Constraint Programming

Emmanuel Hebrard

Toulouse

Outline

1. Constraint Programming
2. Modelisation
3. Propagation
4. Search
5. Practice

Outline

1. **Constraint Programming**
   - Principles
   - Definitions
2. Modelisation
3. Propagation
4. Search
5. Practice

Combinatorial Optimisation

- Finite search space, find a solution that:
  - satisfy some constraints
  - optimize an objective

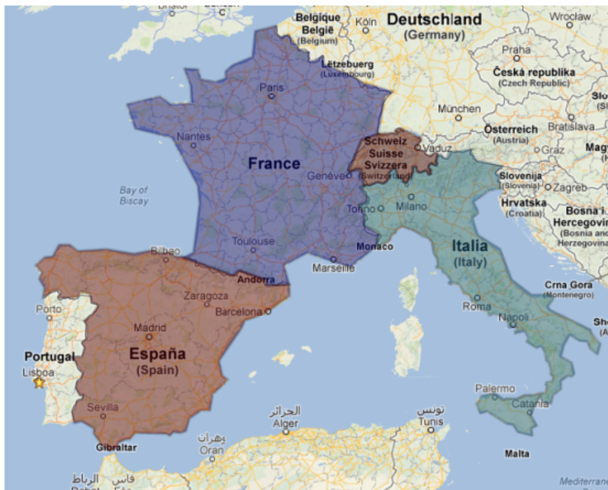
Approaches

- Ad-hoc algorithms: a search algorithm for each problem
- Restriction of the problem formulation:
  - Integer Linear Programming (linear inequations)
  - Boolean Satisfiability (conjunctive normal form)
- Decomposition into known sub-problems
  - A constraint is a sub-problem that we know how to solve efficiently
  - Constraint propagation is the inference method that allow to solve the composite problem

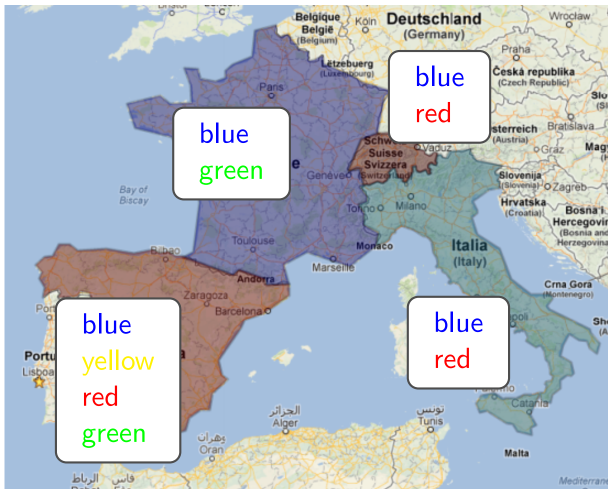
Constraint Programming

- (Almost) no restriction on the constraints
  - Checkable in polynomial time ($\in NP$)
  - Often a known relation for which efficient algorithms exist
- Tree search (backtracking algorithms)
  - Take a decision – an arbitrary reduction of the problem $\rightarrow$ recursion (left branch)
  - If the left branch did not succeed, deduce the complementary reduction $\rightarrow$ recursion (right branch)
- Constraint propagation
  - Given a constraint, detect inconsistent variable/value pairs and remove them
  - Used before each decision, to prune the set of possible decisions

Map Coloring



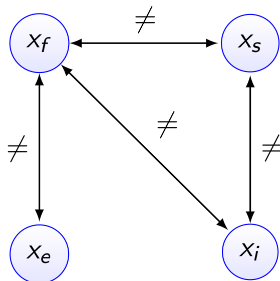
Map Coloring



Map Coloring

$\mathcal{D}(x_f)$: blue, green

$\mathcal{D}(x_s)$: blue, red



$\mathcal{D}(x_e)$: blue, yellow, red, green

$\mathcal{D}(x_i)$: blue, red

Map Coloring

```
from Numberjack import *
import Mistral

france = Variable(['blue','green'], 'france')
switzerland = Variable(['blue','red'], 'switzerland')
spain = Variable(['blue','yellow','red','green'], 'spain')
italy = Variable(['blue','red'], 'italy')

model = Model(
    france != switzerland,
    france != italy,
    france != spain,
    italy != switzerland
    )

solver = Mistral.Solver(model)

if solver.solve():
    for var in [france, switzerland, spain, italy]:
        print name(var), 'in', value(var)
```

Definitions: Constraint Network

- A Constraint Network $\mathcal{P}$ is a triplet $(\mathcal{X}, \mathcal{D}, \mathcal{C})$, where:
  - $\mathcal{X}$ is a set of variables $\{x_1, \ldots, x_n\}$
  - $\mathcal{D}$ is a domain on $\mathcal{X}$, that is, a set $\{\mathcal{D}(x_1), \ldots, \mathcal{D}(x_n)\}$
    - where $\mathcal{D}(x_i) \subset \mathbb{Z}$ is the finite set of values that x_i can take
  - $\mathcal{C}$ is a set of constraints $\{c_1, \ldots, c_m\}$ defining possible relations between variables
- A Constraint c is a pair $(\mathcal{X}(c), \mathcal{R}(c))$ where:
  - $\mathcal{X}(c)$ is a sequence of variables. The length of the sequence $\mathcal{X}(c) = (x_{i_1}, \ldots, x_{i_k})$ is called the arity of the constraint c
  - $\mathcal{R}(c)$ is a relation of arity k over $\mathbb{Z}$, that is, a subset of $\mathbb{Z}^k$

Outline

1. Constraint Programming
2. Modelisation
   - Python
   - Numberjack
   - Expression Tree
     - Ex: Solving XKCD Knapsack
3. Propagation
4. Search
5. Practice

Python

- Interpreted
  - Can call compiled libraries (Modules)
  - Constraint solvers are modules
- Not typed
- Object oriented

Indentation

- Indentations are important in Python
- It defines the block structure

```
understood = False
tries = 0
while not understood:
  if tries > 3:
    break #through
  else:
    print 'still thinking about it'
    tries += 1
print 'got it!'
```

Data Structures

- **list []**: Extendable array

```
primes = [2, 3, 5, 7, 11]
cities = ["Cork", "Toulouse", "Ithaca"]
print cities[2]
>>> "Ithaca"

print cities[1:]
>>> ["Toulouse", "Ithaca"]
```

- **tuple ()**: Fixed size arrays

```
cities = ("Cork", "Toulouse", "Ithaca")
```

- **dict {}**: Indexed by arbitrary objects

```
rainfall = {'Cork':1207, 'Toulouse':632,
'Ithaca':1016}
```

Loops

```
for item in <iterable>:
   do_something_with(item)
```

```
scores = [("Villa", 5),("Sneijder", 5),("Henry", 0)]
for player, goals in scores:
print player, " scored ", goals, " goals"
```

List comprehensions

- Define a list in intention

```
print range(4)
>>> [0, 1, 2, 3]
print [x*x for x in range(4)]
>>> [0, 1, 4, 9]
```

- List comprehensions generally take the following form:

```
[function(x) for x in <Iterable> (if <condition>)]
```

Overview

Module

- Numberjack is a module
- Each solver is a module

```
from Numberjack import *
import Mistral
```

Model

- Variables
- List of constraints

```
X = Variable( [2,4,5,8,10] )
model = Model( X != Y, ... )
model.add( Z == 2*x1 + x2 - 3*x3 )
```

Solver

- Build from a model

```
solver = Mistral.Solver( model )
solver.solve()
```

Expression Tree

Variables - Leaves of the expression tree

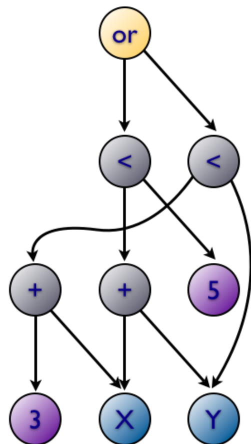
```
X = Variable(0,10)
X = Variable([1,3,5,7])
```

Predicates - Internal nodes of the tree

```
P = X+Y          # arythmetic value
Q = X+3 <= Y     # truth (logic) value
```

Constraints - Root of the expression tree

```
C1 = (X+Y < 5) | (X+3 < Y)
C2 = AllDiff([x,y,z])
C3 = Sum([a,b,c,d]) >= e
```



XKCD Knapsack

MY HOBBY:
EMBEDDING NP-COMPLETE PROBLEMS IN RESTAURANT ORDERS

CHOTCHKIES RESTAURANT

APPETIZERS

MIXED FRUIT	2.15
FRENCH FRIES	2.75
SIDE SALAD	3.35
HOT WINGS	3.55
MOZZARELLA STICKS	4.20
SAMPLER PLATE	5.80

SANDWICHES

BARBECUE 6.55

XKCD Knapsack

```
from Numberjack import *
import Mistral

price = [215, 275, 335, 355, 420, 580]
appetizers = ["Mixed Fruit", "French Fries", "Side Salad",
              "Hot Wings", "Mozzarella Sticks", "Sample Plate"]
total = 1505
num_appetizers = len(appetizers)

quantities = [Variable(0, 1505/price[i], '#'+appetizers[i])
              for i in range(num_appetizers)]

model = Model(
    Sum([quantities[i] * price[i] for i in range(num_appetizers)]) == total
    )

solver = Mistral.Solver(model)
```

Outline

1. Constraint Programming
2. Modelisation
3. Propagation
   - Arc Consistency
   - Bounds Consistency
     - Ex: Golomb Ruler
   - Dedicated Propagators
     - AllDifferent
     - Sum
     - Ex: Return to Golomb Ruler
4. Search
5. Practice

Definitions: Constraint Network

- A Constraint Network $\mathcal{P}$ is a triplet $(\mathcal{X}, \mathcal{D}, \mathcal{C})$, where:
  - $\mathcal{X}$ is a set of variables $\{x_1, \ldots, x_n\}$
  - $\mathcal{D}$ is a domain on $\mathcal{X}$, that is, a set $\{\mathcal{D}(x_1), \ldots, \mathcal{D}(x_n)\}$
    - ★ where $\mathcal{D}(x_i) \subset \mathbb{Z}$ is the finite set of values that x_i can take
  - $\mathcal{C}$ is a set of constraints $\{c_1, \ldots, c_m\}$ defining possible relations between variables
- A Constraint c is a pair $(\mathcal{X}(c), \mathcal{R}(c))$ where:
  - $\mathcal{X}(c)$ is a sequence of variables. The length of the sequence $\mathcal{X}(c) = (x_{i_1}, \ldots, x_{i_k})$ is called the arity of the constraint c
  - $\mathcal{R}(c)$ is a relation of arity k over $\mathbb{Z}$, that is, a subset of $\mathbb{Z}^k$

Definitions: Constraints

- A relation is a subset of $\mathbb{Z}^k$, that is, a list of tuples
  - For instance we can define a constraint $c(x, y, z)$
    - where $\mathcal{X}(c) = (\quad x \quad y \quad z \quad)$
    - and

$$\mathcal{R}(c) = \{ \begin{array}{ccc} 0 & 0 & 1 \\ 0 & 1 & 0 \\ 0 & 1 & 1 \\ 1 & 0 & 0 \\ 1 & 0 & 1 \\ 1 & 1 & 0 \\ 1 & 1 & 1 \end{array} \}$$

Definitions: Constraints

- A relation is a subset of $\mathbb{Z}^k$, that is, a list of tuples
  - Relations can also be defined intentionally:
    - The relation $<$ defines the infinite relation $\{(v, w) \mid v < w\}$

$$\} \quad \begin{array}{cc} -\infty & -\infty + 1 \\ \vdots & \vdots \\ -\infty & \infty \\ \vdots & \vdots \\ 1 & 2 \\ 1 & 3 \\ 1 & 4 \\ \vdots & \vdots \\ 1 & \infty \\ 2 & 3 \\ \vdots & \vdots \\ \infty - 1 & \infty \end{array} \quad \{$$

Definitions: Global Constraints

- Some classes of constraints corresponds to a family of relations
  - For instance ALLDIFFERENT
  - Every variable must take a different value
    - ★ One relation for each arity k

$$\left[\{(v_1, \ldots, v_k) \mid \forall 1 \leq i \neq j \leq k,\ v_i \neq v_j\} \mid k \in \mathbb{N}^+\right]$$

$$\ldots \quad \left\{ \begin{matrix} \vdots & \vdots \\ 0 & 1 \\ 1 & 0 \\ 0 & 2 \\ \vdots & \vdots \\ 5 & 7 \\ 7 & 5 \\ \vdots & \vdots \end{matrix} \right\} \quad \ldots \quad \left\{ \begin{matrix} \vdots & \vdots & \vdots & \vdots \\ 1 & 2 & 3 & 4 \\ 1 & 2 & 4 & 3 \\ 1 & 3 & 2 & 4 \\ \vdots & \vdots & \vdots & \vdots \\ 8 & 1 & 6 & 3 \\ 8 & 6 & 1 & 3 \\ \vdots & \vdots & \vdots & \vdots \end{matrix} \right\} \quad \ldots$$

Definitions: Global Constraints

- Some classes of constraints corresponds to a family of relations
  - For instance AMONG
  - The number of occurrences of values in a set $\mathcal{V}$ is in the range $[l, u]$
    - ⋆ One relation for each arity k, and for each value of some parameters (A set of values $\mathcal{V}$ and two integers l, u)

$$\left[\{(v_1, \dots, v_k) \mid l \leq |\{v_1, \dots, v_k\} \cap \mathcal{V}| \leq u\} \mid u, l \in \mathbb{N}^+, \mathcal{V} \in 2^{\mathbb{Z}} \right]$$

$$\begin{array}{ccccccccccccc} & \} & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & & \\ & & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & & \\ & & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 2 & & \\ & & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 3 & & \\ \dots & & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & & \dots \\ & & 5 & 5 & 0 & 7 & 8 & 1 & 1 & 7 & 0 & & \\ & & 5 & 5 & 0 & 7 & 8 & 1 & 2 & 7 & 0 & & \\ & & 5 & 5 & 0 & 7 & 8 & 1 & 3 & 7 & 0 & & \\ & & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \{ & \end{array}$$

Definitions: Solutions

- Given a constraint network $\mathcal{P} = (\mathcal{X}, \mathcal{D}, \mathcal{C})$:
  - An instantiation σ on a set $\mathcal{Y} = \{x_1, \ldots, x_k\}$ of variables is a mapping from variables to values.
  - An instantiation σ on $\mathcal{Y}$ is said valid iff $\forall x_i \in \mathcal{Y},\ \sigma(x_i) \in \mathcal{D}(x_i)$
  - An instantiation σ on $\mathcal{Y}$ violates a constraint c iff $\mathcal{X}(c) \subseteq \mathcal{Y}$ and $\sigma(\mathcal{X}(c)) \notin \mathcal{R}(c)$
  - An instantiation σ on $\mathcal{Y}$ is said consistent iff it is valid and it does not violate any constraint in $\mathcal{C}$
  - A solution to $\mathcal{P}$ is a consistent instantiation of $\mathcal{X}$

Arc Consistency (AC)

- Let $\mathcal{P} = (\mathcal{X}, \mathcal{D}, \mathcal{C})$ be constraint network, and c a constraint in $\mathcal{C}$
  - A valid tuple σ of the constraint c is called a support of c
    - i.e., A solution σ of the constraint network $(\mathcal{X}(c), \mathcal{D}, \{c\})$
  - A value $v \in \mathcal{D}(x)$ is viable iff there exists a support σ of c
  - A domain $\mathcal{D}$ is Arc Consistent iff $\forall x \in \mathcal{X},\ \forall v \in \mathcal{D}(x),\ v$ is viable

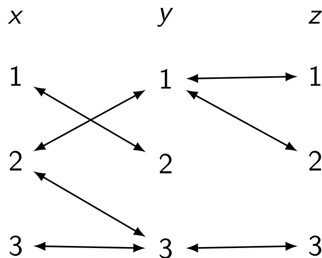
Arc Consistency (AC)

- Let $\mathcal{P} = (\mathcal{X}, \mathcal{D}, \mathcal{C})$ be constraint network, and c a constraint in $\mathcal{C}$
  - A valid tuple σ of the constraint c is called a support of c
    - i.e., A solution σ of the constraint network $(\mathcal{X}(c), \mathcal{D}, \{c\})$
  - A value $v \in \mathcal{D}(x)$ is viable iff there exists a support σ of c
  - A domain $\mathcal{D}$ is Arc Consistent iff $\forall x \in \mathcal{X},\ \forall v \in \mathcal{D}(x),\ v$ is viable
- Iteratively removing non-viable values of a network $\mathcal{P}$ converges toward a unique fix-point
  - The largest Arc Consistent subdomain of $\mathcal{P}$
  - (AC closure of $\mathcal{P}$)

Arc-Consistency

Support

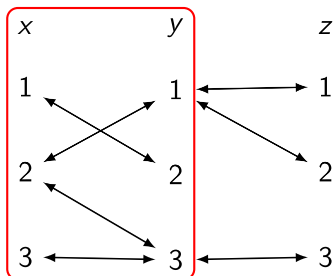
- A support σ of a value v for a constraint c is a solution of this constraint such that every value is itself arc-consistent
  - Propagation until reaching a fix point



Arc-Consistency

Support

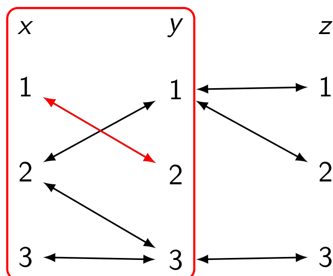
- A support σ of a value v for a constraint c is a solution of this constraint such that every value is itself arc-consistent
  - Propagation until reaching a fix point



Arc-Consistency

Support

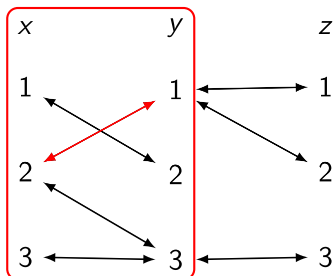
- A support σ of a value v for a constraint c is a solution of this constraint such that every value is itself arc-consistent
  - Propagation until reaching a fix point



Arc-Consistency

Support

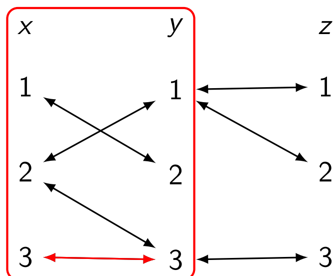
- A support σ of a value v for a constraint c is a solution of this constraint such that every value is itself arc-consistent
  - Propagation until reaching a fix point



Arc-Consistency

Support

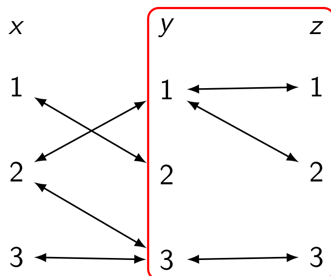
- A support σ of a value v for a constraint c is a solution of this constraint such that every value is itself arc-consistent
  - Propagation until reaching a fix point



Arc-Consistency

Support

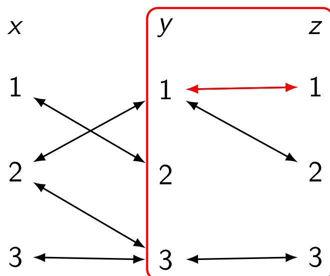
- A support σ of a value v for a constraint c is a solution of this constraint such that every value is itself arc-consistent
  - Propagation until reaching a fix point



Arc-Consistency

Support

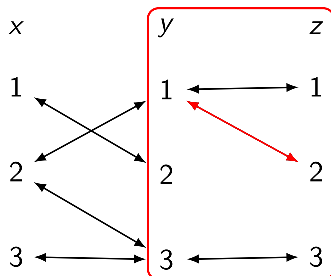
- A support σ of a value v for a constraint c is a solution of this constraint such that every value is itself arc-consistent
  - Propagation until reaching a fix point



Arc-Consistency

Support

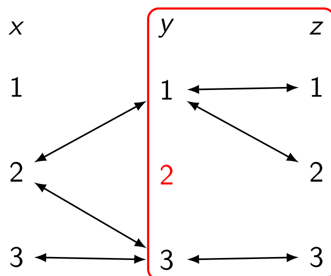
- A support σ of a value v for a constraint c is a solution of this constraint such that every value is itself arc-consistent
    - Propagation until reaching a fix point



Arc-Consistency

Support

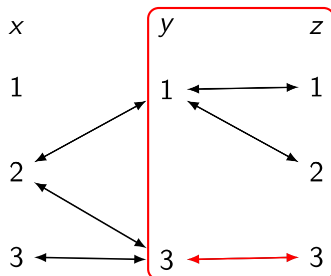
- A support σ of a value v for a constraint c is a solution of this constraint such that every value is itself arc-consistent
  - Propagation until reaching a fix point



Arc-Consistency

Support

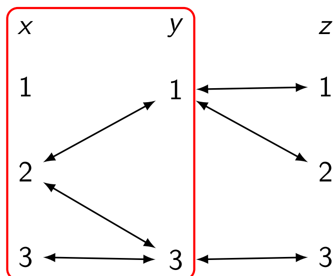
- A support σ of a value v for a constraint c is a solution of this constraint such that every value is itself arc-consistent
  - Propagation until reaching a fix point



Arc-Consistency

Support

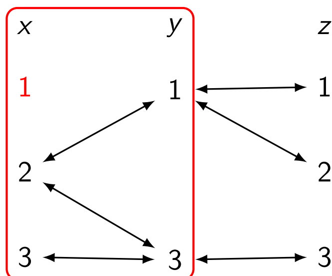
- A support σ of a value v for a constraint c is a solution of this constraint such that every value is itself arc-consistent
  - Propagation until reaching a fix point



Arc-Consistency

Support

- A support σ of a value v for a constraint c is a solution of this constraint such that every value is itself arc-consistent
  - Propagation until reaching a fix point



Bounds Consistency (BC)

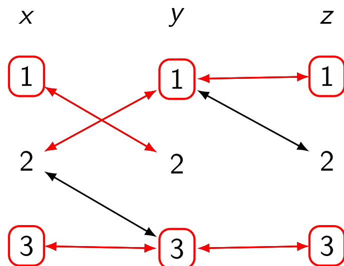
- Notation: $\min(x)$ (resp. $\max(x)$) denotes the smallest, (resp. largest) value in the domain of x
  - Shortcut for $\min(\mathcal{D}(x))$ and $\max(\mathcal{D}(x))$
- The interval $[a, \ldots, b]$ denotes the set of all integers greater or equal to a and smaller or equal to b
- Basic idea: approximate the domains by their "convex" closure
  - $\mathcal{D}(x)$ becomes $[\min(x), \ldots, \max(x)]$

Bounds Consistency (BC)

- Let $\mathcal{P} = (\mathcal{X}, \mathcal{D}, \mathcal{C})$ be constraint network, and c a constraint in $\mathcal{C}$
  - A tuple σ is bounds valid iff $\forall x_i \in \mathcal{Y}, \min(x_i)\ \sigma(x_i) \leq \max(x_i)$
  - A bounds valid tuple σ of the constraint c is a bounds support of c
    - ⋆ i.e., A solution σ of the constraint network $(\mathcal{X}(c), \mathcal{B}, \{c\})$
    - ⋆ Where $\forall x,\ \mathcal{B}(x) = [\min(x), \ldots, \max(x)]$
  - A value $v \in \mathcal{D}(x)$ is bounds viable iff there exists a support σ of c
  - A domain $\mathcal{D}$ is Bounds Consistent iff $\forall x \in \mathcal{X},\ \ \min(x)$ and $\max(x)$ are bounds viable
  - A domain $\mathcal{D}$ is Range Consistent iff $\forall x \in \mathcal{X},\ \forall v \in \mathcal{D}(x),\ \ v$ is bounds viable

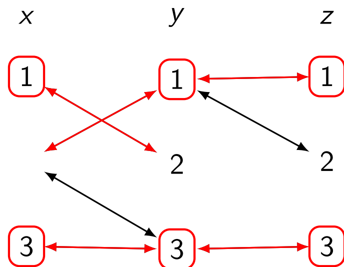
Bounds Consistency (BC)

- Bounds consistency is weaker than arc consistency
- It can be cheaper to achieve the bounds consistent closure

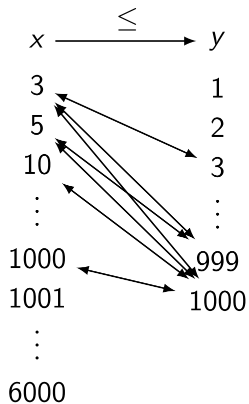


Bounds Consistency (BC)

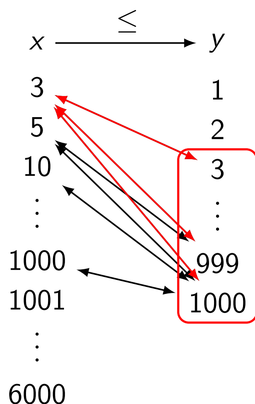
- Bounds consistency is weaker than arc consistency
- It can be cheaper to achieve the bounds consistent closure



Bounds Consistency (BC)

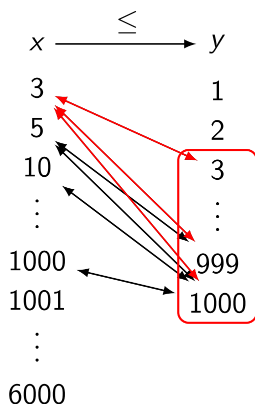


Bounds Consistency (BC)



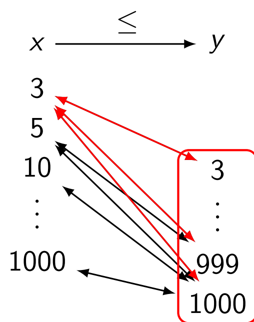
- If (v, w) is a support of $x \leq y$, then for all $u \in \mathcal{D}(y)$ such that $u \geq w$
  - (v, u) is also a support

Bounds Consistency (BC)



- If (v, w) is a support of $x \leq y$, then for all $u \in \mathcal{D}(y)$ such that $u \geq w$
  - (v, u) is also a support
- All bounds inconsistent values can be computed in $O(1)$
  - All values of $\mathcal{D}(x)$ greater than $\max(y)$ are bounds inconsistent
  - All values of $\mathcal{D}(y)$ smaller than $\min(x)$ are bounds inconsistent

Bounds Consistency (BC)

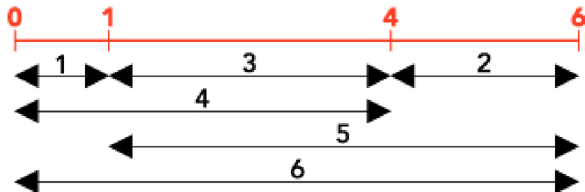


- If (v, w) is a support of $x \leq y$, then for all $u \in \mathcal{D}(y)$ such that $u \geq w$
  - (v, u) is also a support
- All bounds inconsistent values can be computed in $O(1)$
  - All values of $\mathcal{D}(x)$ greater than $\max(y)$ are bounds inconsistent
  - All values of $\mathcal{D}(y)$ smaller than $\min(x)$ are bounds inconsistent
- Given the right data-structures, BC on the constraint $x \leq y$ can be done in $O(1)$
- On this constraint, AC is equivalent to BC

Golomb Ruler

Problem definition

- Place N marks on a ruler
- Distance between each pair of marks is different
- Goal is to minimise the size of the ruler
- Proposed by Sidon [1932] then independently by Golomb and Babcock



Golomb Ruler

- `m` is the number of marks to be on the ruler (input)
- The size of the ruler is at most $\mathtt{n} = 2^{\mathtt{m}-1}$ (`2**(m-1)`)
  - Write a python program to minimize it

Adding an Objective

- `solver.add(Minimise(x))` where `x` is an Expression

Branch & Bound

- Bounds of the objective variable are maintained
  - When a new solution is found, a new upper bound (in the case of minimisation) is deduced
  - Constraint propagation will maintain the lower bound
    - The objective is a variable as any other
    - The way the objective function is defined matters when computing the lower bound

The Data

```
m = 5
n = 2**(m-1)

marks = VarArray(m,n,'m')
distance = [marks[i] - marks[j]
            for i in range(1,m)
            for j in range(i)]
```

A First Model

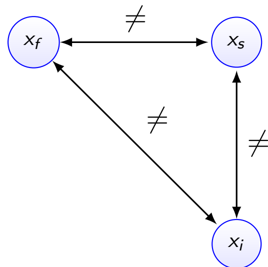
```
m = 5
n = 2**(m-1)

marks = VarArray(m,n,'m')
distance = [marks[i] - marks[j]
            for i in range(1,m)
            for j in range(i)]


model = Model(
    Minimise( Max(marks) ), #objective function

    [m1 != m2 for m1,m2 in pair_of(marks)],
    [d1 != d2 for d1,d2 in pair_of(distance)]
    )
```

Map Coloring

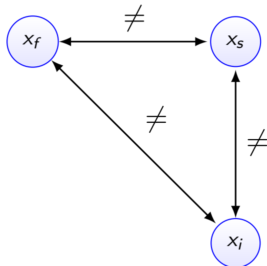


- The network is Arc-Consistent

Map Coloring

$\mathcal{D}(x_f)$: green

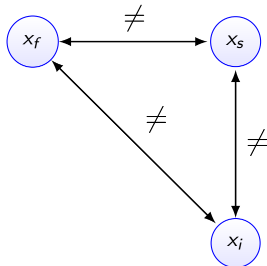
$\mathcal{D}(x_s)$: blue, red



$\mathcal{D}(x_i)$: blue, red

- The network is Arc-Consistent
- However, since Italy and Switzerland must share the colors blue and red, both wil be used
  - blue is not available for France

Map Coloring



- The network is Arc-Consistent
- However, since Italy and Switzerland must share the colors blue and red, both wil be used
  - blue is not available for France
- If we consider the constraint equal to the conjunction of the three inequalities, it is <u>not</u> Arc-consistent

AllDifferent

- $\textsc{AllDifferent}(x_1, \ldots, x_n) \Leftrightarrow \forall 1 \leq i < j \leq n,\ x_i \neq x_j$

- $\textsc{AllDifferent}(x_1, \ldots, x_n) \Leftrightarrow \forall 1 \leq i < j \leq n,\ x_i \neq x_j$
- For instance: $\textsc{AllDifferent}(x_1, x_2, x_3, x_4)$
  - $\mathcal{D}(x_1) = \{1\}$
  - $\mathcal{D}(x_2) = \{1, 2, 3\}$
  - $\mathcal{D}(x_3) = \{1, 2, 3\}$
  - $\mathcal{D}(x_4) = \{1, 2, 3, 4\}$

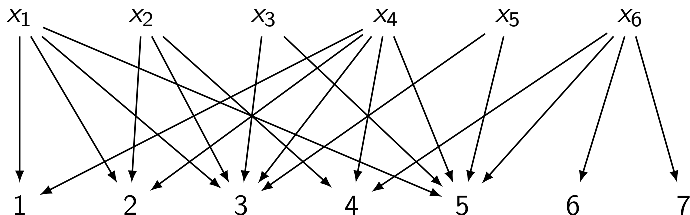
- $\textsc{AllDifferent}(x_1, \ldots, x_n) \Leftrightarrow \forall 1 \leq i < j \leq n,\ x_i \neq x_j$
- For instance: $\textsc{AllDifferent}(x_1, x_2, x_3, x_4)$
  - $\mathcal{D}(x_1) = \{1\}$
  - $\mathcal{D}(x_2) = \{1, 2, 3\}$
  - $\mathcal{D}(x_3) = \{1, 2, 3\}$
  - $\mathcal{D}(x_4) = \{1, 2, 3, 4\}$
- Only two solutions: $(1, 2, 3, 4)$ and $(1, 3, 2, 4)$

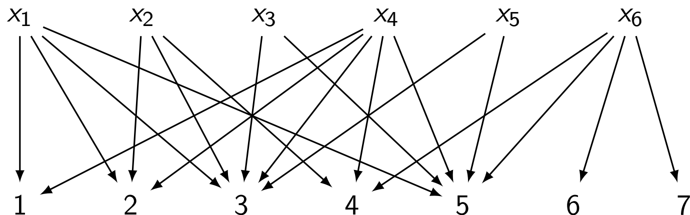
- $\textsc{AllDifferent}(x_1, \ldots, x_n) \Leftrightarrow \forall 1 \leq i < j \leq n,\ x_i \neq x_j$
- For instance: $\textsc{AllDifferent}(x_1, x_2, x_3, x_4)$
  - $\mathcal{D}(x_1) = \{1\}$
  - $\mathcal{D}(x_2) = \{2, 3\}$
  - $\mathcal{D}(x_3) = \{2, 3\}$
  - $\mathcal{D}(x_4) = \{4\}$
- Only two solutions: $(1, 2, 3, 4)$ and $(1, 3, 2, 4)$, therefore:
  - $x_2 = 1$, $x_3 = 1$, $x_4 = 1$, $x_4 = 2$, $x_4 = 3$ are not viable

- $\text{AllDifferent}(x_1, \ldots, x_n) \Leftrightarrow \forall 1 \leq i < j \leq n,\ x_i \neq x_j$
- For instance: $\text{AllDifferent}(x_1, x_2, x_3, x_4)$
  - $\mathcal{D}(x_1) = \{1\}$
  - $\mathcal{D}(x_2) = \{2, 3\}$
  - $\mathcal{D}(x_3) = \{2, 3\}$
  - $\mathcal{D}(x_4) = \{4\}$
- Only two solutions: $(1, 2, 3, 4)$ and $(1, 3, 2, 4)$, therefore:
  - $x_2 = 1$, $x_3 = 1$, $x_4 = 1$, $x_4 = 2$, $x_4 = 3$ are not viable
- How can we compute that efficiently?
  - Generating and testing the validity all permutations would take exponential time

- $\textsc{AllDifferent}(x_1, \ldots, x_n) \Leftrightarrow \forall 1 \leq i < j \leq n,\ x_i \neq x_j$
- For instance: $\textsc{AllDifferent}(x_1, x_2, x_3, x_4, x_5, x_6)$
  - $\mathcal{D}(x_1) = \{1, 2, 3, 5\}$
  - $\mathcal{D}(x_2) = \{2, 3, 4\}$
  - $\mathcal{D}(x_3) = \{3, 5\}$
  - $\mathcal{D}(x_4) = \{1, 2, 3, 4, 5\}$
  - $\mathcal{D}(x_5) = \{3, 5\}$
  - $\mathcal{D}(x_6) = \{4, 5, 6, 7\}$

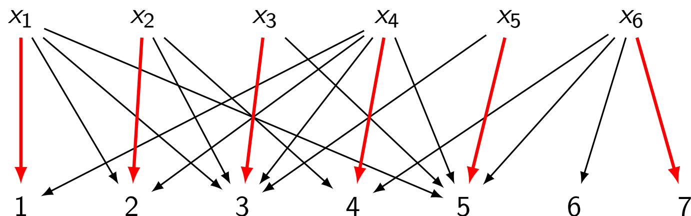
- $\textsc{AllDifferent}(x_1, \ldots, x_n) \Leftrightarrow \forall 1 \leq i < j \leq n,\ x_i \neq x_j$
- For instance: $\textsc{AllDifferent}(x_1, x_2, x_3, x_4, x_5, x_6)$
  - $\mathcal{D}(x_1) = \{1, 2, 3, 5\}$
  - $\mathcal{D}(x_2) = \{2, 3, 4\}$
  - $\mathcal{D}(x_3) = \{3, 5\}$
  - $\mathcal{D}(x_4) = \{1, 2, 3, 4, 5\}$
  - $\mathcal{D}(x_5) = \{3, 5\}$
  - $\mathcal{D}(x_6) = \{4, 5, 6, 7\}$





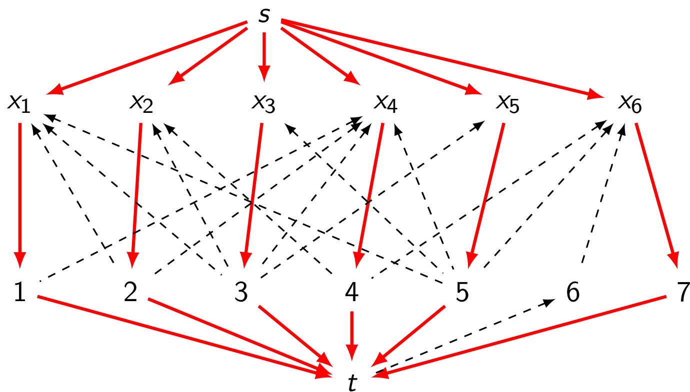
- A solution of the AllDifferent constraint is a maximal matching of the graph

AllDifferent



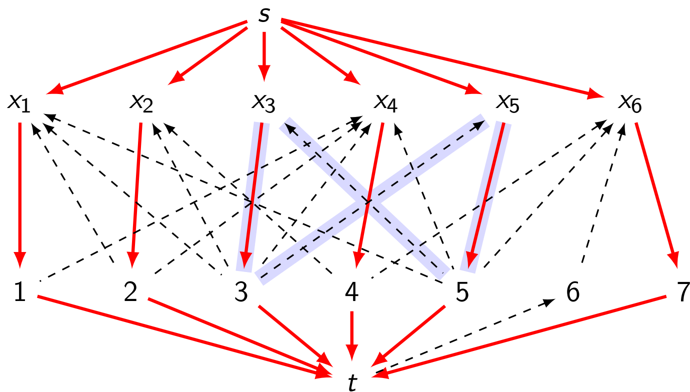
- A solution of the AllDifferent constraint is a maximal matching of the graph
- We can compute a maximal matching in $O(n^{\frac{3}{2}} d)$ (Hopcroft Karp)

AllDifferent

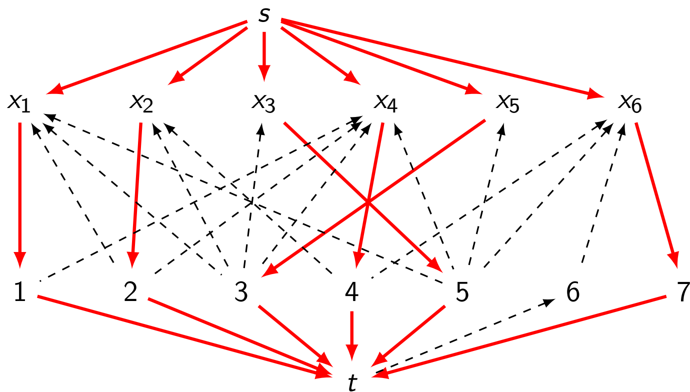


- A solution of the AllDifferent constraint is a maximal matching of the graph
- We can compute a maximal matching in $O(n^{\frac{3}{2}} d)$ (Hopcroft Karp)

AllDifferent

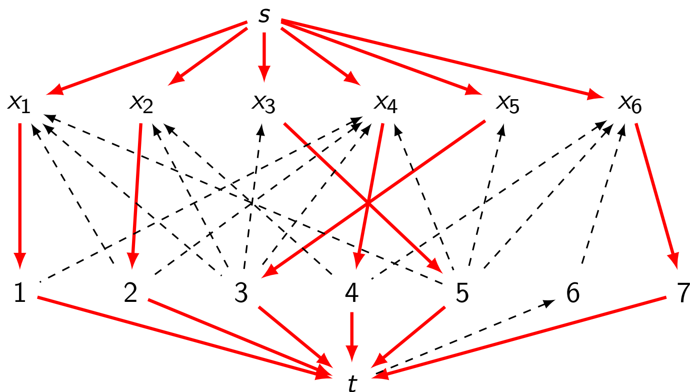


- A solution of the AllDifferent constraint is a maximal matching of the graph
- We can compute a maximal matching in $O(n^{\frac{3}{2}}d)$ (Hopcroft Karp)
- Cycle: alternative matching. Strongly Connected Components are set of vertices all pairwise connected by a cycle. Tarjan's Algorithm finds them all in $O(nd)$



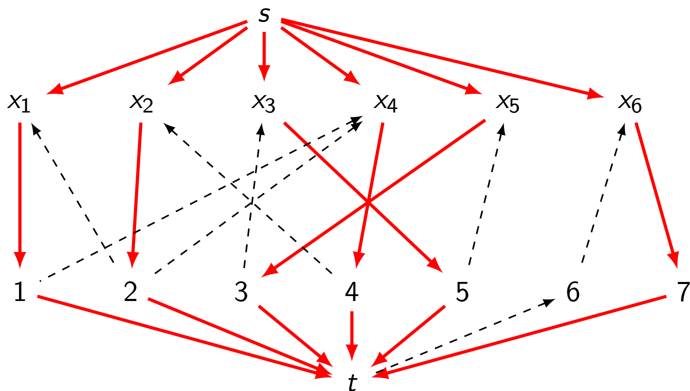
- A solution of the AllDifferent constraint is a maximal matching of the graph
- We can compute a maximal matching in $O(n^{\frac{3}{2}} d)$ (Hopcroft Karp)
- Cycle: alternative matching. Strongly Connected Components are set of vertices all pairwise connected by a cycle. Tarjan's Algorithm finds them all in $O(nd)$

AllDifferent



- A solution of the AllDifferent constraint is a maximal matching of the graph
- We can compute a maximal matching in $O(n^{\frac{3}{2}}d)$ (Hopcroft Karp)
- Cycle: alternative matching. Strongly Connected Components are set of vertices all pairwise connected by a cycle. Tarjan's Algorithm finds them all in $O(nd)$
- An edge (x, v) belongs to a strongly connected component iff the value v is viable for x

AllDifferent



- A solution of the AllDifferent constraint is a maximal matching of the graph
- We can compute a maximal matching in $O(n^{\frac{3}{2}}d)$ (Hopcroft Karp)
- Cycle: alternative matching. Strongly Connected Components are set of vertices all pairwise connected by a cycle. Tarjan's Algorithm finds them all in $O(nd)$
- An edge (x, v) belongs to a strongly connected component iff the value v is viable for $x \Rightarrow$ pruning!

- $\text{AllDifferent}(x_1, \ldots, x_n) \Leftrightarrow \forall 1 \leq i < j \leq n,\; x_i \neq x_j$
- For instance: $\text{AllDifferent}(x_1, x_2, x_3, x_4, x_5, x_6)$
    - $\mathcal{D}(x_1) = \{1, 2, 3, 5\}$
    - $\mathcal{D}(x_2) = \{2, 3, 4\}$
    - $\mathcal{D}(x_3) = \{3, 5\}$
    - $\mathcal{D}(x_4) = \{1, 2, 3, 4, 5\}$
    - $\mathcal{D}(x_5) = \{3, 5\}$
    - $\mathcal{D}(x_6) = \{4, 5, 6, 7\}$

- $\textsc{AllDifferent}(x_1, \dots, x_n) \Leftrightarrow \forall 1 \leq i < j \leq n,\; x_i \neq x_j$
- For instance: $\textsc{AllDifferent}(x_1, x_2, x_3, x_4, x_5, x_6)$
  - $\mathcal{D}(x_1) = \{1, 2\}$
  - $\mathcal{D}(x_2) = \{2, 4\}$
  - $\mathcal{D}(x_3) = \{3, 5\}$
  - $\mathcal{D}(x_4) = \{1, 2, 4\}$
  - $\mathcal{D}(x_5) = \{3, 5\}$
  - $\mathcal{D}(x_6) = \{6, 7\}$

- If $\mathcal{V}$ is a set of variables such that $|\bigcup_{x \in \mathcal{X}} \mathcal{D}(x)| = |\mathcal{X}|$, then $\bigcup_{x \in \mathcal{X}} \mathcal{D}(x)$ is called a Hall set
- For instance: $\textsc{AllDifferent}(x_1, x_2, x_3, x_4)$
  - $\mathcal{D}(x_1) = \{1, 2, 3\}$
  - $\mathcal{D}(x_2) = \{1, 2, 3\}$
  - $\mathcal{D}(x_3) = \{1, 2, 3\}$
  - $\mathcal{D}(x_4) = \{1, 2, 3, 4\}$

- If $\mathcal{V}$ is a set of variables such that $|\bigcup_{x\in\mathcal{X}} \mathcal{D}(x)| = |\mathcal{X}|$, then $\bigcup_{x\in\mathcal{X}} \mathcal{D}(x)$ is called a Hall set
- For instance: $\textsc{AllDifferent}(x_1, x_2, x_3, x_4)$
  - $\mathcal{D}(x_1) = \{1, 2, 3\}$
  - $\mathcal{D}(x_2) = \{1, 2, 3\}$
  - $\mathcal{D}(x_3) = \{1, 2, 3\}$
  - $\mathcal{D}(x_4) = \{4\}$
- $\{1, 2, 3\}$ is a Hall set, therefore $\{1, 2, 3\}$ are not viable for x_4

- If $\mathcal{V}$ is a set of variables such that $|\bigcup_{x \in \mathcal{X}} \mathcal{D}(x)| = |\mathcal{X}|$, then $\bigcup_{x \in \mathcal{X}} \mathcal{D}(x)$ is called a Hall set
- For instance: $\text{ALLDIFFERENT}(x_1, x_2, x_3, x_4)$
  - $\mathcal{D}(x_1) = \{1, 2, 3\}$
  - $\mathcal{D}(x_2) = \{1, 2, 3\}$
  - $\mathcal{D}(x_3) = \{1, 2, 3\}$
  - $\mathcal{D}(x_4) = \{4\}$
- $\{1, 2, 3\}$ is a Hall set, therefore $\{1, 2, 3\}$ are not viable for x_4
- $[1, \ldots, 3]$ is a Hall interval
  - There are at most d^2 intervals of values
  - There exists an algorithm to find all Hall intervals and update the domains accordingly in $O(n \log n)$

Sudoku

- Create the variables

```
grid = Matrix(N*N, N*N, 1, N*N, 'cell_')
```

- The Model

```
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)]
               )
```

  - Every row/column/sub matrix must have different digits
    - **grid.col** returns the transpose of the matrix 'grid'
    - **grid[a:b, c:d]** returns a Matrix construct that contains rows a to b-1 and columns c to d-1.

Sudoku

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

Sudoku

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

Sudoku $\mathrm{AC}(\neq)$

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

Sudoku BC(AllDifferent)

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

Sudoku AC(AllDifferent)

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

Sudoku (Solution)

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

SUM

$$\sum_{i=1}^{n} a_i x_i = K$$

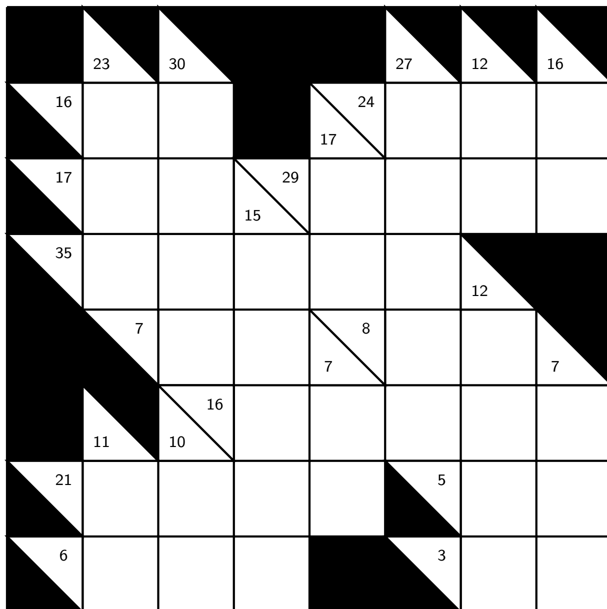
- SUBSET SUM: given a set of integers and an integer K, does there exist a subset whose sum is equal to K
  - A variable with domain $\{0, 1\}$ for each integer, coefficients are the inetegers
- Finding a support is NP-hard
  - Therefore, achieving AC is NP-hard
  - Achieving BC is NP-hard too, since on $\{0, 1\}$ domains, a bounds support is a support
- However, one can enforce BC on each conjunct of:

$$\sum_{i=1}^{n} a_i x_i \leq K \text{ and } \sum_{i=1}^{n} a_i x_i \geq K$$

$$\sum_{i=1}^{n} a_i x_i \leq K$$

- Assume that all coefficients are positive
- $max(x_i) + \sum_{j=1}^{n} a_j \min(x_j) - min(x_i) \leq K$
  - $x_i \leq K - \sum_{j=1}^{n} a_j \min(x_j) - min(x_i)$
- $min(x_i) + \sum_{j=1}^{n} a_j \max(x_j) - min(x_i) \geq K$
  - $x_i \geq K - \sum_{j=1}^{n} a_j \max(x_j) - min(x_i)$

Kakuro



Example: Kakuro

- $\sum_{i=1}^{6} x_i = 39$
- $\text{AllDifferent}(\{x_1, \ldots, x_6\}, \{1, \ldots, 9\})$

x_1 :	{							8	9}
x_2 :	{1	2				6	7	8	9}
x_3 :	{							8	9}
x_4 :	{1				5	6		8	9}
x_5 :	{1	2				6	7	8	9}
x_6 :	{			4	5			8	9}

■	23\	30\	■	■	27\	12\	16\
\16			39\	17\24			
\17							
\35						12\	■
■	\7			7\8			7\
■	11\	10\16					
\21					\5		
\6				■	\3		

Propagation

Example: Kakuro

- $\sum_{i=1}^{6} x_i = 39$
- $\textsc{AllDifferent}(\{x_1, \ldots, x_6\}, \{1, \ldots, 9\})$

$x_1 : \{8 \quad 9\}$

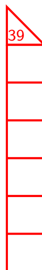
$x_2 : \{1 \quad 2 \quad 6 \quad 7 \quad 8 \quad 9\}$

$x_3 : \{8 \quad 9\}$

$x_4 : \{1 \quad 5 \quad 6 \quad 8 \quad 9\}$

$x_5 : \{1 \quad 2 \quad 6 \quad 7 \quad 8 \quad 9\}$

$x_6 : \{4 \quad 5 \quad 8 \quad 9\}$



Propagation

- $\textsc{AllDifferent}(\{x_1, x_3\}, \{8, 9\})$

Example: Kakuro

- $\sum_{i=1}^{6} x_i = 39$
- $\textsc{AllDifferent}(\{x_1, \ldots, x_6\}, \{1, \ldots, 9\})$

$x_1 : \{ \quad 8 \quad 9 \}$

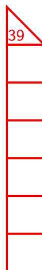
$x_2 : \{1 \quad 2 \quad 6 \quad 7\}$

$x_3 : \{ \quad 8 \quad 9 \}$

$x_4 : \{1 \quad 5 \quad 6\}$

$x_5 : \{1 \quad 2 \quad 6 \quad 7\}$

$x_6 : \{ \quad 4 \quad 5 \}$



Propagation

- $\textsc{AllDifferent}(\{x_1, x_3\}, \{8, 9\})$

Example: Kakuro

- $\sum_{i=1}^{6} x_i = 39$
- $\textsc{AllDifferent}(\{x_1, \ldots, x_6\}, \{1, \ldots, 9\})$

x_1 : { 8 9 }
x_2 : { 1 2 6 7 }
x_3 : { 8 9 }
x_4 : { 1 5 6 }
x_5 : { 1 2 6 7 }
x_6 : { 4 5 }

39

Propagation

- $\sum_{i=1}^{6} = 39$
  - $\Rightarrow \min(x_2) \geq 39 - \sum_{i \neq 2} \max(x_i)$
  - $\Rightarrow \min(x_2) \geq 3, (\& \min(x_5) \geq 3 \ \& \ \min(x_4) \geq 2)$

Example: Kakuro

- $\sum_{i=1}^{6} x_i = 39$
- $\textsc{AllDifferent}(\{x_1, \ldots, x_6\}, \{1, \ldots, 9\})$

x_1 :	{						8	9}
x_2 :	{				6	7		}
x_3 :	{						8	9}
x_4 :	{			5	6			}
x_5 :	{				6	7		}
x_6 :	{		4	5				}

Propagation

- $\sum_{i=1}^{6} = 39$
  - $\Rightarrow \quad \min(x_2) \geq 39 - \sum_{i \neq 2} \max(x_i)$
  - $\Rightarrow \quad \min(x_2) \geq 3, (\& \ \min(x_5) \geq 3 \ \& \ \min(x_4) \geq 2)$

Example: Kakuro

- $\sum_{i=1}^{6} x_i = 39$
- $\textsc{AllDifferent}(\{x_1, \ldots, x_6\}, \{1, \ldots, 9\})$

	4	5	6	7	8	9
x_1 :					8	9
x_2 :			6	7		
x_3 :					8	9
x_4 :		5	6			
x_5 :			6	7		
x_6 :	4	5				

39

Propagation

- $\textsc{AllDifferent}(\{x_2, x_5\}, \{6, 7\})$

Example: Kakuro

- $\sum_{i=1}^{6} x_i = 39$
- $\textsc{AllDifferent}(\{x_1, \ldots, x_6\}, \{1, \ldots, 9\})$

$x_1 : \{ \quad 8 \quad 9 \}$
$x_2 : \{ \quad 6 \quad 7 \quad \}$
$x_3 : \{ \quad 8 \quad 9 \}$
$x_4 : \{ \quad 5 \quad \}$
$x_5 : \{ \quad 6 \quad 7 \quad \}$
$x_6 : \{ \quad 4 \quad 5 \quad \}$

39

Propagation

- $\textsc{AllDifferent}(\{x_2, x_5\}, \{6, 7\})$

Example: Kakuro

- $\sum_{i=1}^{6} x_i = 39$
- $\textsc{AllDifferent}(\{x_1, \ldots, x_6\}, \{1, \ldots, 9\})$

$$
\begin{array}{ll}
x_1: & \{\quad 8 \quad 9\} \\
x_2: & \{\quad 6 \quad 7 \quad\} \\
x_3: & \{\quad 8 \quad 9\} \\
x_4: & \{\quad 5 \quad\} \\
x_5: & \{\quad 6 \quad 7 \quad\} \\
x_6: & \{\quad 4 \quad\}
\end{array}
$$

Propagation

- $\textsc{AllDifferent}(\{x_4\}, \{5\})$

Golomb Ruler: Improving the Model

- Add redundant constraint to reduce the search space
- Use global constraints (AllDifferent) to strengthen propagation

A Better Model

```
model = Model(
     Minimise( marks[-1] ), #objective function

     [marks[i-1] < marks[i] for i in range(1,m)],
     [d1 != d2 for d1,d2 in pair_of(distance)],
     marks[0] == 0, # symmetry breaking
     )
```

Model Variants

- The m marks must all be different in $[1, \ldots, d]$
  - Search space = number of m-permutations: $\frac{d!}{(d-m)!}$
- But they can also be totally ordered
  - Search space = number of m-combinations: $\frac{d!}{m!(d-m)!}$
  - $m!$ times smaller!!
- Instead of marks[0] being any mark on the ruler we can constrain it to be at position 0
  - Reduction of d
- Then we just have to minimise the position of the last mark in the marks array

Using previous solutions

- Deducing information from Golomb Rulers of smaller order
  - If you consider any k consecutive marks of a Golomb Ruler of order $n > k$, they form a Golomb Ruler of order k.
  - Therefore they must span over a distance at least as long as the optimal size of Rulers of order k

Using previous solutions

- Deducing information from Golomb Rulers of smaller order
  - If you consider any k consecutive marks of a Golomb Ruler of order $n > k$, they form a Golomb Ruler of order k.
  - Therefore they must span over a distance at least as long as the optimal size of Rulers of order k

```
[distance[i*(i-1)/2+j] >= ruler[i-j]
 for i in range(1,m)
 for j in range(0,i-1) if (i-j < m)]
)
```

A Better Model

```
model = Model(
    Minimise( marks[-1] ), #objective function

    [marks[i-1] < marks[i] for i in range(1,m)],
    AllDiff(distance),
    marks[0] == 0, # symmetry breaking

    [distance[i*(i-1)/2+j] >= ruler[i-j]
     for i in range(1,m)
     for j in range(0,i-1) if (i-j < m)]
    )
```

Outline

1. Constraint Programming
2. Modelisation
3. Propagation
4. **Search**
   - Backtrack Search
   - Variable Ordering
   - Value Ordering (Branching)
   - Restarts & Randomization
5. Practice

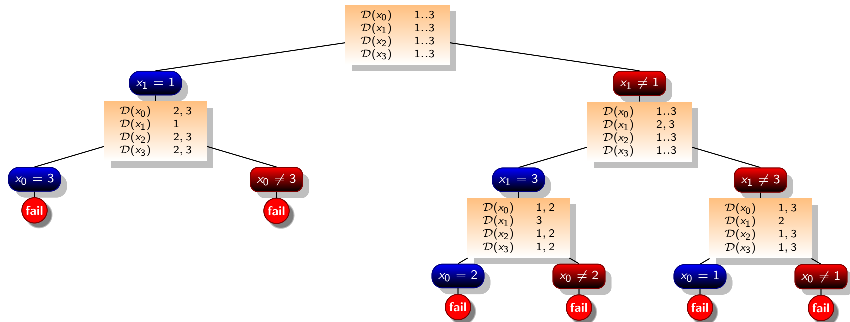
Search Algorithms

- Backtrack
  - Given a variable x and a value $v \in \mathcal{D}(x)$, either:
    - there exists a solution where $x = v$
    - there exists a solution where $x \neq v$
    - there is no solution
  - Exploring both branches gives a complete algorithm
  - Exponential (there is no hope to be polynomial unless $\mathsf{P} = \mathsf{NP}$)
    - In practice, it is often possible to detect inconsistencies after assigning only a few variables
    - Before each decision, there is a propagation step

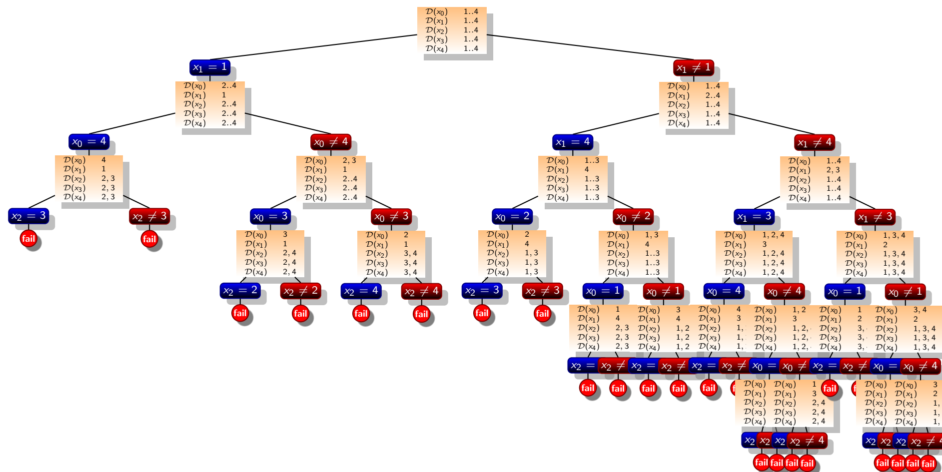
Pigeon Hole Problem

- n pigeons
- $n-1$ holes
- Assign a different hole to each pigeon
  - Clearly impossible if we reason globally ($\textsc{AllDifferent}$)
  - Difficult to detect if we reason about each inequality individually

Search Tree for the Pigeon Hole Problem (4 pigeons, 3 holes)



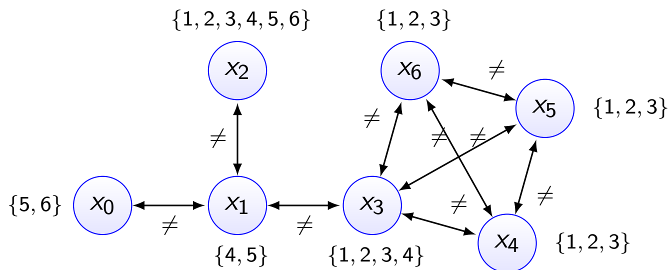
Search Tree for the Pigeon Hole Problem (5 pigeons, 4 holes)



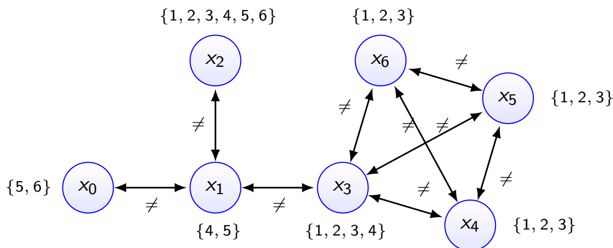
Variable Ordering

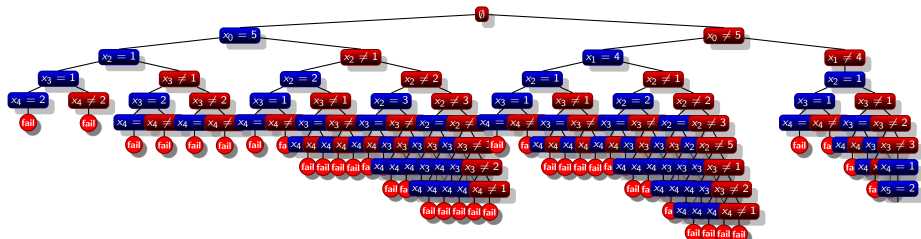
- The order in which variables are explored matters!
  - During search, most of the time is spent in unsatisfiable sub-trees
  - Detecting failure early (fail first)
  - Start with variables that are most likely to “fail”
    - Smallest domain size (less freedom, smaller branching factor)
    - Maximum degree (most constrained variable)
    - Minimum $\frac{\text{domain size}}{\text{degree}}$
    - ...

Coloring



Lexicographic





Lexicographic

$\{1,2,3,5,6\}$ x_2 $\{1,2,3\}$ x_6 x_5 $\{1,2,3\}$

5 x_0 x_1 4 x_3 $\{1,2,3\}$ x_4 $\{1,2,3\}$

$\emptyset$

$x_0 = 5$ $x_0 \neq 5$

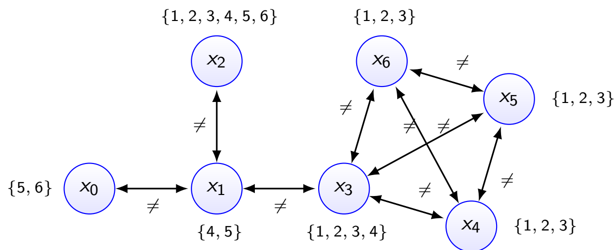
$x_2 = 1$ $x_2 \neq 1$ $x_1 = 4$ $x_1 \neq 4$

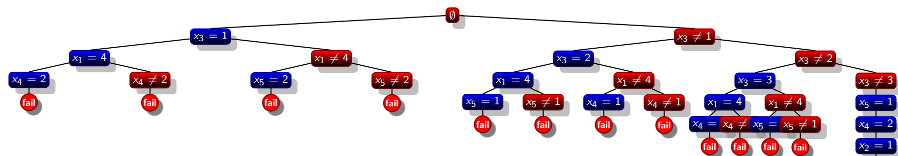
Lexicographic

Lexicographic

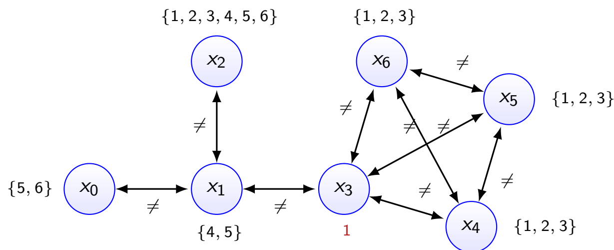
x_0 (5), x_1 (4), x_2 (1), x_3 $\{1, 2, 3\}$, x_4 $\{1, 2, 3\}$, x_5 $\{1, 2, 3\}$, x_6 $\{1, 2, 3\}$

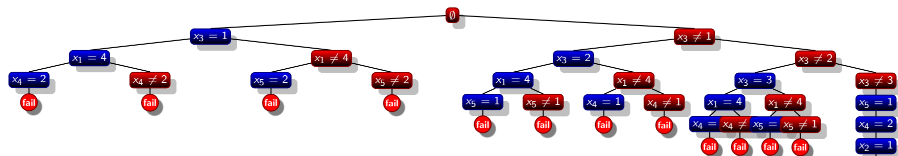
Maximum Degree



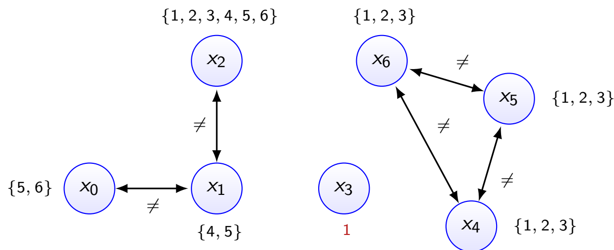


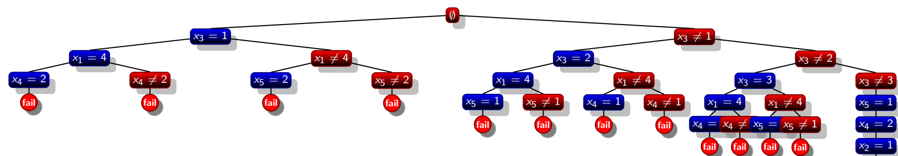
Maximum Degree



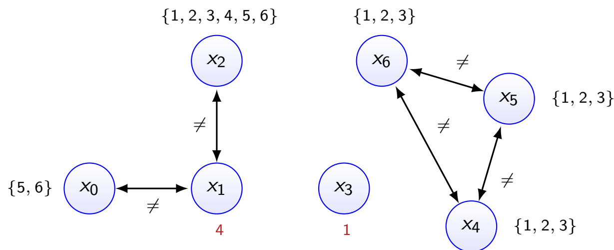


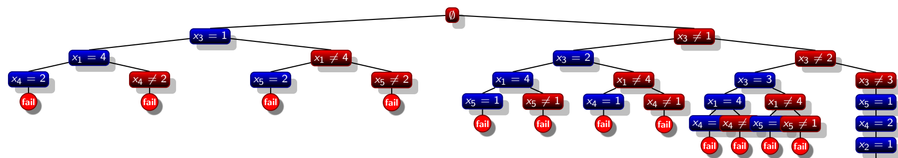
Maximum Degree



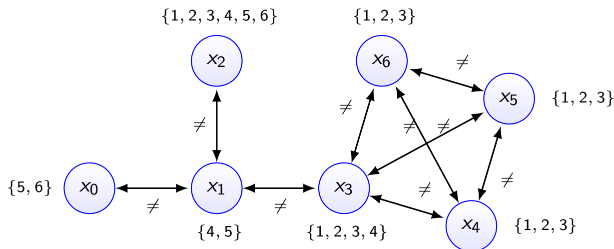


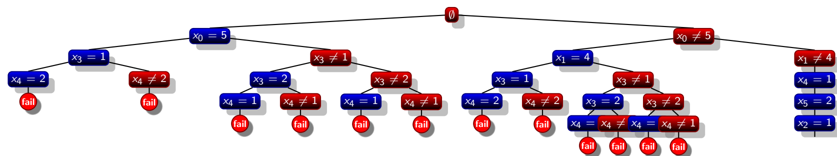
Maximum Degree



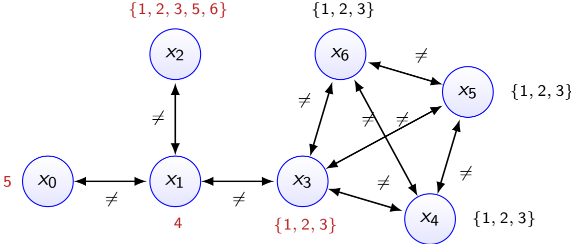


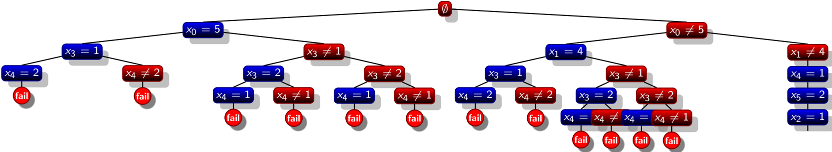
Minimum Domain



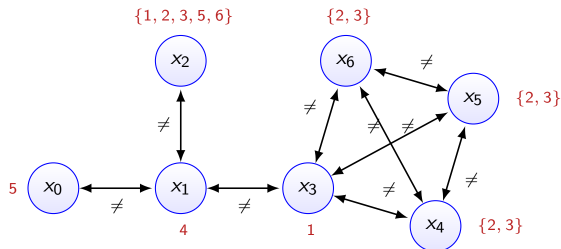


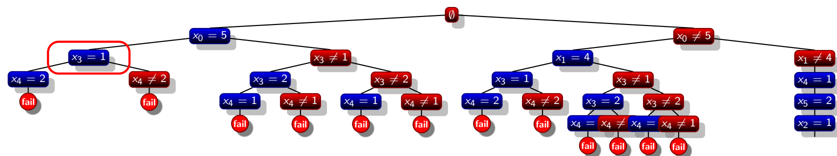
Minimum Domain



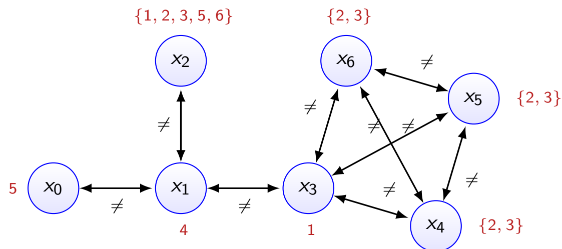


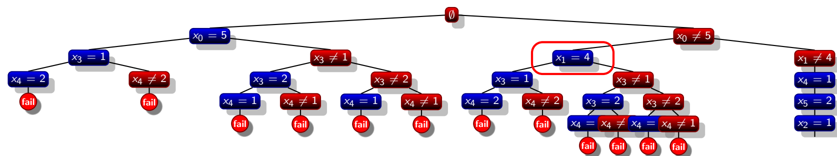
Minimum Domain



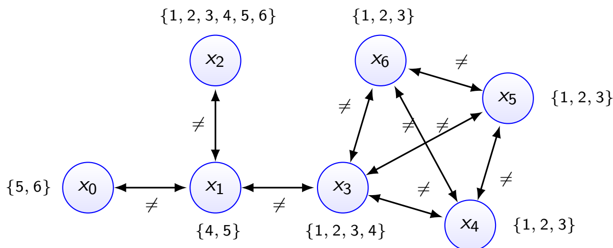


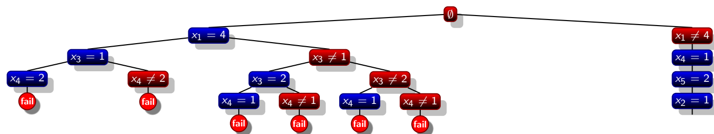
Minimum Domain



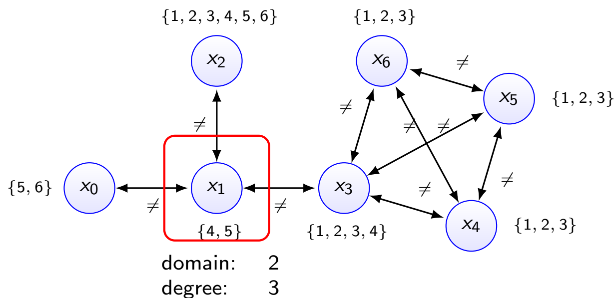


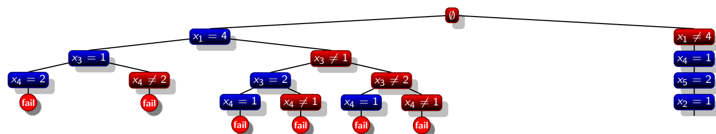
Minimum Domain / Degree



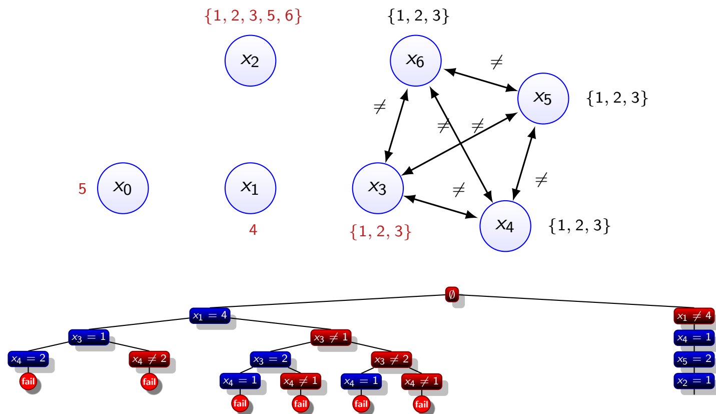


Minimum Domain / Degree

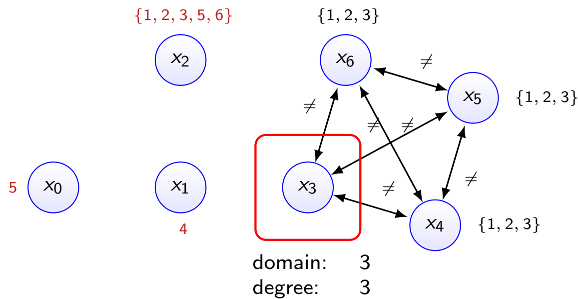


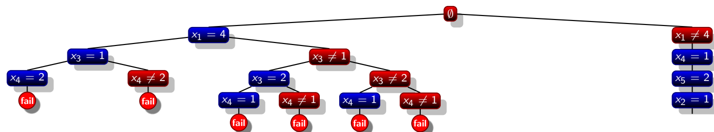


Minimum Domain / Degree

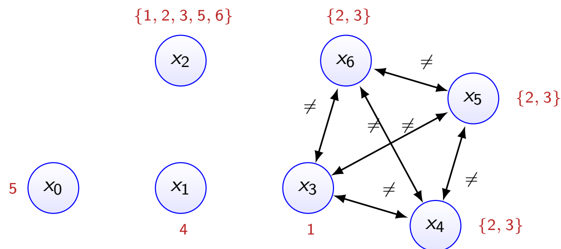


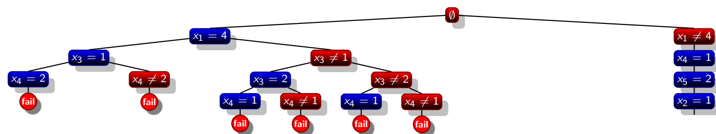
Minimum Domain / Degree



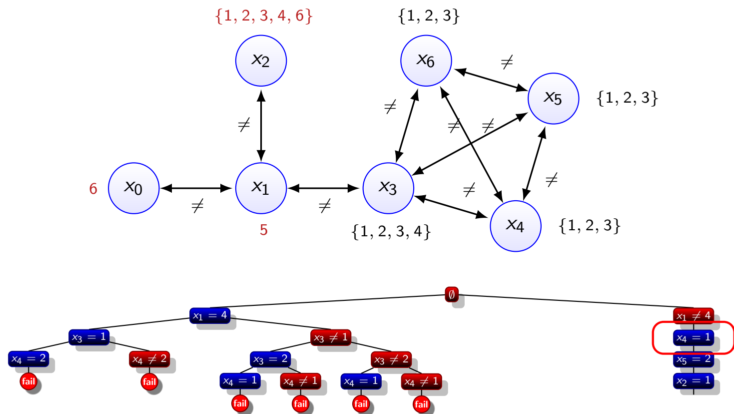


Minimum Domain / Degree

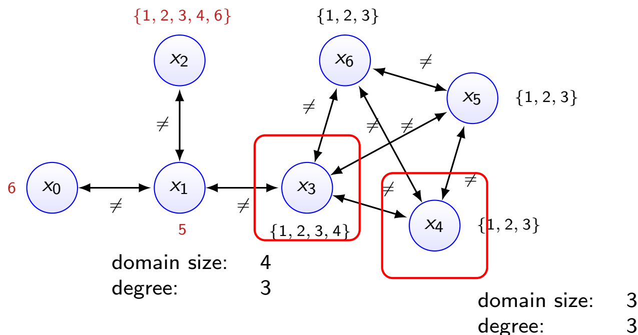


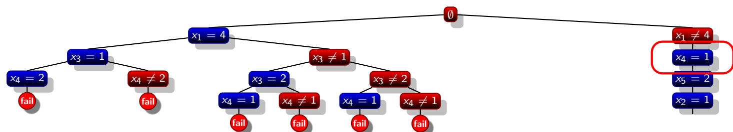


Minimum Domain / Degree



Minimum Domain / Degree





Weighted Degree Heuristic

- The weight of a constraint is initialised to 1
- When propagating, if a contradiction is detected on that constraint, its weight is increased by 1
- The weight of a variable is equal to the sum of the weights of its neighboring constraints

$$\mathrm{weight}(x) = \sum_{c \text{ s.t. } x \in \mathcal{X}(c)} \mathrm{weight}(c)$$

- Domain over weigthed degree:
  - The variable that minimizes the ratio $\frac{\text{domain size}}{\textit{weight}}$ is selected first

Impact

- The impact $I(x, v)$ of a pair (x, v) is initialised to 0
- The size of the search space of a network $\mathcal{P} = (\mathcal{X}, \mathcal{D}, \mathcal{C})$ is bounded by $\prod_{x \in \mathcal{X}} |\mathcal{D}(x)|$
- When the decision $x = v$ is succesful
  - Let b be the size of the search space of $\mathcal{P} = (\mathcal{X}, \mathcal{D}, \mathcal{C})$ (before the decision)
  - Let a be the size of the search space of $\mathcal{P}' = \mathrm{AC}(\mathcal{X}, \mathcal{D}, \mathcal{C} \cup (x = v))$ (after the decision)
  - The impact of this decision corresponds to the reduction of the search space: $\frac{b-a}{a}$
  - The impact of (x, v) is the average impact computed for all decisions $x = v$.
- The weight of a variable x is $\sum_{v \in \mathcal{D}(x)} \frac{1}{I(x,v)}$, the variable with minimum weight is selected first

Value Ordering

- Often considered less important than variable ordering
  - Most of the search effort is spent in unsatisfiable subtrees
  - Value ordering does not really matter in an unsatisfiable subtree since it must be fully explored
- In satisfaction problems:
  - Choosing a satisfiable subtree when branching (promise: choose the least constrained value)
- In optimisation problems:
  - Choosing the assignment that is likely to give the best objective value in order to get good upper bounds quickly

Heavy Tail Behavior

- Given a collection of instances of a problem, we often observe a few exceptionally hard instances
  - These instances are rare, but take exceptionally longer to solve
  - Large impact on the mean runtime for a given set
  - As opposed to normal distributions, the mean does not stabilize when the size of the sample grows
    - When the sample grows, the mean runtime is skewed up
    - Heavy tail behavior
- Not a characteristic of the instance, the same behavior behavior is observed if we run several times the same instance while varying some parameter of the solver
  - For some combination instance / solver parameters, we get trapped into an exponential subtree

Heavy Tail Behavior

- Randomization:
  - Add some randomized parameter in variable or vlue selection (for instance to break ties)
  - Given the same random seed the solver will eplore the same tree, however it will never explore two identical subproblems in the same way
- Restarting:
  - After a given limit r, for instance in number of explored nodes: restart from scratch
- Randomization + restarts eliminates the huge variance in solver performance
  - And therefore reduces the mean runtime when a heavy tail behavior could be observed

- Which limit r should we use?
  - Geometric: $r_i = f^i b$
    - ⋆ $b = 100, f = 2$: 100, 200, 400, 800, ...
  - Luby:
    - ⋆ $r_i = 2^{i-1} b$ if $i = 2^k - 1$
    - ⋆ $r_{i-2^{k-1}+1} b$ if $2^{k-1} \leq i \leq 2^k - 1$
    - ⋆ $b = 10$: 10,10,20,10,10,20,40,10,10,20,10,10,20,40,80 ...

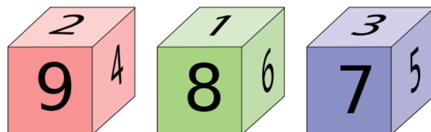
Golomb Ruler: Improving the Resolution

- Try different (randomized) variable heuristics
- Try Different (randomized) value heuristics
- Add Restarts

Outline

1. Constraint Programming
2. Modelisation
3. Propagation
4. Search
5. **Practice**

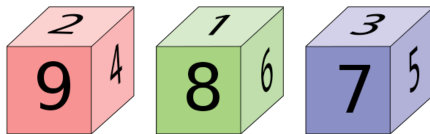
Non Transitive Dice



Non Transitive dice problem

- Set of dice in which the property “beats” is non transitive

Non Transitive Dice



Problem Definition

- 3 dice
- Place digits on faces
- A die beats another if it rolls higher more often
- Need to have three dice such that if a beats b and b beats c then a does not beat c

Non Transitive Dice - Example

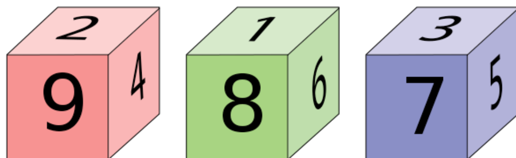
- A solution:

Die A:	1	2	3	4	5	5
Die B:	3	3	3	3	3	3
Die C:	2	2	2	3	6	6

- A beats B:
  - A beats B with probability $\frac{1}{2}$ (18 times out of 36).
  - B beats A with probability $\frac{1}{3}$ (12 times out of 36).
- B beats C:
  - B beats C with probability $\frac{1}{2}$ (18 times out of 36).
  - C beats B with probability $\frac{1}{3}$ (12 times out of 36).
- C beats A:
  - A beats C with probability $\frac{5}{12}$ (15 times out of 36).
  - C beats A with probability $\frac{17}{36}$ (17 times out of 36).

Non Transitive Dice - Exercise

- Find a (different) solution
- Extend the model for any number of dice (D_1 beats D_2, D_2 beats D_3, ..., D_n beats D_1)
- Maximize the sum of the probability gaps ($\frac{1}{2} - \frac{1}{3} + \frac{1}{2} - \frac{1}{3} + \frac{17}{36} - \frac{5}{12}$)
- Maximize the number of 6-faced dice for which there is a cycle



Non Transitive Dice - Correction (1)

```
from Numberjack import *
import Mistral

num_dices = 3
num_sides = 6
min_value = 1
max_value = 6

dice = Matrix(num_dices,num_sides,min_value,max_value,'dice')
wins = VarArray(num_dices, 37, 'wins')
loss = VarArray(num_dices, 37, 'loss')

model = Model(
    [wins[i] == Sum( [side_a > side_b for side_a in dice[i]
                      for side_b in dice[(i+1)%num_dices]] )
     for i in range(num_dices)],
    [loss[i] == Sum( [side_a < side_b for side_a in dice[i]
                      for side_b in dice[(i+1)%num_dices]] )
     for i in range(num_dices)],
    [wins[i] > loss[i] for i in range(num_dices)],
    [dice[i][j-1] <= dice[i][j] for i in range(num_dices)
     for j in range(1,num_sides)],
    dice[0][0] >= dice[1][0],
```

Non Transitive Dice - Correction (2)

```
    Maximise( (num_sides*num_sides*num_dices)*
              Min([wins[i] - loss[i] for i in range(num_dices)])
              +
              Sum([wins[i] - loss[i] for i in range(num_dices)])
              )
    )

solver = Mistral.Solver(model)
solver.setHeuristic('Impact', 'Impact')
solver.solve()

print
for j in range(num_dices):
    print 'dice['+str(j+1)+'] =',
    for i in range(num_sides):
        print dice[j][i],
    print 'wins vs dice['+str((j+1)%num_dices +1)+'] p=', \
        str(wins[j])+'/'+str(num_sides*num_sides) ,
    print 'loses vs dice['+str((j+1)%num_dices +1)+'] p=', \
        str(loss[j])+'/'+str(num_sides*num_sides)
```

Bibliography (books)

- **Handbook of Constraint Programming** Rossi, van Beek and Walsh
- **Constraint Processing** Dechter

Bibliography (propagation)

- Arc Consistency:
  - Waltz, D., **Understanding Line Drawings of Scenes with Shadows.** In: The Psychology of Computer Vision 1975
  - Van Hentenryck, P., Deville, Y., and Teng, C., **A Generic Arc-Consistency Algorithm and its Specializations.** In: Artificial Intelligence 57 (1992), pp. 291-321
  - Bessière, C. and Cordier, M-O., **Arc-Consistency and Arc-Consistency Again.** In: Proceedings of AAAI 1993, pp. 108-113
- AllDifferent (AC):
  - Régin, J-C., **A Filtering Algorithm for Constraints of Difference in CSPs.** In: Proceedings of AAAI 1994, pp. 362-367
- AllDifferent (BC):
  - Lopez-Ortiz, A., Quimper, C-G., Tromp, J. and van Beek, P., **A Fast and Simple Algorithm for Bounds Consistency of the Alldifferent constraint.** In: Proceedings of IJCAI 2003, pp. 245-250

Bibliography (search)

- Articles:
  - Tree search and heuristics:
    - Haralick, R.M. and Elliot G.L., **Increasing Tree Search Efficiency for Constraint Satisfaction Problems** In: Proceedings of IJCAI 1979, pp. 356-364
  - Weighted Degree Heuristic:
    - Boussemart, F., Hemery, F., Lecoutre, C. and Sais, L., **Boosting Systematic Search by Weighting Constraints.** In: Proceedings of ECAI 2004, pp. 146-150
  - Impact Heuristic:
    - Refalo, P., **Impact-Based Search Strategies for Constraint Programming.** In: Proceedings of CP 2004, pp. 557-571
  - Heavy tails and restarts:
    - Gomes, C. P., Selman, B., Crato, N. and Kautz, H., **Heavy-Tailed Phenomena in Satisfiability and Constraint Satisfaction Problems** In: Journal of Automated Reasoning 24:1-2 (2000), pp. 67-100

Bonus: (Water retaining) Magic Square

- N x N grid
- Contains numbers 1 to N^2
- Sum of rows, columns and diagonals equal.
- Known for at least 3000 years!
- Albrecht Dürer in 1514

16	3	2	13
5	10	11	8
9	6	7	12
4	15	14	1

Magic Square

The Model

```
square = Matrix(N,N,1,N*N)
sum_val = N*(N*N+1)/2

model = Model(
    AllDiff( square ),
    [Sum(row) == sum_val for row in square.row],
    [Sum(col) == sum_val for col in square.col],
    Sum([square[a][a] for a in range(N)]) == sum_val,
    Sum([square[a][N-a-1] for a in range(N)]) == sum_val)
```

Water Retention Magic Squares

- Water retention (Craig Knecht 07)
- Magic square = 3D histogram
- Pour water on the histogram
- Maximize the amount of water

Water Retention Magic Squares

Dürer's Square

- One "lake"
- Water level = 9
- Water Depth: 3 and 2
- Water Retention = 5

16	3	2	13
5	10	11	8
9	6	7	12
4	15	14	1

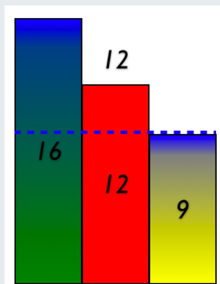
16	3	2	13
5	10	11	8
9	6+3	7+2	12
4	15	14	1

Water Retention Magic Square

How much water will it hold?

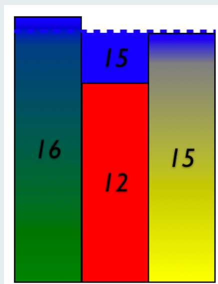
- Keep track of the total height: **sea level**

1st case: landmasses



- The "sea level" around is lower than its own height

2nd case: sea level



- The "sea level" around is greater than its own height

Water Retention Magic Square

Modeling the sea level

- Variables to represent the total level

```
water = Matrix(N,N,1,N*N)
```

Water on the edges

- The edges can't hold any water
  - Matrices allow expression of equality among variable vectors

```
water.row[0] == square.row[0]
water.row[N-1] == square.row[N-1]
water.col[0] == square.col[0]
water.col[N-1] == square.col[N-1]
```

Water Retention Magic Square

Water in the middle

- Sea level at a cell: max of its own height or the surrounding level

```
[water[a][b] ==
 Max((square[a][b],         # own height
      Min((water[a-1][b], water[a][b-1],    # sea level
           water[a+1][b], water[a][b+1])))) # on the cells
                                            # nearby
 for a in range(1,N-1)
 for b in range(1,N-1)]
```

Objective

- Goal is to maximise the water collected

```
    Maximise( Sum( water ) ) # - (N*N*(N*N-1)/2)
```

Water Retention Model

```
square = Matrix(N,N,1,N*N,'s')
water = Matrix(N,N,1,N*N,'w')
model = Model(
    Maximise( Sum( water ) ),   # objective

    # regular magic-square constraints
    AllDiff(square),
    [Sum(row) == N*(N*N+1)/2 for row in square.row],
    [Sum(col) == N*(N*N+1)/2 for col in square.col]
    Sum([square[i][i] for i in range(N)]) == N*(N*N+1)/2,
    Sum([square[i][N-i-1] for i in range(N)]) == N*(N*N+1)/2,

    # channeling with sea level
    water.row[0] == square.row[0],
    water.row[N-1] == square.row[N-1],
    water.col[0] == square.col[0],
    water.col[N-1] == square.col[N-1],
    [water[a][b] ==
     Max((square[a][b], Min((water[a-1][b], water[a][b-1],
                             water[a+1][b], water[a][b+1]))))
     for a in range(1,N-1) for b in range(1,N-1)]
    )
```