Introduction to Constraint Programming

Helmut Simonis

email: helmut.simonis@insight-centre.org
homepage: http://http://insight-centre.org/

Insight SFI Centre for Data Analytics
School of Computer Science and Information Technology
University College Cork
Ireland

ACP Winterschool 2024

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.

Objectives

- Overview of Core Constraint Programming
- Three Main Concepts
  - Constraint Propagation
  - Global Constraints
  - Customizing Search
- Topics will be treated in more detail in later parts of the school
- Based on Examples, not Formal Description

Outline

- Why Constraint Programming?
- Constraint Propagation
- Global Constraints
- Customizing Search

Tutorial Based on ECLiPSe ELearning Course

- Self-study course in constraint programming
- Supported by Cisco Systems and Silicon Valley Community Foundation
- Multi-media format, video lectures, slides, handout etc
- `https://eclipseclp.org/ELearning/index.html`

Also Part of CRT-AI Constraint Week

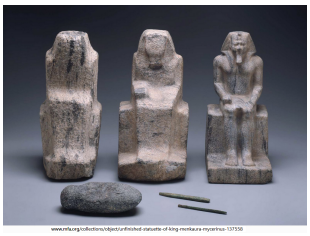
- Annual one week course on CP and Optimization in Ireland
- Part of national training program for PhD students in AI
- `https://www.crt-ai.ie/`

Constraint Programming - in a nutshell

- Declarative description of problems with
  - *Variables* which range over (finite) sets of values
  - *Constraints* over subsets of variables which restrict possible value combinations
  - A *solution* is a value assignment which satisfies all constraints
- Constraint propagation/reasoning
  - Removing inconsistent values for variables
  - Detect failure if constraint can not be satisfied
  - Interaction of constraints via shared variables
  - Incomplete
- Search
  - User controlled assignment of values to variables
  - Each step triggers constraint propagation
- Different domains require/allow different methods

Constraint Programming is Different

- Declarative Programming
  - Concentrate on what you want
  - Not how to get there
  - Program != Algorithm
  - Program = Model
- Applied to Combinatorial Problems
  - No complete polynomial algorithms known (exist?)
  - CP less ad-hoc than heuristics
  - Models can evolve

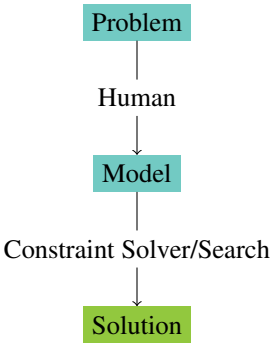


A Subtractive Process

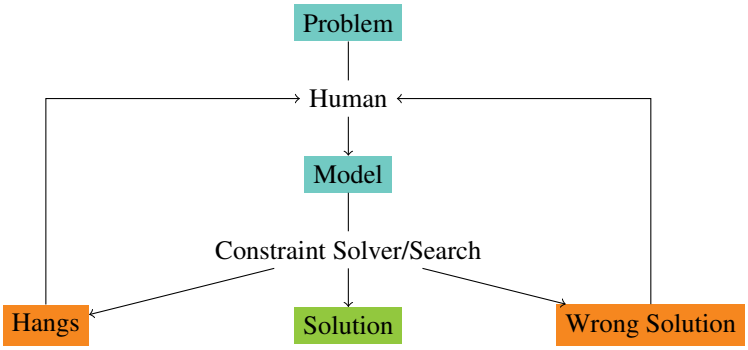
> "Oh, bosh, as Mr. Ruskin says. Sculpture, per se, is the simplest thing in the world. All you have to do is to take a big chunk of marble and a hammer and chisel, make up your mind what you are about to create and chip off all the marble you don't want."-Paris Gaulois.

Source: https://quoteinvestigator.com/2014/06/22/chip-away/

Basic Process



More Realistic



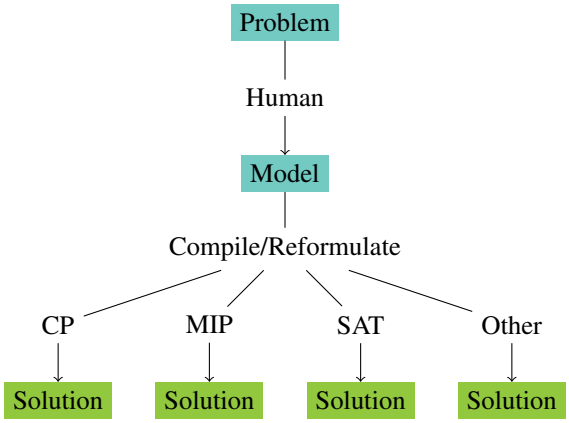
Dual Role of Model

- Allows Human to Express Problem
  - Close to Problem Domain
  - Constraints as Abstractions
- Allows Solver to Execute
  - Variables as Communication Mechanism
  - Constraints as Algorithms

Modelling Frameworks

- MiniZinc (NICTA, Monash University, Australia)
- NumberJack (Insight, Ireland)
- EssencePrime/SavilleRow (UK)
- CPMpy (KU Leuven)
- Allow use of multiple back-end solvers
- Compile model into variants for each solver
- A priori solver independent model(CP, MIP, SAT)

Framework Process



Why use Puzzles as Examples?

- Easy to understand the problem
- Solvable by hand without specialized knowledge
- Possible to compare automated to manual solving process

  The puzzle, though inanimate, is presented as a solvable problem without lasting negative consequences, a very low-risk low-reward situation. By being a puzzle, the object is attempting to convince the user that it must be completed.

Source: Every Day Rhetoric

Part I
Basic Constraint Propagation

Example 1: SEND+MORE=MONEY

- Example of Finite Domain Constraint Problem
- Models and Programs
- Constraint Propagation and Search
- Some Basic Constraints: linear arithmetic, alldifferent, disequality
- A Built-in search
- Visualizers for variables, constraints and search

1 Problem

Let's start with the problem.

Figure 1: Problem Definition

	S	E	N	D
+	M	O	R	E
M	O	N	E	Y

SEND+MORE=MONEY is a crypt-arithmetic puzzle, where characters encode numbers and we have to deduce which character stands for which digit. It is one of the classical, early examples of constraint programming, showing how the constraint reasoning mimics the human reasoning when solving this type of problem. The arithmetic problem is usually shown in the form of a handwritten addition. The puzzle is defined by the following rules:

Rules

- Each character stands for a digit from 0 to 9.
- Numbers are built from digits in the usual, positional notation.
- Repeated occurrence of the same character denote the same digit.
- Different characters denote different digits.
- Numbers do not start with a zero.
- The equation must hold.

2 Program

If we want to model this problem as a finite domain constraint problem, we have to define variables and constraints. *Variables* range over *domains*, which describe the possible values they can take. In this section we consider only finite domains, indeed only finite subsets of natural numbers. We will consider constraint problems with non-integral domains in a later chapter.

An *assignment* is a mapping from the variables to values in their respective domains.

A *constraint* ranges over one or multiple variables and expresses a condition which must hold between these variables. Constraints can take many forms, they may be explicit, describing combinations of values that are allowed or excluded, or symbolic, for example in the form of arithmetic constraints. Formally, we can see a constraint as a subset of the Cartesian product of the domains of its variables. An assignment *satisfies* a constraint if its projection on the variables of the constraint belong to this constrained subset.

An assignment of values to variables is a *solution* if all constraints are satisfied.

For many problems, we have considerable choices in selecting the variables and constraints. Often, models differ dramatically in the number of variables needed, the complexity and number of constraints used to describe the problem, and the speed by which a constraint solver can find a solution.

For the small puzzle considered here, the choice of the variables is fairly obvious: Each character is a variable, which ranges over the values 0 to 9.

For the constraints, a natural description would be:

- An *alldifferent* constraint between all variables, which states that two different variables must have different values. This is a very common constraint, which we will encounter in many other problems later on.
- Two *disequality constraints* (variable X must be different from value V) stating that the variables at the beginning of a number can not take the value 0.
- An arithmetic *equality constraint* linking all variables with the proper coefficients and stating that the equation must hold.

Remember that this is not the only model of this problem, we will discuss some alternatives later on.

SEND+MORE=MONEY Models

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

ECLiPSe Model

```
:- lib(ic).

sendmore1(Digits) :-
    Digits = [S,E,N,D,M,O,R,Y],
    Digits :: [0..9],
    alldifferent(Digits),
    S #\= 0,
    M #\= 0,
                1000*S + 100*E + 10*N + D
              + 1000*M + 100*O + 10*R + E
    #= 10000*M + 1000*O + 100*N + 10*E + Y,
    labeling(Digits).
```

Continue

MiniZinc Model

```
include "alldifferent.mzn";
var 0..9: S;
var 0..9: E;
var 0..9: N;
var 0..9: D;
var 0..9: M;
var 0..9: O;
var 0..9: R;
var 0..9: Y;
constraint S != 0;
constraint M != 0;
constraint           1000 * S + 100 * E + 10 * N + D
                   + 1000 * M + 100 * O + 10 * R + E
       = 10000 * M + 1000 * O + 100 * N + 10 * E + Y;
constraint alldifferent([S,E,N,D,M,O,R,Y]);

solve satisfy;
```

Continue

NumberJack Model (from https://github.com/eomahony/Numberjack/)

```
from Numberjack import *

def get_model():
    model = Model()
    s, m = VarArray(2, 1, 9)
    e, n, d, o, r, y = VarArray(6, 0, 9)
    model.add(      s*1000 + e*100 + n*10 + d +
                    m*1000 + o*100 + r*10 + e ==
          m*10000 + o*1000 + n*100 + e*10 + y)
    model.add(AllDiff((s, e, n, d, m, o, r, y)))
    return s, e, n, d, m, o, r, y, model

def solve(param):
    s, e, n, d, m, o, r, y, model = get_model()
    solver = model.load(param['solver'])
    solver.setVerbosity(param['verbose'])
    solver.solve()
```

Continue

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

```
from cpmpy import *
import numpy as np

s,e,n,d,m,o,r,y = intvar(0,9, shape=8)
model = Model(
    AllDifferent([s,e,n,d,m,o,r,y]),
    (    sum(   [s,e,n,d] * np.array([       1000, 100, 10, 1]) ) \
       + sum(   [m,o,r,e] * np.array([       1000, 100, 10, 1]) ) \
      == sum( [m,o,n,e,y] * np.array([10000, 1000, 100, 10, 1]) ) ),
    s > 0,
    m > 0,
)

model.solve()
```

Continue

Choco-solver Model (from https://choco-solver.org/)

```
Model model = new Model("SEND+MORE=MONEY");
IntVar S = model.intVar("S", 1, 9, false);
IntVar E = model.intVar("E", 0, 9, false);
IntVar N = model.intVar("N", 0, 9, false);
IntVar D = model.intVar("D", 0, 9, false);
IntVar M = model.intVar("M", 1, 9, false);
IntVar O = model.intVar("O", 0, 9, false);
IntVar R = model.intVar("R", 0, 9, false);
IntVar Y = model.intVar("Y", 0, 9, false);

model.allDifferent(new IntVar[]{S, E, N, D, M, O, R, Y}).post();

IntVar[] ALL = new IntVar[]{
    S, E, N, D,
    M, O, R, E,
    M, O, N, E, Y};
int[] COEFFS = new int[]{
    1000, 100, 10, 1,
    1000, 100, 10, 1,
    -10000, -1000, -100, -10, -1};
model.scalar(ALL, COEFFS, "=", 0).post();

Solver solver = model.getSolver();
solver.showStatistics();
solver.showSolutions();
solver.findSolution();
```

Continue

Line **??** shows the module declaration. Each file should be a module, and there should be one module per file. All predicates defined in a module are only visible inside the module, unless they are exported.

Line **??** shows an export directive. It states that the predicate `sendmory` with *arity* one (with one argument) should be exported from this module.

Line **??** contains a library directive. We are loading the `ic` library, which defines the *I*nterval *C*onstraints in the ECLiPSe language.

Lines **??**-**??** define the `sendmory` predicate which is the core of our model. It consists of a single clause with the clause head `sendmory(L)`. The clause has a single argument L, which is a list of variables as defined in line **??**. The square brackets denote lists in Prolog.

We next define the domain of the variables in line **??**. The infix operator `::` takes a variable or list of variables as the first (left) argument, and a domain (here `0..9`) as the second (right) argument.

We then express the `alldifferent` constraint in line **??**. This constraint is a built-in constraint in the `ic` library, and is thus shown in bold.

Lines **??**-**??** express the two disequality constraints. The `ic` library defines the operator `#\=` for this purpose.

The big linear equality constraint is expressed in line **??**. The operator `#=` is used in the `ic` library.

Finally, we call the builtin search routine `labeling` to assign values to the variables.

A Note on Syntax

- Some formulations may seem simpler than others
- This largely is an artifact of a very simple problem
- In most models, you do not write down constraints one by one
- You create constraints based on data
- Ease of integration becomes more important than syntax
- Debugging tools for those who need a debugger :-)

Choice of Model

- This is *one* model, not *the* model of the problem
- Many possible alternatives
- Choice often depends on your constraint system
  - Constraints available
  - Reasoning attached to constraints
- Not always clear which is the *best* model
- Often: Not clear what is the *problem*

The model we have presented here is *one* model of the problem, it is not *the* model of the puzzle. Quite often there is a natural way of expressing the problem, but typically there are still many possible alternatives for expressing particular aspects of a model.

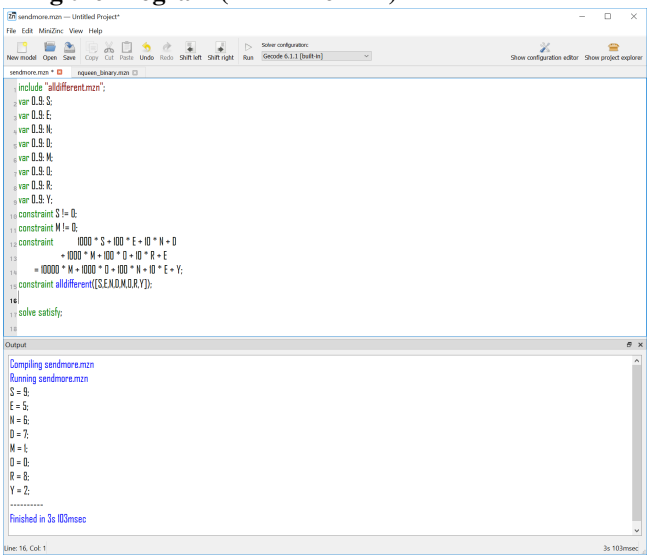
The choice which model to use often depends on your constraint system. A system may or may not contain particular constraints, some constraint system have constraint that others don't have and which are difficult to implement in another system. Some constraint systems have special reasoning attached to some constraints, so that they are much more powerful solving some problems than other systems.

Unfortunately, it is not always clear which is the *best model*, or if there is indeed a best model for a given problem class. Quite often, we don't even know what exactly the problem is, and we can decide which constraints we want to include and which aspects of a problem we can ignore.

One important aspect of constraint programming is that we want to play with the model, experiment by adding or removing some constraints, to really find out which problem we are going to solve and how we are going to do it.

Indeed, for the puzzle here we have developed two alternative models, which are described at the end of this chapter. There are hyperlinks on the slides to look at these alternatives, or you can use the table of contents of the video to look at them now.

Running the Program (MiniZinc IDE)



If we want to run the program, we have to enter a query `sendmory:sendmory(L).` which calls the predicate `sendmory` in the module `sendmory` with a single argument, a free variable L.

The system then returns the answer as a binding for the variable L, and it says `yes`, indicating that it has found a first solution. There may be additional solutions, which can be obtained by backtracking.

Question

- But how did the program come up with this solution?
- We show solution with ECLiPse, other solvers vary slightly

How did the program actually find this solution, what happened in the constraint system before it printed out this message? That is the question we are going to answer in the following sections.

Usually, we are not going to analyze the behaviour of a program in such great detail, but for an introduction I think it is useful to understand how much work can happen inside the constraint engine and how variables, constraints and search interact to find a solution.

3 Constraint Setup

We will now go through the constraint setup, where the constraints are created and perform their initial propagation before the search is started. We are going to use different visualization tools to understand what is happening, this gives a much better conceptual model than tracing through the execution with the line debugger.

3.1 Domain Definition

Domain Definition

```
var 0..9: S;
var 0..9: E;
var 0..9: N;
var 0..9: D;
var 0..9: M;
var 0..9: O;
```

```
var 0..9: R;
var 0..9: Y;
```

When we first encounter our list of variables, we define them to be domain variables with a domain from 0 to 9.

Figure 2: Domain Visualization

	0	1	2	3	4	5	6	7	8	9
S										
E										
N										
D										
M										
O										
R										
Y										

We can represent the variables with a *domain visualizer* (Figure 2). It shows the state of the domain variables at this point. Each line shows one variable, each column one value. Initially, all values are allowed for all variables, this is indicated by white cells.

3.2 Alldifferent Constraint

Alldifferent Constraint

```
include "alldifferent.mzn";

constraint alldifferent([S,E,N,D,M,O,R,Y]);
```

- Built-in alldifferent predicate included
- No initial propagation possible
- *Suspends*, waits until variables are changed
- When variable is fixed, remove value from domain of other variables
- *Forward checking*

3.3 Disequality Constraints

3.4 Equality Constraint

$$
\begin{array}{rrrrr}
 & 1000*S+ & 100*E+ & 10*N+ & D \\
 & +1000*M+ & 100*O+ & 10*R+ & E \\
\hline
10000*M+ & 1000*O+ & 100*N+ & 10*E+ & Y
\end{array}
$$

is transformed into

$$
\begin{array}{rrrrr}
 & 1000*S+ & 91*E+ & & D \\
 & & + & 10*R & \\
\hline
9000*M+ & 900*O+ & 90*N+ & & Y
\end{array}
$$

Simplified Equation

$$1000 * S + 91 * E + 10 * R + D = 9000 * M + 900 * O + 90 * N + Y$$

We now look at the initial propagation of the equality constraint, while noting that the `alldifferent` constraint is suspended, but will be woken when variables are fixed to values.

Propagation

We consider the two sides of the equation, taking the domains of the variables into account.

$$\begin{aligned} 1000 * S^{1..9} + 91 * E^{0..9} + 10 * R^{0..9} + D^{0..9} = \\ 9000 * M^{1..9} + 900 * O^{0..9} + 90 * N^{0..9} + Y^{0..9} \end{aligned}$$

The left hand side ranges from 1000 to 9918, the right hand side from 9000 to 89919.

$$\begin{aligned} \underbrace{1000 * S^{1..9} + 91 * E^{0..9} + 10 * R^{0..9} + D^{0..9}}_{1000..9918} = \\ \underbrace{9000 * M^{1..9} + 900 * O^{0..9} + 90 * N^{0..9} + Y^{0..9}}_{9000..89919} \end{aligned}$$

If the equation holds, then both sides must be equal. We can therefore consider the intersection of the ranges for left and right hand side.

$$\begin{aligned} \underbrace{1000 * S^{1..9} + 91 * E^{0..9} + 10 * R^{0..9} + D^{0..9}}_{9000..9918} = \\ \underbrace{9000 * M^{1..9} + 900 * O^{0..9} + 90 * N^{0..9} + Y^{0..9}}_{9000..9918} \end{aligned}$$

From this we can deduce the following domain restrictions:

Deduction:

$$M = 1, S = 9, O \in \{0..1\}$$

Consider lower bound for S

- Lower bound of equation is 9000
- Rest of lhs (left hand side) $(91 * E^{0..9} + 10 * R^{0..9} + D^{0..9})$ is atmost 918
- S must be greater or equal to $\frac{9000-918}{1000} = 8.082$
  - otherwise lower bound of equation not reached by lhs
- S is integer, therefore $S \geq \lceil \frac{9000-918}{1000} \rceil = 9$
- S has upper bound of 9, so $S = 9$

Consider upper bound of M

- Upper bound of equation is 9918
- Rest of rhs (right hand side) $900 * O^{0..9} + 90 * N^{0..9} + Y^{0..9}$ is at least 0
- M must be smaller or equal to $\frac{9918-0}{9000} = 1.102$
- M must be integer, therefore $M \leq \lfloor \frac{9918-0}{9000} \rfloor = 1$
- M has lower bound of 1, so $M = 1$

Consider upper bound of O

- Upper bound of equation is 9918
- Rest of rhs (right hand side) $9000 * 1 + 90 * N^{0..9} + Y^{0..9}$ is at least 9000
- O must be smaller or equal to $\frac{9918-9000}{900} = 1.02$
- O must be integer, therefore $O \leq \lfloor \frac{9918-9000}{900} \rfloor = 1$
- O has lower bound of 0, so $O \in \{0..1\}$

Figure 3: Propagation of equality: Result

	0	1	2	3	4	5	6	7	8	9
S		-	-	-	-	-	-	-	-	✹
E										
N										
D										
M		✹	-	-	-	-	-	-	-	-
O			✖	✖	✖	✖	✖	✖	✖	✖
R										
Y										

So after the initial reasoning of the equality constraint (Figure 3), we have deduced that two variables (S and M) must have fixed values and that another variable O can only range between 0 and 1.

Figure 4: Propagation of `alldifferent`

	0	1	2	3	4	5	6	7	8	9
S										
E										
N										
D										
M										
O										
R										
Y										

The assignment of the variables will wake up the `alldifferent` constraint. This removes the assigned values 1 and 9 from the domain of the unassigned variables. But variable O could only take values 0 or 1. If we remove value 1, then it must take value 0. This assignment will be propagated through the `alldifferent` constraint and removes the value 0 from the remaining variables (Figure 4.

At this point the equality constraint will be checked again, since some variable bounds have been updated.

For ease of presentation, we first remove the constant parts of the constraint, and obtain a simplified form:

Removal of constants

$$1000 * 9 + 91 * E^{2..8} + 10 * R^{2..8} + D^{2..8} = \\ 9000 * 1 + 900 * 0 + 90 * N^{2..8} + Y^{2..8}$$

$$\mathbf{1000} * \mathbf{9} + 91 * E^{2..8} + 10 * R^{2..8} + D^{2..8} = \\ \mathbf{9000} * \mathbf{1} + \mathbf{900} * \mathbf{0} + 90 * N^{2..8} + Y^{2..8}$$

$$91 * E^{2..8} + 10 * R^{2..8} + D^{2..8} = 90 * N^{2..8} + Y^{2..8}$$

We now go though several steps of updating variable bounds, each triggering a renewed check of the equality constraint.

Propagation of equality (Iteration 1)

$$\underbrace{91 * E^{2..8} + 10 * R^{2..8} + D^{2..8}}_{204..816} = \underbrace{90 * N^{2..8} + Y^{2..8}}_{182..728}$$

$$\underbrace{91 * E^{2..8} + 10 * R^{2..8} + D^{2..8}}_{204..728} = 90 * N^{2..8} + Y^{2..8}$$

$$N \geq 3 = \lceil \frac{204 - 8}{90} \rceil, E \leq 7 = \lfloor \frac{728 - 22}{91} \rfloor$$

Propagation of equality (Iteration 2)

$$91 * E^{2..7} + 10 * R^{2..8} + D^{2..8} = 90 * N^{3..8} + Y^{2..8}$$

$$\underbrace{91 * E^{2..7} + 10 * R^{2..8} + D^{2..8}}_{204..725} = \underbrace{90 * N^{3..8} + Y^{2..8}}_{272..728}$$

$$\underbrace{91 * E^{2..7} + 10 * R^{2..8} + D^{2..8} = 90 * N^{3..8} + Y^{2..8}}_{272..725}$$

$$E \geq 3 = \lceil \frac{272 - 88}{91} \rceil$$

Propagation of equality (Iteration 3)

$$91 * E^{3..7} + 10 * R^{2..8} + D^{2..8} = 90 * N^{3..8} + Y^{2..8}$$

$$\underbrace{91 * E^{3..7} + 10 * R^{2..8} + D^{2..8}}_{295..725} = \underbrace{90 * N^{3..8} + Y^{2..8}}_{272..728}$$

$$\underbrace{91 * E^{3..7} + 10 * R^{2..8} + D^{2..8} = 90 * N^{3..8} + Y^{2..8}}_{295..725}$$

$$N \geq 4 = \lceil \frac{295 - 8}{90} \rceil$$

Propagation of equality (Iteration 4)

$$91 * E^{3..7} + 10 * R^{2..8} + D^{2..8} = 90 * N^{4..8} + Y^{2..8}$$

$$\underbrace{91 * E^{3..7} + 10 * R^{2..8} + D^{2..8}}_{295..725} = \underbrace{90 * N^{4..8} + Y^{2..8}}_{362..728}$$

$$\underbrace{91 * E^{3..7} + 10 * R^{2..8} + D^{2..8} = 90 * N^{4..8} + Y^{2..8}}_{362..725}$$

$$E \geq 4 = \lceil \frac{362 - 88}{91} \rceil$$

Propagation of equality (Iteration 5)

$$91 * E^{4..7} + 10 * R^{2..8} + D^{2..8} = 90 * N^{4..8} + Y^{2..8}$$

$$\underbrace{91 * E^{4..7} + 10 * R^{2..8} + D^{2..8}}_{386..725} = \underbrace{90 * N^{4..8} + Y^{2..8}}_{362..728}$$

$$\underbrace{91 * E^{4..7} + 10 * R^{2..8} + D^{2..8} = 90 * N^{4..8} + Y^{2..8}}_{386..725}$$

$$N \geq 5 = \lceil \frac{386 - 8}{90} \rceil$$

Propagation of equality (Iteration 6)

$$91 * E^{4..7} + 10 * R^{2..8} + D^{2..8} = 90 * N^{5..8} + Y^{2..8}$$

$$\underbrace{91 * E^{4..7} + 10 * R^{2..8} + D^{2..8}}_{386..725} = \underbrace{90 * N^{5..8} + Y^{2..8}}_{452..728}$$

$$\underbrace{91 * E^{4..7} + 10 * R^{2..8} + D^{2..8} = 90 * N^{5..8} + Y^{2..8}}_{452..725}$$

$$N \geq 5 = \lceil \frac{452 - 8}{90} \rceil, E \geq 4 = \lceil \frac{452 - 88}{91} \rceil$$

No further propagation at this point We have reached a fixed-point of the constraint propagation, all constraints have been checked, and no further domain reduction is possible.

Figure 5: Domains after setup

	0	1	2	3	4	5	6	7	8	9
S										
E										
N										
D										
M										
O										
R										
Y										

Figure 5 shows the state of the variables after the constraint setup, three variables have been assigned, and the domains of the unassigned variables have been reduced.

4 Search

We now call the `labeling` routine to assign values to the unassigned variables.

The predicate call `labeling([S,E,N,D,M,O,R,Y])` in line **??** on page **??** will enumerate the variables in the given order, each time starting with the smallest value in the domain. If the assignment of that value fails, the next value will be tried, and so on. If no more values remain, the search will backtrack to the last previous choice and try its next alternative. This will continue until either

- a valid assignment is found for all variables, and the search *succeeds*
- no more untested choices remain, the search *fails*

4.1 Step 1

Figure 6: Search Tree Step 1

Figure 6 shows the search tree at this point. The top node shows the variable being assigned (here S), the label on the edge shows the value that is taken (here 9). The link is dotted, as the variable is already assigned, and no new choice is required.

4.2 Step 2

Figure 7 shows the search tree at this point. We are currently assigning variable E to value 4. As this is a proper choice, a solid link is used.

Assigning E to 4 (Figure 8) will wake the equality constraint

Propagation of $E = 4$, equality constraint

$$91 * 4 + 10 * R^{2..8} + D^{2..8} = 90 * N^{5..8} + Y^{2..8}$$

$$\underbrace{91 * 4 + 10 * R^{2..8} + D^{2..8}}_{386..452} = \underbrace{90 * N^{5..8} + Y^{2..8}}_{452..728}$$

$$\underbrace{91 * 4 + 10 * R^{2..8} + D^{2..8} = 90 * N^{5..8} + Y^{2..8}}_{452}$$

$$N = 5, Y = 2, R = 8, D = 8$$

The state after propagation the equality constraint is shown in Figure 9.

Figure 7: Step 2, Alternative $E = 4$

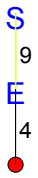


Figure 8: Assignment $E = 4$

	0	1	2	3	4	5	6	7	8	9
S										
E					✹	-	-	-		
N										
D										
M										
O										
R										
Y										

The assignments will trigger the `alldifferent` constraint, but the constraint detects that two variables (D and R) have the same value 8. This is not allowed and the `alldifferent` constraint fails (Figure 10).

We backtrack to the last choice (E), and then try the next value ($E = 5$). This situation is depicted in figure 11. The left child of node E is marked as a failure, we are currently testing value 5, the second alternative. The assignment of value ($E = 5$) will trigger both the `alldifferent` and the equality constraint. Assume that the `alldifferent` is woken first[1].

The propagation of the assignment $E = 5$ in the `alldifferent` constraint will remove value 5 from the variable N (Figure 13). At this point, since the lower bound of variable N is updated to 6, the equality constraint is triggered.

Propagation of equality

$$91 * 5 + 10 * R^{2..8} + D^{2..8} = 90 * N^{6..8} + Y^{2..8}$$

$$\underbrace{91 * 5 + 10 * R^{2..8} + D^{2..8}}_{477..543} = \underbrace{90 * N^{6..8} + Y^{2..8}}_{542..728}$$

$$\underbrace{91 * 5 + 10 * R^{2..8} + D^{2..8} = 90 * N^{6..8} + Y^{2..8}}_{542..543}$$

$$N = 6, Y \in \{2, 3\}, R = 8, D \in \{7..8\}$$

Figure 9: Result of equality propagation

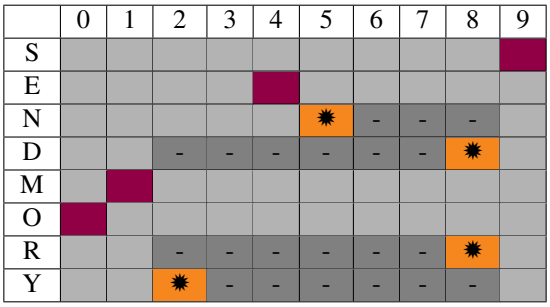
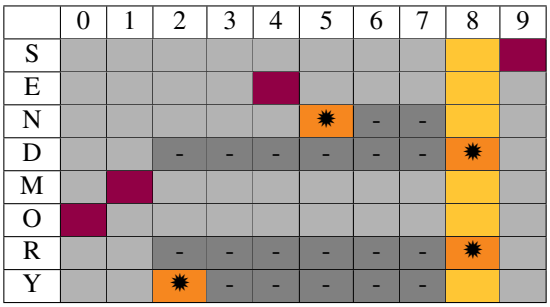


Figure 10: Propagation of `alldifferent` fails!



The `alldifferent` constraint (figure 15) has fixed variable D to 7. We again wake the equality constraint.

Propagation of equality

$$91 * 5 + 10 * 8 + 7 = 90 * 6 + Y^{2..3}$$

$$\underbrace{91 * 5 + 10 * 8 + 7}_{542} = \underbrace{90 * 6 + Y^{2..3}}_{542..543}$$

$$\underbrace{91 * 5 + 10 * 8 + 7 = 90 * 6 + Y^{2..3}}_{542}$$

$$Y = 2$$

This finally assigns the last variable Y to 2 (Figure 16).

4.3 Further Steps

Search Tree with Gecode/GIST

[1] Which of the constraints is woken up first can be difficult to predict without understanding the details of the implementation of the solver

Figure 11: Step 2, Alternative $E = 5$

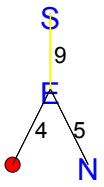
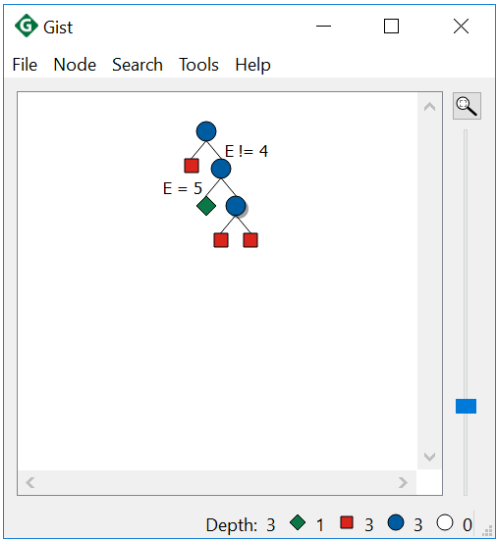


Figure 12: Assignment $E = 5$

	0	1	2	3	4	5	6	7	8	9
S										
E					-	✹	-	-		
N										
D										
M										
O										
R										
Y										



Some Differences

- Uses binary branching
  - var equal value, var not equal value

Figure 13: Propagation of `alldifferent`

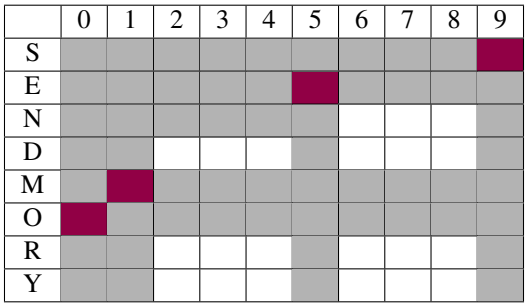
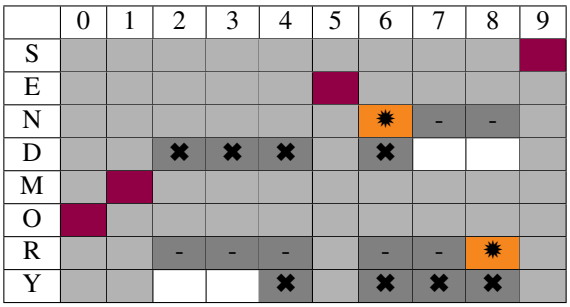


Figure 14: Result of equality propagation



- Solutions in green, failure leafs in red, internal nodes in blue
- By default, shows all failed sub trees collapsed
- By default, uses different search strategy

All variables have been assigned, the search succeeds. Figure 17 shows the resulting search tree. There is only a single variable (E) for which an actual choice had to be made, all other variables are assigned by constraint propagation. The last node at the bottom of the tree is a green success node, indicating that a solution was found.

4.4 Solution

We finally reached a solution (Figure 18), it is easy to check that this indeed solves the puzzle.

Figure 15: Propagation of `alldifferent`

	0	1	2	3	4	5	6	7	8	9
S										
E										
N										
D										
M										
O										
R										
Y										

Figure 16: Last propagation step

	0	1	2	3	4	5	6	7	8	9
S										
E										
N										
D										
M										
O										
R										
Y			✸	-						

Figure 17: Complete Search Tree

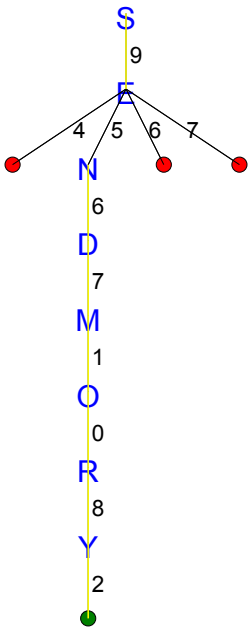


Figure 18: Solution

	9	5	6	7
+	1	0	8	5
1	0	6	5	2

5 Points to Remember

Points to Remember

- Constraint models are expressed by variables and constraints.
- Problems can have many different models, which can behave quite differently. Choosing the best model is an art.
- Constraints can take many different forms.
- Propagation deals with the interaction of variables and constraints.
- It removes some values that are inconsistent with a constraint from the domain of a variable.
- Constraints only communicate via shared variables.

Points to Remember

- Propagation usually is not sufficient, search may be required to find a solution.
- Propagation is data driven, and can be quite complex even for small examples.
- The default search uses chronological depth-first backtracking, systematically exploring the complete search space.
- The search choices and propagation are interleaved, after every choice some more propagation may further reduce the problem.

For Puzzle Purists Only

- We did not follow the puzzler ethos!
- We should solve the puzzle without making choices
- Even a case analysis should be avoided
- The puzzle has a single solution, we should be able to deduce the solution
- In the process shown, we are limited by the underlying assumptions
  - Treat each constraint on its own, they interact only by domains of variables
  - We only use the constraints that we stated in the model
- Can we do better than that?

Skip

Better Reasoning

- Three possible approaches (possibly many more)
  - Full domain reasoning for arithmetic, not just bound reasoning
  - Interaction of *sum* and *alldifferent* constraints
  - Deduced implied constraint

Looking at more than just bounds

- We only considered the smallest and largest values that can be achieved in the sum constraint
- We can do more
  - Can any of the values between be expressed as the sum of the terms
  - Consider holes in the domains, and in the range of possible values for LHS and RHS
- Usually not done in actual solvers for arithmetic constraints
- Easy to do with Dynamic Programming

Consider the interaction of multiple constraints

- Usually ignored, as only interaction is via domains of shared variables
- Here: Sum and *alldifferent* interact
  - When considering the bounds, we cannot assume that each variable takes its smallest/largest value independently
  - Find feasible assignment that minimizes/maximizes the total weight
  - To do this properly, we need some non-trivial reasoning
- Do we do this combined reasoning automatically, or only when prompted by the modeler?

Deduced Implied Constraints

- Look at the partially solved puzzle

```
  9END
+10RE
 ----
```

- `10NEY`
- In the hundreds position, we have $E + 0 + C_{10} = N + 10 * C_{100}$, with C_{10} the 0/1 carry from the tens position
- NB: No carry C_{100} into the thousands, $C_{100} = 0$
- N must be equal to $E + 1$ with $C_{10} = 1$
- If $C_{10} = 0$, then $N = E$, not possible
- We can substitute $N = E + 1$ into our main equation, but keep $N = E + 1$ as well

Expert Mode Reasoning

Starting with

$$91 * E^{4..7} + 10 * R^{2..8} + D^{2..8} = 90 * N^{5..8} + Y^{2..8}$$

we get

$$91 * E^{4..7} + 10 * R^{2..8} + D^{2..8} = 90 * E^{4..7} + 90 + Y^{2..8}$$

Eliminating duplicate occurrences of E

$$\underbrace{E^{4..7} + 10 * R^{2..8} + D^{2..8}}_{26..95} = \underbrace{90 + Y^{2..8}}_{92..98}$$

shared range 92..95 To reach 92, R must be equal to 8, therefore N, E, D, Y must be less than 8

As $N = E + 1$, E must be less than 7

$$E^{4..6} + 10 * 8 + D^{2..7} = 90 + Y^{2..7}$$

Simplification yields

$$\underbrace{E^{4..6} + D^{2..7}}_{6..13} = \underbrace{10 + Y^{2..7}}_{12..17}$$

shared range 12..13 To reach 12 on LHS, E must be greater than 4, D must be greater than 5
To reach 13 in RHS, Y must be smaller than 4

$$E^{5..6} + D^{6..7} = 10 + Y^{2..3}$$

As both N and D must be in 6..7, no other variable can use those values
NB: This requires better reasoning than *forward checking* on the *alldifferent* constraint
If we remove 6 from the domain of E, E must be $E = 5$, and thus $N = 6$, due to $N = E + 1$
Alldifferent forces $D = 7$, leaving

$$5 + 7 = 10 + Y^{2..4}$$

This only leaves $Y = 2$

$$5 + 7 = 10 + 2$$

That is a solution, the constraint is satisfied

Expert Mode Summary

- Often there is more propagation that can be done
- Can be difficult/expensive to do
- Balancing
  - How much work it done at each step of search?
  - How many steps of search you need?
- For hard problems, doing all possible propagation may be exponential
- Not aware that any CP system does the full reasoning shown here

This is how people solve the puzzle by hand

- When writing the first version of this puzzle for CHIP (in 1986), we wanted to mimic the way we solve the puzzle by hand

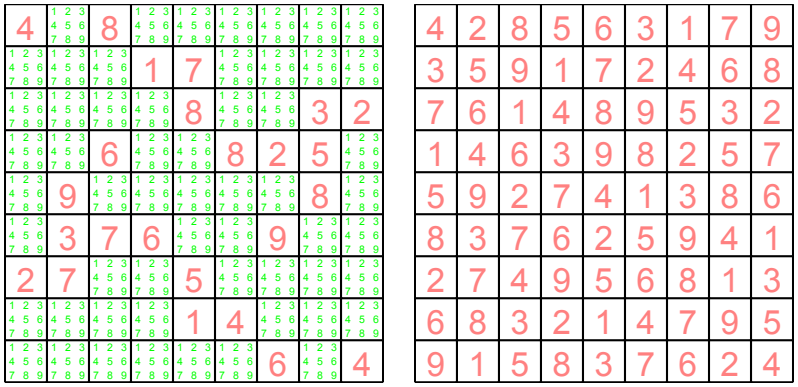
Part II
Global Constraints

Example 2: Sudoku

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

6 Problem

Figure 19: Example Problem and Solution



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(1, [](
    [](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, 1, 7, 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

7 Initial Propagation (Forward Checking)

Figure 20: Initial State (Forward Checking)

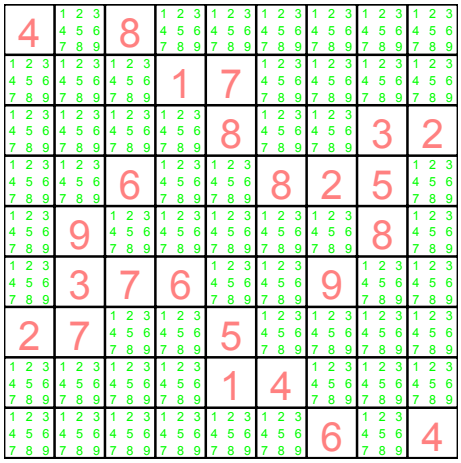


Figure 21: Propagation Steps (Forward Checking)

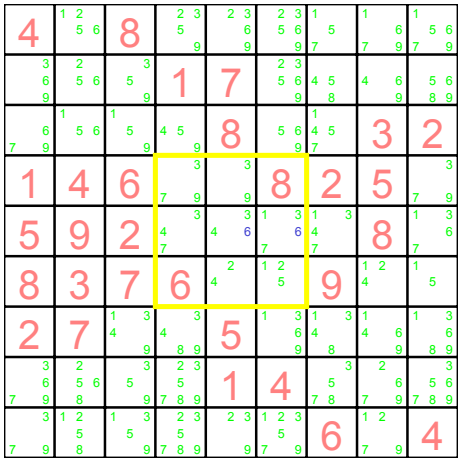
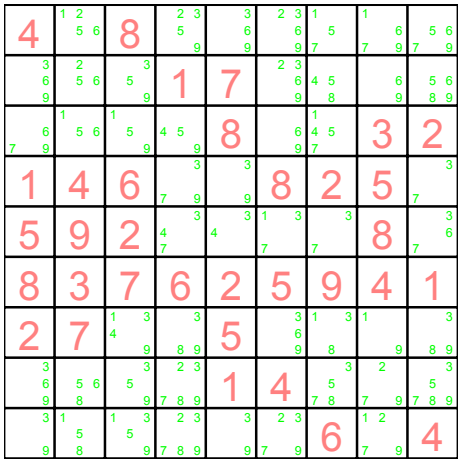


Figure 22: After Setup (Forward Checking)



8 Improved Reasoning

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

8.1 Bounds Consistency

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;
```

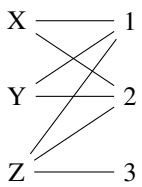
Using Forward Checking

- 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

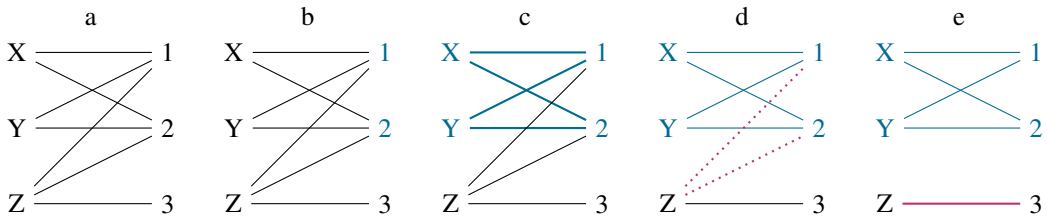
- Show problem as graph with two types of nodes
  - Variables on the left
  - Values on the right
- If value is in domain of variable, show link between them
- This is called a *bipartite* graph

Figure 23: Visualization of alldifferent as bipartite graph



A Simpler Example

Figure 24: A Simpler Example



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

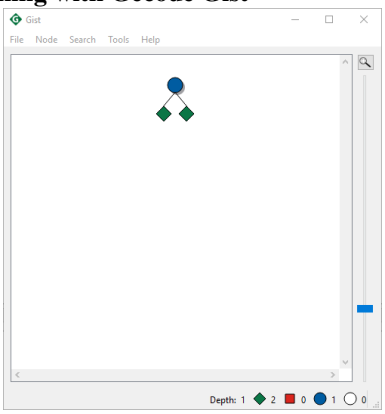
Bounds Consistency

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.

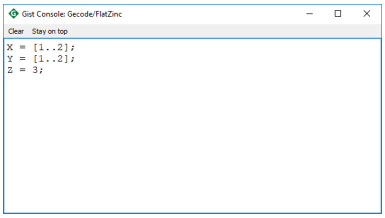
Annotation: :: bounds

```
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)

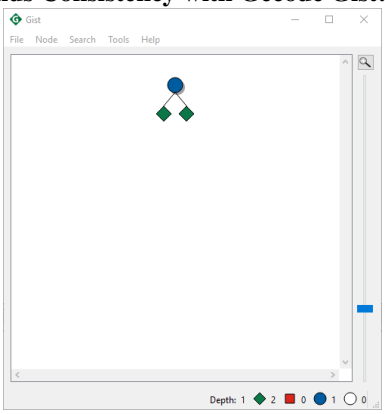
Can we do even better?

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

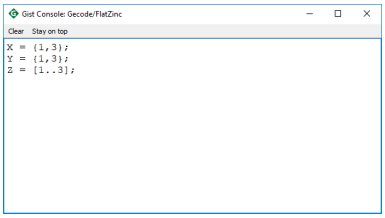
Another Simple Example

```
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;
```

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

Figure 25: Another Example

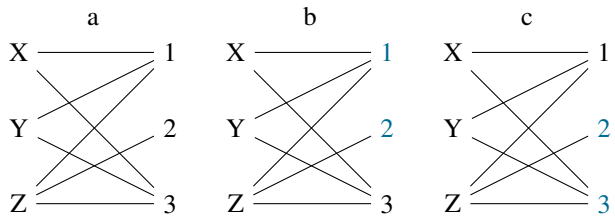
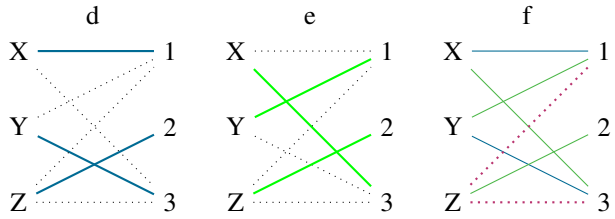


Figure 26: Another Example (Continued)



- 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

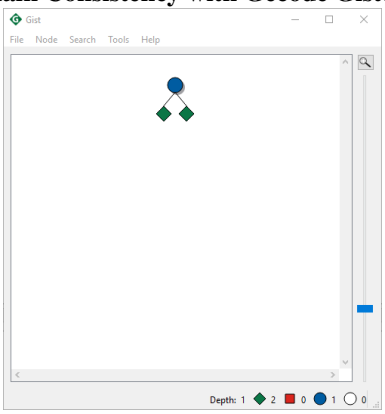
Definition
A constraint achieves *domain consistency*, if for every variable and for every value in its domain, it is possible to find values in the domains of all other variables which satisfy the constraint.

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

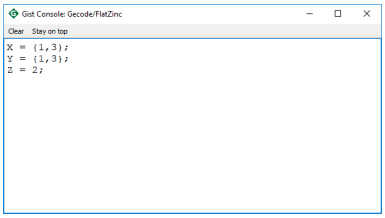
Simple Example Revisited

```
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

8.2 Domain Consistency

Modified MiniZinc Program

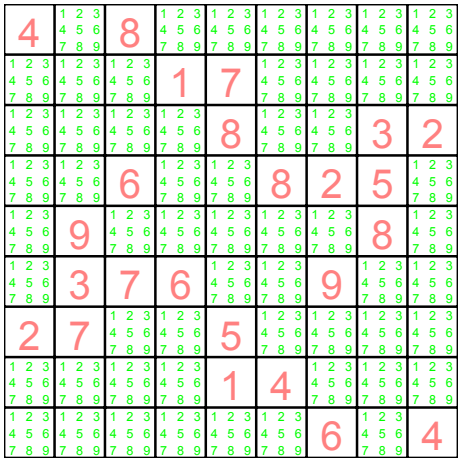
```
int: s;
int: n=s*s;
array[1..n,1..n] of var 1..n: puzzle;

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();
```

Figure 27: Initial State (Domain Consistency)



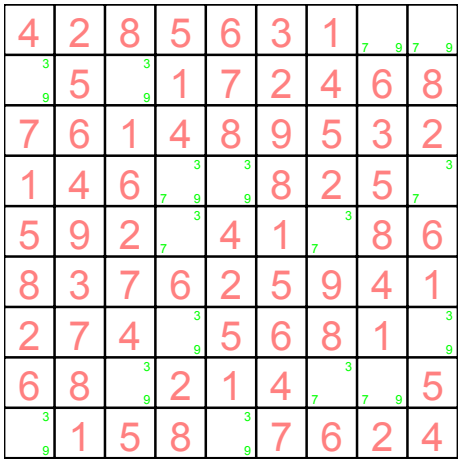
8.3 Comparison

Typical?

- This does not always happen
- Sometimes, two methods produce same amount of propagation
- Possible to predict in certain special cases

Figure 28: Propagation Steps (Domain Consistency)

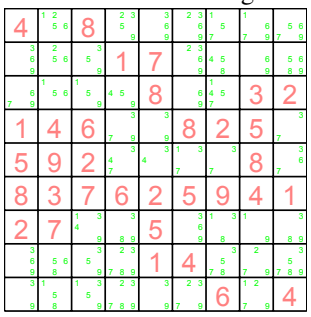
Figure 29: After Setup (Domain Consistency)



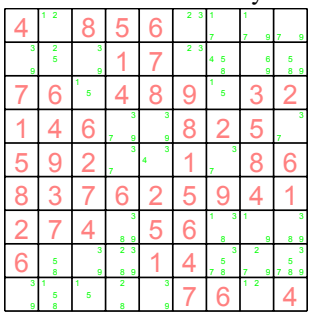
- In general, tradeoff between speed and propagation
- Not always fastest to remove inconsistent values early
- But often required to find a solution at all

Figure 30: Comparison

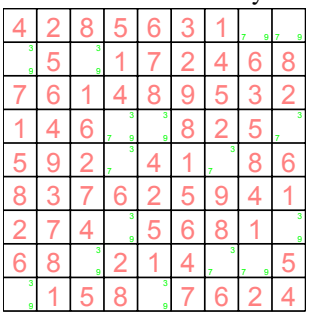
Forward Checking



Bounds Consistency



Domain Consistency



9 Search

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;
```

Figure 31: Search Tree (Forward Checking)

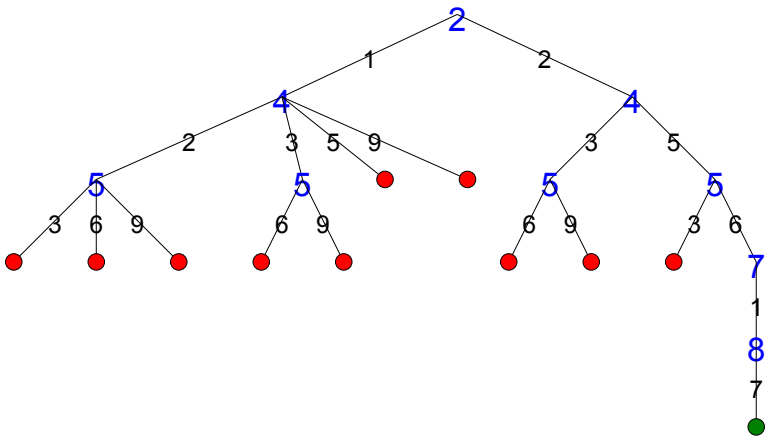
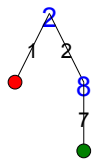


Figure 32: Search Tree (Bounds Consistency)



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

10 Other Global Constraints

Are there other Global Constraints?

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

Figure 33: Search Tree (Domain Consistency)

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

Part III
Customizing Search

What we want to introduce

- Importance of search strategy, constraints alone are not enough
- Two schools of thought
  - Black-box solver, solver decides by itself
  - Human control over process
- Dynamic variable ordering exploits information from propagation
- Variable and value choice
- Hard to find strategy which works all the time
- Different way of improving stability of search routine

Example Problem

- N-Queens puzzle
- Rather weak constraint propagation
- Many solutions, limited number of symmetries
- Easy to scale problem size

As example for the different techniques we use the N-Queens puzzle, a standard problem in Artificial Intelligence and Constraint Programming. The model we use has rather weak constraint propagation, there is not that much information that is deduced from the constraints, so that we rely more on search in solving this problem. The puzzle has many solutions, indeed too many to easily enumerate beyond sizes 10 or 12, and only a limited number of symmetries. A big advantage is that we can easily scale the problem size to create a number of related problem instances, which helps us to understand problems of stability of the methods considered.

11 Problem

Let's look at the problem in a bit more detail.

Problem Definition

8-Queens
Place 8 queens on an 8×8 chessboard so that no queen attacks another. A queen attacks all cells in horizontal, vertical and diagonal direction. Generalizes to boards of size $N \times N$.

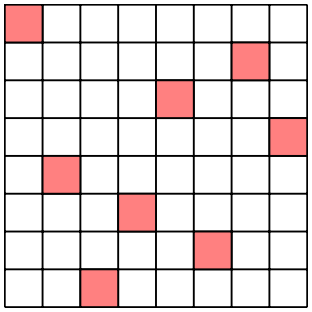
The 8-Queens puzzle is the problem of placing eight queens on a chess board so that no queen attacks another. A queen attacks all cells in horizontal, vertical or diagonal direction. The original puzzle is for an 8×8 chess board. Obviously, we can easily generalize this problem to an arbitary board size $N \times N$.

The diagram (Figure 34) shows a solution for the 8×8 case, the queens are marked in red. It is easy to see that no queen attacks any other queen in this solution.

12 Program

We will now discuss how to model and solve this problem with ECLiPSe.

Figure 34: Solution for board size 8×8



12.1 Model

Basic Model

- Cell based Model
  - A 0/1 variable for each cell to say if it is occupied or not
  - Constraints on rows, columns and diagonals to enforce no-attack
  - N^2 variables, $6N - 2$ constraints
- Column (Row) based Model
  - A 1..N variable for each column, stating position of queen in the column
  - Based on observation that each column must contain exactly one queen
  - N variables, $N^2/2$ binary constraints

There are basically two models for this problem, one based on cells, the other on columns (or rows).

In the first model we introduce 0/1 integer variables for all cells, they describe if a cell contains a queen or not. Linear constraints on the rows, columns and the diagonals enforce that there is at most one queen in each of them. We also have a constraint stating that we must place N queens. This means that we have N^2 variables, and $N + N + 2 * (2N - 1) + 1 = 6N - 1$ constraints.

The alternative to this model is a column (or row) based model, where we have a variable for each column (or row) of the board. This is based on the observation that each column must contain exactly one queen. If a column contains more than one queen, they would attack each other. A column can also not be empty. We need N queens, and we have N columns, each containing atmost one queen. By the *pigeon-hole principle* we know that there can be no empty columns, each must contain exactly one queen. The variable for a column then ranges over the values from 1 to N, and gives the position of the queen in the column.

This choice of model automatically handles the no-attack condition in the vertical direction, so we don't need a special constraint for this. In the horizontal direction we can express the no-attack condition by a binary disequality between any two column variables, i.e. the variables can not have the same values. But this means the variables must be pairwise different, and that is just the definition of the `alldifferent` constraint, which we already encountered in earlier examples. We can therefore express the no-attack condition in horizontal direction by a single `alldifferent` constraint between all variables.

This leaves us with the no-attack condition for the diagonals. They can also be expressed by disequalities, but we have to add an offset between the variables, which is based on their distance $j - i$.

Model

$$\text{assign} \quad [X_1, X_2, ...X_N]$$

s.t.

$$\begin{aligned}
\forall 1 \leq i \leq N: \quad & X_i \in 1..N \\
\forall 1 \leq i < j \leq N: \quad & X_i \neq X_j \\
\forall 1 \leq i < j \leq N: \quad & X_i + j \neq X_j + i \\
\forall 1 \leq i < j \leq N: \quad & X_i + i \neq X_j + j
\end{aligned}$$

We can write down our model in a concise mathematical form as a set of constraints and use this as the basis for our ECLiPSe program.

13 Naive Search

Nqueens Models

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

ECLiPSe N-Queens Model

```
:- lib(lists).
:- lib(ic).
:- lib(ic_search).

top:-
    queens(8,Board),
    search(Board, 0, input_order, indomain, complete).

queens(N, Board) :-
    length(Board, N),
    Board :: 1..N,
    ( fromto(Board, [Q1|Cols], Cols, []) do
        ( foreach(Q2, Cols), param(Q1), count(Dist,1,_) do
            noattack(Q1, Q2, Dist)
        )
    ).

noattack(Q1,Q2,Dist) :-
    Q2 #\= Q1,
    Q2 - Q1 #\= Dist,
    Q1 - Q2 #\= Dist.
```

Continue

MiniZinc N-Queens Model

```
int: n=8;
array[1..n] of var 1..n: queens;
constraint
    forall(i, j in 1..n where i < j) (
         queens[i] != queens[j] /\
         queens[i] + i != queens[j] + j /\
         queens[i] - i != queens[j] - j
    )
 ;
solve :: int_search(
        queens,
        input_order,
        indomain_min)
        satisfy;
```

Continue

NumberJack N-Queens Model

```
from Numberjack import *

def get_model(N):
    queens = VarArray(N, N)
    model = Model(
        AllDiff(queens),
        AllDiff([queens[i] + i for i in range(N)]),
        AllDiff([queens[i] - i for i in range(N)])
    )
    return queens, model

def solve(param):
    queens, model = get_model(param['N'])
    solver = model.load(param['solver'])
    solver.setHeuristic(param['heuristic'], param['value'])
    solver.setVerbosity(param['verbose'])
    solver.setTimeLimit(param['tcutoff'])
    solver.solve()
```

Continue

CPMpy N-Queens Model

```
def nqueens_naive(n=8):
    queens = IntVar(1,n, shape=n)

    model = Model()
    for i in range(n):
        for j in range(i):
            model += [queens[i] != queens[j],
                  queens[i] + i != queens[j] + j,
                  queens[i] - i != queens[j] - j,
                  ]
```

Continue

Choco-solver N-Queens Program

```
int n = 8;
Model model = new Model(n + "-queens problem");
IntVar[] vars = new IntVar[n];
for(int q = 0; q < n; q++){
    vars[q] = model.intVar("Q_"+q, 1, n);
}
for(int i  = 0; i < n-1; i++){
    for(int j = i + 1; j < n; j++){
        model.arithm(vars[i], "!=",vars[j]).post();
        model.arithm(vars[i], "!=", vars[j], "-", j - i).post();
        model.arithm(vars[i], "!=", vars[j], "+", j - i).post();
    }
}
Solution solution = model.getSolver().findSolution();
if(solution != null){
    System.out.println(solution.toString());
}
```

Continue

When it comes to search, it does not matter which variant of the program we use. They create the same constraints, and assign the variables in the same order from X_1 to X_N. We will now use some visualization to see how the search routine finds its first solution.

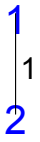
We see (in Figure 35) the search tree on the left, and the current partial assignment on the right. As the first assignment we have fixed X_1 to value 1. This is shown as a red rectangle on the board on the right, indicating the value selected. The currently assigned variable is represented by a yellow outline. The constraint propagation after the assignment has removed values from the other variables. The removed values are shown in blue, the remaining values in the domains are shown as green.

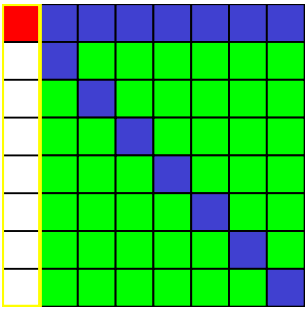
In the second assignment step (Figure 36), we assign a value for variable X_2. We use value 3, since the values 1 and 2 have been removed from the domain by the previous assignment.

In the next step (Figure 37), we encounter a problem. When we try to assign value 5 to variable X_3, we detect a failure in constraint propagation. This is why the variable is outlined in red on our board, it is also shown by a red leaf node in the search tree. After the failure, we backtrack to the last open choice, this is variable X_3, and try the next available value. In this way we continue our search, stopping whenever we detect a failure and continuing until all variables are assigned or no more choices are left.

Finally (Figure 38) we find a solution after quite a few backtracking steps. On the right we see the solution found, on the left the search tree explored. Some links in the tree are shown in yellow, this means they are forced assignments made by constraint propagation. If we go back a few steps, we sees that as soon as variable X_4 is assigned to 6, the remaining variables are assigned by propagation, so we don't have to choose values for those variables.

Figure 35: First Assignment





Observations

- Even for small problem size, tree can become large
- Not interested in all details
- Ignore all automatically fixed variables
- For more compact representation abstract failed sub-trees

What we can see is that even for small problem sizes, the search tree can become quite large. We are normally not interested in all details of the propagation and the search, so we will tell the system to ignore things like automatically fixed variables.

Sometimes we also want to have a shorter representation of the search, a tree in the form shown here may become too wide to fit on our display. In that case, we can choose a more compact representation, which compresses all failed subtrees into little red triangles.

We get a view like this (Figure 39). We have our variable X_1, which we assign to value 1, we then choose variable X_2, for which we first try value 3. This leads to a dead subtree, which does not contain any solutions. This is marked by a triangle, the number inside gives the number of interior nodes (here 4) and the number below gives the number of failures in the subtree (here 7).

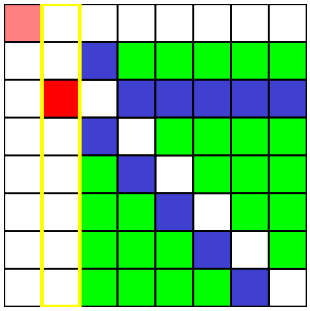
Eventually we come to a solution in the green solution node at the bottom, after having assigned four variables.

Exploring other board sizes

- How stable is the model?
- Try all sizes from 4 to 100
- Timeout of 100 seconds

One interesting question is "How stable is this model?", can we use it to solve the problem for different board sizes. To make this experiment, we try out all sizes from 4 to 100, with a timeout of 100 seconds. Board sizes 2 and 3 do not allow solutions, you may want to check this yourself on a piece of paper.

Figure 36: Second Assignment



In Figure 40 we have a plot of the solutions we obtained. The board size is on the x-axis, time required on the y-axis. We see that initially, we find solutions quite rapidly, up to size 20. For larger sizes, the times vary a lot, for size 25 we use less than a second, but for size 28 we need 25 seconds. More importantly, we don't find any solutions for sizes greater than 30 within the timeout limit of 100 seconds.

This problem is not just linked to problem size. We would expect the program to take longer for a bigger board, but this is not a simple monotonic function. Some problem sizes require much more time to solve than others. What can we do about this?

Figure 37: Third Assignment

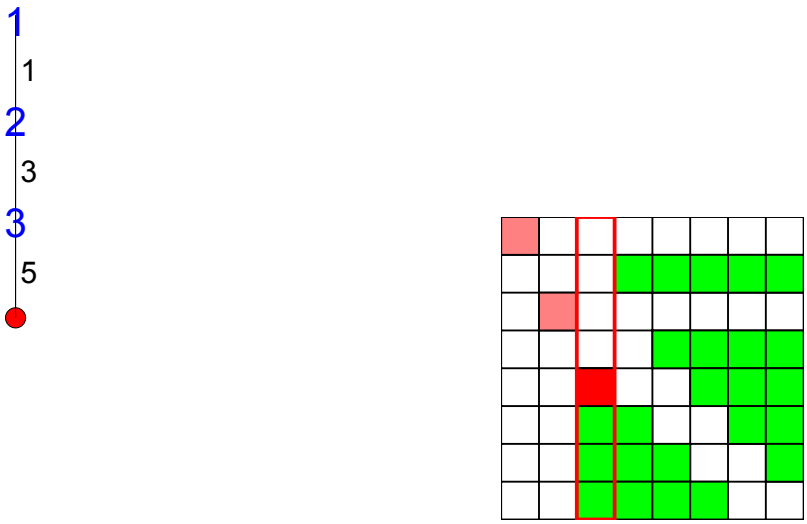


Figure 38: First Solution

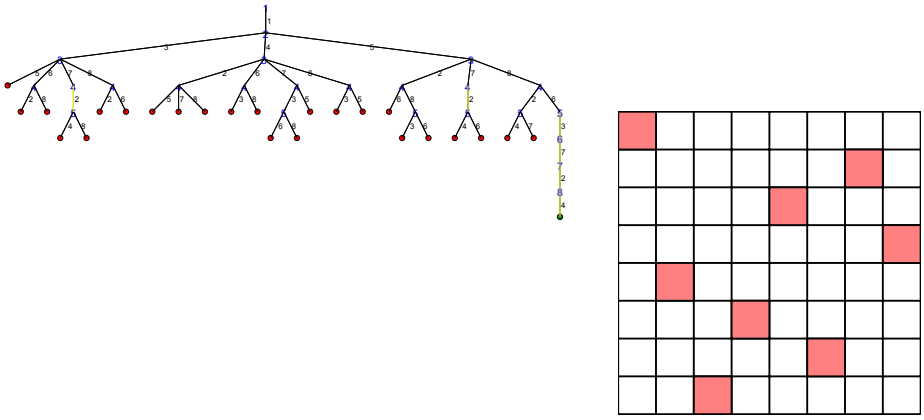


Figure 39: Compact Tree Presentation

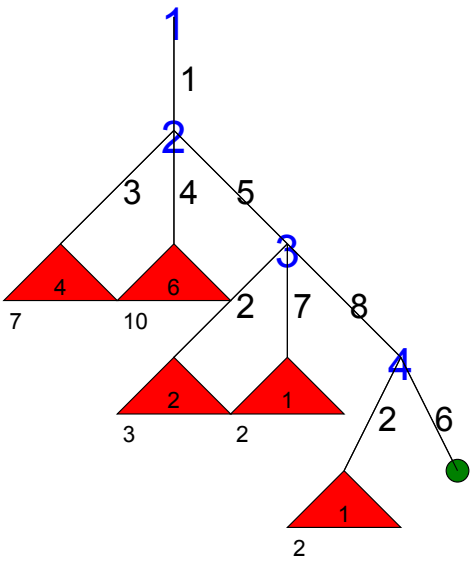
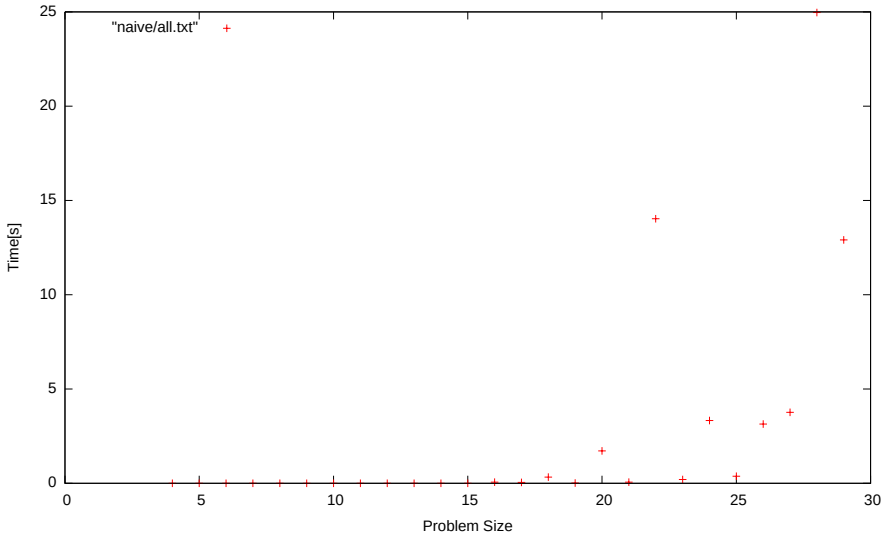


Figure 40: Naive Stategy, Problem Sizes 4-100



14 Improvements

14.1 Dynamic Variable Choice

Can we do better than our naive, straighforward program?

Possible Improvements

- Better constraint reasoning
  - Remodelling problem with 3 `alldifferent` constraints
  - Global reasoning as described before
- Better control of search
  - Static vs. dynamic variable ordering
  - Better value choice
  - Not using complete depth-first chronological backtracking

There are two ways of improving the situation. One would use better constraint reasoning, so that we detect failures earlier and do not have to explore large, dead subtrees. A way to do this is to remodel our problem with three `alldifferent` constraints, and to use the stronger consistency methods that we have explored in the previous chapter. This is not working that well for this problem, and we do not explore this here.

A more promising alternative is to improve our control of the search, introducing a dynamic instead of a static variable ordering. We can also use a better value choice, selecting a more promising value for the selected variable. Finally, we can replace our complete, chronological backtracking search with a more appropriate method.

Static vs. Dynamic Variable Ordering

- Heuristic Static Ordering
  - Sort variables before search based on heuristic
  - Most important decisions
  - Smallest initial domain
- Dynamic variable ordering
  - Use information from constraint propagation
  - Different orders in different parts of search tree
  - Use all information available

We can improve the search routine by either sorting the variables before the search is started based on a heuristic, static ordering or by using a dynamic variable ordering during search, or indeed by a combination of both methods.

For a static ordering, the idea is to make important decisions early in the search, i.e. to choose those variables which have the biggest impact on the search. By making these choices early, we reduce the problem size as quickly as possible. We can also try to assign the variables with small domains first, so that we have fewer alternative branches to explore at the top of the tree. The drawback of such a static heuristic is that we make rather uninformed choices. We can only use the information available at the beginning of the program.

It is usually much better to make the variable selection choice dynamically, this way we can use the current state of the propagation to guide our choice. This can also mean that we use a different ordering in one part of the search tree from the one in another part of the tree. This way we use all information available at every step.

First Fail strategy

- Dynamic variable ordering
- At each step, select variable with smallest domain
- Idea: If there is a solution, better chance of finding it
- Idea: If there is no solution, smaller number of alternatives
- Needs tie-breaking method

One such dynamic selection method is the first fail strategy. At each step, we select the variable which has the smallest domain. There are typically two justifications given for this:

- If there are very few solutions, we have a better chance of finding it by branching on few alternatives.
- If there are no solutions, then we have fewer alternatives to explore, and this should require less time than choosing a variable with a large domain, and exploring many alternatives.

This first fail principle needs a tie-breaking method, to decide what to do if several variables have the same, smallest domain size. In ECLiPSe, we typically use the first variable which has the smallest domain.

Search Stategy Choices

- Minizinc Show
- Choco-solver Show

Modified MiniZinc Program

```
int: n=8;
array[1..n] of var 1..n: queens;
constraint
    forall(i, j in 1..n where i < j) (
         queens[i] != queens[j] /\
         queens[i] + i != queens[j] + j /\
         queens[i] - i != queens[j] - j
    )
 ;
solve :: int_search(
        queens,
        first_fail,
        indomain_min)
        satisfy;
```

Variable Choice (MiniZinc)

- `input_order` assign variables in static order given
- `smallest` assign variable with smallest value in domain first
- `first_fail` select variable with smallest domain first
- `dom_w_deg` consider ratio of domain size and failure count
- Others, including programmed selection for specific solvers

Value Choice (MiniZinc)

- Determines the order in which values are tested for selected variables
- `indomain_min` Start with smallest value, on backtracking try next larger value
- `indomain_median` Start with value closest to middle of domain
- `indomain_random` Choose values in random order
- `indomain_split` Split domain into two intervals

Continue

Modified Choco-solver Model

```
int n = 8;

Model model = new Model(n + "-queens problem");
IntVar[] vars = model.intVarArray("Q", n, 1, n, false);
IntVar[] diag1 = IntStream.range(0, n).
                           mapToObj(i -> vars[i].sub(i).intVar()).
                           toArray(IntVar[]::new);
IntVar[] diag2 = IntStream.range(0, n).
                           mapToObj(i -> vars[i].add(i).intVar()).
                           toArray(IntVar[]::new);

model.post(
    model.allDifferent(vars),
    model.allDifferent(diag1),
    model.allDifferent(diag2)
);

Solver solver = model.getSolver();
solver.showStatistics();
solver.setSearch(Search.domOverWDegSearch(vars));
Solution solution = solver.findSolution();

if (solution != null) {
    System.out.println(solution.toString());
}
```

VariableSelector Choice (Choco-solver)

- Determines the order in which variables are assigned
- `InputOrder` assign variables in static order given
- `Smallest` assign variable with smallest value in domain first
- `FirstFail` select variable with smallest domain first
- `DomOverWDeg` consider ratio of domain size and failure count
- `ActivityBased` dynamic, based on dynamic observed behaviour
- `ImpactBased` dynamic, based on dynamic observed behaviour

IntValueSelector Choice (Choco-solver)

- Determines the order in which values are tested for selected variables
- `IntDomainMin` Start with smallest value, on backtracking try next larger value
- `IntDomainMiddle` Start with value closest to middle of domain
- `IntDomainRandom` Choose values in random order
- `IntDomainRandomBound` Randomly choose between smallest and largest value

Continue

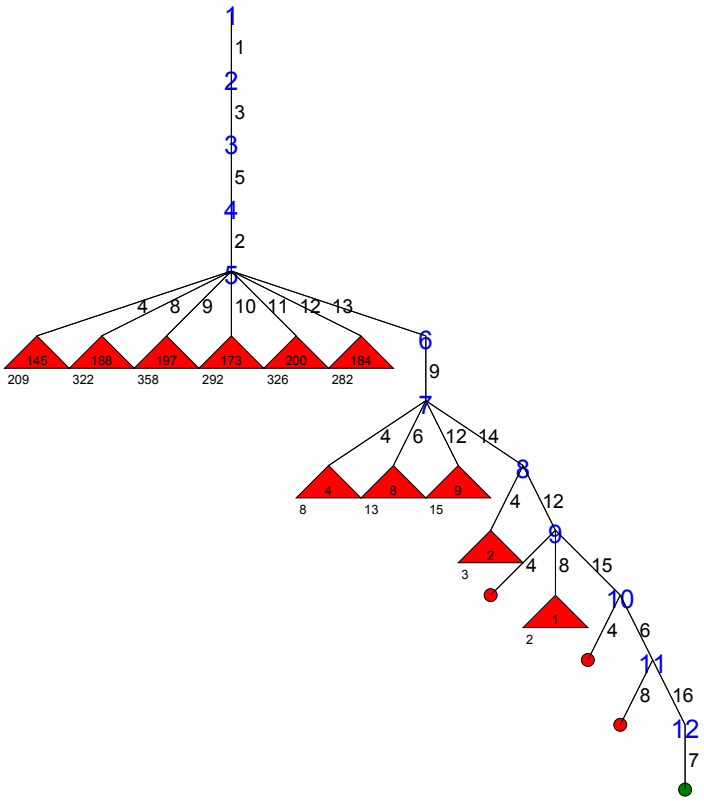
What changes are required to use the first fail princple in our example problem? We have to replace the `labeling` routine with a more general method, the `search` builtin of the ic library, where we give the list of variables to assign, and then a whole set of parameters.

Comparison

- Board size 16x16
- Naive (Input Order) Strategy
- First Fail variable selection

If we want to see how these methods compare, we should use a bigger problem size than the 8×8 board. For demonstration purposes, we use the 16×16 board. We start with the naive method, `input_order` and `indomain` and compare it to the `first_fail` strategy.

Figure 41: Naive (Input Order) Strategy (Size 16) Search Tree



This (Figure 41) is the search tree required for the naive method. This does not look too bad, until you check the size of the failed sub-trees. For variable X_5 we explore 6 failed alternatives with hundreds of failures each, before we make the correct choice.

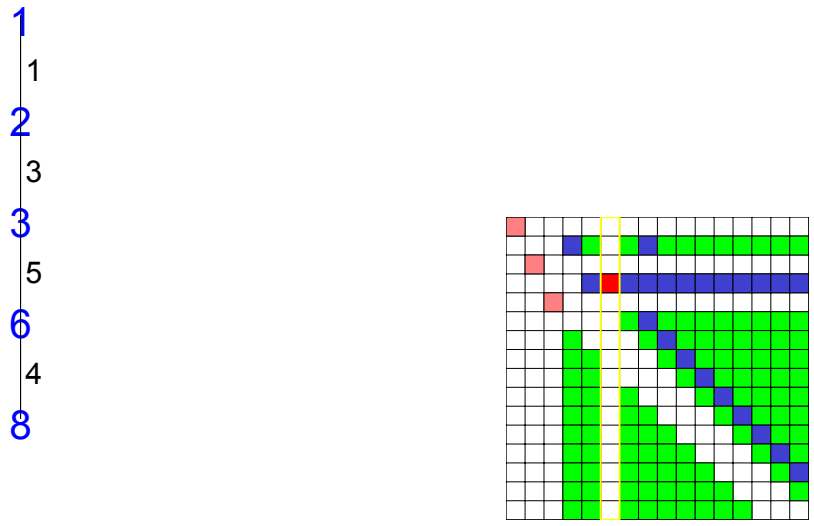
This (Figure 42) is the fourth assignment step in our modified program, and we can see that instead of variable X_4 the program selects X_6 for assignment. From there, the search differs dramatically from the naive method.

If we look at the complete tree for the first fail strategy (Figure 43), we see it is much smaller, requiring only 7 failures before finding a solution.

Also note that the variable order differs in the branches of the tree; on the left side, variable 7 is selected just before the failure, on the right side variable 14 is selected at that level.

It is also important to note that the solutions obtained by the two programs are different, as shown in figure 44. We might be just lucky finding a solution earlier with first fail than the naive method.

Figure 42: First Fail, Fourth Assignment



For this problem we don't care which solution we obtain, we only look for a feasible solution, which satisfies all constraints. For other problems, we may be interested in optimizing a cost function. In that case it is important to generate the feasible solutions in a good order, finding solutions with low cost early on.

We again test our program on all instances from 4 to 100. Here (Figure 45) we have the plot of the execution times required. This is much better than before, note the different time scale on the left. We often find solutions nearly instantly, right up to size 100. But there are still some problems which are more difficult to solve. We still don't find solutions for sizes 88, 91, 93, 97, 98, 99 within the 100 second time limit.

Using the dynamic variable selection with the first fail strategy was a very successful idea, but it does not solve the problem completely.

Figure 43: FirstFail Strategy (Size 16)

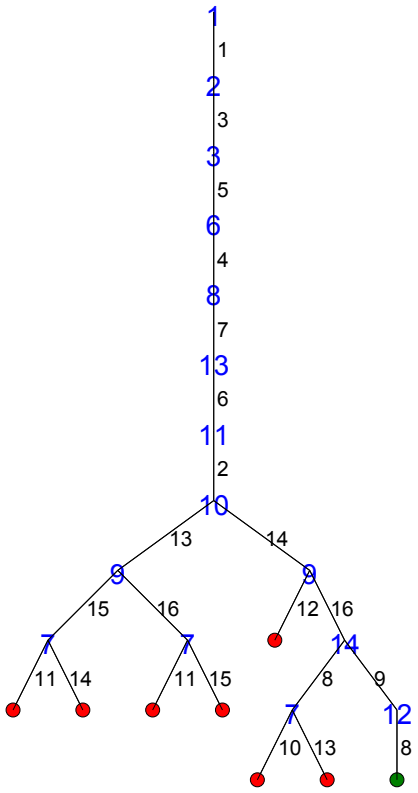


Figure 44: Solutions are different!

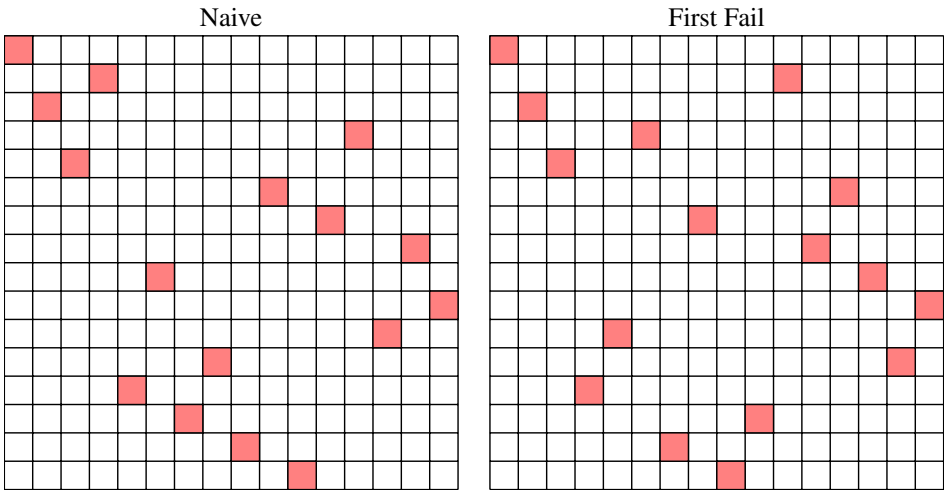
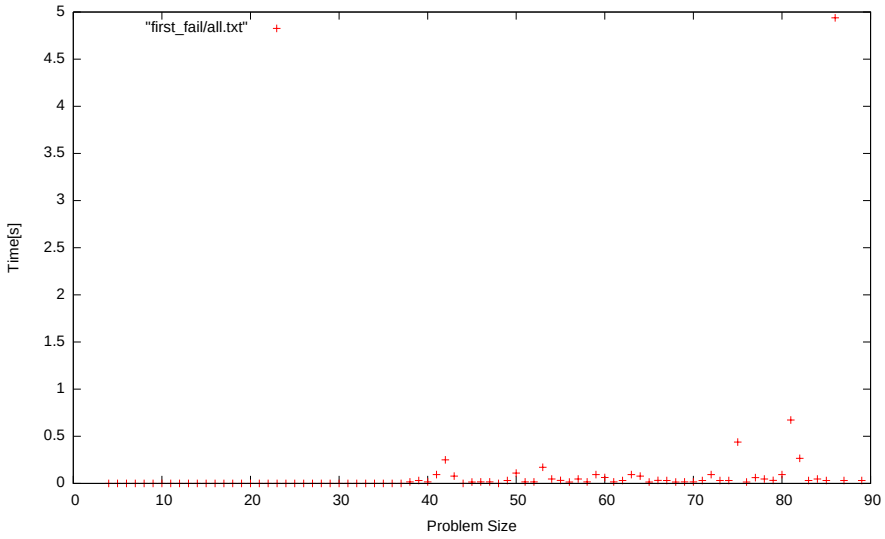


Figure 45: FirstFail, Problem Sizes 4-100



14.2 Weighted Degree

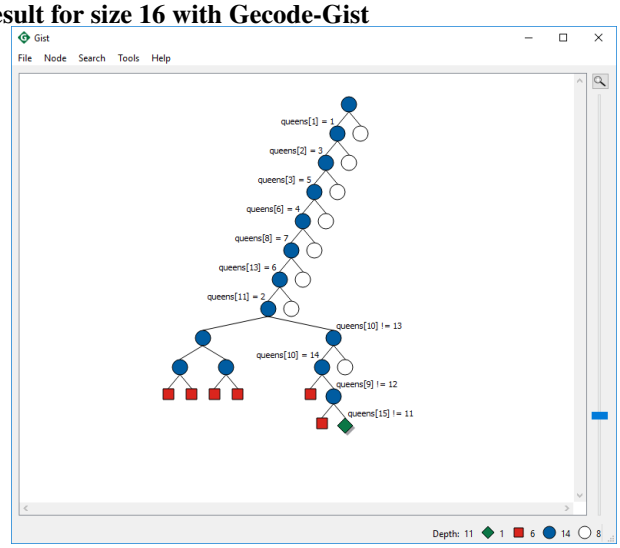
More Reactive Variable Selection

- Domain size is important, but other information is useful as well
- Dom/Weighted Degree: better results in many situations
- Weight Degree: count how often variable has been involved in failure
- Focus on more complicated part of problem
- Changes during search, learns from past performance
- Option **dom_w_deg**

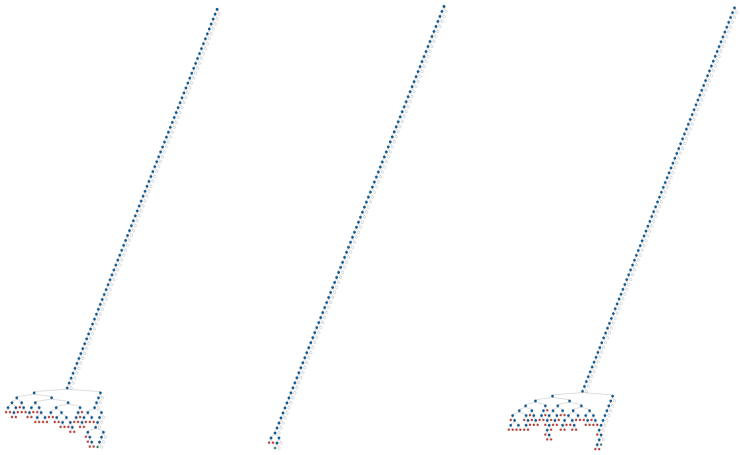
Weighted Degree Variable Selection

```
int: n=8;
array[1..n] of var 1..n: queens;
constraint
    forall(i, j in 1..n where i < j) (
         queens[i] != queens[j] /\
         queens[i] + i != queens[j] + j /\
         queens[i] - i != queens[j] - j
    )
 ;
solve :: int_search(
        queens,
        dom_w_deg,
        indomain_random)
        satisfy;
```

Result for size 16 with Gecode-Gist



Sample Results for Larger Sizes



Size 93 Size 94 Size 95

14.3 Improved Heuristics

14.4 Making Search More Stable

We will now consider some methods to improve the stability of our program.

Approach 1: Heuristic Portfolios

- Try multiple strategies for the same problem
- With multi-core CPUs, run them in parallel
- Only one needs to be successful for each problem

A first idea is to use a portfolio of search strategies, and to run them in parallel. With multi-core CPUs, it is quite possible to use four or more search routines at the same time, and to stop the overall search when one of them has found a solution.

An important point is to select just a few, quite different strategies for the portfolio. We might even be able to select good methods automatically based on the structure of the problem that we are trying to solve. This is an active research area at the moment.

Approach 2: Restart with Randomization

- Only spend limited number of backtracks for a search attempt
- When this limit is exceeded, restart at beginning
- Requires randomization to explore new search branch
- Randomize variable choice by random tie break
- Randomize value choice by shuffling values
- Needs strategy when to restart

Random Variable Choice and Restarts

```
int: n=8;
array[1..n] of var 1..n: queens;
constraint
    forall(i, j in 1..n where i < j) (
         queens[i] != queens[j] /\
         queens[i] + i != queens[j] + j /\
         queens[i] - i != queens[j] - j
    )
 ;
solve :: int_search(
        queens,
        dom_w_deg,
        indomain_random)
        :: random_linear(100)
        satisfy;
```

A perhaps more resource-friendly method uses restart with randomization. The idea is to allow only a limited number of backtracks for a given search attempt. If that limit is exceeded, then the search should restart at the beginning, perhaps exploring another part of the search tree. If the problem has many solutions, we are likely to find one with such a strategy even if constraint propagation is not very powerful.

In order to work, this requires some form of randomization, otherwise we would explore the same part of the search tree over and over again. We can randomize the variable selection, for example by a random tie break, and/or randomize value selection. We also need to decide how much search to allow for each restart attempt.

We will consider such a randomized search routine later in the course.

Approach 3: Partial Search

- Abandon depth-first, chronological backtracking
- Don't get locked into a failed sub-tree
- A wrong decision at a level is not detected, and we have to explore the complete subtree below to undo that wrong choice
- Explore more of the search tree
- Spend time in promising parts of tree

A third way of improving search behavior is to abandon the depth-first chronological backtracking. The idea is not to get locked into a failed sub-tree, which might contain many, many nodes. This happens when we make a wrong descision at some step, but don't detect this immediately. In an ideal world we would just strengthen our constraint reasoning to detect such failures, but unfortunately this is a hard problem that we can't solve in general. Instead, we are left inside a subtree which has no solution, and we have to spend a lot of time exploring all the remaining alternatives, before we have a chance of finding a solution eventually.

In a partial search, we don't continue in such a case, but spend our time exploring other, perhaps more promising parts of the search tree as well.

Example: Credit Search

- Not available in all solvers
- Explore top of tree completely, based on credit
- Start with fixed amount of credit
- Each node consumes one credit unit
- Split remaining credit amongst children
- When credit runs out, start bounded backtrack search
- Each branch can use only K backtracks
- If this limit is exceeded, jump to unexplored top of tree

We can see (in Figure 46) the typical behavior of the credit based search. We start with an initial branch, which at some point starts backtracking. We give up on this branch, jump back to the unexplored part of the credit search and choose another branch there. This may lead to a failure again, but eventually we make a good choice there and find a solution.

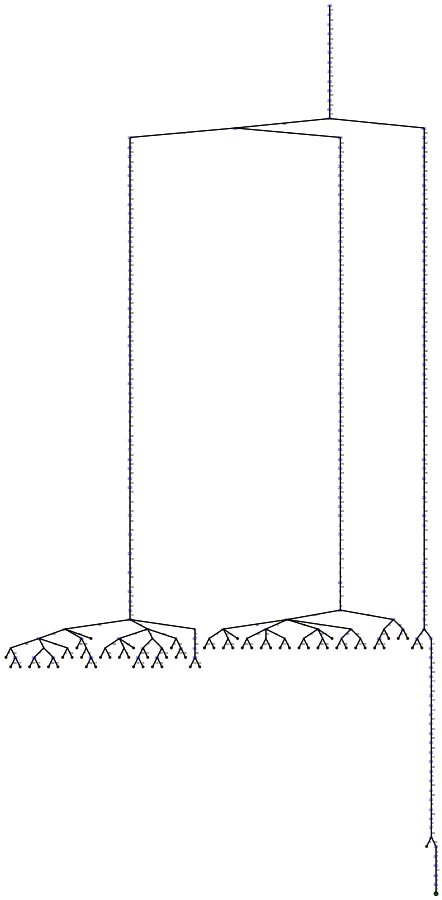
In this example, we have only just started to explore the top part of the tree, there are many unexplored choices left there.

This technique works best if the solutions are not too sparse, and the initial choices really have a big impact on the overall solution obtained. A potential problem is that we might give up too easily with our limited backtracking steps, never finding a solution at all.

We can continue to run the program for larger board sizes, this (Figure 47) shows results for all sizes up to 200. The maximal time needed for any instance is just over 2 seconds.

Nobody really needs to solve the problem with 200 queens or more, so our solution is probably good enough in that sense. But if we really want to solve bigger problem instances, we can use a repair based technique which we will discuss in a later chapter.

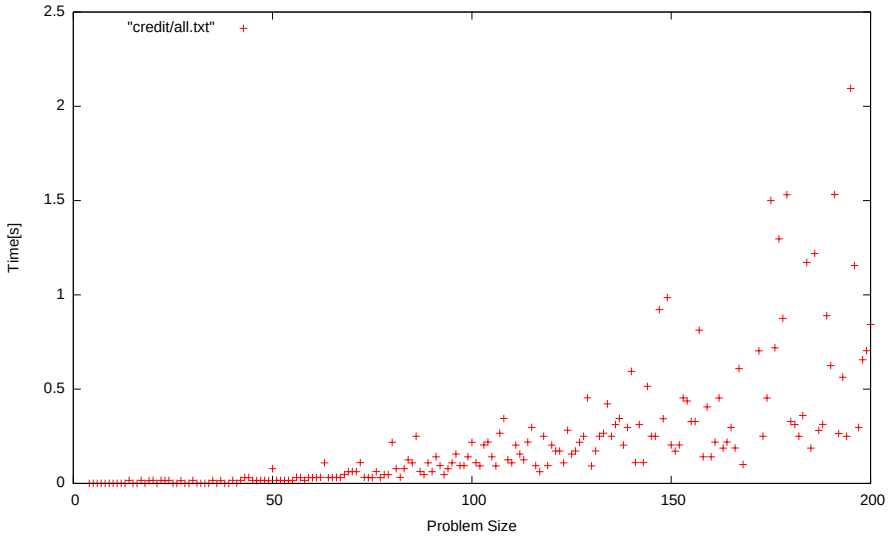
Figure 46: Credit, Search Tree Problem Size 94



Points to Remember

- Choice of search can have huge impact on performance
- Dynamic variable selection can lead to large reduction of search space
- Packaged search can do a lot, but programming search adds even more
- Depth-first chronological backtracking not always best choice
- How to control this explosion of search alternatives?

Figure 47: Credit, Problem Sizes 4-200



Part IV
What is missing?

Many Specialized Topics

- How to design efficient core engine
- Hybrids with LP/MIP tools
- Hybrids with SAT
- Symmetry breaking
- Use of MDD/BDD to encode sets of solutions
- High level modelling tools
- Debugging/visualization

Reformulation

- Just because the user has modelled it this way, it doesn't mean we have to solve it that way
  - Replace some constraint(s) by other, equivalent constraints
  - Because we don't have that constraint in our system
  - For performance

Learning

- While solving the problem we can learn how to strengthen the model/search
  - Understand which constraints/method contribute to propagation and change schedule
  - Learn no-good constraints by explaining failure
  - Adapt search strategy based on search experience

More Learning Resources

- Survey of Methods, Resources, and Formats for Teaching Constraint Programming
  - by Tejas Santanam, Helmut Simonis
  - `https://doi.org/10.48550/arXiv.2403.12717`
  - Based on survey of community for WTCP 2023
  - `https://hsimonis.github.io/WTCP2023/`