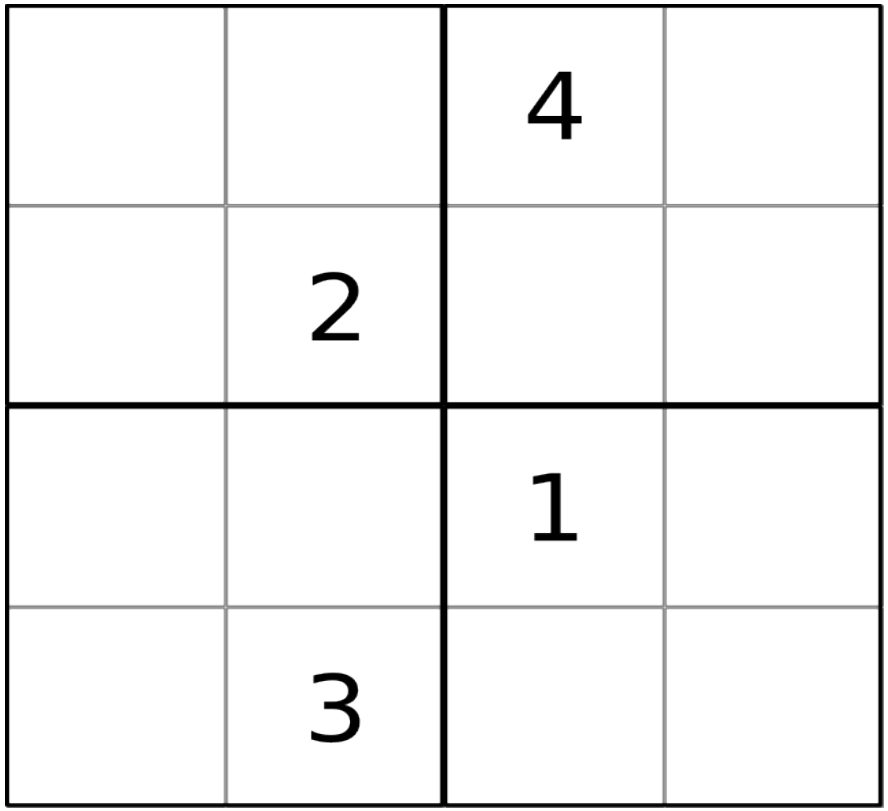
Prolog lecture 6

- Solving Sudoku puzzles
- Constraint Logic Programming
- Natural Language Processing

Playing Sudoku

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

Make the problem easier



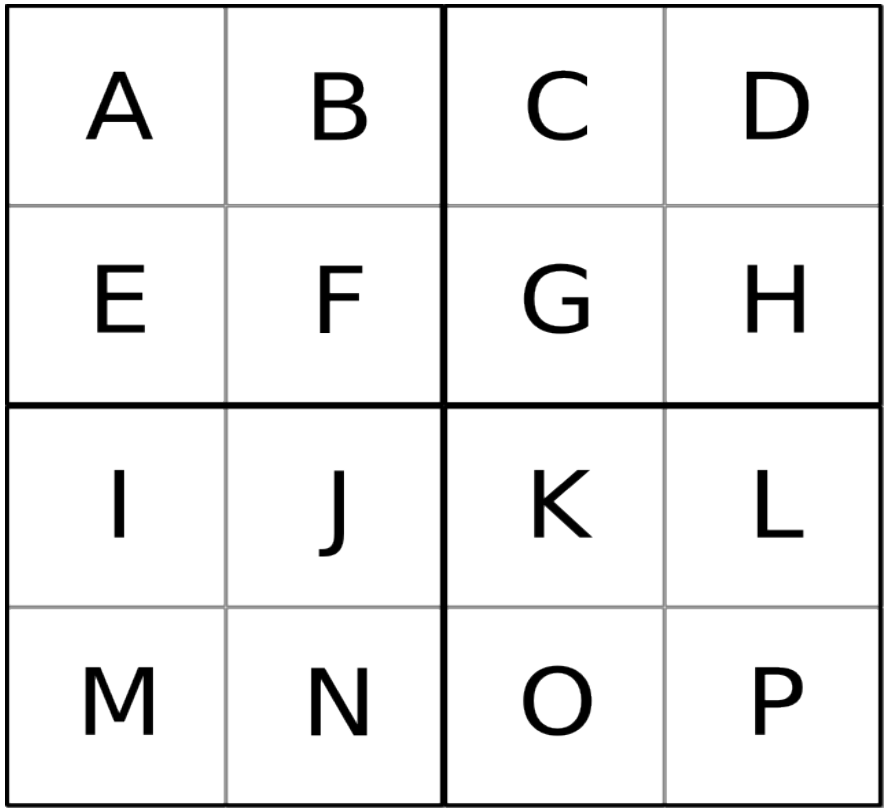
We can model this problem in Prolog using list permutations

Each row must be a permutation of [1,2,3,4]

Each column must be a permutation of [1,2,3,4]

Each 2x2 box must be a permutation of [1,2,3,4]

Represent the board as a list of lists




```
[[A,B,C,D],
 [E,F,G,H],
 [I,J,K,L],
 [M,N,O,P]]
```

The sudoku predicate is built from simultaneous perm constraints

```
sudoku([[X11,X12,X13,X14],[X21,X22,X23,X24],
        [X31,X32,X33,X34],[X41,X42,X43,X44]]) :-
    %rows
    perm([X11,X12,X13,X14],[1,2,3,4]),
    perm([X21,X22,X23,X24],[1,2,3,4]),
    perm([X31,X32,X33,X34],[1,2,3,4]),
    perm([X41,X42,X43,X44],[1,2,3,4]),
    %cols
    perm([X11,X21,X31,X41],[1,2,3,4]),
    perm([X12,X22,X32,X42],[1,2,3,4]),
    perm([X13,X23,X33,X43],[1,2,3,4]),
    perm([X14,X24,X34,X44],[1,2,3,4]),
    %boxes
    perm([X11,X12,X21,X22],[1,2,3,4]),
    perm([X13,X14,X23,X24],[1,2,3,4]),
    perm([X31,X32,X41,X42],[1,2,3,4]),
    perm([X33,X34,X43,X44],[1,2,3,4]).
```

Scale up in the obvious way to 3x3

X11	X12	X13	X14	X15	X16	X17	X18	X19
X21	X22	X23	X24	X25	X26	X27	X28	X29
X31	X32	X33	X34	X35	X36	X37	X38	X39
X41	X42	X43	X44	X45	X46	X47	X48	X49
X51	X52	X53	X54	X55	X56	X57	X58	X59
X61	X62	X63	X64	X65	X66	X67	X68	X69
X71	X72	X73	X74	X75	X76	X77	X78	X79
X81	X82	X83	X84	X85	X86	X87	X88	X89
X91	X92	X93	X94	X95	X96	X97	X98	X99

Brute-force is impractically slow

There are very many valid grids:

$6670903752021072936960 \approx 6.671 \times 10^{21}$

Our current approach does not encode the interrelationships between the constraints

For more information on Sudoku enumeration:

http://www.afjarvis.staff.shef.ac.uk/sudoku/

Prolog programs can be viewed as constraint satisfaction problems

Prolog is limited to the single equality constraint:

- two terms must unify

We can generalise this to include other types of constraint

Doing so leads to Constraint Logic Programming

- and a means to solve Sudoku problems (p319)

Consider variables taking values from domains with constraints

Given:

- the set of variables
- the domains of each variable
- constraints on these variables

We want to find:

- an assignment of values to variables satisfying the constraints

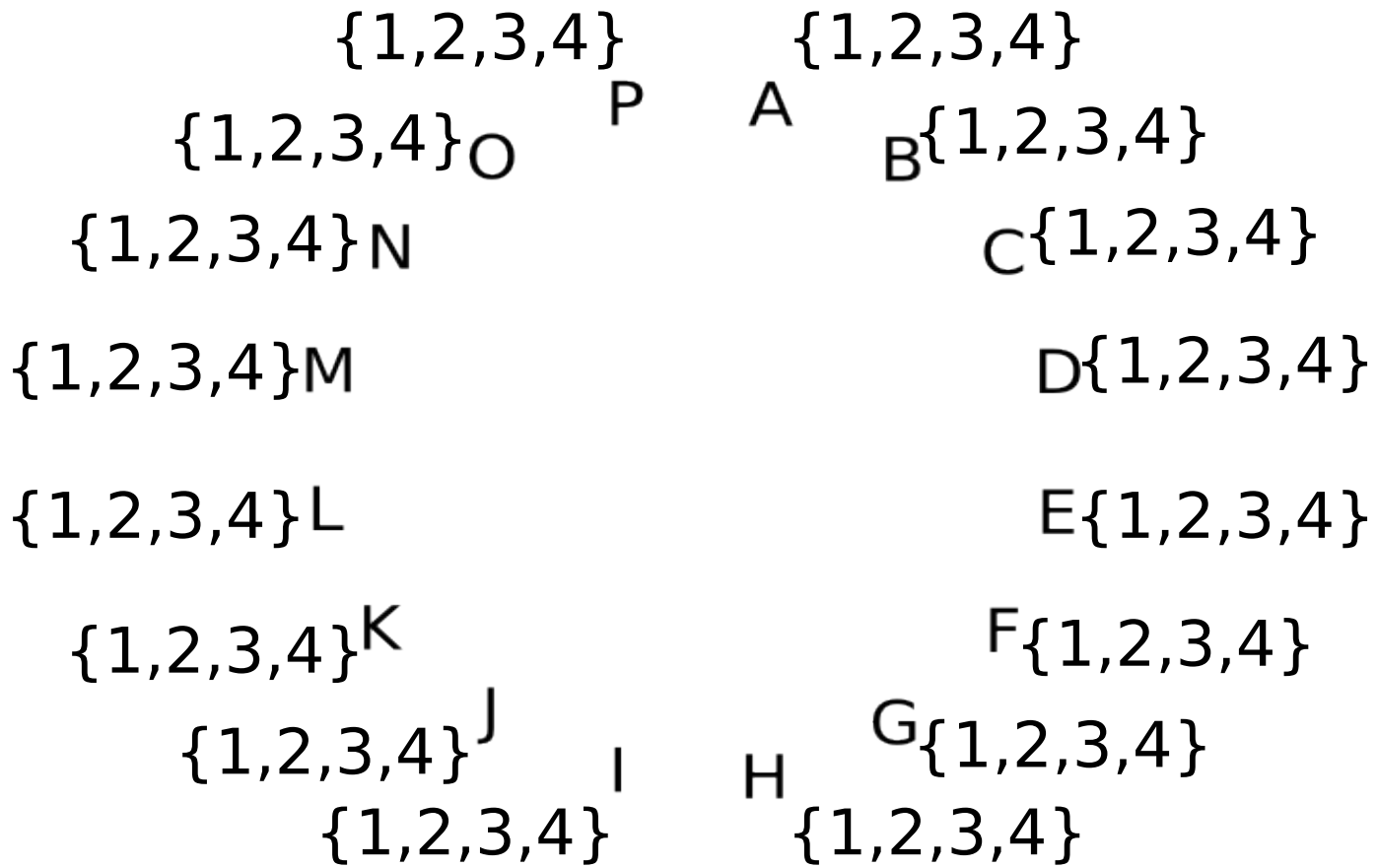
Sudoku can be expressed as constraints

First, we express the variables and domains

$A \in \{1,2,3,4\}$
$B \in \{1,2,3,4\}$
$C \in \{1,2,3,4\}$
$D \in \{1,2,3,4\}$
$E \in \{1,2,3,4\}$
$F \in \{1,2,3,4\}$
$G \in \{1,2,3,4\}$
$H \in \{1,2,3,4\}$
$I \in \{1,2,3,4\}$
$J \in \{1,2,3,4\}$
$K \in \{1,2,3,4\}$
$L \in \{1,2,3,4\}$
$M \in \{1,2,3,4\}$
$N \in \{1,2,3,4\}$
$O \in \{1,2,3,4\}$
$P \in \{1,2,3,4\}$

A	B	C	D
E	F	G	H
I	J	K	L
M	N	O	P

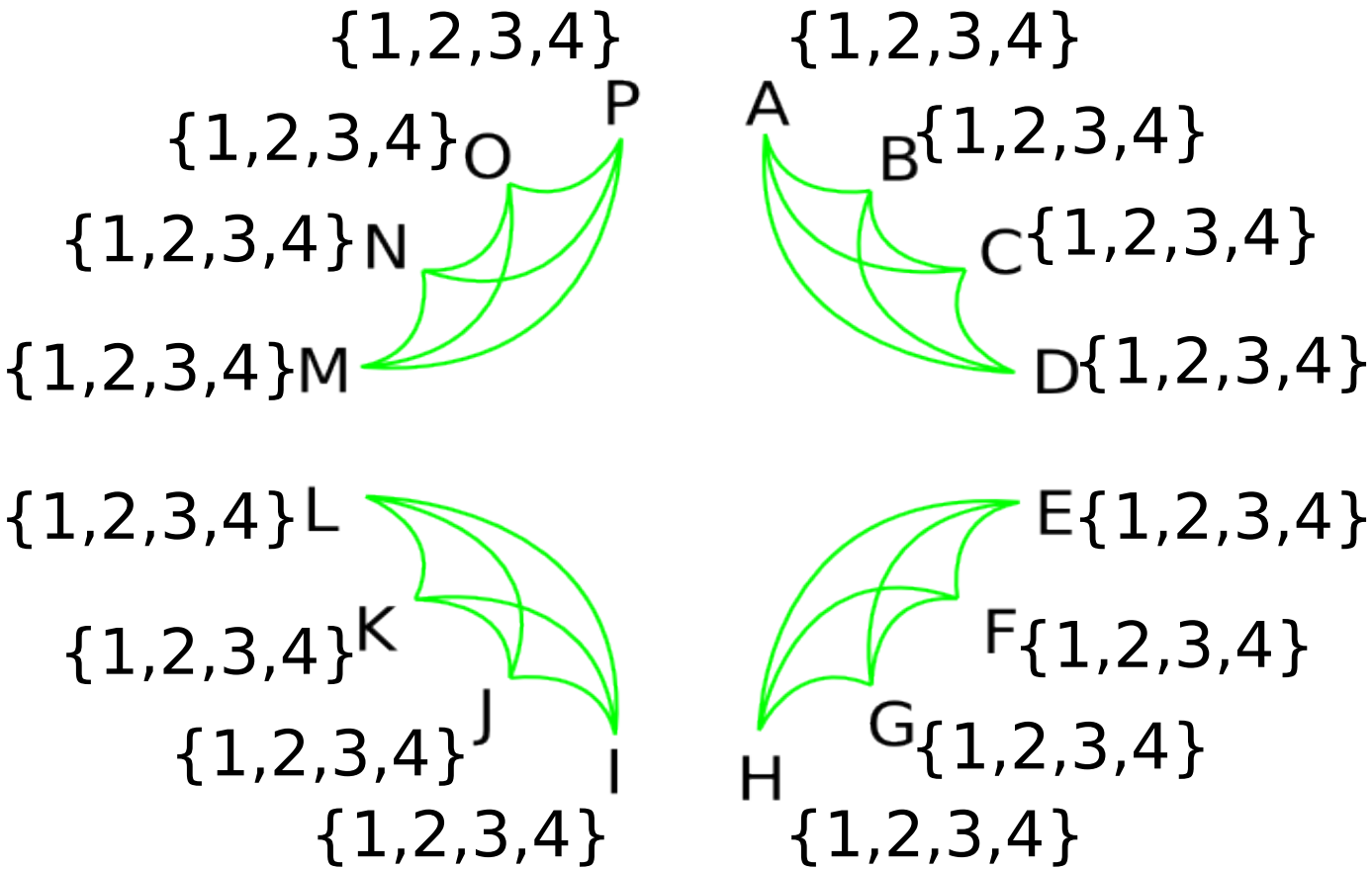
Express Sudoku as a Constraint Graph



A	B	C	D
E	F	G	H
I	J	K	L
M	N	O	P

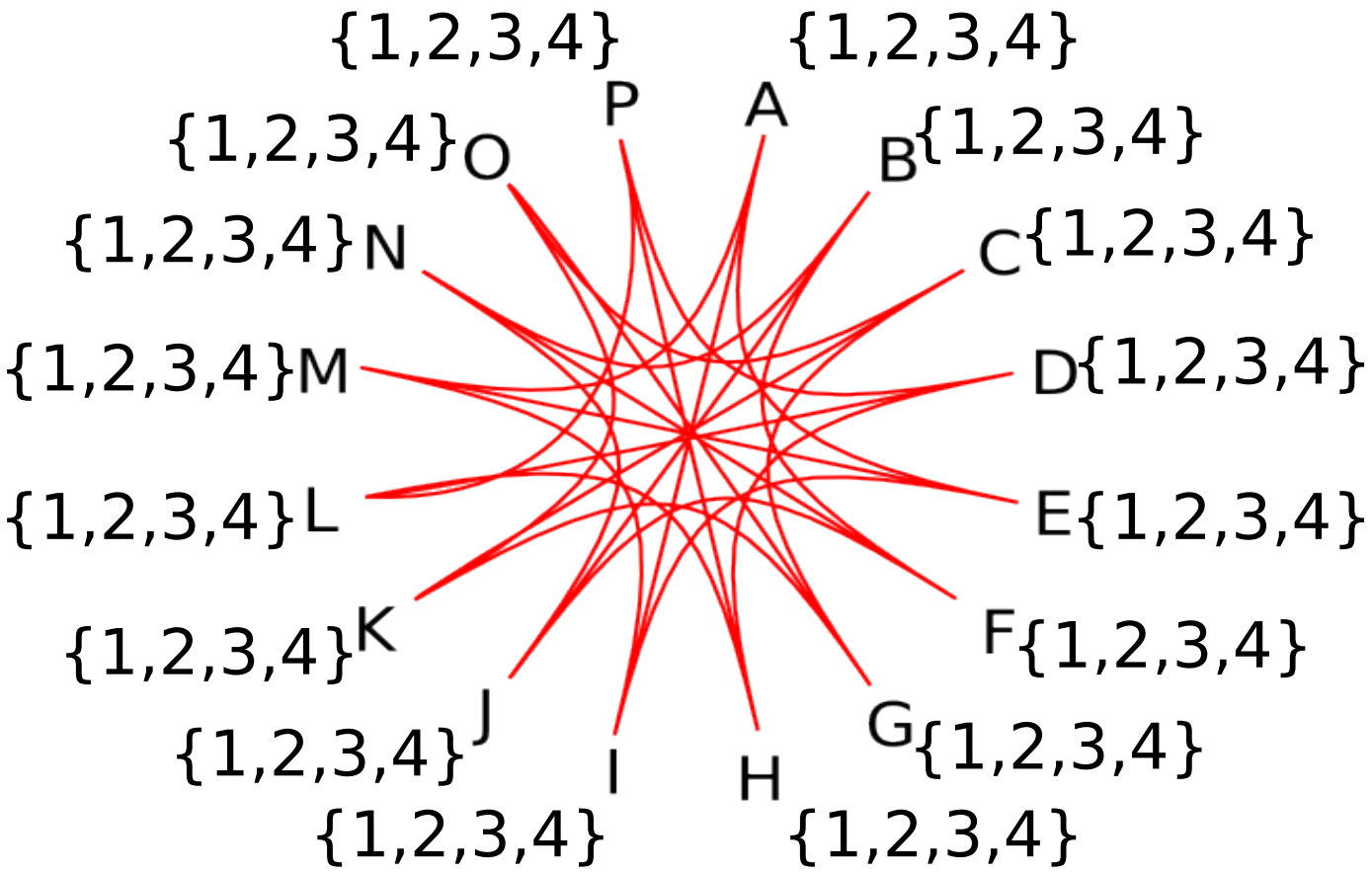
Constraints:
All variables in rows are different

Edges represent inequalities between variables



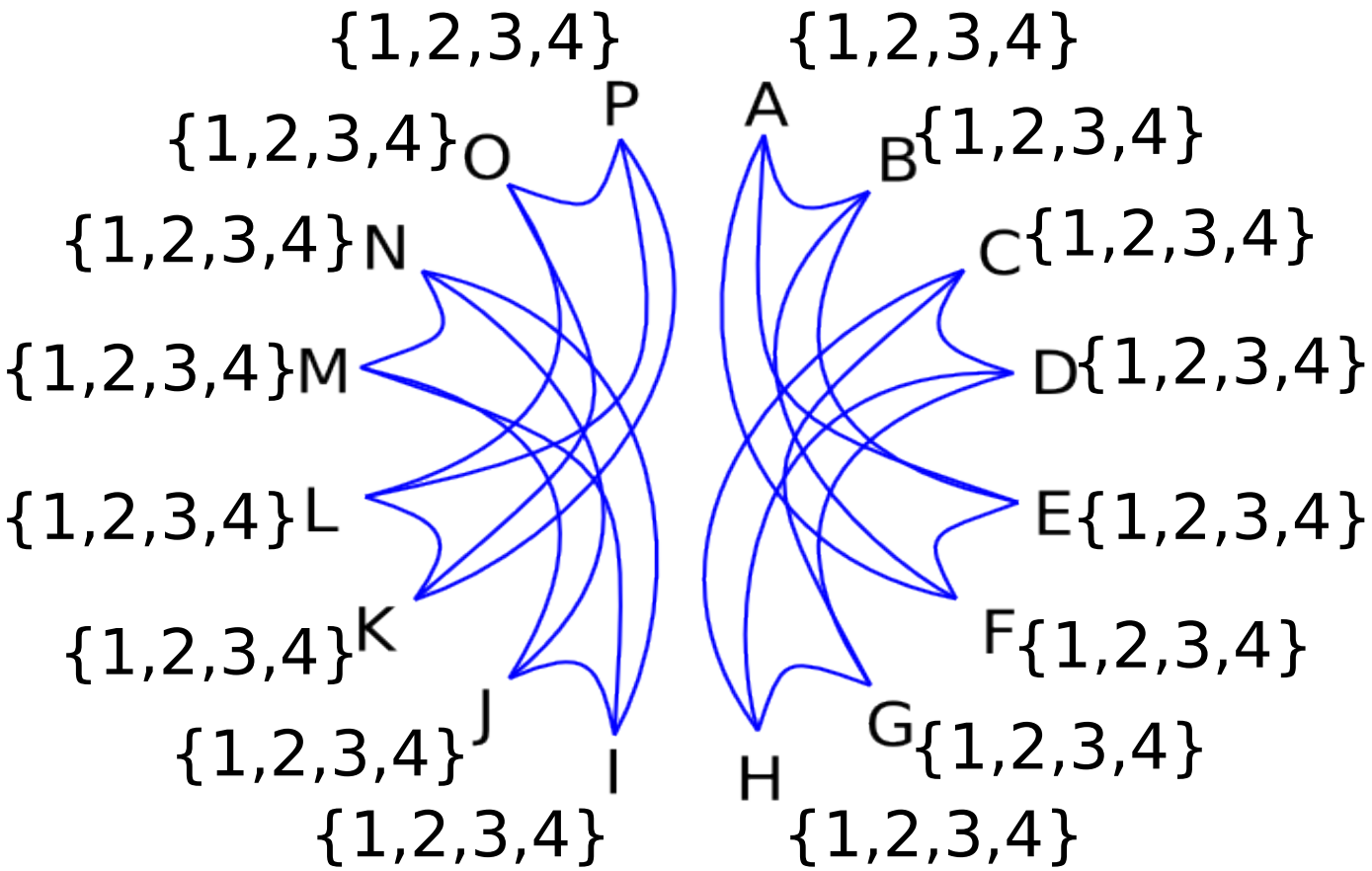
A	B	C	D
E	F	G	H
I	J	K	L
M	N	O	P

Constraints:
All variables in columns are different



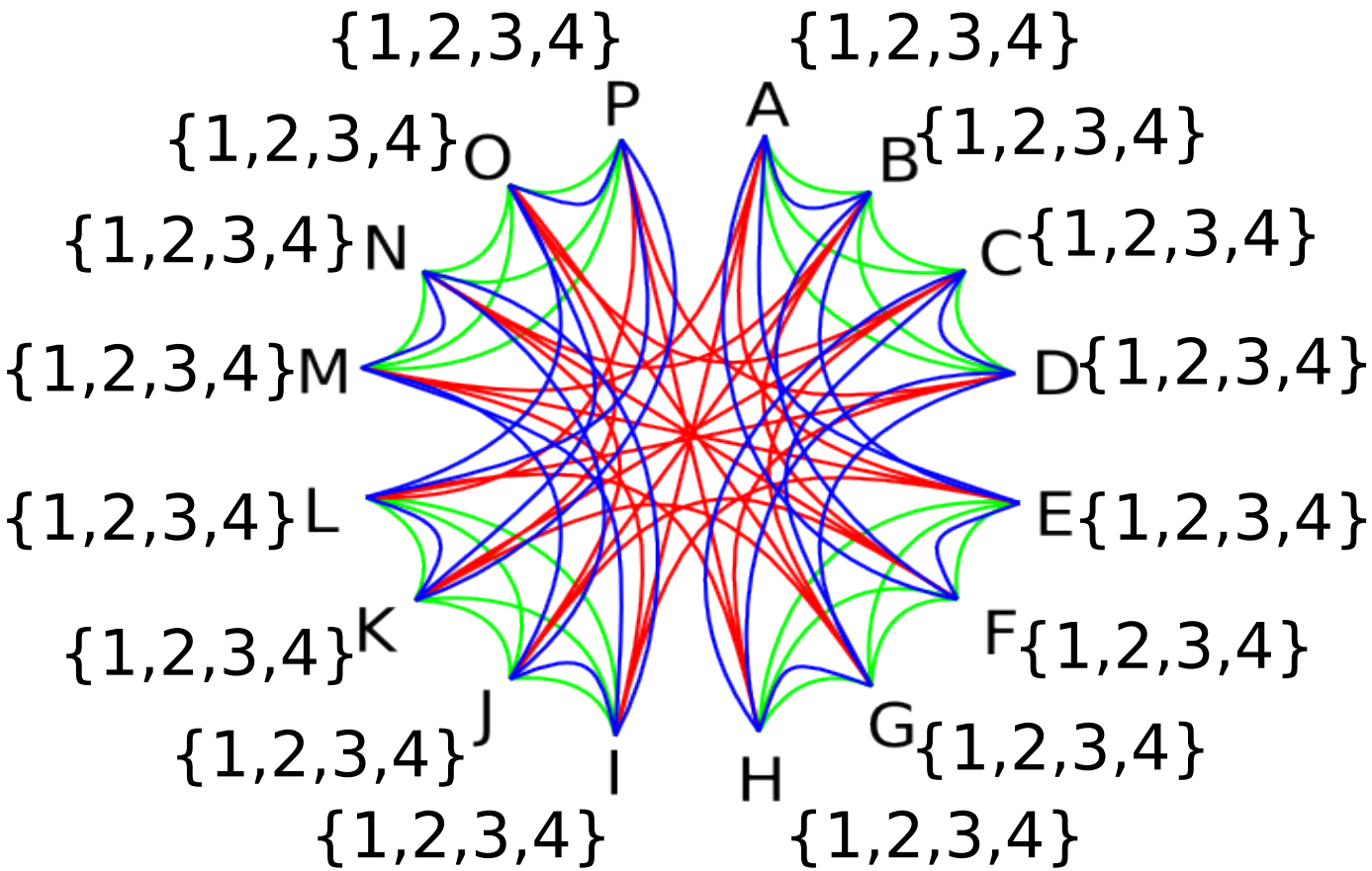
A	B	C	D
E	F	G	H
I	J	K	L
M	N	O	P

Constraints:
All variables in boxes are different



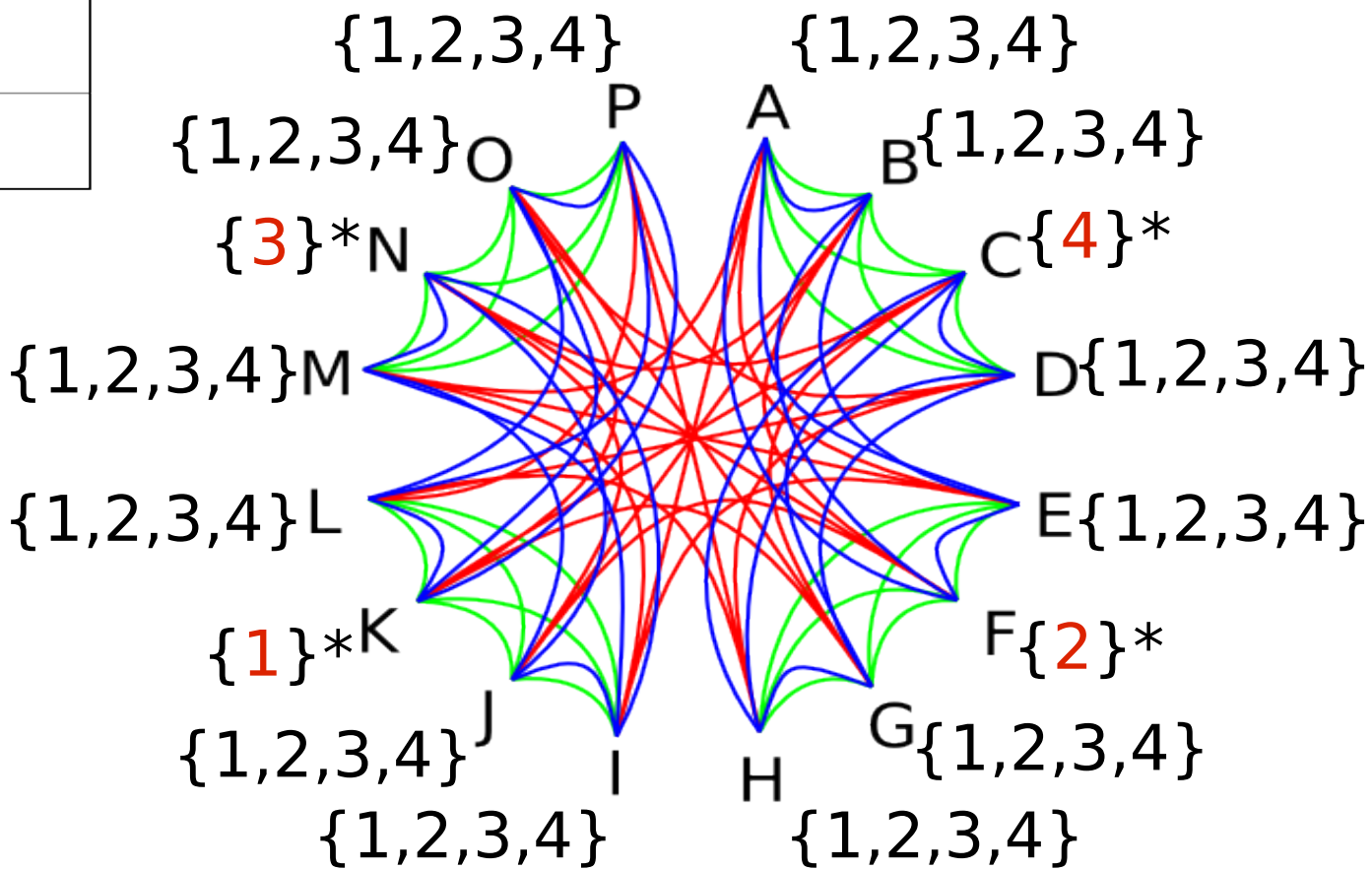
A	B	C	D
E	F	G	H
I	J	K	L
M	N	O	P

All constraints shown together



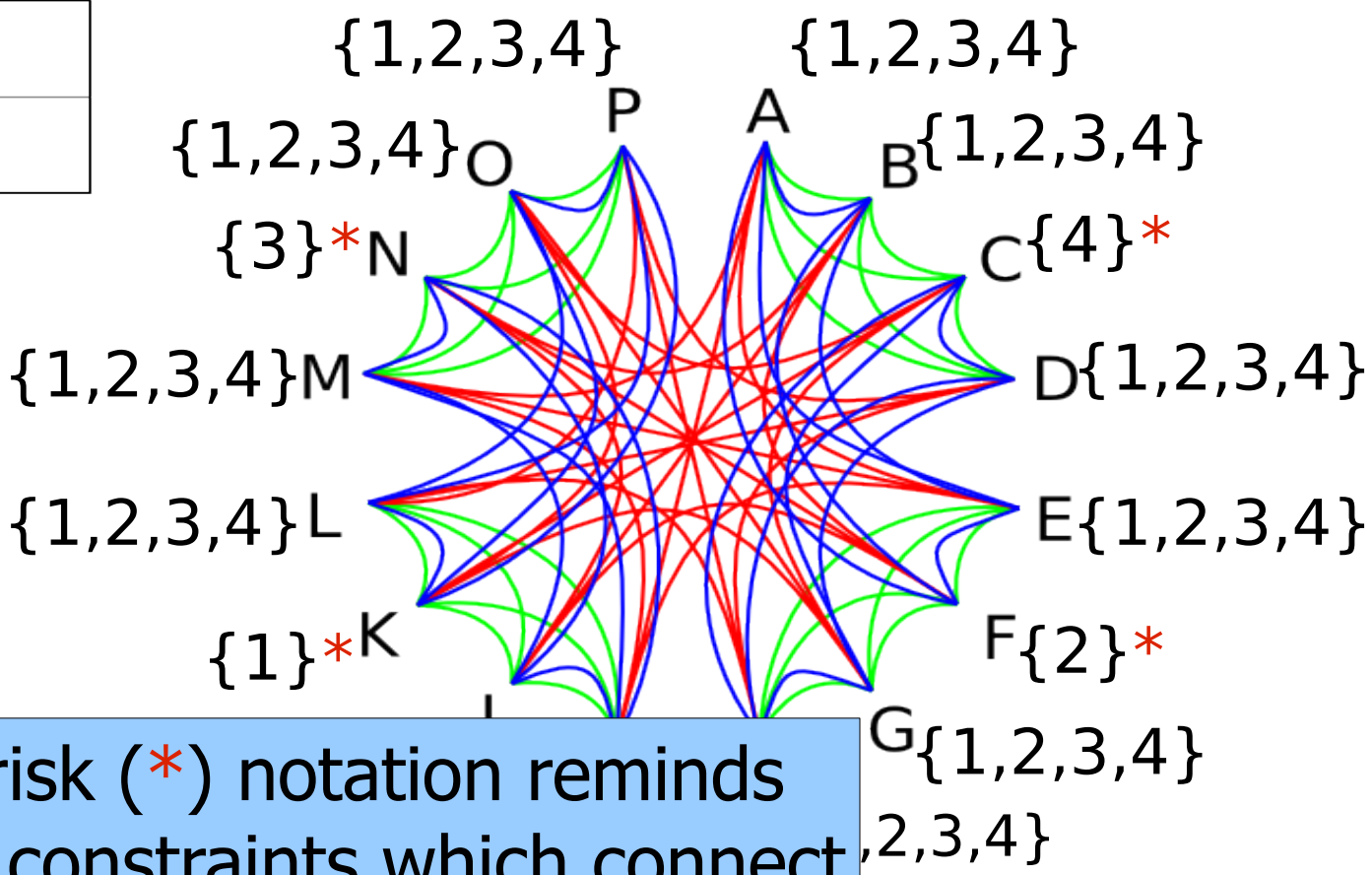
Reduce domains according to initial values

		4	
	2		
		1	
	3		



When a domain changes we update its constraints

		4	
	2		
		1	
	3		

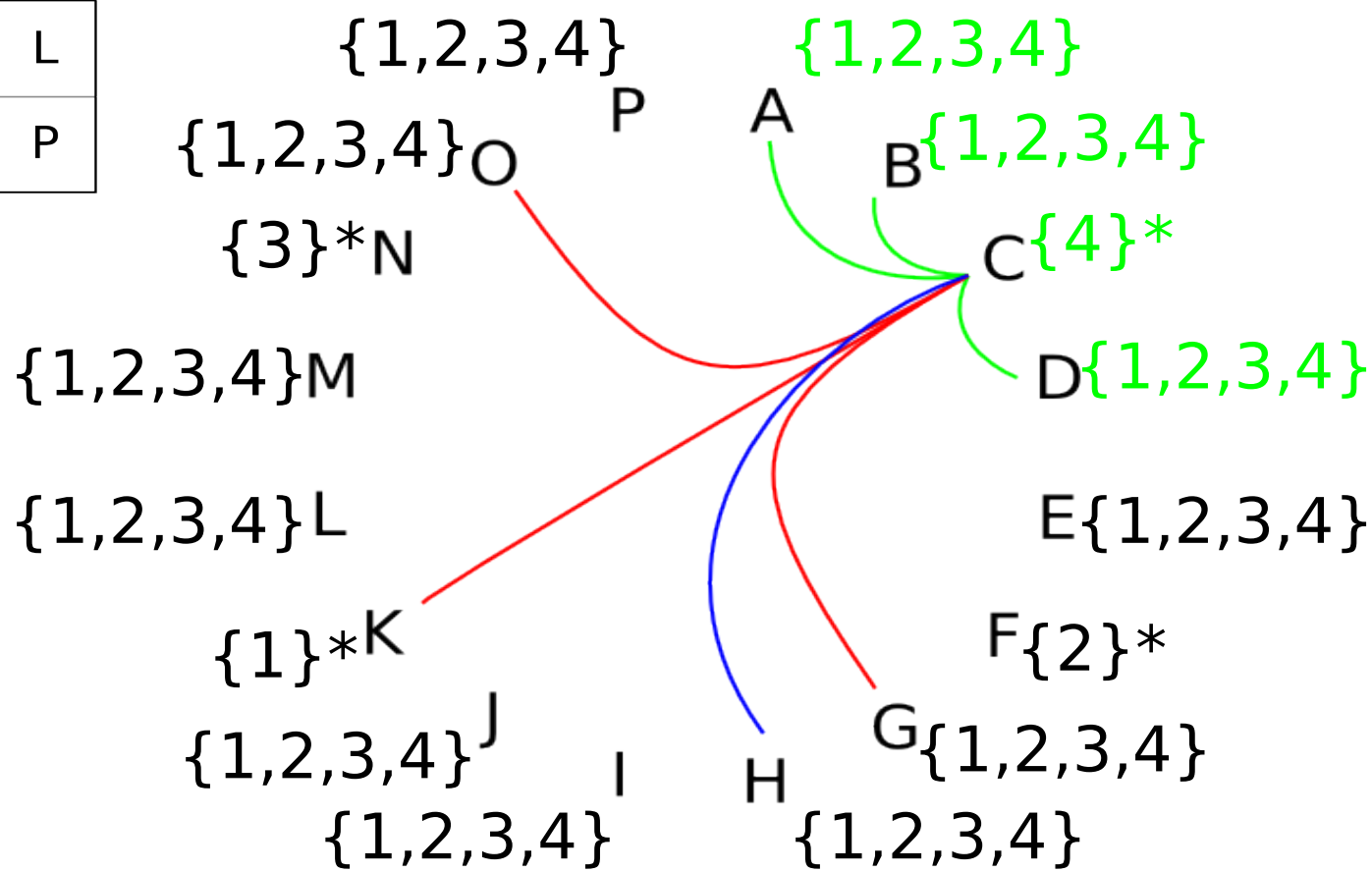


The asterisk (*) notation reminds us that all constraints which connect to this variable need updating

Update constraints connected to C

A	B	C	D
E	F	G	H
I	J	K	L
M	N	O	P

We will remove 4 from the domain of A, B and D

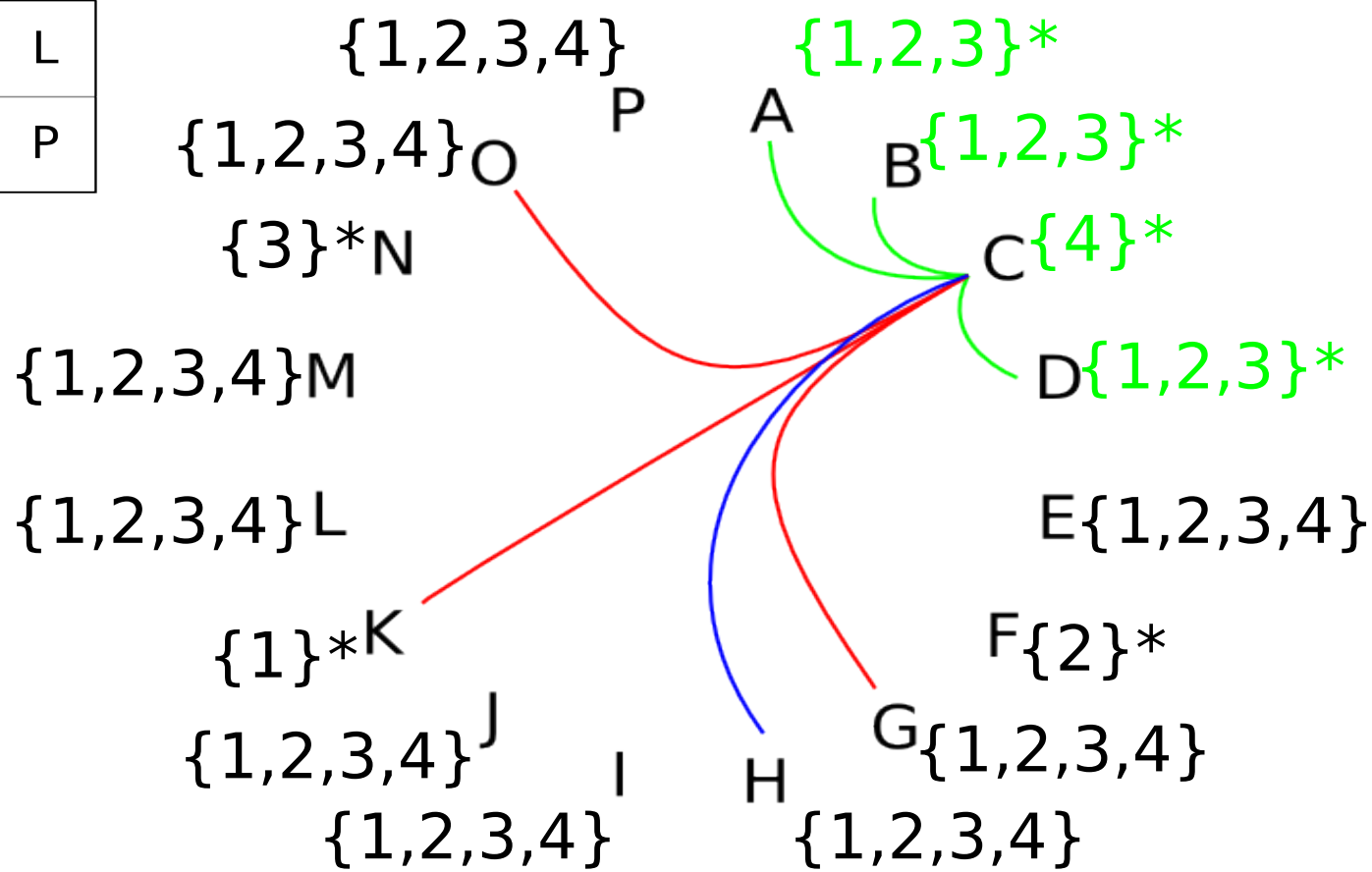


Update constraints connected to C

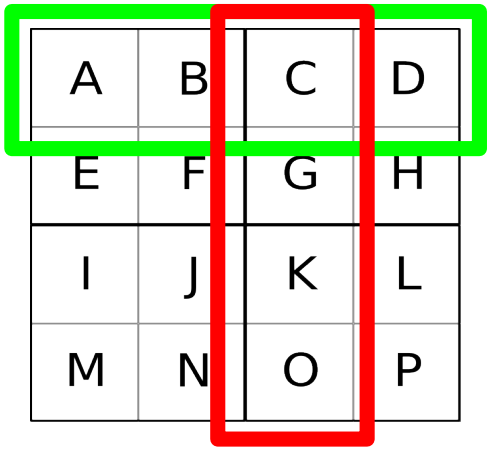
A	B	C	D
E	F	G	H
I	J	K	L
M	N	O	P

We add asterisks to A, B and D

...but will defer looking at them



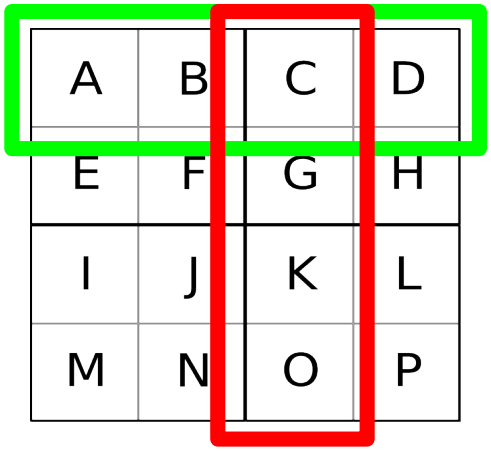
Update constraints connected to C

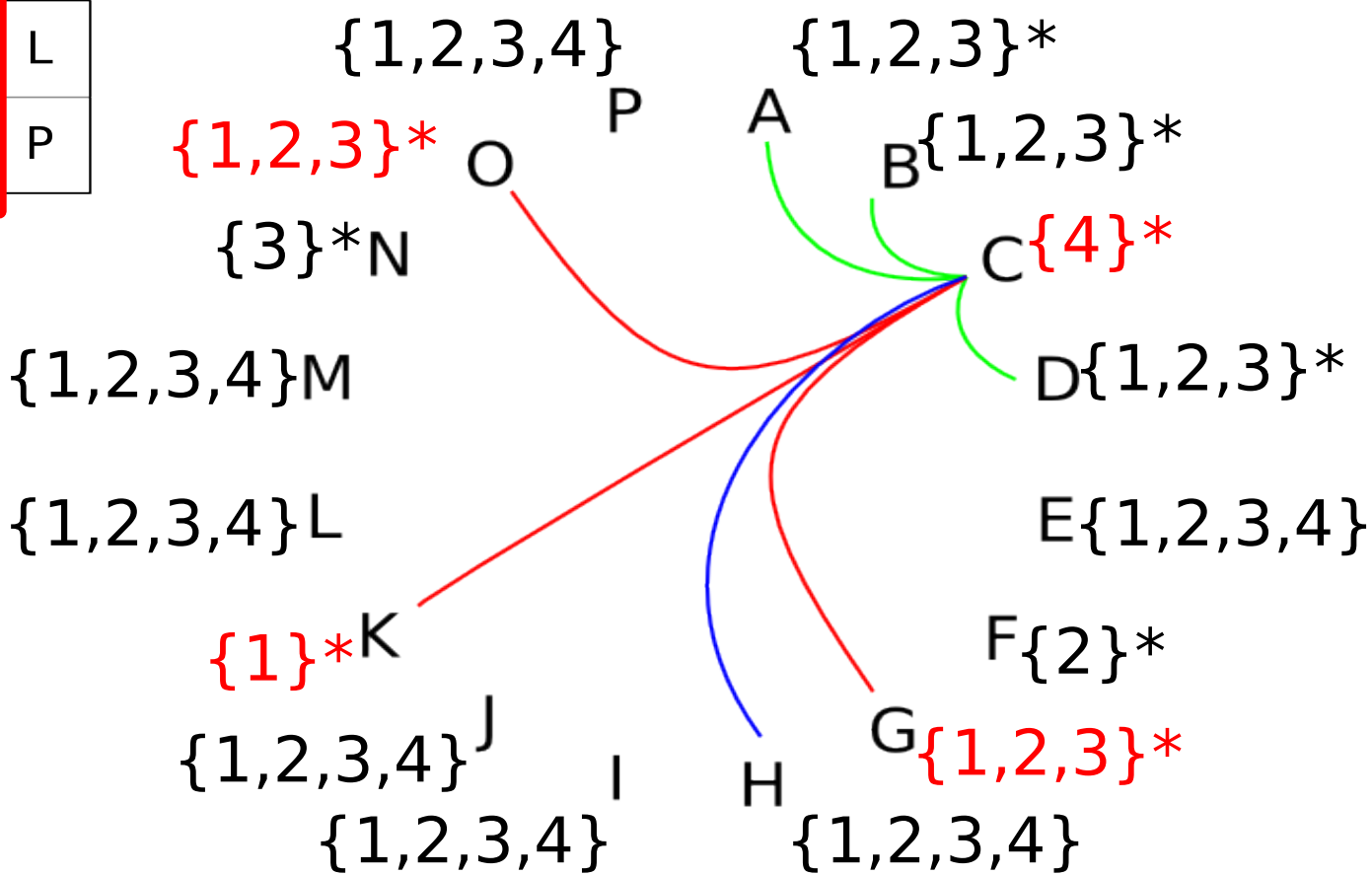


Now examine column constraints

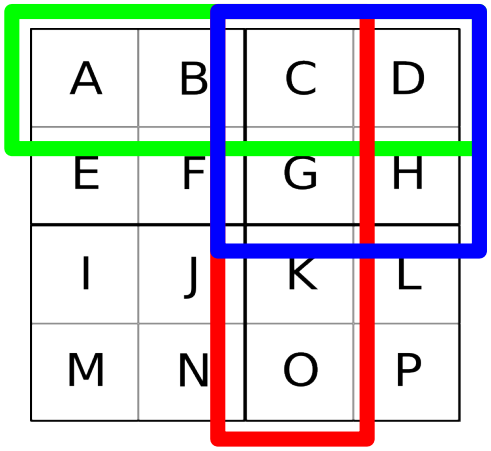
{1,2,3,4} P
{1,2,3,4} O
{3}* N
{1,2,3,4} M
{1,2,3,4} L
{1}* K
{1,2,3,4} J
{1,2,3,4} I
{1,2,3}* A
{1,2,3}* B
{4}* C
{1,2,3}* D
{1,2,3,4} E
{2}* F
{1,2,3,4} G
{1,2,3,4} H

Update constraints connected to C





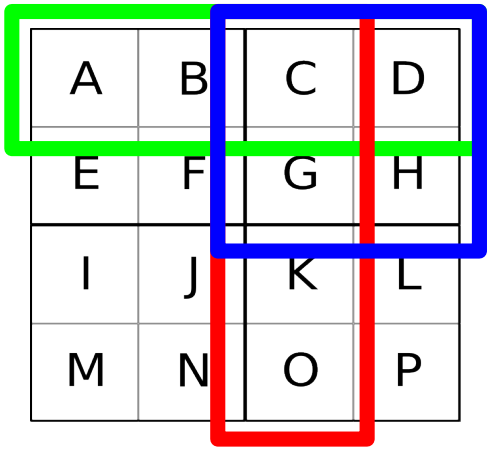
Update constraints connected to C



Note that D and G have already had their domains updated

{1,2,3,4} P
{1,2,3}* O
{3}* N
{1,2,3,4} M
{1,2,3,4} L
{1}* K
{1,2,3,4} J
{1,2,3,4} I
{1,2,3}* A
{1,2,3}* B
{4}* C
{1,2,3}* D
{1,2,3,4} E
{2}* F
{1,2,3}* G
{1,2,3,4} H

Update constraints connected to C

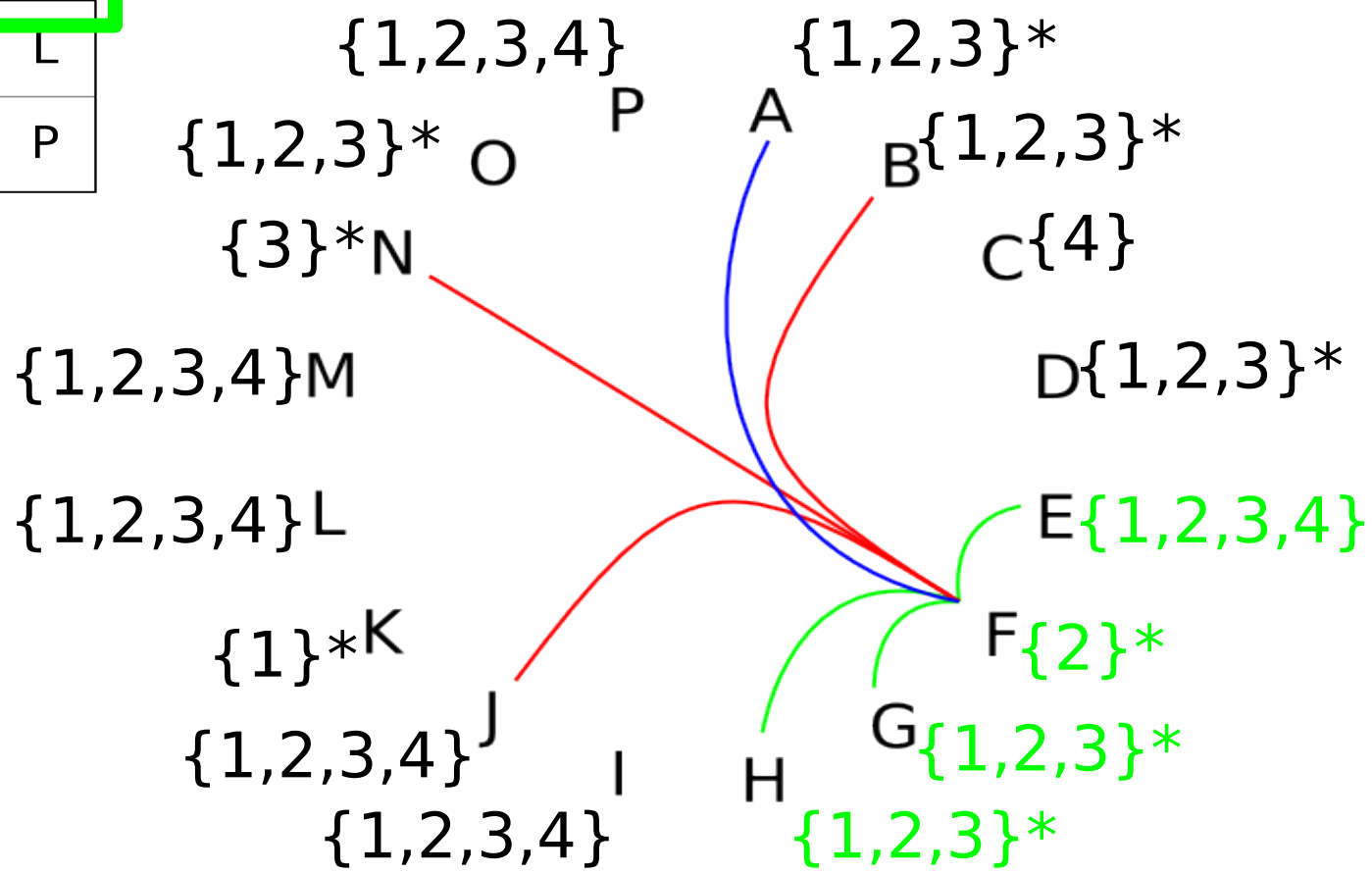


We have exhausted C's constraints for now

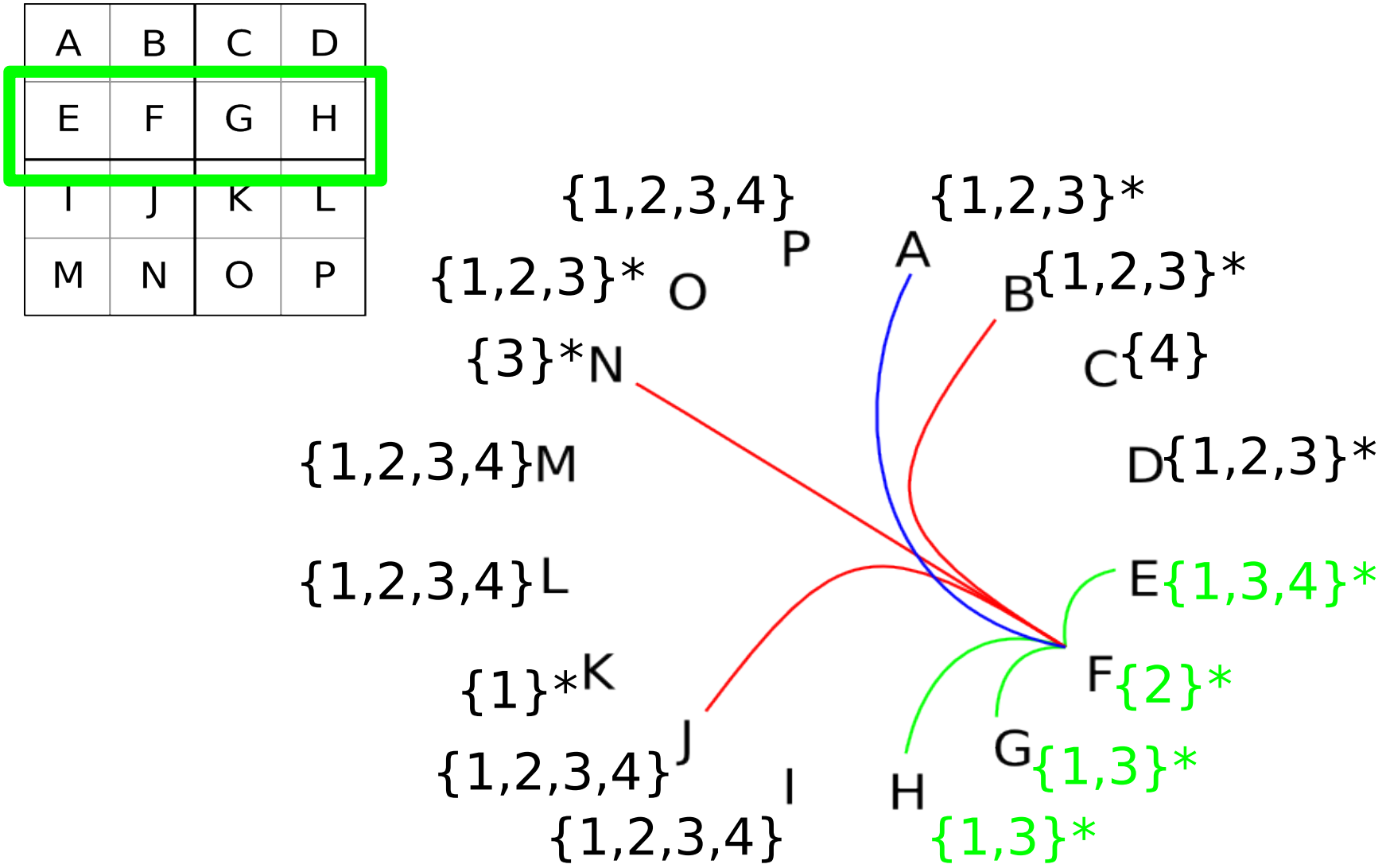
{1,2,3,4} P
{1,2,3}* O
{3}* N
{1,2,3,4} M
{1,2,3,4} L
{1}* K
{1,2,3,4} J
{1,2,3,4} I
{1,2,3}* A
{1,2,3}* B
{4}* C
{1,2,3}* D
{1,2,3,4} E
{2}* F
{1,2,3}* G
{1,2,3}* H

Update constraints connected to F

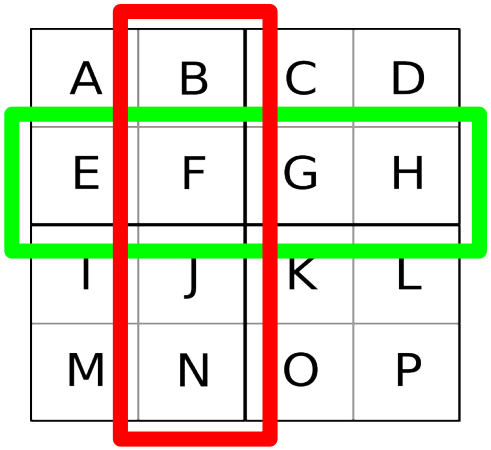
A	B	C	D
E	F	G	H
I	J	K	L
M	N	O	P

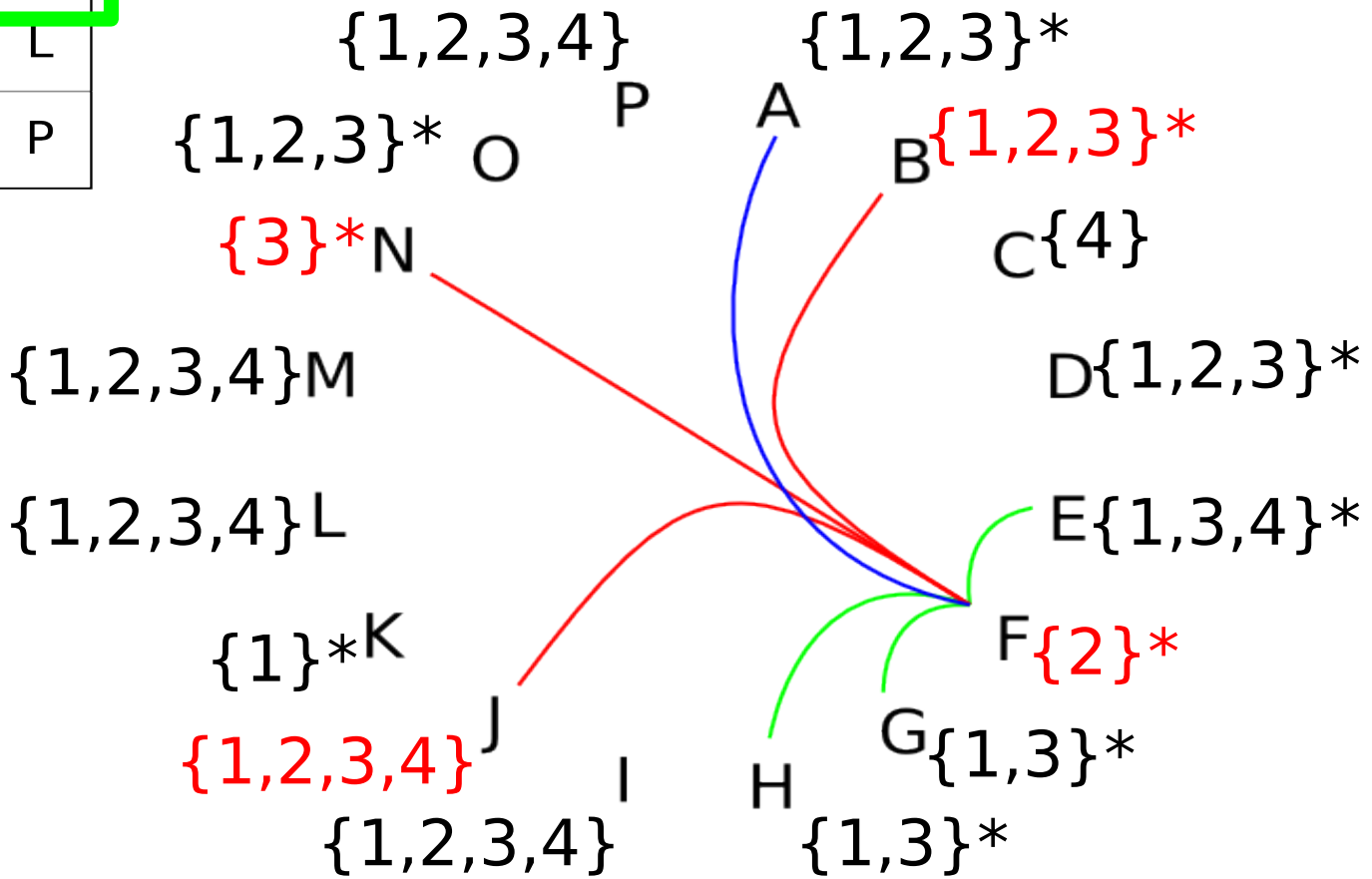


Update constraints connected to F

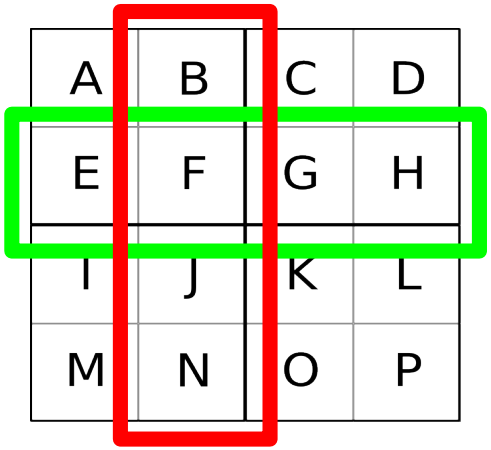


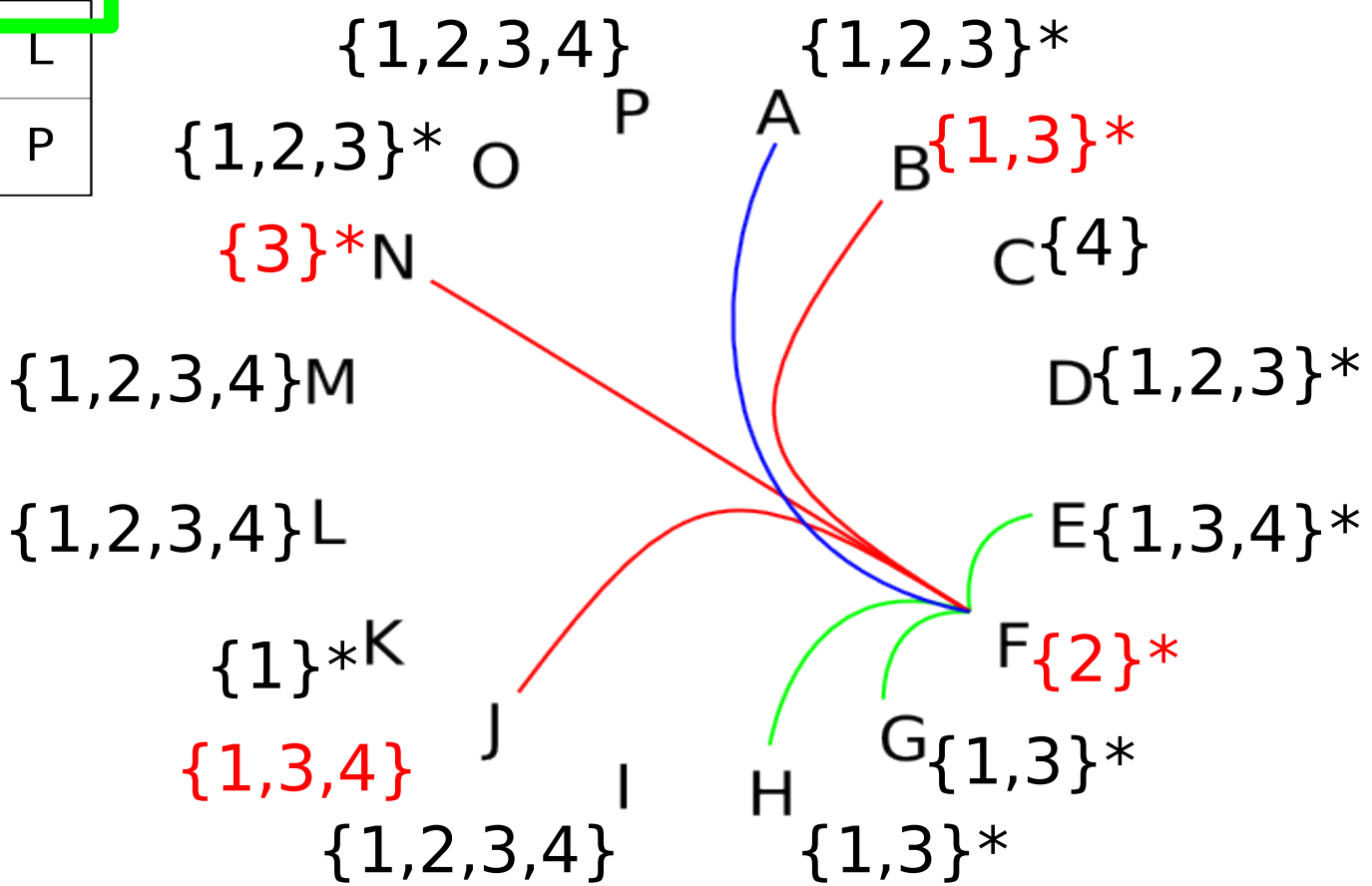
Update constraints connected to F



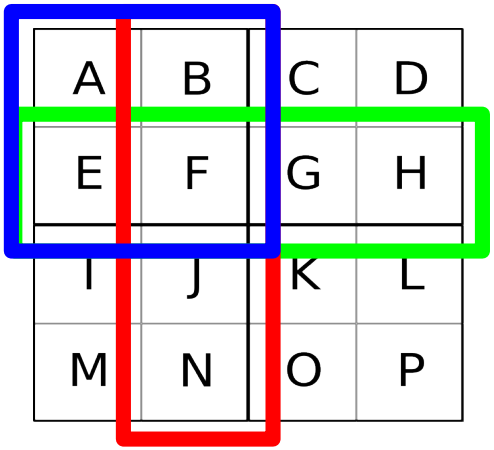


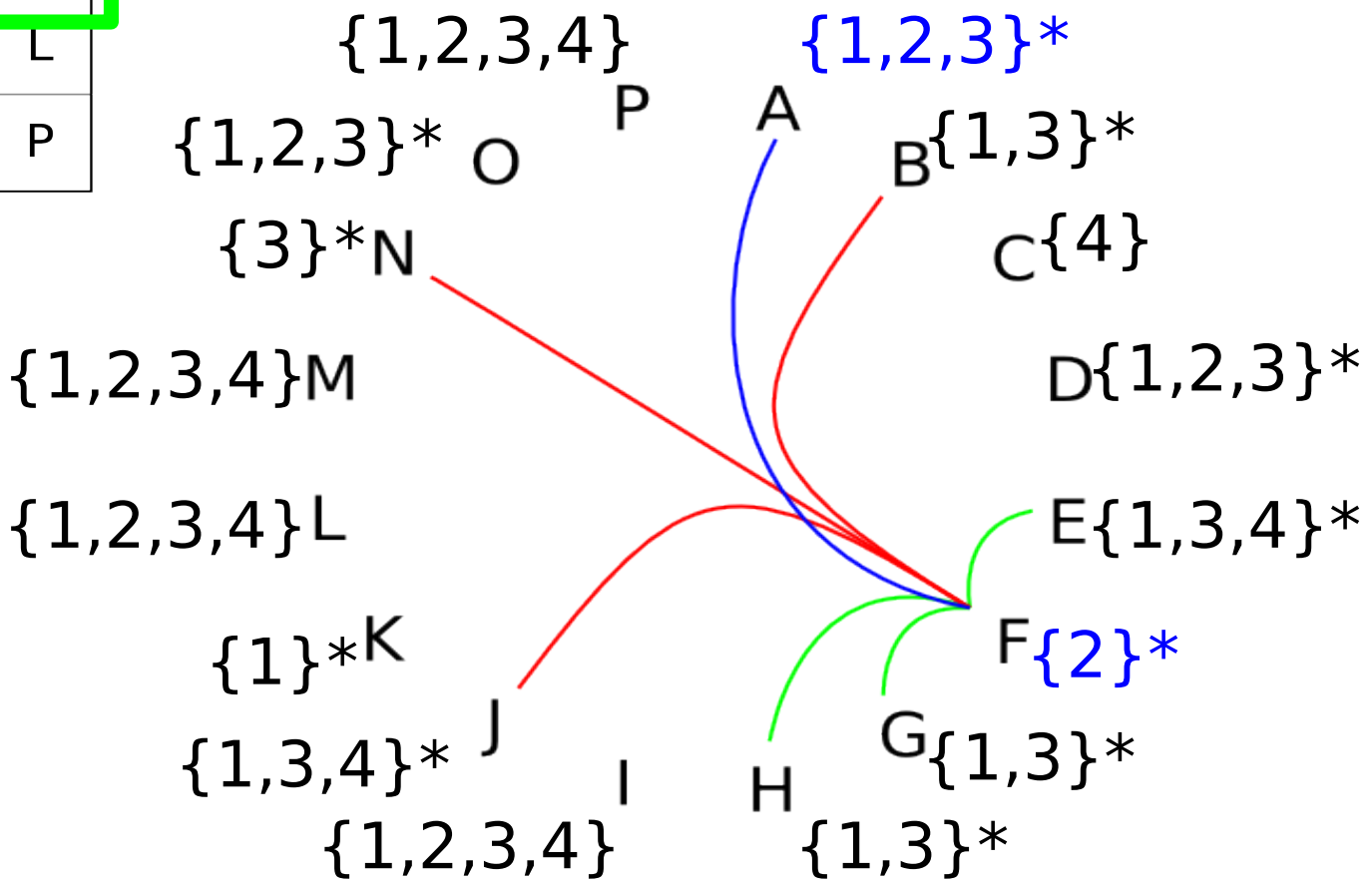
Update constraints connected to F



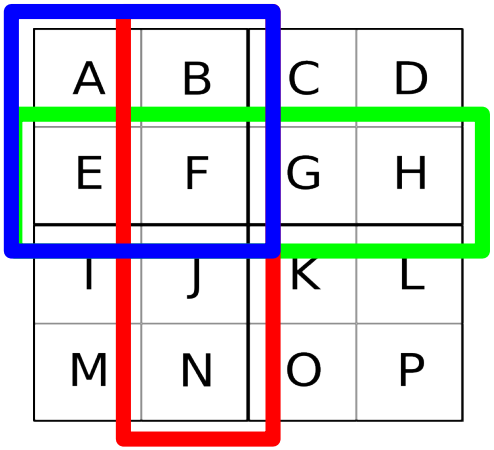


Update constraints connected to F





Update constraints connected to F

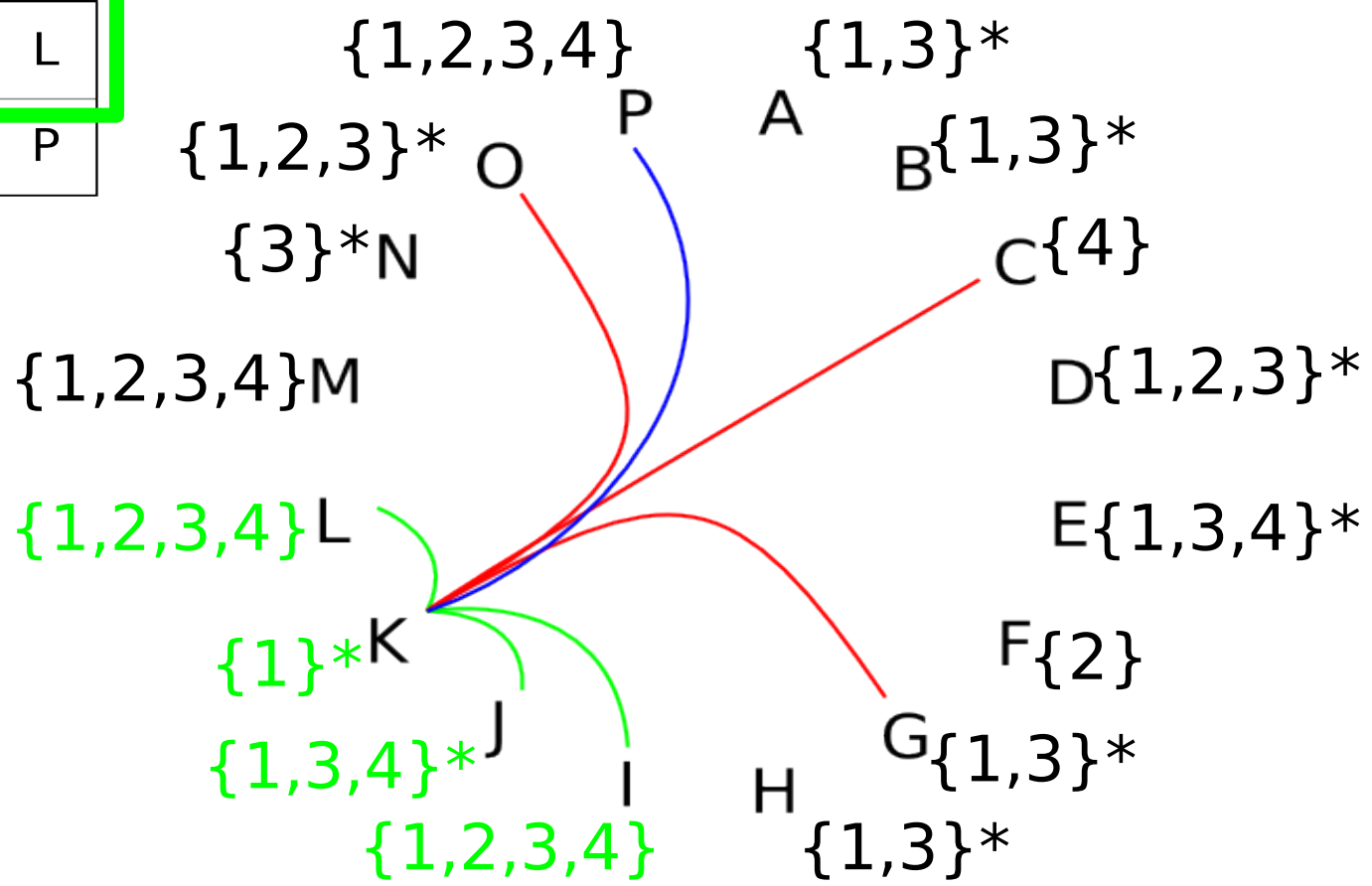


We have exhausted F's constraints for now

{1,2,3,4} P
{1,3}* A
{1,2,3}* O
B {1,3}*
{3}* N
C {4}
{1,2,3,4} M
D {1,2,3}*
{1,2,3,4} L
E {1,3,4}*
{1}* K
F {2}*
{1,3,4}* J
G {1,3}*
I {1,2,3,4}
H {1,3}*

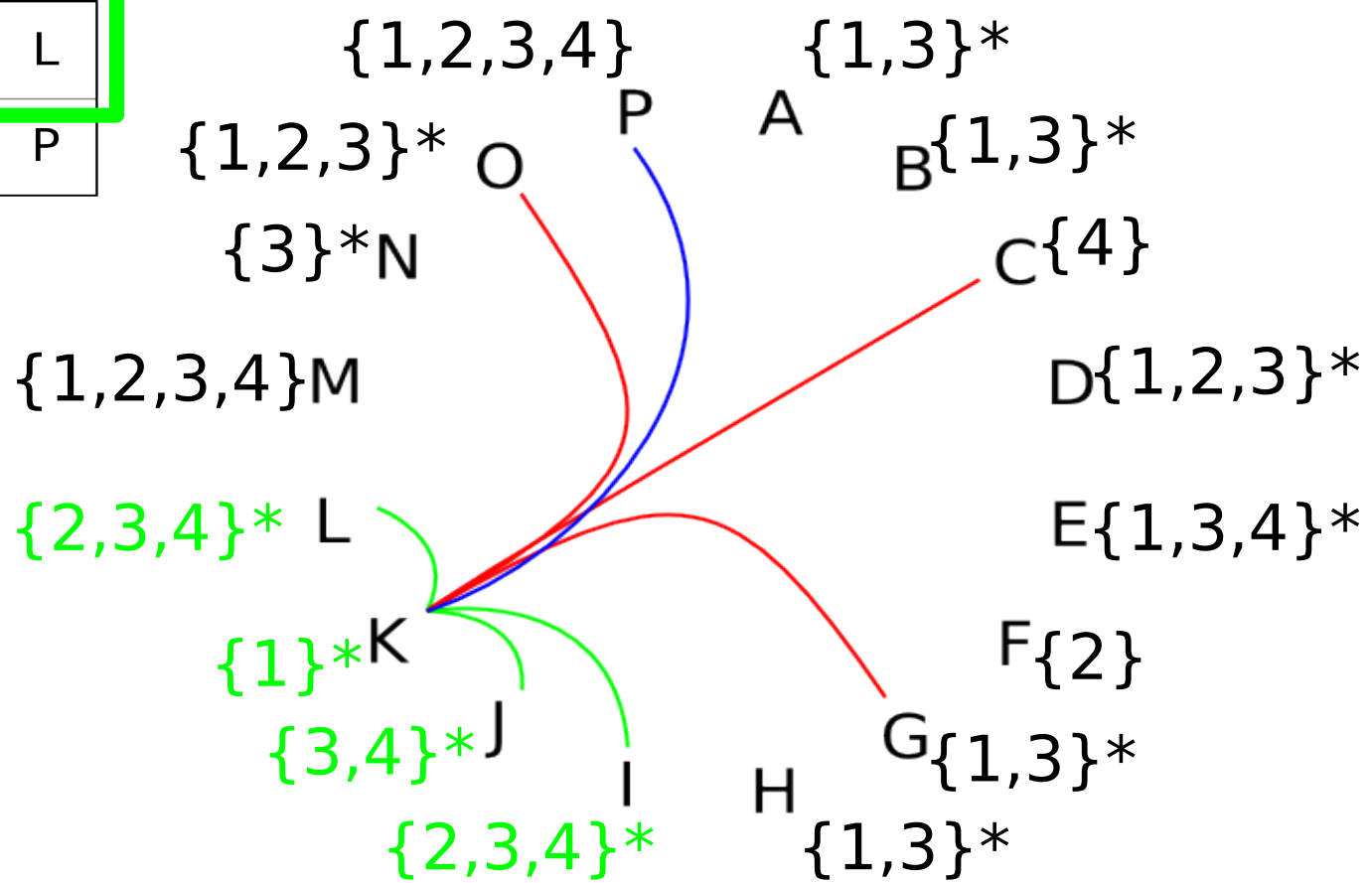
Update constraints connected to K

A	B	C	D
E	F	G	H
I	J	K	L
M	N	O	P

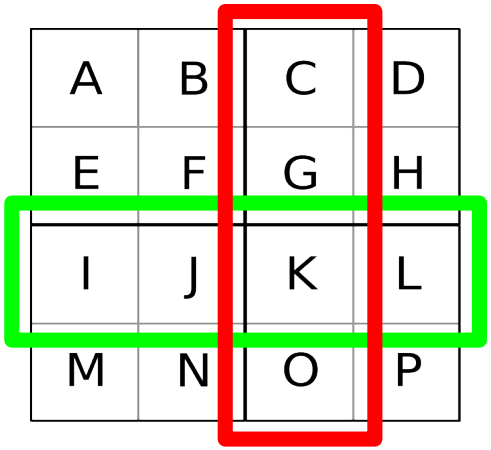


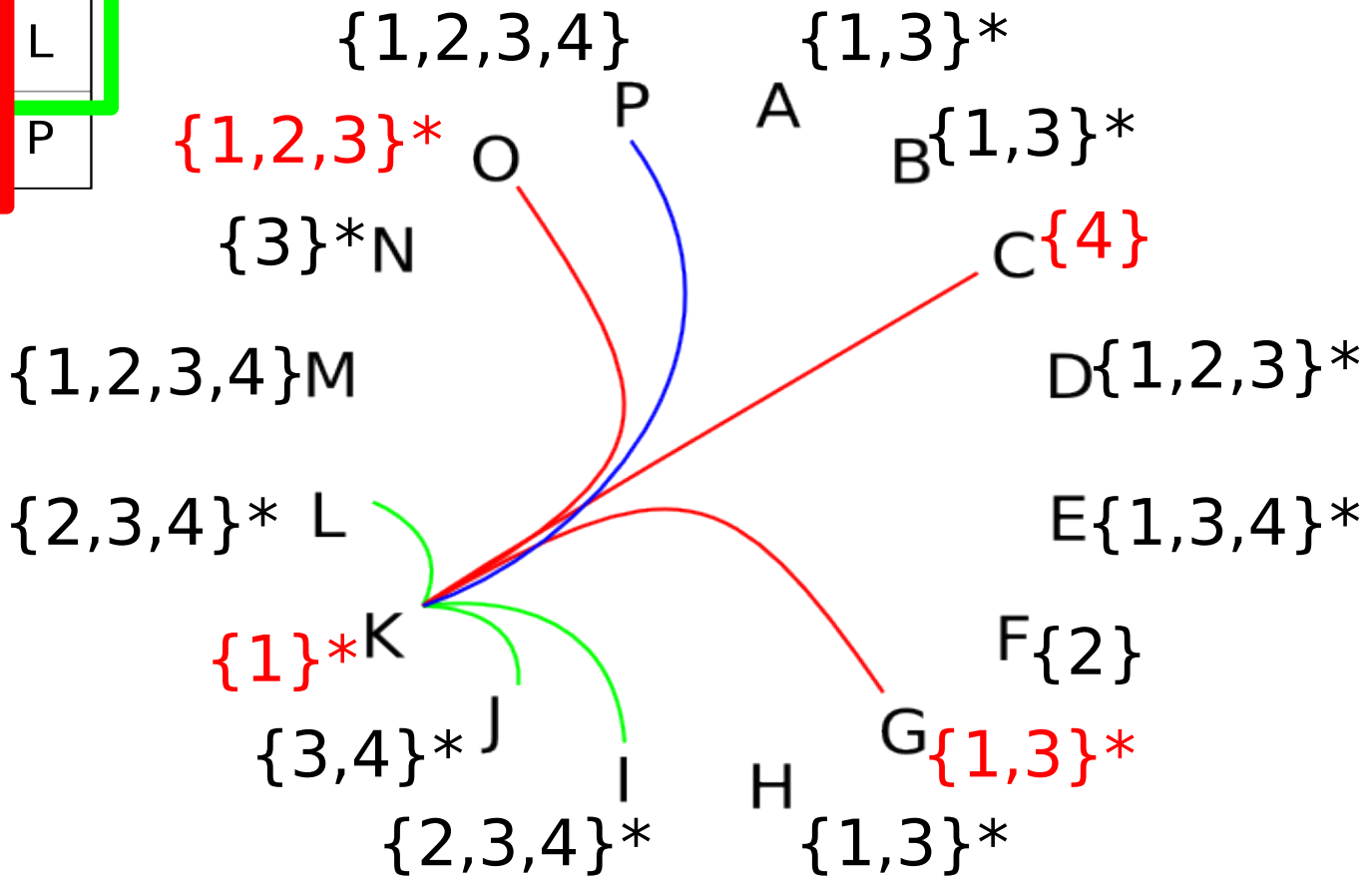
Update constraints connected to K

A	B	C	D
E	F	G	H
I	J	K	L
M	N	O	P

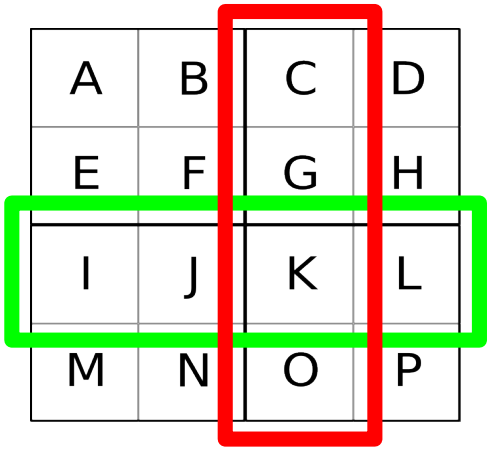


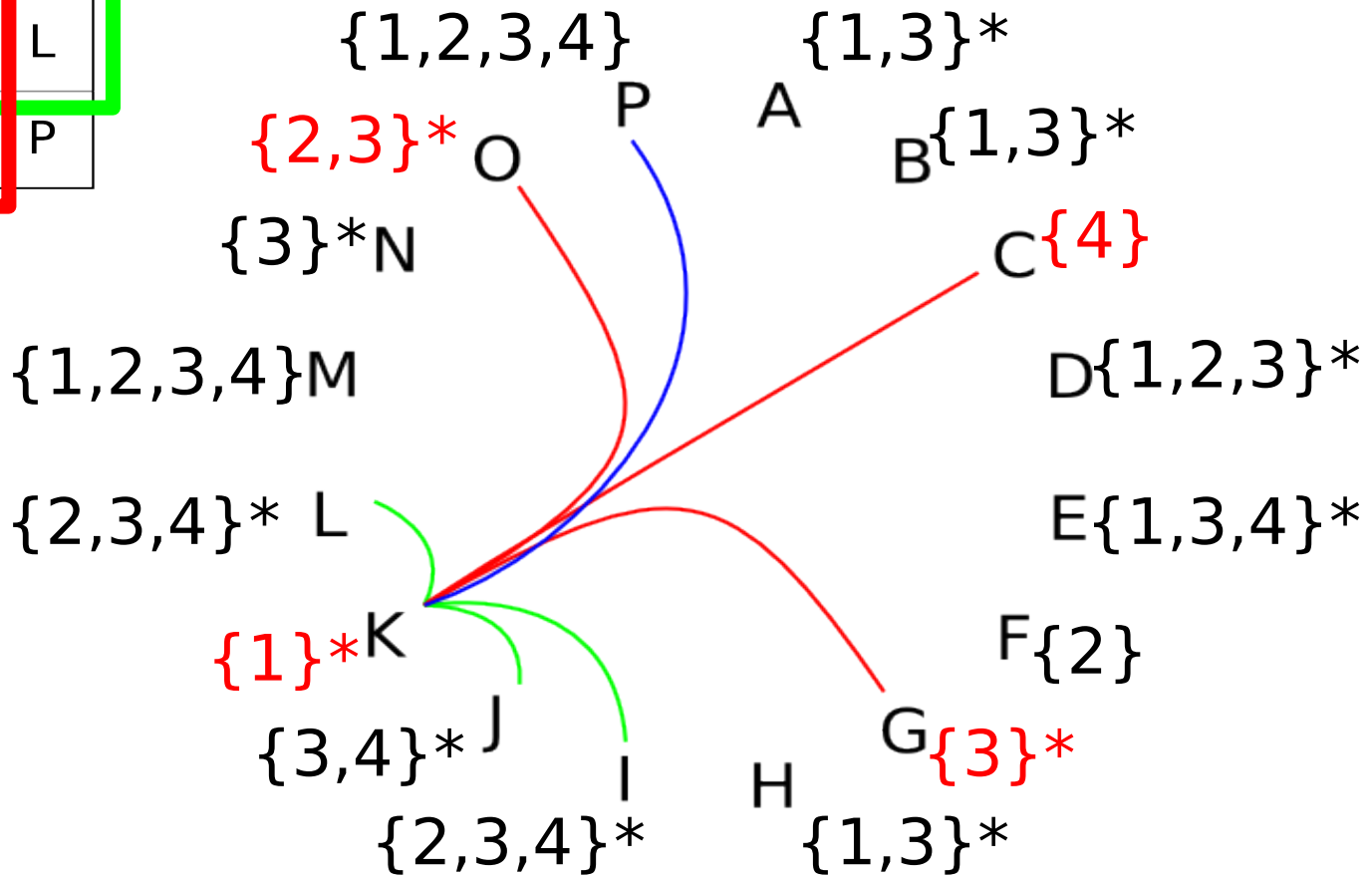
Update constraints connected to K



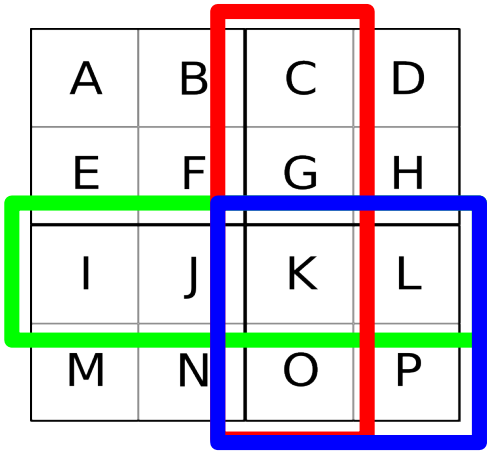


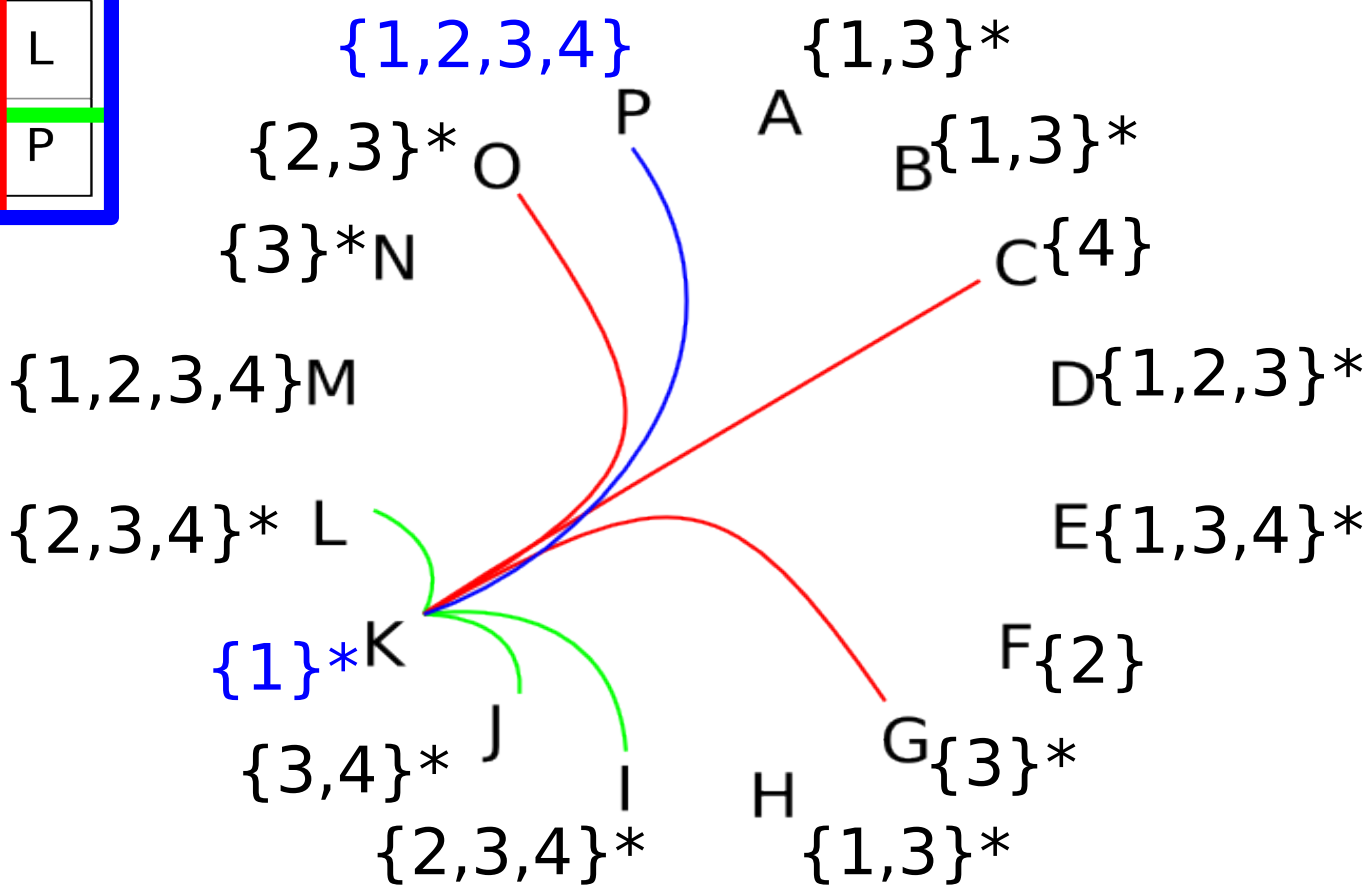
Update constraints connected to K



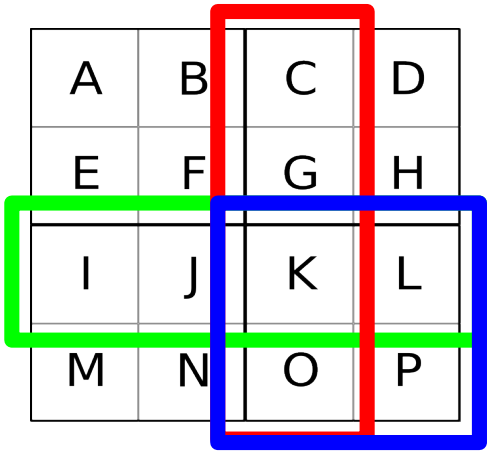


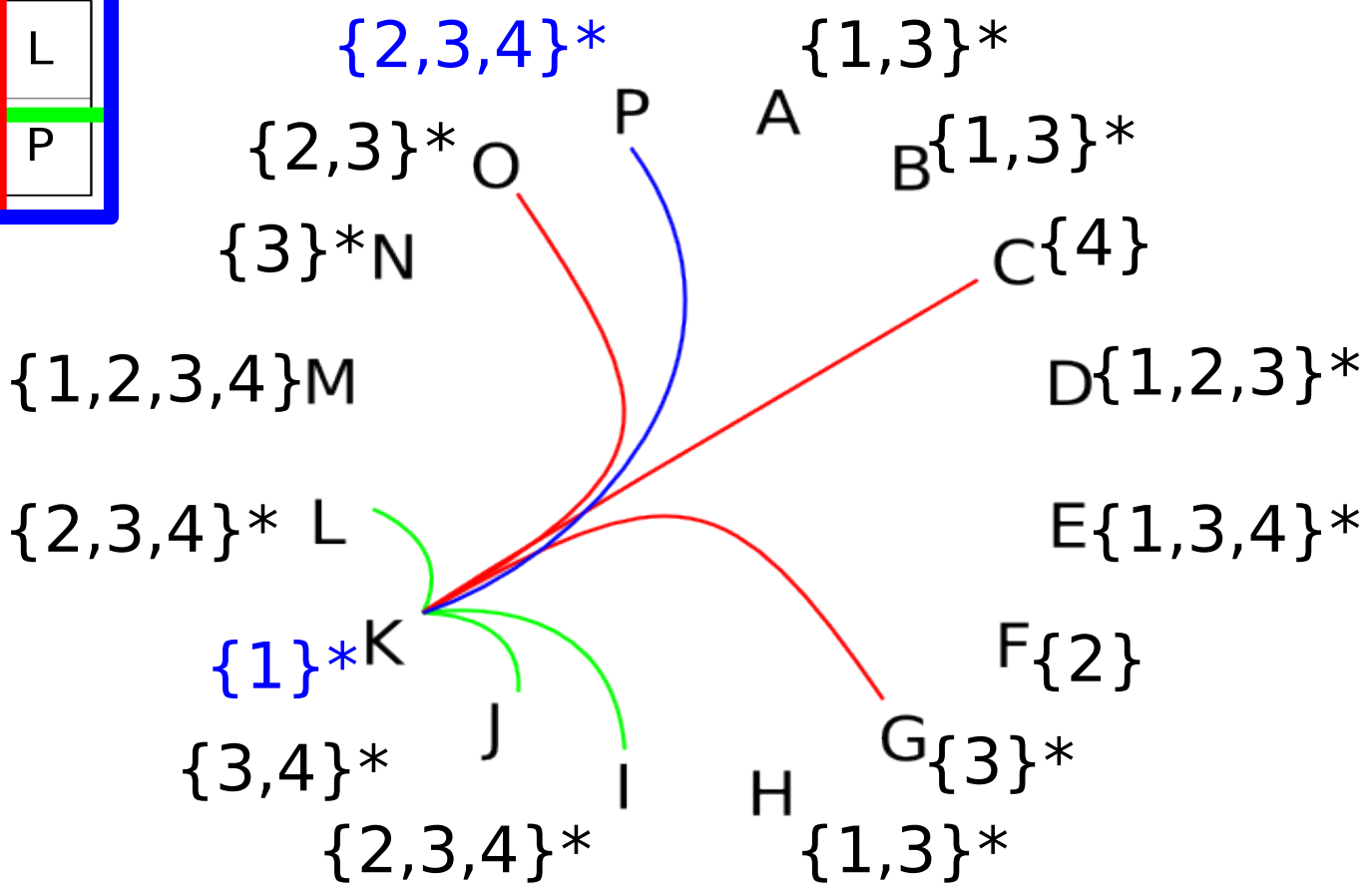
Update constraints connected to K



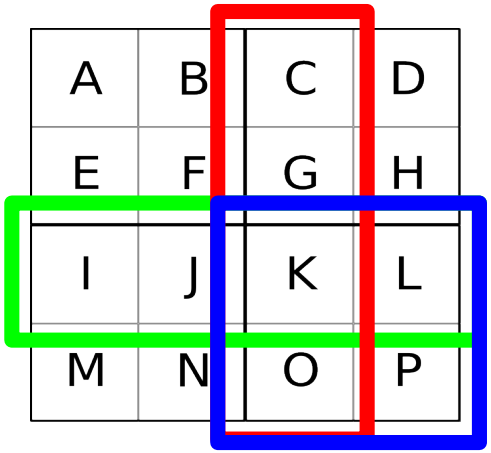


Update constraints connected to K

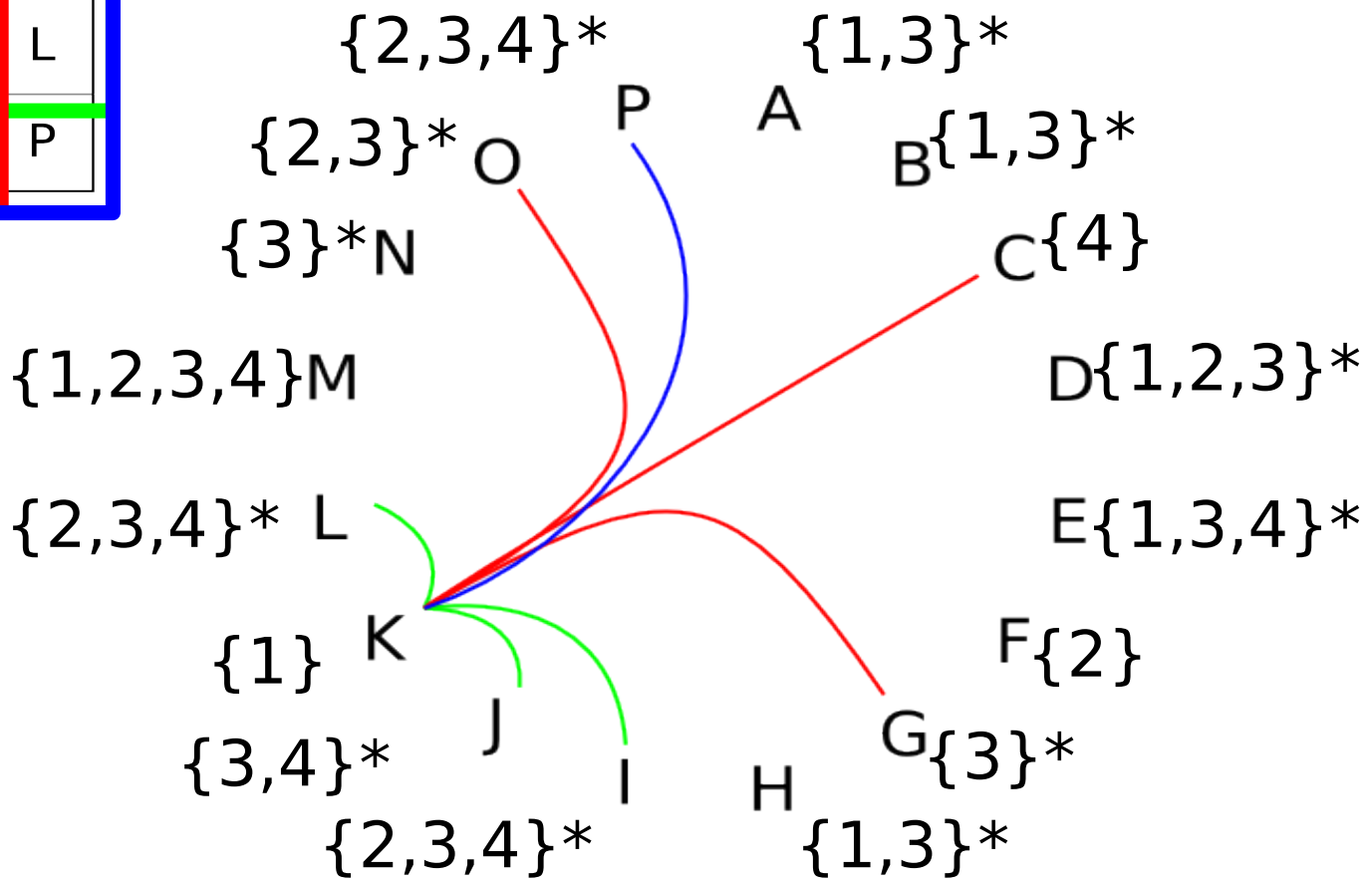




Update constraints connected to K

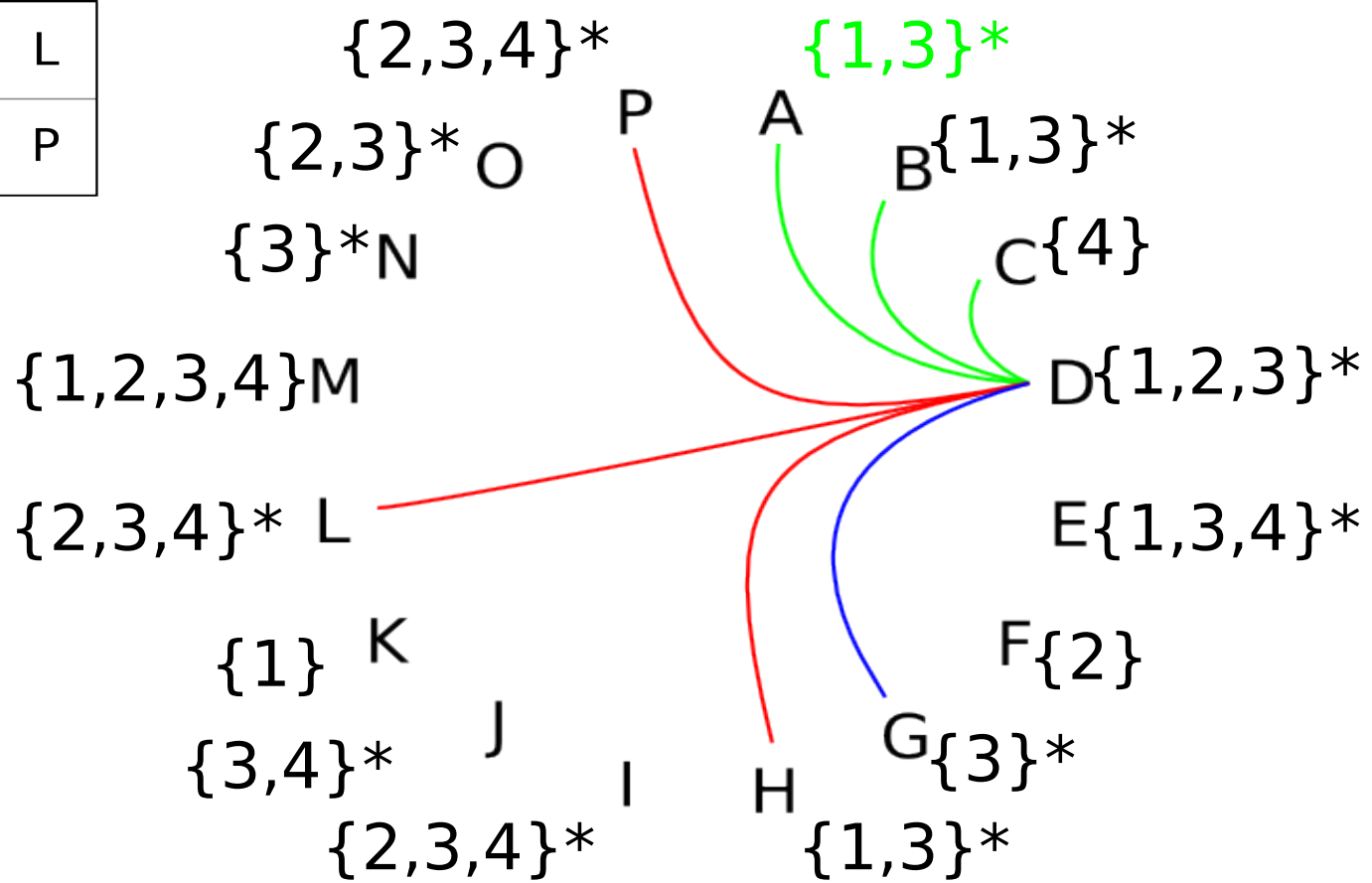


We have exhausted K's constraints for now



Update constraints connected to D

A	B	C	D
E	F	G	H
I	J	K	L
M	N	O	P

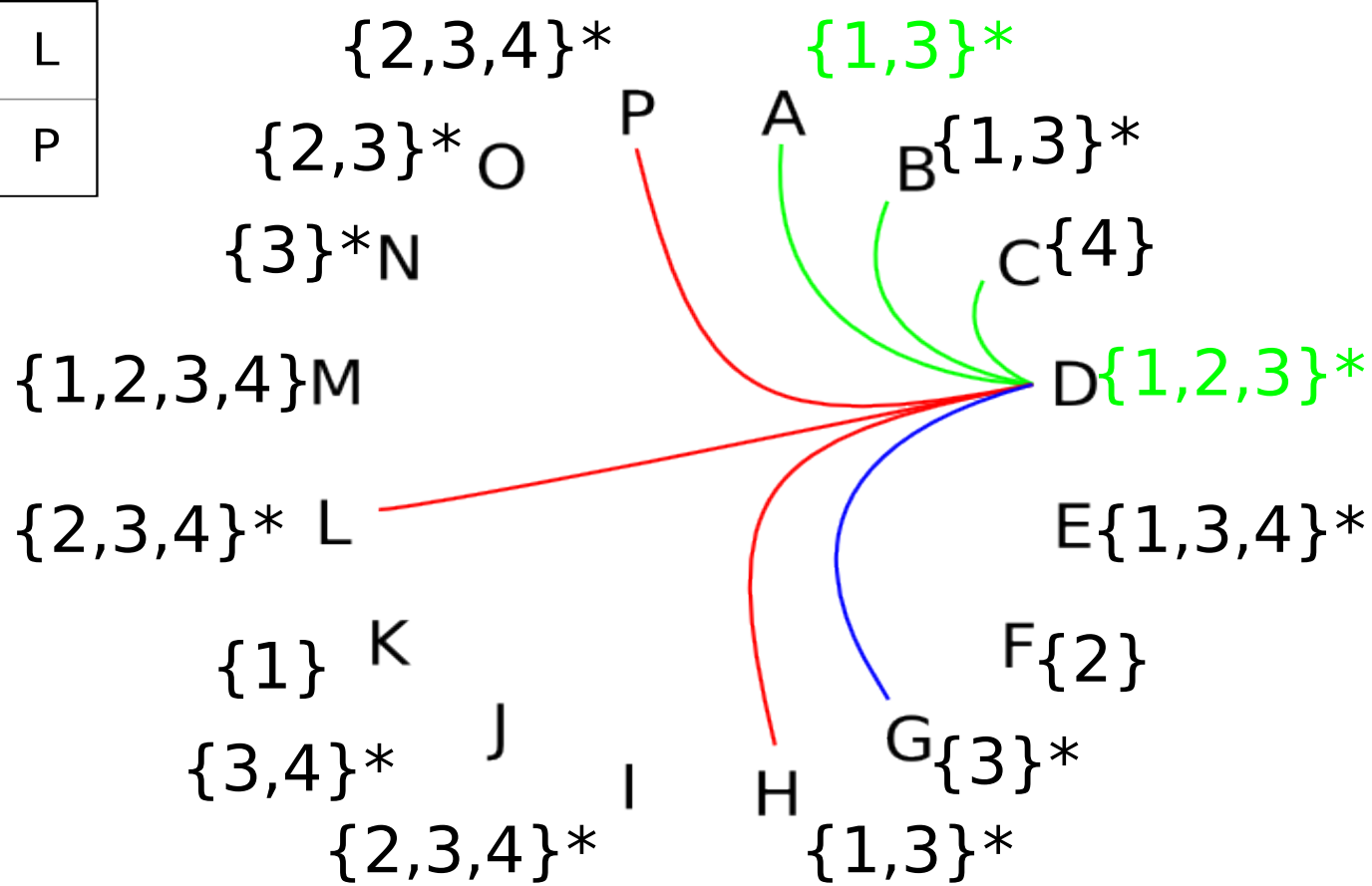


Update constraints connected to D

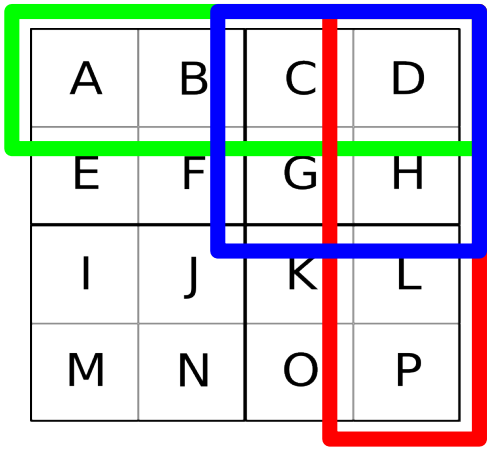
A	B	C	D
E	F	G	H
I	J	K	L
M	N	O	P

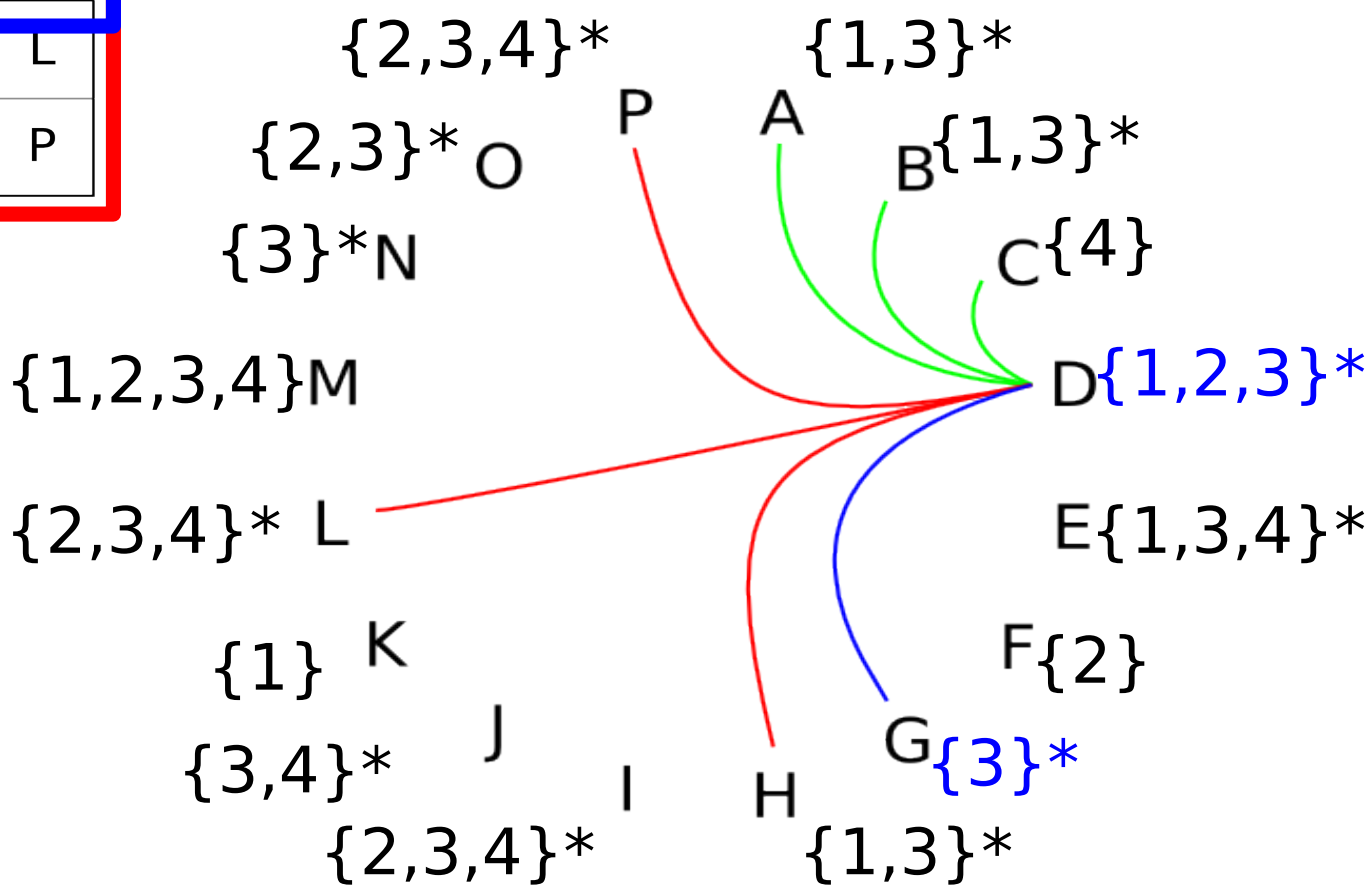
No values can be eliminated directly

(we should see the answer though!)

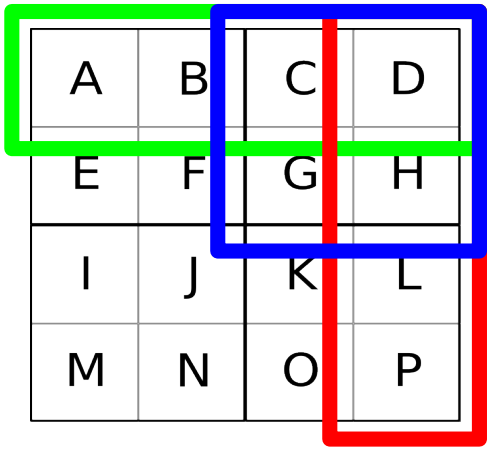


Update constraints connected to D





Change can occur in source domain

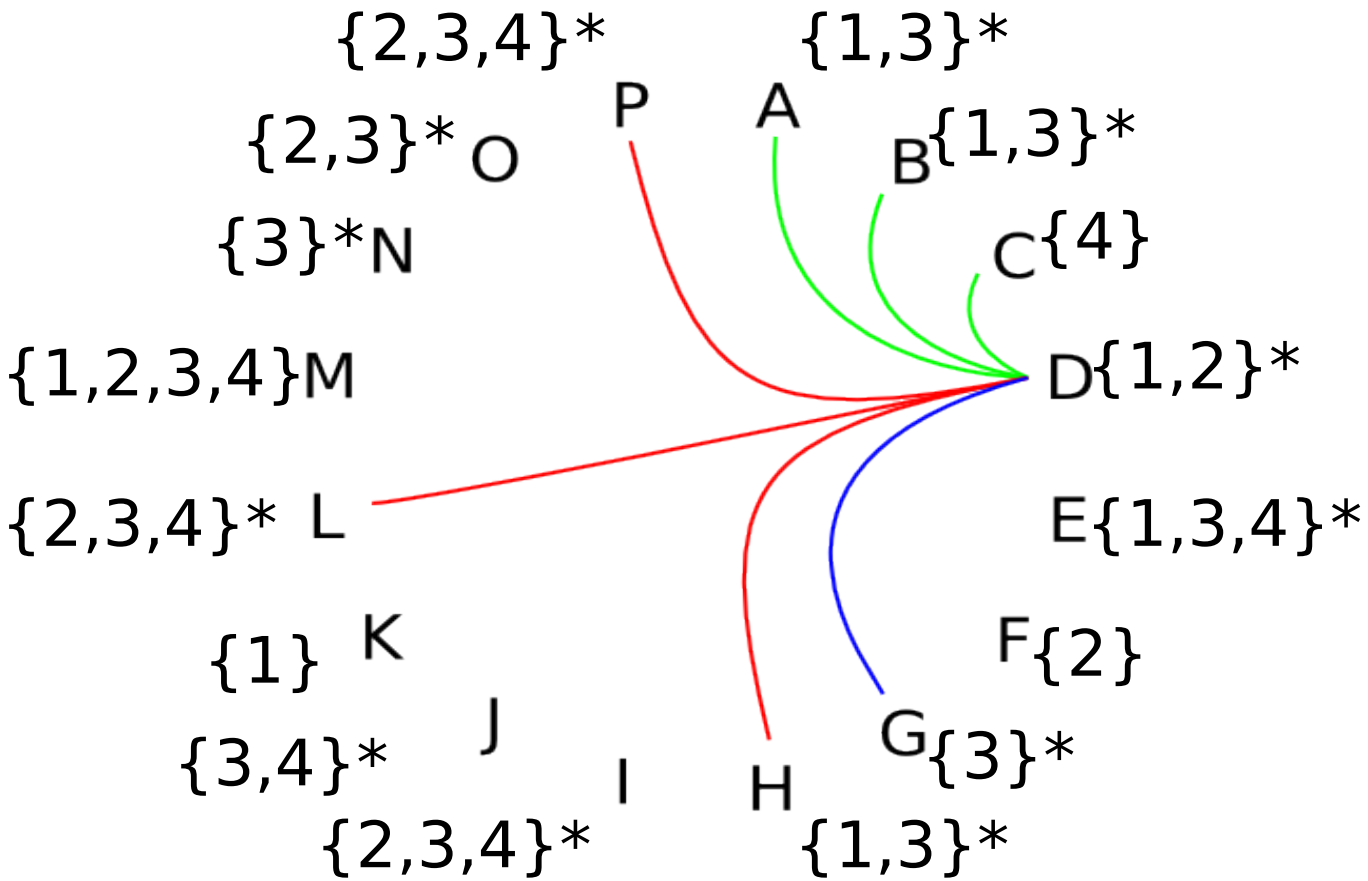


Single 3 at G eliminates D's 3

{2,3,4}* {1,3}*
P A
{2,3}* O B {1,3}*
{3}* N C {4}
{1,2,3,4} M D {1,2}*
{2,3,4}* L E {1,3,4}*
{1} K F {2}
{3,4}* J G {3}*
I H
{2,3,4}* {1,3}*

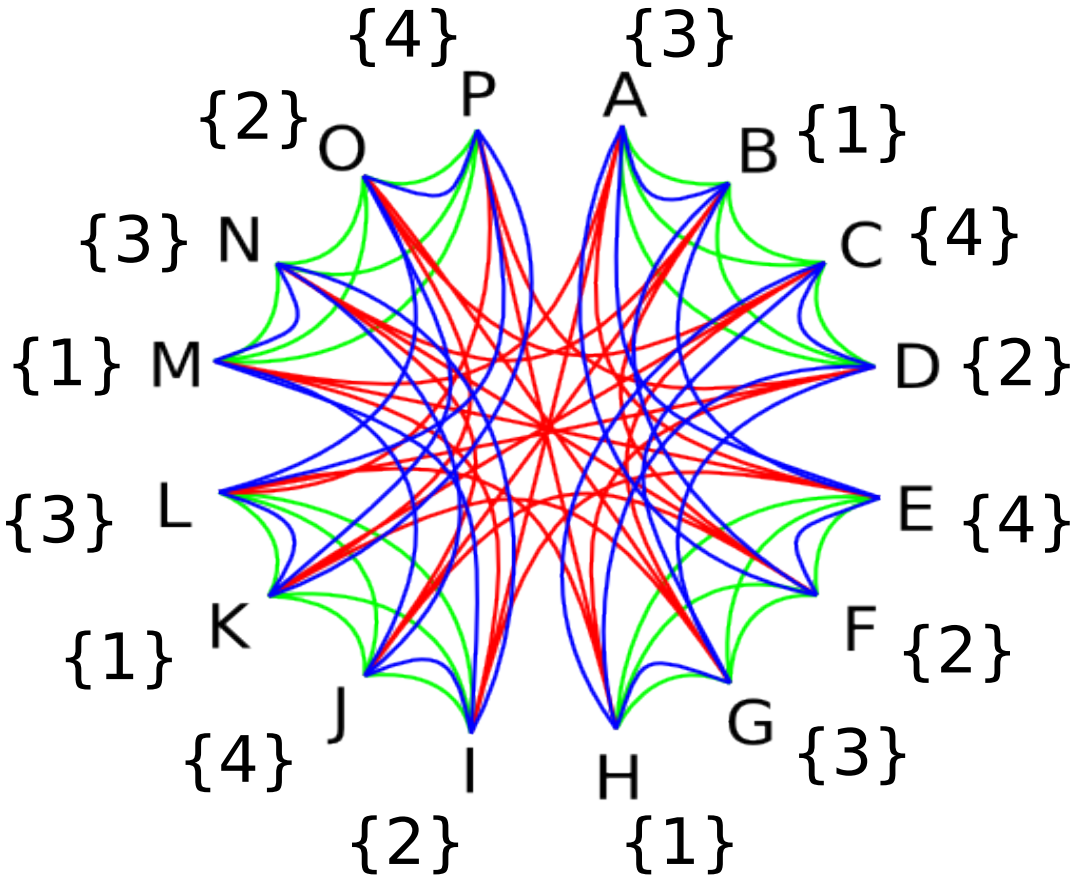
Change can occur in source domain

If the source domain changes we mark all its constraints for update again



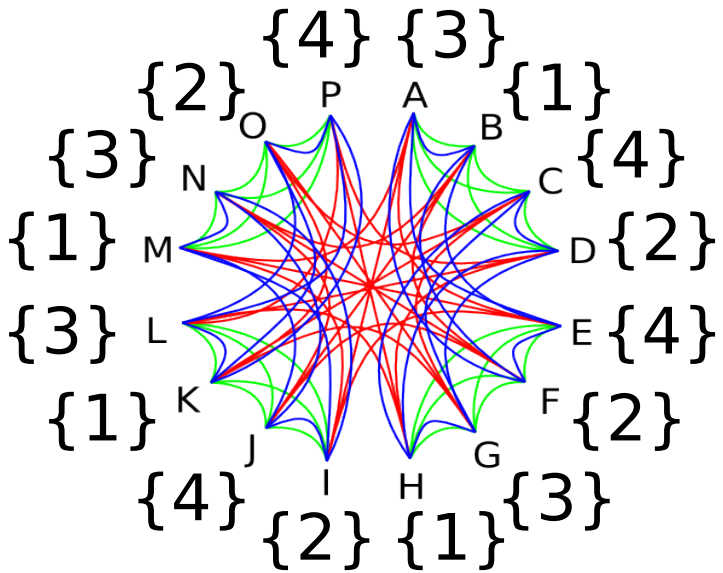
Iterate the algorithm to convergence (no further changes occur)

Why will the algorithm eventually converge?



Outcome 1: Single valued domains

We have found a unique solution to the problem



3	1	4	2
4	2	3	1
2	4	1	3
1	3	2	4

Outcome 2: Some empty domains

Our constraints are shown to be inconsistent

- therefore there is no solution to this problem

Variables

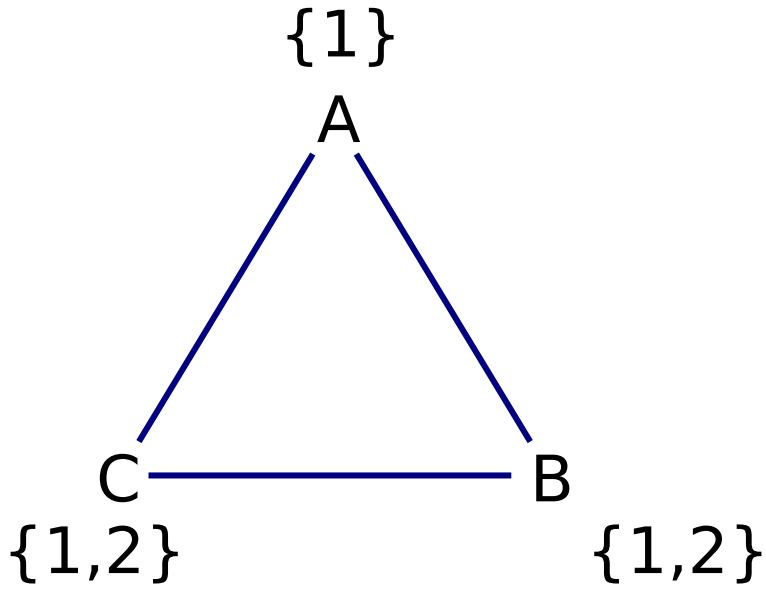
$A \in \{1\}$

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

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

Constraints

$A \neq B$, $A \neq C$, $B \neq C$



Outcome 2: Some empty domains

Our constraints are shown to be inconsistent

- therefore there is no solution to this problem

Variables

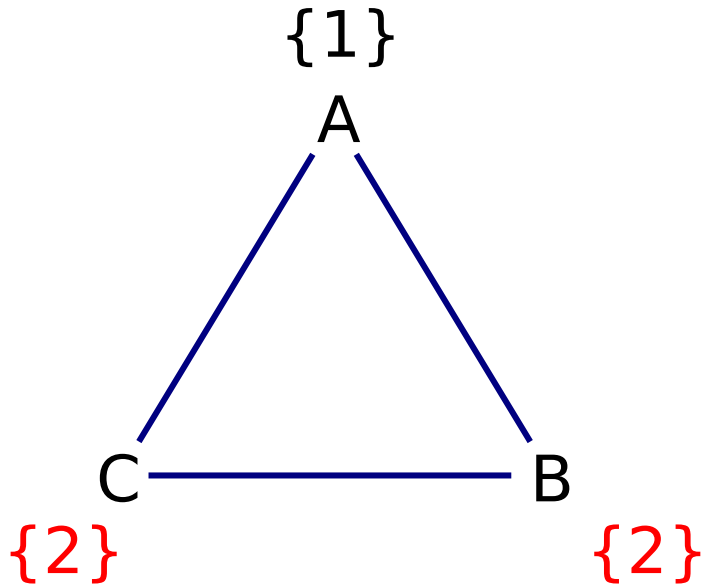
$A \in \{1\}$

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

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

Constraints

$A \neq B$, $A \neq C$, $B \neq C$



Outcome 2: Some empty domains

Our constraints are shown to be inconsistent

- therefore there is no solution to this problem

Variables

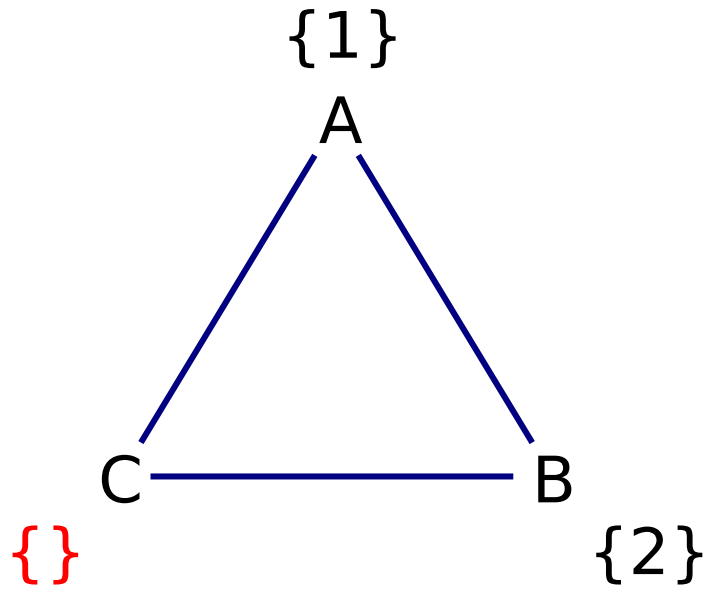
$A \in \{1\}$

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

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

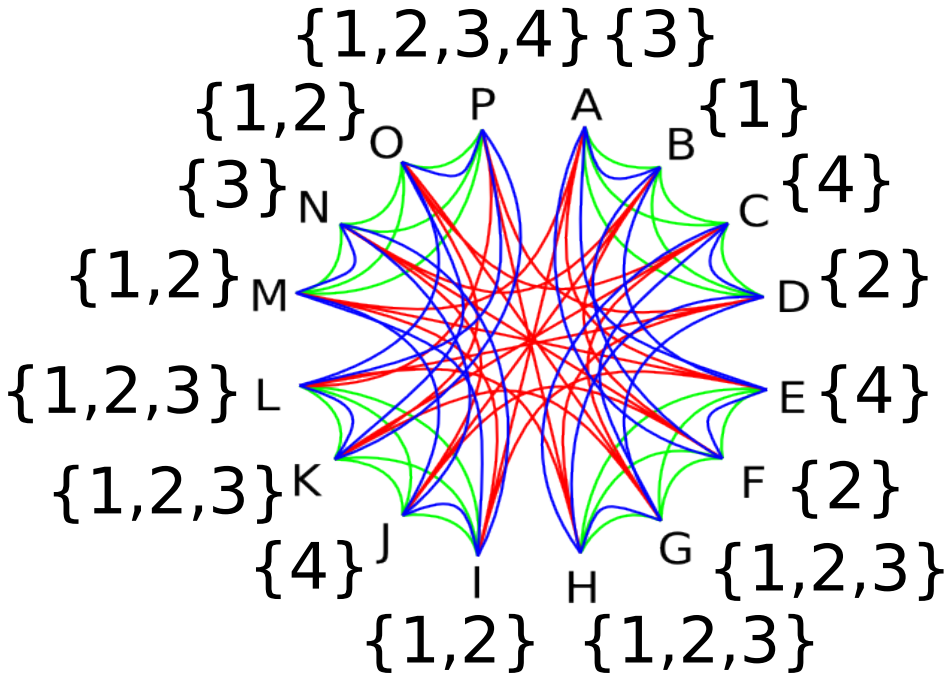
Constraints

$A \neq B$, $A \neq C$, $B \neq C$



Outcome 3: Some multivalued domains

		4	
	2		
	3		



Not all combinations of these possibilities of variable assignments are global solutions...

Outcome 3: Hypothesise labellings

To find global solutions from the narrowed domains we hypothesise a solution in a domain and propagate the changes

Backtrack if something goes wrong

Using CLP in Prolog

```
:- use_module(library(bounds)).

valid4(L) :- L in 1..4, all_different(L).

sudoku2([[X11,X12,X13,X14],[X21,X22,X23,X24],
         [X31,X32,X33,X34],[X41,X42,X43,X44]]) :-
  valid4([X11,X12,X13,X14]),valid4([X21,X22,X23,X24]),
  valid4([X31,X32,X33,X34]),valid4([X41,X42,X43,X44]),
  valid4([X11,X21,X31,X41]),valid4([X12,X22,X32,X42]),
  valid4([X13,X23,X33,X43]),valid4([X14,X24,X34,X44]),
  valid4([X11,X12,X21,X22]),valid4([X13,X14,X23,X24]),
  valid4([X31,X32,X41,X42]),valid4([X33,X34,X43,X44]),
  labeling([],[X11,X12,X13,X14,X21,X22,X23,X24,
               X31,X32,X33,X34,X41,X42,X43,X44]).
```

Rows

Cols

Boxes

Chapter 14 of the textbook has more information

Prolog can be used for parsing context-free grammars (p555)

Here is a simple grammar:

s → 'a' 'b'
s → 'a' 'c'
s → s s

Terminal symbols:

a, b, c

Non-terminal symbols:

s

Parsing by consumption

Write a predicate for each non-terminal that:

- consumes as much of the first list as is necessary to match the non-terminal, and
- returns the remaining elements in the second list

These predicate evaluations will thus be true:

- s([a,b],[])
- s([a,b,c,d],[c,d])

A Prolog program that accepts sentences from our grammar

```
% match a single character
c([X|T],X,T).

% grammar predicates
s(In,Out) :- c(In,a,In2),
             c(In2,b,Out).
s(In,Out) :- c(In,a,In2),
             c(In2,c,Out).
s(In,Out) :- s(In,In2),
             s(In2,Out).
```

```
s → 'a' 'b'
s → 'a' 'c'
s → s s
```

Prolog DCG syntax

Prolog provides us with a shortcut for encoding Definite Clause Grammar (DCG) syntax.

```
s --> [a],[b].
s --> [a],[c].
s --> s,s.
```

```
s → 'a' 'b'
s → 'a' 'c'
s → s s
```

This will both test and generate:

- s([a,c,a,b],[]).
- s(A,[]).

Building a parse tree

```
% match a single character
c([X|T],X,T).

% grammar predicates
s(ab,In,Out) :- c(In,a,In2),
                c(In2,b,Out).
s(ac,In,Out) :- c(In,a,In2),
                c(In2,c,Out).
s(t(A,B),In,Out) :- s(A,In,In2),
                    s(B,In2,Out).

:- s(Result,[a,c,a,b,a,b],[]).
```

Prolog's DCG syntax helps us again

Unfortunately the DCG syntax is not part of the ISO Prolog standard

- Almost all modern compilers will include it though

```
s(ab) --> [a],[b].
s(ac) --> [a],[c].
s(t(A,B)) --> s(A),s(B).
```

Parsing Natural Language (back to Prolog's roots)

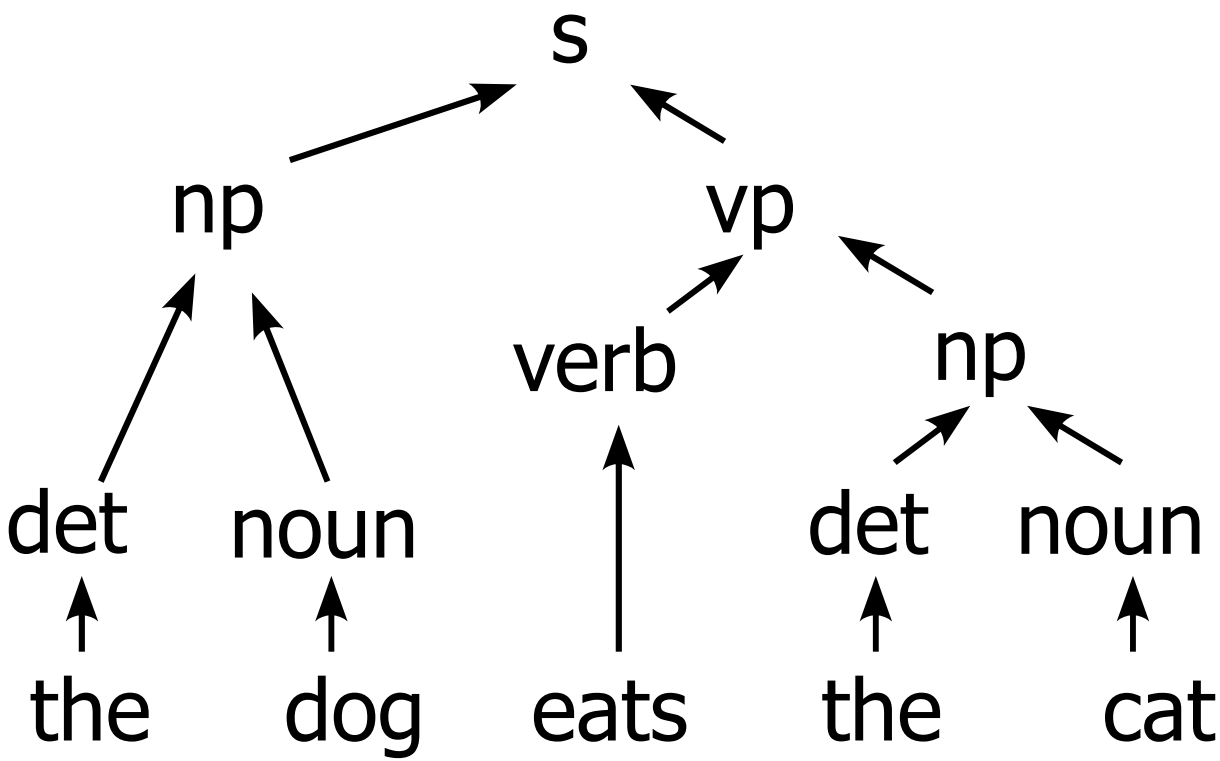
```
s    --> np,vp.
np   --> det,n.
vp   --> v.
vp   --> v,np.

n    --> [cat].
n    --> [dog].
v    --> [eats].
det  --> [the].
```

This is a very limited English grammar subset.

Things get complicated very quickly!

- see the Natural Language Processing course next year (Prolog is not a pre-requisite)

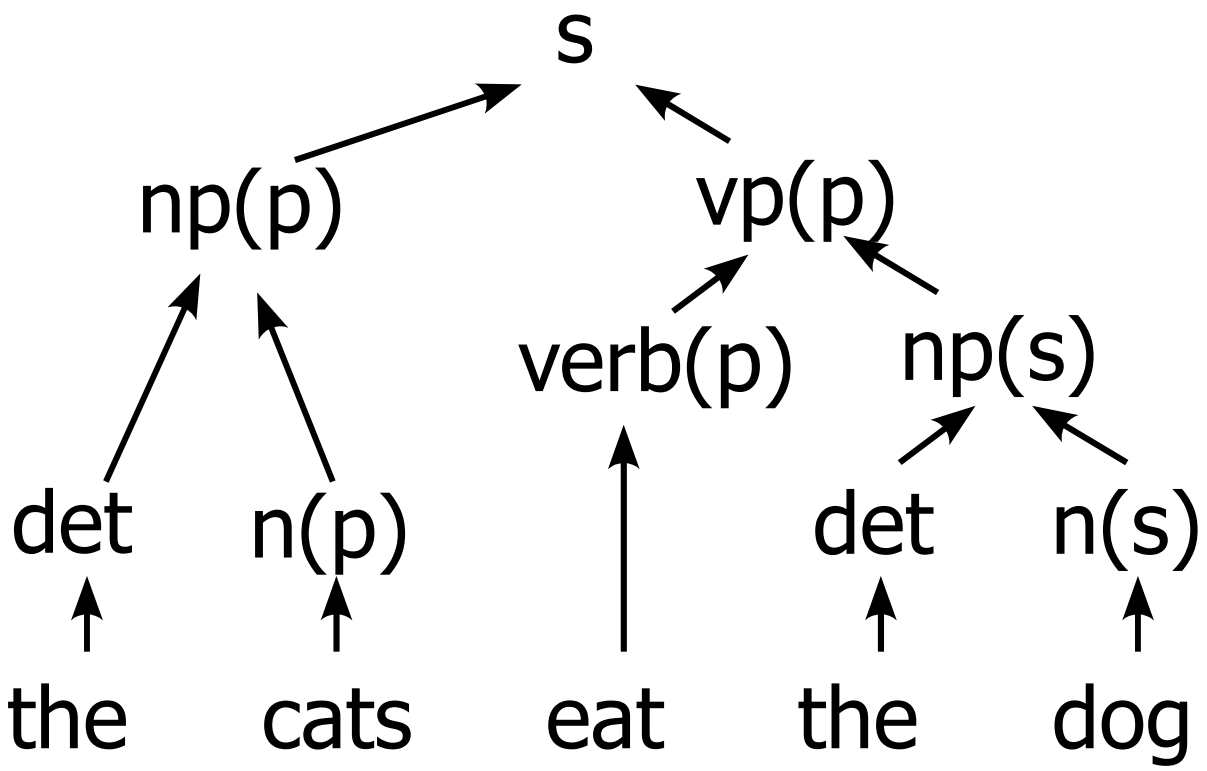

s
np
vp
verb
np
det
noun
det
noun
the
dog
eats
the
cat

We can also handle agreement

```
s(N)   --> np(N),vp(N).
np(N)  --> det,n(N).
vp(N)  --> v(N).
vp(N)  --> v(N),np(_).

n(s)   --> [cat].
n(s)   --> [dog].
n(p)   --> [cats].
v(s)   --> [eats].
v(p)   --> [eat].
det    --> [the].
```

We consider only third-person constructions here!


s
np(p)
vp(p)
verb(p)
np(s)
det
n(p)
det
n(s)
the
cats
eat
the
dog

Real Natural Language Processing

Things get much more complicated very quickly

Ambiguities, special cases and noise all make the approach we have demonstrated hard to scale

- Although people have definitely tried!

Prolog has lasting influence

Languages that have been influenced by Prolog:

- Mercury
  - Compiled language that takes advantage of knowledge about predicate determinism
- Erlang
  - Massively concurrent programming

Projects

- Overlog
  - Declarative networking
- XSB
  - Declarative database: tabled resolution, HiLog

Closing Remarks

Declarative programming is different to Functional or Procedural programming

- Foundations of Computer Science & Programming in Java

Prolog is built on logical deduction

- formal explanation in Logic & Proof

It can provide concise implementations of algorithms such as sorting or graph search

- Algorithms I & Algorithms II

Closing Remarks

Foundations of Functional Programming (Part IB)

- Building computation from first principles

Databases (Part 1B)

- Find out more about representing data and SQL

Artificial Intelligence (Part 1B)

- Search, constraint programming and more

C & C++ (Part 1B)

- Doing useful stuff in the real world

Natural Language Processing (Part II)

- Parsing natural language

End

C

You shoot yourself in the foot.

C++

You accidentally create a dozen instances of yourself and shoot them all in the foot. Providing emergency medical care is impossible since you can't tell which are bitwise copies and which are just pointing at others and saying, "That's me over there."

Prolog

You explain in your program that you want to be shot in the foot. The interpreter figures out all the possible ways to do it, but then backtracks completely, instead destroying the gun.