Centre for
Data Analytics

Insight

Basic Modelling

Helmut Simonis

CRT-AI CP Week 2025

Licence

Acknowledgments

This publication has emanated from research conducted with the financial support of Science Foundation Ireland under Grant number 12/RC/2289-P2 at Insight the SFI Research Centre for Data Analytics at UCC, which is co-funded under the European Regional Development Fund.
A version of this material was developed as part of the ECLiPSe ELearning course: `https://eclipseclp.org/ELearning/index.html`.
Support from Cisco Systems and the Silicon Valley Community Foundation is gratefully acknowledged.

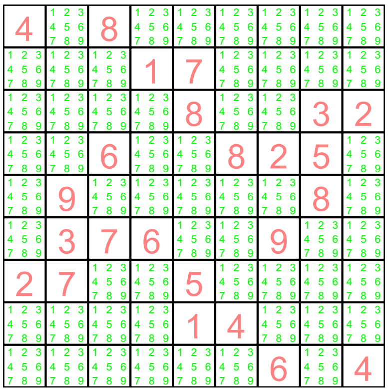
Example 2: Sudoku

- Global Constraints
  - Powerful modelling abstractions
  - Non-trivial propagation
  - Different consistency levels
- Example: Sudoku puzzle

Problem Definition

Sudoku

Fill in numbers from 1 to 9 so that each row, column and 3*x*3 block contain each number exactly once



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

Model

- A variable for each cell, ranging from 1 to 9
- A 9x9 matrix of variables describing the problem
- Preassigned integers for the given hints
- `alldifferent` constraints for each row, column and 3x3 block

Sudoku Models

- ECLiPSe ‣ Show
- MiniZinc ‣ Show
- NumberJack ‣ Show
- CPMpy ‣ Show
- Choco-solver ‣ Show

ECLiPSe Sudoku Model (from `https://eclipseclp.org/`)

```
:- lib(ic).
:- import alldifferent/1 from ic_global.

top :-
    problem(Board),
    print_board(Board),
    sudoku(Board),
    labeling(Board),
    print_board(Board).

sudoku(Board) :-
    dim(Board, [N,N]),
    Board :: 1..N,
    ( for(I,1,N), param(Board) do
        alldifferent(Board[I,*]),
        alldifferent(Board[*,I])
    ),
    NN is integer(sqrt(N)),
    ( multifor([I,J],1,N,NN), param(Board,NN) do
        alldifferent(concat(Board[I..I+NN-1, J..J+NN-1]))
    ).

print_board(Board) :-
    dim(Board, [N,N]),
    ( for(I,1,N), param(Board,N) do
        ( for(J,1,N), param(Board,I) do
        X is Board[I,J],
        ( var(X) -> write("  _") ; printf(" %2d", [X]) )
        ), nl
    ), nl.
```

ECLiPSe Data Definition

```
problem([](
    [](4, _, 8, _, _, _, _, _, _),
    [](_, _, _, 1, 7, _, _, _, _),
    [](_, _, _, _, 8, _, _, 3, 2),
    [](_, _, 6, _, _, 8, 2, 5, _),
    [](_, 9, _, _, _, _, _, 8, _),
    [](_, 3, 7, 6, _, _, 9, _, _),
    [](2, 7, _, _, 5, _, _, _, _),
    [](_, _, _, _, 1, 4, _, _, _),
    [](_, _, _, _, _, _, 6, _, 4))).
```

Continue

MiniZinc Sudoku Model

```
int: s;
int: n=s*s;
array[1..n,1..n] of var 1..n: puzzle;

include "sudoku.dzn";

predicate alldifferent(array[int] of var int: x) =
  forall(i,j in index_set(x) where i < j)
    (x[i] != x[j]);

constraint forall(i in 1..n)
    (alldifferent([puzzle[i,j]| j in 1..n]));
constraint forall(j in 1..n)
    (alldifferent([ puzzle[i,j]| i in 1..n]));
constraint forall(i,j in 1..s)
    (alldifferent([puzzle[s*(i-1)+p, s*(j-1)+q]|
                  p,q in 1..s]));
solve satisfy;
```

MiniZinc Output

```
output [ "sudoku:\\n" ] ++
  [ show(puzzle[i,j]) ++
    if j = n then
       if i mod s = 0 /\ i < n then "\n\n"
       else "\n"
       endif
     else
       if j mod s = 0 then "   "
       else " "
       endif
     endif
    | i,j in 1..n ];
```


MiniZinc Data File (sudoku.dzn)

```
s=3;
puzzle=[|
    4, _, 8, _, _, _, _, _, _|
    _, _, _, 1, 7, _, _, _, _|
    _, _, _, _, 8, _, _, 3, 2|
    _, _, 6, _, _, 8, 2, 5, _|
    _, 9, _, _, _, _, _, 8, _|
    _, 3, 7, 6, _, _, 9, _, _|
    2, 7, _, _, 5, _, _, _, _|
    _, _, _, _, 1, 4, _, _, _|
    _, _, _, _, _, _, 6, _, 4|
|];
```

Continue

NumberJack Sudoku Model

```
from Numberjack import *

def get_model(N, clues):
    grid = Matrix(N*N, N*N, 1, N*N)

    sudoku = Model([AllDiff(row) for row in grid.row],
                   [AllDiff(col) for col in grid.col],
                   [AllDiff(grid[x:x + N, y:y + N]) for x in range(0, N*N, N)
                                                    for y in range(0, N * N, N)],
                   [(x == int(v)) for x, v in
                       zip(grid.flat, "".join(open(clues)).split()) if v != '*']
                   )
    return grid, sudoku

def solve(param):
    N = param['N']
    clues = param['file']

    grid, sudoku = get_model(N, clues)

    solver = sudoku.load(param['solver'])
    solver.setVerbosity(param['verbose'])
    solver.setTimeLimit(param['tcutoff'])

    solver.solve()
```


NumberJack Data File

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

Continue

CPMpy Sudoku Model(from https://github.com/CPMpy/)

```
import numpy as np
from cpmpy import *

# Variables
puzzle = intvar(1,9, shape=given.shape, name="puzzle")

model = Model(
    # Constraints on values (cells that are not empty)
    puzzle[given!=e] == given[given!=e], # numpy’s indexing, vectorized equality
    # Constraints on rows and columns
    [AllDifferent(row) for row in puzzle],
    [AllDifferent(col) for col in puzzle.T], # numpy’s Transpose
)

# Constraints on blocks
for i in range(0,9, 3):
    for j in range(0,9, 3):
        model += AllDifferent(puzzle[i:i+3, j:j+3]) # python’s indexing

model.solve()
```


CPMpy Data Definition

```
e = 0 # value for empty cells
given = np.array([
    [4, e, 8, e, e, e, e, e, e],
    [e, e, e, 1, 7, e, e, e, e],
    [e, e, e, e, 8, e, e, 3, 2],
    [e, e, 6, e, e, 8, 2, 5, e],
    [e, 9, e, e, e, e, e, 8, e],
    [e, 3, 7, 6, e, e, 9, e, e],
    [2, 7, e, e, 5, e, e, e, e],
    [e, e, e, e, 1, 4, e, e, e],
    [e, e, e, e, e, e, 6, e, 4]
])
```

▸ Continue

Choco-solver Sudoku Model

```
Model model = new Model("Sudoku");
int blockSize = 3;
int m = blockSize*blockSize;

IntVar[][] vars = new IntVar[m][m];
for(int i=0;i<m;i++){
    for(int j=0;j<m;j++){
        vars[i][j] =  model.intVar("X"+i+""+j, 1, m);
        if (data[i][j]>0) {
            model.arithm(vars[i][j],"=",data[i][j]).post();
        }
    }
}
for(int i=0;i<m;i++){
    model.allDifferent(row(i,m,vars)).post();
    model.allDifferent(column(i,m,vars)).post();
}
for(int i=0;i<m;i+=blockSize){
    for(int j=0;j<m;j+=blockSize){
        model.allDifferent(block(i,j,blockSize,vars)).post();
    }
}
Solver solver = model.getSolver();
solver.solve();
```


Choco-solver Data

```
int[][] data = new int[m][m]{
    {4, 0, 8, 0, 0, 0, 0, 0, 0},
    {0, 0, 0, 1, 7, 0, 0, 0, 0},
    {0, 0, 0, 0, 8, 0, 0, 3, 2},
    {0, 0, 6, 0, 0, 8, 2, 5, 0},
    {0, 9, 0, 0, 0, 0, 0, 8, 0},
    {0, 3, 7, 6, 0, 0, 9, 0, 0},
    {2, 7, 0, 0, 5, 0, 0, 0, 0},
    {0, 0, 0, 0, 1, 4, 0, 0, 0},
    {0, 0, 0, 0, 0, 0, 6, 0, 4}
};
```

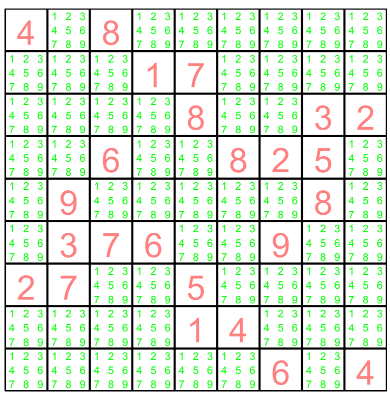
Choco-solver Utilities

```
IntVar[] row(int row, int size, IntVar[][] array){
    return array[row];
}

IntVar[] column(int col,int size,IntVar[][] array){
    IntVar[] column = new IntVar[size];
    for(int i=0; i<size; i++){
        column[i] = array[i][col];
    }
    return column;
}

IntVar[] block(int x,int y,int blockSize,IntVar[][] array){
    IntVar[] block = new IntVar[blockSize*blockSize];
    int k=0;
    for(int i=0;i<blockSize;i++){
        for(int j=0;j<blockSize;j++){
            block[k++] = array[x+i][y+j];
        }
    }
    return block;
}
```

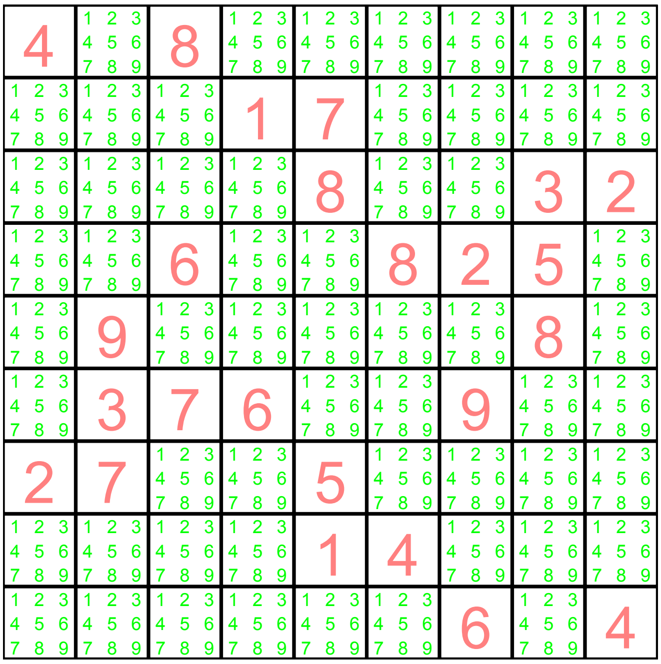
Continue

Domain Visualizer

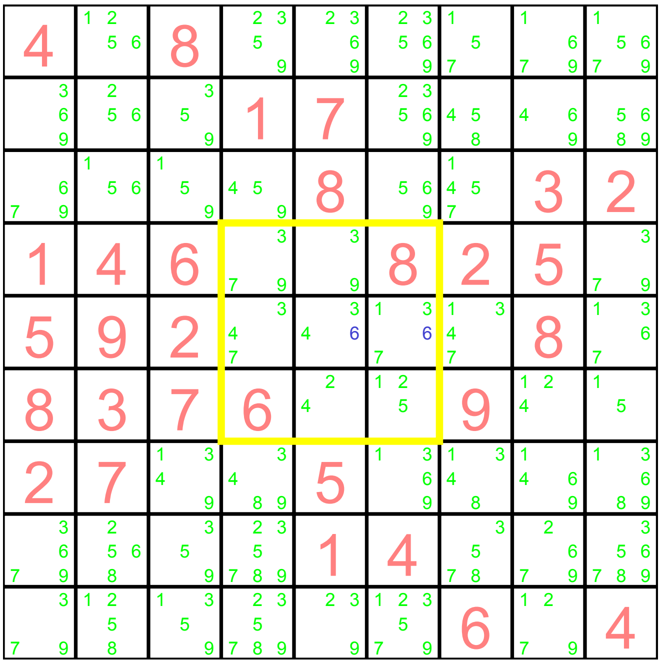
- Problem shown as matrix
- Each cell corresponds to a variable
- Instantiated: Shows integer value (large)
- Uninstantiated: Shows values in domain



Initial State (Forward Checking)



Propagation Steps (Forward Checking)



After Setup (Forward Checking)

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

Can we do better?

- The alldifferent constraint is missing propagation
  - How can we do more propagation?
  - Do we know when we derive all possible information from the constraint?
- Constraints only interact by changing domains of variables

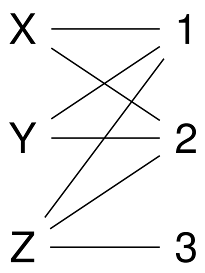
A Simpler Example

```
include "alldifferent.mzn";

var 1..2:X;
var 1..2:Y;
var 1..3:Z;

constraint alldifferent([X,Y,Z]);

solve satisfy;
```


Using Forward Checking

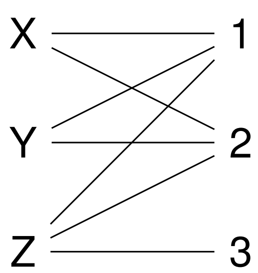
- No variable is assigned
- No reduction of domains
- But, values 1 and 2 can be removed from Z
- This means that Z is assigned to 3

Visualization of alldifferent as Graph



- Show problem as graph with two types of nodes
  - Variables on the left
  - Values on the right
- If value is in domain of variable, show link between them
- This is called a *bipartite* graph

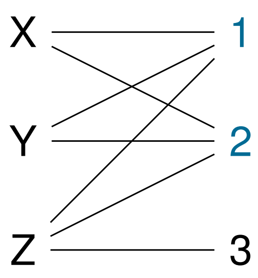
A Simpler Example



Value Graph for

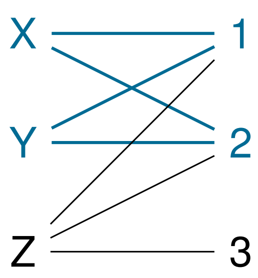
```
var 1..2:X;

var 1..2:Y;

var 1..3:Z;
```

A Simpler Example



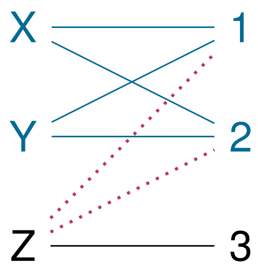
Check interval [1,2]

A Simpler Example



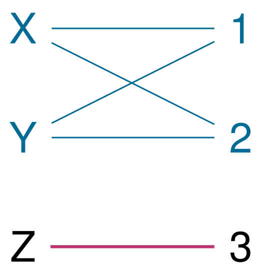
- Find variables completely contained in interval
- There are two: X and Y
- This uses up the capacity of the interval

A Simpler Example



No other variable can use that interval

A Simpler Example



Only one value left in domain of Z, this can be assigned

Idea (Hall Intervals)

- Take each interval of possible values, say size N
- Find all K variables whose domain is completely contained in interval
- If $K > N$ then the constraint is infeasible
- If $K = N$ then no other variable can use that interval
- Remove values from such variables if their bounds change
- If $K < N$ do nothing
- Re-check whenever domain bounds change

Implementation

- Problem: Too many intervals ($O(n^2)$) to consider
- Solution:
  - Check only those intervals which update bounds
  - Enumerate intervals incrementally
  - Starting from lowest(highest) value
  - Using sorted list of variables
- Complexity: $O(n \log(n))$ in standard implementations
- Important: Only looks at min/max bounds of variables

Bounds Consistency

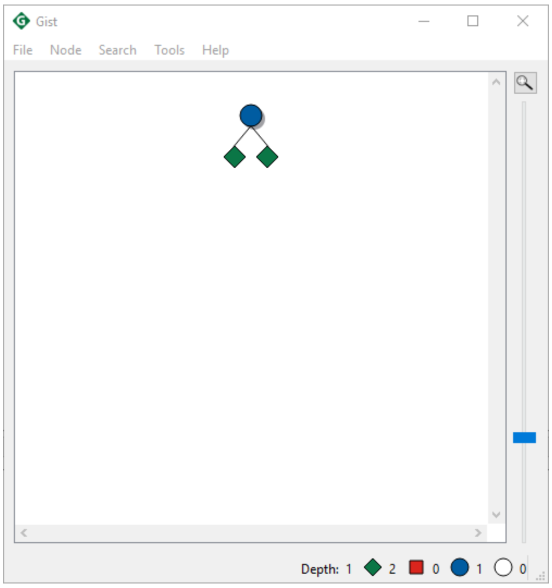
Definition

A constraint achieves *bounds consistency*, if for the lower and upper bound of every variable, it is possible to find values for all other variables between their lower and upper bounds which satisfy the constraint.

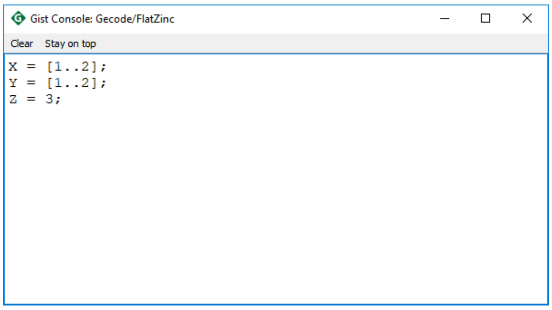
Annotation: :: bounds

```
include "alldifferent.mzn";

var 1..2:X;
var 1..2:Y;
var 1..3:Z;

constraint alldifferent([X,Y,Z]) :: bounds;
solve satisfy;
```

Running with Gecode Gist



All Solutions



Node Inspector (Root)

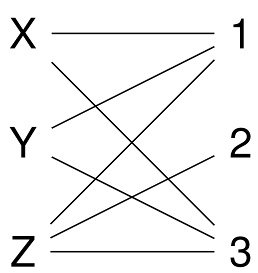
Can we do even better?

- Bounds consistency only considers min/max bounds
- Ignores "holes" in domain
- Sometimes we can improve propagation looking at those holes

Another Simple Example

```
include "alldifferent.mzn";

var {1,3}:X; % note enumerated domain
var {1,3}:Y;
var 1..3:Z; % note domain as interval

% annotated constraint
constraint alldifferent([X,Y,Z]) :: bounds;
solve satisfy;
```

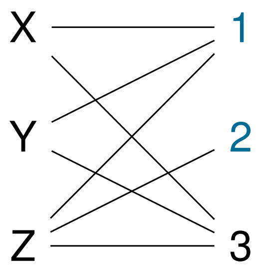

Another Simple Example



Value Graph for

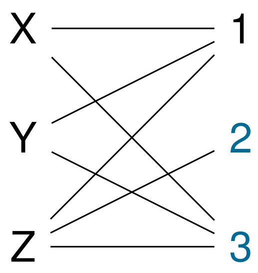
```
var {1,3}:X;

var {1,3}:Y;

var 1..3:Z;
```

Another Simple Example



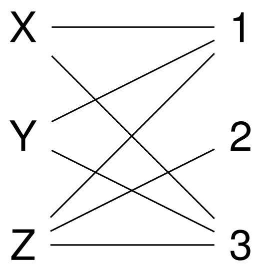
- Check interval [1,2]
- No domain of a variable completely contained in interval
- No propagation

Another Simple Example



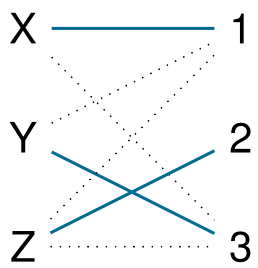
- Check interval [2,3]
- No domain of a variable completely contained in interval
- No propagation

Another Simple Example



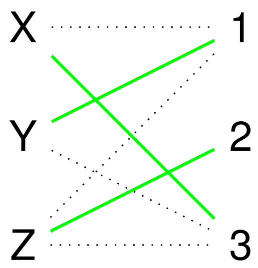
But, more propagation is possible, there are only two solutions

Another Simple Example



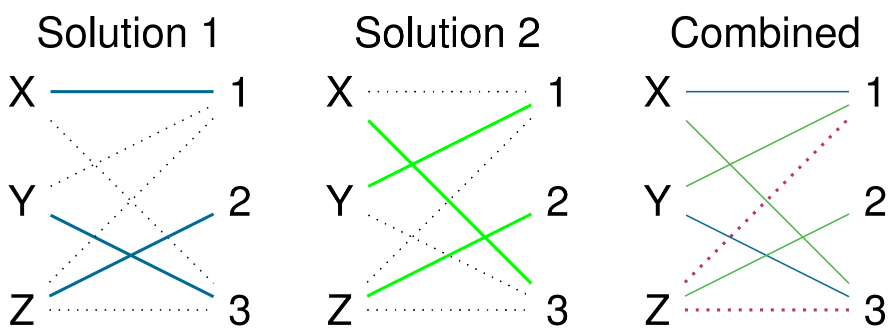
Solution 1: assignment in blue

Another Simple Example



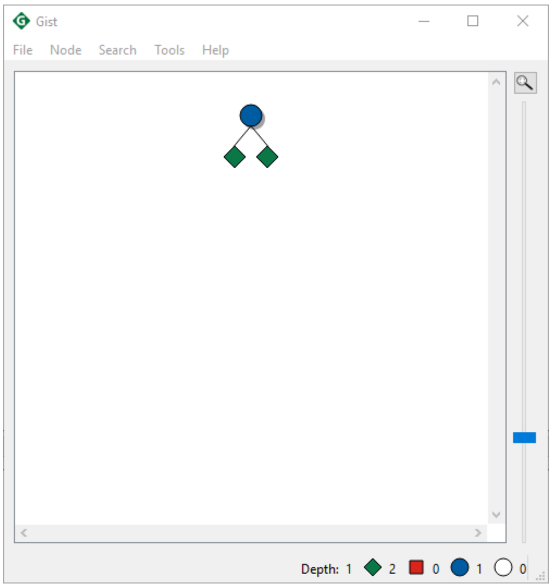
Solution 2: assignment in green

Another Simple Example

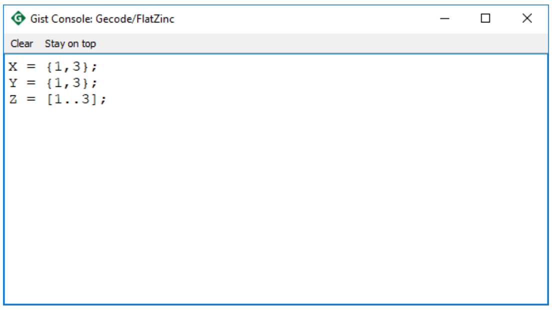


Combining solutions shows that Z=1 and Z=3 are not possible. Can we deduce this without enumerating solutions?

Bounds Consistency with Gecode Gist: No Propagation



All Solutions



Node Inspector (Root)

Solutions and Maximal Matchings

- A *Matching* is subset of edges which do not coincide in any node
- No matching can have more edges than number of variables
- Every solution corresponds to a *maximal matching* and vice versa
- If a link does not belong to some maximal matching, then it can be removed

Implementation

- Possible to compute all links which belong to some matching
  - Without enumerating all of them!
- Enough to compute **one** maximal matching
- Requires algorithm for *strongly connected components*
- Extra work required if more values than variables
- All links (values in domains) which are not supported can be removed
- Complexity: $O(n^{1.5}d)$

Domain Consistency

Definition

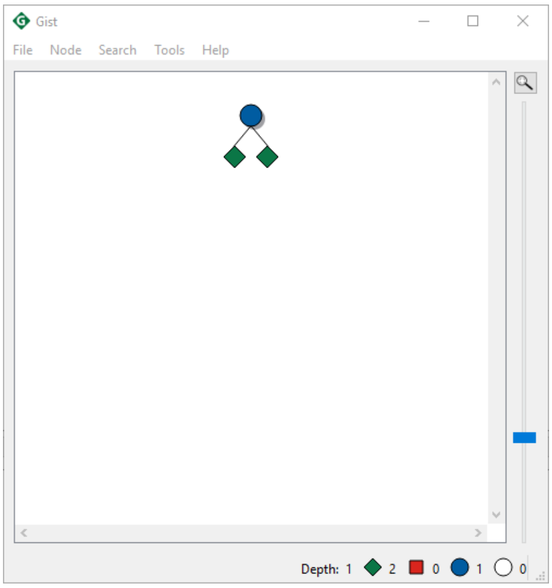
A constraint achieves *domain consistency*, if for every variable and for every value in its domain, it is possible to find values in the domains of all other variables which satisfy the constraint.

- Also called *generalized arc consistency (GAC)*
- or *hyper arc consistency*

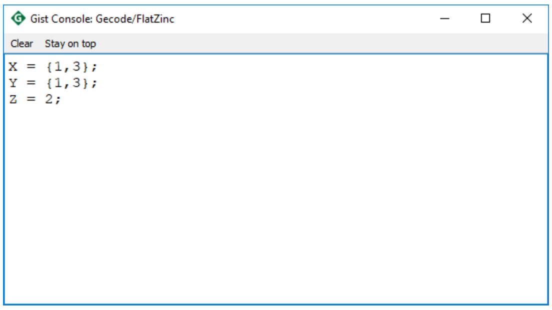
Simple Example Revisited

```
include "alldifferent.mzn";

var {1,3}:X; % note enumerated domain
var {1,3}:Y;
var 1..3:Z; % note domain as interval

% note different annotation
constraint alldifferent([X,Y,Z]) :: domain;
solve satisfy;
```


Domain Consistency with Gecode Gist: Propagation



All Solutions



Node Inspector (Root)

Can we still do better?

- NO! This extracts all information from this one constraint
- We could perhaps improve speed, but not propagation
- But possible to use different model
- Or model interaction of multiple constraints

Should all constraints achieve domain consistency?

- Domain consistency is usually more expensive than bounds consistency
  - Overkill for simple problems
  - Nice to have choices
- For some constraints achieving domain consistency is NP-hard
  - We have to live with more restricted propagation

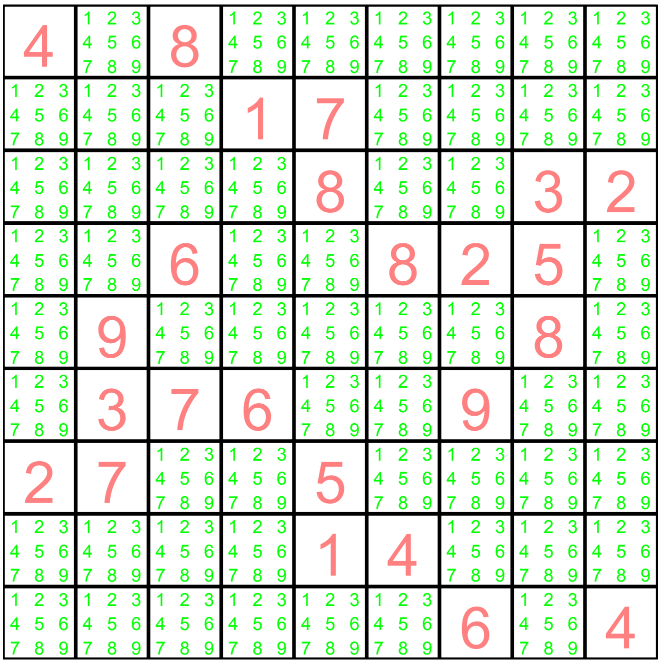
Modified MiniZinc Program

```
int: s;
int: n=s*s;
array[1..n,1..n] of var 1..n: puzzle;

include "sudoku.dzn";

include "alldifferent.mzn";

constraint forall(i in 1..n)
    (alldifferent([puzzle[i,j]| j in 1..n])::domain);
constraint forall(j in 1..n)
    (alldifferent([ puzzle[i,j]| i in 1..n])::domain);
constraint forall(i,j in 1..s)
    (alldifferent([puzzle[s*(i-1)+p, s*(j-1)+q]|
                    p,q in 1..s])::domain);

solve satisfy;
```


Modified Choco-solver Sudoku Model

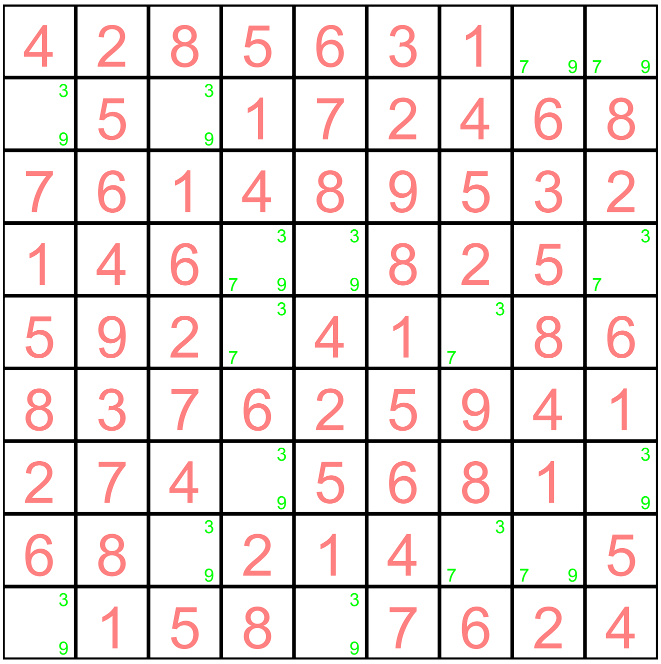
```
    Model model = new Model("Sudoku");
    int blockSize = 3;
    int m = blockSize*blockSize;

    IntVar[][] vars = new IntVar[m][m];
    for(int i=0;i<m;i++){
        for(int j=0;j<m;j++){
            vars[i][j] =  model.intVar("X"+i+""+j, 1, m);
            if (data[i][j]>0) {
                model.arithm(vars[i][j],"=",data[i][j]).post();
            }
        }
    }
// Consistency level AC: domain consistency, BC: bounds consistency, default: mix
    for(int i=0;i<m;i++){
        model.allDifferent(row(i,m,vars),AC).post();
        model.allDifferent(column(i,m,vars),AC).post();
    }
    for(int i=0;i<m;i+=blockSize){
        for(int j=0;j<m;j+=blockSize){
            model.allDifferent(block(i,j,blockSize,vars),AC).post();
        }
    }
    Solver solver = model.getSolver();
    solver.solve();
```

Initial State (Domain Consistency)



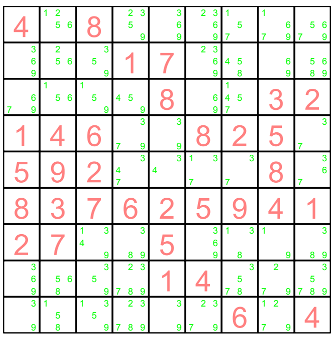
Propagation Steps (Domain Consistency)

After Setup (Domain Consistency)

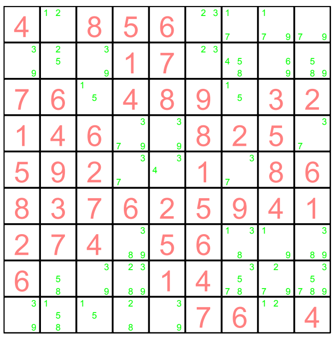


Comparison

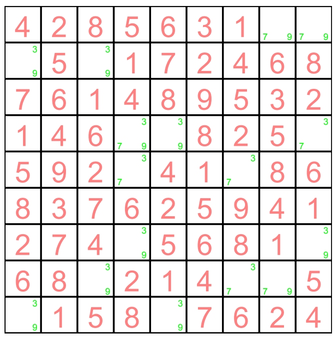
Forward Checking



Bounds Consistency



Domain Consistency



Typical?

- This does not always happen
- Sometimes, two methods produce same amount of propagation
- Possible to predict in certain special cases
- In general, tradeoff between speed and propagation
- Not always fastest to remove inconsistent values early
- But often required to find a solution at all

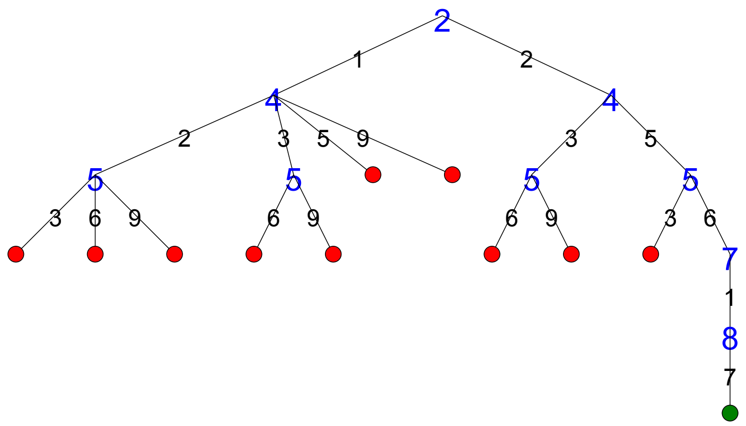
Simple search routine

- Enumerate variables in given order
- Try values starting from smallest one in domain
- Complete, chronological backtracking
- Advantage: Results can be compared with each other
- Disadvantage: Usually not a very good strategy

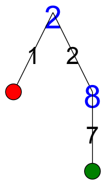
Asking for Naive Search in MiniZinc

```
solve :: int_search(
  puzzle,
  input_order,
  indomain_min)
satisfy;
```


Search Tree (Forward Checking)



Search Tree (Bounds Consistency)



Search Tree (Domain Consistency)

Trading Propagation Against Search

- If we perform more propagation, search is more constrained
- Fewer values left, fewer alternatives to explore in search
- Best compromise is not obvious
- But can be learned from examples or during search
- Annotations are optional
  - Some MiniZinc back-end solvers do the search they want, not the one you specify
  - Some solvers simply do not work in a way that these search annotations apply

Are there other Global Constraints?

- alldifferent is the most commonly used constraint
- Propagation methods can be explained
- But there are many more

Global Constraint Catalog

- `https://sofdem.github.io/gccat/`
- Description of 354 global constraints, 2800 pages
- Not all of them are widely used
- Detailed, meta-data description of constraints in Prolog

Families of Global Constraints

- Value Counting
  - alldifferent, global cardinality
- Scheduling
  - cumulative
- Properties of Sequences
  - sequence, no_valley
- Graph Properties
  - circuit, tree

Common Algorithmic Techniques

- Bi-Partite Matchning
- Flow Based Algorithms
- Automata
- Task Intervals
- Reduced Cost Filtering
- Decomposition