# Symmetry Detection and Exploitation in Constraint Programming

Chris Mears

June, 2008

# Constraint Programming

- What is constraint programming?

# Constraint Programming

- What is constraint programming?
- Programming with constraints!

# Constraint Programming

- For solving combinatorial problems.
- Problems are specified with *constraints*.
  - E.g., only one class per room.
- For both:
  - satisfaction (can it be done?),
  - optimisation (what's the best way?).
- Focus on finite domain problems.

# Example: Latin Square

A CSP is a triple $< X, D, C >$

- $X$ is a set of variables
- $D$ is a set of values
- $C$ is a set of constraints

# Example: Latin Square

| | | |
|---|---|---|
| 1 | 2 | 3 |
| 3 | 1 | 2 |
| 2 | 3 | 1 |

A CSP is a triple $< X, D, C >$

- $X$ is a set of variables
- $D$ is a set of values
- $C$ is a set of constraints

$$
\begin{aligned}
X &= \{x_{11}, x_{12}, x_{13}, x_{21}, x_{22}, x_{23}, x_{31}, x_{32}, x_{33}\} \\
D &= \{1, 2, 3\} \\
C &= \{x_{11} \neq x_{12}, x_{11} \neq x_{13}, x_{12} \neq x_{13}, x_{21} \neq x_{22}, x_{21} \neq x_{23}, x_{22} \neq x_{23}, \\
&\quad x_{31} \neq x_{32}, x_{31} \neq x_{33}, x_{32} \neq x_{33}, x_{11} \neq x_{21}, x_{11} \neq x_{31}, x_{21} \neq x_{31}, \\
&\quad x_{12} \neq x_{22}, x_{12} \neq x_{32}, x_{22} \neq x_{32}, x_{13} \neq x_{23}, x_{13} \neq x_{33}, x_{23} \neq x_{33}\}
\end{aligned}
$$

# Example: Latin Square

| | | |
|---|---|---|
| $x11$ | $x12$ | $x13$ |
| $x21$ | $x22$ | $x23$ |
| $x31$ | $x32$ | $x33$ |

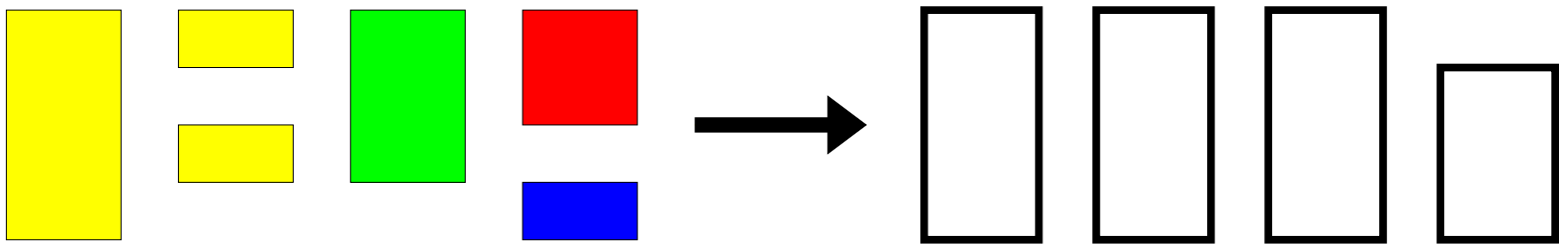
A CSP is a triple $< X, D, C >$

- $X$ is a set of variables
- $D$ is a set of values
- $C$ is a set of constraints

$$
\begin{aligned}
X &= \{x_{11}, x_{12}, x_{13}, x_{21}, x_{22}, x_{23}, x_{31}, x_{32}, x_{33}\} \\
D &= \{1, 2, 3\} \\
C &= \{x_{11} \neq x_{12}, x_{11} \neq x_{13}, x_{12} \neq x_{13}, x_{21} \neq x_{22}, x_{21} \neq x_{23}, x_{22} \neq x_{23}, \\
&\quad x_{31} \neq x_{32}, x_{31} \neq x_{33}, x_{32} \neq x_{33}, x_{11} \neq x_{21}, x_{11} \neq x_{31}, x_{21} \neq x_{31}, \\
&\quad x_{12} \neq x_{22}, x_{12} \neq x_{32}, x_{22} \neq x_{32}, x_{13} \neq x_{23}, x_{13} \neq x_{33}, x_{23} \neq x_{33}\}
\end{aligned}
$$

# Another Example: Steel Mill Slab Design

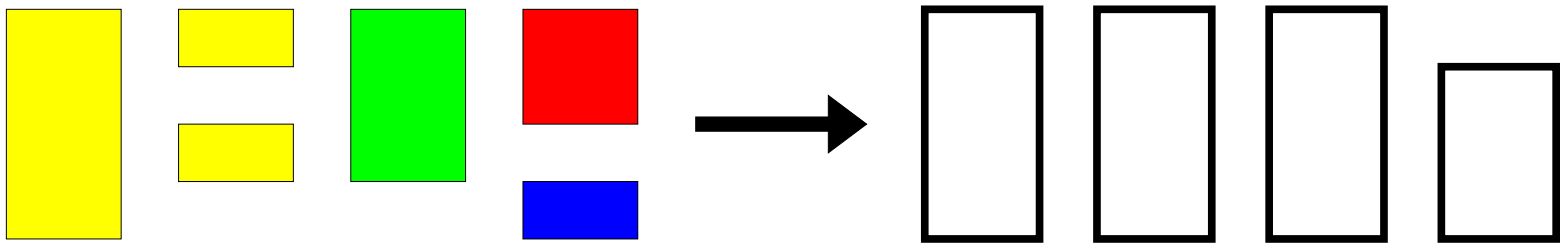
- Put orders in slabs.
- At most two colours per slab.
- Minimise sum of slab sizes.



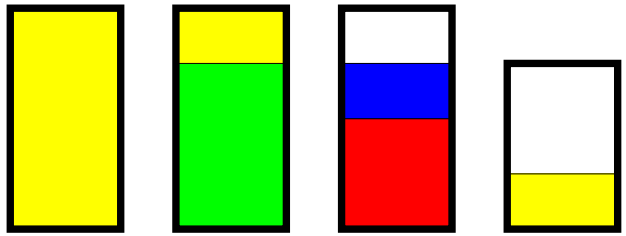
Orders Slabs

# Another Example: Steel Mill Slab Design

- Put orders in slabs.
- At most two colours per slab.
- Minimise sum of slab sizes.

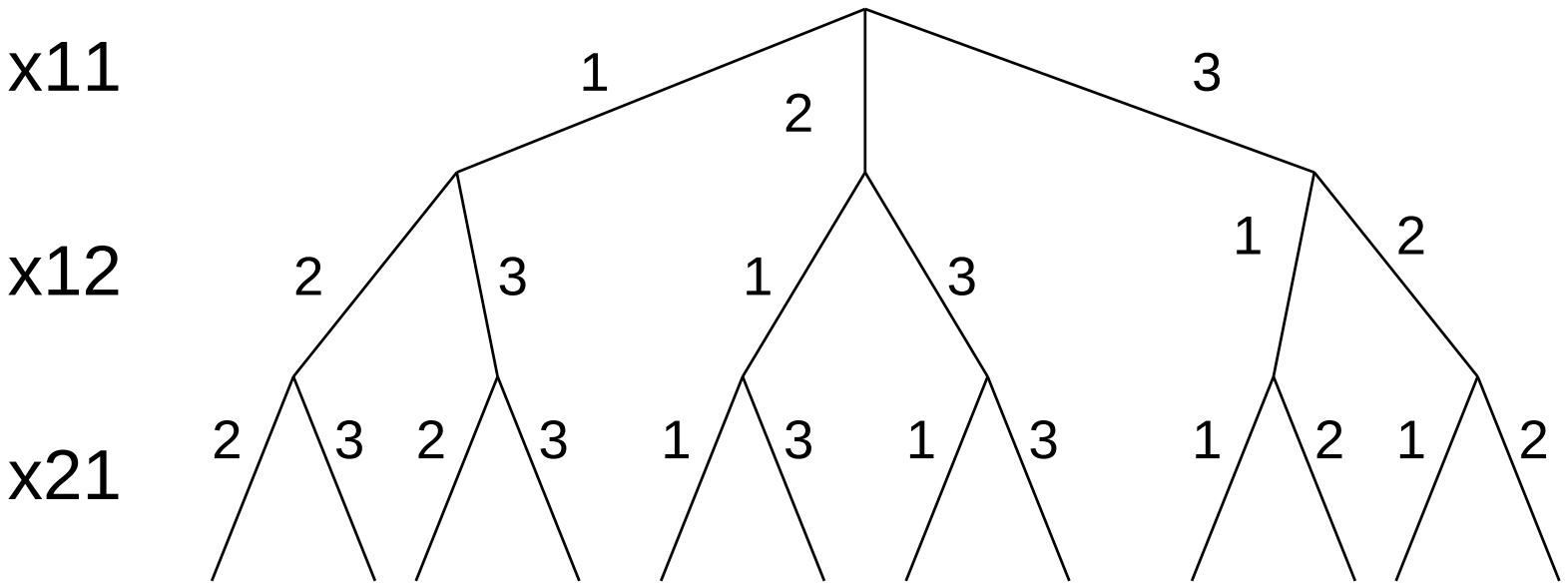


Orders Slabs



# Tree Search

- Constraint problems are usually solved by some form of search.
- E.g., backtracking search.

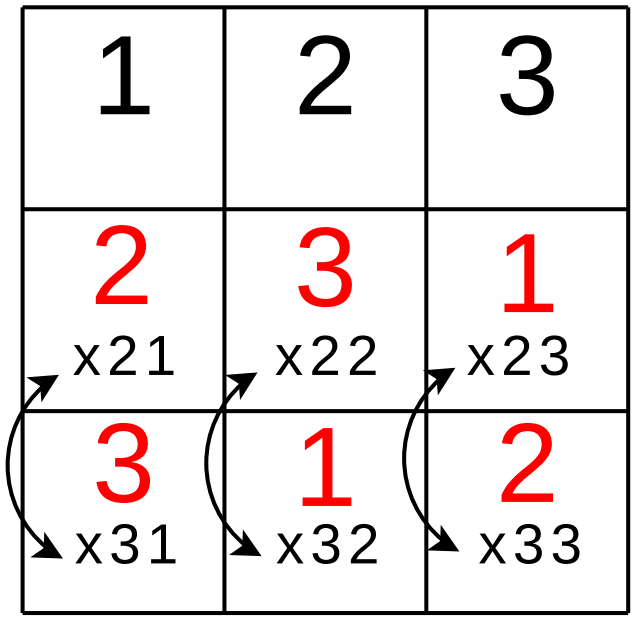


# Symmetry Example: Latin Square

| | | |
|---|---|---|
| 1 | 2 | 3 |
| 3 | 1 | 2 |
| 2 | 3 | 1 |

- A symmetry is a permutation of the problem that doesn't affect solutions.
- E.g. Any two rows can be swapped.

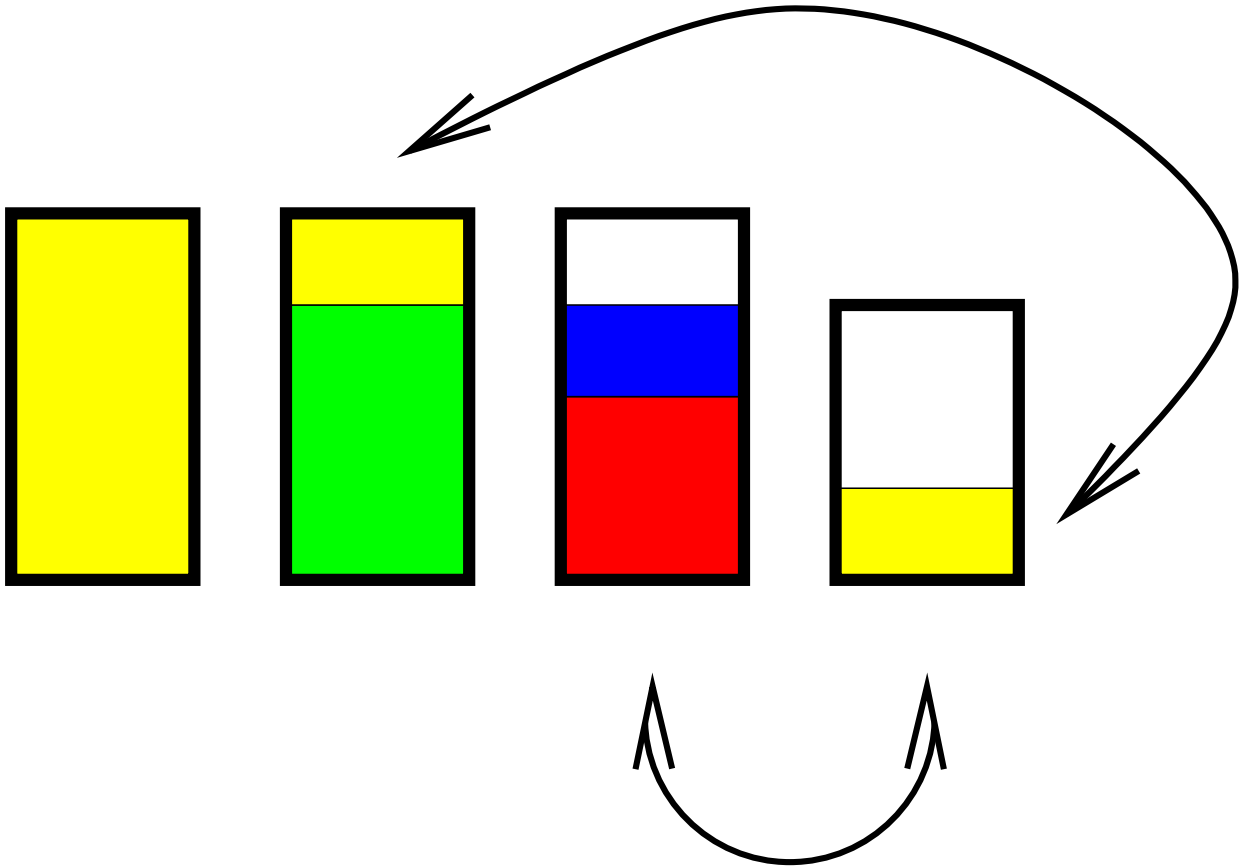
# Symmetry Example: Latin Square




- A symmetry is a permutation of the problem that doesn't affect solutions.
- E.g. Any two rows can be swapped.

# Symmetry Example: Steel Mill Slab Design

- Slab weights can be permuted.
- Identical orders can be permuted.



# Symmetry: a Definition

## Definition

A symmetry is a permutation of literals (variable-value pairs) that maps solutions to solutions (and therefore non-solutions to non-solutions).

## Kinds of Symmetry

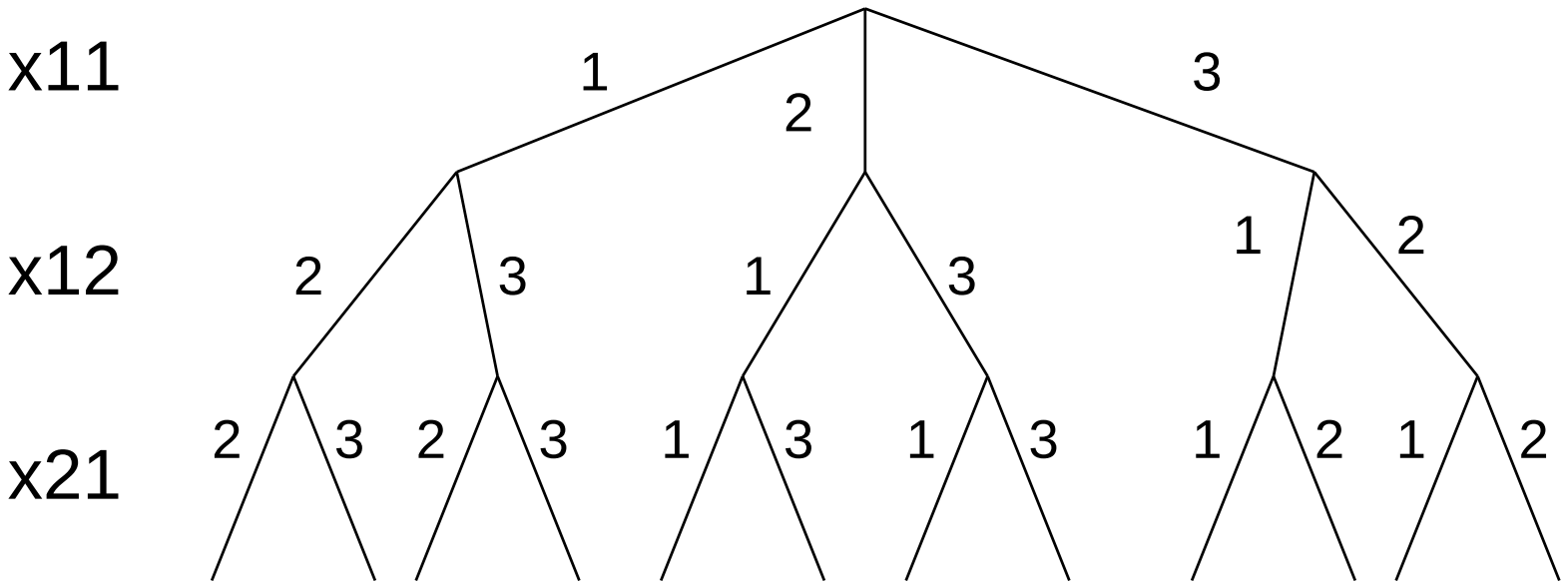
- Variable symmetries (permutation of variables)
- Value symmetries (permutation of values)
- Variable-value symmetries (permutation of literals)

# Symmetries: Who cares?

Interesting property, but who cares?

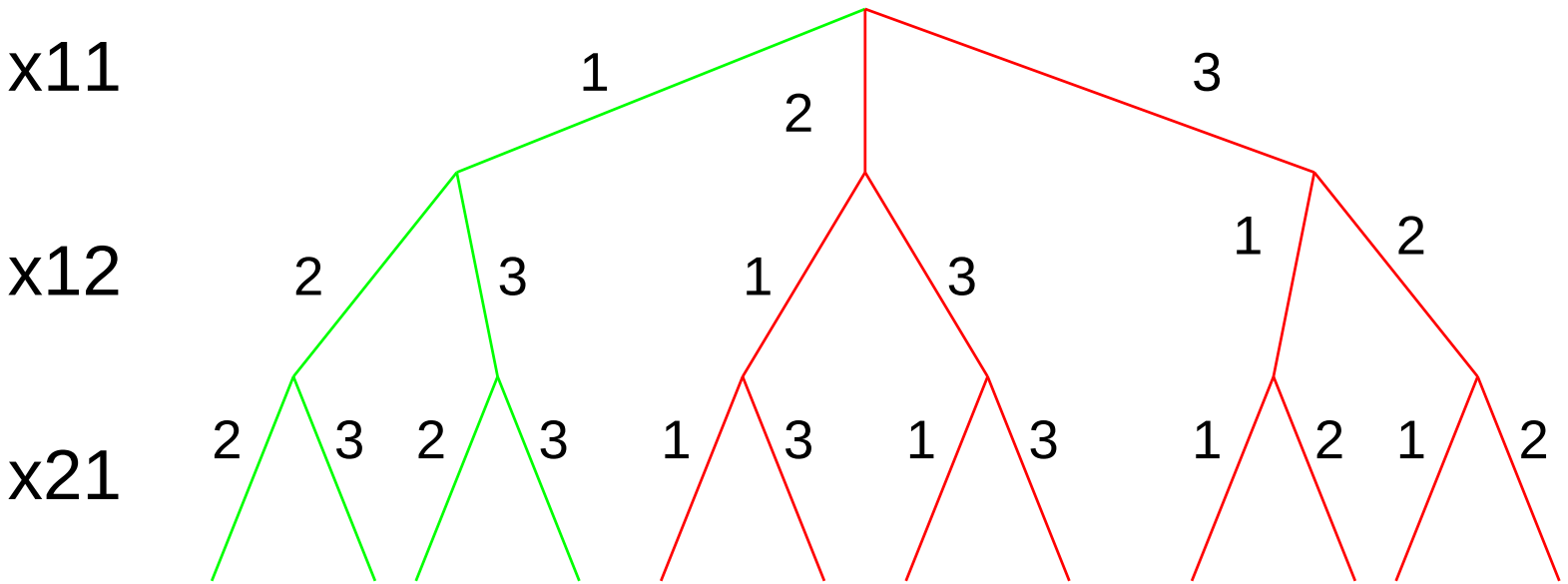
Symmetries can be used to improve search.

# Search



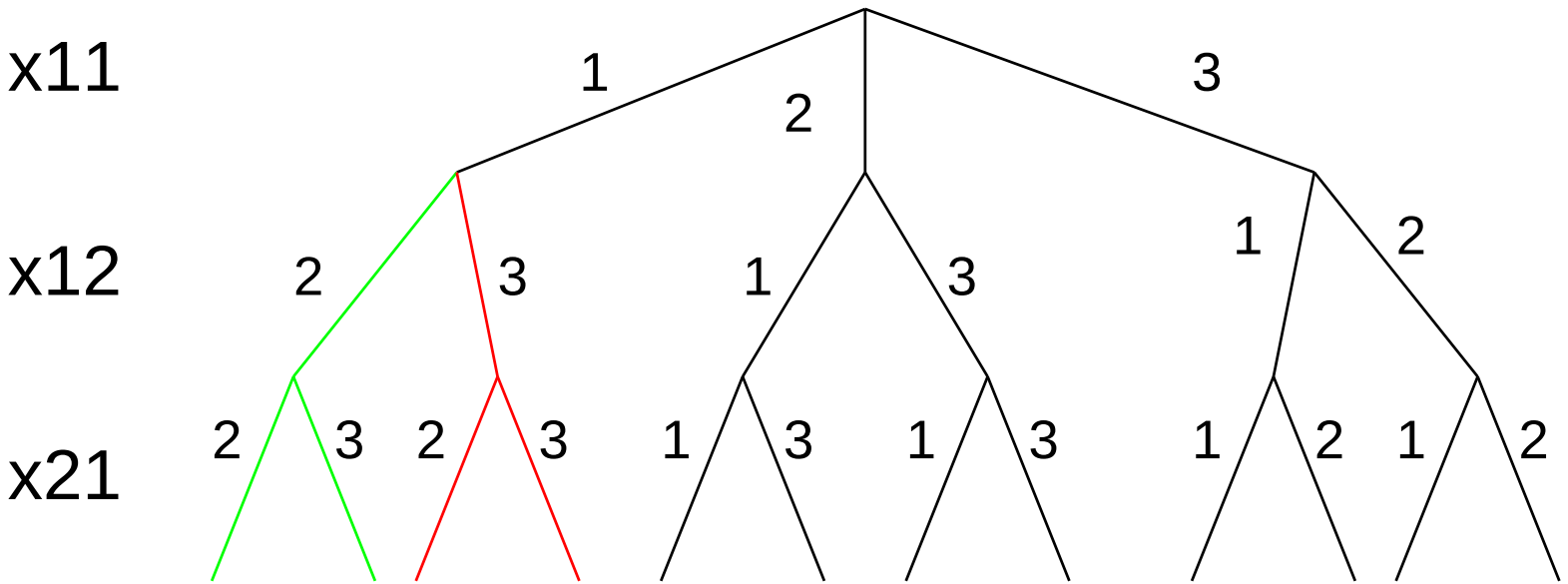
# Symmetries in Search

- Symmetries in the problem lead to symmetric subtrees.
- Only need to search one of each symmetric set.



# Symmetries in Search

- Symmetries in the problem lead to symmetric subtrees.
- Only need to search one of each symmetric set.



# Symmetries

## Avoiding redundant search

1. Detect symmetries.
2. Avoid searching through symmetric subtrees.

# Automatic Symmetry Detection

- We have investigated detecting symmetries in problems automatically.
- There are many methods for detecting symmetries.
- Two main approaches:
  - Instance-based (most) E.g., $3 \times 3$ Latin Square
  - Model-based (only two) E.g., $N \times N$ Latin Square

# Instance-based Detection Methods

## Generate-and-test

Try some permutations and see if the constraints are the same.

## Graph-based

Build a graph of the problem and find its automorphisms.

## Complete

Find all the solutions and examine them.

# Instance-based Detection Methods

## Generate-and-test

Try some permutations and see if the constraints are the same.

## Graph-based

Build a graph of the problem and find its automorphisms.

## Complete

Find all the solutions and examine them.

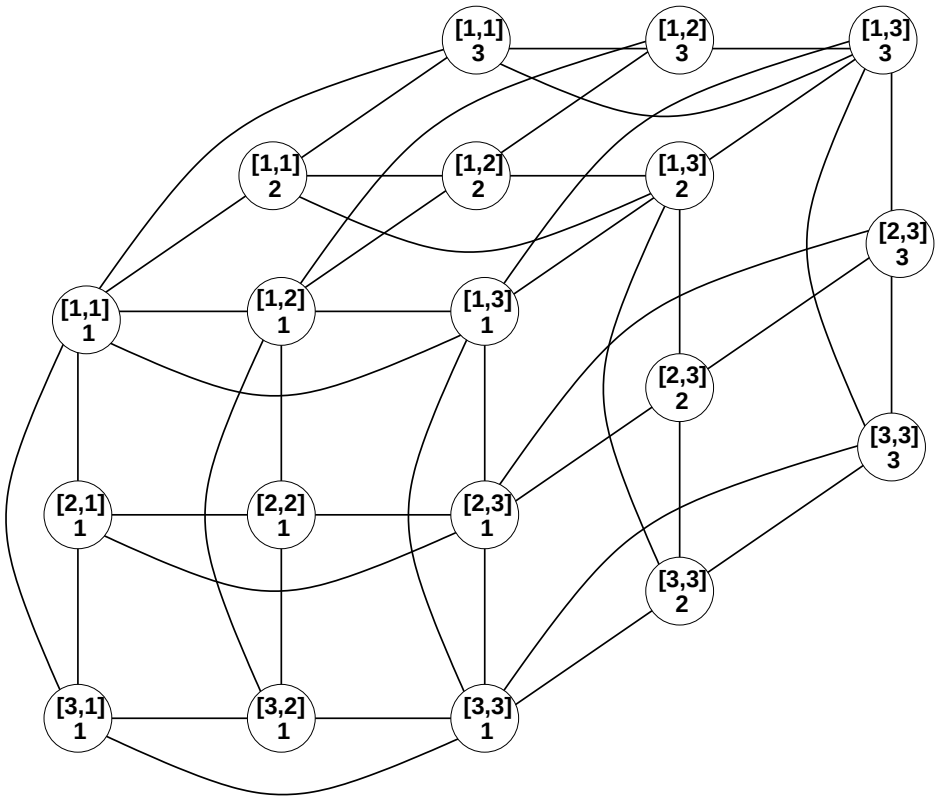
# Graph-based Detection

- Build a graph derived from the CSP.
- Once constructed, find the automorphisms of the graph.
  - Use standard tools, e.g. Saucy.
- The automorphisms correspond directly to symmetries of the instance.
- Main difference between methods is how to build the graph.

# Graph-based Detection

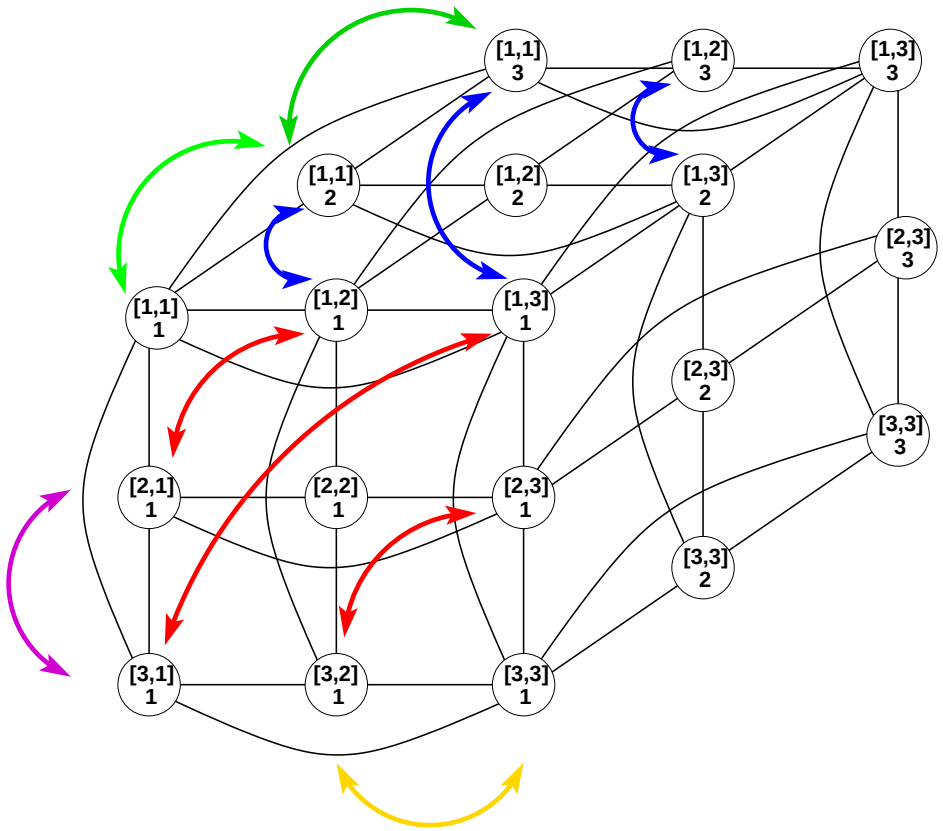
- Graph-based methods are very powerful.
- At best, can find all symmetries in the problem.
  - Variable, value, variable-value.
- Trade-off: completeness versus speed.
  - The more information encoded in the graph, the more complete, but slower.
- Not really practical.
  - Must be re-computed for every instance.

# Our Graph Construction




- Each variable-value pair is a node in the graph.
- Any two mutually exclusive nodes are joined by an edge.

# Our Graph Construction




- Each variable-value pair is a node in the graph.
- Any two mutually exclusive nodes are joined by an edge.

# Instance Detection Results

| Instance | Total | Gr | HR |
|---|---|---|---|
| bibd-6-10-5-3-2 | 1.96 | 0.83 | 0.14 |
| golf-2-2-3 | 2.71 | 0.73 | 0.23 |
| golf-2-3-2 | 6.72 | 0.72 | 0.22 |
| golomb-6 | 7.67 | 0.93 | 0.05 |
| golomb-7 | 24.45 | 0.94 | 0.03 |
| graceful-3-2 | 0.31 | 0.71 | 0.26 |
| graceful-5-2 | 8.41 | 0.82 | 0.14 |
| latin-13 | 9.17 | 0.46 | 0.37 |
| latin-14 | 12.86 | 0.46 | 0.36 |
| mostperfect-4 | 31.70 | 0.85 | 0.10 |
| nnqueens-6 | 0.30 | 0.60 | 0.33 |
| queens-30 | 6.62 | 0.82 | 0.13 |
| queens-40 | 18.43 | 0.84 | 0.11 |
| steiner-6 | 5.92 | 0.74 | 0.21 |
| steiner-7 | 57.49 | 0.76 | 0.17 |

# Existing Detection Methods for Models

- Only two methods.
- Advantages:
  - Practical: Results apply to all instances.
- Disadvantages:
  - Can lose accuracy easily.
    - Due to abstraction of information.
  - Less flexible: Depend on model syntax.
    - Require global constraints.

# Global Constraints

| | | |
|---|---|---|
| $x11$ | $x12$ | $x13$ |
| $x21$ | $x22$ | $x23$ |
| $x31$ | $x32$ | $x33$ |

$$
\begin{aligned}
X &= \{x_{11}, x_{12}, x_{13}, x_{21}, x_{22}, x_{23}, x_{31}, x_{32}, x_{33}\} \\
D &= \{1, 2, 3\} \\
C &= \{\mathit{alldiff}(x_{11}, x_{12}, x_{13}), \mathit{alldiff}(x_{21}, x_{22}, x_{23}), \mathit{alldiff}(x_{31}, x_{32}, x_{33}), \\
&\quad \mathit{alldiff}(x_{11}, x_{21}, x_{31}), \mathit{alldiff}(x_{12}, x_{22}, x_{32}), \mathit{alldiff}(x_{13}, x_{23}, x_{33})
\end{aligned}
$$

# Aim and our Framework

To develop a method of automatic symmetry detection that:

- operates on models rather than instances,
- is flexible: as syntax-independent as possible,
- is accurate: detects as many symmetries as possible,
- is practical: fast enough to be useful.

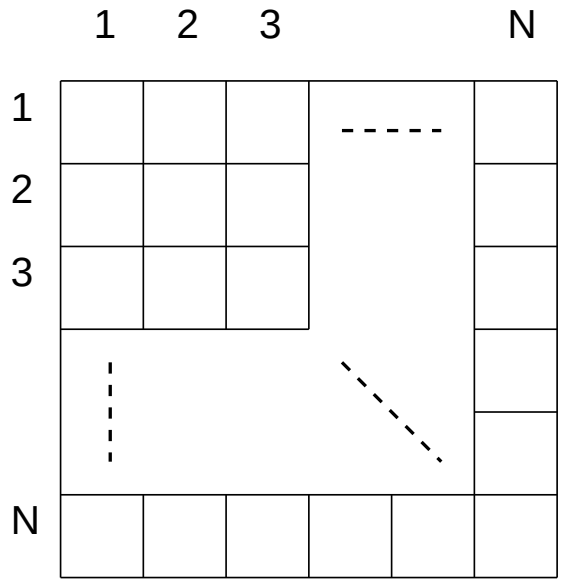
Our approach is to:

- Use the strengths of instance detection: accuracy and flexibility.
- And the strength of model detection: practicality.

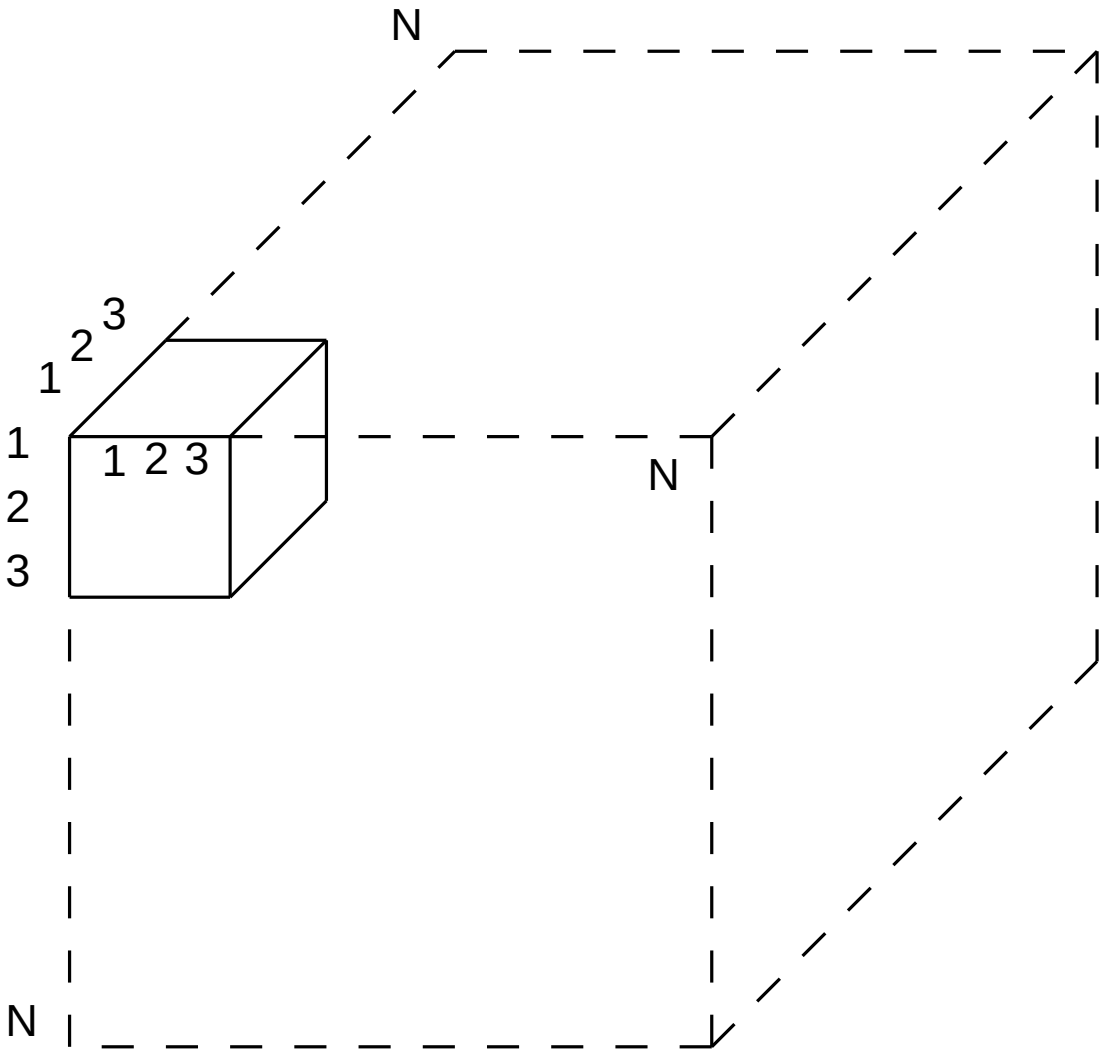
# From an Instance to a Model

- But first: what is a model?
- A model is a *parameterised CSP*.

$$X[N] = \{square_{ij} | i, j \in [1..N]\}$$
$$D[N] = [1..N]$$
$$C[N] = \{square_{ij} \neq square_{ik} | i, j \in [1..N], k \in [j+1..N]\} \cup$$
$$\{square_{ji} \neq square_{ki} | i, j \in [1..N], k \in [j+1..N]\}$$



# Parameterised Graph



# From an Instance to a Model, cont.

- Of course, a parameterised CSP is not a true CSP.
- It can be viewed as a function:

$$ParameterisedCSP : Parameter \longrightarrow CSP$$

- Similarly for the parameterised graph:

$$ParameterisedGraph : Parameter \longrightarrow Graph$$

# From an Instance to a Model, cont.

- Finally, a model symmetry is merely a parameterised symmetry:

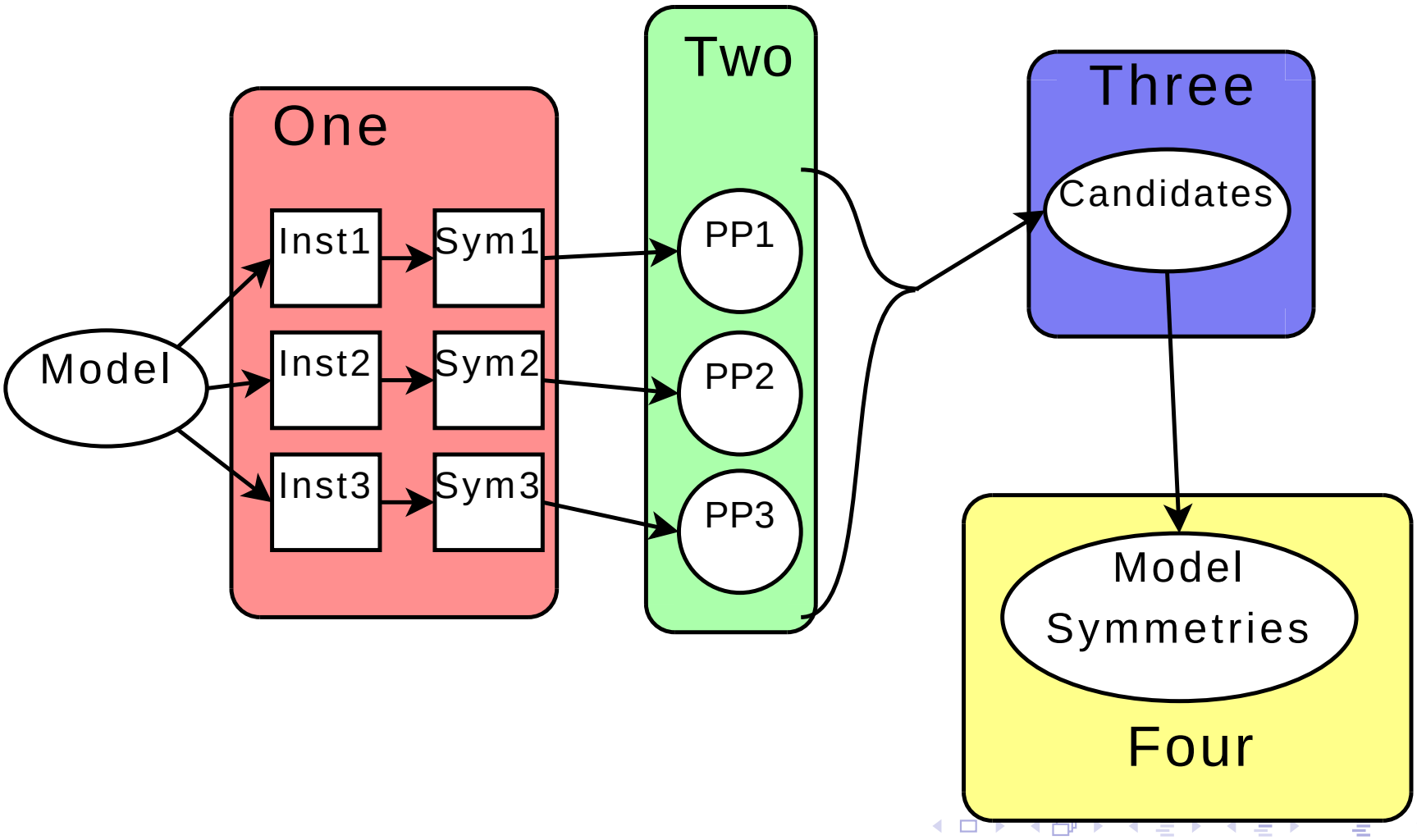
$$ParameterisedSymmetry : Parameter \longrightarrow Symmetry$$

- A parameterised permutation $f$ is a parameterised symmetry of a model $CSP$ if, for any parameter $p$, $f(p)$ is a symmetry of $CSP(p)$.
- Equivalently, $f(p)$ is an automorphism of $Graph(p)$.

# Our Framework

1. Find the symmetries of several small instances.
2. Parameterise these symmetries.
3. Filter the parameterised permutations to produce some candidate symmetries.
4. Prove (or disprove) that the candidates hold.

# Our Framework



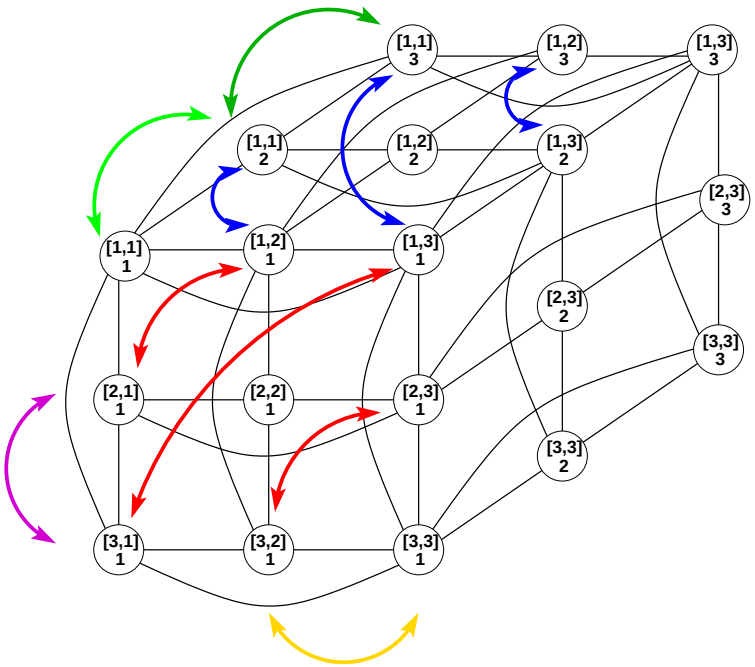
# Step 1: Find Instance Symmetries

- Choose some instances.
- Find the symmetries of these instances.
  - Choose your favourite method.
  - The completeness of the framework depends on this choice.

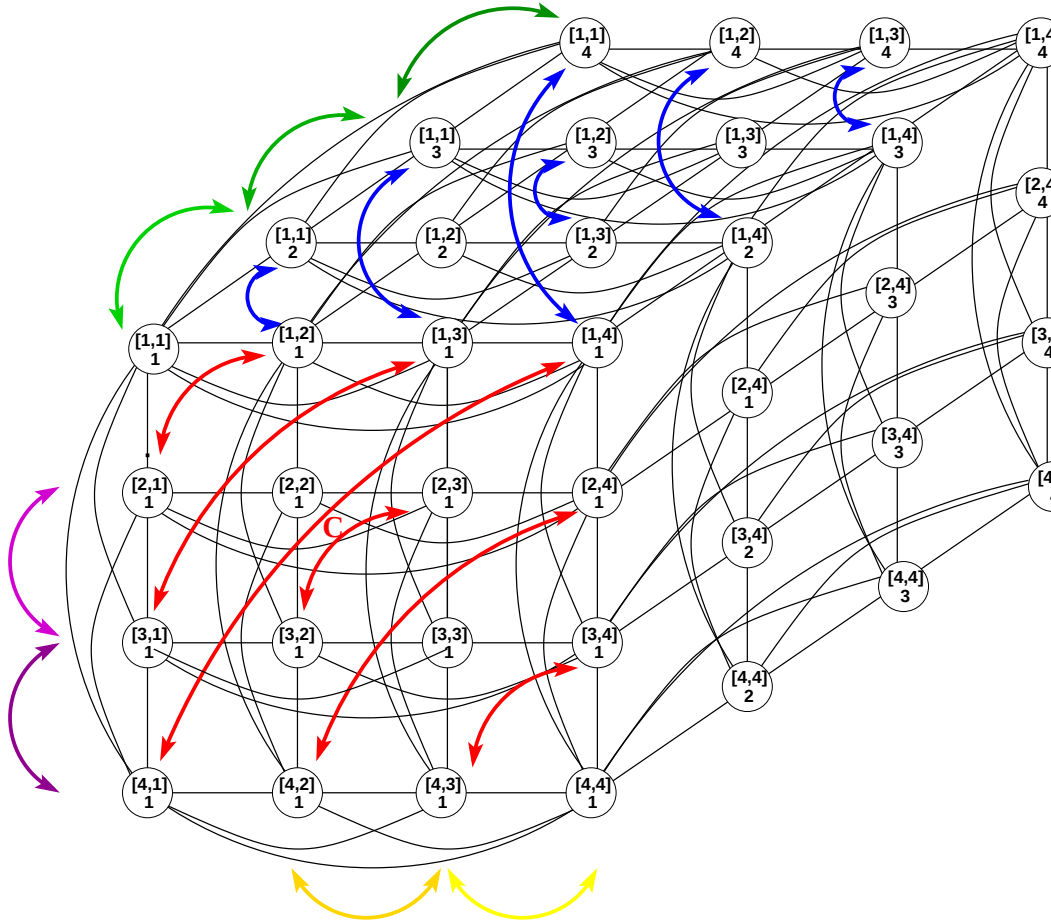
Our implementation:

- Assumes the parameter is a tuple of integers.
- Asks the user to provide some starting parameter $(a, b, c, \dots)$.
- Then tries $(a, b, c)$, $(a+1, b, c)$,$(a+2, b, c)$, $(a, b+1, c)$, etc.
- Uses the instance symmetry detection of Mears et al. (SymCon06).
  - Quite accurate but not complete.
  - Uses Saucy to produce a set of symmetry generators.

N=3


[1,1] 3
[1,2] 3
[1,3] 3
[1,1] 2
[1,2] 2
[1,3] 2
[2,3] 3
[1,1] 1
[1,2] 1
[1,3] 1
[2,3] 2
[3,3] 3
[2,1] 1
[2,2] 1
[2,3] 1
[3,3] 2
[3,1] 1
[3,2] 1
[3,3] 1

N=4


[1,1] 4
[1,2] 4
[1,3] 4
[1,1] 3
[1,2] 3
[1,3] 3
[1,4] 3
[1,1] 2
[1,2] 2
[1,3] 2
[1,4] 2
[2,4] 3
[1,1] 1
[1,2] 1
[1,3] 1
[1,4] 1
[2,4] 1
[3,4] 3
[2,1] 1
[2,2] 1
[2,3] 1
[2,4] 1
C
[3,4] 2
[4,4] 3
[3,1] 1
[3,2] 1
[3,3] 1
[3,4] 1
[4,4] 2
[4,1] 1
[4,2] 1
[4,3] 1
[4,4] 1

# Step 2: Lift Instance Symmetries to Parameterised Permutations

- We have the symmetries (or generators) relative to each instance (from step 1).
- We want to convert these into parameterised permutations.
- That is, we want to "lift" the generators to the model.

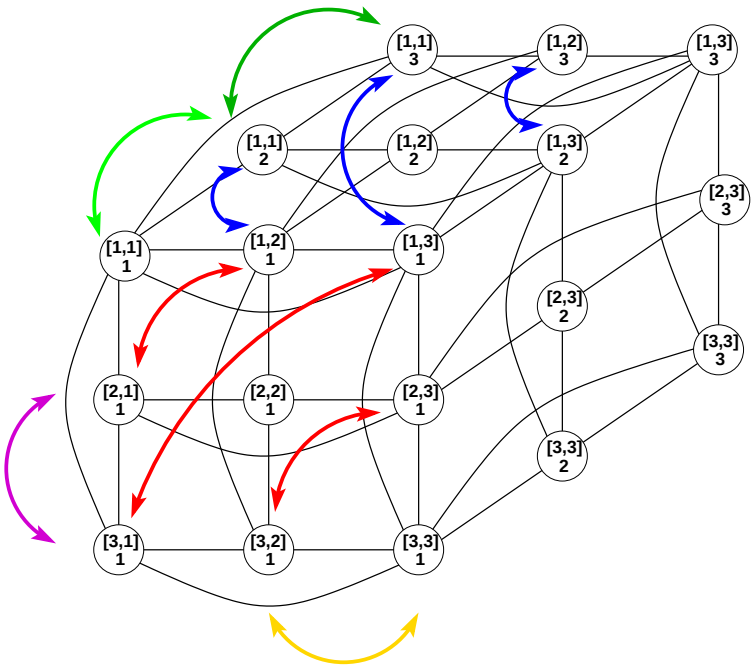
Our implementation:

- Does not attempt to be complete.
- We have identified a set of common symmetry patterns.
- Tries to find these patterns in the instance symmetries.
- Relies on the CSP having a matrix-like structure
  - Consequently, the nodes of the parameterised graph form a matrix.
  - Patterns then correspond to matrix operations (rows swap, reflections, etc.)

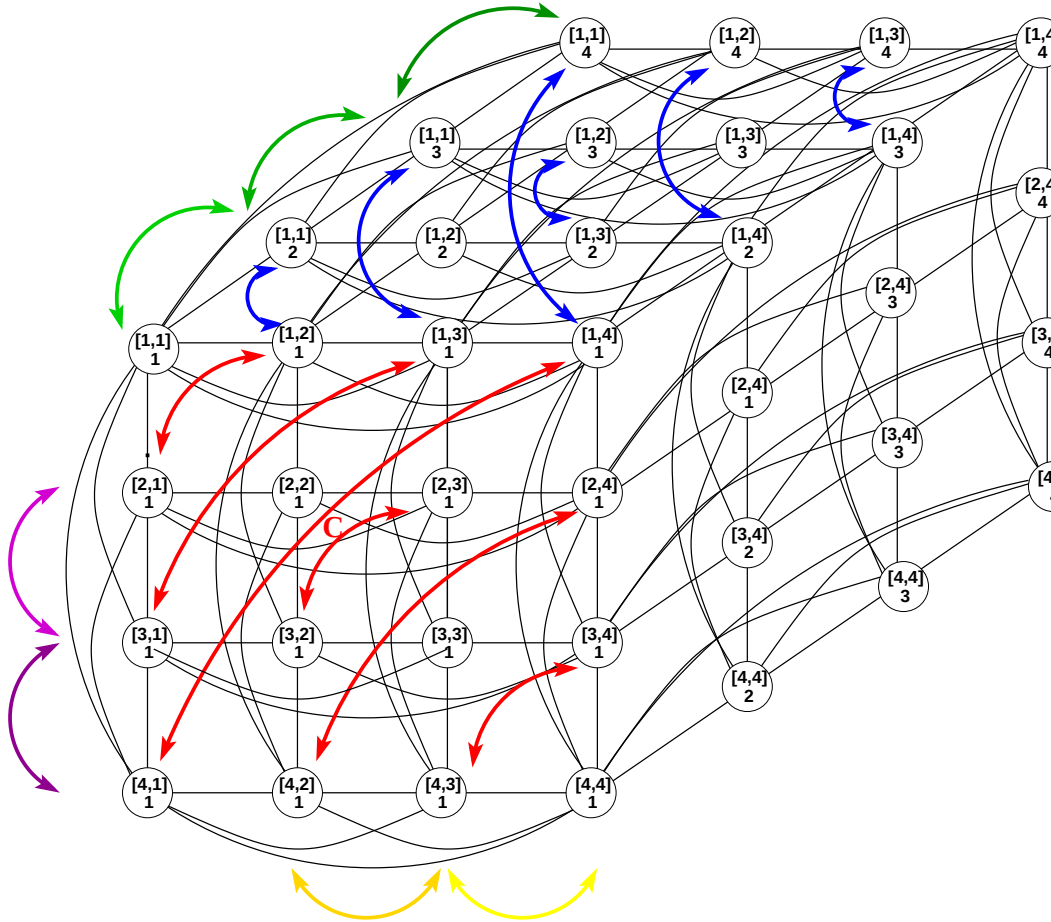
# Step 2: Our Implementation

- Common patterns:
  - Two values swapped in one dimension (row/column swap).
  - The values of a dimension inverted (matrix reflection).
  - Two dimensions swapped (diagonal matrix reflection).

N=3


[1,1] 3
[1,2] 3
[1,3] 3
[1,1] 2
[1,2] 2
[1,3] 2
[2,3] 3
[1,1] 1
[1,2] 1
[1,3] 1
[2,3] 2
[3,3] 3
[2,1] 1
[2,2] 1
[2,3] 1
[3,3] 2
[3,1] 1
[3,2] 1
[3,3] 1

N=4


[1,1] 4
[1,2] 4
[1,3] 4
[1,1] 3
[1,2] 3
[1,3] 3
[1,4] 3
[1,1] 2
[1,2] 2
[1,3] 2
[1,4] 2
[2,4] 3
[1,1] 1
[1,2] 1
[1,3] 1
[1,4] 1
[2,4] 1
[3,4] 3
[2,1] 1
[2,2] 1
[2,3] 1
[2,4] 1
C
[3,4] 2
[4,4] 3
[3,1] 1
[3,2] 1
[3,3] 1
[3,4] 1
[4,4] 2
[4,1] 1
[4,2] 1
[4,3] 1
[4,4] 1

# Step 2: Our Implementation

- Common patterns:
  - Two values swapped in one dimension (row/column swap).
  - The values of a dimension inverted (matrix reflection).
  - Two dimensions swapped (diagonal matrix reflection).
- This treats the generators independently.
- The parameterised permutations themselves can be lifted:
  - Value-swaps in a dimension can be merged.
  - At best, all the values in the dimension are interchangeable.

Our implementation:

- Pros: Simple and fast.
- Cons: Incomplete; possible improvement:
  - More patterns.
  - More complete method, e.g. machine learning.

# Step 3: Determine Candidate Symmetries

- Having gathered the parameterised permutations, we filter them to propose candidate symmetries for the model.

Our implementation:

- Simple way: take the patterns found in every instance.
- This naive intersection may miss some good candidates due to the generator sets given by Saucy.
  - A symmetry group can be described by many different generator sets.

  This can be repaired by a more advanced form of intersection.

  - If a patterns is found in one instance, look explicitly in the other instances.

# Step 4: Proving symmetries of the model

- The final step is to determine whether each candidate is a true symmetry of the model.
- We don't propose a new method for this step yet, but some work has already been done (e.g. Mancini and Cadoli, 2005).
- Such theorem-proving methods are inherently incomplete.
- We would prefer a method based on the construction of the graph (step 1); this is future work.
- Even without this step, the framework (and our implementation) is useful as a semi-automatic method.

# Results

| Problem | Symmetries | | Time | Instance |
|---|---|---|---|---|
| BIBD | objects | ✓ | 19.0 | 20% |
| | blocks | ✓ | | |
| Social Golfers | weeks | ✓ | 376.4 | 96% |
| | groups | ✓ | | |
| | players | ✓ | | |
| Golomb Ruler | flip | X | 6.7 | 99% |
| Graceful Graph | intra-clique | ✓ | 9.0 | 44% |
| | path-reverse | ✓ | | |
| | value | ✓ | | |
| Latin Square | dimensions | ✓ | 13.7 | 10% |
| | value | ✓ | | |
| N × N queens | chessboard | ✓ | 8.0 | 21% |
| | colours | ✓ | | |
| Queens (int) | chessboard | ✓ | 3.6 | 36% |
| Queens (bool) | chessboard | ✓ | 5.4 | 64% |
| Steiner Triples | triples | ✓ | 16.8 | 32% |
| | value | ✓ | | |

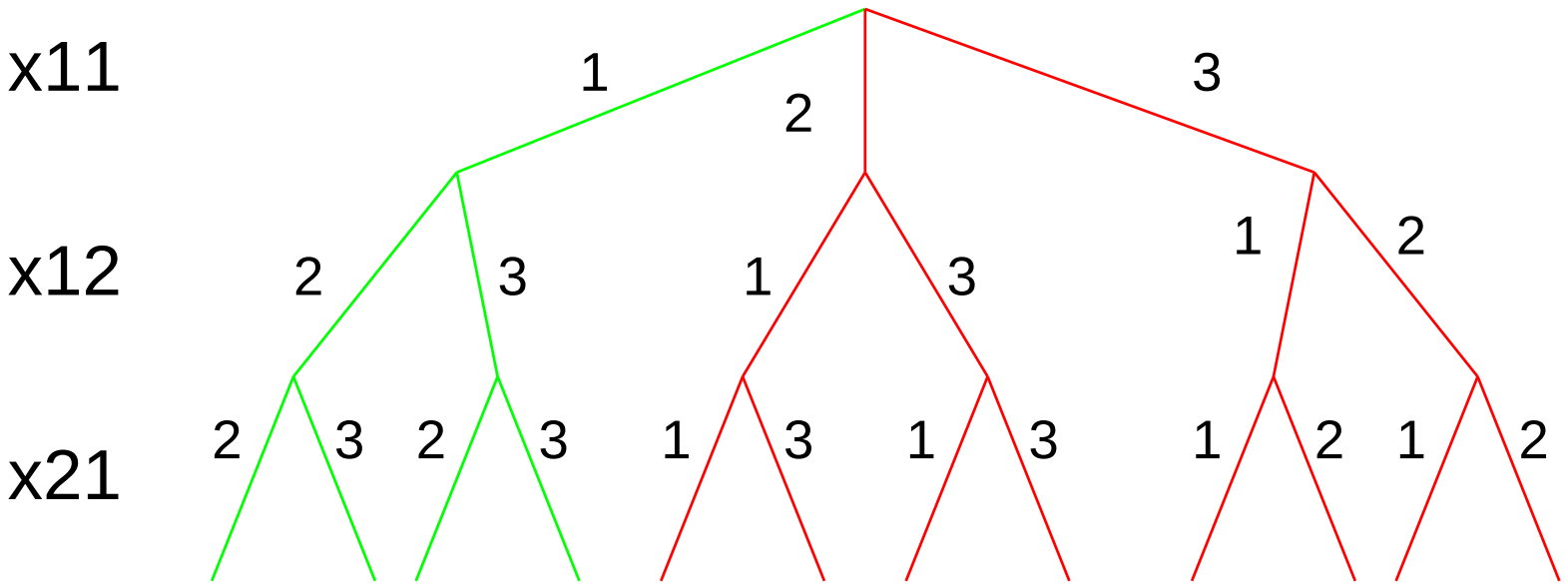
# Small Problem

- In our paper about instance symmetry detection, we showed some results of symmetry breaking.

| Instance | First Solution | | All Solutions | | |
|---|---|---|---|---|---|
| | No SBDS | SBDS(ratio) | No SBDS | SBDS(ratio) | |
| queens12 | 0.26 | 0.59 (0.44) | 9.02 | 21.92 | (0.41) |
| queens13 | 0.25 | 0.59 (0.42) | 48.23 | 118.09 | (0.41) |
| bibd33110 | 0.26 | 0.52 (0.50) | 0.26 | 0.56 | (0.46) |
| bibd77331 | 0.27 | 0.91 (0.30) | 157.89 | 1.01 | (156.33) |
| golomb5 | 0.26 | 0.53 (0.49) | 0.30 | 0.80 | (0.37) |
| golomb6 | 0.18 | 0.94 (0.19) | 8.73 | 67.89 | (0.13) |
| golf322 | 0.23 | 0.59 (0.39) | 0.29 | 0.57 | (0.51) |
| golf332 | 0.24 | 0.72 (0.33) | 76.34 | 0.77 | (99.14) |
| mostperfect4 | 0.27 | 0.60 (0.45) | 0.71 | 0.92 | (0.77) |
| steiner7 | 0.23 | 0.68 (0.34) | 392.55 | 0.87 | (451.21) |
| latin8 | 0.25 | 2.09 (0.12) | 81.12 | 647.21 | (0.13) |

# Symmetry Exploitation

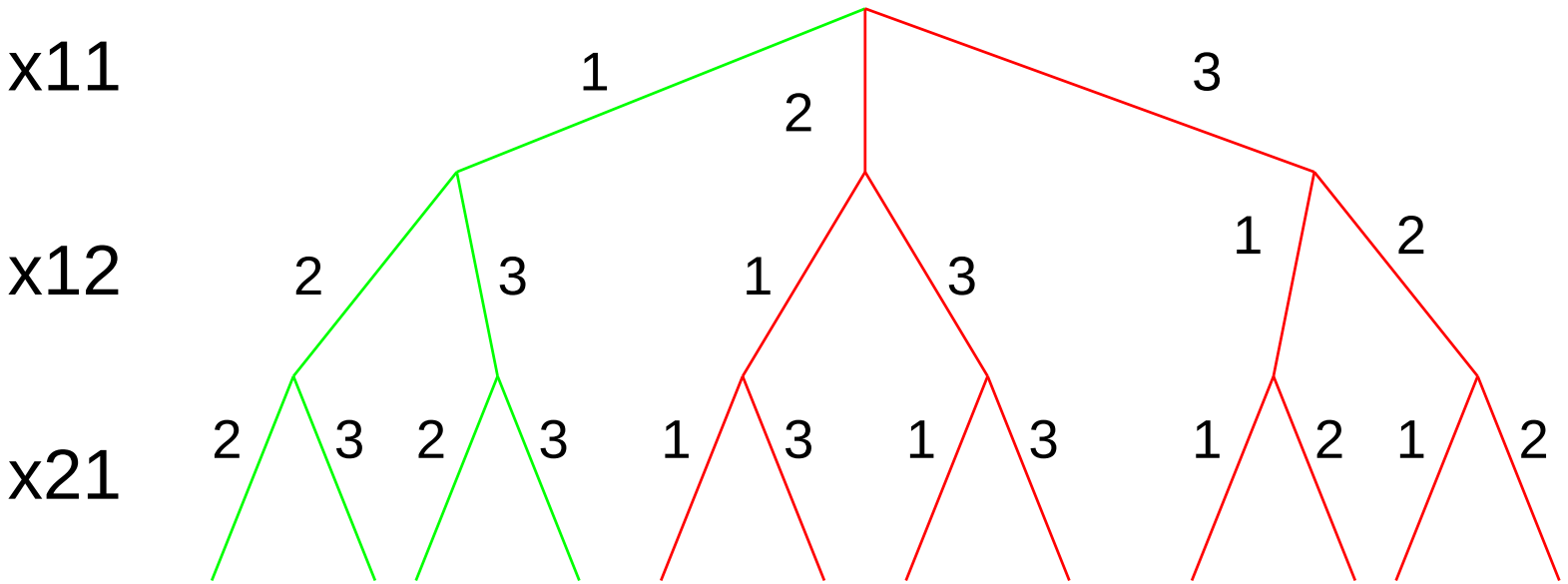
- Reminder: symmetries lead to redundancy in the search tree.
- We can exploit this redundancy to speed up search.



# Static Symmetry Exploitation

- "Break" the symmetries before you start.
- Add constraints to exclude redundant areas of the search tree.
- For example, in the Latin Square problem, fix the value of the top left square (assert $x_{11} = 1$).
- Often effective, but can interfere with search heuristics.

# Static Symmetry Exploitation

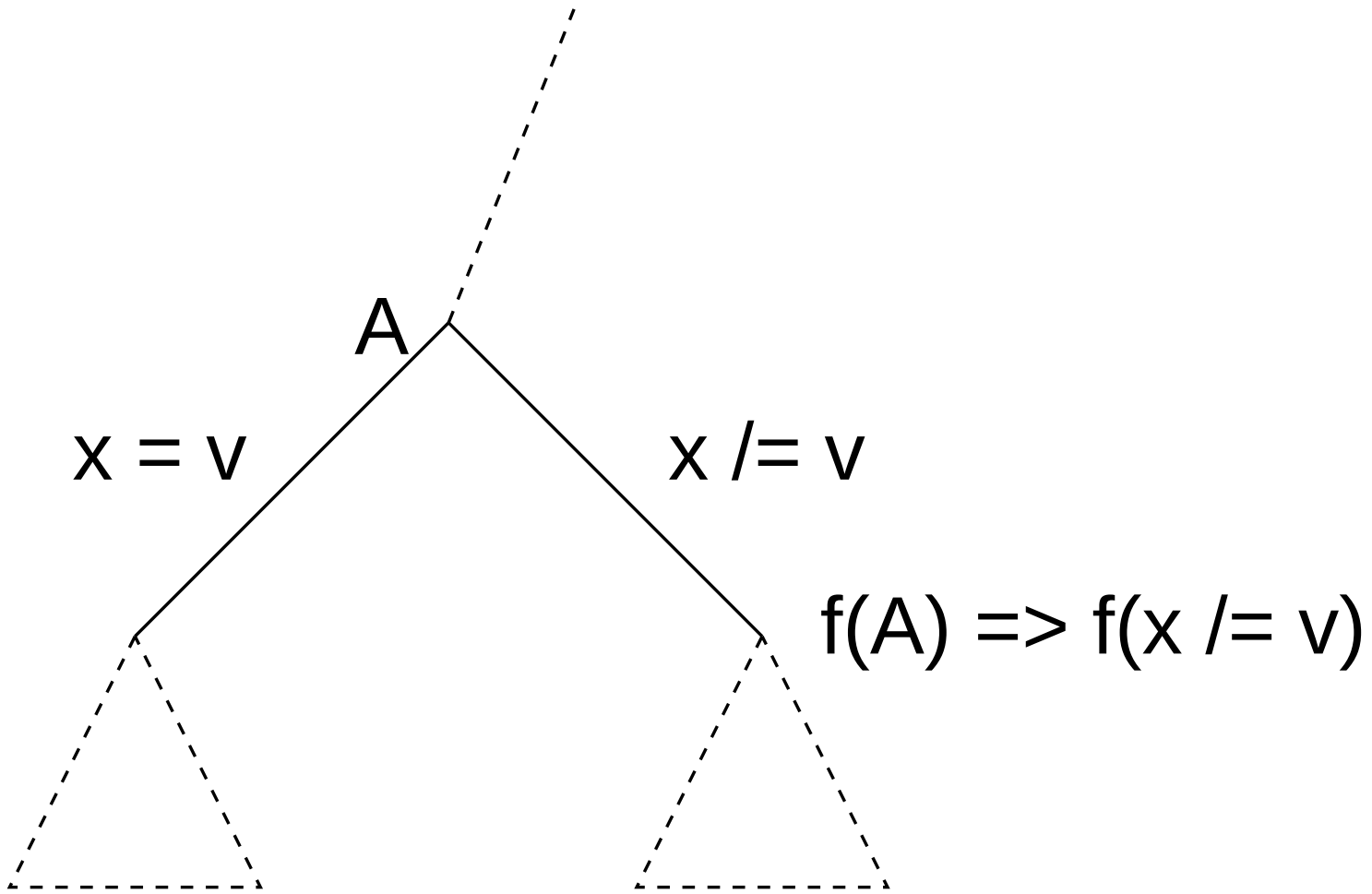


# Dynamic Symmetry Exploitation

- "Break" symmetry during search.
- Alter the search to know about symmetry.
- When the search comes to subtree symmetric to one already explored, ignore it.
- Co-operates better with search heuristic.

# Symmetry Breaking During Search

- One method: SBDS.



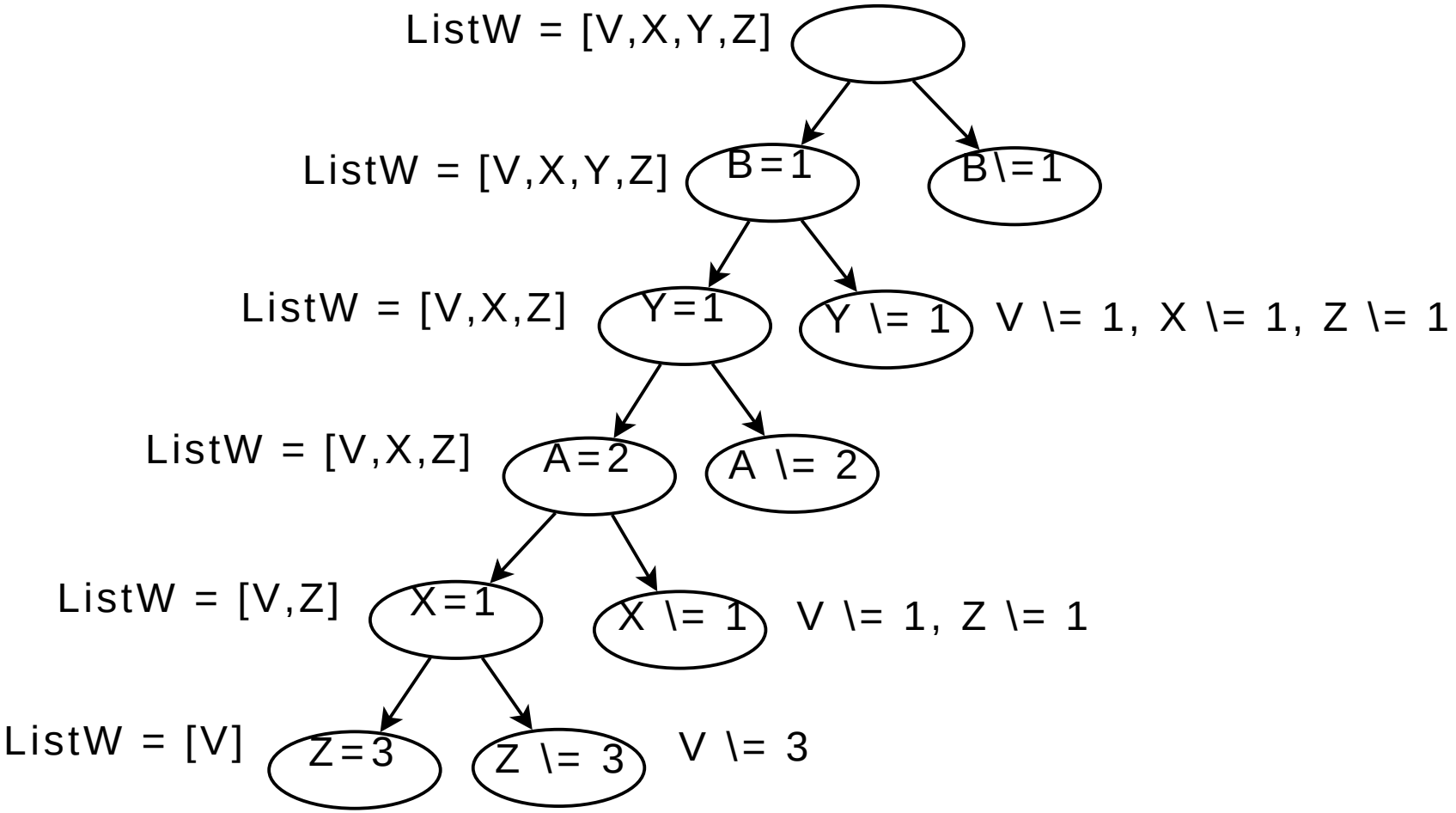
# Symmetry Breaking During Search

- Problem: there may be many, many symmetries.
- E.g. Latin Square has $6(n!)^3$ symmetries.
- Solution: use computational group theory to work with group instead of individual symmetries.
- GAP-SBDS still has large overhead.

# Lightweight Dynamic Symmetry Breaking

- Our aim: a "default" symmetry breaking method.
  - Cannot interfere with search heuristics.
  - Cannot have large overhead.
- We have proposed a simpler symmetry breaking method, LDSB.
- Doesn't handle all symmetries, only those:
  - common,
  - cheap to process.
- Can handle them cheaply, with little overhead.
- LDSB handles:
  - variable swaps
  - multi-variable swaps
  - value swaps
  - multi-value swaps

# LDSB Example



# LDSB Results

| Problem | Time (s) | | | | LDSB |
|---|---|---|---|---|---|
| | None | SBDD | SBDS | LDSB | O'head |
| bibd [12, 12, 6, 6, 4] | TO | 193.71 | TO | 6.24 | 6.1 |
| bibd [13, 13, 6, 6, 4] | TO | TO | MO | 9.07 | 5.9 |
| golf [3, 4, 3] | 0.03 | 0.67 | 17.05 | 0.02 | |
| golf [4, 4, 3] | 1.48 | 8.63 | 177.46 | 0.23 | 11.1 |
| golf [4, 4, 4] | 0.02 | 0.11 | 1.1 | 0.02 | |
| latin [20] | 1.07 | TO | MO | 1.11 | 0.1 |
| latin [25] | 2.53 | 10.49 | MO | 2.62 | 0.0 |
| latin [30] | 5.21 | TO | TO | 5.36 | 0.1 |
| magicsquare [6] | 7.49 | 187.2 | 119.61 | 9.08 | 7.2 |
| nn_queens [8] | TO | 162.72 | 127.17 | 19.09 | 8.4 |
| queens_bool [24] | 6.9 | 236.89 | 105.88 | 7.22 | 4.8 |
| steiner [9] | 1.71 | 3.56 | 7.07 | 0.23 | 11.1 |
| | ↑ First solution ↑ – ↓ All solutions ↓ | | | | |
| bibd [12, 12, 6, 6, 4] | TO | 194.17 | TO | 5.87 | 6.0 |
| bibd [13, 13, 6, 6, 4] | TO | TO | MO | 9.12 | 6.0 |
| golf [3, 4, 3] | TO | 4.62 | 111.32 | 3.85 | 13.1 |
| golf [4, 4, 3] | TO | 18.0 | TO | 42.35 | 10.7 |
| golf [4, 4, 4] | TO | 58.9 | TO | 1.55 | 10.5 |
| latin [6] | TO | 8.2 | 118.26 | 10.19 | 5.6 |
| magicsquare [4] | 23.5 | 30.14 | 14.81 | 2.85 | 9.4 |
| nn_queens [8] | TO | 162.04 | 127.24 | 19.13 | 8.3 |
| queens [14] | 160.26 | 145.84 | 263.67 | 67.88 | 11.1 |
| queens_bool [12] | 23.52 | 78.59 | 36.0 | 21.18 | 5.3 |
| queens_bool [13] | 121.92 | TO | 202.64 | 106.79 | 5.1 |
| steiner [9] | TO | 12.98 | 25.76 | 19.64 | 13.6 |

# Conclusion

## Summary

- Symmetry in constraint programming is an interesting field.
- Still much work to be done on real-world use of symmetries.

## What's next?

- The proof step for model symmetry detection.
- Applying the framework to other properties.
- Further exploration of cheap symmetry breaking.