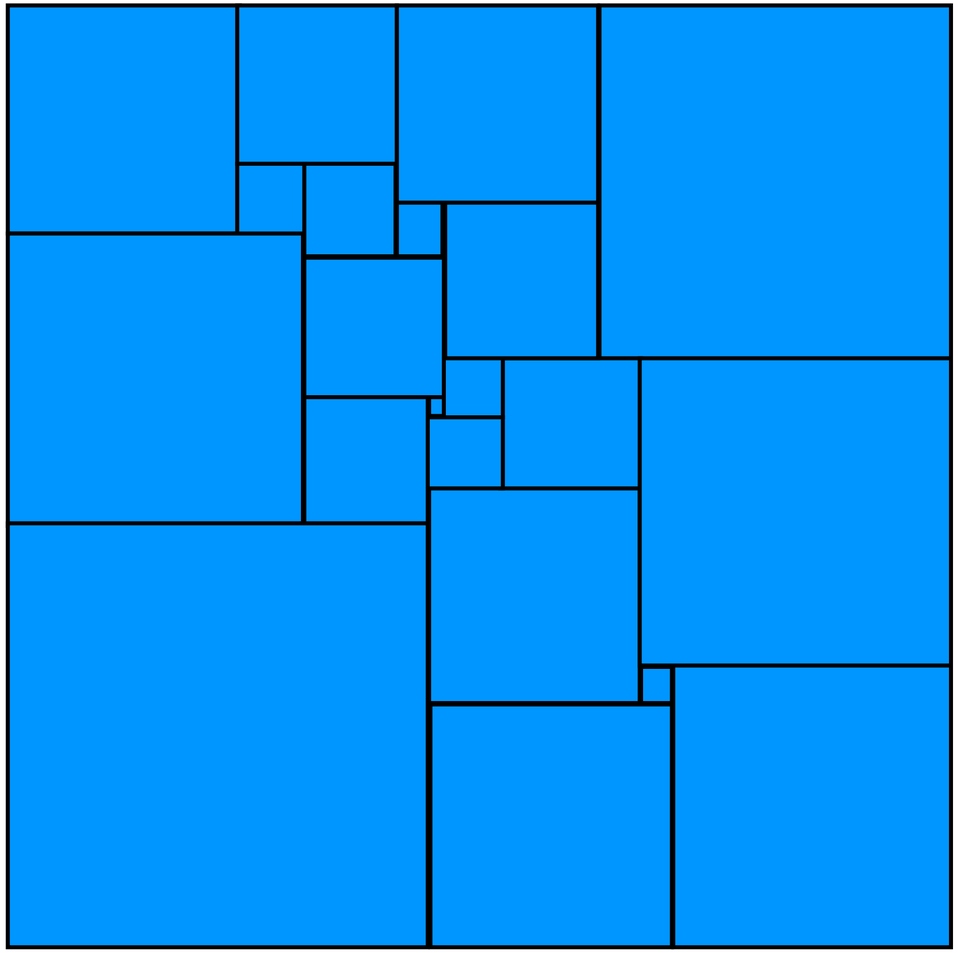
The perfect square problem



The perfect square problem



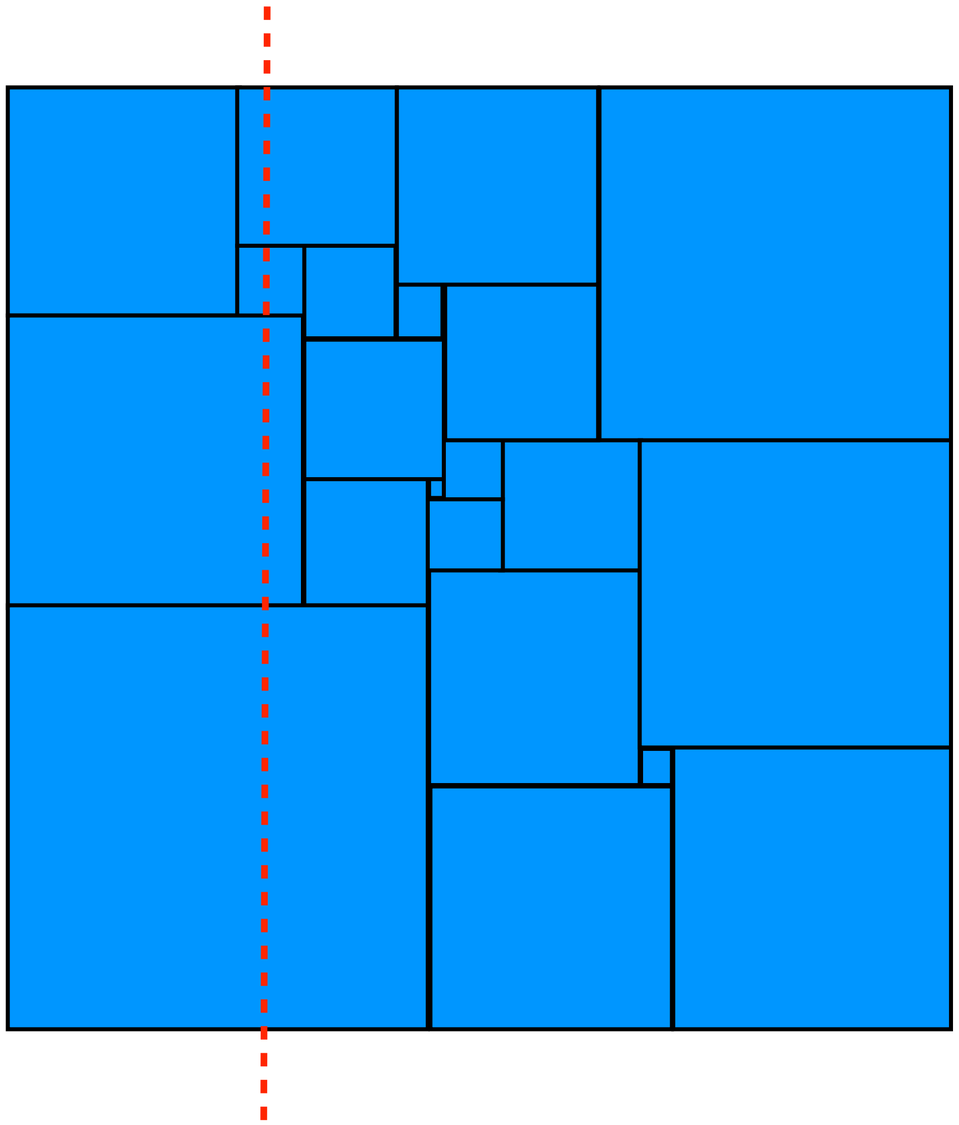
The perfect square problem



The perfect square problem

- What are the decision variables?
  - x and y-coordinates of the bottom-left corner of every square
- What are the constraints?
  - the squares fit in the larger square
  - the squares do not overlap
- There are also redundant constraints
  - here is the intuition

The perfect square problem



The perfect square problem

```
range R = 1..8;
int s = 122; 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{int} x[Square] in Side;
var{int} y[Square] in Side;

solveall {
  forall(i in Square) {
 x[i]<=s-side[i]+1;
   y[i]<=s-side[i]+1;
 }
  forall(i in Square,j in Square: i<j)
    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) {
     sum(i in Square) side[i]*((x[i]<=p) && (x[i]>=p-side[i]+1)) = s;
     sum(i in Square) side[i]*((y[i]<=p) && (y[i]>=p-side[i]+1)) = s;
  }
}
```

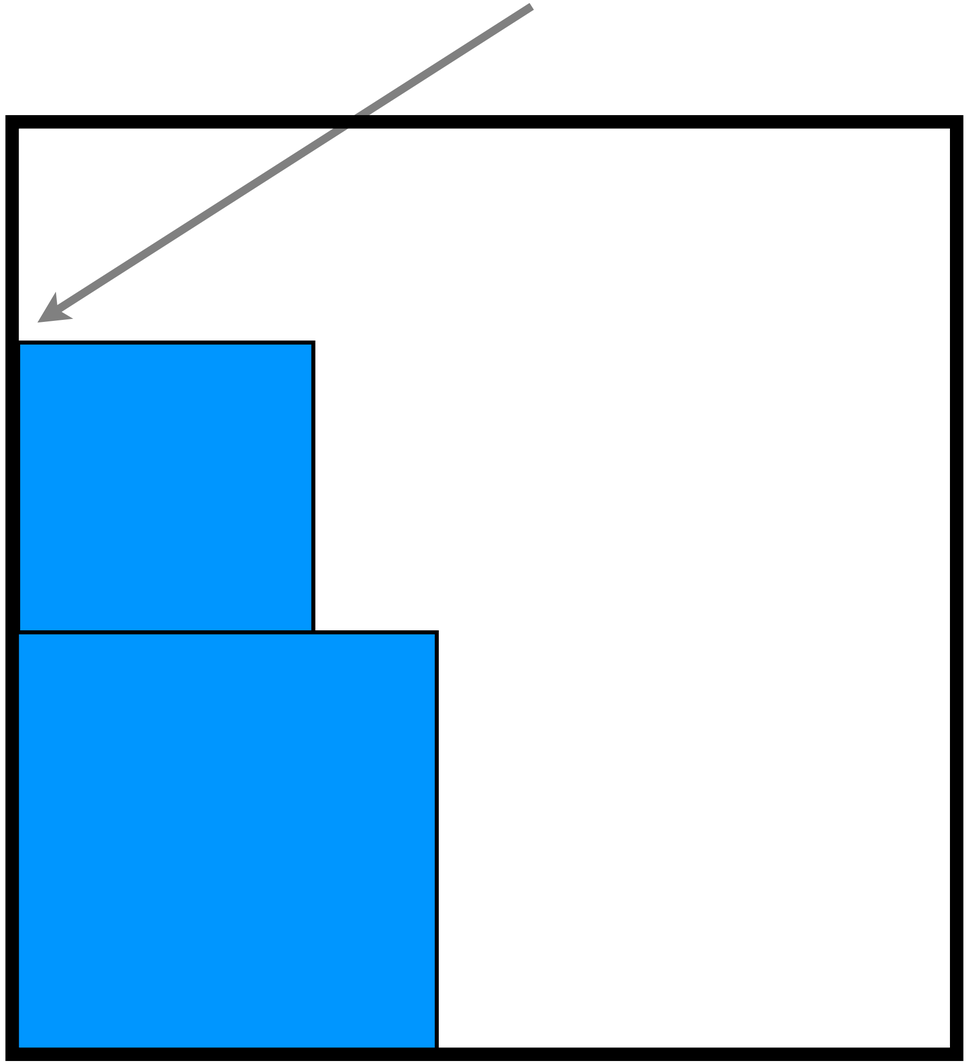
non-overlapping constraints

redundant constraints

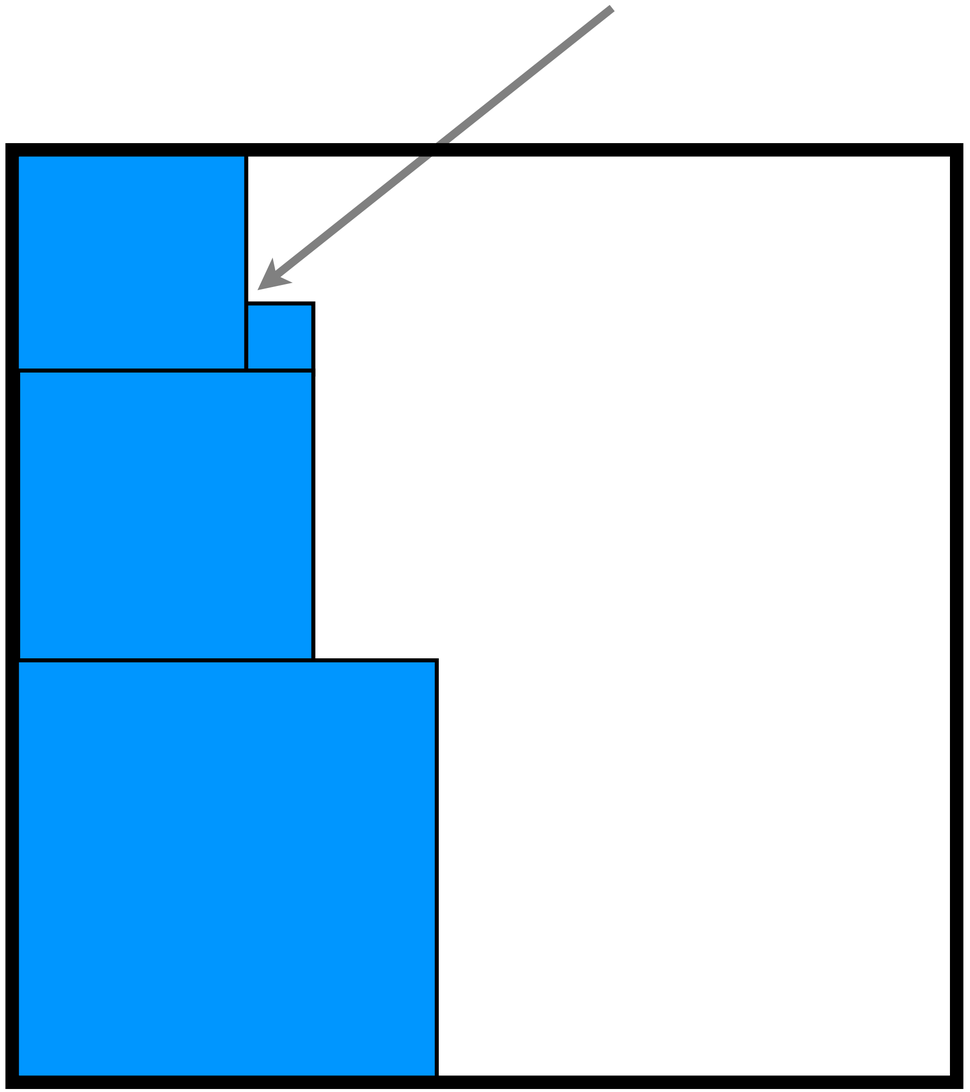
The value/variable labeling

- Why a value/variable labeling
  - we know that there is no empty space in the square to be filled.

The perfect square problem



The perfect square problem



The value/variable labeling

- What is the labeling doing?
  - choose a x-coordinate p
  - for all square i, decide whether to place i at coordinate p
    - that is, whether the bottom-left corner of i has x-coordinate p
  - repeat for all x-coordinates
  - repeat for all y-coordinates

The value/variable labeling

- What is the labeling doing?
  - choose a x-coordinate p
  - for all square i, decide whether to place i at coordinate p
    - that is, whether the bottom-left corner of i has x-coordinate p
  - repeat for all x-coordinates
  - repeat for all y-coordinates

The perfect square problem

```
using {

  forall(p in Side)
    forall(i in Square)
      try
         x[i] = p;
      |
       x[i] != p;

  forall(p in Side)
    forall(i in Square)
      try
         y[i] = p;
      |
         y[i] != p;

}
```

choose a x-coordinate p

consider a square i

decide whether to place i at position p

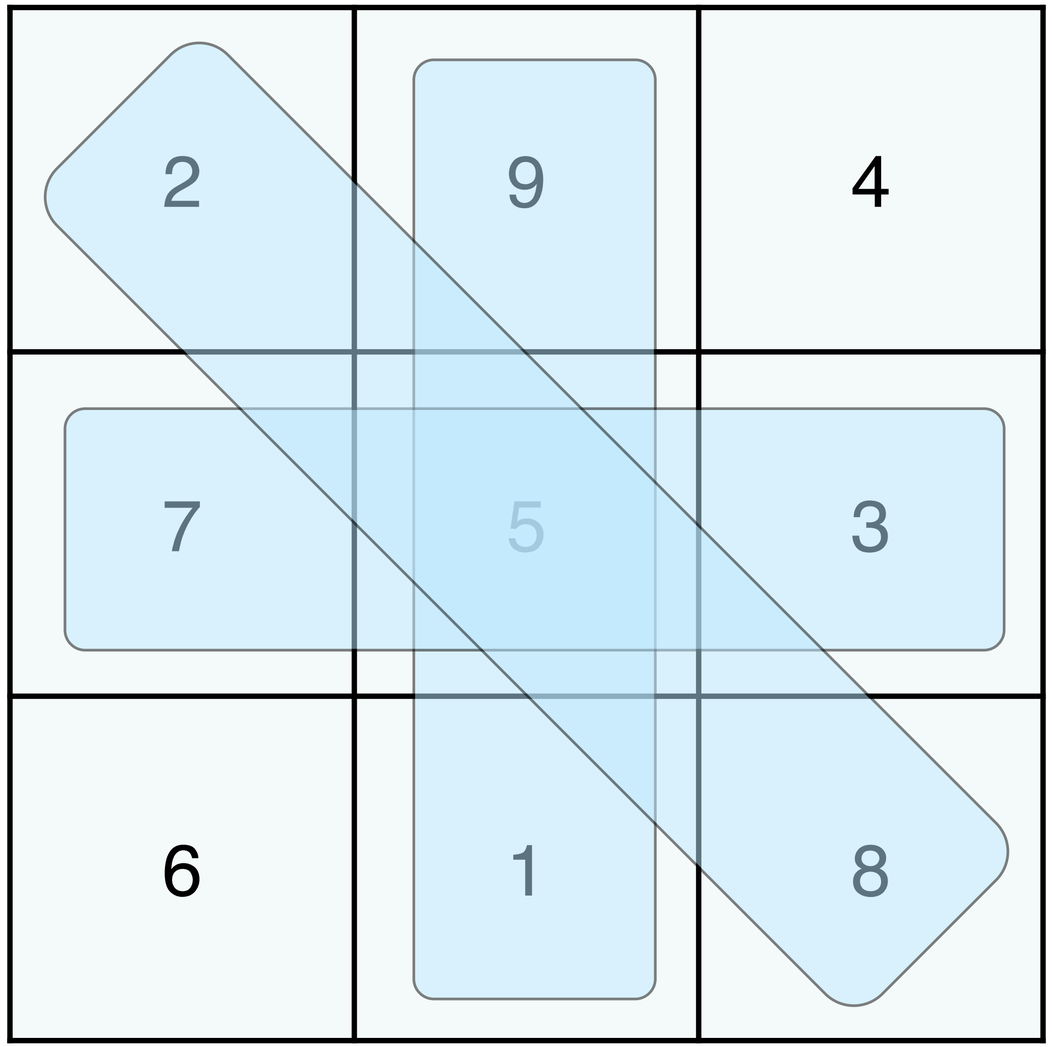
Searching in constraint programming

- variable/value labeling
- focusing on the objective
- value/variable labeling
- domain splitting
- symmetry breaking during search
- randomization and restarts

The magic square problem

- Place numbers in a square
  - all numbers are different
  - all rows, columns, and diagonals sum to the same number

The magic square problem



The magic square problem

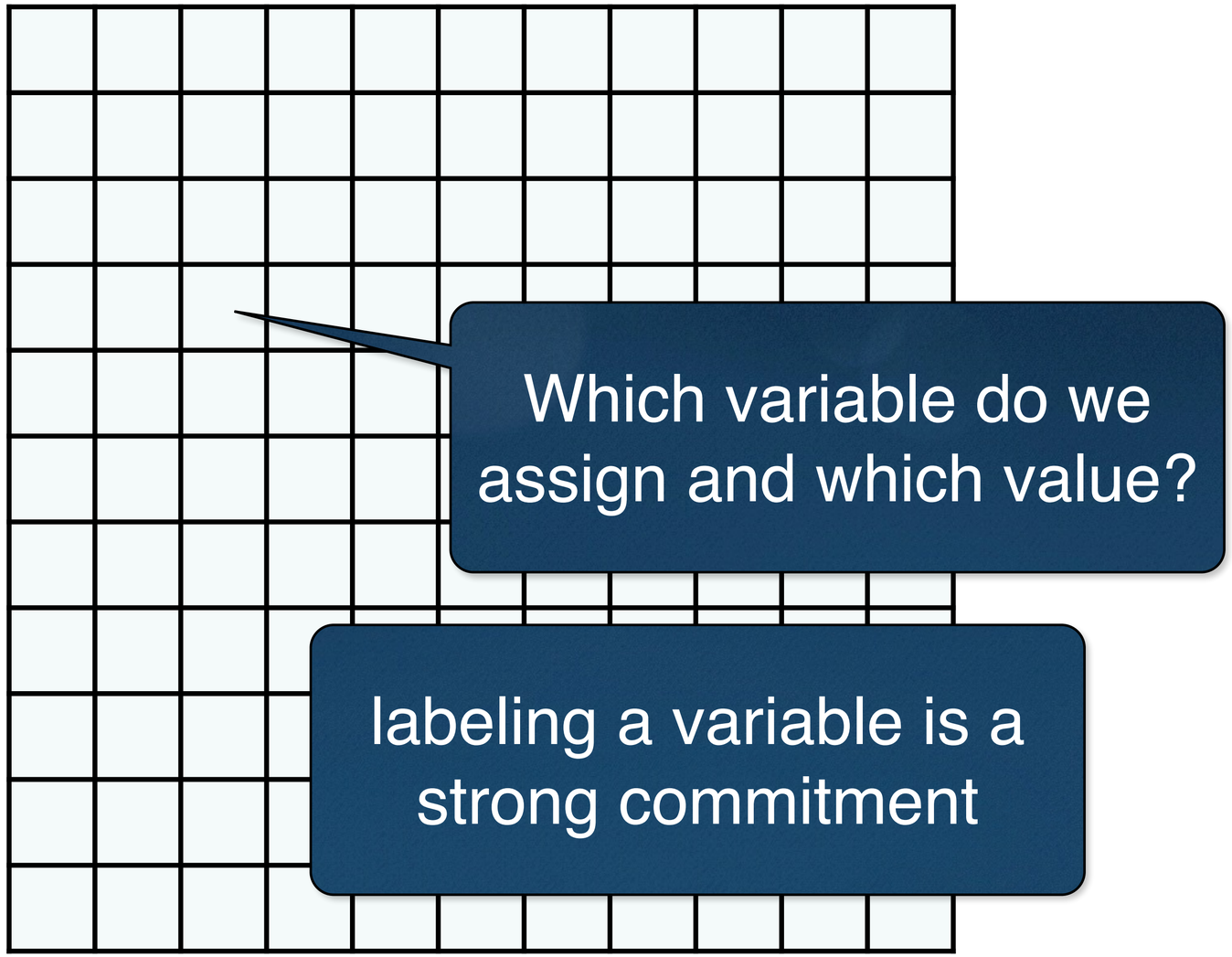
The magic square problem

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

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

The magic square problem



Domain splitting

- Select a variable
- Split its domain in two or more sets
  - much weaker commitment

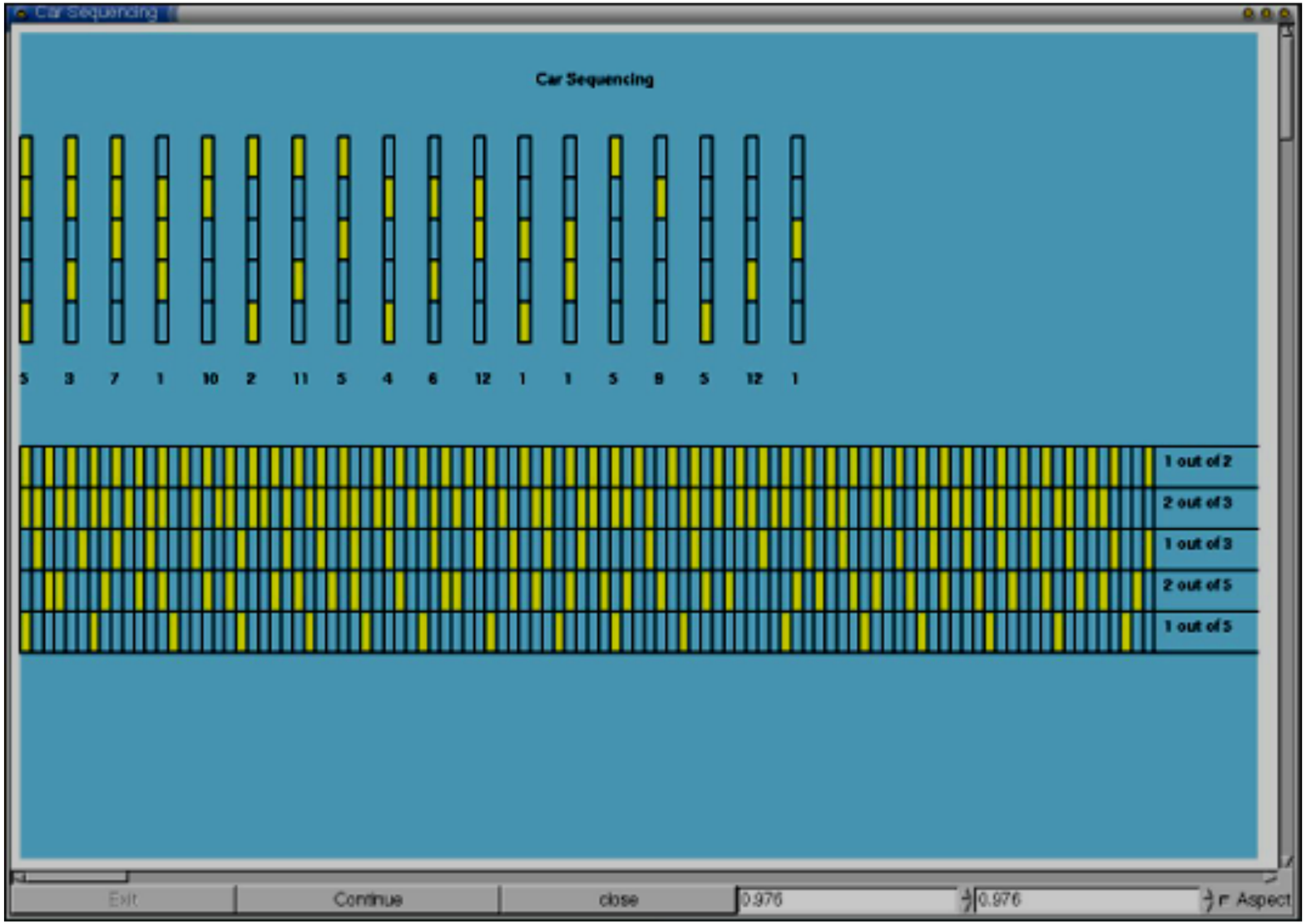
The magic square problem

```
using {
  var{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
        x[i] <= mid;
      |
        x[i] > mid;
    }
  }
}
```

Car sequencing

- Cars on an assembly line
- Cars require specific options
  - leather seats, moonroof
- Capacity constraints on the production units
  - at most 2 out of 5 successive cars can require a moonroof
- Sequence all the cars such that the capacity constraints are satisfied

Car sequencing



Domain splitting in car sequencing

- Motivation
  - focus on difficult options
  - decide which slots take these options
- Domain splitting
  - do not assign configurations to slots
  - decide whether the slot take the option

Domain splitting in car sequencing

```
using {
  forall(o in Options) by (slack[o])
    forall(i in Slots)
      try
        line[i] in options[o];
      |
        line[i] notin options[o]);
}
```

start with the option that has the least slack

Force the slot to take the option

Force the slot not to take the option

Searching in constraint programming

- variable/value labeling
- focusing on the objective
- value/variable labeling
- domain splitting
- symmetry breaking during search
- randomization and restarts

Scene Allocation

- Shooting scenes for a movie
  - an actor plays in some of the scenes
  - at most k scenes can be shot per day
  - each actor is paid by the day
- Objective
  - minimize the total cost
- Symmetries
  - what kind of symmetries do we have here?

Scene allocation

- How can we eliminate these symmetries?
  - consider the first scene s_1. What are the days that we consider for this scene?

    Only one day, say day 1
  - consider the second scene s_2. Which days must be considered for s_2?

    Only two days, day 1 and day 2
  - In general the already used days and one additional new day

existing days

new day

$$D(s_k) = \{1, 2, \ldots, \max(s_1, ..., s_{k-1}) + 1\}$$

Scene allocation

```
range Scenes = 1..n;
range Days  = 1..m;
range Actor = …;
int fee[Actor] = …;
set{Actor} appears[Scenes] = …;
set{int} which[a in Actor] = setof(i in Scenes) member(a,appears[i]);
var{int} shoot[Scenes] in Days;

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

Symmetry-breaking constraints

- Side effect
  - interferences with the search heuristic
- Can we avoid this?
  - symmetry-breaking during search
  - dynamically impose the symmetry-breaking constraints
- How?
  - same constraints
  - the order is different and discovered dynamically

Symmetry-breaking during search

- Choose a scene to shoot
  - use a good heuristic
    - first-fail
    - expensive scene
- Consider existing days + 1 new day
  - to label the scene
- Advantages
  - break symmetries
  - does not interfere with the search heuristic

Symmetry breaking during search

dynamic ordering

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

existing days

new day

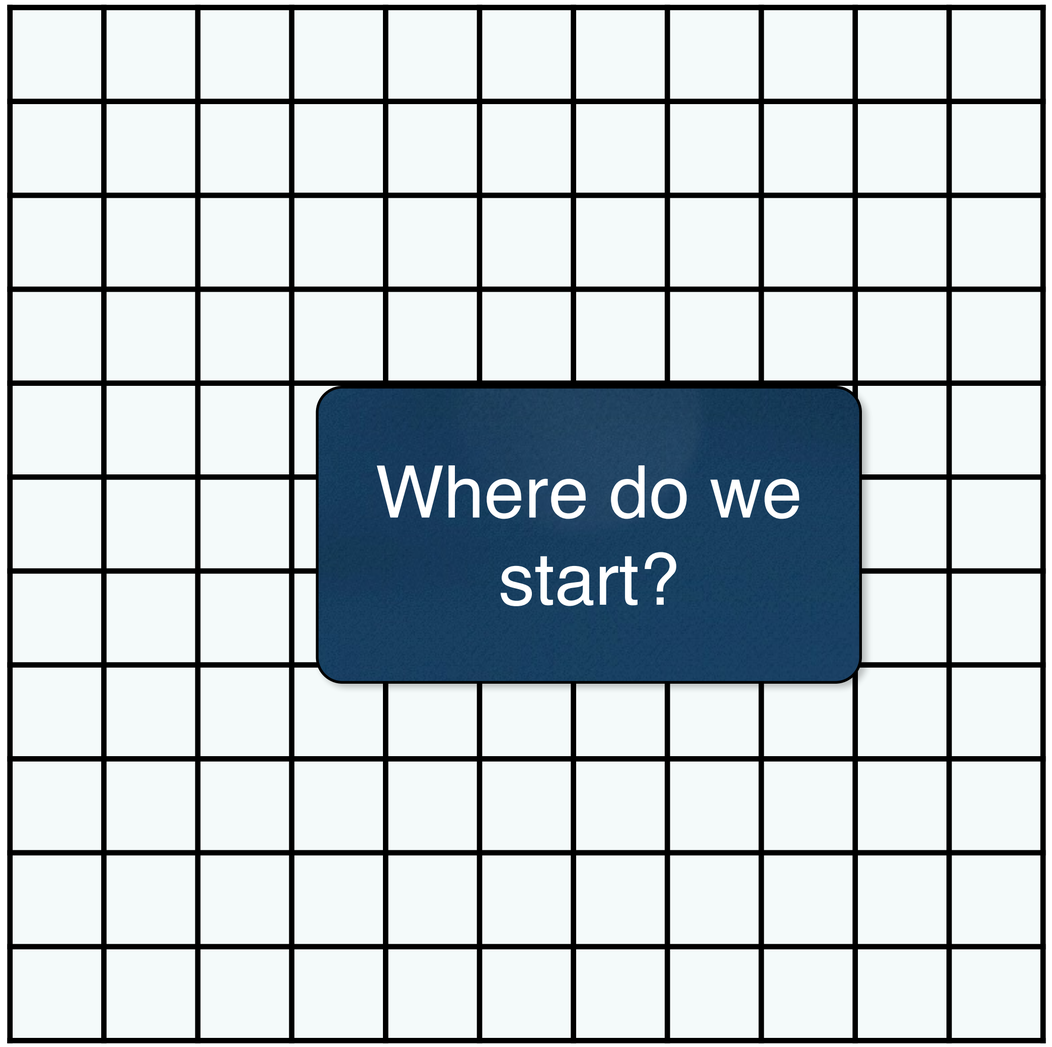
Searching in constraint programming

- variable/value labeling
- focusing on the objective
- value/variable labeling
- domain splitting
- symmetry breaking during search
- randomization and restarts

Randomization and restarts

- Sometimes there is no (obvious) search ordering
  - but there exists some good ones
- How to find them?
  - brute force
  - randomization and restarts
- Key idea
  - try a random ordering
  - if no solution is found after some limit, restart the search

The magic square problem



Randomization and restarts

- Key idea
  - apply a heuristic but with randomization
    - e.g., one of the three best variables
  - limit the time in the search
  - if the limit is reached, restart and possibly increase the limit

The magic square problem

```
using {
 timeLimit = 10;
  repeat {
  limitTime(timeLimit) {
    var{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 x[i] <= mid; | x[i] > mid;
      }
    }
  }
  onFailure {
    timeLimit = 1.1 * timeLimit;
  }
}
```

Citations

Wolfsburg - Volkswagen Assembly Line (http://www.flickr.com/photos/24736216@N07/2994043188/) by Roger Wollstadt (http://www.flickr.com/photos/24736216@N07/) CC BY-SA 2.0 (http://creativecommons.org/licenses/by-sa/2.0/deed.en)

Geely assembly line in Beilun, Ningbo (http://commons.wikimedia.org/wiki/File:Geely_assembly_line_in_Beilun,_Ningbo.JPG) by Siyuwj (Own work) [CC-BY-SA-3.0 (http://creativecommons.org/licenses/by-sa/3.0)], via Wikimedia Commons

66ème Festival de Venise (Mostra)(http://www.flickr.com/photos/nicogenin/3902889281/) by Nicolas Genin (http://www.flickr.com/photos/nicogenin/) CC BY-SA 2.0 (http://creativecommons.org/licenses/by-sa/2.0/deed.en)

Julia Roberts 2011 Shankbone (http://www.flickr.com/photos/shankbone/5656729480/) by David Shankbone (http://www.flickr.com/people/shankbone/) CC BY 2.0 (http://creativecommons.org/licenses/by/2.0/deed.en)

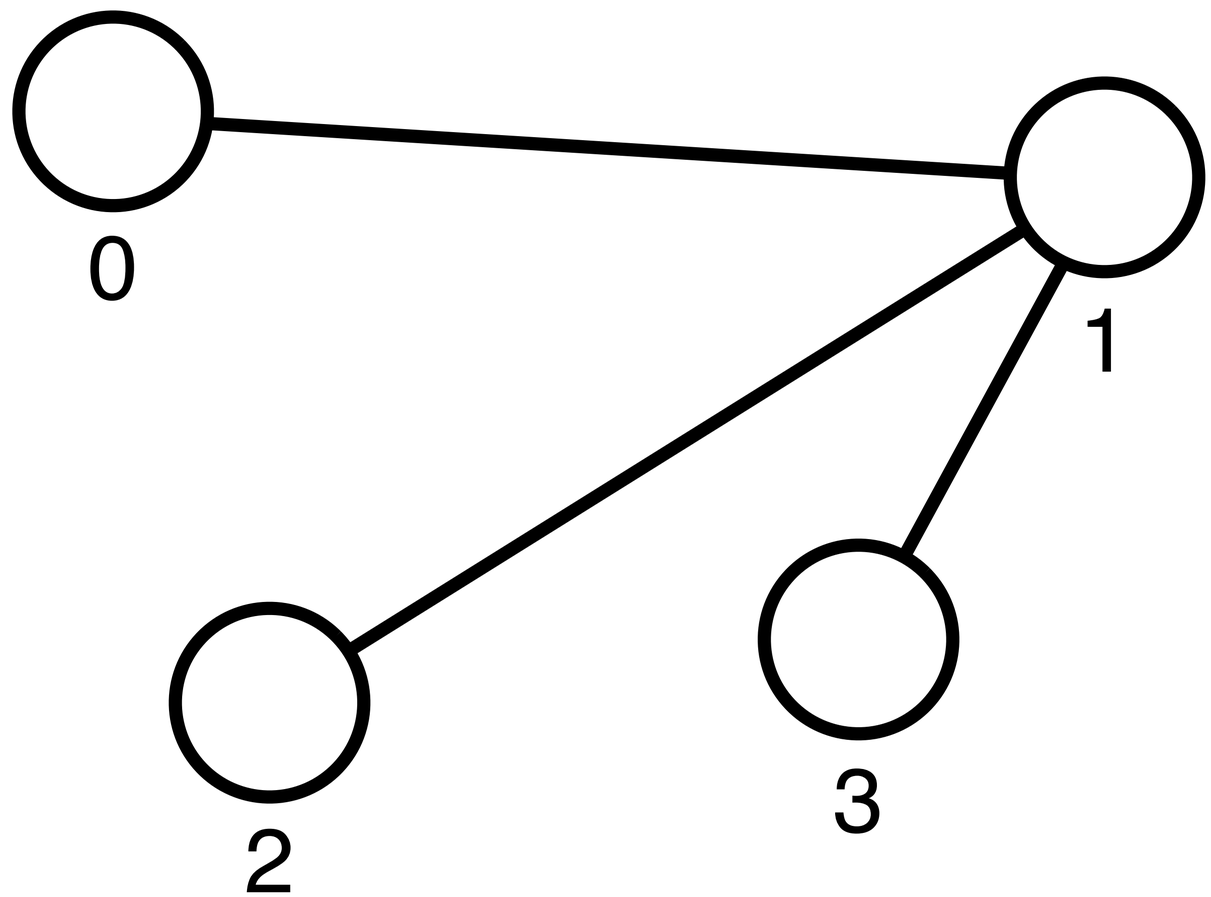
Pride and Prejudice_004 (http://www.flickr.com/photos/tonyshek/7271339306/) by Tony Shek (http://www.flickr.com/photos/tonyshek/) CC BY-SA 2.0 (http://creativecommons.org/licenses/by-sa/2.0/deed.en)

Clive Owen, (http://www.flickr.com/photos/oneras/251106343/) by Mario Antonio Pena Zapateira (http://www.flickr.com/photos/oneras/) CC BY-SA 2.0 (http://creativecommons.org/licenses/by-sa/2.0/deed.en)

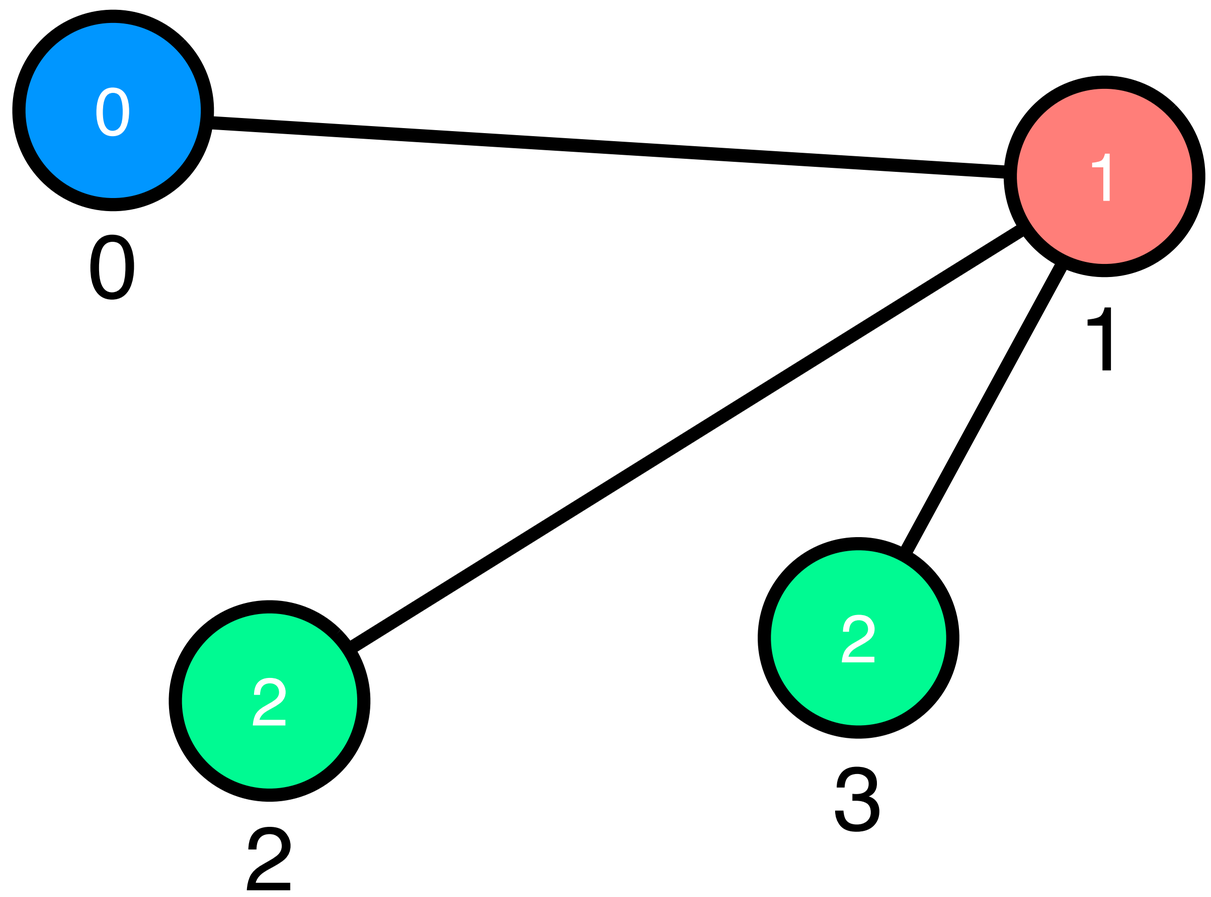
Discrete Optimization

Assignments: Graph Coloring

Graph Coloring



Graph Coloring



Graph Coloring

- n Nodes
- e Edges
- Edge list
- C_i color of node i

$$\text{minimize:} \quad \max_{i \in 0 \ldots n-1} c_i$$

$$\text{subject to:}$$

$$c_i \neq c_j \quad (\langle i, j \rangle \in E)$$

Graph Coloring

$$\begin{aligned} &\text{minimize:} && \max_{i \in 0 \ldots n-1} c_i \\ &\text{subject to:} && \\ & && c_i \neq c_j \quad (\langle i, j \rangle \in E) \end{aligned}$$

Input

```
|N| |E|
u_1 v_1
u_2 v_2
...
u_|E| v_|E|
```

Output

```
obj opt
c_1 c_2 c_3 ... c_n
```

Graph Coloring

$$\begin{aligned} &\text{minimize:} && \max_{i \in 0 \ldots n-1} c_i \\ &\text{subject to:} && \\ & && c_i \neq c_j \quad (\langle i, j \rangle \in E) \end{aligned}$$

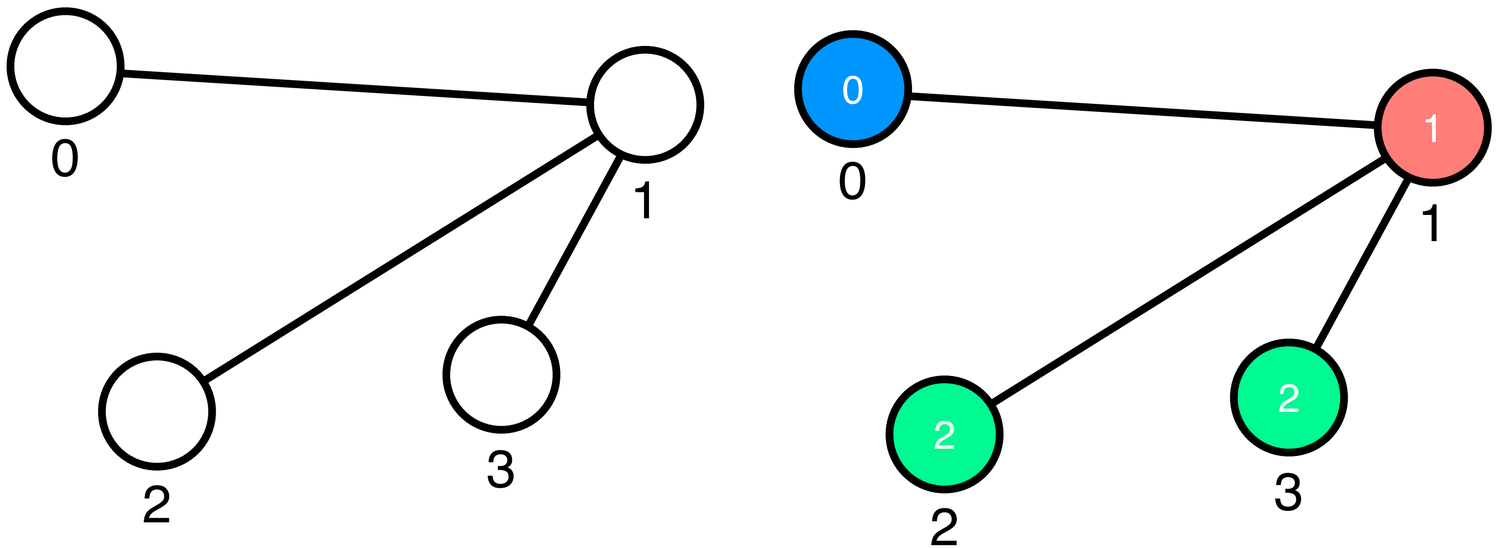
Input

```
4 3
0 1
1 2
1 3
```

Output

```
3 0
0 1 2 2
```

Graph Coloring



Input

```
4 3
0 1
1 2
1 3
```

Output

```
3 0
0 1 2 2
```

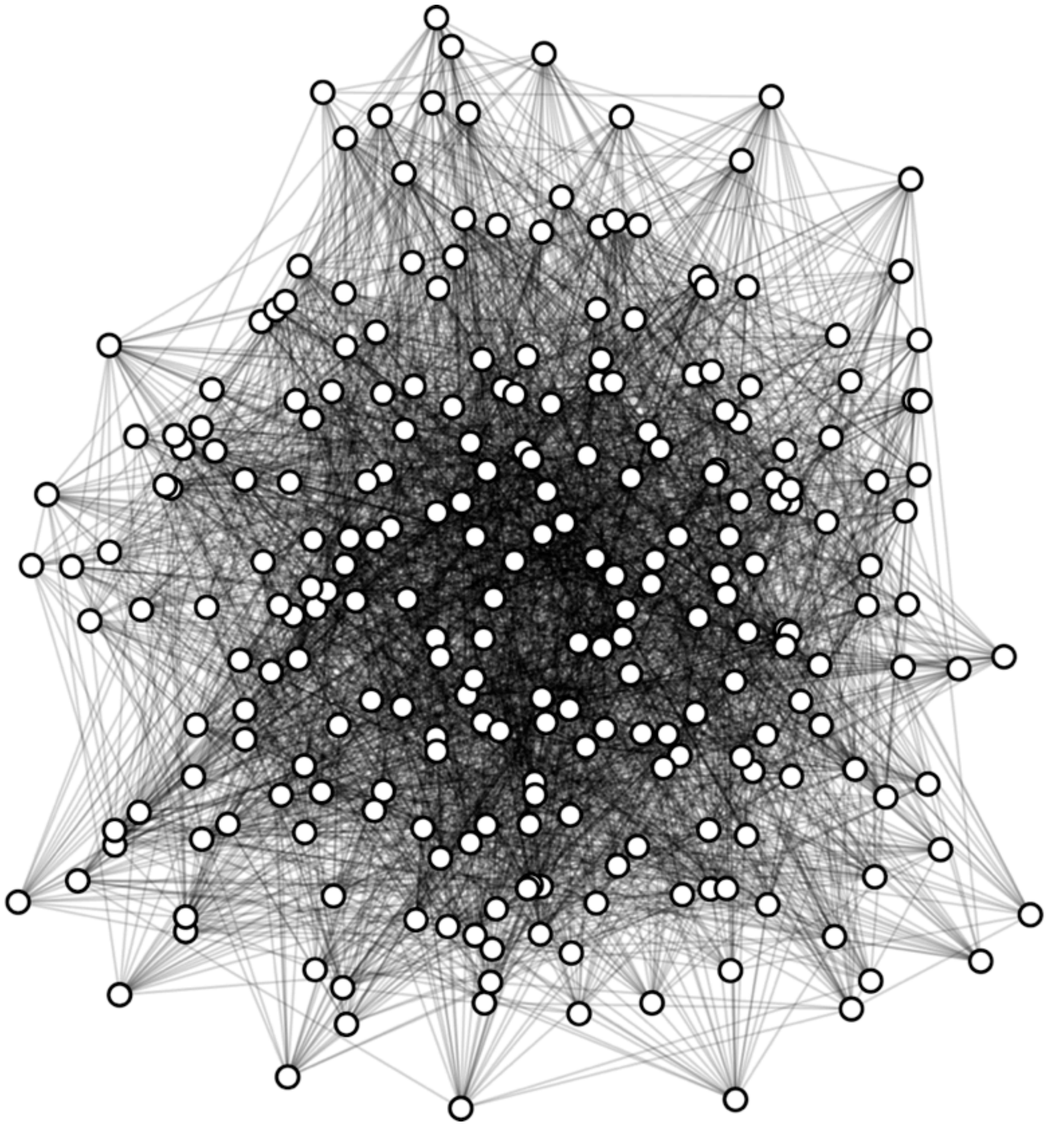
Assignment Tips

- Node Degree

- Symmetries
  - 0 1 2 2
  - 2 0 1 1

- Easy Lower Bound

Have Fun!



Discrete Optimization

Assignments: Optimization Tools

Optimization Tools

- General purpose optimization tools!
  - Let us not reinvent the wheel

Optimization Tools

- General purpose optimization tools!
  - Let us not reinvent the wheel

- Pros:
  - Rapid prototyping of ideas
  - Include decades of research / fancy algorithms

Optimization Tools

- General purpose optimization tools!
  - Let us not reinvent the wheel

- Pros:
  - Rapid prototyping of ideas
  - Include decades of research / fancy algorithms

- Cons:
  - Learning curve, platform/language limitations
  - Often slower than dedicated algorithms
  - May appear to do strange things
    - if you do not know how they work.

Optimization Tools

- Types of Tools
  - CP Solvers
  - MIP Solvers
  - LS Solvers

Optimization Tools

- Types of Tools
  - CP Solvers
  - MIP Solvers
  - LS Solvers

- Availability
  - open source
  - free for non-commercial use
  - free for students
    - May need an *edu* e-mail
  - see the partial list we provide

N-Queens Example

```
range R = 1..8;
var{int} row[R] in R;
solve {
   forall(i in R,j in R: i < j) {
      row[i] ≠  row[j];
      row[i] ≠  row[j] + (j - i);
      row[i] ≠  row[j] - (j - i);
   }
}
```

N-Queens Example

```
range R = 1..8;
var{int} row[R] in R;
solve {
   forall(i in R,j in R: i < j) {
      row[i] ≠  row[j];
      row[i] ≠  row[j] + (j - i);
      row[i] ≠  row[j] - (j - i);
   }
}
```

looks like **alldifferent**

N-Queens Example

```
range R = 1..8;
var{int} row[R] in R;
solve {
   forall(i in R,j in R: i < j) {
      row[i] ≠  row[j];
      row[i] ≠  row[j] + (j - i);
      row[i] ≠  row[j] - (j - i);
   }
}
```

looks like **alldifferent**

would this be a better model?

```
range R = 1..8;
var{int} row[R] in R;
solve {
   alldifferent(row);
   alldifferent(all(i in R) row[i]+i);
   alldifferent(all(i in R) row[i]-i);
}
```

Optimization Tools

- Options
  - Go to the **alldifferent** lecture, implement your own propagator from scratch
  - Learn an optimization tool (e.g. CP solver) try out both models

Optimization Tools

- Options
  - Go to the **alldifferent** lecture, implement your own propagator from scratch
  - Learn an optimization tool (e.g. CP solver) try out both models

- An example of option 2,
  - the queens problem in Comet
    - A domain specific language for optimization, including a CP solver.

N-Queens in Comet

```
import cotfd;

Solver<CP> m();
range R = 1..8;
var<CP>{int} row[i in R](m,R);

solve<m> {
   forall(i in R,j in R: i < j) {
      m.post(row[i] != row[j]);
      m.post(row[i] != row[j]+(j-i));
      m.post(row[i] != row[j]-(j-i));
   }
} using {
   label(row);
   cout << row << endl;
}
```

N-Queens in Comet

```
import cotfd;

Solver<CP> m();
range R = 1..8;
var<CP>{int} row[i in R](m,R);

solve<m> {

   m.post(alldifferent(row));
   m.post(alldifferent(all(i in R) row[i] + i));
   m.post(alldifferent(all(i in R) row[i] - i));

} using {
   label(row);
   cout << row << endl;
}
```

Final Note

- Top results on the leader board are most often dedicated algorithms, not general purpose solvers.

Final Note

- Top results on the leader board are most often dedicated algorithms, not general purpose solvers.

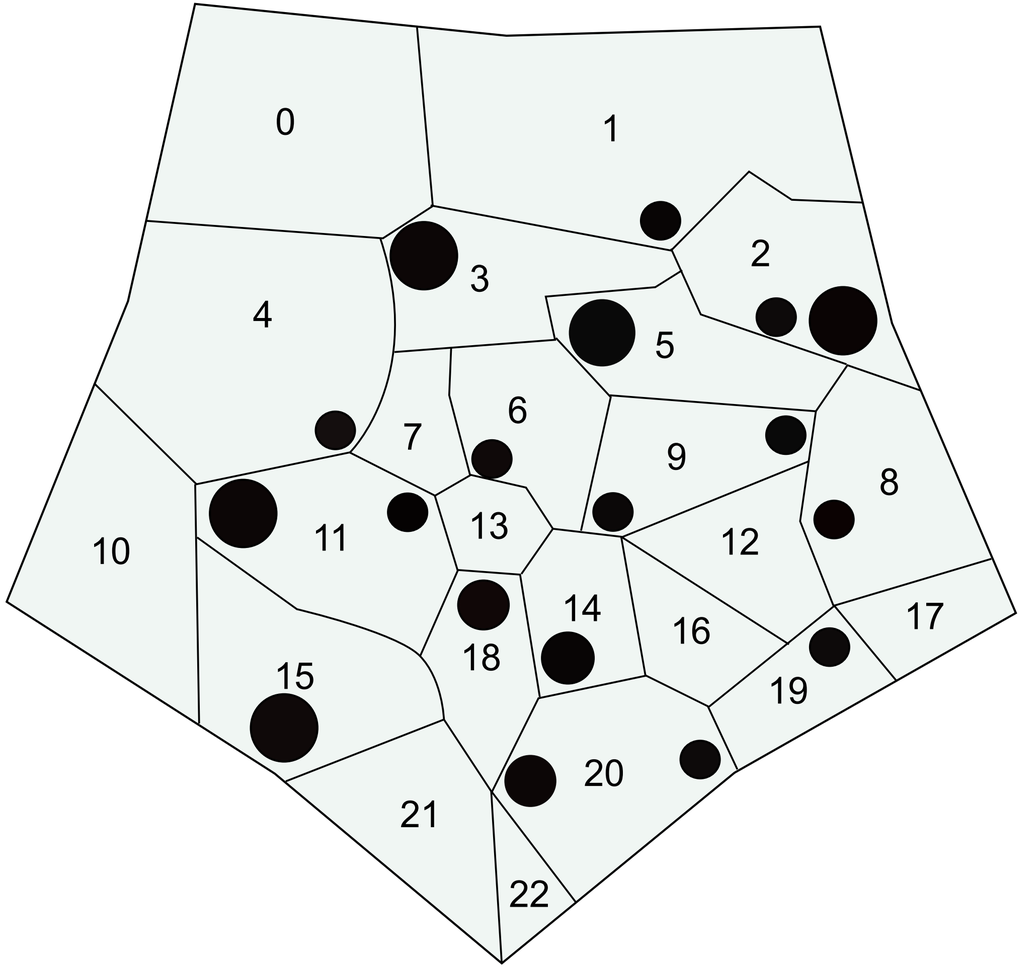
- No solver technology is officially supported or recommended by this class
  - We are not required to answer questions about different solvers
  - Your welcome to help each other learn to use them

Have Fun!

Discrete Optimization

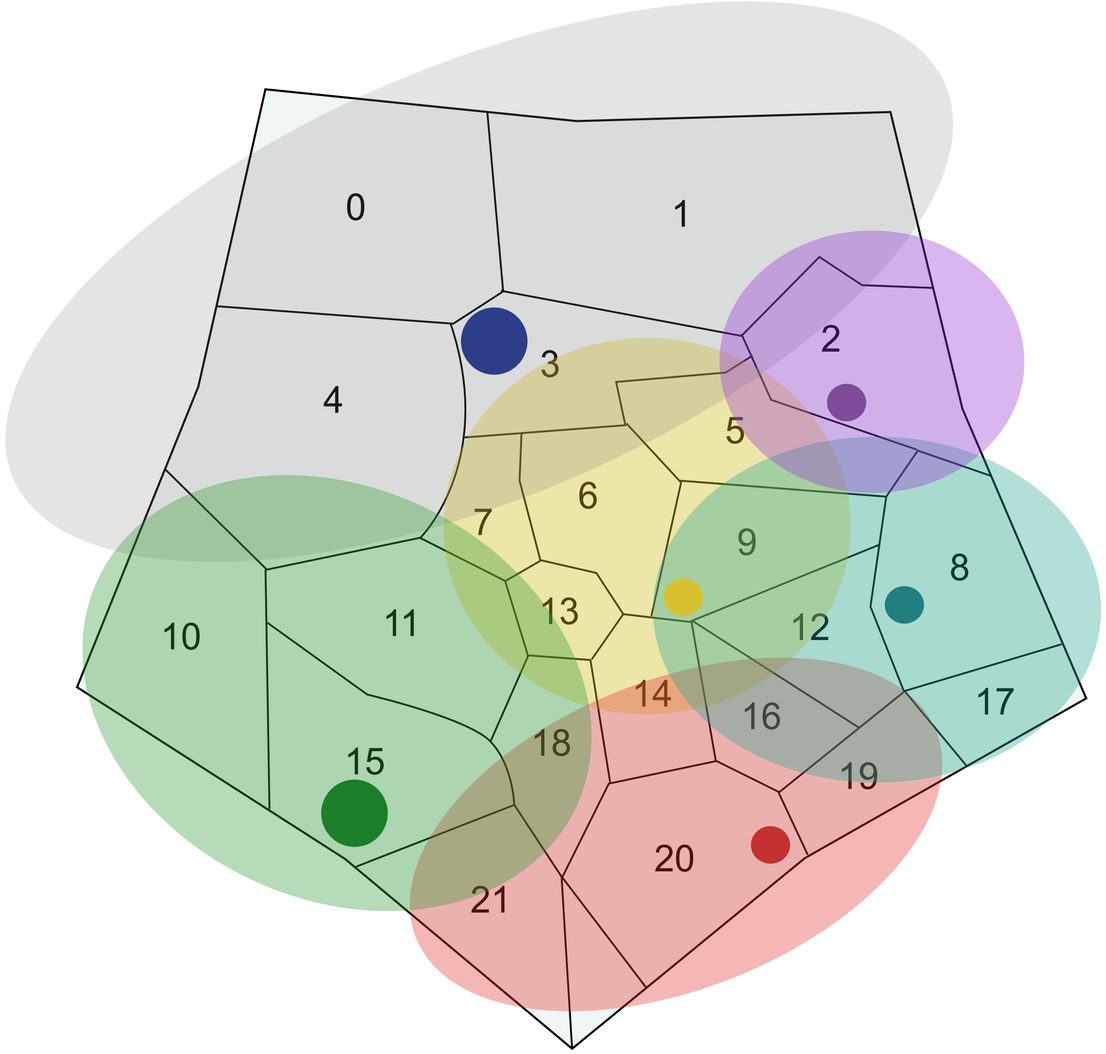
Assignments: Set Cover

Set Cover - Fire Stations Example



22 regions and 18 possible firestations

Set Cover - Fire Stations Example



6 selected firestations and their coverage

Set Cover

- *n* Items (regions),
- *m* Sets (fire stations),
- Set costs c_i
- *Covered items* S_i
- x_i *= 1 if set i is selected*

$$\text{minimize:} \quad \sum_{i \in M} c_i x_i$$

$$\text{subject to:}$$

$$\sum_{i \in M} (j \in S_i) x_i \geq 1 \quad (j \in N)$$

$$x_i \in \{0, 1\}$$

Set Cover

$$\text{minimize:} \quad \sum_{i \in M} c_i x_i$$

$$\text{subject to:}$$

$$\sum_{i \in M} (j \in S_i) x_i \geq 1 \quad (j \in N)$$

$$x_i \in \{0, 1\}$$

Input

```
|N| |M|
c_0 s_0_0 s_0_1 ...
c_1 s_1_0 s_1_1 ...
c_2 s_2_0 s_2_1 ...
...
c_|M|-1 s_(|M|-1)_0  ...
```

Output

```
obj opt
x_0 x_1 ... x_|M|-1
```

Set Cover

$$\text{minimize:} \quad \sum_{i \in M} c_i x_i$$

$$\text{subject to:}$$

$$\sum_{i \in M} (j \in S_i) x_i \geq 1 \quad (j \in N)$$

$$x_i \in \{0, 1\}$$

Input

```
5 4
12.0 0 2
7.0 1 2 3
10.0 1 4
5.0 4
```

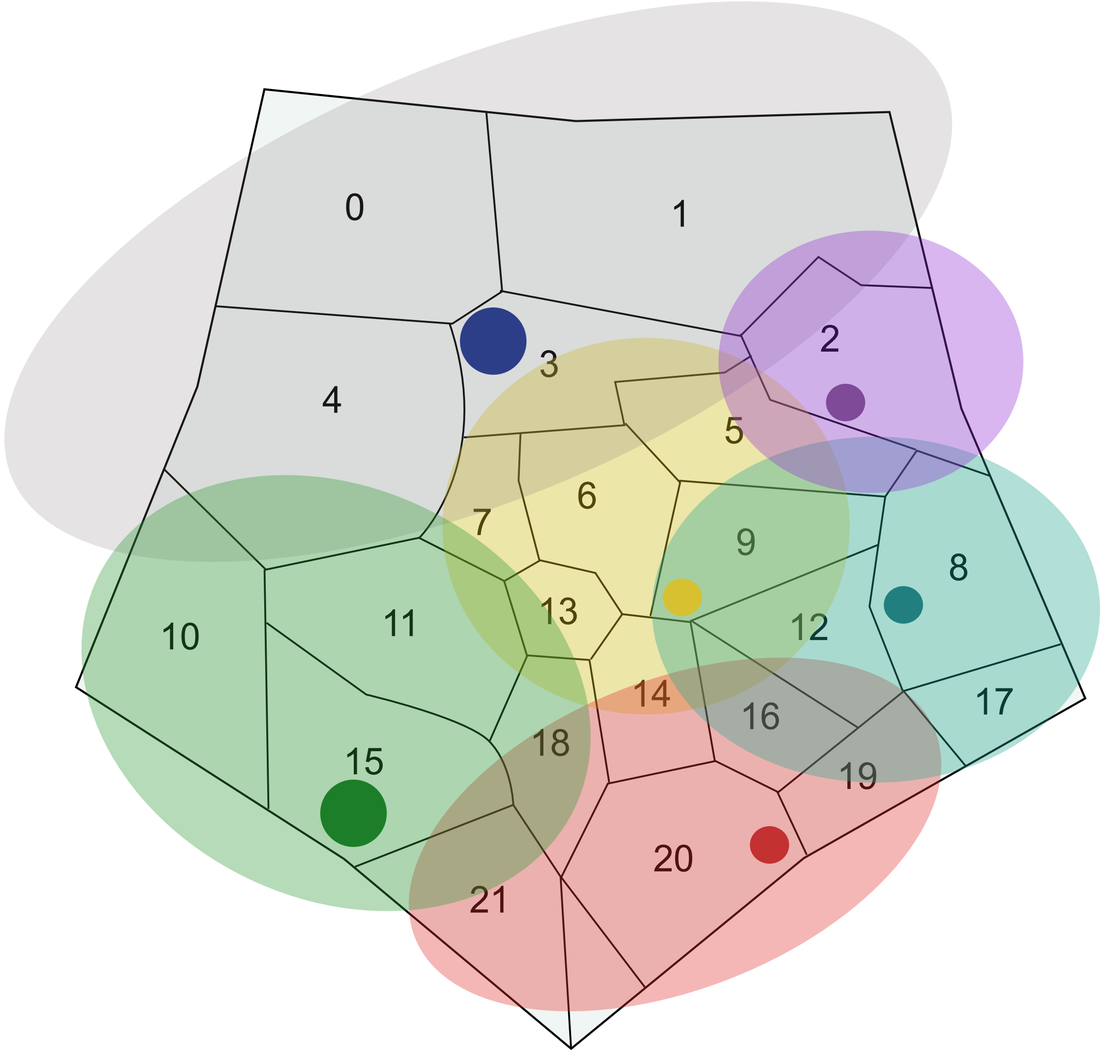
Output

```
24.0 0
1 1 0 1
```

Open Source

- We provide some examples
  - simple greedy solvers
  - simple CP style solver
  - calling an external tool
  - …
- Share code with your classmates
  - github.com/discreteoptimization
- Share algorithms, not specific solutions
  - a lookup table is a really boring algorithm

Have Fun!



6 selected firestations and their coverage